PROGRESS Corticon

Corticon Studio: Rule Language Guide

PROGRESS

Notices

Copyright agreement

© 2013 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

Apama, Business Empowerment, Business Making Progress, Corticon, Corticon (and design), DataDirect (and design), DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, Empowerment Center, Fathom, Making Software Work Together, OpenEdge, Powered by Progress, PowerTier, Progress, Progress Control Tower, Progress Dynamics, Progress Business Empowerment, Progress Empowerment Center, Progress Empowerment Program, Progress OpenEdge, Progress Profiles, Progress Results, Progress RPM, Progress Software Business Making Progress, Progress Software Developers Network, ProVision, PS Select, RulesCloud, RulesWorld, SequeLink, SpeedScript, Stylus Studio, Technical Empowerment, WebSpeed, Xcalia (and design), and Your Software, Our Technology–Experience the Connection are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. AccelEvent, Apama Dashboard Studio, Apama Event Manager, Apama Event Modeler, Apama Event Store, Apama Risk Firewall, AppsAlive, AppServer, BusinessEdge, Cache-Forward, DataDirect Spy, DataDirect SupportLink, Future Proof, High Performance Integration, OpenAccess, ProDataSet, Progress Arcade, Progress ESP Event Manager, Progress ESP Event Modeler, Progress Event Engine, Progress RFID, Progress Responsive Process Management, Progress Software, PSE Pro, SectorAlliance, SeeThinkAct, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, WebClient, and Who Makes Progress are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

See Table of Contents for location of Third party acknowledgements within this documentation.

Table of Contents

Preface.....9

Progress Corticon documentation.....9

Overview of Progress Corticon.....11

Chapter 1: Introduction to Corticon Rule Language.....13

Rule structure.....14

Basic data types.....14

Truth values.....15

Collection operators.....15

Language operators.....15

Vocabulary used in this Language Guide.....16

Chapter 2: Rule operators.....17

Icons.....17

Tool tips.....19

Usage restrictions.....19

Chapter 3: Categories of rule operators.....21

Organization.....22

General terms.....22

Attribute operators.....23

Boolean.....23

DateTime.....24

Date.....29

Time.....32

Decimal.....34

Integer.....38

String.....41

Entity/Association operators.....44

Entity.....45

Collection.....45

Sequence.....47

Extended Operators.....49

Chapter 4: Rule operator details and examples.....51

Absolute value.....54

Add numbers....55
Add strings....56
Add days....57
Add hours....58
Add minutes....59
Add months....60
Add seconds....61
Add years....62
Associate element(s)....63
At....64
Average....65
CellValue....66
Concatenate....70
Contains....71
Day....72
Days between....73
Day of week....74
Day of year....76
Decrement....77
Disassociate element(s)....78
Divide....79
Div....80
Ends with....81
Equals – used as an assignment....82
Equals – used as a comparison....83
Equals ignoring case....85
Equals – strings only....86
Exists....87
Exponent....88
False....89
First....90
Floor....91
For all....92
Greater than....94
Greater than or equal to....95
Hour....97
Hour between....98
Increment....99
Index of....100
Is empty....101
Iterate....103
Last....104
Less than....105
Less than or equal to....106
Logarithm (Base 10)....108

Logarithm (Base x)....109
Lowercase....111
Maximum value....112
Maximum value (Collection)....113
Minimum value....114
Minimum value (Collection)....115
Minute....116
Minutes between....117
Mod....118
Month....119
Months between....120
Multiply....121
Natural logarithm....123
New....124
New unique....125
Not....127
Not empty....128
Not equal to....129
Now....130
Null....131
Other....132
Or....134
Remove element....135
Replace element(s)....137
Round....138
Second....140
Seconds between....141
Size of string....142
Size of collection....143
Sorted by....144
Sorted by descending....146
Starts with....149
Substring....150
Subtract....151
Sum....152
Today....154
To date – casting a dateTime to a date....155
To dateTime – casting a string to a dateTime....156
To dateTime – casting a date to a dateTime....157
To dateTime – casting a time to a dateTime....158
To dateTime – timezone offset....159
To decimal....160
To integer....161
To string....162
To time – casting a dateTime to a time....163

Trend....164
True....166
Uppercase....167
Week of month....168
Week of year....169
Year....170
Years between....171

Chapter 5: Special syntax....173

Value ranges....173
Numeric value ranges in conditions....174
Numeric value ranges in filter rows....174
String value ranges in condition cells....175
String value ranges in filter rows....175
DateTime, date, and time value ranges in condition cells....176
DateTime, date, and time value ranges in filter rows....176
Inclusive & exclusive ranges....177
Overlapping value ranges....178
Using value sets in condition cells....180
Using variables as condition cell values....181
Embedding attributes in posted rule statements....183
Including apostrophes in strings....185
Advanced collection syntax....185
Statement blocks....186

Appendix A: Character precedence: Unicode & Java Collator....189

Appendix B: Arithmetic operator precedence....193

Appendix C: DateTime data type....195

Appendix D: Third party acknowledgments 197

Preface

For details, see the following topics:

- Progress Corticon documentation
- Overview of Progress Corticon

Progress Corticon documentation

The following documentation, as well as a *What's New in Corticon* document, is included with this Progress Corticon release:

Corticon Tutorials	
Corticon Studio Tutorial: Basic Rule Modeling	Introduces modeling, analyzing, and testing rules and decisions in Corticon Studio. Recommended for evaluators and users getting started. *See also the PowerPoint-as-PDF version of this document that is accessed from the Studio for Analysts'* ***Help*** *menu.*
Corticon Studio Tutorial: Advanced Rule Modeling	Provides a deeper look into Corticon Studio's capabilities by defining and testing vocabularies, scope, collections, messages, filters, conditions, transient data, and calculations in multiple rulesheets that are assembled into a Ruleflow. *See also the PowerPoint-as-PDF version of this document that is accessed from the Studio for Analysts'* ***Help*** *menu.*
Corticon Tutorial: Using Enterprise Data Connector (EDC)	Introduces Corticon's direct database access with a detailed walkthrough from development in Studio to deployment on Server. Uses Microsoft SQL Server to demonstrate database read-only and read-update functions.

Corticon Studio Documentation: Defining and Modeling Business Rules	
Corticon Studio: Installation Guide	Step-by-step procedures for installing Corticon Studio and Corticon Studio for Analysts on computers running Microsoft Windows. Also shows how use other supported Eclipse installations for integrated development. Shows how to enable internationalization on Windows.
Corticon Studio: Rule Modeling Guide	Presents the concepts and purposes the Corticon Vocabulary, then shows how to work with it in Rulesheets by using scope, filters, conditions, collections, and calculations. Discusses chaining, looping, dependencies, filters and preconditions in rules. Presents the Enterprise Data Connector from a rules viewpoint, and then shows how database queries work. Provides information on versioning, natural language, reporting, and localizing. Provides troubleshooting and many *Test Yourself* exercises.
Corticon Studio: Quick Reference Guide	Reference guide to the Corticon Studio user interface and its mechanics, including descriptions of all menu options, buttons, and actions.
Corticon Studio: Rule Language Guide	Reference information for all operators available in the Corticon Studio Vocabulary. A Rulesheet example is provided for many of the operators. Includes special syntax issues, handling arithmetic and character precedence issues.
Corticon Studio: Extensions Guide	Detailed technical information about the Corticon extension framework for extended operators and service call-outs. Describes several types of operator extensions, and how to create a custom extension plug-in.
Corticon Server Documentation: Deploying Rules as Decision Services	
Corticon Server: Deploying Web Services with Java	Details installing the Corticon Server as a Web Services Server, and then deploying and exposing Decision Services as Web Services on Tomcat and other Java-based servers. Presents the features and functions of the browser-based Server Console.
Corticon Server: Deploying Web Services with .NET	Details installing the Corticon Server as a Web Services Server, and then deploying and exposing decisions as Web Services with .NET. Provides installation and configuration information for the .NET Framework and Internet Information Services (IIS) on various supported Windows platforms.
Corticon Server: Integration & Deployment Guide	An in-depth, technical description of Corticon Server deployment methods, including preparation and deployment of Decision Services and Service Contracts through the Deployment Console tool. Discusses relational database concepts and implementation of the Enterprise Data Connector. Goes deep into the server to discuss state, persistence, and invocations by version or effective date. Includes samples, server monitoring techniques, and recommendations for performance tuning.

Overview of Progress Corticon

Progress® Corticon® is the Business Rules Management System with the patented "no-coding" rules engine that automates sophisticated decision processes.

Progress Corticon products

Progress Corticon distinguishes its development toolsets from its server deployment environments.

- **Corticon Studios** are the Windows-based development environment for creating and testing business rules:
  - **Corticon Studio for Analysts.** is a standalone application, a lightweight installation that focuses exclusively on Corticon.
  - **Corticon Studio** is the *Corticon Designer* perspective in the **Progress Developer Studio** (PDS), an industry-standard Eclipse and Java development environment. The PDS enables development of applications integrated with other products, such as Progress OpenEdge.

  The functionality of the two Studios is virtually identical, and the documentation is appropriate to either product. Documentation of features that are only in the *Corticon Designer* (such as on integrated application development and Java compilation) will note that requirement. Refer to the *Corticon Studio: Installation Guide* to access, prepare, and install each of the Corticon Studio packages.

  Studio Licensing - Corticon embeds a time-delimited evaluation license that enables development of both rule modeling and Enterprise Data Connector (EDC) projects, as well as testing of the projects in an embedded Axis test server. You must obtain studio development licenses from your Progress representative.

- **Corticon Servers** implement web services for business rules defined in Corticon Studios:
  - **Corticon Server for deploying web services with Java** is supported on various application servers, and client web browsers. After installation on a supported Windows platform, that server installation's deployment artifacts can be redeployed on various UNIX and Linux web service platforms as Corticon Decision Services. The guide *Corticon Server: Deploying web services with Java* provides details on the full set of platforms and web service software that it supports, as well as installation instructions in a tutorial format for typical usage.
  - **Corticon Server for deploying web services with .NET** facilitates deployment of Corticon Decision Services on Windows .NET Framework and Microsoft Internet Information Services (IIS). The guide *Corticon Server: Deploying web services with .NET* provides details on the platforms and web service software that it supports, as well as installation instructions in a tutorial format for typical usage.

  Server Licensing - Corticon embeds a time-delimited evaluation license that enables evaluation and testing of rule modeling projects on supported platform configurations. You must obtain server deployment licenses and server licenses that enable the Enterprise Data Connector (EDC) from your Progress representative.

1

Introduction to Corticon Rule Language

Graphical modeling languages and tools (UML, ER, ORM, for example) are not sufficiently precise for specifications. Additional constraints on the objects in the model must also be defined. While natural languages are easily used by individuals without a programming background, they are often ambiguous. On the other hand, formal programming languages are precise, but not easily used by business analysts and other non-programmers.

The Corticon Rule Language has been developed to resolve this dilemma. Based on the Object Constraint Language (OCL, an extension of the Universal Modeling Language specification 1.1), the *Corticon Rule Language* (CRL) is designed to enable non-programmers to express rules clearly and precisely without the use of procedural programming languages. More information on OCL may be found at www.uml.org.

For details, see the following topics:

- Rule structure
- Basic data types
- Truth values
- Collection operators
- Language operators
- Vocabulary used in this Language Guide

Rule structure

In traditional programming languages (or logic systems), most rules are expressed via IF/THEN structures. The IF clause contains a conditional expression and the THEN clause contains actions the rule should perform if all conditions have been met. This IF/THEN structure is expressed as Conditions and Actions in the `Rulesheet` user interface of Corticon Studio. For more information on building and organizing rules in Corticon Studio, see the *Corticon Studio Tutorial: Basic Rule Modeling*.

Basic data types

The proper expression and execution of rules in Corticon Studio is dependent on the type of data involved. Each attribute in the Corticon Studio Business Vocabulary has a data type, meaning that it has restrictions on the type of data it may contain. Corticon standard data types as listed and described in the following table:

Data Type	Description
String	Any combination of alphanumeric characters, of any length
Integer	A whole number, including zero and negative numbers, of any length
Decimal	A number containing a decimal point, including zero and negative numbers, of any length
Boolean	Values are `true` and `false`. `T` and `F` may also be used.
DateTime	Values must be entered for both date and time.
Date	A value with only date information. No Time information is allowed.
Time	Value with only time information. No Date information is allowed.

In this guide, the data types Integer and Decimal are often referred to by the generic term `<Number>`. Wherever `<Number>` is used, either Integer or Decimal data types may be used.

Syntax such as `<DateTime>` indicates that data must conform to the data type shown in angle brackets (`<..>`). For this example, you might enter `9/13/2013 2:00:00 PM EST`. Do not type the angle brackets themselves.

See DateTime data type on page 195 for further details on formatting DateTime, Date, and Time information.

Truth values

This guide uses the notation <`Expression`> to refer to some combination of terms from the Vocabulary that resolves or evaluates to a single "truth value". A truth value is the Boolean value (`true` or `false`) assigned to an expression upon evaluation by the rule engine. For example, the expression `Patient.name='John'` has a truth value of `true` whenever the patient's name is John. If it is not `John`, then the truth value of this expression is `false`.

Collection operators

Many of the operators provided in the Corticon Rule Language deal exclusively with collections of entities. When using collection operators, the expression **must** use aliases to represent the collection(s) operated on by the collection operator(s). A complete discussion of aliases is included in the *Rule Modeling Guide*. Reminders are included throughout this manual wherever collection operators are referenced.

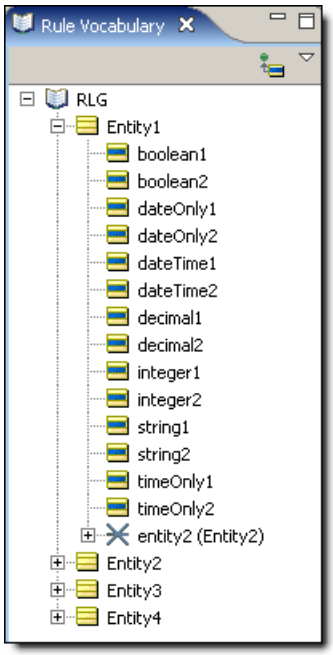
Language operators

The Corticon Rule Language operators can be grouped into various classifications as shown in the following tables. Each operator is subsequently described in detail in the Language Reference section of this document. This section includes a detailed description of the operator, its syntax, usage restrictions, and an example in a Corticon Rulesheet.

Vocabulary used in this Language Guide

This guide uses a generic Vocabulary in all its examples. The Vocabulary contains four entities, each of which contains the same attribute names and types. Attribute names reflect their data types. For example, `integer1` has a data type of Integer. This generic Vocabulary provides sufficient flexibility to create examples using all operators and functions in the Corticon Rule Language. `Entity1` is shown expanded in the following figure:

Figure 1: Vocabulary used in Corticon Language Guide examples



2

Rule operators

This chapter describes the toolset for accessing and using operators.
For details, see the following topics:

- Icons
- Tool tips
- Usage restrictions

Icons

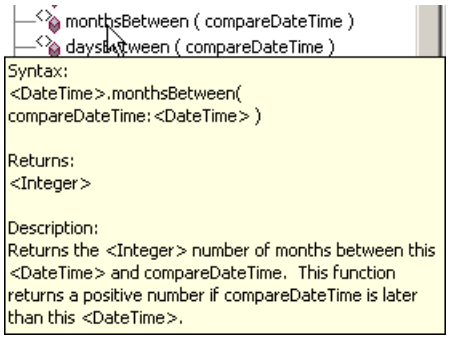
Rule Operators are assigned icons which provide the user with information about their usage. The following table describes these icons:

Icon	Where Found	Purpose	Examples
	General, Literals category	indicates special values or constants	null, true, other
	General, Functions category	indicates system values that are automatically retrieved upon rule execution.	now, today
	Operators, Boolean category	this special "unary" operator icon is used only with not	not
	Operators, all categories	indicates the operator uses a period "." to attach to its operand. Most operators with this icon typically fell into the previous "function" category.	day, round, contains
	Operators, all categories	indicates the operator is used between two operands. Most operators with this icon typically fell into the previous "comparison" category.	equals, multiply
	Operators, Collection & Sequence categories	indicates the operator is used with collections or sequences. Also indicates an alias must be used to represent the collection operated on.	sum, size
	Extended Operators	indicates the operator has been added to the Vocabulary using the extension framework described in *Corticon Extensions Guide*.	-

Tool tips

In Corticon Studio, moving the mouse over a Vocabulary operator and pausing, or "hovering" for a moment, will cause a dynamic "tool tip" text box to display. This tool tip contains information about operator syntax, return data type, and description, all of which are supplied in more detail in this Guide. For questions not answered by the tool tip, refer to the detailed operator descriptions in this Guide. The following figure shows a typical tool tip for the date operator `.monthsBetween`:

Figure 2: Typical Rule Operator Tool Tip



Usage restrictions

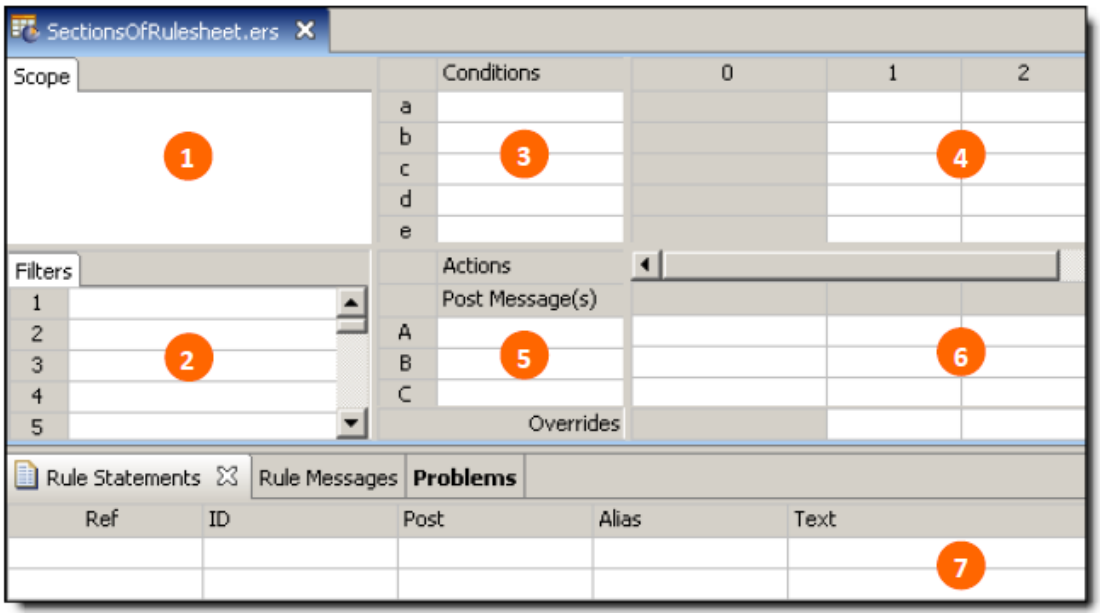
The following illustrations show the general usage restrictions for the various types of Vocabulary terms depending on where they are used in a Rulesheet. This table indicates, for example, that entities (terms from the Vocabulary) may be used in any section of the Rulesheet. Rule Operators, however, are restricted to only three sections.

Note: Some operators have specific restrictions that vary from this general table – see each operator's usage restrictions for details of these exceptions.

Figure 3: Vocabulary usage restrictions in Rulesheet sections

Rulesheet Section Name		Scope	Filter Rows	Condition Rows	Condition Cells	Actions Rows	Action Cells	Rule Statements
Rulesheet Section #		1	2	3	4	5	6	7
Literals			☑	☑	☑	☑	☑	
Functions			☑	☑	☑	☑	☑	☑
Operators			☑	☑		☑		
Data	Values		☑	☑	☑	☑	☑	☑
	Terms	☑	☑	☑	☑	☑	☑	☑

Figure 4: Sections of Rulesheet that correlate with usage restrictions



3

Categories of rule operators

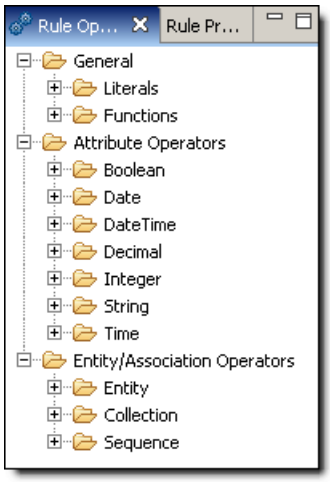
Corticon Studio presents its rule operators in logical groups.
For details, see the following topics:

- Organization
- General terms
- Attribute operators
- Entity/Association operators
- Extended Operators

Organization

Rule Operators are classified based on the data type(s) of the terms *to which the operator may be applied* (known as the "operand"), as shown:

Figure 5: Rule Operator categories



General terms

Corticon's **General** operators are categorized as **Literals** and **Functions**.

Literals

Literal Terms can be used in any section of the Rulesheet, except **Scope** and **Rule Statements**. Exceptions to this general statement exist – see individual literals for detailed usage restrictions.

Corticon's **Literals** operators are as follows:

Name and Syntax	Returns	Description
Null		
`null`	*none*	The null value corresponds to one of three different scenarios: • the absence of an attribute in a Ruletest scenario • the absence of data for an attribute in a Ruletest scenario • an object that has a value of null
True		

Name and Syntax	Returns	Description
`true` or `T`	Boolean	Represents Boolean value true
False		
`false` or `F`	Boolean	Represents the Boolean value false
Other		
`other`	*any*	When included in a condition's Values set, other represents any value not explicitly included in the set, including null.
CellValue		
`cellValue`	*any*	cellValue is a variable whose value is determined by the rule Column that executes

Functions

Corticon's Functions operators are as follows:

Name and Syntax	Returns	Description
Now		
`now`	Date	Returns the current system date and time when the rule is executed.
Today		
`today`	Date	Returns the current system date when the rule is executed.

Attribute operators

The Corticon Rule Language supports attribute operators categorized as Boolean, DateTime, Date, Time, Decimal, Integer, and String.

Boolean

Corticon's **Boolean** attribute operators are as follows:

Name and Syntax	Returns	Description
Equals (used as a comparison)		
<Expression1> = <Expression2>	Boolean	Returns a value of true if <Expression1> has the same value as <Expression2>.
Equals (used as an assignment)		
<Boolean1> = <Expression1>	Boolean	Assigns the truth value of <Expression1> to <Boolean1>
Not Equal To		
<Expression1> <> <Expression2>	Boolean	Returns a value of true if <Expression1> does not have the same truth value as <Expression2>
Or		
<Expression1> or <Expression2> or …	Boolean	Returns a value of true if either <Expression1> or <Expression2> evaluates to true.
And		
<<Boolean1> and <Boolean2>	Boolean	Returns a value of true if both <<Boolean1> and <Boolean2 are true. NOTE: This operator can be used only in Preconditions/Filters section of the Rulesheet.
Not		
not <Expression>	Boolean	Returns the negation of the truth value of <Expression>

DateTime

Note: A DateTime data type **must contain both** date information **and** time information. Applying a DateTime operator to a DateTime attribute should always produce a result. Be sure to use the data type that suits your needs.

Corticon's **DateTime** attribute operators are as follows:

Name and Syntax	Returns	Description
Equals (used as a comparison)		
<DateTime1> = <DateTime2>	Boolean	Returns a value of true if <DateTime1> is the same as <DateTime2>, including both the Date and the Time portions
Equals (used as an assignment)		
<DateTime1> = <DateTime2>	DateTime	Assigns the value of <DateTime2> to <DateTime1>
Not Equal To		
<DateTime1> <> <DateTime2>	Boolean	Returns a value of true if <DateTime1> does not equal <DateTime2>
Less than		
<DateTime1> < <DateTime2>	Boolean	Returns a value of true if <Date1> is less than <Date2>
Greater than		
<DateTime1> > <DateTime2>	Boolean	Returns a value of true if <DateTime1> is greater than or equal to <DateTime2>
Less than or Equal to		
<DateTime1> <= <DateTime2>	Boolean	Returns a value of true if <DateTime1> is less than or equal to <DateTime2>
Greater than or Equal to		
<DateTime1> >= <DateTime2>	Boolean	Returns a value of true if <DateTime1> is greater than or equal to <DateTime2>
Year		
<DateTime>.year	Integer	Returns the century/year portion of <DateTime> as a four digit Integer
Month		
<DateTime>.month	Integer	Returns the month in <DateTime> as an Integer between 1 and 12
Day		
<DateTime>.day	Integer	Returns the day portion of <DateTime> as an Integer between 1 and 31

Name and Syntax	Returns	Description
Hour		
<DateTime>.hour	Integer	Returns the hour portion of <DateTime>. The returned value is based on a 24-hour clock.
Minute		
<DateTime>.min	Integer	Returns the minute portion of <DateTime> as an Integer between 0 and 59
Second		
<DateTime>.sec	Integer	Returns the seconds portion of <DateTime> as an Integer between 0 and 59
Add years		
<DateTime>.addYears (<Integer>)	Date	Adds the number of years in <Integer> to the number of years in <DateTime>
Add months		
<DateTime> .addMonths (<Integer>)	Date	Adds the number of months in <Integer> to the number of months in <DateTime>
Add days		
<DateTime> .addDays (<Integer>)	Date	Adds the number of days in <Integer> to the number of days in <DateTime>
Add hours		
<DateTime> .addHours (<Integer>)	Date	Adds the number of hours in <Integer> to the number of hours in the Time portion of <DateTime>
Add minutes		
<DateTime> .addMinutes (<Integer>)	Date	Adds the number of minutes in <Integer> to the number of minutes in the Time portion of <DateTime>
Add seconds		
<DateTime> .addSeconds (<Integer>)	Date	Adds the number of seconds in <Integer> to the number of seconds in the Time portion of <DateTime>
Years between		

Name and Syntax	Returns	Description
<DateTime1>.yearsBetween (<DateTime2>)	Integer	Returns the Integer number of years between <DateTime1> and <Date2>. This function returns a positive number if <DateTime2> is later than <DateTime1>.
Months between		
<DateTime1> .monthsBetween (<DateTime2>)	Integer	Returns the Integer number of months between <DateTime1> and <DateTime2>. If the month and year portions of <DateTime1> and <DateTime2> are the same, the result is zero. This function returns a positive number if <DateTime2> is later than <DateTime1>.
Days between		
<DateTime1> .daysBetween (<DateTime2>)	Integer	Returns the Integer number of days between <DateTime1> and <DateTime2>. If the two dates differ by less than a full 24-hour period, the value is zero. This function returns a positive number if <DateTime2> is later than <DateTime1>.
Hours between		
<DateTime1> .hoursBetween (<DateTime2>)	Integer	Returns the Integer number of hours between <DateTime1> and <DateTime2>. If the two dates differ by less than a full hour, the value is zero. This function returns a positive number if <DateTime2> is later than <DateTime1>.
Minutes between		
<DateTime1> .minsBetween (<DateTime2>)	Integer	Returns the Integer number of minutes between <DateTime1> and <DateTime2>. This function returns a positive number if <DateTime2> is later than <DateTime1>.
Seconds between		
<DateTime1> .secsBetween (<DateTime2>)	Integer	Returns the Integer number of seconds between <DateTime1> and <DateTime2>. This function returns a positive number if <DateTime2> is later than <DateTime1>.
Day of Week		

Name and Syntax	Returns	Description
<DateTime>.dayOfWeek	Integer	Returns an Integer corresponding to day of the week, with Sunday equal to 1, in <DateTime>.
Week of Year		
<DateTime>.weekOfYear	Integer	Returns an Integer from 1 to 52, equal to the week number within the year in <DateTime>
Day of Year		
<DateTime>.dayOfYear	Integer	Returns an Integer from 1 to 366, equal to the day number within the year in <DateTime>
Week of Month		
<DateTime>.weekOfMonth	Integer	Returns an Integer from 1 to 6, equal to the week number within the month in <DateTime> or <Date>. A week begins on Sunday and ends on Saturday.
To Date		
<DateTime>.toDate	Date	Returns the date portion only of DateTime
To Time		
<DateTime>.toTime	Time	Returns the time portion only of DateTime
To String		
<DateTime>.toString	String	Converts DateTime to a String with date and time information
getMilliseconds		
<DateTime>.getMilliseconds	Integer	Returns the internal date/time, namely the number of milliseconds that have transpired since the epoch 1/1/1970 00:00:00 GMT.
toString		
<DateTime>.toString	Integer	Converts the value of <DateTime> to data type <String>.
toZulu		
<DateTime>.toZulu	String	Returns an ISO-8601-compliant date-time as a String.

Date

Corticon's **Date** attribute operators are as follows:

Name and Syntax	Returns	Description
Equals (used as a comparison)		
<Date1> = <Date2>	Boolean	Returns a value of true if <Date1> is the same as <Date2>.
Equals (used as an assignment)		
<Date1> = <Date2>	DateTime	Assigns the value of <Date2> to <Date1>
Not Equal To		
<Date1> <> <Date2>	Boolean	Returns a value of true if <Date1> does not equal <Date2>
Less than		
<Date1> < <Date2>	Boolean	Returns a value of true if <Date1> is less than <Date2>
Greater than		
<Date1> > <Date2>	Boolean	Returns a value of true if <Date1> is greater than or equal to <Date2>
Less than or Equal to		
<Date1> <= <Date2>	Boolean	Returns a value of true if <Date1> is less than or equal to <Date2>
Greater than or Equal to		
<Date1> >= <Date2>	Boolean	Returns a value of true if <Date1> is greater than or equal to <Date2>
Year		
<Date>.year	Integer	Returns the century/year portion of <Date> as a four digit Integer
Month		
<Date>.month	Integer	Returns the month in <Date> as an Integer between 1 and 12
Day		

Name and Syntax	Returns	Description
<Date>.day	Integer	Returns the day portion of <Date> as an Integer between 1 and 31
Add years		
<Date> .addYears(<Integer>)	Date	Adds the number of years in <Integer> to the number of years in <Date>
Add months		
<Date> .addMonth(<Integer>)	Date	Adds the number of months in <Integer> to the number of months in <DateTime>
Add days		
<Date> .addDays(<Integer>)	Date	Adds the number of days in <Integer> to the number of days in <Date>
Years between		
<Date1> .yearsBetween(<Date2>)	Integer	Returns the Integer number of years between <Date1> and <Date2>. This function returns a positive number if <Date2> is later than <Date1>.
Months between		
<Date1> .monthsBetween(<Date2>)	Integer	Returns the Integer number of months between <Date1> and <Date2>. If the month and year portions of <Date1> and <Date2> are the same, the result is zero. This function returns a positive number if <Date2> is later than <Date1>.
Days between		
<Date1> .daysBetween(<Date2>)	Integer	Returns the Integer number of days between <Date1> and <Date2>. If the two dates differ by less than a full 24-hour period, the value is zero. This function returns a positive number if <Date2> is later than <Date1>.
Day of Week		
<Date> .dayOfWeek	Integer	Returns an Integer corresponding to day of the week, with Sunday equal to 1, in <Date>.
Week of Year		

Name and Syntax	Returns	Description
<Date> .weekOfYear	Integer	Returns an Integer from 1 to 52, equal to the week number within the year in <Date>
Day of Year		
<Date> .dayOfYear	Integer	Returns an Integer from 1 to 366, equal to the day number within the year in <Date>
Week of Month		
<Date> .weekOfMonth	Integer	Returns an Integer from 1 to 6, equal to the week number within the month in <DateTime> or <Date>. A week begins on Sunday and ends on Saturday.
To String		
<Date> .toString	String	Converts DateTime to a String with date and time information
To DateTime		
<Date> .toDateTime	DateTime	Returns a DateTime where the date portion is equal to the value of <Date> and the time portion is equal to 00:00:00 in the system's local timezone
To DateTime with Timezone Offset		
<Date> .toDateTime (<string>)	DateTime	Returns a DateTime where the date portion is equal to the value of <Date> and the time portion is equal to 00:00:00 in the timezone specified by the value of <string>
getMilliseconds		
toString		
<Date>.toString	Integer	Converts the value of <Date> to data type <String>.
<Date>.getMilliseconds	Integer	Returns the internal date/time, namely the number of milliseconds that have transpired since the epoch 1/1/1970 00:00:00 GMT.
nextDay		
<Date>.nextDay	Date	Returns the Date that represents the date that follows this Date instance.

Time

Corticon's **Time** attribute operators are as follows:

Name and Syntax	Returns	Description
Equals (used as a comparison)		
<Time1> = <Time2>	Boolean	Returns a value of true if <Time1> is the same as <Time2>, including both the Date and the Time portions
Equals (used as an assignment)		
<Time1> = <Time2>	DateTime	Assigns the value of <Time2> to <Time1>
Not Equal To		
<Time1> <> <Time2>	Boolean	Returns a value of true if <Time1> does not equal <Time2>
Less than		
<Time1> < <Time2>	Boolean	Returns a value of true if <Time1> is less than <Time2>
Greater than		
<Time1> > <Time2>	Boolean	Returns a value of true if <Time1> is greater than <Time2>
Less than or Equal to		
<Time1> <= <Time2>	Boolean	Returns a value of true if <Time1> is less than or equal to <Time2>
Greater than or Equal to		
<Time1> >= <Time2>	Boolean	Returns a value of true if <Time1> is greater than or equal to <Time2>
Hour		
<Time>.hour	Integer	Returns the hour portion of <Time>. The returned value is based on a 24-hour clock.
Minute		
<Time>.min	Integer	Returns the minute portion of <Time> as an Integer between 0 and 59
Second		

Name and Syntax	Returns	Description
<Time>.sec	Integer	Returns the seconds portion of <Time> as an Integer between 0 and 59
<Time> .addDays (<Integer>)	Date	Adds the number of days in <Integer> to the number of days in <Time>
Add hours		
<Time> .addHours (<Integer>)	Date	Adds the number of hours in <Integer> to the number of hours in the Time portion of <Time>
Add minutes		
<Time> .addMinutes (<Integer>)	Date	Adds the number of minutes in <Integer> to the number of minutes in the Time portion of <Time>
Add seconds		
<Time> .addSeconds (<Integer>)	Date	Adds the number of seconds in <Integer> to the number of seconds in the Time portion of <Time>
Hours between		
<Time1> .hoursBetween (<Time2>)	Integer	Returns the Integer number of hours between <Time1> and <Time2>. If the two times differ by less than a full hour, the value is zero. This function returns a positive number if <Time2> is later than <Time1>.
Minutes between		
<Time1> .minsBetween (<Time2>)	Integer	Returns the Integer number of minutes between <Time1> and <Time2>. This function returns a positive number if <Time2> is later than <Time1>.
Seconds between		
<Time1> .secsBetween (<Time2>)	Integer	Returns the Integer number of seconds between <Time1> and <Time2>. This function returns a positive number if <Time2> is later than <Time1>.
To String		
<Time>.toString	String	Converts <Time> to a String with date and time information

Name and Syntax	Returns	Description
To DateTime		
<Time>.toDateTime	DateTime	Returns a DateTime where the time portion is equal to the value of <Time> and the date portion is equal to the epoch.
toString		
<Time>.toString	Integer	Converts the value of <Time> to data type <String>.
getMilliseconds		
<Time>.getMilliseconds	Integer	Returns the internal date/time, namely the number of milliseconds that have transpired since the epoch 1/1/1970 00:00:00 GMT.
toString		
<Time>.toString	Integer	Converts the value of <Time> to data type <String>.
getTimeName		
<Time>.getTimeName	String	Returns a String that states whether the time is morning, afternoon, or evening.

Decimal

In this section, wherever the syntax includes <Number>, either Integer or Decimal data types may be used.

Corticon's **Decimal** attribute operators are as follows:

Name and Syntax	Returns	Description
Equals (used as a comparison)		
<Number1> = <Number2>	Boolean	Returns a value of true if <Number1> is the same as <Number2>.
Equals (used as an assignment)		
<Number1> = <Number2>	Number	Assigns the value of <Number2> to the value of <Number1>.
Not Equal To		
<Number1> <> <Number2>	Boolean	Returns a value of true if <Number1> is not equal to <Number2>.

Name and Syntax	Returns	Description
Less than		
<Number1> < <Number2>	Boolean	Returns a value of true if <Number1> is less than <Number2>.
Greater than		
<Number1> > <Number2>	Boolean	Returns a value of true if <Number1> is greater than <Number2>.
Less than or Equal to		
<Number1> <= <Number2>	Boolean	Returns a value of true if <Number1> is less than or equal to <Number2>.
Greater than or Equal to		
<Number1> >= <Number2>	Boolean	Returns a value of true if <Number1> is greater than or equal to <Number2>.
Add		
<Number1> + <Number2>	Number	Returns the sum of <Number1> and <Number2>. The resulting data type is the more expansive of either <Number1> or <Number2>. For example, if an Integer value is added to a Decimal value, the resulting value will be a Decimal. See Arithmetic operator precedence on page 193.
Subtract		
<Number1> - <Number2>	Number	Subtracts <Number2> from <Number1>. The resulting data type is the more expansive of either <Number1> or <Number2>. See Arithmetic operator precedence on page 193.
Multiply		
<Number1> * <Number2>	Number	Returns the product of <Number1> and <Number2>. The resulting data type is the more expansive of either <Number1> or <Number2>. See Arithmetic operator precedence on page 193.
Divide		

Name and Syntax	Returns	Description
<Number1> / <Number2>	Number	Divides <Number1> by <Number2>. The resulting data type is the more expansive of either <Number1> or <Number2>. See Arithmetic operator precedence on page 193.
Exponent		
<Number1> ** <Number2>	Number	Raises <Number1> to the power of <Number2>. The resulting data type is the more expansive of either <Number1> or <Number2>. See Arithmetic operator precedence on page 193.
Increment		
<Number1> += <Number2>	Number	Increments <Number1> by <Number2>. The data type of <Number1> must accommodate the addition of <Number2> . See Arithmetic operator precedence on page 193.
Decrement		
<Number1> -= <Number2>	Number	Decrements <Number1> by the value of <Number2>. The data type of <Number1> must accommodate the addition of <Number2>. See Arithmetic operator precedence on page 193.
Absolute Value		
<Decimal>.absVal	Decimal	Returns the absolute value of <Number>. If the <Number> is positive, <Number> itself is returned; if <Number> is negative, the negation of <Number> is returned.
Floor		
<Decimal>.floor	Integer	Returns the largest (closest to positive infinity) Integer that is not greater than <Number>.
Round		
<Decimal>.round	Decimal	Rounds <Decimal> to the nearest Integer.
Round(n)		
<Decimal>.round(<Integer>)	Decimal	Rounds <Decimal> to the number of decimal places specified by <Integer>.
To Integer		

Name and Syntax	Returns	Description
<Decimal>.toInteger	Integer	Converts an attribute of type Decimal to type Integer. Decimals will have the decimal point and fraction (those digits to the right of the decimal point) truncated.
To String		
<Decimal>.toString	String	Converts an attribute of type Decimal to type string
Maximum Value		
<Decimal> .max(<Number>)	Number	Returns the greater of <Decimal> and <Number>.
Minimum Value		
<Decimal> .min(<Number>)	Number	Returns the lesser of <Decimal> and <Number>.
Logarithm (base 10)		
<Decimal> .log	Decimal	Returns the logarithm (base 10) of <Decimal>. <Decimal> may not be zero.
Logarithm (base x)		
<Decimal1> .log(<Decimal2>)	Decimal	Returns the logarithm (base <Decimal2>) of <Decimal1>. <Decimal1> may not be zero.
Natural Logarithm		
<Decimal>.ln	Decimal	Returns the logarithm (base e) of <Decimal>.<Decimal> may not be zero.
truncate		
<Decimal>.truncate	Integer	Truncates "this" Decimal value to an integer by removing the fractional portion.
toString		
<Decimal>.fraction	Decimal	Extracts the fraction portion of "this" Decimal.
movePoint(places)		
<Decimal>.movePoint (places:Integer)	Decimal	Moves the Decimal value's point moved n places where n can be a positive (moves right) or negative (moves left) value.

Integer

In this section, wherever the syntax includes <Number>, either Integer or Decimal data types may be used.

Corticon's **Integer** attribute operators are as follows:

Name and Syntax	Returns	Description
Equals (used as a comparison)		
<Number1> = <Number2>	Boolean	Returns a value of true if <Number1> is the same as <Number2>.
Equals (used as an assignment)		
<Number1> = <Number2>	Number	Assigns the value of <Number2> to the value of <Number1>. The data type of <Number1> must be expansive enough to accommodate <Number2>.
Not Equal To		
<Number1> <> <Number2>	Boolean	Returns a value of true if <Number1> is not equal to <Number2>.
Less than		
<Number1> < <Number2>	Boolean	Returns a value of true if <Number1> is less than <Number2>.
Greater than		
<Number1> > <Number2>	Boolean	Returns a value of true if <Number1> is greater than <Number2>.
Less than or Equal to		
<Number1> <= <Number2>	Boolean	Returns a value of true if <Number1> is less than or equal to <Number2>.
Greater than or Equal to		
<Number1> >= <Number2>	Boolean	Returns a value of true if <Number1> is greater than or equal to <Number2>.
Add		

Name and Syntax	Returns	Description
<Number1> + <Number2>	Number	Returns the sum of <Number1> and <Number2>. The resulting data type is the more expansive of either <Number1> or <Number2>. For example, if an Integer value is added to a Decimal value, the resulting value will be a Decimal. See Arithmetic operator precedence on page 193.
Subtract		
<Number1> - <Number2>	Number	Subtracts <Number2> from <Number1>. The resulting data type is the more expansive of either <Number1> or <Number2>. See Arithmetic operator precedence on page 193.
Multiply		
<Number1> * <Number2>	Number	Returns the product of <Number1> and <Number2>. The resulting data type is the more expansive of either <Number1> or <Number2>. See Arithmetic operator precedence on page 193.
Divide		
<Number1> / <Number2>	Number	Divides <Number1> by <Number2>. The resulting data type is the more expansive of either <Number1> or <Number2>. See Arithmetic operator precedence on page 193.
Exponent		
<Number1> ** <Number2>	Number	Raises <Number1> to the power of <Number2>. The resulting data type is the more expansive of either <Number1> or <Number2>. See Arithmetic operator precedence on page 193.
Increment		
<Number1> += <Number2>	Number	Increments <Number1> by <Number2>. The data type of <Number1> must accommodate the addition of <Number2>. See Arithmetic operator precedence on page 193.
Decrement		

Name and Syntax	Returns	Description
<Number1> -= <Number2>	Number	Decrements <Number1> by the value of <Number2>. The data type of <Number1> must accommodate the addition of <Number2>. See Arithmetic operator precedence on page 193.
Absolute value on page 54.		
<Integer>.absVal	Number	Returns the absolute value of <Integer>. If the <Integer> is positive, <Integer> itself is returned; if <Integer> is negative, the negation of <Integer> is returned.
To Decimal		
<Integer>.toDecimal	Decimal	Converts an attribute of type Integer to type Decimal.
To String		
<Integer>.toString	String	Converts an attribute of type Integer to type String.
Maximum Value		
<Integer1> .max(<Integer2>)	Integer	Returns the greater of <Integer1> and <Integer2>.
Minimum Value		
<Integer1> .min(<Integer2>)	Integer	Returns the lesser of <Integer1> and <Integer2>.
Div		
<Integer1> .div(<Integer2>)	Integer	Returns the whole number of times that <Integer2> fits within <Integer1> - any remainder is discarded.
Mod		
<Integer1> .mod(<Integer2>)	Integer	Returns the whole number remainder that results from dividing <Integer1> by <Integer2>. If the remainder is a fraction, then zero is returned.
Logarithm (base 10)		
<Integer> .log	Decimal	Returns the logarithm (base 10) of <Integer>. <Integer> may not be zero.

Name and Syntax	Returns	Description
Logarithm (base x)		
<Integer> .log(<Decimal>)	Decimal	Returns the logarithm (base <Decimal>) of <Integer>. <Integer> may not be zero.
Natural Logarithm		
<Integer>.ln	Decimal	Returns the natural logarithm (base e) of <Number>. <Integer> may not be zero.
isProbablePrime(certainty)		
<Integer>.isProbablePrime (certainty:Integer)	Boolean	Returns true if this Integer is probably prime; false if definitely is not prime .
gcd(val)		
<Integer>.gcd(val:Integer)	Integer	Returns the greatest common divisor of the absolute value of this and the absolute value of val.
negate		
<Integer>.negate	Integer	Returns the negative value of this integer.

String

Corticon's **String** attribute operators are as follows:

Name and Syntax	Returns	Description
Equals (used as a comparison)		
<String1> = <String2>	Boolean	Returns a value of true if <String1> exactly matches <String2>. Both case and length are examined to determine equality. See Character precedence: Unicode & Java Collator on page 189 for character precedence.
Equals (used as an assignment)		
<String1> = <String2>	String	Assigns the value of <String2> to the value of <String1>.
Not Equal to		
<String1> <> <String2>	Boolean	Returns a value of true if <String1> is not equal to <String2>.

Name and Syntax	Returns	Description
Less than		
<String1> < <String2>	Boolean	Returns a value of true if <String1> is less than <String2>. See Character precedence: Unicode & Java Collator on page 189 for character precedence.
Greater than on page 94		
<String1> > <String2>	Boolean	Returns a value of true if <String1> is greater than <String2>. See Character precedence: Unicode & Java Collator on page 189 for character precedence.
Less than or Equal to		
<String1> <= <String2>	Boolean	Returns a value of true if <String1> is less than or equal to <String2>. See Character precedence: Unicode & Java Collator on page 189 for character precedence.
Greater than or Equal to		
<String1> >= <String2>	Boolean	Returns a value of true if <String1> is greater than or equal to <String2>. See Character precedence: Unicode & Java Collator on page 189 for character precedence.
Adding Strings		
<String1> + <String2>	String	Concatenates <String1> to <String2>. Alternative syntax.
Size		
<String>.size	String	Returns the number of characters in <String>.
Concatenate		
<String1> .concat(<String2>)	String	Concatenates <String1> to <String2>.
Uppercase		
<String>.toUpper	String	Converts all characters <String> to uppercase.
Lowercase		

Name and Syntax	Returns	Description
<String>.toLower	String	Converts all characters in <String> to lowercase.
To DateTime		
<String>.toDateTime	DateTime	Converts the value in <String> to data type DateTime ONLY if all characters in <String> correspond to a valid DateTime mask (format)
To Decimal		
<String>.toDecimal	Decimal	Converts an attribute of type String to data type Decimal ONLY if all characters in <String> are numeric and contain not more than one decimal point. If any non-numeric characters are present (other than a single decimal point or leading minus sign), no value is returned.
To Integer		
<String>.toInteger	Integer	Converts an attribute of type String to type Integer ONLY if all characters in <String> are numeric. If any non-numeric characters are present, no value is returned.
Substring		
<String> .substring (<Integer1>,<Integer2>)	String	Returns that portion of <String> between character positions <Integer1> and Integer2>.
Equals Ignoring Case		
<String1> .equalsIgnoreCase (<String2>)	Boolean	Returns a value of true if <String1> is the same as <String2>, irrespective of case.
Starts with		
<String1> .startsWith (<String2>)	Boolean	Returns a value of true if the <String1> begins with the characters specified in <String2>.
Ends with		
<String1> .endsWith (<String2>)	Boolean	Evaluates the contents of <String1> and returns a value of true if the String ends with the characters specified in <String2>.
Contains		

Name and Syntax	Returns	Description
<String1> .contains (<String2>)	Boolean	Evaluates the contents of <String1> and returns a value of true if it contains the exact characters defined by <String2>
Equals		
<String1> .equals (<String2>)	Boolean	Returns a value of true if <String1> is the same as <String2>.
Index Of		
<String1> .indexOf (<String2)	Integer	Returns the beginning character position number of <String2> within <String1>, if <String1> contains <String2>. If it does not, the function returns a value of zero.
containsBlanks		
<String>.containsBlanks	Boolean	Determines whether the specified String contains any blanks.
characterAt(index)		
<String>.characterAt(index:Integer)	String	Returns the character at the specified position in the String.
isInteger		
<String>.isInteger	Boolean	Determines whether "this" String contains only integer digits.
trimSpaces		
<String>.trimSpaces	String	Trims leading and trailing spaces from "this" String.
charsIn(validSet)		
<String>.charsIn(validSet:String)	Boolean	Determines whether "this" String contains only characters specified in the validSet.

Entity/Association operators

The Corticon rule language supports Entity and Association operators categorized as Entity, Collection, and Sequence.

Entity

Corticon's **Entity** operators are as follows:

Name and Syntax	Returns	Description
New		
<Entity> .new [<Expression1>,...]	Entity	Creates a new instance of <Entity>. Expressions (optional) in square brackets [..] must be written in the form: *attribute* = *value*.
New Unique		
<Entity> .newUnique [<Expression1>,...]	Entity	Creates a new instance of <Entity> only if the instance created is unique as defined by optional <Expression1>,...
Remove		
<Entity>.remove	Entity	Deletes the entity from memory and from the resultant XML document.

Collection

Corticon's **Collection** operators are as follows:

Name and Syntax	Returns	Description
Replace element(s)		
<Collection1> = <Collection2> <Collection1> = <Entity>	*modifies a collection*	replaces all elements in <Collection1> with elements of <Collection2> or with <Entity>, provided the new associations are allowed by the Business Vocabulary.
Associate element(s)		
<Collection1> += <Collection2> <Collection1> += <Entity>	*modifies a collection*	Associates all elements of <Collection2> or <Entity> with <Collection1>. Every <Collection> must be expressed as a unique alias.
Disassociate element(s)		
<Collection1> -= <Collection2>	*modifies a collection*	Disassociates all elements of <Collection2> from <Collection1>. Does not delete the disassociated elements. Every <Collection> must be expressed as a unique alias.

Name and Syntax	Returns	Description
Is empty		
<Collection> ->isEmpty	Boolean	Returns a value of true if <Collection> contains *no* elements
Not empty		
<Collection> ->notEmpty	Boolean	Returns a value of true if <Collection> contains *at least one* element.
Exists		
<Collection> ->exists (<Expression>)	Boolean	Returns a value of true if <Expression> holds true for *at least one* element of <Collection>
For all		
<Collection> ->forAll (<Expression>)	Boolean	Returns a value of true if *every* <Expression> holds true for *every* element of <Collection>
Sorted by		
<Collection> ->sortedBy (<Attribute>)	*converts a collection into a sequence*	Sequences the elements of <Collection> in ascending order, using the value of <Attribute> as the index. <Collection> must be expressed as a unique alias.
Sorted by descending		
<Collection> ->sortedByDesc (<Attribute>)	*converts a collection into a sequence*	Sequences the elements of <Collection> in descending order, using the value of <Attribute> as the index. <Collection> must be expressed as a unique alias.
Iterate		
<Collection> ->iterate(<Expression>)		Executes <Expression> for every element in <Collection>. <Collection> must be expressed as a unique alias.
Size of collection		
<Collection> ->size	Integer	Returns the number of elements in <Collection>. <Collection> must be expressed as a unique alias.
Sum		

Name and Syntax	Returns	Description
<Collection.attribute> ->sum	Number	Sums the values of the specified <attribute> for all elements in <Collection>. <attribute> must be a numeric data type.
Average		
<Collection.attribute> ->avg	Number	Averages all of the specified attributes in <Collection>. <Collection> must be expressed as a unique alias. <attribute> must be a numeric data type
Minimum		
<Collection.attribute> ->min	Number	Returns the lowest value of <attribute> for all elements in <Collection>. <attribute> must be a numeric data type
Maximum		
<Collection.attribute> ->max	Number	Returns the highest value of <attribute> for all elements in <Collection>. <attribute> must be a numeric data type
toSet		
Collection.toSet	String	Returns a single String that is the set of Strings in this collection.
allContain(lookFor)		
Collection.allContain (lookFor:String)	Boolean	Determines whether all the strings in this collection contain the lookFor String
uniqueCount		
Collection.uniqueCount	Integer	Returns the count of the unique Strings in this collection.

Sequence

Sequence operators act on collections that have *already* been ordered by a sorting operator (see sortedBy and sortedByDesc). In other words, sequence operators operate on collections that have been turned into sequences. The notation <Sequence> used below, is shorthand for a completed sorting operation. For example:

<Collection> -> sortedBy(<Attribute>)

produces a <Sequence>, in this case the elements of <Collection> arranged in ascending order using <Attribute> as the index. This <Sequence> can then be used with one of the sequence operators described below. The design of the Object Constraint Language (upon which the Corticon Rule Language is based), allows for the "chaining" of operators, so a collection operator and a sequence operator can be used in the same expression to produce a sequence and identify a particular element of that sequence in the same step. For example:

<Entity.attribute1> = <Collection> sortedBy(<Attribute3>) first.<Attribute2>

performs the following:

1. Sorts <Collection> in ascending order according to <Attribute3>, turning it into a <Sequence>
2. Locates the first element of <Sequence>
3. Reads the value of <Attribute2> of the first element
4. Assigns the value of <Attribute2> of the first element to <Entity.attribute1>

Corticon's **Sequence** operators are as follows:

	Name and Syntax	Returns	Description
At			
	<Sequence> ->at(<Integer>)	Entity	Returns the element at position <Integer>. <Sequence> must be expressed as a unique alias.
First			
	<Sequence> ->first	Entity	Returns the first element of <Sequence>. <Sequence> must be expressed as a unique alias.
Last			
	<Sequence> ->last	Entity	Returns the last element of <Sequence>. <Sequence> must be expressed as a unique alias.
Trend			
	<Attribute> -> <Sequence>.trend	String	Returns a 4-character string, INCR, DECR, CNST, or NONE depending on the trend of <Attribute> within <Sequence>.
mavg(elements)			
	<Sequence.decimal> .mavg(elements:Integer)	Decimal	Returns a single decimal value that is the average of the number of elements specified.

	Name and Syntax	Returns	Description
Sorted Alias: next			
	`->next`		Operates against a Sorted Alias (a special cached Sequence) inside a filter expression. The Rulesheet is set into a Ruleflow that iterates to bind the alias in each successive invocation to the next element in the sequence. For more information, see the topic *"Sorted Alias" in the Collections chapter of the Corticon Studio: Rule Modeling Guide.*.

Extended Operators

Corticon's **Extended** operators are categorized as **RandomGenerator** and **SeMath**.

RandomGenerator

RandomGenerator does just what it says: it generates -- and then returns -- a random number.

Corticon's **RandomGenerator** operator is as follows:

Name and Syntax	Returns	Description
getRandomNumber		
`RandomGenerator.getRandomNumber`	Decimal	Returns the next random number out of the current static Random object.

SeMath

Corticon's **SeMath** operators are as follows:

Name and Syntax	Returns	Description
getCircumference		
`SeMath.getCircumference (radius:<Decimal>)`	Decimal	Converts the specified radius to circumference.
getFahrenheit		
`SeMath.getFahrenheit (centigrade:<Decimal>)`	Decimal	Converts the specified Centigrade temperature to Fahrenheit.

Name and Syntax	Returns	Description
replaceString		
`SeMath.replaceString (lookin:<String>,lookfor:<String>,replacewith:<String>)`	String	Replaces all occurences of a specified string with another string.

4

Rule operator details and examples

The following pages describe each operator in greater detail. Each Rule Operator has the following sections

1. **Syntax** – Describes the standard syntax used with this operator. In this section, as in the previous summary tables, the angle bracket convention `<..>` is used to indicate what types of terms and their data types can be used with the operator. When using the operator with real terms from the Vocabulary, do not include the angle brackets.
2. **Description** – Provides a plain-language description of the operator's purpose and details of its use. Important reminders, tips, or cautions are included in this section.
3. **Usage Restrictions** – Describes what limitations exist for this operator, and where an operator may not be used in a *`Rulesheet`*. Such limitations are rare, but important to a good understanding of Corticon Studio.
4. **Example** – Shows an example of each operator in a *`Rulesheet`*. A screenshot of the example *`Rulesheet`* is provided, with portions of the *`Rulesheet`* not used by the example collapsed or truncated for clarity. The example also includes sample input and output data for *`Ruletest`* scenarios run against the *`Rulesheet`*.

The entire list of operators is presented in alphabetic order.

For details, see the following topics:

- Absolute value
- Add numbers
- Add strings
- Add days
- Add hours

- Add minutes
- Add months
- Add seconds
- Add years
- Associate element(s)
- At
- Average
- CellValue
- Concatenate
- Contains
- Day
- Days between
- Day of week
- Day of year
- Decrement
- Disassociate element(s)
- Divide
- Div
- Ends with
- Equals – used as an assignment
- Equals – used as a comparison
- Equals ignoring case
- Equals – strings only
- Exists
- Exponent
- False
- First
- Floor
- For all
- Greater than
- Greater than or equal to
- Hour
- Hour between

- Increment
- Index of
- Is empty
- Iterate
- Last
- Less than
- Less than or equal to
- Logarithm (Base 10)
- Logarithm (Base x)
- Lowercase
- Maximum value
- Maximum value (Collection)
- Minimum value
- Minimum value (Collection)
- Minute
- Minutes between
- Mod
- Month
- Months between
- Multiply
- Natural logarithm
- New
- New unique
- Not
- Not empty
- Not equal to
- Now
- Null
- Other
- Or
- Remove element
- Replace element(s)
- Round

- Second
- Seconds between
- Size of string
- Size of collection
- Sorted by
- Sorted by descending
- Starts with
- Substring
- Subtract
- Sum
- Today
- To date – casting a dateTime to a date
- To dateTime – casting a string to a dateTime
- To dateTime – casting a date to a dateTime
- To dateTime – casting a time to a dateTime
- To dateTime – timezone offset
- To decimal
- To integer
- To string
- To time – casting a dateTime to a time
- Trend
- True
- Uppercase
- Week of month
- Week of year
- Year
- Years between

Absolute value

SYNTAX

```
<Number>.absVal
```

DESCRIPTION

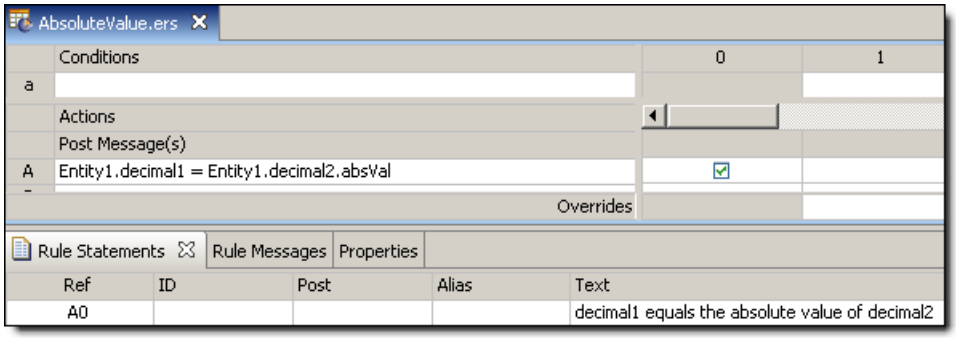
Returns the absolute value of <Number>. If the <Number> is positive, <Number> itself is returned; if <Number> is negative, the negation of <Number> is returned.

USAGE RESTRICTIONS

The Operators row in the table of Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

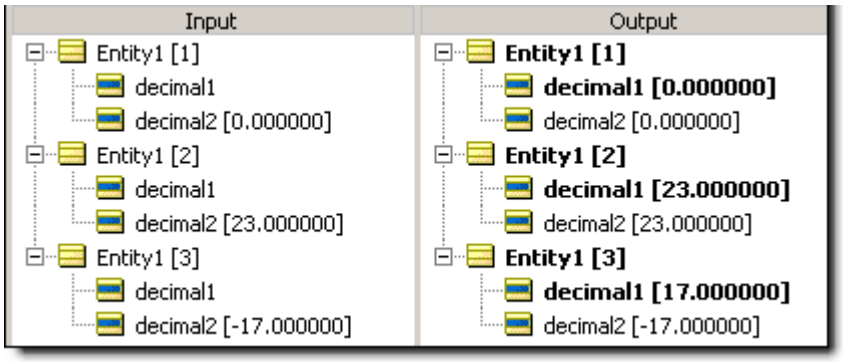
RULESHEET EXAMPLE

This sample *Rulesheet* uses **.absVal** to produce the absolute value of decimal2 and assign it to decimal1



SAMPLE RULETEST

A sample *Ruletest* provides decimal2 values for three different scenarios of Entity1. Input and Output panels are shown below.



Add numbers

SYNTAX

<Number1> + <Number2>

DESCRIPTION

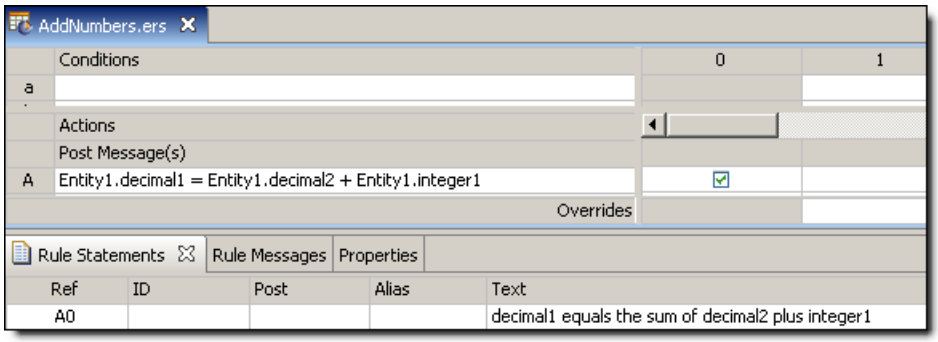
Adds <Number1> to <Number2>. The resulting data type is the more expansive of those of <Number1> and <Number2>. For example, if you are adding an Integer value and a Decimal value, the resulting value will be a Decimal. See Arithmetic operator precedence on page 193.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

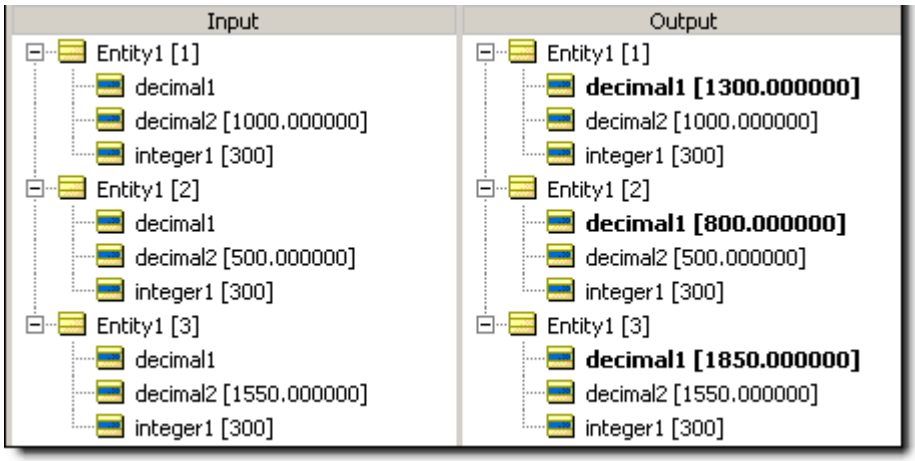
RULESHEET EXAMPLE

This sample Rulesheet uses the **add numbers** operation to add the value of `decimal2` to the value of `integer1` and assign the result to `decimal1`



SAMPLE RULETEST

A sample Ruletest provides an `integer1` value of `300` which is added to the value of `decimal2` and assigned to the value of `decimal1` for three instances of `Entity1`. Input and Output panels are shown below.



Add strings

SYNTAX

<String1> + <String2>

DESCRIPTION

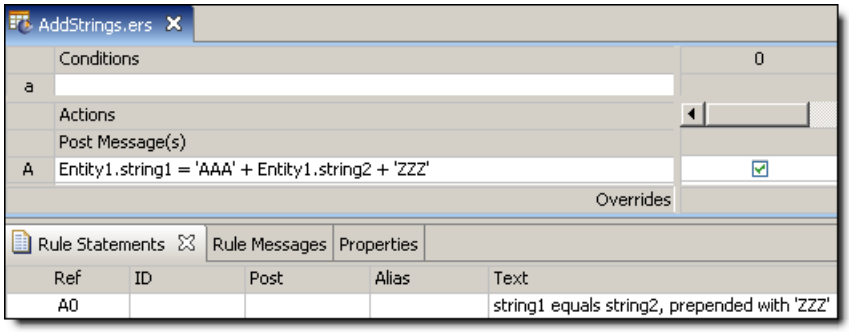
Adds <`String1`> to <`String2`>. This has the same effect as using the .concat operator. However, the "+" syntax permits concatenation of more than two String values without nesting, as shown in the example below.

USAGE RESTRICTIONS

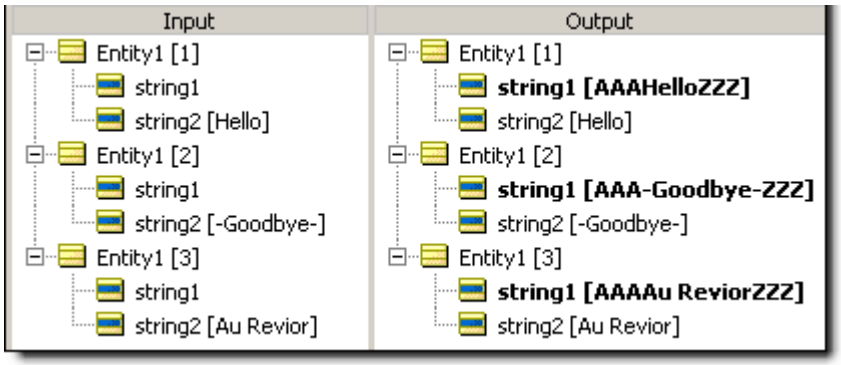
The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

RULESHEET EXAMPLE

This sample Rulesheet uses **add strings** operation to add the String `AAA` to `string2` to `ZZZ` and assign the result to `string1`



SAMPLE RULETEST



Add days

SYNTAX

<`DateTime`>.addDays(<`Integer`>)

<`Date`>.addDays(<`Integer`>)

DESCRIPTION

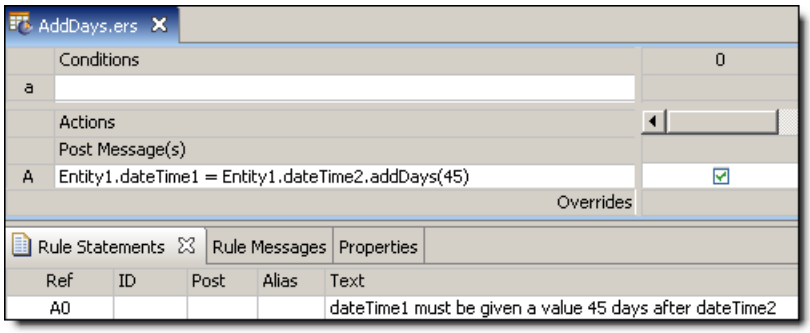
Adds the number of days in <`Integer`> to the number of days in <`DateTime`> or <`Date`>.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

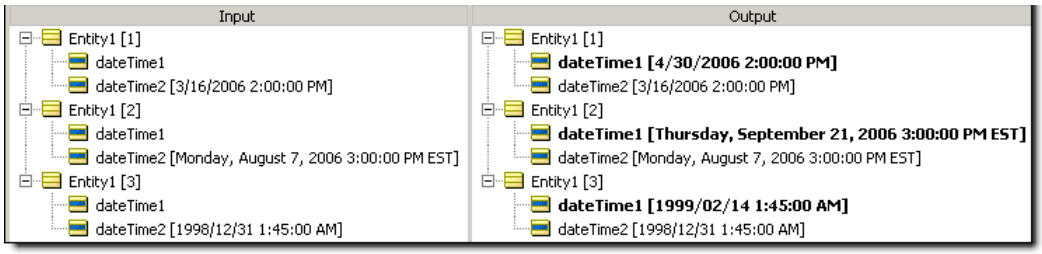
RULESHEET EXAMPLE

This sample Rulesheet uses **.addDays** to add `45` days to the value of `dateTime2` and assign the result to `dateTime1`.



SAMPLE *RULETEST*

A sample Ruletest provides values of `dateTime2` for three instances of `Entity1`. Input and Output panels are shown below. Notice the month portion of `dateTime1` also changes accordingly.



Add hours

SYNTAX

`<DateTime>`.addHours(`<Integer>`)

`<Time>`.addHours(`<Integer>`)

DESCRIPTION

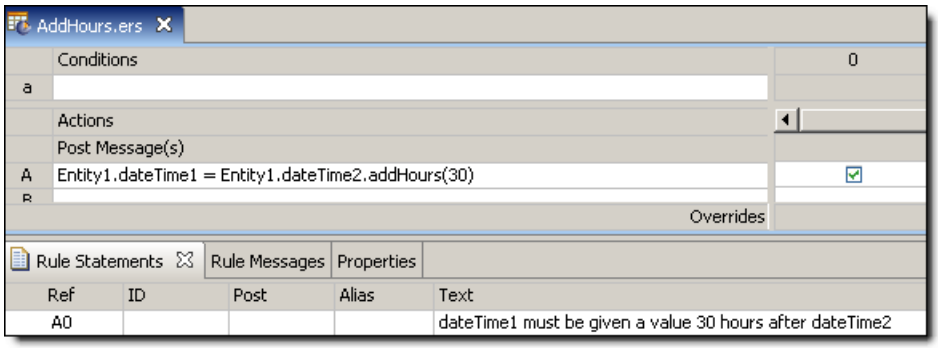
Adds the number of hours in `<Integer>` to the number of hours in the Time portion of `<DateTime>` or `<Time>`.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

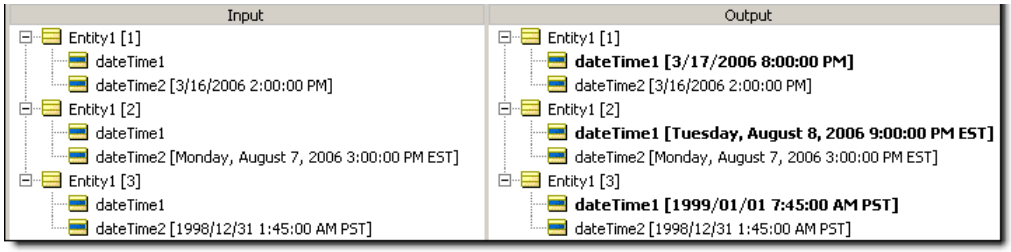
RULESHEET EXAMPLE

This sample Rulesheet uses the **. addHours** to add `30` hours to the value of `dateTime2` and assign the result to `dateTime1`.



SAMPLE RULETEST

A sample Ruletest provides values of `dateTime2` for three instances of `Entity1`. Input and Output panels are shown below.



Add minutes

SYNTAX

`<DateTime>`.addMinutes(`<Integer>`)

`<Time>`.addMinutes(`<Integer>`)

DESCRIPTION

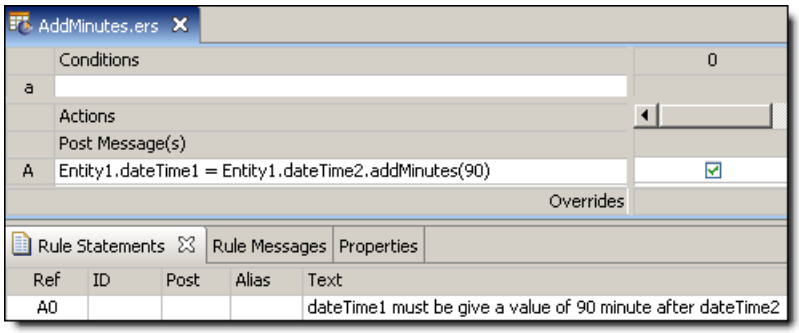
Adds the number of minutes in `<Integer>` to the number of minutes in the Time portion of `<DateTime>` or `<Time>`.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

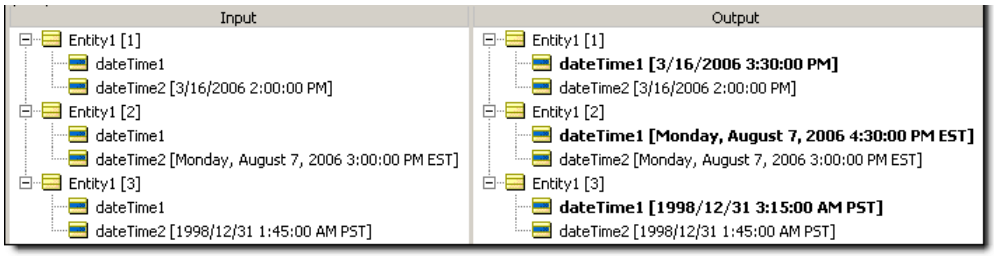
RULESHEET EXAMPLE

This sample Rulesheet uses the **. addMinutes** add `90` minutes to the value of `dateTime2` and assign the result to `dateTime1`.



SAMPLE RULETEST

A sample Ruletest provides values of `dateTime2` for three instances of `Entity1`. Input and Output panels are shown below.



Add months

SYNTAX

`<DateTime>`.addMonths(`<Integer>`)

`<Date>`.addMonths(`<Integer>`)

DESCRIPTION

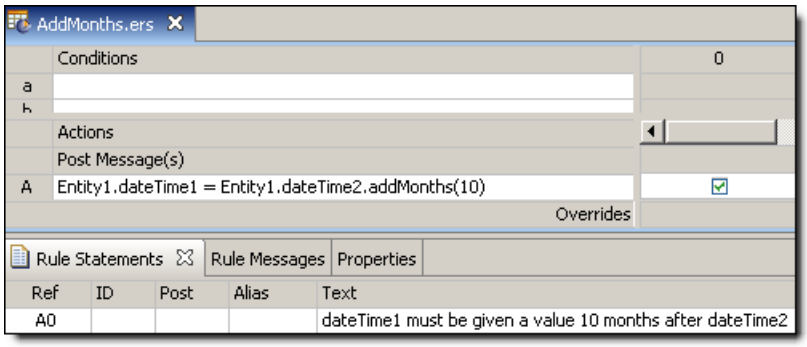
Adds the number of months in `<Integer>` to the number of months in `<DateTime>` or `<Date>`.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

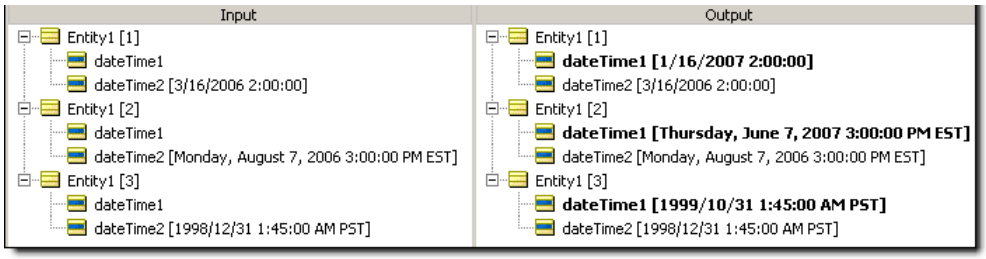
RULESHEET EXAMPLE

This sample Rulesheet uses **. addMonths** in a Nonconditional rule to add `10` months to the value of `dateTime2` and assign the result to `dateTime1`.



SAMPLE RULETEST

A sample Ruletest provides values of `dateTime2` for three instances of `Entity1`. Input and Output panels are shown below. Notice the year portion of `dateTime1` also changes accordingly.



Add seconds

SYNTAX

`<DateTime>`.addSeconds(`<Integer>`)

`<Time>`.addSeconds(`<Integer>`)

DESCRIPTION

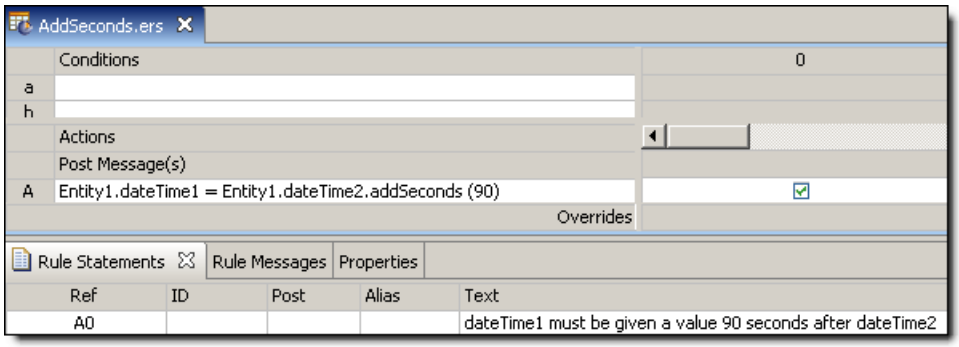
Adds the number of seconds in `<Integer>` to the number of seconds in the Time portion of `<DateTime>` or `<Time>`.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

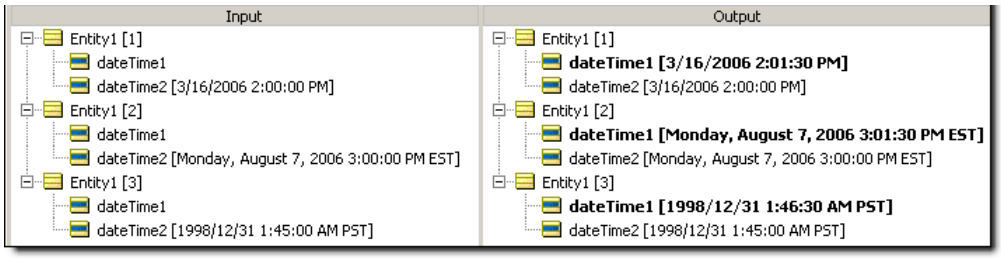
RULESHEET EXAMPLE

This sample Rulesheet uses **.addSeconds** in a Nonconditional rule to add `90` seconds to the value of `timeOnly2` and assign the result to `timeOnly1`.



SAMPLE RULETEST

A sample Ruletest provides values of `timeOnly2` for three instances of `Entity1`. Input and Output panels are shown below. Notice how the time "wraps" around to the beginning of the day, even though Time data type does not include date information.



Add years

SYNTAX

`<DateTime>`.addYears(`<Integer>`)

`<Date>`.addYears(`<Integer>`)

DESCRIPTION

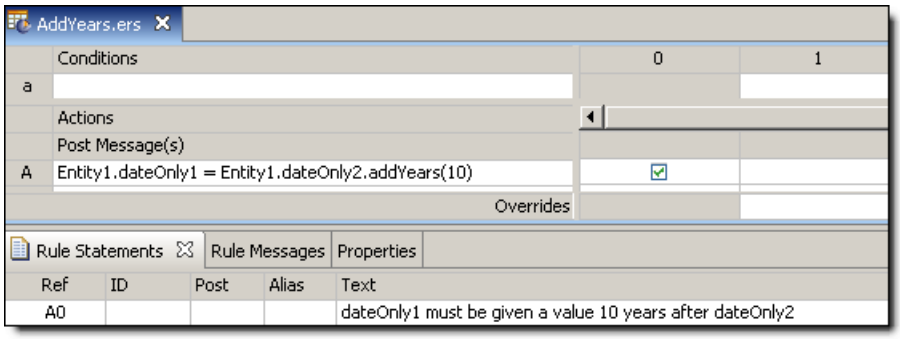
Adds the number of years in `<Integer>` to the number of years in the Date portion of `<DateTime>` or `<Date>`.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

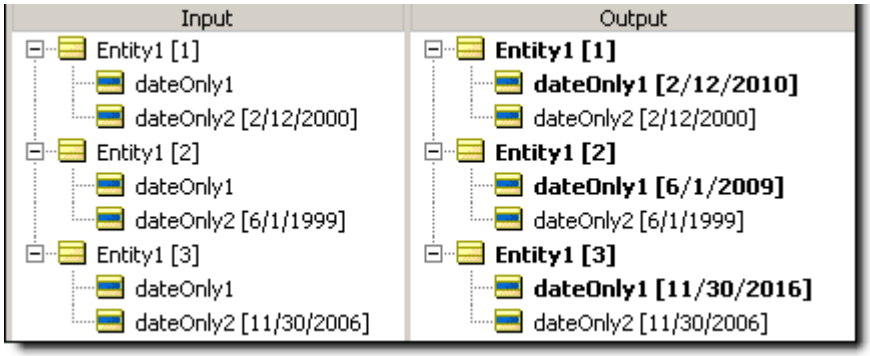
RULESHEET EXAMPLE

This sample Rulesheet uses **.addYears** in a Nonconditional rule to add `10` years to the value of `dateOnly2` and assign the result to `dateOnly1`.



SAMPLE RULETEST

A sample Ruletest provides values of `dateOnly2` for three instances of `Entity1`. Input and Output panels are shown below.



Associate element(s)

SYNTAX

`<Collection1> += <Collection2>`

`<Collection1> += <Entity>`

DESCRIPTION

Associates all elements of `<Collection2>` or a single element named `<Entity>` with `<Collection1>`, provided such an association is allowed by the Vocabulary. Every `<Collection>` must be expressed as a unique alias.

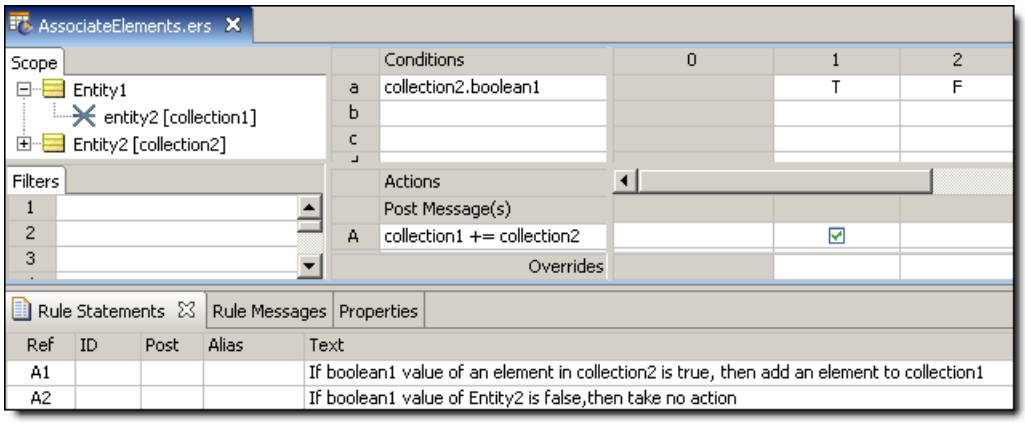
If the cardinality of the association between the parent entity of `<Collection>` and the `<Entity>` being added is "one-to-one" (a straight line icon beside the association in the Rule Vocabulary), then this **associate element** syntax is not used. Instead, replace element syntax is used, since the collection can contain only one element, and any element present will be replaced by the new element.

USAGE RESTRICTIONS

The Operators row of the table in Vocabulary usage restrictions does not apply. Special exceptions: **associate element** may only be used in Action Rows (section 5 in Sections of Rulesheet that correlate with usage restrictions).

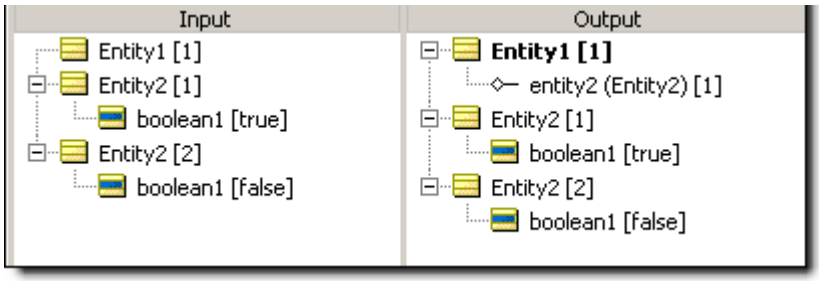
RULESHEET EXAMPLE

The following Rulesheet uses **associate element** to associate an element of `collection2` to `collection1` when `boolean1` value of any element in `collection2` is `true`. Note that the Action is not associating *all* elements in `collection2` with `collection1`, *only* those elements within `collection2` that satisfy the condition.



SAMPLE RULETEST

A sample Ruletest provides two examples of `Entity2` with `boolean1` values, and a single `Entity1`. Input and Output panels are described below:



At

SYNTAX

`<Sequence>` ->at(`<Integer>`).`<Attribute1>`

DESCRIPTION

Returns the value of `<Attribute1>` for the element at position `<Integer>` in `<Sequence>`. Another operator, such as sortedBy, must be used to transform a `<Collection>` into a `<Sequence>` before **at** may be used. `<Sequence>` must be expressed as a unique alias. See Advanced Collection Syntax for more examples of usage.

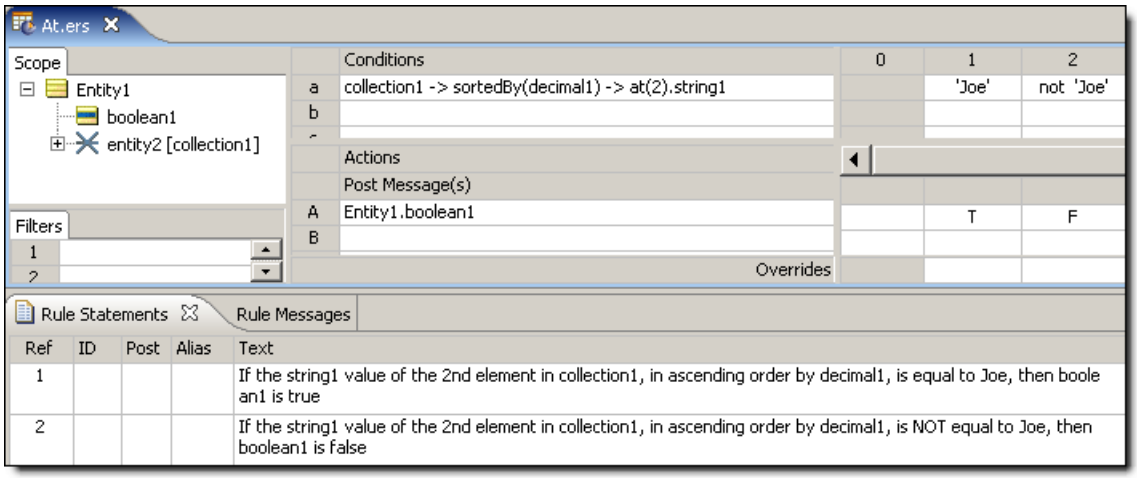
`<Attribute1>` may be of any data type.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

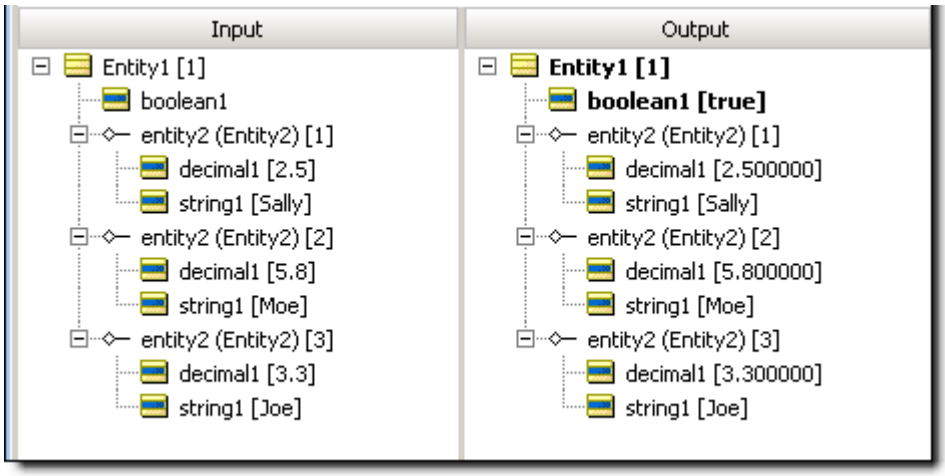
RULESHEET EXAMPLE

This sample Rulesheet uses **->at(2)** to identify the second element of the sequence created by applying sortedBy to `collection1` . Once identified, the value of the `string1` attribute belonging to this second element is evaluated. If the value of `string1` is `Joe`, then `boolean1` attribute of `Entity1` is assigned the value of `true` .



SAMPLE RULETEST

A sample Ruletest provides a collection of three elements, each with a `decimal1` value. Input and Output panels are shown below.



Average

SYNTAX

`<Collection.attribute>` ->avg

DESCRIPTION

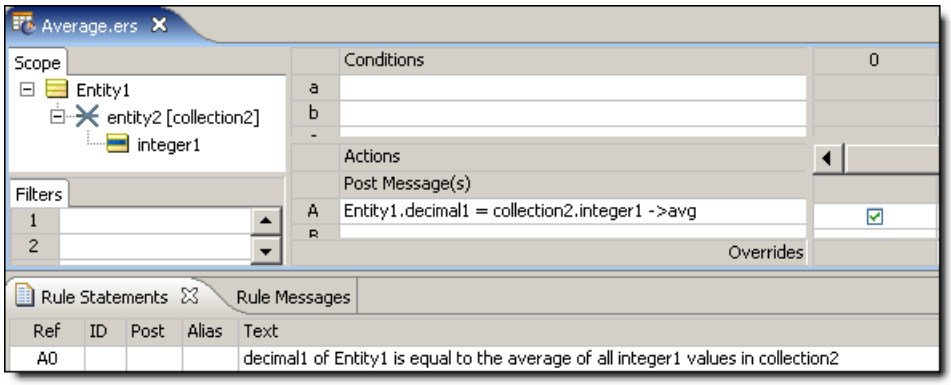
Averages the values of all of the specified attributes in <`Collection`>. <`Collection`> must be expressed as a unique alias. <`attribute`> must be a numeric data type.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

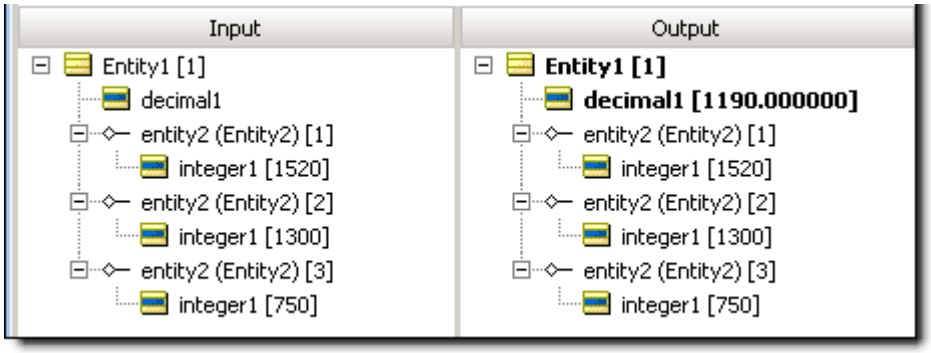
RULESHEET EXAMPLE

This sample Rulesheet uses **->avg** to average the `integer1` values of all elements in `collection2`, then assigns the resulting value to `decimal1` in `Entity1`. Note the use of the alias `collection2` to represent the collection of `Entity2` elements associated with `Entity1`.



SAMPLE RULETEST

A sample Ruletest provides `integer1` values for three elements in `collection2`. The following illustration shows Input and Output panels:



CellValue

SYNTAX

Various, see Examples below

DESCRIPTION

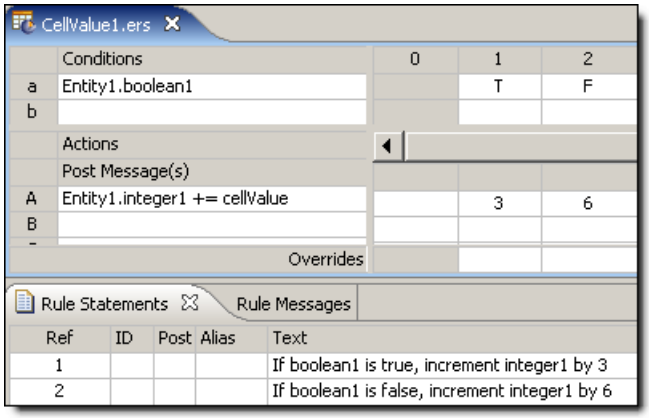
When used in an expression, **cellValue** is essentially a variable whose value is determined by the rule Column that executes. Using **cellValue** in a Condition or Action expression eliminates the need for multiple, separate Rows to express the same logic.

USAGE RESTRICTIONS

The Operators row of the table in Vocabulary usage restrictions does not apply. Special exceptions: **cellValue** may only be used in Condition and Action Rows (sections 3 and 5 in Sections of Rulesheet that correlate with usage restrictions).

RULESHEET EXAMPLE 1

This sample Rulesheet uses **cellValue** to increment `integer1` by the amount in the Action Cell of the rule Column that fires. An equivalent Rulesheet which does not use cellValue is also shown for comparison purposes.



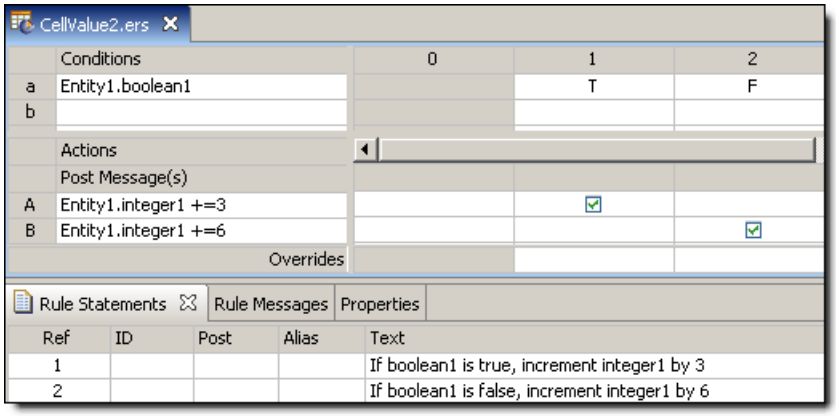
CellValue1.ers

	Conditions	0	1	2
a	Entity1.boolean1		T	F
b				
	Actions			
	Post Message(s)			
A	Entity1.integer1 += cellValue		3	6
B				
	Overrides			

Rule Statements | Rule Messages

Ref	ID	Post	Alias	Text
1				If boolean1 is true, increment integer1 by 3
2				If boolean1 is false, increment integer1 by 6

Equivalent Rulesheet without using **cellValue**:



CellValue2.ers

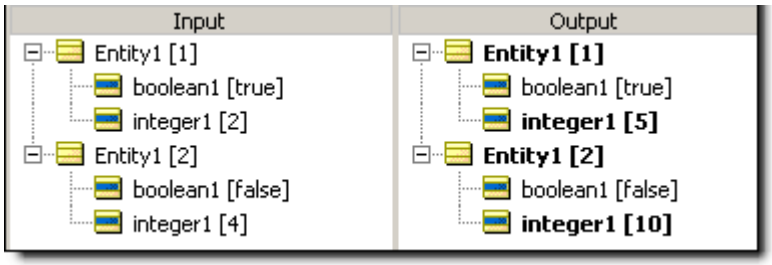
	Conditions	0	1	2
a	Entity1.boolean1		T	F
b				
	Actions			
	Post Message(s)			
A	Entity1.integer1 +=3		☑	
B	Entity1.integer1 +=6			☑
	Overrides			

Rule Statements | Rule Messages | Properties

Ref	ID	Post	Alias	Text
1				If boolean1 is true, increment integer1 by 3
2				If boolean1 is false, increment integer1 by 6

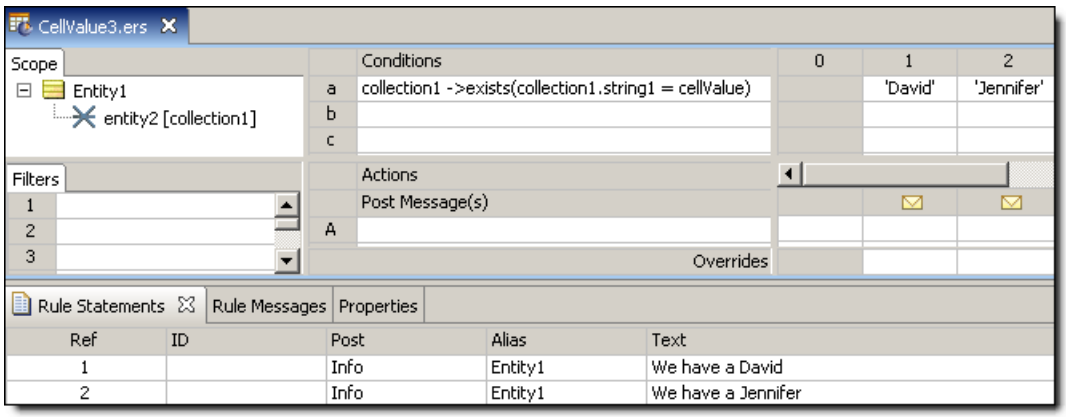
SAMPLE RULETEST 1

A sample Ruletest provides two examples of `boolean1`. The following table shows Input and Output panels.



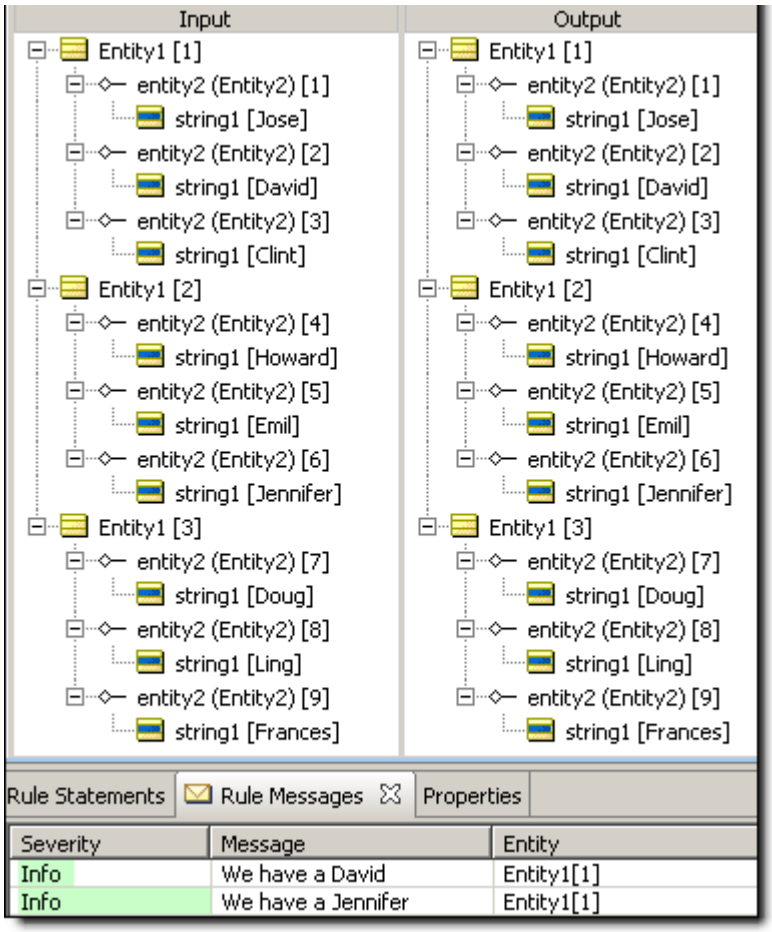
RULESHEET EXAMPLE 2

The following Rulesheet uses **cellValue** to evaluate whether `collection1` includes at least one member with a `string1` value of the entry in the Conditions Cell of the rule Column.



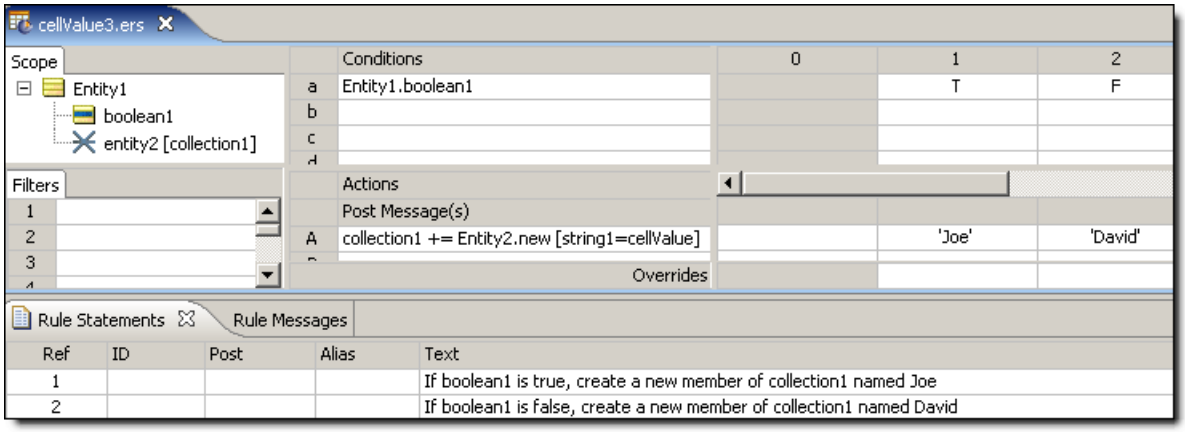
SAMPLE RULETEST 2

A sample Ruletest provides three examples of `collection1` – each member has a `string1` value. Input and Output panels are shown below.



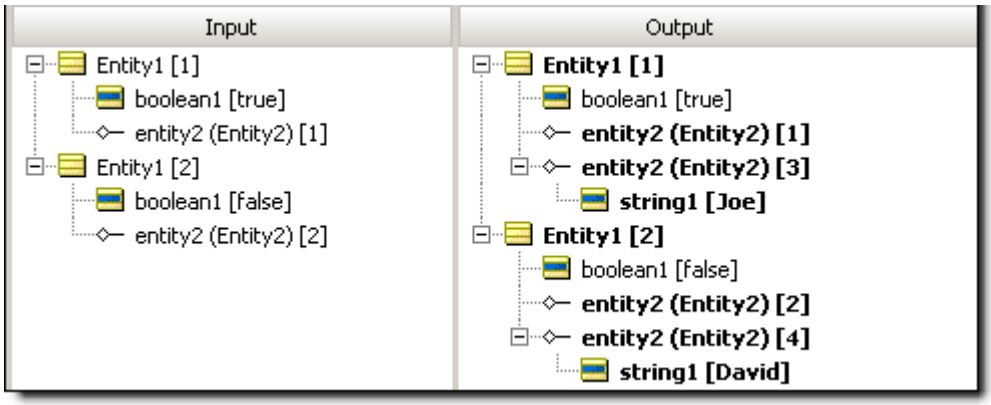
RULESHEET EXAMPLE 3

The following Rulesheet uses **cellValue** to create a new member of `collection1` with `string1` value equal to the Action Cell in the rule Column that fires.



SAMPLE RULETEST 3

A sample Ruletest provides `string1` values for three examples. The following illustration shows Ruletest Input and Output panels. Notice that each `collection1` already has one element prior to executing the test. This simply ensures the results will be displayed in hierarchical style.



Concatenate

SYNTAX

`<String1>`.concat(`<String2>`)

DESCRIPTION

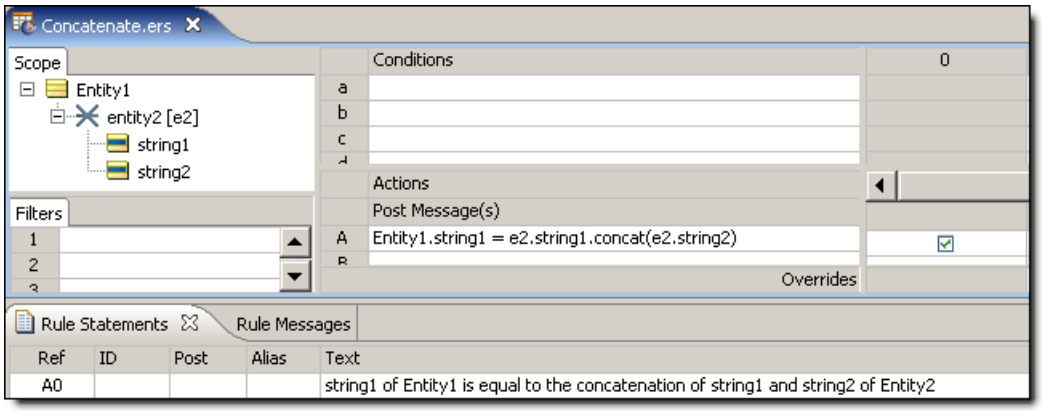
Concatenates `<String1>` to `<String2>`, placing `<String2>` at the end of `<String1>`

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions

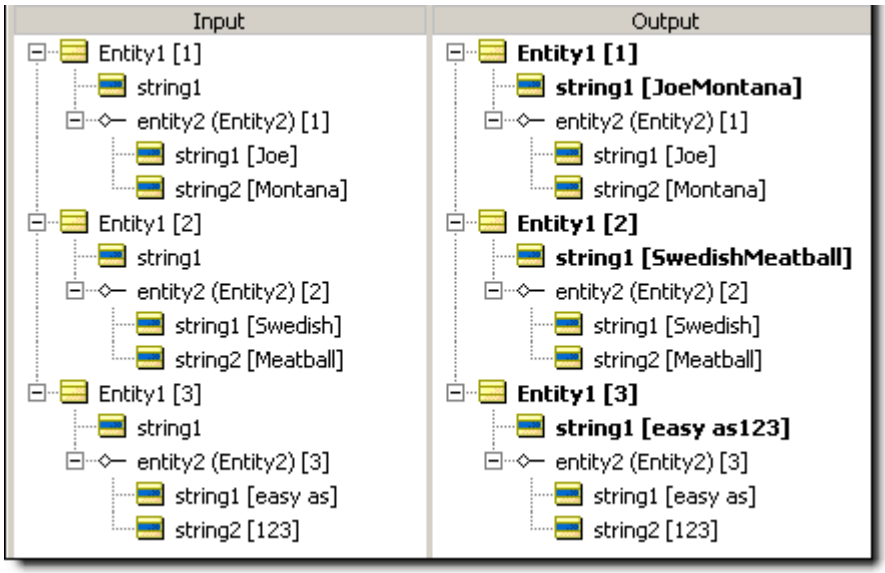
RULESHEET EXAMPLE

This sample Rulesheet uses **.concat** to create `string1` by combining `string1` and `string2` from `Entity1.entity2`.



SAMPLE RULETEST

A sample Ruletest provides three examples of `string1` and `string2`. Input and Output panels are shown below.



Contains

SYNTAX

`<String1>`.contains(`<String2>`)

DESCRIPTION

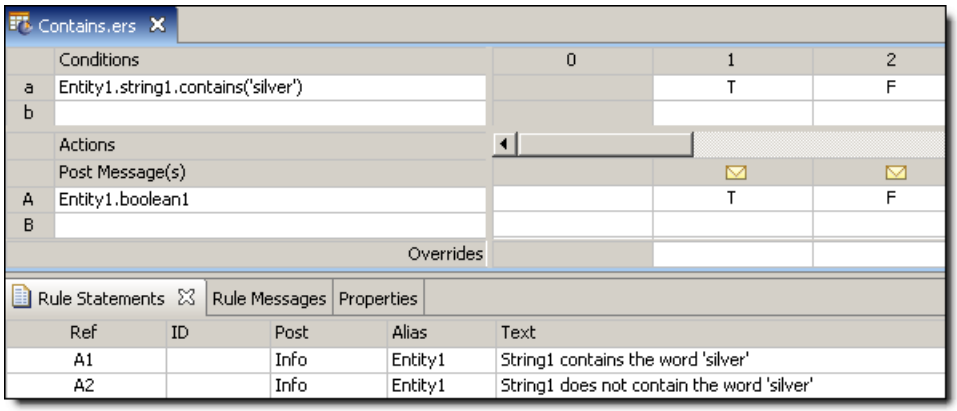
Evaluates `<String1>` and returns a value of true if it contains or includes the exact (case-sensitive) characters specified in `<String2>`.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions

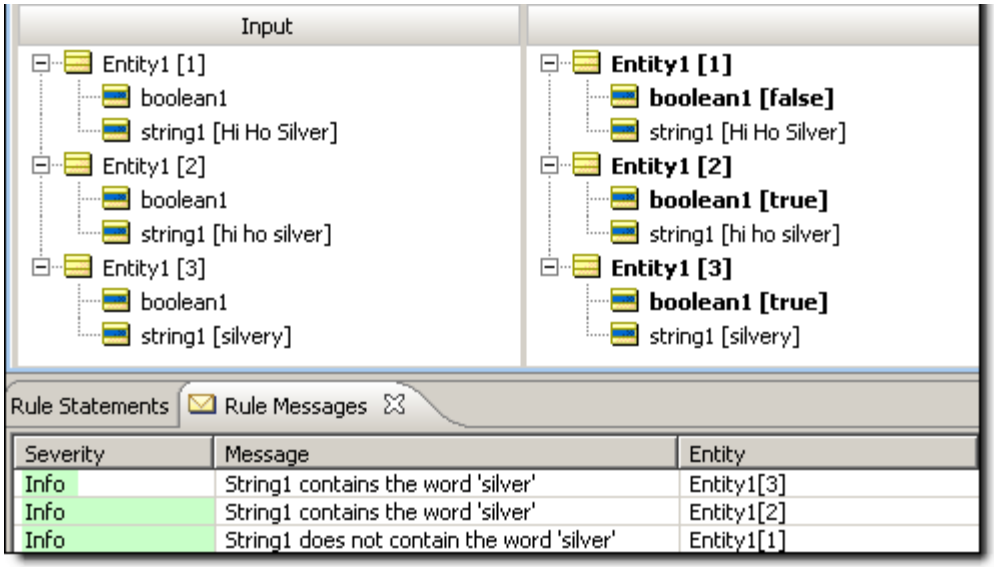
RULESHEET EXAMPLE 1

The following uses **.contains** to evaluate whether `string1` includes the characters `silver` and assigns a value to `boolean1` for each outcome.



SAMPLE RULETEST 1

A sample Ruletest provides `string1` values for three examples. Input and Output panels are shown below. Note case sensitivity in these examples. Posted messages are not shown.



Day

SYNTAX

`<DateTime>`.day

`<Date>`.day

`

DESCRIPTION

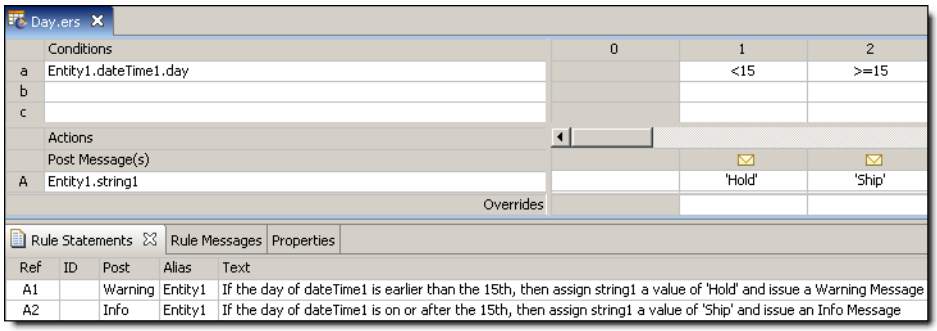
Returns the day portion of `<DateTime>` or `<Date>` as an Integer between 1 and 31.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

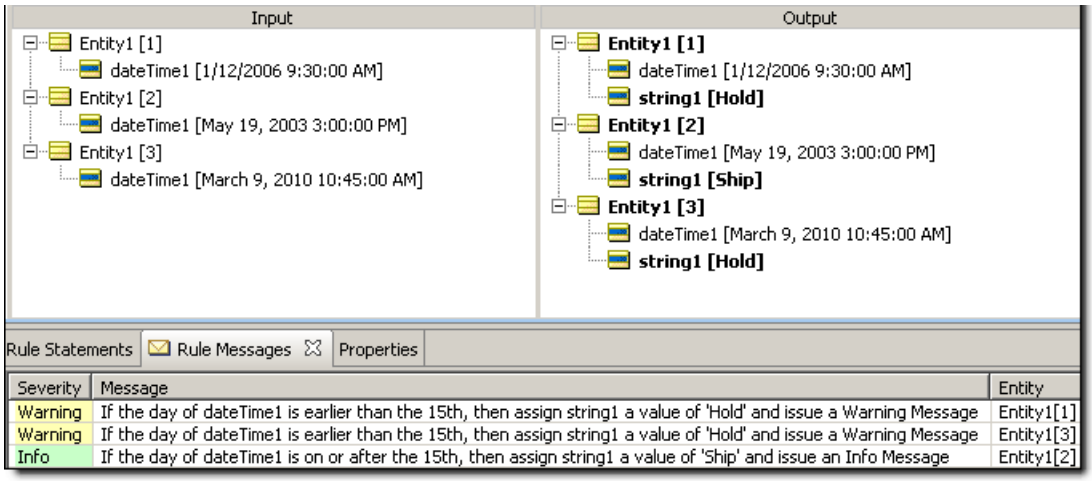
RULESHEET EXAMPLE

The following Rulesheet uses **.day** to assign a value to `string1` and post a message.



SAMPLE RULETEST

A sample Ruletest provides `dateTime1` values for three examples. Input and Output panels are shown below. Posted messages are not shown.



Days between

SYNTAX

`<DateTime1>`.daysBetween(`<DateTime2>`)

`<Date1>`.daysBetween(`<Date2>`)

DESCRIPTION

Returns the Integer number of days between DateTimes or Dates. This function calculates the number of milliseconds between the date values and divides that number by 86,400,000 (the number of milliseconds in a day). Any fraction is truncated, leaving an Integer result. If the two dates differ by less than a full 24-hour period, the value returned is zero. A positive Integer value is returned when `<DateTime2>` occurs after `<DateTime1>`.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions

RULESHEET EXAMPLE

The following Rulesheet uses **.daysBetween** to determine the number of days that have elapsed between `dateTime1` and `dateTime2`, compare it to the values in the Condition cells, and assign a value to `string1`.

DaysBetween.ers

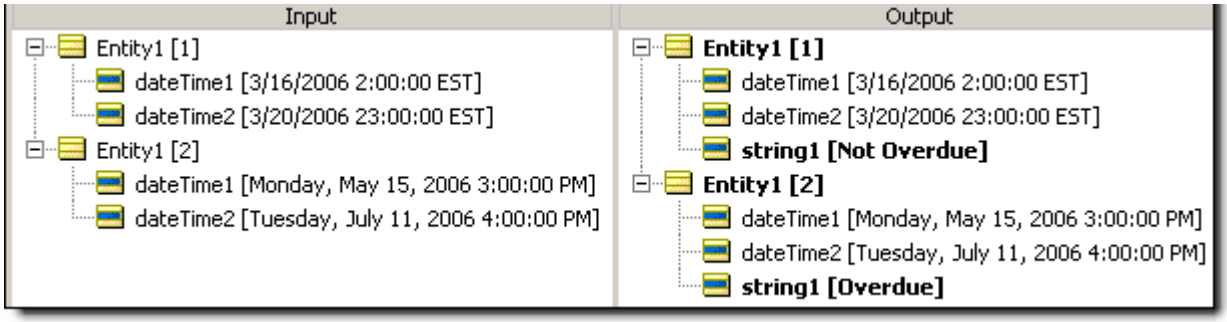
	Conditions	0	1	2
a	Entity1.dateTime1.daysBetween(Entity1.dateTime2)		<=30	>30
b				
	Actions			
	Post Message(s)			
A	Entity1.string1		'Not Overdue'	'Overdue'
-				
	Overrides			

Rule Statements | Rule Messages | Properties

Ref	ID	Post	Alias	Text
A1				If 30 days or fewer have elapsed between dateTime1 and dateTime2, then Entity1 is not overdue
A2				If more than 30 days have elapsed between dateTime1 and dateTime2, then Entity 1 is overdue

SAMPLE RULETEST

A sample Ruletest provides `dateTime1` and `dateTime2` for two examples. Input and Output panels are shown below.



Day of week

SYNTAX

`<DateTime>`.dayOfWeek

`<Date>`.dayOfWeek

DESCRIPTION

Returns an Integer between 1 and 7, corresponding to the table below:

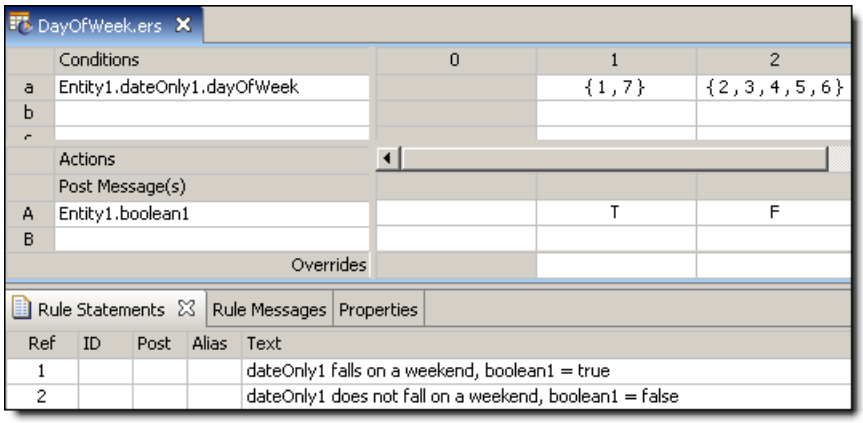
returned Integer	day of the week
1	Sunday
2	Monday
3	Tuesday
4	Wednesday
5	Thursday
6	Friday
7	Saturday

USAGE RESTRICTIONS

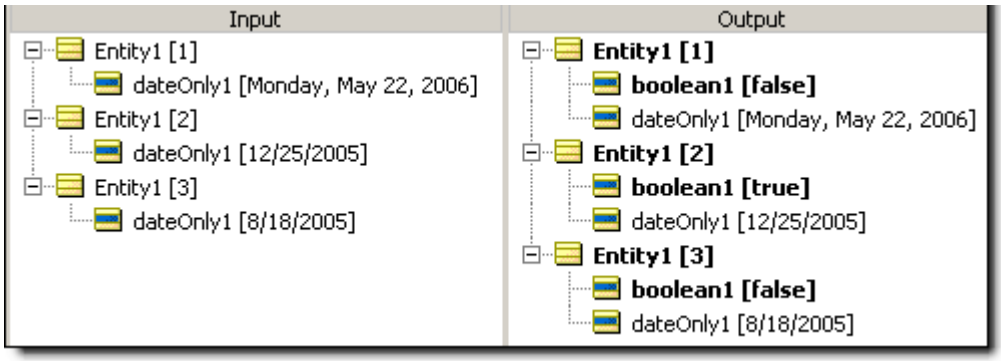
The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions

RULESHEET EXAMPLE

The following Rulesheet uses **.dayOfWeek** to assign a value to `boolean1`.



SAMPLE RULETEST



Day of year

SYNTAX

`<DateTime>`.dayOfYear

`<Date>`.dayOfYear

DESCRIPTION

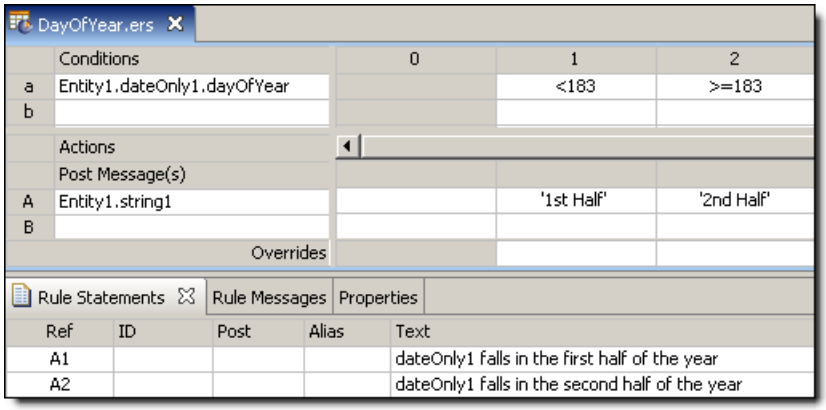
Returns an Integer from 1 to 366, equal to the day number within the year.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions

RULESHEET EXAMPLE

The following Rulesheet uses **.dayOfYear** to assign a value to `string1`.



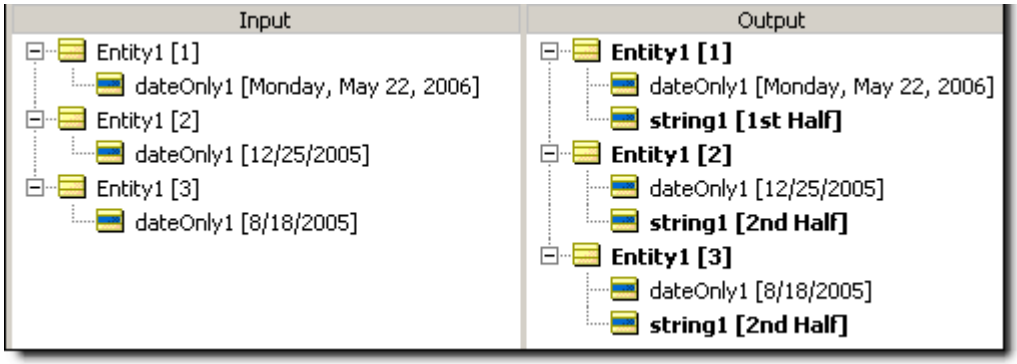
DayOfYear.ers

	Conditions	0	1	2
a	Entity1.dateOnly1.dayOfYear		<183	>=183
b				
	Actions			
	Post Message(s)			
A	Entity1.string1		'1st Half'	'2nd Half'
B				
	Overrides			

Rule Statements | Rule Messages | Properties

Ref	ID	Post	Alias	Text
A1				dateOnly1 falls in the first half of the year
A2				dateOnly1 falls in the second half of the year

SAMPLE RULETEST



Input	Output
Entity1 [1]	**Entity1 [1]**
dateOnly1 [Monday, May 22, 2006]	dateOnly1 [Monday, May 22, 2006]
Entity1 [2]	**string1 [1st Half]**
dateOnly1 [12/25/2005]	**Entity1 [2]**
Entity1 [3]	dateOnly1 [12/25/2005]
dateOnly1 [8/18/2005]	**string1 [2nd Half]**
	Entity1 [3]
	dateOnly1 [8/18/2005]
	string1 [2nd Half]

Decrement

SYNTAX

`<Number1> -= <Number2>`

DESCRIPTION

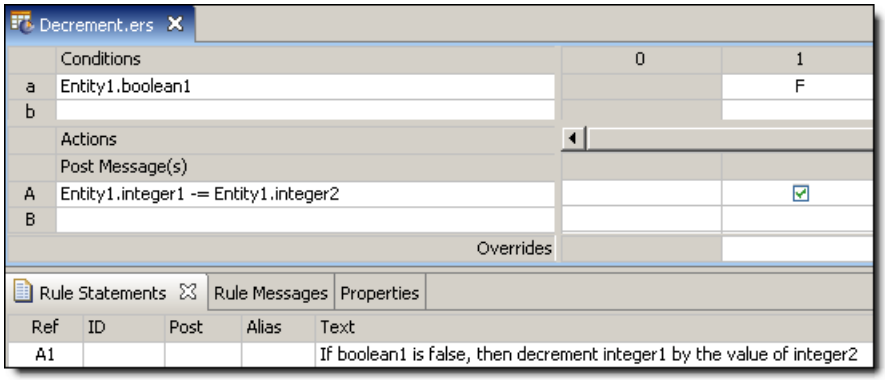
Decrements `<Number1>` by the value of `<Number2>`. The data type of `<Number1>` must accommodate the subtraction of `<Number2>`. In other words, an Integer may not be decremented by a Decimal without using another operator (such as .toInteger or Floor on page 91) to first convert the Decimal to an Integer.

USAGE RESTRICTIONS

The Operators row of the table in Vocabulary usage restrictions does not apply. Special exceptions: **decrement** may only be used in Action Rows (section 5 in Sections of Rulesheet that correlate with usage restrictions).

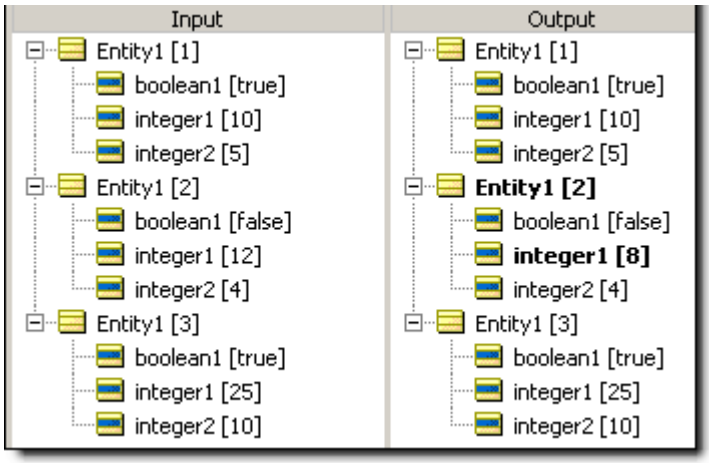
RULESHEET EXAMPLE

This sample Rulesheet uses **decrement** to reduce `integer1` by the value of `integer2` when `boolean1` is `false`.



SAMPLE RULETEST

A sample Ruletest provides three examples of `integer1`, `integer2`, and `boolean1`. Input and Output panels are shown below.



Disassociate element(s)

SYNTAX

`<Collection1> -= <Collection2>`

DESCRIPTION

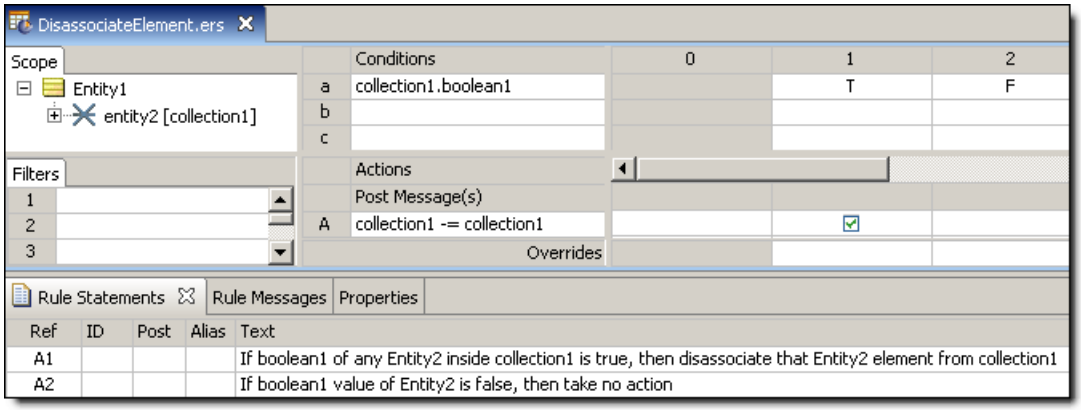
Disassociates all elements of `<Collection2>` from `<Collection1>`. Elements are not deleted, but once disassociated from `<Collection1>`, they are moved to the "root" level of the data. `<Collection1>` must be expressed as a unique alias. Contrast this behavior with remove, which deletes elements entirely.

USAGE RESTRICTIONS

The Operators row of the table in Vocabulary usage restrictions does not apply. Special exceptions: **disassociate element** may only be used in Action Rows (section 5 in Sections of Rulesheet that correlate with usage restrictions).

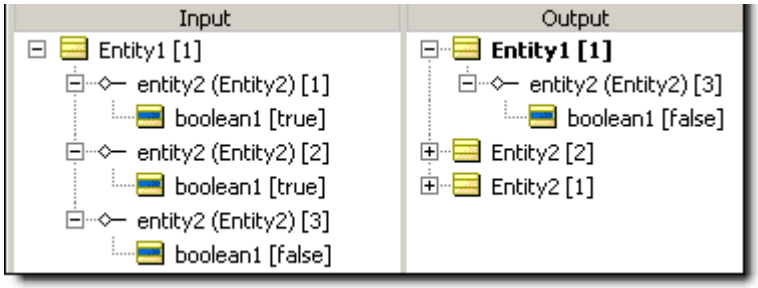
RULESHEET EXAMPLE

This sample Rulesheet removes those elements from `collection1` whose `boolean1` value is true.



SAMPLE *RULETEST*

A sample Ruletest provides a collection with three elements. The illustration shows Input and Output panels:



Divide

SYNTAX

`<Number1>/<Number2>`

DESCRIPTION

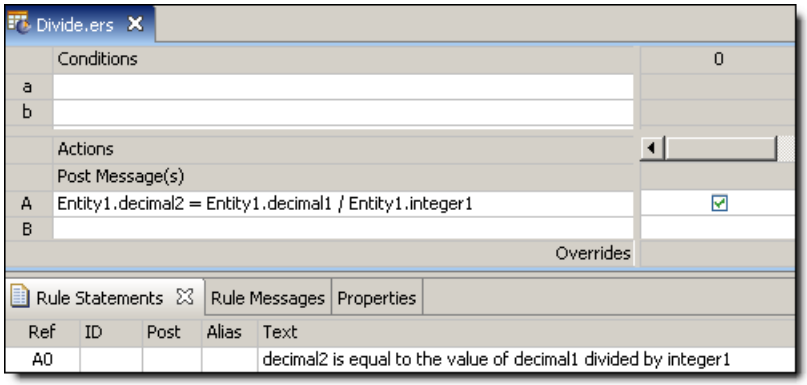
Divides `<Number1>` by `<Number2>`. The resulting data type is the more expansive of those of `<Number1>` and `<Number2>`.

USAGE RESTRICTIONS

The Operators row in the table of Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

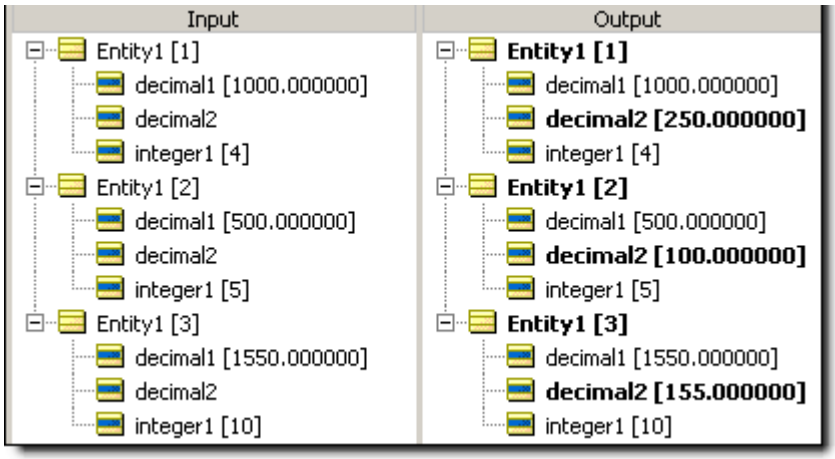
RULESHEET EXAMPLE

This sample Rulesheet uses **divide** to divide `decimal1` by `integer1` and assign the resulting value to `decimal2`



SAMPLE RULETEST

A sample Ruletest provides `decimal1` and `integer1` values for three examples. Input and Output panels are shown below.



Div

SYNTAX

`<Integer1>`.div(`<Integer2>`)

DESCRIPTION

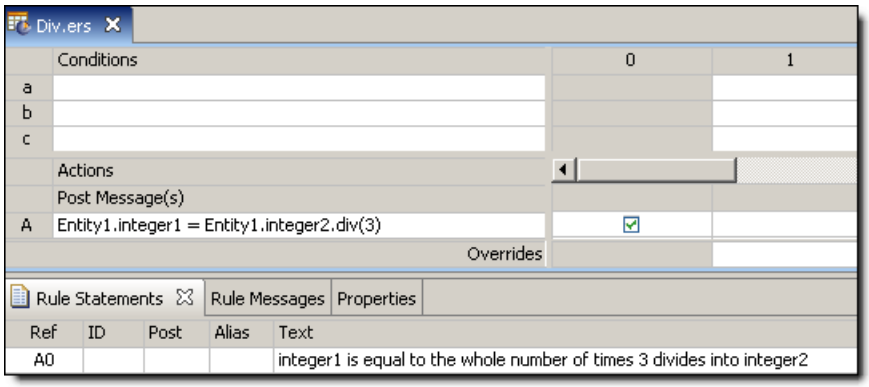
Returns an Integer equal to the whole number of times that `<Integer2>` divides into `<Integer1>`. Any remainder is discarded.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

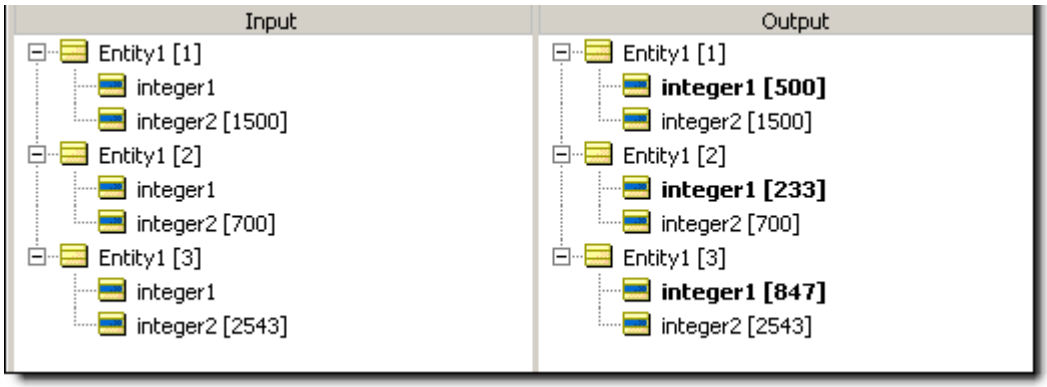
RULESHEET EXAMPLE

This sample Rulesheet uses **.div** to calculate the whole number of times `3` divides into `integer2`, and assigns the resulting value to `integer1` .



SAMPLE RULETEST

A sample Ruletest provides `integer2` values for three examples. Input and Output panels are shown below.



Ends with

SYNTAX

`<String1>`.endsWith(`<String2>`)

DESCRIPTION

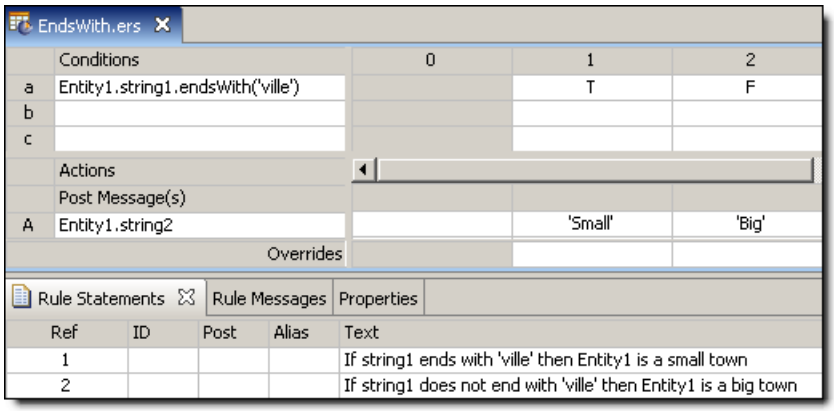
Evaluates `<String1>` and returns a value of true if it ends with the characters specified in `<String2>`.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

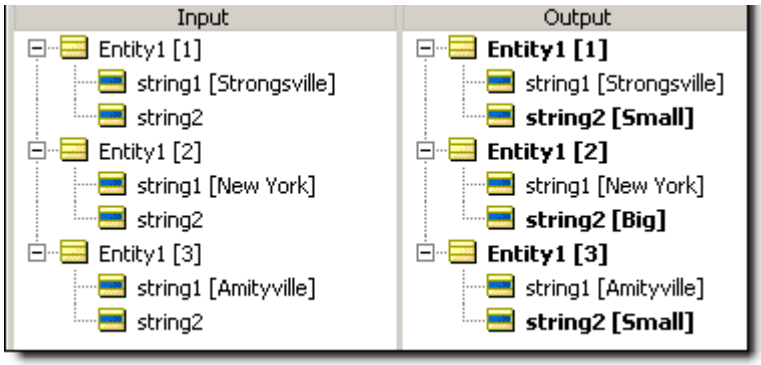
RULESHEET EXAMPLE

The following Rulesheet uses **.endsWith** to evaluate whether `string1` ends with the characters `ville` and assigns a different value to `string2` for each outcome.



SAMPLE RULETEST

A sample Ruletest provides `string1` values for three examples. Input and Output panels are shown below.



Equals – used as an assignment

SYNTAX

Boolean	`<Boolean1>` = `<Expression1>`
DateTime*	`<DateTime1>` = `<DateTime2>`
Number	`<Number1>` = `<Number2>`
String	`<String1>` = `<String2>`

DESCRIPTION

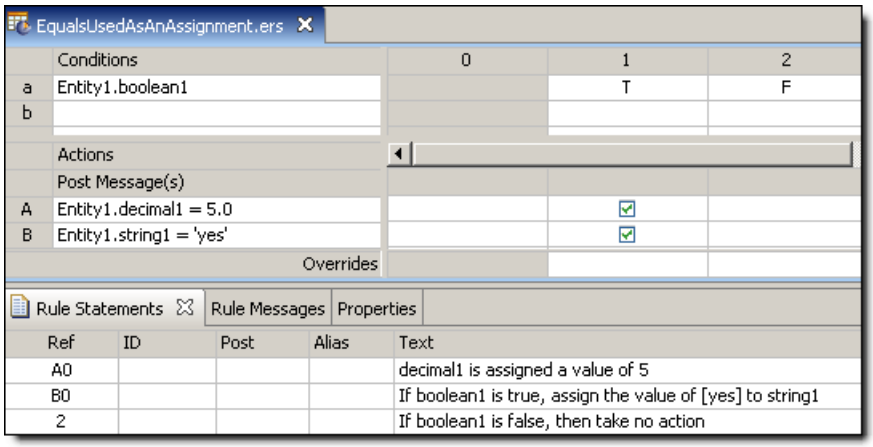
Boolean	Assigns the truth value of `<Expression1>` to `<Boolean1>`.
DateTime*	Assigns the value of `<DateTime2>` to `<DateTime1>`.
Number	Assigns the value of `<Number2>` to `<Number1>`. Automatic *casting* (the process of changing a value's data type) -- which includes DateTime, Date, or Time data types -- will occur when assigning an Integer data type to a Decimal data type. To assign a Decimal value to an Integer value, use the .toInteger operator.
String	Assigns the value of `<String2>` to `<String1>`.

USAGE RESTRICTIONS

The Operators row of the table in Vocabulary usage restrictions does not apply. Special exceptions: **equals** used as an assignment may only be used in Action Rows (section 5 in Sections of Rulesheet that correlate with usage restrictions).

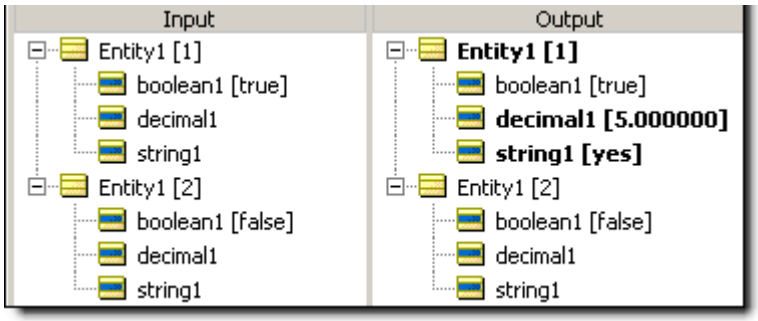
RULESHEET EXAMPLE

The following Rulesheet uses **equals** twice: in an Action row to assign a value to `decimal1`, and in an Action row to assign a value to `string1` based on the value of `boolean1`.



SAMPLE RULETEST

A sample Ruletest provides two examples of `boolean1`. Input and Output panels are shown below:



Equals – used as a comparison

SYNTAX

Boolean	`<Expression1> = <Expression2>`
DateTime*	`<DateTime1> = <DateTime2>`
Number	`<Number1> = <Number2>`
String	`<String1> = <String2>`

DESCRIPTION

Boolean	Returns a value of true if `<Expression1>` is the same as `<Expression2>`.
DateTime*	Returns a value of true if `<DateTime1>` is the same as `<DateTime2>`, including both the Date and the Time portions
Number	Returns a value of true if `<Number1>` is the same as `<Number2>`. Different numeric data types may be compared in the same expression.
String	Returns a value of true if `<String1>` is the same as `<String2>`. Both case and length are examined to determine equality. Corticon Studio uses the ISO character precedence in comparing String values. See Character precedence: Unicode & Java Collator on page 189.

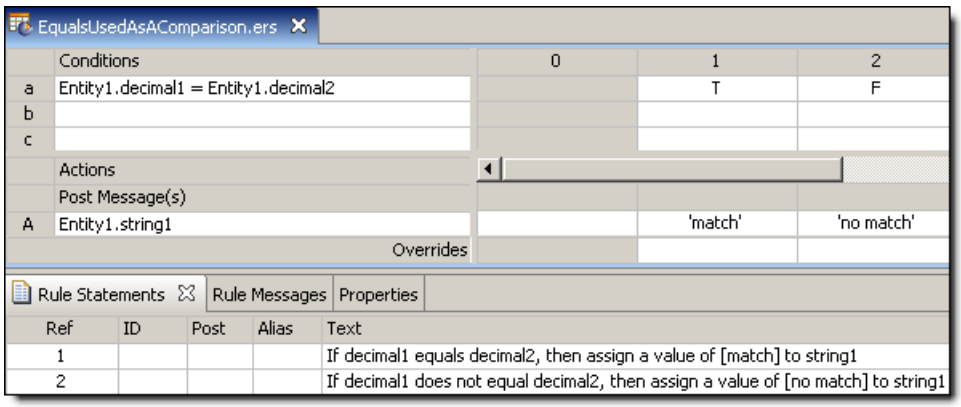
*includes DateTime, Date, and Time data types

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

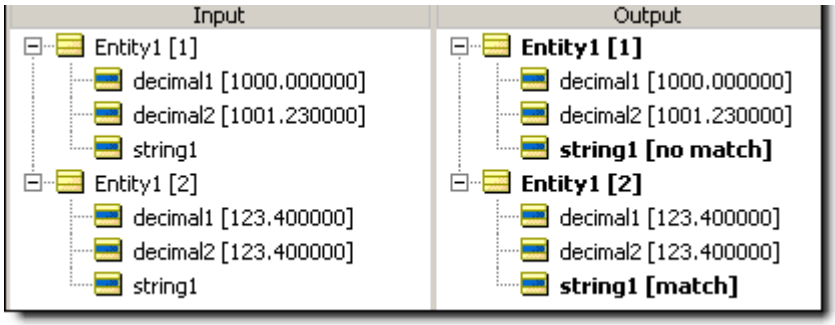
RULESHEET EXAMPLE

The following *`Rulesheet`* uses **equals** to *`Ruletest`* whether `decimal1` equals `decimal2`, and assign a value to `string1` based on the result of the comparison.



SAMPLE RULETEST

A sample *`Ruletest`* provides two examples. Input and Output panels are shown below:



Equals ignoring case

SYNTAX

`<String1>`.equalsIgnoreCase(`<String2>`)

DESCRIPTION

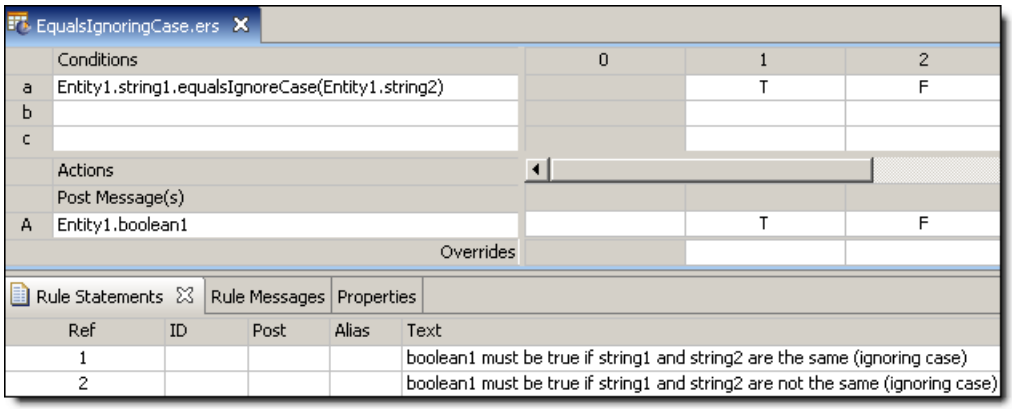
Returns a value of true if `<String1>` is the same as `<String2>`, irrespective of case.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

RULESHEET EXAMPLE

This sample Rulesheet uses **.equalsIgnoreCase** to compare the values of `string1` and `string2`, and assign a value to `boolean1` based on the results of the comparison.



EqualsIgnoringCase.ers

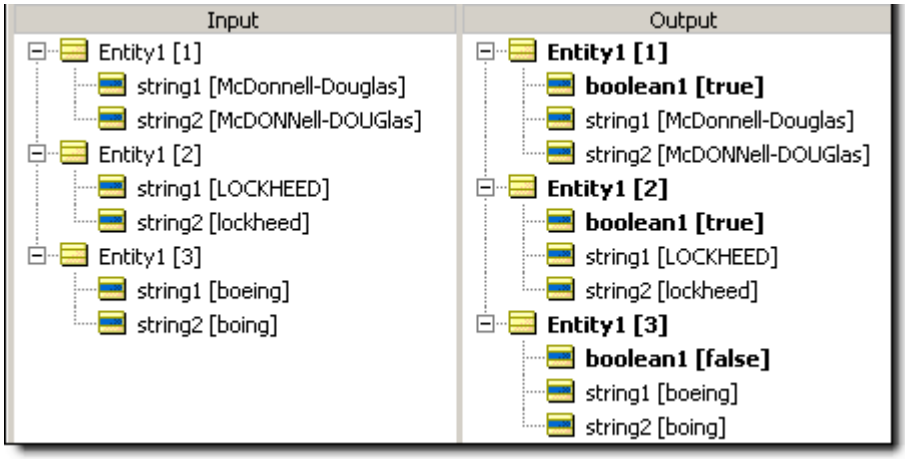
	Conditions	0	1	2
a	Entity1.string1.equalsIgnoreCase(Entity1.string2)		T	F
b				
c				
	Actions			
	Post Message(s)			
A	Entity1.boolean1		T	F
	Overrides			

Rule Statements | Rule Messages | Properties

Ref	ID	Post	Alias	Text
1				boolean1 must be true if string1 and string2 are the same (ignoring case)
2				boolean1 must be true if string1 and string2 are not the same (ignoring case)

SAMPLE RULETEST

A sample Ruletest provides the plane type for three sets of `string1` and `string2`. Input and Output panels are shown below. Notice how these results differ from those shown in the equals example.



Input

- Entity1 [1]
  - string1 [McDonnell-Douglas]
  - string2 [McDONNell-DOUGlas]
- Entity1 [2]
  - string1 [LOCKHEED]
  - string2 [lockheed]
- Entity1 [3]
  - string1 [boeing]
  - string2 [boing]

Output

- **Entity1 [1]**
  - **boolean1 [true]**
  - string1 [McDonnell-Douglas]
  - string2 [McDONNell-DOUGlas]
- **Entity1 [2]**
  - **boolean1 [true]**
  - string1 [LOCKHEED]
  - string2 [lockheed]
- **Entity1 [3]**
  - **boolean1 [false]**
  - string1 [boeing]
  - string2 [boing]

Equals – strings only

SYNTAX

`<String1>`.equals(`<String2>`)

DESCRIPTION

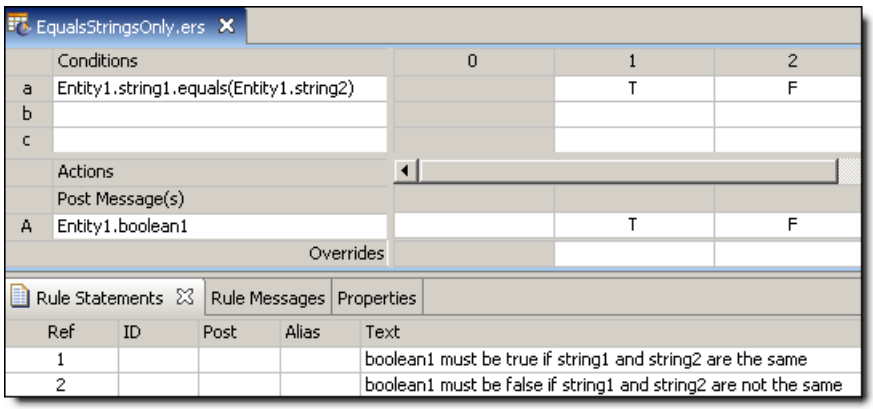
Returns a value of true if `<String1>` is exactly the same as `<String2>`, including character case. This is alternative syntax to equals (used as a comparison).

USAGE RESTRICTIONS

The Operators row in the table Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

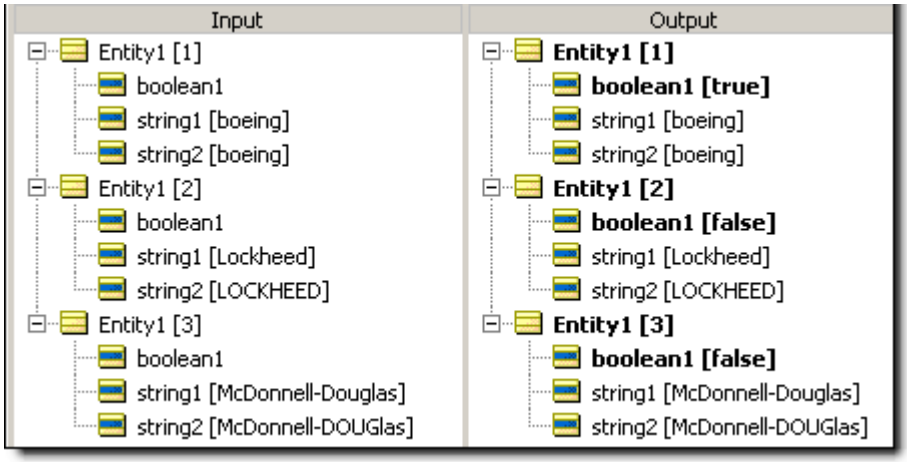
RULESHEET EXAMPLE

This sample Rulesheet uses **.equals** to compare the contents of `string1` and `string2`, and assign a value to `boolean1` as a result.



SAMPLE RULETEST

A sample Ruletest provides three sets of `string1` and `string2`. Input and Output panels are shown below. Notice how these results differ from those shown in the .equalsIgnoreCase example.



Exists

SYNTAX

`<Collection>` ->exists(`<Expression1>`,`<Expression2>`,...)

`<Collection>` ->exists(`<Expression1>` or `<Expression2>` or ...)

DESCRIPTION

Returns a value of true if `<Expression>` holds true for *at least one* element of `<Collection>`. `<Collection>` must be expressed as a unique alias. Multiple `<Expressions>` are optional, but at least one is required.

Both **AND** (indicated by commas between `<Expressions>`) and **OR** syntax (indicated by `or` between `<Expressions>`) are supported within the parentheses `(..)`. However, take care to ensure invariant expressions are not inadvertently created. For example:

```
<Collection> -> exists(integer1=5, integer1=8)
```

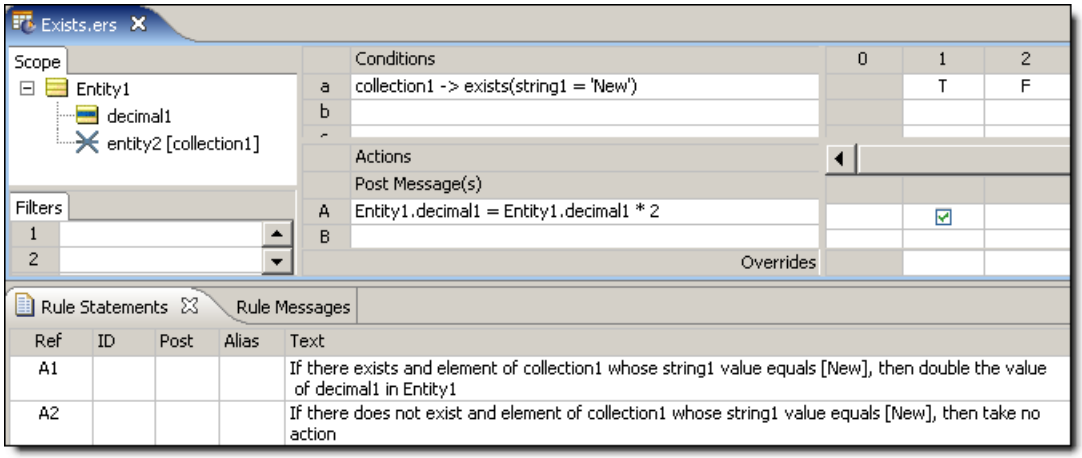
will always evaluate to `false` because no `integer1` value can be both `5` **AND** `8` simultaneously.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

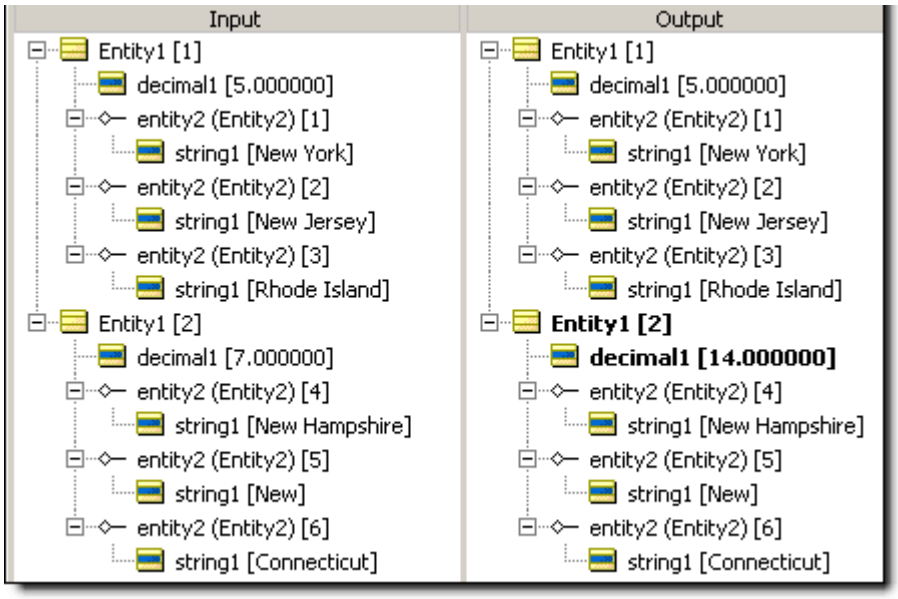
RULESHEET EXAMPLE

This sample Rulesheet uses **exists** to check for the existence of an element in `collection1` whose `string1` value equals `New`, and assigns a value to `decimal1` based on the results of the test. Note the use of unique alias `collection1` to represent the collection of `Entity2` associated with `Entity1`.



SAMPLE RULETEST

A sample Ruletest provides 2 separate collections of `Entity2` elements and `Entity1.decimal1` values. Input and Output panels are shown below.



Exponent

SYNTAX

`<Number1>` ** `<Number2>`

DESCRIPTION

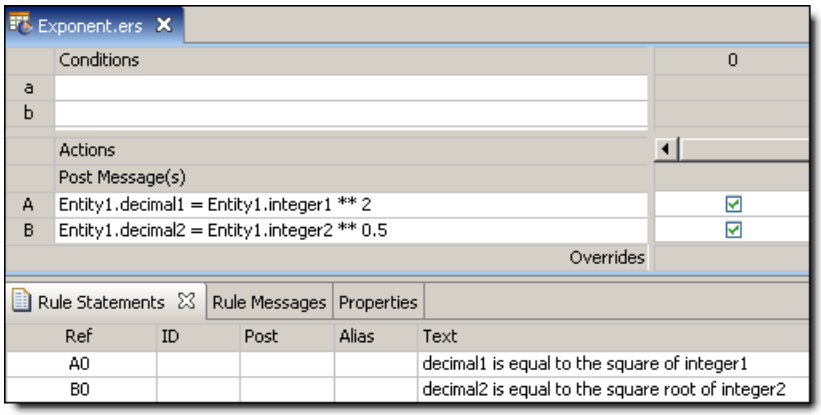
Raises `<Number1>` by the power of `<Number2>`. The resulting data type is the more expansive of those of `<Number1>` and `<Number2>`. To find a root, `<Number2>` should be expressed as a fraction. For example, the square root is expressed as `0.5`

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

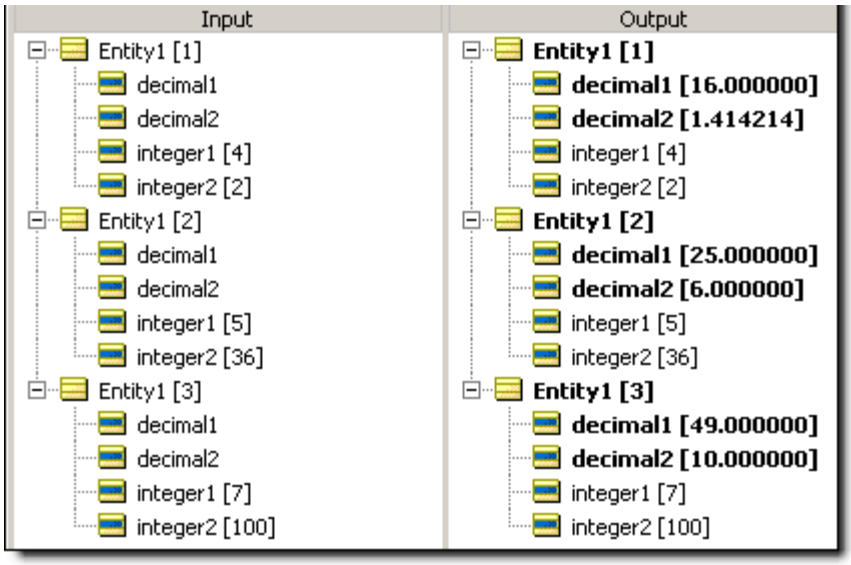
RULESHEET EXAMPLE

This sample Rulesheet uses **exponent** to raise `integer1` and `integer2` by the power of `2` and `0.5`, respectively, and assign the resulting value to `decimal1` and `decimal2`, respectively.



SAMPLE RULETEST

A sample Ruletest provides `decimal1` and `integer1` values for three examples.



False

SYNTAX

false or F

DESCRIPTION

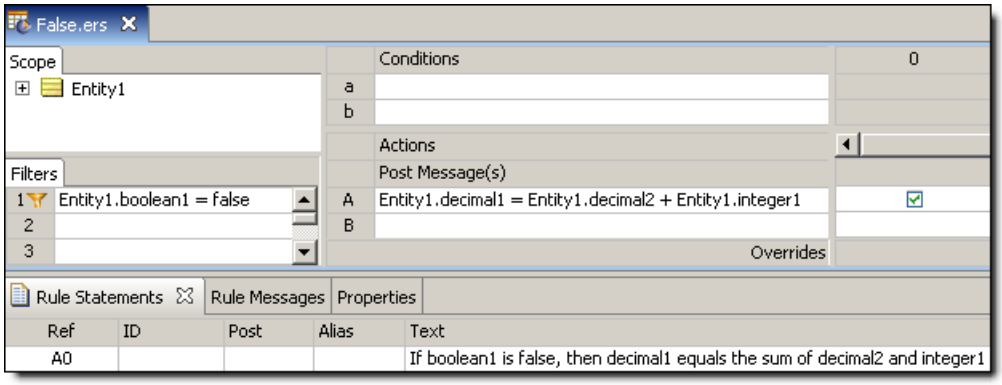
Represents the Boolean value false. Recall from discussion of truth values that an `<expression>` is evaluated for its truth value, so the expression `Entity1.boolean1=false` evaluates to `true` only when `boolean1=false`. But since `boolean1` is Boolean and has a truth value all by itself without any additional syntax, we could simply state `not Entity1.boolean1`, with the same effect. Many examples in the documentation use explicit syntax like `boolean1=true` or `boolean2=false` for clarity and consistency, even though `boolean1` or `not boolean2` are equivalent, respectively, to the explicit syntax.

USAGE RESTRICTIONS

The Operators row of the table in Vocabulary usage restrictions applies. No special exceptions.

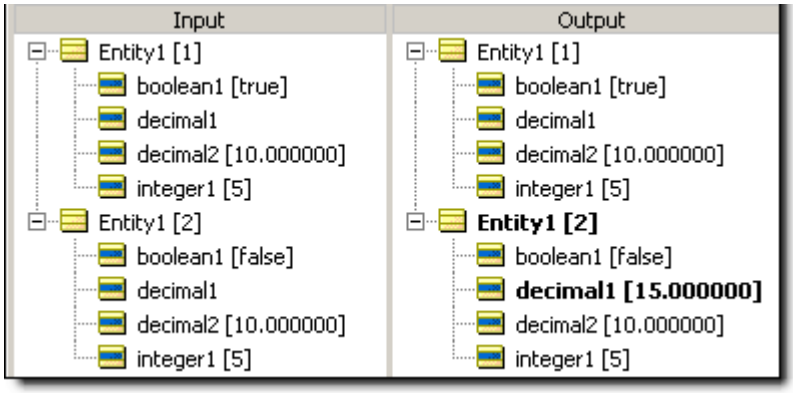
RULESHEET EXAMPLE

The following Rulesheet uses **false** in a Filter row to test whether `boolean1` is false, and perform the Nonconditional computation if it is. As discussed above, the alternative expression `not Entity1.boolean1` is logically equivalent.



SAMPLE RULETEST

A sample Ruletest provides three examples. Assume `decimal2=10.0` and `integer1=5` for all examples. Input and Output panels are shown below:



First

SYNTAX

`<Sequence>` ->first.`<attribute1>`

DESCRIPTION

Returns the value of `<attribute1>` of the first element in `<Sequence>`. Another operator, such as sortedBy, must be used to transform a `<Collection>` into a `<Sequence>` before **first** may be used. `<Sequence>` must be expressed as a unique alias. See Advanced Collection Syntax for more examples of usage.

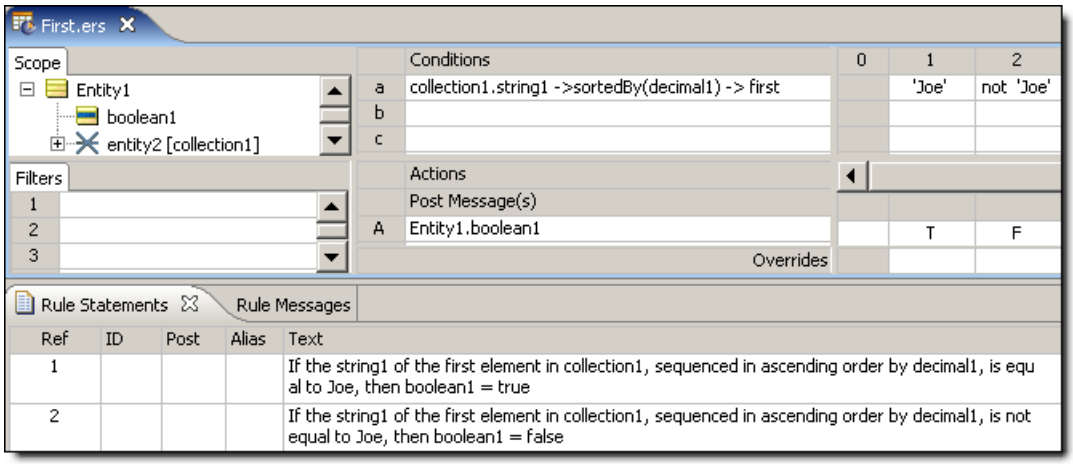
`<attribute1>` may be of any data type.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

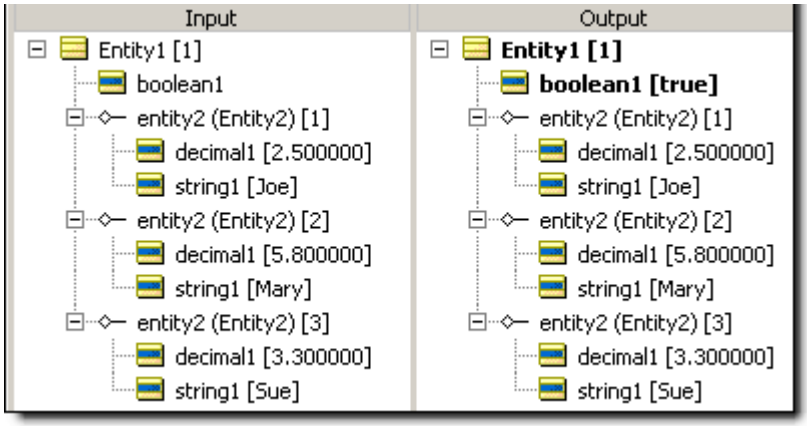
RULESHEET EXAMPLE

This sample Rulesheet uses **first** to identify the first element of the sequence created by applying sortedBy to `collection1`. Once identified, the value of the `string1` attribute belonging to this first element is evaluated. If the value of `string1` is `Joe`, then `boolean1` attribute of `Entity1` is assigned the value of `true` .



SAMPLE RULETEST

A sample Ruletest provides a collection of three elements, each with a `decimal1` value. Input and Output panels are shown below.



Floor

SYNTAX

`<Decimal>`.floor

DESCRIPTION

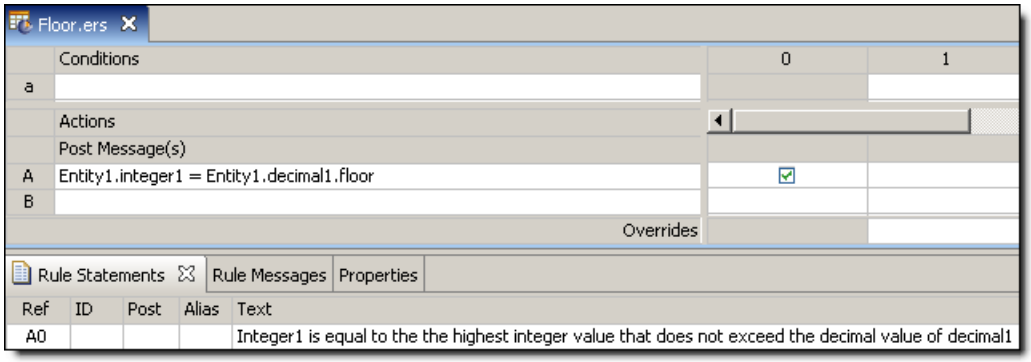
Returns the Integer closest to zero from `<Decimal>`. **.floor** may also be thought of as a truncation of `<Decimal>`.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

RULESHEET EXAMPLE

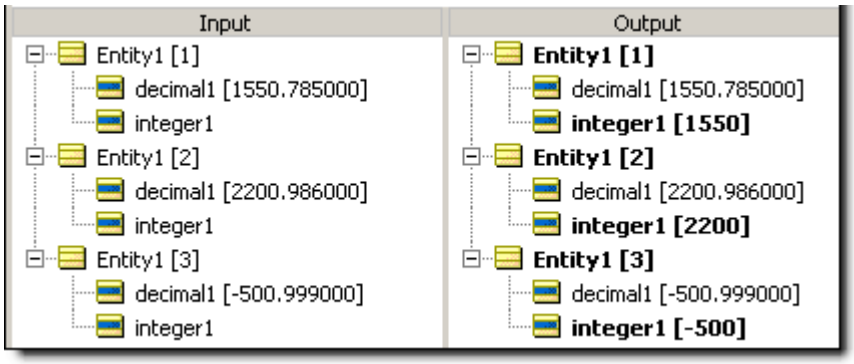
The Rulesheet uses **.floor** to assign Integer values to `integer1` that are closer to zero than the input `decimal1` values.



SAMPLE RULETEST

A sample Ruletest provides three `decimal1` values. Input and Output panels are shown below:

Note: Notice how these results differ from those shown in the Round example.



For all

SYNTAX

`<Collection>` ->forAll(`<Expression1>`, `<Expression2>`,...)

`<Collection>` ->forAll(`<Expression1>` or `<Expression2>` or ...)

DESCRIPTION

Returns a value of true if *every* `<Expression>` holds true for *every* element of `<Collection>`. `<Collection>` must be expressed as a unique alias. Multiple `<Expressions>` are optional, but at least one is required.

Both **AND** (indicated by commas between `<Expressions>`) and **OR** syntax (indicated by `or` between `<Expressions>`) is supported within the parentheses `(..)`. However, take care to ensure invariant expressions are not inadvertently created. For example:

```
<Collection> -> forAll(integer1=5, integer1=8)
```

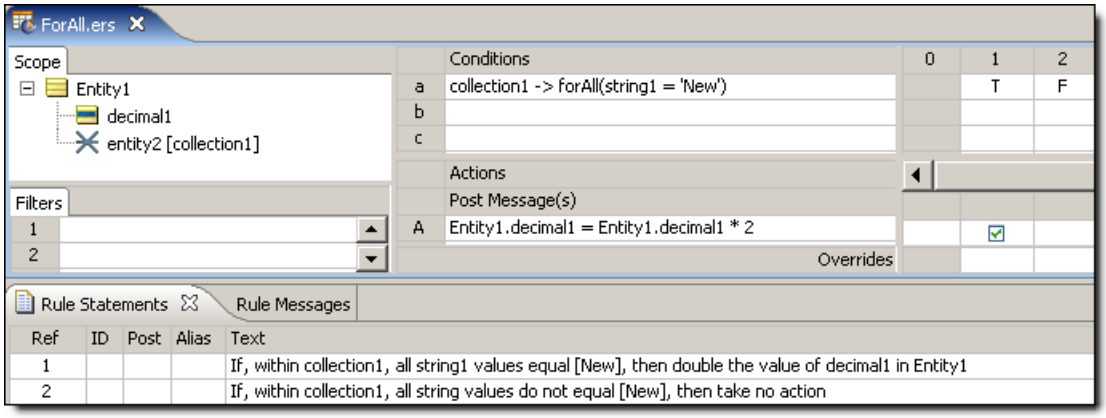
will always evaluate to `false` because no single `integer1` value can be both `5` **AND** `8` simultaneously, let alone all of them.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

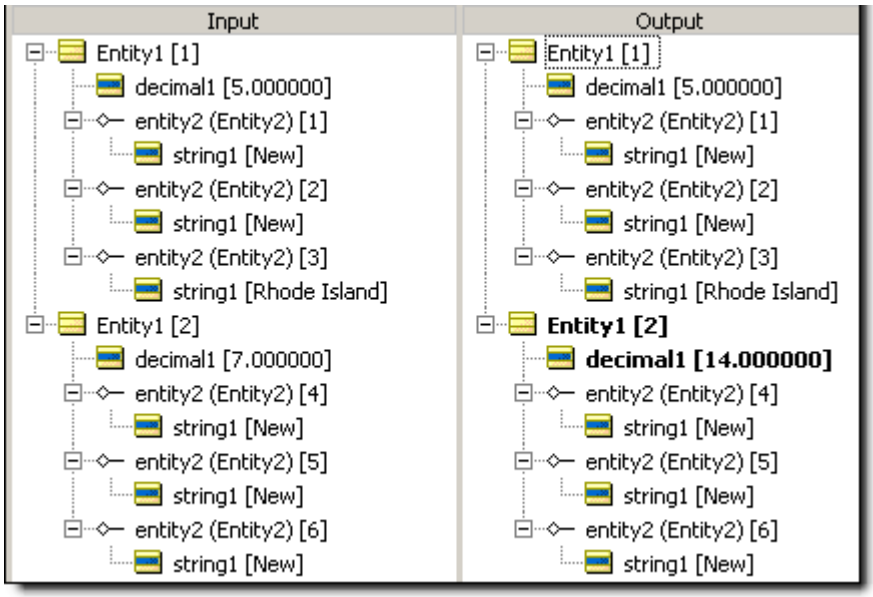
RULESHEET EXAMPLE

This sample Rulesheet uses **forAll** to check for the existence of an element in `collection1` whose `string1` value equals `New`, and assigns a value to `decimal1` based on the results of the test. Note the use of unique alias `collection1` to represent the collection of `Entity2` associated with `Entity1`.



SAMPLE RULETEST

A sample Ruletest provides 2 separate collections of `Entity2` elements and `Entity1.decimal1 values`. The following illustration shows Input and Output panel



Greater than

SYNTAX

DateTime*	`<DateTime1>` > `<DateTime2>`
Number	`<Number1>` > `<Number2>`
String	`<String1>` > `<String2>`

DESCRIPTION

DateTime*	Returns a value of true if `<DateTime1>` is greater than or equal to `<DateTime2>`. This is equivalent to `<DateTime1>` occurring "after" `<DateTime2>`
Number	Returns a value of true if `<Number1>` is greater than `<Number2>`. Different numeric data types may be compared in the same expression.
String	Returns a value of true if `<String1>` is greater than `<String2>`. Studio uses Character precedence: Unicode & Java Collator on page 189 to determine character precedence.

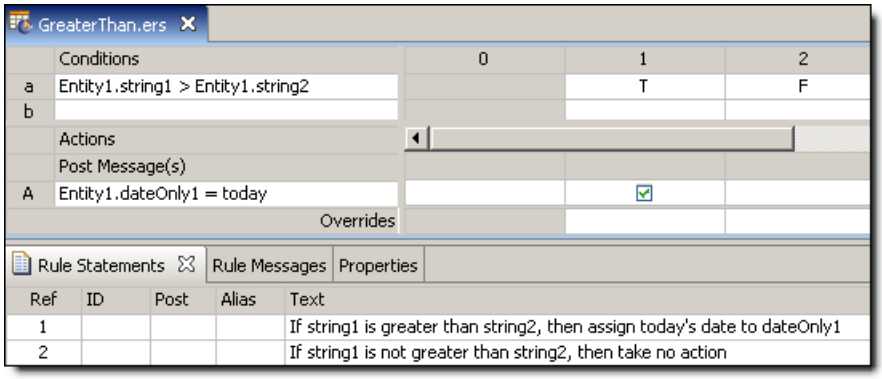
*includes DateTime, Date, and Time data types

USAGE RESTRICTIONS

The Operators row of the table in Vocabulary usage restrictions applies, with the following exception: **greater than** may also be used in Conditional Value Sets & Cells (section 5 in Sections of Rulesheet that correlate with usage restrictions).

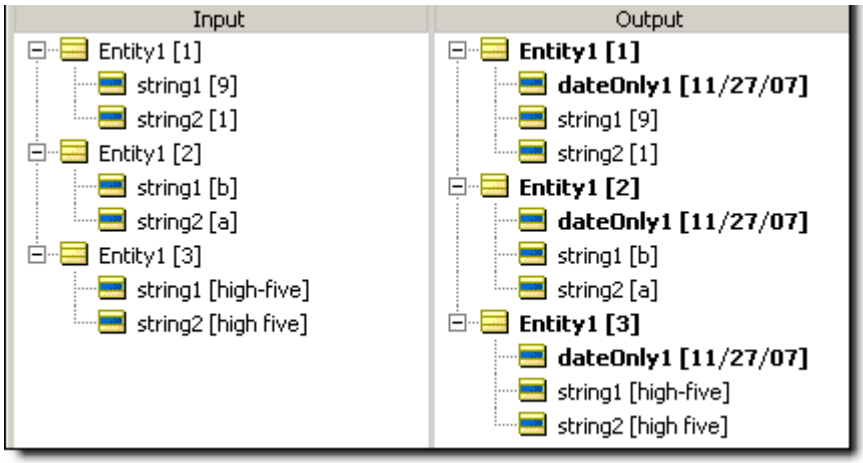
RULESHEET EXAMPLE

The following Rulesheet uses **greater than** to test whether `string1` is greater than `string2`, and assign today's date to `dateTime1` if it is. See today for an explanation of this literal term.



SAMPLE RULETEST

A sample *`Ruletest`* provides three examples. Input and Output panels are shown below:



Greater than or equal to

SYNTAX

DateTime*	`<DateTime1>` >= `<DateTime2>`
Number	`<Number1>` >= `<Number2>`
String	`<String1>` >= `<String2>`

DESCRIPTION

DateTime*	Returns a value of true if `<DateTime1>` is greater than or equal to `<DateTime2>`. This is equivalent to `<DateTime1>` occurring on or after `<DateTime2>`
Number	Returns a value of true if `<Number1>` is greater than or equal to `<Number2>`. Different numeric data types may be compared in the same expression.
String	Returns a value of true if `<String1>` is greater than or equal to `<String2>`. Corticon Studio uses Character precedence: Unicode & Java Collator on page 189 to determine character precedence.

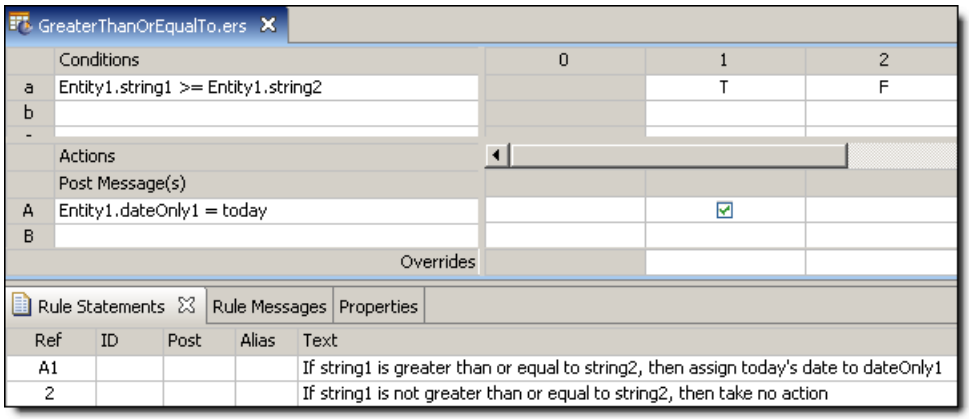
*includes DateTime, Date, and Time data types

USAGE RESTRICTIONS

The Operators row of the table in Vocabulary usage restrictions applies, with the following exception: **greater than or equal to** may also be used in Conditional Value Sets & Cells (section 5 in Sections of Rulesheet that correlate with usage restrictions).

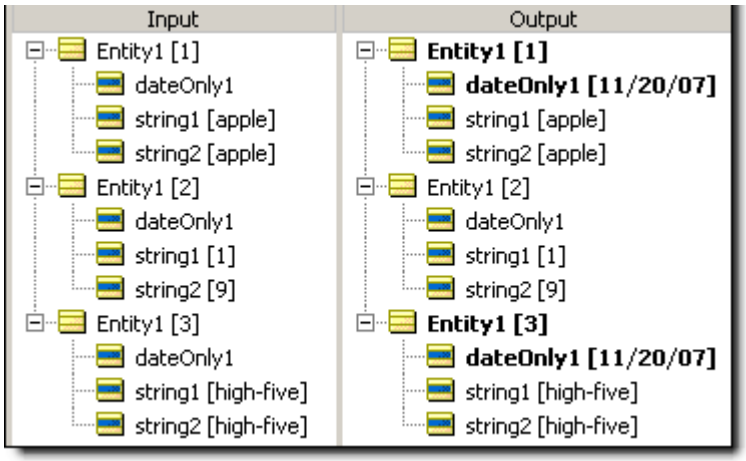
RULESHEET EXAMPLE

The following Rulesheet uses **greater than or equal to** to test whether `string1` is greater than or equal to `string2`, and assign today's date to `dateTime1` if it is. See today for an explanation of this literal term.



SAMPLE RULETEST

A sample Ruletest provides two examples. Input and Output panels are shown below:



Hour

SYNTAX

`<DateTime>`.hour

`<Time>`.hour

DESCRIPTION

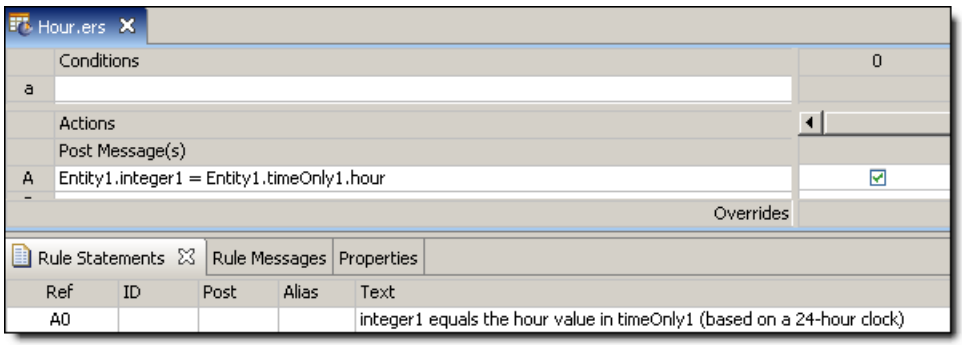
Returns the hour portion of `<DateTime>` or `<Time>`. The returned value is based on a 24-hour clock. For example, 10:00 PM (22:00 hours) is returned as 22.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

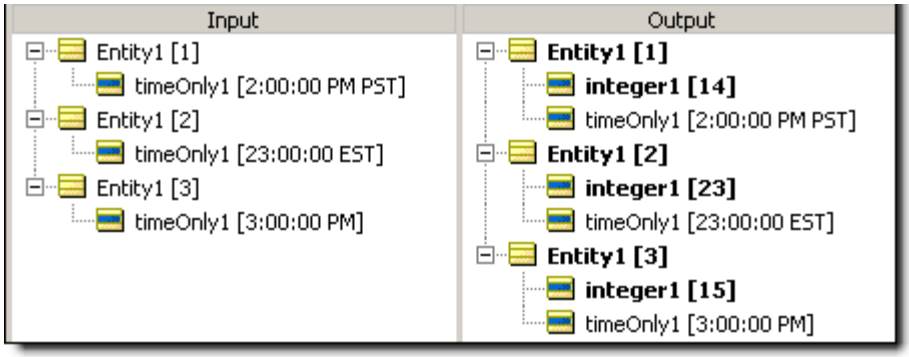
RULESHEET EXAMPLE

The following Rulesheet uses **.hour** to evaluate `dateTime1` and assign the hour value to `integer1`.



SAMPLE RULETEST

A sample Ruletest provides three examples of `dateTime1`. Input and Output panels are shown below. Notice that the hour returned is dependent upon the timezone of the machine executing the rule. The hour returned is independent of the machine running the Ruletest and only depends on the locale/timezone of the data itself.



Hour between

SYNTAX

`<DateTime1>`.hoursBetween(`<DateTime2>`)

DESCRIPTION

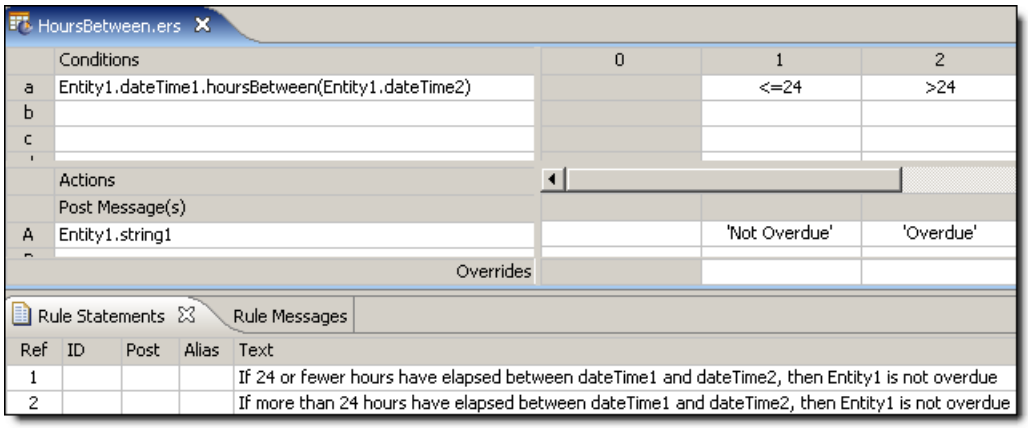
Returns the Integer number of hours between any two DateTimes or Times. The function calculates the number of milliseconds between the two values and divides that number by 3,600,000 (the number of milliseconds in an hour). The decimal portion is then truncated. If the two dates differ by less than a full hour, the value is zero. This function returns a positive number if `<DateTime2>` is later than `<DateTime1>`.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

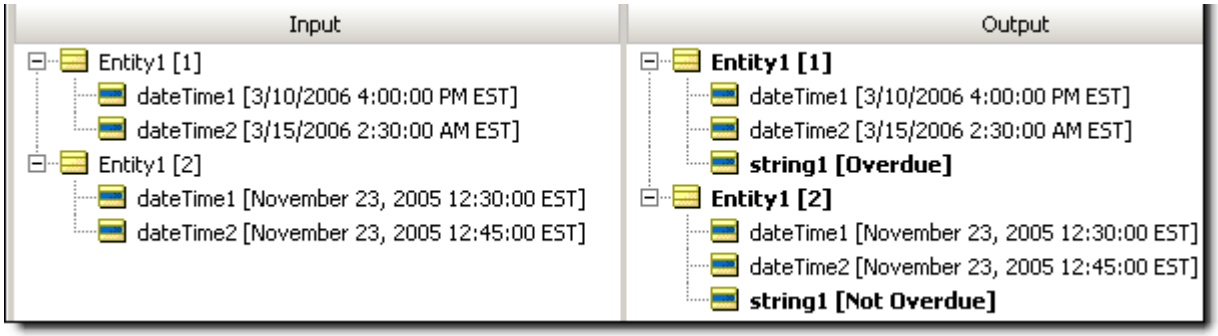
RULESHEET EXAMPLE

The following Rulesheet uses **.hoursBetween** to determine the number of hours that have elapsed between `dateTime1` and `dateTime2`, compare it to the Values set, and assign a value to `string1`.



SAMPLE RULETEST

A sample Ruletest provides `dateTime1` and `dateTime2` for two examples. Input and Output panels are shown below.



Increment

SYNTAX

`<Number1>` += `<Number2>`

DESCRIPTION

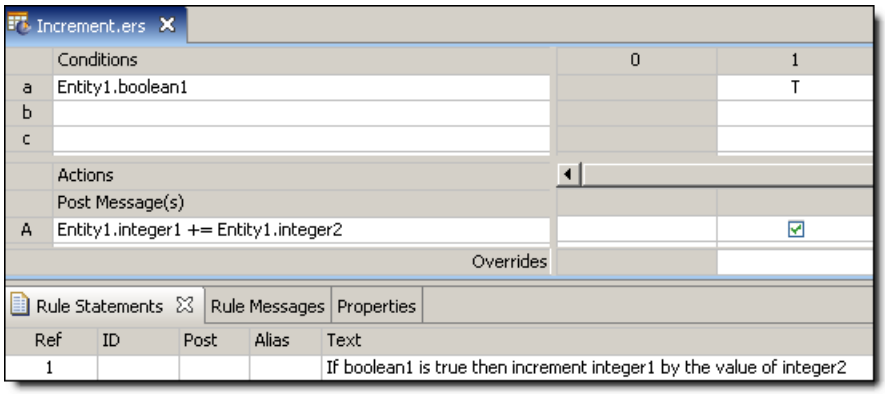
Increments `<Number1>` by the value of `<Number2>`. The data type of `<Number1>` must accommodate the addition of `<Number2>`. In other words, an Integer may not be incremented by a Decimal without using another operator (such as .toInteger or Floor on page 91.floor) to first convert the Decimal to an Integer.

USAGE RESTRICTIONS

The Operators row of the table in Vocabulary usage restrictions does not apply. Special exceptions: **increment** may only be used in Action Rows (section 5 in Sections of Rulesheet that correlate with usage restrictions).

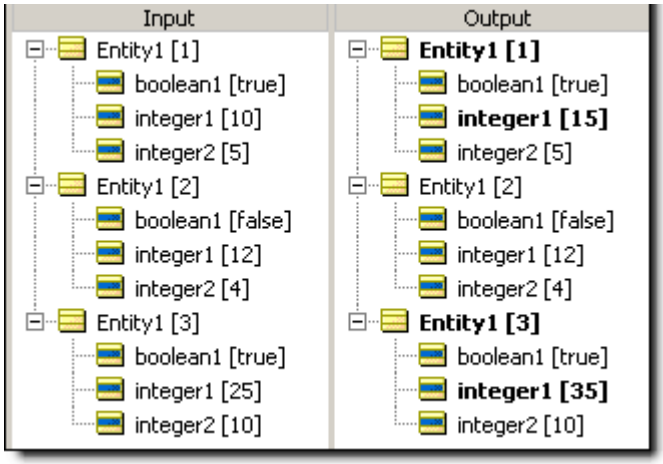
RULESHEET EXAMPLE

This sample Rulesheet uses **increment** to increment `integer1` by the value of `integer2` when `boolean1` is `true`.



SAMPLE RULETEST

A sample Ruletest provides three examples of `integer1`, `integer2`, and `boolean1`. Input and Output panels are shown below.



Index of

SYNTAX

`<String1>`.indexOf(`<String2>`)

DESCRIPTION

Determines if `<String2>` is contained within `<String1>` and returns an Integer value equal to the beginning character position of the first occurrence of `<String2>` within `<String1>`. If `<String1>` does not contain `<String2>`, then a value of `0` (zero) is returned. This operator is similar to .contains but returns different results. A `0` result from **.indexOf** is equivalent to a `false` value returned by the .contains operator.

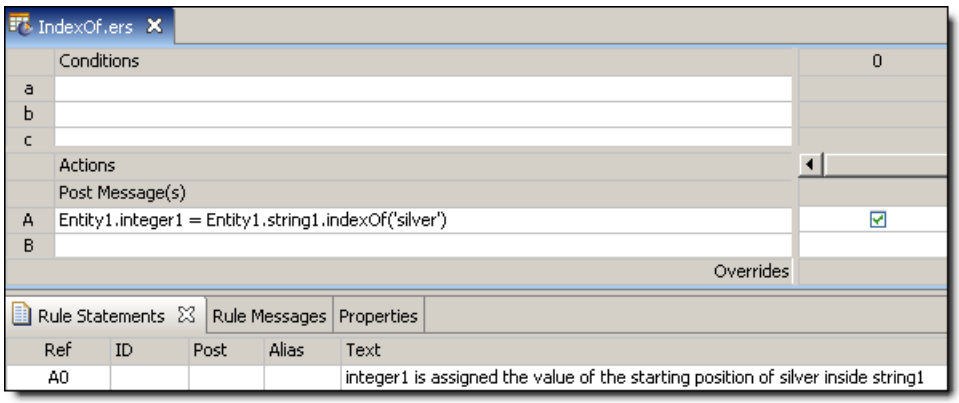
If `<String1>` contains more than one occurrence of `<String2>`, **.indexOf** returns the first character position of the first occurrence. For example: If `<String1>` holds the String value `'Mississippi'` and `<String2>` holds the String value `'ss'`, then the **.indexOf** operator returns `3`. The second occurrence of `'ss'` beginning at position 6 is not identified.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

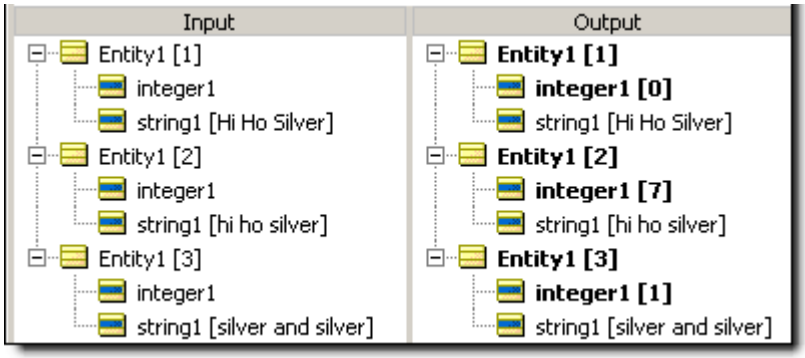
RULESHEET EXAMPLE

The following Rulesheet uses **.indexOf** to evaluate whether `string1` includes the characters `silver` and assigns a value to `integer1` corresponding to the beginning character position of the first occurrence.



SAMPLE RULETEST

A sample Ruletest provides `string1` values for three examples. Input and Output panels are shown below. Notice sensitivity to case in example 1.



Is empty

SYNTAX

`<Collection>` ->isEmpty

DESCRIPTION

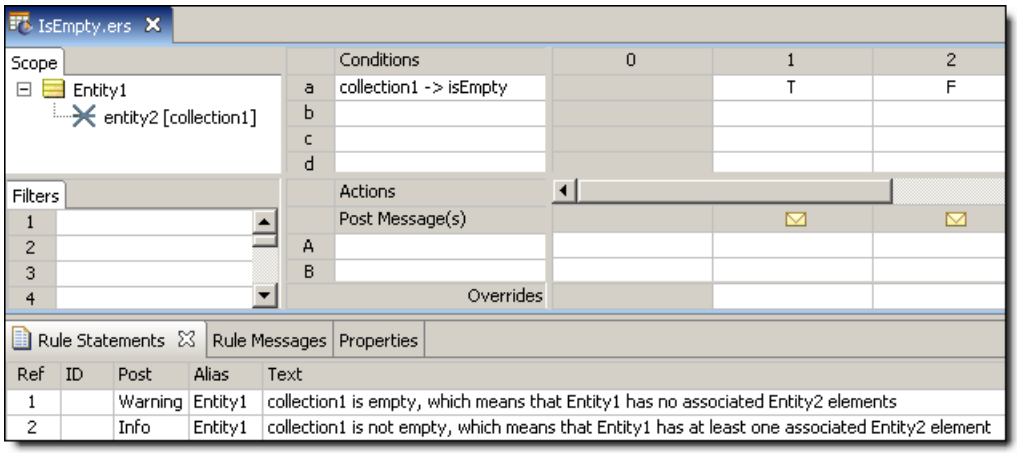
Returns a value of true if `<Collection>` contains *no* elements (that is, has no children). **isEmpty** does not check for an empty or null value of an attribute, but instead checks for *existence* of elements within the collection. As such, a unique alias must be used to represent the `<Collection>` being tested.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

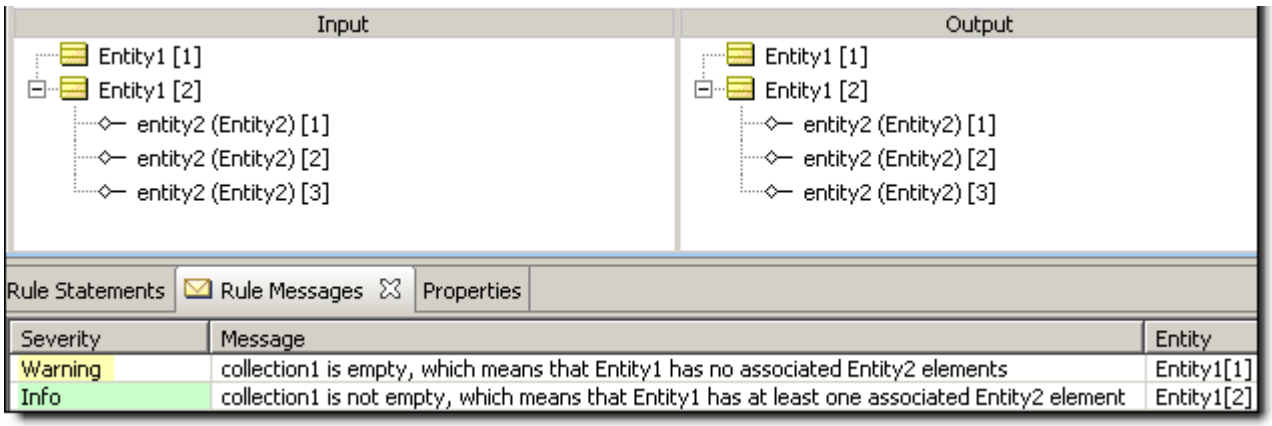
RULESHEET EXAMPLE

This sample Rulesheet uses **isEmpty** to determine if `collection1` has any elements. Note the use of unique alias `collection1` to represent the collection of `Entity2` associated with `Entity1`.



SAMPLE RULETEST

A sample Ruletest provides two example `collection1`. The following illustration shows Input and Output panels



Iterate

SYNTAX

`<Collection>` ->iterate(`<Expression>`)

DESCRIPTION

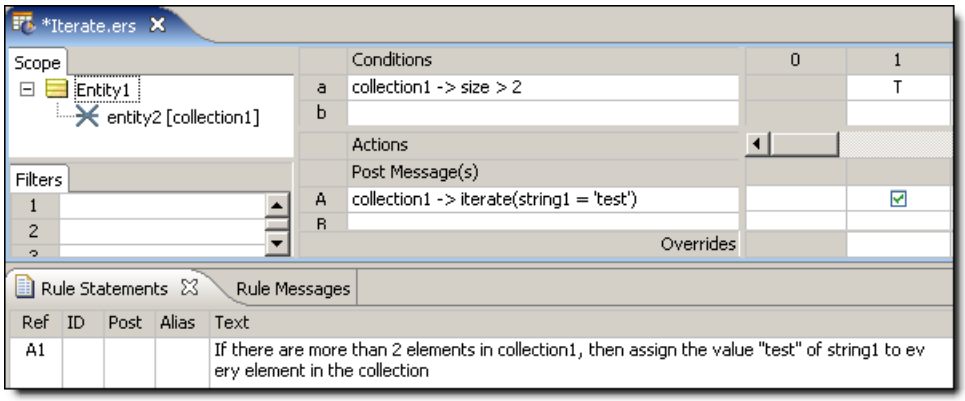
Executes `<Expression>` for every element in `<Collection>`. `<Collection>` must be expressed as a unique alias.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction does not apply. Special exceptions: **iterate** may only be used in Action Rows (section 5 in Sections of Rulesheet that correlate with usage restrictions).

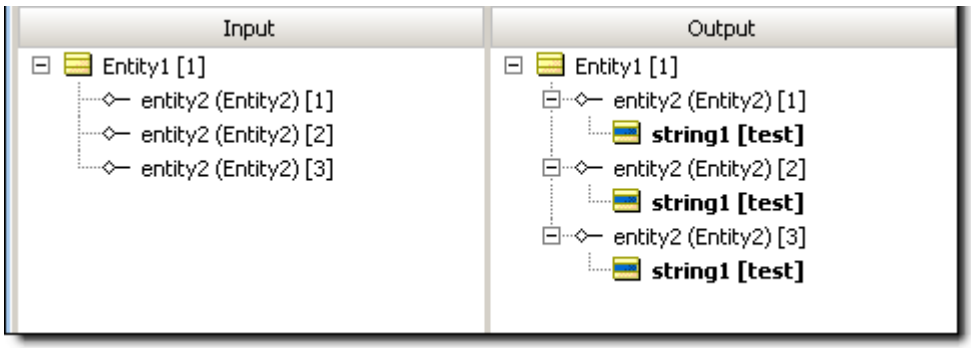
RULESHEET EXAMPLE

This sample Rulesheet uses **iterate** to assign the value of `test` to `string1` in every element in `collection1` . See exists for more information on this operator.



SAMPLE RULETEST

A sample Ruletest provides three elements in `collection1`. Input and Output panels are shown below.



Last

SYNTAX

`<Sequence>` ->last.`<Attribute1>`

DESCRIPTION

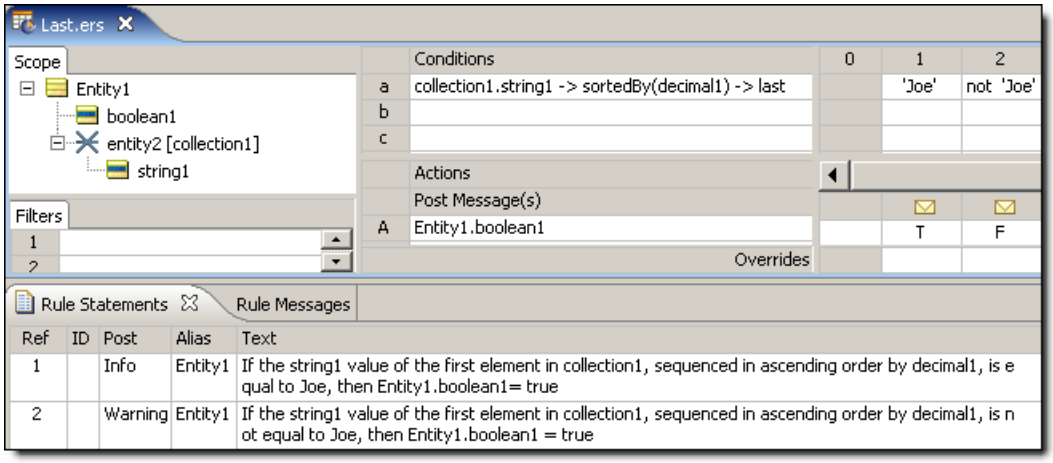
Returns the value of `<Attribute1>` of the last element in `<Sequence>`. Another operator, such as sortedBy, must be used to transform a `<Collection>` into a `<Sequence>` before **last** may be used. `<Sequence>` must be expressed as a unique alias. `<Attribute1>` may be of any data type. See Advanced Collection Syntax for more examples of usage.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

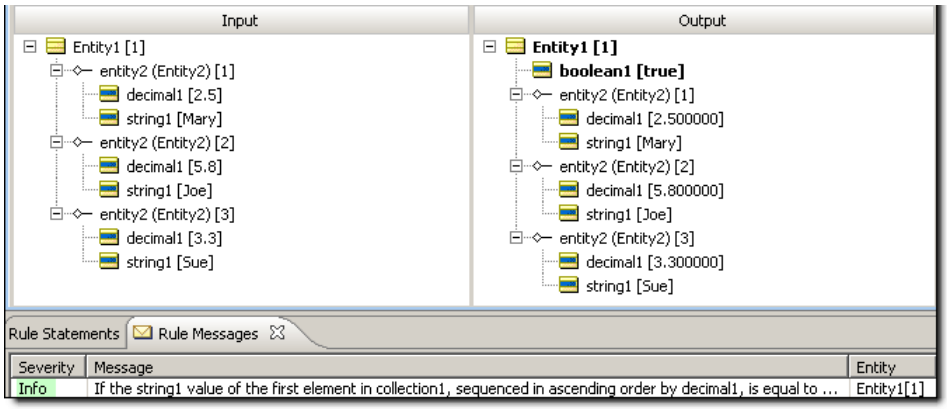
RULESHEET EXAMPLE

This sample Rulesheet uses **last** to identify the last element of the sequence created by applying sortedBy to `collection1`. Once identified, the value of the `string1` attribute belonging to this last element is evaluated. If the value of `string1` is `Joe`, then `boolean1` attribute of `Entity1` is assigned the value of `true`.



SAMPLE RULETEST

A sample Ruletest provides a collection of three elements, each with a `decimal1` value. Input and Output panels are shown below.



Less than

SYNTAX

DateTime*	`<DateTime1>` < `<DateTime2>`
Number*	`<Number1>` < `<Number2>`
String	`<String1>` < `<String2>`

DESCRIPTION

DateTime*	Returns a value of true if `<DateTime1>` is less than `<DateTime2>`. This is equivalent to `<DateTime1>` occurring "before" `<DateTime2>`
Number	Returns a value of true if `<Number1>` is less than `<Number2>`. Different numeric data types may be compared in the same expression.
String	Returns a value of true if `<String1>` is less than `<String2>`. Corticon Studio uses Character precedence: Unicode & Java Collator on page 189.

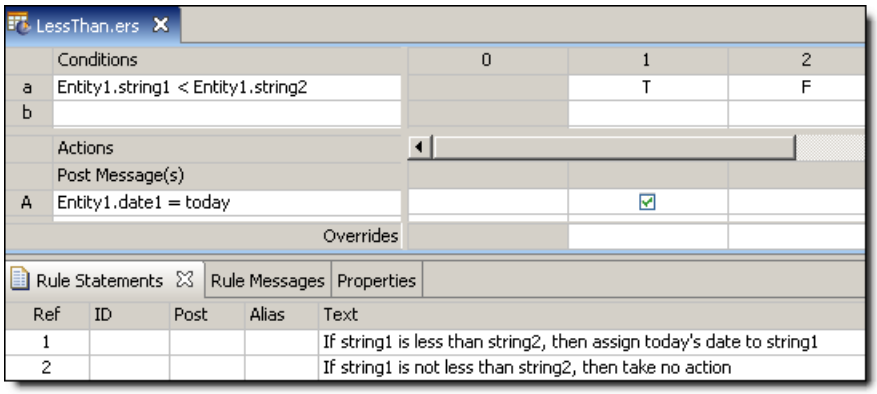
*includes DateTime, Date, or Time data types

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies, with the following exception: **less than** may also be used in Conditional Value Sets & Cells (section 5 in Sections of Rulesheet: Numbers Correlate with Table Above).

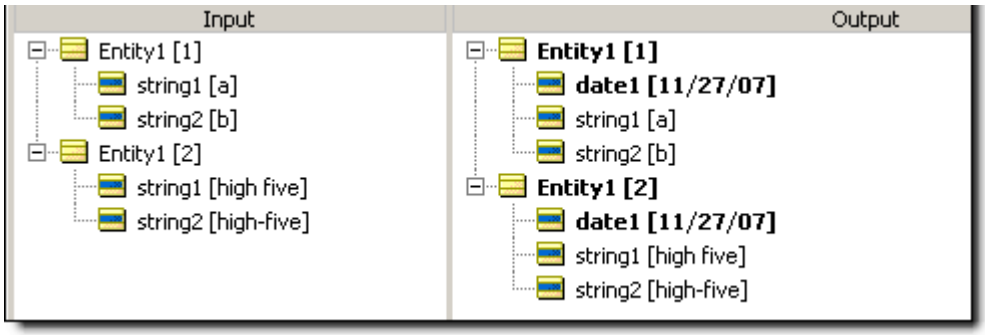
RULESHEET EXAMPLE

The following Rulesheet uses **less than** to test whether `string1` is less than `string2`, and assign today's date to `dateTime1` if it is. See today for an explanation of this literal term.



SAMPLE RULETEST

A sample Ruletest provides two examples. Input and Output panels are shown below:



Less than or equal to

SYNTAX

DateTime*	`<DateTime1>` **<=** `<DateTime2>`
Number*	`<Number1>` **<=** `<Number2>`
String	`<String1>` **<=** `<String2>`

DESCRIPTION

DateTime*	Returns a value of true if `<DateTime1>` is less than or equal to `<DateTime2>`. This is equivalent to `<DateTime1>` occurring "on or before" `<DateTime2>`
Number	Returns a value of true if `<Number1>` is less than or equal to `<Number2>`. Different numeric data types may be compared in the same expression.
String	Returns a value of true if `<String1>` is less than or equal to `<String2>`. Corticon Studio uses Character precedence: Unicode & Java Collator on page 189.

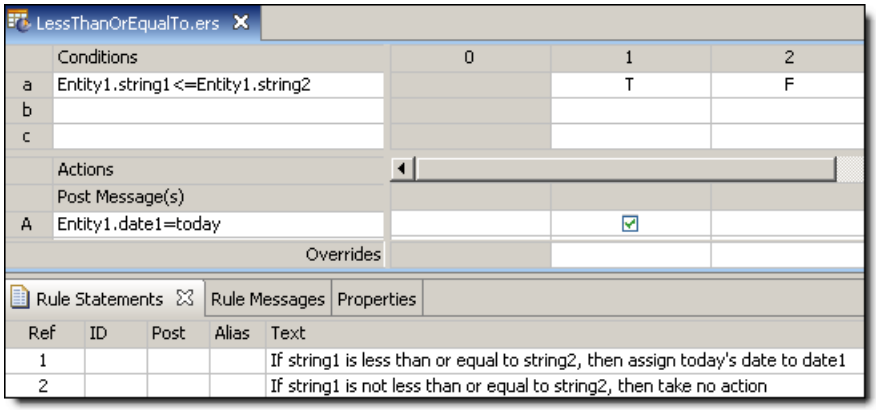
*includes DateTime, Date, or Time data types

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies, with the following exception: **less than or equal to** may also be used in Conditional Value Sets & Cells (section 5 of Sections of Rulesheet that correlate with usage restrictions).

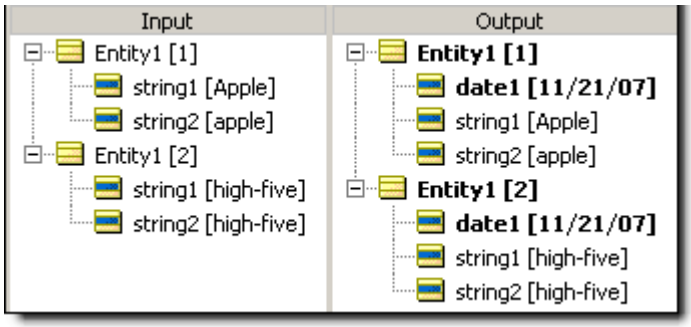
RULESHEET EXAMPLE

The following Rulesheet uses **less than or equal to** to test whether `string1` is less than or equal to `string2`, and assign today's date to `dateTime1` if it is. See today for an explanation of this literal term.



SAMPLE RULETEST

A sample Ruletest provides two examples. Input and Output panels are shown below:



Logarithm (Base 10)

SYNTAX

`<Number>`.log

DESCRIPTION

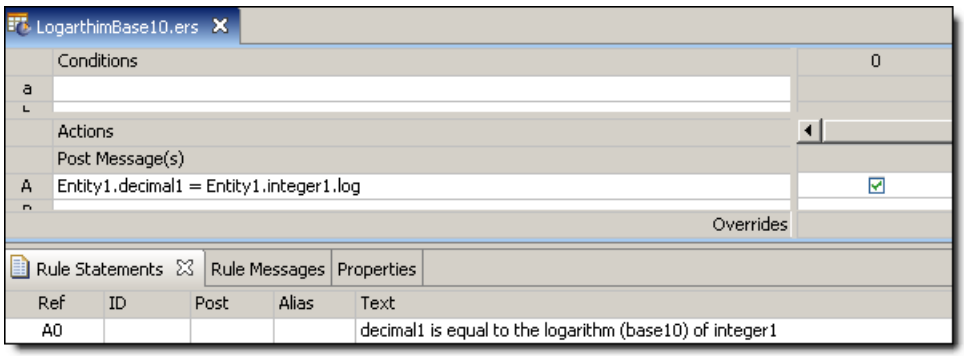
Returns a Decimal value equal to the logarithm (base 10) of `<Number>`. If `<Number>` is equal to `0` (zero) an error is returned when the rule is executed.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

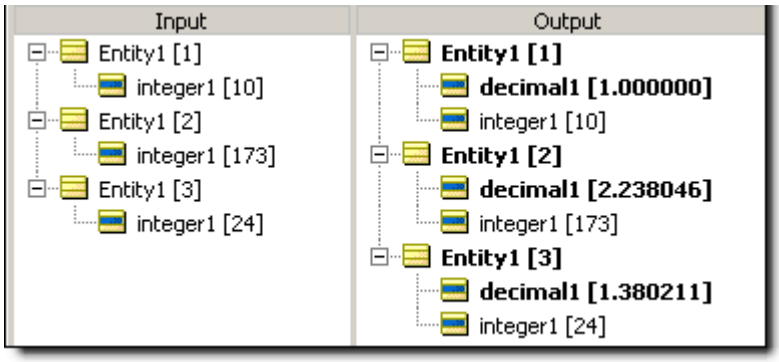
RULESHEET EXAMPLE

The following Rulesheet uses **.log** to calculate the logarithm (base 10) of `integer1` and assign it to `decimal1`.



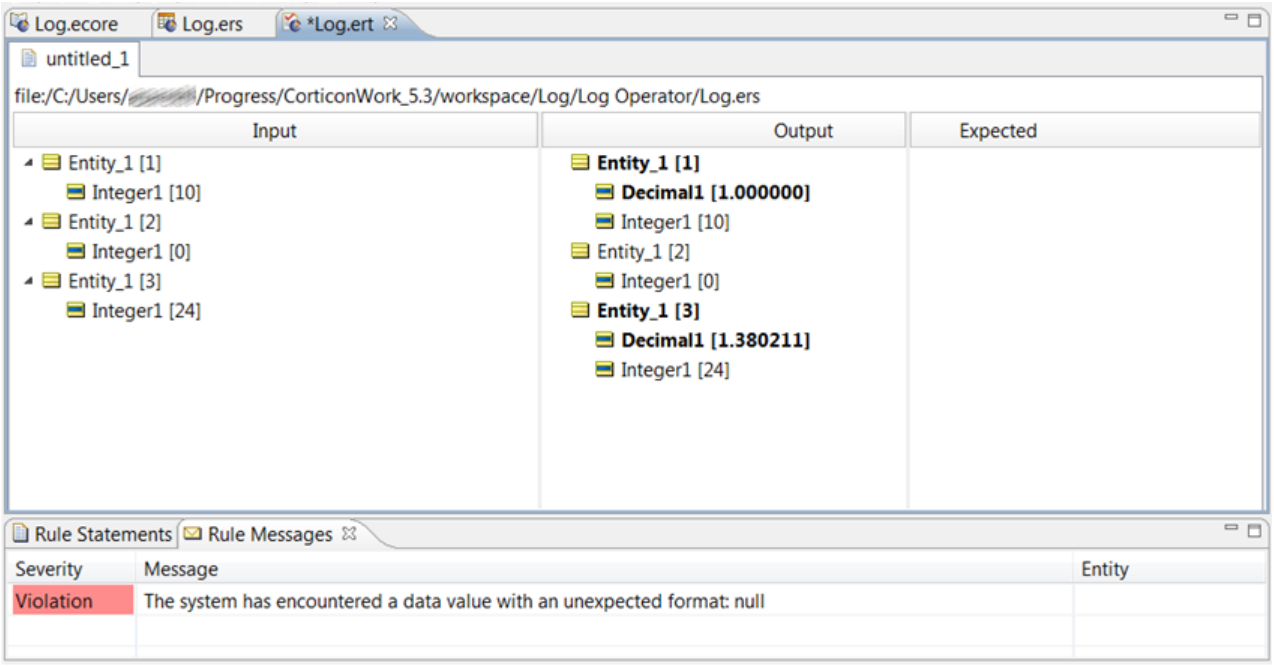
SAMPLE RULETEST 1

A sample Ruletest provides results for three examples of `integer1`. Input and Output panels are shown below:



SAMPLE RULETEST 2

Another sample Ruletest for three examples of `integer1` where one example is equal to zero (0). The resulting error is illustrated below. Notice that the error encountered in the second example causes all Rulesheet execution to halt, leaving the third example unprocessed.



Logarithm (Base x)

SYNTAX

`<Number>`.log(`<Decimal>`)

DESCRIPTION

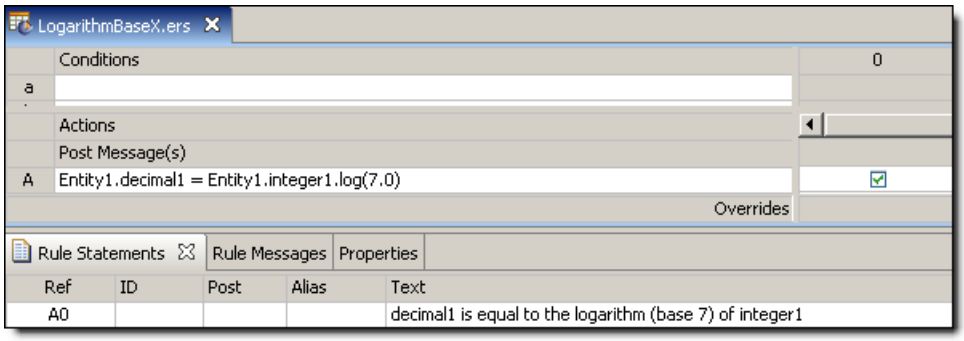
Returns a Decimal value equal to the logarithm (base `<Decimal>`) of `<Number>`. If `<Number>` is equal to `0` (zero) an error is returned when the rule is executed.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

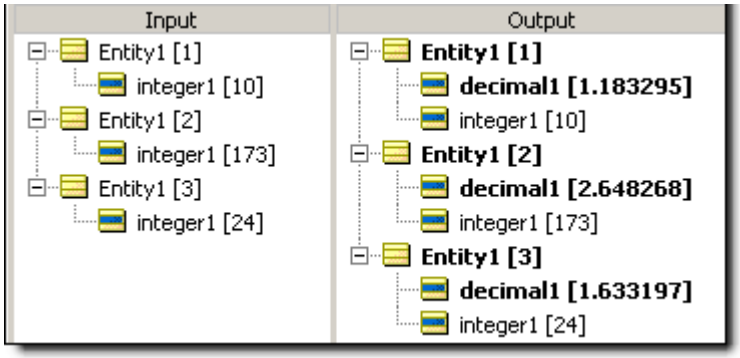
RULESHEET EXAMPLE

The following Rulesheet uses **.log** to calculate the logarithm (base 7.0) of `integer1` and assign it to `decimal1`.



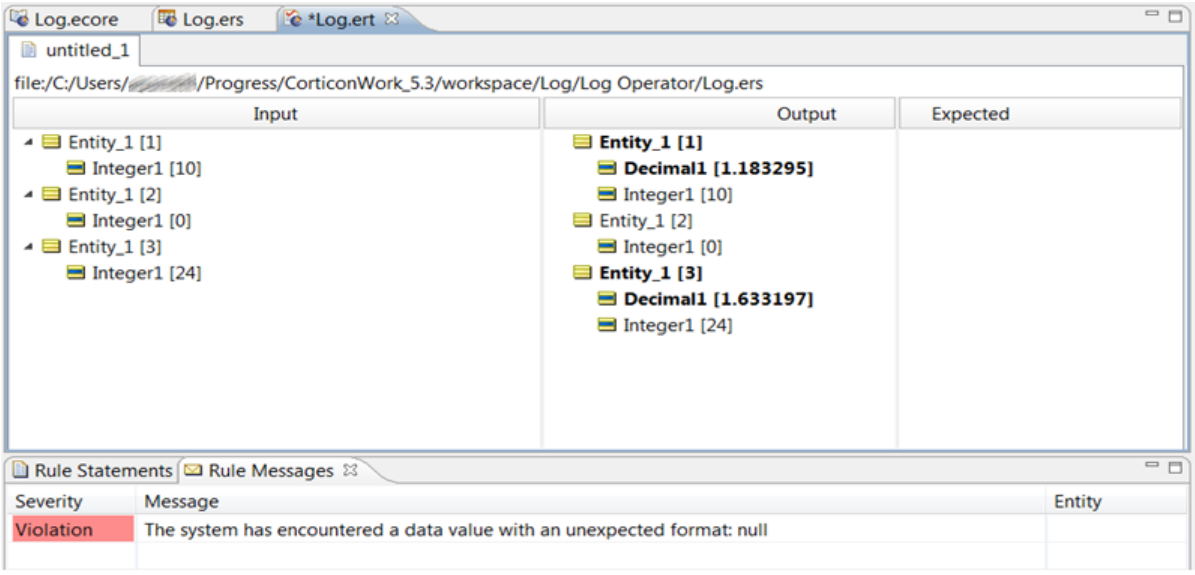
SAMPLE RULETEST 1

A sample Ruletest provides results for three examples of `integer1`. Input and Output panels are shown below:



SAMPLE RULETEST 2

Another sample Ruletest for three examples of `integer1` where one example is equal to zero (0). The resulting error is illustrated below.



Lowercase

SYNTAX

`<String>`.toLower

DESCRIPTION

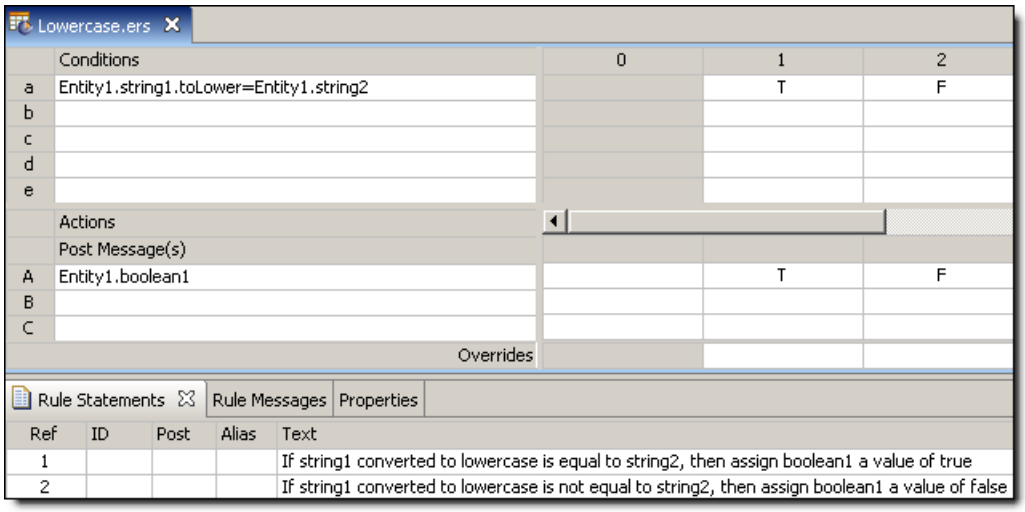
Converts all characters in `<String>` to lowercase characters.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

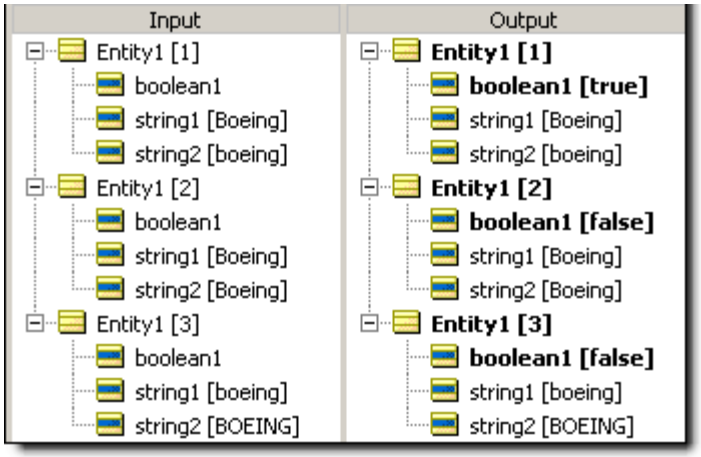
RULESHEET EXAMPLE

The following Rulesheet uses **.toLower** to convert `string1` to lowercase, compare its value with `string2`, and assign a value to `boolean1` based on the results of the comparison.



SAMPLE RULETEST

A sample Ruletest provides three examples of `string1` and `string2`. Input and Output panels are shown below:



Maximum value

SYNTAX

`<Number1>`.max(`<Number2>`)

DESCRIPTION

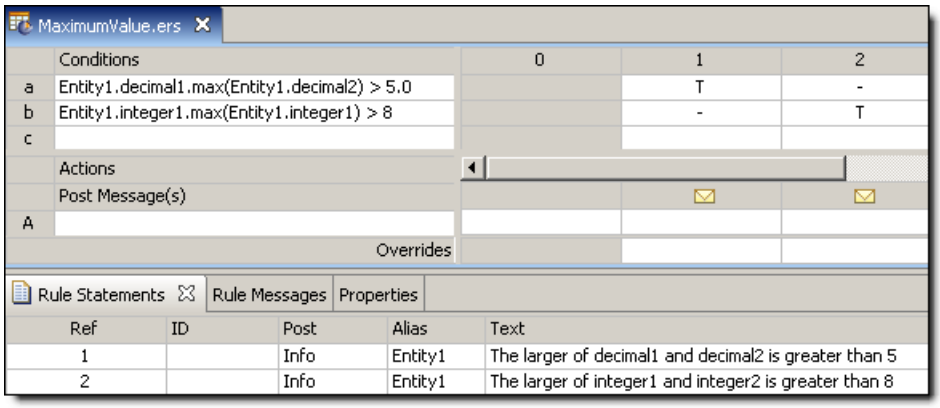
Returns either `<Number1>` or `<Number2>`, whichever is greater.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

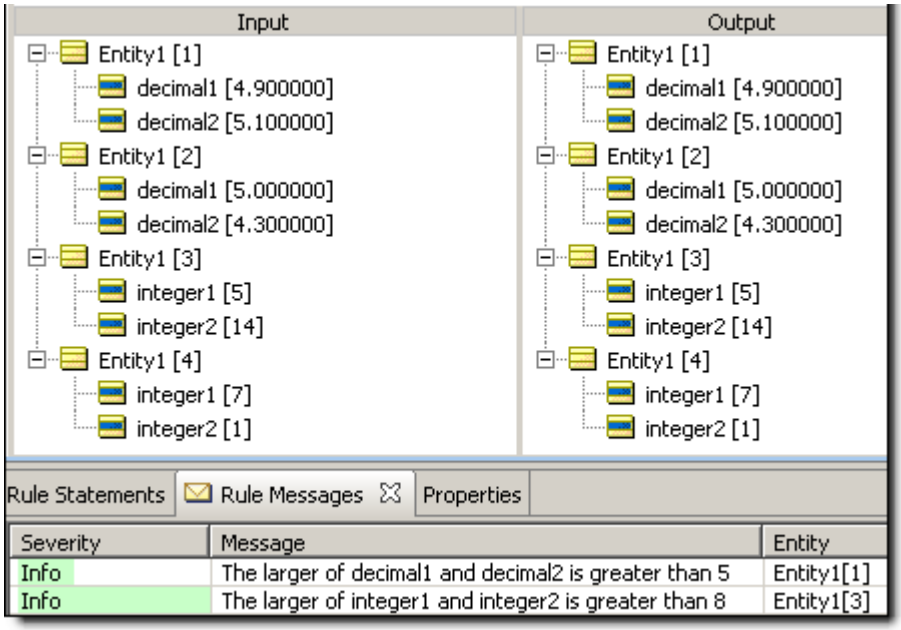
RULESHEET EXAMPLE

The following Rulesheet uses **.max** to compare the values of `decimal1` and `decimal2`, and `integer1` and `integer2`, and posts a message based on their size relative to `5.0` and `8`, respectively.



SAMPLE RULETEST

A sample Ruletest provides four examples, two using `decimal1` and `decimal2`, and two using `integer1` and `integer2` as input data.



Maximum value (Collection)

SYNTAX

`<Collection.attribute>` -> max

DESCRIPTION

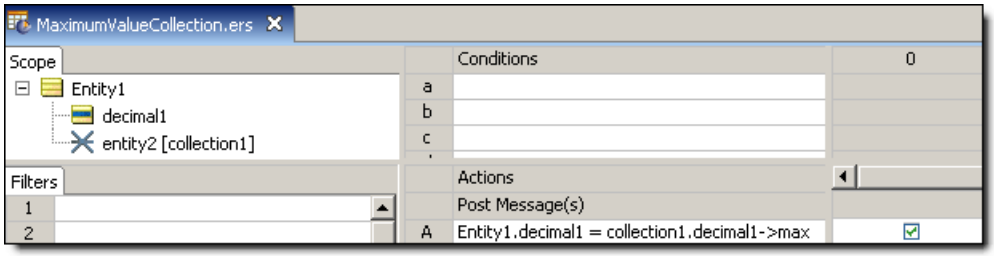
Returns the highest value of `<attribute>` for all elements in `<Collection>`. `<attribute>` must be a numeric data type. `<Collection>` must be expressed as a unique alias.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

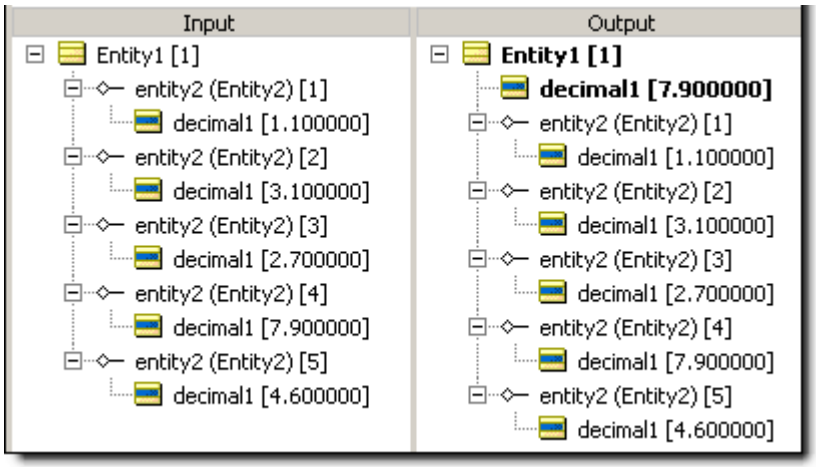
RULESHEET EXAMPLE

The following Rulesheet uses **max** to identify the highest value of `decimal1` in all elements of `collection1`, then assign it to `Entity1.decimal1`.



SAMPLE RULETEST

A sample collection contains five elements, each with a value of `decimal1`.



Minimum value

SYNTAX

`<Number1>`.min(`<Number2>`)

DESCRIPTION

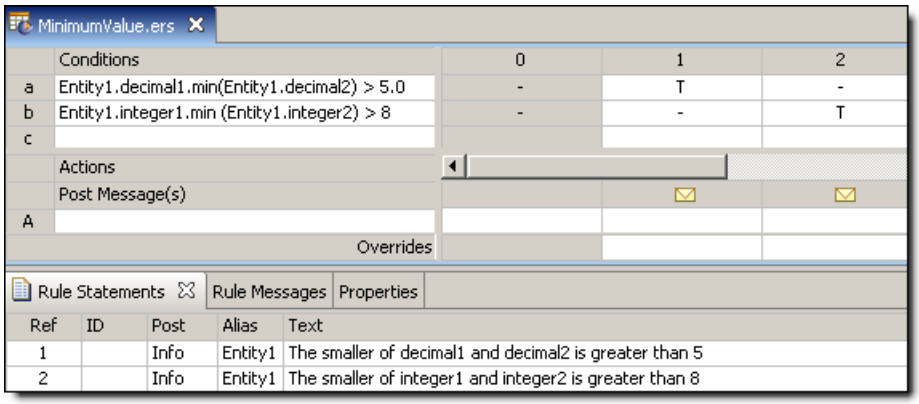
Returns either `<Number1>` or `<Number2>`, whichever is smaller.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

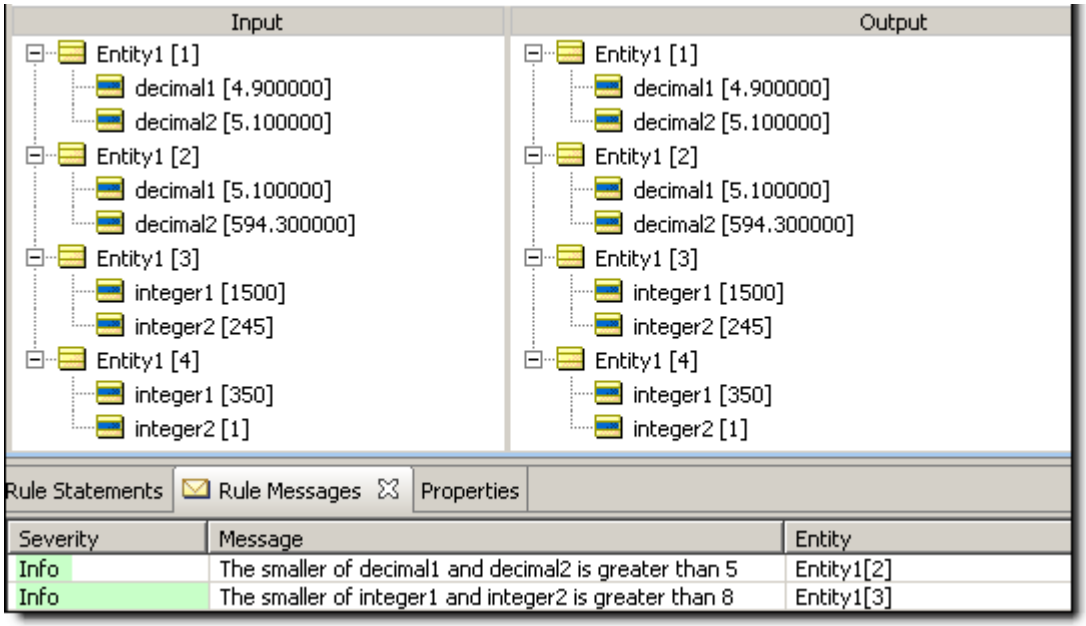
RULESHEET EXAMPLE

The following Rulesheet uses **.min** to compare the values of `decimal1` and `decimal2`, and `integer1` and `integer2`, and posts a message based on their size relative to `5.0` and `8`, respectively.



SAMPLE RULETEST

A sample Ruletest provides four examples, two using decimal inputs, and two using integers.



Minimum value (Collection)

SYNTAX

`<Collection.attribute>` -> min

DESCRIPTION

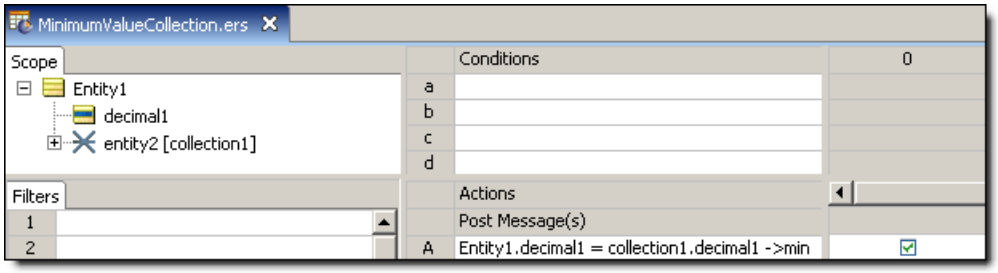
Returns the lowest value of `<attribute>` for all elements in `<Collection>`. `<attribute>` must be a numeric data type. `<Collection>` must be expressed as a unique alias.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

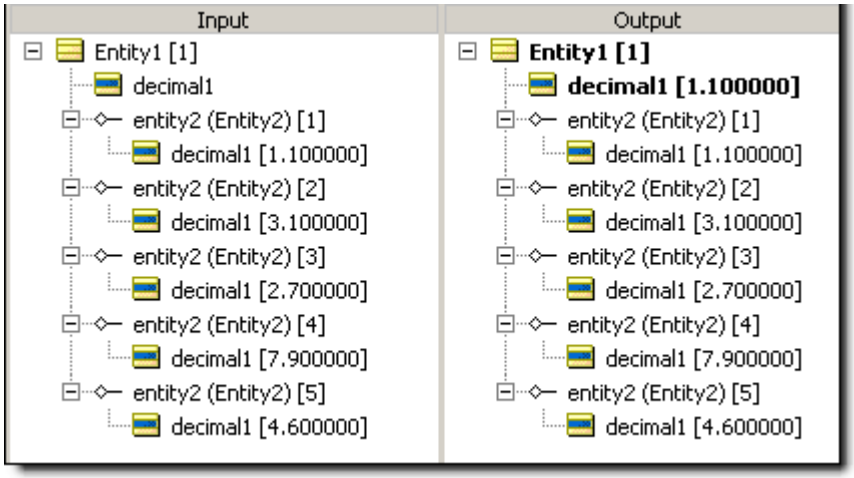
RULESHEET EXAMPLE

The following Rulesheet uses **min** to identify the lowest value of `decimal1` in all elements of `collection1`, then assign it to `Entity1.decimal1`.



SAMPLE RULETEST

A sample collection contains five elements, each with a value of `decimal1`.



Minute

SYNTAX

`<DateTime>`.min

`<Time>`.min

DESCRIPTION

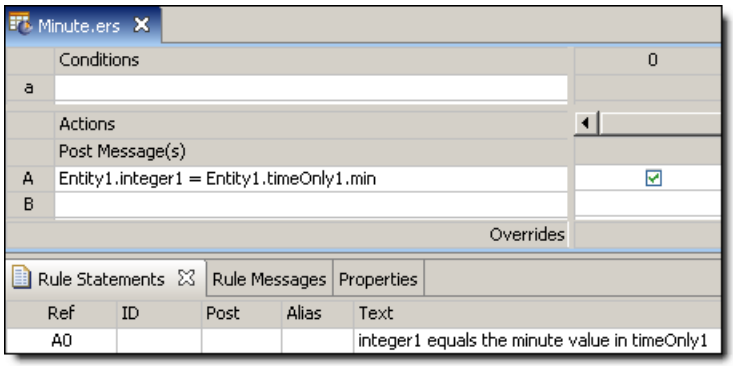
Returns the minute portion of `<DateTime>` or `<Time>` as an Integer between 0 and 59. This operator cannot be used with Date attributes because no time information is present.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

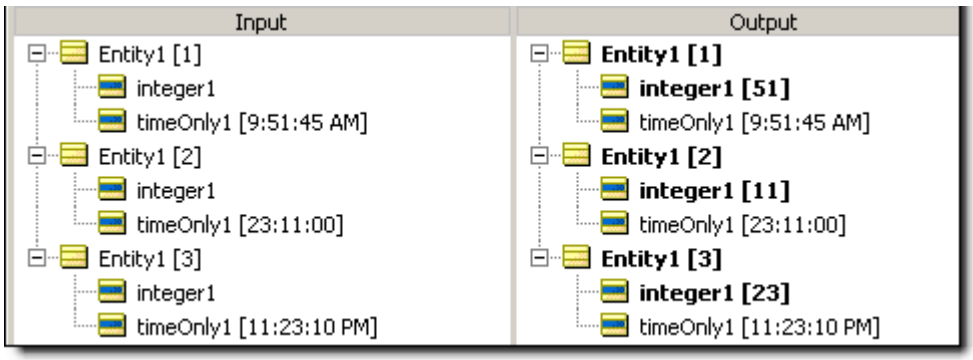
RULESHEET EXAMPLE

The following Rulesheet uses **.min** to evaluate `dateTime1` and assign the minute value to `integer1`.



SAMPLE RULETEST

A sample Ruletest provides three examples of `dateTime1`. Input and Output panels are shown below:



Minutes between

SYNTAX

`<DateTime1>`.minsBetween(`<DateTime2>`)

`<Time1>`.minsBetween(`<Time2>`)

DESCRIPTION

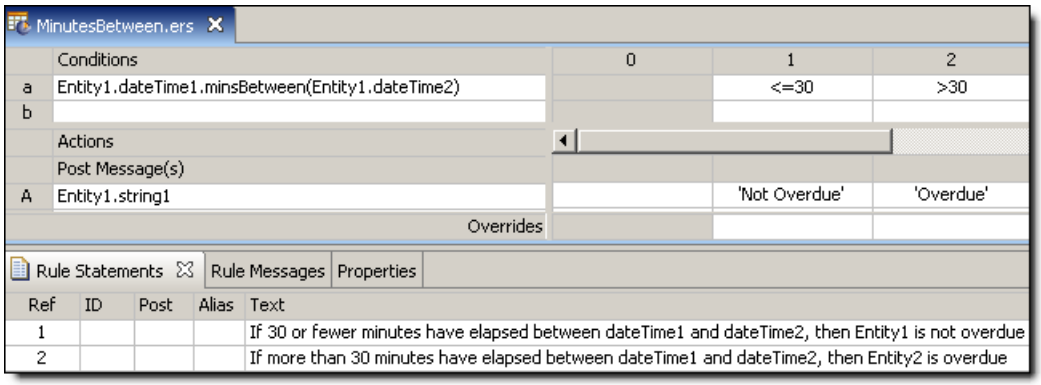
Returns the Integer number of minutes between DateTimes or between Times. The function calculates the number of milliseconds between the two dates and divides that number by 60,000 (the number of milliseconds in a minute). The decimal portion is then truncated. If the two dates differ by less than a full minute, the returned value is zero. This function returns a positive number if `<DateTime2>` is later than `<DateTime1>`.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

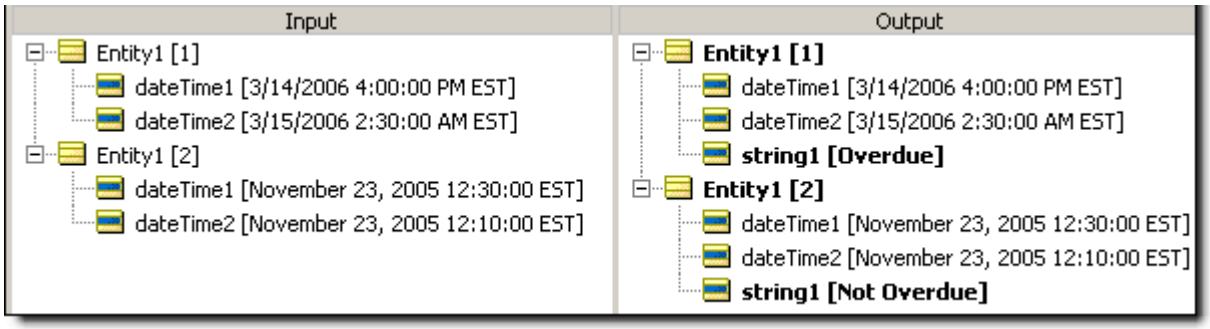
RULESHEET EXAMPLE

The following Rulesheet uses **.minsBetween** to determine the number of minutes that have elapsed between `dateTime1` and `dateTime2`, compare it to the Values set, and assign a value to `string1`.



SAMPLE RULETEST

A sample Ruletest provides `dateTime1` and `dateTime2` for two examples. Input and Output panels are shown below. Notice the different masks (formats) used for the DateTime data.



Mod

SYNTAX

`<Integer1>`.mod(`<Integer2>`)

DESCRIPTION

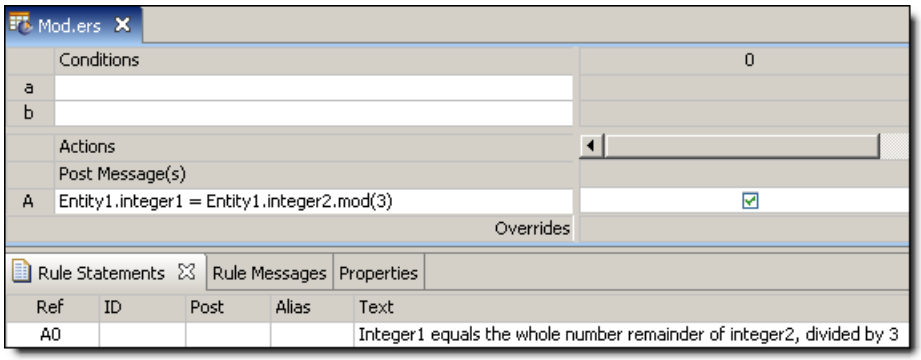
Returns the whole number remainder that results from dividing <`Integer1`> by <`Integer2`>. If the remainder is a fraction, then `0` (zero) is returned.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

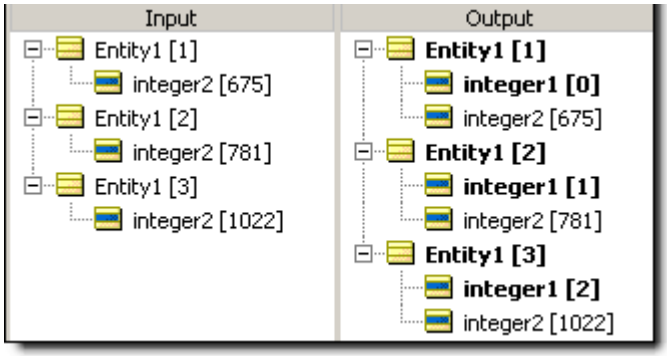
RULESHEET EXAMPLE

The following Rulesheet > uses **.mod** to calculate the whole number remainder resulting from the division of `integer2` by 3. The result is assigned to `integer1`.



SAMPLE RULETEST

A sample Ruletest provides three examples of `integer2`. Input and Output panels are shown below.



Month

SYNTAX

<`DateTime`>.month

<`Date`>.month

DESCRIPTION

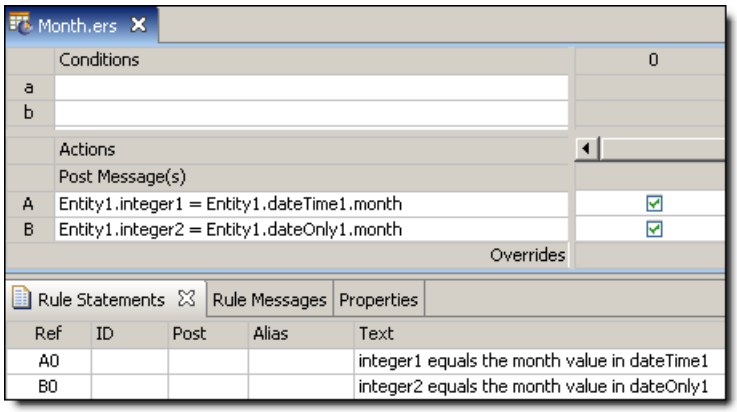
Returns the month in <`DateTime`> or <`Date`> as an Integer between 1 and 12.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

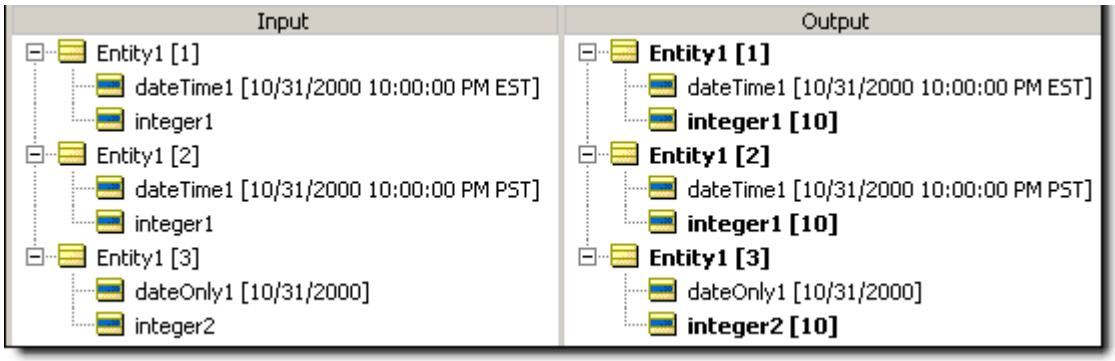
RULESHEET EXAMPLE

The following Rulesheet uses **.month** to evaluate `dateTime1` and `dateOnly1` and assign the month value to `integer1` and `integer2`, respectively.



SAMPLE RULETEST

A sample Ruletest provides three examples of `dateTime1` or `dateOnly1`. Input and Output panels are shown below. The month returned is independent of the machine running the *`Ruletest`* and only depends on the locale/timezone of the data itself.



Months between

SYNTAX

<`DateTime1`>.monthsBetween(<`DateTime2`>)

<`Date1`>.monthsBetween(<`Date2`>)

DESCRIPTION

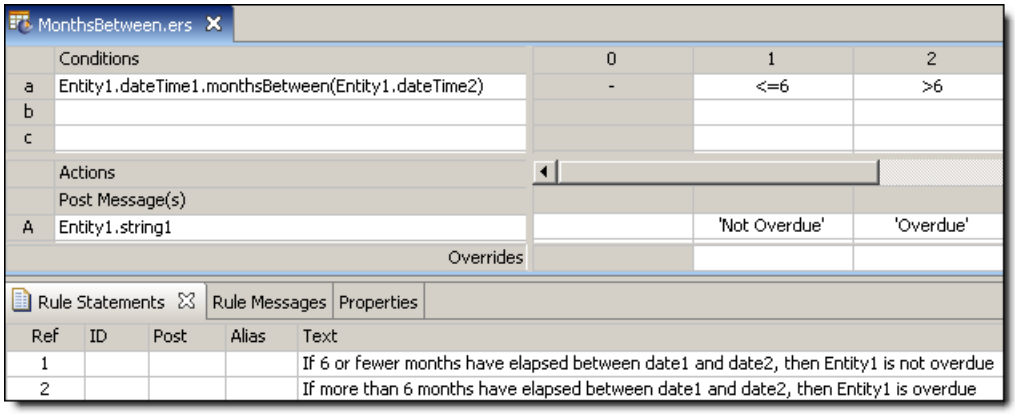
Returns the Integer number of months between DateTimes or between Dates. The month and year portions of the date data are subtracted to calculate the number of elapsed months. The day portions are ignored. If the month and year portions are the same, the result is zero. This function returns a positive number if `<DateTime2>` is later than `<DateTime1>`.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

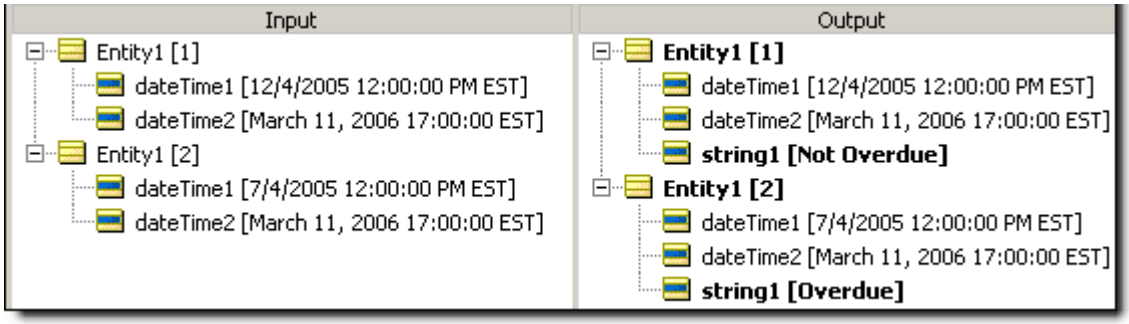
RULESHEET EXAMPLE

The following Rulesheet uses **.monthsBetween** to determine the number of months that have elapsed between `dateTime1` and `dateTime2`, compare it to the values in the Condition Cells, and assign a value to `string1`.



SAMPLE RULETEST

A sample Ruletest provides `dateTime1` and `dateTime2` for two examples. Input and Output panels are shown below. Notice the variations in date masks (formats).



Multiply

SYNTAX

`<Number1> * <Number2>`

DESCRIPTION

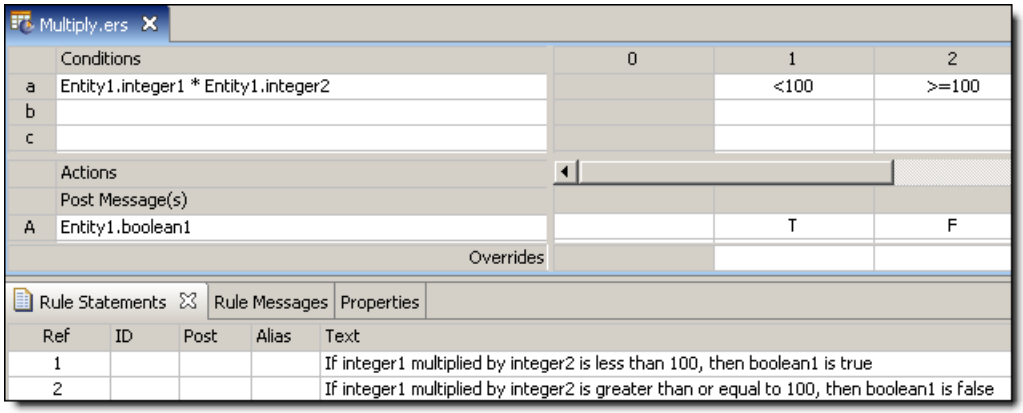
Multiplies <Number1> by <Number2>. The resulting data type is the more expansive of those of <Number1> and <Number2>.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

RULESHEET EXAMPLE

This sample Rulesheet uses **multiply** to multiply `integer1` and `integer2` and compare the result to `100`



Multiply.ers

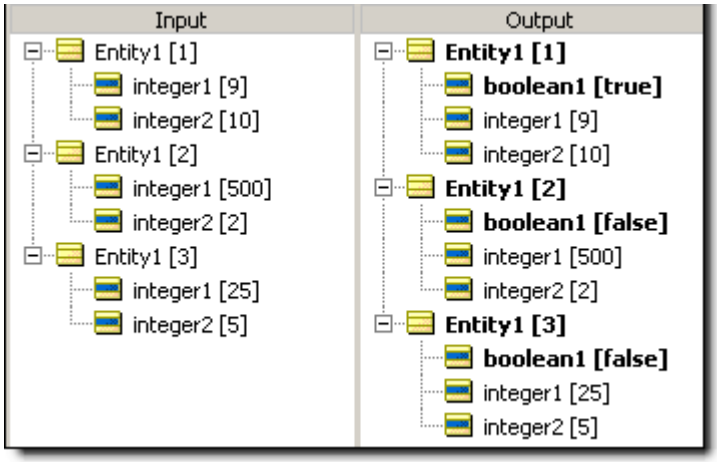
	Conditions	0	1	2
a	Entity1.integer1 * Entity1.integer2		<100	>=100
b				
c				
	Actions			
	Post Message(s)			
A	Entity1.boolean1		T	F
	Overrides			

Rule Statements | Rule Messages | Properties

Ref	ID	Post	Alias	Text
1				If integer1 multiplied by integer2 is less than 100, then boolean1 is true
2				If integer1 multiplied by integer2 is greater than or equal to 100, then boolean1 is false

SAMPLE RULETEST

A sample Ruletest provides three examples of `integer1` and `integer2`. Input and Output panels are shown below.



Natural logarithm

SYNTAX

<Number>.ln

DESCRIPTION

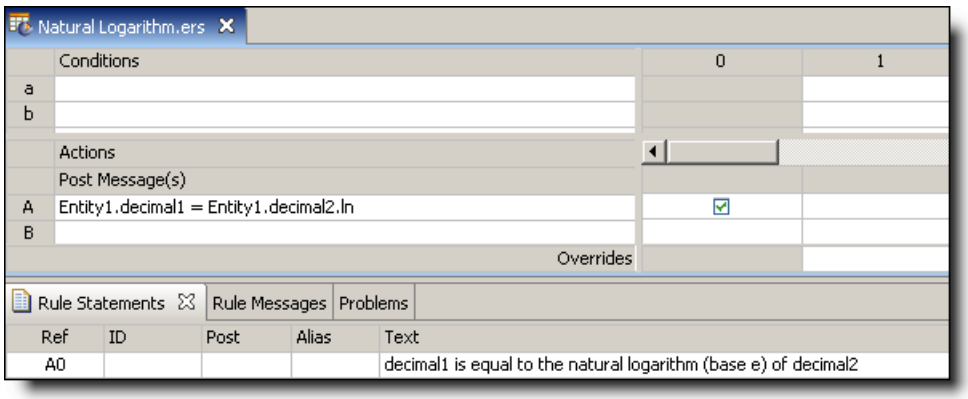
Returns a Decimal value equal to the natural logarithm (base e) of <Number>. If <Number> is equal to 0 (zero), an error is returned when the rule is executed. This error will halt execution for all data present.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

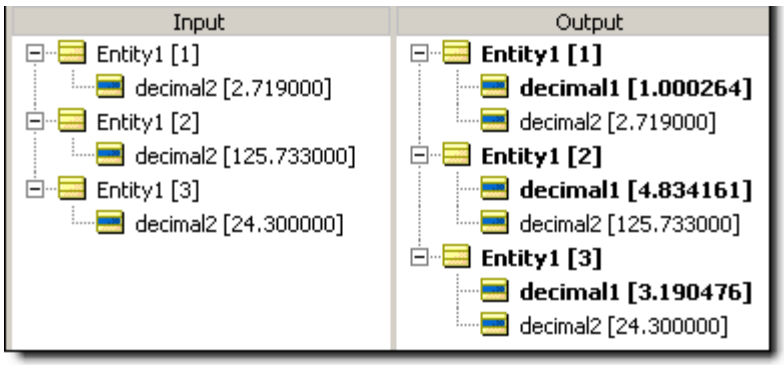
RULESHEET EXAMPLE

The following Rulesheet uses **.ln** to calculate the natural logarithm of decimal2 and assign it to decimal1.



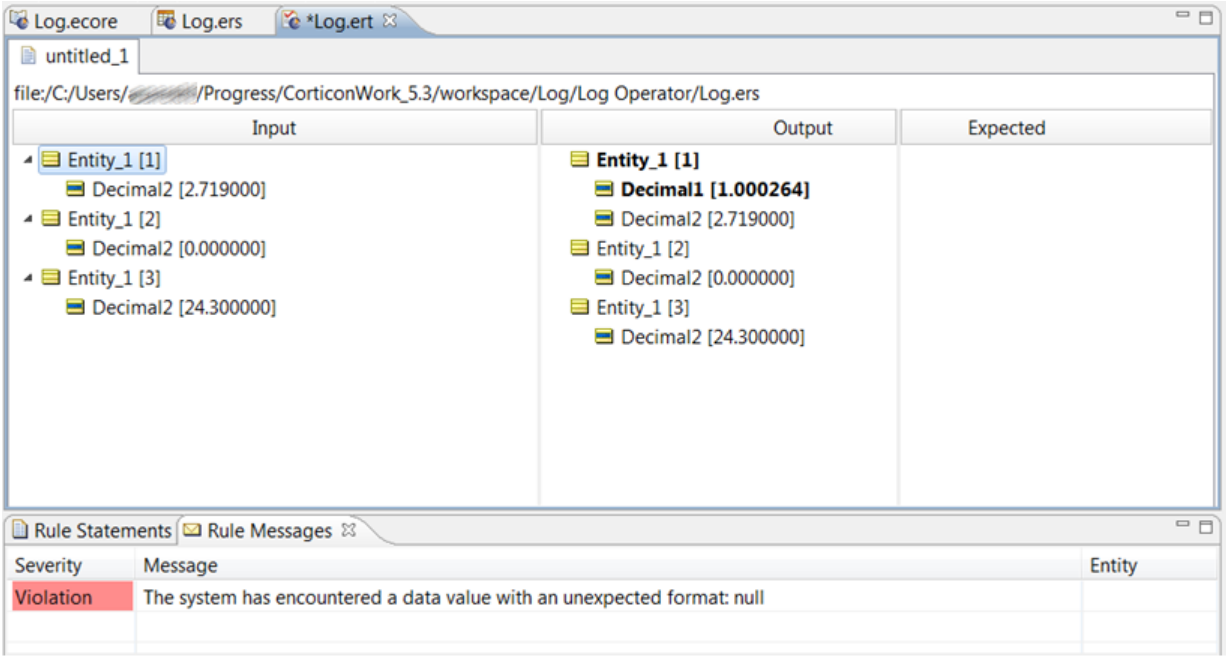
SAMPLE RULETEST 1

A sample Ruletest provides results for three examples of decimal2. Input and Output panels are shown below:



SAMPLE RULETEST 2

Another sample Ruletest for three examples of `decimal2` where one example is equal to zero (0). The resulting error is illustrated below:



New

SYNTAX

<`Entity`>.new[<`Expression1`>,<`Expression2`>...]

DESCRIPTION

creates a new <`Entity`> with attribute values defined by optional <`Expression`>. Expressions (when present) should be written as assignments in the form: *attribute* = *value*. The attribute used in <`Expression`> (when present) must be an attribute of <`Entity`>.

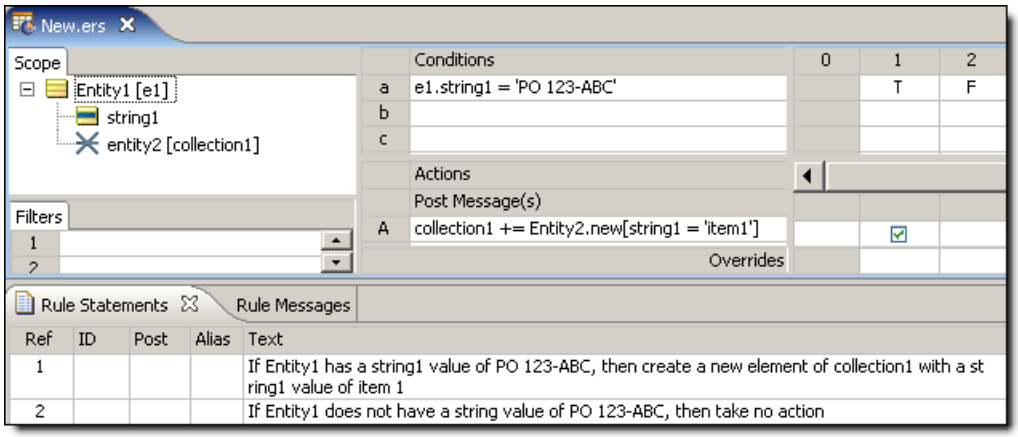
USAGE RESTRICTIONS

The Operators row in the table of Summary Table of Vocabulary Usage Restriction does not apply. Special exceptions: **new** may only be used in Action Rows (section 5 in Sections of Rulesheet that correlate with usage restrictions).

AND logic, expressed as commas between <`Expressions`>, is supported within the square brackets `[..]`. **OR** logic is not supported within the square brackets `[..]`.

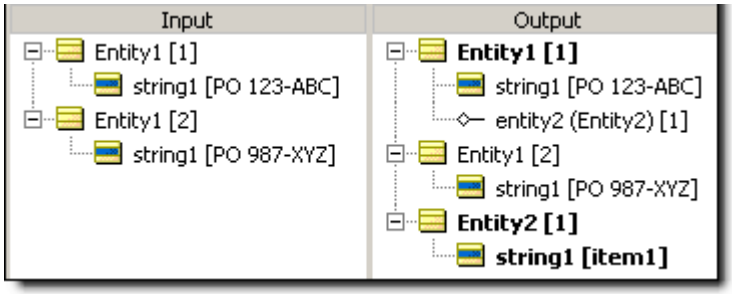
RULESHEET EXAMPLE

The following Rulesheet uses **.new** to create a new `Entity2` element in `collection1` when `Entity1` has a `string1` value equal to `"PO 123-ABC"`. An alias is not required by the **.new** operator, because it is possible to create a new entity at the root level, without inserting it into a collection. The collection alias used here is required by the `+=` (Associate Element to collection) operator.



SAMPLE RULETEST

A sample Ruletest provides 2 collections of `Entity1`. Input and Output panels are illustrated below:



New unique

SYNTAX

<`Entity`>.newUnique[<`Expression1`>,<`Expression2`>...]

DESCRIPTION

newUnique is an unusual operator in that it contains both action *and* condition logic. When an Action containing this operator is executed, a new <`Entity`> will be created only if no other entity exists with the characteristics defined by <`Expression1`> **and** <`Expression2`>, etc. <`Expression1`> and <`Expression2`> are optional. If no expression is present within the square brackets `[..]`, the **newUnique** operator will create a new entity only if none currently exists in memory.

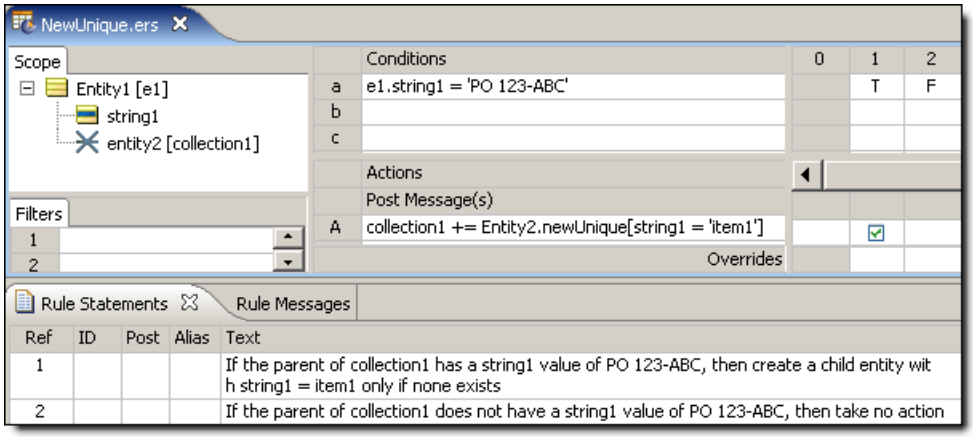
USAGE RESTRICTIONS

The Operators row in the table of Summary Table of Vocabulary Usage Restriction does not apply. Special exceptions: **newUnique** may only be used in Action Rows (section 5 in Sections of Rulesheet that correlate with usage restrictions).

AND logic, expressed as commas between <`Expressions`>, is supported within the square brackets `[..]`. **OR** logic is not supported within the square brackets `[..]`.

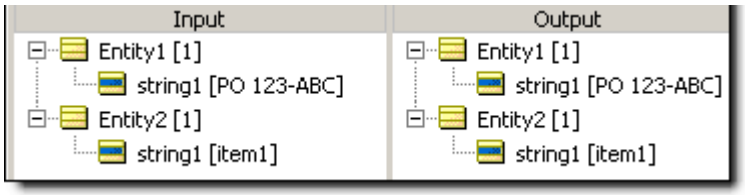
RULESHEET EXAMPLE

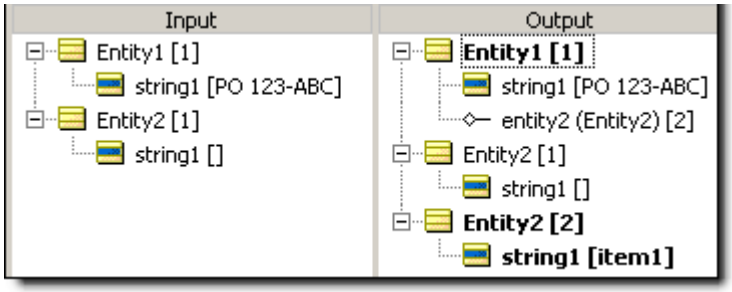
The following Rulesheet uses **.newUnique** to create a new `Entity2` element with `string1="item1"`, and add it to `collection1` only if no existing `Entity2` already has `string1="item1"`. A collection alias is not required by the **.newUnique** operator because it is possible to create a new entity at the root level, without inserting it into a collection. The collection alias used here is required by the `+=` (Associate Element to collection) operator.

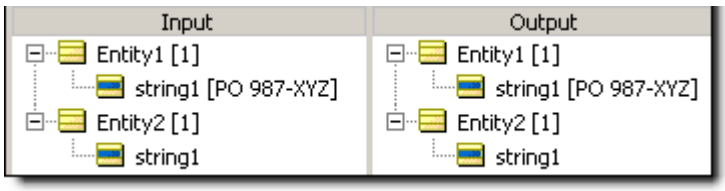


SAMPLE RULETEST 1

Each of three sample tests provides different combinations of `Entity1` and `Entity2`. Input and Output panels are illustrated below:







Not

SYNTAX

not <`Expression`>

DESCRIPTION

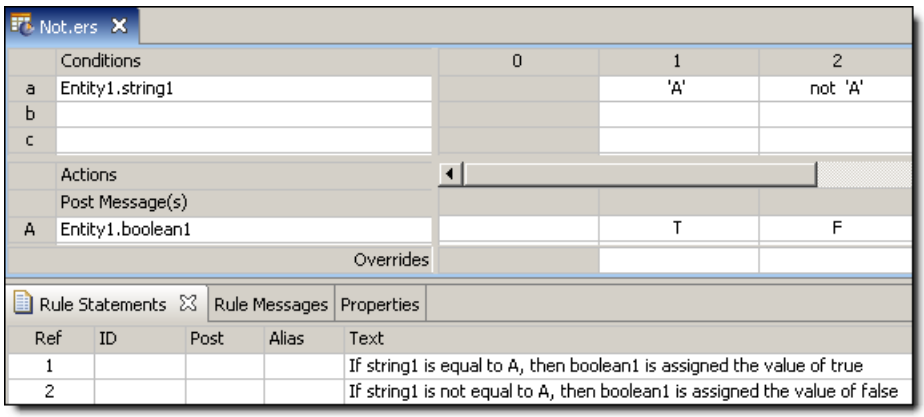
Returns the negation of the truth value of <`Expression`>.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies, with the following special exception: **not** may also be used in Conditional Cells.

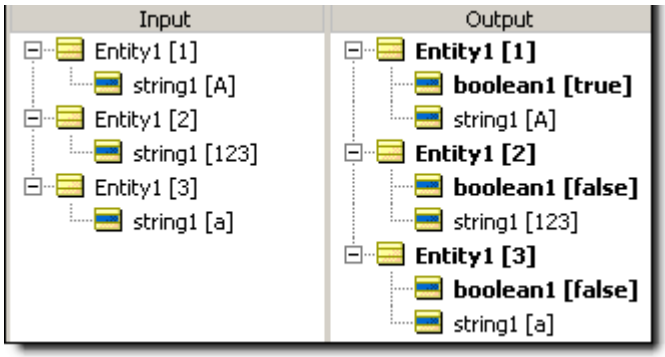
RULESHEET EXAMPLE

The following Rulesheet uses **not** to negate the value of `A` in the Condition Cell of rule 2. **Not** may only be used in this manner if there is at least one other value (including other or null) present in the Condition Cells values drop-down list (i.e., there must be at least one alternative to the value negated by **not**).



SAMPLE RULETEST

A sample Ruletest provides three examples of `string1`. Input and Output panels are shown below:



Not empty

SYNTAX

<`Collection`> ->notEmpty

DESCRIPTION

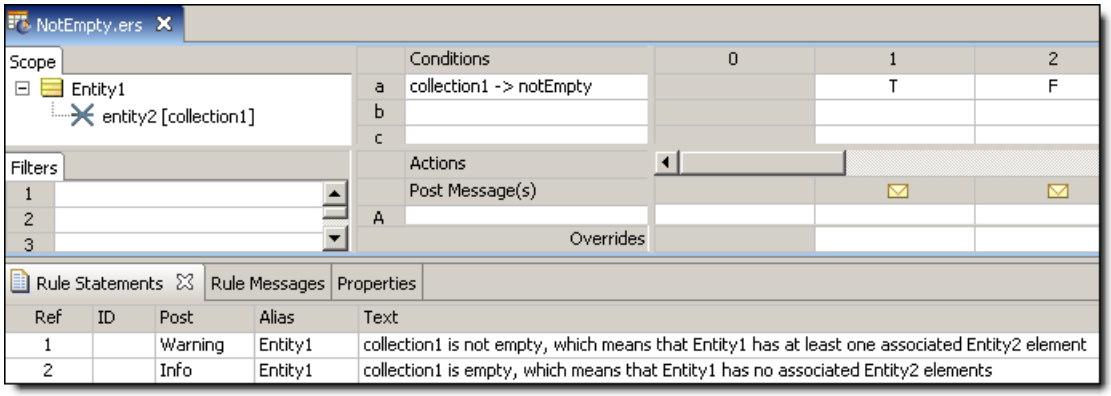
Returns a value of true if <`Collection`> contains *at least one* element. **->notEmpty** does not check for attribute values, but instead checks for the *existence of elements within a collection*. As such, it requires the use of a unique alias to represent the collection being tested.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

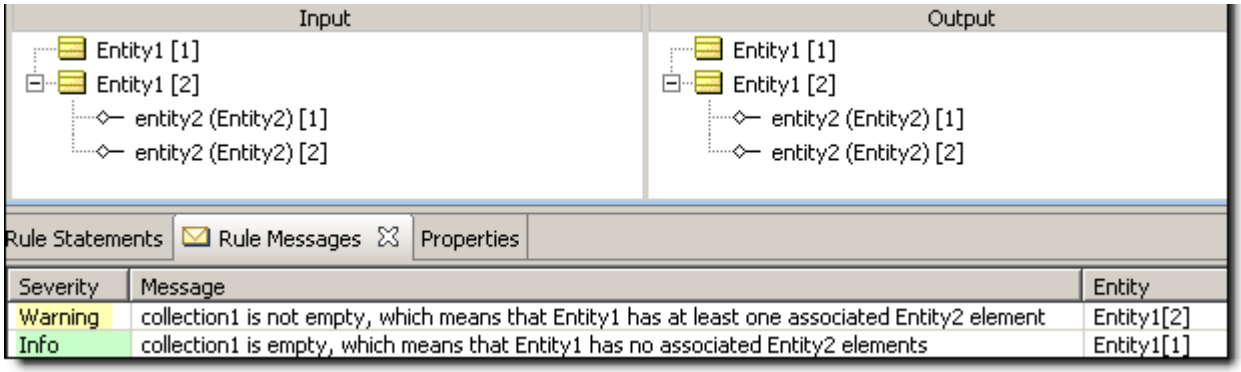
RULESHEET EXAMPLE

This sample Rulesheet uses the **notEmpty** function to determine if `collection1` has elements. Note the use of unique alias `collection1` to represent the collection of `Entity2` associated with `Entity1`.



SAMPLE RULETEST

A sample Ruletest provides two collections. The following illustration shows Input and Output panels



Not equal to

SYNTAX

Boolean	<Expression1> <> <Expression2>
DateTime*	<DateTime1> <> <DateTime2>
Number	<Number1> <> <Number2>
String	<String1> <> <String2>

DESCRIPTION

Boolean	Returns a value of true if <Expression1> does not have the same truth value as <Expression2>.
DateTime*	Returns a value of true if <DateTime1> does not equal <DateTime2>. This is equivalent to <DateTime1> not occurring "on" <DateTime2>
Number	Returns a value of true if <Number1> is not equal to <Number2>. Different numeric data types may be compared in the same expression.
String	Returns a value of true if <String1> is not equal to <String2>. Studio uses Character precedence: Unicode & Java Collator on page 189 to determine character precedence.

*includes DateTime, Date, and Time data types

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

RULESHEET EXAMPLE

The following Rulesheet uses **not equal to** to test whether `decimal1` equals `decimal2`, and assign a value to `string1` based on the result of the comparison.

NotEqualTo.ers

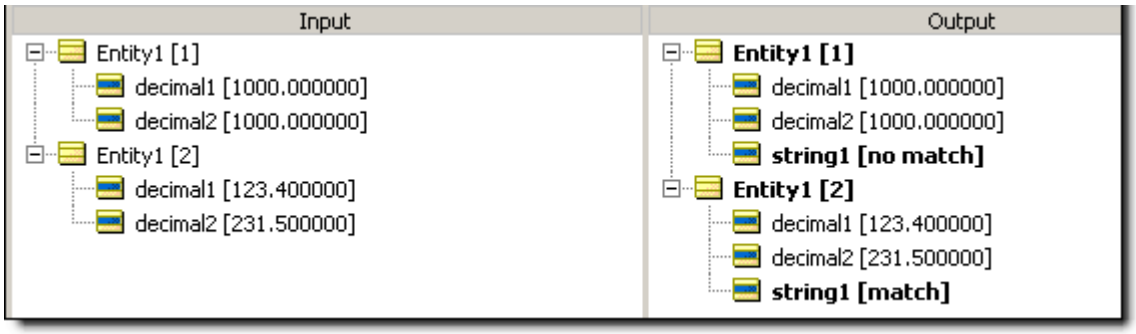
	Conditions	0	1	2
a	Entity1.decimal1 <> Entity1.decimal2		T	F
b				
c				
	Actions			
	Post Message(s)			
A	Entity1.string1		'match'	'no match'
	Overrides			

Rule Statements | Rule Messages | Properties

Ref	ID	Post	Alias	Text
1				If decimal1 does not equal decimal2, then assign a value of [no match] to string1
2				If decimal1 equals decimal2, then assign a value of [match] to string1

SAMPLE RULETEST

A sample Ruletest provides two examples. Input and Output panels are shown below:



Now

SYNTAX

now

DESCRIPTION

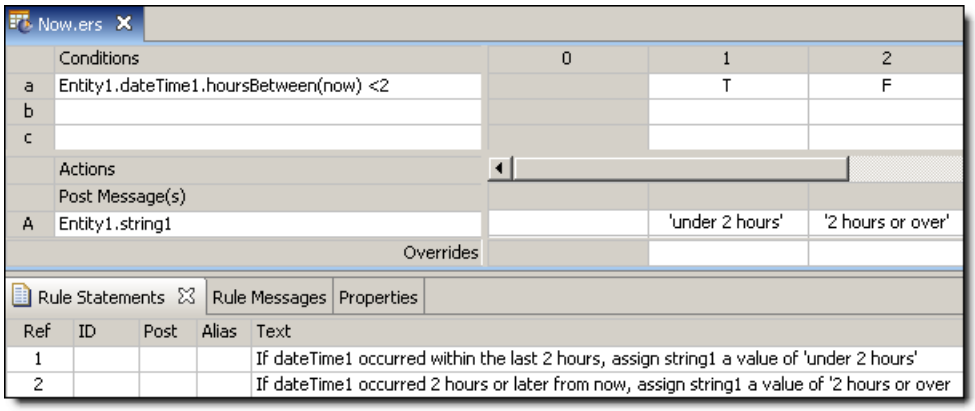
Returns the current system date and time when the rule is executed. This DateTime value is assigned the first time **now** is used in a Decision Service (Rule Set), then remains constant until the Decision Service finishes execution, regardless of how many additional times it is used. This means that every rule in a Rule Set containing **now** will use the same DateTime value.

USAGE RESTRICTIONS

The Literals row in the table of Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

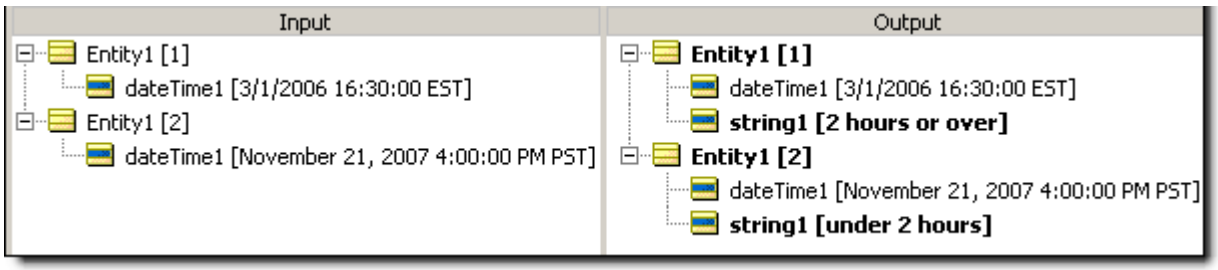
RULESHEET EXAMPLE

The following Rulesheet uses **now** to determine how many hours have elapsed between now and `dateTime1` (see .hoursBetween for more details on this operator), and assign a value to `string1` based on the result.



SAMPLE RULETEST

A sample Ruletest provides two examples of `dateTime1`. Assume **now** is equal to `Thursday,March 16,2006 8:50:00 AM EST`. Input and Output panels are shown below. Notice the variation in DateTime masks (formats).



Null

SYNTAX

null

DESCRIPTION

The null value corresponds to one of three different scenarios:

1. the absence of an attribute in a Ruletest Input pane or request message
2. the absence of data for an attribute in a Ruletest (the value zero counts as data)
3. a business object (supplied by an external application) that has an instance variable of null

A **null** value is different from an empty String (for String data types) or zero for numeric data types. An empty String is represented in a Ruletest as [] -- open then close square brackets. Any attribute value, including any empty strings, may be reset to **null** in a Ruletest by right-clicking the attribute and choosing **Set to null**. Mandatory attributes (property set in the Vocabulary) may not have a null value.

USAGE RESTRICTIONS

The Literals row in the table of Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

RULESHEET EXAMPLE

The following Rulesheet uses **null** to test for the existence of a real value in `decimal1`, and assign a value to `boolean1` as a result.

Null.ers

	Conditions	0	1	2
a	Entity1.decimal1		null	other
b				
c				
	Actions			
	Post Message(s)			
A	Entity1.boolean1		T	F
	Overrides			

Rule Statements | Rule Messages | Properties

Ref	ID	Post	Alias	Text
1		Warning	Entity1	If decimal1 has the value of null, then assign boolean1 the value of true
2		Info	Entity1	If decimal1 has any value other than null (any real number), then assign boolean1 the value of false

SAMPLE TEST

A sample Ruletest provides four examples of `decimal1`. Input and Output panels are illustrated below. Posted messages are not shown.

Input

- Entity1 [1]
  - decimal1 [4.000000]
- Entity1 [2]
  - decimal1
- Entity1 [3]
  - decimal1 [0.000000]
- Entity1 [4]
  - decimal1 [-13.500000]

Output

- **Entity1 [1]**
  - **boolean1 [false]**
  - decimal1 [4.000000]
- **Entity1 [2]**
  - **boolean1 [true]**
  - decimal1
- **Entity1 [3]**
  - **boolean1 [false]**
  - decimal1 [0.000000]
- **Entity1 [4]**
  - **boolean1 [false]**
  - decimal1 [-13.500000]

Rule Statements | Rule Messages | Properties

Severity	Message	Entity
Warning	If decimal1 has the value of null, then assign boolean1 the value of true	Entity1[2]
Info	If decimal1 has any value other than null (any real number), then assign boolean1 the value of false	Entity1[4]
Info	If decimal1 has any value other than null (any real number), then assign boolean1 the value of false	Entity1[3]
Info	If decimal1 has any value other than null (any real number), then assign boolean1 the value of false	Entity1[1]

Other

SYNTAX

other

DESCRIPTION

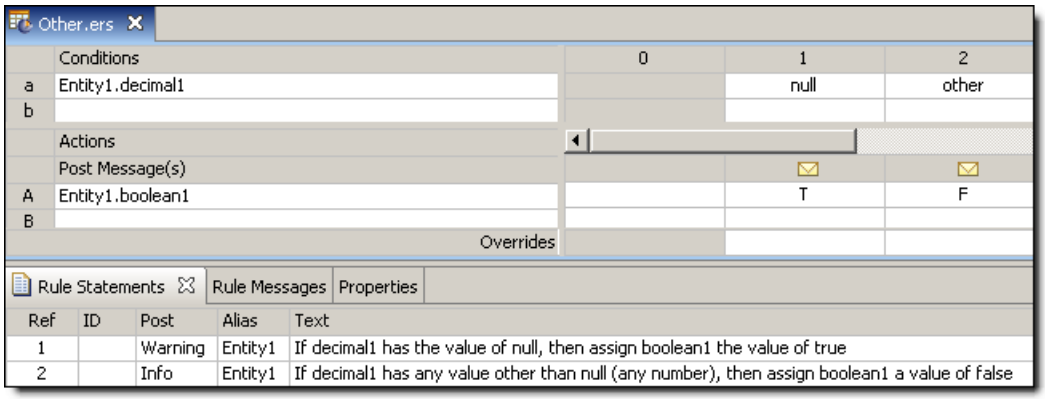
When included in a condition's Values set (the drop-down list of values available in a Conditions Cell), **other** represents any value not explicitly included in the set, including null. If null is explicitly included in the Values set, then **other** does not include null.

USAGE RESTRICTIONS

The Literals row in the table of Summary Table of Vocabulary Usage Restriction does not apply. Special exception: **other** may only be used in Condition Cells (section 4 of the Sections of Rulesheet that correlate with usage restrictions) because it is a non-specific value used in comparisons.

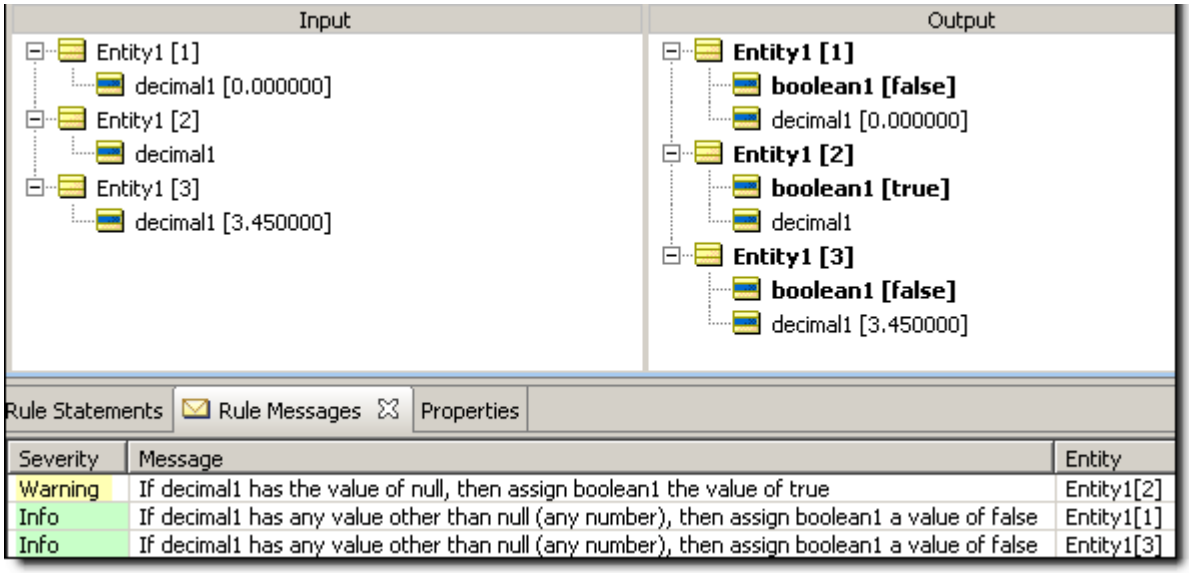
RULESHEET EXAMPLE

The following Rulesheet uses **other** to test the value of `decimal1`. If `decimal1` has any value *other* than null, `boolean1` is assigned the value of `false`.



SAMPLE TEST

A sample Ruletest provides three examples of `decimal1`. Ruletest Input and Output panels are shown below:



Or

SYNTAX

`<Expression1>` or `<Expression2>` or ….

OR may also be used with ->forAll and ->exists expressions

DESCRIPTION

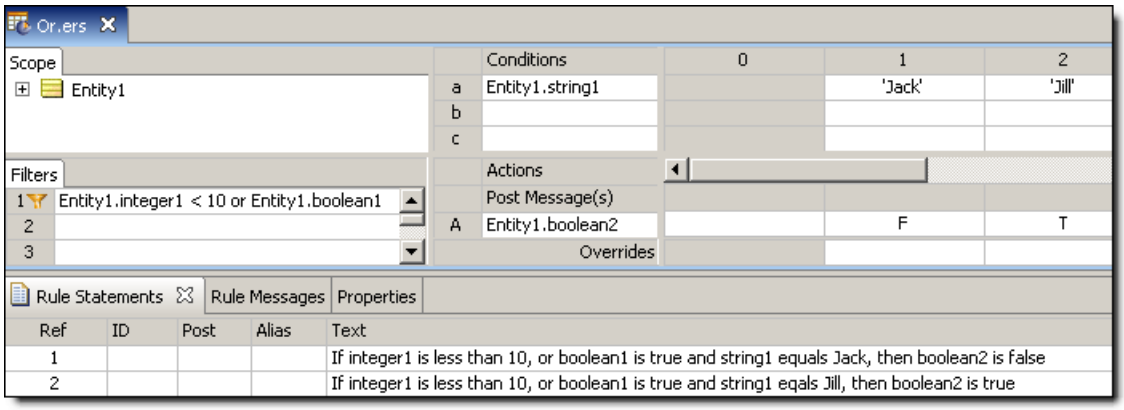
Returns a value of true if either `<Expression1>` or `<Expression2>` evaluates to true. When used between two or more expressions in the Preconditions section, creates a compound filter for the Rulesheet that follows. See *Rule Modeling Guide* for details on using Preconditions as filters. **OR** is not available in the Conditions section because the logical **OR** construction is implemented using multiple Columns in the decision table, or by value sets in Conditions Cells.

USAGE RESTRICTIONS

The Literals row in the table of Sections of Rulesheet that correlate with usage restrictions does not apply. Special exception: **or** may only be used in the Filters section of the Rulesheet to join 2 or more expressions, as shown above, or within ->forAll and ->exists expressions as described in those sections.

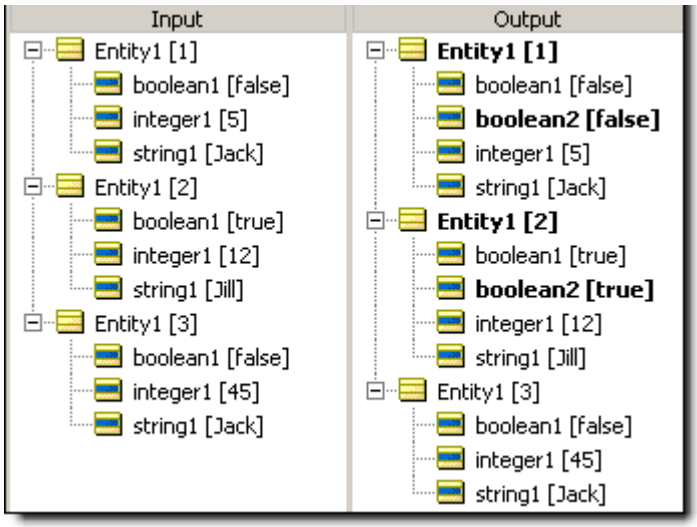
RULESHEET EXAMPLE

The following Rulesheet uses **or** to test the value of `integer1`, `boolean1`, and `string1` to set the value of `boolean2`



SAMPLE TEST

A sample Ruletest provides three examples. Input and Output panels are shown below:



Remove element

SYNTAX

<`Entity`>.remove

<`Collection`>.remove

DESCRIPTION

Removes <`Entity`> or removes elements from <`Collection`> and deletes it/them. If removing from a collection, then using a unique alias to represent the collection is optional since **.remove** is not a collection operator. If any elements in <`Collection`> have one-to-many associations with other entities, then those entities will also be deleted.

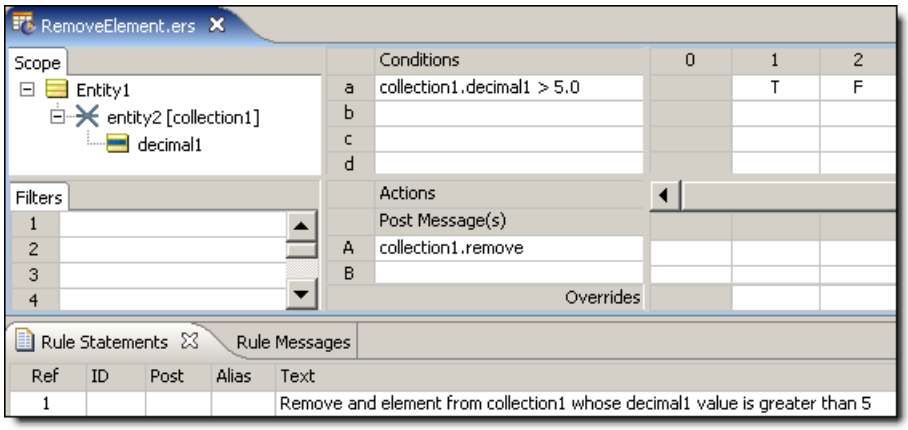
When using **.remove** to delete elements of a collection, associated entities will not be removed. To remove a chain of associated entities, use **.remove** on each level of the association hierarchy, beginning with the last or "lowest" level and working up. For example of this behavior, see example 2 below.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction does not apply. Special exceptions: **.remove** may only be used in Action Rows (section 5 in Sections of Rulesheet that correlate with usage restrictions).

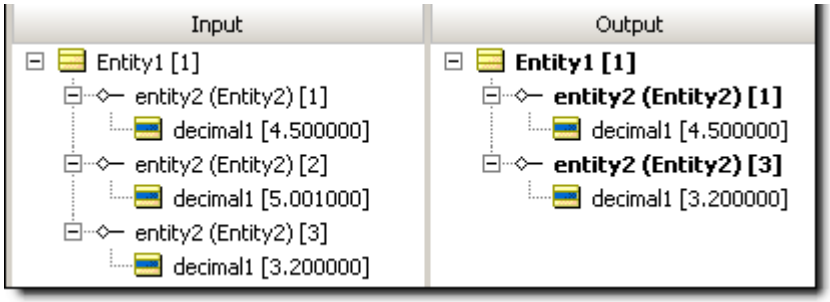
RULESHEET EXAMPLE

This Rulesheet uses the operator to remove elements from `collection1` whose `decimal1` value is greater than `5`. Note the *optional* use of unique alias `collection1` to represent the collection of `Entity2` elements associated with `Entity1`.



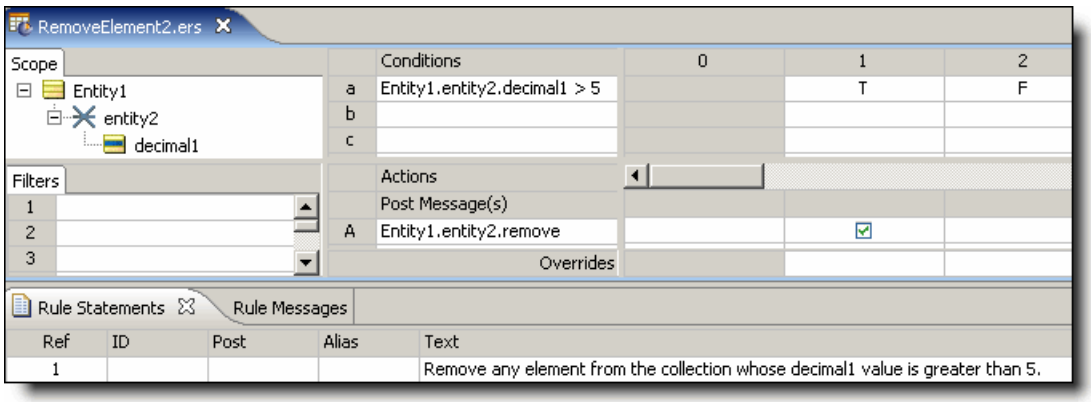
SAMPLE TEST

A sample Ruletest provides a collection with two elements. The illustration shows *`Ruletest`* Input and Output panels



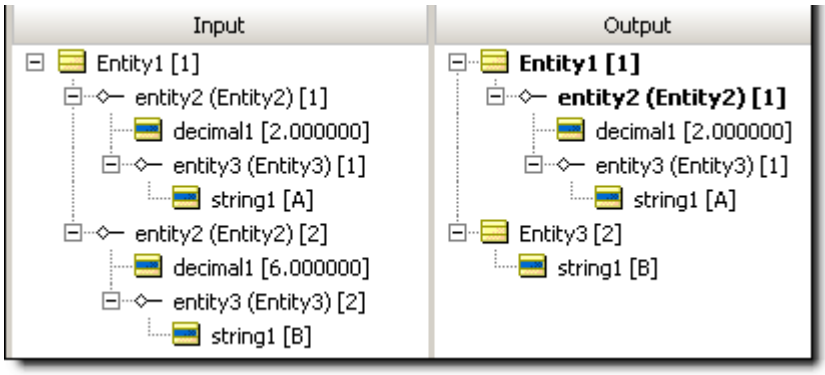
RULESHEET EXAMPLE 2

This *`Rulesheet`* uses the operator to remove elements from `Entity1.entity2` whose `decimal1` value is greater than `5`. Note no unique alias has been used to represent the collection of `Entity2` elements associated with `Entity1`.



SAMPLE RULETEST 2

A sample Ruletest provides an `Entity1` with two `entity2`, each of which has an `entity3` child of its own. The illustration shows Ruletest Input and Output panels. Note that when an `entity2` is removed, its associated `entity3` is moved to the root level.



Replace element(s)

SYNTAX

`<Collection1>` = `<Collection2>`

`<Collection>` = `<Entity>`

DESCRIPTION

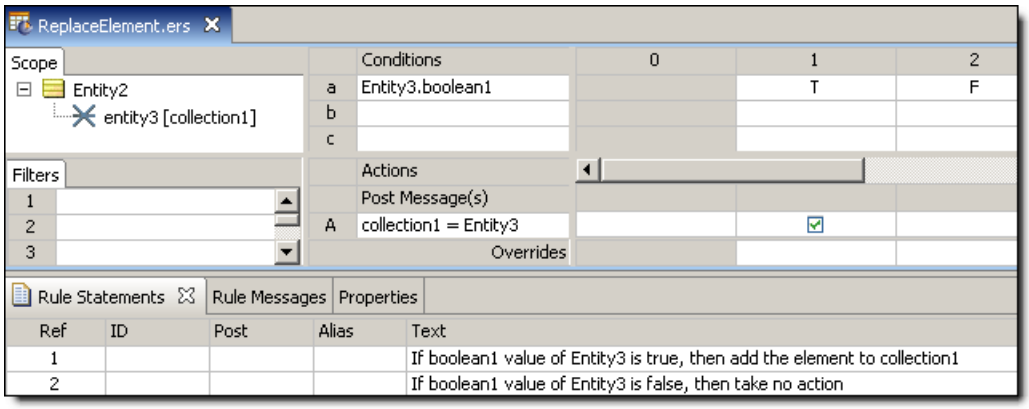
Replaces all elements in `<Collection1>` with the elements in `<Collection2>`, provided the association between the two is permitted by the Business Vocabulary. In the second syntax, `<Entity>` is associated with `<Collection>`, replacing the `<Entity>` already associated, when the association between the two is "one-to-one" in the Business Vocabulary. All collections must be expressed as unique aliases.

USAGE RESTRICTIONS

The Operators row in the table of Summary Table of Vocabulary Usage Restriction does not apply. Special exceptions: **replace elements** may only be used in Action Rows (section 5 in Sections of Rulesheet that correlate with usage restrictions).

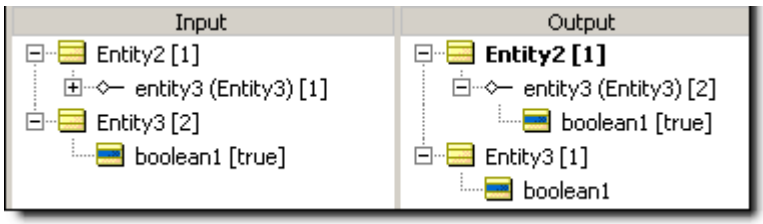
RULESHEET EXAMPLE

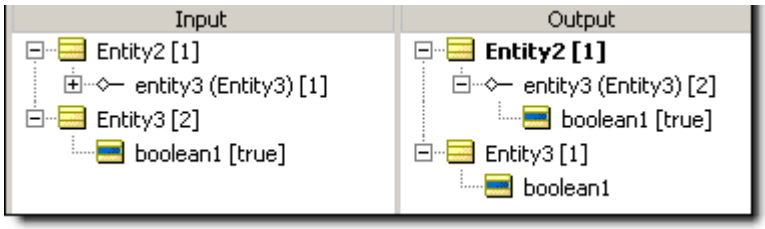
This sample Rulesheet uses the **replace element** operator to add `Entity3` to `collection1` if its `boolean1` value is `true`. Note the use of unique alias `collection1` to represent the collection of `Entity3` elements associated with `Entity2`. Recall from the generic Business Vocabulary, shown in Vocabulary used in this Language Guide on page 16, that the association between `Entity2` and `Entity3` has a cardinality of "one-to-one". This means that if multiple `Entity3` are present, only one will be added to `collection1`.



SAMPLE TEST

Four sample tests provide scenarios of two elements which share a one-to-one association. Input and Output panels are illustrated below:





Round

SYNTAX

`<Decimal>`.round(`<Integer>`)

DESCRIPTION

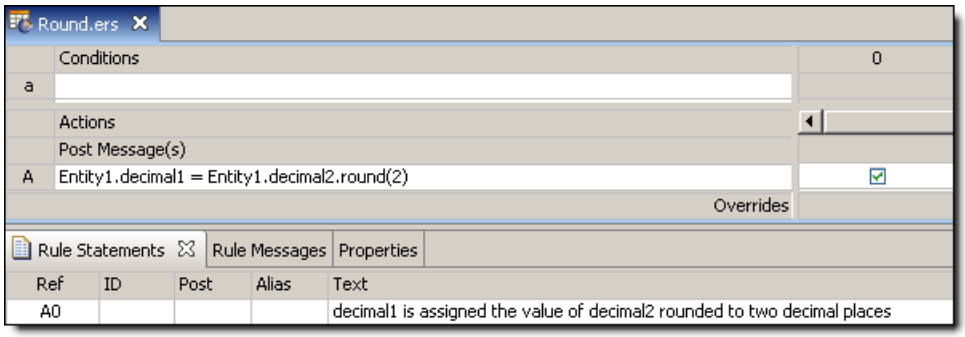
Rounds `<Decimal>` to the number of decimal places specified by `<Integer>`. Standard rounding conventions apply, meaning numbers ending with significant digits of 5 or more round up and numbers ending with significant digits less than 5 round down. `<Integer>` is optional – if no parameter is specified, then `<Decimal>` rounds to the nearest whole number of type Decimal.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

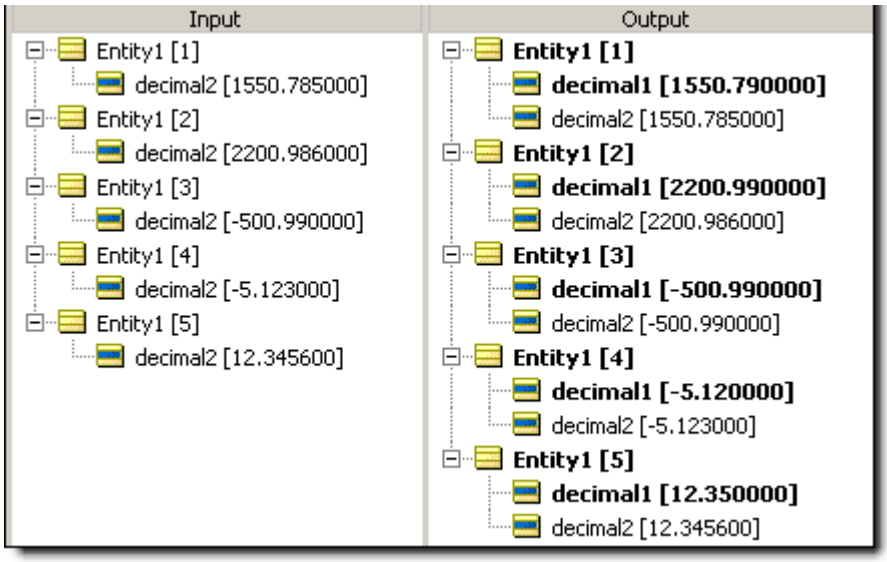
RULESHEET EXAMPLE

The following Rulesheet uses **.round** to round the value of `decimal2` to the 2nd decimal place, and assigns it to `decimal1`.



SAMPLE TEST

A sample Ruletest provides results for five examples of `decimal2`.



Second

SYNTAX

`<DateTime>`.sec

`<Time>`.sec

DESCRIPTION

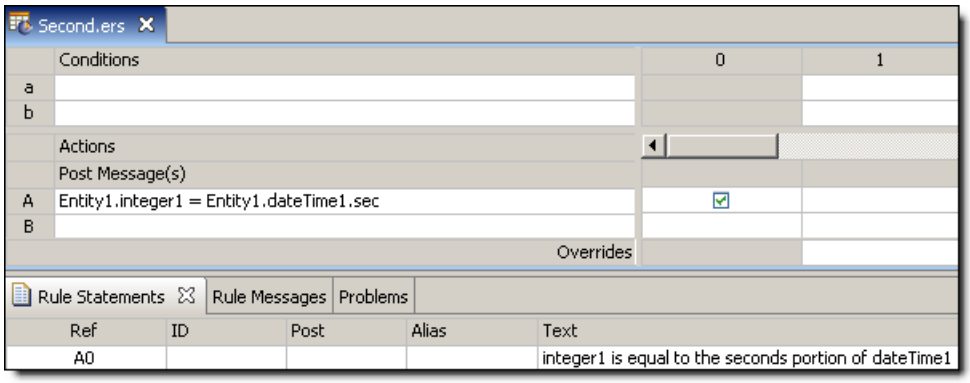
Returns the seconds portion of `<DateTime>` or `<Time>`. The returned value is an Integer between 0 and 59.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

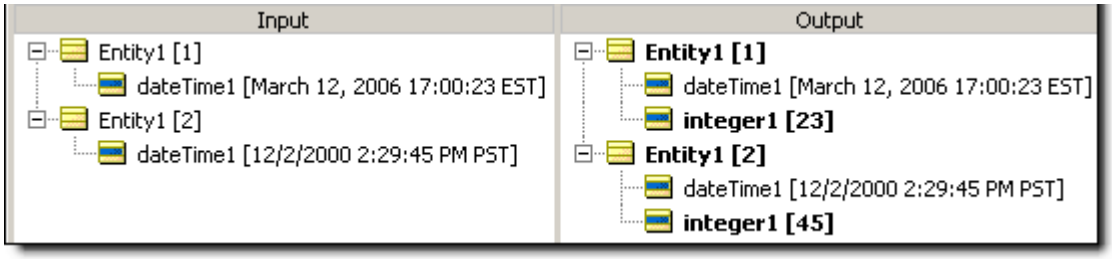
RULESHEET EXAMPLE

The following Rulesheet uses the **.sec** function to evaluate `dateTime1`, return the seconds value, and assign it to `integer1`.



SAMPLE TEST

A sample Ruletest provides results for two examples of `dateTime1`.



Seconds between

SYNTAX

`<DateTime1>`.secsBetween(`<DateTime2>`)

`<Time1>`.secsBetween(`<Time>`)

DESCRIPTION

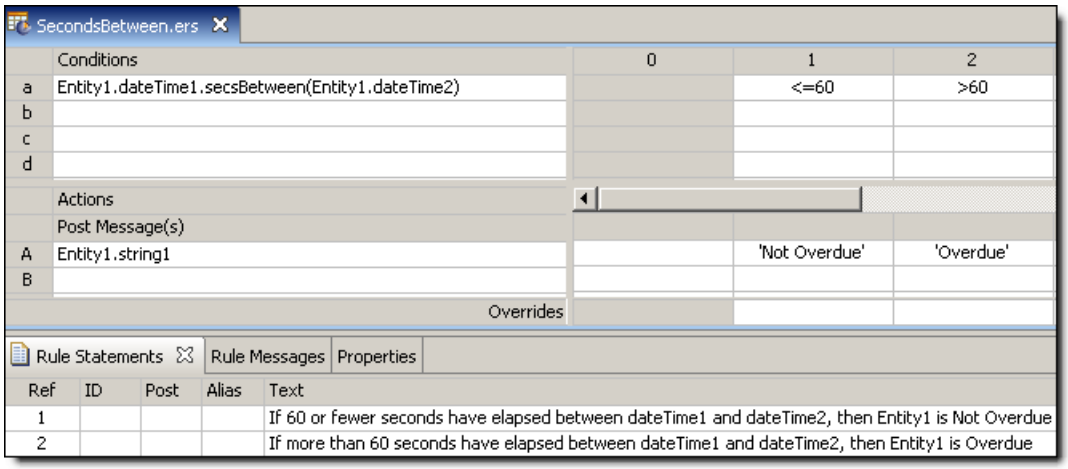
Returns the Integer number of seconds between DateTimes or between Times. The number of milliseconds in `<DateTime1>` is subtracted from that in `<DateTime2>`, and the result divided by 1000 (the number of milliseconds in a second). The result is truncated. This function returns a positive number if `<DateTime2>` is later than `<DateTime1>`.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

RULESHEET EXAMPLE

The following Rulesheet uses **.secsBetween** to determine the number of seconds that have elapsed between `dateTime1` and `dateTime2`, compare it to the Values set, and assign a value to `string1`.



SecondsBetween.ers

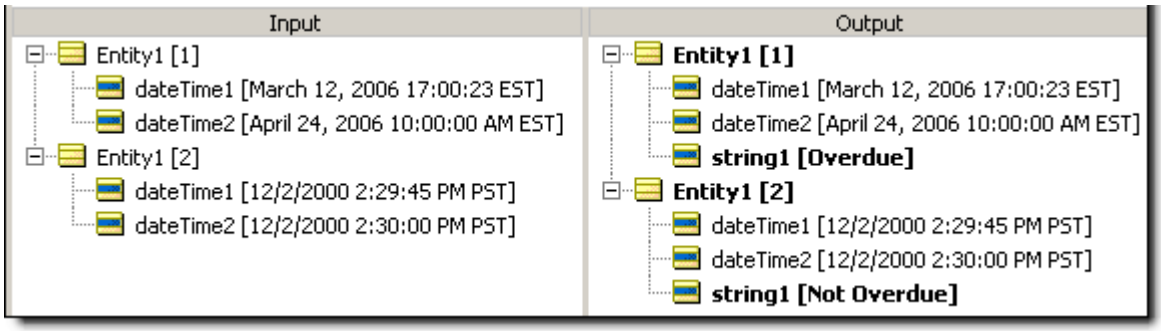
	Conditions	0	1	2
a	Entity1.dateTime1.secsBetween(Entity1.dateTime2)		<=60	>60
b				
c				
d				
	Actions			
	Post Message(s)			
A	Entity1.string1		'Not Overdue'	'Overdue'
B				
	Overrides			

Rule Statements | Rule Messages | Properties

Ref	ID	Post	Alias	Text
1				If 60 or fewer seconds have elapsed between dateTime1 and dateTime2, then Entity1 is Not Overdue
2				If more than 60 seconds have elapsed between dateTime1 and dateTime2, then Entity1 is Overdue

SAMPLE TEST

A sample Ruletest provides `dateTime1` and `dateTime2` for two examples. Input and Output panels are shown below.



Size of string

SYNTAX

`<String>`.size

DESCRIPTION

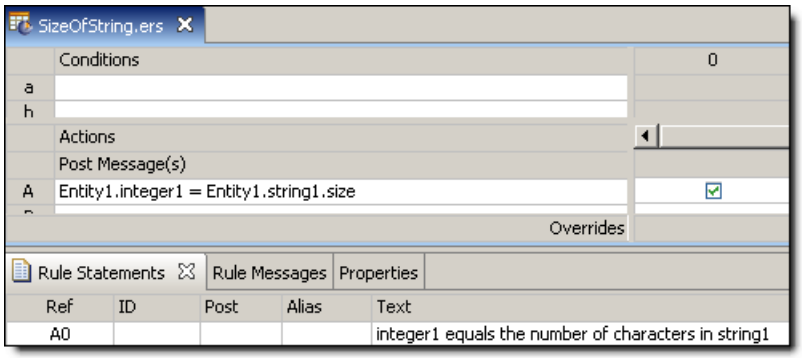
Returns the Integer number of characters in `<String>`. All characters, numbers, symbols, and punctuation marks are counted, including spaces before, within, and after words.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

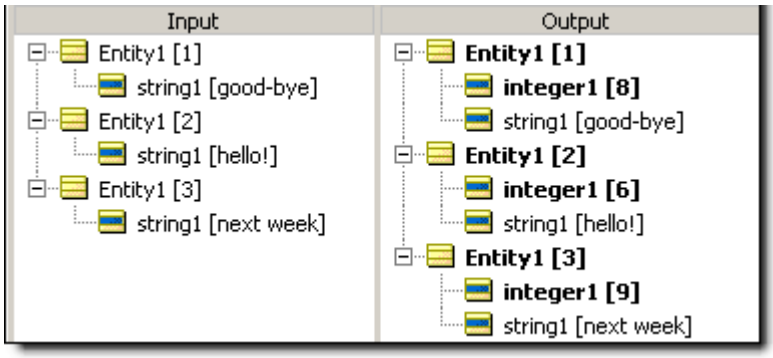
RULESHEET EXAMPLE

The following Rulesheet uses the **.size** function to determine the length of `string1` and assign it to `integer1`



SAMPLE TEST

A sample Ruletest provides three examples. Input and Output panels are shown below:



Size of collection

SYNTAX

`<Collection>` ->size

DESCRIPTION

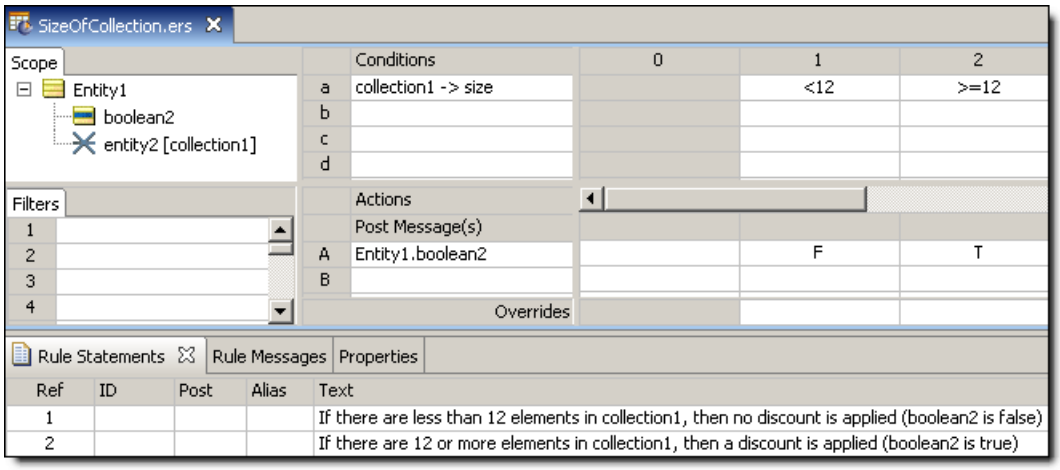
Returns the Integer number of elements in `<Collection>`. `<Collection>` must be expressed as a unique alias.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

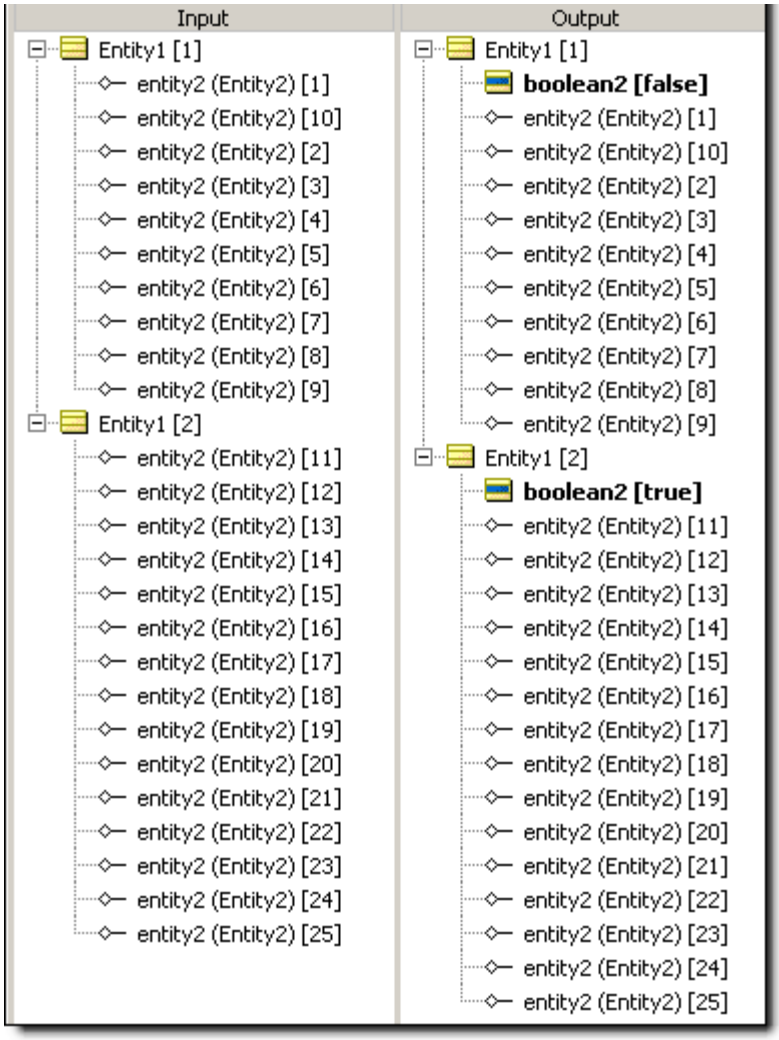
RULESHEET EXAMPLE

This sample Rulesheet uses **->size** to count the number of elements in `collection1`, and assign a value to `boolean2`. Note the use of unique alias `collection1` to represent the collection of `Entity2` associated with `Entity1`.



SAMPLE TEST

A sample Ruletest provides three examples of `collection1`. Input and Output panels are shown below.



Sorted by

SYNTAX

`<Collection>` ->sortedBy(`<Attribute2>`) -> *sequence operator* . `<Attribute1>`

DESCRIPTION

Sequences the elements of `<Collection>` in ascending order, using the value of `<Attribute2>` as the index, and returns the `<Attribute1>` value of the element in the sequence position determined by the sequence operator. A sequence must be created before any sequence operator (first, last, or at) is used to identify a particular element. `<Attribute1>` and `<Attribute2>` must be attributes of `<Collection>`.

`<Attribute2>` may be any data type except Boolean. Strings are sorted according to character precedence – see Character precedence: Unicode & Java Collator on page 189. `<Collection>` must be expressed as a unique alias.

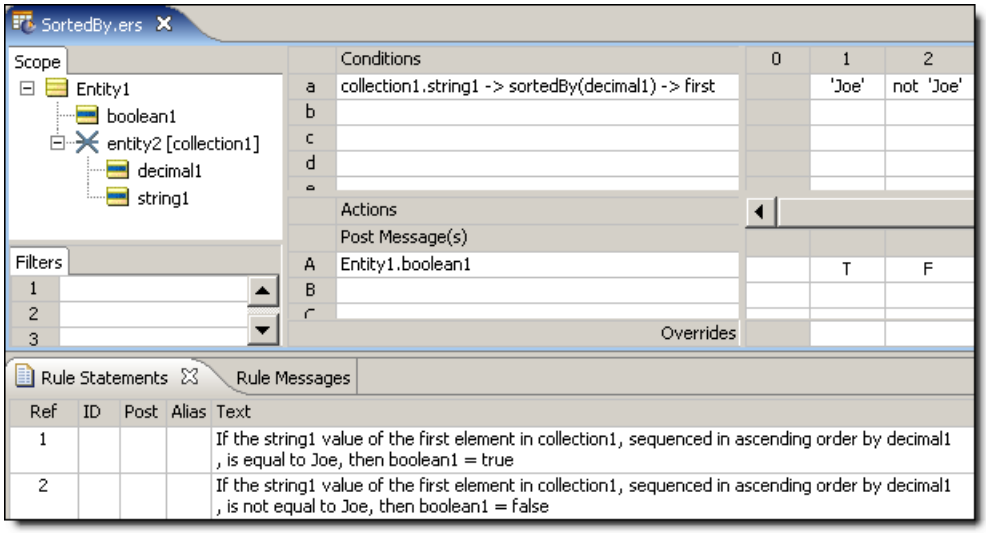
See Advanced Collection Syntax and special statement block syntax for more examples of usage.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

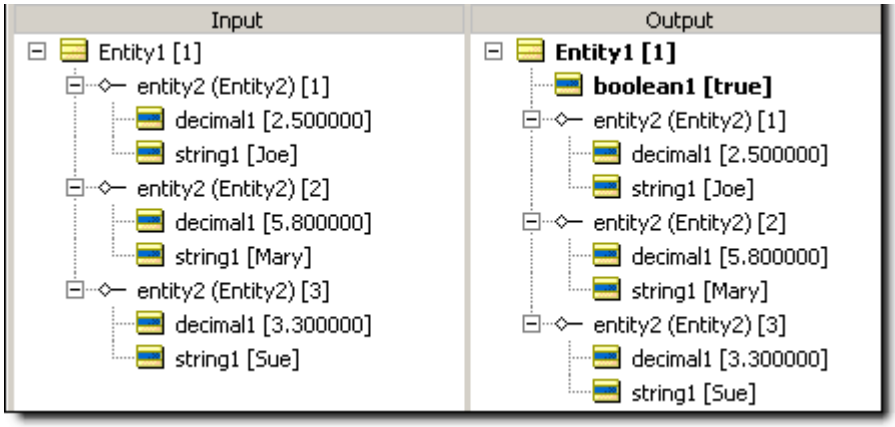
RULESHEET EXAMPLE 1 - USED IN A CONDITION

This sample Rulesheet uses **->sortedBy** in a conditional expression to create an ascending sequence from `collection` with `decimal1` as the index. `first.string1` is used to return the value of the `string1` attribute of the first element of the sequence. If the value of `string1` is `Joe`, then `boolean1` attribute of `Entity1` is assigned the value of `true`.



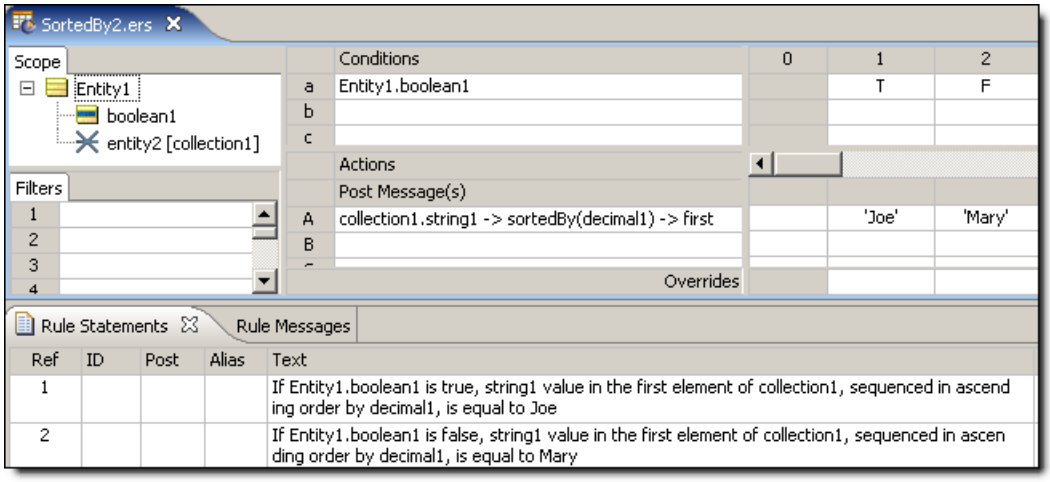
SAMPLE RULETEST 1

A sample Ruletest provides a collection of three elements, each with a `decimal1` and `string1` value. Input and Output panels are shown below.



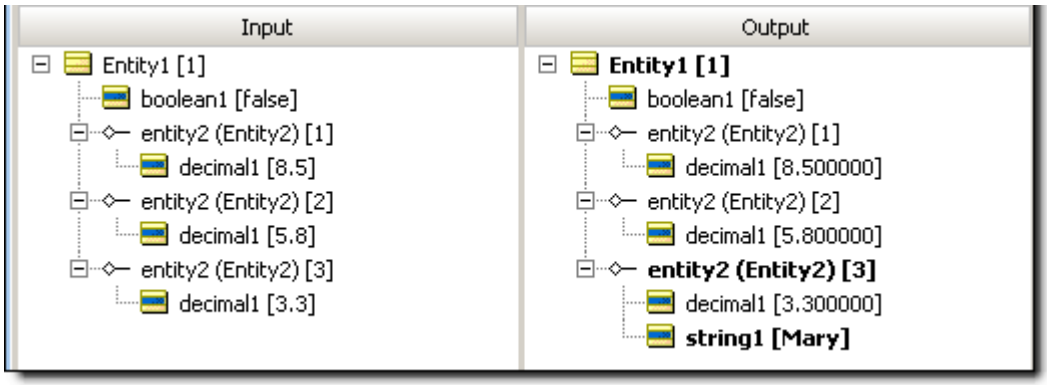
RULESHEET EXAMPLE 2 – USED IN AN ACTION

This sample Rulesheet uses **sortedBy** in an action expression to create an ascending sequence from `collection` with `decimal1` as the index. `first.string1` is used to return the value of the `string1` attribute of the first element of the sequence. The value of `string1` is assigned the value of `Joe` if `boolean1` attribute of `Entity1` is `true`, if `false` it is assigned the value of `Mary`.



SAMPLE RULETEST 2

A sample Ruletest provides a collection of three elements, each with a `decimal1` and `string1` value. Input and Output panels are shown below.



Sorted by descending

SYNTAX

`<Collection>` ->sortedByDesc(`<Attribute2>`) -> *sequence operator*.`<Attribute1>`

DESCRIPTION

Sequences the elements of `<Collection>` in descending order, using the value of `<Attribute2>` as the index, and returns the `<Attribute1>` value of the element in the sequence position determined by the sequence operator. A sequence must be created before any sequence operator (first, last, or at) is used to identify a particular element. `<Attribute1>` and `<Attribute2>` must be attributes of `<Collection>`.

`<Attribute2>` may be any data type except Boolean. Strings are sorted according to their ISO character precedence – see Character precedence: Unicode & Java Collator on page 189. `<Collection>` must be expressed as a unique alias.

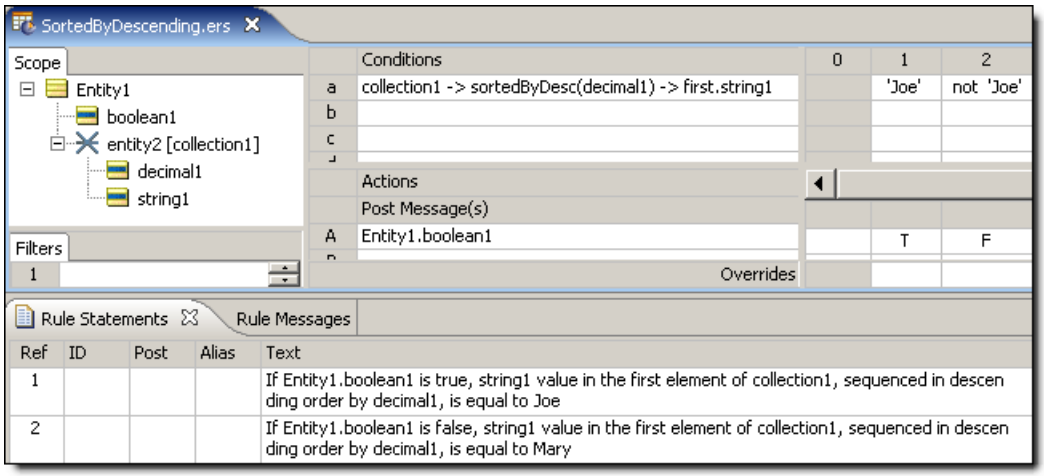
See Advanced Collection Syntax and special statement block syntax for more examples of usage.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

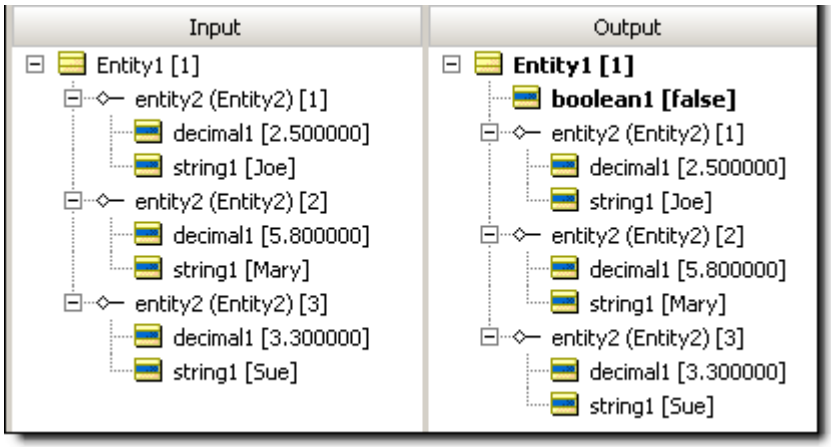
RULESHEET EXAMPLE 1 - USED IN A CONDITION

This sample Rulesheet uses **-> sortedByDesc** in a conditional expression to create an descending sequence from `collection` with `decimal1` as the index. `first.string1` is used to return the value of the `string1` attribute of the first element of the sequence. If the value of `string1` is `Joe`, then `boolean1` attribute of `Entity1` is assigned the value of `true`.



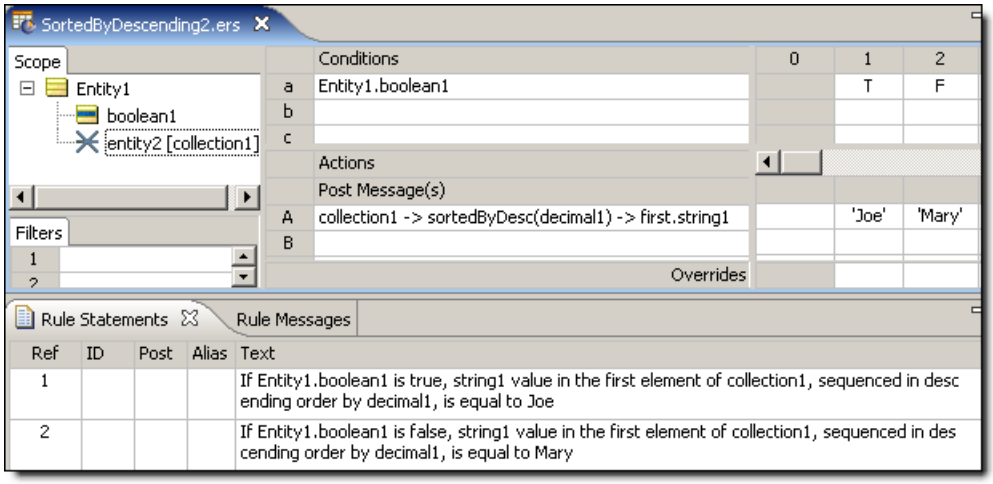
SAMPLE RULETEST 1

A sample Ruletest provides a collection of three elements, each with a `decimal1` value. Input and Output panels are shown below.



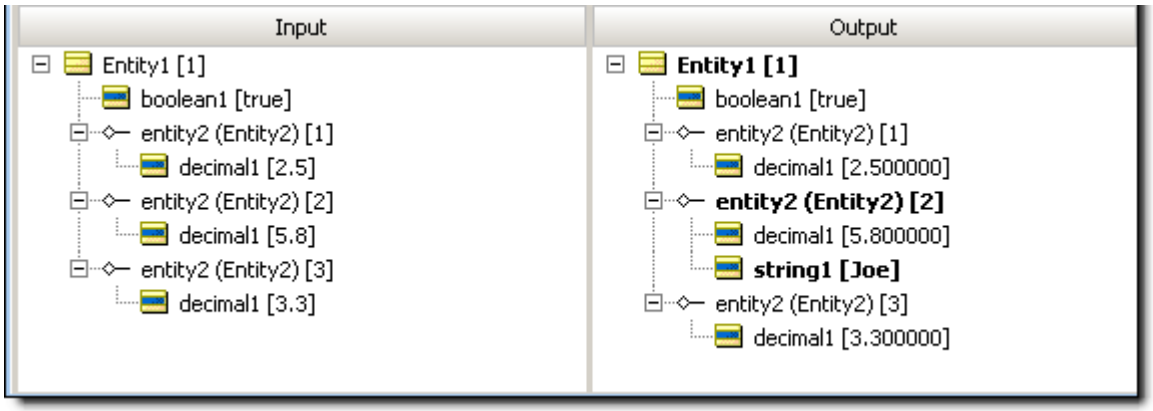
RULESHEET EXAMPLE 2 – USED IN AN ACTION

This sample Rulesheet uses **sortedByDesc** in an action expression to create an descending sequence from `collection` with `decimal1` as the index. `first.string1` is used to return the value of the `string1` attribute of the first element of the sequence. The value of `string1` is assigned the value of `Joe` if `boolean1` attribute of `Entity1` is `true`, if `false` it is assigned the value of `Mary`.



SAMPLE RULETEST 2

A sample Ruletest provides a collection of three elements, each with a `decimal1` value. Input and Output panels are shown below.



Starts with

SYNTAX

`<String1>`.startsWith(`<String2>`)

DESCRIPTION

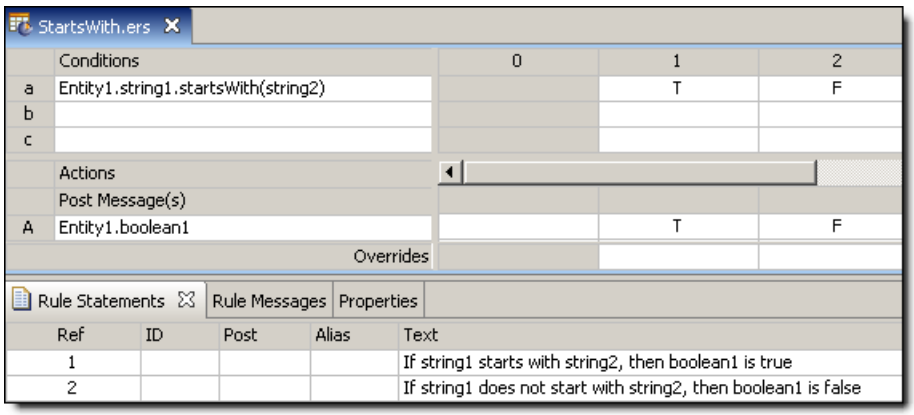
Returns a value of true if `<String1>` begins with the characters specified in `<String2>`. Comparisons are case-sensitive.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

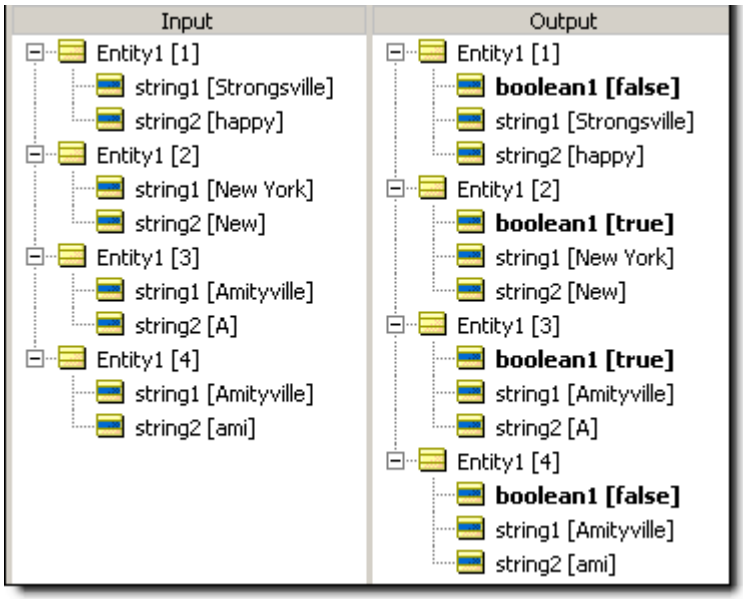
RULESHEET EXAMPLE

The following Rulesheet uses **.startsWith** to evaluate whether `string1` begins with the value of `string2` and assigns a different value to `boolean1` for each outcome.



SAMPLE TEST

A sample Ruletest provides `string1` and `string2` values for four examples. Input and Output panels are shown below.



Substring

SYNTAX

`<String>`.substring(`<Integer1>`, `<Integer2>`)

DESCRIPTION

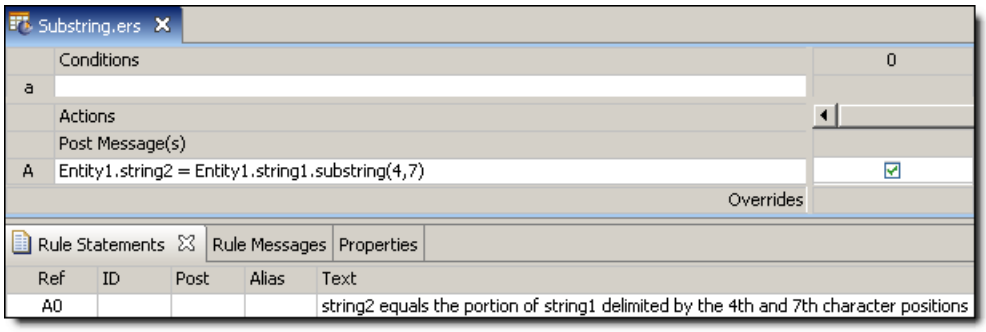
Returns the portion of `<String>` beginning with the character in position `<Integer1>` and ending with the character in position `<Integer2>`. The number of characters in `<String>` must be at least equal to `<Integer2>`, otherwise an error will be produced. Both `<Integer1>` and `<Integer2>` must be positive integers, and `<Integer2>` must be greater than `<Integer1>`.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

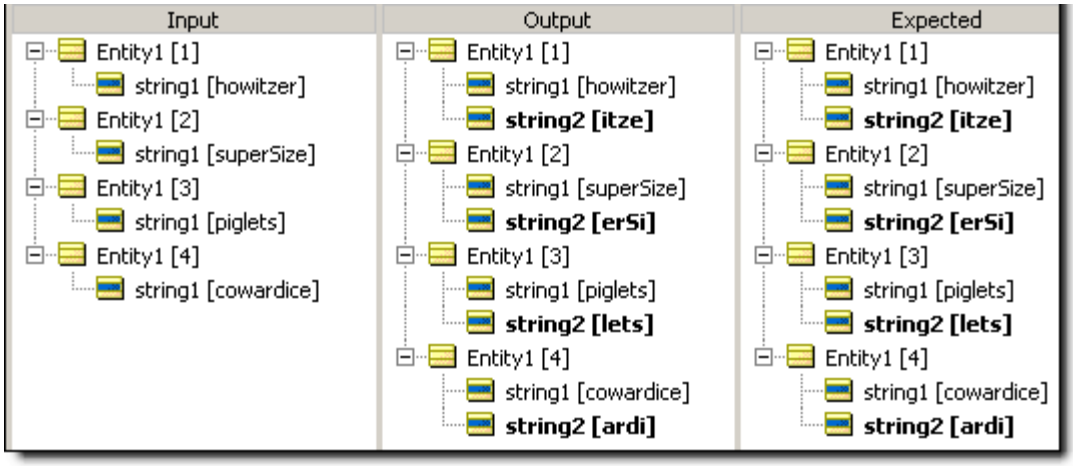
RULESHEET EXAMPLE

This sample Rulesheet uses **.substring** to return those characters of `string1` between positions `4` and `7` (inclusive), and assign the resulting value to `string2`.



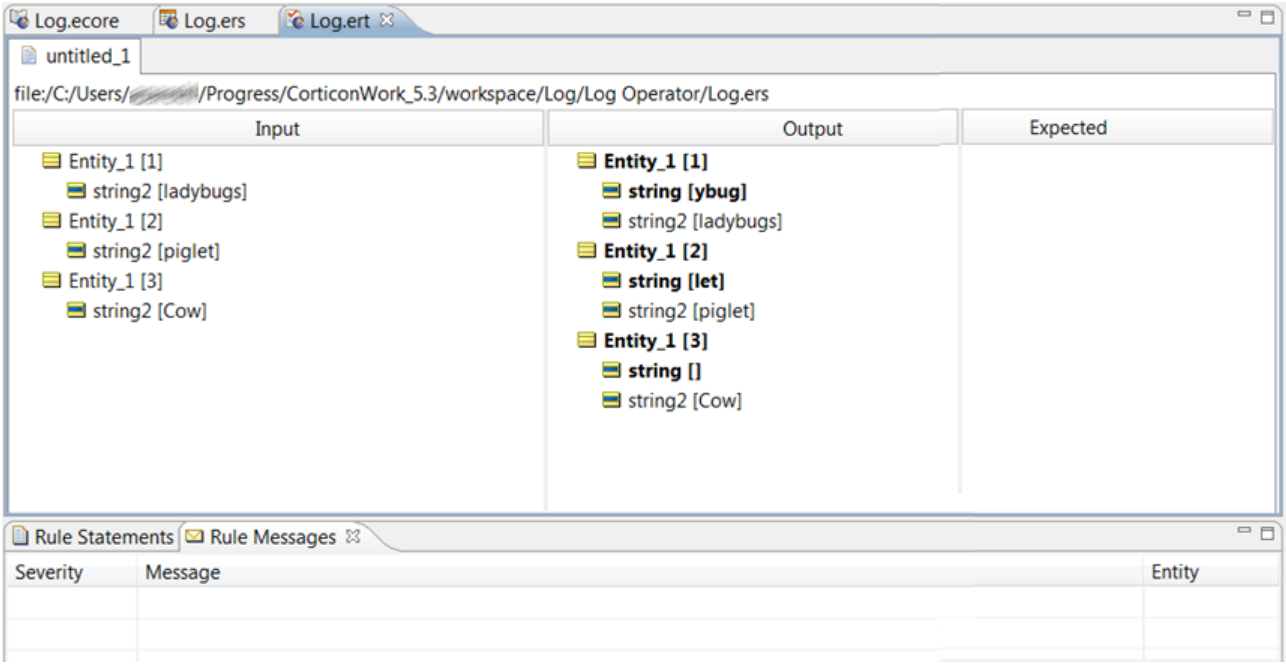
SAMPLE RULETEST 1

A sample Ruletest provides `string1` values for four examples. Input and Output panels are shown below.



SAMPLE RULETEST 2

Another sample Ruletest shows `string1` values with two examples having insufficient character strings. The resulting error message is illustrated below:



Subtract

SYNTAX

```
<Number1> - <Number2>
```

DESCRIPTION

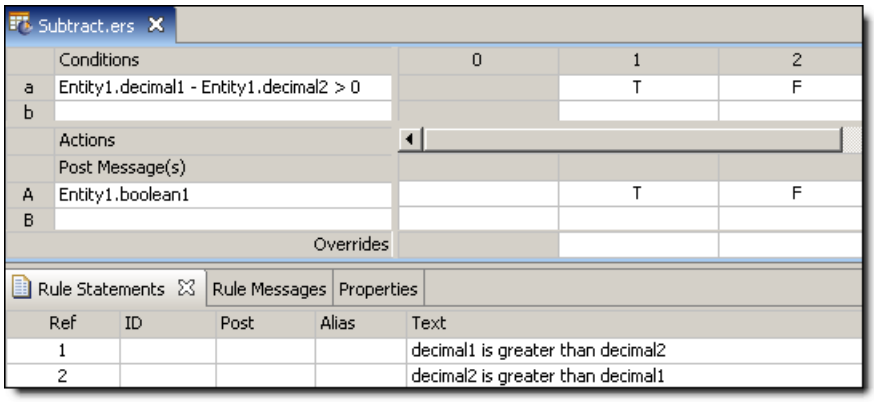
Subtracts the value of `<Number2>` from that of `<Number1>`. The resulting data type is the more expansive of those of `<Number1>` and `<Number2>`.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

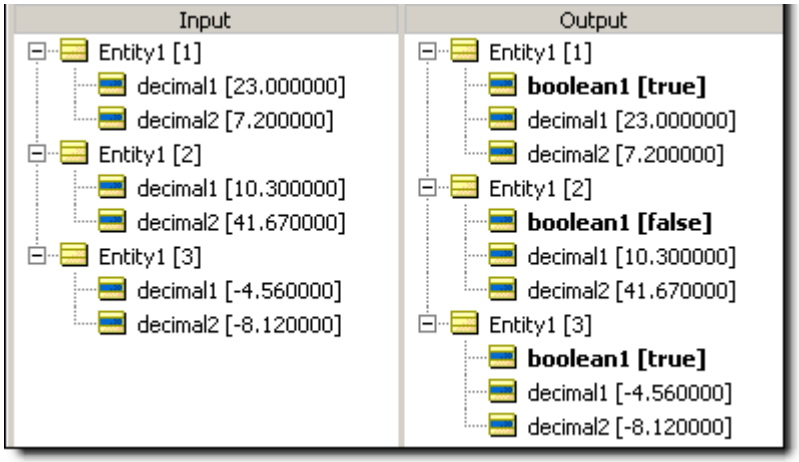
RULESHEET EXAMPLE

This sample Rulesheet uses **subtract** to reduce the value of `decimal1` by `decimal2`, compare the resulting value to zero, and assign a value to `boolean1`



SAMPLE TEST

A Ruletest provides three examples of `decimal1` and `decimal2`. Input and Output panels are shown below.



Sum

SYNTAX

`<Collection.attribute>` ->sum

DESCRIPTION

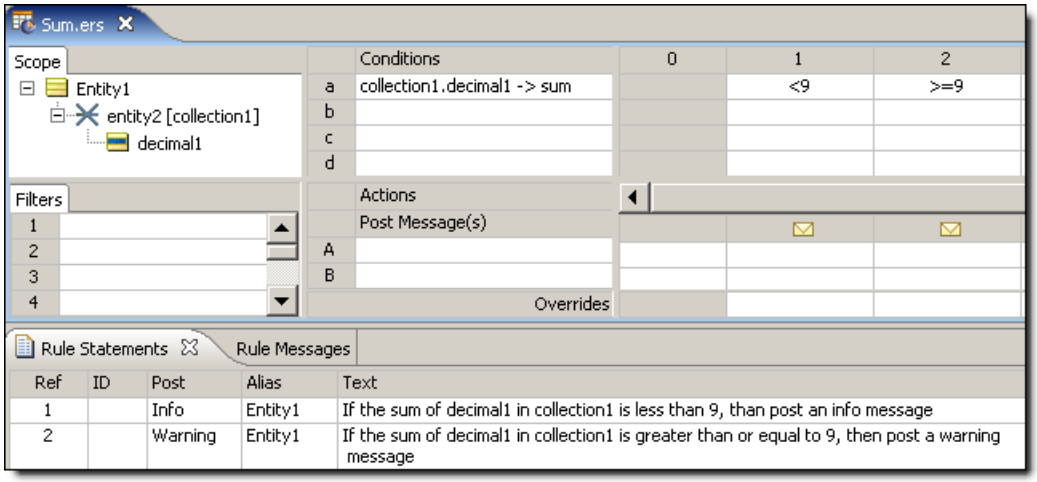
Sums the values of the specified `<attribute>` for all elements in `<Collection>`. `<attribute>` must be a numeric data type. `<Collection>` must be expressed as a unique alias.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

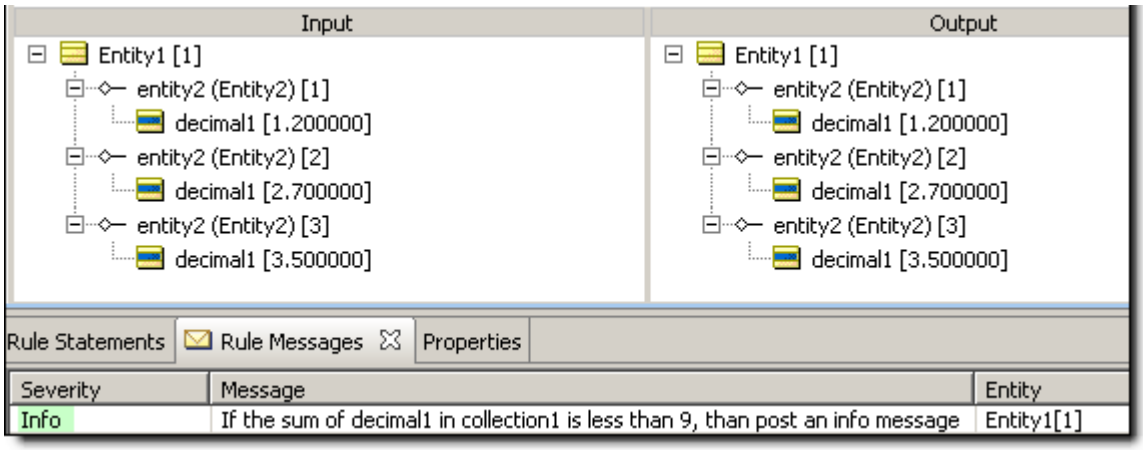
RULESHEET EXAMPLE

This Rulesheet uses the **sum** function to add all `decimal1` attributes within `collection1`. Note the use of unique alias `collection1` to represent the collection of `Entity2` associated with `Entity1`



SAMPLE TEST

A sample Ruletest provides 3 elements in `collection1`. Input and Output panels are shown below.



Today

SYNTAX

today

DESCRIPTION

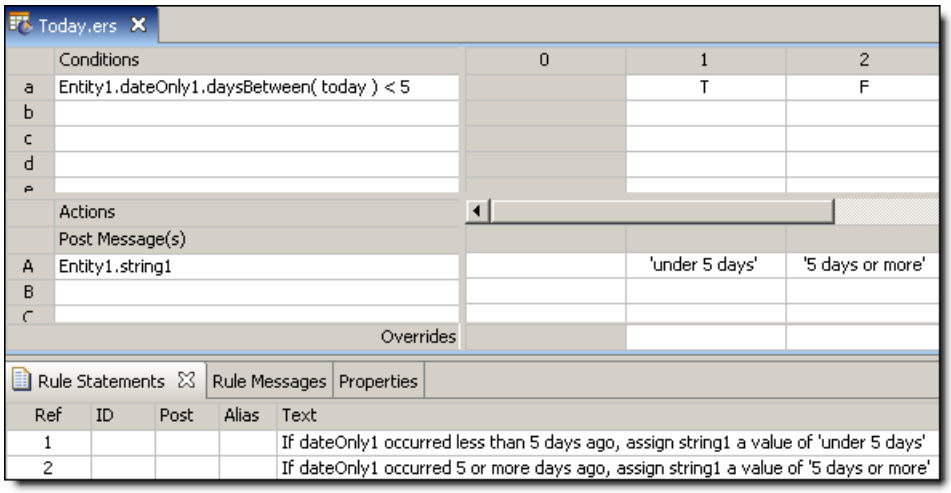
Returns the current system date when the rule is executed. This Date Only value is assigned the first time **today** is used in a Decision Service (Rule Set), then remains constant until the Decision Service finishes execution, regardless of how many additional times it is used. This means that every rule in a Rule Set using **today** will use the same Date Only value. No time portion is assigned

USAGE RESTRICTIONS

The Literals row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

RULESHEET EXAMPLE

The following Rulesheet uses **today** to determine how many days have elapsed between today and `dateTime1`, and assign a value to `string1` based on the result.



Today.ers

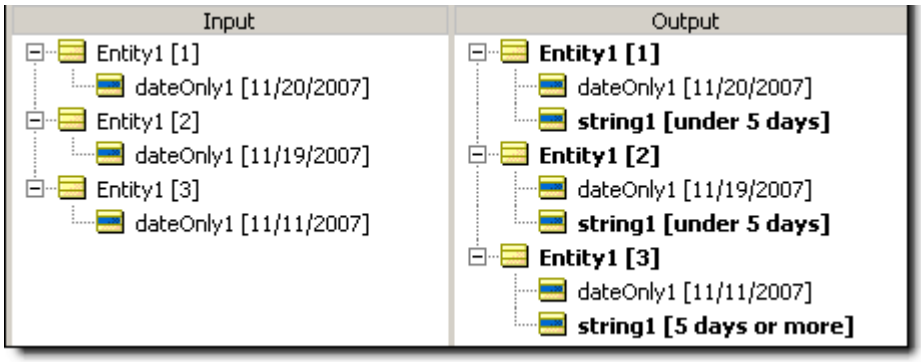
	Conditions	0	1	2
a	Entity1.dateOnly1.daysBetween(today) < 5		T	F
b				
c				
d				
e				
	Actions			
	Post Message(s)			
A	Entity1.string1		'under 5 days'	'5 days or more'
B				
C				
	Overrides			

Rule Statements | Rule Messages | Properties

Ref	ID	Post	Alias	Text
1				If dateOnly1 occurred less than 5 days ago, assign string1 a value of 'under 5 days'
2				If dateOnly1 occurred 5 or more days ago, assign string1 a value of '5 days or more'

SAMPLE TEST

A sample *`Ruletest`* provides three examples of `dateOnly1`. Assume **today** is equal to `Friday,November 23,2007`. Input and Output panels are shown below:



To date – casting a dateTime to a date

SYNTAX

`<DateTime>`.toDate

DESCRIPTION

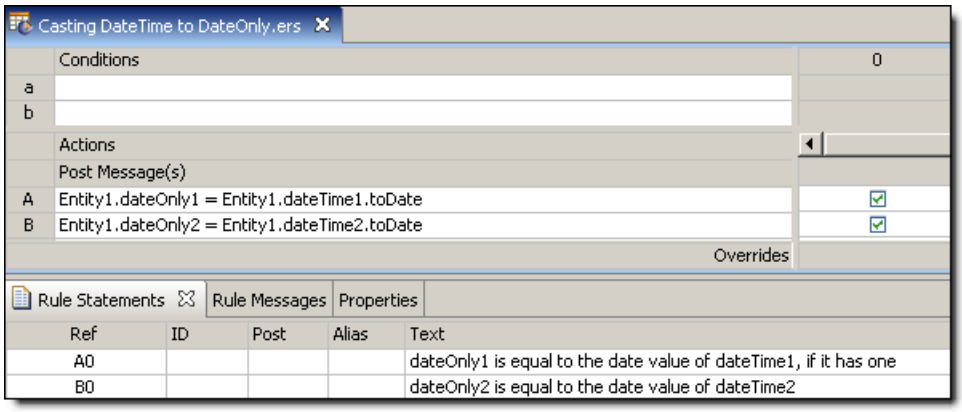
Converts the value in `<DateTime>` to a Date datatype, containing only the date portion of the DateTime. If `<DateTime>` contains no date information, then the system epoch is used.

USAGE RESTRICTIONS

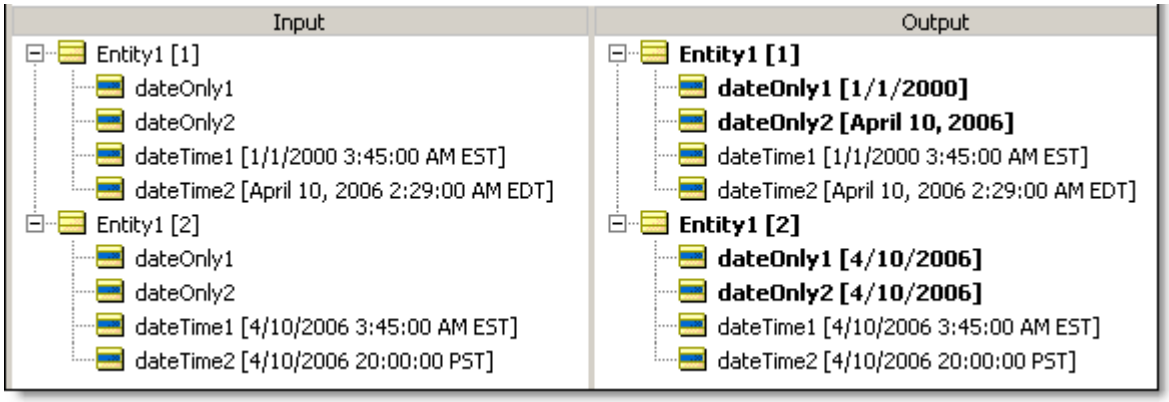
The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

RULESHEET EXAMPLE

The following Rulesheet uses **.toDate** to convert `dateTime1` and `DateTime2` to Date datatypes and assign the values to `dateTime1` and `dateTime2`.



SAMPLE TEST



To dateTime – casting a string to a dateTime

SYNTAX

`<String>`.toDateTime

DESCRIPTION

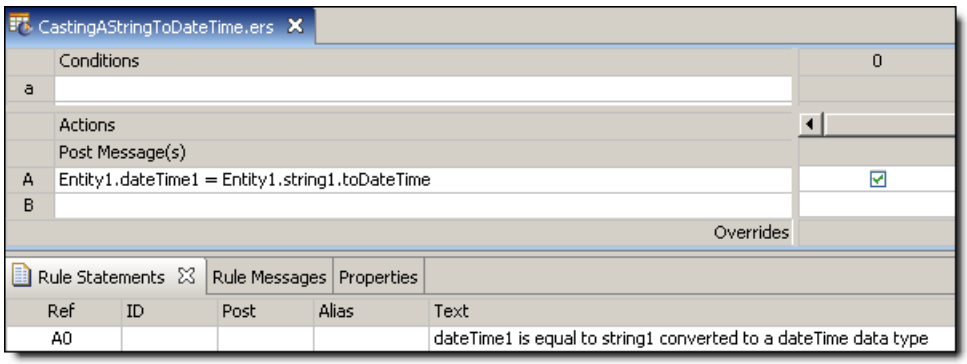
Converts the value in `<String>` to data type DateTime ONLY if all characters in `<String>` correspond to a valid Date, Time, or DateTime mask (format). For complete details on DateTime masks, see *Rule Modeling Guide*.

USAGE RESTRICTIONS

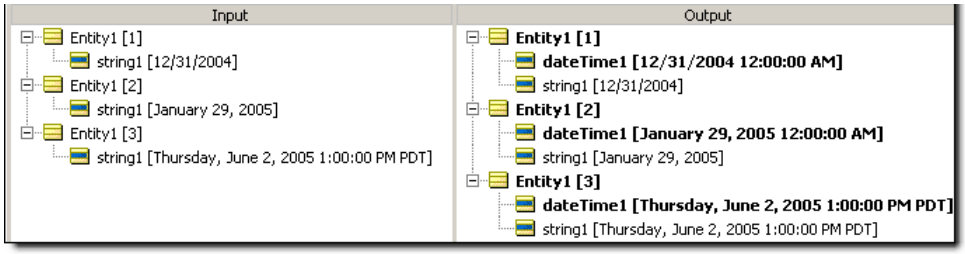
The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

RULESHEET EXAMPLE

The following Rulesheet uses **.toDateTime** to convert `string1` to type DateTime and assign the value to `dateTime1`.



SAMPLE TEST



To dateTime – casting a date to a dateTime

SYNTAX

`<Date>`.toDateTime

DESCRIPTION

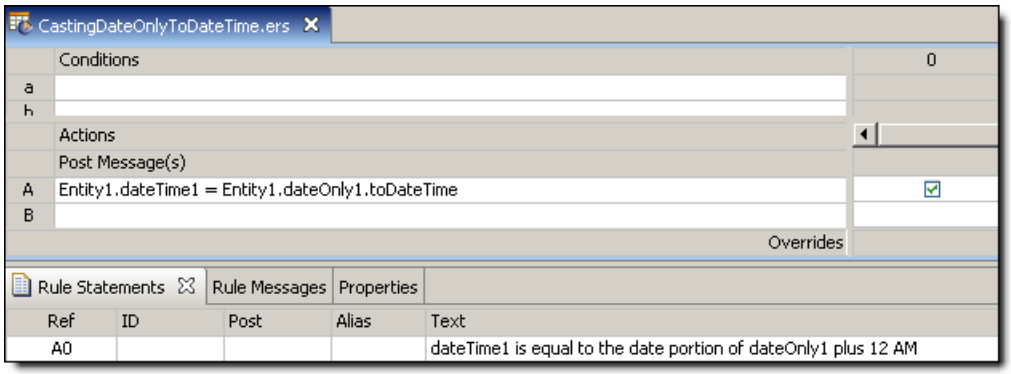
Converts the value in `<Date>` to data type DateTime. The date portion is the same as the `<Date>` value and the time portion is set to `12:00:00 AM` in the current timezone.

USAGE RESTRICTIONS

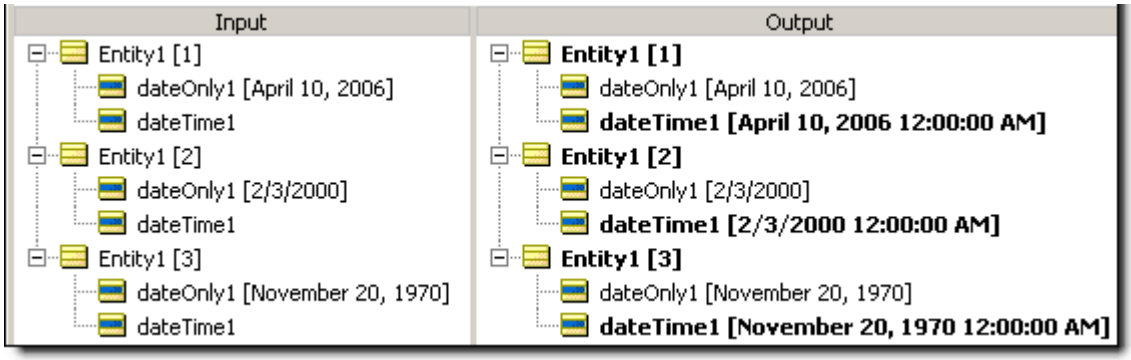
The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

RULESHEET EXAMPLE

The following Rulesheet uses **.toDateTime** to convert `dateOnly1` to type DateTime and assign the value to `dateTime1`.



SAMPLE TEST



To dateTime – casting a time to a dateTime

SYNTAX

`<Time>`.toDateTime

DESCRIPTION

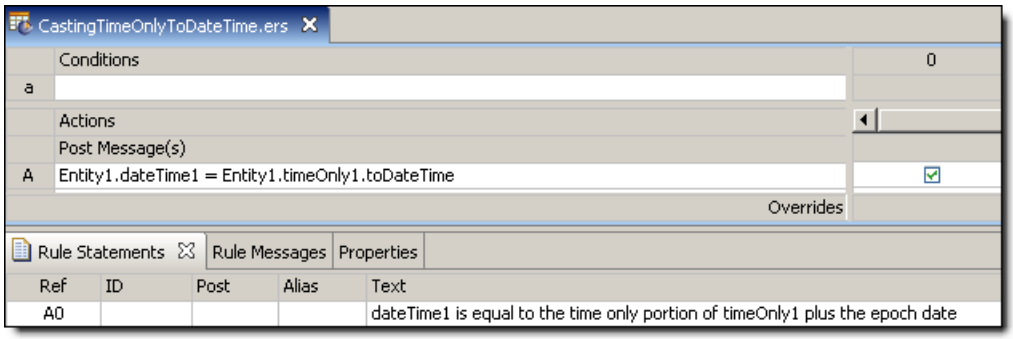
Converts the value in `<Time>` to data type DateTime ONLY if all characters in `<Time>` correspond to a valid DateTime mask (format). The time portion is the same as the `<Time>` value and the date portion is the epoch (see .toTime operator)

USAGE RESTRICTIONS

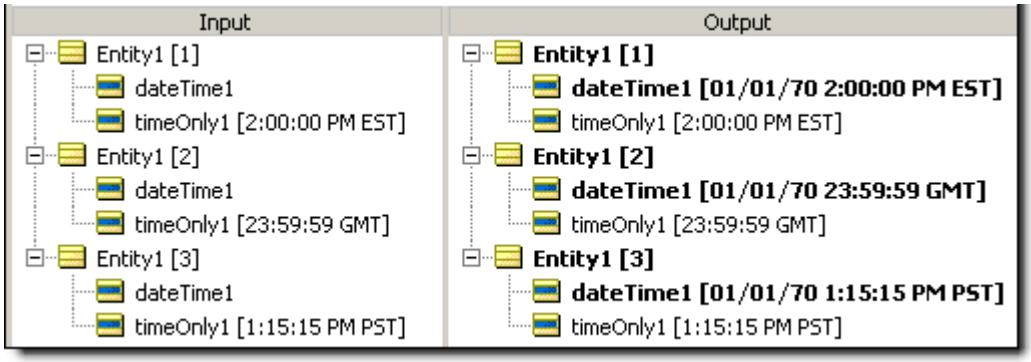
The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

RULESHEET EXAMPLE

The following Rulesheet uses **.toDateTime** to convert `timeOnly1` to type DateTime and assign the value to `dateTime1`.



SAMPLE TEST



To dateTime – timezone offset

SYNTAX

`<Date>`.toDateTime(`<String>`)

DESCRIPTION

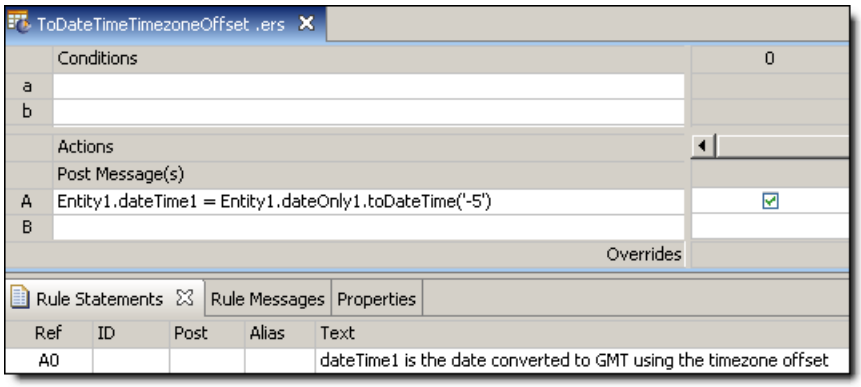
Converts the value in `<Date>` to data type DateTime ONLY if all characters in `<Date>` correspond to a valid DateTime mask (format). The date portion is the same as the `<Date>` value and the time portion is set to `00:00:00` in the timezone specified by `<String>`, which is the timeZoneOffset. The timeZoneOffset must take the form of a valid timezone offset such as `'-08:00'`, `'+03:30'`, `'01:45'`

USAGE RESTRICTIONS

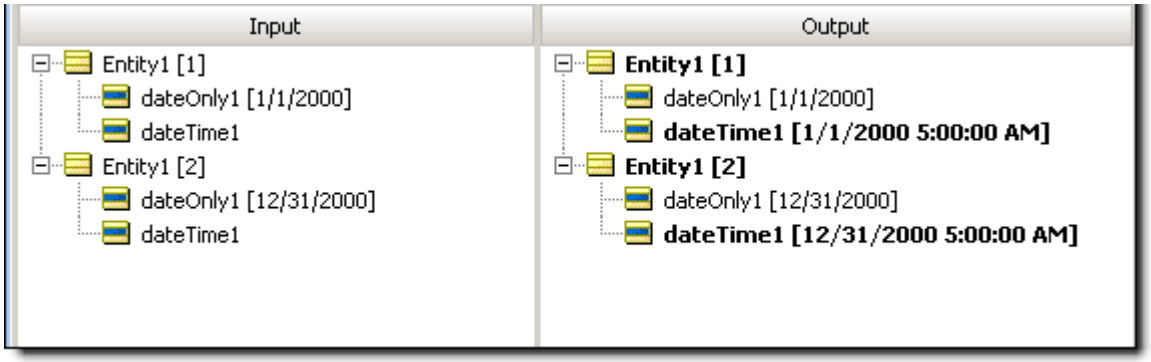
The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

RULESHEET EXAMPLE

The following Rulesheet uses **.toDateTime** to convert `dateOnly1` to type DateTime and assign the value to `dateTime1`, with a timezone offset of `-5`.



SAMPLE TEST



To decimal

SYNTAX

`<Integer>`.toDecimal

`<String>`.toDecimal

DESCRIPTION

Converts the value in `<Integer>` or all characters in `<String>` to data type Decimal. Converts a String to Decimal ONLY if all characters in `<String>` are numeric and contain not more than one decimal point. If any non-numeric characters are present in `<String>` (other than the single decimal point or a leading minus sign), no value is returned by the function.

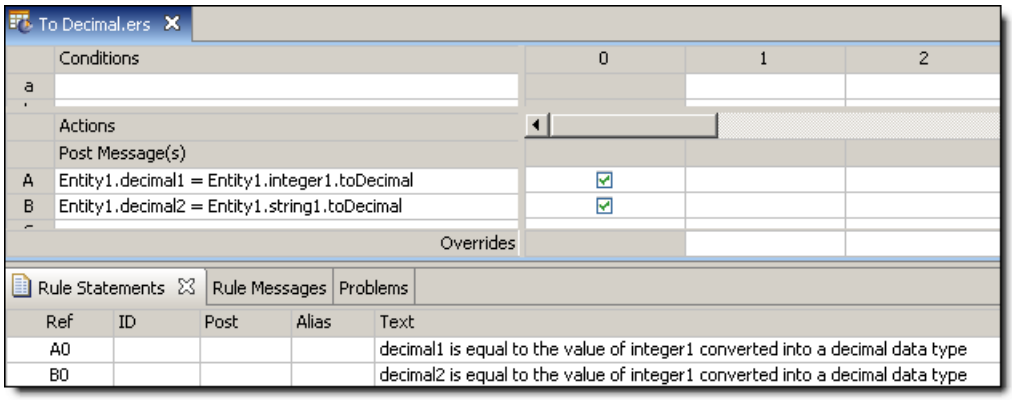
Note: Integer values may be assigned directly to Decimal data types without using the **.toDecimal** operator because a Decimal data type is more expansive than an Integer.

USAGE RESTRICTIONS

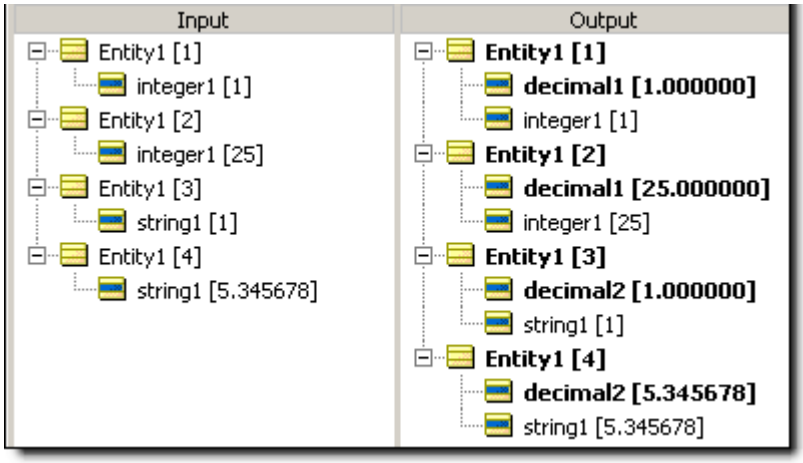
The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

RULESHEET EXAMPLE

The following Rulesheet uses **.toDecimal** to convert `integer1` and `string1` to type Decimal and assign them to `decimal1` and `decimal2`, respectively.



SAMPLE TEST



To integer

SYNTAX

`<Decimal>`.toInteger

`<String>`.tointeger

DESCRIPTION

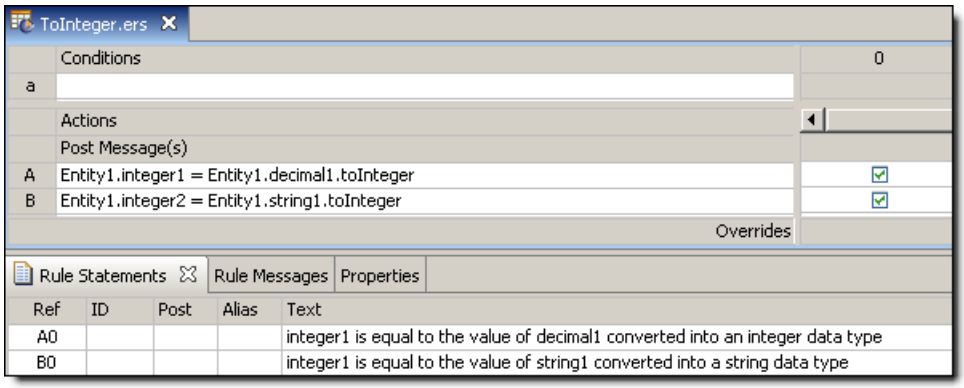
Converts the value in `<Decimal>` or all characters in `<String>` to data type Integer. All decimals have fractional portions truncated during the conversion. Strings are converted ONLY if all characters in `<String>` are numeric, without a decimal point. If any non-numeric characters (with the sole exception of a single leading minus sign for negative numbers) are present in `<String>`, no value is returned by the function. Do not use on String values of `null` or empty String (`''`) -- a pair of single quote marks -- as this will generate an error message.

USAGE RESTRICTIONS

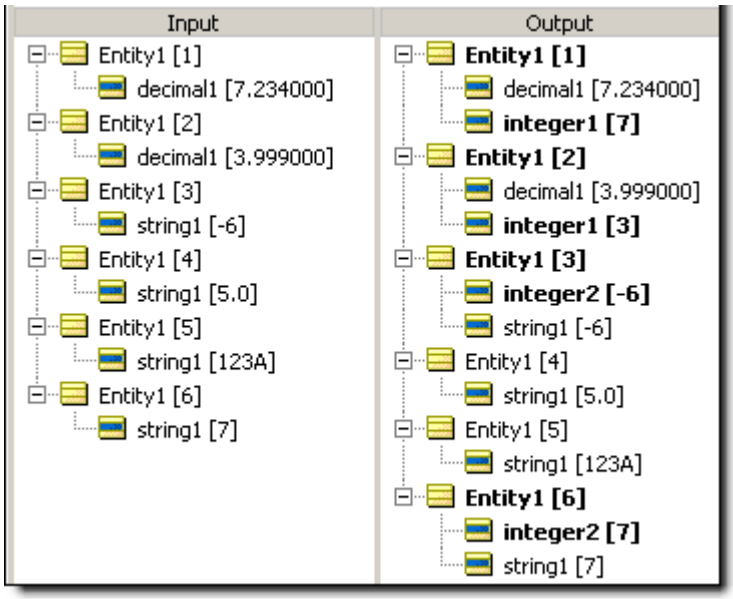
The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

RULESHEET EXAMPLE

The following Rulesheet uses **.toInteger** to convert `decimal1` and `string1` to type Integer and assign them to `integer1` and `integer2`, respectively.



SAMPLE TEST



To string

SYNTAX

`<Number>`.toString

`<DateTime*>`.toString

*includes DateTime, Date, and Time data types

DESCRIPTION

Converts a value to a data type of String.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

RULESHEET EXAMPLE

The following Rulesheet uses **.toString** to convert 3 data types to strings. Rule N.3 also uses the alternative String concatenation syntax. See Add Strings for details.

ToString.ers

	Conditions	0
a		
b		
	Actions	
	Post Message(s)	
A	Entity1.string1 = Entity1.decimal1.toString	☑
B	Entity1.string2 = Entity1.integer1.toString	☑
C	Entity2.string1 = Entity2.dateTime1.toString + 'AD'	☑
	Overrides	

Rule Statements | Rule Messages | Properties

Ref	ID	Post	Alias	Text
A0				Entity1.string1 is equal to the value of decimal1 converted into a string data type
B0				Entity1.string2 is equal to the value of integer1 converted into a string data type
C0				Entity2.string1 is equal to the value of dateTime1 converted into a string data type and appended with AD.

SAMPLE TEST

Input	Output
Entity1 [1]	**Entity1 [1]**
decimal1 [3.456700]	decimal1 [3.456700]
Entity1 [2]	**string1 [3.456700]**
integer1 [5]	**Entity1 [2]**
Entity2 [1]	integer1 [5]
dateTime1 [3/16/2006 2:00:00 PM EST]	**string2 [5]**
	Entity2 [1]
	dateTime1 [3/16/2006 2:00:00 PM EST]
	string1 [3/16/2006 2:00:00 PM ESTAD]

To time – casting a dateTime to a time

SYNTAX

`<DateTime>`.toTime

DESCRIPTION

Converts the value in `<DateTime>` to a Time data type, containing only the time portion of the full DateTime. If `<DateTime>` contains no time information, then the time portion is set to `12:00:00 AM` in the current timezone.

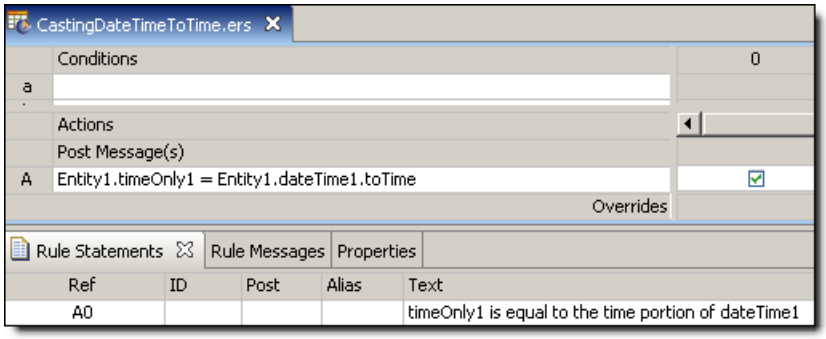
Note: Should this value be saved in a database field supporting a full DateTime, the date portion is saved as the epoch specified in `com.corticon.crml.OclDate.epochForTimeValues`. See *Server Integration & Deployment Guide* for a complete description of this and other properties inside `CcCommon.properties`.

USAGE RESTRICTIONS

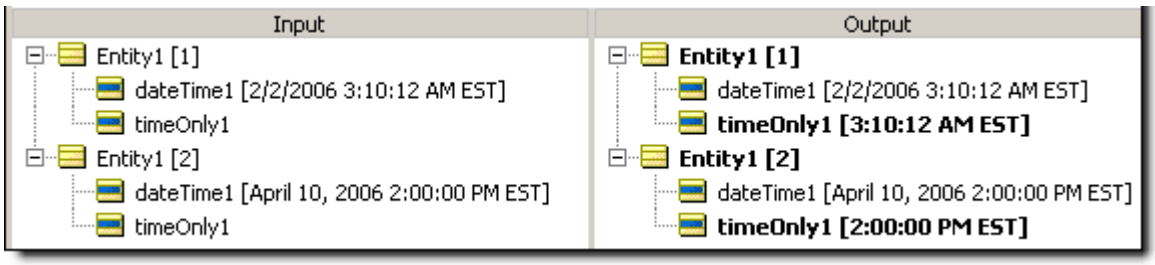
The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

RULESHEET EXAMPLE

The following Rulesheet uses **.toTime** to convert `dateTime1` to Time and assign the value to `TimeOnly1`.



SAMPLE TEST



Trend

SYNTAX

`<Collection.attribute>` -> `<Sequence>`.trend

DESCRIPTION

Returns one of the following 4-character strings depending on the trend of `<Collection.attribute>` once sequenced by the same or different attribute in `<Collection>`. `<Sequence>` is an ordered set of `<Collection>` in the form $\{x_1, x_2, x_3 \ldots x_n\}$, where

INCR	the value of <attribute> of element x_{n+1} is greater than or equal to the value of <attribute> of element x_n for every element. At least one <attribute> value of element x must be greater than that of x_{n-1}
DECR	the value of <attribute> of element x_{n+1} is less than or equal to the value of <attribute> of element x_n for every element. At least one <attribute> value of element x must be less than that of x_{n-1}
CNST	the value of <attribute> of element x_{n+1} is equal to the value of <attribute> for element x_n for every element.
NONE	any <sequence> with elements not meeting the requirements for INCR, DECR, or CNST

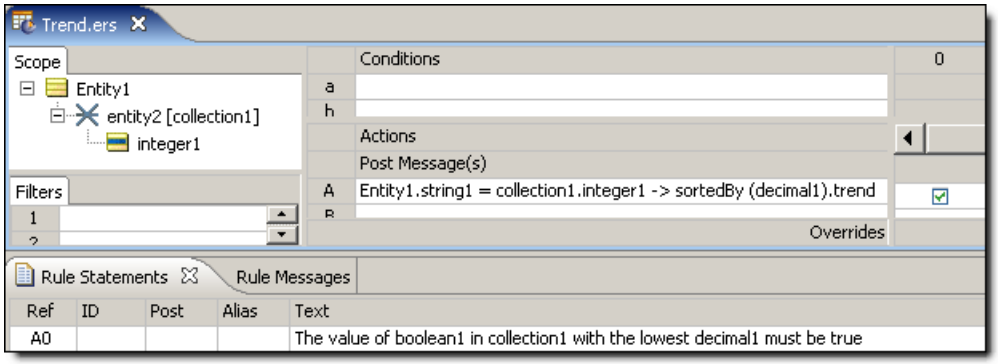
An alternative way to understand this operator is to view the index attribute used to sequence the collection as the *independent* variable (traditionally plotted along the "x" axis in a standard x-y graph) in a set of data pairs. The attribute evaluated by the **.trend** operator, <Collection.attribute>, is the *dependent* variable, plotted along the "y" axis. When so plotted, the 4-character words returned by **.trend** correspond to curves with positive, negative, zero (constant), or arbitrary slopes.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

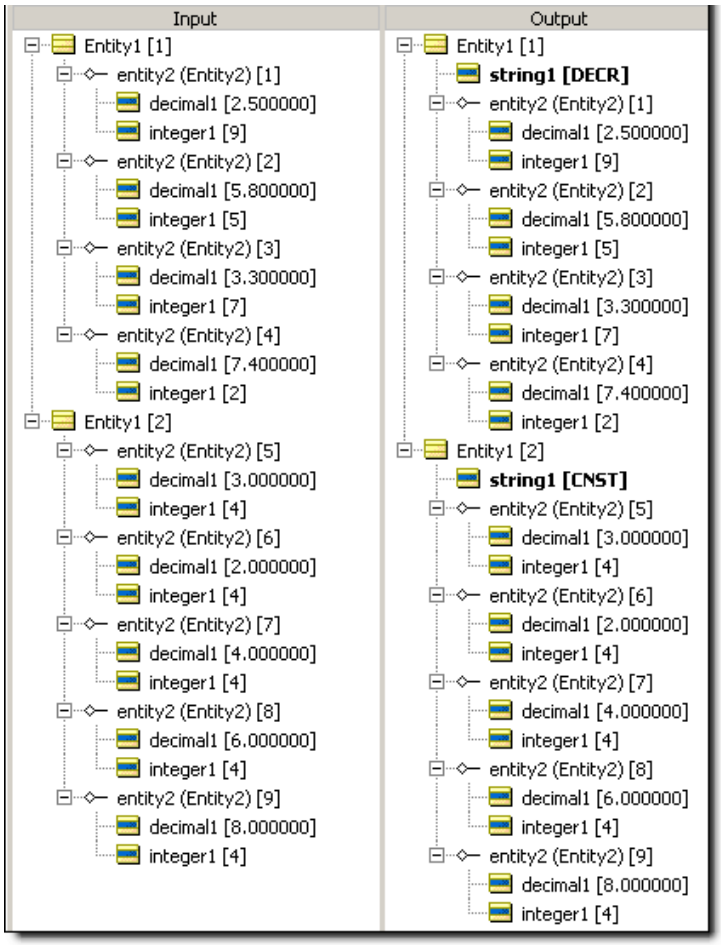
RULESHEET EXAMPLE

This Rulesheet uses the **.trend** function to analyze integer1 attributes within collection1 sorted by decimal1. The resulting trend value is assigned to string1.



SAMPLE TEST

Two sample tests provide two collections of elements, each with a decimal1 and integer1 values. Input and Output panels are shown below.



Note: Technically, the slope of an `INCR` curve need not be positive everywhere, but must have a first derivative (instantaneous slope) that is positive at some point along the curve and never be negative. The slope of a `CNST` curve must be zero everywhere.

True

SYNTAX

true or T

DESCRIPTION

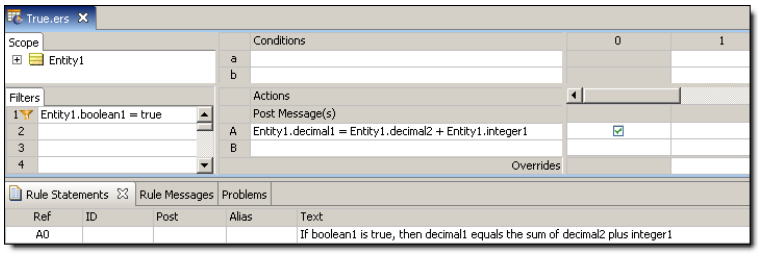
Represents Boolean value true. Recall from the discussion oftruth values that an `<expression>` is evaluated for its truth value, so the expression `Entity1.boolean1=true` will evaluate to `true` only if `boolean1=true`. But since `boolean1` is Boolean and has a truth value all by itself without any additional syntax, we do not actually need the "`=true`" piece of the expression. Many examples in the documentation use explicit syntax like `boolean1=true` or `boolean2=false` for clarity and consistency, even though `boolean1` or `not boolean2` are equivalent logical expressions.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

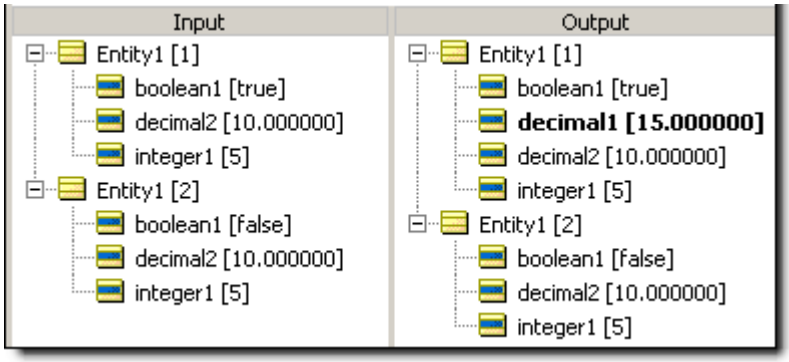
RULESHEET EXAMPLE

The following Rulesheet uses **true** in a Precondition to Ruletest whether `boolean1` is true, and perform the Nonconditional computation if it is. As discussed above, the alternative expression `Entity1.boolean1` is logically equivalent.



SAMPLE TEST

A sample Ruletest provides three examples. Assume `decimal2=10.0` and `integer1=5` for all examples. Input and Output panels are shown below:



Uppercase

SYNTAX

`<String>`.toUpper

DESCRIPTION

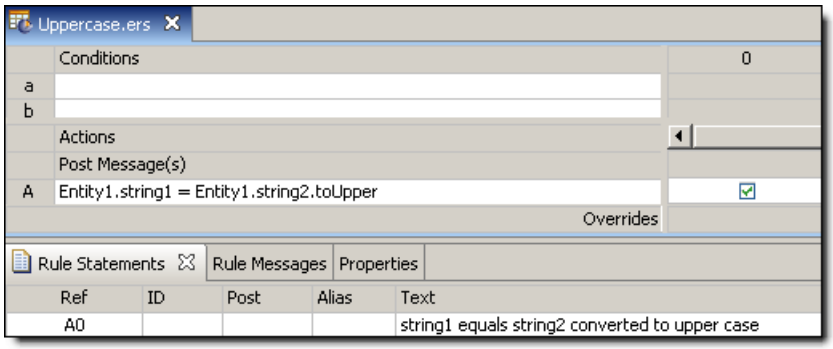
Converts all characters in `<String>` to uppercase.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

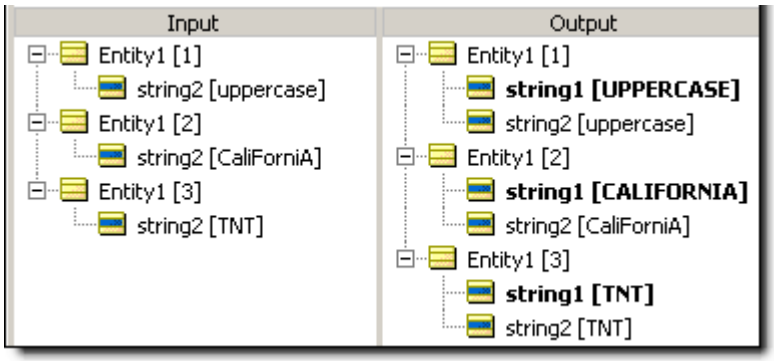
RULESHEET EXAMPLE

The following Rulesheet uses **.toUpper** to convert `string2` to uppercase and assign it to `string1`.



SAMPLE TEST

A sample Ruletest provides three examples. Input and Output panels are shown below:



Week of month

SYNTAX

`<DateTime>`.weekOfMonth

`<Date>`.weekOfMonth

DESCRIPTION

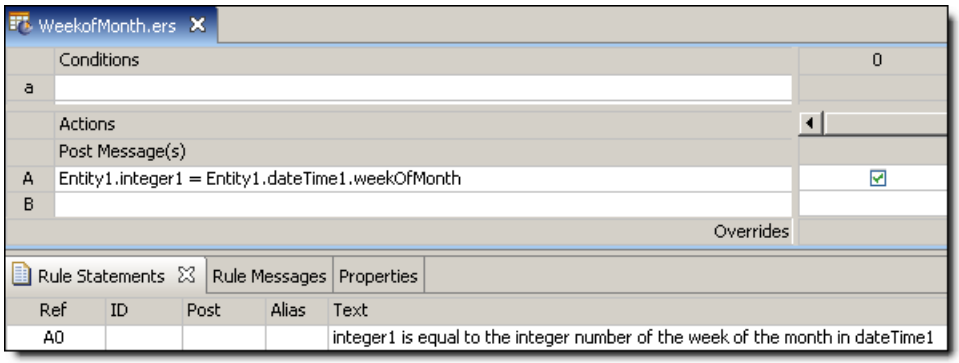
Returns an Integer from 1 to 6, equal to the week number within the month in `<DateTime>` or `<Date>`. A week begins on Sunday and ends on Saturday.

USAGE RESTRICTIONS

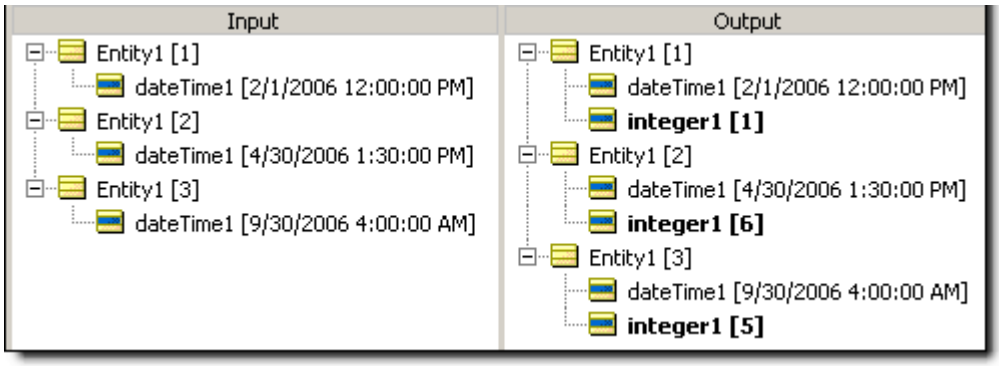
The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

RULESHEET EXAMPLE

The following Rulesheet uses **.weekOfMonth** to assign a value to `integer1`.



SAMPLE TEST



Week of year

SYNTAX

`<DateTime>`.weekOfYear

`<Date>`.weekOfYear

DESCRIPTION

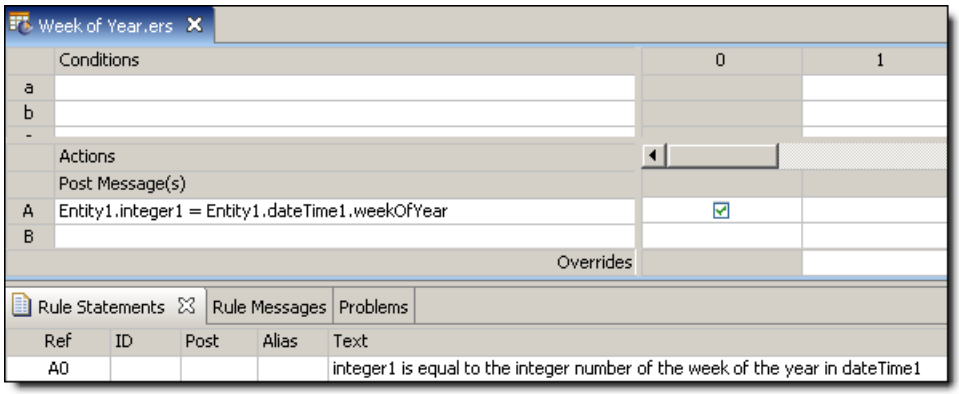
Returns an Integer from 1 to 52, equal to the week number within the year in `<DateTime>` or `<Date>`. A week begins on Sunday and ends on Saturday. When a year ends between Sunday and the next Friday, or in other words when a new year begins between Monday and the next Saturday, the final day(s) of December will be included in week 1 of the new year. For example, 12/29/2002 fell on a Sunday, so 12/29-31 are included in week 1 of 2003.

USAGE RESTRICTIONS

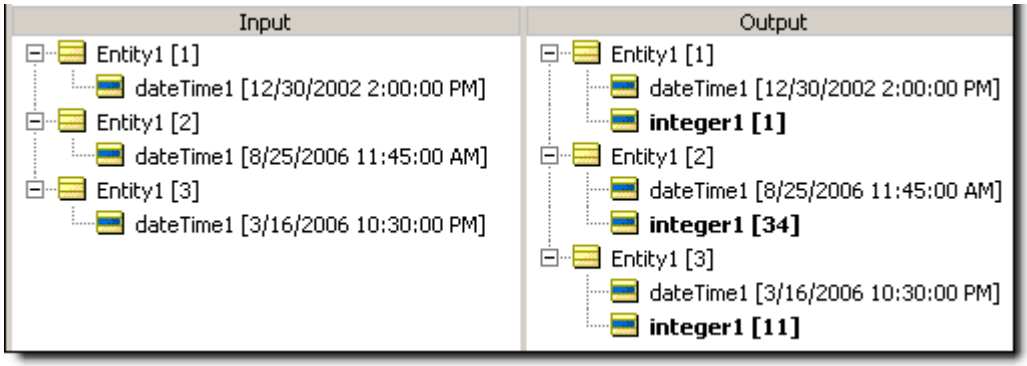
The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

RULESHEET EXAMPLE

The following Rulesheet uses **.weekOfYear** to assign a value to `integer1`.



SAMPLE TEST



Year

SYNTAX

`<DateTime>`.year

`<Date>`.year

DESCRIPTION

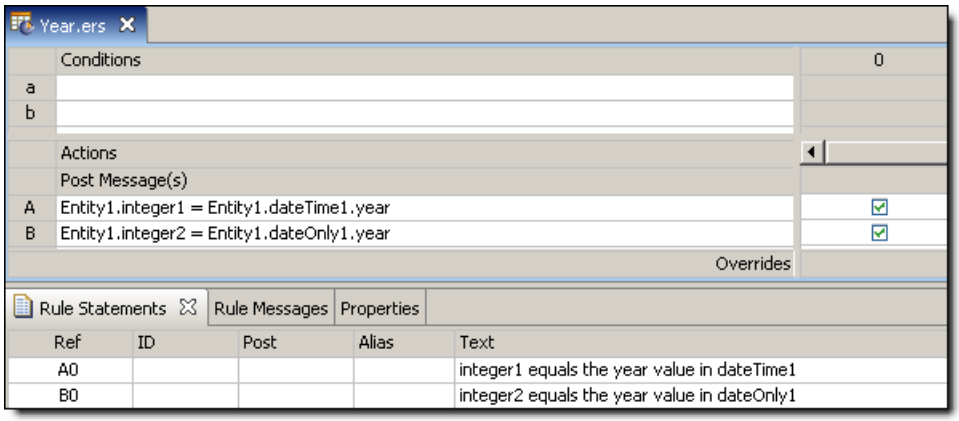
Returns the century/year portion of `<DateTime>` or `<Date>`. The returned value is a four digit Integer.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

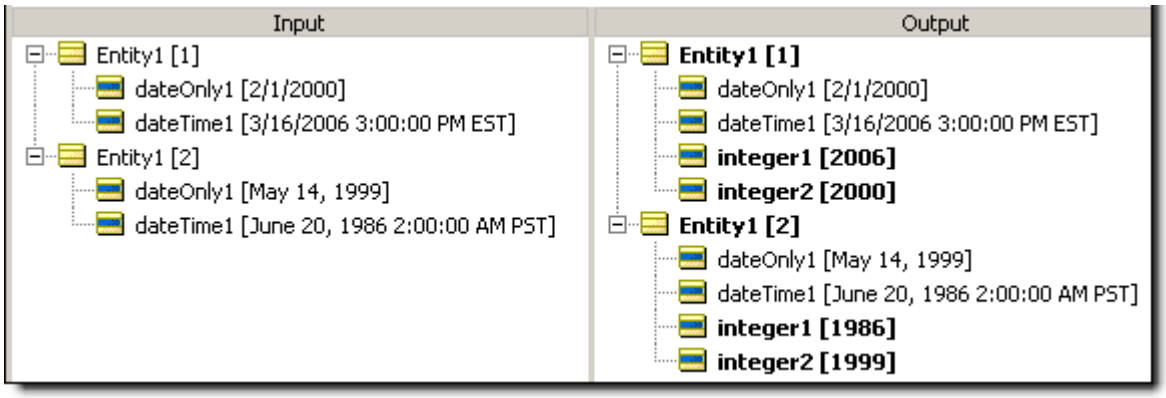
RULESHEET EXAMPLE

The following Rulesheet uses **.year** to evaluate `dateTime1` and `dateOnly1` and assign the year values to `integer1` and `integer2`, respectively.



SAMPLE TEST

A sample Ruletest provides three examples of `dateTime1` and `dateOnly1`. Input and Output panels are shown below:



Years between

SYNTAX

`<DateTime1>`.yearsBetween(`<DateTime2>`)

`<Date1>`.yearsBetween(`<Date2>`)

DESCRIPTION

Returns the Integer number of years between DateTimes or between Dates. The number of months in `<DateTime2>` is subtracted from the number of months in `<DateTime1>`, and the result is divided by 12 and truncated. This function returns a positive number if `<DateTime2>` is later than `<DateTime1>`.

USAGE RESTRICTIONS

The Operators row of the table in Summary Table of Vocabulary Usage Restriction applies. No special exceptions.

RULESHEET EXAMPLE

The following Rulesheet uses **.yearsBetween** to determine the number of months that have elapsed between `dateTime1` and `dateTime2`, compare it to the Values set, and assign a value to `string1`.

YearsBetween.ers

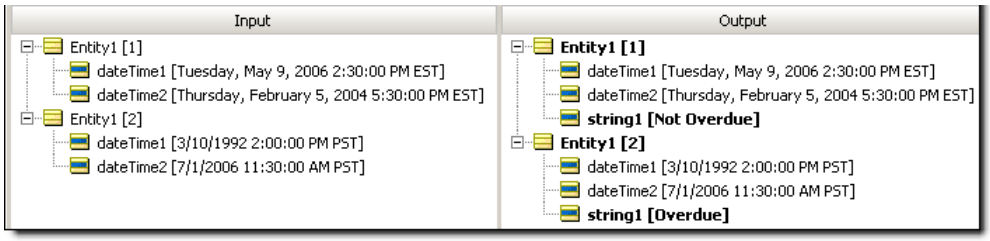
	Conditions	0	1	2
a	Entity1.dateTime1.yearsBetween(Entity1.dateTime2)		<=3	>3
b				
	Actions			
	Post Message(s)			
A	Entity1.string1		'Not Overdue'	'Overdue'
	Overrides			

Rule Statements | Rule Messages | Properties

Ref	ID	Post	Alias	Text
1				If 3 or fewer years have elapsed between dateTime1 and dateTime2, then Entity1 is not overdue
2				If more than 3 years have elapsed between dateTime1 and dateTime2, then Entity1 is overdue

SAMPLE TEST

A sample Ruletest provides `dateTime1` and `dateTime2` for two examples. Input and Output panels are shown below.



5

Special syntax

For details, see the following topics:

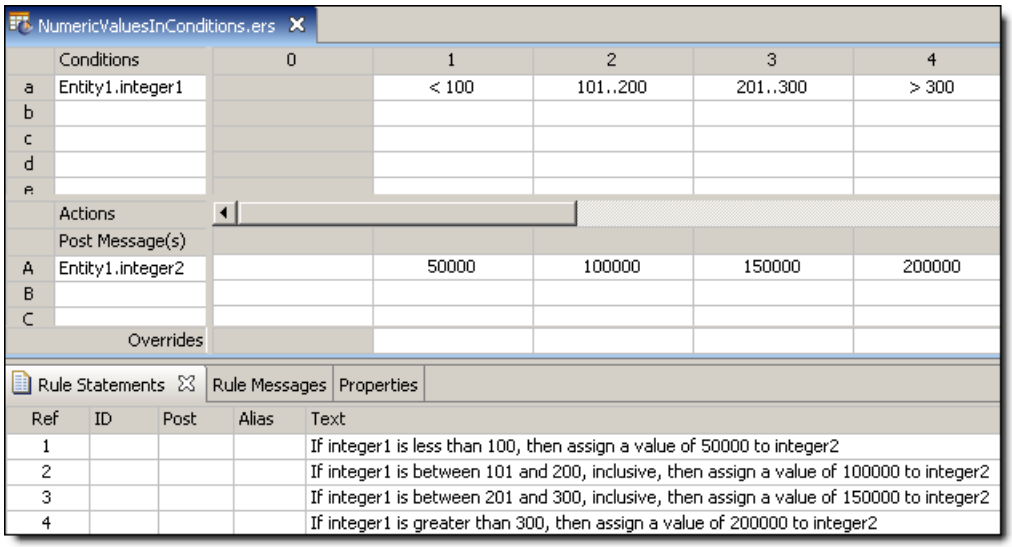
- Value ranges
- Using value sets in condition cells
- Using variables as condition cell values
- Embedding attributes in posted rule statements
- Including apostrophes in strings
- Advanced collection syntax
- Statement blocks

Value ranges

When using values in Condition Cells for Integers, Decimals, Strings, or DateTime data types, the values do not need to be precise – they can be in the form of a range (except for Boolean values.) A value range is typically expressed in the following format: `x..y`, where `x` and `y` are the starting and ending values for the range *inclusive* of the endpoints if there is no other notation to indicate otherwise. This is illustrated in Rulesheet using Numeric Value Ranges in Condition Values Set.

Numeric value ranges in conditions

Figure 6: Rulesheet using Numeric Value Ranges in Condition Values Set



NumericValuesInConditions.ers

	Conditions	0	1	2	3	4
a	Entity1.integer1		< 100	101..200	201..300	> 300
b						
c						
d						
e						
	Actions					
	Post Message(s)					
A	Entity1.integer2		50000	100000	150000	200000
B						
C						
	Overrides					

Rule Statements | Rule Messages | Properties

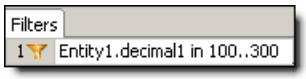
Ref	ID	Post	Alias	Text
1				If integer1 is less than 100, then assign a value of 50000 to integer2
2				If integer1 is between 101 and 200, inclusive, then assign a value of 100000 to integer2
3				If integer1 is between 201 and 300, inclusive, then assign a value of 150000 to integer2
4				If integer1 is greater than 300, then assign a value of 200000 to integer2

In this example, we are assigning an `integer2` value to `Entity1` depending on its `integer1` value. The value range `101..200` represents all values (integers in this case) between `101` and `200`, including `101` and `200`. This is an inclusive range because both the starting and ending values are included in the range.

Numeric value ranges in filter rows

Numeric value ranges can be used as Filter expressions as shown. Note the use of special term `in`, as shown below.

Figure 7: Rulesheet using a Numeric Value Range as a Precondition/Filter



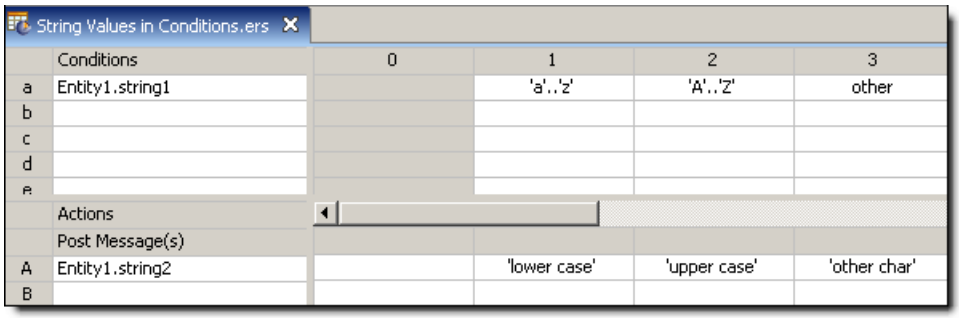
Filters

1	Entity1.decimal1 in 100..300

String value ranges in condition cells

When using value range syntax with String types, be sure to enclose literal values inside single quotes, as shown in the following figure. Corticon Studio will usually perform this for you, but always check to make sure it has interpreted your entries correctly.

Figure 8: Rulesheet using String Value Ranges in Condition Values Set

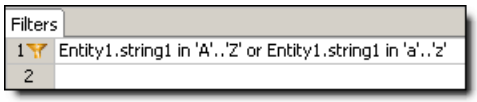


String Values in Conditions.ers					
	Conditions	0	1	2	3
a	Entity1.string1		'a'..'z'	'A'..'Z'	other
b					
c					
d					
e					
	Actions				
	Post Message(s)				
A	Entity1.string2		'lower case'	'upper case'	'other char'
B					

String value ranges in filter rows

String value ranges can be used as Filter expressions as shown in the following figure:

Figure 9: Rulesheet using String Value Ranges in Precondition Value Set



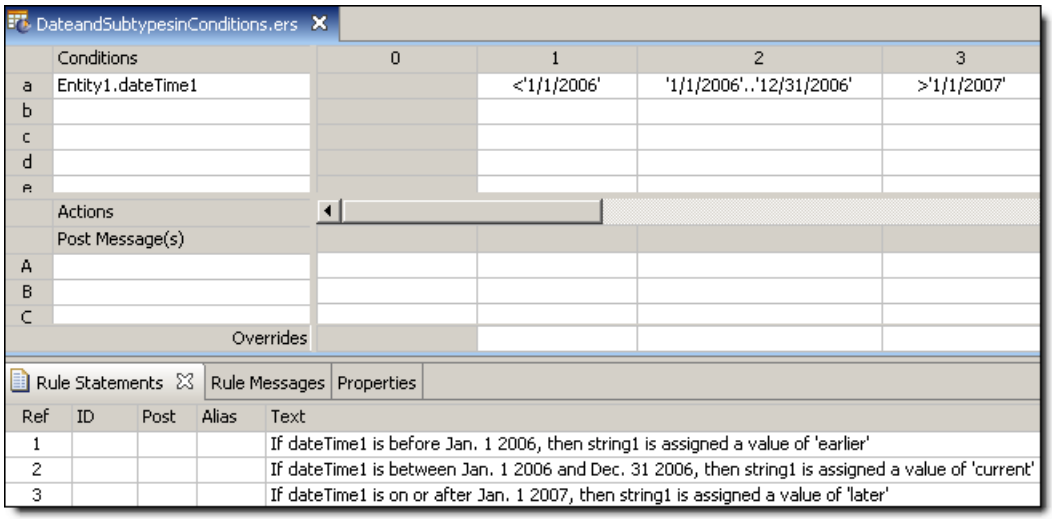
Filters	
1	Entity1.string1 in 'A'..'Z' or Entity1.string1 in 'a'..'z'
2	

Note the use of special term `in`. Also notice the use of Boolean operator `or` to combine two expressions. This single Precondition is satisfied only if `string1` is an English letter.

DateTime, date, and time value ranges in condition cells

When using value range syntax with date types, be sure to enclose literal date values inside single quotes, as shown:

Figure 10: Rulesheet using a Date Value Range in Condition Cells



DateandSubtypesinConditions.ers

	Conditions	0	1	2	3
a	Entity1.dateTime1		<'1/1/2006'	'1/1/2006'..'12/31/2006'	>'1/1/2007'
b					
c					
d					
e					
	Actions				
	Post Message(s)				
A					
B					
C					
	Overrides				

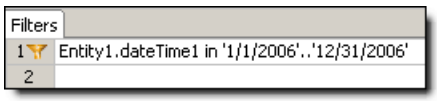
Rule Statements | Rule Messages | Properties

Ref	ID	Post	Alias	Text
1				If dateTime1 is before Jan. 1 2006, then string1 is assigned a value of 'earlier'
2				If dateTime1 is between Jan. 1 2006 and Dec. 31 2006, then string1 is assigned a value of 'current'
3				If dateTime1 is on or after Jan. 1 2007, then string1 is assigned a value of 'later'

DateTime, date, and time value ranges in filter rows

DateTime, Date, and Time value ranges can be used as Filter expressions as shown. Note the use of special term `in`.

Figure 11: Rulesheet using a Date Value Range in a Filter



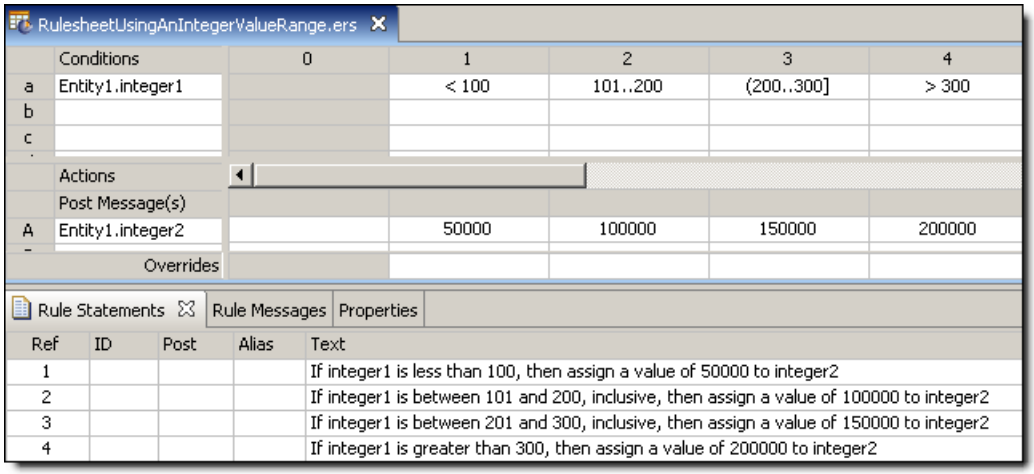
Filters

1	Entity1.dateTime1 in '1/1/2006'..'12/31/2006'
2	

Inclusive & exclusive ranges

Corticon Studio also gives you the option of defining value ranges where one or both of the starting and ending values are "exclusive", meaning that the starting/ending value is **not** included in the range of values. Rulesheet using an Integer Value Range in Condition Values Set shows the same *Rulesheet* as in Rulesheet using Numeric Value Ranges in Condition Values Set, but with one difference: we have changed the value range `201..300` to `(200..300]`. The starting parenthesis `(` indicates that the starting value for the range, `200`, is excluded – it is **not** included in the range of possible values. The ending bracket `]` indicates that the ending value is inclusive. Since `integer1` is an Integer value, and therefore no fractional values are allowed, `201..300` and `(200..300]` are equivalent and our Values set in Rulesheet using an Integer Value Range in Condition Values Set is still complete as it was in Rulesheet using Numeric Value Ranges in Condition Values Set.

Figure 12: Rulesheet using an Integer Value Range in Condition Values Set



RulesheetUsingAnIntegerValueRange.ers

	Conditions	0	1	2	3	4
a	Entity1.integer1		< 100	101..200	(200..300]	> 300
b						
c						
	Actions					
	Post Message(s)					
A	Entity1.integer2		50000	100000	150000	200000
	Overrides					

Rule Statements | Rule Messages | Properties

Ref	ID	Post	Alias	Text
1				If integer1 is less than 100, then assign a value of 50000 to integer2
2				If integer1 is between 101 and 200, inclusive, then assign a value of 100000 to integer2
3				If integer1 is between 201 and 300, inclusive, then assign a value of 150000 to integer2
4				If integer1 is greater than 300, then assign a value of 200000 to integer2

Listed below are all of the possible combinations of parenthesis and bracket notation for value ranges and their meanings:

`(x..y)` - is the range between x & y, excluding both x & y

`(x..y]` - is the range between x & y, excluding x and including y

`[x..y)` - is the range between x & y, including x and excluding y

`[x..y]` - is the range between x & y, including both x & y

As illustrated in Rulesheet using Numeric Value Ranges in Condition Values Set and Rulesheet using an Integer Value Range in Condition Values Set, if a value range has no enclosing parentheses or brackets, it is assumed to be closed. It is therefore not necessary to use the `[..]` notation for a closed range in Corticon Studio; in fact, if you try to create a closed value range by entering `[..]`, the square brackets will be automatically removed. However, should either end of a value range have a parenthesis or a bracket, then the other end must also have a parenthesis or a bracket. For example, `x..y)` is not allowed, and is properly expressed as `[x..y)`.

When using range notation, always ensure `x` is less than `y`, i.e., an ascending range. A range where `x` is greater than `y` (a descending range) may result in errors during rule execution.

Overlapping value ranges

One final note about value ranges: they **might overlap**. In other words, Condition Cells may contain the two ranges `0..10` and `5..15`. It is important to understand that when overlapping ranges exists in rules, the rules containing the overlap are frequently ambiguous and more than one rule may fire for a given set of input *`Ruletest`* data.

Note: In Corticon 4.x and earlier, overlapping Value sets were not allowed.

Rulesheet with Value Range Overlap shows an example of value range overlap.

Figure 13: Rulesheet with Value Range Overlap

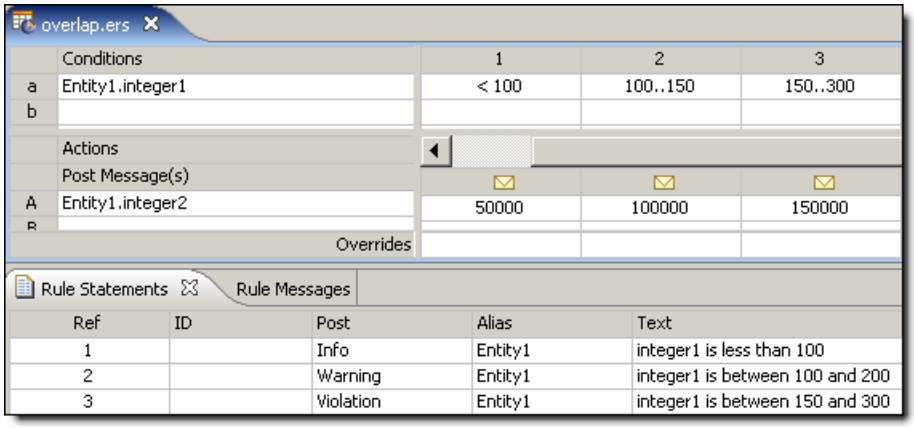


Figure 14: Rulesheet expanded with Ambiguity Check applied

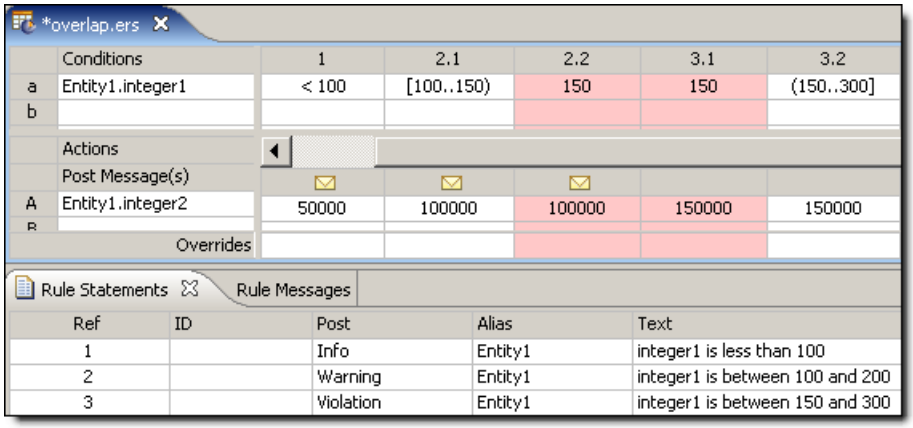
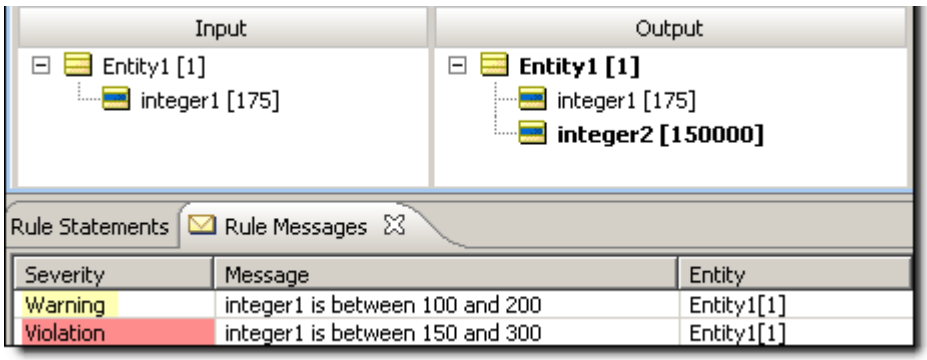


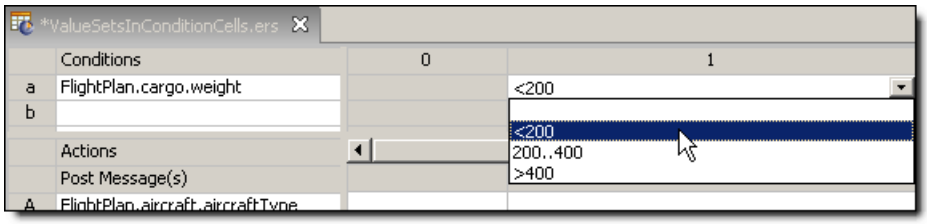
Figure 15: Ruletest showing multiple rules firing for given test data



Using value sets in condition cells

Most Conditions implemented in the Rules section of the Rulesheet use a single value in a Cell, as shown in the following figure:

Figure 16: Rulesheet with One Value Selected in Condition Cell



Sometimes, however, it is useful to combine more than one value in the same Cell. This is accomplished by holding **CTRL** while clicking to select multiple values from the Condition Cell drop-down box. When multiple values are selected in this manner, pressing **ENTER** will automatically enclose the resulting set in curly brackets `{ . . }` in the Cell as shown in the sequence of Rulesheet with Two Values Selected in Condition Cell and Rulesheet with Value Set in Condition Cell. Additional values may also be typed into Cells. Be sure the comma separators and curly brackets remain correct during hand-editing.

Figure 17: Rulesheet with Two Values Selected in Condition Cell

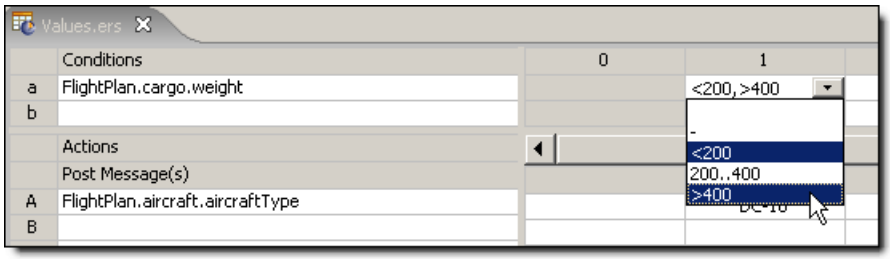
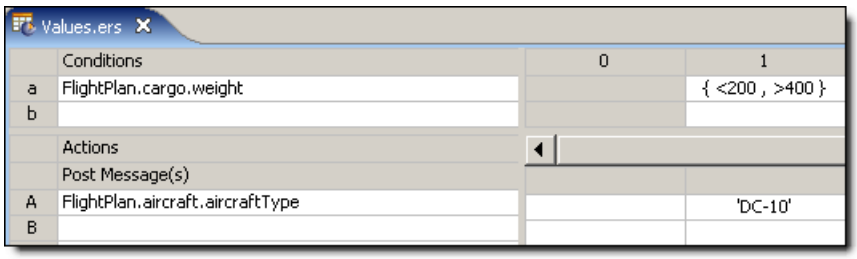


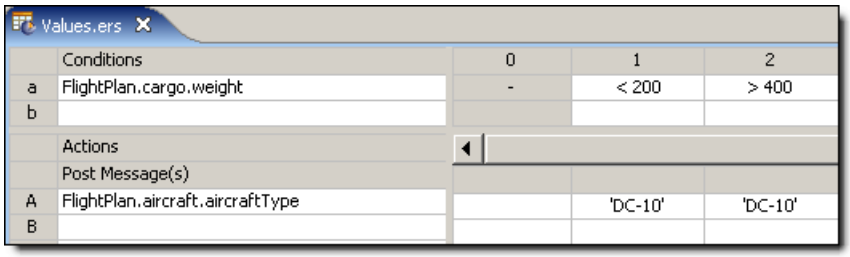
Figure 18: Rulesheet with Value Set in Condition Cell



The rule implemented in Column 1 of Rulesheet with Value Set in Condition Cell is logically equivalent to the Rulesheet shown in Rulesheet with Two Rules in Lieu of Value Set. Both are implementations of the following rule statement:

1. If a flightplan's cargo weight is less than 200 **OR** greater than 400, then the flightplan's aircraft type must be a DC-10

Figure 19: Rulesheet with Two Rules in Lieu of Value Set



Values.ers	Conditions	0	1	2
a	FlightPlan.cargo.weight	-	< 200	> 400
b				
	Actions			
	Post Message(s)			
A	FlightPlan.aircraft.aircraftType		'DC-10'	'DC-10'
B				

If you write rules that are logically **OR**'ed in separate Columns, performing a Compression will reduce the Rulesheet to the fewest number of Columns possible by creating value sets in Cells wherever possible. Fewer Columns results in faster Rulesheet execution, even when those Columns contain value sets. Compressing the Rulesheet in Rulesheet with Two Rules in Lieu of Value Set will result in the Rulesheet in Rulesheet with Value Set in Condition Cell.

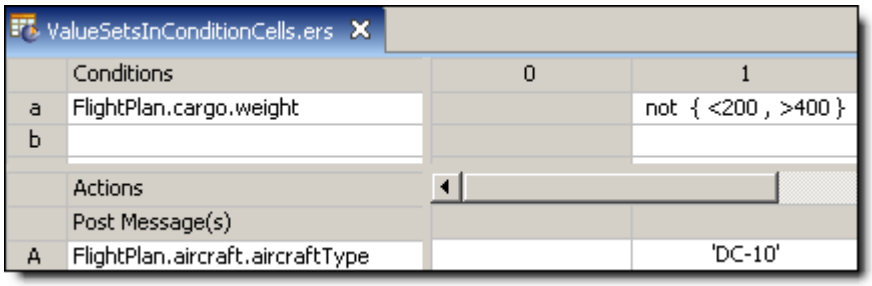
Condition Cell value sets can also be negated using the **not** operator. To negate a value, simply type `not` in front of the leading curly bracket `{` as shown in Negating a Value Set in a Condition Cell. This is an implementation of the following rule statement:

1. If a flightplan's cargo weight is **NOT** less than 200 **OR NOT** greater than 400, then the flightplan's aircraft type must be a DC-10

which, given the Condition Cell's value set, is equivalent to:

1. If a flightplan's cargo weight is between 200 and 400 (inclusive), then the flightplan's aircraft type must be a DC-10

Figure 20: Negating a Value Set in a Condition Cell



ValueSetsInConditionCells.ers	Conditions	0	1
a	FlightPlan.cargo.weight		not { <200 , >400 }
b			
	Actions		
	Post Message(s)		
A	FlightPlan.aircraft.aircraftType		'DC-10'

Value sets can also be created in the Overrides Cells at the foot of each Column. This allows one rule to override multiple rules in the same Rulesheet.

Using variables as condition cell values

You can use a variable as a condition's cell value. However, there are constraints:

- Either **all** of the rule cell values for a condition row contain references to the *same* variable (with the exception of dashes), or **none** of the rule cell values for a condition row reference *any* variable.

- Only one variable can be referenced by various rules for the same condition row.
- Logical expressions in the various rules for the same condition row should be logically non-overlapping.
- A condition value that uses a colon, such as `A:B`, is not valid.

Derived value sets are created by accounting for all logical ranges possible around the variable.

The following Rulesheet uses the `Cargo` Vocabulary to illustrate the valid and invalid use of variables. Note that the Vocabulary editor marks invalid values in red.

	Conditions	0	1	2	3
a	Aircraft.maxCargoVolume		< Cargo.volume	> Cargo.volume	Cargo.volume
b	Aircraft.maxCargoVolume		<= Cargo.volume	> Cargo.volume	-
c	Aircraft.maxCargoVolume		< Cargo.volume	> Cargo.volume	-
d	Aircraft.maxCargoVolume		< Cargo.volume	-	-
e					
f	Aircraft.maxCargoVolume		< Cargo.volume	FlightPlan.cargo.volume	Cargo.volume
g	Aircraft.maxCargoVolume		< Cargo.volume	5	10..15
h	Aircraft.maxCargoVolume		< Cargo.volume	<= Cargo.volume	Cargo.volume
i	Aircraft.maxCargoVolume		A1:B2		

Derived values when using variables

The following tables abbreviate the attribute references shown in the illustration.

Table 1: Rulesheet columns

Conditions	1	2	3	Derived Value Set
`A.maxCV`	`< C.v`	`> C.v`	`C.v`	`{< C.v, > C.v, C.v}`
`A.maxCV`	`<= C.v`	`> C.v`		`{<= C.v, > C.v }`
`A.maxCV`	`< C.v`	`> C.v`		`{< C.v, > C.v, C.v }`
`A.maxCV`	`< C.v`			`{< C.v, >= C.v}`

Improper use of variables

Table 2: Rulesheet condition f: Attempt to use multiple variables

Conditions	1	2	3
A.maxCV	< C.v	**> FP.c.v**	C.v

Table 3: Rulesheet condition g: Attempt to mix variables and literals

Conditions	1	2	3
A.maxCV	< C.v	**5**	**10..15**

Table 4: Rulesheet condition h: Attempt to use logically overlapping expressions

Conditions	1	2	3
A.maxCV	< C.v	**<= C.v**	C.v

Embedding attributes in posted rule statements

It is frequently useful to "embed" attribute values within a Rule Statement, so that posted messages contain actual data. Special syntax must be used to differentiate the static text of the rule statement from the dynamic value of the attribute. As shown in Sample Rulesheet with Rule Statements Containing Embedded Attributes, an embedded attribute must be enclosed by curly brackets `{..}` to distinguish it from the static Rule Statement text.

It may also be helpful to indicate which parts of the posted message are dynamic, so a user seeing a message knows which part is based on "live" data and which part is the standard rule statement. As shown in Sample Rulesheet with Rule Statements Containing Embedded Attributes, square brackets are used immediately outside the curly brackets so that the dynamic values inserted into the message at rule execution will be "bracketed". The use of these square brackets is optional – other characters may be used to achieve the intended visual distinction.

Remember, Action Rows execute in numbered order (from top to bottom in the Actions pane), so a Rule Statement that contains an embedded attribute value must not be posted before the attribute has a value. Doing so will result in a `null` value inserted in the posted message.

Figure 21: Sample Rulesheet with Rule Statements Containing Embedded Attributes

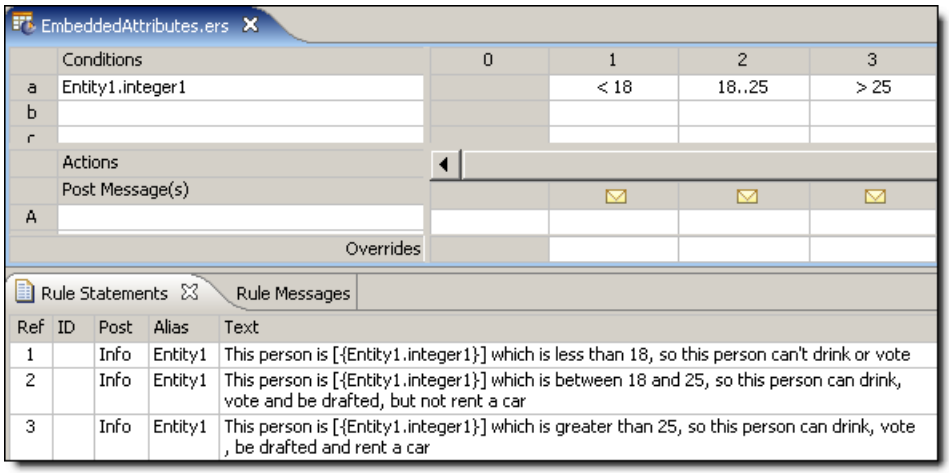
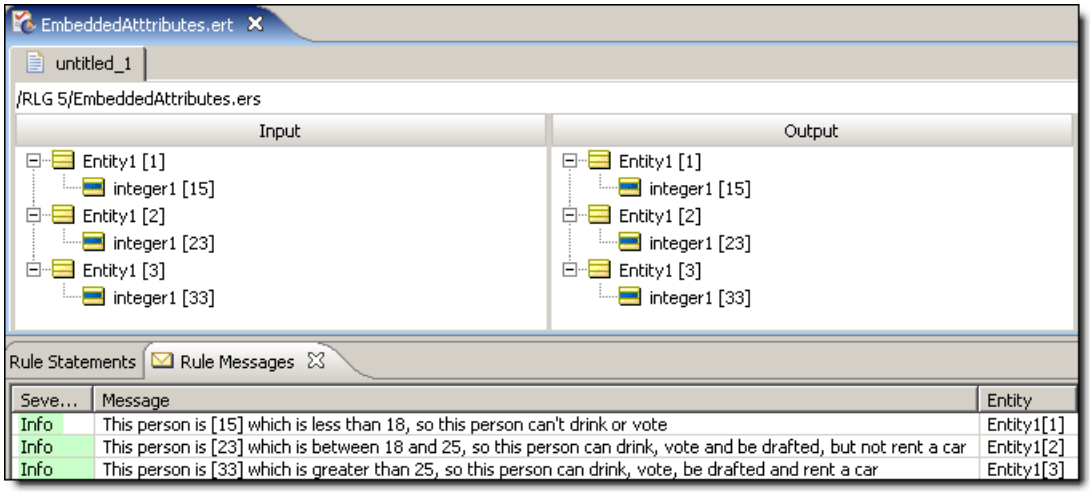


Figure 22: Rule Messages Window Showing Bracketed Embedded Attributes (Orange Box)



When an attribute uses an Enumerated Custom Data Type, the dynamic value embedded in the posted Rule Message will be the Value, not the Label. See the *Rule Modeling Guide*, "Building the Vocabulary" chapter for more information about Custom Data Types.

No expressions in Rule Statements

A reminder about the table in Summary Table of Vocabulary Usage Restriction, which specifies that the only parts of the Vocabulary that may be embedded in Rule Statements are attributes and functions (`today` and `now`). No operators or expressions are permitted inside Rule Statements. Often, operators will cause error messages when you try to save a Rule Set. Sometimes the Rule Statement itself will turn red. Sometimes an embedded equation will even execute as you intended. But sometimes no obvious error will occur, but the rule does not executed as intended. Just remember that operators and expressions are not supported in Rule Statements.

Including apostrophes in strings

String values in Corticon Studio are always enclosed in single quotes. But occasionally, you may want the String value to include single quote or apostrophe characters. If you enter the following text in Corticon Studio:

```
entity1.string1=‘Jane's dog Spot’
```

The text will turn red, because Corticon Studio thinks that the `string1` value is `‘Jane’` and the remaining text `s dog Spot’` is invalid. To properly express a String value that includes single quotes or apostrophes, you must use the special character backslash (`\`) that tells Corticon Studio to ignore the apostrophe(s) as follows:

```
entity1.string1=‘Jane\'s dog Spot’
```

When preceded by the backslash, the second apostrophe will be ignored and assumed to be just another character within the String. This notation works in all sections of the *`Rulesheet`*, including Values sets. It also works in the Possible Values section of the Vocabulary Editor.

Advanced collection syntax

Collection syntax contains some subtleties which are worth learning once you are comfortable with the basics described in the *Rule Modeling Guide's* Collections chapter. It's sometimes helpful when writing collection expressions to step through them, left to right, as if you were reading a sentence. This helps us understand better how the pieces combine to create the full expression. It also helps us to know what else we can safely add to the expression to increase its utility. Let's try this approach in order to dissect the following expression:

```
Collection1 -> sortedBy(attribute1) -> last.attribute2
```

1. `Collection1`

   This expression returns the collection {e_1, e_2, e_3, e_4, e_5,…e_n} where e_x is an element (an entity) in `Collection1`. We already know that alias `Collection1` represents the entire collection.
2. `Collection1 -> sortedBy(attribute1)`

   This expression returns the collection {e_1, e_2, e_3, e_4, e_5,…e_n} arranged in ascending order based on the values of `attribute1` (which we call the "index").
3. `Collection1 -> sortedBy(attribute1) -> last`

   returns {e_n} where e_n is the last element in `Collection1` when sorted by `attribute1`

   This expression returns a ***specific entity*** (element) from `Collection1`. It does not return a specific value, but once we have identified a specific entity, we can easily reference the value of any attribute it contains, as in the following:
4. `Collection1 -> sortedBy(attribute1) -> last.attribute2`

   which returns {e_n.attribute2}

This expression not only returns a specific value, but just as importantly, it also returns the entity the value belongs to. This "entity context" is important because it allows us to do things to the entity itself, like assign a value to one of its attributes. For example:

```
Collection1 -> sortedBy(attribute1) -> last.attribute2='xyz'
```

The above expression will assign the value of `xyz` to `attribute2` of the entity whose `attribute1` is highest in `Collection1`. Contrast this with the following:

```
Collection1.attribute1 -> sortedBy(attribute1) -> last
```

which returns a single integer value, like 14.

Notice that all we have now is a number, a *value*. We have lost the entity context, so we can't do anything to the entity that owns the attribute with value of 14. In many cases, this is just fine. Take for example:

```
Collection1.attribute1 -> sortedBy(attribute1) -> last > 10
```

In this expression, it is not important that we know which element has the highest value of `attribute1`, all we want to know is if the highest value (whomever it "belongs" to) is greater than 10.

Understanding the subtleties of collection syntax and the concept of entity context is important because it helps us to use the returned entities or values correctly. For example:

Return the lower of the following two values:

- 12
- The age of the oldest child in the family

What is really being compared here? Do we care *which* child is oldest? Do we need to know his or her name? No. We simply need to compare the age of that child (whichever one is oldest) with the value of 12. So this is the expression that models this logic:

```
family.age -> sortedByDesc(age) -> first.min(12)
```

.min, as we know, is an operator that *acts upon* numeric data types (Integer or Decimal). And since we also know that `family.age -> sortedByDesc(age) -> first` returns a number, then it is legal and valid to use `.min` at the end of this expression.

What about this scenario: Name the youngest child Junior.

```
family -> sortedByDesc(age) -> last.name='Junior'
```

Now we want to return a *specific entity* – that of the youngest child – and assign to its name a value of `Junior`. We need to keep the entity context in order to make this assignment, and the expression above accomplishes this.

Statement blocks

Sequence operators can easily extract an attribute value from the first, last or other specific element in a sorted collection (see first, last, or at(n) for examples). This is especially useful when the attribute's value is involved in a comparison in a Conditional or Preconditional rule. Sometimes, however, you want to identify a particular element in a sequence and "flag" or "tag" it for use in subsequent rules. This can be accomplished using special syntax called Statement Blocks.

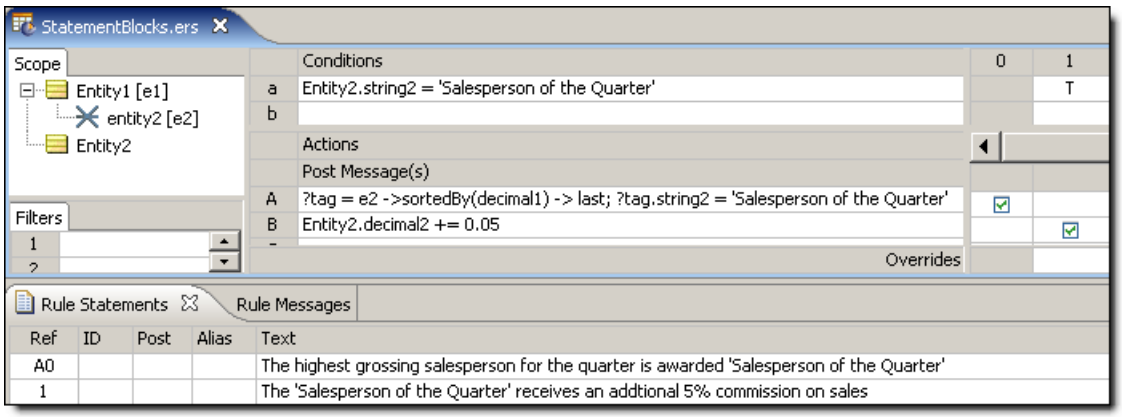
Statement Blocks, permitted only in the Action rows of the Rulesheet, use special variables, prefixed by a `?` character, to "hold" or "pin" an element so that further action may be taken on it, including tagging it by assigning a value to one of its attributes. These special holder variables may be declared "on the fly", meaning they do not need to be defined anywhere prior to use.

Here's an example. In a sales management system, the performance of sales reps is analyzed every quarter, and the highest grossing sales rep is awarded "Salesperson of the Quarter". This special status is then used to automatically increase the rep's commission percentage on sales made in the following quarter. We will use the same generic Vocabulary as in all previous examples, but make these assumptions:

Vocabulary Term	Meaning
`Entity2`	a salesperson
`Entity1.entity2`	collection of salespeople
`Entity2.string1`	a salesperson's name
`Entity2.decimal1`	a salesperson's quarterly sales
`Entity2.string2`	a salesperson's award
`Entity2.decimal2`	a salesperson's commission percentage

Using this Vocabulary, we construct the Rulesheet shown in

Figure 23: Rulesheet using Statement Block to Identify and Reward Winner



Important Notes about Statement Blocks

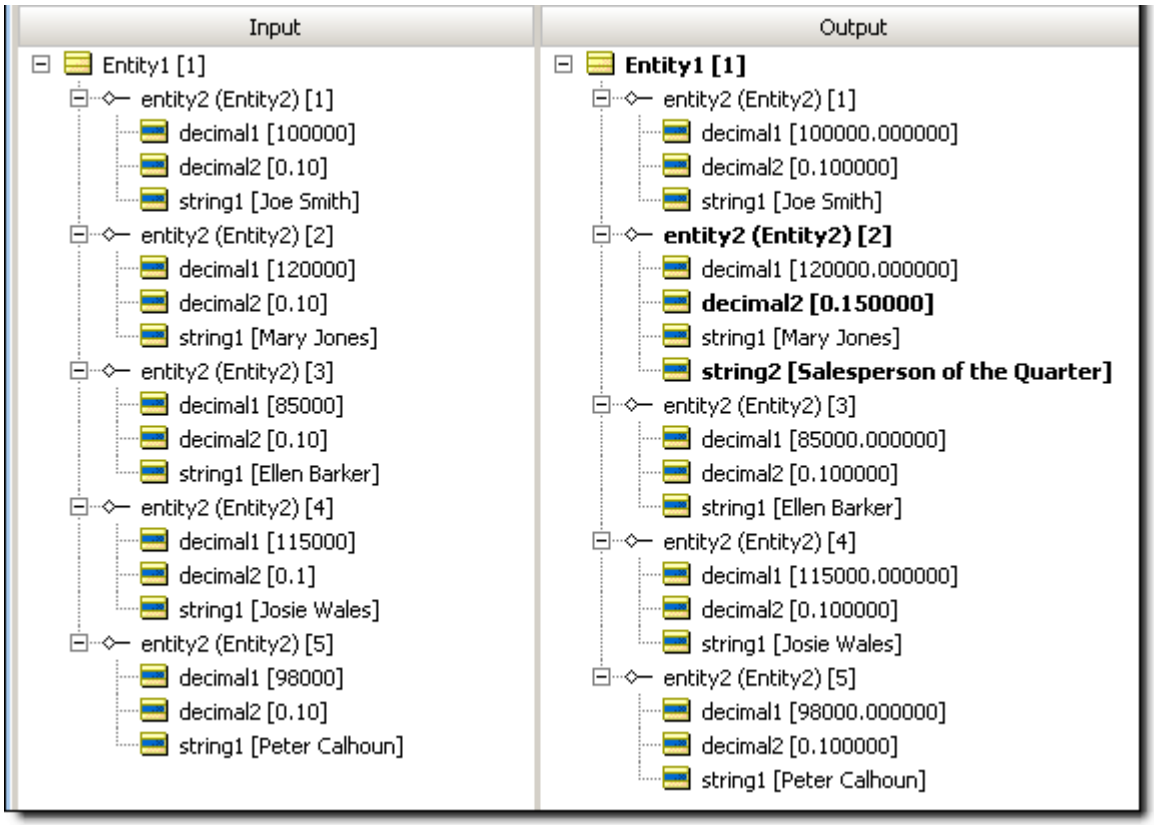
As expressed in Action row A in the figure above, a statement block consists of two separate expressions:

1. the first part assigns an element of a sequence to a special holder variable, prefixed by the `?` character. This variable is unusual because it represents an *element*, not a *value*. Here, the highest grossing salesperson is expressed as the last element of the collection of salespeople (`e2`), sorted in ascending order according to quarterly sales (`decimal1`). Once identified by the sequencing operator `□last`, this salesperson is momentarily "held" by the `?tag` variable, which we declared "on-the-fly".
2. the second part of the statement – the part following the semicolon – assigns a value to an attribute of the element held by the `?tag`. In our example, we are assigning a value of `'Salesperson of the Quarter'` to the `string2` attribute of the salesperson held by `?tag`. In effect, we have "tagged" the highest grossing salesperson with this award.
3. the two parts above must be included on the same Action Row, separated by a semicolon. If the two parts are separated in different sections or in different Rows of the same section, the element represented by the `?` variable is "lost", in other words, `?tag` loses its "grip" on the element identified by the sequencing operator.

Now that we have tagged our winner, we can use the tagged element (awardee) to take additional actions. In the Conditional rule, we increase the commission percentage of the winner by 5% using the increment operator.

The next figure shows a Ruletest Input and Output pane. As expected, our highest grossing salesperson has been awarded "Salesperson of the Year" honors, and has had her commission raised by an additional 5%.

Figure 24: Output Panel with Winner and Adjusted Commission in Bold Black Text



A

Character precedence: Unicode & Java Collator

The Unicode standard assigns a 4 digit (hexadecimal) code to every character, including many that can't be typed on standard keyboards. Java (and hence Progress Corticon software) uses a special method named `Collator` to sort these characters in specific sequences based on the I18n locale of the user.

While sorting by locale allows for regional variations of language-specific characters like accents, the combination of these two systems can also make determining character precedence very complicated. The Unicode code and Java Collator sequence for standard keyboards in US-English locale is shown in the table below.

Sequences for other languages and/or locales may differ, and many other Unicode characters are available but are not shown in the table. We recommend http://www.unicode.org/charts for more information on the Unicode system and http://java.sun.com/docs/books/tutorial/i18n/text/locale.html for more information on the Java Collator method.

- `'Z'='z'` evaluates to `true` because character `Z` has the same precedence as `z` (69=69). A given letter has the same precedence regardless of its case. This is an important difference between character precedence determined by ISO or ASCII systems, and the Java Collator system used by *Corticon*.
- `'C & S' < 'C and S'` evaluates to `true` because character `a` has a higher precedence than `&` (26 < 44). These characters are decisive because they are the first different characters encountered as the two strings are compared beginning with characters in position 1.
- `'B' > 'aardvark'` evaluates to `true` because character `B` has a higher precedence than `a` (45 > 44).
- `'Marilynn' < 'Marilyn'` evaluates to `false` because character `n` has a higher precedence than `<space>` (57 > 1). The first seven characters of each String are identical, so the final character comparison is decisive.

character	name	precedence	Unicode 5.0 code
	typed space	1	0020
-	dash or minus sign	2	002D
_	underline or underscore	3	005F
,	comma	4	002C
;	semicolon	5	003B
:	colon	6	003A
!	exclamation point	7	0021
?	question mark	8	003F
/	slash	9	002F
.	period	10	002E
`	grave accent	11	0060
^	circumflex	12	005E
~	tilde	13	007E
‘	apostrophe	14	0027
“	quotation marks	15	0022
(	left parenthesis	16	0028
)	right parenthesis	17	0029
[	left bracket	18	005B
]	right bracket	19	005D

character	name	precedence	Unicode 5.0 code
{	left brace	20	007B
}	right brace	21	007D
@	at symbol	22	0040
$	dollar sign	23	0024
*	asterisk	24	002A
\	backslash	25	005C
&	ampersand	26	0026
#	number sign or hash sign	27	0023
%	percent sign	28	0025
+	plus sign	29	002B
<	less than sign	30	003C
=	equals sign	31	003D
>	greater than sign	32	003E
|	vertical line	33	007C
0..9	numbers 1 through 9	34-43	0031-0039
a, A	letter a, small and capital	44	0061, 0041
b, B	letter b, small and capital	45	0062, 0042
c, C	letter c, small and capital	46	0063, 0043
d, D	letter d, small and capital	47	0064, 0044
e, E	letter e, small and capital	48	0065, 0045
f, F	letter f, small and capital	49	0066, 0046
g, G	letter g, small and capital	50	0067, 0047
h, H	letter h, small and capital	51	0068, 0048
I, I	letter I, small and capital	52	0069, 0049
j, J	letter j, small and capital	53	006A, 004A
k, K	letter k, small and capital	54	006B, 004B

character	name	precedence	Unicode 5.0 code
l, L	letter l, small and capital	55	006C, 004C
m, M	letter m, small and capital	56	006D, 004D
n, N	letter n, small and capital	57	006E, 004E
o, O	letter o, small and capital	58	006F, 004F
p, P	letter p, small and capital	59	0070, 0050
q, Q	letter q, small and capital	60	0071, 0051
r, R	letter r, small and capital	61	0072, 0052
s, S	letter s, small and capital	62	0073, 0053
t, T	letter t, small and capital	63	0074, 0054
u, U	letter u, small and capital	64	0075, 0055
v, V	letter v, small and capital	65	0076, 0056
w, W	letter w, small and capital	66	0077, 0057
x, X	letter x, small and capital	67	0078, 0058
y, Y	letter y, small and capital	68	0079, 0059
z, Z	letter z, small and capital	69	007A, 005A

B

Arithmetic operator precedence

Order	Operator	Operator Name	Example
1	()	Parenthesis	(5.5 / 10)
2	Unary	Negation	-10
3	*	Multiplication	5.5 * 10
	/	Division	5.5 / 10
4	+	Addition	5.5 + 10
	-	Subtraction	10.0 – 5.5
5	<	Less Than	5.5 < 10
	<=	Less Than Or Equal To	5.5 <= 5.5
	>	Greater Than	10 > 5.5
	>=	Greater Than Or Equal To	10 >= 10
6	=	Equal	5.5=5.5
	<>	Not Equal	5.5 <> 10
7	**	Exponentiation	10 ** 2

C

DateTime data type

FORMATS OR MASKS

DateTime information may take many different formats. Corticon Studios use a common source of acceptable DateTime, Date Only, and Time Only formats, also known as "masks".

For example, a date mask may specify `yyyy-MM-dd` as an acceptable date format, which means that an attribute of type DateTime (or Date) may "hold" or "contain" data that conforms to this format. `'2013-04-12'` conforms to this mask; `'April12th,2013'` does not.

For proper execution, it is important to ensure that date formats used during rule development and testing (and are included in the rule builders' Corticon Studio installations) are also present in the Corticon Server's installation. See *Server Integration & Deployment Guide* for more details about setting and modifying masks in both Corticon Studio and Corticon Server.

PRESENTATION AND PERSISTENCE

Most commercial databases represent dates as DateTimes. Such DateTimes are frequently stored as UTC, namely the number of milliseconds that have transpired from an arbitrary epoch (for example, 1/1/1970 00:00:00 GMT); this is not a universal standard but is a very popular convention. UTC dates can be *rendered* in the user's local time zone, *but this is merely a matter of presentation*. A UTC represents a simultaneous point in time for two observers regardless of where on earth they reside.

However, some date or time concepts, such as *holiday*, cannot be expressed conveniently as a discrete time point. *Christmas* (12/25/XX) actually denotes different time frames depending on the observers' time zones; thus, Corticon *carries* (that is, holds in memory) all dates in GMT with the time portion zeroed (that is, midnight). This approach addresses the holiday problem because a user can enter holiday dates into the database and not have them shift when they are rendered in the user's local time zone.

Carrying GMT dates should be transparent to the user. Dates expressed as strings in incoming XML are parsed and the proper data type is inferred; for dates, they are immediately instantiated as GMT and rendered back in GMT with no conversion.

D

Third party acknowledgments

One or more products in the Progress Corticon v5.3.4 release includes third party components covered by licenses that require that the following documentation notices be provided:

Progress Corticon v5.3.4 incorporates Apache Commons Discovery v0.2 from The Apache Software Foundation. Such technology is subject to the following terms and conditions: The Apache Software License, Version 1.1 - Copyright (c) 1999-2001 The Apache Software Foundation. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The end-user documentation included with the redistribution, if any, must include the following acknowlegement: "This product includes software developed by the Apache Software Foundation (http://www.apache.org/)." Alternately, this acknowledgement may appear in the software itself, if and wherever such third-party acknowledgements normally appear.

4. The names "The Jakarta Project", "Commons", and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org.

5. Products derived from this software may not be called "Apache" nor may "Apache" appear in their names without prior written permission of the Apache Group.

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This software consists of voluntary contributions made by many individuals on behalf of the Apache Software Foundation. For more information on the Apache Software Foundation, please see <http://www.apache.org/>.

Progress Corticon v5.3.4 incorporates Apache SOAP v2.3.1 from The Apache Software Foundation. Such technology is subject to the following terms and conditions: The Apache Software License, Version 1.1 Copyright (c) 1999 The Apache Software Foundation. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Apache Software Foundation (http://www.apache.org/)." Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear. 4. The names "SOAP" and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org. 5. Products derived from this software may not be called "Apache", nor may "Apache" appear in their name, without prior written permission of the Apache Software Foundation. THIS SOFTWARE IS PROVIDED ``AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. This software consists of voluntary contributions made by many individuals on behalf of the Apache Software Foundation. For more information on the Apache Software Foundation, please see <http://www.apache.org/>.

Progress Corticon v5.3.4 incorporates DOM4J v1.6.1. Such technology is subject to the following terms and conditions: Project License BSD style license Copyright 2001-2005 (C) MetaStuff, Ltd. All Rights Reserved.

Redistribution and use of this software and associated documentation ("Software"), with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain copyright statements and notices. Redistributions must also contain a copy of this document.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The name "DOM4J" must not be used to endorse or promote products derived from this Software without prior written permission of MetaStuff, Ltd. For written permission, please contact dom4j-info@metastuff.com.

4. Products derived from this Software may not be called "DOM4J" nor may "DOM4J" appear in their names without prior written permission of MetaStuff, Ltd. DOM4J is a registered trademark of MetaStuff, Ltd.

5. Due credit should be given to the DOM4J Project - http://www.dom4j.org

THIS SOFTWARE IS PROVIDED BY METASTUFF, LTD. AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL METASTUFF, LTD. OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT,

INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Progress Corticon v5.3.4 incorporates Jaxen v1.0. Such technology is subject to the following terms and conditions: JAXEN License - $Id: LICENSE,v 1.3 2002/04/22 11:38:45 jstrachan Exp $ - Copyright (C) 2000-2002 bob mcwhirter and James Strachan. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution.

3. The name "Jaxen" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact license@jaxen.org.

4. Products derived from this software may not be called "Jaxen", nor may "Jaxen" appear in their name, without prior written permission from the Jaxen Project Management (pm@jaxen.org).

In addition, we request (but do not require) that you include in the end-user documentation provided with the redistribution and/or in the software itself an acknowledgement equivalent to the following: "This product includes software developed by the Jaxen Project (http://www.jaxen.org/)." Alternatively, the acknowledgment may be graphical using the logos available at http://www.jaxen.org/. THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE Jaxen AUTHORS OR THE PROJECT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. This software consists of voluntary contributions made by many individuals on behalf of the Jaxen Project and was originally created by bob mcwhirter <bob@werken.com> and James Strachan <jstrachan@apache.org>. For more information on the Jaxen Project, please see <http://www.jaxen.org/>.

Progress Corticon v5.3.4 incorporates JDOM v1.0 GA. Such technology is subject to the following terms and conditions: $Id: LICENSE.txt,v 1.11 2004/02/06 09:32:57 jhunter Exp $ - Copyright (C) 2000-2004 Jason Hunter and Brett McLaughlin. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution.

3. The name "JDOM" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact <request_AT_jdom_DOT_org>.

4. Products derived from this software may not be called "JDOM", nor may "JDOM" appear in their name, without prior written permission from the JDOM Project Management <request_AT_jdom_DOT_org>.

In addition, we request (but do not require) that you include in the end-user documentation provided with the redistribution and/or in the software itself an acknowledgement equivalent to the following: "This product includes software developed by the JDOM Project (http://www.jdom.org/)."

Alternatively, the acknowledgment may be graphical using the logos available at http://www.jdom.org/images/logos. THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE JDOM AUTHORS OR THE PROJECT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. This software consists of voluntary contributions made by many individuals on behalf of the JDOM Project and was originally created by Jason Hunter <jhunter_AT_jdom_DOT_org> and Brett McLaughlin <brett_AT_jdom_DOT_org>. For more information on the JDOM Project, please see <http://www.jdom.org/>.

Progress Corticon v5.3.4 incorporates Saxpath v1.0. Such technology is subject to the following terms and conditions: Copyright (C) 2000-2002 werken digital. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution.

3. The name "SAXPath" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact license@saxpath.org.

4. Products derived from this software may not be called "SAXPath", nor may "SAXPath" appear in their name, without prior written permission from the SAXPath Project Management (pm@saxpath.org).

In addition, we request (but do not require) that you include in the end-user documentation provided with the redistribution and/or in the software itself an acknowledgement equivalent to the following: "This product includes software developed by the SAXPath Project (http://www.saxpath.org/)." Alternatively, the acknowledgment may be graphical using the logos available at http://www.saxpath.org/ THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE SAXPath AUTHORS OR THE PROJECT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. This software consists of voluntary contributions made by many individuals on behalf of the SAXPath Project and was originally created by bob mcwhirter <bob@werken.com> and James Strachan <jstrachan@apache.org>. For more information on the SAXPath Project, please see <http://www.saxpath.org/>.