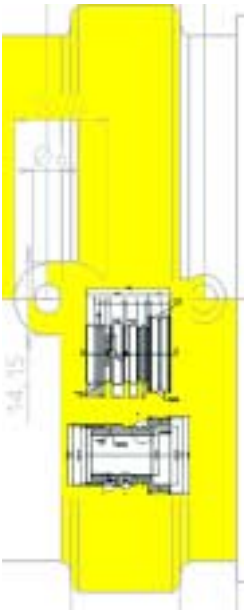
EUKLID Design

euklid
CAD CAM



Offene Architektur AQL

2002

Oktober 2001

So können Sie uns erreichen:

In Deutschland:

EUKLID Software GmbH
Vor dem Lauch 19
D-70567 Stuttgart
Tel: +49 / 711 / 7 22 84-0 · Fax: +49 / 711 / 7 22 84- 293
Internet: http://www.euklid-software.de

In der Schweiz:

EUKLID Software GmbH
St. Gallerstrasse 151
CH-8645 Jona, SG
Tel: +41 / 55 / 21270-43 · Fax: +41 / 55 / 21270-44

Haben Sie Fragen, Anregungen zu ...

... Produkten

... oder einem anderen Thema der
Technischen Informationssysteme?

Auskunft und Informationen erhalten Sie bei unseren Geschäftsstellen.

Inhalt

Inhalt

Inhalt

1 Einleitung

Die Programmiersprache AQL stellt das gesamte Spektrum der im CAD-System ObjectD integrierten Objekte und Aktionen zur Verfügung. Außerdem bietet sie Kontrollfunktionen sowie Steuerungsmöglichkeiten für die Ein- und Ausgabe. Mit Hilfe von AQL-Programmen können Sie den Inhalt von mit ObjectD erstellten Modellen analysieren, erzeugen, verändern u.v.m.

AQL kann auch als Implementierungssprache zum Einbringen neuer Funktionalität genutzt werden.

Aufbau der Handbuchreihe

Die vorliegende Benutzerdokumentation besteht aus folgenden Bänden:

ObjectD, Grundlagen und Anwendung
: Dieses Handbuch enthält einen Überblick über die Funktionalität und die Grundlagen, sowie ein Anwendungsbeispiel, das in die Arbeit mit dem System einführt.

ObjectD, Offene Architektur (AQL) – vorliegender Band
: Diese Dokumentation enthält die Beschreibung der Programmiersprache AQL, die für das Erzeugen, Ändern und Abfragen von Modellen zur Verfügung steht.

Zielgruppe und Verwendungszweck des Handbuches

Ziel dieses Benutzerhandbuchs ist die Beschreibung des Aufbaus, der Erstellung und der Ausführung von AQL-Programmen. Diese Sprachbeschreibung richtet sich an Anwender, die Erfahrung mit dem CAD-System ObjectD haben. Die Kenntnis einer Programmiersprache, speziell C oder Pascal, ist für die Einarbeitung von Vorteil.

Sie können zu diesem Themenkomplex auch Kurse besuchen. Genauere Informationen hierzu können Sie bei der Geschäftsstelle der *strässle GmbH* erhalten.

Einleitung

Aufbau des Handbuchs

Die Sprachbeschreibung ist wie folgt gegliedert:

Das Kapitel „Einführung in AQL“ enthält die grundlegenden Informationen zu den Eigenschaften von AQL, zu Programmaufbau und Programmausführung.

Die Formatierung der Ausgabe, die Variablen und Funktionen, sowie die Dateibearbeitung werden im Kapitel „Programmiersprache AQL“ beschrieben.

Die modellbezogenen Aspekte von AQL sind im Kapitel „AQL als Kommandosprache“ dargestellt. Hier wird erklärt, wie AQL die Erzeugung, Manipulation und Abfrage des CAD-Modells ermöglicht. Außerdem ist der Zugriff auf die Daten von ObjectD erläutert.

Die nachfolgenden Kapitel enthalten Funktionen mit Erläuterungen und Beispielen.

Beispiele zu Datenstrukturen, Dateibearbeitung und Zeichensuche in einer Datei sind in Kapitel „Beispielprogramme“ aufgenommen.

Das Kapitel „Syntaxdiagramme“ bietet eine Übersicht über die Syntax von AQL in Diagrammform.

Erläuterung der typographischen Darstellungsmittel

Zur leichteren Lesbarkeit des Handbuchs dienen folgende Darstellungsmittel:

Handlungsanweisungen sind durch das Zeichen ☞ dargestellt.

Fest definierte Sprachwörter werden bei der Beschreibung durch **Fettdruck** hervorgehoben, z.B. **return**.

Programmcode ist in `Schreibmaschinenschrift` dargestellt.

Teile innerhalb von Programmcode, welche Sie im jeweiligen AQL-Programm durch Ausdrücke ersetzen müssen, werden in <spitze Klammern> eingeschlossen, z.B. **while** `<expression>` **do**.

Funktionsargumente, deren Angabe optional ist, stehen in [eckigen Klammern].

Beispiele zu Erläuterungen sind durch „**Beispiel**“ gekennzeichnet.

Tips zur Arbeitserleichterung oder Ausnahmen sind vom Fließtext durch das nebenstehende Symbol abgesetzt, ebenso wie die Beschreibungen von Sonderfällen z.B. Störungen, Fehlerquellen.

2 Einführung in AQL

In diesem Kapitel wird der Aufbau, die Erstellung und die interaktive Ausführung von AQL-Programmen an einem einfachen Beispiel, das die Zeichenfolge „Hello World!“ ausgibt, gezeigt (Siehe „Programmbeispiel“ auf Seite 2-6.).

2.1 Eigenschaften von AQL

Die Programmiersprache AQL (Attribute Query Language) ist eine precompilierte Interpretersprache mit hoher Sprachmächtigkeit und Systemtiefe. AQL ist objektorientiert. Erweitert um CREATE-, EDIT- und DELETE-Funktionalität ist sie die Prozedursprache von ObjectD.

AQL ermöglicht die spezifische Anpassung der CAD-Applikation. Die Bearbeitung des ObjectD-Modells wird mit den ObjectD-Funktionen vorgenommen.

AQL ist durch Makros, Automatisierung von Gestaltsvarianten, Einbinden von Fremdprogrammen und Animation vielseitig einsetzbar.

Die Sprache entspricht in ihren Grundzügen dem Aufbau anderer bekannter Programmiersprachen. Bei der Gestaltung der AQL-Sprache wurde der **syntaktische Aufbau** durch C angeregt, der Programmaufbau lehnt sich an Pascal an. Als interpretierte Sprachen dienten Lisp und Basic als Vorbilder und SQL als Abfragesprache.

AQL bietet

- alle gängigen **Kontrollstrukturen** wie „if...then...else“ für eine bedingte Verzweigung, „switch...case...“ für Fallunterscheidungen und „for..“ für Schleifen
- alle gängigen **Operatoren** und **Funktionen**, z.B. +, -, /, *, **, min( ), sin( ), cos( ), tan( )
- dynamische **Datentypen** wie „real“, „integer“, „boolean“, „string“, „date“, „group“
- automatische **Variablen**

- eine weitreichende **Modellabfrage** (Query).
- vollständige Einbindung in Bedienoberfläche und Systemarchitektur durch UDAs

Binden und Übersetzen der Programme ist nicht erforderlich.

Im Gegensatz zu den konventionellen Programmiersprachen kann AQL jedoch mit den Objekten des ObjectD-Modells arbeiten, die nicht durch den AQL-Sprachprozessor verwaltet werden. Damit die Unabhängigkeit der Sprache von den Objekten gewährleistet ist, kennt AQL keine Typvereinbarung. Variable und Funktionsargumente werden nur mit ihrem Namen deklariert.

Da das System zur Laufzeit die entsprechenden Fehlermeldungen bei unzulässiger Kombination von Typen ausgibt, ist es nicht notwendig, Typkonflikte syntaktisch zu erfassen.

Verallgemeinert kann AQL als eine Sprache angesehen werden, die nur einen Typ kennt: das **AQL-Symbol**. Dieses Symbol kann ein beliebiges Objekt darstellen. Entweder einen der standardmäßig in AQL integrierten Typen „real“, „integer“, „string“, „date“, „group“ (Gruppe von Symbolen) oder ein Objekt des ObjectD-Modells mit dessen Datentypen.

Der AQL-Interpreter kann als **Instrument zur Abfrage des ObjectD-Modells** und als **Kommandosprache** verwendet werden.

- Die Abfrage stützt sich auf die Eigenschaft, daß alle ObjectD-Objekte benannte Attribute besitzen. Die Attribute können in einem Selektionsausdruck verwendet werden. Mit Hilfe von Selektionsausdrücken werden solche Objekte einer beliebigen Gruppe erfaßt, die z.B. einer bestimmten Bedingung („where“-Klausel) genügen.
- Makros, z.B. die das ObjectD-Modell erweitern, verändern o. ä., können als AQL-Programme implementiert werden.

Beim **Aufruf eines AQL-Programms** wird der lesbare Code zunächst übersetzt und in Form einer Baumstruktur intern abgespeichert. Diese Baumstruktur repräsentiert alle Anweisungen und Ausdrücke des AQL-Programms. Bei der Ausführung wird nur noch auf diese Struktur zugegriffen, das Quellprogramm (source code) bleibt unberücksichtigt.

Dieses Verfahren der Vorübersetzung erhöht die Ausführungsgeschwindigkeit beträchtlich. Der Aufwand für die Vorübersetzung (ca. 700 Statements/Sekunde*mips) kann bei

einer Ausführungsgeschwindigkeit von ca. 3000 Statements/Sekunde*mips vernachlässigt werden.

Um AQL-Programme vor unerwünschten Veränderungen zu schützen, besteht die Möglichkeit, **verschlüsselte AQL-Programme** auszuführen.

2.2 Programmaufbau

Jedes AQL-Programm befindet sich in einer ASCII-Datei und kann mit einem beliebigen Texteditor bearbeitet werden. Es besteht aus den folgenden Teilen:

- Anweisungen (statements)
- Variablenvereinbarungen (declarations)
- Funktionsdefinitionen (function definitions)

Die Reihenfolge dieser Teile ist beliebig. Es ist praktisch und zulässig, diese Teile nacheinander in der Reihenfolge aufzuführen, in der sie verwendet werden. Es besteht nur die eine Bedingung, daß eine Funktion vor ihrer ersten Verwendung definiert wird.

Ein AQL-Programm muß aus mindestens einer Anweisung bestehen. Das Vorhandensein von Variablenvereinbarungen und Funktionsdefinitionen ist optional. Der Inhalt der AQL-Datei wird wie ein Hauptprogramm betrachtet, in dem alle Anweisungen nacheinander abgearbeitet werden.

2.2.1 Trennung von Anweisungen

Die Syntax von AQL ist so aufgebaut, daß auf ein Trennzeichen für Anweisungen (statements) verzichtet werden kann. Um die Lesbarkeit der AQL-Programme zu erhöhen oder um das Erscheinungsbild für C- und Pascal-Programmierer vertrauter zu gestalten, kann der Strichpunkt **;** als Trennzeichen verwendet werden.

2.2.2 Kommentare

Ein Kommentar wird bei der Syntaxanalyse wie eine Anweisung behandelt. Deshalb kann er zwischen beliebige Anweisungen eingeschoben werden. Er orientiert sich an der Sprache C und wird mit dem Kommentaranfang **/*** und dem Kommentarende ***/** begrenzt. Dazwischenliegende Zeichen werden zusammen mit den Kommentarbegrenzern überlesen. Wie in C++ wird auch in AQL die Zeichenfolge **//** als Kommentar bis Zeilenende interpretiert, d. h. alle Zeichen bis zum Ende der Zeile, die auf **//** folgen, werden überlesen. Die Schachtelung von Kommentaren ist nicht zulässig.

2.2.3 Einfügen anderer Dateien – include

Durch die *include*-Anweisung kann an der aktuellen Position der Inhalt einer anderen Datei eingefügt werden. Der Dateiname muß konstant sein, d.h. er darf weder Bezeichner noch Ausdrücke enthalten.

```
include <dateiname>
```

- Bei Vergabe von Dateinamen ist das zugrunde liegende Betriebssystem zu beachten.
- Environmentvariable aus dem Betriebssystem sind in AQL über die Funktion „getenv“ bekannt.
- Dateinamen, die mit **!** beginnen, werden relativ zum aufrufenden AQL-Programm interpretiert.

```
include !my_include
```

Die Datei *my_include* wird im selben Verzeichnis gesucht, in dem auch das AQL-Programm steht.

Die Dateien der mit „include“ eingefügten Programmsegmente müssen zum Ablaufzeitpunkt verfügbar sein.

2.2.4 Verschlüsselung von AQL-Programmen – encrypt

Um AQL-Programme vor unbefugten Veränderungen bzw. unlizenziertem Gebrauch zu schützen, können diese Programme mit der AQL-Funktion *encrypt* verschlüsselt werden.

Verschlüsselte Programme sind ohne AQL-Entwicklungslizenz ablauffähig.

Um die Funktion „encrypt“ zu benutzen benötigen Sie eine AQL-Entwicklerlizenz.

```
encrypt(<ascii_source_name>,
        <crypted_name>,
        <my_company_name>,
        <my_key>)
```

`<ascii_source_name>`	`:string`	Dateiname des zu verschlüsselnden Programmes
`<crypted_name>`	`:string`	Dateiname des verschlüsselten Programmes
`<my_company_name>`	`:string`	Name der Firma, die das AQL-Programm erstellte
`<my_key>`	`:string`	Persönliches Schlüsselwort

Beispiel

```
‘before encrypt‘ nl
encrypt ("my_original.aql",
         "my_encrypted.aql",
         "my_company",
         "my_key")
‘after encrypt‘ nl
```

2.2.5 Verschlüsselte AQL-Programme – key

Verschlüsselte Programme beginnen mit der *key*-Anweisung, gefolgt vom Schlüsselwort. Der Rest des Programms ist nicht lesbar. Wenn ein verschlüsseltes Programm durch den Anwender modifiziert wird, kann es nicht mehr ausgeführt werden. Es wird dann eine Fehlermeldung ausgegeben.

2.2.6 Programmbeispiel

Um die Nachricht „Hello World!“ auszugeben, ist nur eine Anweisung für die Ausgabe konstanter Zeichenketten erforderlich, wie sie weiter unten im Detail erläutert wird.

Beispiel: Hello_World.aql

PROGRAMM:
```
'Hello World!'
```

AUSGABE:
```
Hello World!
```

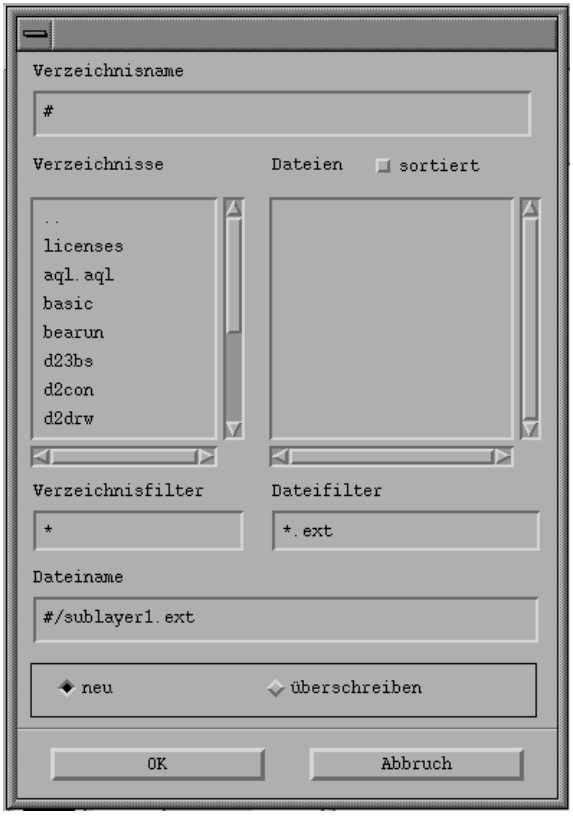
2.3 Programmausführung

Es gibt verschiedene Möglichkeiten, AQL-Programme zu starten:
- interaktiv
- im Batch-Modus
- über UDAs
- über ein anderes AQL-Programm
- über Optionen für spezielle Funktionalität

2.3.1 Interaktive Ausführung

Interaktiv starten können Sie AQL-Programme durch

- Anklicken der AQL-Ikone im zweiten Menü
- Auswahl des AQL-Programms mit dem eingeblendeten Dateiauswahlformular



AQL-Programme, die die Standardein- und -ausgabe verwenden, d.h. Ein- und Ausgabefunktionen ohne vorhergehende *file*-Anweisung, benutzen das Fenster, in dem ObjectD gestartet wurde.

Während der Entwicklungs- und Testzeit ist der Programmname *f1.aql* empfehlenswert. Diese Programme können mit den Funktionstasten <F1> bis <F4> bzw. deren Kombination mit der Taste <SHIFT> gestartet werden.

Wenn Sie die ersten Beispiele nachvollziehen wollen, können Sie die restlichen Ausführungsarten in diesem Kapitel überspringen.

2.3.2 Batch Modus

Soll ObjectD nur ein AQL-Programm ausführen, ohne daß Interaktionen des Anwenders erforderlich sind, so kann es im Batch-Modus gestartet werden:

objectd `-batch <aql-dateiname> <modellname>`

Die Option *-batch* und die darauf folgenden Namen von AQL-Dateien veranlassen ObjectD, die AQL-Programme auszuführen und sich danach zu beenden, ohne ein Grafikfenster zu eröffnen. Damit können z.B. Umsetzer oder automatisches Plotten von Modellen elegant in Shell-Prozeduren ausgeführt werden.

2.3.3 Kopplung von UDAs mit AQL-Programmen

Ein AQL-Programm wird an eine definierte Aktion (UDA) gekoppelt, wenn dies bei der Erstellung der UDA angegeben wurde.

Beispiel: Das UDA *myuda.uda* ist mit dem AQL-Programm *myuda.aql* gekoppelt.

Je nach Angabe verschiedener Attribute werden Funktionsrümpfe vorgeneriert, die vom Entwickler mit Semantik zu füllen sind.

Es sind dies die Funktionen:

pre_action
: Ablauf vor Durchlauf von „execute“ und Parameterfunktionen

post_action
: Ablauf nach Durchlauf von „execute“ und Parameterfunktionen

execute
: eigentliche Evaluierungsfunktion

pre_par_<parname>
: Ablauf vor Parametereingabe

post_par_<parname>
: Ablauf nach Parametereingabe

select_par_<parname>
: Selektorfunktion für Parameter

traverse_values_par_<parname>
: liefert die möglichen Zustände der Selektoren

draft_par_<parname>
: liefert skizzierten Wert für Parameter

move_cursor_par_<parname>
: Cursorroutine bei Mausbewegung ohne Tastendruck

drag_cursor_par_<parname>
: Cursorroutine bei Mausbewegung mit gedrückter linker Maustaste

show_par_<parname>
: Darstellungsmethode für Parameterhilfsdarstellung z.B. Länge

init_par_<parname>
: Eintragen einer Voreinstellung für Parameter

get_prompt_par_<parname>
: Parametereigenes Prompting

check_par_<parname>
: Intervalltest für Parameterwerte

suppress_property
: Funktion zum Unterdrücken von Properties in Resultat und Effekten

res_fill_suppress_props
: füllen von Standardwerten für suppressed Resultatproperties

2.3.4 Ausführung von AQL-Programmen aus

Ein AQL-Programm wird von anderen AQL-Programmen aus mit der AQL-Funktion *aql*, die den Namen des AQL-Programms als erstes Argument erhält, aufgerufen:
`aql (<dateiname>)`

Weitere Argumente sind möglich.

Beispiel:
```
aql("!my_subprog", "Arg1", "Arg2")
```

Die Funktion liefert folgende Werte als Rückgabewert:

```
   0   keine Fehler
  -1   Laufzeitfehler
  -2   Syntaxfehler
  -3   Die angegebene Datei mit dem AQL-Programm kann nicht
       geöffnet werden.
   1   Fehler in externen Funktionen
```

- Bei Vergabe von Dateinamen ist das zugrundeliegende Betriebssystem zu beachten.
- Environmentvariable aus dem Betriebssystem sind in AQL über die Funktion „getenv“ bekannt.
- Dateinamen, die mit **!** beginnen, werden relativ zum aufrufenden AQL-Programm interpretiert.

Sollen übergebene Argumente von der gerufenen Prozedur übernommen werden, gibt es dafür zwei vordefinierte Symbole:

`aql_argument_list`	Gruppe der Parameterstrings
`aql_arg_<n>`	n-tes Argument der Parameterleiste; aql_arg_Ø enthält den Dateinamen.

Beispiel 1
```
j=1
for i in aql_argument_list do
       ‘Argument‘[j]‘;‘[i] nl
        j=j+1
end
```

Beispiel 2
```
‘Unterprogramm‘[aql_arg_Ø]‘gestartet!‘ nl
```

2.3.5 Sonderfunktionen im Zusammenhang mit Optionen

Für die AQL-Programmausführung sind folgende Optionen beim Start von ObjectD wichtig:

`-access <aqlnam>`	Vor und nach jedem Speicher- und Ladevorgang sowie bei Aufruf neuer Modelle wird je eine AQL-Routine (siehe auch mitgeliefertes Beispiel im Installationsverzeichnis #/aql.aql/file_aql.aql) aufgerufen. Diese wird nacheinander in folgenden Verzeichnissen gesucht:
	
	im Arbeitsverzeichnis
	`~/user_data`
	`#/aql.aql`

`-start <aqlname>`	erst AQL-Datei, dann interaktiv
`-stop <aqlname>`	Ablauf vor Beenden der Sitzung

ObjectD kann **AQL-Protokolle** mitschreiben. Sie dienen als gute Programmiererstützen, wenn bestimmte AQL-Funktionen unklar sind.

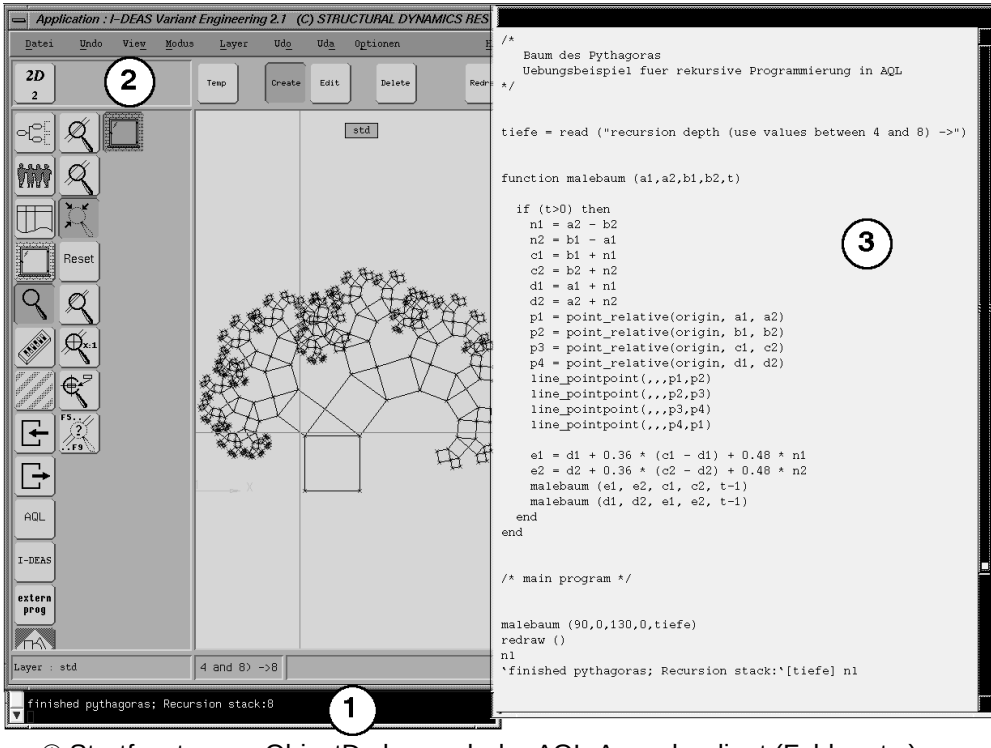
AQL-Protokolle werden bei folgender Start-Option mitgeschnitten:

`-protocol`	Protokollmitschnitt	Mitschnitt der Modell-Logik, d.h. Objektidentifikation über Name

Die Mitschnitte unterliegen keiner Ablaufgewähr, können aber Datenstrukturzusammenhänge gut widerspiegeln und sind damit eine Hilfe bei der AQL-Programmierung. Die Mitschnitte werden im Arbeitsverzeichnis abgelegt.

2.3.6 AQL-Programmierumgebung

Das Bild zeigt den Bildschirm einer typischen AQL-Programmierumgebung.



① Startfenster von ObjectD, das auch der AQL-Ausgabe dient (Fehler etc.)
② Ablauffenster von ObjectD
③ Editorfenster mit dem gerade bearbeiteten AQL-Programm

2.4 Abbrechen von AQL-Programmen

Ein laufendes AQL-Programm kann durch die Taste <DELETE> (Signal SIGKILL) aus dem Startfenster abgebrochen werden.

3 Programmiersprache AQL

Dieses Kapitel beschreibt die Formatierung der Ausgabe und stellt die Hauptstrukturelemente von AQL vor. Außerdem wird die Behandlung von Programmfehlern erläutert.

3.1 Formatierung der Alpha-Ausgabe

Die Alpha-Ausgabe, die mit den nachfolgenden Anweisungen formatiert wird, erfolgt immer auf den aktuellen Ausgabestrom, wie er mit der *file*-Anweisung vereinbart wird. Ist kein Ausgabestrom vereinbart, erfolgt sie in das Fenster, in dem ObjectD gestartet wurde (*stdout*).

3.1.1 Ausgabe von konstanten Zeichenketten – ‘

Die auszugebende Zeichenfolge wird in Rückwärtshochkommata **‘** eingeschlossen. In ihr können Tabulatorzeichen enthalten sein. Diese werden als solche ausgegeben und werden dann von den u.U. angesprochenen Ausgabegeräten wie Bildschirm oder Drucker interpretiert. Die eingeschlossene Zeichenfolge kann sich über mehrere Zeilen erstrecken. Die dann enthaltenen Zeilenendezeichen (*newline*) werden wie die Tabulatorzeichen behandelt.

3.1.2 Umdefinition des Ausgabebegrenzers – escape

Mit dieser Anweisung, die als Argument das neue Begrenzungszeichen für die Ausgabe von konstanten Zeichenketten erhält, wird das Begrenzungszeichen umdefiniert. Dies ist dann notwendig, wenn das Rückwärtshochkomma im Text mit ausgegeben werden soll und im AQL-Programm nicht durch einen Rückwärtsstrich **** geschützt wird.

```
escape <char>

string = "geschuetztes \’ Hochkomma")
```

3.1.3 Entwerten von Zeichen – \

Zeichen, die in AQL-Programmen als Programmsteuerzeichen interpretiert werden, können in Zeichenketten übernommen werden, indem ihnen ein Rückwärtsstrich **** vorangestellt wird.

Beispiel 1

```
\Newline(10)
```

Entwerten des ASCII-Codes *Newline(10)* für das AQL-Programm bei Verwendung innerhalb einer Zeichenkette.

3.1.4 Ausgabe eines Zeilenendes – nl

Die *nl*-Anweisung generiert ein Zeilenende auf dem aktuellen Ausgabestrom. Damit müssen AQL-Programme nicht jedes Zeilenende durch ein in Rückwärtshochkommata eingeschlossenes wirkliches Zeilenende erzeugen. Sie hat keine Argumente.

```
nl
```

3.1.5 Ausgabe von Leerzeichen – sp

Die *sp*-Anweisung (space) dient dazu, Lücken mit einer vom AQL-Programm berechneten Anzahl von Leerzeichen (Blanks) zu füllen, um variabel lange Zeichenketten spaltengerecht auszugeben. Sie erhält als Argument die Anzahl der zu generierenden Leerzeichen.

```
sp <int-expression>
```

Beispiel 2: Hello_World_1.aql

PROGRAMM:

```
/* 1. Version von Hello World */
escape @
sp 7; @Hello World it‘s great@ nl
```

AUSGABE :

```
       Hello World it‘s great
```

3.1.6 Ausgabe von Variableninhalten – [...]

Wie bei der Ausgabe von konstanten Zeichenfolgen besteht auch die Ausgabeanweisung für den Inhalt von Variablen oder Ausdrücken aus einer Zeichenklammer. Es sind drei Formen der Anweisung möglich.


```
[<expression>]
[<expression>;<min_field_width>]
[<expression>;<min_field_width>;<precision>]
```

Das erste Argument ist ein beliebiger Ausdruck, die beiden weiteren sind Ausdrücke mit einem Integerergebnis.
Der Mechanismus für die minimale Feldlänge <*min_field_width*> und die Steuerung der Ausgabe mit <*precision*> wurden von der Sprache C übernommen.

3.1.6.1 Ausgabe in der Standardform

Wird nur ein Argument angegeben, so wird dieser Ausdruck möglichst genau in eine lesbare Form umgewandelt. Dies geschieht nach Typ des Ausdrucks verschieden. Um den Test von AQL-Programmen zu vereinfachen, ist für jeden vorhandenen Typ eine Ausgabe vorhanden. Beispiele zu dieser und den folgenden Ausgabeformen finden Sie bei der Beschreibung von Ausdrücken.

```
[<expression>]
```

3.1.6.2 Ausgabe mit Längenangaben

Die Mindestlänge bestimmt die Mindestzahl der für die Ausgabe erzeugten Zeichen. Ist die Ausgabe kleiner, so wird der Rest mit Leerzeichen (Blanks) aufgefüllt. In diesem Fall wird bei einem negativen Vorzeichen die Ausgabe linksbündig erzeugt (d.h. es werden am Ende der Ausgabe Leerzeichen angefügt), ansonsten wird rechtsbündig justiert.

```
[<expression>;<min_field_width>]
```

Wird zusätzlich ein drittes Argument angegeben, so wird damit die Maximallänge der Ausgabe näher spezifiziert (siehe „Beispiel 6: real.aql“ auf Seite 3-12 und „Beispiel 7: group.aql“ auf Seite 3-13). Bei Zeichenfolgen wird damit die maximale Anzahl der auszugebenden Zeichen bestimmt, bei Zahlen die Anzahl der Nachkommastellen (Ausgabegenauigkeit)

```
[<expression>;<min_field_width>;<precision>]
```

3.1.6.3 Hexadezimale Ausgabe von Integerwerten

Hierfür wird der Angabe <*precision*> der Buchstabe „x“ vorangestellt. Dabei zeigt der Kleinbuchstabe „x“ an, daß auch für die hexadezimale Darstellung Kleinbuchstaben verwendet werden. Ein X bedeutet analog eine Ausgabe mit Großbuchstaben.

```
[<expression>;<min_field_width>;x]
[<expression>;<min_field_width>;X]
[<expression>;<min_field_width>;x <precision>]
```

Die erzeugte hexadezimale Darstellung wird durch die *precision*-Angabe nicht verkürzt, sondern nur verlängert. Dabei werden führende Nullen (0) erzeugt.

3.1.6.4 Ausgabe von Gleitkommawerten

Hierfür wird der Angabe *precision* einer der folgenden Buchstaben vorangestellt:

f Ausgabe mit der angegebenen Anzahl von Stellen nach dem Dezimalpunkt

e Ausgabe in Exponentialschreibweise mit der angegebenen Anzahl von Stellen nach dem Dezimalpunkt

E Ausgabe in Exponentialschreibweise wie bei Format „e“ mit dem Großbuchstaben E vor dem Exponenten

g Ausgabe mit der angegebenen Zahl von Dezimalstellen. Das System entscheidet auf Grund der Länge, ob die Exponentialdarstellung gewählt wird oder die Festpunktdarstellung

G Ausgabe wie bei Format „g“, jedoch mit dem Großbuchstaben E vor dem Exponenten.

3.2 Symbole – die Variablen in AQL

3.2.1 Begriffsdefinition

Ein Symbol ist die Bezeichnung für ein von AQL behandeltes Sprachobjekt. Ein Symbol kann beliebige Typen repräsentieren, während der Begriff der Variablen immer mit einem Typ verbunden ist.

Ein Symbol kann innerhalb seines Geltungsbereiches Sprachobjekte verschiedener Typen repräsentieren.Typkonflikte werden immer bei der Verwendung eines Symbols innerhalb eines Ausdrucks oder durch eine Funktion geprüft.

3.2.2 Namensaufbau

Ein Symbolname kann aus maximal 32 Zeichen bestehen. Zulässig sind alle Buchstaben in Groß- oder Kleinschreibung, die Zahlen und der Unterstrich _. Ein Name darf nicht mit einer Zahl beginnen.

Wie das UNIX-System ist auch AQL „case-sensitive“. Das bedeutet, daß dieselbe Buchstabenfolge in Großbuchstaben und in Kleinbuchstaben verschiedene Symbole bezeichnet (symbol != SYMBOL).

3.2.3 Geltungsbereich

Ein Symbol gilt immer innerhalb der Funktion, in der es vereinbart wird. Dabei gilt die Datei, in der sich das AQL-Programm befindet, als eine Funktion (main).

Bei geschachtelten Funktionen gelten für alle inneren Funktionen die Symbole der umgebenden Schalen. Das bedeutet, daß alle Symbole des Hauptprogramms (main) globalen Charakter haben. Die neue Vereinbarung eines Symbols in einer inneren Schale ist möglich. Sie gilt dann nur für die Funktion, in der sie getroffen ist, und für die in dieser Funktion eingeschlossenen Funktionen.

Symbole gelten nur innerhalb einer Source Datei, also nicht für AQL-Programme, die mit *aql(„...“)* aufgerufen wurden.

3.2.4 Deklaration von Symbolen – var

Es muß nicht jedes Symbol vereinbart werden, da standardmäßig eine Vereinbarung bei der ersten Verwendung eines Symbols für den Bereich der aktuellen Funktion vorgenommen wird (implizite Deklaration). Die explizite Deklaration ist nur dann zwingend, wenn der Geltungsbereich des Symbols außerhalb derjenigen Funktion liegt, in der es zum ersten Mal verwendet wird.

```
var <name>
```

3.2.5 Initialisierung von Symbolen

Nach einer Deklaration hat ein Symbol keinen Wert. Es besitzt den Typ „**invalid**“. Der Wert kann mit der AQL-Funktion „**valid**“ abgefragt werden.

Die Initialisierung mit einem Wert erfolgt mit einer Zuweisung (assignment), die dem Symbol auch den aktuellen Typ zuordnet. Dieser Typ ist der Typ des Ergebnisses des zugewiesenen Ausdrucks.


```
my_symbol_name = expr
```

Namen von Objekten, die interaktiv vergeben wurden, sind konvertibel zu AQL-Symbolen. Hat eine Linie z. B. den Namen L1 erhalten, so kann über diesen Namen in AQL wie auf ein Symbol zurückgegriffen werden.

Deklaration und Initialisierung können in einer Anweisung zusammengefaßt werden.
VAR a = 10 ;

Beispiel: Lösche Linie mit Namen L1

```
delete (L1)
```

3.2.6 Symboltypen

Folgende Symboltypen (Datentypen) sind bekannt:

`string`	zur Aufnahme von Zeichenfolgen
`boolean`	zur Aufnahme von "logisch wahr" (true) und "logisch falsch" (false)
`integer`	zur Aufnahme ganzer Zahlen
`real`	zur Aufnahme von reellen Zahlen
`group`	zur Zusammenfassung von Objekten
`date`	zur Aufnahme von Datum
`object`	allgemeiner Symboltyp (z.B. Zeiger auf Objekte und Aktionen)

Alle Symboltypen werden, soweit möglich, mit einer Zuweisung einer Konstanten desselben Typs erläutert. Der Typ *object*, der für die Bearbeitung des ObjectD-Modells und seiner Teile gebraucht wird, ist in der Online-Hilfe – „Systemattribute“ beschrieben.

AQL kennt die Typen seiner Objekte und weist diese den Symbolen automatisch zu. Werden Objekte miteinander verknüpft oder verglichen, so ist auf die Kompatibilität bzw. Verträglichkeit der Datentypen zu achten. Andernfalls erfolgt eine Fehlermeldung.

3.2.6.1 string

Konstante Zeichenfolgen werden in doppelte Hochkommata „ “ als Zeichenfolgenbegrenzer eingeschlossen. Falls dieses Zeichen Bestandteil der Zeichenfolge ist, wird ihm ein ückwärtsstrich \ vorangestellt. Eine Zeichenfolge kann aus beliebig vielen Zeichen bestehen.

Bei der Verwendung als Funktionsargument ist eine dafür bestehende Beschränkung einzuhalten (z.B. können Dateinamen im UNIX aus maximal 32 Zeichen bestehen).

Beispiel 3: string.aql (vgl. Abschnitt „Operatoren für Zeichenketten“ auf Seite 3-20)

PROGRAMM:

```
var string;
string = "inhalt "
delimiter = "mit Begrenzungszeichen \""
[string+delimiter] nl
[string+(delimiter-"* Begrenzungszeichen*")] nl
```

AUSGABE :

```
inhalt mit Begrenzungszeichen "
inhalt mit "
```

3.2.6.2 logical

Symbole mit diesem Typ können nur die Werte für „logisch wahr“ und „logisch falsch“ annehmen.

Bei der expliziten Belegung sind „true“ und „false“ in Kleinbuchstaben zu schreiben.

Beispiel 4: logical.aql

PROGRAMM:

```
      log_symbol              = false;
(1)   [log_symbol] nl;
      var not_init
      log_symbol              = valid(not_init);
(2)   [log_symbol;11] nl;
      not_init                = false;
(3)   [valid(not_init)] nl;
      log_next                = true
(4)   [log_next] nl
```

AUSGABE :

```
(1)   false
(2)   false
(3)   true
(4)  true
```

3.2.6.3 integer

Der Wertebereich für ganze Zahlen erstreckt sich von

```
-2147483646   (hexadezimal -80000000)
bis
2147483646    (hexadezimal 7FFFFFFF, 32 Bit Integer).
```

Bei der expliziten hexadezimalen Zahlenangabe ist immer der Kleinbuchstabe „x“ zu verwenden.

Beispiel 5: integer.aql

PROGRAMM:

```
      int   = 140488
(1)   ‘dec:‘[int;11]‘ hexa:‘[int;11;x] nl
      int   = 0xabcdef
(2)   ‘dec:‘[int;11]‘ hexa:‘[int;11;X] nl
      int   = 0x1111111
(3)   ‘dec:‘[int;11]‘ hexa:‘[int;11;X] nl
      int   = 1111111
(4)   ‘dec:‘[int;-11]‘ hexa:‘[int;11;X] nl
```

AUSGABE :

```
(1)   dec:     140488 hexa:    224c8
(2)   dec:   11259375 hexa:   ABCDEF
(3)   dec:   17895697 hexa:  1111111
(4)   dec:1111111     hexa:   10F447
```

3.2.6.4 real

Die Angabe der reellen Zahlen (Konstanten) ist als Festkommazahl (mit Angabe des Dezimalpunktes) oder in Exponentialdarstellung möglich. Der Wertebereich bewegt sich zwischen „-10^{308}“ und „10^{308}“. Wird eine für *integer* gebräuchliche Notation gewählt, so wird dem Symbol der Typ *integer* zugewiesen. Da das System bei Bedarf automatisch eine notwendige Konversion von *integer* nach *real* vornimmt, kann eine solche Zahl auch im Gleitkommaformat ausgegeben werden.

Beispiel 6: real.aql

PROGRAMM:
```
      real = 0xffffff;
(1)   [real;;f2] nl
      real = 1.11111111111e11;
(2)   ‘fixed:‘[real;12;f0]  ‘ standard:‘[real] nl
      real = 1.1111111111;
(3)   ‘fixed:‘[real;12;f0]  ‘ standard:‘[real] nl
(4)   ‘fixed:‘[real;-12;f10]‘ g:      ‘[real;;g] nl
```

AUSGABE :
```
(1)   16777215
(2)   fixed:111111111111 standard:1.111111e+11
(3)   fixed:           1 standard:1.111111
(4)   fixed:1.1111111111 g:       1.111111
```

3.2.6.5 group

Eine Gruppe Symbole kann verschiedene Typen (auch Gruppen) enthalten, da der Typ eines Symbols (Objekts) erst dann relevant ist, wenn sein Inhalt bearbeitet wird.

Beispiel 7: group.aql

PROGRAMM:

```
      group = { 1, 22, 333}
(1)   ‘group: ‘[group] nl
      group = { "a", "bb", "ccc" }
(2)   ‘group: ‘[group] nl
      group = { false, 1, 2.2, "drei", group}
(3)   ‘group: ‘[group] nl
      log = true;
      int = 0xFF;
      real= 1.4
      str = "str"
      group = { log, int, real, str, nl, group }
(4)   ‘group   : ‘[group] nl
(5)   ‘group.first: ‘[group.first] nl
(6)   ‘group.tail : ‘[group.tail] nl
(7)   ‘group   : ‘[group;2;2] nl
```

AUSGABE :

```
(1)   group: { 1, 22, 333 }
(2)   group: { a, bb, ccc }
(3)   group: { false, 1, 2.2, drei, { a bb ccc } }
(4)   group   : { true, 255, 1.4, str
       { false, 1, 2.2, drei, { a, bb, ccc } } }
(5)   group.first: true
(6)   group.tail : { 255, 1.4, str
       { false, 1, 2.2, drei, { a, bb, ccc } } }
(7)   group   : { tr, 255, 1.4, st
       { fa, 01, 2.2, dr, { a, bb, cc } } }
```

3.2.6.6 date

Symbole dieses Typs werden mit der AQL-Funktion *date* belegt (siehe Kapitel „Datumsfunktion – date“ auf Seite 5-3).

Beispiel 8: date.aql

PROGRAMM:
```
heute = date();
‘Erstellt: ‘[heute] nl
```

AUSGABE:
```
Erstellt: Tue May 28 17:48:54 1991
```

3.3 Ausdrücke

In diesem Kapitel werden folgende Themen behandelt:
- Operatoren
- Ergebnisse gemischter Arithmetik
- Ergebnisse gemischter Vergleiche
- Operatoren für Zeichenketten
- Ausdrücke mit Gruppen

3.3.1 Übersicht der Operatoren

Die folgende Tabelle gibt eine Übersicht der Operatoren:

Typ des Ausdrucks:	*Priorität*[*]	*gültig für*
`expr1 and expr2`	2	logical [**], integer (bitwise and)
`expr1 or expr2`	1	logical [**], integer (bitwise or)
`expr1 + expr2`	4	real, integer, string [***]
`expr1 - expr2`	4	real, integer, string [***]
`expr1 << expr2`	4	integer (bitshift)
`expr1 >> expr2`	4	integer (bitshift)
`expr1 * expr2`	5	real, integer [***]
`expr1 / expr2`	5	real, integer [***]
`expr1 ** expr2`	5	real, integer [***]

Typ des Ausdrucks:	*Priorität*[*]	*gültig für*
`expr1 in expr2`	3	rechter Operand Gruppe
`- expr`	6	real, integer
`expr1 < expr2`	3	real, integer, string, date ***
`expr1 <= expr2`	3	real, integer, string, date ***
`expr1 = expr2`	3	integer, string, date, real**** ***
`expr1 >= expr2`	3	real, integer, date, string ***
`expr1 > expr2`	3	real, integer, date, string ***
`expr1 != expr2`	3	real, integer, date, string, logical ***
`not expr`	6	logical, integer (bitwise not)
`(expr)`	7	all
`function()`	7	siehe Kapitel „Vordefinierte Funktionen“ auf Seite 5-1, Online-Hilfe ❍ AQL-Funktionen für Objekte, ❍ AQL-Funktionen für Aktionen, ❍ Values ❍ Systemattribute

* Die bei der Spalte "Priorität" angegebene Zahl bezeichnet die Stärke der Bindung der Operanden an den Operator. Je höher die Zahl, desto stärker die Bindung. Steht ein Operand zwischen zwei Operatoren, so wird zuerst die Operation mit der stärkeren Bindung, bzw. der höheren Priorität ausgeführt. Haben benachbarte Operatoren die gleiche Priorität, so erfolgt die Abarbeitung von links nach rechts.

** Bei den logischen Ausdrücken *and* und *or* wird der zweite Operand nur dann bearbeitet, wenn es notwendig ist. Das ist der Fall bei:
true and <expression2>
false or <expression2>

*** Der Typ des Ergebnisses ist bei Operanden verschiedenen Typs der Tabelle „Ergebnisse gemischter Arithmetik“ auf Seite 3-18 zu entnehmen.

**** Da die Prüfung auf Gleichheit bei real-Werten intern mit der maximalen Genauigkeit ausgeführt wird, ist es empfehlenswert, auf die Gleichheit innerhalb einer bestimmten Toleranz abzufragen.

Beispiel 9: priority.aql

PROGRAMM:

```
[true  or  nix ] nl;
[false and nix ] nl;
[3-2-1]          nl;
[3-(2-1)]        nl;
[1+2*3]          nl;
[2<<2-1]         nl;
[32>>2-1]        nl;
[32>>2**2-1]     nl;
[(32>>2)**2-1]   nl;
[1+2**2*3]       nl;
```

AUSGABE:

```
true
false
0
2
7
7
7
1
63
13
```

8

3.3.2 Ergebnisse gemischter Arithmetik

Die folgende Tabelle zeigt die Ergebnisse gemischter Arithmetik:

+ - * / ** *zweiter Operand*	*erster Operand* *real*	*integer*	*string*	*group*
real	real	invalid	invalid	nur +, -
integer	real	integer	invalid	nur +, -
string	invalid	invalid	nur +, - *	nur +, -
group	invalid	invalid	invalid	nur +, - **
object	invalid	invalid	invalid	nur +, -

* (concat, substring)
** (siehe Kapitel „Ausdrücke mit Gruppen“ auf Seite 3-21)

Es wird nicht geprüft, ob bei *integer* und *real* der Wertebereich überschritten wird (overflow).

Beispiel 10: mixed_arit.aql

PROGRAMM:
```
[3/2;;f3]              nl;
[3/2*1.0;;f3]          nl;
[3.0/2;;f3]            nl;
["Mein Hut der hat "+"drei Ecken"] nl;
```

AUSGABE:
```
001
1.000
1.500
Mein Hut der hat drei Ecken
```

3.3.3 Ergebnisse gemischter Vergleiche

Die folgende Tabelle zeigt die Ergebnisse gemischter Vergleiche:

< > <= >= = *!=*	*real*	*integer*	*string*	*logical*	*date*
real	logical	logical*	**	**	**
integer	logical*	logical	invalid	**	**
string	**	**	logical	**	**
logical	**	**	**	**	**
date	**	**	**	**	logical

* Das Ergebnis des *real*-Ausdrucks wird vor dem Vergleich durch Abschneiden der Dezimalstellen in eine Integer umgewandelt. Falls dabei ein Bereichsüberlauf auftritt, wird "invalid" als Ergebnis zurückgeliefert.

** Vergleich ist nur für die Vergleichsoperatoren "=" und "!=" zulässig.

= Bei einem Vergleich, bei dem nur ein Wert "invalid" ist, ist das Ergebnis immer der logische Wert "false". Sind beide Werte "invalid", so hat das Ergebnis den logische Wert "true".

!= Bei einem Vergleich, bei dem nur ein Wert "invalid" ist, ist das Ergebnis immer der logische Wert "true". Sind beide Werte "invalid", so hat das Ergebnis den logischen Wert "false".

3.3.4 Operatoren für Zeichenketten

Folgende Operatoren für Zeichenketten können verwendet werden:

+ Verkettung von Zeichenketten (concatenation)

- Teilkettenbildung: Dabei muß der rechte Operand (nach dem Zeichen -) eine Zeichenkette sein, die mindestens ein Wildcard * enthält. Dieses Zeichen steht stellvertretend für eine beliebige Zeichenfolge. Das Ergebnis der Operation besteht aus den durch die Wildcards repräsentierten Zeichen. Kann die explizit angegebene Zeichenkette des rechten Operanden nicht im linken gefunden werden, so besteht das Ergebnis des Ausdrucks aus dem Wert "invalid".

Beispiel 11: string.aql

PROGRAMM:
```
var                   string;
string                = "inhalt "
delimiter = "mit Begrenzungszeichen \""
[string+delimiter] nl
[string+(delimiter-"* Begrenzungszeichen*")] nl
```

AUSGABE :
```
inhalt mit Begrenzungszeichen "
inhalt mit "
```

3.3.5 Ausdrücke mit Gruppen

Folgende Attribute können für Gruppen verwendet werden:

`.first` Dieses standardmäßig für jede Gruppe definierte Attribut liefert als Ergebnis das erste Gruppenelement.

`.tail` Dieses immer vorhandene Attribut jeder Gruppe liefert eine Gruppe mit allen Gruppenelementen außer dem ersten Element (Komplement zu *.first*).

`.el_*` Dieses Attribut liefert das n-te Element der Gruppe. Damit kann gezielt auf einzelne Gruppenelemente zugegriffen werden (Arrays in anderen Programmiersprachen).

`or` Ergebnis ist eine Gruppe, die alle Elemente der beiden beteiligten Gruppen enthält. Es wird dabei gewährleistet, daß in der Ergebnisgruppe kein Element doppelt auftritt.

`*` Ergebnis ist eine Gruppe, die nur die Elemente enthält, welche in beiden beteiligten Gruppen enthalten sind.

`-` Ergebnis ist eine Gruppe, die nur die Elemente enthält, welche jeweils nur in einer der beiden beteiligten Gruppen enthalten sind.

`+` Die als linker Operand angegebene Gruppe bzw. das angegebene Gruppeneinzelelement wird um das nach dem Zeichen + angegebene Symbol erweitert. Falls es sich dabei um eine Gruppe handelt, werden beide Gruppen addiert (concatenation). Es wird nicht geprüft, ob das Ergebnis ein Element doppelt enthält. Falls es sich um ein Einzelement handelt, wird es je nach Schreibweise am Anfang oder Ende der Gruppe eingehängt.

`in` Falls der erste Operand in der als zweiter Operand aufgeführten Gruppe enthalten ist, hat der Ausdruck den logischen Wert "true", andernfalls "false".

`where`	Dieser Operator verbindet eine Gruppe mit einem logischen Ausdruck. Ergebnis sind alle Elemente, die diese Bedingung erfüllen. Voraussetzung ist, daß jedes Element bezüglich seines Typs in diese Bedingung eingesetzt werden kann. In der Bedingung wird das zu untersuchende Gruppenelement durch einen Punkt dargestellt. Soll ein Attribut des zu untersuchenden Gruppenelements geprüft werden, wird es mit *.<attributname>* angegeben.
`sort by`	Nach dem Operator *sort by* wird das Sortierkriterium angegeben. Dabei wird das Gruppenelement oder ein Attribut davon in der gleichen Form notiert wie im logischen Ausdruck der *where*-Bedingung. Das Sortierkriterium kann durch folgende Schlüsselworte näher spezifiziert werden: asc aufsteigende Sortierung (Voreinstellung) desc absteigende Sortierung
`and group`	Mit diesem Schlüsselwort werden Untergruppen von allen Elementen mit gleichem Attributwert gebildet. Es ist möglich, mehrere Sortierkriterien durch Komma getrennt anzugeben. Bei der Kombination mit *where* muß *where* den Sortierkriterien vorangestellt werden.
`select from`	Mit diesem Schlüsselwort kann aus einer Gruppe attributgesteuert eine andere Gruppe erzeugt werden Syntax <group> = select <attribute> from <group_expression>

An **Funktionen** zur Behandlung von Gruppen stehen zur Verfügung:

`len ( )`	Ermitteln der Elemente einer Gruppe (siehe Kapitel „Stringlänge – len“ auf Seite 5-9)
`index ( )`	Ermitteln eines Gruppenelements (siehe Kapitel „Ermitteln eines Gruppenelements bzw. Character aus String – index“ auf Seite 5-3)
`empty( )`	Test auf leere Gruppen (siehe Kapitel „Test auf leere Gruppen – empty“ auf Seite 5-3)
`pos ( )`	Ermitteln des Aufrufwertes eines Gruppenelementes

Beispiel 12: group_op.aql

PROGRAMM:

```
     zahlen = { 1, 2, 3, 4, 5 }
     gruppen_gruppe = {zahlen, zahlen, zahlen}
(1)  ‘zahlen:              ‘ [zahlen] nl
     diverses = { 4, "gelb", 2, 34.5, "leicht", false }
(2)  ‘diverses:            ‘ [diverses] nl
     gruppe_gesamt = zahlen + diverses
(3)  ‘Erweiterung:         ‘ [gruppe_gesamt] nl
     kopf = gruppe_gesamt.first
(4)  ‘Erstes Gruppenelement: ‘ [kopf] nl
(5)  ‘Rest der Gruppe:     ‘ [gruppe_gesamt.tail] nl
     /*‘Vereinigung:  ‘ [zahlen or diverses] nl */
(6)  ‘Durchschnitt:        ‘ [zahlen and diverses] nl
(7)  ‘Differenz 1-2:       ‘ [zahlen - diverses] nl
(8)  ‘Differenz 2-1:       ‘ [diverses - zahlen] nl
(9)  ‘Bedingung:           ‘ [zahlen where . >2 ] nl
(10) ‘Sort aufsteigend:    ‘ [zahlen sort by .  ] nl
(11) ‘Sort absteigend:     ‘ [zahlen sort by .  desc] nl
(12) ‘Sort absteigend mit Bedingung:  ‘
                       [zahlen where . > 1 sort by .  desc] nl
(13) ‘Einzelelement‘       ‘[zahlen.el_3] nl
(14) ‘Gruppe von Gruppen: ‘[gruppen_gruppe] nl
(15) ‘1.Auswahl:         ‘[select .first from gruppen_gruppe] nl
```

AUSGABE:

```
(1)  zahlen:              { 1, 2, 3, 4, 5 }
(2)  diverses:            { 4, gelb, 2, 34.5, leicht, false }
(3)  Erweiterung:         { 1, 2, 3, 4, 5, 4, gelb, 2, 34.5,
                            leicht, false }
(4) Erstes Gruppenelement: 1
(5)  Rest der Gruppe:     { 2, 3, 4, 5, 4, gelb, 2, 34.5,
                            leicht, false }
(6)  Durchschnitt:        { 2, 4 }
(7)  Differenz 1-2:       { 1, 3, 5 }
(8)  Differenz 2-1:       { gelb, 34.5, leicht, false }
(9)  Bedingung:           { 3, 4, 5 }
(10) Sort aufsteigend:    { 1, 2, 3, 4, 5 }
```

```
(11) Sort absteigend:     { 5, 4, 3, 2, 1 }
(12) Sort absteigend mit Bedingung:  { 5, 4, 3, 2 }
(13) Einzelelement:       3
(14) Gruppe von Gruppen: {{1,2,3,4,5} {1,2,3,4,5} {1,2,3,4,5}}
(15 1.Auswahl:             {1,1,1}
```

3.4 Kontrollanweisungen

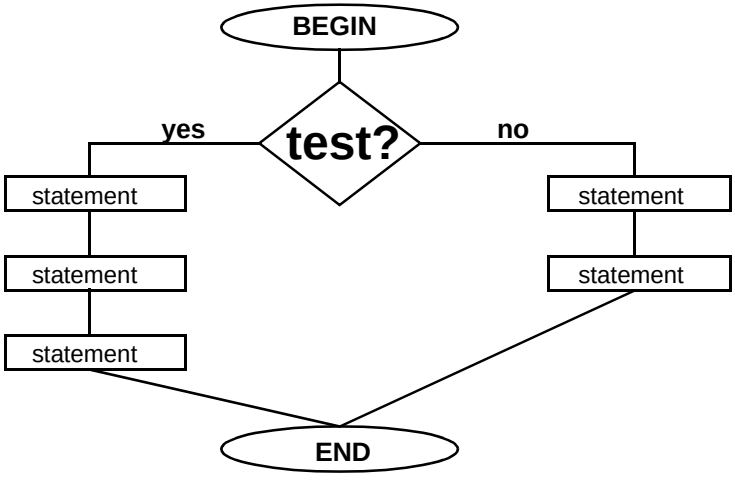
Folgende Kontrollanweisungen können verwendet werden:

Bedingung	`if...then...else`
Kontrollierte Schleife	`while...do`
Schleife über Gruppenelemente	`for..in..do`
Fallunterscheidung	`switch...case...`

3.4.1 Bedingung – if...then...else

Diese Anweisung ist in zwei **Formen** zulässig:

```
(1)   if <logical_expression> then
      <statements für logical_expression true>
      end
(2)   if <logical_expression> then
      <statements für logical_expression true>
      else
      <statements für logical_expression false>
      end
```



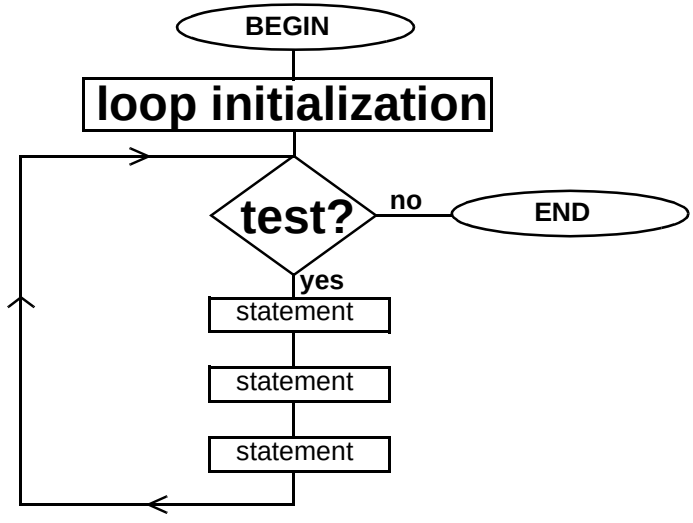
Diese Anweisung wird wie die anderen Kontrollanweisungen auch mit einem *end* abgeschlossen. Die *else..if*-Konstruktion wird durch einen *if*-Block innerhalb des *else*-Blocks realisiert.

3.4.2 Kontrollierte Schleife – while...do

While-Schleifen sind die Schleifenkonstrukte von AQL. Da *for*-Schleifen im herkömmlichen Sinn nicht existieren, werden auch diese über *while* abgebildet.

Diese Anweisung hat die **Form**:

```
while <logical_expression> do
   <statements für logical_expression true>
end
```



Solange der logische Ausdruck in der *while*-Anweisung den Wert „true“ annimmt, werden die Anweisungen bis zur zugehörigen *end*-Anweisung ausgeführt, und die Ausführung wird bei der *while*-Anweisung fortgesetzt. Ergibt der Kontrollausdruck in der *while*-Anweisung den Wert „false“, so wird hinter die zugehörige *end*-Anweisung verzweigt.

Es wird nicht untersucht, ob eine Endlosschleife vorliegt. Um eine Endlosschleife zu vermeiden, muß die Bedingung nach einer Ausführung „false“ werden.

3.4.3 Schleife über Gruppenelemente – for..in..do

Diese Anweisung, die sich von gleichlautenden Anweisungen anderer Sprachen unterscheidet und nur zum Durchlaufen von Gruppen dient, hat die **Form**:

```
for <loop_var> in <group_expression> do
   <statements für jedes element in group_expression>
end
```

Das Schleifensymbol <*loop_var*> wird automatisch angelegt.
Als <*group_expression*> sind alle zulässigen Gruppenausdrücke beliebiger Komplexität erlaubt.

3.4.4 Fallunterscheidung – switch...case...

Diese Anweisung ist in zwei **Formen** zulässig:

```
(1)   switch <expression>
         case <alternative_1>:
            <statements für expression=alternative_1>
         case <alternative_2>:
            <statements für expression=alternative_2>
      ...
         case <alternative_n>:
            <statements für expression=alternative_n>
      end
(2)   switch <expression>
         case <alternative_1>:
            <statements für expression=alternative_1>
         case <alternative_2>:
            <statements für expression=alternative_2>
      ...
         case <alternative_n>:
            <statements für expression=alternative_n>
         otherwise:
            <statements für alle anderen Fälle>
      end
```

Es ist Voraussetzung, daß der Typ von <*expression*> und <*alternative_1*> einen Vergleich zuläßt. Falls in der Form ohne *otherwise* keine Alternative gefunden wird, wird keine Anweisung ausgeführt.

Im Gegensatz zu C wird nur die Alternative durchlaufen, die gültig ist. Die nachfolgenden Alternativen werden übersprungen!

3.5 Funktionen

Es werden zwei **Arten** von Funktionen unterschieden:
- Vordefinierte Funktionen (siehe Kapitel „Vordefinierte Funktionen“ auf Seite 5-1)
- Funktionen, die der Anwender im AQL-Programm deklariert

3.5.1 Deklaration einer Funktion – function

```
function <function_name> (<parameter>, ...)
   <statements zu Ausführung der Funktion>
end
```

`<function_name>`	Der Name einer Funktion entspricht in seinem Aufbau dem eines Symbols.
`<parameter>`	Bei den Parametern handelt es sich um Symbole beliebigen Typs, welche in der Funktion nicht vereinbart werden. Die Zulässigkeit des Typs eines Parameters wird wie bei allen Symbolen zum Zeitpunkt der Ausführung in einem Ausdruck geprüft. Die Klammern nach dem Funktionsnamen sind auch dann notwendig, wenn die Funktion keine Parameter benötigt. Wird von der rufenden Funktion ein Parameter nicht angegeben, was zulässig ist, so besitzt er den Wert "**invalid**".

AQL behandelt alle Parameter als Ausdrücke und versorgt die gerufene Funktion mit einem Hilfssymbol, das das Ergebnis enthält (*call by value*) oder mit der Referenz (Zeiger) auf das Original-Symbol bei vorangestelltem Schlüsselwort *var* in der Aufrufleiste

(*call by reference*). Das heißt, wenn die Funktion diese manipuliert, wird auch der Wert der Variablen in der aufrufenden Funktion manipuliert.

Beispiel:

```
function my_function (var reference_parameter,
   value_parameter)
         .
         .
         .
```

Funktionswert

Jede Funktion liefert dem verwendenden Programm einen Funktionswert. Dieser Funktionswert wird behandelt wie das Ergebnis eines Ausdrucks. Damit können Funktionsaufrufe in Ausdrücke integriert werden, welche u.U. auch wiederum Funktionsparameter sein können. In einer Funktionsdeklaration wird der Rückgabewert als Argument der *return*-Anweisung definiert. Erfolgt der Rücksprung aus einer Funktion nicht über ein „return“ mit Argument, so ist der Funktionswert *invalid*. Fehlt das *return*, so erfolgt der Rücksprung bei der *end*-Anweisung.

Inhalt

Jede Funktion wird wie das Hauptprogramm behandelt und umgekehrt, d.h. sie kann Symbol- und Funktionsvereinbarungen enthalten. Wie in PASCAL sind auch geschachtelte Funktionsdeklarationen möglich. Bei größeren AQL-Programmen ist auf besondere Sorgfalt bei der Verwendung globaler Symbole zu achten.

Beispiel 13: param.aql

PROGRAMM:

```
function noparam()
                ‘function with no parameters‘ nl
end
noparam()

function noretvalue()
                ‘function noretvalue with no return value‘ nl
end
[type(noretvalue())] nl

function retvalue()
               return ("function retvalue returns this string")
end
[retvalue()] nl

function param(first_par,second_par)
                if not valid(first_par) then
                            first_par = "not given"
                end
                if not valid(second_par) then
                            second_par = "not given"
                end
                ‘function param with parameter:‘
                [first_par]‘,‘[second_par] nl
end
param(1,2)
param(1,)
param(,)
```

AUSGABE :
```
function with no parameters
function noretvalue with no return value
invalid
function retvalue returns this string
function param with parameter:1,2
function param with parameter:1,not given
function param with parameter:not given,not given
```

3.5.2 Vorausdeklaration einer Funktion – function...forward

```
function <function_name> (<parameter>, ...) forward
```

Hiermit kann eine Funktion im voraus ohne Inhalt vereinbart werden. Dabei muß die Anzahl der Argumente dieser Deklaration mit der Anzahl der Argumente bei der Deklaration der Funktion mit dem Inhalt übereinstimmen. Diese Form ist dann angebracht, wenn sich rekursive Funktionen gegenseitig aufrufen:
```
function a (count) forward

function b (count)
  if count != 0 then a(count-1) end
end

function a (count)
  if count != 0 then b(count-1) end
end
```

3.5.3 Rücksprung aus Funktionen – return

Die Verwendung ist in vier **Formen** zulässig:

Die **erste und zweite** Formen sind in ihrer Wirkungsweise gleich. Hier wird ein Funktionswert *invalid* zurückgeliefert.

```
return
return()
```

Bei der **dritten** Form spezifiziert der Ausdruck den Funktionswert. Ein *return* im Hauptprogramm bewirkt die Beendigung des Hauptprogramms.

```
return (<return_value_expression>)
```

Bei der **vierten** Form wird der Rückgabewert des AQL-Programms an ein aufrufendes AQL-Programm geliefert.
Ein Rücksprung ins aufrufende AQL-Programm erfolgt jedoch erst durch eine *return*-Anweisung oder bei Erreichen des Dateiendes.

```
return_code (<integer>)
```

3.5.4 Funktionszeiger

Um Programme variabel zu gestalten kann mit sogenannnten Funktionszeigern gearbeitet werden. Die allgemeine Form der Anweisung ist:

```
fp = FUNCTION <my_function>
```

3.5.5 Aufruf von Funktionen

Der Aufruf einer Funktion kann in Form einer Anweisung erfolgen. Die Funktion wird anhand ihres Namens oder anhand des Funktionszeigers identifiziert.
`<function_name> ([<parameter>,...])`

oder
`<function_pointer> ([<parameter>, ...]) ...`

Dabei wird ein möglicher Funktionswert ignoriert. Soll der Funktionswert weiterverarbeitet werden, so ist die folgende Form des Ausdrucks zu wählen:
`<symbol> <operator> <function_name> ([<parameter>,...]) ...`

Dabei ist zu berücksichtigen, daß die verwendete Funktion einen Funktionswert liefert und daß dieser im Umfeld seiner Verwendung typverträglich ist. Jede Funktion muß vor ihrer Verwendung vereinbart worden sein. Die Rekursion – die Verwendung einer Funktion durch sich selbst – ist möglich, wie im folgenden klassischen Beispiel gezeigt wird.

Beispiel 14: function.aql

PROGRAMM:
```
function hanoi(stack, from, to, use)
   if empty(stack.tail) then
      ‘moving ‘[stack.first;-6]
      ‘ block from ‘[from;-]‘ to ‘[to] nl;
   else
      hanoi(stack.tail,    from, use, to)
      hanoi({stack.first}, from, to,  use)
      hanoi(stack.tail,    use , to,  from)
   end
end

hanoi(   {"red" "green" "blue"},
          "source ", "target ", "parking")
```

AUSGABE :

```
moving blue  block from source  to target
moving green block from source  to parking
moving blue  block from target  to parking
moving red   block from source  to target
moving blue  block from parking to source
moving green block from parking to target
moving blue  block from source  to target
```

3.6 Dateibearbeitung

Folgende Funktionen können verwendet werden:
- Allgemeine Form der Dateizuweisung – file
- Zuweisung der Ausgabedatei – file output
- Zuweisung der Eingabedatei – file input
- Test von Zugriffsverfahren – file_access
- Test auf Existenz einer Datei – existf
- Test auf Existenz eines Verzeichnisses – existd
- Zuweisung der Fehlerdatei – file error_output
- Öffnen einer Binärdatei – bin_open
- Einlesen eines AQL-Symbols – get
- Positionieren in der Eingabedatei – get, relget
- Ermitteln der Position in der Eingabedatei – line
- Schreiben auf Binärdatei – bin_write, bin_write_byte, bin_write_short
- Lesen einer Binärdatei – bin_read_byte, bin_read_short, bin_read_int
- Einlesen eines Namens – parse_name
- Einlesen einer Zahl – parse_number
- Einlesen einer Zeile – parse_line
- Einlesen eines Schlüsselworts – parse_keyword
- Einlesen eines Kommentars – parse_comment
- Einlesen eines Strings – parse_string
- Schließen der Eingabedatei – close
- Schließen einer Binärdatei – bin_close

Siehe „Beispiel zur Dateibearbeitung“ auf Seite 7-4.

3.6.1 Allgemeine Form der Dateizuweisung – file

Die allgemeine Form der *file*-Anweisung ist:

```
file <file_name_string> <file_type>
```

Datei <*file_name_string*> bezeichnet einen Ausdruck, der als Ergebnis den Namen der angesprochenen Datei als String ergibt. Dieser Name muß den Systemkonventionen für Dateinamen entsprechen. Als Sonderform ist die Bezeichnung <*screen*> zulässig, die das Fenster anspricht, in dem ObjectD gestartet wurde. Dies ist auch die Voreinstellung für alle Dateitypen. Jede Zuweisung ist sofort wirksam.

- Bei Vergabe von Dateinamen ist das zugrundeliegende Betriebssystem zu beachten.
- Environmentvariable aus dem Betriebssystem sind in AQL nicht bekannt.

3.6.2 Zuweisung der Ausgabedatei – file output

Mit dieser Zuweisung wird die Ausgabe von konstanten Zeichenketten, sowie die Ausgabe der Anweisungen *[...]*,*nl* und *sp* gesteuert.

```
file <file_name_string> output
file <file_name_string>
```

Die Angabe des Schlüsselwortes *output* ist optional. Wird eine Datei innerhalb eines Programmlaufs zum ersten Mal angesprochen, so wird sie mit *write* eröffnet. D.h. falls sie noch nicht vorhanden ist, wird sie angelegt; falls sie existiert, wird ihr bisheriger Inhalt durch die nachfolgenden Ausgaben überschrieben.

Bei einer nochmaligen Dateieröffnung in einem AQL-Programmlauf, wird diese mit *append* eröffnet, d.h. sie wird durch alle nachfolgenden Ausgaben erweitert.

Dateien werden automatisch bei Programmbeendigung oder implizit durch Eröffnen einer neuen Ausgabedatei geschlossen. Auch das Rücksetzen auf die Voreinstellung <*screen*>, welche dem Standardstrom *stdout* entspricht, schließt eine offene Ausgabedatei.

3.6.3 Zuweisung der Eingabedatei – file input

Damit wird der Eingabestrom, der standardmäßig vom Bildschirm (<*screen*>) einliest (UNIX-Strom *stdin*), einer existierenden Datei zugeordnet.

```
file <file_name_string> input
```

Mit der ersten Anweisung *file input* auf eine Datei innerhalb eines Programmlaufs, wird diese Datei in einen internen Speicher eingelesen. Alle lesenden Zugriffe mit *get* werden von diesem internen Speicher ausgeführt, auch wenn die Datei zwischendurch neu eröffnet wurde.

3.6.4 Test von Zugriffsverfahren – file_access

Die Funktion *file_access* testet die Zugriffsrechte. Der Return-Wert ist vom Typ *boolean*.

```
<res_boolean> = file_access (<string1>,
                             <string2>)
<string1>   Dateiname
<string2>   r:  read
            w:  write
            x:  executable
```

Beispiele:

Die Reihenfolge der Buchstaben „r“, „w“ und „x“ ist beliebig.

```
file_access ("file_name", "rwx")
```
Ist Lesen, Schreiben und Ausführen möglich?

```
file_access ("file_name", "r")
```
Ist Lesen möglich?

```
file_access ("file_name", "w")
```
Ist Schreiben möglich?

```
file_access ("file_name", "rw")
```
Ist Lesen und Schreiben möglich?

3.6.5 Sonderzeichen am Anfang eines Dateinamens ersetzen – file_extend

Die Funktion *file_extend* ersetzt Sonderzeichen (#, +, ~) am Anfang eines Dateinamens.

```
<res_string> = file_extend (<string>)
<string>   Dateiname
```

3.6.6 Test auf Existenz einer Datei – existf

Die Funktion liefert „true“, wenn die angegebene Datei existiert.

```
<exists_bool> = existf(<filename>)
```

3.6.7 Test auf Existenz eines Verzeichnisses – existd

Die Funktion liefert „true“, wenn das angegebene Verzeichnis existiert.

```
<exists_bool> = existd(<dirname>)
```

3.6.8 Zuweisung der Fehlerdatei – file error_output

In diesen Strom, der standardmäßig dem Bildschirm (*stderr*) entspricht, werden alle Fehlermeldungen ausgegeben. Er wird in derselben Weise behandelt wie der Ausgabestrom.

```
file <file_name_string> error_output
```

3.6.9 Öffnen einer Binärdatei – bin_open

Die Funktion öffnet eine Binärdatei.

```
<fd> = bin_open (<filename>, <mode>)
<fd>        file_descriptor, wird bei den Funktionen bin_write*,
            bin_read* und bin_close benötigt.
<filename>  Dateiname
<mode>      "input"/"output"/"append"
```

Das erste Argument identifiziert die zu öffnende Datei. Die Funktion liefert im Erfolgsfall *file_descriptor*.

3.6.10 Einlesen eines AQL-Symbols – get

Das angegebene Symbol wird mit dem nächsten im Eingabestrom (siehe *Anweisung file .. input*) gefundenen Symbol versehen. Gleichzeitig wird der interne Lesezeiger an das Ende des gelesenen Symbols gesetzt, damit mit dem nächsten Lesezugriff das nächste Symbol eingelesen wird.

```
<symbol> = get()
```

Beim Lesen werden Leerzeichen und Zeilenenden ignoriert, sofern sie sich nicht innerhalb von Zeichenketten befinden, die in Anführungszeichen eingeschlossen sind. Ganze Zahlen und Gleitkommawerte (*integer* und *real*) beginnen mit einer Zahl oder dem Minuszeichen. Zeichenketten beginnen mit allen anderen Zeichen und enden mit dem ersten Leerzeichen oder dem Zeilenende, falls sie nicht in die Anführungszeichen eingeschlossen sind.

Siehe „Beispiel für Zeichensuche in einer Datei“ auf Seite 7-6.

3.6.11 Positionieren in der Eingabedatei – get, relget

Mit der folgenden Form der *get*-Funktion wird der Lesezeiger an die angegebene Position in der Eingabedatei positioniert.

```
<symbol> = get(<line_number>)
```

Die Lesefunktion *relget* positioniert den Lesezeiger relativ zur aktuellen Position. Positive Werte für „offset“ verschieben den Zeiger in Richtung „Dateiende“, negative Werte in Richtung „Dateianfang“. Falls die angegebene oder berechnete Zeilennummer einen Wert ergibt, der negativ oder größer als die Zeilenzahl der Eingabedatei ist, so wird der Wert *invalid* zurückgegeben. Ist die aktuelle Eingabedatei der Bildschirm (<*screen*>), so sind die Positionierfunktionen wirkungslos.

```
<symbol> = relget(<offset>)
```

3.6.12 Ermitteln der Position in der Eingabedatei – line

Die Nummer der aktuellen Zeile der Eingabedatei wird ausgegeben. Ist die aktuelle Eingabedatei der Bildschirm <*screen*>, so wird der Wert *invalid* zurückgegeben.

```
<line_number> = line()
```

Siehe „Beispiel für Zeichensuche in einer Datei“ auf Seite 7-6.

3.6.13 Schreiben in eine Binärdatei – bin_write, bin_write_byte, bin_write_short

Die Funktion *bin_write* schreibt jedes Argument in eine Binärdatei. Die Argumente müssen *integer*- oder *real*-Werte bzw. Strings sein.

```
bin_write (file_descriptor, arg1, arg2, ..., argn)
```

Die Funktion *bin_write_byte* schreibt das erste Byte jedes Arguments in eine Binärdatei. Die Argumente müssen Integerwerte sein.

```
bin_write_byte (file_descriptor, arg1, arg2, ..., argn)
```

Die Funktion *bin_write_short* schreibt den ersten Datentyp *short* jedes Arguments in eine Binärdatei. Die Argumente müssen Integerwerte sein.

```
bin_write_short (file_descriptor, arg1, arg2, ..., argn)
```

3.6.14 Lesen einer Binärdatei – bin_read_byte, bin_read_short, bin_read_int

Die Funktion *bin_read_byte* liest ein Byte aus einer Binärdatei. Die Funktion liefert im Erfolgsfall einen Typ „Integer“.

```
<i> = bin_read_byte (file_descriptor)
```

Die Funktion *bin_read_short* liest den ersten Datentyp *short* aus einer Binärdatei. Die Funktion liefert im Erfolgsfall einen Typ *integer*.

```
<i2> = bin_read_short (file_descriptor)
```

Die Funktion *bin_read_int* liest einen Datentyp *integer* aus einer Binärdatei. Die Funktion liefert im Erfolgsfall einen Typ *integer*.

```
<i4> = bin_read_int (file_descriptor)
```

3.6.15 Einlesen eines Namens – parse_name

Die Funktion liefert den Namensstring aus dem Eingabestrom (max 31 Zeichen).

```
<res_string> = parse_name()
```

3.6.16 Einlesen einer Zahl – parse_number

Die Funktion liefert den Wert der Zahl als Typ *integer* oder *real*.

```
<number> = parse_number()
```

3.6.17 Einlesen einer Zeile – parse_line

Die Funktion liefert den Inhalt der verbleibenden Zeile bis zum Zeilenende als String.

```
<res_string> = parse_line()
```

3.6.18 Einlesen eines Schlüsselworts – parse_keyword

Die Funktion liefert den Wert „true“, wenn der nächste zu lesende Eintrag dem *keyword* entspricht. Der Lesezeiger steht dann am nächsten Eintrag. Andernfalls bleibt der Lesezeiger an der Stelle stehen und die Funktion liefert „false“ als Ergebnis (Siehe „Beispiel für Zeichensuche in einer Datei“ auf Seite 7-6.).

```
<res_bool> = parse_keyword()
```

3.6.19 Einlesen eines Kommentars – parse_comment

Die Funktion liefert den nächsten Kommentar aus der Eingabedatei.

```
parse_comment ( )
```

3.6.20 Einlesen eines Strings – parse_string

Die Funktion liefert den nächsten String (zwischen doppelten Hochkommata „ “) aus der Eingabedatei.

```
parse_string ( )
```

3.6.21 Schließen der Eingabedatei – close

AQL schließt automatisch alle eröffneten Dateien bei Programmende. Das explizite Schließen der Eingabedatei ist deshalb nur dann notwendig, falls eine Datei im selben Programmlauf verändert wird und danach gelesen werden soll. Der Funktionswert ist vom Typ *logical* und sagt aus, ob das Schließen erfolgreich war (true) oder nicht (false). Nach dem Schließen der Eingabedatei ist die Standardeinstellung <*screen*> aktiv.

```
close()
```

3.6.22 Schließen einer Binärdatei – bin_close

Die Funktion schließt eine Binärdatei.

```
bin_close (file_descriptor)
file_descriptor     siehe bin_open
```

3.7 Fehler

In diesem Kapitel werden folgende Themen behandelt:
- Fehlerbehandlung
- Fehlersuche

3.7.1 Fehlerbehandlung

Während einer AQL-Sitzung können verschiedene **Fehler** auftreten:

Syntaxfehler	Syntaxfehler werden während der Vorübersetzungsphase von der AQL-Syntaxanalyse erkannt.
AQL-Ausführungsfehler	Diese Fehler werden während der Ausführungsphase des Programms erkannt. Typische Beispiele sind: Attributabfrage eines undefinierten Symbols Verwendung nicht initialisierter Variablen Ausdrücke mit inkompatiblen Operanden (z.B. Vergleich von *real* und *string*)

Im interaktiven Betrieb erscheinen Fehlermeldungen in einem Popup-Fenster auf dem Bildschirm.

Mit der Funktion ***error_response*** ist das Verhalten einer AQL-Prozedur nach dem Auftreten von Laufzeitfehlern zu steuern:

`error_response ("stop")`	AQL wird abgebrochen
`error_response ("continue")`	AQL läuft weiter

Im Batch-Modus schreibt der AQL-Interpreter alle Fehlermeldungen in die gerade aktive Meldungsausgabedatei. Standardmäßig ist dies der Bildschirm. Mit der Anweisung ***error_file*** kann die Ausgabe auf eine andere Datei umgelenkt werden.

Während der Übersetzungsphase ist die Ausgabeumlenkung nicht aktiv. Syntaxfehler werden also immer auf die Standardausgabe geschrieben.

Während der Ausführungsphase wird ein bestimmter Fehler nur einmal gemeldet, selbst wenn der Fehler innerhalb einer Schleife auftritt.

Syntax- und Ausführungsfehler werden in Englisch ausgegeben, Aktionsfehler in der für ObjectD eingestellten Sprache.

In einigen Fällen kann der AQL-Programmierer das Auftreten von Fehlern vorhersehen, und so programmieren, daß im Fehlerfall entsprechende Maßnahmen ergriffen werden (siehe Rückgabewerte der AQL-Funktionen und Kapitel „Meldung in eigenem Fenster ausgeben – display_error“ auf Seite 5-14).

Mit dem globalen AQL-Symbol ***errmes*** ist es möglich, den Text der Fehlermeldung auszuwerten. Folgende Attribute von *errmes* sind definiert:

`.text` Text der Fehlermeldung
`.line` Zeilennummer des Fehlerauftretens
`.file` Dateiname der AQL-Prozedur, die den Fehler enthält

errmes wird vor jedem Aufruf von modellbezogenen Funktionen auf „invalid“ gesetzt. Das *Text*-Attribut enthält im Fehlerfall den Text der Fehlermeldung. Die Identifizierung eines Fehlers anhand seiner Textmeldung ist jedoch sprach- und eventuell versionsabhängig.

Bei Aktionen kann auch das Attribut *valid* des erzeugten Aktionszeigers getestet werden. Es wird empfohlen diesen Test vor dem Zugriff auf *errmes* zu machen. Der Test des valid-Attributs führt nämlich zuerst zu einem Versuch, das Objekt neu auszuwerten.

Wenn das System sich nicht im Update-Modus befindet, werden erzeugte Objekte nicht sofort ausgewertet. Das *errmes*-Symbol wird dann erst durch den *valid*-Test gesetzt.

Beispiel

```
p = point_intersection(, 11, 12, 0)
if not valid(p) then [errmes.Text] end
```

3.7.2 Ausgabe von Fehlermeldungen unterdrücken – disable_messages

Die Funktion *disable_messages* unterdrückt die Ausgabe von Fehlermeldungen in AQL.

```
disable_messages ( )
```

3.7.3 Ausgabe von Fehlermeldungen zulassen – enable_messages

Die Funktion *enable_messages* läßt die Ausgabe von Fehlermeldungen in AQL zu.

```
enable_messages ( )
```

3.7.4 Dialog-Fehlerausgabe – short_messages

Die Funktion schaltet die Dialog-Fehlerausgabe für Laufzeitfehler ein oder aus.

```
short_messages (<boolean>)
```

3.7.5 Fehlersuche

Um Fehler in AQL-Programmen aufzudecken, kann der Programmierer an beliebiger Stelle innerhalb des AQL-Programms die Anweisung ***break*** einfügen.
`break(<prompt_string>)`

Diese Anweisung ist als AQL-Funktion *break* implementiert. Ihr einziges Argument ist ein String, der beim Erreichen dieser Funktion als Eingabeaufforderung durch den Interpreter ausgegeben wird und zwar nach der Anweisung, in das diese Funktion integriert ist. Die ausgegebene Anweisung ist zu diesem Zeitpunkt noch nicht ausgeführt. Der Anwender hat nun drei Möglichkeiten, die Verarbeitung fortzusetzen:

- Ausgabe von Variableninhalten mit dem Anweisung *[...]*
- schrittweise Ausführung dieses und der folgenden Anweisungen durch Drücken von <RETURN>
- Fortsetzung des AQL-Programms durch Eingabe des Buchstabens *g*

In diesem Zustand ist auch die Ein- und Ausgabe unterbrochen. Die genaue Stelle dafür kann wegen der Benutzerunterbrechung nicht definiert werden und ist auch von der Betriebsart (Dialog, Batch) des Systems abhängig.

Besondere Möglichkeiten für die Ausgabesteuerung in *prompt_string* sind:

```
%f   Dateiname
%n   Zeilennummer
%l   aktuelle Zeichen
```

Beispiele:

```
1. break ("%f line %n : %l --- break> ");
2. break ("%l>);
```

Die Funktion kann auch ohne Argumente aufgerufen werden:
`break()`

Dieser Aufruf entspricht dem oben angeführten 1. Beispiel.

Das Öffnen von Dateien kann mit der AQL-Funktion ***verbose*** verfolgt werden. Sie erhält einen Parameter, der diesen Modus ein- oder ausschaltet.
`verbose(logical_expression)`

4 AQL als Kommandosprache

Bisher wurde nur auf die nicht modellbezogenen Aspekte von AQL eingegangen. Im Rahmen von ObjectD dient AQL jedoch als eine Kommandosprache, die Erzeugung, Manipulation und Abfrage des CAD-Modells sowie die Erweiterung der Applikation ermöglicht.

4.1 Kompatibilität von AQL-Programmen

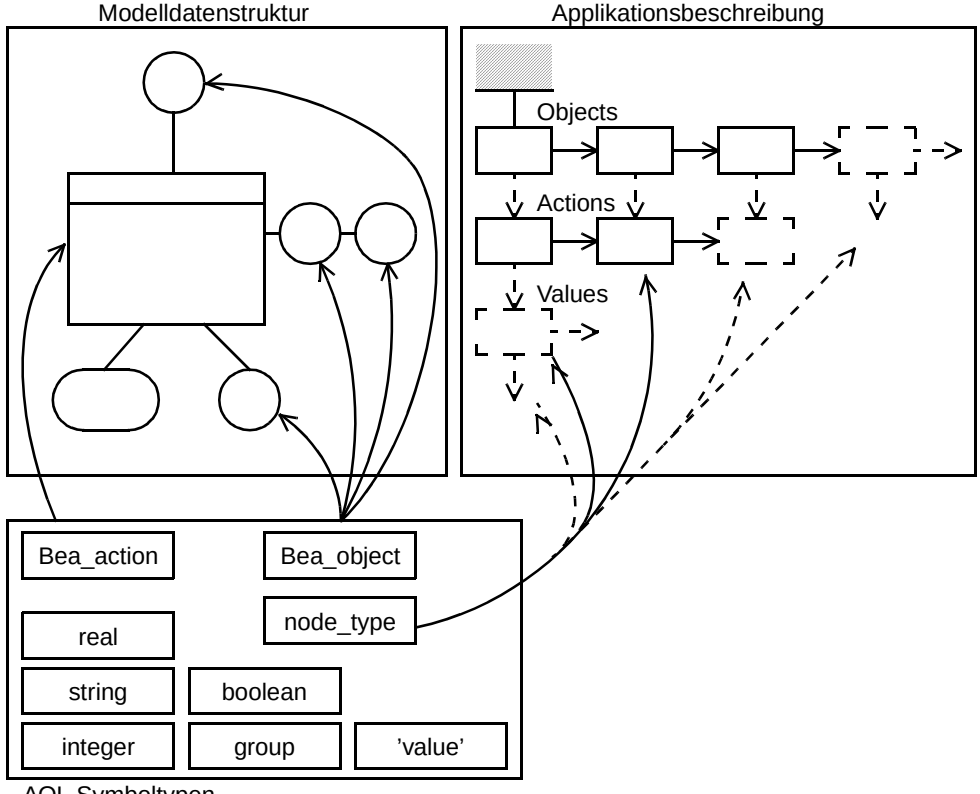
AQL als Sprache bleibt in ihrer Syntax und den Schlüsselworten grundsätzlich über die Versionen kompatibel. Durch Fehlerbehebungen kann die bisherige Toleranz des AQL-Compilers jedoch eingeschränkt werden. Der Programmierer sollte deswegen, wie in jeder Programmiersprache, besondere Syntaxtoleranzen nicht auszunutzen und streng an der spezifizierten Sprachsyntax bleiben (siehe Kapitel „Syntaxdiagramme“ auf Seite 8-1).

In AQL durch ObjectD implementierte Aktionen bleiben, soweit sie nicht der Fortentwicklung der Applikation entgegenstehen, versionskompatibel. Der AQL-Programmierer sollte sich nicht auf Implementierungstechniken von Funktionsrandeffekten verlassen.

Die Programmierung über Adressen ist wie in anderen Sprachen nur innerhalb eines laufenden AQL-Programmes möglich, da Adressen und Objektidentifikationen nicht über mehr als einen AQL-Lauf konstant gehalten werden können.

4.2 AQL-Symbole und ObjectD-Datenstruktur

Für den Zugriff auf die Datenstruktur existieren 3 Zeigertypen, die die Identifikation von Aktionen und Objekten in der Datenstruktur erlauben. Mit Hilfe dieser Identifikatoren kann dann auf Objekte und Aktionen zum Erzeugen, Löschen, Ändern und Abfragen Bezug genommen werden.



AQL Symboltypen

Bea_object	Zeiger auf beliebiges Objekt der Modelldatenstruktur
Bea_action	Zeiger auf beliebige Aktion der Modelldatenstruktur
node_type	Zeiger auf Applikationsbeschreibung; hier ist die nähereBeschreibung der Objekte, Aktionen, Values, Enums, ... abgelegt

4.3 Allgemeine Erzeugungs- und Kommandosyntax

4.3.1 Erzeugen von Absolutobjekten

Die allgemeine Syntax zum Erzeugen von Absolutobjekten durch AQL ist wie folgt aufgebaut:
`<res> = <objektname>_absolute (prop1, prop2, ..., propn)`
`<res>` ist vom Typ "Bea_object"

Beispiel: Absoluter Punkt an Position (100, 100)
```
p = point_absolute ({first_world, {100, 100}})
```

Benutzerdefinierte Objekte gehorchen der gleichen Regel; <objektname> ist dann, wie vom Benutzer definiert.

4.3.2 Einbringen und Ablaufen von Aktionen

Die allgemeine Syntax ist je nach Aktionstyp wie folgt festgelegt:

Aktion mit Resultatobjekt

```
<act_obj> = <actionname> ([id], prop1, ..., propn,
                          par1, ..., parn)
```

`<act_obj>`	Je nach Anwendung verwendbar als Bea_object (= Resultatobjekt) oder als Bea_action (= Zeiger auf Aktion)
`<actionname>`	gehorcht folgender Konvention <Objektname>_<Erzeugungsart>
`[id]`	Identifikator (optional) wird nur gebraucht innerhalb von execute functions von UDAs, z.B. zur Identifikation/Definition von Effektobjekten
`prop1 ... propn`	Properties des Resultatobjektes
`par1 ... parn`	Parameter der Aktion

Aktion ohne Resultatobjekt

```
<act> = <actionname> (par1, par2, ..., parn)
```

`<act>`	Zeiger auf Aktion vom Typ "Bea_action"
`<actionnname>`	Name der Aktion nach der Konvention <Hauptname>_<Nebenname>
`par1 ... parn`	Parameter der Aktion

Aktionen mit dem Attribut "drop"

```
<bool> = <actionname> (par1, par2, ..., parn)
```

`<bool>`	Ergebnisvariable "true" = Ablauf ok "false = Ablauf fehlerhaft
`<actionname>`	Name der Aktion nach der Konvention <Hauptname>_<Nebenname> Hauptname evtl. Objekt, auf das die Aktion wirkt

Benutzerdefinierte Aktionen

Je nach definierter Eigenschaft wie a. bis c.. Als <actionname> dient der Name der Aktion, wie vom Anwender definiert.

Wahlparameter, die mit dem Standardwert belegt werden sollen, können mit **,** übersprungen werden.

Für Parameter ist die verkürzte Schreibweise für Absolutobjekte möglich (siehe Kapitel „Implizite Schreibweise für absolute Objekte“ auf Seite 4-11). Damit können ungewollte Datenstruktureffekte bei oberflächlicher Programmierung auftreten ("Explosion" der Datenstruktur z.B. bei Bewegungssimulationen).

4.4 Funktionen auf das Modell

4.4.1 Neues Modell öffnen – input_new

Die Funktion *input_new* öffnet ein neues Modell.

Nur aus Kompatibilitätsgründen implementiert. Neu: **model_close** und **model_load/model_create**

```
input_new (<boolean>,
           <string>)
```

`<boolean>` `true:` löscht alle Objekte im aktuellen Modell
`<string>` Dateiname

4.4.2 Modell einlesen – input_read

Einlesen eines Modells

Nur aus Kompatibilitätsgründen implementiert. Neu: **model_load**

```
input_read (<string1>,
            <string2>)
```

`<string1>` Dateiname
`<string2>` Art des Einlesens (add, save, forget)

4.4.3 Maßeinheit umstellen – drawing_inch_mm

Die Funktion *drawing_inch_mm* gibt an, in welcher Einheit gearbeitet wird.

```
drawing_inch_mm (<boolean>)
```

`<boolean>` true bedeutet inch

4.4.4 Selektierte Objekte kopieren – inrect_copy

Die Funktion *inrect_copy* kopiert selektierte Objekte.

Nur aus Kompatibilitätsgründen implementiert. Neu: **group_copy**

```
inrect_copy (<integer>,
             <vector>,
             <select_rect>)
```

```
<integer>      Anzahl der Kopien
<vector>       Kopiervektor
<select_rect>  Selektionsmenge
```

4.4.5 Selektierte Objekte redefinieren – inrect_cut

Die Funktion *inrect_cut* redefiniert selektierte Objekte.

Nur aus Kompatibilitätsgründen implementiert. Neu: **redefine_cutall**

```
<redefined object> = inrect_cut (<world>,
                                 <select_rect>)
```

```
<world>             Koordinatensystem
<select_rect>       Selektionsmenge
<redefined object>  redefinierte Objekte
```

4.4.6 Selektierte Objekte duplizieren – inrect_duplicatecopy

Die Funktion *inrect_duplicatecopy* dupliziert selektierte Objekte.

Nur aus Kompatibilitätsgründen implementiert. Neu: **multi_duplicate**

```
<duplicated objects> = inrect_duplicatecopy (<integer>,
                                             <deleted_elm>,
                                             <vector>,
                                             <world>,
                                             <select_rect>)
```

`<integer>`	Anzahl
`<deleted_elem>`	gelöschte Beziehungen
`<vector>`	Kopiervektor
`<world>`	Koordinatensystem
`<select_rect>`	Selektionsmenge

4.4.7 Selektierte Objekte spiegeln – inrect_mirror

Die Funktion *irect_mirror* spiegelt die selektierten Objekte.

Nur aus Kompatibilitätsgründen implementiert. Neu: **group_copymirror**

```
<mirrored objects> = inrect_mirror (<integer>,
                                    <line>,
                                    <select_rect>)
```

`<integer>`	Anzahl
`<line>`	Spiegelachse
`<select_rect>`	Selektionsmenge
`<mirrored objects>`	gespiegelte Objekte

4.4.8 Separieren selektierter Objekte – inrect_separate

Die Funktion *inrect_separate* separiert selektierte Objekte.

Nur aus Kompatibilitätsgründen implementiert. Neu: **redefine_cutborder**

```
<selected> = inrect_separate (<deleted_elem>,
                              <world>,
                              <select_rect>)
```

`<deleted_elem>`	gelöschte Beziehungen
`<world>`	Koordinatensystem
`<select_rect>`	Selektionsmenge
`<selected>`	separierte Elemente

4.4.9 Modellidentifikator bestimmen – model

Gibt das Modell mit dem angegebenen Namen aus, bzw. das aktive Modell, wenn kein name angegeben wurde.

```
model (<string>)
```

`<string>`	Modellname (optional)

4.4.10 Modell sichern – output_save

Sichern eines Modells

Nur aus Kompatibilitätsgründen implementiert. Neu: **model_save**

```
output_save (<filename>)
```

4.4.11 Erzeugen eines neuen Modells aus der Selektionsmenge – output_savesection

Erzeugt aus der Selektionsmenge ein neues Modell

Nur aus Kompatibilitätsgründen implementiert. Neu: **model_outofselset**

```
output_savesection (<world>,
                    <select_rect>,
                    <filename>)
```

`<world>`	Koordinatensystem
`<select_rect>`	Selektionsmenge
`<filename>`	Dateiname

4.4.12 Aktives Modell definieren – set_active_model

Definiert das aktive Modell

`set_active_model`	`(<string>)`
`<string>`	Modellname

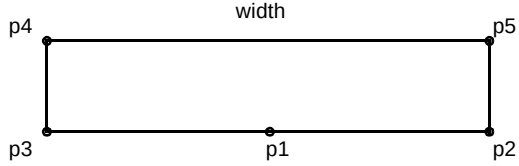
oder

`set_active_model`	`(<object>)`
`<object>`	Modell

4.5 Erzeugungsfunktionen – create

Beispiel

Ein Beispielprogramm erzeugt einen Block, der aus 4 Punkten und den zugehörigen Verbindungslinien besteht. Der Punkt *p1* erhält den Namen *reference_point* (Referenzpunkt), die Breite den Namen *width*.




```
width = length_absolute(100)
p1 = point_relative(origin, 100.0, 100.0)
p2 = point_samey(p1,length_qotient(width,2.0))
p3 = point_mirrorpoint(p2,p1)
p4 = point_samex(p3, 20.0)
p5 = point_relativex1y2(p2,p4)
line_pointpoint(,,,p3,p2)
line_pointpoint(,,,p4,p5)
line_pointpoint(,,,p2,p5)
line_pointpoint(,,,p3,p4)
name(p1, "reference_point")
name(width, "width")
```

length_absolute benötigt einen Realwert als Argument. In diesem Beispiel wird ein Integer-Wert übergeben. Integerwert-Argumente werden automatisch in Realwert-Argumente umgewandelt, wenn es erforderlich ist.

Die Aktionen *point_samey*, *point_samex* und *point_relative* werden im Normalfall mit einer Länge als zweitem Argument verwendet:

```
p4 = point_samey (p3,length_absolute(20.0))
```

Implizite Schreibweise für absolute Objekte

Für absolute Objekte (*length*, *angle*, *coordinates*, *number, ...*) kann der Wert direkt eingegeben werden:

```
p4 = point_samey(p3,20.0)
```

Das System erzeugt automatisch das Absolut-Objekt *length_absolute (20)* als Parameter für *p4*.

Beispiel 1:

Im Beispiel werden die AQL-Variablen *p1*, *p2*, *p3*, *p4*, *p5* und *width* verwendet. Dies bedeutet nicht, daß die erzeugten Objekte diese Namen erhalten. Um einem Objekt einen Namen zu geben, sollte die AQL-Funktion *name* benutzt werden. Die Variablen *p1*, *p2*, *p3*, *p4*, *p5* und *width* existieren nur zur Laufzeit des AQL-Programms. Sie werden nicht im CAD-Modell gespeichert. Die erzeugten Objekte bleiben jedoch permanent in der Datenstruktur.

Das gleiche Programm kann auch als AQL-Funktion geschrieben werden. Die Funktion hat dann zwei Argumente: den Referenzpunkt und die Breite des Blocks. In dieser Version werden dem Referenzpunkt und der Breite keine Namen gegeben. *p2* und *p3* werden auf anderem Wege konstruiert.

```
function block ( pref, width )
  p2 = point_samey(pref, width / 2)
  p3 = point_samey(p2, -width)
  p4 = point_samex(p3, 20.0)
  p5 = point_relativex1y2(p2,p4)
  line_pointpoint(,,,p3,p2)
  line_pointpoint(,,,p4,p5)
  line_pointpoint(,,,p2,p5)
  line_pointpoint(,,,p3,p4)
end

/* Hauptprogramm */

block ( point_relative (origin,100.0,100.0) , 100.0)
```

In dieser Version werden drei Objekte vom Typ *Länge* implizit konstruiert. Die ersten beiden haben die Werte *width/2* und *-width*, es besteht jedoch keine Relation zwischen ihnen.

Optionale Argumente können weggelassen werden. Das trennende Komma ist jedoch Pflicht.

Beispiel 2:

Die folgende Funktion erzeugt einen Turm von Blöcken mit nach oben hin abnehmender Breite. DIe Argumente sind der Referenzpunkt des untersten Blocks und die Anzahl der Blöcke.

```
function tower ( pref, count )
  i = 1
  p = pref
  group = {}
  while (i<= count) do
    block( p , (count + 1 -i) * 10.0)
    group = group + { p }
    p = point_samex(pref, i * 22.0)
    i = i + 1;
   end
  return(group)
end

/* Hauptprogramm */

tower ( point_relative (origin,100.0,100.0) , 10 )
```

Die Funktion liefert eine Gruppe der Referenzpunkte aller Blöcke zurück, die jedoch in diesem Beispiel nicht verwendet wird.

4.6 Dialogverzweigung bei ObjectD-Parametern

Der String "$" als Parameter hat die Spezialfunktion, Parameter über das 4-Spaltenmenü nachzufordern. Damit werden alle nicht versorgten Parameter nacheinander abgefragt, bis alle Parameter versorgt sind. Das System reagiert also wie bei einer Parameternachforderung im Dialog. Implizites Erzeugen über die vierte Spalte (und deren Rekursionen) ist dabei möglich.

Das Dialogforderungszeichen $ funktioniert nur auf oberster Ebene, d.h. Parameterlisten sind komplett einzugeben und abzuschließen (z.B. Parameterliste in Sequenz der Start-, End- und Hilfspunkte bei der automatischen Konturerstellung).

Oft empfiehlt sich, $ im Zusammenhang mit der Funktion *prompt* zu verwenden. Damit wird eine Eingabeaufforderung ausgegeben.

Beispiel:

```
prompt ("Bitte Relativpunkt eingeben!")
p = point_relative ("$",100,20)
```

Der Punkt "p" wird relativ zu einem vom Anwender festzulegenden Punkt definiert. Die beiden Längen "x" und "y" werden nicht nachgefragt, da sie versorgt sind.

```
prompt   ("Bitte geben Sie Start und Hilfspunkte für die
         Flächenerstellung ein")
plane_tracing (,,,,,"$",3,45)
```

Es wird eine Fläche AQL-gesteuert erzeugt. Die Liste der Start- und Hilfspunkte ist einzugeben und abzuschließen.

4.7 Änderungsfunktionen – edit

Die Funktion ***edit*** entspricht den Möglichkeiten im EDIT-Modus von ObjectD. Die allgemeine Form ist wie folgt definiert:

```
edit   (<Action_or_Object>,
          Prop1,
          Prop2,
           ...,
          Propn)
```

Parameter, die nicht verändert werden sollen, können mit "**,**" übersprungen werden.

Für Parameter ist die verkürzte Schreibweise für Absolutobjekte möglich (siehe Abschnitt „Implizite Schreibweise für absolute Objekte“ auf Seite 4-11). Damit können ungewollte Datenstruktureffekte bei oberflächlicher Programmierung auftreten ("Explosion" der Datenstruktur z.B. bei Bewegungssimulationen).

Die nächste Funktion zeigt, wie ein Block von einer Position zu einer anderen verschoben werden kann. Die Funktion hat drei Argumente:

- Das erste Argument identifiziert den Referenzpunkt des zu verschiebenden Blocks.
- Das zweite Argument identifiziert den Referenzpunkt, der über dem Block plaziert werden soll.
- Das dritte Argument zeigt, wie hoch der Block über dem Referenzpunkt zu plazieren ist.

```
function move ( block, reference_point, height )
  edit ( block, reference_point, height)
end
```

Diese Funktion ***move*** verschiebt den Block durch Editieren seines Referenzpunktes mit Hilfe der AQL-Funktion *edit*. Der Block wird automatisch durch den ObjectD-Mechanismus der Neuberechnung verschoben.
Im obigen Beispiel werden durch die *edit*-Operation gleich zwei Parameter auf einmal geändert. Es ist jedoch auch möglich, nur jeweils einen Parameter zu ändern:

```
function move ( block, reference_point, height )
  edit ( block, reference_point,)
  edit ( block, , height)
end
```

Die zweite Form von edit erlaubt das direkte Ansprechen **eines** Parameters. Dies erspart das Schreiben von Kommas bzw. das komplizierte Aufbauen von "edit strings" für die eval function bei unbekannten Aktionen.

```
edit (<par_node>, <Action_or_Object>, <new_value>)
```

4.8 Ändern einer objekterzeugenden Aktion

4.8.1 Redefinieren einer Aktion – redefine

Um die objekterzeugende Aktion zu ändern, kann die AQL-Funktion *redefine* benutzt werden. Das erste Argument der *redefine*-Funktion ist das Objekt, das geändert werden soll. Das zweite Argument ist das neue Objekt. Das zweite Argument muß den gleichen Objekttyp wie das erste haben.

redefine ist nur für Objekte im gleichen Layer sinnvoll.

```
redefine (block,point_samey (reference_point, height) )
```

4.8.2 Redefinieren einer Aktion eines Effekt-Objektes – redefine_effect

```
redefine_effect (<object1>,
                 <object2>)
```

`<object1>`	zu änderndes Objekt
`<object2>`	geändertes Objekt

4.9 Löschen

4.9.1 Löschen von Objekten – delete, undo delete

Objekte können mit der AQL-Funktion ***delete*** gelöscht werden. Das Argument zu dieser Funktion ist entweder ein Objekt oder eine Gruppe von Objekten.
`delete(block)`

Wird ein Objekt mit *delete* gelöscht, das bereits gelöscht war, können ungewollte Effekte entstehen ("toggle", siehe Attribut *.deleted*, Online-Hilfe – Attribute des Objekttyps Bea_object").

Die AQL-Funktion ***undo_delete*** hebt den Löschvorgang wieder auf (siehe Band *Grundlagen und Anwendung*, Kapitel "Sichtbarmachen gelöschter Objekte").
`undo_delete (block)`

4.9.2 Löschen einer Aktion – remove_action

Alle Objekte, die von dieser Aktion erzeugt wurden, werden gelöscht.
`remove_action (action_identifyer)`

4.9.3 Manipulatoraktion löschen – remove_manipulator

Macht eine Manipulatoraktion rückgängig
`remove_manipulator (<action>)`
`<action>` Manipulatoraktion

4.10 Namensvergabe – name

Ein Objekt kann mit der AQL-Funktion *name* benannt und umbenannt werden. Die maximale Länge eines Objektnamens ist 31 Zeichen. Das erste Argument identifiziert das zu benennende Objekt.

```
name (object,"the_name")
```

4.11 Definition von Benutzerelementparametern – inpar

Ein Objekt wird als Benutzerelementparameter definiert durch Anwendung der AQL-Funktion *inpar* (siehe Band *Grundlagen und Anwendung*, Kapitel "Parametrierung des Benutzerelements"):

4.11.1 Definition von Inputparametern – inpar

Es wird das übergebene Objekt oder alle Objekte einer übergebenen Gruppe als Inputparameter für das in Arbeit befindliche Modell definiert.

Nur aus Kompatibilitätsgründen implementiert. Neu: **udo_define / uda_define**

```
inpar (<object_or_group_of_objects>)
<object_or_group_of_objects>   Objekte (point, line, length,
                                        angle ...)
```

Die Funktion darf nur am Programmende vor output_save aufgerufen werden.

Falls Charakteristika (*linestyle*, *font*, ...) als Parameter deklariert werden sollen, muß die AQL- Funktion *inpar_ttype* verwendet werden.

4.11.2 Definition von Charakteristikparametern – inpar_ttype

Mit dieser AQL-Funktion können nur Charakteristika (*linestyle*, *font* ...) als Charakteristikparameter deklariert werden.

Nur aus Kompatibilitätsgründen implementiert. Neu: **udo_define/uda_define**

`inpar_ttype (<object>,<seq_number_of_char_parameter>)`

`<object>`	Objekt, dessen Charakteristik Inputparameter sein soll
`<seq_number_of_char_parameter>`	Nummer des Parameters innerhalb der Parameterfolge des Objektes

Objekte (*point*, *line*, *length*, *angle* ...) werden mit der AQL-Funktion *inpar* vereinbart.

Die Funktion darf nur am Programmende vor output_save aufgerufen werden.

4.12 Layertechnik

4.12.1 Erzeugen von Layern – layer_normal

Layer werden mit Hilfe der AQL-Funktion *layer* erzeugt:

Nur aus Kompatibilitätsgründen implementiert. Neu: **layer_create**

```
<new_layer> = layer_normal (<string1>,
                            <string2>,
                            <boolean>)
<string1>     Layername
<string2>     selectable
              inactive
              active
<boolean>     locked
<new_layer>   neuer Layer
new_layer = layer_normal("layer_name", "selectable",false/true)
new_layer = layer_normal("layer_name", "inactive")
new_layer = layer_normal("layer_name", "active")
```

Layer können mit der *edit*-Funktion editiert werden:

```
edit (my_layer,,,"active")
edit (my_layer, "new_name",status,gesperrt/nicht gesperrt)
```

4.12.2 Layer Statusdialog eröffnen – layer_status

Eröffnet den Dialog für Layerstatus.

```
layer_status ()
```

4.12.3 Zuweisen einer Layerfarbe – color_layer

Einem Layer wird eine Farbe zugewiesen.
`color_layer (layer, color index)`

4.12.4 Zuweisen einer Objektgruppe – move_set_to_layer

Eine Gruppe von Objekten wird einem Layer zugewiesen.

Nur aus Kompatibilitätsgründen implementiert. Neu: **layer_move_objects**

```
move_set_to_layer (<group_of_objects>,<layer>)
```

4.12.5 Objekt einem Layer zuweisen – move_object_to_layer

Ordnet ein Objekt einem Layer zu

```
move_object_to_layer (<object>,
                      <layer>)
```

`<object>`	Objekt, das zugewiesen werden soll
`<layer>`	Ziel-Layer

4.12.6 Layer unterordnen – move_sublayer_to_layer

Ein Layer wird zu einem Sublayer eines anderen.

Nur aus Kompatibilitätsgründen implementiert. Neu: **layer_move_sublayer**

```
move_sublayer_to_layer (<layer_to_be_moved>,<target_layer>)
```

Der gewählte Sublayer darf nicht gesperrt sein.

4.13 Benutzerdefinierte Attribute

4.13.1 Erzeugen von Attributen – create_attrib

Attribute werden mit der AQL-Funktion *create_attrib* erzeugt. Der Value kann mit dieser Funktion oder mit *set_attrib* angegeben werden.

Diese Form setzt einen Leerstring als Value

```
create_attrib (<object>,
               <string>)
```

`<object>`	Objekt
`<string>`	Attributname

oder

```
create_attrib (<object>,
               <string>,
               <value>)
```

`<object>`	Objekt
`<string>`	Attributname
`<value>`	`integer | real | boolean | string`

oder

Diese Form beschreibt den Value des Attributs in der Form: Value des Attributs <Name> des Objekts <Referenz>

```
create_attrib (<object1>,
               <string1>,
               <object2>,
               <string2>)
```

`<object1>`	Objekt
`<string1>`	Attributname
`<object2>`	Referenz
`<string2>`	Attributname der Referenz

4.13.2 Erzeugen von Attributen – create_attrib_expr

Attribute werden mit der AQL-Funktion *create_attrib* erzeugt. Der Value kann mit dieser Funktion oder mit *set_attrib* angegeben werden.

Diese Form setzt einen Leerstring als Value

```
create_attrib (<object>,
               <string>)
```

`<object>`	Objekt
`<string>`	Attributname

oder

```
create_attrib (<object>,
               <string1>,
               <string2>)
```

`<object>`	Objekt
`<string1>`	Attributname
`<string2>`	Ausdruck

oder

Diese Form beschreibt den Value des Attributs in der Form: Value des Attributs <Name> des Objekts <Referenz>

```
create_attrib (<object1>,
               <string1>,
               <object2>,
               <string2>)
```

`<object1>`	Objekt
`<string1>`	Attributname
`<object2>`	Referenz
`<string2>`	Attributname der Referenz

4.13.3 Vergabe von Attributwerten – set_attrib, set_attrib_expr

Attribute werden einem ObjectD-Objekt mit der AQL-Funktion *set_attrib* zugeordnet.

```
set_attrib (p1,"attribname",value)
```

Die AQL-Funktion *set_attrib_expr* ermöglicht es dem Benutzer, ein Attribut zu definieren, das aus einem AQL-Ausdruck besteht. Dieser Attributausdruck wird jedesmal, wenn er gelesen wird, neu ausgewertet.

```
set_attrib_expr (<object>,
                 <string>,
                 <aql_expression>)
```

<object>	Objekt
<string>	Attributname
<aql_expression>	AQL-Ausdruck

4.13.4 Kopieren einer Attributmenge – copy_user_atts

Bei der AQL-Funktion *copy_user_atts* erhält das Zielobjekt (target) alle Attribute des Ausgangsobjektes (source).

```
copy_user_atts (<target>, <source>)
```


4.13.5 Löschen von Attributen – delete_attrib

Attribute können mit der AQL-Funktion *delete_attrib* wieder gelöscht werden. Das erste Argument jeder dieser Funktionen kann ein Objekt oder eine Gruppe von Objekten sein.

```
delete_attrib (p1,"attribname")
```

4.14 Systemattribute

4.14.1 Gemeinsame Attribute

Attribute, die allen Objekttypen gemeinsam sind, sind als Attribute des Typs *Bea_object* im Kapitel "Systemattribute" aufgeführt.

Wurde das Objekt durch eine Aktion (z.B. als Resultatobjekt) erzeugt, so können zusätzlich die Attribute der Aktion (Bea_action) abgefragt werden.

4.14.2 Attribute für UDOs, Layers und Gruppen

Attribute für UDOs, Layer und Gruppen sind als Attribute der Objekttypen *group_type* und *layer_user_type* in der Online-Hilfe – "Systemattribute" aufgeführt.

4.14.3 Attribute für andere Objekttypen

Über das Symbol *program* kann auf Systeminformation zugegriffen werden. Die Attribute für den Objekttyp *program_type* sind in der Online-Hilfe – "Systemattribute" aufgeführt.

4.15 Zugriff auf die ObjectD-Datenstruktur

4.15.1 Zugriff über Namen und Datenstrukturdurchlauf

Auf die ObjectD-Datenstruktur kann über zwei verschiedene **Wege** zugegriffen werden:
- direkter Zugriff auf die Objekte über ihren Namen
- Durchlauf durch die Datenstruktur ausgehend vom AQL-Symbol *top*.

Der Zugriff auf Objekte **über Namen** auf ein Objekt ist nur möglich, wenn der Anwender dem Objekt einen Namen gegeben hat.

Der namentliche Zugriff ist z.B. dann sinnvoll, wenn der Anwender dem CAD-Modell Daten entnehmen, sie in einem Berechnungsprogramm verwenden, und dann das Modell abhängig von diesen Daten modifizieren möchte. In diesem Fall sind meist nur eine geringe Anzahl von Objekten von Interesse. Sie können deshalb interaktiv benannt werden.

Geht es um eine größere Anzahl von Objekten, können diese zu einer Gruppe zusammengefaßt werden, der dann ein Name gegeben wird.

Das AQL-Symbol *top* repräsentiert die oberste Ebene des aktiven Modells. Über seine Attribute kann die komplette Datenstruktur erreicht werden (jede Ebene hat diese Attribute):

`list_all`	Liste aller Objekte in einer Modelldatei
`list_out`	Liste aller Objekte in einer Modelldatei sortiert nach Abhängigkeiten (unabhängige Objekte zuerst)
`list_<obj>`	Liste aller Objekte eines bestimmten Typs, z.B. liefert *list_line* die Liste aller Linien.
`el_<number>`	Das Objekt mit der Identifikationszahl <*number*>.

Ein typisches Programm sieht folgendermaßen aus:

```
for i in top.list_line do
  Linie : [i.x1] [i.y1] [i.x2] [i.y2] nl
end
```

Das Programm durchläuft die Liste aller Linien und gibt die Positionsinformation jeder Linie aus.

Man könnte das gleiche Programm auch folgendermaßen schreiben:

```
for i in top.list_all where .prim_type = "line" do
  Linie : [i.x1] [i.y1] [i.x2] [i.y2] nl
end
```

Diese zweite Version bildet die Liste aller Objekte, reduziert jedoch die selektierte Gruppe auf die Objekte, die die Bedingung *prim_type = "line"* erfüllen. Diese Version ist weniger effizient, da sie nicht die Abkürzung über das Attribut *list_line* verwendet.

Die Entwicklung von AQL-Programmen erfordert einen Überblick über die für jeden Typ definierten **Attribute**.

Manche Attribute sind allen Objekttypen gemeinsam (z.B. *name*, *number*, *deleted*, *sons*). Andere Attribute sind abhängig vom Typ (wie z.B. *x1*, *y1*, *x2*, *y2* beim Typ *Linie*).

Die wichtigsten Attribute sind die der Objekte *UDO*, *layer* und *group* (Gruppe), durch die das Modell strukturiert wird. Über diese Attribute ist das gesamte Modell erreichbar. Beispiele sind die Attribute *list_all*, *list_<Objekttyp>* und *list_out*.

Eine Liste aller Attribute ist bezogen auf die Objekttypen in der Online-Hilfe – Systemattribute" aufgeführt.

4.15.2 Zugriff auf Attribute

Ein Bezeichner besteht aus einem Symbolnamen, gefolgt von einer Liste von Attributnamen. Der Symbolname und die einzelnen Attribute werden durch einen Punkt voneinander abgegrenzt. Der Bezeichner wird ausgewertet, indem das Attribut der Variablen genommen wird (erstes aus der Liste), davon wieder das Attribut (zweites der Liste) usw. bis die gesamte Liste abgearbeitet ist.

Beispiel :
```
obj.par_p1.name
```

wird ausgewertet, indem zuerst das *par_p1*-Attribut von obj und dann das Attribut *name* des resultierenden Attributs genommen wird.

Die Felder eines Bezeichners können wie oben konstant sein (*son*, *name*). Sie können jedoch auch Stringausdrücke sein. In diesem Fall muß der Ausdruck in eckige Klammern eingefaßt werden. Der erste Teil eines Bezeichners darf nicht variabel sein. Diese Möglichkeit ist der Verwendung der Funktion *search* vorzuziehen.

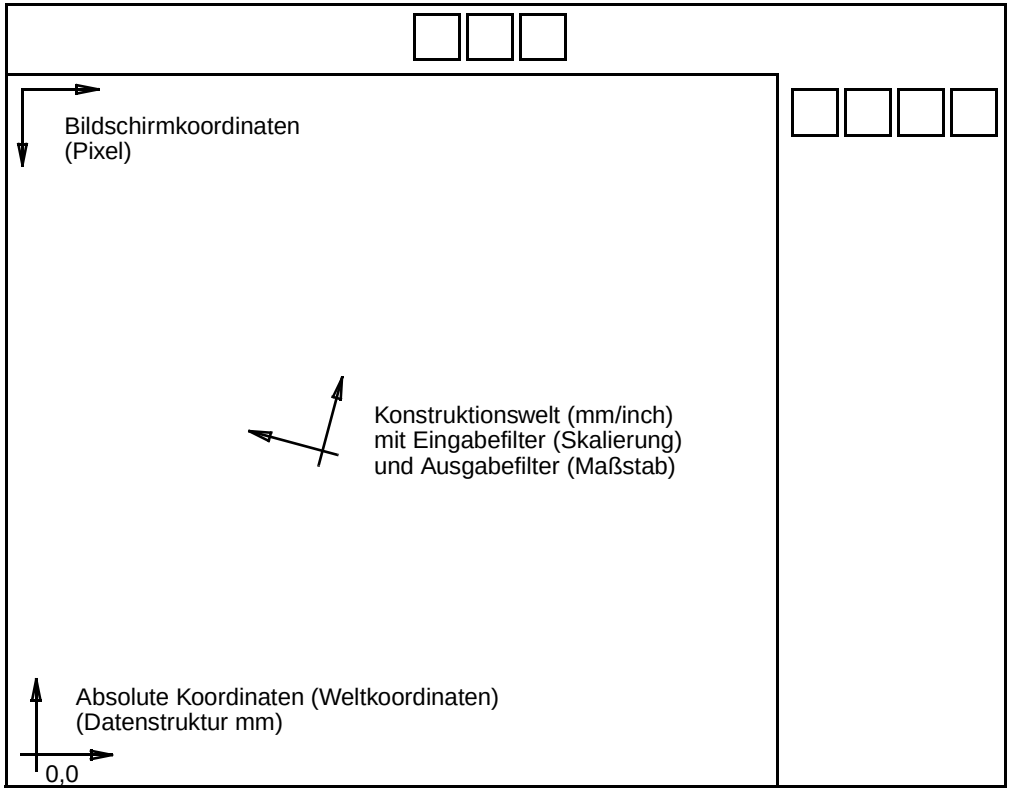
Beispiel:
```
obj.[i.type].color
obj.[i.type + "_list"]
```

4.15.3 Zugriff auf benutzerdefinierte Attribute

Einem vom Benutzer definierten Attribut muß beim Zugriff der String *user_* vorangestellt werden. Dadurch wird sichergestellt, daß die benutzerdefinierten Attribute nicht die vordefinierten Attribute überschreiben (siehe Attribute *user_<attrib>* und *attribs_user* des Objekttyps Bea_object, Online-Hilfe – "Systemattribute").

4.16 Bildschirmaufteilung

Viele AQL-Funktionen liefern oder verarbeiten Bildschirmkoordinaten. Diese werden in einer Gruppe {x,y} abgelegt, wobei *x* und *y* in Bildschirmpixeln von links oben in der Zeichenfläche gemessen werden.



4.17 Grafische Eingabe

4.17.1 Identifikation und Selektion von Objekten – pick, pick_group, pick_rectangle, pick_multiple_objects

Mit der AQL-Funktion *pick,* der AQL-Funktion *pick_group* und der AQL-Funktion *pick_rectangle* kann der Anwender mit der Maus Objekte auswählen (picken). Das Argument ist entweder ein Objekttypidentifikator, oder eine Gruppe von Objekttypidentifikatoren. Dadurch werden die auswählbaren Objekttypen eingeschränkt. Folgende Resultate werden geliefert:

`pick`	Objekt
`pick_group`	Gruppe von Objekten
`pick_rectangle`	Objektmenge, die, wie gewohnt, mit der Rechteckselektion zusammengestellt werden kann. Als Argument wird eine Gruppe der erlaubten Objekttypen angegeben.
`pick_multiple_objects`	Gruppe mit *boolean* und der Objektmenge. *boolean* besagt, ob die Selektion durchgeführt oder abgebrochen wurde. Die Objektmenge kann im Multiselektionsmodus zusammengestellt werden. Als Argumente werden eine Gruppe der erlaubten Objekttypen (als *node_type*) und eine Gruppe mit bereits selektierten Objekten angegeben.

```
obj = pick("line")
obj = pick( {"line", "circle"} )
obj = pick_group( {"line", "circle", "point"} )
obj_group = pick_rectangle( {"line", "circle", "point"} )
obj_group = pick_multiple_objects( {<allow_nod>}, {<presel_obj>} )
```

Das **erste Argument** einer Pickfunktion kann ein UDO oder ein Layer sein. In diesem Fall wird die Pickfunktion auf das entsprechende Benutzerelement, oder den entsprechenden Layer beschränkt.

```
obj = pick(user, "line")
obj = pick(my_layer, {"line", "circle"} )
obj_group = pick_group(my_layer, {"line", "circle", "point"} )
```

Als **letztes Argument** kann die Pick-Funktion eine Gruppe von Bildschirmkoordinaten enthalten, die den Anwenderklick an dieser Stelle simulieren.

Beispiel:
```
p = pick("point", {200,250})
```
Diese Funktion identifiziert einen Punkt, nächstgelegen zu den Bildschirmkoordinaten x = 200, y = 250.

Beispiel für *pick_multiple_objects*:
```
obj_group = {"point","line","circle"}
allowed_obj = application.list_objects where .name in obj_group
pre_sel_obj = top.list_point
sel_group = pick_multiple_objects ( allowed_obj, pre_sel_obj )
```

Diese Funktionen können nicht im Batch-Betrieb verwendet werden. Sie führen in diesem Fall zu einem Abbruch.

4.17.2 Einlesen einer Maus-Klickposition – pick_mouse

Die Funktion wartet auf eine Mauseingabe im aktiven Fenster und liefert eine Gruppe mit den Bildschirmkoordinaten zurück.
```
group = pick_mouse ()
```

Beispiel:
```
file <screen>
gp_pick = pick_mouse ()
gp_world = world_from_screen (gp_pick)
x = gp_world.first
y = gp_world.tail.first
[x] ‘ ‘ [y] nl
```

Die Funktion darf nicht im Batch-Modus angewendet werden.

4.17.3 Cursorposition ausgeben – peek_cursor

Die Funktion gibt die aktuelle Cursorposition aus.

```
grp = peek_cursor ()
grp:   group { x, y, view }
```

Die Funktion darf nicht im Batch-Modus angewendet werden.

Beispiel:

```
// create a circle
tx = circle_absolute (0,{"solid",3},"no_axes",
{first_world,{100,100},0,20})
vp = true; pos = true
// move this circle by moving the mouse
while vp or valid(pos) do
   pos = peek_cursor ()
   // if cursor in valid viewport > move
   if valid (pos) then
      vi = pos.el_3
      p = {real(pos.el_1) + real(pos.el_2)
      p = world_from_screen (p, first_world, vi)
      edit (tx,,,,{first_world,p,0,20})
      vp = false
   end // if
end // while loop
```


4.17.4 Koordinatentransformation nach Bildschirmkoordinaten – world_to_screen

Die Funktion liefert eine Gruppe mit den Bildschirmkoordinaten zurück.

```
screen_coord_gp = world_to_screen(<world_coord_gp>,
                                  [world_obj],
                                   [view])
```

4.17.5 Koordinatentransformation nach Weltkoordinaten – world_from_screen

Die Funktion liefert eine Gruppe mit den Weltkoordinaten zurück.

```
world_coord_gp = world_from_screen(<screen_coord_gp>,
                                   [world_obj],
                                    [view])
```

Beispiel:

Das folgende Beispiel zeigt das sinnvolle Zusammenspiel der drei Funktionen *world_to_screen*, *world_from_screen* und *pick_mouse*:

```
x = 1000
while x > 100 do
  gp = pick_mouse ()
  gpworld = world_from_screen (gp)
  gpscreen = world_to_screen (gpworld)
  [gp] [gpworld] [gpscreen] nl
  x = gp.first
end
```

4.17.6 Gedrehtes Koordinatensystem transformieren – trans_from_world

Die Funktion transformiert ein gedrehtes Koordinatensystem in ein horizontales Koordinatensystem.

```
trans_from_world (<pin>,
                  <world>
                  [, <inputtype>,
                  <outputtype>])
```

`<pin>`	Gruppe der Values, die transformiert werden sollen	
	Eingabetyp karthesisch: Eingabetyp polar:	{x y} {r alpha}
`<world>`	neues Koordinatensystem	
`<inputtype>`	Eingabetyp:	karthesisch (standard) polar
`<outputtype>`	Ausgabetyp:	karthesisch (standard) polar
`<result>`	AQL-Gruppe der transformierten Values (siehe pin)	

4.17.7 Koordinatensystem drehen – trans_to_world

Die Funktion transformiert ein horizontales Koordinatensystem in ein gedrehtes Koordinatensystem.

```
trans_to_world (<pin>,
                <world>
                [, <inputtype>,
                <outputtype>])
```

`<pin>`	Gruppe der Values, die transformiert werden sollen	
	Eingabetyp karthesisch:	{x y}
	Eingabetyp polar:	{r alpha}
`<world>`	neues Koordinatensystem	
`<inputtype>`	Eingabetyp:	karthesisch (standard) polar
`<outputtype>`	Ausgabetyp:	karthesisch (standard) polar
`<result>`	AQL-Gruppe der transformierten Values (siehe pin)	

4.18 Steuerung der Grafikausgabe

4.18.1 Grafikausgabe – redraw, flush, update_mode

Grafiken können mit der AQL-Funktion ***redraw*** nach Änderungen aktualisiert werden.

```
redraw ()
```

Die AQL-Funktion ***flush*** erzwingt ein Leeren der Puffer des Grafiksystems, um den Bildschirm sofort auf den aktuellen Stand zu bringen:

```
flush()
```

Der Benutzer kann entscheiden, ob das Modell nach jeder Änderung dargestellt werden soll oder nicht. Die entsprechende Einstellung erfolgt über die AQL-Funktion ***update_mode***:

```
update_mode(true)  /* Alle Änderungen werden sofort gezeigt */
update_mode(false)  /* Änderungen werden erst nach einem redraw
                                                    gezeigt */
```

4.18.2 Fenster zurücksetzen – view_reset

Die Funktion setzt Fenster zurück

```
view_reset ()
```

Die Funktion darf nicht im Batch-Modus angewendet werden.

4.18.3 Fenster verschieben – view_translate

Die Funktion verschiebt alle Fenster.

```
view_translate {<integer1> <integer2>}
<integer1>   from_x from_y
<integer2>   to_x to_y
```

Die Funktion darf nicht im Batch-Modus angewendet werden.

4.18.4 Darstellen von Objekten – show_normal, show_highlighted

Das angegebenene Objekt oder die in der angegebenen Gruppe enthaltenen Objekte werden neu dargestellt. Es wird nicht geprüft, ob es sich um ein darstellbares Objekt handelt, d.h. ob sich das Objekt in einem darstellbaren Layer befindet. Dabei wird jedes Objekt zunächst in der Hintergrundfarbe gezeichnet. Daraufhin wird es bei
`show_normal(group_or_object)`
mit der dem Objekt zugeordneten Farbe gezeichnet.
`show_highlighted(group_or_object)`
zeichnet es in der Markierungsfarbe.

Beispiel:

```
del_gp = pick_rectangle ("point","line","circle")
show_highlighted (del_gp)
prompt ("delete this Objects (y/n)")
ky = read_key ()
if ky = "y" then
   delete (del_gp)
else
   show_normal(del_gp)
end
```

4.18.5 Pop-Funktionen – lower_window, raise_window

Die Funktion ***lower_window*** schaltet das ObjectD-Ausgabefenster in den Hintergrund, so daß es kein benachbartes Fenster verdeckt. Dabei bleiben seine x- und y-Koordinaten unverändert.
`lower_window()`

Die Funktion ***raise_window*** schaltet das ObjectD-Ausgabefenster
in den Vordergrund. Dabei bleiben seine x- und y-Koordinaten unverändert.
`raise_window()`

Diese Funktionen können nicht im Batch-Modus angewendet werden.

4.19 Gruppen (Datenstrukturgruppen) – group_abs

Gruppen im CAD-Modell (im Gegensatz zu den internen AQL-Gruppen, die nur zur Laufzeit des AQL-Programms existieren) werden mit der AQL-Funktion *group_abs* erzeugt. Das erste Argument ist ein Symbol vom Type "Gruppe" (AQL-Gruppe). Alle Objekte dieser Gruppe müssen Objekte sein.

Nur aus Kompatibilitätsgründen implementiert. Neu: **group_create**

```
group_abs( "name",{p1, p2, p3} )
```

4.20 UDO/UDAs – user_symbol, user_outofstring

UDO/UDAs werden mit der AQL-Funktion *user_symbol* oder *user_outofstring* ins Modell eingetragen. Das erste Argument enthält den Dateinamen des Benutzerelements. Die restlichen Argumente werden als Parameter des UDO/UDAs betrachtet.

Die Funktionen sind nur aus Kompatibilitätsgründen implementiert. Benutzen Sie bei UDAs den Namen der Aktion, z.B. <my_action> (par1, par2, ...), bei UDOs die absolut erzeugt werden <UDO_name>_absolute (prop1, prop2, ...)

5 Vordefinierte Funktionen

Dieses Kapitel ist folgendermaßen gegliedert:
- Test und Umwandlung von Datentypen
- Basis-Funktionen
- Dialogschnittstellen
- Sonderfunktionen
- Systemkommandos
- In AQL nicht unterstützte Funktionen

5.1 Test und Umwandlung von Datentypen

5.1.1 Typermittlung – type

Diese Funktion liefert den Typ eines Ausdrucks in Form eines Stringwerts zurück. Für die vordefinierten Typen sind die möglichen **Rückgabewerte**:
- real
- integer
- string
- boolean
- date
- group
- invalid/valid
- Bea_object
- Bea_action
- node_type

... und alle applikationsdefinierten Typen

Für Objekte des Modells wird der Typname des Objekts zurückgeliefert.
`x = type(expression)`

5.1.2 Objekttyp abfragen – object_type

Gibt den Typ des Objektes aus
`object_type (<object>)`
`<object>` Objekt

5.1.3 Test auf Gültigkeit – valid

Die vordefinierte Funktion *valid (expression)* testet, ob ein Ausdruck ausgewertet werden kann. Wenn ja, wird der *boolean* Wert "true" zurückgeliefert, wenn nicht, der Wert "false". Alle Variablen werden so initialisiert, daß sie den Wert "invalid" (ungültig) haben. Die *valid*-Funktion testet, ob eine Variable initialisiert ist oder ob ein Attribut eines Objekts existiert.
Der valid-Test entspricht dem Test des Typs eines Ausdrucks auf "invalid":

```
function my_valid(expression)
  return( type(expression) != "invalid" )
end
```

Es wird empfohlen, den *valid*-Test zu verwenden, da er für diesen Zweck optimiert ist.

5.1.4 Test auf leere Gruppen – empty

Diese Funktion testet, ob eine Gruppe leer ist oder nicht. Ist sie leer, wird der *boolean* Wert "true" zurückgeliefert, wenn nicht, der Wert "false". Wenn das Argument keine Gruppe ist, wird eine Fehlermeldung ausgegeben, und die Funktion liefert "invalid" zurück.

```
x = empty(<group_expression>)
```

Beispiel

```
selection = {}
while empty (selection)
   prompt ("pick at least one point")
   selection = pick_group ("point")
end
```

5.1.5 Ermitteln eines Gruppenelements bzw. Character aus String – index

Diese Funktion liefert das n-te Element einer Gruppe. Das erste Argument identifiziert die Gruppe. Das zweite Argument muß ein Integerwert sein.

```
index (group, integer)
```

Die Funktion kann auch für die Übergabe des n-ten Zeichens eines Strings verwendet werden. Das erste Argument muß dann ein String sein.

Beispiel

```
index ( {1, "abc", 50.0},2) => "abc"
index("abc",2) ==> 98 (d.h. ASCII-Code von "b")
```

5.1.6 Datumsfunktion – date

Diese Funktion hat keine Argumente und liefert das aktuelle Datum zurück.

```
x = date()
```

5.1.7 Umwandlung in String – string

Alle Symbole können in ein Symbol vom Typ *string* umgewandelt werden. Dies geschieht mit der vordefinierten Funktion *string*. In der allgemeinen Form hat diese Funktion vier Argumente:

- Wert, der in einen String konvertiert werden soll
- Mindestlänge des resultierenden Strings
- optionale Genauigkeitsangabe (*precision*)
- Formatangabestring

Die Angaben für Mindestlänge und Genauigkeitsangabe sind im Kapitel „Formatierung der Alpha-Ausgabe“ auf Seite 3-1, beschrieben.

Der Formatstring enthält den Buchstaben, der bei der Ausgabefunktion der Genauigkeitsangabe vorangestellt wird.

Beispiel:

```
zahl = 1234.12345678
x1   = string(zahl,10,2,"f")
x2   = string(zahl,-10,2,"f")
x3   = string(zahl,10,2,"E")

‘ => x1 ========‘ [x1] ‘=========‘ nl
‘ => x2 ========‘ [x2] ‘=========‘ nl
‘ => x3 ========‘ [x3] ‘=========‘ nl
```

Ergibt die Ausgabe:

```
 => x1 ========   1234.12=========
 => x2 ========1234.12   =========
 => x3 ========  1.23E+03=========
```

5.1.8 Umwandlung in Großschrift – uppercase

Die Funktion liefert den in Großschrift gewandelten Ergebnisstring.

```
<res_string> = uppercase(<in_string>)
```

5.1.9 Umwandlung in Kleinschrift – lowercase

Die Funktion liefert den in Kleinschrift gewandelten Ergebnisstring.

```
<res_string> = lowercase(<in_string>)
```

5.1.10 Teilstringbildung – substr

Die Funktion liefert den Teilstring zwischen *pos_start* und *pos_end* (inclusive).

```
str = substr(<in_string>,
             <pos_start>
             [,<pos_end>])
```

5.1.11 Suchen nach einem String – pos

Gibt die Position des ersten Vorkommens des Suchstrings im durchsuchten String an. Die Suche beginnt vom Anfang her.

```
pos (<string1>,
     <string2>)
```

`<string1>`	zu durchsuchender String
`<string2>`	Suchstring
`<result>`	integer

Gibt das Vorkommen eins AQL-Symbols in einer Gruppe an.

```
pos (<group>,
     <aql_symbol>)
```

`<group>`	Gruppe
`<aql_symbol>`	AQL-Symbol
`<result>`	integer

5.1.12 Suchen nach einem String – rpos

1. Gibt die Position des letzten Vorkommens des Suchstrings im durchsuchten String an.

```
rpos (<string1>,
      <string2>)
```

`<string1>`	zu durchsuchender String
`<string2>`	Suchstring
`<result>`	integer

2. Gibt die Position des letzten Vorkommens des Symbols in der durchsuchten Gruppe an.

```
rpos (<group>,
      <AQL symbol>)
```

`<string1>`	zu durchsuchende Gruppe
`<string2>`	AQL-Symbol
`<result>`	integer

Beispiele

```
i = rpos ("/usr/home/bands/beatles", "/")           Ergebnis: 16
i = rpos ({"John", "Paul", "Ringo", "Paul"}, "Paul") Ergebnis: 4
```

5.1.13 Umwandlung von String nach Transformationsvorschrift - translate, translation_table

Die Funktion *translate* wandelt einen String nach einer mit *translation_table* vordefinierten Transformationsvorschrift um.

```
<res_string> = translate (<in_string>)
translation_table (<group_of_group_of_integer_and_string>)
```

`<group_of_group_of_integer_and_string>` beliebig lange Gruppe aus Einzelvorschriften: Diese müssen den ASCII-Code des umzuwandelnden Zeichens und des zu ersetzenden Strings als Gruppe erhalten.

Beispiel:

```
translation_table ({{97,"aa"}{98,"boe"}})
translate ("abba")
```

wandelt den String *abba* durch die Transformationsvorschrift in den String *aaboeboeaa* um. *a* (ASCII-Code 97) wird zu *aa* und *b* (ASCII-Code 98) zu *boe*.

5.1.14 Umwandlung in Real – real

Integer- und String-Symbole können mit der vordefinierten Funktion *real* in ein Real-Symbol konvertiert werden.

```
x = real(10)  => 10.0
```

5.1.15 Umwandlung in Integer – int, round

Real- und String-Symbole können mit der vordefinierten Funktion *int* in ein Integer-Symbol umgewandelt werden.

```
x = int(10.03)  => 10
```

Real-Zahlen verlieren bei der Konversion ihren Nachkommateil. Soll die Umwandlung mit Rundung erfolgen, so wird die AQL-Funktion *round* verwendet.

```
int = round(<real_or_string>)
```

5.1.16 Umwandlung von ASCII-Code nach String – chr

Ein Integer-Symbol aus dem Bereich 0...255 (ASCII-Code) wird in einen String umgewandelt.

Beispiel: `chr(97) => "a"`

5.2 Basis-Funktionen

5.2.1 Ermitteln des absoluten Wertes – abs

Die Funktion liefert einen absoluten Wert.

```
x = abs (real expr)
```

5.2.2 Stringlänge – len

Diese Funktion liefert die Länge eines Strings bzw. die Zahl der Gruppenelemente (einstufig).

```
x = len("ein String") => 10
x = len({a, b, c}) => 3
```

5.2.3 Stringlänge in mm ermitteln – get_world_length_of_string

Die Funktion *get_world_length_of_string* gibt die Länge eines Strings in mm an.

```
get_world_length_of_string (<string1>,
                            <string2>,
                            <real1>,
                            <real2>)
```

`<string1>`	Eingabestring
`<string2>`	Schriftart
`<real1>`	Schrifthöhe
`<real2>`	Schriftbreite

5.2.4 Definieren des n-ten Elementes in einer Gruppe, des n-ten Zeichens in einem String – set

Definiert das n-te Element in einer Gruppe

```
set (<group>,
     <integer>,
     <symbol>)
```

`<group>`	Gruppe
`<integer>`	Stelle
`<symbol>`	Objekt

Definiert das n-te Zeichen in einem String

```
set (<string>,
     <integer>,
     <string>)
```

`<string>`	zu bearbeitender String
`<integer>`	Stelle
`<string>`	Zeichen (1 character)

5.3 Mathematische Funktionen

5.3.1 Trigonometrische Funktionen

Folgende Funktionen haben *real*-Argumente und liefern einen *real*-Wert zurück, vorausgesetzt das Argument liegt im Gültigkeitsbereich der Funktion:

```
x = sin(real expr)
x = cos(real expr)
x = tan(real expr)
x = asin(real expr)
x = acos(real expr)
x = atan(real expr)
x = atan2(y, x)
```

Die Funktionen ***sin***, ***cos*** und ***tan*** erwarten eine Winkelangabe in Grad. Wenn bei der Berechnung von ***tan*** ein Überlauf eintritt, wird der Wert "invalid" zurückgeliefert.

Die Funktionen ***asin*** und ***atan*** liefern einen Wert zwischen -90 und 90 Grad.

Das Argument der Funktionen ***asin*** und ***acos*** muß im Bereich -1<=expr<=1 liegen. Ist das nicht der Fall, wird der Wert "invalid" zurückgeliefert.

Die Funktion ***acos*** liefert einen Wert zwischen 0 und 180 Grad.

Die Funktion ***atan2*** liefert den Arkustangens von "y/x". Durch Verwendung der Zeichen von "y" und "x" kann sie einen Wert zwischen -180 und 180 Grad zurückliefern.

5.3.2 Logarithmen

Folgende Funktionen haben *real*-Argumente und liefern einen *real*-Wert zurück, vorausgesetzt das Argument liegt im Gültigkeitsbereich der Funktion:

```
x = log(real expr)
x = ln(real expr)
```

Die Funktionen ***log*** und ***ln*** benötigen ein positives Argument. Ist das Argument negativ, wird der Wert "invalid" zurückgeliefert.

5.3.3 Exponentialfunktion

```
x = exp(real expr)
```

Folgende Funktion hat *real*-Argumente und liefert einen *real*-Wert zurück.

5.3.4 Quadratwurzel

Folgende Funktion hat *real*-Argumente und liefert einen *real*-Wert zurück, vorausgesetzt das Argument liegt im Gültigkeitsbereich der Funktion:

```
x = sqrt(real expr)
```

Die Funktion ***sqrt*** benötigt ein positives Argument. Ist das Argument negativ, wird der Wert "invalid" zurückgeliefert.

5.3.5 Modulofunktion – mod

Die Modulofunktion erwartet zwei Argumente. Die Argumente müssen *integer*- oder *real*-Werte sein. Die Funktion liefert den Rest der Division des ersten durch das zweite Argument zurück. Wenn beide Argumente *integer*-Werte sind, ist das Resultat ebenfalls ein *integer*-Wert. Andernfalls ist das Resultat ein Realwert. Wenn ein Fehler auftritt, wird der Wert "invalid" zurückgeliefert.

```
x = mod (a,b)
```

5.3.6 Minimum-/Maximum-Funktionen – min, max

Diese Funktionen verarbeiten Listen von Argumenten. Es wird der Minimal-(*min*) oder Maximalwert(*max*) der Liste zurückgeliefert.

```
x = min (real exp1, real exp2, real exp3, ... real expn)
x = max (real exp1, real exp2, real exp3, ... real expn)
```

5.3.7 Zufallszahl erzeugen – random

Erzeugt eine Zufallszahl zwischen 0 und 1

```
random ()
```

5.4 Dialogschnittstellen

5.4.1 Promptfunktion – prompt

Mit der Funktion *prompt* läßt sich eine Eingabeaufforderung ausgeben. Im Batch-Modus geschieht dies über *stdout*, im Dialog im Textbereich.
`prompt(prompt_expression)`

Der Wert von *prompt_expression* wird vor jeder Eingabe ausgegeben.

Beispiel:
```
prompt ("hier passiert Ausgabe im Textbereich")
```

5.4.2 Meldung im Meldungsfenster ausgeben – prompt_comment

Gibt eine Meldung im Meldungsfenster aus
`prompt_comment (<string>)`
`<string>` Meldungstext

5.4.3 Meldung in eigenem Fenster ausgeben – display_error

Der angegebene String mit der Nachricht an den Anwender wird in einem eigenen Fenster am Bildschirm ausgegeben. Nach der Bestätigung wird das Fenster wieder weggeblendet.
`display_error(<message>)`

Beispiel:
```
display_error("1.Zeile%2.Zeile%3.Zeile%")
```

Diese Funktion ist im Batch-Modus nicht zulässig.

Im String <*message*> wird das Zeichen % als Zeilenumbruchzeichen interpretiert. Am Ende des Strings <*message*> muß mindestens ein %-Zeichen enthalten sein.

5.4.4 Eingabefunktion – read

Mit der Funktion *read* kann eine Eingabezeile vom Textbereich (im Batch-Modus vom ObjectD-Fenster) gelesen werden. Das Argument muß ein *string*-Wert sein und wird als Eingabeaufforderung "???" geschrieben. Die Eingabezeile muß die Syntax eines AQL-Ausdrucks haben.

```
each_type = read (<prompt_string>)
```

Beispiel

```
default = 2
inp = false
while type(inp) != "integer" or inp > 3 or inp < 1 do
   inp = read ("Please give Integer between 1 and 3 =>|default")
end
```

prompt_string kann am Ende durch das Zeichen ‘|‘ getrennt den Standardwert erhalten, der durch einfaches <RETURN> übernommen werden kann.

```
<prompt> ‘|‘ <default value>
```

Weitere | werden als Buchstabe in den Defaultwert übernommen.

Die Funktion *read* liest den Datentyp, den der Anwender eingibt:

```
"asdf" ->  string
12     ->  Integer
3.4    ->  real
```

5.4.5 Stringeingabefunktion – read_string

Mit der Funktion *read_string* kann ein String im Textbereich (im Batch-Modus: im Prozeßfenster) gelesen werden. Im Gegensatz zur Eingabefunktion *read* muß der String nicht in doppelte Hochkommata " " eingeschlossen werden.

```
str = read_string (<prompt_string>)
```

Beispiel

```
str_from_user = read_string ("please type in a string!>")
```

prompt_string kann am Ende durch das Zeichen ‘|‘ getrennt den Standardwert erhalten, der durch einfaches <RETURN> übernommen werden kann.

```
<prompt> ‘|‘ <default value>
```

5.4.6 Einlesen eines Textblocks – read_textblock

Die Funktion öffnet ein spezielles Eingabeformular und fordert den Anwender zur Texteingabe auf. Als Argument wird ein Dateiname erwartet. Existiert die angegebene bereits, so wird ihr Inhalt im Eingabeformular dargestellt.

```
read_textblock (<filename>)
```

5.4.7 Lesen eines Tastaturanschlages – read_key

Das Programm wartet, bis eine Eingabe des Anwenders von der Tastatur erfolgt.

```
<key> = read_key()
<key>   String mit dem eingegebenen Zeichen
```

Beispiel

```
function change_view ()
   while true do
      prompt ("Use following Keys: All=<A>, Zoom In=<I>,
               Zoom Out=<O>, End=each other")
      key = uppercase (read_key ())
      switch key
         case "A":   view_full ()
         case "I":   view_zoomrectangle ("$")
         case "O":   view_unzoomrectangle ("$")
         otherwise:  prompt ("")
                     return ()
      end
   end
end // of function change_view
```

Diese Funktion ist im Batch-Modus nicht zulässig.

5.4.8 Tastatureingabe lesen – peek_key

Die Funktion liest einen Tastenanschlag und gibt diesen als String zurück. Ist gerade keine Taste gedrückt, wird ein Leerstring zurückgegeben.

```
<key> = peek_key ()
```

Beispiel

```
// example for peek_key fuction
k = ""; n = 1
prompt ("hit a key, <e> for exit")
// repeat until <e> key is pressed
while k != "e" do
   n = n + 1
   k = peek_key ()
   if k != "" then
      [k] ’ reaction time: ’[n] ’ loop passes’ nl
      n = 1
   end
end
```

Die Funktion darf nicht im Batch-Modus angewendet werden. Die Funktion wartet **nicht** auf einen Tastendruck.

5.4.9 Funktionstaste definieren – add_function_key

Die Funktion *add_fun_key* definiert eine Funktionstaste.

```
define_fun_key (<string1>,
                <string2>,
                <string3>)
```

`<string1>`	Funktionstaste
`<string2>`	Modifyer
`<string3>`	Aktionsname

5.4.10 Value einlesen – read_value

Die Funktion liest einen Value aus dem Textfeld oder Zeichenbereich. Bei einigen Values wird ein Formular geöffnet, in dem Sie den Value eingeben können. Es existiert nicht für jeden Value eine Eingabemöglichkeit, z.B. "circle_rec".

```
read_value (<node>,
            <initial>,
            <prompt>)
```

`<node>`	node des Value
`<initial>`	Vorbelegung (optional)
`<prompt>`	Eingabeaufforderung (optional)

5.4.11 Auswahl aus zwei Möglichkeiten – popup_boolean

Ein Formular, mit dem die zwei Möglichkeiten zur Auswahl angeboten werden, wird eingeblendet und erwartet die Eingabe des Anwenders.

```
<users_choice> = popup_boolean (<title_string>,
                                <true_choice>,
                                <false_choice>)
```

`<title_string>`	String mit dem Formulartitel (Platzangebot im Formular beachten)
`<true_choice>`	String mit der Beschreibung der einen Auswahlmöglichkeit
`<false_choice>`	String mit der Beschreibung der anderen Auswahlmöglichkeit
`<users_choice>`	logische Variable mit dem Wert "true" oder "false", je nachdem, welche Möglichkeit gewählt wurde

Diese Funktion kann nicht im Batch-Modus angewendet werden.

5.4.12 Auswahl aus drei Möglichkeiten – popup_3choices

Ein Formular, mit dem die drei Möglichkeiten zur Auswahl angeboten werden, wird eingeblendet und erwartet die Eingabe des Anwenders.

```
<users_choice> = popup_3choices (<title_string>,
                                 <first_choice>,
                                 <second_choice>,
                                 <third_choice>)
```

`<title_string>`	String mit dem Formulartitel (Platzangebot im Formular beachten)
`<first_choice>`	String mit der Beschreibung der ersten Auswahlmöglichkeit
`<second_choice>`	String mit der Beschreibung der zweiten Auswahlmöglichkeit
`<third_choice>`	String mit der Beschreibung der dritten Auswahlmöglichkeit
`<users_choice>`	Integer mit dem Wert 1, 2 oder 3, je nachdem welche Möglichkeit gewählt wurde

Diese Funktion kann nicht im Batch-Modus angewendet werden.

5.4.13 Auswahl aus einer Liste von Möglichkeiten – popup_list

Ein Formular, mit dem die angegebenen Möglichkeiten zur Auswahl angeboten werden, wird eingeblendet und erwartet eine Eingabe.

```
<users_choice> = popup_list (<title_string>,
                             <choice_list>)
```

`<title_string>`	String mit dem Formulartitel (Platzangebot im Formular beachten)
`<choice_list>`	Gruppe mit den Strings der einzelnen Auswahlmöglichkeiten
`<users_choice>`	ausgewählter String

Diese Funktion kann nicht im Batch-Modus angewendet werden.

5.4.14 Auswahl aus einer Liste von Möglichkeiten – popup_largelist

Das Formular, in dem die angegebenen Möglichkeiten zur Auswahl angeboten werden, wird eingeblendet und erwartet eine Eingabe. Ausgegeben wird ein String entsprechend der ausgewählten Möglichkeit im Formular.

```
<selected> = popup_largelist (<title string>,
                              <button string>,
                              <group elements>)
```

`<title string>`	String mit dem Formulartitel (Platzangebot im Formular beachten)
`<button string>`	String der Schaltfläche *Abbruch*(nach Anklicken wird das Formular verlassen)
`<group elements>`	Gruppe von Strings, die im Formular enthalten sein sollen

Beispiel

```
my_selected = popup_largelist("Select:",
                              "exit",
                              {"first_choice",
                               "second_choice",
                               "third_choice"})
[my_selected] nl
```

Diese Funktion kann nicht im Batch-Modus angewendet werden.

5.4.15 Formular mit Mehrfachauswahl – popup_multiplelist

Ein Formular mit einer Liste von Auswahlmöglichkeiten wird aufgeblendet und wartet auf eine Eingabe des Anwenders.

```
popup_multiplelist (<string1>,
                    <string2>
                    <string3>
                    <boolean>
                    <group of strings1>
                    <group of strings2>
```

`<string1>`	Titel
`<string2>`	OK-Schaltfläche
`<string3>`	CANCEL-Schaltfläche
`<boolean>`	Mehrfachauswahl erlaubt
`<group of strings1>`	Vorauswahl
`<group of strings2>`	Einträge
`<result>`	Ausgewählte Einträge

Beispiel

```
ala_cart = {"Nudelsuppe",
            "Gemuesesuppe"
            "Eierflockensuppe"
            "Schweinebraten"
            "Schnitzel"
            "Kalbs-Hormonbraten"
            "Vegetarierliebe"
            "Pudding"
            "Eis" }
menue    = {"Nudelsuppe"
            "Kalbs-Hormonbraten"
            "Pudding" }

a = popup_multiplelist("Tageskarte",
                       "Bestellen",
                       "Erbrechen",
                       true,
                       menue,
                       ala_cart)
[a] nl
```

Diese Funktion kann nicht im Batch-Modus angewendet werden.

5.4.16 Eingabe eines Dateinamens – popup_filename

Das Formular für die Eingabe von Dateinamen wird eingeblendet und erwartet die Eingabe des Anwenders.

```
<filename> = popup_filename (<iprompt>,
                             <choice_list_spec>,
                             <initial_dir>,
                             <initial_filename>,
                             <open_mode>)
```

`<iprompt>` String mit der Eingabeaufforderung und mit Angabe der Verwendung der auszuwählenden Datei

`<choice_list_spec>` String mit der Spezifikation (mit Wildcards) für die Dateien, die dem Anwender zur Auswahl angeboten werden

`<initial_dir>` String mit dem Verzeichnis, in dem die angebotenen Dateien gesucht werden. Ein leerer String ("") entspricht dem aktuellen Verzeichnis (Working Directory ".").

`<initial_filename>` String mit Vorbelegung für den auszuwählenden Dateinamen

`<open_mode>` String mit der Eröffnungsart für die Datei. Es sind folgende Eingaben dafür möglich:

"r"	read:	Öffnen zum Lesen
"w"	write:	Öffnen zum Erzeugen und Beschreiben: Falls die Datei existiert, erfolgt eine Fehlermeldung und ein neuer Dateiname wird angefordert.
"o"	overwrite:	Öffnen zum Schreiben bzw. Erzeugen, wenn nicht vorhanden.
"a"	append:	Erweitern: Die Datei muß existieren.

<filename> Name der gewählten Datei (als absoluter oder relativer Dateiname)

Beispiel: Sie finden ein „Beispiel zur Dateibearbeitung“ auf Seite 7-4.

Für Eingaben gelten die Bedienungsregeln des Datei-Formulars (siehe Band 1, Kapitel "Datei-Formular").

Diese Funktion kann nicht im Batch-Modus angewendet werden.

5.4.17 Farbauswahl – popup_color

Das Formular zur Farbauswahl wird eingeblendet und erwartet die Eingabe des Anwenders.

`<color> = popup_color (<back_ground_allowed>)`

`<back_ground_allowed>`	ohne Wirkung; wird wegen Aufwärtskompatibilität beibehalten
`<color>`	gewählte Farbe in RGB Gruppe

Diese Funktion kann nicht im Batch-Modus und auch nicht bei schwarz- weißen Bildschirmen angewendet werden.

5.4.18 Eingabe des Grades einer Spline – popup_degree

Das Formular für die Angabe des Grades einer Spline wird eingeblendet und erwartet die Eingabe des Anwenders.

`<selected> = popup_degree(<initial>)`

`<initial>`	Initialwert (z.B. 2 für Spline 2. Grades)
`<selected>`	ausgewählter Wert

Diese Funktion kann nicht im Batch-Modus angewendet werden.

5.4.19 Eingabe von Nachkommastellen – popup_digits

Das Formular für die Angabe der Nachkommastellen wird eingeblendet und erwartet die Eingabe des Anwenders.

```
<selected> = popup_digits(<initial>)
```

`<initial>` Initialwert, z.B.

bei 14:
1: Nachkomma-Null nicht unterdrücken
4: vier Nachkommastellen

Beispiel: 123,2000

bei 3:
Nachkomma – Null unterdrücken
3: 3 Nachkommastellen

`<selected>` ausgewählter Wert

Diese Funktion kann nicht im Batch-Modus angewendet werden.

5.4.20 Auswahl des Interface – popup_interface

Das Formular für die Auswahl des Interface wird eingeblendet und erwartet die Eingabe
des Anwenders.

```
<selected> = popup_interface(<initial>)
```

`<selected>` ausgewählter Wert

`<initial>` Initialwert vom Typ *formattype*
(siehe Online-Hilfe – "Values")

Diese Funktion kann nicht im Batch-Modus angewendet werden.

5.4.21 Auswahl der Sprache – popup_language

Das Formular für die Auswahl der Sprache wird eingeblendet und erwartet die Eingabe des Anwenders.

```
<selected> = popup_language()
```

`<selected>` String mit der ausgewählten Sprache

Diese Funktion kann nicht im Batch-Modus angewendet werden.

5.4.22 Auswahl des Strichmodus – popup_linestyle

Das Formular für die Auswahl des Strichmodus wird eingeblendet und erwartet die Eingabe des Anwenders.

```
<selected> = popup_linestyle(<styletype>,
                             <broken_on_off>)
```

`<selected>` ausgewählter Strichmodus

`<styletype>` Vorbelegung für den Strichmodus vom Typ *styletype_rec* (siehe Online-Hilfe – "Values")

`<broken_on_off>` logischer Wert, mit welchem dem Anwender die Auswahl unterbrochener Linien gestattet wird (=true) oder nicht

Falls dieser Parameter weggelassen wird, wird die Auswahl unterbrochener Linien gestattet.

Wird die Funktion ohne Parameter aufgerufen, wird als Vorbelegung der Wert {"solid" 3} angeboten.

Diese Funktion kann nicht im Batch-Modus angewendet werden.

5.4.23 Auswahl der Zeichnungsgröße – popup_size

Das Formular für die Auswahl der Zeichnungsgröße wird eingeblendet und erwartet die Eingabe des Anwenders.

```
<selected> = popup_size()
```

`<selected>` ausgewählte Größe

Diese Funktion kann nicht im Batch-Modus angewendet werden.

5.4.24 Auswahl des Formats – popup_format

Das Formular für die Auswahl des Formats wird eingeblendet und erwartet die Eingabe des Anwenders.

```
<selected> = popup_format(<initial>)
```

`<selected>` ausgewählter Wert

`<initial>` Initialwert vom Typ *formattype*_rec (siehe Online-Hilfe – "Values")

5.4.25 Auswahl der Schriftart – popup_font

Das Formualr zur Auswahl der Schriftart wird eingeblendet.

```
<font> = popup_font (<initial_value>)
initial value: string "Helvetica"
```

Diese Funktion kann nicht im Batch-Modus angewendet werden.

5.4.26 Eingabe von Fülleigenschaften einer Fläche – popup_plane

Das Formular *Fläche füllen* wird eingeblendet.

```
<selected> = popup_plane (<initial>)
```

`<initial>` Initialwert vom Typ *planetype_rec* (siehe Online-Hilfe – "Values")

Beispiel

```
my_plane = popup_plane ({"hatch_single" false})
[my_plane] nl
```

Diese Funktion kann nicht im Batch-Modus angewendet werden.

5.4.27 Funktionstaste belegen – define_fun_key

Die Funktion *define_fun_key* belegt eine Funktionstaste mit einer Aktion.

```
define_fun_key (<string1>,
                <string2>,
                <string3>)
```

`<string1>` Funktionstaste
`<string2>` Modifyer
`<string3>` Aktionsname

5.4.28 Funktionstastenbelegung löschen – reset_fun_keys

Löscht alle Funktionstastenbelegungen

```
reset_fun_keys ()
```

5.5 Sonderfunktionen

5.5.1 Beenden des Programmlaufs – quit_application

Diese Funktion beendet den Programmlauf.

```
quit_application ( )
```

5.5.2 Optionaleinstellung eines Parameters / Properties setzen – set_optional_on

Die Funktion definiert den ausgewählten Parameter bzw. die ausgewählte Property als optional.

```
set_optional_on (<node>, <string>)
```

<node>	node
<string>	Value

oder

```
set_optional_on (<node>, <node>)
```

<node>	node
<node>	Objekt

5.5.3 Optionaleinstellung eines Parameters / Properties rücksetzen – set_optional_off

Hebt die Vorbelegung eines Parameters oder eines Properties auf

```
set_optional_off (<node>)
```

<node>	node des Parameters, oder node des Properties

5.5.4 Abfragen, ob Optionalwert eines Parameters / Properties gesetzt ist – optional_set

Prüft, ob ein Parameter oder Property mit einem Optionalwert vorbelegt ist.

```
<result> = optional_set (<node>)
```

`<node>`	Node des Parameters oder Properties
`<result>`	true, wenn Parameter oder Property optional gesetzt ist

5.5.5 Objekt-Suchfunktion – search

Die *search*-Funktion sucht ein Objekt in der Datenstruktur anhand seines Namens. Diese Funktion steht nur noch aus Kompatibilitätsgründen zur Verfügung, da Stringausdrücke, die in eckigen Klammern eingeschlossen sind, Teile von Bezeichnern sein können. Das Argument muß ein Stringwert sein. Die Funktion liefert das gesuchte Objekt, wenn es gefunden werden kann. Gibt es kein Objekt mit dem angegebenen Namen, wird der Wert "invalid" zurückgeliefert.

```
p = search("prim_" + p2.name)
```

5.5.6 Objekt suchen nach dem Namen oder der id-Nummer – search_obj

Sucht ein Objekt nach dem Namen oder der id-Nummer im angegebenen user-Bereich

```
search_obj (<string>,
            <user>)
```

`<string>`	Name oder id-Nummer
`<user>`	user (optional)

5.5.7 Punkt-Suchfunktion – search_point

Die Funktion sucht einen Punkt in einer Gruppe von Objekten innerhalb eines angegebenen Toleranzkreises mit Mittelpunkt (*x*,*y*) und Radius (*epsilon*). Die Funktion liefert das Objekt, wenn es gefunden werden kann. Wurde kein Objekt gefunden, wird der Wert "invalid" zurückgeliefert.

```
object = search_point (<group of objects>,
                       double x,
                       double y,
                       double epsilon,
                       boolean nearest)
```

`<group g>`	Gruppe von Objekten	
`<double x>`	x-Weltkoordinate	
`<double y>`	y-Weltkoordinate	
`<double epsilon>`	Toleranz	
`<boolean nearest>`	true:	Der nächstliegende Punkt zu (x,y) wird selektiert.
	false:	Der erste Punkt, der innerhalb der Toleranz gefunden wird, wird selektiert.

Beispiel

```
function point (x, y)
// returns a object 'point' by using an existing
// point p(x,y) in a tolerance circle (radius 0.001)
// in the active model
   p = search_point (top.list_point, x, y, 0.001, false)
   if not valid (p) then
      p = point_relative (origin, x, y)
   end
   return (p)
end // of function point
```

5.5.8 Farbzuweisung für Objekte – color, layer_set_color

Dem angegebenen Objekt wird eine Farbe zugewiesen.
`color (object,<colortype>)`

Dem angegebenen Layer wird eine Farbe zugewiesen.
`layer_set_color (layer,<colortype>)`

`<colortype>` -64 < Wertebereich < 64

`negative Werte`	Objekt erhält dieselbe Farbe, falls der Layer, zu dem das Objekt gehört, eine bestimmte Farbe besitzt.
`Wert "0"`	Objekt erhält die für diesen Typ eingestellte Standardfarbe.

5.5.9 Layerfarbe abfragen – layer_get_color

Gibt die Farbe eines Layers zurück
`layer_get_color (<layer>)`

5.5.10 Definition von Parameterikonen – inpar_icon, inpar_iconascii

Der angegebene Inputparameter wird mit einer Ikone versehen, deren Inhalt hexadezimal angegeben ist.

Nur aus Kompatibilitätsgründen implementiert. Neu: **uda_define**, **udo_define**

```
inpar_icon(<number_of_parameter>,
           <icon>)
```

<icon> Jedes Pixel einer Ikone von 48 Zeilen mit je 48 Pixeln wird durch ein Bit einer hexadezimal angegebenen Zahl dargestellt. Ist das Bit nicht gesetzt, d.h. ist sein Wert 0, so wird das adressierte Pixel in der Hintergrundfarbe dargestellt. Ist es gesetzt, wird die Vordergrundfarbe gewählt. Jede andere als eine hexadezimale Darstellung wird nicht berücksichtigt. Die Adressierung geschieht nach folgendem Schema:

Byte 1:	pixel	1 bis 8	in Zeile	1
Byte 2:	pixel	9 bis 16	in Zeile	1
...				
Byte 6:	pixel	41 bis 48	in Zeile	2
Byte 7:	pixel	1 bis 8	in Zeile	2
...				
Byte 288:	pixel	41 bis 48	in Zeile	48

Das höchstwertige Bit ist immer dem Pixel *n*8+1* zugeordnet, das niederwertigste dem Pixel *n*8+8*.

Der angegebene Inputparameter wird mit einer Ikone versehen, deren Inhalt in Zeichenform angegeben ist.

`inpar_iconascii(<number_of_parameter>,<iconascii>)`

`<iconascii>` Jedes Pixel einer Ikone von 48 Zeilen mit je 48 Pixeln wird durch ein Bit eines Zeichens dargestellt. Ist es 0 oder " " (Blank), so wird das adressierte Pixel in der Hintergrundfarbe dargestellt, bei allen anderen darstellbaren Zeichen wird die Vordergrundfarbe gewählt. Nicht abdruckbare Zeichen werden nicht berücksichtigt. Die Adressierung geschieht nach folgendem Schema:

Byte	1:	pixel	1	in Zeile	1
Byte	2:	pixel	2	in Zeile	1
...					
Byte	48:	pixel	48	in Zeile	1
Byte	49:	pixel	1	in Zeile	2
Byte	2304:	pixel	48	in Zeile	48

5.5.11 Ausführung von AQL-Ausdrücken – eval

Diese Funktion führt den in einem String übergebenen AQL-Ausdruck aus. Falls dieser String AQL-Symbole enthält, so müssen diese an derselben Stelle, an der diese Funktion aufgerufen wird, schon definiert sein.
`<symbol>=eval(<string>)`

Beispiel

```
// a very simple commandline interpreter for expressions
while true do
   exp = read_string("give expression string
                      or 'end' for exit =>")
   if lowercase (exp) = "end" then return() end
   result = eval (exp)
   if not valid(result) then
      display_error("% sorry an error occured %")
   end
end
```

5.5.12 Ausführung externer Programme – exec

Die *exec*-Funktion führt ein externes (System-) Programm aus. Der volle Pfadname des Programms muß als erster Parameter angegeben werden (als Stringausdruck). Alle weiteren Argumente werden in Strings konvertiert und als Parameter an das Programm übergeben. Die *exec*-Funktion liefert einen Integerwert zurück. Bei erfolgreichem Abschluß wird der Wert 0 zurückgeliefert.

```
exec("/bin/ls","-l","*")
exec("/bin/who")
```

5.5.13 Lesen eines Wahlparameter- oder Wahlpropertywertes – get_optional

Die Funktion liest den derzeit eingestellten Wert eines Wahlparameters oder Wahlproperties.

```
<result> = get_optional (<node>)
```

`<node>`	node
`<result>`	object value node

5.5.14 Aufruf des Ikoneneditors – icon_editor

Aufruf des Ikoneneditors

```
icon_editor (<icon>)
```

5.5.15 Aktionen der Aktionsgruppe laden – load_actions_of_ag

Laden einer Aktion aus einer Aktionsgruppe

```
load_actions_of_ag (<node>)
```

`<node>` node der Aktionsgruppe

5.5.16 Aktionsgruppen vom Menü laden – load_ag_of_menu

Laden von Aktionsgruppen vom Menü
`load_ag_of_menu (<node>)`
`<node>` node des Menüs

5.5.17 Objekte aus einer Objektgruppe laden – load_objects _of_og

Laden von Objekten aus einer Objektgruppe
`load_objects _of_og (<node>)`
`<node>` node der Objektgruppe

5.5.18 Belegten Speicherplatz abfragen – memory_use

Maßzahl für den Speicherverbrauch durch den ObjectD (Modelle, AQL, ...)
`memory_use ()`

5.5.19 Standardparameter für Bemaßung setzen – optional_digits

Setzen von Standardparametern für die Bemaßung
`optional_digits (<control>)`
`<control>` Maßeinstellung

5.5.20 Zuordnen einer Ikone – set-icon

Definieren oder ändern der Zuordnung einer Ikone

```
set_icon (<node>,
          <icondefinition>)
```

`<node>` Node
`<icondefinition>` Ikonenbeschreibung

5.6 Systemkommandos

5.6.1 Ausführung von Shell-Kommandofolgen – system

Durch die *system*-Funktion wird der UNIX-Kommandointerpreter *sh* aufgerufen und ihm der übergebene String weitergereicht. Die Parameter für die hiermit gestarteten Kommandos sind weniger kritisch als bei der Funktion *exec*, da hier die in der Shell übliche Suchfolge für Kommandos gilt. Außerdem können Wildcards und die Ein- und Ausgabeumleitung verwendet werden. Werden einzelne Kommandos mit dieser Funktion ausgeführt, so ist deren Abarbeitung wegen der vorangehenden Interpretation durch die Shell geringfügig langsamer als mit der AQL-Funktion *exec*.

```
system ("/bin/ls -l *.aql")
system ("/bin/who >me")
```

5.6.2 Lesen und Setzen des aktuellen Verzeichnisses – wd, cd

Die Funktion ***wd*** liefert den Pfadnamen des aktuellen Verzeichnisses als String. Sie hat keine Argumente.

```
oldwd = wd ()
```

Die Funktion ***cd*** wechselt das aktuelle Verzeichnis. Sie liefert im Erfolgsfall den boolean-Wert "true".

```
ok = cd ("/usr/new_dir")
```

5.6.3 Ausgabe des Anwendernamens – who

Die Funktion *who* liefert den Anwendernamen als String. Sie hat keine Argumente.

```
iam = who ()
```

5.6.4 Inhalt des Verzeichnisses anzeigen – dir

Die *dir*-Funktion liefert den Inhalt des angegebenen Verzeichnisses als eine Gruppe von Strings. Wenn kein Argument angegeben wird, liefert sie den Inhalt des aktuellen Verzeichnisses. Ein optionales zweites Argument dient als Wildcard-Spezifizierung der auszugebenden Dateien. Wenn das Verzeichnis nicht gefunden oder nicht geöffnet werden kann, wird der Wert "invalid" zurückgeliefert.

Beispiel

```
gr = dir ()
gr = dir ("src")
gr = dir ("src","*.c")

for i in dir (".","*.c") do
  exec ("/bin/cc",i)
end
```

5.6.5 Ausgabe des Programmnamens im Textbereich – write_name

Der Programmname wird im Textbereich ausgegeben.

```
write_name ()
```


5.6.6 Wert einer Environment Variablen einlesen – getenv

Die Funktion *getenv* liest den value einer Environment Variablen ein.

```
<res_string> = getenv (<string>)
```

`<string>`	Name der Variablen
`<res_string>`	Value der Variablen

5.7 Funktionen auf die Datenstruktur

5.7.1 Voraussichtlichen Value berechnen – calculate_objval

Die Funktion *calculate_objval* berechnet den Value, wenn die Aktion ausgeführt würde.

```
<res_value> = calculate_objval (<string>, <params>)
```

`<string>`	Name der Aktion
`<params>`	Liste der nötigen Parameter

5.7.2 id-Typ für Effektobjekte errechnen – id

Errechnet den id-Typ von Effektobjekten.

```
<id_type> = id (<effect>,
                <integer>)
```

`<effect>`	Effektobjekt
`<integer>`	Sequenznummer

5.7.3 Objektvalue versorgen – set_objval

Versorgt den Value eines Objekts

```
set_objval (<object>,
            <string>,
            <aql_symbol>)
```

`<object>`	Objekt
`<string>`	Name des Objekt-Value
`<aql_symbol>`	Objekt-Value

5.7.4 Objektvalue versorgen – set_objval_in_user

Versorgt den Value eines Objekts, das einem bestimmten user gehört

```
set_objval_in_user (<user>
                    <object>,
                    <string>,
                    <aql_symbol>)
```

`<user>`	User
`<object>`	Objekt
`<string>`	Name des Objekt-Value
`<aql_symbol>`	Objekt-Value

5.7.5 Gruppen von Unterobjekten ausgeben – subobject

Gibt eine Gruppe von Unterobjekten aus

```
subobject (<object>)
```

`<object>`	Objekt
`<result>`	Gruppe

5.8 Funktionen zur Wahrung der Kompatibilität

5.8.1 Automatische Konturerstellung – scetch_makecont

Erzeugen einer Kontur durch Konturverfolger (nur aus Kompatibilitätsgründen, verwenden Sie **contour_tracing**).

```
scetch_makecont
```

5.8.2 Tabelle erzeugen – tab_one

Erzeugen einer Tabelle (nur aus Kompatibilitätsgründen, verwenden Sie tab_...).

```
tab_one <string1>,
        <string2>
       [,{{namestr enu_symboltype len_typ} ...},
        univ_ptr]
```

`<string1>`	Dateiname
`<string2>`	Name der Tabelle (optional)
`namestr`	Name der Tabellenspalte
`enu_symboltype`	Datentyp der Spalteneinträge
`len_typ`	Länge des Datentyps String
`univ_ptr`	Tabelleneinträge
`<result>`	Tabelle

5.8.3 Darstellung zurücksetzen – view_unzoomrectangle

Darstellung in Fenstern zurücksetzen.

```
view_unzoomrectangle {<integer1> <integer2>}
<integer1>   from_x from_y
<integer2>   to_x to_y
```

5.8.4 Darstellung vergrößern – view_zoomrectangle

Darstellung in Fenstern vergrößern.

```
view_zoomrectangle {<integer1> <integer2>}
<integer1>   from_x from_y
<integer2>   to_x to_y
```

Vordefinierte Funktionen

6 Datenstruktur – Erzeugung und Abfrage

Im Installationsverzeichnis finden Sie unter *#/aql.aql/electric_manual.aql* eine AQL-Prozedur, die eine ASCII-Datei *electric_manual* im aktuell eingestellten Verzeichnis erzeugt.

Die Datei *electric_manual* enthält eine vollständige AQL-Sprachbeschreibung, die aus System-Metainformationen gewonnen wird. Sie ist immer aktuell (versionsunabhängig) und vollständig. Sie ist folgendermaßen gegliedert:

- Liste der Typen (Values)
- Liste der Aufzählungs-Typen (Enumeration Types)
- Liste der Objekte
- Liste der Aktionen
- Liste der Attribute
- Liste der Events
- Liste der vordefinierten Funktionen
- Liste der vordefinierten Variablen

Die Datei hat den Vorteil, daß beim Programmiervorgang am Rechner die wichtigsten Informationen zu AQL verfügbar sind. Dazu öffnen Sie die Datei in einem Editorfenster. Mit den entsprechenden Editorfunktionen können Sie komfortabel nach Funktionen suchen. Durch Kopieren aus dieser Datei können Sie syntaxsichere Programme erstellen.

Die folgenden Beispiele zeigen, wie Sie mit der Datei *electric_manual* arbeiten.

Die mit der Datei *electric_manual* gewonnenen Informationen stellen die ungefilterte Gesamtinformation über die ObjectD-Applikation dar. Funktionen und Attribute, die in diesem Handbuch nicht näher beschrieben sind (aber in der Datei enthalten sind), stellen intern verwendete, aber nicht freigegebene Funktionalität dar und werden für Folgeversionen nicht garantiert.

1. Beispiel für Objekterzeugung

Datenstruktur – Erzeugung und Abfrage

Sie wollen eine Linie (line) zwischen zwei Punkten mit den Parametern *strichpunktiert* und *dick* (Strichdicke der Vollinie der Liniengruppe) erzeugen. Die Linie soll zwischen den Punkten *p1* und *p2* liegen, die schon im Programmsegment vorher erzeugt wurden.

Die Syntaxkonvention der Erzeugungsaktionen lautet:
`<Objekt>_<Erzeugungsart>`

Vorgehensweise

- Suchen Sie bei den Aktionen (DEFINED ACTIONS) mit dem Muster *line_* nach den Aktionen zur Linienerzeugung.
- Wählen Sie die gewünschte Erzeugungsart aus (alphabetische Sortierung)
- Erzeugen Sie eine Linie zwischen zwei Punkten mit der ersten Aktion, der Funktion *line_pointpoint*.

```
 line_pointpoint ( length,
                   styletype_rec,
                   endless,
                   point,
                   point )
   z                    : length          [opt]
   spec                 : styletype_rec   [opt]
   end                  : endless         [opt]
   p1                   : point
   p2                   : point
```

Hinter dem Funktionsnamen sind in Klammern alle **Parameter** in ihrer Typrepräsentation aufgeführt (hier *length*, *styletype_rec*, usw.). Darunter sind aufgelistet:

Namen der Parameter	(hier *z*, *spec*, usw.)
ihre Typen	(hier *length*, *styletype_rec*, usw.)
evtl. Parameterattribute	(hier *[opt]*)

Mögliche Parametertypen sind:

Parametertyp	*Kennzeichen*	*Bemerkungen*
Wahlparameter (optional voreingestellt)		Solche Parameter können bei Standardwert mit Leereintrag belegt werden, d.h. durch „,“ übersprungen werden.
Listenparameter	„star“	Darunter findet sich eingerückt der Listentyp. Die Parameterlisten werden über AQL-Gruppen abgebildet ({..}).
Parametersequenz (innerhalb einer Liste)	„sequence“	Die Parametersequenzen werden über AQL-Gruppen abgebildet ({..})
Alternativparameter	„alternative“	Darunter finden sich eingerückt die möglichen Typen aufgelistet. Parameter, die einen Datenverbundtyp fordern, werden über AQL-Gruppen abgebildet ({..}). Die zugehörige Parametersyntax finden Sie in der Online-Hilfe - „Values“

Es ist Ihnen nun bekannt, daß die gefragte Funktion *line_pointpoint* heißt und die ersten drei Parameter Wahlparameter sind, die Sie durch Leereinträge belegen können.

Da der Strichmodus strichpunktiert und die Strichdicke 0,5 mm sein soll, müssen Sie den **Wahlparameter *spec*** anders versorgen. Dieser ist vom Typ *styletype_rec*, der im Abschnitt DEFINED VALUES beschrieben ist.

```
styletype_rec  { <string:pattern>,
                 <real or integer:thickness>
                 [, <integer:dashes> ] }
                pattern:       ″solid″  ″dashed″  ″dashdotted″
                 ″dashdotdotted″  ″dotted″
                ″solid_invisiblepartdashed″  ″broken″  ″userdef″
                 thickness:     1 (=0.25), 2 (=0.35), 3 (=0.5)
                 default = 0.5
                 dashes:        on/off length segment lengthes
                                (0.1 mm pieces)
                 only if pattern == ″userdef″
                 { integer, ... }
```

- Erzeugen Sie eine strichpunktierte (dashdotted) Linie mit dem ersten „Wert“ des Datenverbundes (String) *dashdotted*.
- Die Strichdicke ist mit 3 zu belegen. Die Numerierung der Strichdicken erfolgt in der Reihenfolge innerhalb der Liniengruppe. Alternativ können Sie die Strichdicke absolut angeben (z.B. 1,2).
  Die Liniengruppe 0.5 gliedert sich demnach in 1 = 0.25, 2 = 0.35 und 3 = 0.5
  Die Liniengruppe 0.7 gliedert sich demnach in 1 = 0.35, 2 = 0.5 und 3 = 0.7 usw.

Ergebnis: `line_pointpoint (,{″dashdotted″,3}, , p1, p2)`

2. Beispiel für Objekterzeugung

Sie möchten eine Fläche (Objekt *plane*) aus den Linien l1, l2 und dem Bogen b1 erzeugen. Die Randelemente haben Sie schon im Programmsegment vorher erzeugt. Die Elemente begrenzen die zu erzeugende Fläche vollständig. Zusätzliche Merkmale sind:

- gefüllt mit Farbe des Index 10
- Schraffur im Kreuzmuster in einem Winkel von 45 Grad mit 2 mm Schraffurabstand
- Ausblenden, mit dem z-Wert 100

Vorgehensweise

- Suchen Sie in der Datei *electric_manual* nach Erzeugungsaktionen für *plane* mit dem Muster *plane_*.
- Wählen Sie die Fläche aus Einzelelementen. Die benötigte AQL-Funktion lautet also *plane_ofelements(...)*.

```
plane = plane_ofelements ( length,
                           color,
                           fillstyle_rec,
                           {line/circle/ellips
                            line/circle/ellips ...},
                           length,
                           angle )
    z                    : length
    fill_color           : color
    fill                 : fillstyle_rec
    poe                  : star
    poea                 : alternative
          l              : line
          c              : circle
          e              : ellips
    d                    : length
    a                    : angle
```

- Erzeugen Sie den **ersten Parameter** *z* als *length_absolute* implizit durch Eingabe des Wertes 100 in den Parameter (gleichbedeutend mit Eintrag *length_absolute(100)*)
- Tragen Sie für den **zweiten Parameter** *color* vom Typ *color* ein.

```
color  <string: colorname> or <integer: color number> or
      { <integer: red> <integer: green> <integer: blue> }
       when "" is specified, standard color is taken
```

- Belegen Sie den **dritten Parameter** *fill* als Verbundyp *planetype_rec* (Online-Hilfe - „Values“) mit dem String *hatch_cross* für „Kreuzschraffur“ und dem *boolean* „true“ für „Füllen mit Ausblenden“.

```
fillstyle_rec   { <string:hatch>, <boolean:hidden> }
                 hatch: ”hatch_no”, ”hatch_single”, ”hatch_cross”
```

- Übergeben Sie für den **vierten Parameter** als Typ *star* (Liste) eine AQL-Gruppe alternativ (Parametertyp *alternative*) mit den Objekten (common_type) *line* oder *circle*.

```
{l1, l2, b1}
```

- Erzeugen Sie den **fünften und sechsten Parameter** vom Typ *length* und *angle* wie für den ersten Parameter als *length_absolute* implizit durch Eingabe des Wertes 2 bzw. als *angle_absolute* durch Eingabe des Wertes 45.

Ergebnis

```
plane_ofelements (100,
                  10,
                  {”hatch_cross”,true},
                  {l1, l2, b1},
                  2,
                  45)
```

Beispiel für Attributabfrage

Sie wollen aus der Menge aller Objekte eines Modells alle unendlichen Linien löschen. Sie suchen also eine Möglichkeit, solche Linien zu identifizieren. Die Liste aller Linien eines Modells ist über *top.list_line* ansprechbar (siehe Kapitel „Zugriff über Namen und Datenstrukturdurchlauf“ auf Seite 4-25).

Vorgehensweise

- Suchen Sie in der Datei *electric_manual* nach *line_type* (Attribute von Linien), Sie finden den Typ `D2con_line_rec`.

```
           BASE CLASS       see also attributes and
                              methods of Bea_value
 ------------------------------------------------------------
       angle |real           | angle of the line
          ax |real           | equation:
             |               | ax * x + by * y + ct  = 0
             |               | (-1.0 <= ax <= 1.0)|
          by |real           | equation:
             |               | ax * x + by * y + ct  = 0
             |               | (-1.0 <= by <= 1.0)|
 constr_type |string         | ”not_limited”, ”is_limited”,
             |               | ”is_midline”
          ct |real           | equation :
             |               | ax * x + by * y + ct  = 0

         ...
         ...
```

- Durchsuchen Sie die angebotenen Attribute für *line_type* nach dem gesuchten Attribut (*constr_type*). Sie müssen also das Attribut *constr_type* auf den Inhalt *not_limited* prüfen.

Ergebnis:

```
del_gp = top.list_line where .constr_type = ”not_limited” delete
(del_gp)
```

Datenstruktur – Erzeugung und Abfrage

7 Beispielprogramme

7.1 Beispiel zu Datenstrukturen

Das folgende Beispielprogramm dient zum Durchlaufen der ObjectD-Datenstruktur (siehe auch weitere AQL-Programme im Verzeichnis *#/examples.aql*).

```
PROGRAMM #/aql/count_element.aql:
var count_total
var count_total_del
function counter_total ()
   count_total = 0
   count_total_del = 0
   for i in top.list_all  do
      count_total = count_total + 1
      if i.deleted then
           count_total_del = count_total_del + 1
      end
   end
end
function write_number (str, count , count_del)
   [str;-13] ':' [count;10]
   if count > 0 then
               [(100 * count / count_total );10] ' %'
               [count_del;15] [(count_del * 100 / count);10] '%'
   else '         -'
   end
   nl
end
function counter_element ( prim )
   var count
   var count_del
   count     = 0
```

```
   count_del = 0
   for i in top.[”list_” + prim.content]  do
      count = count + 1
      if i.deleted then
           count_del = count_del + 1
      end
   end
   write_number (prim.content, count , count_del)
end
nl
‘Characteristics of file :   ‘ [top.name]
nl
‘==========================================‘
nl
nl

counter_total ()

‘element         #    % of total      # deleted   % deleted  ‘ nl
‘---------------------------------------------------------- ‘ nl

for prim in application.list_prim sort by .content do
    counter_element ( prim )
end

‘---------------------------------------------------------- ‘ nl
nl
write_number (”total       ”, count_total , count_total_del)
nl

/* that’s all, folks */
```

AUSGABE:

```
Characteristics of file :   std
==========================================

element              #    % of total      # deleted   % deleted
-------------------------------------------------------------------
angle        :         2          3 %              2       100%
circle       :         3          5 %              0         0%
contour      :         0          -
coord        :         4          7 %              4       100%
dummy        :         0          -
ellips       :         0          -
group        :         0          -
layer        :         0          -
length       :        17         29 %             17       100%
line         :         6         10 %              0         0%
measure      :         0          -
nilprim      :         1          1 %              1       100%
number       :         0          -
plane        :         0          -
point        :        10         17 %              0         0%
posmeas      :         0          -
posttext     :         1          1 %              1       100%
prop         :         6         10 %              6       100%
rawval       :         1          1 %              1       100%
spline       :         0          -
string       :         4          7 %              4       100%
symbol       :         0          -
tab          :         0          -
tab_instance :         0          -
text         :         0          -
user         :         0          -
variable     :         0          -
vector       :         0          -
viewdata     :         1          1 %              0         0%
world        :         1          1 %              0         0%
-------------------------------------------------------------------

total        :        57        100 %             36        63%
```

7.2 Beispiel zur Dateibearbeitung

Das zweite Beispielprogramm zeigt das Zusammenspiel der Funktionen *popup_filename*, *file_access*, *file*, *open*, *close*, *get* und *parse_...*

```
// this program reads an existing ASCII File containing
// calculation expressions (like 1 + 5 * 6) and write
// the result of calculation (31) back to a binary file
// comment lines (//..) are ignored
input_filename = ″expression_file.txt″
// test for existance
if not existf (input_filename) then
   display_error (″% input file does not exist %″)
   return()
end
// test for access
if not file_access (input_filename, ″r″) then
   display_error (″% could not read the input file %″)
   return()
end
// get output_filename
output_filename = ″″
while output_filename = ″″ do
        output_filename   =   popup_filename   (″binary   output   file-
name?″,″*.bin″,″″,″result_file.bin″,″o″)
end
// open error file
file ″/usr/tmp/errors″ error_output
// open input file
file input_filename input
// open binary output file
o_file = bin_open (output_filename, ″output″)
nl nl nl
exp = ″″
allowed_types = {″integer″,″real″}
while valid (exp) do
   parse_comment () // skip over comments
   exp = parse_line ()
   result = eval (exp)
   result_type = type(result)
   if result_type in allowed_types then
      [exp] ‘ ===> ‘[result] nl
```

```
      bin_write (o_file, result)
   else
         display_error  (”%Error  in  input  file:%”+exp+”%invalid  type
’”+result_type+”’%”)
   end
end
// close and reset files
close ()
bin_close (o_file)
```

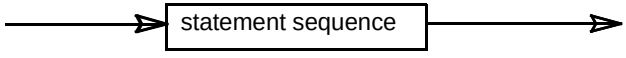
7.3 Beispiel für Zeichensuche in einer Datei

Das folgende Beispielprogramm dient der Suche nach einem bestimmten Zeichenmuster in einer Datei mit Hilfe der Funktionen *parse_keyword*, *line* und *get*.

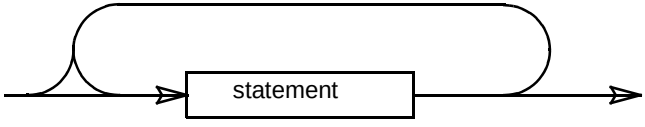
```
//============================================================
// aql programm to parse inputfile for a certain searchstring
//============================================================
// preset of variables
count = 0
g = ""
// open file for input
file "textfile" input
// ask user for search string
mykey = read_string ("Search string: ")
// traverse file till EOF
while valid (g) do
     // ask for keyword mykey
     h = parse_keyword (mykey)
     if h then
        // if found, print line number and add 1 to counter
        ' keyword ' [mykey] ' found in line: ' [line()] nl
        count = count + 1
     else
        // if not skip over
        g = get ()
     end
end  // of while loop
// close files and resume
close ()
'I have found '[count]' entities of search string '[mykey] nl
```

8 Syntaxdiagramme

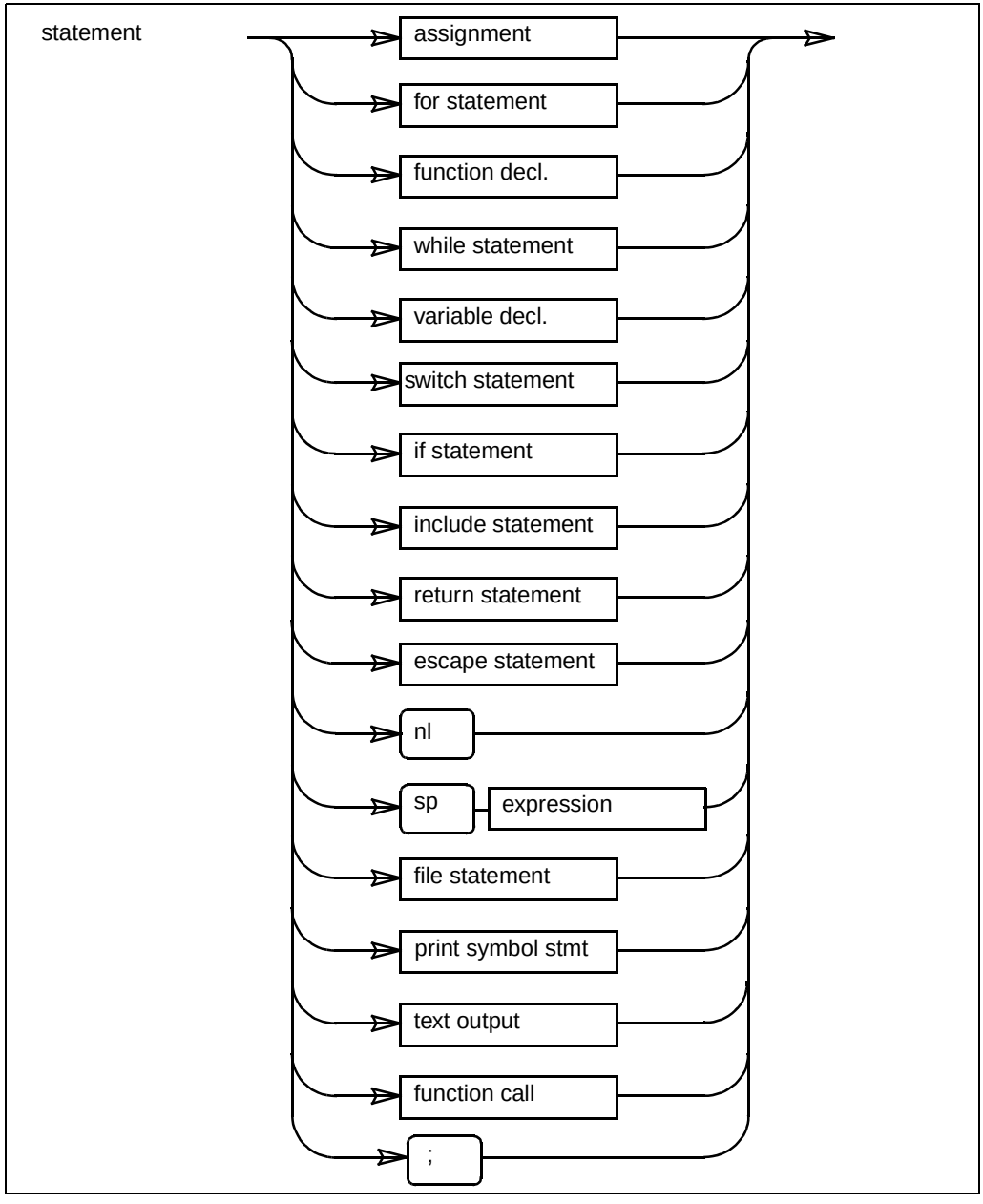
AQL program



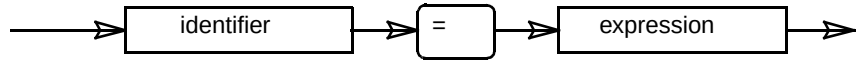
statement sequence



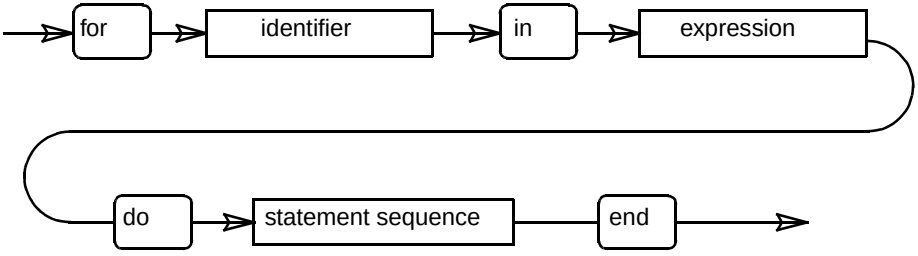
Syntaxdiagramme



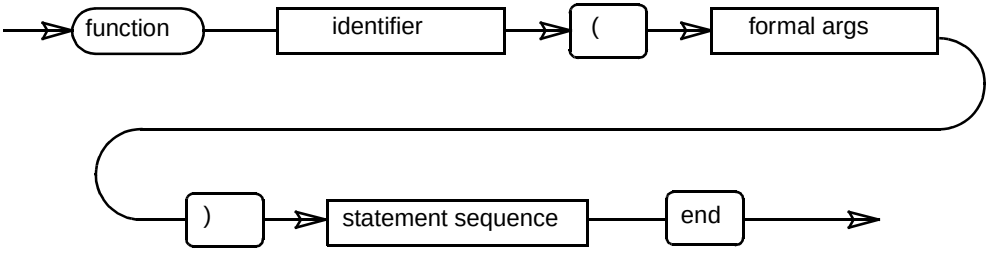
assignment



for statement

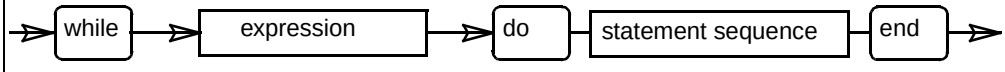


function declaration

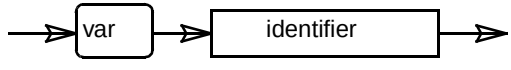


Syntaxdiagramme

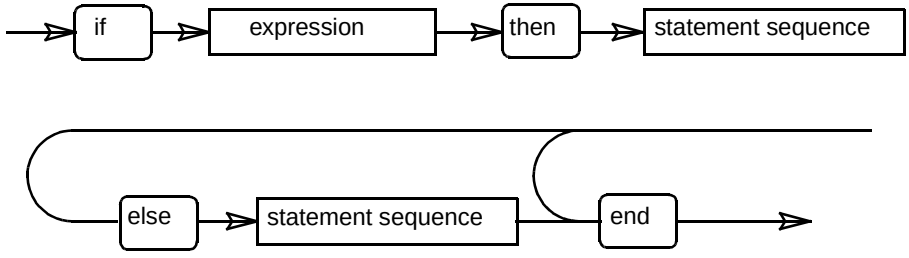
while statement



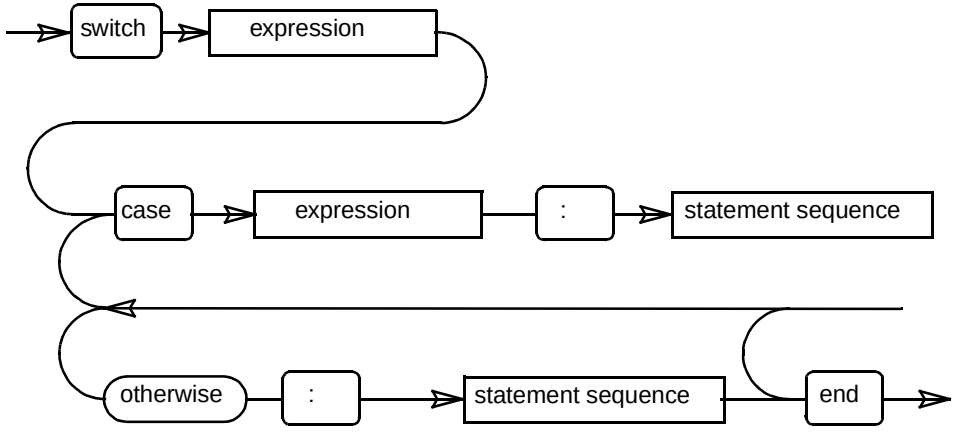
variable decl



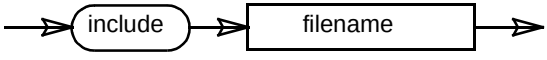
if statement



switch statement

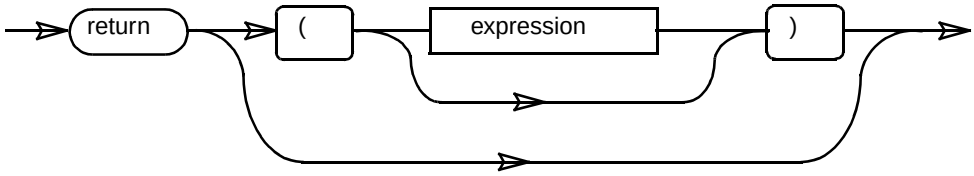


include statement

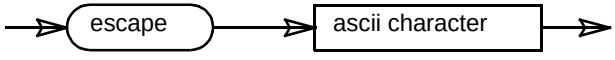


Syntaxdiagramme

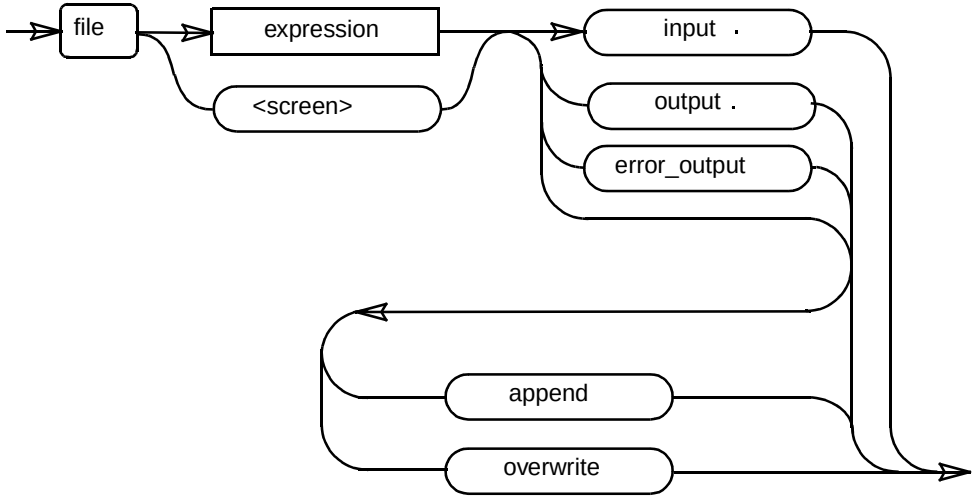
return statement



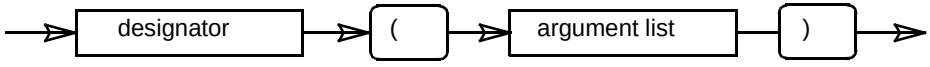
escape statement



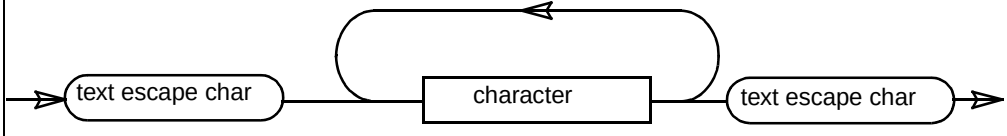
file statement



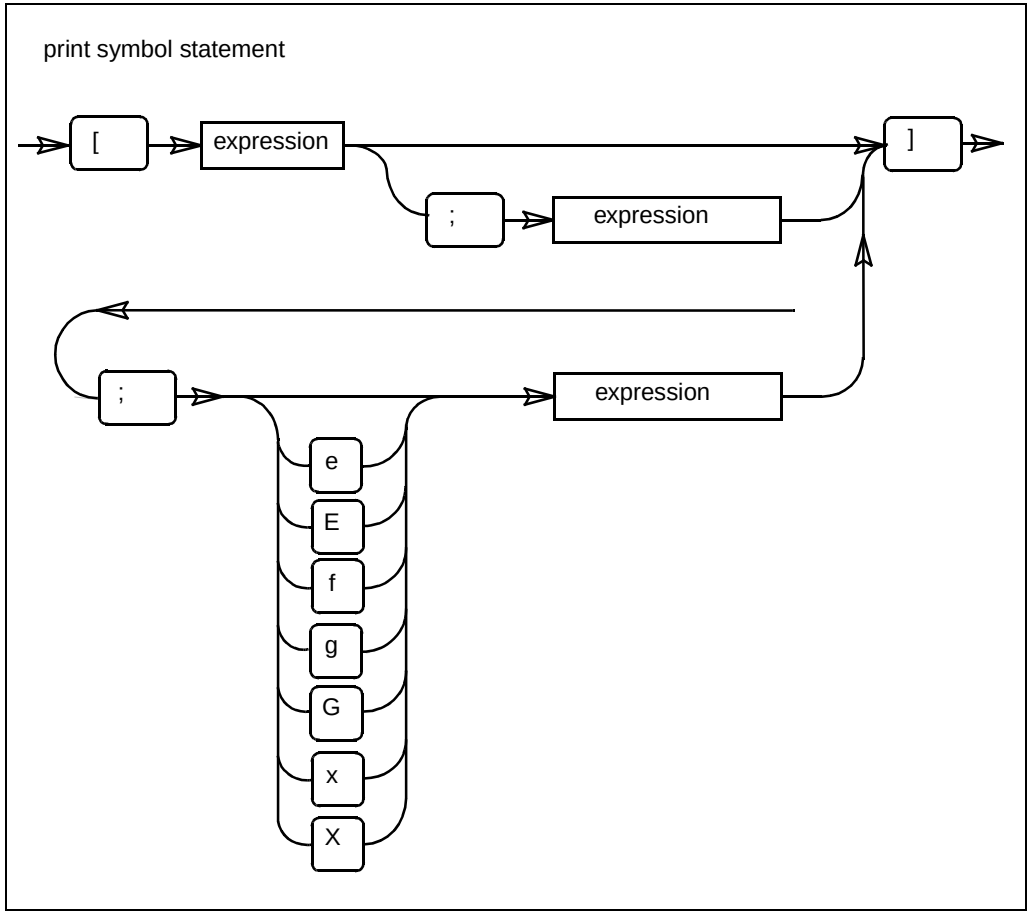
function call



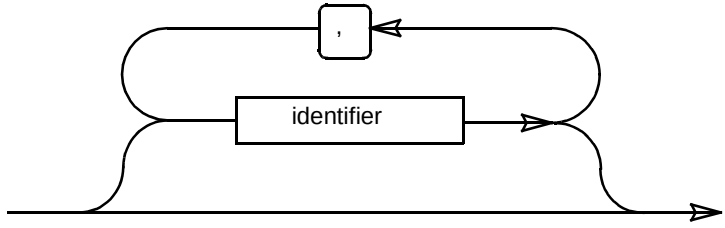
text output



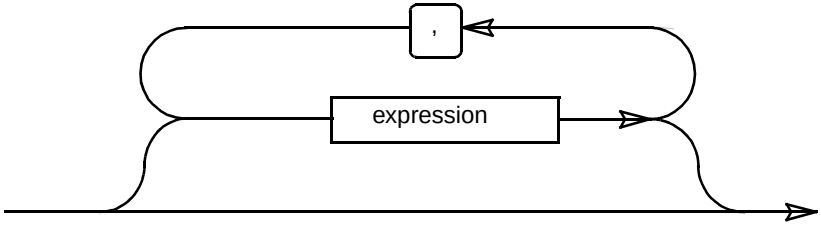
Syntaxdiagramme



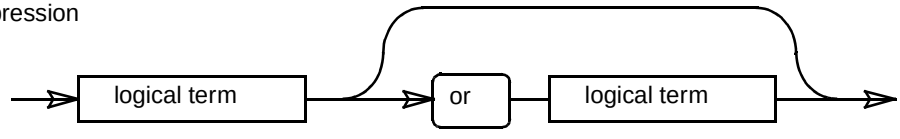
formal args



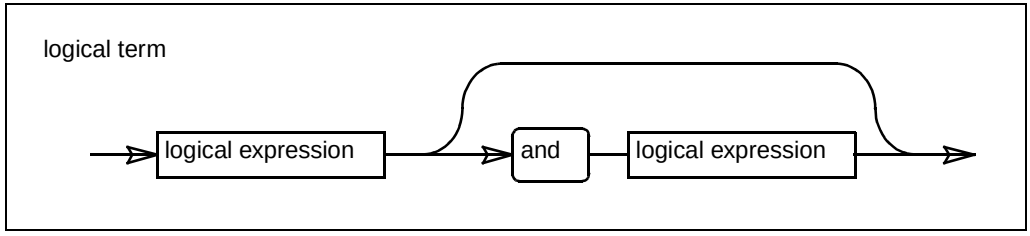
argument list

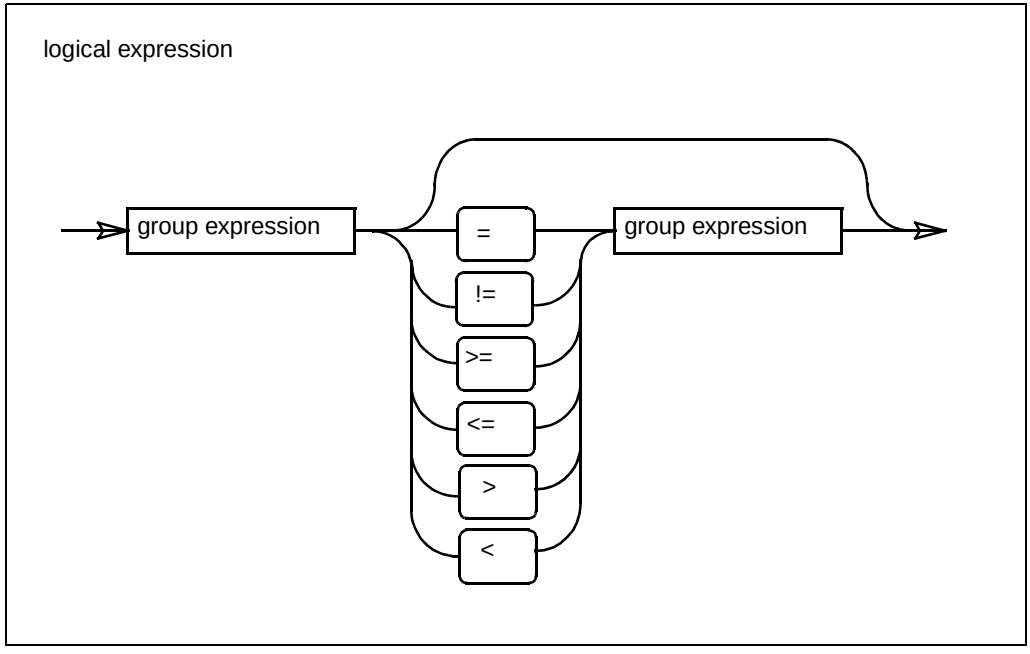


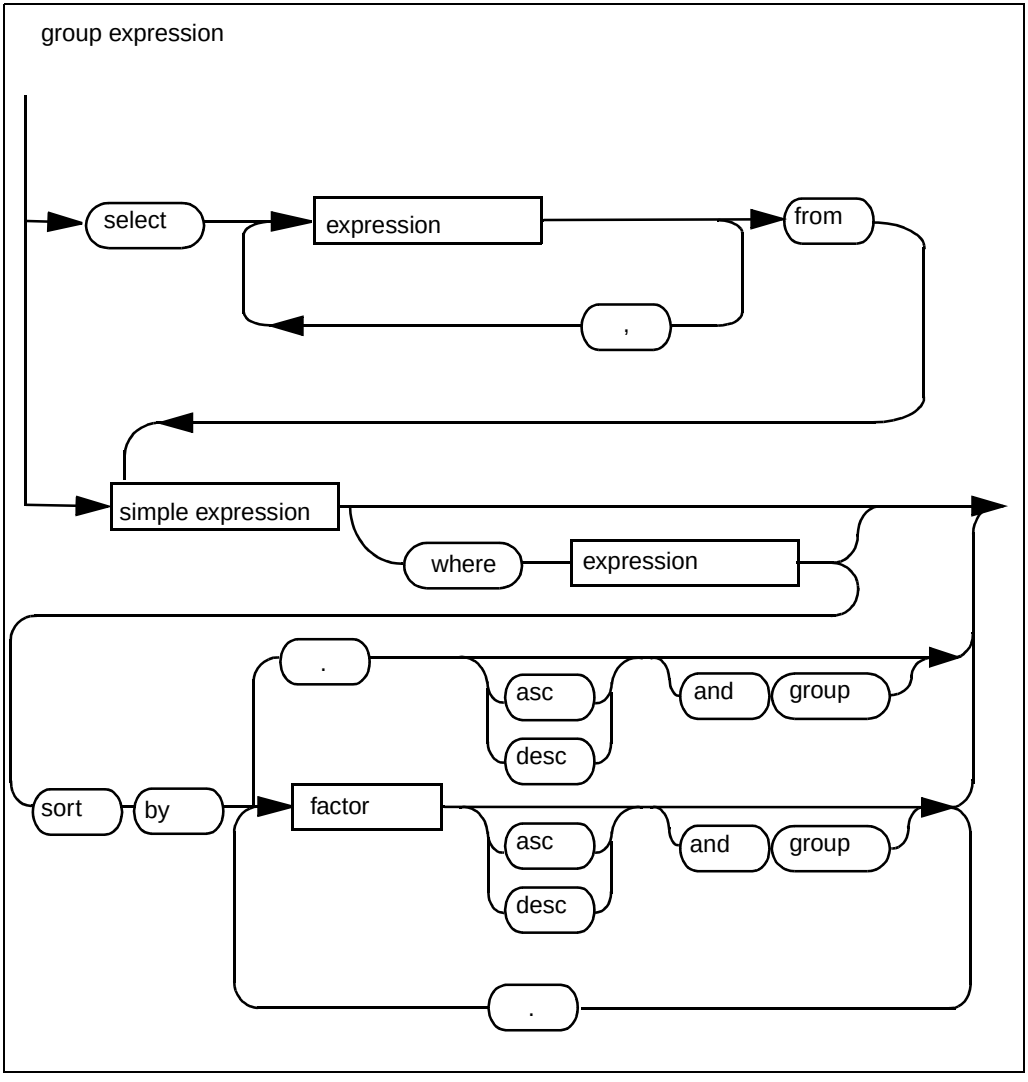
expression



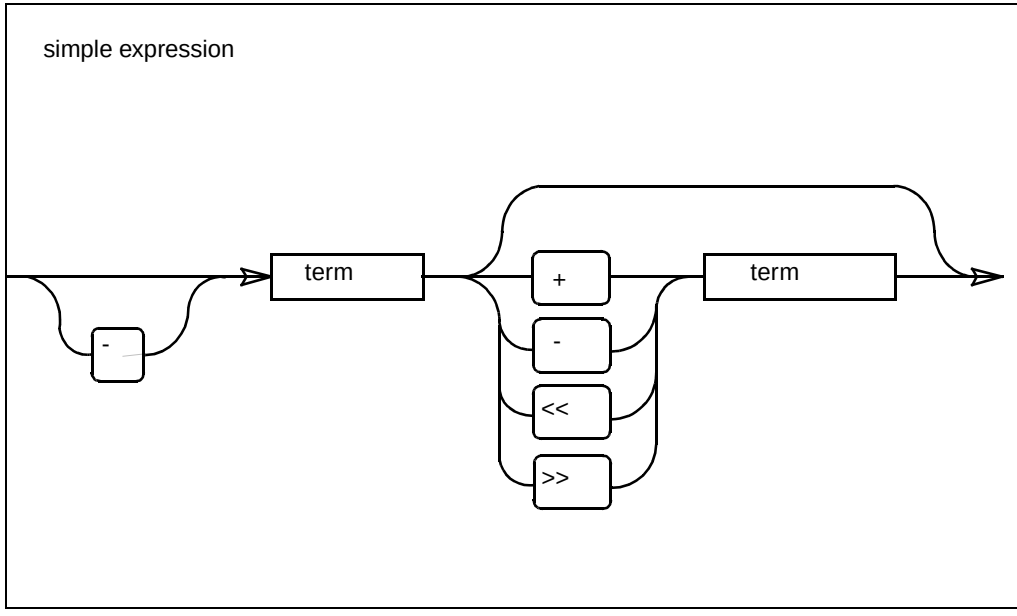
Syntaxdiagramme

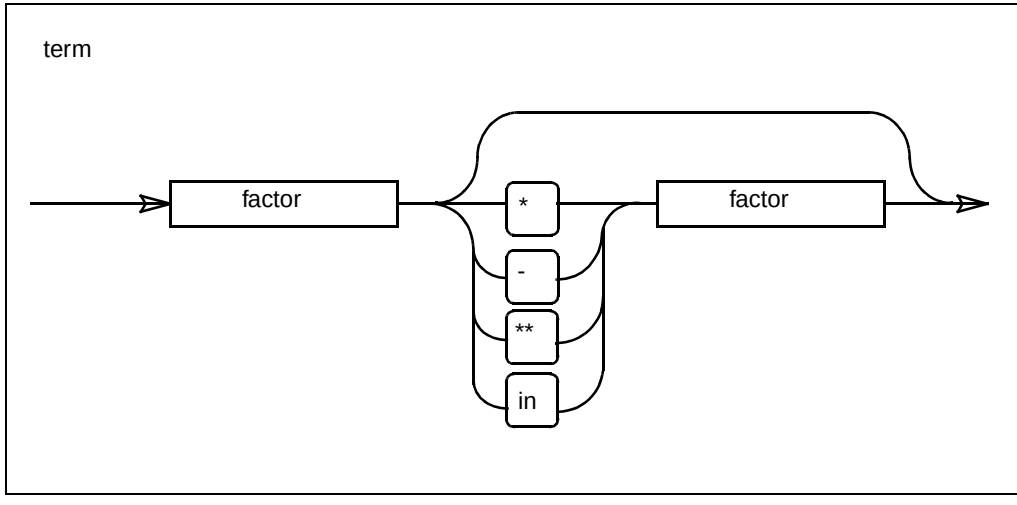


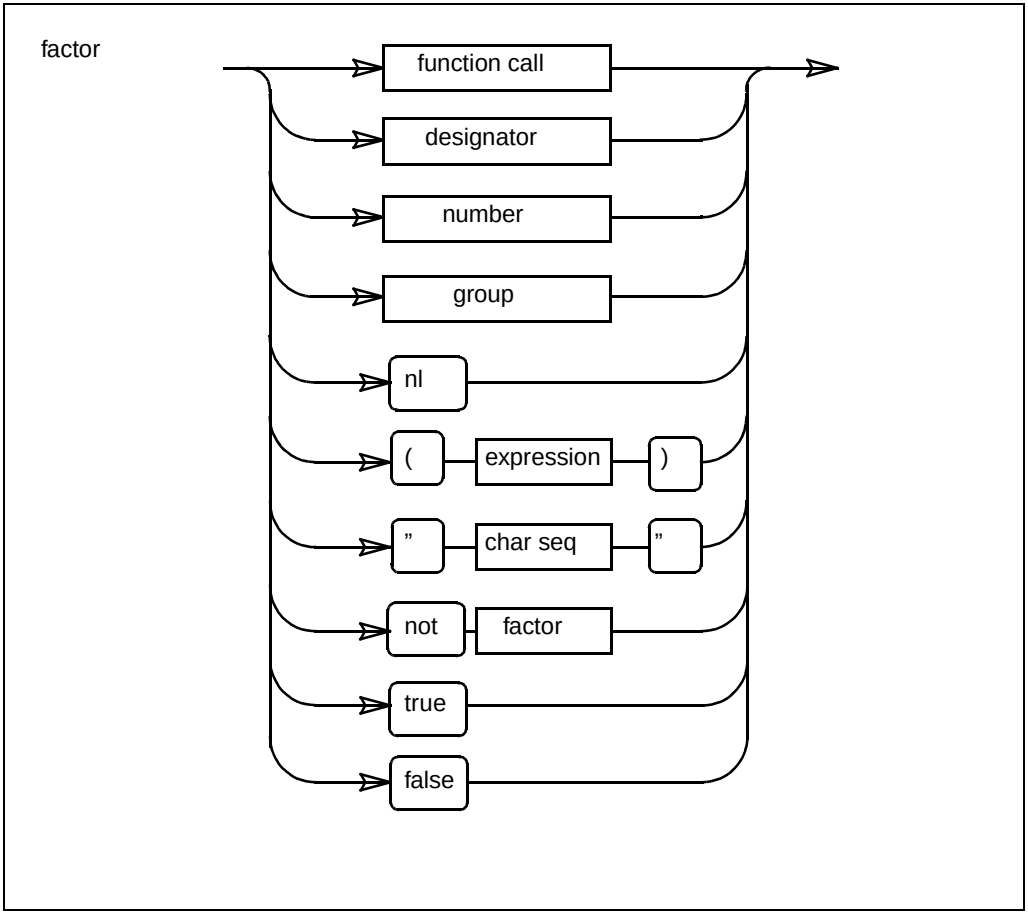



group expression
select
expression
from
,
simple expression
where
expression
.
asc
desc
and
group
sort
by
factor
asc
desc
and
group
.

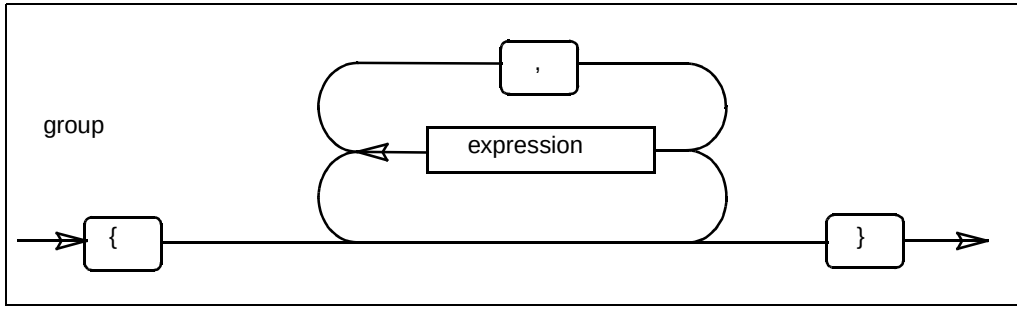
Syntaxdiagramme

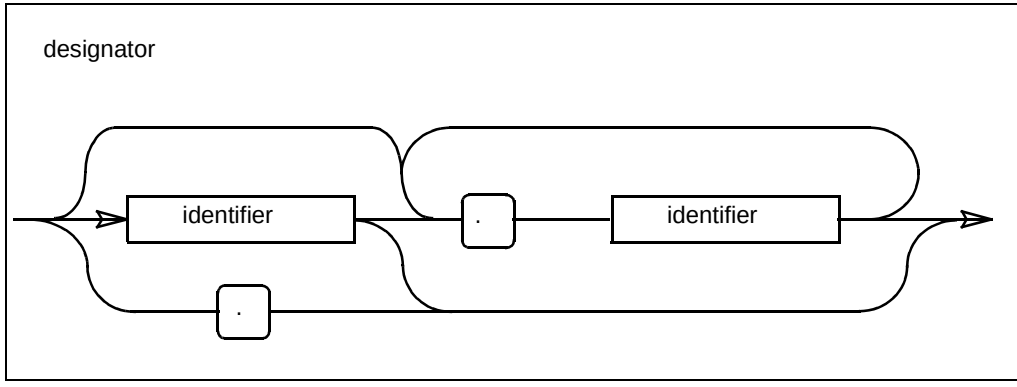


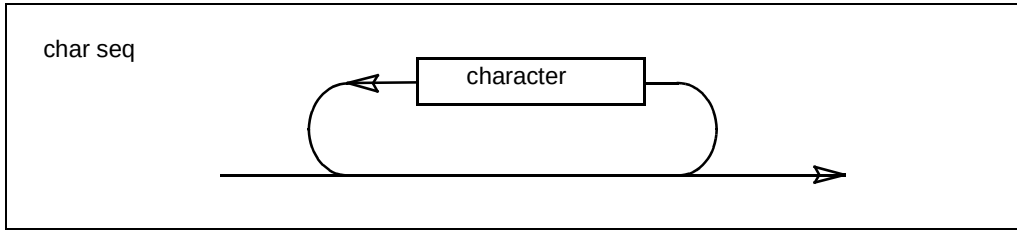



factor
function call
designator
number
group
nl
(
expression
)
”
char seq
”
not
factor
true
false

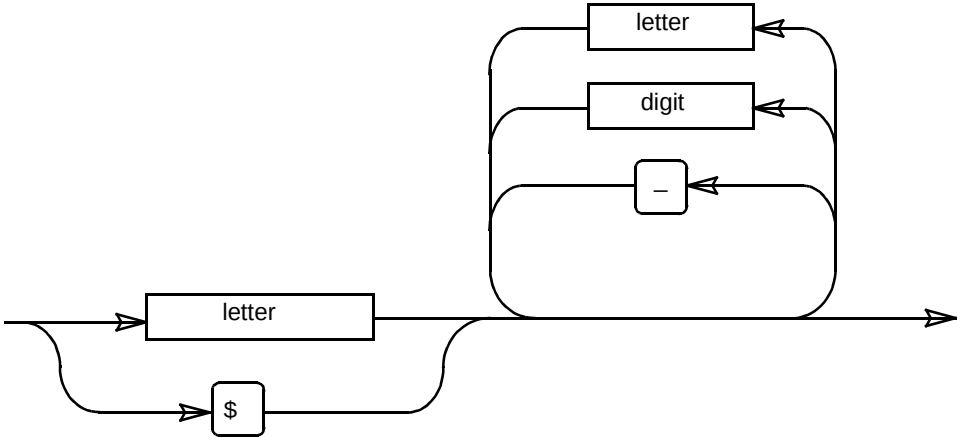
Syntaxdiagramme



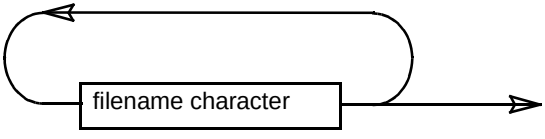




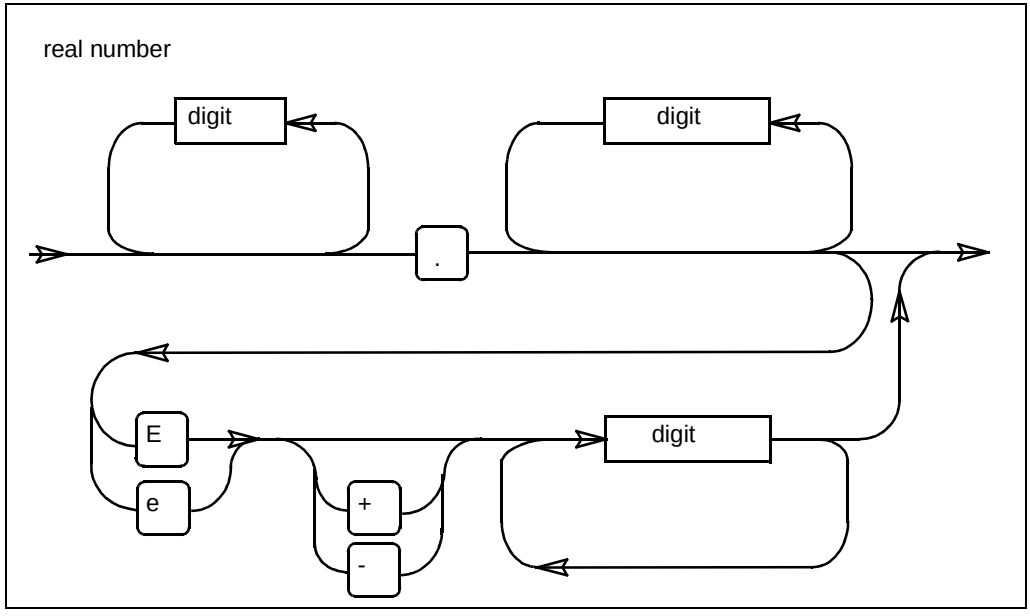
identifier



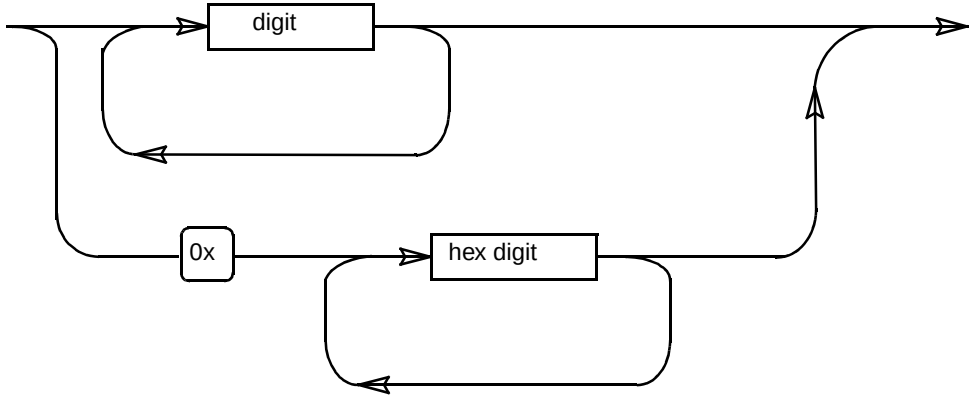
filename



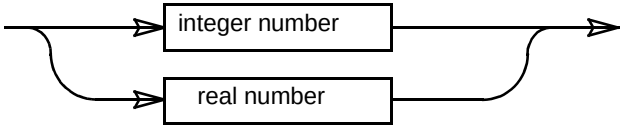
Syntaxdiagramme



integer number



number



Syntaxdiagramme

Index

A

B

C

D

E

F

G

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z