Ausdrücke in Prüfregel

Zugehörige Informationen			
Abfragecode und Ausdrücke, **Ausdrücke in Prüfregeln**, Ausdrücke in Skripten, Reguläre Ausdrücke	Allgemeine Bedienungshinweise	—	—
Prüfregeln			

Erstellen von Regeln

Diese Seite informiert über die Möglichkeiten der Formulierung von Prüfregelcode. Auf die Seite zur Bedienung der Administrator-Maske zur Definition einer Prüfregel und zur Defintion eines Prüfplans sei von hier nur verwiesen.

Regeln werden in **JavaScript** erstellt. Der Regelcode wird zur Laufzeit von ASYS interpretiert, wobei zuvor Platzhalter für Datensatz- oder Nachrichtenfeldinhalte von der ASYS-Mittelschicht durch die jeweiligen Werte ersetzt werden. Innerhalb einer Regel können eine oder mehrere Abfragen bzw. ihre Ergebnisattribute verwendet werden, wobei ebenfalls eine Platzhalterersetzung durch die ASYS-Mittelschicht erfolgt. Der nach dieser Ersetzung ausgeführte Code muss immer **true** (WAHR) oder **false** (FALSCH) zurück liefern (bzw. das Ergebnis wird entsprechend als true oder false weiterverarbeitet).

Objekte, die dem Skriptinterpreter übergeben werden

Prinzipiell können alle Methoden und Funktionen, die JavaScript bietet, für Regeln verwendet werden. Damit allerdings nicht für jede Regel ein umfangreiches Skript geschrieben werden muss, gibt es einige Vereinfachungen sowie einige vordefinierte Funktionen, die in der Mittelschicht abgelegt sind und direkt verwendet werden können. Um diese Funktionalitäten nutzen zu können, werden dem Skriptinterpreter drei Objekte übergeben.

dc (Data Context)

Der **dc** (Data Context) ist das Hauptobjekt für den Regelmechanismus. Er bietet alle Methoden/Funktionen, Attribute und Daten für den Zugriff auf die aktuellen Daten einer Maske oder einer Nachricht. Die vereinfachte Schreibweise für den Zugriff auf die Daten (**{%Klasse.Attribut%}**) wird vor der Ausführung intern in einen **dc**-Aufruf umgewandelt (bei Textfeldern z.B. in **dc.getStringValue('Klasse.Attribut')**). Relevant für den direkten Aufruf einer dc-Funktion

innerhalb einer Regel ist in erster Linie die Methode **dc.isNullValue('Klasse.Attribut')**, über die festgestellt werden kann, ob ein Feld einen Inhalt besitzt.

Bei Masken (Aufgabenbereichen) enthält der **dc** die Daten der Datendarstellungsdefinition des gerade aktuell auf der Maske (Aufgabenbereichs) angezeigten Datensatzes (nicht aber die Daten von ggf. auf der Maske ebenfalls angezeigten abhängigen Datensätzen).

Bei Nachrichten enthält der **dc** alle Daten einer einzelnen Nachricht, die in einer 1:1-Beziehung vorliegen. Um welche Daten es sich hierbei handelt wird über den Kommunikationsbaum festgelegt.

sc (Session Context)

Der **sc** (Session Context) stellt einige Funktionen zur Verfügung, über die insbesondere Angaben über den aktuellen Repository-Standort ermittelt werden können.

dco (Data Context Objecttree)

Das **dco**-Objekt ist nur für Nachrichten relevant. Es enthält den gesamten Dateninhalt einer einzelnen Nachricht mit allen 1:n:m-Beziehungen. Aufgrund der Komplexität dieses Objektes wird auf eine weitergehende Beschreibung verzichtet.

Rückgabewerte true und false und das Ankreuzfeld Invers

Eine Prüfregel ist so zu definieren, dass sie **true** zurückliefert, wenn die Bedingung erfüllt ist und **false** (also einen Fehler), wenn die Bedingung nicht erfüllt ist. Manchmal ist es jedoch einfacher, zu prüfen, ob eine Bedingung nicht erfüllt ist oder der Sprachgebrauch legt eine Umkehrung der Prüfung nahe. Aus diesem Grund kann eine Prüfregel auf inverse Logik gesetzt werden (s. Prüfregeln).

Beispiel: Die Frage 'Liegt das Eingangsdatum in der Zukunft?' hat als Antwort Ja, wenn das Datum in der Zukunft liegt. Das wäre in der BGS-Prüfung aber der Fehlerfall, der nur durch den Rückgabewert **false** sichtbar werden würde. Würde ein Regelcode vereinfacht formuliert also lauten 'Eingangsdatum > heute', muss das Feld inverse Logik angekreuzt werden, damit **false** (also ein Fehler) gemeldet wird, falls das Eingangsdatum tatsächlich in der Zukunft liegt.

Konventionen für die Verwendung von Attributen, Abfragen und 'is null' Prüfungen

Alle Attribute des aktuellen Datenkontextes können in den Prüfungen verwendet werden. Die Definitionsmaske des Administrators bietet diese Attribute zur Auswahl an. Folgende Konventionen sind dabei zu beachten:

Verwendung von Attributen

```
{%BGS UNS.Abfallschluessel%}
```

Ein Attribut muss in geschweifte Klammern und Prozentzeichen eingefasst sein **{%...%}**. Zuerst wird der Klassenname und - durch einen Punkt verbunden - anschließend der Name des Attributes angegeben. Die Schreibweise des Tabellennamens und des Attributes muss exakt stimmen (also BGS UNS statt bgsuns oder Bgs Uns etc.). Äquivalent könnte auch die Schreibweise **dc.getStringValue(„BGS UNS.Abfallschluessel“)** verwendet werden. Hierbei ist der korrekte Datentyp zu beachten.

Gezielte Abfrage auf NULL

Am sinnvollsten wird ein Feld mit der Methode **dc.isNullValue(„Klasse.Attribut“)** daraufhin überprüft, ob ein Inhalt in diesem Feld vorhanden ist. Prinzipiell ist es jedoch auch möglich, die geschweifte Klammer/Prozent – Notation zu verwenden

```
{%Klasse.Attribut%}==null
```

Regelergebnis false, wenn eines der verwendeten Attribute NULL ist

Sofern für eine Regel nicht das Ankreuzfeld keine automatische Vorprüfung der Feldinhalte auf NULL gesetzt ist, wird vor der Ausführung der eigentlichen Regel jedes Feld, das in der Regel angegeben ist, auf einen NULL-Wert überprüft. Wenn einer der Feldinhalte NULL ist, liefert die Regel **false** zurück, ohne dass der Regelcode ausgeführt wurde! Verhindert werden soll mit diesem Mechanismus, dass Regeln unsinnige Werte zurückgeben.

Die hier erwähnte **Vorprüfung auf NULL** wird nur in Prüfregeln ausgeführt!

In Skripten (z.B. für die Vorgangssteuerung), für welche die auf dieser Seite erläuterten Regeln im Grundsatz auch gelten, wird eine solche Vorprüfung nicht ausgeführt, da Skripte als Ergebnis nicht nur *true* und *false* haben können, sondern beliebige Ergebniswerte (z.B. Zahlen oder Zeichenketten).

Verwendung von Regelabfragen

```
{%*IKA STD BGS AVV gueltig.trefferzahl%} > 0
```

Wird eine Abfrage verwendet, muss nach dem ersten Prozentzeichen ein * folgen - also **{%*...%}**. Dann folgt der Name der Abfrage, ein Punkt und dann der Name des Rückgabewertes der Abfrage (einer der Einträge im Abschnitt RESULTS der Abfrage). Der Ausdruck **{%*IKA STD BGS AVV gueltig.trefferzahl%} > 0** bewirkt, dass der Inhalt der Abfrage in einen SQL-Befehl an die Datenbank umgewandelt wird, der unter dem Spaltenaliasnamen **trefferzahl** einen numerischen Wert zurück liefert. Auch hier gilt, dass die Schreibweise exakt stimmen muss.

Aufbau des Regelcodes

Die folgende Tabelle zeigt die allgemeine Syntax für den Aufbau einer Regel und einige häufig verwendete **JavaScript**-Funktionen.

Syntax	Erläuterung
(**Regel**)	Klammern dienen der Gliederung einer komplexen Regel, damit eindeutig ist, welche Regelbestandteile zusammengehören bzw. getrennt ausgewertet werden sollen.
{%KLASSE.ATTRIBUT%}	Liefert den Feldinhalt eines Attributes.
dc.isNullValue(„KLASSE.ATTRIBUT“)	Überprüft den Feldinhalt eines Attributes auf null; liefert true, wenn der Inhalt null ist und false, wenn er nicht null ist. Bitte beachten Sie die Anführungsstriche um KLASSE.ATTRIBUT, sie müssen mit in den Code geschrieben werden!
{%*ABFRAGE.ATTRIBUT%}	Liefert den Feldinhalt eines Attributes einer Abfrage.
Feld==**Wert**	Vergleicht die rechts und links stehenden Ausdrücke bei Zahlen und einem Zeichen (Character, z.B. von charAt() s.u.); zum Vergleich von Zeichenketten, auch, wenn sie nur ein Zeichen lang sind muss .equals (s.u.) verwendet werden.
Feld <, >, >=, <= **Wert**	Operatoren für Zahlenvergleiche.
Regel 1 && **Regel 2** (und)	Und (and)–Verknüpfung mehrerer Regeln.
Regel 1 || **Regel 2** (oder)	Oder (or)–Verknüpfung mehrerer Regeln.
! (nicht) **Regel**	Negiert den Ausdruck.
Wenn ? **Dann** : **Sonst**	Wenn … dann Regel; liefert der **Wenn**-Ausdruck vor dem Fragezeichen **true**, wird der **Dann**-Ausdruck vor dem Doppelpunkt ausgeführt, bei **false** wird der **Sonst**-Ausdruck nach dem Doppelpunkt ausgeführt. Wird in den meisten Fällen mit Klammerausdrücken verwendet und kann dann auch geschachtelt werden, z.B. `( (Wenn 1) ? ( (Wenn 2) ? (Dann 2) : (Sonst 2) ) : (Sonst 1) )` Der **Dann 1**-Teil ist wieder eine eigene Wenn … dann Regel; **Dann 2** wird nur ausgeführt, wenn sowohl **Wenn 1** als auch **Wenn 2 true** ergeben.
Feld.charAt(**Zahl**)	Die Methode charAt() liefert das Zeichen (als Charakter) an der übergebenen Zahlenposition; 0 indiziert, d.h. charAt(0) liefert das 1. Zeichen.
Feld/Wert.equals(**Wert/Feld**)	Die Methode equals() vergleicht den Inhalt zweier Strings.
Feld/Wert.equalsIgnoreCase(**Wert/Feld**)	Die Methode equalsIgnoreCase() vergleicht den Inhalt zweier Strings, wobei die Groß- und Kleinsschreibung unberücksichtigt bleibt.
Feld/Wert.indexOf(**Wert/Feld**)	Die Methode indexOf() liefert die Position des Strings in der Klammer in dem vor der Funktion genannten String.
Feld.startsWith(**Wert**)	Die Methode startsWith() stellt fest, ob der String vorne mit dem in der Klammer genannten beginnt.
Feld.endsWith(**Wert**)	Die Methode endsWith() stellt fest, ob der String vorne mit dem in der Klammer genannten endet.

Syntax	Erläuterung
Feld.length	Die Methode length liefert die Länge eines Strings.
Feld.substr(**Anfang**, **Länge**)	Die Methode substr() liefert einen String; in der Klammer werden die Anfangsposition (beginnend bei 0) und die Anzahl der Zeichen angegeben.
Feld.substring(**Start**, **Ende**)	Die Methode substring() liefert einen String; in der Klammer werden die Startposition (beginnend bei 0) und die Position nach dem letzten gewünschten Zeichen angegeben.
Feld.valueOf(**Wert**)	Die Methode valueOf() liefert einen String; in der Klammer können beliebige Typen von Werten oder Feldinhalten übergeben werden (Zahlen, boolesche Werte...)

Erläuterungen zu den Parametern in der vorstehenden Tabelle:

- **Regel 1, Regel 2, Wenn, Dann, Sonst**: In sich vollständige (Teil-)Regeln, die **true** oder **false** liefern.
- **Wert**: Ein fester Wert (z.B. 1 oder „EN“) oder das Ergebnis einer Regel.
- **Feld**: Meistens ein Attribut aus dem 'Data Context' oder einer Abfrage (z.B. {%BGS UNS.BGS UNS Nummer%} oder {%*IKA STD BGS vorhanden.trefferzahl%})

Allgemeine Hinweise zur Definition und Ausführung:

Feste **Strings** müssen immer in doppelten Anführungsstrichen stehen. Bei Stringvergleichen kann es sich anbieten, den festen String an den Anfang einer Regel zu stellen, um Fehler aufgrund von NULL-Werten zu vermeiden.

Beispiel:

```
"eANV / XML".equals({%BGS UNS.Datenursprung Typ%})
```

und

```
{%BGS UNS.Datenursprung Typ%}.equals("eANV / XML")
```

liefern im Prinzip das gleiche Ergebnis. Wenn jedoch der Feldinhalt von 'Datenursprung Typ' NULL ist, kommt es in der zweiten Variante zu einer 'NullPointerException', da auf einem NULL-Wert keine Funktionen ausgeführt werden können. Die erste Variante liefert dagegen einfach ein korrektes **false**, denn „IrgendeineZeichenkette“.equals(null) ist erlaubt und kann ausgeführt werden.

Feste **Charakter**-Zeichen müssen immer in einfachen Anführungszeichen stehen.

Bei **&&**-Verknüpfungen von Regeln werden die einzelnen Teilregeln nacheinander abgearbeitet und der Skriptinterpreter bricht ab, wenn der logische Ausdruck feststeht.

Beispiel:

```
!dc.isNullValue("Rolle Befoerderer.Behoerdliche Nummer") &&
eq({%Rolle Befoerderer.Behoerdliche Nummer%},sc.blKennung())
```

Wenn die behördliche Nummer des Beförderers NULL ist, liefert der erste Teil der Regel **false**. Damit kann der gesamte logische Ausdruck nur noch **false** liefern und der zweite Teil der Regel wird nicht mehr ausgeführt.

Codeblock

Meist besteht der Regelcode nur aus einer Zeile (der ggf. im Fenster für den Regelcode automatisch umgebrochen wird). Es besteht aber auch die Möglichkeit, mehrzeiligen Code zu schreiben, um z.B. komplexere Fallunterscheidungen vorzusehen. Hierfür muss der Code als **Codeblock** gekennzeichnet werden:

```
[
mehrzeiliger
Codeblock
]
```

Der Codeblock wird durch ein Paar eckiger Klammern [ ] gekennzeichnet. Dieses Klammerpaar muss in der ersten und letzten Zeile des Regelcodes stehen. **Nach der schließenden Klammer darf kein weiteres Zeichen folgen, auch kein Zeilenumbruch!**

Innerhalb des Codeblocks können Variablen definiert und JavaScript-Sprachelemente verwendet werden.

Beispiel[1)2)]:

```
[
<-Der Beginn des Codeblocks.
var nummer = {%Verantwortliche Erklaerung.Erzeugernummer%};
<-Deklaration der Variablen 'nummer' und Zuweisung des Inhalts aus dem
aktuellen Datensatz.
var ergeb = false;
<-Deklaration der Variablen 'ergeb' und Initialisierung mit einem
feststehenden Wert.

if (nummer == null)
<-Prüfung, ob 'nummer' undefiniert ist.
  {
  ergeb = false;
<-Zuweisung eines feststehenden Wertes, wenn 'nummer' undefiniert ist.
  }
else
<-Codeteil der ausgeführt wird, wenn 'nummer' nicht undefiniert ist.
  {
   if (nummer.equals("A12345678") ||
<-Prüfung, ob die 'nummer' einen bestimmten Inhalt besitzt - liefert eine
```

```
Fehlermeldung, wenn 'nummer' undefiniert ist.
       nummer.equals("A23456789") ||
<-weitere Prüfung gegen einen anderen Inhalt, die einzelnen Prüfungen sind
per ODER verknüpft -> || = ODER; && = UND.
       ...
       nummer.equals("A98765432")
      )
      {
      ergeb = true;
<- Zuweisung eines feststehenden Wertes, wenn eine der vorhergehenden
Prüfungen ein positives Ergebnis erbracht hat.
      }
   }

return ergeb;
<- Rückgabe des Inhalts der Variablen 'ergeb' als Ergebnis des Regel-
/Scriptcodes.
]
<- Ende des Codeblocks. Bitte beachten: Hier darf kein weiteres Zeichen oder
ein Zeilenumbruch folgen.
```

JavaScript bietet weitere Sprachkonstrukte, die hier aber nicht ausgeführt werden können. Eine erste Übersicht findet sich bereits im **Wikipedia-Artikel zu JavaScript**. Weitergehende Einführungen und Referenzen finden sich z.B. unter **JavaScript Tutorial (englisch)** und **JavaScript and HTML DOM Reference (englisch)**[3].

Wichtig: ASYS ist keine Web-Anwendung. JavaScript ist aber primär ein Werkzeug bei der Entwicklung von Web-Anwendungen, die in Internet-Browsern laufen. Daher kann keine Gewähr übernommen werden, dass der in ASYS zum Einsatz kommende Javascript-Interpreter alle in den vorstehend verlinkten externen Quellen genannten Sprachkonstrukte unterstützt bzw. dass ihre Nutzung in ASYS in gleicher Weise möglich ist, wie in einer Web-Anwendung.

Spezielle Session-Context-Funktionen (sc) und Data-Context-Funktionen (dc), die bei der Definition von Regeln und Skripten verwendet werden können

Session-Context-Funktionen lassen sich mittels vorangestelltem **sc.** erreichen. Nach Eingabe von **sc.** erscheint automatisch eine Liste aller bereitstehender Funktionen jeweils mit einem Erläuterungstext. Die Liste kann auch mit **Strg+Leer** aufgerufen werden.

Für Data-Context-Funktionen gilt entsprechendes, nur das statt sc. ein vorangestelltes **dc.** einzugeben ist.

Vordefinierte ASYS-Funktionen, die bei der Definition von Regeln und Skripten verwendet werden können

Eine Reihe von vordefinierten Funktionen können in Prüfregeln und Skripten eingesetzt werden. Eine Übersicht über diese Funktionen kann über die kontextsensitive Hilfe mittels **Strg+Leer** aufgerufen werden. Jeder Funktion ist ein kurzer Erläuterungstext - meist mit Beispielen - beigestellt.

Weitere Informationen zu diesem Thema																
Prüfregeln																
landesspezifische Zusatzinformationen:	SH	HH	NI	HB	NW	HE	RP	BW	BY	SL	BE	MV	ST	BB	TH	SN

[1)]

Das Beispiel stammt aus einem VG-Skript, ist aber auch für Prüfregelcode gültig, da als Ergebnis *true* oder *false* zurückgegeben werden.

[2)]

Die mit '←' beginnenden Textteile sind Erläuterungen für dieses Beispiel und **kein** Bestandteil des Regel-/Scriptcodes!

[3)]

Beide zuletzt aufgerufen unter dieser Adresse: November 2022