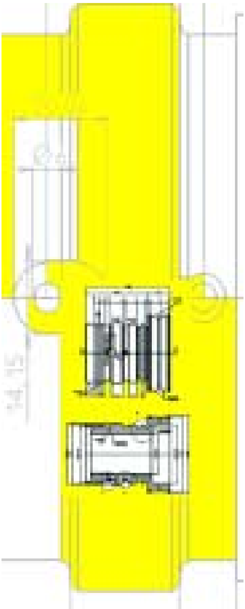
EUKLID Design

euklid
CAD CAM



Open Architecture AQL

2002

October 2001

How to contuct us:

In Germany:

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 Switzerland:

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

You have questions, recomments to...

... products

... or other subjects of the technical
information system?

For further information please call our offices.

AcrobatReader™ is a registered trademark of Adobe Inc.
I-DEAS™ is a registered trademark of SDRC Inc.
PARASOLID™ is a registered trademark of EDS Corp.
SolidWorks® is a registered trademark of SolidWorks Corp.

Contents

1 General Introduction

The programming language AQL affords the user the entire range of objects and actions integrated in the ObjectD CAD system. It also provides control and I/O functions. AQL programs can thus be used to analyse, generate and modify the contents of ObjectD models, plus a wide range of other options.

AQL can also be used as an implementation language to incorporate new functions.

Structure of the Manual Series

This user documentation consists of the following volumes:

ObjectD, Exploring

> This manual contains an overview of the functionality and the principles of ObjectD and an introductory example to learn the handle of the basic functions.

ObjectD, Open Architecture (AQL) — this manual.

> This documentation contains the description of the programming language which can be used to analyze, generate and modify the contents of ObjectD models.

Use of this Manual

This manual describes the structure, generation and execution of AQL programs. It is intended for users already familiar with ObjectD. Those with a working knowledge of a programming language such as C or Pascal will find this advantageous in learning AQL.

Training courses are also available on various aspects of ObjectD. More detailed information may be obtained from the *strässle GmbH* Sales Office.

General Introduction

Structure of this Manual

The language description is structured as follows:

The chapter "Introduction to AQL" contains basic information about the characteristics of AQL, about program structure and about program execution.

The formatting of outputs, the variables and functions and the processing routines are described in the chapter "Programming Language AQL".

The model oriented aspects of AQL are presented in the chapter "AQL as Command Language". It describes how AQL is used to generate, manipulate and query CAD models. Access to ObjectD data is also described.

The next few chapters list, explain and demonstrate parameters, attributes and functions.

Examples of data structures, file processing and character search operations in a file are included in the chapter "Example Program".

The chapter "Syntax Diagrams" provides an overview of AQL syntax.

Modes of Illustration

The following modes of illustration were chosen for the sake of clarity:

Instructions for the user are identified by a ☞ character.

Invariable keywords are emphasized in **bold type**, e.g. **return**.

Source code is given in `typewriter script`.

Parts of the source code which are to be substituted by the user for literal expressions are enclosed in <angled brackets>, e.g. **while** `<expression>` **do**.

Function arguments whose task is optional are shown in [square brackets].

Examples are marked by the heading **"Example"** and are indented.

Time-saving tips or exceptions are separated from ordinary text by the mark left from this text.

Descriptions of special cases such as disturbances, error sources etc. are highlighted by the mark left from this text.

2 Introduction to AQL

The structure, generation and interactive execution of AQL programs is illustrated in this chapter by a simple example, which outputs the character string "Hello World!" (See "Program Example" on page 2-6.).

2.1 Characteristics of AQL

AQL (Attribute Query Language) is a precompiled interpreted language, providing both power and flexibility. AQL is object oriented. It is the procedure language of ObjectD enhanced with CREATE, EDIT and DELETE functions.

AQL enables users to adapt CAD applications to their own requirements. Processing of a ObjectD model is carried out using ObjectD functions.

AQL features macros, automation of design variants, incorporation of external programs and animation, enabling it to be used for a wide variety of purposes.

The AQL language framework is similar to other widely-used programming languages. **Syntax** was influenced by C and program structure by Pascal. The interpreted languages Lisp and Basic and the query language SQL were also drawn upon.

AQL includes
- all the usual **control structures** such as "if...then...else" for a conditional branch, "switch...case..." for case differentiation and "for.." for loops
- all the usual **operators** and **functions** such as +, -, /, *, **, min( ), sin( ), cos( ), tan()
- dynamic **data types** such as "real", "integer", "boolean", "string", "date", "group"
- automatic **variables**
- powerful model **query functions**.
- full integration in the user interface and the system architecture by means of UDAs

Introduction to AQL

It is not necessary to link or compile AQL programs.

In contrast to conventional programming languages, however, AQL is able to work with objects of a ObjectD model which are not managed by the AQL language processor. To preserve the independence of language from the objects, AQL recognizes no type declaration. Variables and function arguments are declared by name only.

As the system outputs a corresponding error message for any invalid combination of types, it is not necessary to input type conflicts syntactically.

AQL may be regarded generally as a language which knows only one data type, namely the **AQL Symbol**. This symbol may represent any objects, either one of the standard integrated types "real", "integer", "string", "date", "group" (groups of symbols) or an object from a ObjectD model with these data types.

The AQL interpreter may also be used to **query a ObjectD model** or as a **command language**.

- The query facility exploits the fact that all ObjectD objects possess attributes. These may be used in a selection expression. These are in turn used to input those objects from any given group, which satisfy a specific condition (i.e. a "where" clause).
- Macros which extend or change the ObjectD model may be implemented in the form of AQL programs.

When an **AQL program is called**, the readable code is first interpreted and stored in a tree structure. This tree structure represents all statements and expressions of the AQL program. On execution, only this tree structure is referred to; the source code is ignored.

This type of precompilation method improves speed of execution considerably. The precompling effort of approx. 700 statements/second*mips is negligible when compared to the speed of execution of approx. 3000 statements/second*mips.

In order to protect AQL programs from unwanted alteration, programs can be executed in **encrypted** form.

2.2 Program Structure

All AQL programs are held in ASCII files and may be processed with any normal text editor. An AQL program consists of the following elements**:**

- statements
- declarations
- function definitions

These elements may be in any sequence. It is both advisable for the sake of efficiency to execute these elements in the sequence in which they are used. There is one prerequisite, namely that a function defined before it is used for the first time.

An AQL program must contain at least one instruction. The presence of declarations and function definitions is optional. The contents of the source file are regarded as a maon program in which all statements are executed in sequence.

2.2.1 Separation of Statements

The syntax of AQL is so structured that a separator between statements is not required. The legibility of AQL programs may be enhanced and made more familiar to usC and Pascal programmers, however, by using thesemicolon; as a separator.

2.2.2 Comments

Comments are treated during syntax analysis in the same way as a statement, and may therefore be inserted any where between statements. A comment is oriented to C language and is delimited by the comment start **/*** and the comment end ***/**. Characters lying between these are ignored together with the delimiters themselves. As in C++, the string **//** is interpreted as a comment up to line end. All following characters up to the end of the line are ignored. Comments may not be nested.

2.2.3 Insertion of Other Files – include

The *include* statement may be used to insert the contents of another file at its current position. The file name must not include either identifiers or expressions.

```
include <filename>
```

- When file names are assigned, the rules of the operating system must be taken into account.
- The environment variables of the operating system are supported by AQL via the "getenv" function.
- File names beginning with "!" are interpreted relative to the calling AQL program, e.g.
  `include !my_include`
  The file *my_include* will be seached for in the directory containing the AQL program.
  Files used in include statements must be available during program execution.

2.2.4 Encrypted AQL programs – encrypt

AQL programs can be encrypted using the AQL *encrypt* function, to protect them against unauthorized changes and unlicensed use.

Encrypted programs can be run without an AQL development license.

An AQL development license is needed to use the "encrypt" function.

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

`<ascii_source_name>`	`:string`	File name of the program which must be encrypted
`<crypted_name>`	`:string`	File name of the encrypted program
`<my_company_name>`	`:string`	Name of the company which wrote the AQL program
`<my_key>`	`:string`	Personal keyword

Example

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

2.2.5 Encrypted AQL Programs – key

Encrypted programs begin with the *key* statement, followed by the keyword. The rest of the program is not readable. If an encrypted program is modified by the user, it is no longer executable. In this case an error message is output.

2.2.6 Program Example

For the message "Hello World!" one statement for the output of alphabetical character strings is required, as detailed below.

Example: Hello_World.aql

PROGRAM:

```
‘Hello World!‘
```

OUTPUT:

```
Hello World!
```

2.3 Program Execution

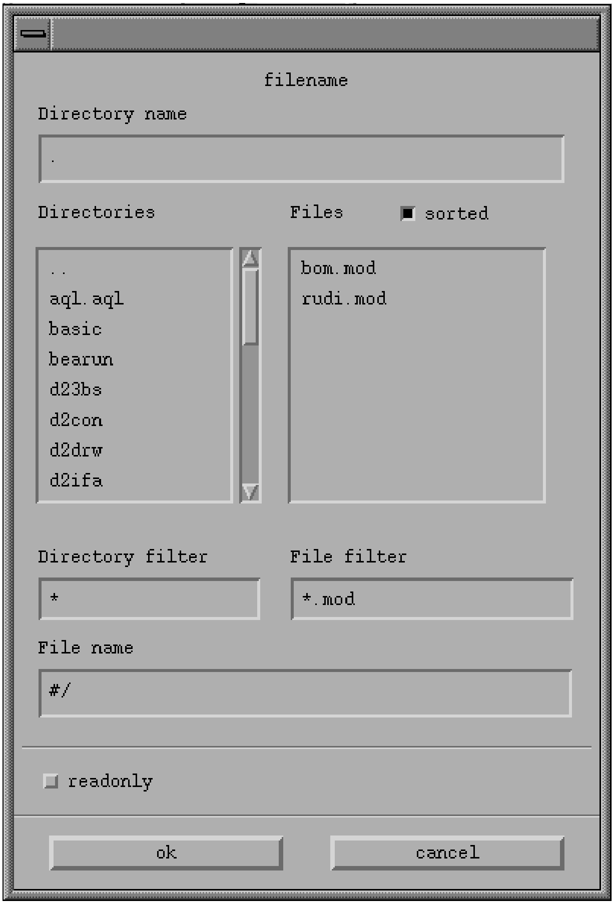
There is a number of ways of starting AQL programs:

- interactively
- in batch mode
- via UDAs
- via another AQL program
- with options for special functions

2.3.1 Interactive Execution

To start AQL interactively

- select AQL icon in second menu
- select AQL program via file selection menu displayed.



filename

Directory name: .

Directories:
..
aql.aql
basic
bearun
d23bs
d2con
d2drw
d2ifa

Files ☒ sorted:
bom.mod
rudi.mod

Directory filter: *

File filter: *.mod

File name: #/

☐ readonly

AQL programs using standard I/O, i.e. I/O functions without previous *file* declaration, use the window in which ObjectD was started.

The program name *f1.aql* is recommended during development and testing. These programs can be started with the function keys <F1> to <F4> or combinations of these keys with the <SHIFT> key.

Readers wishing to follow through the first few examples may skip the rest of this chapter if they wish.

2.3.2 Batch Mode

If ObjectD is merely to execute an AQL program without user intervention, the program may be started in Batch mode:

objectd `-batch <aql-filename> <modelname>`

The option *-batch* followed by the name of the AQL file causes ObjectD to execute the program and afterwards to terminate without opening a graphics window. This is an elegant way to carry out a format conversion or automatic plot of a model within a shell procedure.

2.3.3 Interface between User Elements and AQL Programs

An AQL program is interfaced to a user-defined action (UDA) if this was specified when the UDA was written.

Example: The UDA *myuda.uda* is interfaced with the AQL program *myuda.aql*.

Skeleton functions, which must then be filled with semantics by the developer, are pregenerated according to the specified attributes.

The functions are as follows:
pre_action
    Executed before carrying out "execute" and parameter functions
post_action
    Executed after carrying out "execute" and parameter functions

execute
Actual evaluation function
pre_par_<parname>
Executed before parameter input
post_par_<parname>
Executed after parameter input
select_par_<parname>
Selector function for parameters
traverse_values_par_<parname>
Returns the possible states of the selectors
draft_par_<parname>
Returns a draft value for a parameter
move_cursor_par_<parname>
Cursor routine when the mouse is moved with no buttons pressed
drag_cursor_par_<parname>
Cursor routine when the mouse is moved with the left button pressed
show_par_<parname>
Visualization method for help information about parameters, e.g. length
init_par_<parname>
Causes a default value to be stored for a parameter
get_prompt_par_<parname>
Parameter-internal prompting function
check_par_<parname>
Interval test for parameter values
suppress_property
Function for suppressing properties of the result and effects
res_fill_suppress_props
Fills in default values for suppressed result properties

2.3.4 Execution of AQL Programs

An AQL program is called by another AQL program with the AQL function "*aql*"**,** in which the name of the AQL program is the first argument:

```
aql(<filename>)
```

Further arguments are possible.

Example:

```
aql(”!my_subprog”, ”Arg1”, ”Arg2”)
```

The function outputs the following return codes:

```
   0   no error occurred
  -1   run time error
  -2   syntax error
  -3   the file specified with the AQL program cannot be opened
   1   error in external functions.
```

- When file names are assigned, the rules of the operating system must be taken into account.
- The environment variables of the operating system are supported by AQL via the "getenv" function.
- File names beginning with "!" are interpreted relative to the calling AQL program.

If arguments are to be passed on from the called procedure, two predefined symbols may be used:

`aql_argument_list`	group of parameter strings
`aql_arg_<n>`	nth argument of parameter list; aql_arg_0 contains the file name.

Example 1

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

Example 2

```
‘Subroutine‘[aql_arg_0]‘started‘ nl
```

2.3.5 Special Functions in Connection with Options

The following options can be used when starting ObjectD:

`-access <aqlnam>`	An AQL routine is called before and after every store or load operation, and whenever a new model is called (please also refer to the enclosed example in the installation directory *#/aql.aql/file_aql.aql*).The system looks for this routine in the following directories (in this order):
	in the working directory
	`~/user_data`
	`#/aql.aql`

`-start <aqlname>`	first AQL file, then interactive
`-stop <aqlname>`	Sequence before terminating the session

ObjectD can write **AQL logs**. These are useful to programmers in cases where specific AQL ObjectD functions are unclear.

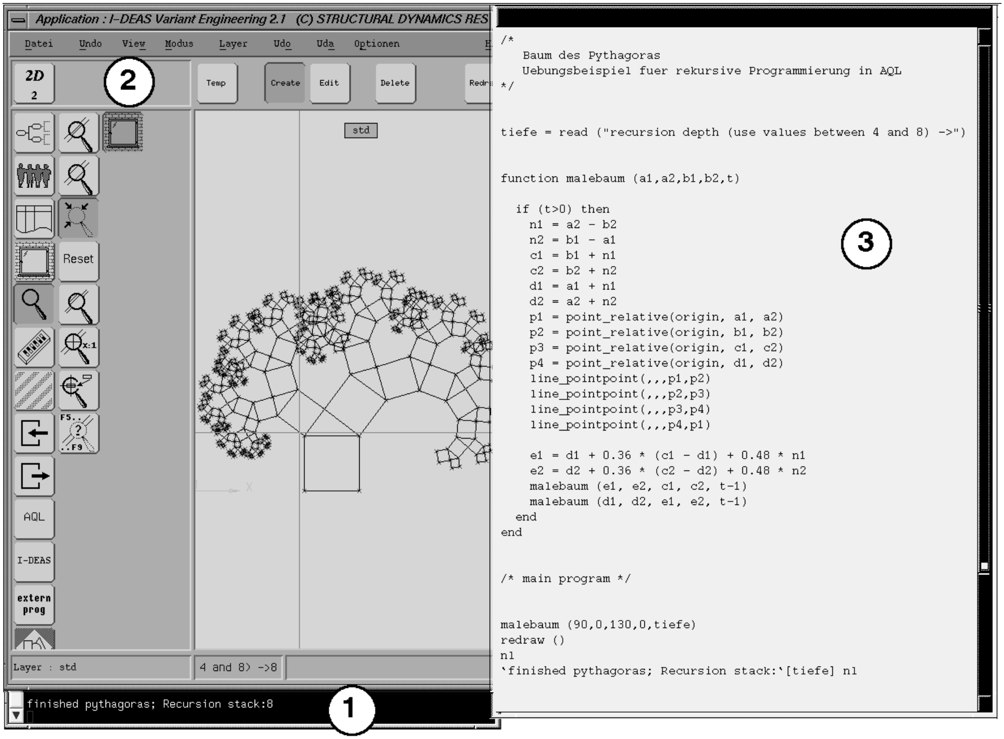
AQL logs are written when the following start options are used:

`-protocol`	model logging	log of model logic, i.e. object identification by name

It is not explicitly guaranteed that all logs are executable, however these logs help reveal data structure relationships and are primarily intended as an aid to AQL programming. The logs are stored in the current working directory.

2.3.6 AQL programming environment

The following screen dump shows a typical AQL programming environment.



```
finished pythagoras; Recursion stack:8
```

```
/*
   Baum des Pythagoras
   Uebungsbeispiel fuer rekursive Programmierung in AQL
*/


tiefe = read ("recursion depth (use values between 4 and 8) ->")


function malebaum (a1,a2,b1,b2,t)

  if (t>0) then
    n1 = a2 - b2
    n2 = b1 - a1
    c1 = b1 + n1
    c2 = b2 + n2
    d1 = a1 + n1
    d2 = a2 + n2
    p1 = point_relative(origin, a1, a2)
    p2 = point_relative(origin, b1, b2)
    p3 = point_relative(origin, c1, c2)
    p4 = point_relative(origin, d1, d2)
    line_pointpoint(,,,p1,p2)
    line_pointpoint(,,,p2,p3)
    line_pointpoint(,,,p3,p4)
    line_pointpoint(,,,p4,p1)

    e1 = d1 + 0.36 * (c1 - d1) + 0.48 * n1
    e2 = d2 + 0.36 * (c2 - d2) + 0.48 * n2
    malebaum (e1, e2, c1, c2, t-1)
    malebaum (d1, d2, e1, e2, t-1)
  end
end


/* main program */


malebaum (90,0,130,0,tiefe)
redraw ()
nl
`finished pythagoras; Recursion stack:`[tiefe] nl
```

① Start window of ObjectD, also used for AQL output (e.g. error messages)
② Execution window of ObjectD
③ Editor window with AQL program currently being processed

2.4 Termination of AQL Programs

Execution of an AQL program can be terminated by depressing <DELETE> (signal SIGKILL) in the start window.

3 Programming Language AQL

This chapter describes output formatting and introduces the main structure elements of AQL. The handling of program errors is also described.

3.1 Formatting of Alpha Outputs

Alpha output, which is formatted as described below, is always made to the current output stream, as declared with the *file* statement. If no output stream has been declared, output is made to the window in which ObjectD was started (*stdout*).

3.1.1 Output of Character Strings – ‘

The character string to be output is enclosed in reversed quotes. Tabulator characters may be included. These are output unchanged and then interpreted by output device concerned, such as display terminal or printer. The enclosed character string may extend over more than one line. The end of line (*newline*) characters are treated in the same way as tabulator characters.

3.1.2 Redefinition of an Output Delimiter – escape

In this statement, which is used to redefine the delimiter for the output of alpha character strings, the argument is the new delimiter. This is necessary when the text in the reversed quotes is also to be output and is not protected in the AQL program by a \.

```
escape <char>

string = ”protected \, quote”)
```

3.1.3 Invalidation of Characters – \

Characters which can be interpreted by AQL programs as program control characters may be used in character strings if they are preceded by a backslash \.

Example 1

```
\new_line(10)
```

The ASCII code for ”*new_line(10)*” is now invalidated for the AQL program when it is used within a character string.

3.1.4 Output of End of Line – nl

The *nl* statement generates an end of line in the current output stream. This saves the AQL program from having to generate an end of line with a true end of line enclosed in reversed quotes. The nl statement has no arguments.

```
nl
```

3.1.5 Output of Spaces – sp

The *sp* (space) statement is used when outputting character strings of variable length to fill the gaps with a quantity of blanks, calculated by AQL. The argument is the number of spaces to be generated.

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

Example 2: Hello_World_1.aql

PROGRAM:

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

OUTPUT:

```
       Hello World it‘s great
```

3.1.6 Output of Contents of Variables – [...]

As with the output of alpha character strings, the output statement for the contents of variables or expressions is generated by enclosing them in square brackets. The statement can take three forms:

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

The first argument is any expression, the other two expressions with an integer for a result. The mechanism for the minimum field length (<*min_field_width*>) and the control of the output with,<*precision*> were taken from C language.

3.1.6.1 Output in Standard Form

If only one argument is specified, the expression is converted to readable form as precisely as possible. This is done differently acording to the type of expression. The test by the AQL program is simplified by having a separate output for each existing type. Examples of these and the following forms of output may be found in the description of expressions.

```
[<expression>]
```

3.1.6.2 Output with Length Specification

The minimum length governs the minimum number of characters generated for output. If the string to be output is smaller, the rest of the length is filled with blanks. If the length is specified with a minus sign the output is left-justified (i.e. the spaces are added to the right), otherwise right-justified.

```
[<expression>,<min_field_width>]
```

If a third argument is specified, the maximum length of the output is defined with more precision (see "Example 6: real.aql" on page 3-12 und "Example 7: group.aql" on page 3-13). With alpha strings this defines the maximum number of characters to be output, with numerics the number of decimal places (output precision).

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

3.1.6.3 Hexadezimal Output of Integers

The `<precision>` option is prefixed with "x". The lower case "x" indicates that lower case is also to be used for the hexadecimal values. A capital "X" indicates output using all capitals.

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

The hexadecimal value generated is not truncated by the `<precision>` option, only extended. Leading zeroes (0) are generated.

3.1.6.4 Output of Floating Point Figures

The following prefixes are used with the "`precision`" option:

f	Output with specified number of decimal places.
e	Output is generated in exponential form with the specified number of decimal places.
E	Output is generated as with format "e" with a capital "E" in front of the exponent.
g	Output of the specified number of decimal digits. The system decides according to the length specified whether the ouput is to be in exponential form or fixed point.
G	Output is generated as with format "g" with a capital "E" in front of the exponent.

3.2 Symbols – AQL Variables

3.2.1 Definition of Terms

Symbol is the term used for a language object in AQL. A symbol may represent any type, whereas the concept of a variable is always associated with one type only.

A symbol may, within the limits of its scope, represent objects of different types. Type conflicts are always checked when a symbol is used within an expression or by a function.

3.2.2 Name Structure

A symbol name may consist of a maximum of 32 characters. All upper and lower case letters, numerals and the underscore _ are permissble. A symbol name must not begin with a numeral.

AQL is "case-sensitive" like UNIX; this means that one and the same string of letters in upper case and in lower case can describe different symbols (symbol != SYMBOL).

3.2.3 Scope

A symbol is always effective within the function in which it is declared. The file containing the AQL program is regarded as a function (main).

With nested functions, the symbols of the enclosing function are effective for all enclosed functions. This means that all symbols of the main program are of global effectivity. A symbol may be redeclared within an enclosed function and is then effective for this function and any other funtions it may enclose.

Symbols are only effective within a source file, in other words not within AQL programs invoked with *aql("...")*.

3.2.4 Declaration of Symbols – var

Not every symbol must be declared, since it is normal for a declaration to be made for the current function when the symbol is first used (implicit declaration). An explicit declaration is only needed when the scope of a symbol lies outside the function in which the symbol is first used.

```
var <name>
```

3.2.5 Initialization of Symbols

A symbol possesses no value after declaration and is of type **"invalid"**. The value may be enquired with the AQL function **"valid"**.

The initialization with a value is achieved by means of an assignment which also assigns the symbol the current type. This type is that of the result of an assigned expression.

```
my_symbol_name = expr
```

Names which are assigned to objects interactively are convertible to AQL symbols. If, for example, a line has been assigned the name L1, this name can be used to access it in AQL in the same way as a symbol.

The declaration and initialization can be combined in a single statement.
VAR a = 10;

Example: Delete the line with the name L1

```
delete (L1)
```

3.2.6 Symbol Types

The following symbols are available:

`string`	for declaring strings
`boolean`	for declaring logical values "true" and "false"
`integer`	for declaring integers
`real`	for declaring real numbers
`group`	for declaring groups of objects
`date`	for declaring dates
`object`	general symbol type (e.g. pointer to objects and actions)

All symbol types, as far as possible, are labelled by the assignment of a constant of the same type. The type *object*, which is needed for the processing of the ObjectD model and its parts, is described in the Online Help – "Attributes".

AQL knows the type of its objects and allocates symbols automatically. If objects are to be linked or compared, the compatibility of the data types should be checked. An error message is output in cases of incompatibility.

3.2.6.1 string

Constant character strings are enclosed in double quotes **""**. These are the character string delimiters. If this character is itself included in a character string, it is preceded by a \. A character string may be of any length.

When used as a function argument, constant strings are subject to constraints (e.g. file names under UNIX have a maximum length of 32 characters).

Example 3: string.aql (see section "Operators for Character Strings" on page 3-20)

PROGRAM:
```
var string;
string  = ”contents ”
delimiter = ”with delimiters \””
[string+delimiter] nl
[string+(delimiter-”* delimiter*”)] nl
```

OUTPUT:
```
contents with delimeters ”
contents with ”
```

3.2.6.2 logical

Symbols of this type may only contain the values for "true" and "false".

It should be noted that with explicit definition "true" und "false" may be written in lower case only.

Example 4: logical.aql

PROGRAM:

```
      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
```

OUTPUT:

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

3.2.6.3 integer

The value range of these whole numbers extends from

```
-2147483646   (hexadecimal -80000000)
to
2147483646    (hexadecimal 7FFFFFFF, 32 bit integer).
```

With explicit hexadecimal numbers the small letter "x" must always be used.

Example 5: integer.aql

PROGRAM:

```
      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
```

OUTPUT:

```
(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

Real numbers (constants) may be specified either as fixed point numbers (*with* decimal point) or in exponential form. The value range extends from -10^{308} to 10^{308}. If a notation which is usual for *integers* is used, the symbol is assigned the type *integer*. Since the system will, if required, convert from *integer* to *real* automatically, this type of number may be output in floating point format.

Example 6: real.aql

PROGRAM:

```
      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
```

OUTPUT:

```
(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

As the type of a symbol (object) only becomes relevant when its contents are procesed, it is possible for a group of symbols to contain symbols of different types and groups.

Example 7: group.aql

PROGRAM:

```
      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
```

OUTPUT:

```
(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

Symbols of this type are furnished with the AQL function *date* (see section "Date Function – date" on page 5-3).

Example 8: date.aql

PROGRAM:

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

OUTPUT:

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

3.3 Expressions

The following topics are discussed in this chapter:

- Overview of Operators
- Results of Mixed Type Expressions
- Results of Mixed Type Comparsions
- Operators for Character Strings
- Group Expressions

3.3.1 Overview of Operators

The table below contains an overview of the operators:

type of expression	*priority*[*]	*valid for*
`expr1and 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 ***

type of expression	*priority*[*]	*valid for*
`expr1 in expr2`	3	right operand group
`- 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	see chapter "Predefined Functions", Online Help ❍ AQL Functions for Objects, ❍ AQL Functions for Actions, ❍ Parameter and Values, ❍ System Attributes

[*] The figure given in the column "Priority" defines the strength of the tie between the operand and the operator. The higher the number, the stronger the tie. If an operand lies between two operators, the operation with the stronger tie, i.e. the higher priority, is carried out first. Adjacent operations with the same priority are carried out in sequence from left to right.

** With the logical expressions *and* and *or* the second operand is only processed when necessary. This is the case when:
true and <expression2>
false or <expression2>

*** Where operators are of different types the type of the result may be taken from the table "Results of Mixed Type Expressions" on page 3-18.

**** Since with *real* values the system carries out the check for equality with the maximum precision, it is advisable to check for equality within a given tolerance.

Example 9: priority.aql

PROGRAM:

```
[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;
```

OUTPUT:

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

3.3.2 Results of Mixed Type Expressions

The table below shows the results of mixed type expressions:

+ - * / **		*first operand*		
second operand	*real*	*integer*	*string*	*group*
real	real	invalid	invalid	only +, -
integer	real	integer	invalid	only +, -
string	invalid	invalid	only +, -[*]	only +, -
group	invalid	invalid	invalid	only +, -[**]
object	invalid	invalid	invalid	only +, -

[*] (concat, substring)

[**] (see section "Group Expressions" on page 3-21)

No check is made with "*integer*" and "*real*" of whether the value range has been exceeded (overflow).

Example 10: mixed_arit.aql

PROGRAM:

```
[3/2;;f3]              nl;
[3/2*1.0;;f3]          nl;
[3.0/2;;f3]            nl;
[”My hat it has ”+”three corners”] nl;
```

OUTPUT:

```
001
1.000
1.500
My hat it has three corners
```

3.3.3 Results of Mixed Type Comparisons

The table below shows the results of mixed type comparisons:

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

* The result of the real expression is converted to an integer by truncating the decimal places before the compare. If an overflow occurs, the result invalid is returned.

** Only permissible for the compare operators "=" and "!=".

= With a compare in which only one value is "invalid", the result always has the logical value "false". If both compare values are "invalid", the result has the logical value "true".

!= With a compare in which only one value is "invalid", the result always has the logical value "true". If both compare values are "invalid", the result has the logical value "false"

3.3.4 Operators for Character Strings

The following operators can be used for character strings:

- `+` Concatenation of character strings
- `-` Generation of substrings. In this case the operand on the right (after the sign -) must be a character string containing at least one wild card `*`. This character represents any character string. The reult of the operation consists of the characters represented by the wild cards. If the explicitly specified character string of the operand on the right cannot be found, the result of the expression ihas the value invalid.

Example 11: string.aql

PROGRAM:

```
var string;
string                = ″contents ″
delimiter = ″with delimiter \″″
[string+delimiter] nl
[string+(delimiter-″* delimiter*″)] nl
```

OUTPUT:

```
contents with delimiter ″
contents with ″
```

3.3.5 Group Expressions

The following attributes can be used for groups:

`.first`	The result of this attribute, defined for every group, is the first group .
`.tail`	The result of this attribute, present in every group, is a group with all group elements except the first (complementary to *.first*.)
`.el_*`	This attribute returns the nth element of a group. This means that individual group elements can be accessed (arrays in other programming languages).
`or`	The result is a group containing all elements of the two groups involved. This guarantees that no element occurs twice in the resulting group.
`*`	The result is a group containing only those elements which occur in both groups involved.
`-`	The result is a group containing only those elements occurring in only one of the two groups involved.
`+`	The symbol after the sign+ is added to the group on the left or to the specified individual group element.If a group is involved, the two groups are concatenated. There is no check for whether the any element occurs more than once in the resulting group. If the element involved is a single group element, it is added to the beginning or end depending on syntax.
`in`	If the first operand is included in the group specified as the second operand, the expression has the logical value "true", otherwise "false".
`where`	This operator links a group to logical expression. The result is all elements satisfying this condition. It is a prerequisite that every element is permitted by its type to be included in this condition. In the expression the group element to be examined is represented by a dot**.**. An attribute of a group element to be examined, which is to be checked, is specified with *.<attributename>*.

`sort by`	The operator *sort by* is followed by the sort criterion. The group element or an attribute of it is given in the same form as in the logical expression of the *where* condition. The sort criterion may be specified more precisely by means of the following keywords: asc ascending sequence (default). desc descending sequence
`and group`	This keyword is used to form subgroups of all elements with the same attributes.It is possible to specifiy more than sort criterion, separated by commas. When a combination includes *where* the *where* must preceed the sort criteria.
`select from`	select from This keyword is used to generate a second group from the original group on the basis of attributes Syntax <group> = select <attribute> from <group_expression>

The following **functions** are provided for processing groups:

`len ( )`	Determines the elements of a group (see section "String Length – len" on page 5-9)
`index ( )`	Determines an element of a group (see section "Determine a Group Element or Character from String – index" on page 5-3)
`empty( )`	Tests for empty groups (see section "Test for Empty Groups – empty" on page 5-3)
`pos ( )`	Determining the call value of an element of a group

Example 12: group_op.aql

PROGRAM:

```
      numbers = { 1, 2, 3, 4, 5 }
     groups_group = {numbers, numbers, numbers}
 (1)  ‘numbers:              ‘ [numbers] nl
      misc = { 4, ”yellow”, 2, 34.5, ”light”, false }
 (2)  ‘misc:            ‘ [misc] nl
      entire_group = numbers + misc
 (3)  ‘extension:         ‘ [entire_group] nl
      header = entire_group.first
 (4)  ‘first group element: ‘ [header] nl
 (5)  ‘rest of group:     ‘ [entire_group.tail] nl
      /*‘combination:  ‘ [numbers or misc] nl */
 (6)  ‘average:        ‘ [numbers and misc] nl
 (7)  ‘difference 1-2:       ‘ [numbers - misc] nl
 (8)  ‘difference 2-1:       ‘ [misc - numbers] nl
 (9)  ‘condition:           ‘ [numbers where . >2 ] nl
(10)  ‘Sort ascending:    ‘ [numbers sort by .  ] nl
(11)  ‘Sort descending:     ‘ [numbers sort by .  desc] nl
(12)  ‘Sort descending with condition:  ‘
                        [numbers where . > 1 sort by .  desc] nl
(13) ‘single element‘       ‘[numbers.el_3] nl
(14) ‘group of groups: ‘[groups_group] nl
(15) ‘1st choice:          ‘[select .first from groups_group] nl
```

OUTPUT:

```
 (1) numbers:              { 1, 2, 3, 4, 5 }
 (2)  misc:             { 4, yellow, 2, 34.5, light, false }
 (3)  extension:          { 1, 2, 3, 4, 5, 4, yellow, 2, 34.5,
                             light, false }
 (4)  first group element: 1
 (5)  rest of group:    { 2, 3, 4, 5, 4, yellow, 2, 34.5, light,
                             false }
 (6)  average:          { 2, 4 }
 (7)  difference 1-2:       { 1, 3, 5 }
 (8)  difference 2-1:       { yellow, 34.5, light, false }
 (9)  condition:           { 3, 4, 5 }
(10)  Sort ascending:    { 1, 2, 3, 4, 5 }
```

```
(11)  Sort descending:     { 5, 4, 3, 2, 1 }
(12)  Sort descending with condition:  { 5, 4, 3, 2 }
(13) single element:      3
(14) group of groups: {{1,2,3,4,5} {1,2,3,4,5} {1,2,3,4,5}}
(15) 1st choice:            {1,1,1}
```

3.4 Control Statements

The following control statements can be used:

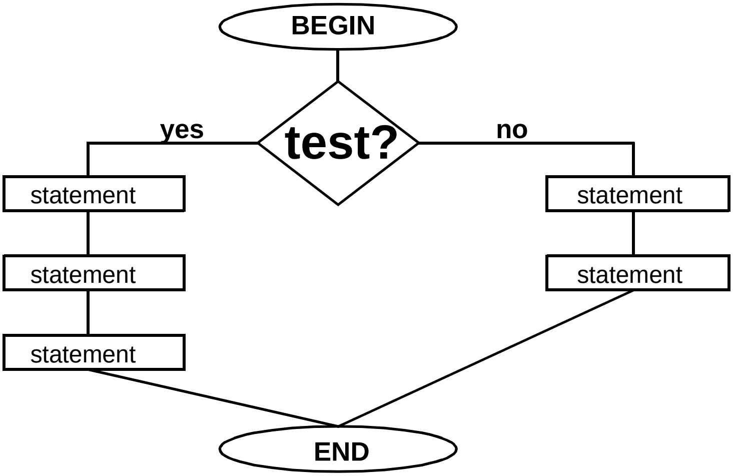
Condition	`if...then...else`
Conditional Loop	`while...do`
Group Element Loop	`for..in..do`
Case Differentation	`switch...case...`

3.4.1 Condition – if...then...else

Two **forms** of this statement are permissible:

```
(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
```



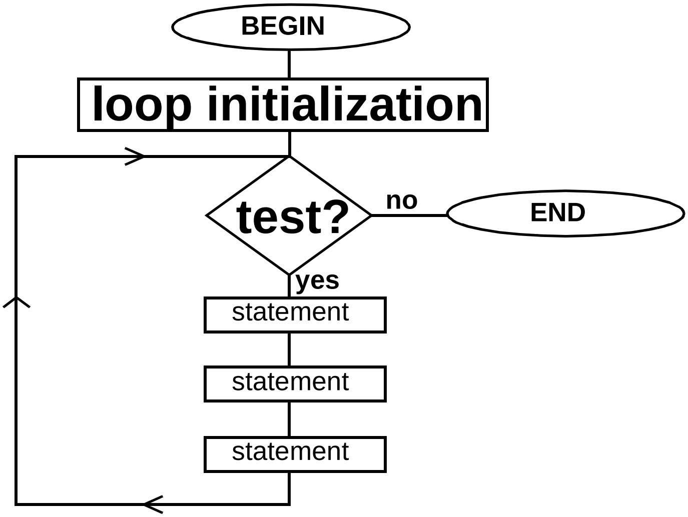
This statement is concluded, like other control statements, with an *end*. The *else..if* structure is achieved by an *if* block within the *else* block.

3.4.2 Conditional Loop – while...do

While loops are the only loop constructs in AQL. As *for* loops in the normal sense are not available, these must be constructed using *while*.

This statement takes the following **form**:

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



As long as the logical expression in the *while* statement receives the value "true", the statements will be executed up to the corresponding *end* statement and the execution then continued from *while* statement. If the control expression in the *while* statement returns the value "false", a branch is made to after the corresponding *end* statement.

No check is made for an endless loop. In order to prevent this, the condition should become "false" after testing. In order to prevent an endless loop, the condition must become "false" after an execution.

3.4.3 Group Element Loop – for..in..do

This statement, which differs from other similar ones in other languages in only serving to carry out groups, takes the following **form**:

```
for <loop_var> in <group_expression> do
   <statements for each element in group_expression>
end
```

The loop symbol <*loop_var*> is added automatically. All permissible group expressions may be used as <*group_expression*>.

3.4.4 Case Differentation – switch...case...

This statement may take two **forms**:

```
(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 for all other cases>
      end
```

It must be possible for a compare to be made between the types of <*expression*> and <*alternative_1*>. In the form of the statement without *otherwise*, if no alternative is found, the statement will not be executed.

In contrast with C, only the valid alternative is carried out. All subsequent alternatives are ignored!

3.5 Functions

A distinction is made between two **types** of function:

- predefined functions (see chapter on "Predefined Functions" on page 5-1)
- functions declared by the user in the AQL program.

3.5.1 Function Declaration – function

```
function <function_name> (<parameter>, ...)
   <statements for execution of function>
endend
```

`<function_name>`	The name of a function is structured in the same way as that of a symbol.
`<parameter>`	The parameters are symbols of any type, which are not declared in the function. The validity of the type of a parameter is checked for all symbols in an expression at the time of execution. The brackets following the function name are still necessary when the function does not require parameters. If no parameter is specified by the calling function (this is permissible), the parameter has the value **"invalid"**.

AQL treats all parameters as expressions and furnishes the called function with an auxiliary symbol containing the result (*call by value*) or with the reference (pointer) to the original symbol prefixed with the keyword *var* in the call list (*call by reference*). This means that when the function manipulates these parameters, the value of the variables in the calling function is also manipulated.

Example:

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

Function Value

Every function supplies a function value to the program using it. Ths function value is treated like the result of an expression. This allows function calls to be included in expressions, which may in turn be function parameters. The return value is defined in a function declaration as an argument of the *return* statement. If the branch back from a function is not made via a *return* with argument, the function value is "*invalid*". If there is no *return*, the branch back takes place on reaching the *end* statement.

Contents

Every function is treated in the same way as the main program and vice-versa, and so the function may contain symbol and function declarations. Nested function declarations are also possible, as in Pascal. Particular care is needed with larger AQL programs when using global symbols of this kind.

Example 13: param.aql

PROGRAM:

```
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(,)
```

OUTPUT:

```
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 Forward Declaration of a Function – function...forward

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

This is a way to declare a function in advance without its contents. It is necessary for the number of arguments in this declaration to agree with the number of arguments in the declaration of the function with contents. This form is relevant when recursive functions call each other:

```
function a (count) forward

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

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

3.5.3 Branch back from Functions – return

This may be used in four **forms**.

The **first two** forms have the same effect. The function value "*invalid*" is returned.

```
return
return()
```

With the **third** form, the expression specifies the function value. A *return* in the main program causes this to terminate.

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

With the **fourth** form, the return code of the AQL program is sent to an invoking AQL program.
The actual return to the invoking AQL program takes place either after a *return* statement or at the end of the file, however.

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

3.5.4 Function Pointers

Function pointers can be used to make the programs more flexible. The general form of the statement is as follows:

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

3.5.5 Function Call

A function may be called in the form of a statement. The function is identified either by its name or by the function pointer.
`<function_name>([<parameter>,...]).`

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

This ignores a possible function value. If the function value is to be further processed, the following form of expression is required:
`<symbol> <operator> <function_name>([<parameter>,...]) ...`

It is necessary to take account of the fact that the function being used supplies a function value and that this value is of compatible type where it is used. Every function must be declared before use. Recursion, i.e. the use of a function by itself, is also possible, as shown in the classical example below.

Example 14: function.aql

PROGRAM:

```
function hanoi(stack, from, to, use)
   if empty(stack.tail) then
      ‘moving ‘[stack.first;-5]
      ‘ 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”)
```

OUTPUT:

```
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 File Handling

The following functions can be used:

- General Form of File Assignment – file
- Output File Assignment – file output
- Input File Assignment – file input
- Test for File Access – file_access
- Test for Existence of a File – existf
- Test for Existence of a Directory – existd
- Error File Assignment – file error_output
- Open a Binary File – bin_open
- Read an AQL Symbol – get
- Positioning in the Input File- get, relget
- Find Position in Input File – line
- Write to a Binary File – bin_write, bin_write_byte, bin_write_short
- Read a Binary File – bin_read_byte, bin_read_short, bin_read_int
- Read a Name – parse_name
- Read a Number – parse_number
- Read a Line – parse_line
- Read a Keyword – parse_keyword
- Read a Comment – parse_comment
- Read a String – parse_string
- Close Input File – close
- Close a Binary File – bin_close

A file handling example can be found on "Example of File Handling" on page 7-4.

3.6.1 General Form of File Assignment – file

The general form of the *file* statement is as follows:

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

where <*file_name_string*> describes an expression of which the result is the name of the file concerned in string form. This name must conform to system file name conventions. The description <*screen*> is a permissible special form which refers to the window in which ObjectD was started. This also the default type for all files. Every asignment becomes effective immediately.

- When file names are assigned, the rules of the operating system must be taken into account.
- The environment variables of the operating system are not supported by AQL.

3.6.2 Output File Assignment – file output

This statement is used to control the output of constant character strings and of the statements n *[...]*,*nl* and *sp*.

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

Output of the keyword *output* is optional. A file accessed for the first time during program execution is opened with *write*, i.e. if not already in existence, the file is set up, and if in existence, its current contents are overwritten by the new outputs.

If a file is opened again during execution of an AQL program, it is opened with *append*, i.e. the new outputs are added to it.

Files are closed automatically when the program terminates or implicitly when another output file is opened. A return to the default <*screen*>, which corresponds to the standard stream *stdout*, closes any files still open.

3.6.3 Input File Assignment – file input

The input stream which normally reads from <*screen*> (UNIX stream *stdin*) is assigned to an existing file.

```
file <file_name_string> input
```

The first *file input* statement referring to a file during program execution causes the file to be read into working storage. All read accesses with *get* are supplied from this working storage area, even if the file has since been opened again.

3.6.4 Test for File Access – file_access

The function *file_access* checks the access authorizations. It returns a "*boolean*" value.

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

Examples:

The order of the letters "r", "w" and "x" has no significance.

`file_access ("file_name", "rwx")`
Can file be read, written to and executed?

`file_access ("file_name", "r")`
Can file be read?

`file_access ("file_name", "w")`
Can file be written to?

`file_access ("file_name", "rw")`
Can file be read and written to?

3.6.5 Replace Special Characters at Start of File Name – file_extend

The function *file_extend* replaces special characters (`#`, `+`, `~`) at the start of a file name.

```
<res_string> = file_extend (<string>)
<string>   file name
```

3.6.6 Test for Existence of a File – existf

This function returns the value "true" if the specified file exists.

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

3.6.7 Test for Existence of a Directory – existd

This function returns the value "true" if the specified directory exists.

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

3.6.8 Error File Assignment – file error_output

All error messages are output to this stream, which normally corresponds to the screen (*stderr*). The stream is treated in the same way as the output stream.

```
file <file_name_string> error_output
```

3.6.9 Open a Binary File – bin_open

This function opens a binary file.
`<fd> = bin_open (<filename>, <mode>)`
`<fd>` *file_descriptor*, `will be used with the functions` *bin_write*, bin_read** und *bin_close*.
`<filename>` `filename`
`<mode>` `"input"/"output"/"append"`

The first argument identifies the file to be opened. If successful, the function returns *file_descriptor*.

3.6.10 Read an AQL Symbol – get

The specified symbol is furnished with the next symbol to be found in the input stream (see statement *file .. input*). At the same time the internal read pointer is positioned at the end of the symbol being read to allow the next symbol to be read with the next access.
`<symbol> = get()`

Spaces and end of lines are ignored on reading, as long as they are not included within character strings which are enclosed in quotes. Whole numbers and floating point numbers (*integer* and *real*) begin with a numeral or a minus sign. Character strings begin with all other characters and end with the first space or the end of line, as long as they are not enclosed in quotes.

An example program can be found on "Example of Character Search in a File" on page 7-6.

3.6.11 Positioning in the Input File- get, relget

This form of the *get* function is used to position the read pointer at the position specified within the input file.

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

The read function *relget* positions the read pointer relative to its current position. Positive values for "offset" move the pointer towards the "end of file", negative values towards the "start of file". If the line number specified or calculated is negative or greater than the total number of lines in the input file, i.e. outside the limits of the file, the value *invalid* is returned. If the current input file is the screen (<*screen*>), the positions are ignored.

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

3.6.12 Find Position in Input File – line

The number of the current line within the input file is output. If the current input file is the screen (<*screen*>), the value *invalid* is retured.

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

An example program can be found on "Example of Character Search in a File" on page 7-6.

3.6.13 Write to a Binary File – bin_write, bin_write_byte, bin_write_short

The function *bin_write* writes every argument to a binary file. The arguments must be *integers*, *real* values or strings.

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

The function *bin_write_byte* writes the first byte of every argument to a binary file. The arguments must be integers.

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

The function *bin_write_short* writes the first data type *short* of every argument to a binary file. The arguments must be integers.

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

3.6.14 Read a Binary File – bin_read_byte, bin_read_short, bin_read_int

The function *bin_read_byte* reads a byte from a binary file. The function returns an integer if successful.

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

The function *bin_read_short* reads the first data type *short* from a binary file. The function returns an *integer* if successful.

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

The function *bin_read_int* reads a data type *integer* from a binary file. The function returns an *integer* if successful.

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

3.6.15 Read a Name – parse_name

The function returns the name string from the input stream (max. 31 characters).

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

3.6.16 Read a Number – parse_number

The function returns the value of the number as an *integer* or *real* number.
`<number> = parse_number()`

3.6.17 Read a Line – parse_line

The function returns the contents of the rest of the line as a string up to n*ew_line*.
`<res_string> = parse_line()`

3.6.18 Read a Keyword – parse_keyword

This function returns the value "true" if the next item read corresponds to the "*keyword*". In this case, the read pointer points to the next entry. Otherwise, the read pointer remains where it is and the function returns the value "false" as its result (see example program on "Example of Character Search in a File" on page 7-6).
`<res_bool> = parse_keyword()`

3.6.19 Read a Comment – parse_comment

The function returns the next comment in the input file.
`parse_comment ( )`

3.6.20 Read a String – parse_string

The function returns the next string (in double quotes"").
`parse_string ( )`

3.6.21 Close Input File -close

AQL automatically closes all files still open at program termination. The explicit closing of an input file is thus only necessary if a file is updated during the same run of the program and is to be read afterwards. The type of the function value is *logical* and states whether the close was successful (true) or not (false). After closing the input file the standard assignment <*screen*> is active.

```
close()
```

3.6.22 Close a Binary File – bin_close

The function closes a binary file.

```
bin_close (file_descriptor)
```

`file_descriptor` see *bin_open*

3.7 Errors

This chapter contains the following topics:

- Error Handling
- Error Search

3.7.1 Error Handling

Various types of **error** may occur during an AQL session:

Syntax errors:	Syntax errors are detected by the AQL syntax analysis facility during the precompilation phase.
AQL-execution errors	These errors are detected during execution of the program. Typical examples are: inquiry of the attributes if an undefined symbol use of uninitialized variables expressions with incompatible operands (e.g. compare *real* and *string*)

Error messages are displayed during interactive working in a popup window on the screen.

The ***error_response*** function makes it possible to control the further processing of an AQL program in the event of a run time error:

`error_response (”stop”)`	AQL is terminated
`error_response (”continue”)`	AQL continues

In batch mode the AQL interpreter writes all error messages into the currently active message output file. This is normally the screen. The ***error_file*** statement may be used to divert the output to a different file.

The output diversion facility is not active during the compilation phase. Syntax errors are thus always written to the standard output.

A given error is only reported once during execution, even if it occurs within a loop.

Syntax and execution errors are output in English. Action errors are output in the language specified in I-DEAS Variant Engineering.

It is possible in some cases to predict the occurrence of errors and to cause the program to take appropriate measures when necessary (see return values of AQL functions and chapter "Message Output – display_error" on page 5-14).

The global AQL symbol ***errmes*** makes it possible to evaluate the text of an error message. *errmes* has the following attributes:

`.text`	Text of the error message
`.line`	Line number in which error occurs
`.file`	File name of the AQL procedure containing the error

errmes is set to "invalid" before each call of a model oriented function.The text attribute contains the text of error message. The identification of an error from the text of the message depends however on the language and possibly the version number.

With actions it is also possible to test the *valid* attribute of the action pointer being generated. It is recommended that this test be carried out before accessing *errmes*. The valid attribute test triggers first of all a new attempt to evaluate the object.

Generated objects are not evaluated immediately if the system is not in update mode. The *errmes* symbol is then only set by the valid test.

Example

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

3.7.2 Disable Output of error messages – disable_messages

The function *disable_messages* disables outputs of error messages in AQL.

```
disable_messages ( )
```

3.7.3 Enable Output of error messages – enable_messages

The function *enable_messages* enables outputs of error messages in AQL.

```
enable_messages ( )
```

3.7.4 Interactive Error Output – short_messages

This function switches interactive error outputs for runtime errors on or off.

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

3.7.5 Error Search

The detection of errors in AQL programs is assisted by inserting ***break*** statements at any desired point in the program code.

```
break(<prompt_string>).
```

This statement is part of the AQL function *break*. Its sole argument is a string containing a prompt for input by the user. The string is output when the interpreter reaches the function after the statement in which the break function is included. The statement displayed has at this time not yet been executed. The user has three options for continuing processing:

- Output of contents of variables with the *[...]* statement.
- Step by step execution of this and the subsequent statements by depressing <RETURN>.
- Continuing the AQL program by inputing the letter *g*.

Input and ouput are also uninterrupted in this program status. Due to the user interrupt, the exact point for this cannot be defined precisely and is also dependent on the processing mode (interactive or batch) of the system.

There are 3 special options for controlling the output in *prompt_string*:

```
%f   file name
%n   line number
%l   current character.
```

Examples:

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

The function can also be called without arguments:

```
break()
```

This call corresponds to the 1st example.

It is possible to trace the opening of files with the AQL funktion ***verbose***, which contains a parameter by means of which this mode is activated or inactivated.

```
verbose(logical_expression)
```

4 AQL as Command Language

Up to now, only the non-model oriented aspects of AQL have been discussed. Within I-DEAS Variant Engineering, however, AQL serves as a command language for the generation, manipulation and retrieval of CAD models and for the further development of the application.

4.1 Compatibility of AQL Programs

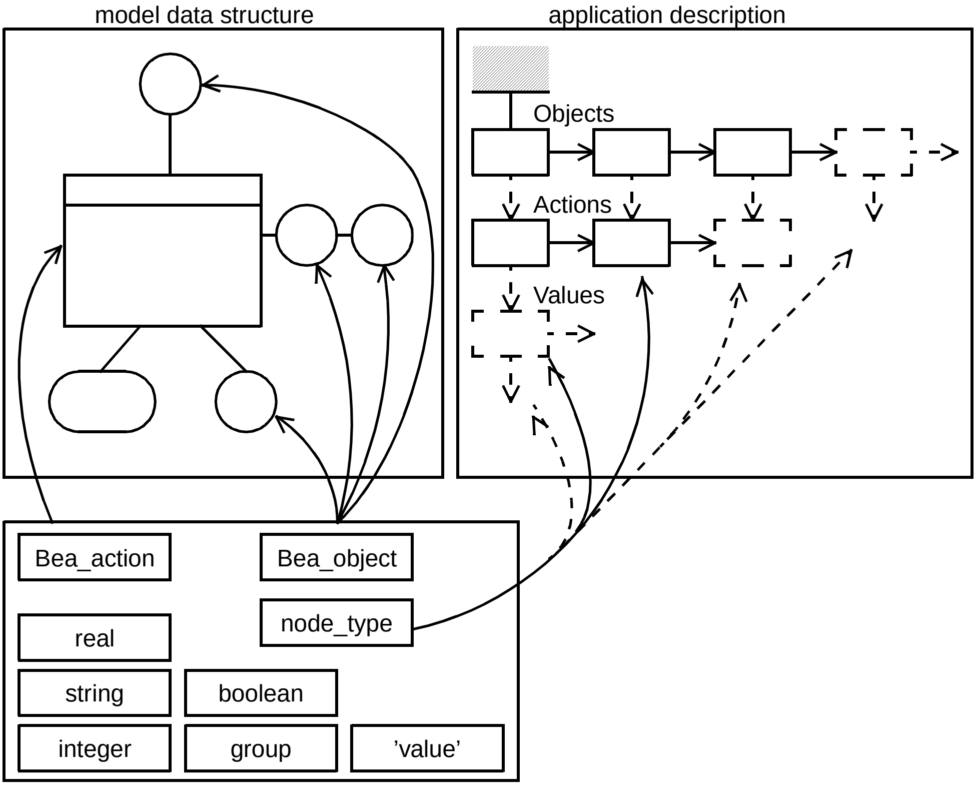
As far as syntax and keywords are concerned, the AQL language remains generally compatible between versions. It is possible, however, that previous AQL compiler tolerances have been adversely affected by the eradication of errors. It is therefore strongly recommended as in all programming languages that programmers refrain from exploiting syntax tolerances and adhere closely to the preferred syntax (see chapter "Syntax Diagrams" on page 8-1).

I-DEAS Variant Engineering actions implemented in AQL also remain compatible between versions, as long as they do not impede the further development of the application. The AQL programmer should not rely on the implementation techniques of function side-effects.

As in other languages, programming via addresses is only possible within one executing AQL program, since addresses and object identifications cannot be kept constant over more than one AQL run.

4.2 AQL Symbols and ObjectD Data Structure

Three types of pointer, which allow actions and objects in the data structure to be identified, are available for accessing this structure. These identifiers may be used to refer to objects and actions for generation, deletion, modification and retrieval.



AQL symbol types

Bea_object	Pointer to any object in the model data structure
Bea_action	Pointer to any object in the model data structure
node_type	Pointer to the application description This is a detailed description of the objects, actions, values, enums, etc.

4.3 General Generation and Command Syntax

4.3.1 Generation of Absolute Objects

The general syntax for generating absolute objects in AQL is defined as follows:

```
<res> = <Objectname>_absolute (prop1, prop2, ..., propn)
```

`<res>` belongs to the type "Bea_object"

Example: Absolute point at the position (100, 100)

```
p = point_absolute ({first_world, {100, 100}})
```

User-defined objects obey the same convention; <objectname> is then as defined by the user.

4.3.2 Integration and Execution of Actions

The general syntax is defined as follows, according to the type of action:

Action with result object

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

`<act_obj>`	Can be used either as a Bea_object (= result object) or as a Bea_action (= pointer to an action) depending on the application
`<actionname>`	Obeys the following convention: <object name>_<generationtype>
`[id]`	Identifier (optional); only used within execute of UDAs,e.g. for identifying / defining effect objects
`prop1 ... propn`	Properties of the result object
`par1 ... parn`	Parameters of the action

Action without result object

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

`<act>`	Pointer to an action belonging to the type "Bea_action"
`<actionnname>`	Name of the action according to the convention <mainname>_<secondaryname>
`par1 ... parn`	Parameters of the action

Actions with the attribute "drop"

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

`<bool>`	Result variable "true" = execution successful "false" = execution unsuccessful
`<actionname>`	Name of the action according to the convention <mainname>_<secondaryname> mainname may be an object on which the action acts

User-defined actions

Cf. a. to c., according to the defined attribute. <actionname> is the name of the action as defined by the user.

Optional parameters which are to be filled with the default value may be skipped by specifying "**,**".

Parameters for absolute objects may be specified in a shortened form (see chapter "Implicit specification for absolute objects" on page 4-11).If insufficient attention is paid to the programming, this may result in unintentional data structure effects ("explosion" of the data structure during movement simulation, for example).

4.4 Functions Acting on the Model

4.4.1 Opening a New Model – input_new

The function *input_new* opens a new model.

Only implemented for compatibility reasons. New: **model_close** and **model_load/model_create**

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

`<boolean>`	`true:` Deletes all objects from the actual model
`<string>`	File name

4.4.2 Reading a Model – input_read

Reads a model.

Only implemented for compatibility reasons. New: **model_load**

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

`<string1>`	File name
`<string2>`	Type of reading (add, save, forget)

4.4.3 Changing the Dimensional Unit – drawing_inch_mm

The function *drawing_inch_mm* specifies the dimensional unit for the drawing.

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

`<boolean>`	true = inches

4.4.4 Copying Selected Objects – inrect_copy

The function *inrect_copy* copies selected objects.

Only implemented for compatibility reasons. New: **group_copy**

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

`<integer>`	Number of copies
`<vector>`	Copy vector
`<select_rect>`	Selection set

4.4.5 Redefining Selected Objects – inrect_cut

The function *inrect_cut* redefines selected objects.

Only implemented for compatibility reasons. New: **redefine_cutall**

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

`<world>`	Coordinate system
`<select_rect>`	Selection set
`<redefined object>`	Redefined objects

4.4.6 Duplicating Selected Objects – inrect_duplicatecopy

The function *inrect_duplicatecopy* duplicates selected objects.

Only implemented for compatibility reasons. New: **multi_duplicate**

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

`<integer>`	Number
`<deleted_elem>`	Deleted relations
`<vector>`	Copy vector
`<world>`	Coordinate system
`<select_rect>`	Selection set

4.4.7 Mirroring Selected Objects – inrect_mirror

The function *inrect_mirror* mirrors selected objects.

Only implemented for compatibility reasons. New: **group_copymirror**

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

`<integer>`	Number
`<line>`	Mirror axis
`<select_rect>`	Selection set
`<mirrored objects>`	Mirrored objects

4.4.8 Separating Selected Objects – inrect_separate

The function *inrect_separate* separates selected objects.

Only implemented for compatibility reasons. New: **redefine_cutborder**

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

`<deleted_elem>`	Deleted relationships
`<world>`	Coordinate system
`<select_rect>`	Selection set
`<selected>`	Separated elements

4.4.9 Define model identifier – model

Outputs the model with the specified name or the active model if a name is not specified.

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

`<string>`	Model name (optional)

4.4.10 Saving a Model – output_save

Saves a model.

Only implemented for compatibility reasons. New: **model_save**

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

4.4.11 Generating a New Model from the Selection Set – output_savesection

Generates a new model from the selection set.

Only implemented for compatibility reasons. New: **model_outofselset**

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

`<world>`	Coordinate system
`<select_rect>`	Selection set
`<filename>`	File name

4.4.12 Defining the Active Model – set_active_model

Defines the active model

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

`<string>`	Model name

or

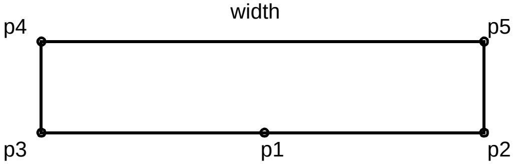
```
set_active_model   (<object>)
```

`<object>`	Model

4.5 Create Functions

Example: Generation Functions – create

A typical program is given below. The program generates a block consisting of four points and the correesponding lines between them. The point *p1* receives the name *reference_point*, and the width the name *width*.



```
width = length_absolute(100)
p1 = point_relative(origin, 100.0, 100.0)
p2 = point_samey(p1,length_quotient(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 requires an argument which is a real value. In this example an integer value is given. Integer value arguments are converted automatically to real value arguments if necessary.

The actions *point_samey*, *point_samex* and *point_relative* are normally used with a length as the second argument:

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

Implicit specification for absolute objects

The values of absolute objects (*length*, *angle*, *coordinates*, *number ...*) may be input directly:

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

The system automatically generates the absolute object *length_absolute (20)* as a parameter for *p4*.

Example 1:

The AQL variables *p1*, *p2*, *p3*, *p4*, *p5* and *width* are used. This does not mean that the objects generated receive the names *p1*, *p2*, *p3*, *p4*, *p5* and *width*. An object is given a name by means of the AQL function *name*. The variables *p1*, *p2*, *p3*, *p4*, *p5* and *width* exist only while the AQL program is running. They are not stored in the CAD model. The objects, however, remain in the data structure permanently.

The same program may also be written as an AQL function. The function has two arguments: the reference point and the width of the block. In this version the reference point and the width of the block do not receive names. *p2* and *p3* are constructed by a different method.

```
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

/* main program */

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

In this version, three objects of type *length* are constructed implicitly. The first two have the values *width/2* and *-width*; there is however no relationship between them.

Optional arguments may be omitted. The comma is a mandatory separator.

Example 2:

The following function generates a tower of blocks of smaller and smaller width towards the top. The arguments are the reference point of the bottom block and the number of blocks.

```
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

/* main program */

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

The function returns a group containing the reference points of all the blocks; this is not used in this example, however.

4.6 Interactive Specification of ObjectD Parameters

The string "$" used as a parameter serves to interactively request parameters from the column 4 menu. All parameters not previously supplied are requested one after the other until they are complete. The system behaves in exactly the same way as when parameters are requested in dialog mode. Implicit generation via column 4 (and recursions) is possible.

$ functions at the top level only. Parameter lists must be input in complete form and suitably terminated (e.g. parameter list for automatic contour generation must be in sequence start point, end point, auxiliary point).

It is often recommendable that $ be used together with the *prompt* function. An input prompt is then output.

Example:

```
prompt (”Please input relative point!”)
p = point_relative (”$”,100,20)
```

The point "p" is defined relative to a point specified by the user. The lengths "x" and "y" are not requested as they have already been supplied.

```
prompt   (”Please enter start and auxiliary points for plane
          generation”)
plane_tracing (,,,,,”$”,3,45)
```

AQL generates a plane. The list of start and auxiliary points must be input and terminated.

4.7 Editing Functions – edit

The ***edit*** function corresponds in its functionality to the EDIT mode in ObjectD. The general definition form is as follows:

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

Parameters which are not to be changed may be skipped using "**,**".

Parameters for absolute objects may be specified in a shortened form (see chapter "Implicit specification for absolute objects" on page 4-11). If insufficient attention is paid to the programming, this may result in unintentional data structure effects ("explosion" of the data structure during movement simulation, for example).

The next function shows how a block is moved from one position to another. The function has three arguments:

- The first argument identifies the reference point of the block to be moved.
- The second argument identifies the reference point over which the block is to be positioned.
- The third argument shows how high above the reference point the block is to be positioned.

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

This ***move*** function moves the block by editing its reference point with the aid of the AQL function *edit*. The block is moved automatically by the ObjectD evaluation mechanism.

In the example above two parameters are changed at once with the *edit* operation. It is also possible to change just one parameter at a time:

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

The second form of edit allows **one** parameter to be selected directly.
This avoids the need to type in commas or create complex edit strings for the eval function if an action is unknown.

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

4.8 Redefining an Object creating Action

4.8.1 Redefining an Action – redefine

The action used to generate the object may be changed with the AQL function *redefine*. The first argument of the *redefine* function is the object to be changed. The second argument is the new object. The second argument must be of the same object type as the first.

Redefine is only appropriate for objects belonging to the same layer.

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

4.8.2 Redefining an Action of Effect Objects – redefine_effect

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

`<object1>`	Object to be changed
`<object2>`	Changed object

4.9 Deleting

4.9.1 Deleting objects – delete, undo delete

Objects may be deleted with the AQL function ***delete***. The argument of this function is either an object or a group of objects.

```
delete(block)
```

If *delete* is used to delete an object that has already been deleted, this may lead to unintended side effects ("toggle", see attribute *.deleted*, Online Help – "Attributes of the Object Type Bea_object").

The AQL function ***undo_delete*** reverts a previous deletion action (see volume *exploring*, chapter "Redisplay of Deleted objects")

```
undo_delete(block)
```

4.9.2 Removing an Action – remove_action

All the objects generated with this action are removed again.

```
remove_action (action_identifyer)
```

4.9.3 Canceling a Manipulator Action – remove_manipulator

Cancels a manipulator action.

```
remove_manipulator (<action>)
```

`<action>` Manipulator action

4.10 Assigning Names – name

An object may be named or renamed with the AQL function *name*. The maximum length of an object name is 31 characters. The first object identifies the object concerned.

```
name(object,”the_name”)
```

4.11 Definition of User Element Parameters

An object is defined as a user element parameter by using the AQL function *inpar* (see User Guide I-DEAS Variant Engineering volume *exploring*, chapter "Parameterization of the User Element"):

4.11.1 Definition of Input Parameters – inpar

The specified object or group of objects is defined as an input parameter for the current model.

Only implemented for compatibility reasons. New: **udo_define/uda_define**

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

The function may only be called at end of program before output_save.

If characteristics (*linestyle, font, ...*) are to be declared as parameters, the AQL function *inpar_ttype* should be used.

4.11.2 Definition of Characteristic Parameters – inpar_ttype

Characteristics only (*linestyle, font ...*) may be defined as characteristic parameters using this AQL function.

Only implemented for compatibility reasons. New: **udo_define/uda_define**

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

`<object>`	Object whose characteristics are to be input parameters
`<seq_number_of_char_parameter>`	Number of parameters within the parameter sequence of the object

Objects (*point, line, length, angle ...*) are declared using the AQL function *inpar*.

The function may only be called at end of program before output_save.

4.12 Layer Technique – layer

4.12.1 Generation of Layers – layer_normal

Layers are generated with the AQL function *layer*:

Only implemented for compatibility reasons. New: **layer_create**

```
<new_layer> = layer_normal (<string1>,
                            <string2>,
                            <boolean>)
<string1>     Layer name
<string2>     Selectable
              Inactive
              Active
<boolean>     Locked
<new_layer>   New Layer
new_layer = layer_normal(”layer_name”, ”selectable”,false/true)
new_layer = layer_normal(”layer_name”, ”inactive”)
new_layer = layer_normal(”layer_name”, ”active”)
```

Layers may be edited with the normal *edit* function:

```
edit(my_layer,,,”active”)
edit(my_layer, ”new_name”,status,locked/unlocked)
```

4.12.2 Opening the Layer Status Dialog – layer_status

Opens the dialog for the layer status.

```
layer_status ( )
```

4.12.3 Assignment of Layer Color- color_layer

A color is assigned to a layer.

```
color_layer(layer, color index)
```

4.12.4 Assignment of Object Group – move_set_to_layer

A group of objects is assigned to a layer.

Only implemented for compatibility reasons. New: **layer_move_objects**

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

4.12.5 Assignment of an Object to a Layer – move_object_to_layer

Assigns an object to a layer.

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

`<object>`	Object to be assigned
`<layer>`	Target layer

4.12.6 Moving a Layer to a Sublayer – move_sublayer_to_layer

A level is made a sublayer of another level.

Only implemented for compatibility reasons. New: **layer_move_sublayer**

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

The selected sublayer may not be locked.

4.13 User-defined Attributes

4.13.1 Creation of Attributes – create_attrib

Attributes are created with the AQL function *create_attrib*. The value may be specified either with this function or with *set_attrib*.

This form sets an empty string as the value.

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

`<object>`	Object
`<string>`	Attribute name

or

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

`<object>`	Object
`<string>`	Attribute name
`<value>`	`integer | real | boolean | string`

or

This form describes the value of the attribute in the following form: value of attribute <name> of object <reference>.

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

`<object1>`	Object
`<string1>`	Attribute name
`<object2>`	Reference
`<string2>`	Attribute name of reference

4.13.2 Creation of Attributes – create_attrib_expr

Attributes are created with the AQL function *create_attrib*. The value may be specified either with this function or with *set_attrib*.

This form sets an empty string as the value.

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

`<object>`	Object
`<string>`	Attribute name

or

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

`<object>`	Object
`<string1>`	Attribute name
`<string2>`	Expression

or

This form describes the value of the attribute in the following form: value of attribute <name> of object <reference>.

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

`<object1>`	Object
`<string1>`	Attribute name
`<object2>`	Reference
`<string2>`	Attribute name of reference

4.13.3 Assignment of Attribute Values – set_attrib, set_attrib_expr

Attributes are assigned to an object with the AQL function *set_attrib*.

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

The AQL function *set_attrib_expr* makes it possible for the user to define an attribute comprising an AQL expression. This attribute expression is newly evauated each time it is read.

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

<object>	Object
<string>	Attribute name
<aql_expression>	AQL expression

4.13.4 Copying an Attribute Set – copy_user_atts

The AQL function *copy_user_atts* copies all attributes from the source object (source) to the target object (target).

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

4.13.5 Deleting Attributes – delete_attrib

Attributes can be deleted again with the AQL function *delete_attrib*. The first argument of each of these functions can be an object or a group of objects.

```
delete_attrib(p1,”attribname”)
```

4.14 System Attributes

4.14.1 Common Attributes

Attributes common to all object types are listed as attributes belonging to the type *Bea_object* in the Online Help, chapter "System Attributes".

If the object was generated by means of an action (e.g. as a result object), the attributes of this action (Bea_action) may be retrieved as well.

4.14.2 Attributes for UDOs, Layers and Groups

Attributes for UDOs, layers and groups are listed as attributes belonging to the object types *group_type* and *layer_user_type* in the Online Help – "Attributes".

4.14.3 Attributes for Other Object Types

The symbol *program* can be used to access system information. The attributes for the object type *program_type* are listed in the Online Help – "Attributes".

4.15 Accessing the ObjectD Data Structure

4.15.1 Access by Name and Data Structure Traversal

Acess to the ObjectD data structure is possible by two **routes**:

- direct access to objects by name
- traversing the data structure starting with the AQL symbol *top.*

Accessing an object **by name** is only possible if the object has been assigned a name by the user.

Access to objects by name is appropriate, for example, when data is to be taken from a CAD model and used in a calculation program, and the model then modified on the basis of this data. In such cases only a small number of objects is generally involved. These may therefore be named interactively.

If a larger number of objects is involved, it is better to assign them all to a group, which is then given a name.

The AQL symbol *top* represents the uppermost layer of the active model.

`list_all`	list of all objects in a model file
`list_out`	list of all objects in a model file sorted by interdependencies (independent objects first)
`list_<obj>`	list of all objects of a given type: *list_line* supplies a list of all lines
`el_<number>`	the object with the identification number <*number*>

A typical program is as follows:

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

The program runs through the list of all lines and outputs the position data for each line.

The program could also be written as follows:

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

This second version forms a list of all objects, but then reduces the selected group to those objects which satisfy the condition *prim_type = "line"*. This version is less efficient, since it does not take the short cut via the *list_line* attribute.

The development of AQL programs requires a knowledge of the **attributes** defined for each object type.

Some attributes are common to all object types (e.g. *name*, *number*, *deleted*, *sons*). Other attributes are specific to the object type (e.g. *x1*, *y1*, *x2*, *y2* for objects of type *line*).

The most important attributes are those of the objects *UDO*, *layer* and *group*, by means of which the model is structured. The entire model can be accessed via these attributes. Examples are the attributes *list_all*, *list_<object type>* and *list_out*.

The Online Help "Attributes" contains a list of all attributes together with the object types concerned.

4.15.2 Accessing Attributes

An identifier consists of a symbol name followed by a list of attribute names. The symbol name and the individual attributes are separtated from each other by full stops. The identifier is evaluated by taking the attribute of the variable (the first in the list), then its attribute in turn (the second in the list), and so on until the whole list has been processed.

Example:

```
obj.par_p1.name
```

is evaluated by first taking the *par_p1* attribute from obj and then the *name* attribute of the resulting attribute.

The fields of the identifier may be constants (*son, name*), as above. They can also be string expressions. In this case the expression must be enclosed in square brackets. The first part of an identifier is not allowed to be variable.

Example:

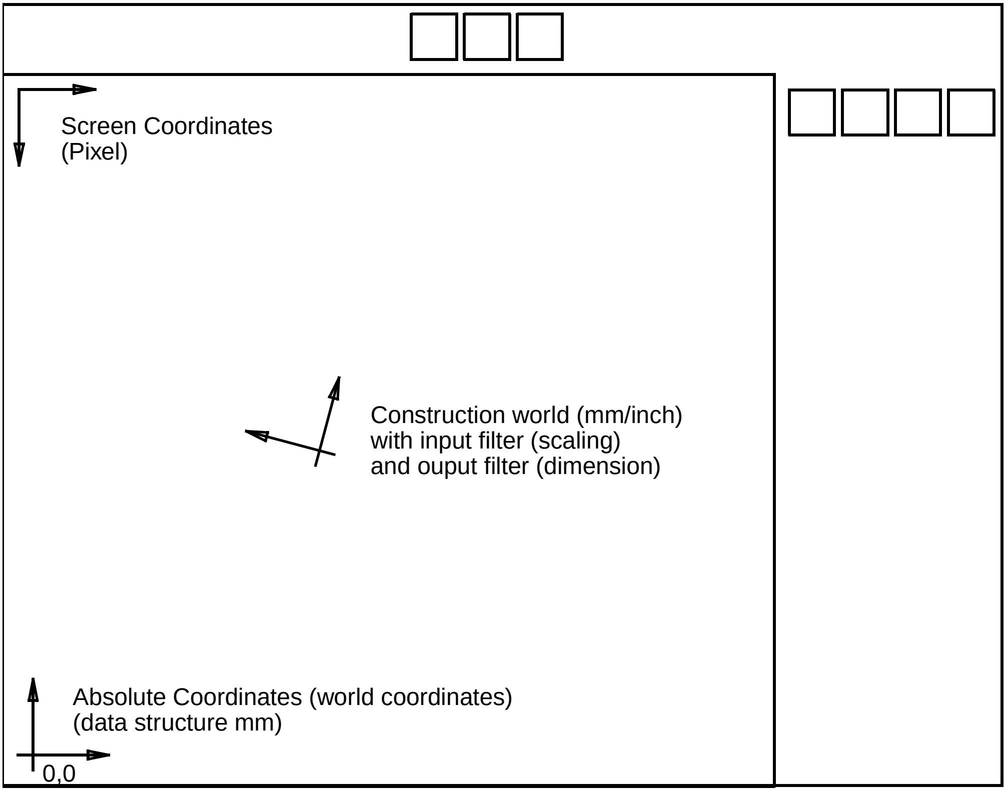
```
obj.[i.type].color
obj.[i.type + ”_list”]
```

4.15.3 Accessing User-defined Attributes

When a user-defined attribute is accessed, this must be prefixed with the string *user_*. This ensures that the user-defined attributes do not overwrite the predefined attributes (see *user_<attrib>* and *attribs_user* attributes belonging to the object type Bea_object, Online Help – "Attributes").

4.16 Screen Format

A number of AQL functions return or process screen coordinates. These are stored in a group {x,y} where *x* and *y* are measured in screen pixels from the upper left of the drawing area.


Screen Coordinates
(Pixel)
Construction world (mm/inch)
with input filter (scaling)
and ouput filter (dimension)
Absolute Coordinates (world coordinates)
(data structure mm)
0,0

4.17 Graphics Input

4.17.1 Identification and Selection of Objects – pick, pick_group, pick_rectangle, pick_multiple objects

The AQL function *pick*, the AQL function *pick_group* and the AQL function *pick_rectangle* allow the user to select elements from the screen with the mouse, i.e. to pick. The argument is either an object type identifier or a group of object type identifiers. This limits the object types to be picked. The following results are obtained:

`pick`	object
`pick_group`	group of objects
`pick_rectangle`	set of objects put together with the menu for selection in a rectangle. The argument is a group of permissible object types.
`pick_multiple_objects`	pick_multiple_objects group with *boolean* and set of objects.*boolean* tells whether the selection was transformed or canceled. The set of objects can be put together with the multi selection mode. The arguments are a group of permissible object types (as *node_type*) and a group with already selected objects.

```
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>} )
```

The **first argument** of a pick function may be an UDO or a layer. In this case the pick function is restricted to the user element or layer concerned.

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

As its **last argument** the pick function can contain a group of screen coordinates which simulate the user selection at this position.

Example:

```
p = pick(”point”, {200,250})
```

This function identifies a point nearest to the screen coordinates x = 200, y = 250.

Example for *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 )
```

These functions cannot be used in batch mode. This causes an abnormal termination.

4.17.2 Reading a Mouse Selection – pick_mouse

The function waits for a mouse input in the active window and returns a group with the screen coordinates.

```
group = pick_mouse()
```

Example:

```
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
```

The function may not be used in batch mode.

4.17.3 Outputting the Cursor Position – peek_cursor

The function outputs the current cursor position.

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

The function may not be used in batch mode.

Example:

```
// 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 Coordinate Transformation to Screen Coordinates – world_to_screen

The function returns a group with the screen coordinates.

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

4.17.5 Coordinate Transformation to World Coordinates – world_from_screen

The function returns a group with the world coordinates.

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

Example:

The following example demonstrates interaction of the three functions *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 Transforming a Rotary Coordinate System – trans_from_world

The function transforms a rotary coordinate system into a horizontal coordinate system.

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

`<pin>`	Group of values to be transformed	
	Input type Cartesian:	{x y}
	Input type polar:	{r alpha}
`<world>`	New coordinate system	
`<inputtype>`	Input type:	Cartesian (default) polar
`<outputtype>`	Output type:	Cartesian (default) polar
`<result>`	AQL group of transformed values (see pin)	

4.17.7 Rotating a Coordinate System – trans_to_world

The function ttransforms a horizontal coordinate system into a rotary coordinate system.

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

`<pin>`	Group of values to be transformed	
	Input type Cartesian:	{x y}
	Input type polar:	{r alpha}
`<world>`	New coordinate system	
`<inputtype>`	Input type:	Cartesian (default) polar
`<outputtype>`	Output type:	Cartesian (default) polar
`<result>`	AQL group of transformed values (see pin)	

4.18 Graphics Output Control

4.18.1 Graphics Output – redraw, flush, update_mode

The AQL function ***redraw*** may be used to cause production of a new drawing.

```
redraw()
```

The AQL function ***flush*** empties the graphics system buffers and brings the screen up to date:

```
flush()
```

The user decides whether the model should be redisplayed after each change with the AQL function ***update_mode***:

```
update_mode(true) /* all changes will be displayed immediately */
update_mode(false) /* changes will only be displayed after
                                                        redraw */
```

4.18.2 Resetting a Viewport – view_reset

The function resets viewports.

```
view_reset ()
```

The function may not be used in batch mode.

4.18.3 Translating a viewport – view_translate

The function translates all viewports.

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

The function may not be used in batch mode.

4.18.4 Display of Objects – show_normal, show_highlighted

The object specified or all objects in the group specified are displayed anew. No check is made whether a displayable object is involved, i.e. whether the object is in a displayable layer. In this case the object is first displayed in the background color.

```
show_normal(group_or_object)
```

is then used to draw the object in the color assigned to it.

```
show_highlighted(group_or_object)
```

draws the object in a highlight color.

Example:

```
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 Functions – lower_window, raise_window

The ***lower_window*** function puts the ObjectD output window in the background so that it hides no neighboring window. The x and y coordinates of the window remain the same.

```
lower_window()
```

The ***raise_window*** function puts the ObjectD output window in the foreground. Its x and y coordinates remain the same.

```
raise_window()
```

These functions may not be used in batch mode.

4.19 Groups (Data Structure Groups) – group_abs

Groups in the CAD model (as opposed to internal AQL groups, which only exist while the AQL program is actually running) are generated with the AQL function *group_abs*. The first argument is a symbol of type group (AQL group). All elements of the group must be objects.

Only implemented for compatibility reasons. New: **group_create**

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

4.20 UDOs/UDAs – user_symbol, user_outofstring

UDOs/UDAs are entered in the model with the AQL function *user_symbol* or *user_outofstring*. The first argument contains the file name of the user element. The remaining arguments are regarded as parameters of the UDO/UDA.

The functions are only implemented for compatibility reasons. Use the name of the action, e.g. <my_action> (par1, par2, ...) for UDAs, and for UDOs generated as absolute objects <UDO_name>_absolute (prop1, prop2, ...).

5 Predefined Functions

This chapter is subdivided as follows:

- Testing and Conversion of Data types
- Basic Functions
- Interactive Interfaces
- Special Functions
- System Commands
- Functions not Supported in AQL

5.1 Testing and Conversion of Data Types

5.1.1 Type Determination – type

This function returns the type of an expression in string format. The following **return values** are possible for predefined functions:

- real
- integer
- string
- boolean
- date
- group
- invalid/valid
- Bea_object
- Bea_action
- node_type

... and all application-defined types

For objects of a model the type name of the object concerned is returned.

```
x = type(expression)
```

5.1.2 Retrieve Object Type – object_type

The type of the object is output

```
object_type (<object>)
<object>   Object
```

5.1.3 Test for Validity – valid

The predefined function *valid*(*expression*) tests whether an expression can be evaluated. If so, the *boolean* value "true" is returned, if not, the value "false". All variables are initialized to have the value "invalid". The *valid* function tests whether a variable is initialized or whether an attribute of the object is in existence.

The valid test corresponds to the expression type test for "invalid":

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

It is advisable to use the *valid* test, which has been optimized for this purpose.

5.1.4 Test for Empty Groups – empty

This function tests whether a group is empty or not. If the group is empty, the *boolean* value "true" is returned, if not, the value "false". If the argument is not a group, an error message is output and the function returns "invalid".

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

Example

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

5.1.5 Determine a Group Element or Character from String – index

This function returns the nth element of a group. The first argument identifies the group. The second argument must be an integer value.

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

The function can also be used for transferring the nth character of a string, in which case the first argument must be a string.

Example

```
index( {1, ”abc”, 50.0},2) => ”abc”
index(”abc”,2) ==> 98 (i.e. ASCII-code for ”b”)
```

5.1.6 Date Function – date

This function has no arguments and returns the date.

```
x = date()
```

5.1.7 Convert to String – string

All symbols may be converted to a symbol of type *string*. This is done with the predefined function *string*. The most general form of this function has four arguments:

- the value to be converted to a string
- the minimum length of the resulting string
- an optional precision specification
- format specification string.

The specifications for minimum length and precision are described in the chapter "Formatting of Alpha Outputs" on page 3-1.

The format string contains the code letters to be prefixed to *precision* in the output function.

Example:

```
number = 1234.12345678
x1   = string(number,10,2,”f”)
x2   = string(number,-10,2,”f”)
x3   = string(number,10,2,”E”)

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

Result:

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

5.1.8 Convert to Uppercase – uppercase

The function returns the string in uppercase.

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

5.1.9 Convert to Lowercase – lowercase

The function returns the string in lowercase.

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

5.1.10 Create Substring – substr

The function returns the substring between *pos_start* and *pos_end* (inclusive).

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

5.1.11 Seek String – pos

Specifies the position of the first occurrence of the search string in the string to which the search applies. The search starts at the beginning.

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

`<string1>`	String to which search applies
`<string2>`	Search string
`<result>`	Integer

Specifies the occurrence of an AQL symbol in a group.

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

`<group>`	Group
`<aql_symbol>`	AQL symbol
`<result>`	Integer

5.1.12 Seek String – rpos

1. Specifies the position of the last occurrence of the search string in the string to which the search applies.

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

`<string1>`	string to which the search applies
`<string2>`	search string
`<result>`	integer

2. Specifies the position of the last occurrence of the symbol in the group to which the search applies.

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

`<string1>`	group to which the search applies
`<string2>`	AQL symbol
`<result>`	integer

Examples

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

5.1.13 Convert from String According to Translation Rule – translate, translation_table

The *translate* function converts a string according to a translation rule which has been predefined with *translation_table*.

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

`<group_of_group_of_integer_and_string>`	Group of any length made up of individual rules: these must be combined in a group and given the ASCII codes of the character to be converted and the string to be replaced.

Example:

```
translation_table ({{96,”aa”}{97,”boe”}})
translate (”abba”)
```

converts the string *abba* to the string *aaboeboeaa* by applying the translation rule. *a* (ASCII code 97) becomes *aa* and *b* (ASCII code 98) becomes *boe*.

5.1.14 Convert to Real – real

An integer or string symbol may be converted to a real symbol with the predefined function *real*.

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

5.1.15 Convert to Integer – int, round

A real or string symbol may converted to an integer symbol with the predefined function *int*:

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

Real numbers lose their decimal places on conversion. If the conversion is to be accompanied by rounding, the AQL function *round* is used.

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

5.1.16 Convert from ASCII Code to String – chr

An integer symbol in the range 0...255 (ASCII code) is converted to a string.

Example: `chr(97) => ″a″`

5.2 Basic Functions

5.2.1 Determine an Absolute Value – abs

The function returns an absolute value

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

5.2.2 String Length – len

This function returns the length of a string or the number of group elements (single-level).

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

5.2.3 Determine String Length in mm – get_world_length_of_string

The function *get_world_length_of_string* specifies the length of a string in mm.

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

`<string1>`	Input string
`<string2>`	Font
`<real1>`	Font height
`<real2>`	Font width

5.2.4 Define nth Element in a Group, nth Character in a String – set

Defines the nth element in a group.

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

`<group>`	Group
`<integer>`	Position
`<symbol>`	Object

Defines the nth character in a string.

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

`<string>`	String to be edited
`<integer>`	Position
`<string>`	Character (1 only)

5.3 Mathematical Functions

5.3.1 Trigonometrical Functions

The following functions have *real* arguments and return a *real* value, provided that the argument lies within the scope of the function.

```
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)
```

The ***sin***, ***cos*** and ***tan*** functions require an angle to be given in degrees. If an overflow occurs during ***tan*** calculation, the value invalid is returned.

The ***asin*** and ***atan*** functions return a value between -90 and +90 degrees.

The argument of the functions ***asin*** and ***acos*** must lie within the range -1<=expr<=1. If this condition is not met, the value "invalid" is returned.

The ***acos*** function returns a value between 0 and 180 degrees.

The ***atan2*** function returns the arc tangent of y/x. By using the characters x and y, this may be returned as a value between -180 and +180 degrees.

5.3.2 Logarithms

The following functions have real arguments and return a *real* value, provided that the argument lies within the scope of the function.

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

The functions *log* and *ln* require a positive argument. If the argument is negative, the value "invalid" is returned.

5.3.3 Exponential Function

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

The following function has *real* arguments and returns a *real* value.

5.3.4 Square Root

The following function has *real* arguments and return a *real* value, providing the argument is within the value range of the function.

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

The function ***sqrt*** requirse a positive argument. If the argument is negative, the value "invalid" is returned.

5.3.5 Modulus Function – mod

The modulus function requires two arguments. These must be *real* or *integer* values. The function returns the remainder after dividing the first argument by the second. If both arguments are *integer* values, the result is also an *integer*. Otherwise the result is a real number. An error causes the value "invalid" to be returned.

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

5.3.6 Minimum/maximum Functions – min – max

These functions process lists of arguments. The lowest (*min*) and highest (*max*) values from the list are returned.

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

5.3.7 Generate Random Number – random

Generates a random number between 0 and 1

```
random ()
```

5.4 Interactive Interfaces

5.4.1 Prompt Function – prompt

The function *prompt* is used to cause a prompt to be output. This is done via *stdout* in Batch and via the text area in interactive working.

```
prompt (prompt_expression)
```

The contents of *prompt_expression* are output before every input.

Example:

```
prompt(”here the output is in the text area”)
```

5.4.2 Message Output in the Message Window – prompt_comment

Causes a message to be output in the message window.

```
prompt_comment (<string>)
<string>   Message text
```

5.4.3 Message Output – display_error

The specified string containing a message to the user is output in a screen window. The window disappears after an acknowledgement.

```
display_error(<message>)
```

Example:

```
display_error(”Line1%Line2%Line3%”)
```

This function may not be used in batch mode.

In the <*message*> string, the **%** character is interpreted as a line feed. There must be at least one % character at the end of the <*message*> string.

5.4.4 Input Function – read

The function *read* may be used to read an input line from the text area (in batch from the ObjectD window). The argument must be a *string* and written in the form of a prompt "???". The input line must have the syntax of an AQL expression.

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

Example

```
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 may contain the default value at the end, separated by the `‘|‘` character; the default value can be accepted simply by pressing <RETURN> .

```
<prompt> ‘|‘ <default value>
```

If there is more than one **|** character, the remainder are included in the default value as letters.

The function *read* reads the data type entered by the user:

```
”asdf” ->  string
12     ->  integer
3.4    ->  real
```

5.4.5 String Input Function – read_string

The *read_string* function reads a string from the text area (in batch from the process window). In contrast to the *read* input function, the string need not be input in double quotes " ".

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

Example

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

The default value can be appended to the "*prompt_string*", using a `’|’` character to separate it, and then confirmed simply by pressing <RETURN>.

```
<prompt> ‘|‘ <default value>
```

5.4.6 Read a Text block – read_textblock

The function opens a special input form and requests text input from the user. The argument *filename* is required. If the specified file already exists it’s content will be displayed in the form.

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

5.4.7 Read a Key Depression – read_key

The program waits until the user makes an input via the keyboard.

```
<key> = read_key()
<key>   string containing input character
```

Example

```
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
```

The function may not be used in batch mode.

5.4.8 Read Keyboard Input – peek_key

This function reads a key depression and returns it as a string. If no keys are pressed, an empty string is returned instead.

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

Example

```
// 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
```

The function may not be used in batch mode. It does **not** wait for a key to be depressed.

5.4.9 Define Function Key – add_function_key

The function *add_fun_key* defines a function key.

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

`<string1>`	Function key
`<string2>`	Modifier
`<string3>`	Action name

5.4.10 Read Value – read_value

The Function reads a value from the text area or drawing area. A form will be opened with several values in order to get your input. Not all values have a possibility for input, e.g. "circle_rec".

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

`<node>`	Node of value
`<initial>`	Initial value (optional)
`<prompt>`	Input prompt (optional)

5.4.11 Selection from Two Options – popup_boolean

A form offering two options is displayed and awaits a user input.

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

`<title_string>`	string with form title (note space available in the form)
`<true_choice>`	string describing one option
`<false_choice>`	string describing the other option
`<users_choice>`	logical variable with the value "true" or "false," according to which option was selected.

This function cannot be used in batch mode.

5.4.12 Selection from Three Options – popup_3choices

A form offering three options is displayed and awaits a user input.

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

`<title_string>`	string with form title (note space available in the form)
`<first_choice>`	string describing the first option
`<second_choice>`	string describing the second option
`<third_choice>`	string describing the third option
`<users_choice>`	integer with the value 1, 2 or 3, according to which option was selected.

This function cannot be used in batch mode.

5.4.13 Selection from a List of Options – popup_list

A form offering the available options is displayed and awaits a user input.

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

`<title_string>`	string with menu title (note space available in the form)
`<choice_list>`	group containing the strings of the individual options
`<users_choice>`	selected string.

This function cannot be used in batch mode.

5.4.14 Selection from a List of Options – popup_largelist

A form offering the available options is displayed and awaits a user input. The output string is the option selected in the popup menu.

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

`<title string>`	string with form title (note space available in the menu)
`<button string>`	string of "*abort*" menu field exit menu after selection)
`<group elements>`	group of strings to be contained in the popup menu.

Example

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

This function cannot be used in batch mode.

5.4.15 Form with Multiple Choice – popup_multiplelist

A form containing a list of the available options is displayed and awaits a user input.

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

`<string1>`	Title
`<string2>`	OK button
`<string3>`	CANCEL button
`<boolean>`	Multiple choice allowed
`<group of strings1>`	Preselection
`<group of strings2>`	Entries
`<result>`	Selected entries

Example

```
ala_cart = {”vermicelli soup”,
            ”vegetable soup”,
            ”egg soup”,
            ”roast pork”,
            ”escalope”,
            ”roast veal hormones”,
            ”vegetarian’s delight”,
            ”blancmange”,
            ”ice cream”}
menue    = {”vermicelli soup”,
            ”roast veal hormones”,
            ”blancmange”}

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

The function may not be used in batch mode.

5.4.16 File name Input – popup_filename

The form to input file names is displayed and awaits a user input.

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

`<iprompt>`	string containing the prompt and sepcifying the application of the file to be selected
`<choice_list_spec>`	string containing the specification (with wildcards) for the files to be made available for selection
`<initial_dir>`	string containing the directory in whcih the files to be selected are to to be sought; an empty string ("") is interpreted as the working directory "."
`<initial_filename>`	string containing the default file name to be selected
`<open_mode>`	string containing the access mode of the file; the following inputs are possible:

"r"	read:	open for read only;
"w"	write:	open to set up and write; if the file already exists an error message is output with a prompt for a new file name.
"o"	overwrite:	open to write and set up if not already in existence
"a"	append:	extend; the file must already exist.

<filename>	name of file selected (absolute or relative path name).

Example: A file handling example can be found on "Example of File Handling" on page 7-4.

The rules for using the file selection form apply to inputs (see Volume *exploring*, chapter on "File Selection form").

This function cannot be used in batch mode.

5.4.17 Color Selection – popup_color

The color selection form is displayed and awaits a user input.
`<color> = popup_color(<back_ground_allowed>)`

`<back_ground_allowed>`	No effect; merely preserved to ensure upward compatibility
`<color>`	selected color in RGB group

This function cannot be used in batch mode or with monochrome (black and white) screens.

5.4.18 Input of Spline Degree – popup_degree

The form for specifying the spline degree is displayed and awaits a user input.
`<selected> = popup_degree(<initial>)`

`<initial>`	initial value (e.g. "2" for spline of 2nd degree)
`<selected>`	selected value.

This function cannot be used in batch mode.

5.4.19 Input of Decimal Digits – popup_digits

The form for specifying the number of decimal places is displayed and awaits a user input.

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

`<initial>` initial value e.g.

if "14":
1: do not suppress trailing zeros
4: four decimal places

Example: 123,2000

if "3":
3 decimal places
3: selected value

`<selected>` ausgewählter Wert

This function cannot be used in batch mode.

5.4.20 Interface Selection – popup_interface

The interface selection menu is displayed and awaits a user input.

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

`<selected>` selected value

`<initial>` initial value of the type "*formattype*"
(see Online Help – "Parameters and Values")

This function cannot be used in batch mode.

5.4.21 Language Selection – popup_language

The language selection form is displayed and awaits a user input.

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

`<selected>`	string containing selected language.

This function cannot be used in batch mode.

5.4.22 Line style Selection – popup_linestyle

The line style selection form is displayed and awaits a user input.

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

`<selected>`	selected line style.
`<styletype>`	default line style of the type "*styletype_rec*" (see Online Help – "Values")
`<broken_on_off>`	logical value indicating whether the user is allowed to select other than continuous lines (= true) or not. If this parameter is omitted, selection of other than continuous lines is allowed.

If the function is invoked without parameters, the default value {"solid" 3} is proposed.

This function cannot be used in batch mode.

5.4.23 Selection of Drawing Size – popup_size

The form for selection of drawing size is displayed and awaits a user input.

`<selected> = popup_size()`

`<selected>`	selected size

This function cannot be used in batch mode.

5.4.24 Selection of Format – popup_format

The format selection form is displayed and awaits a user input.

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

`<selected>`	selected value
`<initial>`	initial value of the type *formattype*_rec (see Online Help – "Values")

5.4.25 Selection of Fonts – popup_font

The form of character sets is displayed.

`<font> = popup_font(<initial_value>)`

`initial value:` string "Helvetica"

This function cannot be used in batch mode.

5.4.26 Input of Fill Characteristics for a Plane – popup_plane

The form for *filling planes* is displayed.

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

`<initial>`	initial value of the type *planetype_rec* (see Online Help "Values")

Example

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

This function cannot be used in batch mode.

5.4.27 Assign Function Key – define_fun_key

The function *define_fun_key* assigns an action to a function key.

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

`<string1>`	Function key
`<string2>`	Modifier
`<string3>`	Name of action

5.4.28 Reset Function Keys – reset_fun_keys

Resets all function key assignments.

```
reset_fun_keys ()
```

5.5 Special Functions

5.5.1 Quit Program – quit_application

This function terminates the program.
`quit_application ( )`

5.5.2 Set Optional Parameter or Property – set_optional_on

The function defines the selected parameter or the selected property as optional.
`set_optional_on (<node>, <string>)`

`<node>`	node
`<string>`	value

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

`<node>`	node
`<node>`	object

5.5.3 Reset Optional Parameter or Property – set_optional_off

Resets the initial value of a parameter or a property.
`set_optional_off (<node>)`

`<node>`	node of parameter or node of property

5.5.4 Query if Parameter or Property is set optional – optional_set

Queries if a parameter or property is set optional.

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

`<node>`	node of parameter or property
`<result>`	true if the parameter or property is set optional

5.5.5 Object Seek Function – search

The *search* function seeks an object by name within the data structure. This function is provided for compatability purposes only, since string expressions which are enclosed in square brackets can be parts of identifiers. The argument must be a string. The function returns the object if it can be found. If there is no object with the name specified, the value invalid is returned.

```
p = search(”prim_” + p2.name)
```

5.5.6 Seek Object by Name or ID Number – search_obj

Seeks an object by name or ID number in the specified user area.

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

`<string>`	Name or ID number
`<user>`	user (optional)

5.5.7 Point Seek Function – search_point

The function seeks a point in a group of objects within a specified tolerance with middle point (*x,y*) and radius (*epsilon*). The function returns the object if found. If no object is found, the value "invalid" is returned.

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

`<group g>`	group of objects	
`<double x>`	x world coordinate	
`<double y>`	y world coordinate	
`<double epsilon>`	tolerance	
`<boolean nearest>`	true:	the nearest point (x,y) is selected
	false:	the first point found within the tolerance is selected.

Example

```
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 Color Assignment for Objects – color, layer_set_color

A color is assigned to the specified object.
`color(object,<colortype>)`

A color is assigned to the specified layer.
`layer_set_color (layer,<colortype>)`

`<colortype>`	The value range of this option lies between "-64" and "+ 64".	
	`A negative value`	means that if the layer to which the object belongs possesses a certain color, this color is also assigned to the object concerned.
	`A zero`	causes the object to be asigned the standard color for its type.

5.5.9 Retrieve Layer Color – layer_get_color

Returns the color of a layer.
`layer_get_color (<layer>)`

5.5.10 Definition of Parameter Icons – inpar_icon, inpar_iconascii

The specified input parameter is assigned an icon; the contents of the icon are specified in hexadecimal.

Only implemented for compatibility reasons. New: **uda_define/udo_define**

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

`<icon>` Each pixel of an icon with 48 lines of 48 pixels is represented by one bit of a hexadecimal number. If the bit is not set, i.e. its value is "0", the pixel concerned is displayed in the background color; if the bit is set, the pixel concerned is displayed in the foreground color. All formats other than hexadecimal are ignored. Addressing is as follows:

Byte 1:	pixel	1 to 8	in line	1
Byte 2:	pixel	9 to 16	in line	1
...				
Byte 6:	pixel	41 to 48	in line	2
Byte 7:	pixel	1 to 8	in line	2
...				
Byte 288:	pixel	41 to 48	in line	48

The highest valued bit is always assigned to pixel n*8+1, the lowest valued to pixel n*8+8.

The specified input parameter is assigned an icon whose contents are in character form.

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

`<iconascii>` Each pixel of an icon with 48 lines of 48 pixels is represented by one character. If the character is "0" or " " (blank), the pixel concerned is displayed in the background color; for all other values the pixel concerned is displayed in the foreground color. Non-displayable characters are ignored.
Addressing is as follows:

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

5.5.11 Evaluation of AQL Expressions – eval

This function evaluates the AQL expression contained in a given string. If the string contains AQL symbols, these must be defined at the same position as where the function concerned is already invoked.
`<symbol> = eval(<string>)`

Example

```
// 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 External Program Call – exec

The *exec* function executes an external program or system routine. The full path name of the program must be given as a string in the first parameter. All other arguments are converted to strings and passed to the program as parameters. The *exec* function returns an integer value, on successful completion a zero (0).

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

5.5.13 Read Optional Parameter or Property Value – get_optional

Reads an optional parameter value or an optional property value.

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

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

5.5.14 Call Icon Editor – icon_editor

Calls the icon editor.

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

5.5.15 Load Actions of Action Group – load_actions_of_ag

Loads actions in an action group.

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

`<node>`	node of action group

5.5.16 Load Action Groups from Menu – load_ag_of_menu

Loads action groups from the menu.

`load_ag_of_menu (<node>)`

`<node>`	node of menu

5.5.17 Load Objects from Object Group – load_objects_of_og

Loads objects from an object group.

`load_objects _of_og (<node>)`

`<node>`	node of object groups

5.5.18 Retrieve Occupied Memory – memory_use

Shows the amount of memory occupied by the ObjectD process (models, AQL, etc.).

`memory_use ()`

5.5.19 Set Default Parameters for Dimensioning – optional_ digits

Sets default parameters for dimensioning.

`optional_digits (<control>)`

`<control>`	dimension setting

5.5.20 Assign Icon – set_icon

Defines or modifies the assignment of an icon.

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

`<node>`	node
`<icondefinition>`	icon description

5.6 System Commands

5.6.1 Execution of Shell Command Sequences – system

The *system* function calls the UNIX command interpreter *sh* and passes to it the string concerned. The parameters for programs started in this way are less critical than with *exec*, since in the shell the normal command seek sequence applies. Wild cards may also be used, as may the I/O redirection feature. If commands are executed separately with this function, the processing is slightly slower than with the AQL function *exec* due to the preceeding interpetation by the shell.

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

5.6.2 Read and Set Current Working Directory – wd – cd

The function ***wd*** supplies a string containing the path name of the current working directory. The function has no arguments.

```
oldwd = wd()
```

The function ***cd*** changes the current working directory. If successful, the boolean value "true" is returned.

```
ok = cd(”/usr/new_dir”)
```

5.6.3 Output of User Names – who

The function *who* supplies a string containing the name of the user. The function has no arguments.

```
iam = who()
```

5.6.4 Directory Listing – dir

The *dir* function supplies the contents of the directory specified in a group of strings. If no argument is specified, the contents of the current working directory are returned. An optional second argument allows the wildcard specification of the file names to be output. If the directory specified cannot be found or opened, the value "invalid" is returned.

Example

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

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

5.6.5 Output Program Name to Text Area – write_name

The name of the program is output to the text area.

```
write_name()
```

5.6.6 Read Value of Environment Variable – getenv

The function *getenv* reads the value of an environment variable.

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

`<string>`	name of the variable
`<res_string>`	value of the variable

5.7 Functions Acting on the Data Structure

5.7.1 Calculate Estimated Value – calculate_objval

The function *calculate_objval* calculates the value which would be obtained if the action were to be executed.

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

`<string>`	name of the action
`<params>`	list of the necessary parameters

5.7.2 Derive ID Type for Effect Objects – id

Derives the ID type of effect objects.

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

`<effect>`	effect object
`<integer>`	sequential number

5.7.3 Set Object Value – set_objval

Sets the value of an object.

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

`<object>`	object
`<string>`	name of the object value
`<aql_symbol>`	object value

5.7.4 Set Object Value – set_objval_in_user

Sets the value of an object belonging to a particular user.

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

`<user>`	user
`<object>`	object
`<string>`	name of object value
`<aql_symbol>`	object value

5.7.5 Output Group of Subobjects – subobject

Outputs a group of subobjects.

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

`<object>`	object
`<result>`	group

5.8 Functions for Preserving Compatibility

5.8.1 Contour tracing – scetch_makecont

Generates a contour with the contour tracer (only implemented for compatibility reasons, please use **contour_tracing**).

```
scetch_makecont ()
```

5.8.2 Generate Table – tab_one

Generates a table (only implemented for compatibility reasons, please use tab_...).

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

`<string1>`	file name
`<string2>`	name of the table (optional)
`namestr`	name of the table column
`enu_symboltype`	data type of the column entries
`len_typ`	length of the data type string
`univ_ptr`	table entries
`<result>`	table

5.8.3 Reset Zoom – view_unzoomrectangle

Resets the zoomed view in viewports.

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

5.8.4 Zoom – view_zoomrectangle

Zooms in viewports.

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

6 Data Structure – Generation and Retrieval

The installation directory contains an AQL procedure in *#/aql.aql/electric_manual.aql* which creates an ASCII file *electric_manual* in the current working directory.

The file *electric_manual* contains a complete AQL language description taken from system meta information. This information is always complete and up-to-date. It is structured as follows:

- list of types (values)
- list of enumeration types
- list of objects
- list of actions
- list of attributes
- list of events
- list of predefined functions
- list of predefined variables

This file ensures that the most important AQL information is available to the programmer on the computer itself. The file is opened in an editor window. The normal editor functions can be used to search for the desired functions. Syntactically correct programs can be copied from this file.

The following example shows how you can use the *electric_manual*.

The information in the file *electric_manual* is unfiltered information about the ObjectD application. Functions and attributes described in the file but not in the manual are not officially supported and cannot be guaranteed in future versions.

Data Structure – Generation and Retrieval

Example 1: Object generation

You want to generate a line between two points with the parameters *dashdotted* and *thick* (greatest line thickness). The line is to join the points *p1* and *p2* which have already been created.

The syntax convention for the generation actions is as follows:
`<object>_<generation type>`

Procedure

- Use the actions (DEFINED ACTIONS) to find the line generation actions with the pattern l*ine_*.
- Now choose the desired generation type (alphabetical order).
- You generate a line between two points with the first action, the function "*line_pointpoint*".

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

The **parameters** are listed in parentheses after the function name according to type (here *length, styletype_rec*, etc.).

Name of parameter	(here *z*, *spec*, etc.)
Types	(here *length*, *styletype_rec*, etc.)
Possible parameter attributes	(here [*opt*]).

The possible parameter types are:

Parameter type	*Indicator*	*Remarks*
Optional parameter	[opt]	These parameters can be skipped with "," if default values are desired.
List parameter	"star"	The list type is indented. The parameter lists are shown as AQL groups ({..}).
Parameter sequence (within a list)	"sequence"	The parameter sequences are shown as AQL groups ({..})
Alternative parameter	"alternative"	The possible types are further indented. Parameters which requires string types are shown as AQL groups ({..}). The corresponding parameter syntax can be found in the manual "Functions – Values".

You now know that the desired function is called "*line_pointpoint*" and that the first three parameters are optional and may be skipped.

Since the line type is to be dashdotted and the line width 0.5 mm, the **optional parameter *spec*** must be supplied with a different value. This parameter has the type *styletype_rec*. It is described in the section DEFINED VALUES.

```
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 lengths
                                 (0.1 mm pieces)
                 only if pattern == ”userdef”
                 { integer, ... }
```

- Since you want to generate a dashdotted line, this must be the first value of the string.
- Line thickness must be 3. The line thickness numbering corresponds to the sequence within the line group. Alternatively, the line thickness may be specified as an absolute value (e.g. 1.2).

The line group 0.5 is subdivided as follows: 1 = 0.25, 2 = 0.35 und 3 = 0.5

The line group 0.7 is subdivided as follows: 1 = 0.35, 2 = 0.5 und 3 = 0.7 etc.

Result: `line_pointpoint (,{”dashdotted”,3}, , p1, p2)`

Example 2: Object generation

You want to generate a plane from the lines l1, l2 and the arc b1. You have already defined the boundary elements in the program segment. The elements completely define the plane which is to be generated. Additional characteristics are:

- filled with color of index 10
- cross-hatched at an angle of 45 degrees with 2 mm interval
- hidden, with z value 100

Procedure

- Search for generation actions for *plane* in the file *electric_manual* using the pattern *plane_.*
- Choose the plane from single elements. The required AQL function is thus *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              : ellipse
   d                    : length
   a                     : angle
```

- Generate the **first parameter** *z* implicitly as *length_absolute* by entering the value 100 in it (same effect as entry *length_absolute(100)*).
- For the **second parameter** enter *color* with the type *color.*

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

- For the **third parameter** *fill* with the string type *planetype_rec* (Online Help – "Values") enter the string *hatch_cross* for "cross-hatching" and the *boolean* "true" for "fill and hide".
  ```
  fillstyle_rec   { <string:hatch>, <boolean:hidden> }
                    hatch: ”hatch_no”, ”hatch_single”, ”hatch_cross”
  ```
- For the **fourth parameter** with type *star* enter an AQL group "alternative" with the objects (common_type) *line* or *circle*.
  ```
  {l1, l2, b1}
  ```
- Specify the **fifth and sixth parameters** of type *length* and *angle* implicitly (in the same way as *length_absolute* in the first parameter) by entering the value 2 or as *angle_absolute* by entering the value 45.

Result

```
plane_ofelements (100,
                  10,
                  {”hatch_cross”,true},
                  {l1, l2, b1},
                  2,
                  45)
```

Example for Attribute Retrieval

You want to delete all continuous lines from the set of objects in a model. You are therefore looking for a way of identifying these lines. The list of all the lines which belong to a model can be accessed via *top.list_line* (see chapter "Access by Name and Data Structure Traversal" on page 4-25).

Procedure

- Search for *line_type* (line attributes) in the file *electric_manual*. You will find the type `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


       ...
       ...
```

- Search in the displayed attributes for the desired attribute (*constr_type*). You now know that you have to test the attribute *constr_type* for the contents *not_limited*.

Result:

```
del_gp = top.list_line where .constr_type = ”not_limited” delete
(del_gp)
```

7 Example Program

7.1 Example of Data Structures

This example program traverses the ObjectD data structure (see also other AQL programs in the *#/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 */
```

OUTPUT:

```
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 Example of File Handling

The second example shows the interaction of the functions *popup_filename*, *file_access*, *file*, *open*, *close*, *get* und *parse_...*

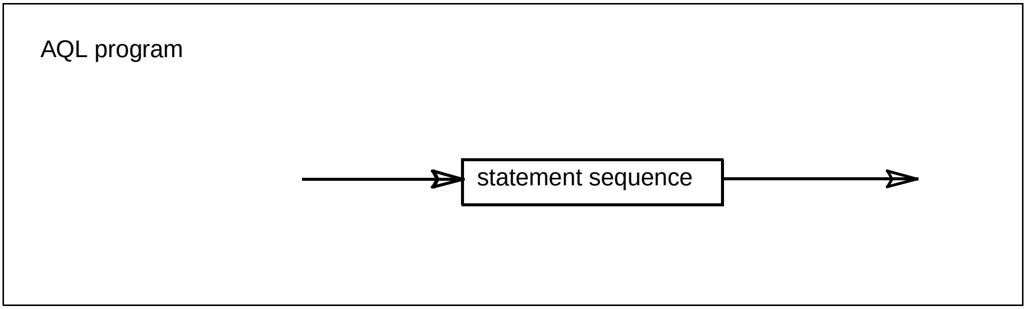
```
// this program reads an existing ASCII File containing
// calculation expressions (like 1 + 5 * 6) and write
// the result of calculation (31) back to a binary file
// comment lines (//..) are ignored
input_filename = ”expression_file.txt”
// test for existance
if not existf (input_filename) then
   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 filename?”,”*.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)
```

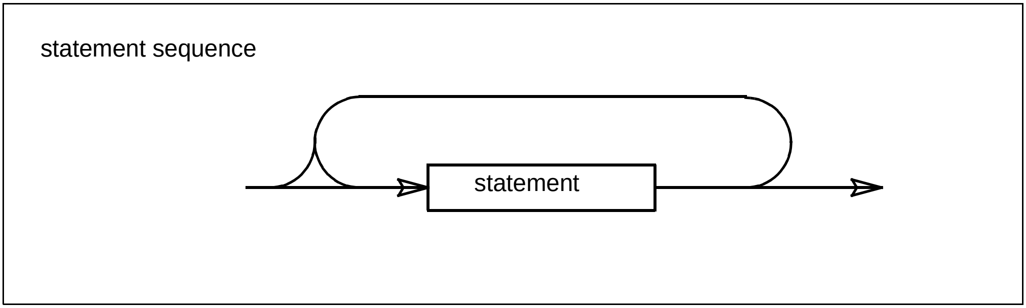
7.3 Example of Character Search in a File

This example program can be used to search for a particular character string in a file with the aid of the functions *parse_keyword*, *line* und *get*.

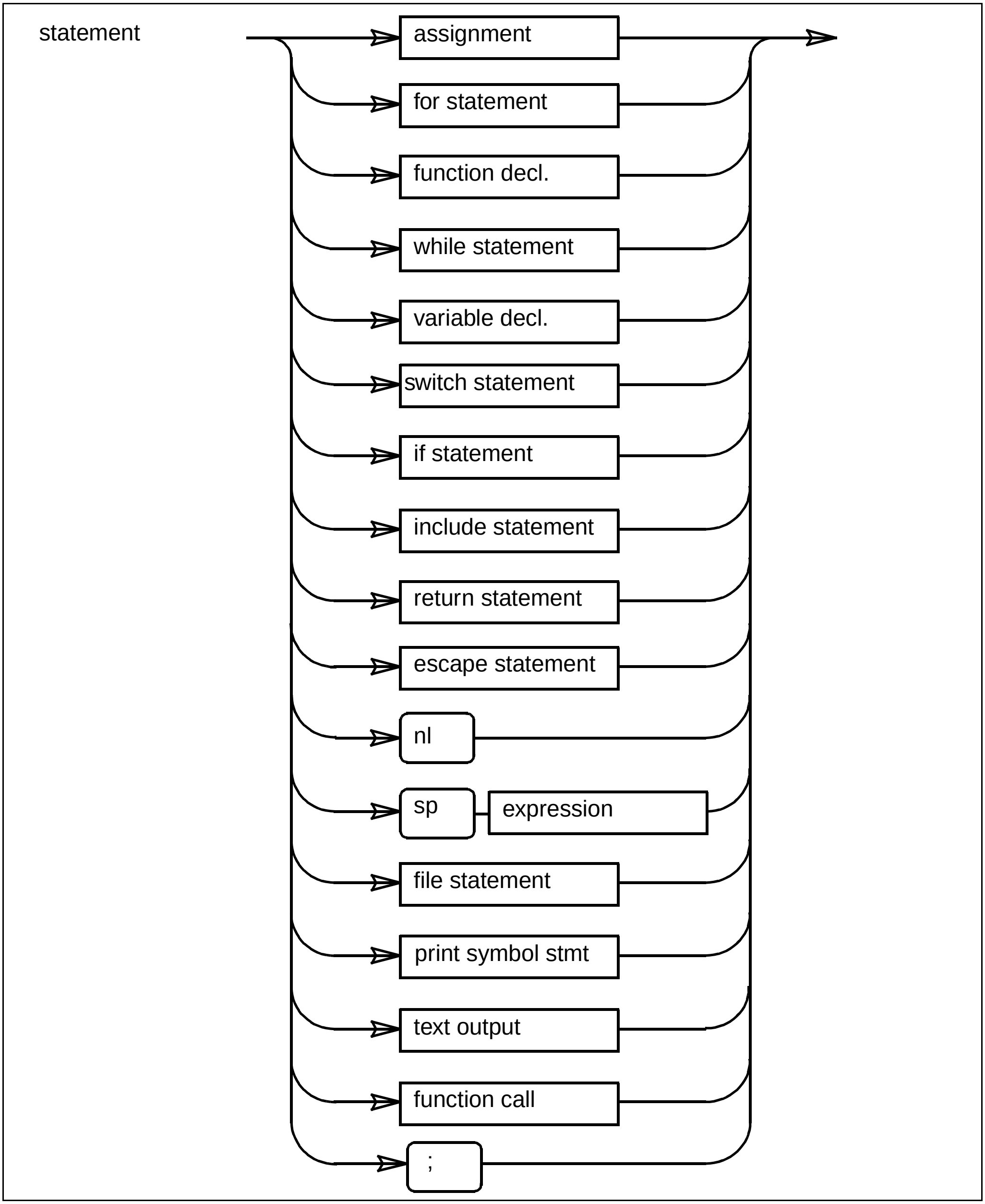
```
//=============================================================
// aql programm to parse inputfile for a certain searchstring
//=============================================================
// preset of variables
count = 0
g = ””
// open file for input
file ”textfile” input
// ask user for search 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 Syntax Diagrams

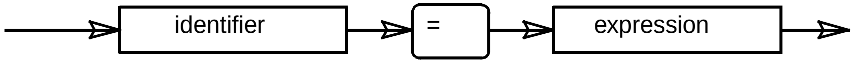




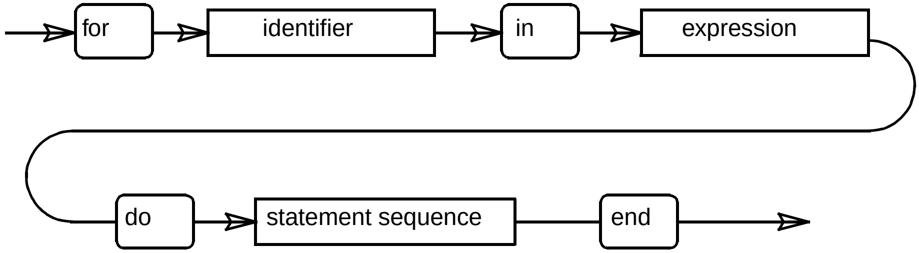
Syntax Diagrams



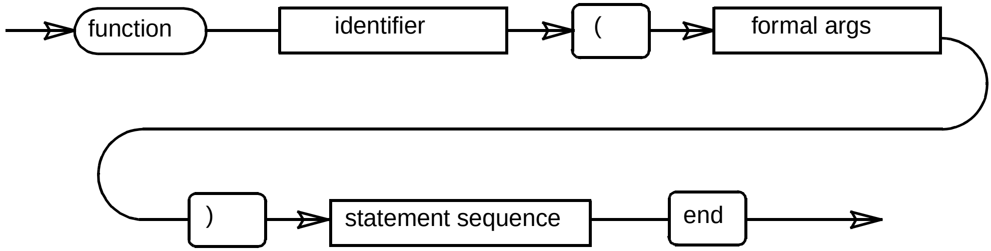
assignment



for statement

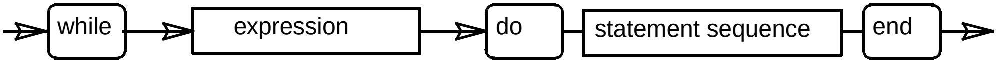


function declaration

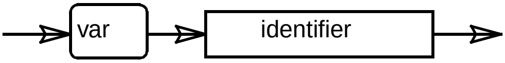


Syntax Diagrams

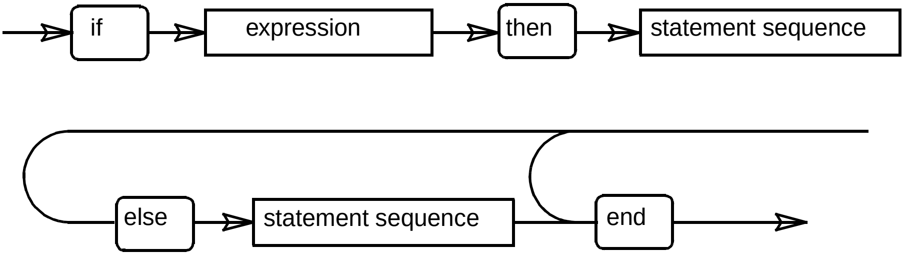
while statement



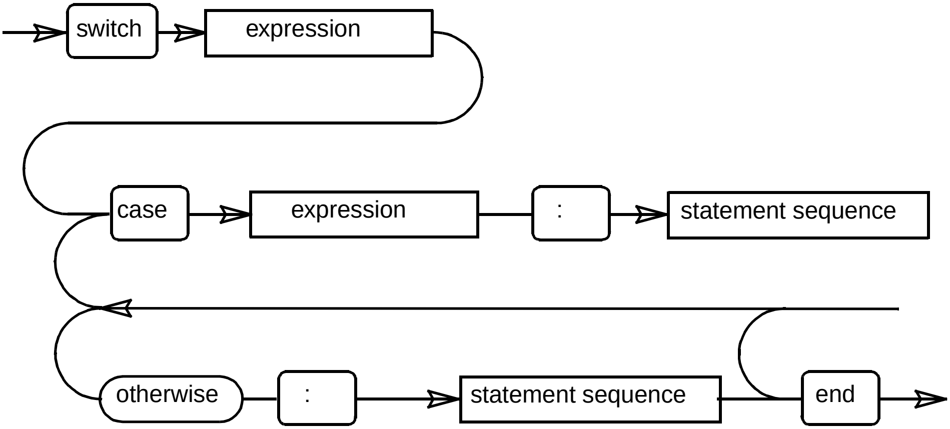
variable decl



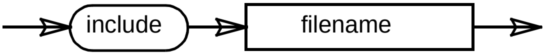
if statement



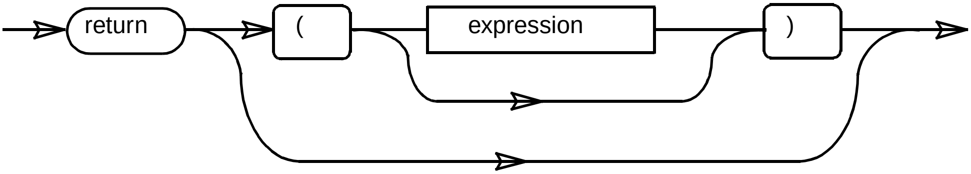
switch statement



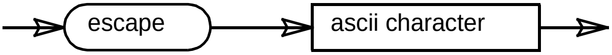
include statement



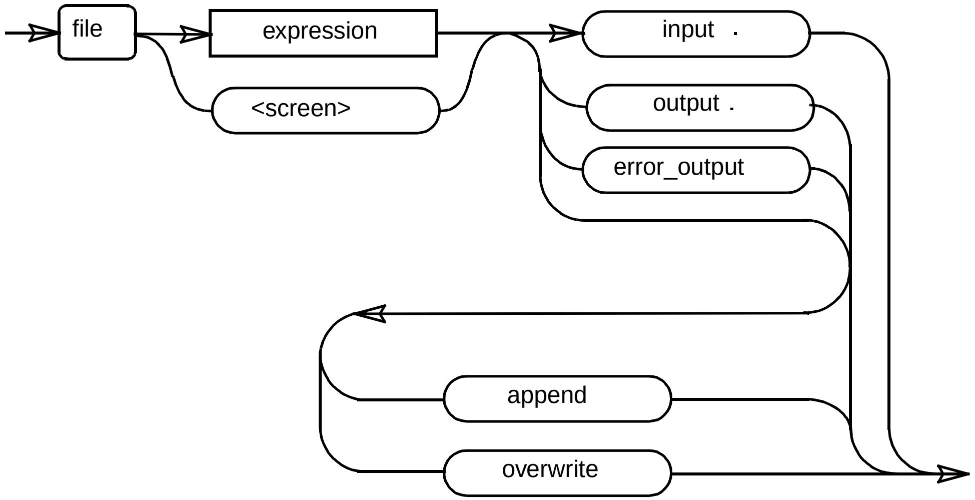
return statement



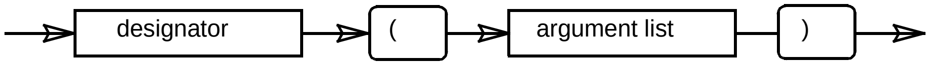
escape statement



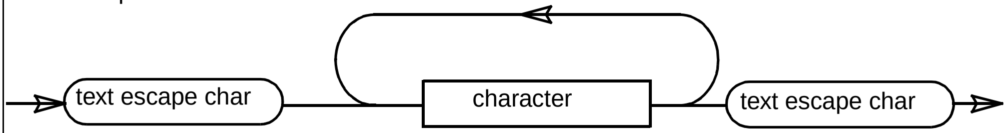
file statement



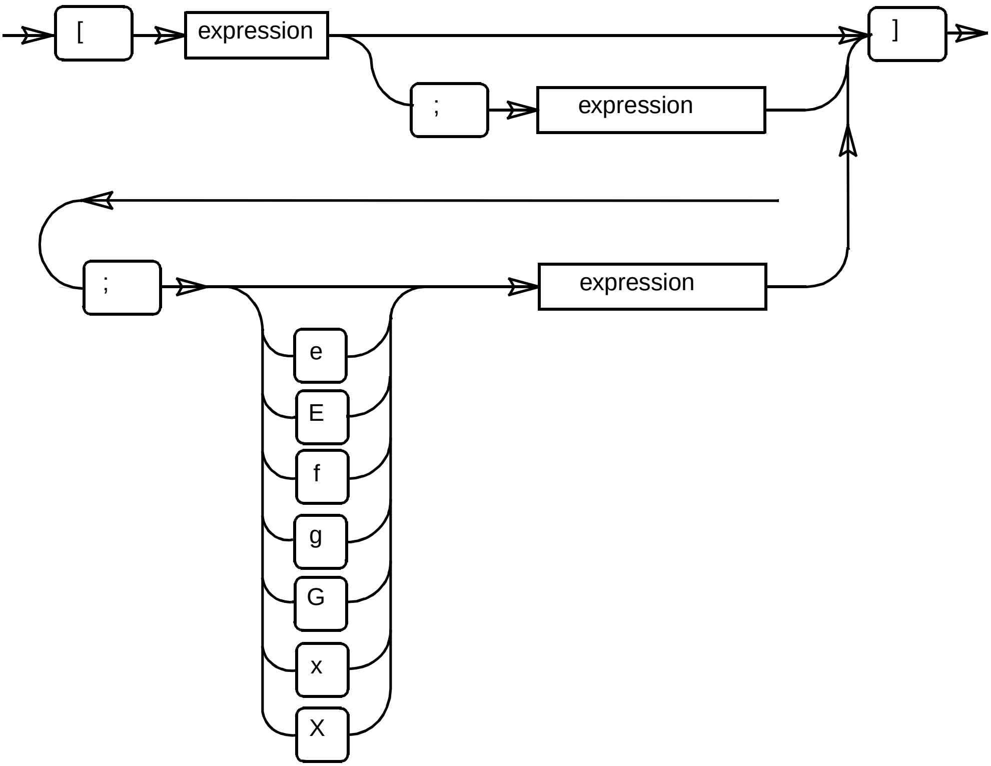
function call



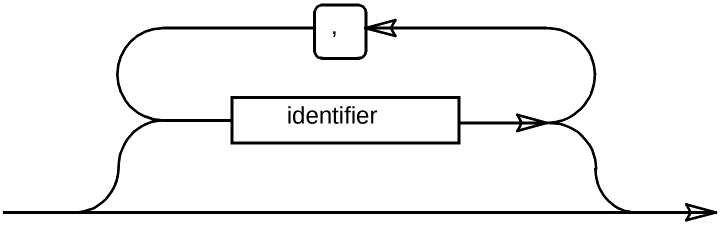
text output



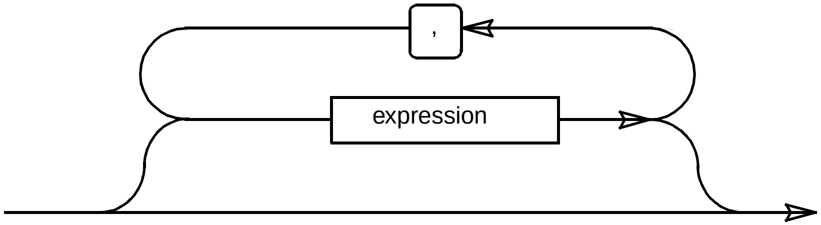
print symbol statement



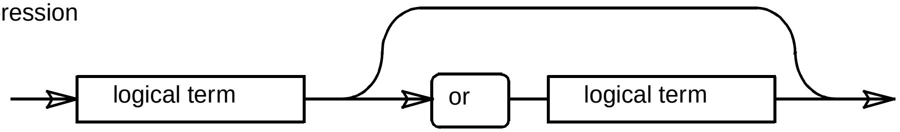
formal args



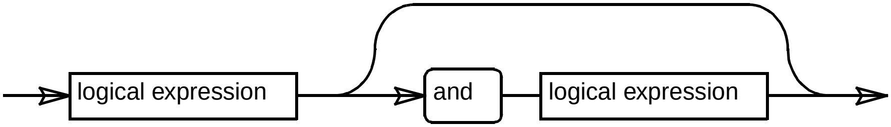
argument list



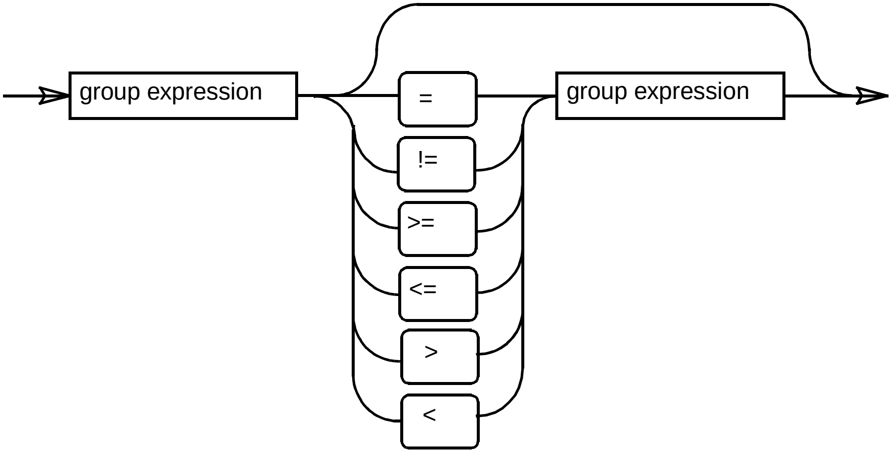
expression

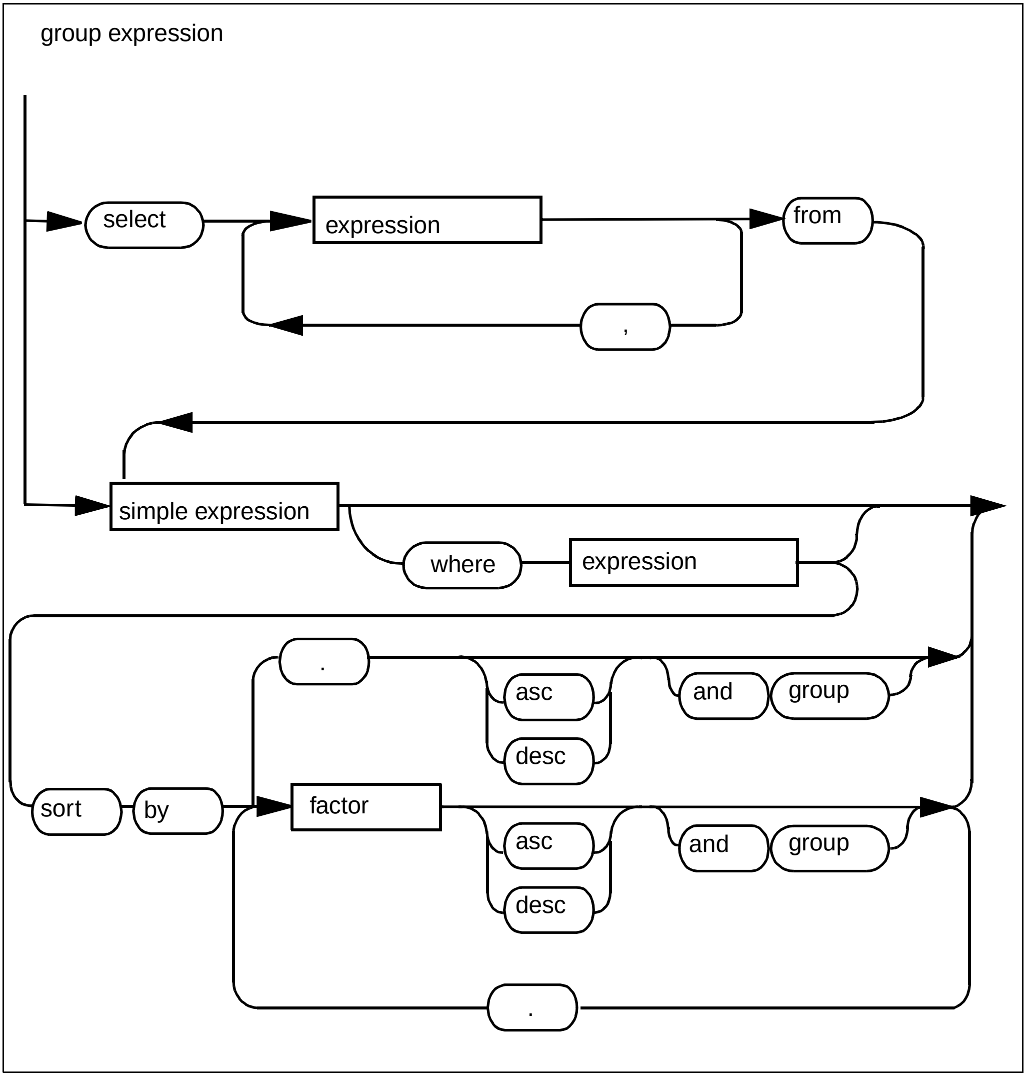


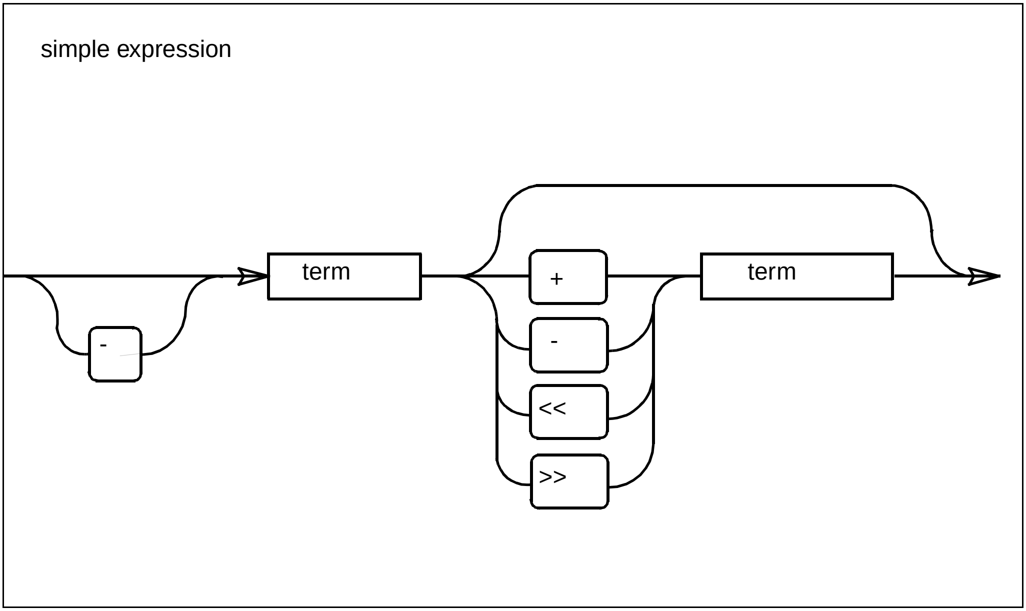
logical term

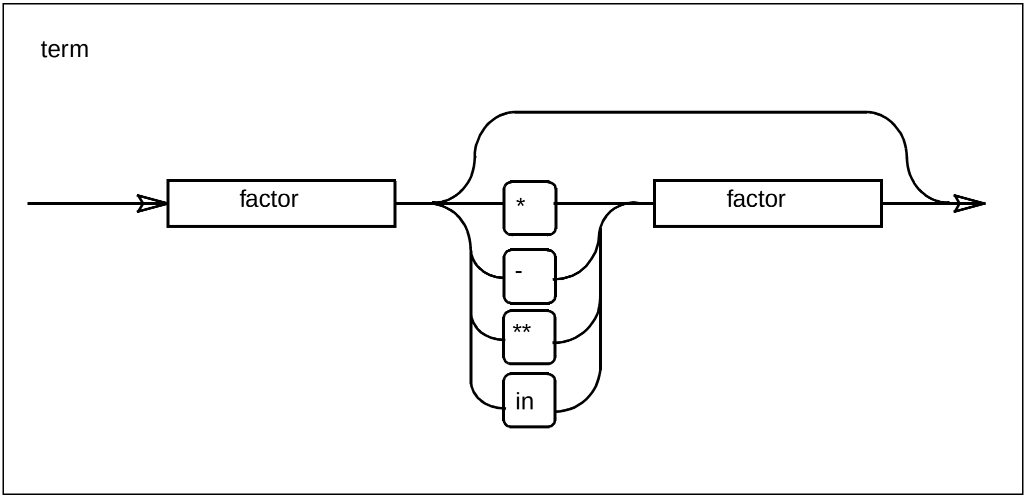


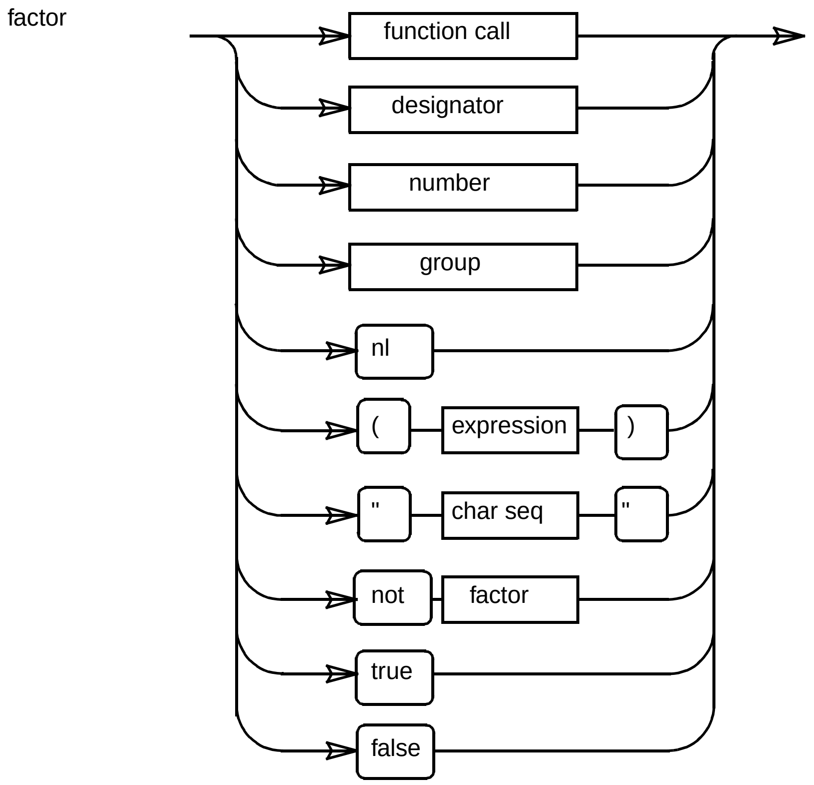
logical expression

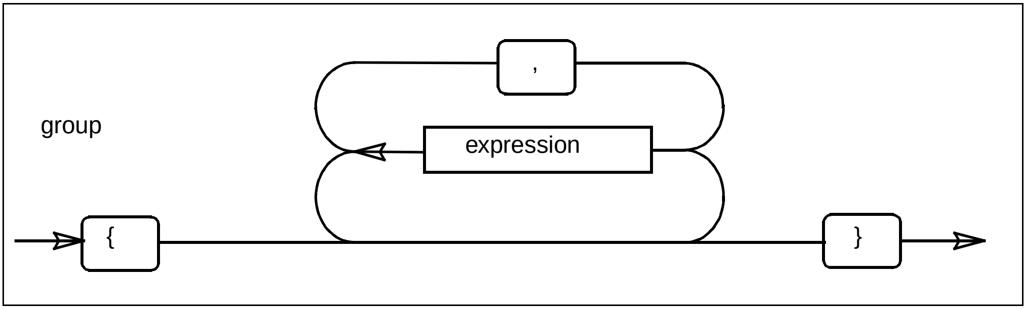


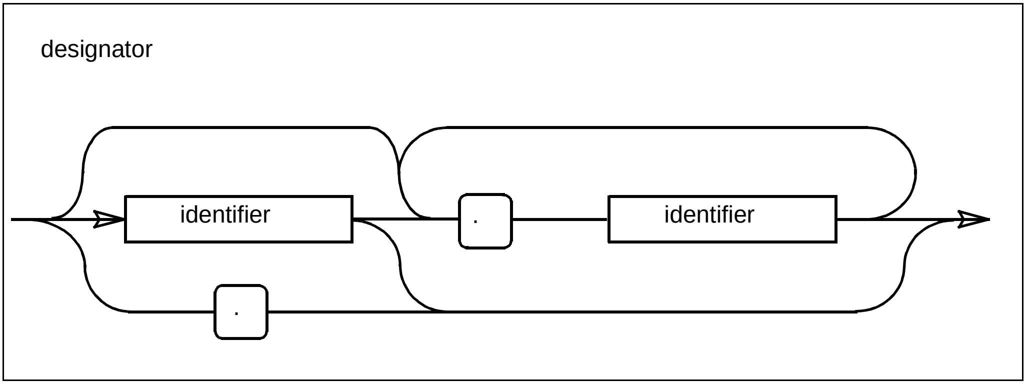

group expression
select
expression
from
,
simple expression
where
expression
sort
by
.
asc
desc
and
group
factor
asc
desc
and
group
.

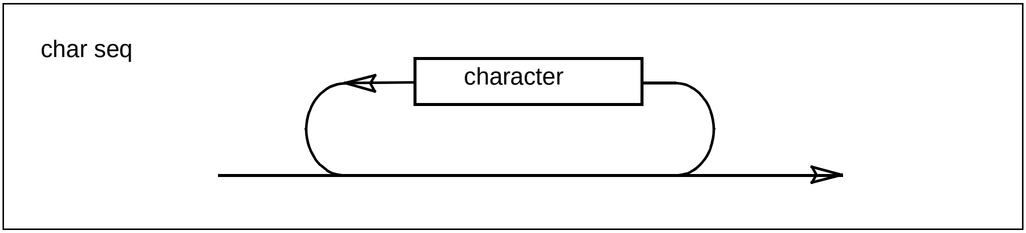

simple expression


term

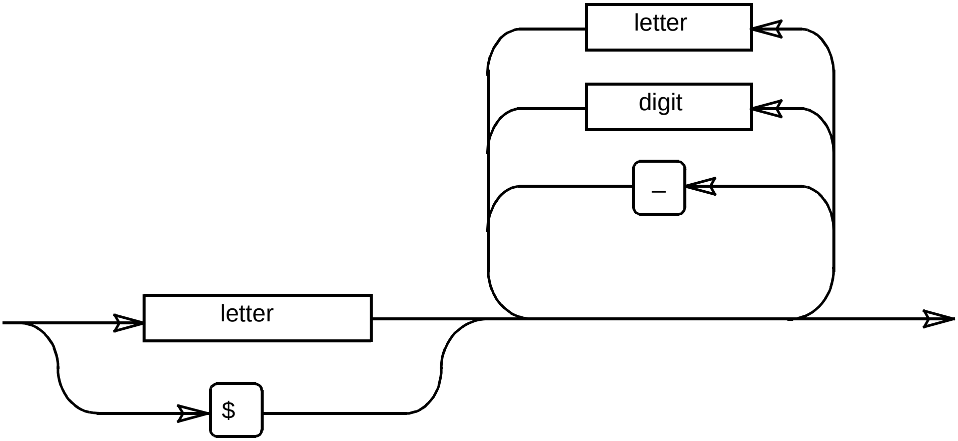

factor
function call
designator
number
group
nl
(
expression
)
"
char seq
"
not
factor
true
false


group
,
expression
{
}

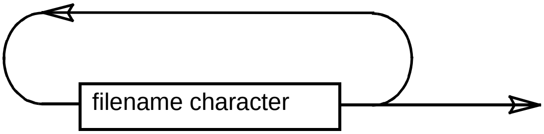

designator
identifier
.
identifier
.

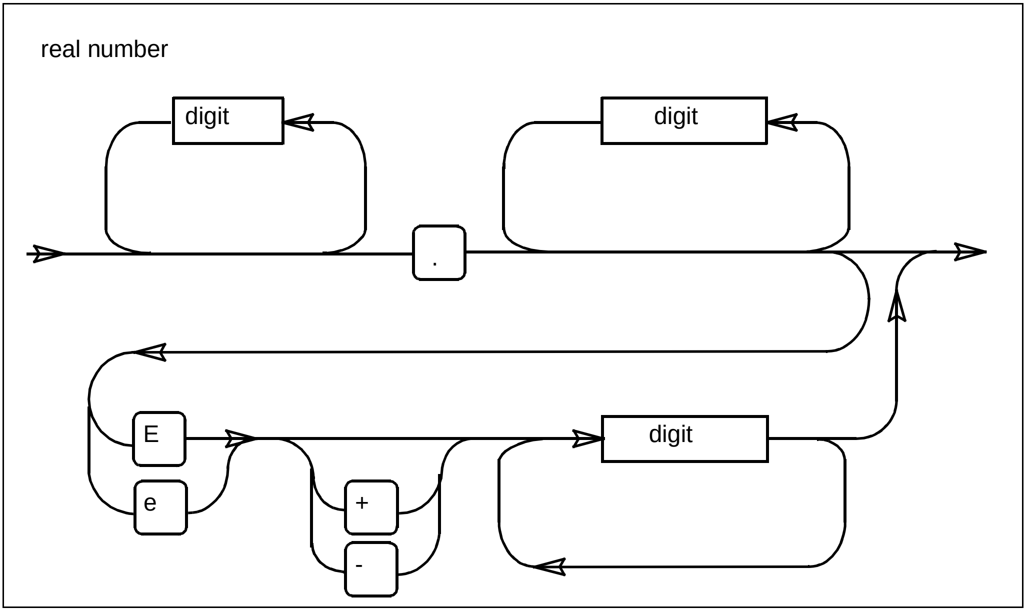

char seq
character

identifier

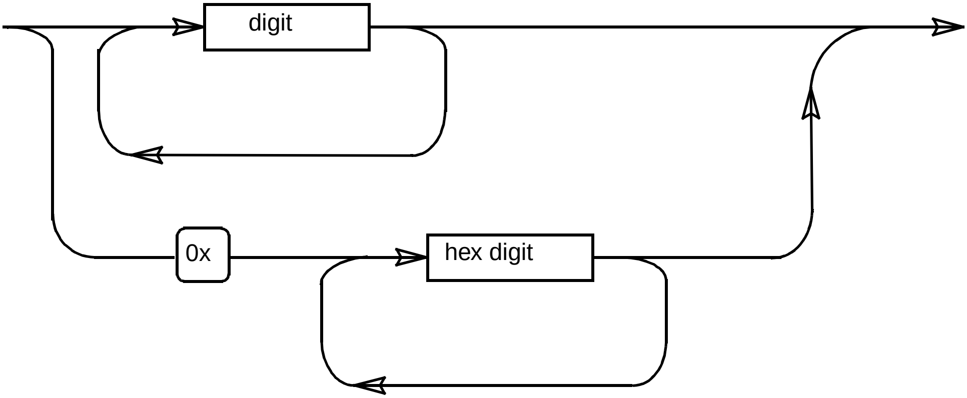


filename

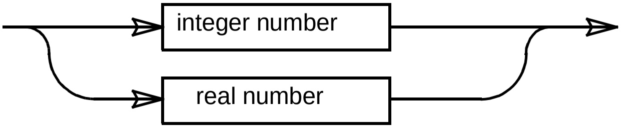



real number
digit
digit
.
E
e
+
-
digit

integer number



number



Index