ISA Dialog Manager

REGELSPRACHE

A.06.03.b

Dieses Handbuch erläutert die Regelsprache des ISA Dialog Managers in der das dynamische Verhalten der Benutzeroberfläche programmiert werden kann. Im Handbuch sind beispielsweise Ereignis- und Regelverarbeitung, Datentypen, Sprachumfang der Regelsprache, Syntax von Anweisungen und eingebaute Funktionen beschrieben.

ISA Informationssysteme GmbH

Meisenweg 33

70771 Leinfelden-Echterdingen

Deutschland

Darstellungskonventionen

DM wird in diesem Handbuch synonym zu "Dialog Manager" verwendet.

Die Bezeichnung UNIX schließt generell alle unterstützten UNIX-Derivate ein - außer in den explizit angegebenen Fällen.

Dort wo für geläufige englische Fachbegriffe keine gängigen deutschen Übersetzungen existieren, wird zur Vermeidung von Unklarheiten der englische Begriff verwendet.

< >	muss durch einen entsprechenden Wert ersetzt werden
color	Schlüsselwort ("keyword")
.bgc	Attribut
{ }	optional (0 oder einmal)
[ ]	optional (0 oder n-mal)
<A> | <B>	entweder <A> oder <B>

Beschreibungsmodus

Alle Schlüsselwörter sind fett und unterstrichen, z.B.

variable **integer** **function**

Indizierung von Attributen

Syntax für indizierte Attribute:

[I]

[I,J] bzw. [row,column]

Identifikatoren

Identifikatoren müssen mit einem Großbuchstaben oder einem "Unterstrich" ('_') beginnen. Die weiteren Zeichen können Groß-, Kleinbuchstaben, Zahlen oder Unterstriche sein.

Der Bindestrich ('-') ist für die Benennung von Identifikatoren als Zeichen ***nicht*** zugelassen!

Die maximale Länge eines Identifikators beträgt *31* Zeichen.

Beschreibung der zugelassenen Identifikatoren in Backus-Naur-Form

<Identifikator> ::= <erstes Zeichen>{<Zeichen>}

<erstes Zeichen> ::= _ | <Großbuchstabe>

<Zeichen> ::= _ | <Kleinbuchstabe> | <Großbuchstabe> | <Ziffer>

<Ziffer> ::= 1 | 2 | 3 | … 9 | 0

<Kleinbuchstabe> ::= a | b | c | … x | y | z

<Großbuchstabe> ::= A | B | C | … X | Y | Z

Inhalt

1 Einleitung

Das Verhalten der erzeugten Dialogobjekte wird in der DM-Regelbasis beschrieben.

Eine Regel besteht aus

- dem sogenannten Ereignis und
- dem Aktionsteil.

Grundsätzlich bestehen die DM-Regeln aus der sogenannten "Ereigniszeile" und einem "Programmblock" (in geschweiften Klammern), der ausgeführt wird, sobald das in der Ereigniszeile beschriebene Ereignis auftaucht.

Die Regelabarbeitung wird durch ein Ereignis ausgelöst, das direkt oder indirekt vom Benutzer initiiert wird. Dieses Ereignis wird in der Ereigniszeile beschrieben.

Ein Ereignis ist eine Aktion des Benutzers oder des DM, das sich auf ein DM-Objekt oder dessen Attribute bezieht (z.B. Fenster verschieben und vergrößern, Buttons anklicken, Text eingeben, usw.).

Es gibt folgende Ereignistypen:

- Benutzerereignisse,
- Tastatur- und Hilfe-Ereignisse,
- interne Ereignisse,
- Objektklassenereignisse,
- Systemereignisse,
- externe Ereignisse.

Die Funktionalität der Regeln wird durch den der Ereigniszeile folgenden, in geschweiften Klammern stehenden Programmblock beschrieben. Durch diese Regelkomponente wird erreicht, dass der gesamte Dialogablauf einschließlich seiner Funktionalität durch den DM beschrieben werden kann.

Aktionen können z.B. sein:

- Ändern von Attributen bzw. Attributwerten,
- Aufruf von Anwendungsfunktionen,
- Aufruf von Builtin-Funktionen.

```
<Ereigniszeile>
{
  <Programmblock>
}
```

Mit dieser Definition wird der Programmblock immer dann ausgeführt, wenn das entsprechende Ereignis auftaucht.

Vorbedingung

Mit der folgenden Definition wird der Programmblock nur ausgeführt, wenn eine zusätzliche Vorbedingung erfüllt ist.

Das Schlüsselwort ("keyword") für diese zusätzliche Vorbedingung lautet **if**.

Der Programmblock besteht aus einer Abfolge von Zuweisungen und Statements, die die zusätzliche Programmausführung betreffen.

```
<Ereigniszeile>
if( <Zusatzbedingung> )
{
  <Programmblock>
}
```

Es gibt noch weitere Kontrollstrukturen, mit denen der Programmfluss beeinflusst werden kann, z.B. mögliche Sub-Regeln "if then else endif"

Anmerkung

Die Vorbedingung ("if") hat im Prinzip die gleiche Struktur wie das "if" im konditionierten Programmfluss. Es gibt jedoch einen Unterschied:

Das Vorbedingungs-"if" wird vor jeder Regelausführung abgeprüft, wobei das konditionierte "if" nur geprüft wird, wenn die Regel ausgeführt wird.

Daher ist die Regelabfolge bei dem Vorbedingungs-"if" nicht relevant, da eine Regel nicht eine andere Regel blockieren kann, wenn die Vorbedingung erfüllt ist.

2 Ereignisse

Die Ereigniszeile wird durch das Schlüsselwort **on** eingeleitet.

Jede Regel darf dabei maximal eine Ereigniszeile haben, die aber aus mehreren Ereignissen für ein Objekt bestehen kann.

Die regelauslösenden Ereignisse lassen sich in die folgenden Gruppen einteilen:

- Benutzerereignisse,
- Tastatur- und Hilfe-Ereignisse,
- interne Ereignisse,
- Systemereignisse,
- externe Ereignisse.

2.1 Benutzerereignisse

Ereignisse, die sich auf ein bestimmtes Objekt, eine Vorlage oder eine Objektklasse beziehen, werden Benutzerereignisse genannt. Für diese Art von Ereignissen ist entscheidend, dass bei allen der Benutzer als Ursache für das Auftreten des Ereignisses verantwortlich ist. Immer wurden Aktionen vom Benutzer ausgeführt, die dazu geführt haben, dass das entsprechende Ereignis ausgelöst worden ist.

Die Syntax für diese Art von Ereigniszeile sieht wie folgt aus:

```
on <Objekt-ID> <Ereignis> { , <Ereignis> }
```

Wie man dabei dieser Definition entnehmen kann, kann eine Regel an nur ein Objekt, aber an mehrere durch Kommata getrennte Ereignisse gebunden sein.

Objekt-IDs sind die Namen für bestehende DM-Objekte (Identifikatoren) wie z.B. „OK_Knopf“, „TestFenster“ usw.).

Ereignistypen beschreiben benutzerinitiierte Ereignisse wie „selektiert“, „bewegt“, „geschlossen“.

Beispiel

```
on Pushbutton select
{
  Aktion;
}
```

Bei Clipboard-Aktionen werden die entsprechenden Events an das Objekt geschickt. Eine Drag & Drop-Operation löst beim Quellobjekt beim Cut einen Cut-Event aus und am Zielobjekt einen Paste-Event. Da bei einem Objekt bei Copy nichts verändert wird, gibt es keinen Copy-Event.

Achtung: Die Reihenfolge der Events ist willkürlich!

Beim Cut wird vom DM aus keine Löschaktion vorgenommen. Die Reaktion muss auf den Cut-Event hin erzeugt werden. Die Attribute thisevent.type und thisevent.value sind nicht belegt.

Die Paste-Reaktion kann mit diesem Event ausgelöst werden.

thisevent.type enthält einen type_enum und zeigt das angenommene Format an.

thisevent.value enthält die Daten (DM_Datentyp).

Bei der Bearbeitung sollte man darauf achten, dass .cut_pending = true ist, wenn Quell- und Zielobjekt bei einer Move-Operation (Cut + Paste) **das gleiche** sind.

Die Beispiele hierzu befinden sich im Verzeichnis ...\examples\dragdrop.

Folgende Ereignistypen stellt der DM zur Verfügung:

Ereignistyp	Beschreibung
activate	Das angegebene Objekt wurde aktiviert. Dieses Ereignis kann für alle Objekte ausgelöst werden, deren aktiver Zustand deutlich für den Benutzer sichtbar ist.
changed	Der Wert einer angegebenen Variable oder eines angegebenen Objektattributs wurde geändert.
charinput	Ein Zeichen wurde in einen editierbaren Text eingegeben. Kann nun z.B. geprüft werden.
close	Schließmechanismus (z.B. Closebox) eines Fensters wurde betätigt. Bei einem Treeview wurde ein Teilbaum geschlossen.
cut	Benutzer hat mit Drag&Drop ein Objekt ausgeschnitten.
dbselect	Benutzer hat das Objekt durch einen Doppelklick selektiert, d.h. ein Objekt editiert.
deactivate	Benutzer hat ein Objekt deaktiviert, z.B. wenn er eine bereits aktivierte Checkbox nochmals selektiert, so dass diese nun deaktiviert wird.
deiconify	Icon wurde wieder zu einem Fenster geöffnet.
deselect	Objekt wurde deselektiert (durch Anwahl eines anderen Objekts oder durch Beenden der Eingabe mit `Return`-Taste beim editierbaren Text).
deselect_enter	Eingabe in das Textfeld wurde durch das Drücken der `Return`-Taste beendet.

Ereignistyp	Beschreibung
extevent	Externes Ereignis ist aufgetreten.
finish	Ende des Dialoges oder Moduls oder Beenden einer Verbindung zu einer Applikation.
focus	Eingabefokus wurde auf das angegebene Objekt gesetzt.
help	Benutzer hat für das aktuelle Objekt Hilfe angefordert. Die Anforderung der Hilfe erfolgt dabei so, wie es das jeweilige Fenstersystem vorsieht (z.B. für Motif & Windows: `F1`-Taste, unter Qt Tastenkombination `SHIFT+F1`).
hscroll	Der Benutzer hat horizontal gescrollt.
iconify	Iconifybox eines Fensters wurde betätigt; -> Fenster wird dadurch zum Icon.
key	Benutzer hat für das Objekt, für das diese Regel geschrieben wurde, den nach dem Schlüsselwort **key** angegebenen Accelerator gedrückt (siehe auch Kapitel „Benutzerereignisse mit besonderer Vererbung“).
modified	Benutzer hat das Editieren eines Textes beendet und diesen dabei aktiv verändert.
move	Benutzer hat das Fenster verschoben. Eine Toolbar wurde verschoben. Bei einer eingedockten Toolbar tritt das Ereignis nur auf, wenn sich durch das Verschieben die Attribute *.dock_offset* oder *.dock_line* ändern.
open	Eine Menübox wurde geöffnet. Bei einem Treeview wurde ein Teilbaum geöffnet.
paste	Der Benutzer hat ein Objekt mit Hilfe von Drag&Drop verschoben und über einem anderen Objekt losgelassen.
resize	Fenster wurde in seiner Größe verändert. Bei einer Splitbox wurde die Größe eines Splitbereichs geändert.
scroll	Benutzer hat gescrollt.
select	Objekt wurde vom Benutzer selektiert.
start	Start des Dialoges, Moduls oder Verbindungsaufbau einer Applikation.
vscroll	Der Benutzer hat vertikal gescrollt.

Beispiele für Ereigniszeilen

```
on dialog start
{
  program block
  /* execution on DM program start */
}
```

```
on WINDOW close
{
  program block
  /* on closing any window, i.e. on
     activation of close button */
}
```

```
on OK_Button select
{
    program block
    /* on activation of OK_Button */
}
```

```
on Window.title changed
{
  program block
  /* on change of window title */
}
```

```
on Window.title changed
if ( Variable_1 = 12 )
{
  program block
  /* on change of window title
     AND if Variable_1 contains the value 12 */
}
```

2.1.1 Drag und Drop-Ereignisse

In diesem Kapitel werden die Ereignisse beschrieben, die durch Drag&Drop-Operationen ausgelöst werden. Wie bei Clipboard-Aktionen werden die entsprechenden Events an das Objekt geschickt. Eine Drag & Drop-Operation löst beim Quellobjekt beim Cut einen Cut-Event aus und am Zielobjekt einen Paste-Event.

Achtung: Die Reihenfolge der Events ist willkürlich!

Das hier beschriebene Drag & Drop funktioniert zur Zeit nur auf den Microsoft Fenstersystemen.

2.1.1.1 Cut-Event

Beim Cut wird vom DM aus keine Löschaktion vorgenommen. Die Reaktion muss auf den Cut-Event hin erzeugt werden. Die Attribute thisevent.type und thisevent.value sind nicht belegt.

2.1.1.2 Paste-Event

Die Paste-Reaktion kann mit diesem Event ausgelöst werden.

thisevent.type enthält einen type_enum und zeigt das angenommene Format an.

thisevent.value enthält die Daten (DM_Datentyp).

Bei der Bearbeitung sollte man darauf achten, dass .cut_pending = true ist, wenn Quell- und Zielobjekt bei einer Move-Operation (Cut + Paste) **das gleiche** sind.

2.1.1.3 Beispiele

Die Beispiele zu den hier beschriebenen Ereignissen befinden sich im Verzeichnis *…\examples\dragdop*.

Modul mit Standard-Regeln

Das Modul **dnd_defa.mod** enthält Regeln für das Standardverhalten einiger Objekte.

Derzeit sind Regeln zur Behandlung von Cut und Paste des Formats *type_text* für die Objekte ***Edittext***, ***Listbox*** und ***Tablefield*** enthalten.

Allgemeine Regeln

- `rule boolean DnD_default_cut (object Obj input);`
  Cut-Regel, die je nach Objekt die passende Regel aufruft
- `rule boolean DnD_default_paste (object Obj input, anyvalue Event input);`
  Paste-Regel, die je nach Objekt die passende Regel aufruft

Spezielle Regeln

- `rule boolean Edittext_cut (object Obj);`
- `rule boolean Edittext_paste (object Obj, anyvalue Type, anyvalue Value, anyvalue Index);`
- `rule boolean Listbox_cut (object Obj);`
- `rule boolean Listbox_paste (object Obj, anyvalue Type, anyvalue Value, anyvalue Index );`
- `rule boolean Tablefield_cut (object Obj);`
- `rule boolean Tablefield_paste (object Obj, anyvalue Type, anyvalue Value, anyvalue Index );`

Der Rückgabewert aller Regeln zeigt an, ob die Bearbeitung erfolgreich ausgeführt werden konnte.

Dialog „dnd_et.dlg“

Dieser Dialog ist ein Drag&Drop-Beispiel für einen ***Edittext***. Die Regeln aus dem Modul **dnd_defa.-mod** werden hier verwendet.

Dialog „dnd_lb.dlg“

Dieser Dialog ist ein Drag&Drop-Beispiel für eine ***Listbox***. Die Regeln aus dem Modul **dnd_defa.-mod** werden hier verwendet.

Dialog „dnd_tb.dlg“

Dieser Dialog ist ein Drag&Drop-Beispiel für ein ***Tablefield***. Die Regeln aus dem Modul **dnd_defa.-mod** werden hier verwendet.

Dialog „solitair.dlg“

Dieser Stand-alone Dialog zeigt, wie der Drag&Drop-Datentyp *type_object* verwendet werden kann.

2.1.1.4 Tipps & Tricks zu Drag & Drop und Clipboard

Cut-Regeln und Paste-Regeln sollten immer unabhängig von einander arbeiten können. Dafür gibt es folgende Gründe:

- Quelle und Ziel sind oft verschiedene Objekte, die sich nicht einmal in der gleichen Applikation befinden müssen.
- Je nach Benutzereingabe können die Events in beliebiger Reihenfolge kommen. Der Benutzer kann z.B. am gleichen Objekt einmal über Tastatur ausschneiden oder kopieren, eine Drag & Drop-Operation ausführen und zwei Stunden später 10x einfügen.

Sobald an einem Objekt "Cut" erlaubt wird, kann diese Operation an jedem Zeitpunkt ausgelöst werden. Bis die "on cut" Regel ausgeführt wird, könnten andere Regeln vorher auf das Objekt zugreifen. Damit nicht versehentlich wichtige Informationen, auf die sich die Cut-Regel verlässt, verfälscht werden, wird das Attribut *.cut_pending* auf *true* gesetzt.

Aus diesen Gründen ist bei der Programmierung folgendes zu beachten:

Während *.cut_pending* = *true* ist, können am Objekt keine Attribute verändert werden. D.h. Regeln, die unter Umständen zwischen dem Auslösen des "Cuts" und dem *cut*-Event aufgerufen werden könnten (z.B. durch `on focus`), können per Default nichts am Objekt verändern.

Achtung

Bei einer Drag & Drop Move-Operation auf dem selben Objekt, kommt der Paste-Event vor dem Cut-Event, also mit *.cut_pending* = *true*.

Wenn eine Regel trotzdem Attribute neu setzen soll, kann *.cut_pending := false* gesetzt werden. Damit die nachfolgende Cut-Regel dies bemerken kann, wird dabei *.cut_pending_changed := true* gesetzt.

Bei der Verarbeitung des Formats DM_Objekt, in der Paste-Regel, sollte geprüft werden, ob das Objekt tatsächlich zu diesem Zeitpunkt existiert.

2.1.2 Benutzerereignisse mit besonderer Vererbung

Um in der Regelsprache des DM auch Tastaturereignisse verarbeiten zu können, können diese wie folgt abgefragt werden:

```
on <Objekt-ID> key <accelerator>
```

<Objekt-ID>
Name eines im Dialog definierten Objekts, Modells oder Defaults.

<accelerator>
Name eines im Dialog definierten Accelerators.

Damit nicht für jedes Objekt eine Regel definiert werden muss (z.B. ob die Taste `F1` gedrückt wurde), wird dieses Ereignis intern im DM nach folgendem Schema vererbt:

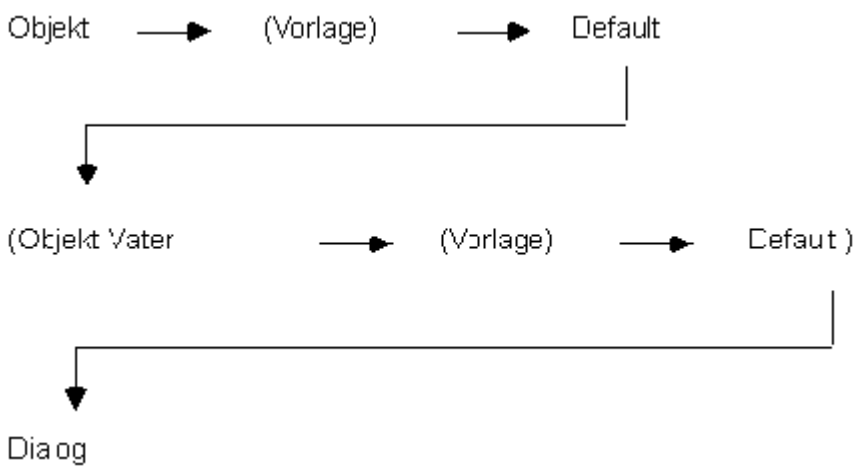

Abbildung 1: *Vererbung bei Key- und Help-Regeln*

Diese Vererbung wird unterbrochen, sobald eines dieser Objekte eine Regel für diese Taste definiert hat.

Beispiel

```
dialog Beispiel
accelerator FK5
{
  0: F5;
```

```
}

window Window
{
  .xleft  10;
  .width  100;
  .ytop   10;
  .height 200;

  child pushbutton
  {
    .xleft 10;
    .ytop  10;
    .text  "pushbutton";
  }
}

on Beispiel key FK5
{
  print "Taste F5 wurde gedrückt";
}
```

Sobald die Taste `F5` in diesem Fenster gedrückt wurde, wird die Regel ausgeführt.

Derselbe Mechanismus hierarchischer Vererbung gilt für das Hilfe-Ereignis. Dadurch wird erreicht, dass nur eine Regel für die "Hilfe" zuständig ist, die am Dialog hängt.

2.2 Interne Ereignisse

Ereignisse, die sich auf die Änderung eines Objektattributes oder einer Variable beziehen, sind „interne Ereignisse“.

```
on <Objekt-ID><Attributname> changed
```

Die Attribute, die hier verwendet werden können, entnehmen Sie bitte den Beschreibungen in der „Attributreferenz“.

Für Variablen gilt folgendes:

```
on <Variablenname>.value changed
```

Beispiel

```
on Variable_A.value changed
{
}
on Pushbutton.xleft changed
{
```

```
}
```

2.3 Systemereignisse

Ereignisse, die sich auf das gesamte DM-Programm-System beziehen, werden Systemereignisse genannt. Mit Hilfe dieser Ereignisse lassen sich Regeln für den Dialog- oder Modulstart und das Dialog- oder Modulende formulieren.

Dasselbe gilt auch für das Objekt application.

Start-Regel

```
on dialog start
  Ereignis, das den Beginn der Abarbeitung des DM-Programms anzeigt.

on module start
  Ereignis, das den Beginn der Abarbeitung des Moduls anzeigt.

on application start
  Ereignis, das den Beginn der Abarbeitung des application-Objektes anzeigt.
```

Ende-Regel

```
on dialog finish
  Ereignis, das das Ende der Abarbeitung des DM-Programms anzeigt.

on module finish
  Ereignis, das das Ende der Abarbeitung des Moduls anzeigt.

on application finish
  Ereignis, das das Ende der Abarbeitung des application-Objektes anzeigt.
```

Folgende Systemereignisse sind möglich:

Ereignistyp	Beschreibung
application start	Ereignis, das den Start der Abarbeitung des Application-Objektes anzeigt.
application finish	Ereignis, das das Ende der Abarbeitung des Application-Objektes anzeigt.
dialog start	Ereignis, das den Start der Dialogabarbeitung anzeigt.
dialog finish	Ereignis, das das Ende der Dialogabarbeitung anzeigt.
module start	Ereignis, das den Start des Moduls anzeigt.
module finish	Ereignis, das das Ende des Moduls anzeigt.

Beispiel

```
on dialog start
{
}

on dialog finish
{
}
```

Typischerweise beinhaltet die Start-Regel Statements, um Daten zu initialisieren, die Ende-Regel beinhaltet Statements für ein kontrolliertes Verlassen des DM (z.B. für das Schließen aller Fenster).

Die Start-Regel sollte immer definiert werden; zusätzlich muss mindestens eine Regel definiert werden, die das Schlüsselwort **exit** beinhaltet.

Die Ausführung der Regel, die das Schlüsselwort **exit** beinhaltet, ruft die Ende-Regel auf.

2.4 Externe Ereignisse

Externe Ereignisse sind Ereignisse, die zwar von ihrer Zuordnung und Abarbeitung her mit den Dialogereignissen im ISA Dialog Manager gleichzustellen sind, deren Quelle allerdings nicht unter der Kontrolle des IDM steht. Eine mögliche Quelle solcher Ereignisse kann z.B. ein Signal-Handler sein, der dieses Ereignis erzeugt, damit der Dialog darauf reagieren kann.

Ein externes Ereignis stellt eine Mischung aus Dialogereignis und parametrisierter Regel dar.

Wie bei den parametrisierten Regeln ist die Anzahl der Parameter auf *16* limitiert.

Allerdings ist zu beachten, dass bei der Nutzung aus der Regelsprache nur *14* Parameter verwendet werden können und es daher sinnvoll ist, externe Ereignisse mit maximal *14* Parametern zu definieren (siehe auch eingebaute Funktion **sendevent()**).

```
on <Objekt-ID> extevent <Ereignis-ID> ([<Datentyp> <Parametername> <Typ>])
```

anyvalue <Ereignis-ID>
: Bezeichnet einen für das jeweilige Objekt eindeutigen Identifikator, über den das externe Ereignis referenziert wird. Hier kann beispielsweise auch eine message-Ressource genutzt werden.

Externe Ereignisse werden an ein Objekt geschickt und von dort bis zum Default des Objekts weitergereicht bis ein Objekt das Ereignis verarbeitet.

Beispiel

```
on Object1 extevent 4711 (integer ErrCode, string ErrText);
on Object2 extevent EvSpeichern (boolean Erfolg, string Text);
```

Die Ausführung einer dem externen Ereignis zugeordneten Regel wird durch die Anwendung über die Schnittstellenfunktionen **DM_QueueExtEvent** oder **DM_SendEvent** vorgemerkt. „Vorgemerkt“

bedeutet, dass die Regel nicht sofort ausgeführt wird, sondern dass das externe Ereignis in eine Queue eingetragen wird und entsprechend den Mechanismen für Dialogereignisse weiterverarbeitet wird.

Siehe auch

C-Funktionen DM_QueueExtEvent() und DM_SendEvent() im Handbuch „C-Schnittstelle - Funktionen“

3 Regelabarbeitung

3.1 Normale Regeln

Unter normalen Regeln werden die Regeln verstanden, die kein weiteres Schlüsselwort in der Ereigniszeile haben. Diese Regeln haben dabei eine festdefinierte Art und Weise der Abarbeitung, die hier beschrieben werden soll.

Die Objekte, zu denen Regeln definiert werden können, haben intern eine Hierarchie. So kann ein sichtbares Objekt (Instanz) auf einem Modell beruhen, das seinerseits direkt vom zugehörigen Default abgeleitet ist. Bei der Vererbung von Attributen und bei der Vererbung von Regeln spielt diese Objekt-Hierarchie eine wichtige Rolle. Diese Hierarchiestufen darf man auf keinen Fall mit der Vater-Kind Beziehung verwechseln, die in der Regel keinen Einfluss auf die Vererbung von Attributen und Regeln hat.

Werden nun für unterschiedliche Hierarchiestufen normale Regeln definiert, so gilt wie bei den Attributen, was auf einer weiter unterliegenden Hierarchiestufe definiert ist, gilt und der Rest wird ignoriert. Werden also für ein und dasselbe Ereignis Regeln an einem Objekt, am zugehörigen Modell und Default definiert, so werden nur die Regeln am Objekt ausgeführt, da nur diese für das Objekt gelten. Die am Modell oder Default definierten Regeln kommen nicht zum Tragen, da sie von den Objektregeln überlagert werden. In diesem Fall verhalten sich die Regeln also identisch mit den Attributen.

Da dieses Verhalten für die Abarbeitung von Regeln zu starr ist, können mit Hilfe von weiteren Schlüsselworten zusätzliche Regeln ausgeführt werden. Diese werden in den nachfolgenden Kapiteln erläutert.

3.2 before-Regeln

Mit Hilfe des Schlüsselwortes **before** können Regeln so gekennzeichnet werden, dass sie auf jeden Fall vor den normalen Regeln ausgeführt werden. Dieses Schlüsselwort wird hinter die Ereigniszeile geschrieben.

Syntax

```
on { <Objekt-ID> } <Ereignis> { , <Ereignis> } before
```

oder

```
on { <Objekt-ID> } <Attribut> changed before
```

<Objekt-ID> kann **nur** entfallen, wenn die Regel innerhalb einer Objektdefinition definiert wird.

Bei der Suche nach diesen before-Regeln fängt der IDM beim zugehörigen Default des Objektes an zu suchen und geht dann über die Modellhierarchie zum eigentlichen Objekt. Alle dabei gefundenen Regeln mit **before** werden ausgeführt.

3.3 after-Regeln

Mit Hilfe des Schlüsselwortes **after** können Regeln so gekennzeichnet werden, dass sie auf jeden Fall nach den normalen Regeln ausgeführt werden. Dieses Schlüsselwort wird hinter die Ereigniszeile geschrieben.

Syntax

```
on { <Objekt-ID> } <Ereignis> { , <Ereignis> } after
```

oder

```
on { <Objekt-ID> } <Attribut> changed after
```

<Objekt-ID> kann **nur** entfallen, wenn die Regel innerhalb einer Objektdefinition definiert wird.

Bei der Suche nach diesen after-Regeln fängt der IDM beim jeweiligen Objekt an und geht dann über die Modellhierarchie zum zugehörigen Default. Alle dabei gefundenen Regeln mit **after** werden ausgeführt.

3.4 Ausführungsreihenfolge

Aus diesen verschiedenen Arten von Regeln ergibt sich eine fest definierte Reihenfolge, die wie folgt beschrieben werden kann:

Werden für ein Objekt und ein Ereignis mehrere Regeln definiert, so ist die Definitionsreihenfolge entscheidend für die Ausführungsreihenfolge. Dieses sollte nur dann genutzt werden, wenn die Regeln untereinander unabhängig sind.

Wenn ein Ereignis auftritt, fängt der DM an, nach für das Ereignis passenden Regeln zu suchen. Dabei wird bei dem zugehörigen Default des Objektes angefangen nach Regeln mit **before** zu suchen. Vor dort aus wird bei dem oder den zu dem Objekt gehörenden Modellen nach diesen Regeln gesucht. Abschließend wird beim Objekt selbst überprüft, ob eine Regel mit **before** definiert worden ist. Dabei werden alle gefundenen Regeln ausgeführt.

Danach beginnt der DM bei dem Objekt nach einer normalen Regel zu suchen und geht, falls er bei dem Objekt keine findet, zu dem zugehörigen Modell. Falls aber an dem Objekt eine Regel für das eingetroffene Ereignis definiert worden ist, geht er nicht mehr zu dem Modell. Bei dem Modell überprüft er, ob hier eine passende Regel definiert ist. Ist das nicht der Fall, geht er zum zugehörigen Modell des Modells bzw. zum Default. Falls aber eine Regel am Modell vorhanden ist, wird diese ausgeführt und die Suche abgebrochen.

Abschließend fängt der DM wieder bei dem Objekt an, nach passenden Regeln für das Ereignis zu suchen, diesmal aber solche, die mit dem Schlüsselwort **after** gekennzeichnet sind. Vom Objekt geht er zu dessen Modell und beginnt dort wieder mit der Suche. Vom Modell geht er dann zu dessen Modell bzw. dem Default und sucht dort auch noch nach passenden Regeln. Hierbei werden aber alle gefundenen Regeln ausgeführt.

Eine Abweichung von diesem Schema gibt es für Regeln die an Tastaturereignisse (*key*) und an das Hilfeereignis (*help*) gebunden sind. Für diese Ereignisse gilt das obengenannte Schema, zusätzlich wird aber noch zum Vater des Objektes gegangen, falls auf dem gesamten Weg keine passende Regel gefunden worden ist. Damit kann man sehr einfach in Form einer Regel ein Hilfesystem anbinden, wenn man die entsprechende Regel im Dialog hinterlegt.

Das nachfolgende Schaubild veranschaulicht die Suche nach Regeln. Dabei wird in diesem Bild davon ausgegangen, dass dem Objekt ein Modell zugeordnet ist, das direkt vom Default abgeleitet ist.

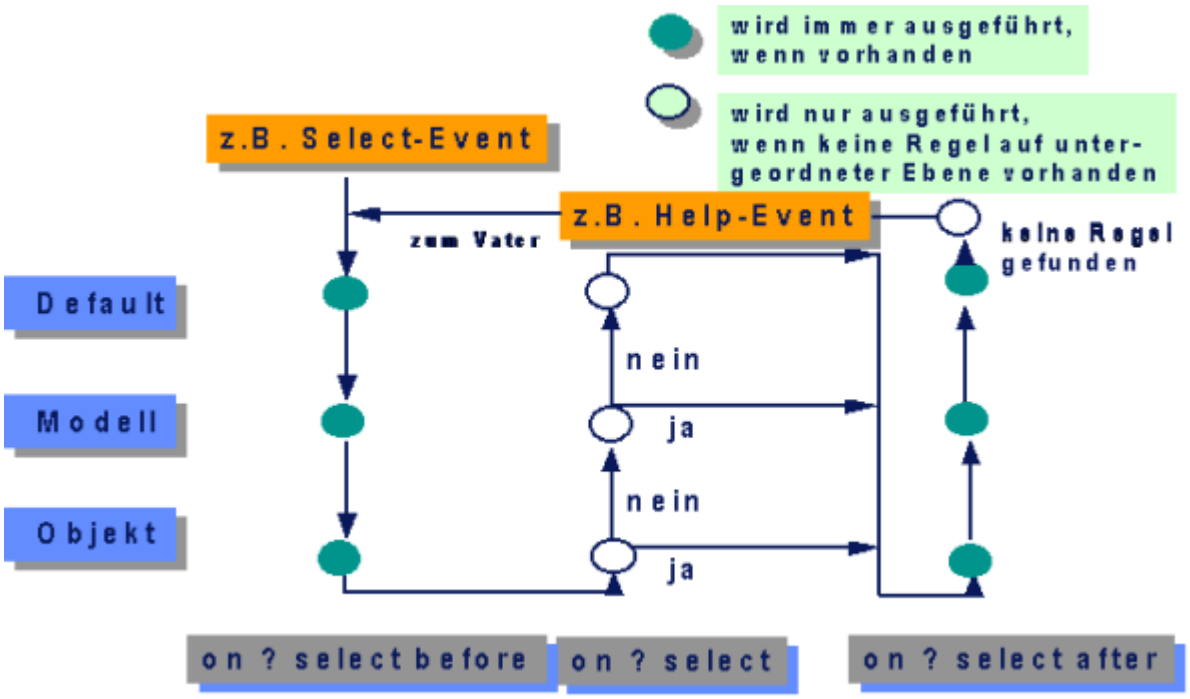

Abbildung 2: *Regelabarbeitungs-Reihenfolge*

Beispiel

Für ein Fensterobjekt seien folgende 4 Regeln definiert:

1. `on WINDOW close before`
2. `on Window1 close`
3. `on WINDOW close`
4. `on Window1 close after`

Damit ergibt sich folgende Reihenfolge: 1-2-4.

Die Regel 3 wird nie ausgeführt wegen der Existenz der Regel 1.

Für einen Pushbutton seien folgende 5 Regeln definiert:

1. `on PUSHBUTTON select`
2. `on OKButton select`

3. `on PUSHBUTTON select after`
4. `on OKButton select before`
5. `on OKButton select after`

Damit ergibt sich folgende Reihenfolge: 4-2-5-3.

Die Regel 1 wird nie ausgeführt wegen der Existenz der Regel 2.

Als Anwendungsbeispiel kann hier ein System zur Dateneingabe angeführt werden. In diesem System werden viele Fenster definiert, die über einen OK-Button den Inhalt verarbeiten und über einen Abbrechen-Button den Inhalt verwerfen. Bei beiden Buttons wird jeweils das Fenster geschlossen.

Die Regeln kann man nun z.B. so definieren, dass am Modell des Abbrechen-Buttons eine Regel hängt, die das jeweilige Fenster schließt. Im Fall vom OK-Button ist es schwieriger, da zunächst eine fensterspezifische Verarbeitung erfolgen muss. Deshalb ergibt sich folgende sinnvolle Möglichkeit:

An jeder Instanz des OK-Buttons hängt eine Regel, die die aktuelle Funktion aufruft und am Modell des OK-Buttons hängt eine after-Regel, die immer das Fenster schließt. Damit wird erreicht, dass das Schließen des Fensters zentral erfolgt und nicht immer programmiert werden muss.

4 Benannte Regeln (Unterprogramme)

Neben der Möglichkeit, Regeln an Objekte zu binden, gibt es noch die Möglichkeit Regeln unabhängig von einem Objekt zu definieren. Diese Regeln werden dann wie Unterprogramme genutzt und von mehreren Programmstellen aus aufgerufen. Der IDM kann also diese Regel nie direkt aufgrund einer Benutzeraktion aufrufen, sondern sie wird immer durch eine andere Regel aufgerufen.

Diese sogenannten benannten Regeln können bis zu 16 Parameter haben. Diese Parameter können dabei als reine Eingabewerte, als reine Ausgabewerte oder auch als Ein-/Ausgabewerte aus Sicht der Regel betrachtet werden. Dieses muss bei der Deklaration der Parameter mitdefiniert werden.

Anstelle des Schlüsselwortes **on** fangen diese Regeln mit dem Schlüsselwort **rule** an, um zu kennzeichnen, dass es sich um einen komplett anderen Regeltyp handelt.

Syntax

```
rule <Rückgabetyp> <Regelname> ( { <Parameter> { , <Parameter> } } )
{
  <Anweisungen>
}

Parameter ::=
  <Datentyp> <Parametername> { := <Standardwert> } { <Art> }
```

<Parameter> kann bis zu 16-mal vorkommen, d.h., die Regel kann bis zu 16 Parameter haben.

<Rückgabetyp>

<Datentyp>
Alle Datentypen, die im IDM existieren, sind als gültige Datentypen für Regel-Parameter und Regel-Rückgabetypen verfügbar. Beim Rückgabetyp ist zusätzlich der Typ *void* verfügbar, der kennzeichnet, dass die Regel eigentlich nichts zurückliefert.

<Regelname>

<Parametername>
Muss ein gültiger Identifikator sein.

<Standardwert>
Für Eingabeparameter (**input**) können Standardwerte definiert werden. Diese Parameter müssen dann beim Aufruf der Regel nicht angegeben werden.
Parameter mit Standardwerten müssen am Ende der Parameterliste stehen!

Achtung
Optionale Parameter werden vom ***control***-Objekt und von der ***message***-Ressource nicht unterstützt. Die Benutzung von Standardwerten bei der Programmierung eines OLE Servers oder eines OLE Clients mit der OLE Schnittstelle ist momentan nicht möglich.

<Art>

- **input** (Standard)
  nur Eingabeparameter
- **output**
  nur Ausgabeparameter
- **input output**
  sowohl Eingabe- als auch Ausgabeparameter

Dabei ist input und output immer aus Sicht der Regel, die hier definiert wird.

Beispiel

```
rule integer Add (integer Arg1 input, integer Arg2 input)
```

Diese Addierungsregel hat zwei Zahlen als Eingabeparameter, addiert die zwei Parameter und hat das Ergebnis "Summe der Addition".

```
rule void CheckObjects ()
```

Dieses definiert eine Regel ohne Parameter, die auch keinen Rückgabewert hat.

5 Spezielle Objekte und Objektreferenzierung

In den folgenden Kapiteln finden Sie die Beschreibung wie Objekte im Inneren der Regeln referenziert werden können. Zusätzlich werden in den nachfolgenden Kapiteln zwei spezielle Objekte vorgestellt, die den Zugriff auf Objekte wesentlich erleichtern.

5.1 this

In dem in jeder Regel verfügbaren ***this***-Objekt ist immer das aktuelle Objekt abgespeichert, das die Regelverarbeitung ausgelöst hat. Damit können zum Beispiel Regeln an Modelle gebunden werden, die aber nur auf die aktuelle Instanz des Modells wirken. Daher werden im Normalfall alle an Modellen definierten Regeln über das ***this*** auf das aktuelle Objekt zugreifen und von dort aus die entsprechenden Aktionen einleiten. Nur in den seltensten Fällen wird man in den Regeln direkt auf das Modell zugreifen wollen und dort Änderungen vornehmen wollen.

Beispiel

Es ist ein Modell eines Pushbuttons wie folgt definiert:

```
model pushbutton OK {}
```

In einer Regel soll auf das zugehörige Fenster zugegriffen werden. Das sieht dann wie folgt aus:

```
!! Regel ist am Modell definiert
on OK select
{
  !! Regel greift über die Instanz auf das
  !! zugehörige Fenster zu
  this.window.visible := false;
}
```

Würde die Regel über den Objektnamen zugreifen, wäre die Reaktion undefiniert, da in der Regel die Instanzen einzeln sichtbar gemacht werden und sie daher die Sichtbarkeit nicht mehr vom Modell erben.

5.2 Objektreferenzierung

Es gibt zwei Arten, Objekte zu referenzieren:

1. Mit eindeutigen Namen, d.h. das Objekt hat einen eindeutigen Namen oder einen kompletten Pfad:

   Beispiele

   `END_Button`

```
MyWindow.OKButton
```

2. Durch Spezifikation des Vaters, d.h. durch Inbeziehungsetzen des Objektes bzw. des Vaters und der Kinder:

   Beispiele

   - `Object.parent`
     adressiert den Vater des Objektes.
   - `Object.window`
     adressiert das Fenster, in das das Objekt positioniert wurde.
   - `Object.groupbox`
     adressiert die Groupbox, in die das Objekt positioniert wurde.

Die oben beschriebenen zwei Möglichkeiten der Objektreferenzierung können in dem Ereignis, der (Vor-)Bedingung und den Aktionszeilen verwendet werden, d.h. auf ein Objekt kann mit einem eindeutigen Namen oder über Beziehungen ("relations") Bezug genommen werden.

Der Begriff Relation ("Verwandtschaftsbeziehung") fasst die Referenzierungsmöglichkeiten der Objekte untereinander zusammen.

Folgende Schlüsselworte stehen zur Bildung einer Relation zur Verfügung:

- **child[i]**
  Setzt das "i"-te Kind eines Objektes (Fenster, Groupbox, Menübox).
- **firstchild**
  Setzt das erste Kind eines Objektes.
- **firstmenu**
  Bezeichnet das erste Menü eines Objektes.
- **groupbox**
  Bezeichnet die Groupbox, in der das Objekt liegt.
- **lastchild**
  Setzt das letzte Kind eines Objektes.
- **lastmenu**
  Bezeichnet das letzte Menü eines Objektes.
- **menu[i]**
  Erfragt das "i"-te Kind der Menühierarchie eines Objektes.
- **parent**
  Bezeichnet den Vater des Objektes.
- **this**
  Bezieht sich auf das Objekt, das das anstehende Ereignis ausgelöst hat.
- **window**
  Bezeichnet das nächste in der Hierarchie oberhalb (Richtung Vater) liegende Fenster.

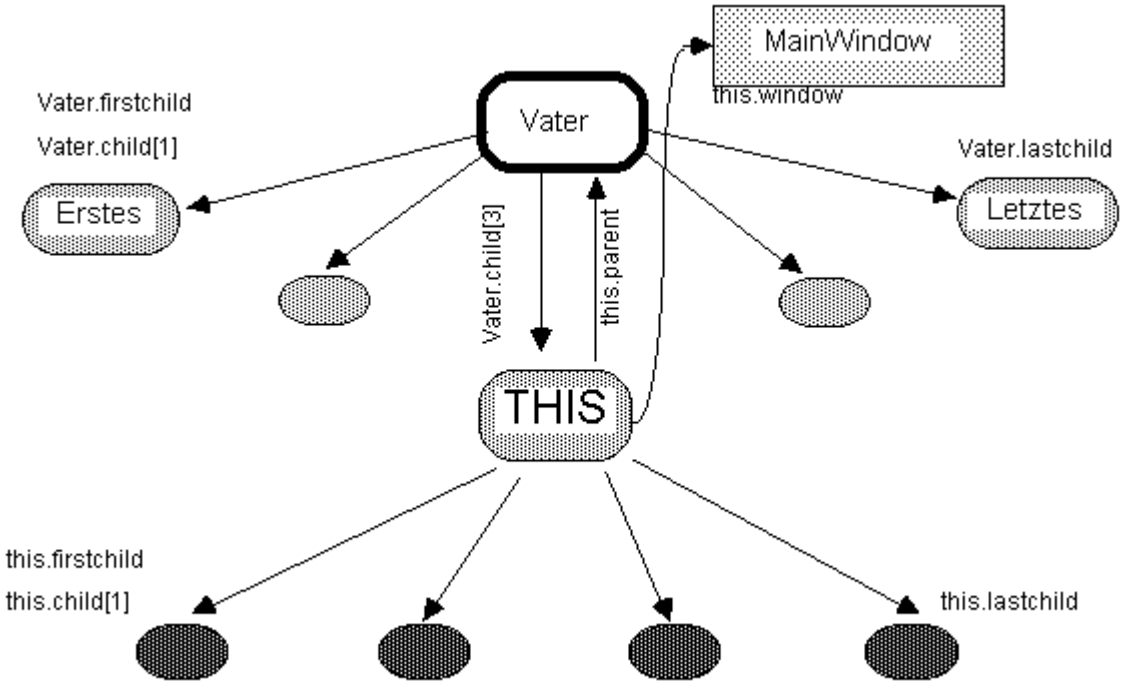

Abbildung 3: *Referenzierungsmöglichkeiten*

Um auf ein bestimmtes Objekt in der Hierarchie zuzugreifen, können einige der oben genannten Schlüsselworte zusammengefügt werden. Am Ende einer solchen Referenzaktion können die Attribute des so gefundenen Objektes spezifiziert werden.

Definition

```
<Objekt>::=
  <Objektpfad>{<Relation>}[<Attribut>]

<Objektpfad>::=
  <Objektidentifikator>{.<Objektidentifikator>}|
  <Variable>|
  this
```

Beispiel

```
{
  MainWindow.ytop := this.window.ytop + this.window.height;
}
```

Das Fenster "MainWindow" wird hier an die untere Kante des Fensters positioniert, in dem das Objekt ***this*** liegt.

Anmerkung

Die Verwendung der Relations ist nur dann angebracht, wenn die Beziehungen der Objekte zueinander im Wesentlichen bekannt sind.

Werden dynamische Objekte erzeugt, so ist diese Art der Objektreferenzierung oft der beste Weg, ein Objekt zu bezeichnen.

Forward Referencing

Neben der Referenzierung für bereits definierte Objekte, können an bestimmten Stellen Objekte auch erst nach der Referenzierung definiert werden, und zwar

- in Regeln (Expressions) und
- in Attributen von Objekten.

5.3 null-Objekt

Es gibt ein Objekt mit einem speziellen Identifikator, mit dessen Hilfe das Zurücksetzen von Ressource-Attributen ermöglicht wird. Dieses Objekt wird ***null***-Objekt genannt und hat die Dialog Manager ID ***null***.

Dieses ***null***-Objekt kann dazu benutzt werden, alle Ressource-Attribute zurückzusetzen. Es kann bei Zuweisungen auf alle Objekt-Attribute benutzt werden.

Implikationen

- `object.bgc := null;`
  Entfernt die Farbe an dem Objekt.
- `object.text := null;`
  Eliminiert den Text des Objekts.
- Definition:

```
default pushbutton
{
}
model pushbutton PBM
{
  .fgc RED;
}
pushbutton PB1
{
}
pushbutton PB2
{
  .model PBM;
}
pushbutton PB3
{
  .model PBM;
  .fgc null;
}
```

Bei dieser Definition haben die Pushbuttons PB1 und PB3 keine Farbe, PB2 hat die Vordergrundfarbe RED.
Die Zuweisung verbietet explizit, dass Pushbutton PB3 die Farbe seines Modells erbt.

5.4 Ereignisobjekt thisevent

Neben den anderen speziellen Objekten ***setup*** und ***this*** gibt es noch das spezielle Objekt ***thisevent***.

Mit Hilfe des Schlüsselwortes **thisevent** können Informationen zum aufgetretenen Ereignis erhalten werden. Es liefert das (virtuelle) Ereignis, das verschiedene Attribute besitzt. Diese Attribute können ausgelesen werden.

Attribute des Ereignisobjektes thisevent

Attribut	Typ	Bedeutung	Gültig bei
.accelerator	object	Accelerator-Identifikator	key
.attribute	attribute	geändertes Attribut	changed
.count	integer	Anzahl der aufgetretenen Timer-Ereignisse	*select* in Timer
.event_code	anyvalue	Code des externen Ereignisses	extevent
.eventcount	integer	Anzahl der Ereignisse	allen Ereignissen
.event[EV]	boolean	true, wenn EV enthalten	allen Ereignissen
.event[I]	event	Typ des i-ten Ereignisses	allen Ereignissen
.index	integer	selektierter Eintrag	*select*, *cut*, *paste* in Listbox, Poptext, Treeview
.index	index	selektiertes Feld	*select*, *cut*, *paste* im Tablefield
.type	enum	Art der von der Drag&Drop-Operation betroffenen Objekts	*paste*
.value	anyvalue	Wert, der mit der Drag&Drop Operation verschoben wurde	*paste*
.x	integer	Maus X-Koordinate (Pixel)*	*select* in Fenster, Groupbox
.y	integer	Maus Y-Koordinate (Pixel)*	*select* in Fenster, Groupbox

* Die Mauskoordinaten sind nur bei den Objekten Fenster und Groupbox verfügbar und relativ zum Objekt, auf das sich das Ereignis bezieht.

Ist ein Attribut ungültig, so enthält es *void*-Data. Dies kann mit der Builtin-Funktion **typeof** ermittelt werden.

Eine Regel kann von mehreren Ereignissen gleichzeitig ausgelöst werden. Alle diese Ereignisse sind im (indizierten) Attribut *.event[ ]* enthalten.

Mit *.eventcount* kann die Anzahl der Ereignisse ermittelt werden.

Wird *.event* mit einem Integerwert indiziert, erhält man ein gleichzeitig aufgetretenes Ereignis.

Wird *.event* mit einem Wert vom Typ *event* indiziert, ist das Ergebnis vom Typ *boolean* mit dem Wert *true*, falls das Index-Ereignis eines der aufgetretenen Ereignisse ist.

Als EV kann jegliches Ereignis eingetragen werden, so kann z.B. erfragt werden, ob ein spezielles Ereignis vorliegt, z.B. *.event[select]*.

6 Datentypen

In der Regelsprache sind wie in anderen Programmiersprachen auch verschiedene Datentypen definiert, die hier erklärt werden sollen.

Datentyp	Bedeutung
anyvalue	Dieser Datentyp stellt einen beliebigen Datentyp dar. Erst durch die aktuelle Belegung erhält er seinen eigentlich Datentyp. In einem so definierten Wert kann also alles abgespeichert werden, bei der Abfrage ist dann entscheidend, welche Art von Wert zuletzt abgespeichert worden ist.
attribute	Dieser Datentyp ist ein Attribut des Dialog Managers. Er kann daher mit allen vom Dialog Manager definierten Attributen wie .visible oder einem benutzerdefinierten Attribut belegt werden.
boolean	Dieser Datentyp stellt einen booleschen Wert dar. Seine Werte sind entweder *true* oder *false*.
class	Dieser Datentyp gibt die Klasse eines Objekts oder einer Ressource an. Werte umfassen z.B. Klassen wie z.B. *pushbutton, color, function, ...*
datatype	Dieser Datentyp stellt einen Datentyp dar. In ihm können also Werte wie *string, object, integer, ...* hinterlegt werden.
enum	Dieser Datentyp ist ein Aufzählungstyp. In ihm werden vom Dialog Manager definierte Werte hinterlegt. Dieser Aufzählungstyp kommt z.B. bei der Messagebox zum Einsatz, wenn der Benutzer einen von den angebotenen Pushbuttons drückt.
event	Dieser Datentyp stellt ein Ereignis im Dialog Manager. In ihm können also Werte wie *select*, *dbselect*, *key* und *help* abgespeichert werden.
index	Dieser Datentyp dient der Adressierung von zweidimensionalen Attributen wie z.B. dem Inhalt eines Tablefields. In diesem Datentyp können zwei ganze Zahlen abgespeichert werden, die zusammen eine Zeile und eine Spalte, also genau eine Zelle adressieren.
integer	Dieser Datentyp stellt eine ganze Zahl dar. Der Wertebereich für diese Zahl ist dabei von -2^{31} bis 2^{31}.
method	Dieser Datentyp stellt eine Methode im Dialog Manager dar. Er kann zum Beispiel die Werte **:insert**, **:delete**, **:clear** oder **:exchange** annehmen.

Datentyp	Bedeutung
object	Dieser Datentyp stellt ein Objekt im Sinn des Dialog Managers dar. Im Sinn des Dialog Managers ist alles ein Objekt, was einen Namen hat oder einen Namen bekommen kann. Damit sind alle Objekte, Ressourcen, Funktionen, Variablen und Regeln Objekte in diesem Sinn.
pointer	Dieser Datentyp stellt einen für die Regelsprache unbekannten Typ dar. Er kann genutzt werden, um anwendungsspezifische Daten im Dialog zu speichern, ohne dass der Dialog den Inhalt dieser Daten kennen muss.
string	Dieser Datentyp stellt eine beliebig lange Zeichenkette im Dialog Manager dar. Die Länge wird dabei automatisch bei einer Zuweisung auf diesen Typ angepasst, so dass hier keine Maßnahmen durch die Anwendung erfolgen müssen.

6.1 Datentypen für Sammlungen

Folgende **Sammlungsdatentypen** sind im IDM erlaubt. Dabei ist zu beachten, dass für die Adressierung wie auch die adressierten Einzelwerte die Beschränkung gilt, dass nur skalare Typen erlaubt sind. Ein Aufbau von mehrdimensionalen Listen ist also nicht erlaubt.

Tabelle 1: *Datentypen für Sammlungen (collections)*

Datentyp	Bedeutung
list	Dieser Datentyp definiert eine Liste mit beliebigen Werten die im Adressierungsbereich von 0 … 2^{31} liegen. Dabei wird über *0* der Standardwert adressiert, der an alle Listenwerte vererbt wird. Dieser Datentyp ist für die Manipulation von Listen in der Regelsprache vorgesehen und optimiert.
vector	Dieser Datentyp definiert eine Liste, die Werte des gleichen Typs beinhaltet. Einen Standardwert gibt es nicht. Der Adressierungsbereich ist somit von 1 … 2^{31}. Dieser Datentyp ist speziell für die Kommunikation mit Objekt-Attributen vorgesehen und **nicht** für eine schnelle Manipulation optimiert.
refvec	Dieser Datentyp definiert eine Liste, die nur Werte des Typs *object* beinhaltet. Der Adressierungsbereich geht von 1 … 2^{31} und bietet keinen Standardwert. Jede object-ID wird nur ein einziges Mal aufgenommen, ein *null* wird nicht in die Liste aufgenommen.
matrix	Dieser Datentyp definiert ein zweidimensionales Feld dessen Werte mit dem *index*-Datentyp adressiert werden können. Ist die angegebene Zeile (first-Wert des Index) oder die Spalte (second-Wert des Index) *0*, so handelt es sich um einen Standardwert der in der Adressierungsreihenfolge [0,0] → [Zeile,0] → [0,Spalte] → [Zeile,Spalte] weitervererbt wird

Datentyp	Bedeutung
hash	Dieser Datentyp definiert ein assoziatives Feld mit beliebigem skalaren Adressierungsbereich und beliebigen Werten und einem optionalen Standardwert der zurückgegeben wird für einen Zugriff bei dem der Schlüssel nicht enthalten ist.

Analog zu vordefinierten Vektor- und Matrixattributen wie *.content[]*, *.userdata[]* usw. ist eine Erweiterung von Sammlungen direkt hinter ihrem letzten Element möglich und erlaubt.

Allerdings ist die Verwendung dieser Sammlungsdatentypen für benutzerdefinierte Attribute nicht möglich. Benutzerdefinierte und vordefinierte Attribute besitzen keine eindeutige Zuordnung (Attribute existieren an einem Objekt in einer skalaren **und** feldhaften Variante) und erlauben einen unindizierten Zugriff auf den Standardwert.

6.1.1 Syntax von Ausdrücken

Liste und Hashes

Sammlungen mit den Datentypen *hash*, *list*, *matrix*, *refvec* oder *vector* können über die folgende Syntax innerhalb des Regelcodes definiert werden. Eine Definition im statischen Teil eines Dialogs oder Moduls ist nicht als Expression möglich, lediglich konstante Werte sind hier erlaubt. Ohne explizite Angabe eines Listen-Datentyps entsteht eine Liste vom Datentyp *list*. Bei Verwendung des Verweisoperators => entsteht automatisch ein *hash* (assoziatives Feld), wobei der Ausdruck vor dem => den Schlüssel bzw. Index definiert, der zweite den Wert.

Achtung

Aus Kompatibilitätsgründen genießt die Index-Expression Vorrang. Will man also eine zweielementige Liste mit zwei Integer-Werten aufbauen, so sollte man den Listendatentyp voranstellen.

Syntax

```
<Liste> = [ <Datentyp> ] '[' [ <Expression> { ',' <Expression> } ] ']'
<Hash-Liste> = [ <Datentyp> ] '[' [ <Expression> '=>' <Expression>
    { ',' <Expression> '=>' <Expression> } ] ']'
```

Beispiele

```
[1, .xleft, 17+4, Wi.Pb]
["ZDF" => 2, itoa(7) => 7]
matrix[ [1,1]=>"Vorname", [1,2]=>"Nachname", [1,3]=>"Telefonnr." ]
```

6.1.2 Listengröße und Standardwerte

Grundsätzlich hat jeder Sammlungsdatentyp eine aktuelle Größe, die sich entsprechend den gesetzten Werten ergibt. Wertelücken in diesem aktuell gültigen Indizierungsbereich gibt es nicht, lediglich ungesetzte Werte (siehe hierzu auch die Beschreibungen zu **itemcount()**, **countof()**).

Um die Konsistenz zu den bisherigen statischen lokalen Variablen herzustellen erlauben die Datentypen *hash*, *list* und *matrix* einen Standardwert.

Dieser wird angezeigt an ungesetzten Werte-Positionen innerhalb des aktuell gültigen Indizierungsbereiches bis zur aktuellen Größe. Bei *hash*-Datentypen wird bei gesetztem Standardwert dieser für alle möglichen Indexierungen zurückgeliefert.

Die Reihenfolge für den Zugriff auf die Standardwerte bei ungesetztem Wert an der Position *<Row>,<Col>* in einer Matrix *M* ist *M[<Row>,<Col>] → M[0,<Col>] → M[<Row>,0] → M[0,0]*.

Die Zugriffsreihenfolge bis zum Standardwert bei ungesetztem Wert an *<Index>* in einer Liste *L* ist *L [<Index>] → L[0]*.

6.1.3 Zugriff und Zuweisungen von Sammlungen

Der Zugriff auf die indizierten Werte (Elemente) einer Sammlung erfolgt, wie bisher von lokalen Variablen oder vordefinierten und benutzerdefinierten Attribute gewohnt, über die [ ]-Operation.

Ohne die [ ]-Operation wird der gesamte Wert geholt oder gesetzt.

Achtung

Es ist zu beachten, dass ein Zugriff auf ein benutzerdefiniertes oder vordefiniertes Attribut ohne [ ]-Index den Standardwert bzw. das skalare Attribut holt oder setzt. Möchte man alle Werte eines Attributs holen oder setzen, so sollten die Funktionen **getvector()** oder **setvector()** verwendet werden.

Beispiele

```
variable list Primes := [2,3,5,7,11,13];
variable vector[integer] Numbers;
variable hash Months := [1=>"Jan", 2=>"Feb", 3=>"Mar"];

print Primes[1];
Months[4] := "Apr";
Numbers := Primes;
Primes[0] := -1;
Primes := [17, 19, 23];
```

Das Ausgeben von Sammlungen über **print** oder **sprintf()** ist ebenso möglich. Dabei werden aber keine Standardwerte mit ausgegeben. Bei den Datentypen *hash* und *matrix* erfolgt die Ausgabe inklusive Index.

Besonderheiten *refvec*

Weiterhin ist zu beachten das der Datentyp *refvec* eine besondere Behandlung genießt, da er für die Eindeutigkeit der Werte sorgt und keine *null*-Werte speichert.

Bei einer indizierten Zuweisung einer schon vorhandenen object-ID wird die object-ID an der Position ersetzt und danach für die Eindeutigkeit gesorgt, was damit auch eine Verschiebung der schon vor-

handenen object-ID an eine andere Position und eine Verkleinerung der *refvec*-Liste zur Folge haben kann.

Bei einer indizierten Zuweisung einer *null* wird die object-ID aus der *refvec*-Liste gelöscht und die nachfolgenden object-IDs rücken vor, was wiederum eine Verkleinerung der *refvec*-Liste zur Folge hat.

6.1.4 Automatische Konvertierung

In Zuweisungen oder Parameteraufrufen findet bei Sammlungen eine automatische Konvertierung in den verlangten Listentyp statt. Dabei werden alle Werte ohne Index in ihrer „natürlichen“ Reihenfolge kopiert, wenn möglich inklusive der Standardwerte, z.B. *[0]* bei einer *list*-Variablen wird weitergegeben, wenn der Zieltyp ebenso einen Standardwert besitzt.

Ausnahme

Bei der Zuweisung von einem *hash* mit *index*-Datentyp an eine *matrix* wird der Index übernommen. Bei der umgekehrten Zuweisung von einer *matrix* an einen *hash* wird ebenso der Index mit übernommen. Will man dies verhindern, so sollte die eingebaute **values()**-Funktion verwendet werden.

6.1.5 Performanz

Grundsätzlich sollte bedacht werden, dass die Verwendung von Sammlungen (Gesamtwerte) innerhalb von Ausdrücken initial ein Kopieren der Werte erforderlich macht, dann aber bei lesenden Operationen lediglich Referenzen dieser Liste, also ohne weiteres Kopieren, Anwendung finden. So wird die nötige Performanz bei Umgang mit Sammlungen gewährleistet. Die Weitergabe an Applikationsfunktionen oder die Manipulation durch eingebaute Funktionen machen aber eine erneute Kopieraktion, unter Umständen mit einer Codepage-Konvertierung von String-Werten, erforderlich.

Insofern sollte bedacht werden, dass bei komplexen Ausdrücken, welche Sammlungen bearbeiten, durchaus temporär große Datenmengen auflaufen können. Die Kodierung und Aufteilung von Ausdrücken sollte mit Berücksichtigung der maximalen Listengrößen erfolgen.

6.1.6 Lokale und globale Variablen

Die Sammlungsdatentypen sind bei lokalen und bei globalen Variablen verfügbar. Eine Initialisierung einer globalen Sammlung ist ebenso mit konstanten Werten möglich.

Bisher konnten lokale, statische Variablen auch ein Feld oder assoziatives Feld sein, welches nur sehr beschränkt initialisiert werden konnte. Sammlungsdatentypen sind nun auch genauso für lokale und auch lokale, statische Variablen erlaubt. Der Initialisierungsausdruck ist dabei nicht mehr eingeschränkt.

Achtung Verhaltensänderung

Die bisherige Schreibweise für Felder und assoziative Felder für statische lokale Variablen ist wie bisher auch erlaubt und wird auf die Datentypen *list* oder *hash* abgebildet. Die statische Initialisierung

über `.<Bezeichner>[<Index>] := <Wert>;` nach dem Variablenteil ist aber **nicht mehr möglich** um die Konsistenz zu anderen lokalen Variablen zu gewährleisten! Der Initialwert setzt den gesamten Variablenwert, und nicht wie früher nur den Standardwert!

Beispiel

```
dialog D

variable hash Prices := [ "iMac" => 1300, "Samsung Tab" => 300 ];
variable vector[string] Weekdays := ["Mon","Tue","Wed"];

on dialog start
{
  variable string Stations[integer] := [1=>"ARD", 2=>"ZDF"];
  variable hash Stations2 := [3=>"SWR", 4=>"RTL"];
  static variable list AllStations := join(Stations,Stations2);
  /* Nicht mehr erlaubt:
   * variable string Stations[integer] := "UNBEKANNT";
   * .Stations[1] := "ARD";
   * oder
   * static variable integer AssArray[string] := -1;
   * Stattdessen möglich:
   * static variable integer AssArray[string] := [nothing=>-1];
   */
  exit();
}
```

6.1.7 Vergleichsoperatoren

Für Sammlungen sind nur die Vergleichsoperatoren = und <> erlaubt.

Gleichheit herrscht bei folgenden Bedingungen:

1. Der Datentyp muss gleich sein.
2. Alle enthaltenen Werte bzw. Index/Wert-Paare müssen gleich sein, inklusive der Standardwerte.
3. Die Anzahl der enthaltenen Werte und ihre Positionen müssen gleich sein.

Beim Vergleich zwischen einem String und einer Text-ID erfolgt der Vergleich auf String-Basis.

Bei der Verwendung von Vergleichsoperatoren ist insbesondere auf den Datentyp zu achten.

Beispiel

```
dialog D
window Wi {
  listbox Lb {
    .content[1] "Sat";
    .content[2] "Sun";
```

```
  }
}
on dialog start {
  variable list WeekendDays := ["Sat", "Sun"];
  variable hash DayNumber := ["Sat"=>6,"Sun"=>7];
  variable list Days;

  print WeekendDays = getvector(Lb, .content);
  print vector["Sat","Sun"] = getvector(Lb, .content);
  print values(WeekendDays) = values(getvector(Lb, .content));
  print WeekendDays = keys(DayNumber);
  Days := WeekendDays;
  WeekendDays[0] := "??";
  print Days=WeekendDays;
  exit();
}
```

Ausgabe

```
false => Datentyp verschieden (list<>vector)
true
true
true
false => WeekendDays hat einen Standardwert, Days aber nicht
```

6.1.8 Uninitialisierte Variablen

Wie bisher auch verhält sich der IDM beim Zugriff auf uninitialisierte Werte unterschiedlich. Wird auf eine uninitialisierte lokale oder statische Variable zugegriffen, erhält man *nothing*. Beim Zugriff auf eine uninitialisierte globale Variable (entspricht einem Variablen-Objekt) oder ein benutzerdefiniertes Attribut wird der Zugriff mit einem Fehler *„cannot get value“* verweigert.

7 Funktionen (functions)

Die Funktionen stellen das Interface zu der verwendeten Programmiersprache dar. Das Schlüsselwort hierzu ist **function**. Anschließend müssen der Funktionstyp und der Funktionsname angegeben werden. Im Sinne von Funktionsprototypen müssen statt der Funktionsargumente die Datentypen der Parameter angegeben werden. Bei dieser Definition müssen die Funktionen nach ihrem späteren Einsatz unterschieden werden.

Prinzipiell werden die folgenden Arten von Funktionen unterschieden:

- Funktion, die aus den Regeln heraus aufgerufen wird
  Dieser Funktionstyp kann bis zu 16 beliebige Parameter haben, die bei der Deklaration der Funktion entsprechend angegeben werden müssen.
- Funktion, die direkt an einem Objekt hängt (Callback-Funktion)
  Diese direkt bei der Definition des Objektes über *.function* angegebenen Funktionen müssen als Callback-Funktion definiert werden. Dabei definiert der ISA Dialog Manager, welche Parameter diese Funktion erhält. Sie dürfen daher bei der Deklaration nicht angegeben werden. Bei der Definition dieser Funktion wird weiterhin angegeben, bei welche Benutzerereignissen sie aufgerufen werden soll.
- Funktion, die an einem Objekt Canvas hängt (Canvas-Funktion)
  Da dieses Objekt in Bezug auf den IDM eine große Ausnahme darstellt und es auch von der Anwendung anders behandelt werden muss, gibt es hierfür einen eigenen Funktionstyp `canvasfunc`. Ihre Parameter werden ebenfalls vom ISA Dialog Manager fest definiert und dürfen bei der Deklaration nicht angegeben werden.
- Datenfunktion
  Diese Funktion kann im *.datamodel*-Attribut bei allen Objekten angegeben werden, die auch benutzerdefinierte Attribute unterstützen. Eine Datenfunktion kann dabei die Rolle eines Datenmodells (Model-Komponente) erfüllen um die Präsentationsobjekte mit Datenwerten zu versorgen. Die Parameter einer Datenfunktion sind vom ISA Dialog Manager fest definiert und können vom Benutzer nicht verändert werden.
- Format-Funktion
  Diese Funktion wird bei der Definition von Format-Ressourcen sowie bei editierbaren Texten und Tablefields im Attribut *.formatfunc* angegeben. Ihre Parameter werden vom ISA Dialog Manager definiert und dürfen bei der Deklaration nicht angegeben werden.
- Nachlade-Funktion für Tablefields
  Diese Funktion wird bei der Definition von Tablefields im Attribut *.contentfunc* angegeben. Ihre Parameter werden vom ISA Dialog Manager definiert und dürfen bei der Deklaration nicht angegeben werden.

Zulässige Sprachen sind:

- C
- COBOL

Wird keine Sprache angegeben, wird C als Default genommen.

Zulässige Datentypen sind:

Datentyp	Wertebereich
anyvalue	Beliebiger undefinierter Datentyp; wird erst durch aktuelle Belegung definiert.
attribute	Datentyp der Attribute im IDM.
boolean	Boolescher Wert (true | false)
class	Klasse eines Objekts, z.B. "window".
datatype	IDM-Datentyp.
enum	Symbolische Konstante; Aufzählungstyp für verschiedene spezielle Attribute, z.B. "sel_row".
event	Ereignisart, z.B. "select".
index	Indexwert für 2-dimensionale Attribute [I,J].
integer	Ganzzahl ("long integer"), z.B. *42*.
method	Methode, z.B. "delete".
object	Objekt im Sinne des IDM (eigentliche Objekte, Ressourcen, Funktionen, Regeln…).
pointer	Zeiger aus der Anwendung.
string	Beliebig lange Zeichenkette (Null-terminiert), z.B. "Hallo".
void	Datentyp mit keinem Wert.

Anmerkung zum IDM für Windows

Der IDM-Datentyp *integer* entspricht dem Datentyp *long* in der C-Schnittstelle.

Der ISA Dialog Manager verwendet die **Pascal-Calling-Convention** als Default. Es ist ratsam, den Funktionsprototyp vom IDM schreiben zu lassen und einzubinden.

7.1 Funktion aus Regel

Diese Funktionen können explizit in den Regeln aufgerufen werden. Funktionen müssen einen der zulässigen Funktionstypen haben und sind mit maximal 16 Argumenten parametrisierbar. Der Datentyp eines jeden Argumentes kann innerhalb der Funktionsklammern angegeben werden; bei mehr als einem Argument sind die Datentypen durch Kommas zu trennen.

Syntax

```
{ export | reexport } function { <Sprache> } <Datentyp> <Funktionsname>
  ( { <Parameter> { , <Parameter> } } ) ;

Sprache ::= c | cobol

Parameter ::=
  <Datentyp> { [ <Größe> ] } { <Parametername> { := <Standardwert> } }
  { input } { output }
```

Innerhalb dieser Deklaration der Funktionsparameter können die Parameter-Werte definiert werden. Dafür stehen drei Möglichkeiten zur Verfügung:

- Nur Eingabewert (**input**)
  Soll ein Parameter nur als Eingabewert in eine Funktion dienen, so ist nach der Angabe des Datentyps das Schlüsselwort **input** optional, denn dieser Typ ist gleichzeitig der Default. Dieser Parameter kann durch diese Deklaration nur lesend in der Funktion verarbeitet werden.
- Nur Ausgabewert (**output**)
  Soll ein Parameter nur als Ausgabewert einer Funktion dienen, so muss nach der Definition des Datentyps das Schlüsselwort **output** angegeben werden. Dieses bedeutet, dass dieser Parameter in der Anwendung verändert werden muss und anschließend im IDM weiterbearbeitet werden kann.
- Sowohl Eingabe- als auch Ausgabewert (**input** **output**)
  Soll ein Parameter sowohl als Eingabewert in eine Funktion als auch als Ausgabewert dieser dienen, so müssen nach der Definition des Datentyps die Schlüsselworte **input** und **output** angegeben werden. Mit Hilfe dieser Deklaration hat man also die Möglichkeit, einen Attributwert in eine Funktion als Eingabeparameter zu geben, den Wert dort zu manipulieren und das Ergebnis dieser Manipulation wieder in dem entsprechenden Attribut anzuzeigen.

Parameter von Funktionen können auch mit Namen definiert werden, um z.B. besser erkennen zu können, wozu die einzelnen Parameter verwendet werden.

Beispiel

```
function c boolean ReadData(string Filename,
                            integer Index,
                            string Value output);
```

Es ist auch möglich, Defaultwerte für Parameter des Typs input zu definieren. Diese Parameter müssen dann beim Aufruf der Funktion nicht angegeben werden.

Zu beachten

Parameter mit Defaultwerten müssen am Ende der Parameterliste stehen.

Die Länge der Parameter kann auch an das System gegeben werden. Dies ist besonders relevant, wenn ein String (Zeichenkette) an die Anwendung übergeben werden soll, z.B. mit COBOL.

Die Größe muss in eckigen Klammern angegeben werden [<Größe>].

Zusätzlich können Parameter auch auf bestimmte Typen eingeschränkt werden, näheres siehe Kapitel „Gültigkeitsbereich zur besseren Typ-Prüfung“.

7.1.1 Alias-Name mit optionaler Codepage-Definition

Verfügbarkeit

Ab IDM-Version A.06.01.d

Bei der Definition von Anwendungsfunktionen kann nach den Funktionsparametern das Schlüsselwort **alias** gefolgt von einem String angegeben werden.

Syntax

```
{ export | reexport } function <language> <return_data_type>
    <function_identifier> ( { parameter [ , parameter ] } )
    alias <alias_string>  ; | <code_block>

alias_string ::=
    "<alias-name>{ ;<code_page> } | <code_page>"
code_page ::=
    [CP=]<code_page_identifier>
```

Die Alias-Angabe dient dazu, bei einer dynamischen Anbindung den korrekten Funktionsnamen (also ohne Reglementierung wie beim Funktionsidentifikator) vorzugeben.

Im Alias-String kann außerdem eine funktionsspezifische Codepage für die String-Verarbeitung vorgegeben werden. Diese ist mit „CP=“ einzuleiten, direkt gefolgt vom Codepage-Bezeichner (analog zu den CP-Defines aus **IDMuser.h**, aber ohne-Präfix „CP_“).

Ist zusätzlich ein Alias-Name definiert, muss die Codepage-Angabe mit Semikolon („;“) getrennt danach erfolgen.

Beispiele

```
function integer Atoi (string S) "myatoi;CP=utf8";
function string CurTime() "CP=utf16b";
```

7.1.2 Abfrage der Parameter einer Funktion zur Laufzeit

Die Funktion kann auch zur Laufzeit nach der Art seiner Parameter gefragt werden. Dazu müssen die Attribute *.type* für den Datentyp und *.input* für Inputwert bzw. *.output* für einen Ausgabewert jeweils mit dem Index des zu erfragenden Parameters angegeben werden. Wenn es sich bei dem Parameter

um einen Parameter vom Typ *string* handelt, kann mit dem Attribut *.size* auf die Größe des Parameters zugegriffen werden.

Damit sind folgende Attribute für eine Funktion zulässig

Attribut	Bedeutung
.count	Anzahl der Parameter
.language	Programmiersprache, in der die Funktion geschrieben werden soll
.type	Datentyp des Rückgabewerts der Funktion
.input[I]	Parameter I ist Eingabewert
.output[I]	Parameter I ist Ausgabewert
.size[I]	Größe des String-Parameters (nur relevant bei COBOL)
.type[I]	Datentyp des I-ten Parameters

Beispiel

```
dialog Test
{
}
function void MeineTestFkt(integer A input output,
  string[20] B output,
  boolean C input,
  pointer D input output);

on dialog start
{
  variable integer I;

  print MeineTestFkt.count;
  print MeineTestFkt.type;
  print MeineTestFkt.language;
  for I := 1 to MeineTestFkt.count do
    print MeineTestFkt.input[I];
    print MeineTestFkt.output[I];
    print MeineTestFkt.size[I];
    print MeineTestFkt.type[I];
  endfor
}
```

ergibt folgende Ausgabe im Tracefile

```
[XR] rule on dialog start (this=dialog Test)
    4
    void
```

```
    "default"
    true
    true
    0
    integer
    false
    true
    20
    string
    true
    false
    0
    boolean
    true
    true
    0
    pointer
[XD] rule on dialog start
```

7.2 Callback-Funktion

Eine Callback-Funktion wird aufgerufen, wenn ein Objekt, an das diese Funktion gebunden ist, ein Ereignis erhält, das bei der Definition dieser Callback-Funktion angegeben wurde. Die Callback-Funktion wird vor der Abarbeitung der Ereignisregeln aufgerufen. Die entsprechenden Ereignisregeln werden nur abgearbeitet, wenn die Callback-Funktion den Wert *true* zurückliefert.

Syntax

```
{ export | reexport } function { <Sprache> } callback <Funktionsname> ( )
    for <Ereignis> [ , <Ereignis> ] ;
```

Diese Funktionen können bei der Objektdefinition im Attribut *.function* angegeben werden. Bei der Definition der Funktion wird angegeben, bei welchen Benutzerereignissen – z.B. *select*, *activate*, *deselect* oder *charinput* – diese Funktion aufgerufen werden soll.

Die Parameter der Funktion sind vom IDM vorgegeben und können nicht verändert werden.

Beispiel

```
function callback ObjectCall () for close, move, activate;

window TestFenster
{
  .function ObjectCall;
  .xleft 17;
}
```

Diese Funktion wird dann aufgerufen, wenn der Benutzer das Fenster schließt, verschiebt oder aktiviert.

Siehe auch

Kapitel „Objektcallback-Funktionen“ im Handbuch „C-Schnittstelle - Grundlagen“

7.3 Canvas-Funktion

Canvas-Funktionen dienen zur Bearbeitung von speziellen Canvas-Ereignissen und erlauben das Darstellen beliebiger Grafiken innerhalb einer Canvas. Die jeweiligen Canvas-Ereignisse sowie die Art und Weise, wie Grafiken in der Canvas zur Anzeige gebracht werden, sind fenstersystemspezifisch.

Syntax

```
{ export | reexport } function { c } canvasfunc <Funktionsname> ( ) ;
```

Diese Funktionen können bei der Definition von Canvas-Objekten im Attribut *.canvasfunc* angegeben werden.

Die Parameter der Funktion sind vom IDM vorgegeben und können nicht verändert werden.

Beispiel

```
function canvasfunc HandleCanvas ();

window CanvasFenster
{
  .xleft 17;
  child canvas Dynamic
  {
    .canvasfunc HandleCanvas;
    .width 100;
  }
}
```

Siehe auch

Kapitel „Canvas-Funktionen“ im Handbuch „C-Schnittstelle - Grundlagen“

7.4 Datenfunktion

Eine Datenfunktion kann die Rolle eines Datenmodells (Model-Komponente) erfüllen um die Präsentationsobjekte mit Datenwerten zu versorgen. Aufgerufen wird die Funktion wenn eine Synchronisierung der Datenwerte erfolgen soll, also entweder um Datenwerte vom Datenmodell zu holen oder um sie zuzuweisen.

Syntax

```
{ export | reexport } function { <Sprache> } datafunc <Funktionsname> ( ) ;
```

Diese Funktionen können im *.datamodel*-Attribut bei allen Objekten angegeben werden, die auch benutzerdefinierte Attribute unterstützen.

Die Datenfunktion kann in C/C++ geschrieben werden, ab IDM-Version A.06.01.d wird mit der COBOL-Schnittstelle für MICRO FOCUS VISUAL COBOL auch COBOL als Applikationssprache unterstützt.

Die Implementierung der Datenfunktion kann auf der DDM-Serverseite liegen.

Die Parameter der Funktion sind vom IDM vorgegeben und können nicht verändert werden.

Siehe auch

Kapitel „Datenfunktion“ im Handbuch „C-Schnittstelle - Grundlagen“

Kapitel „Datenfunktionen“ im "Handbuch „COBOL-Schnittstelle“

Kapitel „Datenmodell“ im Handbuch „Programmiertechniken“

Attribute .dataget[attribute], .datamodel[attribute], .dataoptions[enum], .dataset[attribute]

7.5 Format-Funktion

Format-Funktionen dienen zur Formatierung von editierbaren Text- und Tablefieldinhalten.

Syntax

```
{ export | reexport } function formatfunc <Funktionsname> ( ) ;
```

Diese Funktionen können bei der Definition von Format-Ressourcen angegeben werden. Bei der Definition von editierbaren Texten und Tablefields können sie auch im Attribut *.formatfunc* angegeben werden.

Die Parameter der Funktion sind vom IDM vorgegeben und können nicht verändert werden.

Siehe auch

Kapitel „Format-Funktionen“ im Handbuch „C-Schnittstelle - Grundlagen“

7.6 Nachlade-Funktion

Nachlade-Funktionen dienen zum dynamischen Nachladen des Tablefieldinhalts und werden vom Dialog Manager aufgerufen, wenn in einem Tablefield vom Benutzer so gescrollt wird, dass der danach sichtbare Bereich Zeilen oder Spalten enthält, die keinen Inhalt haben. Die Nachlade-Funktion kann dann den fehlenden Inhalt in das Tablefield einfüllen und gegebenenfalls nicht mehr benötigte Inhalte außerhalb des sichtbaren Bereich löschen.

Syntax

```
{ export | reexport } function { <Sprache> } contentfunc <Funktionsname> ( )
 ;
```

Diese Funktionen können bei der Definition von Tablefields im Attribut *.contentfunc* angegeben werden.

Die Parameter der Funktion sind vom IDM vorgegeben und können nicht verändert werden.

Siehe auch

Kapitel „Nachlade-Funktionen“ im Handbuch „C-Schnittstelle - Grundlagen“

7.7 Simulation von Funktionen

Funktionen, die nicht an den ISA Dialog Manager gebunden sind, können durch Regeln simuliert werden. Beim Entwickeln großer Anwendungen stehen dem Entwickler der graphischen Benutzeroberfläche oftmals nicht alle Funktionen zur Verfügung, die er zum Ablauf bzw. Simulation seiner Oberfläche benötigt. Dies behindert den Entwickler oder lässt ihn nur Teile seiner Anwendung testen. Um Dialoge unabhängig von den nicht vorhandenen Funktionen ablauffähig zu machen, können diese Funktionen durch Regeln simuliert werden. Funktionen, die an den Dialog Manager angebunden wurden, werden weiterhin wie gewohnt aufgerufen.

Folgende Funktionen können simuliert werden:

- Funktionen, die aus Regeln heraus aufgerufen werden
- Callback-Funktionen

Bei der Definition der Funktion wird die Regel mitangegeben. Statt die Funktionsdefinition wie gewohnt mit einem Semikolon abzuschließen, wird eine geschweifte Klammer geöffnet, um danach die Simulationsregel zu definieren. Die Simulationsregel hat die gleichen Eigenschaften wie eine benannte Regel.

Beispiel einer Funktionsdefinition ohne Simulationsregel:

```
function c boolean CheckOnline();
```

Beispiel derselben Funktion mit Simulationsregel:

```
function c boolean CheckOnline()
{
return( true );  // always online simulation
}
```

Die Simulationsregeln lassen sich wie benannte Regeln benutzen. Sie besitzen die identischen Parameter wie die simulierte Funktion.

Beispiel

```
function c void ConvertDate(boolean CurrentDate input,
```

```
    string Date input output)
{
  // Date will be in format YYYYMMDD and we are to lazy to do
  // this here, it will be sufficient for us to do this with
  // two different dates
  if ( CurrentDate ) then
    Date := "11.11.2011";    // simulate always with this
                             // date as current
  else
    Date := "1.4.2010";      // no conversion supplied,
                             // this is sufficient
  endif
}

function cobol boolean GetExchangeRate( string From input,
    string To input, string ExchangeRate output)
{
    case From
    in "EUR":
        case To
        in "EUR":    ExchangeRate := "1.000";
        in "USD":    ExchangeRate := "1.3249";
        in "GBP":    ExchangeRate := "0.8550";
        ...     // many more currencies
        endcase
    in "USD":
        case To
        in "USD":    ExchangeRate := "1.000";
        in "EUR":    ExchangeRate := "0.7548";
        ...
    ...
    endcase
    return (true);
}

record Currency
{
    string Which;
    string EUR;
    string USD;
    string CHF;
    ...
}

function cobol boolean GetCurrencyRates(record Currency
    input output)
```

```
{
    case Currency.Which
    in "EUR":
        Currency.EUR := "1.000";
        Currency.USD := "1.3249";
        ...
    in "USD":
    ...
}
```

Diese Simulation der Funktionen dürfte den meisten Anwendungen genügen. Eine vollständige Ersetzung der Funktionen ist in den allermeisten Fällen auch nicht nötig, da es reicht, den Dialogablauf sinnvoll zu steuern.

Der ISA Dialog Manager ermöglicht es dem Entwickler, Funktionen für Ereignisse direkt anzugeben. Diese werden beim Auftreten des angegebenen Ereignisses direkt aufgerufen. Bei diesen Callback-Funktionen können ebenfalls Simulationsfunktionen angegeben werden. Damit kann hier ebenfalls eine für die Dialogsimulation und Testverfahren vorteilhafte Funktionssimulation programmiert werden. Die simulierte Callback-Funktion hat dieselben Eigenschaften wie eine Ereignisregel. Die zugehörigen Ereignisregeln werden, falls vorhanden, immer nach der simulierten Callback-Funktion ausgeführt.

Beispiel

```
function callback PushbuttonSelect() for select, focus
{
    if ( thisevent.event[select] ) then
        // the simulation code for select
    endif
    if ( thisevent.event[focus] ) then
        // the simulation code for focus
    endif
}
default pushbutton
{
    .function PushbuttonSelect;
}
on PUSHBUTTON select
{
    // is also executed
}
```

Dadurch kann der Entwickler den Dialogablauf steuern und auf die Ereignisse reagieren ohne dass er die Ereignisregeln zusätzlich implementieren muss.

7.8 Funktionen in modularisierten Dialogen

Für die Benutzung von Funktionen in modularisierten Dialogen stehen die Attribute *.**application*** am ***import***-Objekt sowie *.masterapplication* am ***modul***- bzw. ***dialog***-Objekt zur Verfügung . Damit können alle Funktionen eines Modules von„außen“ an eine spezielle Applikation gekoppelt werden. Die Vorgehensweisen für beide Varianten - für **import** und für **use** - sind im Kapitel „Programmiertechniken“/ „Modularisierung“ / „Objekt Application“ beschrieben.

8 Globale Variablen (variables)

Der Aufbau einer Variablendeklaration setzt sich aus dem Schlüsselwort **variable**, dem Datentyp, dem Variablenidentifikator und dem Endezeichen (Terminator) **;** zusammen. In einer Anweisung können mehrere Variablen mit demselben Datentyp deklariert werden. Dabei sind die Identifikatoren der Variablen durch **,** (Komma) zu trennen.

Wird die Variable direkt im Dialog oder Modul und nicht innerhalb einer Regel oder Methode (siehe auch Kapitel „Lokale Variablen“) definiert, handelt es sich um eine **globale** Variable.

Globale Variablen werden im IDM ähnlich wie Objekte behandelt, sodass nicht nur der Inhalt der Variablen sondern auch die Variable selbst zur Laufzeit geändert werden kann (siehe Kapitel „Änderbare Attribute von globalen Variablen“).

8.1 Variablendefinition

Syntax

```
{ export | reexport } variable <Datentyp> <Variablenname>
  [ , <Variablenname> ] ;
```

Es stehen folgende Datentypen zur Verfügung:

Datentyp	Wertebereich
anyvalue	beliebiger undefinierter Datentyp; wird erst durch aktuelle Belegung definiert.
attribute	Datentyp der Attribute im IDM.
boolean	Boolescher Wert (true | false).
datatype	IDM-Datentyp.
class	Klasse eines Objekts, z.B. "window".
enum	Symbolische Konstante; Aufzählungstyp für verschiedene spezielle Attribute, z.B. "sel_row".
event	Ereignisart, z.B. "select".
index	Indexwert für 2-dimensionale Attribute [I,J].
integer	Ganzzahl ("long integer"), z.B. *42*.
method	Methode, z.B. "delete".
object	Objekt im Sinne des DM (eigentliche Objekte, Ressourcen, Funktionen, Regeln…).

Datentyp	Wertebereich
pointer	Zeiger aus der Anwendung.
string	beliebig lange Zeichenkette (Null-terminiert), z.B. "Hallo".

8.2 Variablendefinition mit Initialisierung

Die folgende Definition initialisiert die erzeugte Variable sofort mit einem Wert. Der Datentyp des Wertes muss selbstverständlich mit dem Datentyp der Variablen identisch sein.

Syntax

```
{ export | reexport } variable <Datentyp> <Variablenname> := <Wert>
  [ , <Variablenname> := <Wert> ] ;
```

Objektreferenzen, Zahlen, Farben, Strings und boolesche Werte, d.h. alles was vom IDM unterstützt wird, kann in den Variablen vom Typ *anyvalue* gespeichert werden.

Beispiel

```
variable integer Fuellstand;
variable boolean Ventil_Zustand := true;
variable string  Ventil_Name := "Hauptventil";
variable enum    Mode;
variable object  PbSchalter;  // PbSchalter sei ein Pushbutton
```

Ein String ist immer eine in doppelte Hochkommas eingeschlossene Zeichenkette.

```
Ventil_Name := "Hauptventil";
```

In diesem Beispiel ist das Wort "Hauptventil" ein String.

8.3 Konfigurierbare Variablen

Das zusätzliche Schlüsselwort **config** kennzeichnet eine globale Variable als konfigurierbar, d.h. ihr Wert kann in einer Konfigurationsdatei gesetzt werden.

Syntax

```
{ export | reexport } { config } variable <Datentyp>
  <Variablenname> { := <Wert> } [ , <Variablenname> { := <Wert> } ] ;
```

Die Konfigurationsdatei wird mit den Funktionen **loadprofile()**, DM_LoadProfile oder DMcob_LoadProfile geladen.

Siehe auch

Attribut .configurable

Kapitel „Konfigurationsdatei“ im Handbuch „Entwicklungsumgebung“

8.4 Variable vor Änderungen schützen

Durch Verwendung des Schlüsselworts **constant** statt **variable** oder des Setzens des Attributs *.constant* an der Variablen auf *true* kann eine Änderung des Inhalts der globalen Variablen verhindert werden.

Syntax

```
{ export | reexport } constant <Datentyp> <Variablenname> := <Wert>
  [ , <Variablenname> := <Wert> ] ;
```

Siehe auch

Attribut .constant

Beispiel

```
dialog D

variable integer V := 123;
constant integer C := 456;

on dialog start
{
  C:=V;      // Evaluationsfehler da C nicht veränderbar
  V := 234;  // OK
  V.constant := true;
  V := 345;  // Fehler – Variable nun schreibgeschützt
}
```

8.5 Änderbare Attribute von globalen Variablen

Bei globalen Variablen können folgende Attribute dynamisch zur Laufzeit verändert werden:

Attribut	Wertebereich	Bedeutung
.value	anyvalue	Wert
.type	datatype	Datentyp
.constant	boolean	Änderbarkeit des Inhalts

Um eine Variable und nicht ihren Inhalt anzusprechen, ist das Attribut *self* zu verwenden. Danach kann dann das Attribut folgen, das man von der Variablen ansprechen möchte.

Beispiel

```
variable integer I := 2;

print I.self.type    // integer
print I.self.value   // 2

I.self.type := boolean;
print I.self.type;  // boolean
```

Hinweis

Dies ist nur bei globalen, nicht jedoch bei lokalen Variablen (Variablen, die innerhalb von Regeln und Methoden deklariert werden), möglich.

9 Gültigkeitsbereich zur besseren Typ-Prüfung

Benutzerdefinierte Attribute, Variablen, lokale Variablen (in Regeln bzw. Methoden), Parameter und Rückgabetypen von Regeln, benutzerdefinierten Methoden, extevent-Regeln und Applikationsfunktionen können um einen Gültigkeitsbereich erweitert werden. Dies wurde eingeführt, um eine striktere **Typ**-Prüfung schon beim Laden zu ermöglichen, um so frühzeitig Fehler in Dialogen aufzuzeigen. Konkrete Anwendung - die Restriktion von Attributen/Variablen auf Objekte die von einem bestimmten Modell abgeleitet sind.

Der Gültigkeitsbereich ist ein Zusatz zu einem „Wertebehälter“ wie einem Attribut oder einer Variablen (inkl. Parameter) und stellt keinen eigenen Typ dar. Mit einem definierten Gültigkeitsbereich kann sichergestellt werden, dass nur Setzungen erlaubt sind die dem Gültigkeitsbereich entsprechen, also der „Wertebehälter“ immer einen konsistenten Wert behält. Prüfungen hierzu passieren während des Ladens sowie beim Ausführen von Regelcode.

Zweitens kann der Gültigkeitsbereich auch den weiteren Zugriff vom „Wertebehälter“ aus einschränken. Mit Zugriff ist dabei der Zugriff auf ein Kind, Attribut oder Methode gemeint. Ein Zugriff kann dabei auch zur Rückgabe eines Wertes mit einem abgeleiteten Gültigkeitsbereich führen.

Beispiel

```
dialog D
// Basismodell mit einem Kind
model window MWi {
  statictext StHeader {}
}
// Modell abgeleitet und erweitert
MWi Wi {
  .StHeader {
    integer Width := 10;
  }
  edittext Et {
  }
}
// Ein zweites Modell
model window MWi2 {}
// Instanz aus zweitem Modell
MWi2 Wi2 {
  // Attribut das nur von MWi2 abgeleitetes Objekt beinhalten darf
  object[MWi2] Ref := Wi; // Syntaxfehler da Gültigkeitsbereich verletzt
}
// Regel die nur auf von MWi abgeleitete Objekte angewendet werden darf
rule void SetHeader(object[MWi] O, string Header) {
  O.StHeader.text := Header;  // O kann niemals Wi2 sein!
```

```
  O.Et.xauto := 0; // Syntaxfehler wegen Zugriffsverletzung
  O.StHeader.Width := 20; // Syntaxfehler wegen Zugriffsverletzung
}
on dialog start {
  variable object[MWi] O; // Variable nur für von MWi abgeleitete Objekte
  SetHeader(Wi,”First”);
  SetHeader(Wi2,”Second”); // Syntaxfehler da Gültigkeitsbereich verletzt
  O := D.child[2]; // Dynamische Verletzung des Gültigkeitsbereichs
}
```

9.1 Beschränkungen

Folgende Beschränkungen sind momentan für die aufgelisteten Datentypen möglich:

Datentyp	Gültigkeitswert	Wert- und Zugriffsbeschränkungen, Ableitung des Gültigkeitsbereiches
string	Integerwert (>0)	Keine Einschränkungen. Dieser Gültigkeitswert dient zur bisherigen Repräsentation der String-Größe die für die Generierung von Trampolin-Dateien für COBOL nötig ist.
object	Objekt	Wert darf null sein oder muss identisch mit dem Gültigkeitswert bzw. davon Abgeleitet (vergleichbar zu instance_of-Methode) sein. Zugriffsbeschränkungen bei benutzerdefinierten Attributen, Kinder und Methoden auf diejenigen die am Objekt vorhanden sind. Ableitung des Gültigkeitsbereiches beim Zugriff auf ein Kind-Objekt.
object	Klasse	Wert darf null sein oder muss ein Objekt der entsprechenden Klasse sein. Keine Zugriffsbeschränkungen.

Ein Gültigkeitsbereich kann für folgende Fälle definiert werden:

1. Datentyp eines benutzerdefinierten Attributes
2. Index-Datentyp eines benutzerdefinierten Assoziativen Feldes
3. Datentyp einer globalen Variablen
4. Rückgabetyp einer Applikationsfunktion, benannten Regel oder benutzerdefinierten Methode
5. Parametertypen von Applikationsfunktionen, benannten Regeln und benutzerdefinierten Methoden sowie Ereignisregeln
6. Datentyp von lokale Variablen in benannten Regeln und benutzerdefinierten Methoden sowie Ereignisregeln

Beispiel

```
dialog D
model window MWi {}                                    // Folgende Nummern
MWi Wi {                                               // beziehen sich auf
record Rec {                                           // vorangegangene Liste
  object[MWi] Ref := Wi;                               // 1)
  boolean AssArray[object[MWi]];                       // 2)
  .AssArray[Wi] := true;
}
variable object[MWi] GlobaleVariable := Wi;            // 3)
function object[color] GetBgc(object[MWi] Window);     // 4), 5)
rule object[MWi] GetModel(object[Wi] P) {              // 4), 5)
  return P.model;
}
on Wi extevent 123 (object[MWi] P) {                   // 6
  variable object[MWi] V := P;
}
```

9.2 Zugriffs- und Werteprüfung

Statische Definitionen die Gültigkeitsbereiche beinhalten werden grundsätzlich zur Ladezeit geprüft, ebenso Zuweisungen und Zugriffe in Regeln die als Konstant erkennbar sind. In diesem Sinne sind auch Objektpfad (z.B. „Wi“ im Beispiel unten) als „konstant“ Wert anzusehen der zur Ladezeit geprüft werden kann.

Dynamische Prüfungen passieren grundsätzlich bei Zuweisungen auf Wertebehältern mit gesetztem Gültigkeitsbereich bzw. Werteholungen oder Zugriffen von diesen heraus und erzeugen ein *Fail*, können also über Klammerung in ein fail(…)-Konstrukt abgefangen werden. Aufrufparameter und Rückgaben von benutzerdefinierten Methoden u.U. nur dynamisch geprüft. Dies begründet sich aus der nicht zwangsläufig zum Ladezeitpunkt vorliegenden Gültigkeitsinformationen bei der Verwendung von Attributen und Methoden von importierten Objekten.

Grundsätzlich sind vordefinierte „Variablen“ wie `this` oder `thisevent` nicht an einen Gültigkeitsbereich gebunden. Ebenso wenig wie Rückgabetypen und Parameter von vordefinierten Attributen, Methoden und Builtin-Funktionen. Insofern hat die Verwendung mit besonderer „Umsicht“ zu geschehen.

Da der Gültigkeitsbereich keinen Typ darstellt, gibt es auch keine Typprüfung oder Typkonvertierung. Ein „Casting“ wie man es von anderen Programmiersprachen kennt ist meist nicht notwendig bzw. kann durch Zwischenspeicherung auf eine Variable vom Datentyp *object* geschehen.

Die Zugriffsprüfung betrachtet immer nur gerade aktuellen Zustand (Objekte, vorhandene Kinder und Attribute).

Nachfolgendes kurzes Beispiel beinhaltet korrekte und fehlerhafte Verwendungen und zeigt den erwarteten Prüfungszeitpunk (Ladezeit/Laufzeit) auf.

```
dialog D
```

```
:
model window MWi { child edittext Et{} }
MWi Wi { child pushbutton Pb {} }
model window MWi2 { }
MWi2 Wi2 { checkbox Cb {} }
:
variable object[MWi] I1 := Wi
variable object[MWi2] I2 := Wi2
variable object O;
:
O := I1; // Gültigkeitsprüfung nicht notwendig
I1 := O; // Gültigkeitsprüfung zur Laufzeit
I2 := I1; // Gültigkeitsprüfung zur Ladezeit – GÜLTIGKEITSFEHLER!
I2 := W1; // Gültigkeitsprüfung zur Ladezeit – GÜLTIGKEITSFEHLER!
:
// Gültiger/erlaubter Zugriff auf Kind-Objekt:
print I1.Et;
print O.Pb;

// ZUGRIFFSFEHLER auf Kind-Objekt und deren Prüfungszeitpunkt:
print I1.Pb;     // Zugriffsprüfung zur Ladezeit da I1 Gültigkeitbereich hat
print O.Unkown;  // Zugriffsprüfung zur Laufzeit
print I2.Cb;     // Zugriffsprüfung zur Ladezeit
print I2.Unkown; // Zugriffsprüfung zur Ladezeit
print getvalue(I2,. Unkown); // Zugriffprüfung zur Laufzeit
```

9.3 Ableitung des Gültigkeitsbereiches

Der Gültigkeitsbereich wird beim Zugriff auf Kinder/Attribute weiter abgeleitet.

Dies geschieht beim Zugriff auf ein Attribut mit Gültigkeitsbereich, sowie wenn ausgehenden von einem Wertebehälter mit Gültigkeitsbereich ein Zugriff auf ein Kind-Objekt erfolgt.

```
dialog D
model window MWi {
  child groupbox Gb {

    child edittext Et {
      rule void Apply() {}
    }
  }
}
MWi Wi {
  object[MWi] Ref := MWi;
  .Gb {
    checkbox Cb { rule void Apply(); }
  }
  rule void Apply() {
```

```
    variable object[MWi] This := this;
    This.Gb.Et:Apply();
    This.Gb.Cb:Apply(); // Zugriffsfehler da Cb nicht in MWi.Gb vorhanden
    this.Ref.Gb.Cb:Apply();// Zugriffsfehler da Cb nicht in MWi.Gb da
  }
}
```

9.4 Erweiterung der IDM-Syntax

Syntaktische Erweiterungen/Änderungen an der Regelsprache sind wie folgt (rote Markierung):

Allgemeine Definitionen

```
<Datentyp> ::= anyvalue | attribute | boolean | class | enum | event |
        index | integer | method | object | pointer
<Parameterart> ::= { input } { output }
<Rückgabetyp> ::=  void | <Datentyp>
<Gültigkeit> ::= '[' <Gültigkeitswert> ']'
<Gültigkeitswert> ::= <Klasse> | <Objekt> | null | <Integerwert>
<Objekt> ::= <Identifikator> [ .<Identifikator> { :'[' <Integerwert> ']' } ]
```

Definitionen für benannte Regeln und Methoden

```
<Regel> ::= { export | reexport } { public } { extern } rule
        <Rückgabetyp> { <Gültigkeit> } <Identifikator>
        '(' { <Parameter> [ , <Parameter> ] } ')' { <ReposId> }
        <OptionalerProgrammblock>
<Parameter> ::=
        <Datentyp> { <Gültigkeit> } <Identifikator> { := <Wert> }
<Parameterart>
<OptionalerProgrammblock> ::= ';' | <Programmblock>
```

Definitionen für externe Ereignisse

```
<ExterneEreignisregel> ::= on <Objekt> extevent <KonstanterWert>
        { '(' { <Parameter> [ , <Parameter> ] } ')' }
        { <Regelbedingung> }
        <Programmblock>
```

Definitionen für lokale Variablen

```
<Variablenanweisung> ::= { static } variable <Variable> [ ',' <Variable> ]
';'
<Variable> ::= <Datentyp> { <Gültigkeit> } <Identifikator> { := <Wert> }
```

Definitionen für globale Variablen

```
<GlobaleVariable> ::= { export | reexport } { config } <Variablenart>
<Datentyp>
        { <Gültigkeit> } <Identifikator> { <ReposId> } { := <Wert> } ';'
<Variablenart> ::= constant | variable
```

Definitionen für benutzerdefinierte Attribute

```
<Attributdefinition> ::=
        <Datentyp> { <Gültigkeit> } <Identifikator> <Attributart> ';'
<Attributart> ::= <SkalaresAttribut> | <IndiziertesAttribut> |
        <AssoziativesAttribut> | <ShadowAttribut>
<SkalaresAttribut> ::= { := <Wert> } ';'
<IndiziertesAttribut> ::= '[' <Integerwert> ']' { := <Wert> }
<AssoziativesAttribut> ::= '[' <Datentyp> { <Gültigkeit> } ']' { := <Wert> }
<ShadowAttribut> ::=
        shadows { instance } <Objekt> <Attribut> { '[' <Index> ']' }
```

Definitionen für Applikationsfunktionen

```
<Applikationsfunktion> ::= { export | reexport } function { <Sprache> }
        <Rückgabetyp> { <Gültigkeit> } <Identifikator>
        '(' { <Funktionsparameter> [ , <Funktionsparameter> ] } ')'
        { alias <String> } { <ReposId> } <OptionalerProgrammblock>
<Funktionsparameter> ::= <Parameter> | <RecordParameter>
<RecordParameter> ::= record <Objekt> { := <Wert> } <Parameterart>
```

9.5 Attribute zur Nutzung mit Gültigkeitsbereichen

Die folgenden Attribute sind zur dynamischen Abfrage der Gültigkeitsbereiche bestimmt. Näheres ist in der „Attributreferenz“ bei der jeweiligen Attributbeschreibung zu entnehmen.

- scope[attr]
  Erfragen des Gültigkeitsbereichs bei benutzerdefinierten Attributen.
- indexscope[attr]
  Auslesen des Gültigkeitsbereichs des Index von benutzerdefinierten assoziativen Attributen (assoziativen Arrays).
- typescope
  Auslesen des Gültigkeitsbereichs des Rückgabetyps von benutzerdefinierten Rückgabetypen.
- typescope[l]
  Erfragen des Gültigkeitsbereichs von Parametern.

10 Aktionen im Programmblock

Die Funktionalität der Regeln wird im Programmblock beschrieben, der in Klammern gesetzt ({ }) der Ereigniszeile folgt. Mit dieser Regelkomponente kann die gesamte Dialogausführung inklusive der Funktionalität vom DM beschrieben werden.

Aktionen können z.B. sein:

- Ändern von Attributen bzw. Attributwerten,
- Aufruf von Anwendungsfunktionen,
- Aufruf von Built-in-Funktionen,
- "if-then-else-endif"-Konstrukte,
- Sub-Regeln,
- Variablen.

Der Programmblock beinhaltet Zuweisungen jeder Art (logische, arithmetische und Zeichenketten), Berechnungen und Mechanismen zur bedingten Ausführung von Anweisungen.

Beispiel

```
on PB_Show select                          /* 1 */
{                                          /* 2 */
  if ( Hauptfenster.visible = false )      /* 3 */
  then                                     /* 4 */
    Hauptfenster.visible := true;          /* 5 */
  endif                                    /* 6 */
}                                          /* 7 */
```

In Zeile 1 (Ereigniszeile) wird definiert, dass bei Selektion des Pushbuttons „PB_Show“ der nachfolgende Programmblock in den Zeilen 2 bis 7 ausgeführt werden soll.

Die geschweiften Klammern in Zeile 2 und 7 begrenzen den zu der Ereigniszeile gehörenden Programmblock. Zeile 3 löst eine bedingte Programmbearbeitung aus. Sie führt zur Bearbeitung der Anweisungen in der Zeile 5, wenn der Ausdruck in den runden Klammern *true* ist. Ist dieser Klammerausdruck *false*, so wird die Programmzeile 5 **nicht** ausgeführt.

Zeile 5 weist dem Objektattribut *.visible* des Hauptfensters den Wert *true* zu, dieses Fenster wird also sichtbar.

10.1 Genereller Aufbau eines Statements

Jede Anweisung in der Regelsprache wird durch ein Semikolon abgeschlossen. Dieses Zeichen dient als Endekennzeichen eines Statements. Als Ausnahme sind hier nur das Schlüsselwort **end** und die davon abgeleiteten Worte wie **endif**, **endfor**, **endcase**, **endwhile** zu sehen, denn hier braucht kein Semikolon stehen.

Syntax

```
<Anweisung> ;
```

Fehlt dieses Semikolon, so gibt das System in der Regel für die nachfolgende Zeile einen Fehler aus.

10.2 Kommentierung von Anweisungen

Kommentare können fast beliebig in den Regelaufbau eingebaut werden. Kommentare werden durch

```
!!
```

eingeleitet und kennzeichnen, dass die gesamte nachfolgende Zeile als Kommentar zu betrachten ist. Kommentare dürfen vor der Ereigniszeile und vor jedem Statement im Regelinneren stehen. Sie dürfen nicht vor der Deklaration lokaler Variablen und innerhalb von Expressions auftauchen.

Beispiel

```
!!In Liste wird etwas selektiert, darum werden die
!!Pushbuttons Löschen und Informationen freigeschaltet
on LListe select
{
  !! Umschalten des Löschen-Pushbuttons
  this.window.PLoeschen.sensitive := true;

  !! Umschalten des Info-Pushbuttons
  this.window.PInfos.sensitive := true;
}
```

Kommentare können natürlich auch bei der Definition der Attribute stehen. Es können die eingebauten Attribute wie etwa .visible oder .text kommentiert werden. Ebenso können indizierte Attribute kommentiert werden. Das Gleiche gilt für Benutzerdefinierte Attribute.

Beispiel

```
!! Beispiel
dialog Kommentare

!! Kommunikationsmodell fuer Daten
model record MRComm
{
  !! eindeutige Kennung des entsprechenden Records
  integer ID;
  !! Datenbankkennung
  string S[3];
  !! Host
  .S[1] := "aldadin";
  !! Database
  .S[2] := "inforaclemix";
```

```
  !! query language
  .S[3] := "SQL";
}
model listbox MList
{
  !! Erster Eintrag
  .content[1] "Ja";
  !! Zweiter
  !! Eintrag mit zwei Zeilen
  .content[2] "Es geht";
  !! "normale" gehen auch
  .width 100;
}
!! Ende des Dialoges
```

Zu beachten ist bei Benutzung des IDM Editors und einer Kommentierung im Textfile allerdings folgendes:

Der Dialog Manager unterscheidet bei !!-Kommentaren, welche vor Attributen gesetzt werden, nicht zwischen Attribut-Deklaration und Defaultwert-Zuweisung, da diese bei einer Ausgabe ohnehin in einer Zeile stehen.

Im Gegensatz zu den Versionen vor A.05.01.e werden nun !!-Kommentare für das gleiche Attribut zusammengeführt, so dass z.B. alle Kommentare von Mehrfachsetzungen als auch getrennte Deklarations- und Defaultwertsetzungen zusammen mit dem letzten Wert erhalten bleiben.

Vorsicht ist bei Kommentaren zu shadow-Attributen gegeben. Diese Kommentare können nur für die Attributdeklaration erhalten werden, da der eigentliche Wert sich meist an einem anderen Objekt gespeichert wird.

10.3 Operatoren in der Regelsprache

10.3.1 Zuweisungsoperatoren

Durch einen Zuweisungsoperator werden der linken Seite des Operators der Wert der rechten Seite zugewiesen. Ein Zuweisungsoperator wird in der Regelsprache durch einen Doppelpunkt gefolgt von einem Gleichheitszeichen

```
:=
```

oder durch zwei Doppelpunkte gefolgt von einem Gleichheitszeichen

```
::=
```

dargestellt.

Mit der Zuweisung „:=“ werden immer *changed*-Ereignisse erzeugt. Mit dem Zuweisungsoperator „::=“ werden **keine** *changed*-Ereignisse erzeugt.

Bei einer Zuweisung ist wichtig, dass die Datentypen auf beiden Seiten übereinstimmen, d.h. einem Zahlenwert kann nur ein Zahlenwert und z.B. kein String zugewiesen werden.

Beispiel

```
dialog D
record R
{
  integer I;
  on .I changed
  {
    print "Event!";
  }
}

on dialog start
{
  !! loest Ereignis aus
  R.I := 1;
  !! loest kein Ereignis aus
  R.I ::= 2;
}
```

10.3.2 Vergleichsoperatoren

In der Regelsprache können wie in anderen Programmiersprachen auch, Werte mit Hilfe von Vergleichsoperatoren verglichen werden.

Dabei existieren in der Regelsprache folgende Vergleichsoperatoren:

Operatorzeichen	Bedeutung
<	kleiner
<=	kleiner gleich
=	gleich
>=	größer gleich
>	größer
<>	ungleich

Mit Hilfe dieser Ausdrücke können *integer*-Werte verglichen werden. Für die Operatoren „=“ und „<>“ sind alle Datentypen zugelassen.

10.3.3 Arithmetische Operatoren

In der Regelsprache können wie in anderen Programmiersprachen auch Ausdrücke oder Werte mit Hilfe von arithmetischen Operatoren verknüpft werden.

Dabei existieren in der Regelsprache folgende arithmetische Operatoren:

Operatorzeichen	Bedeutung
+	Addition von zwei Werten.
-	Subtraktion von zwei Werten.
/	Division von zwei Werten.
*	Multiplikation von zwei Werten.
%	Modulo-Bildung von zwei Werten (Rest bei Division).

Alle diese Operationen können auf *integer*-Werte angewendet werden.

Mit dem Operator „+“ können auch zwei Strings verkettet werden.

10.3.4 Logische Operatoren

In der Regelsprache können Ausdrücke mit Hilfe von logischen Operatoren verknüpft werden.

Dafür existieren folgende logische Operatoren:

Operator	Bedeutung
`and`	Durch diesen Operator werden zwei Ausdrücke durch ein logisches UND miteinander verknüpft. Dabei werden, auch wenn das Ergebnis schon feststeht, alle Teilausdrücke ausgewertet und erst abschließend das Ergebnis gebildet.
`andthen`	Wie `and`, wertet den Ausdruck allerdings nur so weit aus, bis der restliche Teilausdruck das Ergebnis nicht mehr beeinflussen kann. Dann wird die Auswertung abgebrochen und das Resultat zurückgegeben.
`or`	Durch diesen Operator werden zwei Ausdrücke durch ein logisches ODER miteinander verknüpft. Dabei werden, auch wenn das Ergebnis schon feststeht, alle Teilausdrücke ausgewertet und erst abschließend das Ergebnis gebildet.
`orelse`	Wie `or`, wertet den Ausdruck allerdings nur so weit aus, bis der restliche Teilausdruck das Ergebnis nicht mehr beeinflussen kann. Dann wird die Auswertung abgebrochen und das Resultat zurückgegeben.
`not`	Durch diesen Operator wird ein Ausdruck negiert.

Der Operator `andthen` ist zum Beispiel nützlich, um erst zu prüfen, ob ein Objekt angelegt wurde, bevor eine Aktion mit dem Objekt ausgeführt wird:

```
variable object O := null; !! noch nicht instanziiert

if O <> null andthen O:TuWas() then
```

Hier kommt es nicht zu einer Fehlermeldung, da `andthen` die Auswertung abbricht nachdem die Evaluierung von *O <> null* als Ergebnis *false* liefert.

10.4 Klammern in der Regelsprache

In der Regelsprache existieren drei Arten von Klammern mit sehr unterschiedlichen Bedeutungen.

Klammerart	Bedeutung
{ }	Mit Hilfe der geschweiften Klammern werden die Regelkörper zusammengefasst. Innerhalb dieser Klammer wird die eigentliche Regel, also der Teil, der etwas ausführen soll, geschrieben.
[ ]	Mit Hilfe dieser eckigen Klammern wird ein Wert indiziert. Damit kann dann z.B. auf das fünfte Element eines Feldes zugegriffen werden. Werden für den Zugriff zwei Indizes benötigt, so werden die beiden Indizes in die Klammern durch Kommata getrennt geschrieben.
( )	Mit Hilfe dieser runden Klammern werden Funktions- oder Regelaufrufe durchgeführt. Innerhalb dieser Klammern werden dann die aktuellen Parameter der Regel oder Funktion angegeben. Als weitere Bedeutung können diese Klammern zur Veränderung der Abarbeitungsreihenfolge innerhalb eines Statements genommen werden. Die in den Klammern geschriebenen Ausdrücke werden zuerst miteinander verknüpft. Das Ergebnis hiervon wird dann erst mit den äußeren Ausdrücken verknüpft.

10.5 Ändern von Attributwerten

Jedes Attribut des Objektes mit Schreibzugriff kann beliebig geändert werden.

Attribute können geändert werden mit Hilfe der Kombination von

- Attributen
  Bitte vergleichen Sie für weitere Details das Kapitel „Attribute in alphabetischer Reihenfolge“ in der „Attributreferenz“.
- Funktionen
  Bitte vergleichen Sie für weitere Details das Kapitel „Schnittstellen-Funktionen“ im Handbuch „C-Schnittstelle - Funktionen“.
- Variablen
  Bitte vergleichen Sie für weitere Details die Kapitel „Globale Variablen (variables)“ und „Lokale Variablen“.

Beispiele

» Setzen der Sichtbarkeit eines Fensters:

```
MyWindow.visible := true;
```

» Setzen von zwei Zeilen in einer Listbox:

```
MyListbox.content[1] := "first string";
MyListbox.content[2] := "second string";
```

» Kombination von zwei editierbaren Texten in einem dritten editierbaren Text, getrennt durch ein Leerzeichen:

```
Edittext3.content := Edittext1.content + " " + Edittext2.content;
```

10.6 Verzweigung des Programmflusses

Zur Steuerung des Programmablaufes in einer Regel stehen verschiedene Konstrukte zur Verfügung:

» if-then-else

» if-elseif-else

» case-Anweisung („switch“)

10.6.1 if-then-else

Mit einer **if**-Anweisung kann ein Programmzweig mit Anweisungen definiert werden, der nur durchlaufen wird, wenn eine Bedingung erfüllt ist. Optional kann ein weiterer Programmzweig mit Anweisungen definiert werden, der ausgeführt wird, wenn die Bedingung nicht erfüllt ist.

Syntax

```
if <boolescher Ausdruck>
then
  <Anweisungen>
{ else
    <Anweisungen> }
endif
```

Ist <boolescher Ausdruck> wahr (*true*), so werden die <Anweisungen> zwischen **then** und **else** ausgeführt;

Ist <boolescher Ausdruck> falsch (*false*), so werden die <Anweisungen> zwischen **else** und **endif** ausgeführt.

In diesem Statement kann der **else**-Teil entfallen.

Beispiel

```
on PbBlink select
{
```

```
  if (WnMain.visible)
  then
    WnMain.visible := false;
  else
    WnMain.visible := true;
  endif
}
```

Bei Bestätigen des Knopfes „PbBlink“ wird zunächst geprüft, ob das Fenster „WnMain“ sichtbar ist (Wert von *WnMain.visible*).

Wenn das Fenster sichtbar ist (*WnMain.visible = true*), so wird es durch das Statement *WnMain.visible := false* unsichtbar gemacht.

Ist das Fenster nicht sichtbar (*WnMain.visible = false*), so wird es durch das Statement *WnMain.visible := true* sichtbar gemacht.

10.6.2 if-elseif-else

Das „if-elseif-else“-Konstrukt ist eine verkürzte Schreibweise für verschachtelte „if-then-else-if“.

Syntax

```
if <boolescher Ausdruck>
then
  <Anweisungen>
[ elseif <boolescher Ausdruck>
  then
    <Anweisungen> ]
{ else
    <Anweisungen> }
endif
```

In diesem Statement können die **elseif**-Teile beliebig oft wiederholt werden.

Beispiel

```
on PbBlink select
{
  if (WnMain.childcount = 0)
  then
    print "Habe keine Objekte.";
  elseif (WnMain.childcount = 1)
  then
    print "Habe ein Objekt.";
  else
    print "Habe mehrere Objekte.";
  endif
```

```
}
```

10.6.3 case-Anweisung

Eine **case**-Anweisung kann anstatt mehreren „if-then-endif“-Konstrukten verwendet werden. Soll sich daher ein Programm an einer Stelle in mehrere Äste verzweigen, eignet sich hierfür eine **case**-Anweisung.

Syntax

```
case <Ausdruck>
[ in <Kriterium> [ , <Kriterium> ] :
    <Anweisungen> ]
{ otherwise :
    <Anweisungen> }
endcase

Kriterium ::= <Wert> | <Min> .. <Max>
```

Beispiel

```
case Edittext.content
  in "Hello":
    print "Yes";
  in "Bye":
    exit();
  otherwise:
endcase
```

Eine case-Anweisung prüft für jeden **in**-Ausdruck, ob dieser wahr ist.

Beispiel

```
variable integer V := 10;
case V
  in 10:
    print "10";
  in 1..100:
    print "1..100";
  in 10..19:
    print "10..19";
  in 20..29:
    print "20..29";
  otherwise:
    print "nothing";
endcase
```

Ausgabe

```
"10"
"1..100"
"10..19"
```

Hier werden alle Statements aufgeführt, die den Wert erfüllen.

Wurde keiner der **in**-Ausdrücke erfüllt, dann – und nur dann – wird **otherwise** ausgeführt, soweit es angegeben worden ist.

Sobald ein oder mehrere **in**-Ausdrücke erfüllt wurden, wird **otherwise** nicht mehr ausgeführt.

Im Gegensatz zu C und anderen Programmiersprachen dürfen in **in**-Ausdrücken auch variable Werte vorkommen.

Beispiel

```
on PushbuttonOK select
{
  case true
    in Checkbox_1.active:
      // Aktion für die erste Checkbox, wenn aktiv
    in Checkbox_2.active:
      // Aktion für die zweite Checkbox, wenn aktiv
    in Checkbox_3.active:
      // Aktion für die dritte Checkbox, wenn aktiv
    // otherwise ist hier nicht notwendig
  endcase
}
```

Da in den **in**-Ausdrücken auch Berechnungen zugelassen sind, kann das obige Beispiel auch für den Fall erweitert werden, dass für inaktive Checkboxen (`not`) eine bestimmte Aktion ausgeführt wird.

```
on PushbuttonOK select
{
  case true
    in Checkbox_1.active:
      // Aktion für die erste Checkbox, wenn sie aktiv ist
    in Checkbox_2.active:
      // Aktion für die zweite Checkbox, wenn sie aktiv ist
    in Checkbox_3.active:
      // Aktion für die dritte Checkbox, wenn sie aktiv ist
    in not Checkbox_1.active:
      // Aktion für die erste Checkbox, wenn sie nicht aktiv ist
    in not Checkbox_2.active:
      // Aktion für die zweite Checkbox, wenn sie nicht aktiv ist
    in not Checkbox_3.active:
      // Aktion für die dritte Checkbox, wenn sie nicht aktiv ist
    // otherwise ist hier immer noch nicht notwendig
```

```
  endcase
}
```

Anmerkung

Es sind auch beliebige Variablen zugelassen. Bitte beachten Sie dabei, dass der richtige Typ verwendet wird.

10.7 Schleifen-Konstrukte

10.7.1 for-Schleife

Dieses Konstrukt dient zur Formulierung von Schleifen, wenn die Anzahl der Durchläufe vor Eintritt in die Schleife berechnet werden kann.

Syntax

```
for <Zähler> := <Start> to <Ende> { step <Schrittweite> } do
  <Anweisungen>
endfor
```

Beim Schleifenstart bekommt <Zähler> den Wert <Start> zugewiesen. Nach jedem Schleifendurchlauf wird <Schrittweite> zu <Zähler> addiert. Wurde <Schrittweite> nicht angegeben, wird immer die Standardschrittweite *1* addiert. Die Schleife wird solange durchlaufen, bis <Zähler> größer als <Ende> ist. Bei jedem Schleifendurchlauf werden die <Anweisungen> ausgeführt, die auf den aktuellen Wert von <Zähler> zugreifen können. <Start>, <Ende> und <Schrittweite> müssen *integer*-Werte sein. <Zähler> muss *integer*-Werte annehmen können. <Start>, <Ende> und <Schrittweite> werden nur einmal vor dem ersten Schleifendurchlauf berechnet.

Beispiel

```
for Index := 1 to Window.childcount do
  print Window.child[Index];
endfor
```

Es gibt keine Möglichkeit, eine `for`-Schleife vorzeitig zu beenden.

Wenn bei `for`-Schleifen das Abbruchkriterium nicht erreichbar ist, wird die Schleife nicht ausgeführt.

Beispiel

```
for I := 1 to 10 step -1 do
```

10.7.2 foreach-Schleife

Dieses Konstrukt dient zur Formulierung von Schleifen, die über alle Elemente einer Sammlung gehen.

Syntax

```
foreach <Element> in <Ausdruck> do
  <Anweisungen>
endfor
```

Die Schleife bricht mit einem Fehler ab, wenn der *<Ausdruck>* keine Sammlung ist oder der Schleifenwert nicht dem *<Element>* zugewiesen werden kann. Der *<Ausdruck>* wird nur einmal Initial ausgewertet. Die Laufvariable *<Element>* darf innerhalb der *<Anweisungen>* verändert werden. Eine Änderung der Laufvariablen hat keinen Einfluss auf die Durchlaufzahl oder den Wert der Laufvariablen im nächsten Durchlauf.

Beispiel

```
dialog D
window Wi
{
  listbox Lb
  {
    .xauto 0;  .yauto 0;
  }
  on close { exit(); }
}

on dialog start
{
  variable list StationList:= ["ARD", "ZDF", "SWR3", ""];
  variable string Station;

  foreach Station in StationList do
    Lb.content[Lb.itemcount+1] := Station;
  endfor
}
```

10.7.3 while-Schleife

Dieses Konstrukt ist eine Wiederholungsanweisung, also eine Schleifenanweisung, bei der die Anzahl der Wiederholungen von einer Bedingung abhängig ist. Die Bedingung wird zu Beginn eines neuen Durchlaufes geprüft.

Syntax

```
while <Bedingung> do
  <Anweisungen>
endwhile
```

Vor jedem Schleifendurchlauf wird der Wert <Bedingung> berechnet. Er muss ein *boolean*-Wert sein. Ist er *true*, werden die <Anweisungen> ausgeführt und die Schleifen von vorne begonnen. Sobald der Wert *false* ist, wird die Schleife abgebrochen. Anders kann eine `while`-Schleife nicht beendet werden.

Beispiel

```
while (Index < Window.childcount) do
  print Window.child;
  Index := Index + 1;
endwhile
```

10.8 Aufruf benannter Regeln

Benannte Regeln werden aufgerufen, indem der Name der Regel und die aktuelle Belegung der Parameter angegeben werden.

Beispiel

Eine Regel wird wie folgt definiert:

```
rule integer Calculate(integer Val1 input, integer Val2 input)
```

Der Aufruf dieser Regel kann dann wie folgt aussehen:

```
Calculate(14, 3);
```

Hinweis zum IDM für Windows

Um die maximale Rekursionstiefe ohne **Exception** zu erreichen, muss die Stackgröße erhöht werden. Eine Größe von 8 MByte (8.388.608 Byte) wird empfohlen.

Die Stackgröße kann mit der Linkeroption /STACK:<size> gesetzt werden. In Visual Studio 2017 kann man den Wert in den Projekteigenschaften unter Konfigurationseigenschaften / Linker / System / Stapelreservierungsgröße setzen.

10.9 Lokale Variablen

Lokale Variablen können dazu verwendet werden, innerhalb einer Regel beliebige Werte zu speichern. Sie sind nur in der Regel bekannt, in der sie definiert wurden. Darin unterscheiden sie sich von globalen Variablen, die im gesamten Dialog oder Modul bekannt sind (siehe Kapitel „Globale Variablen (variables)“).

Innerhalb der Regel, in der sie definiert werden, überdecken lokale Variablen eventuell vorhandene globale Variablen mit demselben Identifikator.

Lokale Variablen können als „normale“ Variablen oder als **statische** Variablen deklariert werden:

- Normale Variablen verlieren ihren Wert nach Beendigung der Regel. Sie werden bei jeder Regelausführung neu initialisiert.
- Statische Variablen behalten ihren Wert auch nach Beendigung der Regel. Sie werden nur bei der erstmaligen Regelausführung initialisiert.

10.9.1 Normale lokale Variablen

Syntax

```
variable <Datentyp> <Variablenname> { := <Ausdruck> }
                [ , <Variablenname> { := <Ausdruck> } ] ;
```

In einer Anweisung können mehrere Variablen mit demselben Datentyp deklariert werden. Dabei sind die Identifikatoren der Variablen durch , (Komma) zu trennen.

Lokale Variablen können direkt bei der Definition mit einem Wert oder einem Ausdruck initialisiert werden. Wird ein Ausdruck zur Initialisierung verwendet, ist folgendes zu beachten:

- Die Auswertung des Ausdrucks muss einen Wert mit dem Datentyp der Variablen ergeben.
- Im Ausdruck verwendete Variablen müssen vorher deklariert werden, d.h. schon bekannt sein.

Beispiel

```
rule integer R1(integer X)
{
  variable integer Z := X * 3;

  if (X > 100) then
    Z := Z / 2;
  endif

  return Z;
}
```

Auch eine Definition ohne direkte Initialisierung ist möglich. Hier muss aber vor dem ersten lesenden Zugriff auf die Variable ein Wert zugewiesen worden sein.

Beispiel

```
rule void R2 ()
{
  variable integer MyNumber;
```

```
  MyNumber := 5;

  print MyNumber;
}
```

Lokale Variablen können, sobald sie einen Wert enthalten, Attributen von Objekten zugewiesen werden. Es ist nur darauf zu achten, dass die Datentypen der lokalen Variablen und des Attributs dies zulassen (identische Datentypen oder Objekt-Attribut vom Typ *anyvalue*).

Beispiel

```
rule void R3 ()
{
  variable string MyWord;

  MyWord := "Hello world";
  Edittext.content := MyWord;
}
```

Eine Variable des Typs *object* kann wie ein Objekt verwendet werden, d.h. jedes Attribut des Objektes, welches in der Variablen gespeichert ist, kann verändert werden.

Beispiel

```
rule void R4 ()
{
  variable object MyObject;

  MyObject := MyWindow;
  MyObject.title := "New Title";
  MyObject.visible := true;
}
```

10.9.2 Statische Variablen (static variable)

Statische Variablen werden durch das zusätzliche Schlüsselwort **static** vor dem Schlüsselwort **variable** deklariert. Im Gegensatz zu normalen lokalen Variablen können sie nur mit Werten oder anderen Variablen initialisiert werden, aber nicht mit Ausdrücken.

Syntax

```
static variable <Datentyp> <Variablenname> { := <Wert> | <Variable> }
                      [ , <Variablenname> { := <Wert> | <Variable> } ] ;
```

Statische Variablen können von den selben Datentypen sein wie globale oder lokale Variablen.

Die **Initialisierungen** in den Deklarationen statischer Variablen werden **nur beim ersten Regelaufruf** durchgeführt, bei weiteren Aufrufen nicht mehr. Das gilt auch dann, wenn sich der Wert der Variablen, mit der initialisiert wurde, mittlerweile geändert hat. Stattdessen bleibt ihr Wert von der letzten Regelausführung bestehen.

Zur Initialisierung statischer Variablen dürfen nur Variablen verwendet werden, die **vorher** im Regelcode oder als globale Variablen im Dialog bzw. Modul definiert wurden.

Beispiele

```
variable integer G:=10;
rule Beispiel (object O input)
{
  variable boolean B;
  static variable integer X:=1, Y:=G, Z:=Y;
  static variable object  Q:=O;
  static variable boolean J:=B;
  variable integer I:=X;
}
```

```
rule Beispiel2
{
  static variable anyvalue A;
  if A = 1 then
    print "Irgendwas";
  else
    A := 1;
    print "Weitere Initalisierungen";
  endif
}
```

10.10 Rückgabe von Werten (return)

Mit Hilfe der `return`-Anweisung kann jede Regel an der aktuellen Stelle verlassen werden und wenn nötig ein Rückgabewert an die aufrufende Regel zurückgegeben werden.

Syntax

```
return { <Ausdruck> } ;
```

<Ausdruck> ist erforderlich, wenn die Regel einen anderen Rückgabetyp als *void* besitzt. Der Datentyp von <Ausdruck> muss dem Rückgabetyp der Regel entsprechen.

Beispiel

```
rule integer Addivide (integer Arg1 input, integer Arg2 input,
                       integer Arg3 input)
```

```
{
  variable integer Sum;
  Sum := Arg1 + Arg2;
  if (Arg3 <> 0) then
    return (Sum / Arg3);
  endif
  return (0);
}
```

10.11 Aufruf von Anwendungsfunktionen

In der Dialogdatei deklarierte Funktionen können direkt aus den Regeln aufgerufen werden. Eine Ausnahme bilden **Canvas-Funktionen**, **Objekt-Callback-Funktionen**, **Format-Funktionen** und **Nachlade-Funktionen** - sie können **nicht** aus den Regeln aufgerufen werden!

Die Parameter der Funktionen müssen durch die gegenwärtigen Werte ersetzt werden. Ihre Typen müssen der Deklaration der Funktion entsprechen.

Beispiele

Deklaration

```
function c integer Add (integer, integer);
```

Aufruf

```
Edittext.content := itoa (Add (atoi (Edittext2.content),
                    atoi (Edittext3.content)));
```

Bedeutung

Eingabe: zwei Zahlen

Ausgabe: Summe wird zurückgegeben

Deklaration

```
function c void GetAddress (string, string output, string output);
```

Aufruf

```
GetAddress (Edittext1.content, Edittext2.content,
    Edittext3.content);
```

Bedeutung

Eingabe: z.B. Name

Ausgabe: z.B. Adresse (Straße und Ort)

11 Eingebaute Funktionen (Builtin-Funktionen)

Der DM verfügt über Standardfunktionen zur Bestimmung der Länge einer Zeichenkette sowie zur Umwandlung von Zahlen- und Zeichenketten ("string").

Daher müssen Sie diese Funktionen nicht selbst definieren.

Diese eingebauten Funktionen ("builtin functions") können im Aktionsteil einer Regel einbezogen werden.

11.1 append()

Mit dieser Funktion kann an eine Sammlung (Datentypen *hash*, *list*, *matrix*, *refvec* und *vector*) oder einen String ein neuer Wert einmal oder mehrmals angehängt werden.

Allgemein gilt, dass der Datentyp des (optionalen) neuen Wertes ein Skalar sein muss und zum Wertetyp der Sammlung passen muss.

Die Wiederholungszahl des Anhängens (*Count*-Parameter) muss >= *0* sein.

Im Fehlerfall wird der Funktionsaufruf mit einem `fail` abgebrochen.

Definition

```
anyvalue append
(
      anyvalue ListValue  input,
      anyvalue NewValue   input
  { , integer  Count := 1 input }
  { , enum     Dir   := dir_row input }
)
```

Parameter

anyvalue *ListValue* input

In diesem Parameter wird die Sammlung bzw. der String angegeben an die ein Wert angehängt werden soll.

anyvalue *NewValue* input

Dieser Parameter definiert den neu anzufügenden Wert. Der Wert muss zum Wertetyp der Sammlung passen.

integer ***Count*** *:= 1 input*

In diesem optionalen Parameter wird die Anzahl angegeben, wie oft ein neuer Wert (bzw. wie viele Zeilen oder Spalten) angehängt werden soll. Erlaubt sind Werte >= *0*, wobei bei *0* kein Anhängen erfolgt. Standardmäßig wird der neue Wert einmal angehängt.

enum ***Dir*** *:= dir_row input*

In diesem optionalen Parameter wird für Sammlungen vom Datentyp *matrix* definiert, ob neue Zeilen (*dir_row*) oder neue Spalten (*dir_column*) angehängt werden.

Rückgabewert

Die Funktion gibt die veränderte Sammlung bzw. den veränderten String zurück.

Besonderheiten

» *hash*
Da dieser Sammlungsdatentyp keinen definierten Indexbereich besitzt, erfolgt das Anhängen hinter dem letzten *integer*-Index. Es wird also von einem *hash* mit einer Indizierung von *1,2,3…* ausgegangen. Der letzte gesetzte Index vom Datentyp *integer* wird ermittelt und daran anschließend angehängt.

» *matrix*
Das Anhängen in eine Matrix erfolgt immer zeilen- oder spaltenweise. Die Steuerung erfolgt über den *Dir*-Parameter. Standardmäßig (*dir_row*) werden neue Zeilen angefügt. Bei Angabe von *dir_column* werden neue Spalten angefügt.

» *refvec*
Das Anhängen von *null*-Objekten ist nicht erlaubt. Falls die Objekt-Id schon vorhanden ist, erfolgt ein Umpositionieren an die letzte Position.

» *string*
Der neue Wert wird am String-Ende angefügt.

» *Default-Werte*
Um die Indexposition für die neuen Werte zu ermitteln werden auch Default-Werte beachtet, außer es handelt sich um eine *hash*-Sammlung.

Beispiele

» Mehrfaches Anhängen an einen String

```
print append("hi", " ho", 3);
```

Ausgabe

```
"hi ho ho ho"
```

» Mehrfaches Anfügen an eine *list*-Sammlung

```
variable list L := [4711, "koeln"];
print append(L, true, 2);
```

Ausgabe

```
[4711,"koeln",true,true]
```

» Anfügen von zwei Zeilen an eine Matrix

```
variable matrix M := [ [0,0]=>"?", [1,1]=>"a", [1,2]=>"b",
                       [2,2]=>"c", [3,1]=>"end" ];
print append(M, "new", 2);
```

Ausgabe

```
[[1,1]=>"a",[1,2]=>"b",[2,1]=>"?",[2,2]=>"c",[3,1]=>"end",[3,2]=>"?",
[4,1]=>"new",[4,2]=>"new",[5,1]=>"new",[5,2]=>"new"]
```

» Anfügen an einen Hash mit gemischter Indizierung

```
variable hash H := [false=>.xleft, 31=>.width, 33=>.ytop, "x"=>1];
print append(H, .height);
```

Ausgabe

```
[false=>.xleft,31=>.width,33=>.ytop,34=>.height,"x"=>1]
```

» Anhängen an eine *vector*-Liste

```
variable vector[integer] V := [9, 8, 7];
print append(V, 6);
```

Ausgabe

```
[9,8,7,6]
```

Verfügbarkeit

Ab IDM-Version A.06.02.g

Siehe auch

Eingebaute Funktion **insert()**

11.2 applyformat()

Mit Hilfe dieser Funktion kann ein String durch die Zuweisung eines Formats umgewandelt werden.

Der Rückgabewert ist derselbe String, der im Edittext angezeigt wird, wenn er dasselbe Format verwendet und den gegebenen String als seinen Inhalt hat.

Definition

```
string applyformat
(
  string FormatString input | object FormatResource input ,
  string String input
)
```

Eine weitere Definition der Funktion sieht wie folgt aus:

```
string applyformat
(
  object FormatFunc   input,
  string FormatString input,
  string String input
)
```

Parameter

string *FormatString* input
: In diesem Parameter wird der Formatstring angegeben, mit dessen Hilfe der String formatiert werden soll.

object *FormatResource* input
: In diesem Parameter wird die Formatressource angegeben, mit deren Hilfe der String formatiert werden soll.

object *FormatFunc* input
: In diesem Parameter wird die Formatfunktion angegeben, die den String formatieren soll.

string *String* input
: In diesem Parameter wird der String angegeben, der formatiert werden soll.

Rückgabewert

Als Ergebnis wird der formatierte String zurückgegeben.

Beispiel

```
applyfomat("NN-NN-NN", "121295");
```

Ausgabe

```
12-12-95
```

11.3 atoi()

Diese Funktion wandelt eine optional vorzeichenbehaftete Ziffernkette in eine Dezimalzahl um (**a**scii **to i**nteger). Dabei sind erlaubt:

- Ein Vorzeichen + | -
- Ziffern *0…9*
- Leerzeichen als Füllzeichen an jeder Stelle

Angenommen, die Zahl *5* soll zu einer Zahl *123* in einem editierbaren Feld addiert werden. Zu diesem Zweck werden die Zeichenketten "123" und "5" als Zahl *123* bzw. *5* interpretiert. Das Ergebnis ist *128*.

Definition

```
integer atoi
(
  string IntString input
)
```

Parameter

string *IntString* input

In diesem Parameter wird der String angegeben, der in eine Zahl umgewandelt werden soll.
In this parameter the string to be changed into a number is indicated.

Rückgabewert

Zahl, in die der String konvertiert worden ist.

Anmerkung

Dieser Funktionsaufruf endet mit einem Fehler, der über die Funktion **fail()** abgefangen werden kann, wenn keine Ziffer oder andere Zeichen im String vorkommen.

Beispiel

```
Sum := atoi(Edittext1.content) + atoi(Edittext2.content);
```

11.4 beep()

Mit Hilfe dieser Funktion können Sie einen Ton erzeugen. Dabei kann je nach zugrundeliegendem Fenstersystem über Parameter spezifiziert werden, in welcher Form und wie lange der Ton ausgegeben werden soll.

Definition

```
void beep
(
  { enum Mode input }
)

void beep
(
  integer Volume    input,
  integer Duration  input,
  integer Frequency input
)
```

Parameter

*enum **Mode** input*

Über diesen Parameter kann ein Ton mittels der folgenden Symbole ausgewählt werden:

beep_error
: Fenstersystem-spezifischer Ton wenn ein Fehler aufgetreten ist.

beep_note
: Fenstersystem-spezifischer Ton für einen Hinweis.

beep_ok
: Fenstersystem-spezifischer Ton für eine Bestätigung.

beep_question
: Fenstersystem-spezifischer Ton für eine Aufforderung an den Benutzer.

beep_warning
: Fenstersystem-spezifischer Ton für eine Warnung.

Dabei ist die Ausprägung dieser Töne bzw. ob sich die verschiedenen Töne überhaupt unterscheiden sehr fenstersystemspezifisch.

integer *Volume* input

Dieser Parameter spezifiziert die Lautstärke, in der der Ton ausgegeben werden soll. Hierbei wurde die im XWindows System gebräuchliche Definition verwendet:
Der Wert muss im Bereich zwischen -100 und 100 liegen. Der Wert 0 definiert die Defaultlautstärke. Bei negativen Werten variiert die Lautstärke prozentual zwischen „unhörbar“ (= -100) und der Defaultlautstärke. Bei positiven Werten variiert die Lautstärke prozentual zwischen der Defaultlautstärke und der maximal möglichen Lautstärke (= 100).
Hinweis: Dieser Parameter muss unter Microsoft Windows zwar angegeben werden, er wird aber ignoriert. Es wird die im System eingestellte Lautstärke verwendet.

integer *Duration* input

In diesem Parameter kann die Dauer des Tons in Millisekunden im Bereich zwischen 0 - 60000 definiert werden. Mit dem Wert 0 wird die Defaultdauer definiert.
Hinweis: Unter Microsoft Windows wird bei dem Wert *0* ein Standardton erzeugt.

integer *Frequency* input

In diesem Parameter kann die Tonfrequenz (**frequency**) im Bereich zwischen 0 - 20000 Hz definiert werden. Mit dem Wert 0 wird die Defaultfrequenz definiert.
Hinweis:Unter Microsoft Windows ist der Wertebereich 37 - 32767 Hz. Bei anderen Werten wird ein Standardton erzeugt.

Warnung

Ob und wie diese Parameter bei der Tonerzeugung beachtet werden, hängt sehr stark von dem verwendeten Fenstersystem und eventuell auch von der darunterliegenden Hardware ab. Diese Systemabhängigkeit gilt auch für das Verhalten, wenn sehr rasch hintereinander verschiedene Töne erzeugt werden.

Diese Funktion ist nicht dafür gedacht, Klangabfolgen oder Melodien zu definieren, sondern dafür, um den Endanwender mittels einzelner Töne auf besondere Situationen aufmerksam zu machen.

Der Dialog Manager versucht beim Aufruf dieser Funktion, einen Ton zu erzeugen, der dem durch die Parameter spezifizierten Ton so nahe kommt, wie es im Rahmen des jeweiligen Systems möglich ist.

Anmerkungen

Wenn diese Funktion ohne Parameter aufgerufen wird, wird ein einfacher Ton erzeugt.

Die Parameter *Volume*, *Duration* und *Frequency* müssen in der angegebenen Reihenfolge angegeben werden, wobei nicht benötigte Parameter am Ende weggelassen werden können.

Beispiele

```
beep();
beep(beep_warning);
beep(90, 10000);
beep(0, 0, 1200);
```

11.5 closequery()

Die Funktion **closequery()** schließt eine Dialogbox (Fenster mit gesetztem Attribut *.dialogbox true*) durch setzen des Atttributs *.visible* auf *false*. Gleichzeitig mit dieser Funktion auch ein Rückgabewert für die Dialogbox festgelegt.

Definition

```
void closequery
(
  anyvalue RetVal input
)
```

Parameter

anyvalue *RetVal* input

Wert, den die entsprechende Dialogbox in der Funktion **querybox()** zurückgibt.

Hinweise

Die Funktion **closequery()** sollte nur eingesetzt werden, wenn die Dialogbox mit der Funktion **querybox()** geöffnet wurde.

Ob ein *changed*-Ereignis auf .visible an die Dialogbox gesandt wird ist abhängig vom Parameter *ShipEvent* der Funktion **querybox()**. Ist dieser *true* wird das *changed*-Ereignis verschickt, bei *false* hingegen nicht.

Beispiel

```
dialog D

window Box
{
  .dialogbox true;
  .visible   false;
  .title     "Enter password";

  edittext E
  {
    .format "S";
  }

  pushbutton P
  {
    .text "OK";

    on select
    {
      closequery(E.content);
```

```
    }
  }
}

window WnLogin
{
  child pushbutton PbLogin
  {
    .text "Login";

    on select
    {
      if (querybox(Box) <> "password") then
        exit();
      endif
    }
  }
}
```

11.6 concat()

Die Funktion **concat()** verkettet mehrere Strings zu einem String. Bei der Verkettung kann eine Trennzeichenfolge zwischen den Strings eingefügt werden oder die Strings können wiederholt aneinander gehängt werden.

Definition

Wenn der erste Parameter ein String ist, wird dieser String immer zwischen zwei zu verkettenden Strings eingefügt.

```
string concat
(
      string   Separator input,
      anyvalue Value1    input
  { , anyvalue Value2    input
  ...
  { , anyvalue Value15   input } }
)
```

Ist der erste Parameter ein *integer*-Wert, dann definiert er, wie oft die Verkettung der Strings wiederholt wird.

```
string concat
(
      integer  Repeat  input,
      anyvalue Value1  input
  { , anyvalue Value2  input
  ...
  { , anyvalue Value15 input } }
)
```

Parameter

string *Separator* input

Dieser Parameter enthält eine Trennzeichenfolge, die bei der Verkettung zwischen zwei Strings eingefügt wird.
Separator kann nicht zusammen mit *Repeat* verwendet werden.

integer *Repeat* input

Dieser Parameter definiert, wie oft die Verkettung der Strings wiederholt werden soll. Zunächst wird ein String aus den übergebenen *Value*-Parametern gebildet, anschließend wird der gebildete String so oft aneinander gehängt, wie in *Repeat* angegeben. Ist *Repeat* <= *0*, wird ein Leerstring zurückgegeben.
Repeat kann nicht zusammen mit *Separator* verwendet werden.

anyvalue *Value1* input

*anyvalue **Value2** input*

…

*anyvalue **Value15** input*

In diesen (optionalen) Parametern werden die Werte übergeben, die zu einem String verkettet werden sollen.
Skalare Werte werden für die Verkettung zu Strings gewandelt. Enthalten die Parameter Sammlungen, dann werden zunächst die darin enthaltenen Werte (ohne Defaultwerte) verkettet und anschließend die Parameter miteinander verkettet. Die Vorgehensweise entspricht einem Aufruf von *sprintf("%s", <Value>)*.

Rückgabewert

String in dem die übergebenen Werte verkettet sind.

Beispiele

» Verketten von drei Werten mit Trennzeichen

```
text TxTres "Tres";
...
print concat(",", 1, "zwei", TxTres);
```

Ausgabe

```
"1,zwei,Tres"
```

» Wiederholte Verkettung

```
print concat(3, "|", [".", ".."]);
```

Ausgabe

```
"|...|...|..."
```

11.7 countof()

Diese Funktion liefert die Größe einer Sammlung zurück. Dies ist typischerweise der höchste Indexwert.

Typischerweise liefern **countof()** und **itemcount()** für Werte des Datentyps *refvec*, *vector* und *list* den gleichen Wert zurück. **itemcount()** bietet aber die generischere Verwendung, wohingegen **countof()** sich für die strukturierte, typangepasste Verwendung besser eignet.

Definition

```
anyvalue countof
(
  anyvalue Value input
)
```

Parameter

anyvalue *Value* input

In diesem Parameter wird der Wert angegeben für den der Indizierungstyp, bzw. der höchste Indexwert, ermittelt werden soll.

Rückgabewert

nothing
: Der übergebene Wert ist skalar.

1 … 2^{31}
: Höchster Index der übergebenen Liste (Datentypen *list*, *vector*, *refvec*).

[0 … 65535,0 … 65535]
: Höchster Index der übergebenen Matrix.

anyvalue
: Datentyp des Index des übergebenen assoziativen Felds (Datentyp *hash*).

Beispiel

```
dialog D

on dialog start
{
  variable matrix Matrix := [
    [0,0] => "-?-",
    [1,1] => "germany",
    [1,2] => "berlin",
    [2,1] => "france"
    /* [2,2] => inherited from default [0,0] */ ];
  variable integer Row, Col;
  variable anyvalue Count, Idx;
```

```
  /* print the Matrix values [0,0] [0,1] ... [2,2] */
  Count := countof(Matrix);
  for Row:=0 to first(Count) do
    for Col:=0 to second(Count) do
      Idx := [Row,Col];
      print sprintf("%s : %s", Idx, Matrix[Idx]);
    endfor
  endfor
  exit();
}
```

Siehe auch

Eingebaute Funktionen itemcount(), valueat()

Methode index

Attribut count

C-Funktion DM_ValueCount

11.8 create()

Mit Hilfe dieser Funktion können zur Laufzeit neue Objekte innerhalb des Dialog Managers generiert werden. Diese Funktion liefert als Ergebnis das neu generierte Objekt zurück.

Definition

```
object create
(
      object  Class  input,
      object  Parent input
  { , object  Dialog input }
  { , integer Type   := 3 input }
  { , boolean CreateInvisible := false input }
)
```

Parameter

object *Class* input

Dieser Parameter bezeichnet das Objekt, von dem eine neue Instanz erstellt werden soll. Damit kann das Objekt also ein Default oder eine Vorlage sein.

object *Parent* input

Dieser Parameter bezeichnet den Vater des Objektes.

*object **Dialog** input*

Dieser optionale Parameter bezeichnet den Dialog, zu dem das Objekt gehören soll.

*integer **Type** := 3 input*

Mit diesem optionalen Parameter können Sie angeben, ob Sie einen Default, ein Modell oder ein Objekt erzeugen wollen (optional). Dabei erzeugt:

1
: Default

2
: Modell

3
: Objekt

Wenn dieser Parameter nicht angegeben ist, wird automatisch ein Objekt generiert.

*boolean **CreateInvisible** := false input*

In diesem optionalen Parameter wird angegeben, ob das Objekt unsichtbar oder wie in der Vorlage definiert generiert werden soll. Dabei bedeutet

true
: Das Objekt wird immer unsichtbar angelegt.

false
: Sichtbarkeit wird aus Modell oder Default übernommen.

Rückgabewert

objectId
: Id des neu generierten Objekts.

null
: Null-ID, da das Objekt nicht generiert werden konnte.

Beispiel

```
dialog Beispiel
{
}
model window MWnDetail
{
  !! Detailanzeige von Datensätzen
  .title "Detailanzeige der Daten";

  rule void GetData()
  {
    !! Holt sich die Daten wie abgesprochen
    !! macht Fenster sichtbar
    this.visible ::= true;
  }

  child pushbutton PbExit
  {
    .text "&Abbrechen";

    on select
    {
      !! Zerstört das Instanzen-Fenster
      destroy(this.window);
    }
  }
}

window WnMain
{
  .title "Datenübersicht";

  child tablefield TfDaten
  {
    !! Enthält eine Datenübersicht
    on dbselect
    {
      variable object NewObj := null;

      !! sichtbares Erzeugen:
```

```
      !! NewObj ::= MWnDetail:create(this.dialog);
      !! unsichtbares Erzeugen:
      NewObj ::= create(MWnDetail , this.dialog, true);
      NewObj:GetData();
    }
  }
}
```

Siehe auch

Methode :create()

C-Funktion DM_CreateObject im Handbuch „C-Schnittstelle - Funktionen“

COBOL-Funktion DMcob_CreateObject im Handbuch „COBOL-Schnittstelle“

11.9 delete()

Mit dieser Funktion können aus einer Sammlung (Datentypen *hash*, *list*, *matrix*, *refvec* und *vector*) einer oder mehrere Werte an einer vorgegebenen Position gelöscht werden. Dahinterliegende Werte werden nach vorne geschoben. Außerdem ist der Datentyp *string* erlaubt um Zeichen aus einem String zu entfernen.

Das Löschen der Werte erfolgt dabei an der angegebenen Position. Die Positionsangabe muss >= *1* sein, d.h. ein Löschen von Default-Werten an den Indizes *[0]* bzw. *[0,0]* ist mit dieser Funktion nicht möglich. Die größtmögliche erlaubte Position entspricht dem aktuellen *countof(ListValue)*.

Die Anzahl der zu löschenden Werte wird über den *Count*-Parameter angegeben und sollte >= *0* sein.

Im Fehlerfall wird der Funktionsaufruf mit einem `fail` abgebrochen.

Als fehlerhaft wird auch angesehen, wenn versucht wird mehr Werte bzw. Zeilen oder Spalten zu löschen als überhaupt vorhanden sind.

Definition

```
anyvalue delete
(
      anyvalue ListValue input
      integer  Pos   input
  { , integer  Count := 1 input }
  { , enum     Dir   := dir_row input }
)
```

Parameter

anyvalue *ListValue* input

In diesem Parameter wird die Sammlung bzw. der String angegeben aus der Werte gelöscht werden sollen.

integer *Pos* input

In diesem Parameter wird die Indexposition angegeben, ab der einschließlich das Löschen erfolgen soll. Erlaubt sind Werte >= *1*. Bei *matrix*-Sammlungen stellt *Pos* je nach *Dir*-Parameter entweder eine Zeilenposition (*dir_row*) oder eine Spaltenposition (*dir_column*) dar.

*integer **Count** := 1 input*

In diesem optionalen Parameter wird die Anzahl der Werte (bzw. Zeilen oder Spalten bei einer Matrix) vorgegeben, die zu löschen sind. Erlaubt sind Werte >= *0*, wobei bei *0* kein Löschen erfolgt. Durch die Anzahl sollte beim Löschen der mögliche Positionsbereich nicht überschritten werden.

*enum **Dir** := dir_row input*

In diesem optionalen Parameter wird die Ausrichtung der Indexposition definiert, was für eine Matrix von Bedeutung ist. Dabei bedeutet der Standardwert *dir_row*, dass in der Matrix Zeilen gelöscht werden. Bei *dir_column* werden Spalten gelöscht.

Rückgabewert

Die Funktion gibt die veränderte Sammlung bzw. den veränderten String zurück.

Besonderheiten

» *hash*
Da dieser Sammlungsdatentyp keinen definierten Indexbereich besitzt, erfolgt das Löschen bezogen auf die *integer*-Indizes. Es wird also von einem *hash* mit einer Indizierung von *1,2,3…* ausgegangen. Es gibt keine Beschränkungen hinsichtlich der größtmöglichen Indexposition.

» *matrix*
Das Löschen in einer Matrix erfolgt immer zeilen- oder spaltenweise. Die Steuerung erfolgt über den *Dir*-Parameter. Standardmäßig (*dir_row*) werden Zeilen gelöscht. Bei Angabe von *dir_column* werden Spalten gelöscht. Dies bedeutet auch, dass sich die Positionsangabe im ersten Fall auf eine Zeile und im zweiten Fall auf eine Spalte bezieht. Entsprechend verkleinert sich damit auch die Matrix.

» *string*
Hier bezieht sich die Position auf die Zeichenposition im String, d.h. der String wird als Array von Zeichen behandelt.

Beispiele

» Mehrfaches Löschen von Werten aus einer Liste

```
variable list L := [17, "x", window, .xleft, null];
print delete(L, 2, 3);
```

Ausgabe

```
[17,null]
```

» Löschen einer Spalte aus einer Matrix

```
variable matrix M := [ [0,0]=>"?", [1,1]=>"a1", [1,2]=>"b1",
                       [2,2]=>"b2", [2,3]=>"c2" ];
M := delete(M, 2, dir_column);
print M;
print countof(M);
```

Ausgabe

```
[[1,1]=>"a1",[1,2]=>"?",[2,1]=>"?",[2,2]=>"c2"]
[2,2]
```

» Entfernen von zwei Zeichen aus einem String

```
print delete("hallo", 3, 2);
```

Ausgabe

```
"hao"
```

» Löschen von Elementen aus einem Hash mit *integer*-Indizierung

```
variable hash H := [2=>.xleft, 31=>.width, 33=>.ytop];
H := delete(H, 30, 3);
print H;
```

Ausgabe

```
[2=>.xleft,30=>.ytop]
```

» Löschen aus einer *vector*-Liste

```
variable vector[boolean] V := [true, false, false, false, true];
print delete(V, 2, 3);
```

Ausgabe

```
[true,true]
```

» Löschen der ersten beiden Elemente aus einer *refvec*-Liste

```
variable refvec R := [PbApply, PbCancel, PbRevert];
print delete(R, 1, 2);
```

Ausgabe

```
[pushbutton D.PbRevert]
```

Verfügbarkeit

Ab IDM-Version A.06.02.g

Siehe auch

Eingebaute Funktion **insert()**

11.10 destroy()

Mit Hilfe dieser Funktion können beliebige Objekte oder Modelle eines Dialogs gelöscht werden. Dabei werden alle Kinder des Objektes mit gelöscht.

Definition

```
boolean destroy
(
      object  Object input
  { , boolean DoIt   := true input }
)
```

Parameter

object *Object* input

In diesem Parameter wird das zu löschende Objekt angegeben.

*boolean **DoIt** := true input*

Mit Hilfe dieses optionalen Parameters wird gesteuert, wie dieses Objekt gelöscht werden soll. Wenn hier *true* angegeben wird, wird das Objekt gelöscht sowie alle Regelteile, die dieses Objekt benutzen abgeändert, so dass die entsprechenden Anweisungen entfernt werden. Falls es sich bei dem zu löschenden Objekt um ein Modell handelt, wird – falls der zweite Parameter *false* ist – das Modell nur gelöscht, wenn es von keinem anderen Objekt benutzt wird. Wird hier aber *true* angegeben, wird das Modell gelöscht und die Objekte, die dieses Modell benutzen, beziehen sich wieder auf das nächsthöhere Modell bzw. den Default.

Rückgabewert

true
: Objekt konnte gelöscht werden.

false
: Objekt konnte nicht gelöscht werden.

destroy() ruft die :clean()-Methode des zu zerstörenden Objekts auf.

Beispiel

```
dialog Beispiel
{
}
model window MWnDetail
{
  !! Detailanzeige von Datensätzen
  .title "Detailanzeige der Daten";

  rule void GetData()
  {
```

```
    !! Holt sich die Daten wie abgesprochen
    !! macht Fenster sichtbar
    this.visible ::= true;
  }

  child pushbutton PbExit
  {
    .text "&Abbrechen";

    on select
    {
      !! Zerstört das Instanzen-Fenster
      destroy(this.window);
    }
  }
}

window WnMain
{
  .title "Datenübersicht";

  child tablefield TfDaten
  {
    !! Enthält eine Datenübersicht
    on dbselect
    {
      variable object NewObj := null;

      !! sichtbares Erzeugen:
      !! NewObj ::= MWnDetail:create(this.dialog);
      !! unsichtbares Erzeugen:
      NewObj ::= create(MWnDetail , this.dialog, true);
      NewObj:GetData();
    }
  }
}
```

Siehe auch

Methode :destroy()

C-Funktion DM_Destroy im Handbuch „C-Schnittstelle - Funktionen“

COBOL-Funktion DMcob_Destroy im Handbuch „COBOL-Schnittstelle“

11.11 dumpstate()

Mit dieser Funktion werden IDM Zustandsinformationen in die Log- bzw. Tracedatei oder in eine definierte Datei herausgeschrieben.

Der Dumpstate ist eine Zustandsinformation von IDM-relevanten Informationen um die Fehleranalyse einer IDM-Applikation zu erleichtern.

Der Inhalt des Dumpstate ist in verschiedene Abschnitte unterteilt, die variabel und der Fehlersituation angepasst sind. Außerdem wird er von zuvor aufgetretenen Fehlern beeinflusst. Beispielsweise führt eine erfolglose Speicherallokierung bei der nächsten Dumpstate-Ausgabe dazu, dass Informationen bzgl. der Speichernutzung durch den IDM ausgegeben werden. Konnten keine IDM-Objekte oder Bezeichner mehr angelegt werden, wird die Auslastung von IDM-Objekten und Bezeichnern ausgegeben.

Die Dumpstate-Information ist immer zwischen *„*** DUMP STATE BEGIN ***“* und *„*** DUMP STATE END ***“* eingeschlossen und kann folgende Abschnitte aufweisen:

- PROCESS: Prozess- und Thread-Nummer, Datum/Uhrzeit.
- ERRORS: Kompletter Inhalt der gesetzten Fehlercodes.
- CALLSTACK: Aufruf-Stack – beinhaltet Regeln, DM-Schnittstellenfunktionen und die obersten, vom IDM aufgerufenen, Applikationsfunktionen.
- THISEVENTS und EVENT-QUEUE: Aktuell verarbeitete thisevent-Objekte und deren Werte sowie Ereignisse die noch in der Warteschlange stehen.
- USAGE: Anzahl angelegter Objekte, Module und Bezeichner, Speichergröße die vom Regelinterpreter und für die String-Übergabe genutzt wird.
- MEMORY: Speichernutzung die vom IDM erfassbar ist.
- SLOTS: Hinweise zu IDM-Objekten die nicht richtig freigegeben wurden.
- VISIBLE OBJECTS: Liste der sichtbaren Objekte mit ihren Werten.

Um die Ausgabe möglichst klein zu halten, erfolgt sie normalerweise in gekürzter Form. Grundsätzlich immer erfolgt eine Kürzung von IDM-Strings (in "…") auf maximal 40 Zeichen. Ihre Gesamtlänge wird in [ ] angehängt. Bytegrößen-Angaben erfolgen gekürzt auf Kilo-, Mega oder Gigabyte (k/m/g).

Definition

Ab IDM-Version A.05.02.g:

```
void dumpstate
(
  {   anyvalue Filename input
  { , enum     State    := dump_error input } }
)
```

```
void dumpstate
(
  { { anyvalue Filename input, }
      enum     State    := dump_error input }
)
```

Bis IDM-Version A.05.02.f4:

```
void dumpstate
(
  enum State input
)
```

Parameter

anyvalue ***Filename*** *input*

Datei, in die die Zustandsinformationen ausgegeben werden (optionaler Parameter, verfügbar ab IDM-Version A.05.02.g).
Standardwert: Wenn der Parameter fehlt, werden die Zustandsinformationen in die Trace- bzw. Log-Datei ausgegeben.
Bis einschließlich IDM-Version A.05.02.f4 erfolgt die Ausgabe immer in die Trace- bzw. Log-Datei.

enum ***State*** *:= dump_error input*

Dieser Parameter beeinflusst, welche Abschnitte der Zustandsinformationen herausgeschrieben werden.
Ab IDM-Version A.05.02.g ist dieser Parameter optional. Wenn der Parameter fehlt, gilt der Standardwert *dump_error*.

Wertebereich

dump_all
: Alle Abschnitte werden gekürzt herausgeschrieben.
Entspricht der Ausgabe bei einem FATAL ERROR.

dump_error
: Die Abschnitte ERRORS, CALLSTACK und EVENTS werden gekürzt herausgeschrieben.
Dies ist auch die normale Ausgabe im Falle eines EVAL ERRORS.

dump_events
: Die Abschnitte THISEVENTS und EVENT QUEUE werden ungekürzt herausgeschrieben.

dump_full
: Alle Abschnitte werden ungekürzt herausgeschrieben.

dump_locked
: Der Abschnitt SLOTS wird ungekürzt herausgeschrieben. Zusätzlich werden für gesperrte (locked) Objekte deren Attributwerte herausgeschrieben.

dump_memory
: Der Abschnitt MEMORY wird ungekürzt herausgeschrieben.

dump_none
: Nichts passiert (kein Herausschreiben).

dump_process
: Der Abschnitt PROCESS wird ungekürzt herausgeschrieben.

dump_short
: Alle Abschnitte (außer SLOTS) werden gekürzt herausgeschrieben.

dump_slots
: Der Abschnitt SLOTS wird ungekürzt herausgeschrieben.

dump_stack
: Der Abschnitt CALLSTACK wird ungekürzt herausgeschrieben.

dump_usage
: Der Abschnitt USAGE wird ungekürzt herausgeschrieben.

dump_uservisible
: Der Abschnitt VISIBLE OBJECTS wird ungekürzt für alle sichtbaren Toplevel-Objekte inklusive deren Kindobjekte, vordefinierten und benutzerdefinierten Attribute herausgeschrieben.

dump_visible
: Der Abschnitt VISIBLE OBJECTS wird ungekürzt herausgeschrieben.

Die Ausgabe von IDM Zustandsinformationen kann auch über die Schnittstellenfunktion **DM_DumpState** und die Startoptionen **-IDMdumpstate <enum>** und **-IDMdumpstateseverity <string>** erreicht werden.

Beispiel

```
dialog D

window Wi
{
  .title "dumpstate()-example";

  edittext Et
  {
    .content "66";
    .xauto 0;

    on deselect_enter
    {
      variable string Content := this.content;

      if fail(Pg.curvalue := atoi(Content)) then
        print "Konvertierungsfehler!";
        dumpstate(dump_error);
      endif
    }
  }

  pushbutton Pb
  {
```

```
    .ytop 33;
    .text "dump objects";

    on select
    {
      dumpstate(dump_visible);
    }
  }

  progressbar Pg
  {
    .yauto -1;
    .xauto 0;
  }

  on close
  {
    exit();
  }
}
```

Verfügbarkeit

IDM-Versionen A.05.01.g3, A.05.01.h, ab A.05.02.e

Siehe auch

Kapitel „Dumpstate (Zustandsinformationen)“ im Handbuch „Entwicklungsumgebung“

11.12 exchange()

Mit dieser Funktion können zwei Werte einer Sammlung (Datentypen *hash*, *list*, *matrix*, *refvec* und *vector*) vertauscht werden. Außerdem ist der Datentyp *string* erlaubt um einzelne Zeichen innerhalb eines Strings zu vertauschen.

Die beiden Positionsangaben (*Pos1* und *Pos2*) geben die Indexpositionen an, deren Werte vertauscht werden. Die Positionsangaben müssen >= *1* sein und dürfen nicht über der möglichen Anzahl von Elementen in der Sammlung liegen.

Im Fehlerfall wird der Funktionsaufruf mit einem `fail` abgebrochen.

Definition

```
anyvalue exchange
(
     anyvalue ListValue input,
     integer  Pos1 input,
     integer  Pos2 input
 { , enum     Dir  := dir_row input }
)
```

Parameter

anyvalue *ListValue* input

In diesem Parameter wird die Sammlung bzw. der String angegeben in der Werte ausgetauscht werden sollen.

integer *Pos1* input

In diesem Parameter wird die erste Indexposition angegeben, deren Wert mit dem der zweiten Indexposition (*Pos2*) vertauscht wird. Erlaubt sind Werte >= *1*. Bei *matrix*-Sammlungen stellt *Pos1* je nach *Dir*-Parameter entweder eine Zeilenposition (*dir_row*) oder eine Spaltenposition (*dir_column*) dar.

integer *Pos2* input

In diesem Parameter wird die zweite Indexposition angegeben, deren Wert mit dem der ersten Indexposition (*Pos1*) vertauscht wird. Erlaubt sind Werte >= *1*. Bei *matrix*-Sammlungen stellt *Pos2* je nach *Dir*-Parameter entweder eine Zeilenposition (*dir_row*) oder eine Spaltenposition (*dir_column*) dar.

*enum **Dir** := dir_row input*

In diesem optionalen Parameter wird die Ausrichtung der Indexposition definiert, was für eine Matrix von Bedeutung ist. Dabei bedeutet der Standardwert *dir_row*, dass in der Matrix Zeilen vertauscht werden. Bei *dir_column* werden Spalten vertauscht.

Rückgabewert

Die Funktion gibt die veränderte Sammlung bzw. den veränderten String zurück.

Besonderheiten

- *hash*
  Da dieser Sammlungsdatentyp keinen definierten Indexbereich besitzt, erfolgt das Austauschen bezogen auf die *integer*-Indizes. Es wird also von einem *hash* mit einer Indizierung von *1,2,3…* ausgegangen.
- *matrix*
  Das Austauschen in einer Matrix erfolgt immer zeilen- oder spaltenweise. Die Steuerung erfolgt über den *Dir*-Parameter. Standardmäßig (*dir_row*) werden ganze Zeilen getauscht. Bei Angabe von *dir_column* werden ganze Spalten getauscht. Dies bedeutet auch, dass sich die Positionsangabe im ersten Fall auf eine Zeile und im zweiten Fall auf eine Spalte bezieht.
- *string*
  Hier bezieht sich die Position auf die Zeichenposition im String, d.h. der String wird als Array von Zeichen behandelt.
- *Default-Werte*
  Die Sammlungsdatentypen *list*, *hash* und *matrix* erlauben Default-Werte. Der Austausch solcher geerbten Werte an gültigen Positionen wird konsistent durchgeführt.

Beispiele

- Austauschen von Werten in einer Liste

```
variable list L := ["a", "c", "b", "d"];
print exchange(L, 2, 3);
```

Ausgabe

```
["a","b","c","d"]
```

- Austauschen von zwei Zeilen einer Matrix

```
variable matrix M := [ [0,0]=>"?", [1,1]=>"a", [1,2]=>"b",
                       [2,2]=>"end", [3,1]=>"c" ];
M := exchange(M, 2, 3);
print M;
```

Ausgabe

```
[[1,1]=>"a",[1,2]=>"b",[2,1]=>"c",[2,2]=>"?",[3,1]=>"?",[3,2]=>"end"]
```

- Austausch von zwei Zeichen in einem String

```
print exchange("X-Y", 1, 3);
```

Ausgabe

```
"Y-X"
```

- Austausch von zwei Elementen in einem Hash mit *integer*-Indizierung

```
variable hash H := [2=>.xleft, 31=>.width, 33=>.ytop];
print exchange(H, 31, 32);
```

Ausgabe

```
[2=>.xleft,32=>.width,33=>.ytop]
```

» Austausch von Werten in einer *vector*-Liste

```
variable vector[boolean] V := [true, false];
print exchange(V, 1, 2);
```

Ausgabe

```
[false,true]
```

» Austausch der letzten beiden Werte einer *refvec*-Liste

```
variable refvec R := [PbApply, PbCancel, PbRevert];
print exchange(R, 2, 3);
```

Ausgabe

```
[PbApply,PbRevert,PbCancel]
```

Verfügbarkeit

Ab IDM-Version A.06.02.g

11.13 execute()

Mit Hilfe dieser Funktion kann aus dem Dialogskript ein anderes Programm gestartet werden. Je nach dem zugrundeliegendem Betriebssystem werden hier unterschiedliche Arten von zu startenden Programmen unterstützt.

Definition

```
boolean execute
(
      string  Command   input
  { , string  Arguments input }
  { , boolean Synchronous := false  input }
  { , enum ExeType    := exenormal  input }
  { , enum WindowType := showwindow input }
  {   object  Object input
    , integer Event  input
  { , anyvalue ReplyData input } }
)
```

Parameter

string *Command* input

In diesem Parameter wird der Name des auszuführenden Programms und dessen Pfad übergeben. Dabei kann der Pfad in der für das Betriebssystem üblichen Art angegeben werden, relative Pfade sind auch erlaubt. Zusätzlich kann der Pfad auch in der für den IDM üblichen Art und Weise mit Hilfe von Umgebungsvariablen definiert angegeben werden.
Wenn diese Funktion auf Microsoft Windows benutzt wird, kann hier stattdessen ein spezielles Windows-Kommando angegeben werden, das ausgeführt werden soll. In diesem Fall startet Microsoft Windows dann das zu dem Dokument gehörende Programm und führt den angegebenen Befehl aus, vorausgesetzt, dass der Befehl vom Programm unterstützt wird.
Zur Zeit unterstützt Microsoft Windows folgende Befehle: „open“, „print“ und „explore“. Alle aktuell gültigen Kommandos können dem Microsoft Windows Handbuch für die Funktion **ShellExecute()** entnommen werden. Wenn dieses vom Dokument abhängige Starten eines Programms genutzt werden soll, muss beim Parameter *ExeType execommand* angegeben werden.

*string **Arguments** input*

Dieser Parameter ist optional. In diesem Parameter können die Argumente für das zu startende Programm übergeben werden.

*boolean **Synchronous** := false input*

In diesem optionalen Parameter kann angegeben werden, ob das Programm synchron oder asynchron zu dem aktuellen Programm gestartet werden soll. Beim synchronen Start wartet der IDM bis das gestartete Programm beendet worden ist. Während dieser Zeit ist keinerlei Verarbeitung im IDM möglich.

Hinweis

Der synchrone Modus ist nicht zu empfehlen, da dies auf jeden Fall die weitere Bearbeitung des aktuellen Programms durch den Benutzer verhindert. In diesem Fall werden **alle** Ereignisse in eine Queue gestellt, bis die Verarbeitung wieder fortgesetzt wird.

*enum **ExeType** := exenormal input*

In diesem optionalen Parameter kann der Typ des auszuführenden Programms angegeben werden. Dieser Parameter wird nur auf dem Betriebssystemen Microsoft Windows ausgewertet.

Wertebereich

exenormal
: Bei dem Programm handelt es sich um ein „normales“ Programm.

exeshell
: Bei dem angegebenen Programm handelt es sich um ein Programm, das nur innerhalb einer Shell gestartet werden kann, z.B. ein Kopierkommando. Daher wird für das angegebene Programm zuerst eine Shell und dann erst das eigentliche Programm gestartet.

execommand (nur IDM FÜR WINDOWS)
: Bei dem angegebenen Programm handelt es sich um einen Befehl, der von dem zum Dokument gehörenden Programm ausgeführt werden soll. In diesem Fall ist es nicht möglich, den Rückgabewert des Programms abzufragen. Zusätzlich wird bei dieser Art von Programm der *Synchronous*-Parameter ignoriert (siehe Parameter *Command*).

*enum **WindowType** := showwindow input*

In diesem optionalen Parameter kann die Art, wie das Startfenster des gestarteten Programms erscheinen soll, angegeben werden. Dieser Parameter wird nur auf dem Betriebssystemen Microsoft Windows ausgewertet.

Wertebereich

hidewindow
: Das Startfenster des zu startenden Programms soll verborgen bleiben, da in der Regel nur ein Befehl ausgeführt werden soll.

maxwindow
: Hiermit wird die angegebene Anwendung aufgefordert maximiert zu starten.

minwindow
: Das Startfenster des zu startenden Programms soll verkleinert zum Symbol geöffnet werden.

showinactive
: Das Startfenster des zu startenden Programms soll zwar geöffnet werden, soll aber nicht das aktive Fenster werden.

showwindow
: Das Startfenster des zu startenden Programms soll das aktive Fenster werden.

Achtung

Die gestartete Anwendung muss diesen Parameter nicht beachten.
Wenn der *ExeType*-Parameter den Wert *exeshell* besitzt, wird der Kommandointerpreter maximiert gestartet und nicht die Anwendung, die der Kommandointerpreter startet.

object ***Object*** *input*

In diesem optionalen Parameter kann ein Objekt angegeben werden, an das ein externes Ereignis geschickt werden soll, wenn das zu startende Programm beendet worden ist. Wenn dieser Parameter angegeben ist, muss auch der nachfolgende Parameter *Event* angegeben werden.

integer ***Event*** *input*

Dieser Parameter darf und muss nur angegeben werden, wenn der Parameter *Object* angegeben worden ist. In diesem Parameter wird dann die Nummer des externen Ereignisses angegeben, das an das angegebene Objekt nach Beendigung des Programms geschickt werden soll.

anyvalue ***ReplyData*** *input*

Dieser optionale Parameter darf nur dann angegeben werden, wenn ein Objekt für ein externes Ereignis angegeben worden ist. In diesem Parameter wird dann ein beliebiger Wert angegeben, der an die Regel nach Ende des Programms übergeben werden soll.

Rückgabewert

true

Programm konnte gestartet werden.

false

Das Programm konnte nicht gestartet werden.

Reaktion auf Beendigung des Programms

Um auf die Beendigung eines Programms zu reagieren, bietet der IDM die Möglichkeit, ein externes Ereignis mit dem Exitcode des Programms und einem benutzerdefinierten Wert zu schicken. Daher hat die Regel, die auf das Ende des Programms reagieren soll, zwei Parameter. Im ersten *integer*-Parameter wird der Exitcode des Programms übergeben, im zweiten ein *anyvalue*-Parameter, in dem der beim Aufruf von **execute()** angegebene Wert übergeben wird

Auswertung der Parameter

Die Auswertung der einzelnen Parameter ist extrem plattformabhängig. Dies wird in der nachfolgenden Tabelle dargestellt:

	Synchronous	ExeType	WindowType	Externes Ereignis
Windows	wird unterstützt	wird unterstützt	wird unterstützt	wird unterstützt (nicht bei *ExeType = execommand*)
Motif	wird ignoriert	wird ignoriert	wird ignoriert	wird unterstützt

Unter Windows kann bei *ExeType = execommand* nicht auf den Rückgabewert des Kommandos gewartet werden.

Unter Windows kann auf maximal 32 Prozesse im Dialog reagiert werden.

Hinweis zu Anführungszeichen

Ein Einschließen des *Command*-Parameters in Anführungszeichen ist nicht notwendig, für die weiteren Parameter hingegen schon.

Zur Vereinfachung wird unter MICROSOFT WINDOWS der *Command*-Parameter automatisch mit doppelten Anführungszeichen eingeschlossen, wenn dieser Parameter Leerzeichen aber keine doppelten Anführungszeichen enthält. Zusätzlich muss hier der *ExeType*-Parameter auf den Wert *exenormal* oder *exeshell* gesetzt werden.

Außerdem werden aus dem *Command*- und *Arguments*-Parametern zusammengesetzte Kommandos automatisch mit Anführungszeichen eingeschlossen, wenn der *Command*-Parameter mit doppeltem Anführungszeichen (schon angegeben oder automatisch hinzugefügt) beginnt und der *ExeType*-Parameter den Wert *exeshell* besitzt.

Sollte dieses Verhalten nicht gewünscht sein, so kann der *Command*-Parameter leer (*null* oder *""*) und das eigentlich auszuführende Kommando – in diesem Fall zwingend – im *Arguments*-Parameter übergeben werden.

Beispiel

```
dialog Exe

on dialog start
{
  variable boolean Started;

  !! start idm without waiting and event
  Started := execute("idm", "-IDMversion");
  print "idm started";
  print Started;

  !! start with event
  Started := execute("idm", "-IDMversion", this, 100, "testmessage");
  print "External event 100 should appear";
  print Started;

  !! start with options for the startwindow
  !! idm ignores this option
  Started := execute ("idm", "-IDMversion", hidewindow, this, 100);
  print "Window should not appear";
  print Started;

  !! start idm synchronous
  Started := execute ("idm", "-IDMversion", true);
  print "Synchronous start";
  print Started;

  !! example with searchsymbol
```

```
  Started := execute ("IDM_PATH:idm", "-IDMversion");
  print "Started with searchsymbol";
  print Started;
}

on dialog extevent 100 (integer Exitcode, anyvalue Message)
{
  print "Returnvalue: " + itoa(Exitcode) + " " + Message;
}
```

11.14 exit()

Diese Funktion dient einem kontrollierten Verlassen aller Dialoge. Alle laufenden Dialoge werden normalerweise beendet, d.h. ihre *on dialog finish* Regeln werden aufgerufen. Danach wird die Ereignisschleife des IDM verlassen. Die Dialoge werden aber nicht gelöscht.

Definition

```
void exit
(
  { boolean StopDialog := true input }
)
```

Parameter

boolean ***StopDialog*** *:= true input*

Ist dieser Parameter mit *true* belegt (Default) kommt es zu oben beschriebenem Verhalten. Wird hier hingegen *false* übergeben kommt es zum Beenden der Ereignisschleife ohne das der Dialog beendet wird, d.h. es wird auch nicht die *finish*-Regel des Dialog ausgeführt.

Beispiel

```
on END_Button select
{
  exit();
}
```

Wichtig

Eine Regel wird mit Hilfe der **exit()**-Funktion nicht sofort verlassen, sondern wird noch bis zum Ende abgearbeitet. Soll die Regel sofort verlassen werden, muss nach dem **exit()**-Aufruf noch ein `return` stehen.

```
on END_Button select
{
  exit();
  print "Wird noch ausgefuehrt";
}

on END_Button select
{
  exit();
  return;
  print "Wird nicht mehr ausgefuehrt";
}
```

Siehe auch

Eingebaute Funktion **stop()**

11.15 fail()

Diese Funktion dient dazu, den fehlerlosen Ablauf einer Funktion zu prüfen. Wenn der in **fail()** enthaltene Funktionsaufruf einen Fehler zurückliefert, so kann durch den Einsatz dieser Funktion verhindert werden, dass dieser Fehler in der Fehlerdatei mitprotokolliert wird. Die Anwendung kann dann selber auf diesen Fehler reagieren.

Definition

```
boolean fail
(
  anyvalue Expression input
)
```

Parameter

anyvalue *Expression* input

In diesem Parameter kann ein beliebiger Ausdruck enthalten sein, der eine interne Funktion des ISA Dialog Managers benutzt.

Rückgabewert

true

Bei dem Aufruf der enthaltenen Funktion ist ein Fehler passiert.

false

Die Funktion wurde korrekt durchlaufen.

Abfangen und Weiterleiten von Fehlern

Für das Prüfen und Abfangen von Fehlern mit **fail()** sowie das Weiterleiten durch `pass` und `throw` gilt folgendes:

- Aus überdefinierten **:get()**- und **:set()**-Methoden sowie aus benutzerdefinierten Regeln, Methoden und Simulationsregeln werden Fehler an den Aufrufer weitergeleitet.
- Fehler aus den Methoden **:init()**, **:clean()**, **:setclip()** und **:action()** werden nicht an den Aufrufer weitergeleitet.

Beispiel

Eine typische Anwendung dieser Funktion hat die Form:

```
if ( fail( atoi(TextVariable) ) )
then
  Value := -1;                  /* Fehlerfall */
else
  Value := atoi(TextVariable);  /* Normalfall */
endif
```

Obiges Beispiel belegt die Variable Wert mit *-1*, wenn die Umwandlung von *TextVariable* in einen Integerwert nicht erfolgreich war.

Der Fehlerfall tritt beispielsweise dann auf, wenn der Inhalt der Textvariablen "123abc45" war. Im Normalfall enthält die Textvariable den Inhalt "12398745".

Die Funktion **fail()** gibt den booleschen Wert *true* zurück, wenn die eingeschlossene Funktion, im Beispiel *atoi(Textvariable)*, nicht durchgeführt werden konnte. Konnte die eingeschlossene Funktion ausgeführt werden, so liefert **fail()** den Wert *false*.

11.16 find()

Diese Funktion sucht einen angegebenen Wert in einer Werteliste und gibt den ersten gefundenen Index zurück an der dieser Suchwert zu finden ist.

Die Suche schließt dabei niemals das Standardelement (also z.B. Index *[0]* oder *[0,0]*) ein.

Durch Angabe eines Start- und Endindex kann die Suche beschränkt werden. Die gültigen Werte sind:

- Kein Wert angegeben.
- *integer*-Wert im Bereich von *1 … Feldgröße* (muss aber *> 0* sein) für Sammlungen mit den Datentypen *vector*, *refvec*, *list* oder *hash*.
- *index*-Wert im Bereich von *[1,1]* bis *[rowcount,colcount]* wobei auch hier *rowcount* und *colcount* *> 0* sein müssen.
- Gültiger Index in einem *hash*.

Bei der Suche nach Strings kann auch „case-insensitive“ oder nach dem ersten Vorkommen als Präfix gesucht werden.

Da die Indizes von Hashes keiner Ordnung unterworfen sind, sollte als Index nur ein *integer*-Wert im Bereich *1 … itemcount()* angegeben werden, der die Position des Start- bzw. End-Elements angibt.

Bei zweidimensionalen Feldern (analog zur **:find()**-Methode für *.content[]* am ***tablefield***) bewirkt die Angabe eines Indexbereichs *[Sy,Sx]* bis *[Ey,Ex]* die Einschränkung des Suchbereichs auf den Bereich *[min(Sy,Ey),min(Sx,Ex)] … [max(Sy,Ey),max(Sx,Ex)]*. Somit ist die Einschränkung auf Zeilen und Spalten problemlos möglich.

Definition

```
anyvalue find
(
      anyvalue ListValue  input,
      anyvalue Value      input
  { , anyvalue StartIndex input
  { , anyvalue EndIndex   input } }
  { , boolean  CaseSensitive := false       input }
  { , enum     MatchType     := match_exact input }
)
```

Parameter

anyvalue *ListValue* input

In diesem Parameter wird ein Listenwert erwartet in dem gesucht wird.

anyvalue *Value* input

Nach dem in diesem Parameter angegebenen Wert wird gesucht. Grundsätzlich werden nur Werte verglichen, deren Typen „gleich“ bzw. überführbar sind (ein ***text*** wird hierzu in einen *string* überführt).

anyvalue ***StartIndex*** *input*

In diesem optionalen Parameter wird der Startindex angegeben, ab dem in der Sammlung nach einem Wert gesucht werden soll. Wird hier kein Wert angegeben, wird *1* bei eindimensionalen bzw. *[1,1]* bei zweidimensionalen Sammlungen angenommen.

anyvalue ***EndIndex*** *input*

In diesem optionalen Parameter wird der Endindex angegeben, bis zu dem in der Sammlung nach einem Wert gesucht werden soll. Die Angabe ist nur zulässig, wenn auch ein Startindex angegeben wurde.

boolean ***CaseSensitive*** *:= false input*

Dieser optionale Parameter findet nur bei *string*-Werten Beachtung und definiert, ob die Suche unter Beachtung der Groß-/Kleinschreibung erfolgen soll oder nicht. Der Standardwert ist *false*.

enum ***MatchType*** *:= match_exact input*

Dieser optionale Parameter findet nur bei *string*-Werten Beachtung und definiert wie nach einem String gesucht werden soll.

Wertebereich

match_begin
: Findet den ersten String, der mit dem Suchstring beginnt.

match_exact
: Findet den ersten String, der mit dem gesuchten String exakt übereinstimmt.

match_first
: Findet den String, dessen Anfang die größte Übereinstimmung mit dem Suchstring aufweist. Gibt es einen exakt mit dem Suchstring übereinstimmenden String, wird dessen Index zurückgegeben.

match_substr
: Findet den ersten String, der den gesuchten String enthält.

Rückgabewert

0
: Element wurde in der übergebenen Liste (Datentypen *list*, *vector*, *refvec*) nicht gefunden.

[0,0]
: Element wurde in der übergebenen Matrix nicht gefunden.

nothing
: Element wurde in dem übergebenen assoziativen Feld (Datentyp *hash*) nicht gefunden.

sonst
: Index des Elements in der übergebenen Sammlung. Der zurückgegebene Wert hat den Index-Datentyp der übergebenen Sammlung.

Fehlerverhalten

Der Aufruf der Funktion schlägt fehl, wenn der *ListValue*-Parameter keine Sammlung ist, die Indexbereiche außerhalb des gültigen Bereichs liegen oder ein anderer ungültiger Parameterwert vorliegt.

Anmerkung

Wenn beim Durchsuchen von Hashes die Parameter *StartIndex* und *EndIndex* einen anderen Datentyp als *integer* haben, wird die Position der Start- und End-Elemente aus den angegebenen Indizes berechnet. Hierin unterscheidet sich das Durchsuchen von Hashes mit der Funktion **find()** vom Durchsuchen eines assoziativen Arrays mit der Methode **:find()**.

Beispiel

```
dialog D

on dialog start
{
  variable hash Hash := [
    "030"  => "berlin",
    "0711" => "stuttgart",
    "089"  => "munich" ];
  variable matrix Matrix := [
    [1,1] => "area code", [1,2] => "call number",
    [2,1] => 089,         [2,2] => 713400,
    [3,1] => 0711,        [3,2] => 805439 ];
  variable anyvalue Idx;

  Idx := find(Hash, "stuttgart");
  if Idx <> nothing then
    print "Found: stuttgart => " + Idx;
    Idx := find(Matrix, atoi(Idx), [2,1], [3,1]);
    if Idx <> [0,0] then
      print "Call: " + Matrix[first(Idx),2];
    endif
  endif
  exit();
}
```

Siehe auch

Eingebaute Funktion **sort()**

Methode :find()

11.17 first()

Diese Funktion dient dazu, den ersten Wert eines Index abzufragen.

Definition

```
integer first
(
  index Idx input
)
```

Parameter

index *Idx* input

Dieser Parameter bezeichnet den Index, aus dem der erste Anteil – in der Regel der Zeilenanteil – extrahiert werden soll.

Rückgabewert

Als Ergebnis dieser Funktion erhält man den ersten der beiden in dem Index enthaltenen Werte.

Beispiel

```
on Table select
{
  variable integer Row := 0;

  !! Abfragen des Zeilenanteils der aktiven Zelle.
  Row := first(Table.activeitem);
}
```

Siehe auch

Eingebaute Funktion **second()**

11.18 getvalue()

Mit Hilfe dieser Funktion können Sie Attribute von Objekten erfragen.

Die dabei zulässigen Attribute für den jeweiligen Objekttyp entnehmen Sie bitte der „Objektreferenz“.

Definition

```
anyvalue getvalue
(
      object    Obj  input,
      attribute Attr input
  { , integer IntIdx input | index IdxIdx input }
)
```

Parameter

object *Obj* input
Dieser Parameter beschreibt das Objekt, dessen Attribut Sie erfragen möchten.

attribute *Attr* input
Dieser Parameter beschreibt das Attribut, das Sie von dem Objekt erfragen möchten.

*integer **IntIdx** input*
*index **IdxIdx** input*
In diesem optionalen Parameter wird der Index des gesuchten Attributes angegeben. Dabei muss hier ein Integerwert angegeben werden, wenn das abgefragte Attribut eindimensional ist (z.B. bei einer Listbox oder einem Poptext), und es muss ein Indexwert angegeben werden, wenn es sich bei dem Attribut um ein zweidimensionales Attribut handelt.

Rückgabewert

Wert des gesuchten Attributs.

Beispiele

» Abfrage des Inhalts in der Zelle *1,1* eines Tablefields

```
getvalue(Tablefield, .content, [1,1]);
```

» Abfrage des Inhalts eines Eingabefeldes

```
getvalue(Edittext, .content);
```

» Mit Hilfe dieser Funktion können auch Attribute über Variablen geändert werden, z.B.

```
rule void Set(attribute A)
{
  setvalue(Win1, A, true);
  Pb1.sensitive := getvalue(Win2, A);
}
```

Siehe auch

Methode :get()

C-Funktion DM_GetValue im Handbuch „C-Schnittstelle - Funktionen“

11.19 getvector()

Diese Funktion kann angewendet werden um von Vektor- oder Matrixattributen (benutzerdefinierten wie auch vordefinierten) den Gesamtwert, das heißt eine Auflistung der indizierten Werte, oder einen Ausschnitt daraus, in einem *vector* zu erhalten.

Definition

```
vector getvector
(
      object    Object     input,
      attribute Attribute  input
  { , anyvalue  FirstIndex input
  { , anyvalue  LastIndex  input } }
)
```

Parameter

object *Object* input

Dieser Parameter definiert das Objekt dessen Attributwerte erfragt werden soll.

attribute *Attribute* input

Dieser Parameter definiert das Attribut welches erfragt werden soll.

anyvalue ***FirstIndex*** *input*

Dieser optionale Parameter definiert den Startindex ab dem die Elementwerte erfragt werden. Für eindimensionale Feldattribute sollte hier ein *integer*-Wert angegeben werden, für zweidimensionale Felder ein *index*-Wert. Der Standardwert für eindimensionale Feldattribute ist dabei der *integer*-Wert *1*, für zweidimensionale Felder der *index*-Wert *[1,1]*.

anyvalue ***LastIndex*** *input*

In diesem optionalen Parameter wird der Endindex angegeben bis zu dem einschließlich die Werte aus dem Attribut in die Resultatswerteliste übernommen werden. Auch hier wird für eindimensionale Feldattribute ein *integer*-Wert erwartet, für zweidimensionale Felder ein *index*-Wert. Wird kein *LastIndex*-Parameter angegeben, so werden alle Werte bis zum Ende des Feldes aufgenommen. Der *LastIndex*-Wert sollte hinter dem *FirstIndex*-Wert liegen.

Rückgabewert

Die vom Objektattribut ermittelten indizierten Werte werden in Form einer Werteliste vom Typ *vector* zurückgegeben.

Fehlerverhalten

Der Aufruf der Funktion schlägt fehl bei einem ungültigen Objekt bzw. Attribut oder bei einem ungültigen Indexbereich.

Beispiel

```
dialog D
{
  string Extra[integer];
  .Extra[1] := "Anna";
  .Extra[2] := "Wolfgang";
}

window Wi
{
  .title "Names";

  listbox Lb
  {
     .xauto 0;  .yauto 0;
     .content[1] "..NAMES..";
     .content[2] "Ines";
     .content[3] "Dieter";
     .content[4] "Hugo";
  }

  on close { exit(); }
}

on dialog start {
  variable vector Names;

  Names := join(getvector(Lb, .content, 2), getvector(D,.Extra));
  setvector(Lb, .content, sort(Names), 2);
}
```

Siehe auch

Eingebaute Funktion **setvector()**

C-Funktionen DM_GetVectorValue(), DM_SetVectorValue()

11.20 indexat()

Diese Funktion liefert den Index einer Sammlung an einer bestimmten Position. Die erlaubten Positionen gehen von *1 … itemcount()* und ermöglichen somit einen Schleifendurchgang durch alle indizierten Werte.

Für Werte des Typs *list*, *refvec* und *vector* ist die Position mit dem eigentlichen Index identisch. Für einen *hash*-Wert gibt es keine definierte Reihenfolge der gelieferten Indizes. Für *matrix*-Werte erfolgt die Abbildung der aufsteigenden Positionen auf eine Ordnung, bei der alle (Spalten-)Werte einer Zeile aufsteigend hintereinander folgen.

Die Funktion gibt einen Fehler (`fail`) zurück falls die Position außerhalb des erlaubten Bereichs liegt oder es sich beim *Value*-Parameter nicht um eine Sammlung handelt.

Definition

```
anyvalue indexat
(
  anyvalue Value input,
  integer  Pos   input
)
```

Parameter

anyvalue *Value* input
In diesem Parameter wird die Werteliste angegeben, von welcher der Index erfragt werden soll.

integer *Pos* input
Position, für welche der Indexwert ermittelt werden soll.

Rückgabewert

Indexwert an der als Parameter übergebenen Position.

Beispiel

```
dialog D

on dialog start
{
  variable matrix Matrix := [
    [0,0] => "-?-",
    [1,1] => "germany",
    [1,2] => "berlin",
    [2,1] => "france" ];
  variable integer Pos;

  /* print the Matrix elements [1,1] [1,2] ... [2,2] */
  for Pos:=1 to itemcount(Matrix) do
    print sprintf("%s : %s", indexat(Matrix, Pos), valueat(Matrix, Pos));
```

```
  endfor
  exit();
}
```

Siehe auch

Eingebaute Funktionen itemcount(), **keys()**, **valueat()**

Methode index

C-Funktion DM_ValueIndex()

11.21 inherited()

Diese Funktion prüft, ob der Wert geerbt ist. Sie liefert *true*, wenn der zuletzt ausgelesene Attributwert von einem Modell geerbt wurde; ansonsten liefert sie false.

Definition

```
boolean inherited
(
)
```

Rückgabewert

true
: Der zuletzt abgefragte Attributwert wurde geerbt.

false
: Der zuletzt abgefragte Attributwert wurde nicht geerbt.

Beispiel

Abfrage des Inhalts eines Eingabefeldes und anschließende Überprüfung, ob dieser Wert von einem zugrundeliegenden Modell geerbt worden ist.

```
print Edittext.content;
print inherited();
// true falls der Inhalt vom Modell oder Default geerbt wurde.
```

11.22 insert()

Mit dieser Funktion kann in einer Sammlung (Datentypen *hash*, *list*, *matrix*, *refvec* und *vector*) oder einem String ein neuer Wert einmal oder mehrmals eingefügt werden.

Das Einfügen der neuen Werte erfolgt dabei vor der angegebenen Position. Die Positionsangabe muss >= *1* sein, d.h. ein Einfügen von Default-Werten an den Indizes *[0]* bzw. *[0,0]* ist mit dieser Funktion nicht möglich. Die größtmögliche erlaubte Position ist um *1* höher als der aktuelle *countof(ListValue)* um das Anhängen an eine bestehende Sammlung zu ermöglichen.

Allgemein gilt, dass der Datentyp des (optionalen) neuen Wertes ein Skalar sein muss und zum Wertetyp der Sammlung passen muss.

Die Wiederholungszahl des Einfügens (*Count*-Parameter) muss >= *0* sein.

Im Fehlerfall wird der Funktionsaufruf mit einem `fail` abgebrochen.

Definition

```
anyalue insert
(
      anyvalue ListValue  input,
      integer  Pos   input
  { , integer  Count := 1 input }
  { , enum     Dir   := dir_row input }
  { , anyvalue NewValue   input }
)
```

Parameter

anyvalue *ListValue* input

In diesem Parameter wird die Sammlung bzw. der String angegeben in die ein Wert eingefügt werden soll.

integer *Pos* input

In diesem Parameter wird die Indexposition angegeben, vor der das Einfügen erfolgen soll. Erlaubt sind Werte >= *1*. Bei *matrix*-Sammlungen stellt *Pos* je nach *Dir*-Parameter entweder eine Zeilenposition (*dir_row*) oder eine Spaltenposition (*dir_column*) dar.

*integer **Count** := 1 input*

In diesem optionalen Parameter wird die Anzahl angegeben, wie oft ein neuer Wert (bzw. wie viele Zeilen oder Spalten) eingefügt werden soll. Erlaubt sind Werte >= *0*, wobei bei *0* kein Einfügen erfolgt.

*enum **Dir** := dir_row input*

In diesem optionalen Parameter wird die Ausrichtung der Indexposition definiert, was für eine Matrix von Bedeutung ist. Dabei bedeutet der Standardwert *dir_row*, dass in der Matrix Zeilen eingefügt werden. Bei *dir_column* werden Spalten eingefügt.

anyvalue ***NewValue*** *input*

Dieser optionale Parameter definiert den neu einzufügenden Wert. Der Wert muss zum Wertetyp der Sammlung passen.

Rückgabewert

Die Funktion gibt die veränderte Sammlung bzw. den veränderten String zurück.

Besonderheiten

» *hash*
Da dieser Sammlungsdatentyp keinen definierten Indexbereich besitzt, erfolgt das Einfügen bezogen auf die *integer*-Indizes. Es wird also von einem *hash* mit einer Indizierung von *1,2,3…* ausgegangen. Es gibt keine Beschränkungen hinsichtlich der größtmöglichen Indexposition.

» *matrix*
Das Einfügen in eine Matrix erfolgt immer zeilen- oder spaltenweise. Die Steuerung erfolgt über den *Dir*-Parameter. Standardmäßig (*dir_row*) werden neue Zeilen eingefügt. Bei Angabe von *dir_column* werden neue Spalten eingefügt. Dies bedeutet auch, dass sich die Positionsangabe im ersten Fall auf eine Zeile und im zweiten Fall auf eine Spalte bezieht.

» *refvec*
Das Einfügen von *null*-Objekten ist nicht erlaubt. Ein mehrfaches Einfügen wird eben so wenig unterstützt, da die gleiche Objekt-Id nur einmal vorhanden sein darf. Falls die Objekt-Id schon vorhanden ist, erfolgt ein Umpositionieren.

» *string*
Hier bezieht sich die Position auf die Zeichenposition im String, d.h. der String wird als Array von Zeichen behandelt.

» *Default-Werte*
Grundsätzlich macht es Sinn beim Einfügen einen neuen Wert anzugeben. Wird dieser weggelassen, so hängt es vom Sammlungstyp ab, welche Werte an den eingefügten Positionen stehen. Die Sammlungstypen *list*, *hash* und *matrix* erlauben Default-Werte, sodass beim Einfügen ohne neuen Wert die Default-Werte an den eingefügten Positionen erscheinen.

Beispiele

» Mehrfaches einfügen des Strings *"X"* in eine Liste

```
variable list L := ["a", "b", "c"];
print insert(L, 1, 2, "X");
```

Ausgabe

```
["X","X","a","b","c"]
```

» Anhängen einer Zeile an das Ende einer Matrix

```
variable matrix M := [ [0,0]=>"?", [1,1]=>"a1", [1,2]=>"b1",
                       [2,1]=>"a2", [2,2]=>"b2" ];
M := insert(M, 3, "NEW");
print M;
```

Ausgabe

```
[[1,1]=>"a1",[1,2]=>"b1",[2,1]=>"a2",[2,2]=>"b2",[3,1]=>"NEW",
[3,2]=>"NEW"]
```

Einfügen einer neuen 1. Spalte ohne Wert

```
M := insert(M, 1, dir_column);
print M;
```

Ausgabe

```
[[1,1]=>"?", [1,2]=>"a1",[1,3]=>"b1",[2,1]=>"?",[2,2]=>"a2",[2,3]=>"b2",
[3,1]=>"?",[3,2]=>"NEW",[3,3]=>"NEW"]
```

» Einfügen von *"ll"* in einen String

```
print insert("hao", 3, 2, "l");
```

Ausgabe

```
"hallo"
```

» Einfügen in ein Hash mit *integer*-Indizierung

```
variable hash H := [2=>.xleft, 33=>.ytop];
H := insert(H, 10, .width);
print H;
```

Ausgabe

```
[2=>.xleft,10=>.width,34=>.ytop]
```

» Einfügen in eine *vector*-Liste

```
variable vector[boolean] V := [true, true];
print insert(V, 2, 3, false);
```

Ausgabe

```
[true,false,false,false,true]
```

» Einfügen eines schon vorhandenen *refvec*-Elements an einer anderen Stelle

```
variable refvec R := [PbApply, PbCancel, PbRevert];
print insert(R, 1, PbRevert);
```

Ausgabe

```
[pushbutton D.PbRevert,pushbutton D.PbApply,pushbutton D.PbCancel]
```

Verfügbarkeit

Ab IDM-Version A.06.02.g

Siehe auch

Eingebaute Funktionen **append()**, **delete()**

11.23 itemcount()

Diese Funktion liefert die Anzahl von indizierten Werten in einer Sammlung zurück. Ausgenommen von der Anzahl sind der/die Standardwert(e). Zusammen mit **indexat()** und **valueat()** lassen sich somit einfach Schleifen durch beliebige Sammlungen realisieren.

Typischerweise liefern **countof()** und **itemcount()** für Werte des Datentyps *refvec*, *vector* und *list* den gleichen Wert zurück. **itemcount()** bietet aber die generischere Verwendung, wohingegen **countof()** sich für die strukturierte, typangepasste Verwendung besser eignet.

Definition

```
integer itemcount
(
  anyvalue Value input
)
```

Parameter

anyvalue *Value* input
In diesem Parameter wird der Wert angegeben für den die Anzahl ermittelt werden soll.

Rückgabewert

0
Der übergebene Wert ist skalar oder eine leere Sammlung.

1 … 2^{31}
Anzahl der Werte in der Sammlung ohne Standardwerte.

Beispiel

```
dialog D

on dialog start
{
  variable matrix Matrix := [
    [0,0] => "-?-",
    [1,1] => "germany",
    [1,2] => "berlin",
    [2,1] => "france"
    /* [2,2] => inherited from default [0,0] */ ];
  variable integer I;
  variable anyvalue Idx;

  /* print the Matrix values [1,1] [1,2] ... [2,2] */
  for I:=1 to itemcount(Matrix) do
    Idx := indexat(Matrix, I);
    print sprintf("%s : %s", Idx, Matrix[Idx]);
  endfor
```

```
  exit();
}
```

Ausgabe

```
"[1,1] : germany"
"[1,2] : berlin"
"[2,1] : france"
"[2,2] : -?-"
```

Siehe auch

Eingebaute Funktionen **countof()**, **indexat()**, **valueat()**

Methode index

Attribut itemcount

C-Funktion DM_ValueCount()

11.24 itoa()

Diese Funktion wandelt eine Zahl in eine Zeichenkette um (**i**nteger **to** **a**scii).

Definition

```
string itoa
(
  integer IntValue input
)
```

Parameter

integer IntValue input

In diesem Parameter wird die Zahl angegeben, die in einen String umgewandelt werden soll.

Rückgabewert

Als Ergebnis liefert diese Funktion die in eine Zeichenkette umgewandelte Zahl.

Beispiel

Füllen eines Eingabefelds mit einer Zahl.

```
Edittext.content := itoa(5);
```

11.25 join()

Diese Funktion bildet aus den angegebenen Parametern eine neue Werteliste. Der Typ der Werteliste ergibt sich dabei aus dem ersten Aufrufparameter. Ist dieser eine Sammlung oder ein Sammlungsdatentyp (*list*, *vector*, *refvec*, *hash*, *matrix*), so hat die Resultatswerteliste den gleichen Typ. Ein Datentyp als erster Parameterwert fließt grundsätzlich nicht in die Resultatsliste ein. Handelt es sich bei den angegebenen Parametern um Sammlungen, so werden hiervon nur die Werte (ohne Standardwerte) herangezogen. Skalare Werte werden zur Resultatswerteliste hinzugenommen.

Die Reihenfolge der Werte in der Resultatswerteliste entspricht der Reihenfolge der Parameter und bei Sammlungen deren „normaler“ Reihenfolge (siehe hierzu **indexat()**).

Die Indexierung erfolgt normalerweise über den erhöhten letzten Indexwert oder die Werteanzahl (siehe **countof()** und **itemcount()**). Das heißt, die Werte in der Resultatsliste erhalten normalerweise die Indexierung über *1, 2, 3, …* oder *[1,1], [1,2], [1,3], ….*

Handelt es sich aber beim Typ der Resultatsliste wie auch beim Parameterwert um einen *hash*, so wird der Index der Hash-Werte übernommen. Hinzugefügte skalare Parameter erhalten als Index den erhöhten **itemcount()**-Wert der Resultatsliste.

Zu beachten ist bei der Verwendung von *refvec* als Resultatslistentyp, dass es zu einer Rückgabe eines Fehlers (`fail`) kommt, falls ein Wert nicht vom Datentyp *object* ist.

Definition

```
anyvalue join
(
      anyvalue Value input
  { , anyvalue Par2  input }
  ...
  { , anyvalue Par16 input }
)
```

Parameter

anyvalue *Value* input
: In diesem Parameter wird der Wert angegeben auf den die Funktion angewendet wird. Dieser Parameter wird auch genutzt um den Typ für die Resultatsliste zu bestimmen.

*anyvalue **Par2** input*

…

*anyvalue **Par16** input*
: Zusätzliche optionale Parameter auf welche die Funktion angewendet wird.

Rückgabewert

Sammlung, die alle Werte aus den übergebenen Parametern enthält. Wenn der erste Parameter ein Sammlungsdatentyp oder eine Sammlung ist, hat die zurückgegebene Sammlung diesen Datentyp. Ein Datentyp als erster Parameter wird nicht in die zurückgegebene Sammlung übernommen.

Änderung ab IDM-Version A.06.01.b bei der Übergabe des Datentyps *string* als ersten Parameter

Modification as of IDM Version A.06.01.b When Passing the Data Type *string* as First Parameter

Wird der Funktion **join()** als erstes Argument der Datentyp *string* übergeben, dann werden die nachfolgenden Argumente in den Datentyp *string* konvertiert und anschließend verkettet. Sind die nachfolgenden Argumente Sammlungen, werden zunächst deren Werte in der natürlichen Reihenfolge des Index zu einem String verkettet.

If the data type *string* is passed as first argument to the function **join()**, the subsequent arguments are converted to the data type *string* and then concatenated. If the subsequent arguments are collections, first their values are concatenated to a string in the natural order of the index.

Falls das erste Argument ein Datentyp, aber weder *string* noch ein Sammlungsdatentyp ist, gibt **join()** einen Fehler zurück.

If the first argument is a data type that is neither *string* nor a collection data type, **join()** returns an error.

Beispiel

```
dialog D

on dialog start
{
  variable hash DomainHash := [
    ".de" =>" germany",
    ".us" => "usa",
    ".fr" => "france",
    ".uk" => "united kingdom" ];
  variable list Countries;
  variable hash Domains;

  /* join the values to a list */
  Countries := join(list, DomainHash, "norway", "spain");
  /* join the values and keys into a hash */
  Domains := join(DomainHash, [".us" => "united states of america"]);

  print Countries;
  print Domains;

  exit();
}
```

Ausgabe

```
["norway","spain","germany","usa","france","united kingdom"]
[".de"=>"germany",".us"=>"united states of america",".fr"=>"france",
".uk"=>"united kingdom"]
```

Siehe auch

Eingebaute Funktion **keys()**

C-Funktion DM_ValueChange()

11.26 keys()

Diese Funktion liefert eine Liste aller Indexwerte einer Sammlung zurück. Die Indizes von Standardwerten fließen dabei nicht in die Liste ein. Wird als Parameter ein Skalar angegeben, so erhält man eine leere Liste.

Definition

```
list keys
(
  anyvalue Value input
)
```

Parameter

anyvalue *Value* input

In diesem Parameter wird der Wert angegeben auf den die Funktion angewendet wird.

Rückgabewert

Liste mit allen Indizes einer Sammlung.

Beispiel

```
dialog D

on dialog start
{
  variable matrix Matrix := [
    [0,0] => "-?-",
    [1,1] => "germany",
    [1,2] => "berlin",
    [2,1] => "france" ];
  variable anyvalue Idx;

  /* print the Matrix elements [1,1] [1,2] ... [2,2] */
  foreach Idx in keys(Matrix) do
    print sprintf("%s : %s", Idx, Matrix[Idx]);
  endfor
  exit();
}
```

Siehe auch

Eingebaute Funktionen **countof()**, **indexat()**, **itemcount()**, **values()**

Kapitel „foreach-Schleife“

11.27 length()

Diese Funktion liefert die Länge eines Strings zurück.

Definition

```
integer length
(
  string Str input
)
```

Parameter

string *Str* input

In diesem Parameter wird der String übergeben, dessen Länge erfragt werden soll.

Rückgabewert

Als Ergebnis liefert diese Funktion die Länge des Strings zurück.

Beispiel

Im DM-Programm ist folgende Zeile gegeben:

```
WnMain.title := "Beispiel";
```

Dann erhält man durch

```
Len := length ( WnMain.title );
print Len;
```

oder

```
Len := length ( "Beispiel" );
print Len;
```

den Wert von „Len“ mit *8*.

11.28 load()

Mit Hilfe dieser Funktion wird ein Dialog hinzugeladen, aber noch nicht gestartet.

Definition

```
object load
(
  string Filename input
)
```

Parameter

string *Filename* input

In diesem Parameter wird der Dateiname des zu ladenden Dialogs angegeben. Die Angabe kann über den im IDM üblichen Mechanismus
<Umgebungvariable>:<Dateiname>
erfolgen, um das Laden so flexibel wie möglich zu gestalten. Die Umgebungsvariable wird dabei als Pfad interpretiert, in dem nach der angegebenen Datei gesucht werden soll.

Rückgabewert

DialogID
Hier wird die ID des neu geladenen Dialogs zurückgeliefert.

null
Der Dialog konnte nicht geladen werden.

Beispiel

Laden eines Dialogs und Starten des neu geladenen Dialogs

```
variable object Dialog := null;

Dialog := load("Suchpfad:Test.dlg");
!! pruefen, ob der Dialog geladen werden konnte
if Dialog <> null then
  !! starten des Dialogs
  run (Dialog);
endif
```

Siehe auch

Eingebaute Funktion **run()**

C-Funktion DM_LoadDialog im Handbuch „C-Schnittstelle - Funktionen“

11.29 loadprofile()

Diese Funktion liest die Werte der konfigurierbaren ***Record***-Instanzen (*.configurable* = *true*) und **globalen Variablen** (deklariert mit **config**) eines Dialogs oder Moduls aus einer Konfigurationsdatei (profile) ein.

Definition

```
boolean loadprofile
(
      string Filename  input
  { , object Module := null input }
)
```

Parameter

string *Filename* input

Dieser Parameter definiert den Dateinamen der Konfigurationsdatei. Es kann ein Dateipfad angegeben werden, der auch eine Umgebungsvariable enthalten darf.

object ***Module*** *:= null input*

Dieser optionale Parameter enthält den Identifikator des Dialogs oder Moduls, dessen ***Record***- und Variablenwerte aus der Datei eingelesen werden sollen.
Bei *Module* = *null* wird die Modul-ID der aktuellen Regel verwendet.

Rückgabewert

true

Das Einlesen der Werte in der Konfigurationsdatei war erfolgreich.

false

Die Werte konnten nicht eingelesen werden.
Dies kann an Fehlern beim Zugriff auf die Datei oder einer ungültigen Modul-ID liegen.

Verfügbarkeit

Ab IDM-Version A.06.02.g

Siehe auch

Eingebaute Funktion **saveprofile()**

C-Funktion DM_LoadProfile()

11.30 max()

Diese Funktion liefert den größten Integer-Wert, der in allen seinen Aufrufparametern vorkommt.

Die Aufrufparameter können dabei skalare Integer-Werte oder Sammlungen (*vector*, *list*, *matrix*, *hash*) sein. Für letztere werden die indizierten Elemente (ohne die Standardwerte) für die Bestimmung des Maximalwerts herangezogen.

Definition

```
integer max
(
      anyvalue Par1  input
  { , anyvalue Par2  input }
  ...
  { , anyvalue Par16 input }
)
```

Parameter

anyvalue *Par1* input

*anyvalue **Par2** input*

…

*anyvalue **Par16** input*

In diesen Parametern werden die Werte angegeben, auf welche die Funktion angewendet wird.

Rückgabewert

Der größte vorkommende Integer-Wert wird geliefert.

Fehlerverhalten

Der Aufruf der Funktion schlägt fehl, wenn in den Argumenten kein skalarer Integer-Wert vorkommt oder Argumente bzw. deren Elemente einen Wert beinhalten, der kein Integer-Wert ist.

Beispiel

```
dialog D

on dialog start
{
  variable list List := [17, 3, 23, 5];

  print max(List, 24, 4);  /* print 24 */
  exit();
}
```

Siehe auch

Eingebaute Funktion **min()**

11.31 min()

Diese Funktion liefert den kleinsten Integer-Wert, der in allen seinen Aufrufparametern vorkommt.

Die Aufrufparameter können dabei skalare Integer-Werte oder Sammlungen (*vector*, *list*, *matrix*, *hash*) sein. Für letztere werden die indizierten Elemente (ohne die Standardwerte) für die Bestimmung des Minimalwerts herangezogen.

Definition

```
integer min
(
      anyvalue Par1  input
  { , anyvalue Par2  input }
  ...
  { , anyvalue Par16 input }
)
```

Parameter

anyvalue *Par1* input

*anyvalue **Par2** input*

…

*anyvalue **Par16** input*

In diesen Parametern werden die Werte angegeben, auf welche die Funktion angewendet wird.

Rückgabewert

Der kleinste vorkommende Integer-Wert wird geliefert.

Fehlerverhalten

Der Aufruf der Funktion schlägt fehl, wenn in den Argumenten kein skalarer Integer-Wert vorkommt oder Argumente bzw. deren Elemente einen Wert beinhalten, der kein Integer-Wert ist.

Beispiel

```
dialog D

on dialog start
{
  variable list List := [17, 3, 23, 5];

  print min(List, 24, 4);  /* print 3 */
  exit();
}
```

Siehe auch

Eingebaute Funktion **max()**

11.32 parsepath()

Mit dieser Funktion kann mithilfe des Objektnamens ein Objekt ermittelt werden.

Definition

```
object parsepath
(
      string  ObjectName input
  { , object  Parent := null input }
  { , object  Dialog := null input }
  { , integer Index  := 0 input }
)
```

Parameter

string *ObjectName* input

Dieser Parameter gibt den Namen des gesuchten Objekts an.
Der im Parameter *ObjectName* angegebene Wert entspricht einem Suchpfad, der vom Objekt, welches im Parameter *Parent* angegeben ist, ausgeht. Ein Leerstring im Parameter *ObjectName* (*""*) entspricht dabei einem leeren Suchpfad.

*object **Parent** := null input*

Dieser Parameter bezeichnet den Vater des gesuchten Objekts. Wenn dieser Parameter angegeben ist, wird unterhalb dieses Objekts nach dem angegebenen Objekt gesucht. Wird für diesen Parameter das *null*-Objekt angegeben, so wird im angegebenen Dialog oder im aktuellen Dialog, wenn kein Dialog angegeben ist, nach dem Objekt gesucht. Wenn nach Ressourcen gesucht wird, muss dieser Parameter mit dem *null*-Objekt belegt sein.
Wird nach Funktionen in Applikationen gesucht, muss die Applikation angegeben werden.

*object **Dialog** := null input*

Dieser optionale Parameter bezeichnet den Dialog des gesuchten Objekts. Wenn dieser Parameter nicht angegeben wird, wird im aktuellen Dialog nach dem Objekt gesucht. Soll in einem Modul nach einem Objekt gesucht werden, so muss das entsprechende Modul in diesem Parameter angegeben werden.

*integer **Index** := 0 input*

Ist *Index* größer als *0*, dann werden in den untergeordneten Kindern des Objekts alle Kinder gesucht auf die der entsprechende *ObjectName* passt. Der Index > *0* gibt der Funktion an, nach dem wievielten Vorkommen von *ObjectName* gesucht werden soll. Ist für diesen Index kein Kind vorhanden, dann wird *null* zurückgegeben.

Rückgabewert

ObjektID
: Das gesuchte Objekt wird zurückgegeben.

null

Das angegebene Objekt wurde nicht gefunden oder der Name ist nicht eindeutig. Das heißt, es gibt keins oder mehrere Objekte mit dem angegebenen Namen.

Beispiele

» Suchen im aktuellen Dialog nach der Farbe „ROT“

```
Obj := parsepath("ROT");
```

» Suchen nach der Regel „Test“ im Modul „M1“

```
Obj := parsepath("Test", null, M1);
```

» Suchen nach dem Pushbutton „OK“ im Fenster „W1“

```
Obj := parsepath("OK", W1);
```

» Suchen nach der Funktion „ApplFunc“ in der Applikation „A1“

```
Obj := parsepath("ApplFunc", A1);
```

» Suchen nach dem I-ten Vorkommen von „SubR“ im Fenster „W“

```
dialog D
window W
{
  record R
  {
    record SubR {}
  }
  child groupbox G
  {
    record SubR {}
  }
  rule void DoSomething()
  {
    variable integer I := 1;
    variable object O := D;
    while (O) do
      O := parsepath("SubR", this, null, I);
      if O <> null then
        I := I + 1;
        print O;
      endif
    endwhile
  }
}
on dialog start
{
```

```
  W:DoSomething();
}
```

Siehe auch

Funktion DM_ParsePath im Handbuch „C-Schnittstelle - Funktionen“

11.33 print()

Diese Funktion dient während der Testphase der Dialogentwicklung der Ausgabe von beliebigen im Dialog definierten Werten. Dazu gehören z.B. die Werte von Variablen und die Inhalte von Attributen.

Hinweis

Beim Aufruf dieser Funktion können die Klammern weggelassen werden.

Definition

```
void print
(
  anyvalue PrintValue input
)
```

Parameter

anyvalue *PrintValue* input

In diesem Parameter wird der Wert angegeben, der ausgedruckt werden soll.

Beispiel

Die folgenden Zeilen sind im DM-Programm angegeben:

```
WnMain.title := "XYZ";
Counter      := 650;
```

Dann werden mit den Anweisungen

```
print WnMain.title;
print Counter;
```

folgendes in der Fehlerdatei ausgegeben:

```
WnMain.title = "XYZ"
Counter = 650
```

11.34 querybox()

Mit der eingebauten Funktion **querybox()** können Messageboxen oder Dialogboxen (Fenster mit gesetztem Attribut *.dialogbox true*) geöffnet werden. Die Regelbearbeitung im Dialog Manager wartet so lange, bis die Messagebox bzw. Dialogbox wieder geschlossen ist.

Definition

```
anyvalue querybox
(
      object  Messagebox input
  { , object  Parent     input }
  { , boolean ShipEvent  := true input }
)
```

Parameter

object *Messagebox* input
Dieser Parameter gibt die zu öffnende Messagebox bzw. Dialogbox an.

*object **Parent** input*
Dieser optionale Parameter gibt das Vaterobjekt an, über dem die Messagebox geöffnet soll. Dieser Parameter darf dabei ein Fenster oder die Null-ID sein. Falls das Fenstersystem es erlaubt, wird die Messagebox über dem Vater-Fenster zentriert dargestellt. Andernfalls wird die Position vom Fenstersystem selbst festgelegt (z.B. Bildschirmmitte).
Dieser Parameter wird bei Dialogboxen ignoriert.

*boolean **ShipEvent** := true input*
Ist dieser Parameter auf *true* gesetzt (Default) wird ein *changed*-Ereignis auf *.visible* an die Dialogbox verschickt. Mit dem Wert *false* wird dies unterdrückt.

Rückgabewert

Messageboxen

button_abort
Die ***messagebox*** wurde mit der Schaltfläche „Abbrechen“ geschlossen.
button_cancel
Die ***messagebox*** wurde mit der Schaltfläche „Abbrechen“ geschlossen.
button_ignore
Die ***messagebox*** wurde mit der Schaltfläche „Ignorieren“ geschlossen.
button_no
Die ***messagebox*** wurde mit der Schaltfläche „Nein“ geschlossen.
button_ok
Die ***messagebox*** wurde mit der Schaltfläche „OK“ geschlossen.
button_retry
Die ***messagebox*** wurde mit der Schaltfläche „Wiederholen“ geschlossen.

button_yes
Die ***messagebox*** wurde mit der Schaltfläche „Ja“ geschlossen.
nobutton
Die ***messagebox*** wurde wegen eines Fehlers nicht geöffnet.

Dialogboxen

Bei einer **Dialogbox** wird der Rückgabewert über die Funktion **closequery()** an der jeweiligen Dialogbox bestimmt.

Hinweis zum IDM für Motif

Bitte beachten Sie in Bezug auf die Anordnung eines Dialogfelds vor oder hinter anderen Fenstern und Dialogfeldern auf dem Bildschirm (Z-Ordnung) den Hinweis im Kapitel „Z-Ordnung von Fenstern und Dialogfeldern“ der „Objektreferenz“.

Beispiel

Öffnen einer Messagebox und abfragen, welchen der angebotenen Pushbuttons der Benutzer selektiert hat.

```
!! Programm wartet bis der Benutzer sich für einen Pushbutton entschieden hat
if button_ok = querybox(MeinMeldungsfenster) then
  !! Benutzer möchte die Aktion ausgeführt haben
```

11.35 queryhelp()

Ohne Argument bzw. mit *null* als Argument versetzt diese Funktion den IDM in den Kontexthilfe-Modus, in dem der nächste Mausklick auf ein Objekt ein *help*-Ereignis für dieses Objekt auslöst.

Wird beim Aufruf eine Objekt-ID übergeben, dann löst die Funktion ein *help*-Ereignis für dieses Objekt aus.

Definition

```
void queryhelp
(
  { object Id := null input }
)
```

Parameter

object ***Id*** *:= null input*

In diesem optionalen Parameter kann ein Objekt angegeben werden, für das ein *help*-Ereignis ausgelöst werden soll.

Verfügbarkeit

Ab IDM-Version A.06.02.g

11.36 random()

Mit Hilfe dieser Funktion kann eine Zufallszahl ermittelt werden

Definition

```
integer random
(
  integer Range := 100 input
)
```

Parameter

*integer **Range** := 100 input*

In diesem Parameter wird der Bereich angegeben, aus dem die Zufallszahl stammen soll.

Rückgabewert

Der Rückgabewert dieser Funktion ist ein integer Wert zwischen 0 und (Range - 1).

Beispiel

```
!! ergibt einen integer-Wert im Bereich 0-4
random(5);
```

11.37 regex()

Mit dieser Funktion kann ein Regulärer Ausdruck auf einen String oder eine Liste von Strings angewendet werden.

Um einen möglichst hohen Funktionsumfang zu ermöglichen, bindet der IDM hierzu die freie PCRE-Bibliothek dynamisch ein. Damit sind Musterausdrücke analog zu PERL möglich. Falls Sie diese Funktion also in ihrem Produkt einsetzen, sollten Sie Lizenzbedingungen und Dokumentation von PCRE (**www.pcre.org**) beachten. Bezüglich der Anbindung ist Kapitel „PCRE-Bibliothek zur Unterstützung Regulärer Ausdrücke“ zu berücksichtigen. Die ISA empfiehlt eine PCRE-Bibliothek in der Version *8.** für ein ordnungsgemäßes Funktionieren. Die PCRE-Funktionen werden dabei für „Pattern-Matching“ und „Capturing“ von String-Teilen benutzt. Der IDM übernimmt dann Auswertung und Ersetzung.

Syntaktisch werden folgende Reguläre Ausdrücke vom IDM akzeptiert um so die Operationen „Matching“ und „Substitution“ zu ermöglichen:

- Matching

```
m/<pattern>/<modifiers>
```

- Substitution

```
s/<pattern>/<replacement>/<modifiers>
```

Für die Konfiguration der Operation sind folgende Modifikatoren (*<modifiers>*) erlaubt:

s	single-line string (PCRE_DOTALL)
m	multi-line (PCRE_MULTILINE)
i	ignore case (PCRE_CASELESS)
g	global search – Mustererkennung wird wiederholt
x	extended (PCRE_EXTENDED)
f	first line (PCRE_FIRSTLINE)
W	unicode (wide) character classes (PCRE_UCP)
X	extra (PCRE_EXTRA)
U	ungreedy (PCRE_UNGREEDY)

In Klammern steht dabei, wie die Modifikatoren an PCRE weitergereicht werden. Deren Anwendung erfolgt also nicht durch den IDM sondern durch die PCRE-Bibliothek.

Ohne Fehlermeldung überlesen werden die Modifikatoren „o“ und „e“.

Die eigentliche Anwendung von *<pattern>* und *<replacement>* erfolgt dann mit der PCRE-Bibliothek. Weitere Details und Informationen zu den Regulären Ausdrücken sind deshalb der Dokumentation der verwendeten PCRE-Bibliothek zu entnehmen.

Der IDM erlaubt auch die Aufteilung der Bestandteile *<pattern>* und *<replacement>* des Regulären Ausdrucks in zwei getrennte Parameter, wobei dann allerdings keine Modifikatoren *<modifiers>* möglich sind. Normalerweise bestimmt die Operation die Art der zurückgegebenen Ergebnisse. Durch einen *Action*-Parameter kann dies aber gesteuert werden um so z.B. nur die Anzahl der gefundenen „Matches“ zu liefern, eine Filterung zu erreichen oder um die Werte der *<pattern>*-Variablen zu erhalten.

Folgende Aktionen mit entsprechenden Auswertungen sind möglich:

Tabelle 2: *Aktionen, Rückgabetypen und Auswertungen der regex-Funktion*

Aktion	Rückgabetyp	Auswertung
regex_ eval	*boolean string list[string]*	Je nach Operation wird entweder zurückgeliefert, ob wenigsten einer der Strings dem Regulären Ausdruck entspricht (Matching-Operation *true* oder *false*). Oder im Substitution-Fall werden die ersetzten String(s) zurückgeliefert, falls das Muster übereinstimmt oder eben der Orginal-String.
regex_ match	*boolean list[string]*	Führt nur eine Prüfung des Musters durch und liefert *true* wenn das Muster übereinstimmt, ansonsten *false*. Bei einer String-Liste werden in die Resultatsliste nur Strings aufgenommen, welche das Muster erfüllen.
regex_ unmatch	*boolean list[string]*	Führt nur eine Prüfung des Musters durch und liefert *false* wenn das Muster übereinstimmt, ansonsten *true*. Bei einer String-Liste werden in die Resultatsliste nur Strings aufgenommen, welche das Muster **nicht** erfüllen
regex_ count	*integer*	Zählt die Anzahl gefundener Muster im String bzw. der String-Liste. Pro String wird maximal *+1* gezählt, sodass bei der Anwendung auf einen Einzel-String als Wert *0* oder *1* herauskommt und bei einer String-Liste ein Wert im Bereich *0 … itemcount()*.
regex_ locate	*integer list[integer]*	Liefert die Zeichenposition der ersten Übereinstimmung bei der Anwendung auf einen Einzel-String. Für eine String-Liste werden die Indexpositionen in der Liste geliefert, bei denen das Mustern erfüllt ist.

Der IDM erlaubt die Anwendung der **regex**-Funktion auf jeden beliebigen Sammlungsdatentyp (*list*, *hash*, *matrix*, *vector*). Eine generierte Rückgabeliste ist aber immer vom Typ *list* ohne Übernahme einer besonderen Indizierung der Ausgangsliste. Vor der Anwendung des Regulären Ausdrucks

werden Werte, die nicht vom Datentyp *string* sind, in einen String umgewandelt, wie es auch z.B. bei *print <Value>;* geschieht.

Definition

```
anyvalue regex
(
      anyvalue StringOrList input,
      string   Pattern input
  { , string   Replace input }
  { , enum     Action  := regex_eval input }
)
```

Parameter

anyvalue *StringOrList* input

Dieser Parameter beinhaltet den String bzw. die Liste von Strings auf die der Reguläre Ausdruck angewendet wird.

string *Pattern* input

Dieser Parameter beinhaltet entweder den Regulären Ausdruck oder den Muster-String (*<pattern>*), falls der Reguläre Ausdruck in *<pattern>* und *<replacement>* aufgeteilt wird.

*string **Replace** input*

Dieser optionale Parameter sollte den Ersetzungs-String (*<replacement>*) beinhalten. wenn für den Funktionsaufruf der Reguläre Ausdruck in *<pattern>* und *<replacement>* aufgetrennt wird.

*enum **Action** := regex_eval input*

Dieser optionale Parameter steuert die Ergebnisauswertung. Folgende Aktionen sind dabei möglich:

regex_eval (Standard)
: Evaluation des Regulären Ausdrucks („Matching“ oder „Substituition“).

regex_match
: Filterung auf Strings, die dem Muster entsprechen.

regex_unmatch
: Filterung auf Strings, die dem Muster **nicht** entsprechen.

regex_count
: Anzahl der gefundenen Muster (*+1* pro Suchstring).

regex_vars
: Liefert den Inhalt aller vom Suchmuster definierten Variablen.

regex_locate
: Liefert die String-Position bzw. Indexposition des gefundenen Musters.

Rückgabewert

Rückgabewert und -typ sind abhängig von der Auswertungsaktion und in „Tabelle 2“ (oben) beschrieben.

Beispiele

1. Auf Ziffern in einem String prüfen

```
print regex("Ist 127 eine Zahl?", "\\d+");
```

Ausgabe

```
true
```

2. Alle Dezimalzahlen durch ein „N“ ersetzen

```
print regex("42 ist größer als 10", "s/(\d+)/N/g");
```

Ausgabe

```
"N ist größer als N"
```

3. Nur Strings ausgeben auf die das Muster passt

```
print regex("127,5", "^\\d+,\\d+$", regex_match);
print regex("1275", "^\\d+,\\d+$", regex_match);
print regex(["3,7", "17,5", "0", "21,03"], "^\\d+,\\d+$", regex_match);
```

Ausgabe

```
"127,5"
""
["3,7","17,5","21,03"]
```

4. Mehrere reguläre Ausdrücke auf eine Liste anwenden
   - nur Werte auflisten die ein Wort beinhalten
   - die Anzahl der Wörter zählen
   - jeden Eintrag in „>> <<“ einrahmen
   - alle Geburtsjahre auflisten

```
variable list BirthDays := ["12-13-1973", "Ina", "1-7-1965", "Udo"];
print regex(BirthDays, "^\\w+$", regex_match);
print regex(BirthDays, "^\\w+$", regex_count);
print regex(BirthDays, "s/(.*)/>> $1 <</", regex_match);
print regex(BirthDays, "s/\\d+-\\d+-(\\d+)/$1/", regex_match);
```

Ausgabe

```
["Ina","Udo"]
2
[">> 12-13-1973 <<",">> Ina <<",">> 1-7-1965 <<",">> Udo <<"]
["1973","1965"]
```

5. Auflistung der Variablenwerte die in einem Regulären Ausdruck enthalten sind

```
print regex("+2500 Euro oder mehr", "(\\d+)\\s+(\\w+)", regex_vars);
```

Ausgabe

```
["2500 Euro","2500","Euro"]
```

6. Auflistung der Zugriffsindizes für die gefundenen Muster in einer Liste oder einem String

```
variable list Locales := [ "de_DE.UTF8", "C.UTF-8", "de_AT.utf8",
                           "en_AU.utf8", "en_ZM", "POSIX", "de_CH.uf8" ];
print regex(Locales, "/^de_/", regex_locate);
print regex("Hallo Welt", "/W/", regex_locate);
```

Ausgabe

```
[1,3,7]
7
```

7. Ausnutzung der automatischen Konvertierung von Werten in einer Liste zu *string*-Werten

```
record Rec4711 {}
print regex([123, winsys_x11, "Bond 007", opt_w2kprefsize_compat,
Rec4711],
            "s/\\d+/N/g");
```

Ausgabe

```
["N","winsys_xN","Bond N","opt_wNkprefsize_compat","RecN"]
```

Verfügbarkeit

Ab IDM-Version A.06.02.g

PCRE-Bibliothek zur Unterstützung Regulärer Ausdrücke

Um Reguläre Ausdrücke über die eingebaute-Funktion **regex()** oder als Format im IDM zu verwenden, ist die freie Bibliothek PCRE (Perl Compatible Regular Expression, siehe hierzu auch **www.pcre.org**) notwendig. Falls Sie diese Funktion also in ihrem Produkt einsetzen, sollten Sie die Lizenzbedingungen von PCRE beachten.

Der IDM setzt eine PCRE-Bibliothek ab der Version *3* mit aktivierter Unicode-Unterstützung und mit der „normalen“ PCRE-Schnittstelle voraus. Die mit PCRE-Version *10* eingeführte PCRE2-Schnittstelle wird noch nicht unterstützt. Empfohlen wird die **aktuell stabile Version *8.**** der PCRE-Bibliothek. Typischerweise sind die meisten aktuellen Linux-Distributionen schon von Haus aus mit der PCRE-Bibliothek ausgestattet oder erlauben eine problemlose Nachinstallation. Für die Nutzung unter Windows bietet sich neben der eigenen Kompilierung auch der Download einer vorkompilierten Bibliothek, z.B. von **www.pcre.org** oder **www.airesoft.co.uk** an.

Wichtig

Je nach Version ist immer mit einem unterschiedlichem Funktionsumfang und Fehlerstand der PCRE-Bibliothek zu rechnen. Es ist zu beachten, dass die ISA keine Gewährleistung auf die PCRE-Bibliothek und ihre Funktionen geben kann.

Die PCRE-Bibliothek wird normalerweise dynamisch angebunden indem die Funktionen **pcre_compile**, **pcre_study**, **pcre_exec**, **pcre_version** und **pcre_free** gesucht werden. Der IDM übergibt die Strings in einer UTF8-Kodierung weshalb die angebundene PCRE-Bibliothek auch mit UTF8-Support ausgestattet sein sollte.

Folgende Anbindungsarten und damit verbundenen Suchreihenfolgen lässt der IDM zu:

Tabelle 3: *Anbindungsarten und Suchreihenfolgen der PCRE-Bibliothek*

Anbindungsart		Windows	Unix/Linux[1]
E	Funktionssuche direkt im Executable		
A	Applikationsbezogen (relativ zum Pfad der Anwendung)	pcre3.dll dll\pcre3.dll pcre.dll dll\pcre.dll	pcre.(so|sl) lib/pcre.(so|sl) ../lib/pcre.(so|sl)[2]
S	Systemspezifische Library-Suche (z.B. über die Pfadvariablen PATH oder LD_LIBRARY_PATH)	pcre3.dll pcre.dll	pcre.(so|sl)

Bei Erstellung einer Anwendung mit den IDM-Bibliotheken erfolgt der Versuch einer Anbindung in der Reihenfolge E – A – S. Damit ist die einfachste Art, die eigene IDM-Anwendung mit der Unterstützung für Reguläre Ausdrücke auszustatten, das Hinzulegen der dynamischen PCRE-Bibliothek neben dem Executable. Andernfalls wird die im System befindliche Bibliothek angezogen.

Die von der ISA ausgelieferten IDM-Anwendungen zum Entwickeln und Simulieren (IDM, RIDM*, IDMED und Debugger) haben die PCRE-Bibliothek schon statisch eingebaut und benutzen die Anbindungssuche A – E – S, sodass die Nutzung einer externen PCRE-Bibliothek damit ebenso möglich ist.

Falls in der eigenen IDM-Anwendung ebenso eine statische Anbindung erwünscht ist, sollte folgendes beachtet werden: Falls ein Linken der Anwendung ohne Referenzierung der PCRE-Funktionen erfolgt, muss diese zwangsweise komplett angezogen werden (typische Linkoptionen sind z.B. **--whole-archive**, **+forceload** oder **/opt:notref**) und dafür gesorgt werden, dass die PCRE-

[1]Entsprechend der jeweiligen Plattform wird die Endung „.so“ oder „.sl“ für den Dateinamen verwendet.

[2]Die Suche über *../lib/* erfolgt nur wenn die Anwendung in einem Verzeichnis mit dem Namen „bin“ liegt.

Funktionen über die systemspezifische Funktionspointersuche gefunden wird (dies macht unter Umständen ein „Exportieren“ der Funktionen von der Anwendung notwendig). Die ISA empfiehlt aber die Anbindung über eine externe Bibliothek (DLL, Shared Library), sodass für die eigene Produktauslieferung ein Austausch der PCRE-Fassung erleichtert wird.

Die Anbindungsreihenfolge der PCRE-Bibliothek kann vom Anwendungsprogrammierer mit der Schnittstellenfunktion DM_Control bzw. DM_ControlEx über die Aktion *DMF_PCREBinding* gesteuert werden.

11.38 run()

Mit Hilfe dieser Funktion wird ein Dialog gestartet.

Definition

```
void run
(
  object Dialog input
)
```

Parameter

object *Dialog* input

In diesem Parameter wird der Dialog angegeben, der gestartet werden soll.

Beispiel

Laden eines Dialogs und Starten des neu geladenen Dialogs

```
variable object Dialog := null;

Dialog := load("Suchpfad:Test.dlg");
!! pruefen, ob der Dialog geladen werden konnte
if Dialog <> null then
  !! starten des Dialogs
  run (Dialog);
endif
```

Siehe auch

Eingebaute Funktion **load()**

C-Funktion DM_StartDialog im Handbuch „C-Schnittstelle - Funktionen“

11.39 save()

Mit Hilfe dieser Funktion kann ein Objekt abgespeichert werden. Dieses wird nur ausgeführt, wenn der zugehörige Dialog in ASCII-Form geladen worden ist. Wurde der Dialog aus einer Binärdatei geladen, so wird dieser Befehl nicht ausgeführt und die Funktion liefert *false* zurück.

Definition

```
boolean save
(
      object SaveObj  input
  { , string Filename input }
)
```

Parameter

object *SaveObj* input
In diesem Parameter wird das Objekt angegeben, das abgespeichert werden soll.

*string **Filename** input*
In diesem optionalen Parameter wird der Name der Datei angegeben, in der das Objekt und seine Kinder abgespeichert werden. Wird dieser Parameter nicht angegeben, wird das Objekt in der Log-Datei abgespeichert.

Rückgabewert

true
Das Objekt wurde in der angegebenen Datei abgespeichert.
false
Beim Abspeichern ist ein Fehler aufgetreten.

Hinweis

Diese Funktion funktioniert nur in der Entwicklungsversion des ISA Dialog Managers. Bei der Laufzeitversion wird immer *false* zurückgeliefert.

Beispiel

Ein Objekt soll von seinen Attributen her untersucht werden. Dazu wird es in der Log-Datei abgespeichert.

```
save (this);
```

11.40 saveprofile()

Diese Funktion schreibt die aktuellen Werte aller konfigurierbaren ***Record***-Instanzen (*.configurable* = *true*) und **globalen Variablen** (deklariert mit **config**) eines Dialogs oder Moduls in eine Konfigurationsdatei (profile), aus der sie mit der Funktion **loadprofile()** wieder geladen werden können.

Bei ***Records*** werden standardmäßig nur Werte, die nicht geerbt sind, in die Datei geschrieben. Um auch die geerbten Werte in die Datei zu schreiben ist der Parameter *All* auf *true* zu setzen.

Es werden nur Werte aus dem angegebenen Dialog oder Modul gespeichert. Aus anderen Modulen importierte ***Records*** und Variablen werden nicht berücksichtigt.

Definition

```
boolean saveprofile
(
      string  Filename   input
  { , object  Module  := null input }
  { , string  Comment := ""   input }
  { , boolean All :=  false   input }
)
```

Parameter

string *Filename* input

Dieser Parameter definiert den Dateinamen der Konfigurationsdatei. Es kann ein Dateipfad angegeben werden, der auch eine Umgebungsvariable enthalten darf.

*object **Module** := null input*

Dieser optionale Parameter enthält den Identifikator des Dialogs oder Moduls, dessen ***Record***- und Variablenwerte in die Datei geschrieben werden sollen.
Bei *Module* = *null* wird die Modul-ID der aktuellen Regel verwendet.

*string **Comment** := "" input*

In diesem optionalen Parameter kann ein Text angegeben werden, der als Kommentar in die Konfigurationsdatei geschrieben wird.

*boolean **All** := false input*

Wird dieser optionale Parameter auf *true* gesetzt, werden zusätzlich die geerbten Werte in die Konfigurationsdatei geschrieben. Beim Standardwert *false* werden nur nicht geerbte Werte gespeichert.

Rückgabewert

true

Das Speichern der Werte in der Konfigurationsdatei war erfolgreich.

false

Die Werte konnten nicht gespeichert werden.
Dies kann an Fehlern beim Zugriff auf die Datei oder einer ungültigen Modul-ID liegen.

Verfügbarkeit

Ab IDM-Version A.06.02.g

Siehe auch

Eingebaute Funktion **loadprofile()**

C-Funktion DM_SaveProfile()

11.41 second()

Diese Funktion dient dazu, den zweiten Wert eines Index abzufragen.

Definition

```
integer second
(
  index Idx input
)
```

Parameter

index *Idx* input

Dieser Parameter bezeichnet den Index, aus dem der zweite Anteil – in der Regel der Spaltenanteil – extrahiert werden soll.

Rückgabewert

Als Ergebnis dieser Funktion erhält man den zweiten der beiden in dem Index enthaltenen Werte.

Eine typische Anwendung dieser Funktion ist z.B. die Abfrage des zweiten Wertes des Index beim Attribut *.focus* des Tablefield, der die Spaltennummer des Tablefield-Feldes angibt.

Beispiel

Im Dialog wurde ein Tablefield mit vertikaler Dynamikrichtung definiert. Bei einem Mausklick sollen die Zeile und Spalte berechnet werden, die selektiert worden ist.

```
on Table select
{
  variable integer Row;
  variable integer Column;

  Row    := first(thisevent.index);
  Column := second(thisevent.index);
}
```

Siehe auch

Eingebaute Funktion **first()**

11.42 sendevent()

Sie können mit Hilfe der Funktion **sendevent() externe Ereignisse** in Regeln an andere Objekte verschicken. Zur Zuordnung dient der *EventID*-Parameter, welcher mit einer Zahl wie auch mit jedem anderen beliebigen Wert (z.B. einer vorher definierten ***message***-Ressource) belegt werden kann. An dem im *Object*-Parameter angegebenen Objekt muss ein entsprechendes externes Ereignis definiert sein.

Definition

```
void sendevent
(
      object   Object  input,
      anyvalue EventID input
  { , anyvalue Arg1    input }
      ...
  { , anyvalue Arg14   input }
)
```

Parameter

object *Object* input
Objekt, an welches das externe Ereignis verschickt wird.

anyvalue *EventID* input
Eindeutiger Identifikator für das Ereignis.

*anyvalue **Arg1** input*
…
*anyvalue **Arg14** input*
Parameter des externen Ereignisses.

Anmerkungen

Werden im Parameter *EventID* Objekte, beispielsweise message-Ressourcen, genutzt, so sollten diese Objekte keinesfalls **vor** der asynchronen Verarbeitung des Ereignisses zerstört werden. Ansonsten kann das Ereignis nicht mehr zugeordnet werden.

Außerdem ist darauf zu achten, dass das genutzte Ereignis (siehe auch Kapitel „Externe Ereignisse“) maximal 14 Parameter benötigt, da durch die Limitierung auf 16 Parameter in der Regelsprache durch die Funktion **sendevent()** nicht mehr Parameter übergeben werden können (die ersten zwei Parameter sind mit Objekt bzw. Ereignis fest belegt).

Bitte vermeiden Sie jegliche Zyklen in Messages. Dies kann Endlosschleifen verursachen.

Beispiel

```
dialog ExampleDialog
```

```
on dialog start
{
  sendevent(ExampleDialog, 42, "Answer");
}

on dialog extevent 42 (string Question)
{
  print Question;
  exit();
}
```

Siehe auch

Eingebaute Funktion **sendmethod()**

C-Funktionen DM_QueueExtEvent() und DM_SendEvent() im Handbuch „C-Schnittstelle - Funktionen“

11.43 sendmethod()

Mit dieser Funktion wird ein Methodenaufruf in die Ereignis-Warteschlange gestellt und asynchron aus der Ereignisschleife (DM_EventLoop) heraus ausgeführt. Die Funktion ist damit eine einfachere Alternative zum Verschicken eines externen Ereignisses mit **sendevent()** und Aufrufen der Methode in der Ereignisregel für dieses externe Ereignis.

sendmethod() unterstützt maximal *14* Argumente für den Methodenaufruf und kann nicht für Methoden mit `output`-Parametern verwendet werden.

Rückgabewerte von Methoden können nicht verarbeitet werden.

Definition

```
void sendmethod
(
      object   Object input,
      method   Method input
  { , anyvalue Arg1   input
    ...
    , anyvalue Arg14  input }
)
```

Parameter

object *Object* input
Objekt dessen Methode asynchron aufgerufen werden soll.

method *Method* input
Identifikator der aufzurufenden Methode.

*anyvalue **Arg1** input*
…
*anyvalue **Arg14** input*
Argumente die der Methode übergeben werden.

Verfügbarkeit

Ab IDM-Version A.06.02.g

Siehe auch

Eingebaute Funktion **sendevent()**

C-Funktion DM_SendMethod()

11.44 setinherit()

Mit dieser Funktion können Sie Attribute von Objekten auf den Wert der entsprechenden Modelle oder Defaults zurücksetzen.

Die dabei zulässigen Attribute für den jeweiligen Objekttyp entnehmen Sie bitte der „Objektreferenz“.

Definition

```
void setinherit
(
      object    Obj  input,
      attribute Attr input
  { , integer IntIdx input | index IdxIdx input }
  { , boolean SendEvent := true input }
)
```

Parameter

object *Obj* input
Dieser erste Parameter beschreibt das Objekt, dessen Attribut Sie zurücksetzen möchten.

attribute *Attr* input
Dieser zweite Parameter beschreibt das Attribut, das Sie an dem Objekt zurücksetzen möchten.

*integer **IntIdx** input*
*index **IdxIdx** input*
Dieser optionale Parameter kann entweder ein Integerwert oder ein Indexwert sein. Er muss gesetzt werden, wenn ein vektorielles Attribut auf den im Modell hinterlegten Wert zurückgesetzt werden soll. Soll dabei ein eindimensionales Attribut behandelt werden, so muss hier eine Zahl angegeben werden. Soll ein zweidimensionales Attribut zurückgesetzt werden, so muss hier ein Index angegeben werden.

*boolean **SendEvent** := true input*
Über diesen optionalen Parameter wird gesteuert, ob der IDM durch das erfolgreiche Zurücksetzen des Attributes die Regelbearbeitung auslösen soll oder nicht. Der Parameter muss mit *false* belegt sein, wenn keine Events verschickt werden sollen. Sollen Events verschickt werden, muss dieser Parameter mit *true* belegt oder weggelassen sein.

Beispiele

» Zurücksetzen des Inhalts der Zelle *1,1* in einem Tablefield ohne Regelauslösung

```
setinherit (Tablefield, .content, [1,1], false);
```

» Zurücksetzen des Inhalts eines Eingabefeldes mit Regelauslösung

```
setinherit(Edittext, .content);
```

11.45 setvalue()

Mit Hilfe dieser Funktion werden Sie in die Lage versetzt, Attribute von Objekten zu verändern.

Die dabei zulässigen Attribute für den jeweiligen Objekttyp entnehmen Sie bitte der „Objektreferenz“.

Mit **setvalue()** können Sie Werte ändern, ohne dass dadurch ein *changed*-Ereignis ausgelöst wird. Dies ist sinnvoll, wenn mit den Schleifenkonstrukten innerhalb einer Regel sehr viele Werte auf einmal abgeändert werden sollen. Es besteht sonst die Gefahr, dass die interne Setvalue-Ereignisqueue überläuft.

Definition

```
boolean setvalue
(
      object    Obj  input,
      attribute Attr input,
      anyvalue  NewVal input
  { , integer   IntIdx input | index IdxIdx input }
  { , boolean   SendEvent := true input }
)
```

Parameter

object *Obj* input
Dieser Parameter beschreibt das Objekt, dessen Attribut Sie ändern möchten.

attribute *Attr* input
Dieser Parameter beschreibt das Objektattribut, das Sie ändern möchten.

anyvalue *NewVal* input
In diesem Parameter wird der Wert übergeben, den das Attribut annehmen soll.

integer ***IntIdx*** *input*
index ***IdxIdx*** *input*
Dieser optionale Parameter kann entweder ein Integerwert oder ein Indexwert sein. Er muss gesetzt werden, wenn ein vektorielles Attribut gesetzt werden soll. Soll dabei ein eindimensionales Attribut gesetzt werden, so muss hier eine Zahl angegeben werden. Soll ein zweidimensionales Attribut gesetzt werden, so muss hier ein Index angegeben werden.

boolean ***SendEvent*** *:= true input*
Über diesen optionalen Parameter wird gesteuert, ob der Dialog Manager durch das erfolgreiche Setzen des Attributes die Regelbearbeitung auslösen soll oder nicht. Der Parameter muss mit *false* belegt sein, wenn keine Events verschickt werden sollen. Sollen Events verschickt werden, muss dieser Parameter mit *true* belegt oder weggelassen sein.

Rückgabewert

true
Das Setzen des Attributes wurde erfolgreich ausgeführt.

false
Das Attribut konnte nicht gesetzt werden.

Beispiele

» Setzen der Zelle 1,1 in einem Tablefield

```
setvalue(Tablefield, .content, true, [1,1]);
```

» Mit Hilfe dieser Funktion können auch Attribute über Variablen geändert werden, z.B.

```
rule void Set(attribute A)
{
  setvalue(Win1, A, true);
  Pb1.sensitive := getvalue(Win2, A);
}
```

Siehe auch

Methode :set()

C-Funktion DM_SetValue im Handbuch „C-Schnittstelle - Funktionen“

11.46 setvector()

Diese Funktion kann zur Modifikation von vektoriellen Attributen (vordefiniert oder benutzerdefiniert) verwendet werden. Dabei kann das Attribut als Ganzes oder auch nur ein zusammenhängender Teilbereich mit neuen Werten belegt werden.

Definition

```
boolean setvector
(
      object    Object     input,
      attribute Attribute  input,
      vector    NewValue   input
  { , anyvalue  FirstIndex input
  { , anyvalue  LastIndex  input } }
  { , boolean   SendEvent := true input }
)
```

Parameter

object *Object* input
Dieser Parameter definiert das Objekt dessen Attributwerte geändert werden sollen.

attribute *Attribute* input
Dieser Parameter definiert das Attribut, das gesetzt werden soll.

vector *Value* input
In diesem Parameter wird die Werteliste angegeben, die dem vektoriellen Attribut gesetzt wird.

anyvalue ***FirstIndex*** *input*
Dieser optionale Parameter definiert den Startindex ab dem die Attributwerte modifiziert werden. Für eindimensionale Feldattribute sollte hier ein *integer*-Wert angegeben werden, für zweidimensionale Felder ein *index*-Wert. Der Standardwert für eindimensionale Feldattribute ist dabei der *integer*-Wert *1*, für zweidimensionale Felder der *index*-Wert *[1,1]*.

anyvalue ***LastIndex*** *input*
In diesem optionalen Parameter wird der Endindex angegeben bis zu dem einschließlich die Werte im Attribut durch die Werte aus der Werteliste modifiziert werden sollen. Auch hier wird für eindimensionale Feldattribute ein *integer*-Wert erwartet, für zweidimensionale Felder ein *index*-Wert. Der *LastIndex*-Wert sollte hinter dem *FirstIndex*-Wert liegen.
Wird kein *LastIndex*-Parameter angegeben (Standardwert *nothing*), so definiert die Größe des *Value*-Vektors die Anzahl bis zu der das vektorielle Attribute modifiziert wird. Die dahinterliegenden Elementwerte des Attributs werden dann abgeschnitten.

boolean ***SendEvent*** *:= true input*
Über diesen optionalen Parameter wird gesteuert, ob das Setzen des Attributs ein *attribute-changed*-Ereignis auslösen soll. Der Standardwert ist *true*, was eine Verschickung des *changed*-Ereignisses für das Attribut bewirkt. Bei einem Parameterwert von *false* wird kein Ereignis verschickt.

Rückgabewert

true
: Attribut wurde geändert.

false
: Änderung des Attributs konnte nicht ausgeführt werden.

Fehlerverhalten

Der Aufruf der Funktion schlägt fehl bei einem ungültigen Objekt bzw. Attribut, bei einem ungültigen Indexbereich oder wenn das Setzen z.B. wegen einer Diskrepanz zwischen Attributtyp und Wertetyp nicht möglich ist.

Beispiel

```
dialog D

window Wi
{
  .title "Cities";
  .width 300;

  tablefield TfCities
  {
    .xauto 0;  .yauto 0;
    .rowheight[0] 25;  .colwidth[0] 80;
    .rowcount 1;  .colcount 2;
    .rowheader 1;
    .direction 2;
  }

  on close { exit(); }
}

on dialog start
{
  variable hash Capitals := [
    "germany" => "berlin",
    "france"  => "paris",
    "england" => "london" ];

  // fill the title row
  setvector(TfCities, .content, ["Country", "City"]);
  // fill the country and city column
  setvector(TfCities, .field, keys(Capitals), [1,1],
            [itemcount(Capitals),1]);
  setvector(TfCities, .field, values(Capitals), [1,2]);
}
```

Siehe auch

Eingebaute Funktion **getvector()**

C-Funktionen DM_GetVectorValue(), DM_SetVectorValue()

11.47 sort()

Diese Funktion liefert eine sortierte Kopie des angegebenen Listenwertes zurück. Es erfolgt nur eine Sortierung der Werte, nicht der Indizes. Insofern ist auch eine Wertesortierung bei assoziativen Feldern (*hash*-Datentyp) möglich.

Die Sortierung erfolgt gruppiert in den Datentypen der Werteelemente, aufsteigend nach dem Ordnungswert des Datentyps und des Wertes. Bei der Sortierung von Texten bzw. Strings erfolgt immer ein Vergleich im für eine Sortierung besser geeigneten Datentyp *string*.

Die Sortierung kann wie folgt gesteuert werden:

» Mit dem *Reverse*-Parameter kann die Sortierreihenfolge umgedreht werden.

» Über den *SortType*-Parameter kann die Sortierordnung von Strings und Texten beeinflusst werden

Definition

```
anyvalue sort
(
      anyvalue ListValue input
  { , boolean  Reverse  := false       input }
  { , enum     SortType := sort_binary input }
)
```

Parameter

anyvalue *ListValue* input
In diesem Parameter wird der Listenwert angegeben, der sortiert werden soll.

boolean ***Reverse*** *:= false input*
Steht dieser optionale Parameter auf *true*, erfolgt die Sortierung in absteigender Reihenfolge. Der Standardwert ist *false*, was einer aufsteigenden Sortierung entspricht.

enum ***SortType*** *:= sort_binary input*
Dieser optionale Parameter findet nur Anwendung bei Elementwerten, die vom Datentyp *string* sind bzw. zur Sortierung temporär in einen String gewandelt werden.

Wertebereich

sort_binary (Standard)
Sortierung entsprechend dem Unicode-Zeichencode.

sort_linguistic
Sortierung entsprechend der Sprach- und Regionseinstellungen (locale) des Systems. Abhängig von den Systemeinstellungen und der eingestellten Codepage werden Groß- und Kleinschreibung sowie Umlaute berücksichtigt.
Da bei dieser Sortierung Codepage-Konvertierungen notwendig sind, ist sie langsamer als *sort_binary*.
Für Unix bzw. Linux empfehlen wir die Verwendung des UTF8-Locales.

Siehe auch
Dokumentation von locale, LC_CTYPE, LC_COLLATE, strcoll ihres Betriebssystems.

Rückgabewert

Sortierte Sammlung mit demselben Datentyp wie die übergebene Sammlung.

Fehlerverhalten

Der Aufruf der Funktion schlägt fehl, wenn der *ListValue*-Parameter keine Sammlung ist oder ein anderer Parameter einen unerlaubten Wert aufweist.

Beispiel

```
// UTF8
dialog D

on dialog start
{
  variable list List := [ "Äcker", "Bande", "Bäcker", "binden",
                          "Bund", "Aachen", "an" ];
  variable string S;

  foreach S in sort(List, sort_linguistic) do
    print S;
  endfor
  exit();
}

/* liefert: Aachen Äcker an Bäcker Bande binden Bund */
```

Siehe auch

Eingebaute Funktionen **find()**, **keys()**, **values()**

11.48 split()

Mit der Funktion **split()** kann ein String an Trennzeichen in eine Liste von Teilstrings aufgeteilt werden. Sind keine Trennzeichen angegeben, dann wird der String zeichenweise zerlegt.

Definition

```
list split
(
  string Separators input,
  string String     input
)
```

Parameter

string *Separators* input

In diesem Parameter werden die Trennzeichen angegeben, an denen der Parameter *String* aufgeteilt wird. Die Trennzeichen sind in der Ergebnisliste nicht enthalten.
Enthält *Separators* eine Zeichenfolge, wird jedes einzelne Zeichen daraus als Trennzeichen verwendet. Folgen in *String* zwei Trennzeichen direkt aufeinander, wird an dieser Stelle ein Leerstring in die Ergebnisliste aufgenommen.
Ist *Separators* ein Leerstring, dann wird der Parameter *String* in eine Liste einzelner Zeichen zerlegt.

string *String* input

In diesem Parameter wird der String übergeben, der zerlegt werden soll.

Rückgabewert

Eine Liste von Teilstrings.

Beispiele

» Zerlegung in einzelne Zeichen

```
print split("", "Udo");
```

Ausgabe

```
["U","d","o"]
```

» Zerlegung an verschiedenen Trennzeichen

```
print split(",.;", "1,23,4.5;;6,478-9");
```

Ausgabe

```
["1","23","4","5","","6","478-9"]
```

11.49 sprintf()

Mit dieser Funktion kann ein String gemäß den Vorgaben in einem Formatstring formatiert werden. Sie ist ähnlich zu der in der C-Bibliothek verfügbaren Funktion.

Definition

```
string sprintf
(
      string   Format     input,
      anyvalue Argument1  input
  { , anyvalue Argument2  input }
      ...
  { , anyvalue Argument15 input }
)
```

Parameter

string *Format* input

Format beschreibt die Formatierung des Ausgabestrings. Die einzelnen Formate lehnen sich sehr stark an die Notation in ANSI-C an. Gewisse Einschränkungen sind jedoch nötig. Der IDM kann nicht überprüfen, ob der übergebene Formatstring stimmt. Falsche Angaben in den Argumenten der Funktion führen zu Fehlern.
Die Datentypen des Dialog Managers werden sämtlich über „%s“ (plus eventuelle Formatierungen) ausgegeben; Ausnahmen bilden die Datentypen *integer* und *pointer*, diese werden über numerische Formatierungstypen ausgegeben. Die möglichen Werte für den Dialog Manager werden im Kapitel „Syntax des Formatstrings“ erklärt.

anyvalue *Argument1* input

anyvalue ***Argument2*** *input*

…

anyvalue ***Argument15*** *input*

Diese (optionalen) Parameter müssen entsprechend dem Formatstring angegeben werden. Sie werden für das korrespondierende Formatzeichen in den Ausgabestring eingesetzt.

Rückgabewert

Der Rückgabewert ist ein String entsprechend dem Eingabeformat *Format*, bei Fehlern wird ein Leerstring zurückgegeben.

Syntax des Formatstrings

Ein Format wird mit einem „%“ (Prozentzeichen) eingeleitet. Der folgende Ausdruck beschreibt die Syntax eines Formates des IDM **sprintf()**:

```
%[argument$] [flags] [width] [.precision] type
%[argument$] [flags] [width] [.precision] ([singular],[plural])
```

Die einzelnen Elemente der Ausdrücke werden wie folgt definiert:

argument

Dieser Parameter dient der Steuerung der Reihenfolge der angegebenen Parameter. Im Normalfall werden die angegebenen Argumente der Reihe nach genommen. Mit Hilfe dieser Option kann statt dem nächsten Argument ein beliebiges Argument, das über seine Position (Nummer) angesprochen wird, genommen werden. Hierbei ist die Mehrfachverwendung von Parametern möglich.

Beispiel

```
text DateFormat "Date"
{
  0: "%02d.%02d.%04d";        // German
  1: "%2$02d/%1$02d/%3$04d";  // British
}
...
sprintf(DateFormat,7,9,1965);
// variant 0:"07.09.1965"
// variant 1:"09/07/1965"
```

flags

Der IDM unterstützt folgende Flags: +, -, *0*, #

Über die Flags + und - kann definiert werden, ob der String links- (-) oder rechtsbündig (+) formatiert werden soll, d.h. ob Leerzeichen, die auf Grund der Breite notwendig sind, am Anfang oder Ende des Strings eingefügt werden sollen.

Zusätzlich kann für die Ausgabe von Zahlen hier eine *0* angegeben werden, dass die Strings mit Nullen aufgefüllt werden. Wird hier ein # angegeben, ergibt dies beim hexadezimalen Zahlenformat *0x* bzw. *0X*.

width

Mit Hilfe dieses Parameters wird gesteuert, wie viele Zeichen minimal ausgegeben werden sollen. Wenn dabei weniger Zeichen als hier angegeben ausgegeben werden, so werden am Anfang oder Ende des Strings, gesteuert über den *flag*-Parameter, Leerzeichen eingefügt. Ist der String länger als die hier angegebene Feldlänge, werden alle Zeichen ausgegeben.

Zusätzlich kann dieser Parameter auch aus einem Stern (*) bestehen. Dies bedeutet, dass das folgende Argument aus der Argumentliste benutzt wird, um die Ausgabegröße zu definieren.

Beispiele

`sprintf("%0*x",4,255)` wird interpretiert als `sprintf("%04x",255)`

`sprintf("%s %*s","Hallo",40,"Welt")` wird interpretiert als sprintf("%s %40s","Hallo","Welt")

.precision

Über diesen Parameter kann gesteuert werden, in welcher Länge ein String ausgegeben werden soll. Diese Angabe wird immer mit einem Punkt „.“ eingeleitet und führt anders als beim Parameter

width zu einem Abschneiden des Strings.

Beispiele

```
sprintf("%.5s", "123456789")  // Ergebnis "12345"
sprintf("%.*s", 2, "1234")    // Ergebnis "12"
```

type

Über den Typ wird definiert, wie das entsprechende Argument betrachtet werden soll. Dabei gibt es im IDM folgende Möglichkeiten

Typ	Ausgabeformat
d	Zahl mit Vorzeichen.
u	Zahl ohne Vorzeichen
b	Zahl im Binärformat
o	Zahl im Oktalformat
x	Zahl im Hexadezimalformat, wobei „abcdef“ benutzt wird.
X	Zahl im Hexadezimalformat, wobei „ABCDEF“ benutzt wird.
c	Ausgabe eines Zeichens
s	Ausgabe eines Strings

Für den Einsatz im IDM wurden die Formate für Strings (*%s*) und Zahlen (*%d*, *%u*, *%x*, *%o*) erweitert. Dieses Verhalten wird in der nachfolgenden Tabelle dargestellt. Dabei wird *[numerisch]* als Ersatz für *d*, *x*, *X*, *b* oder *u* verwendet.

Datentyp	Steuerung über %	Ergebnis
void	%s %[numerisch]	Leerstring ERROR
string	%s sonst	Eingabestring ERROR
text	%s %[numerisch	zugehöriger String Textid im Zahlenformat
object	%s %[numerisch	Name des Objekts DM_ID im Zahlenformat

Datentyp	Steuerung über %	Ergebnis
pointer	%s %[numerisch	ERROR Adresse des Pointers im Zahlenformat
event	%s %[numerisch	Name des Ereignisses Nummer des Ereignisses im Zahlenformat
enum	%s %[numerisch	Name der Enumeration Nummer der Enumeration im Zahlenformat
attribute	%s %[numerisch	%Name des Attributes Nummer des Attributes im Zahlenformat
method	%s %[numerisch	Name der Methode Nummer der Methode im Zahlenformat
integer	%s %[numerisch	ERROR Zahl im Zahlenformat
index	%s %[numerisch	Beide Werte des Index ERROR
boolean	%s %[numerisch	Ausgabe von *true* oder *false* als String Ausgabe von *0* (*false*) oder *1* im Zahlenformat
class	%s %[numerisch	Name der Klasse Interne Nummer der Klasse im Zahlenformat
datatype	%s %[numerisch	Datentyp als String Datentyp im Zahlenformat

Pluralism

Über diesen Parameter kann eine von dem Wert eines Arguments abhängige gesteuerte Ausgabe erfolgen. Dabei kann zwischen dem angegebenen Singular- oder Pluralwert ausgewählt werden. Diese Werte müssen dabei immer in eckigen Klammern angegeben und durch ein „|“ getrennt werden. Welcher Wert angezeigt wird, entscheidet der angegebene Wert, der für diesen Teil des Strings im Parameterbereich angegeben wird.

Dabei bewirkt die Angabe von

1
: Singularwert

<> 1
: Pluralwert

Beispiel

```
sprintf("%d %[Haus|Häuser]", 1);  // "1 Haus"
sprintf("%d %[Haus|Häuser]", 2);  // "2 Häuser"
sprintf("%d %[Haus|Häuser]", 0);  // "0 Häuser"
```

Zusammenfassende Beispiele

```
!! Ausgabe einer hexadezimalen Zahl in einen statischen Text.

dialog HexOut

window W
{
  child statictext HexOutput { }
}

on dialog start
{
  variable integer Answer := 42;
  variable string S;
  S := sprintf("Dies ist eine Hexzahl: %X", Answer);
  W.HexOutput.text := S;
}
```

```
!! Ausgabe eines Datums in einen statischen Text.
!! Dabei werden Varianten verwendet, um Länderspezifika zu unterstuetzen
!! (Reihenfolge von Tag, Monat und Jahr).
dialog VariantDate
text DateFormat "DateFormat"
{
  0: "%02d.%02d.%04d";
  1: "%2$02d/%1$02d/3$04d";
}

window W
{
  child statictext Date{ }
}

on dialog start
{
  variable integer Day := 7;
  variable integer Month := 9;
  variable integer Year := 1965;
  variable string S;
  S := sprintf(DateFormat,Day,Month,Year);
  !! language variant 0: S = "07.09.1965"
```

```
  !! language variant 1: S = "09/07/1965"
  W.Date.text := S;
}
```

```
!! Beispiel für das Ausnutzen von Singular und Pluralwerten.
dialog SingularPlural

on dialog start
{
  sprintf("%d %[Haus|Häuser]", 1);  // "1 Haus"
  sprintf("%d %[Haus|Häuser]", 2);  // "2 Häuser"
  sprintf("%d %[Haus|Häuser]", 0);  // "0 Häuser"

  print sprintf("%3d Datei%[|en] kopiert", 1);
  // Ergebnis:  1 Datei kopiert
  print sprintf("%3d Datei%[|en] kopiert", 34);
  // Ergebnis: 34 Dateien kopiert
  print sprintf("%3d Datei%[|en] kopiert", 0);
  // Ergebnis:  0 Dateien kopiert
}
```

```
!! Dieser Dialog zeigt Spezifika von sprintf() im IDM.
dialog DMInterna

window W1 { }

on dialog start
{
  print sprintf("Attribut %s hat Nummer %1$d", W1.visible);
  // Ergebnis: Attribut .visible hat Nummer 1

  print sprintf("Fenster %s hat DM_ID %d", W1, W1);
  // Ergebnis: Fenster W1 hat DM_ID 5

  print sprintf("Index %s", [1,2]);
  // Ergebnis: Index [1,2]
}
```

11.50 stop()

Mit Hilfe dieser Funktion wird ein Dialog beendet. War dies der letzte laufende Dialog, wird die Ereignis-Schleife verlassen. Ist kein Dialog angegeben, wird der aktuelle Dialog beendet. Läuft nur ein einziger Dialog, hat **stop()** die gleiche Wirkung wie **exit()**.

Definition

```
void stop
(
      object  Dialog  input
  { , boolean Destroy := false input }
)
```

Parameter

object *Dialog* input
In diesem Parameter wird der zu stoppende Dialog angegeben.

*boolean **Destroy** := false input*
Hat dieser optionale Parameter den Wert *true*, wird der Dialog nach Ausführung seiner *finish*-Regeln gelöscht.

Siehe auch

Eingebaute Funktion **exit()**

C-Funktion DM_StopDialog im Handbuch „C-Schnittstelle - Funktionen“

11.51 strcmp()

Mit Hilfe dieser Funktion können zwei Strings auf ihre Gleichheit überprüft werden. Dabei kann über die Parameter gesteuert werden, wie viele Zeichen maximal verglichen werden sollen und ob die Groß-/Kleinschreibung unterschieden werden soll oder nicht.

Definition

```
integer strcmp
(
      string  String1 input,
      string  String2 input
  { , integer Length  input }
  { , boolean IgnoreCase := false input }
)
```

Parameter

string *String1* input
In diesem Parameter wird der erste der zu vergleichenden Strings angegeben.

string *String2* input
In diesem Parameter wird der zweite der zu vergleichenden Strings angegeben.

*integer **Length** input*
Wird dieser optionale Parameter angegeben, wird definiert, wie viele Zeichen maximal in den beiden Strings verglichen werden sollen.

*boolean **IgnoreCase** := false input*
Wird dieser optionale Parameter mit dem Wert *true* angegeben, wird bei dem Vergleich der Strings keine Unterscheidung zwischen Groß- und Kleinschreibung gemacht.

Rückgabewert

Das Ergebnis der Funktion kann folgende Werte annehmen (immer unter den angegebenen Bedingungen wie Vergleichslänge oder Beachtung der Groß-/Kleinschreibung):

-1
String1 kommt lexikalisch vor *String2*

0
beide Strings sind identisch

1
String1 kommt lexikalisch nach *String2*

Beispiele

» Vergleich zweier Strings bis zur Länge *4*

```
strcmp ("ABCDefgh", "ABCDfghijk", 4);
// Ergebnis: 0
```

» Vergleich ohne Beachtung von Groß-/Kleinschreibung

```
strcmp("ABCDEF", "abcdef", true);
// Ergebnis: 0
```

11.52 stringpos()

Mit Hilfe dieser Funktion kann in einem String nach dem Vorkommen eines anderen Strings gesucht werden.

Definition

```
integer stringpos
(
      string  String   input,
      string  Pattern  input
  { , integer StartIdx := 1 input }
)
```

Parameter

string *String* input

In diesem Parameter wird der String angegeben, in dem nach einem gegebenen Muster gesucht werden soll.

string *Pattern* input

In diesem Parameter wird der gesuchte String angegeben.

*integer **StartIdx** := 1 input*

In diesem optionalen Parameter wird angegeben, ab welcher Stelle in dem angegebenen *String* nach dem *Pattern* gesucht werden soll. Fehlt dieser Parameter, wird vom Anfang des Parameters *String* an nach dem gesuchten *Pattern* gesucht.

Rückgabewert

0

Der gesuchte String kommt in diesem String nicht vor.

> 0

Der gesuchte String kommt in diesem String vor und der Startindex wird als Ergebnis zurückgeliefert.

Falls der gesuchte String mehrmals auftritt, wird immer die Position des ersten Auftretens zurückgegeben.

Wenn der gesuchte String ein Leerstring ist, wird der Wert *1* zurückgegeben.

Beispiel

In einem Eingabefeld soll nach dem Pattern „EUR“ gesucht werden.

```
on Edittext charinput
{
  variable integer Idx;

  Idx := stringpos(this.content, "EUR");
  if Idx > 0 then
```

```
    print "Gefunden";
  endif
}
```

11.53 strreplace()

Die Funktion **strreplace()** ersetzt Teilstrings in einem String durch einen oder mehrere Ersatzstrings. Die zu ersetzenden Teile können entweder in Form einer Position mit einer optionalen Längenangabe, einer Zeichenfolge oder einer Liste von Zeichenfolgen definiert werden.

Definition

In der ersten Form wird im Parameter *Index* die Position des Teilstrings übergeben, der innerhalb von *String* durch den Ersatzstring *Replace* ersetzt werden soll. Der optionale Parameter *Length* definiert, wie viele Zeichen ersetzt werden.

```
string strreplace
(
      string  String  input,
      integer Index   input,
      string  Replace input
  { , integer Length  input }
)
```

Bei der zweiten Form wird in *String* jedes Vorkommen von *Match* durch den Ersatzstring *Replace* ersetzt. Der optionale Parameter *IgnoreCase* bestimmt, ob beim Vergleich die Groß- und Kleinschreibung beachtet wird.

```
string strreplace
(
      string  String  input,
      string  Match   input,
      string  Replace input
  { , boolean IgnoreCase := false input }
)
```

In der dritten Form von **strreplace()** werden in *String* alle Teilstrings aus der Sammlung *MatchList* durch den einzelnen Ersatzstring in *ReplaceListOrString* bzw. den entsprechenden Ersatzstring in der Liste *ReplaceListOrString* ersetzt. Der optionale Parameter *IgnoreCase* bestimmt, ob beim Vergleich die Groß- und Kleinschreibung beachtet wird.

```
string strreplace
(
      string   String    input,
      anyvalue MatchList input,
      anyvalue ReplaceListOrString input
  { , boolean  IgnoreCase := false input }
)
```

Parameter

string *String* input

In diesem Parameter wird der String übergeben in dem Teilstrings ersetzt werden sollen.

integer *Index* input

Dieser Parameter bestimmt, an welcher Position innerhalb von *String* der Teilstring beginnt, der ersetzt werden soll. Ist *Index* größer als die Länge von *String*, wird der Ersatzstring an *String* angehängt. *Index* < *1* wird wie *Index* = *1* behandelt.

string *Match* input

In diesem Parameter wird der Teilstring übergeben, der innerhalb von *String* ersetzt werden soll. Wenn *Match* ein Leerstring ist, erfolgt keine Ersetzung.

anyvalue *MatchList* input

In diesem Parameter wird eine Liste von Teilstrings übergeben, die innerhalb von *String* ersetzt werden sollen. *MatchList* kann eine beliebige Sammlung sein, deren Werte jedoch alle den Datentyp *string* oder ***text*** haben müssen.

string *Replace* input

In diesem Parameter wird der Ersatzstring übergeben, durch den die Teilstrings in *String* ersetzt werden. Die Teilstrings werden in der Reihenfolge ihres Auftretens innerhalb von *String* ersetzt. Es erfolgt keine rekursive Ersetzung.

anyvalue *ReplaceListOrString* input

Dieser Parameter enthält einen einzelnen Ersatzstring oder eine Liste von Ersatzstrings. Die Liste kann eine beliebige Sammlung sein, deren Werte jedoch alle den Datentyp *string* oder ***text*** haben müssen und deren Länge mit der Länge von *MatchList* übereinstimmen muss.
Enthält *ReplaceListOrString* eine Sammlung, wird jedes Vorkommen des Teilstrings *MatchList[I]* durch den Ersatzstring *ReplaceListOrString[I]* ersetzt. Die Reihenfolge der Ersetzung wird durch die Reihenfolge der Teilstrings in *MatchList* bestimmt, wobei die einzelnen Teilstrings in der Reihenfolge ihres Auftretens innerhalb von *String* ersetzt werden.
Es erfolgt keine rekursive Ersetzung, das heißt jeder Teilbereich von *String* wird maximal einmal ersetzt.

*integer **Length** input*

Dieser optionale Parameter definiert die Anzahl der Zeichen, die durch den Ersatzstring ersetzt werden. *Length* kann nur in Verbindung mit *Index* verwendet werden. Wenn *Length* nicht angegeben ist, dann ersetzt *Replace* den Teilstring ab *Index* bis zum Ende von *String*. Bei *Length* <= *0* wird der Ersatzstring eingefügt ohne Zeichen zu ersetzen.

*boolean **IgnoreCase** := false input*

Dieser optionale Parameter steuert, ob beim Vergleich der Teilstrings die Groß- und Kleinschreibung berücksichtigt wird.

IgnoreCase	Bedeutung
false (Standardwert)	Beim Vergleich der Teilstrings wird die Groß- und Kleinschreibung berücksichtigt und muss übereinstimmen.
true	Die Groß- und Kleinschreibung wird beim Vergleich der Teilstrings ignoriert und muss nicht übereinstimmen.

Rückgabewert

String in dem Teilstrings durch Ersatzstrings ersetzt wurden.

Beispiele

» Mehrfaches Ersetzen des Buchstaben *"l"* durch den Buchstaben *"L"*

```
print strreplace("Hallo Welt", "l", "L");
```

Ausgabe

```
"HaLLo WeLt"
```

» Ersetzen von mehreren Teilstrings

```
print strreplace("Otto", ["Ot", "t"], ["U", "d"]);
```

Ausgabe

```
"Udo"
```

» Ersetzen von mehreren Teilstrings
Jeder Teilbereich des Strings *"abc"*wird höchstens einmal ersetzt, also *"a"* durch *"b"*, *"b"* durch *"c"* und *"c"* durch *"x"*. Es wird **nicht** rekursiv, also nicht *"a"* durch *"b"* durch *"c"* durch *"x"*, ersetzt.

```
print strreplace("abc", ["a", "b", "c"], ["b", "c", "x"]);
```

Ausgabe

```
"bcx"
```

» Ersetzen von mehreren Teilstrings durch das Zeichen *"."* ohne Berücksichtigung der Groß- und Kleinschreibung

```
print strreplace("Abrakadabra Rizinus", ["A", "r", "IZI", "u"], ".",
true);
```

Ausgabe

```
".b..k.d.b.. ..n.s"
```

» Ersetzen eines Teilbereichs durch einen anderen String

```
print strreplace("Bamjok", 3, "ngk", 2);
```

Ausgabe

```
"Bangkok"
```

» Einfügen am Anfang eines Strings

```
print strreplace("Eins", 0, "Null ", 0);
```

Ausgabe

```
"Null Eins"
```

» Anhängen am Ende eines Strings

```
print strreplace("Eins", 5, " Zwei");
```

Ausgabe

```
"Eins Zwei"
```

» Anhängen außerhalb der String-Länge

```
print strreplace("Eins", 100, "...");
```

Ausgabe

```
"Eins..."
```

11.54 substring()

Mit Hilfe dieser Funktion können Teile aus Strings extrahiert werden.

Definition

```
string substring
(
      string  String input,
      integer Index  input
  { , integer Length input }
)
```

Parameter

string *String* input

In diesem Parameter wird der String übergeben, aus dem ein Teilstring herauskopiert werden soll.

integer *Index* input

In diesem Parameter wird die Stelle angegeben, ab der aus dem String ein Teilstring herauskopiert werden soll.

Hinweis

Dieser Parameter muss kleiner als Länge des Parameters *string* und größer *0* sein – ansonsten wird die Funktion nicht ausgeführt.

*integer **Length** input*

Über diesen optionalen Parameter wird gesteuert, wie viele Zeichen maximal aus dem String herauskopiert werden sollen. Ist dieser Parameter nicht angegeben, wird ab der angegebenen Stelle bis zum Ende des Strings alles kopiert.

Rückgabewert

""

Der leere String wird zurückgegeben, falls die Bereiche falsch angegeben sind.

kopierter String

Der kopierte String wird als Ergebnis geliefert.

Beispiele

» Kopieren ab der Stelle 4 aus einem String:

```
print substring("123456789", 4);
// Ausgabe: "456789"
```

» Kopieren ab der Stelle 4 in der Länge 3 aus einem String:

```
print substring("12345", 4, 3);
// Ausgabe: "45"
```

11.55 tolower()

Mit Hilfe dieser Funktion kann ein String komplett in Kleinbuchstaben umgewandelt werden.

Diese Funktion basiert auf den Funktionen des jeweiligen Betriebssystems. Daher kann es vorkommen, dass Umlaute nicht richtig behandelt werden.

Definition

```
string tolower
(
  string String input
)
```

Parameter

string *String* input

In diesem Parameter wird der String übergeben, in dem alle Buchstaben in Kleinbuchstaben umgewandelt werden sollen.

Rückgabewert

Das Ergebnis dieser Funktion ist ein String, in dem alle enthaltenen Buchstaben in Kleinschreibweise enthalten sind. Die anderen Zeichen sind unverändert.

Beispiel

```
tolower("AbCdEf");
// Ergebnis: "abcdef"
```

11.56 toupper()

Mit Hilfe dieser Funktion kann ein String komplett in Großbuchstaben umgewandelt werden.

Diese Funktion basiert auf den Funktionen des jeweiligen Betriebssystems. Daher kann es vorkommen, dass Umlaute nicht richtig behandelt werden.

Definition

```
string toupper
(
  string String input
)
```

Parameter

string *String* input

In diesem Parameter wird der String übergeben, der in Großbuchstaben umgewandelt werden soll.

Rückgabewert

Das Ergebnis dieser Funktion ist ein String, in dem alle enthaltenen Buchstaben in Großschreibweise enthalten sind. Die anderen Zeichen sind unverändert.

Beispiel

```
toupper("AbCdef1");
// Ergebnis: "ABCDEF1"
```

11.57 trace()

Diese Funktion dient während der Testphase der Dialogentwicklung zur Ausgabe von Werten in ein Tracefile.

Hinweis

Beim Aufruf dieser Funktion können die Klammern weggelassen werden.

Definition

```
void trace
(
  anyvalue TraceVal input
)
```

Parameter

anyvalue *TraceVal* input
In diesem Parameter wird der auszugebende Wert angegeben.

Beispiel

Ausgabe des aktuellen Objektes in einer Regel:

```
trace this;
```

11.58 trimstr()

Mit Hilfe dieser Funktion können Leerzeichen am Anfang und/oder am Ende eines Strings entfernt werden.

Definition

```
string trimstr
(
  string  String input,
  boolean Start  input,
  boolean End    input
)
```

Parameter

string *String* input

In diesem Parameter wird der String übergeben, bei dem am Anfang und/oder Ende die Leerzeichen entfernt werden sollen.

boolean *Start* input

Wird dieser Parameter mit *true* belegt, werden alle führenden Leerzeichen entfernt, ansonsten bleiben führende Leerzeichen erhalten.

boolean *End* input

Wird dieser Parameter mit *true* belegt, werden alle Leerzeichen am Ende des Strings entfernt, ansonsten bleiben Leerzeichen am Stringende erhalten.

Rückgabewert

Als Ergebnis dieser Funktion wird ein String geliefert, in dem wie angegeben die Leerzeichen entfernt worden sind.

Beispiele

Entfernen von Leerzeichen aus einem String

»
```
trimstr("      123     ", true, true);
// Ergebnis: "123"
```

»
```
trimstr("      123     ", false, true);
// Ergebnis: "      123"
```

11.59 typeof()

Mit Hilfe dieser Funktion kann der Datentyp eines beliebigen Ausdrucks ermittelt bzw. bestimmt werden.

Definition

```
datatype typeof
(
  anyvalue ReqVal input
)
```

Parameter

anyvalue ReqVal input
: In diesem Parameter wird der Wert angegeben, dessen Datentyp bestimmt werden soll.

Rückgabewert

Als Rückgabewert dieser Funktion können alle vom Dialog Manager definierten Datentypen auftreten.

Beispiele

» Datentyp des Attributs *.xleft*

```
typeof(this.xleft);
// Ergebnis: integer
```

» Datentyp des Attributs *.bgc*

```
typeof(this.bgc);
// Ergebnis: color
```

11.60 updatescreen()

Mit Hilfe dieser Funktion können Sie sich alle internen SetVal-Events auf dem Bildschirm ausgeben lassen.

Diese Funktion sorgt dafür, dass alle internen Änderungen der Attribute auf dem Bildschirm dargestellt werden.

Definition

```
void updatescreen
(
)
```

Beispiel

Nach mehreren Zuweisungen zu einer Listbox soll eine Funktion aufgerufen werden. Zuvor sollen die Werte aber sichtbar werden.

```
variable integer I;

for I := 1 to 1000 do
  Listbox.content[I] := itoa(I);
endfor
updatescreen();
SomeFunction();
```

11.61 valueat()

Diese Funktion liefert den indizierten Wert einer Sammlung an der angegebenen Position zurück. Die erlaubten Positionen gehen von *1 … itemcount()* und ermöglichen somit einen Schleifendurchgang durch alle indizierten Werte ohne den tatsächlichen Index zu kennen.

Die Funktion gibt einen Fehler (`fail`) zurück falls die Position außerhalb des erlaubten Bereichs liegt oder es sich beim *Value*-Parameter nicht um eine Sammlung handelt.

Definition

```
anyvalue valueat
(
  anyvalue Value input,
  integer  Pos   input
)
```

Parameter

anyvalue *Value* input

In diesem Parameter wird die Werteliste angegeben, von welcher der indizierte Wert ermittelt werden soll.

integer *Pos* input

Position, für welche der indizierte Wert ermittelt werden soll.

Rückgabewert

Wert, der sich in der Sammlung an der als Parameter übergebenen Position befindet.

Beispiel

```
dialog D

on dialog start
{
  variable matrix Matrix := [
    [0,0] => "-?-",
    [1,1] => "germany",
    [1,2] => "berlin",
    [2,1] => "france" ];
  variable integer Pos;

  /* print the Matrix elements [1,1] [1,2] ... [2,2] */
  for Pos:=1 to itemcount(Matrix) do
    print sprintf("%s : %s", indexat(Matrix, Pos), valueat(Matrix, Pos));
  endfor
  exit();
}
```

Siehe auch

Eingebaute Funktionen **indexat()**, **values()**

Methode index

C-Funktionen DM_ValueGet(), DM_ValueIndex()

11.62 values()

Diese Funktion liefert eine Liste aller Werte einer Sammlung. Standardwerte fließen dabei nicht ein. Auf einen skalaren Wert angewandt wird dieser als Liste zurückgeliefert.

Definition

```
anyvalue values
(
  anyvalue Value input
)
```

Parameter

anyvalue *Value* input
: In diesem Parameter wird der Wert angegeben auf den die Funktion angewendet wird.

Rückgabewert

Liste mit allen Werten einer Sammlung ohne die Standardwerte.

Bei einem Skalar als Parameter wird eine Liste zurückgegeben, die nur diesen Parameter enthält.

Beispiel

```
dialog D

on dialog start
{
  variable hash DomainHash := [
    ".de" => "germany",
    ".us" => "usa",
    ".fr" => "france",
    ".uk" => "united kingdom" ];
  variable string Country;

  /* print all country names from the DomainHash */
  foreach Country in values(DomainHash) do
    print Country;
  endfor
  exit();
}
```

Ausgabe

```
"germany"
"france"
"united kingdom"
"usa"
```

Siehe auch

Eingebaute Funktionen **keys()**, **valueat()**

Methode index

C-Funktion DM_ValueCount()

12 Formale Syntax von Regeln und Statements

12.1 Operatoren

Operatoren dienen der Formulierung von "Expressions".

- Arithmetische Operatoren dienen der Berechnung von Zahlen (arithmetische Werte).
- Vergleichende ("komparative") Operatoren dienen der Verbindung von arithmetischen und logischen Expressions.
- Logische Operatoren verbinden zwei Expressions, um eine neue Expression zu bilden.

Die Operatoren werden im Folgenden beschrieben:

Arithmetischer Operator	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo Operator

Komparativer Operator	Bedeutung
=	Gleichheit
<>	Ungleichheit
>=	größer gleich
<=	kleiner gleich
<	kleiner
>	größer

Logischer Operator	Bedeutung
and	logisches UND
andthen	logisches UND wird nur so weit ausgewertet, bis das Ergebnis feststeht

Logischer Operator	Bedeutung
or	logisches ODER
orelse	logisches ODER wird nur so weit ausgewertet, bis das Ergebnis feststeht
not	logische Negation

Sonstiger Operator	Bedeutung
()	Prioritätensteuerung, Klammerung
:=	Zuweisung
::=	Zuweisung ohne Ereignisauslösung
-	Vorzeichenangabe

Priorität der Operatoren

```
( )                          hoch
not

*     /     %
+     -
<>    >=    <=    <     >
=

and   andthen
or    orelse
:=    ::=                    niedrig
```

Beispiel

```
variable  boolean  Ergebnis;
:
Ergebnis := Button_1.active = true and Anzahl = 10;
```

Die Variable „Ergebnis“ erhält dann den Wert *true*, wenn „Button_1.active“ gleich *true* ist, und wenn gleichzeitig die Variable „Anzahl“ den Wert *10* enthält, sonst erhält die Variable den Wert *false*.

12.2 Expression

"Expression" ist der Überbegriff für die Namen von Konstanten, Variablen, Bezeichnern (Identifikator), Funktionsaufrufen, Objekten und Objektattributen.

Auch Zahlen und Strings sind Expressions.

Zwischen Expressions müssen Operatoren stehen.

Eine Expression kann folgende Elemente haben:

- String
- Variable
- Funktion
- Objekt
- Integer-Wert
- Boolean-Wert
- Datentyp
- Objektklasse
- Ereignis
- Attribut
- Aufzählungswert
- Index
- Builtin-Funktion
- Expression
- Expression-Attribut
- Expression-Attribut Index
- Expression Operator Expression
- negative Expression
- negierte Boolean-Expression

Diese Elemente werden im folgenden beschrieben:

String

Ein String kann alle Buchstaben des lateinischen Alphabets enthalten sowie die meisten nationalen Spezialzeichen, z.B. mutierte Vokale. Ein String ist normalerweise ein beliebiger Name, der in der Regel verwendet werden soll. Er wird mit doppelten Hochkommata dargestellt.

Beispiel

```
"Cancel"
"Dialog Manager"
```

Variable

Wenn Sie eine Variable einbeziehen wollen genügt es, den Namen der Variable zu spezifizieren

Siehe Kapitel „Globale Variablen (variables)“.

Funktion

Wenn Sie eine Funktion einbeziehen wollen, müssen Sie zuerst den Funktionsnamen und dann die Parameter angeben. Die Parameter sind in Form von Expressions angegeben.

Siehe Kapitel „Aufruf von Anwendungsfunktionen“.

Beispiel

```
Add (a,b);
```

Objekt

Ein Objekt wird durch seinen Objektpfad referenziert. Dabei muss der Name des ersten Objektes eindeutig sein. Die weiteren Namen im Objektpfad beziehen sich auf das Objekt, welches durch den vorangehenden Teil des Objektpfads referenziert wird.

Beispiel

```
OK_Button
Hauptfenster.Groupbox1.ExitButton
```

Integer-Wert

Ein Integer-Wert ist ein Integer zwischen -2^{31} und $+2^{31}$.

Beispiel

```
30
```

Boolean-Wert

Ein Boolean-Wert kann den Wert *true* oder *false* haben.

Datentyp

Folgende Werte sind zulässig:

- anyvalue
- attribute
- boolean
- class
- datatype
- enum
- event
- index
- integer
- object
- string
- void

Objektklasse

Folgende Werte sind zulässig:

- accelerator
- application
- canvas
- checkbox
- control
- color
- cursor
- dialog
- edittext
- font
- function
- groupbox
- image
- import
- listbox
- menubox
- menuitem
- menusep
- messagebox
- module
- notebook
- notpage
- poptext
- pushbutton
- radiobutton
- rectangle
- rule
- spinbox
- scrollbar
- statictext
- tablefield
- treeview
- text
- tile

- timer
- variable
- window

Ereignis

Folgende Werte sind zulässig:

- activate
- changed
- charinput
- cut
- close
- dbselect
- deactivate
- deiconify
- deselect
- deselect_enter
- extevent
- finish
- focus
- help
- hscroll
- iconify
- key
- modified
- open
- paste
- move
- resize
- scroll
- select
- start
- vscroll

Attribut

Die Identifikatoren der in der „Attributreferenz“ beschriebenen Attribute sind hier zulässig.

Beispiel

```
.visible
.sensitive
.xleft
```

Aufzählungswert (Enum)

Folgende Werte sind zulässig:

- noicon, icon_hand, icon_question, icon_exclamation, icon_asterisk,
- icon_information, icon_query, icon_warning, icon_error
- nobutton, button_ok, button_cancel, button_retry, button_abort,
- button_ignore, button_yes, button_no
- sel_row, sel_column, sel_area, sel_header, sel_single
- store_never, store_onfree, store_onchange
- edit_online, edit_offline, edit_locking
- winsys_x11, winsys_windows, winsys_pm, winsys_none
- toolkit_motif, toolkit_isa, toolkit_windows, toolkit_pm, toolkit_alpha
- coltype_bw, coltype_grey, coltype_color
- os_dos, os_nt, os_os2, os_unix, os_vms, os_mpe

Index

Syntax für Index

```
[ Integer-Expression , Integer-Expression ]
```

Beispiel

```
[3, TF1.colheader]
```

Builtin-Funktion

Builtin-Funktionen sind Funktionen, die dem DM als Standard bekannt sind (siehe Kapitel „Eingebaute Funktionen (Builtin-Funktionen)“).

Expression

Eine Expression kann wiederum eine andere Expression enthalten.

Beispiel

```
a+b
```

Expression-Attribut

Eine Expression kann mit einem Attribut verbunden werden. Dabei muss die Expression ein Objekt liefern.

Beispiel

```
P1.visible
```

Expression-Attribut Index

Eine Expression kann mit einem indizierten Attribut verbunden werden. Dabei muss die Expression ein Objekt liefern. Es können ein- und zweidimensionale Indizes angegeben werden.

Beispiel

```
L1.content[17]
T1.active[3,7]
```

Expression Operator Expression

Eine Expression kann mit einer anderen Expression mit einem arithmetischen Operator verbunden werden.

Beispiel

```
B.x := B.x + B.4;
```

Negative Expression

Eine negative Expression kann z.B. ein negativer Integer sein.

Beispiel

```
-7
```

Negierte Expression

Vor der Expression kann das Wort NOT stehen, so dass ein Switch möglich wird.

Beispiel

```
Window.visible := not Window.visible;
```

Die genaue Syntax der verschiedenen Komponenten einer Zuweisung wird im folgenden Backus-Naur-Modus aufgeführt.

Definition

```
<Expression>::=
                <String>|
                <Variable>|
                <Funktion>|
                <Objekt>|
                <Integerwert>|
                <Booleanwert>|
                <Datentyp>|
                <Objektklasse>|
                <Ereignis>|
                <Attribut>|
                <Enum>|
                <Index>|
                <Builtin-Funktion>|
                (<Expression>)|
                <Expression><Attribut>[<Expression>]|
                <Expression><Attribut><Index>|
                <Expression><Operator><Expression>|
                -<Expression>|
                not<Expression>
<Objekt>::=
                <Objektpfad>{{<Relation>}{<Objektpfad>}}
<Builtin-Funktion>::=
                <Builtin>(<Expression>)
<Datentyp>::=
        anyvalue | attribute | boolean | class | datatype | enum | event |
index |
        integer | object | string | void
<Objektklasse>::=
        accelerator | application | ... (vgl. Liste oben)
<Ereignis>::=
        activate | changed | ... (vgl. Liste oben)
<Enum>::=
        noicon | icon_hand | ... (vgl. Liste oben)
<Index>::=
        [<Expression>, <Expression>]
<Operator>::=
                +       |
                -       |
                *       |
                /       |
                %       |
                <       |
                <=      |
                =       |
                <>      |
                >=      |
                >       |
```

```
                and      |
                or
<Objektpfad>::=
                <Objektidentifikator>{.<Objektidentifikator>}        |
                <Variable>
<Objektpfad>::=
                <Objektpfad>        |
                this
<Relation>::=
                .parent        |
                .window        |
                .menubox       |
                .groupbox      |
<String>::=
                "<ASCII>"
<ASCII>::=
                {<Ziffer>|<Buchstabe>|<Leerzeichen>|\<Oktalzahl>|
                <Sonderzeichen>}
```

12.3 Statements

Zur Bildung von Statements werden Operatoren und Expressions verwendet.

```
Statementlist::=  <Statement>{<Statement>}
<Statement>::=
  <Objekt>    :=  <Expression>;|
  <Variable>  :=  <Expression>;|
  if <Expression> then<Statementliste>{else<Statementliste>}
  endif|
  exit             ;|
  print<Expression>  ;|
  <Funktion>         ;
```

Syntax (konditionierter Programmablauf)

Die Syntax eines konditionierten Programmablaufes wird im folgenden aufgelistet:

```
<If_then_else>::=
  if
                <Bedingungszeile>
  then
                <Statementliste>
  [elseif
                <Bedingungszeile>
   then
                <Statementliste>]
  {else
                <Statementliste>}
  endif
```

Beispiele für Statements

```
window.visible  := true;
window2.x       := window1.height+window1.x+10;
```

Metasprachliche Beispiele für die Verwendung von Statements in den Regeldefinitionen

```
on Identifier event
if ( Statement mit booleschem Ergebnis)
{
  Statements
}
on Identifier event
{
  if ( Statement mit booleschem Ergebnis)
  then
    Statements
  endif
}
on Identifier event
{
  if ( Statement mit booleschem Ergebnis )
  then
    Statements
  else
    Statements
  endif
}
```

Anmerkung

Boolesche Ergebnisse werden durch logische Operatoren gebildet.

Funktionsaufrufe

Funktionsaufrufe zur Applikation können auf zwei Arten verwendet werden:

- `function`
  Die Funktion wird **nicht** aufgerufen, es wird das letzte Resultat der Funktion verwendet.
- `function(parameter)`
  Die Funktion wird aufgerufen, das Ergebnis dieses Aufrufs wird verwendet. Optional können der Funktion Parameter übergeben werden.

Diese Funktionen stellen die Schnittstelle zu Ihrer Applikation dar.

Beispiel

```
on Variable_A.value changed
{
```

```
  print Variablenwert;
}
on Mainwindow close

  exit;
}
on Pushbutton select
{
  AP_Funktion (this.text);
}
```

Index

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z