

## InRule Rule Authoring String Matching Examples

This document and supporting rule applications are intended to provide working examples of using the string functions and templates that are provided within irAuthor. It is important to note that the supporting execution patterns, schema definitions, and rule structures are only used to highlight the use of string functions and do not necessarily reflect best rule authoring practices.

### String Matching Examples

The rule application “1.StringMatchingExamples.ruleapp” provides basic examples of using the most common string functions. The rule application “2.OrderStringsExample.ruleapp” provides more advanced examples by implementing example order processing requirements that rely on the use of various string functions and templates. As there are often multiple approaches that can be taken by a rule author to solve string matching problems, this rule application will provide examples within Business Language (BL) as well as with Syntax expressions.

#### 1.StringMatchingExamples.RuleApp

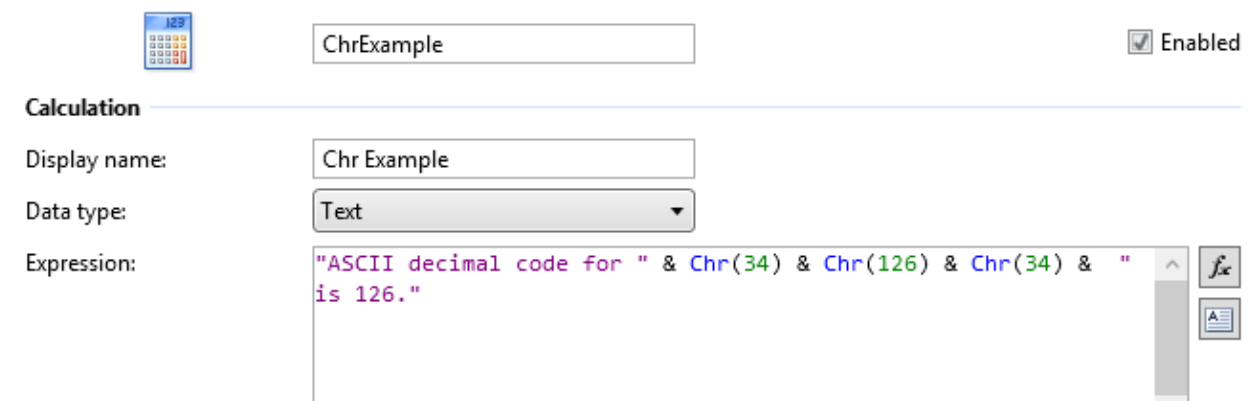
The “1.StringMatchingExamples.ruleapp” provides simple examples of commonly used string functions. Most of the examples work against the fields “TargetString” (with a default value of "abcdefghijklmnopqrstuvwxy") and “EmailString” (with a default value of "johnQ@test.nodomain.com") which can be altered within irVerify when testing.

#### Entity1 Calculations

The calculation fields within the “Entity1” Entity demonstrate some basic uses of commonly used string functions:

**LengthExample:** This field uses the Length() function to return the number of characters within the TargetString.

**ChrExample:** This field provides examples of concatenating strings using the ampersand “&” character, and using the Chr() function to insert special ASCII characters within a string:



The screenshot shows the InRule authoring interface for a calculation field named "ChrExample". At the top, there is a small icon of a calculator with the number 123, the field name "ChrExample" in a text box, and a checkbox labeled "Enabled" which is checked. Below this is a section titled "Calculation" with a horizontal line. Under "Calculation", there are three labels: "Display name:" with a text box containing "Chr Example", "Data type:" with a dropdown menu showing "Text", and "Expression:" with a large text area. The text area contains the following code: `"ASCII decimal code for " & Chr(34) & Chr(126) & Chr(34) & "` on the first line and `is 126."` on the second line. To the right of the text area are two icons: a function icon (f with an x) and a template icon (A with a list).

**LeftMidRightExample:** This field provides an example of using the Left, Mid, and Right functions to obtain the left 3, right 3, and middle 3 (starting from position 12) characters within the TargetString:



LeftMidRightExample

☒ Enabled

#### Calculation

Display name:

Left Mid Right Example

Data type:

Text

Expression:

```
((("Left3:" & Left(TargetString, 3)) & " Middle3:") & Mid  
(TargetString, 12, 3)) & " Right3:") & Right(TargetString, 3)
```

**ProperCaseExample:** This field provides an example of using the ProperCase function against the EmailString.

**RegexReplaceExample:** This field provides an example of using the RegexReplace function to replace the username and "@" symbol from the EmailString. The regex expression "\w+@" selects the beginning of the string through the "@" symbol:



RegexReplaceExample

☒ Enabled

#### Calculation

Display name:

Regex Replace Example

Data type:

Text

Expression:

```
RegexReplace(EmailString, "\w+@", "")
```

**ReplaceExample:** This field uses the Replace function to replace defined literal strings. In this example, the function is called in a nested fashion to remove both the periods "." as well as the at symbol "@" and replace them with the strings of " DOT " and " AT ", respectively:



ReplaceExample

☒ Enabled

#### Calculation

Display name:

Replace Example

Data type:

Text

Expression:

```
Replace(Replace(EmailString, ".", " DOT "), "@", " AT ")
```

**FormatExample:** This field uses the Format and today functions to provide the current date (using the .NET "D" format specifier):



FormatExample

☒ Enabled

#### Calculation

Display name:

Format Example

Data type:

Text

Expression:

"Today is: " & Format(Today(), "D") & "!"



**InStrExample:** This field uses the InStr function to display the position number of the "@" symbol within the EmailString.

#### Examples Rule Set

The Entity1.Examples rule set provides some simple examples of using the IsWildCardMatch and RegExMatch templates.

**IsWildCardMatchExamples:** This rule demonstrates the use of the "matches wild card expression" template. The asterisk "\*" represents any number of characters, while the question mark "?" represents any single character:



IsWildCardMatchExamples

#### Language Rule

##### Take the following actions:

If

Target String matches wild card expression "a\*" »

Then

fire notification 'FireNotification1'

[add action]

If

Target String matches wild card expression "a??d\*" »

Then

fire notification 'Fire\_Notification1'

[add action]

[add action]

#### RegExMatchExamples

The RegExMatchExamples rule demonstrates the use of the "matches regular expression" template. In the first condition expression, the regex expression "[a-z]{26}" will evaluate to true if there are 26 lower case letters within the string:

If

**Target String** matches regular expression "[a-z]{26}" with case-sensitivity »

Then

fire notification 'Fire\_Notification3'  
[add action]

The second condition expression provides a regex expression that matches a simplified concept of a valid email address. The expression “(\w+@[a-zA-Z\_]+?\.[a-zA-Z]{2,6})” will evaluate to true if the beginning of the string contains any letters before the “@” symbol, contains any number of characters before a dot “.” And contain between 2 and 6 characters after the first dot (note that this example is intended to display a very basic case and most email validations requirements will require additional pattern matching than demonstrated here):

If

**Email String** matches regular expression "(\w+@[a-zA-Z\_]+?\.[a-zA-Z]{2,6})" »

Then

fire notification 'Fire\_Notification2'  
[add action]  
[add action]

## 2.OrderStringsExample.RuleApp

### NormalizationRulesExample and NormalizationRulesSyntaxExample Rule Sets

The NormalizationRulesExample (and disabled syntax version “NormalizationRulesSyntaxExample”) rule set contain rules that process the “Name” and “PhoneNumber” fields and parse them into their atomic parts (FirstName, LastName, AreaCode, Prefix, and LineNumber).

A

NameRule

Enabled

Language Rule

If

get position of " " in Name is equal to the length of Name »

Then

set FirstName to Name  
[add action]

Else

set FirstName to the left get position of " " in Name characters in Name  
set LastName to the right the length of Name minus get position of " " in Name plus 1 characters  
in Name  
fire notification 'Fire\_Notification1'  
[add action]

The first Rule, “NameRule” uses the “get position of” template to identify the location of a space character within the name string. It also leverages the “the length of” template to identify cases where there is not a space in the string (the “get position of” template will return the length of the string if the match pattern is not found). If there is not a space in the name, we assign the Name to the FirstName field.

In the else alternative, we leverage the “the left of” and “the right of” to select a substring from the Name field. In the case of FirstName, we simply select the left number of characters with the number of characters being determined by the “get position of” template. In the case of LastName, because the “get position of” template will give the *left-based* position, we must derive the right-based position by subtracting the value from the length of the string and add 1 so that we do not select the space character. For example, the name “John Doe” will find the space at position 4 and assign the left 4 characters of “John Doe” to FirstName. For the LastName field, it will select the right 3 characters as

derived from the length of the string “John Doe” = 8, so  $8 - (4+1) = 3$ . Right clicking on a BL expression and selecting “Show Parenthesis” will show the order of operations for this BL expression.

The Second Rule, “PhoneNumberRule” has a top level length condition as well. However, it is matching the length of the substring generated using the “replace pattern” template. The “replace pattern” template uses a regular expression (regex) to generate a substring. In this particular example the regex expression “\D” to select all “non-digits” and replace them with an empty string (“”). The resultant substring based on the “PhoneNumberField” only then contains the numbers contained within the original string. In the first If/Then template we evaluate a 10 digit phone number and in the second If/Then template, we evaluate a 7 digit phone number:

**Language Rule**

**Take the following actions:**

**If**

the length of replace pattern “\D” in **Phone Number** with “” is equal to 10 »

**Then**

set **AreaCode** to the left 3 characters in replace pattern “\D” in **Phone Number** with “”

set **Prefix** to the middle 3 characters in replace pattern “\D” in **Phone Number** with “” starting with character 3

set **LineNumber** to the right 4 characters in replace pattern “\D” in **Phone Number** with “”

[add action]

**If**

the length of replace pattern “\D” in **Phone Number** with “” is equal to 7 »

**Then**

set **Prefix** to the left 3 characters in replace pattern “\D” in **Phone Number** with “”

set **LineNumber** to the right 4 characters in replace pattern “\D” in **Phone Number** with “”

[add action]

[add action]

Similar to the NameRule, the PhoneNumberRule leverages the “left” and “right” templates.

The first Prefix assignment leverages the “middle” template which requires specifying the number of characters to select along with the starting character position which in this case is provided as a literal value.

These rules can be tested within irVerify by testing in the context of Order. The entity schema defaults the Name field to “John Doe” and the PhoneNumber field to “(000)555-1234” but these values can be changed within irVerify as needed. It is also worth noting that the rule set “NormalizationRulesExample” can be disabled and the “NormalizationRulesSyntaxExample” rule set can be enabled to test the syntax versions of the above rules:

**Rules**

+ Add Rule Set Collapse All

Order

- NormalizationRulesExample
- NameRule
- PhoneNumberRule
- NormalizationRulesSyntaxExample**

**Entity Rule Set**

Activated by default: ☒

Fire mode: ☒ Auto ☐ Explicit

Run mode: ☐ Optimized ☐ Sequential ☒ Single-Pass Sequential

Go to the entity to change settings, add/remove fields, etc.

[Go to entity](#)

## CreditCardValidationRulesExample Rule Set Overview

The “CreditCardValidationRulesExample” rule set (and logically equivalent “CCVR\_FirstPassBLExample” and “CCVR\_SyntaxExample” rule sets) contain rules that identify and validate an Order’s CreditCardNumber.

The rules within the “CreditCardValidationRulesExample” leverage the use of custom vocabulary templates and classifications. By using vocabulary templates and classifications, we can concisely document the business concept without the need to review the low-level expression logic (this however can be seen by right clicking on an expression and selecting “Navigate-To”). Once we review the concepts of the rules within this rule set, we will review the rules within the “CCVR\_FirstPassBLExample” rule set which use the low-level expression logic directly.

The credit card validation rules are contained within two rules: “CardTypeDetermination” and “LuhnValidationRules”. The card type determination rule interrogates the credit card number and uses the rules regarding account prefixes and length to determine the credit card type. If a known type is not identified, the credit card is set to invalid. The Luhn validation rule uses a standard credit card validation algorithm used to determine if the credit card account is in the set of possible valid credit cards. This process is defined as:

1. Assign the rightmost digit of the credit card number as the check digit.
2. The remaining numbers are considered the credit card account number.

Account Number														Check Digit
5	1	0	5		1	0	5	1		0	5	1	0	0

3. From the rightmost digit, we then double every other value:

Account Number														Check Digit
5	1	0	5		1	0	5	1		0	5	1	0	0
x2		x2			x2		x2			x2		x2		
10	1	0	5		2	0	10	1		0	5	2	0	

4. And for any digit greater than 9, we sum the two digits together. For example, 10 = 1+0, or 11 = 1+1:

Account Number														Check Digit
5	1	0	5		1	0	5	1		0	5	1	0	0
x2		x2			x2		x2			x2		x2		
10	1	0	5		2	0	10	1		0	5	2	0	
1	1	0	5		3	0	1	1		0	5	2	0	

5. We then sum the values and multiply the result by 9:

Account Number													Check Digit
5	1	0	5		1	0	5	1		0	5	1	0
x2		x2			x2		x2			x2		x2	
10	1	0	5		2	0	10	1		0	5	2	0
1	1	0	5		2	0	1	1		0	5	2	0
													Sum
													20
													Result
													180

- The rightmost digit of the result represents the Luhn check digit. If the Luhn check digit matches the credit card number check digit, the credit card number is a valid credit card number. In the example above  $0 = 0$ , so the number is valid.

The rule “LuhnValidation” follows this process directly and leverages vocabulary templates to convey this process directly:



LuhnValidation

Language Rule

Take the following actions:

set **creditCard Check Digit** to the rightmost character in **Credit Card Number**

set **creditCard Account Number** to the remaining characters less the CheckDigit in **Credit Card Number**

While

**creditCard Current Position From Right** is not at the end of **creditCard Account Number** »

Do the following:

set **creditCard Current Position Value** to the value in **creditCard Account Number** at the **creditCard Current Position From Right**

Provide variable state information

If

**creditCard Current Position From Right** is odd »

Then

double the **creditCard Current Position Value**

If

**creditCard.CurrentPositionValue** > 9 »

Then

Sum the digits in **creditCard Current Position Value**

[add action]

[add action]

set **luhn Sum** to **luhn.Sum** + **creditCard.CurrentPositionValue**

Increment **creditCard Current Position From Right** by one

[add action]

set **luhn Sum** to **luhn Sum** multiplied by 9

set **luhn Check Digit** to the rightmost character in **luhn Sum** ...

If

**luhn Check Digit** is equal to **creditCard Check Digit** »

Then

set **Valid CC** to true

[add action]

Else

set **Valid CC** to false

[add action]

[add action]

## CreditCardValidationRulesExample Rule Set Details

The disabled rule sets “CCVR\_FirstPassBExample” and “CCVR\_SyntaxExample” are the logical equivalent to the “CreditCardValidationRulesExample” rule set. They represent the equivalent business language and syntax expressions that accomplish the same results. They also represent a common 2-pass approach to authoring business rules in that the “first pass” focuses on creating expressions that solve the business problem. A second pass (represented by “CreditCardValidationRulesExample”) is then taken to more directly convey the business intent by encapsulating the expression logic in classifications and vocabulary templates. We will focus on the rules within the business language example “CCVR\_FirstPassBExample”.

In the CardTypeDetermination rule, we leverage the “matches regular expression” template to use a regex expression on the credit card number to determine the credit card type:

- American Express cards must begin with 34 or 37 and are 15 digits long. This is represented by the regular expression “3[47][0-9]{13}” where “3[47]” matches 34 or 37, and “[0-9]{13}” matches the remaining 13 digits totaling 15.
- Visa cards begin with a 4 and are either 16 or 13 digits long. This is represented by the regular expression “^4[0-9]{12}(:[0-9]{3})?\$” where “^4” represents the start of the line = 4, “[0-9]{12}” represents any numbers for the remaining 12 digits (to total 13) and “(:[0-9]{3})?\$” represents an optional 3 numbers at the end of the string (to total 16).
- Discover cards begin with either 6011 or 65 and are 16 digits long. This is represented by the regular expression “6(?:011|5[0-9]{2})[0-9]{12}” where “(?:011|5[0-9]{2})” represents either “011” or any 4 digit number starting with 65. “[0-9]{12}” represents the remaining 12 digits (to total 16).
- MasterCard cards begin with either 51, 52, 53, 54, or 55 and are 15 digits long. This is represented by the regular expression “5[1-5][0-9]{14}” where “5[1-5]” represents the range 51-54.


The LuhnValidationRule then provides examples of:

- Using the “the Right” template to select the right 1 most character from the Credit Card Number:  
**set creditCard Check Digit to the right 1 characters in Credit Card Number**
- Using the “the Left” template to select the remaining digits by calculating the length of the string using the “Length” function:  
**set creditCard Account Number to Left(CreditCardNumber, Length(CreditCardNumber) - 1)**
- Using the “the Length” template to determine the length of the creditCard.AccountNumber:  
**While**  
**creditCard Current Position From Right is less or equal to the length of creditCard Account Number »**
- Using the “the Middle” template (along with nested length and minus templates) to determine the current digit value:  
**set creditCard Current Position Value to the middle 1 characters in creditCard Account Number starting with character the length of creditCard Account Number minus creditCard Current Position From Right**
- Using the “Left” and “Right” syntax functions to sum 2 digits within a string:



set **creditCard Current Position Value** to `Left(creditCard.CurrentPositionValue, 1) + Right(creditCard.CurrentPositionValue, 1)`

The version of the LuhnValidationRule that does not leverage templates is shown below:



LuhnValidation

☒ Enabled

---

Language Rule

**Take the following actions:**

- set **creditCard Check Digit** to the right **1** characters in **Credit Card Number**
- set **creditCard Account Number** to `Left(CreditCardNumber, Length(CreditCardNumber) - 1)`

**While**  
**creditCard Current Position From Right** is less or equal to the length of **creditCard Account Number** »

**Do the following:**

- set **creditCard Current Position Value** to the middle **1** characters in **creditCard Account Number** starting with character the length of **creditCard Account Number** minus **creditCard Current Position From Right**
- fire notification 'Fire\_VariableStateNotification'

**If**  
`Mod(creditCard.CurrentPositionFromRight, 2) > 0` »

**Then**

- set **creditCard Current Position Value** to `creditCard.CurrentPositionValue * 2`

**If**  
`creditCard.CurrentPositionValue > 9` »

**Then**

- set **creditCard Current Position Value** to `Left(creditCard.CurrentPositionValue, 1) + Right(creditCard.CurrentPositionValue, 1)`

[add action]

[add action]

- set **luhn Sum** to `luhn.Sum + creditCard.CurrentPositionValue`
- set **creditCard Current Position From Right** to `creditCard.CurrentPositionFromRight + 1`

[add action]

- set **luhn Sum** to **luhn Sum** multiplied by **9**
- set **luhn Check Digit** to the right **1** characters in **luhn Sum ...**

**If**  
**luhn Check Digit** is equal to **creditCard Check Digit** »

**Then**

- set **Valid CC** to true

[add action]

**Else**

- set **Valid CC** to false

[add action]

[add action]