**Beyond4P**

**Beyond former expectations of**

**Performance,   
 Productivity,   
 Predictability and   
 Professionalism**

**Turning Big Data to Smart Data**

**An unparalleled programming language for  
high performance data and table processing**

**By Georg A. zur Bonsen**

d

d

dd

**Release 7.04 07.04.2020  
A Release on Independence Day**

* **Another performance boost particularly on large and very large tables and data structures**
* **Extended locale support**
* **Supports text and background color on console outputs**
* **Smart tokenizer function provided to parse text for cleanup purposes**
* **Overhaul of 'literal' function and loading HTML tables**
* **Runs embedded B4P code in github markdown documents**
* **Increased support for scientific notation in tables**

**More than 780 functions, incl. 200 table manipulation functions**

All rights reserved, copyright © 2012…2020 by Georg zur Bonsen

Cover photo: Office building with colorful window shades located in Schlieren near Zürich, Switzerland.

|  |  |
| --- | --- |
| **Key Versions** | **3.00 – Introducing Beyond4P as Script Language** (Earlier versions were C++ libraries) |
|  | **4.00 – Unicode** |
|  | **5.00 – Formatting and Style introduced** |
|  | **6.00 – Unleashed 64 bit performance** |
|  | **7.00 – Runs on both LINUX and WINDOWS platforms** |

# Text, File Format and I/O Conventions

This chapter provides a brief overview of supported data types and conventions.

## Directory and File Names

## Definitions

**White spaces:** One or more invisible space symbols including the space bar (ASCII 32) and tabulator (ASCII 9) as well as the non-ASCII control symbols like non-break space. New line symbols (ASCII 13 and 10) are not treated as white spaces. In source codes, literals cannot continue on a following line, e.g. with use of a hyphen.

**Blank:** A blank relates to an empty literal which contains zero characters. Literals containing one or more white spaces are not considered blank.

## HTML Entity References

Tables in supported HTML / MHTML, as well as softquoted literals (referenced in program code with single quotation marks) support entity references. Examples:

&euro; € Euro sign (case sensitive, i.e. &Euro; will not be converted)

&#8364; € ", represented with decimal UNICODE number

&#x20AC; € ", represented with hexadecimal UNICODE number

&#X20ac; € Same. The letters X and of hexadecimal digits are not case sensitive

All *non-break space* characters specified with the entity reference **&nbsp;** (code 160 / Hex A0), are converted to a regular space bar symbol ( code 32 / Hex 20 ). The reason is that various tables use this symbol to separate between words or specify multiple consecutive spaces where needed. The function **decode entities** ( … ) can also decode during entity references during run time.

# Language Basics

## Overall Program Syntax

## Variable Storage Model and Scopes (System, Global and Local Variables)

Beyond4P supports 4 different variable scopes: *System Variables* are pre-defined variables related to specific system functions and are always visible to the program code everywhere. *Global Variables*, also visible everywhere, can be defined in the main program block (not a called user function, procedure or other program file) or inside a global declaration block ( **global** { … } ). Local variables are visible inside a procedure, function and called program code. They will be hidden when another user procedure or function or program is called. *Local variables* will be deleted when returning from a called user procedure, function or program. *Regional Variables* are supported in special circumstances: They are local in the context created, but visible in the called user functions.

Whenever a variable is referenced, then it checks for System Variables, Local Variables, Regional Variables and Global Variables. New variables created in the main program are global (or inside a global declaration block). Otherwise the variables created are local.

|  |  |  |
| --- | --- | --- |
| **Scope** | **Description** | **Sequence, Examples** |
| System Variable | Pre-defined variables containing system related information or able to set specific system parameters. Most of these variables are protected, e.g. read only.  Example: Local settings (e.g. applicable number and date formats)  See section 16 for details. | Read: Check for System Variables first, otherwise  continue with Local Variables.  Write: Same  Create: Not possible.  Delete: Forbidden  Example: version info[ ] |
| Local Variable | Variables declared in a function block or in a program file which was called from a main program with **start**(…) / **include** (…) are automatically declared as local variables.  They are only visible until executing the function or program has ended. All locally declared variables will be removed entirely.  Variables declared in code pieces passed as parameters to specific functions, for example **table** **process**(…), will be local to while this function is executed. | Read: Check for Local Variables second, otherwise  continue with Regional Variables.  Write: Same  Create: In any code location except the main program  code started initially (they are global variables  instead)  Delete: Allowed  When leaving the called function or program,  all local variables will be deleted entirely. |
| Regional Variable | In some functions, access to the local variables of the code which called the function continues to be available.  This feature is insofar useful for code pieces passed into functions like **table process**(…) so these codes can read and modify the local variables of the code which called that function.  Multiple levels of regional variables are possible, e.g. the code inside **table process**(…) calls **another table process**(…). | Read: Check for Regional Variables third, otherwise  continue with Global variables  Write: Same  Create: Not possible.  Delete: Forbidden  (except where created: Treated as local variables) |
| Global Variable | Freely definable variables which can be created in the main program and then accessed from all levels. | Read: Last attempt is to access global variables  Write: Same  Create: Only from main program block.  Delete: Only from main program block,  forbidden elsewhere |

Below you see the variable model. When the main program is started, it works with Global Variables and can access to System Variables. With every further call to a user procedure, user function or other program, the stack builds up with one level for local variables. Calling another procedure / function / program, even the same one recursively, adds another layer on the stack. As long the variables are not regionally accessible, they will be hidden until returning.

|  |  |  |
| --- | --- | --- |
| Prio 2 | Local Variable Space | (used by called function) |
| Prio 3.1 | Regional Variable Space | (accessible by called function, e.g. parameters) |
| Prio 3.2 | " |  |
|  | Hidden Variables | (created by intermediary calling functions) |
|  | " |  |
|  | : |  |
|  | " |  |
| Prio 4 | Global Variable Space | (always visible and accessible) |
| Prio 1 | System Variable Space | (always visible, protections apply) |

## Defining Global Variables outside the main program

Normally, global variables can only be defined in the main program initially started with Beyond4P. Code in called user defined procedures and functions, as well as program code called from the main program using the **start** (…) and **include** (…) calls, can normally declare local variables only which will be discarded when returning to the calling code.

In order to prevent this, a special purpose function called **global** is available to declare a subsequent statement or code block to define global variables in case variables are defined. This feature is insofar useful if called functions or programs should define global variables which shall then be preserved for later use.

**global**

{

global variable 1[] = 'This variable is visible anywhere.';

global variable 2[] = 'This one, too.';

}

Alternative use, e.g. if only 1 global variable needs to be defined. Don't forget the empty parentheses for clean separation of function name **global** and the following variable name to assign.

**global** **()** global variable 1[] = 'This variable is visible anywhere.';

Assuming following code is not in a main program:

a[] = 1234; // Defines local variable a[]

**global**() a[] = 357; // Defines global variable a[]

a[] += 1000; // Adds 1000 to local variable

**global**() a[]+= 1000; // Adds 1000 to global variable

Global A contains 1357 and local A contains 1234.

Deleting global variables outside the main program also requires the global function:

**global**() **delete**(a[]); // Deletes global variable a[]

## Structures

Structures can be built up dynamically by defining member (child) variables as well as further sub-members. They can be manipulated individually and removed, given the applicable protection settings to not forbid this. Every member carries the similar type of context information like base variables (variable name, value, protection settings, optionally further member variables).

Member variables are referenced with a member name inside the brackets. Similar to variable names, the member names must be literals, too.

The number of member variables is only constrained by the available memory space in the system.

Member (sub-member) variables may be created even if a base variable does not exist yet. In this case, the base and precedent member variables are automatically created and assigned with void values.

Example 1:

|  |  |
| --- | --- |
| // You can do the following  // five assignments in any  // order. The structural  // outcome of variable a[]  // with its members and sub-  // members will be the same.  a[] = Ha;  a[e] = He;  a[o] = Ho;  a[u,y] = Huy;  a[e,y] = Hey; |  |

Example 2:

{

complex number[real] = 3;

complex number[imag] = 4;

// Calculate the magnitude and assign to the base variable

complex number[] = **sqrt**( complex number[real]\*complex number[real]  
 + complex number[imag]\*complex number[imag] );

b[] = complex number[]; // Will only transfer the calculated magnitude  
 // and not the members.

c[] <== a[]; // Will transfer all contents (transaction)

**echo**("A. magnitude of ", a[real], " and ", a[imag], " is ", a[] );

**echo**("B. magnitude of ", b[real], " and ", b[imag], " is ", b[] );

**echo**("C. magnitude of ", c[real], " and ", c[imag], " is ", c[] );

// The 2nd echo asserts an error because it cannot find b[]'s members.

}

Declaring a member variable without a parent is allowed. In this case, the value in the parent variable is 'void'. Example:

{

a[b] = 1;

**echo**(a[]); // Will output a void value.

}

## Default Member Variables

Attempting to non-existing member variables will normally assert an exception (error message, program execution will stop). However, a default member (or sub-member, …) variable can be declared so all other references will lead to the default variable. Declaring default member variables is easy: Assign an empty literal (regardless if using single or double quotation marks) and the default variable will always be referenced if the matching member name is not found.

Example.

{

age[''] = "Don't know";

age[Jim] = 35;

age[Jan]= 36;

**echo**("Ages of Jim: ", age[Jim], ", of Jan: ", age[Jan],

" and Jane: ", age[Jane]);

}

The output will be:

Ages of Jim: 35, of Jan: 36 and Jane: Don't know

Following structure is also valid. It is insofar useful if you define 2-dimensional matrix with most values assumed zero.

default matrix value['',''] = 0;

--

One restriction applies: A default for base variables is not supported. In the example below, the assignment works well, but variable x[ ] will result to an error.

''[] = Default value; // Yes, a blank variable name works and is unique in this language

**echo**(''[]); // Outputs "Default value"

**echo**(x[]); // Sorry, will not refer base variables to default values.

## Referencing Structures with Numbers

Alternatively to member names, structures can also be addressed by numbers. Since the list of members is always maintained in an alphabetic order, index 0 refers to the first member in the alphabetic order and index n-1 to the last one.

*Background info regarding keeping the members in an alphabetic order:*

*The alphabetic order is more than just a beautiful clean-up. It allows a bi-sectional search algorithm which allows for a very fast search procedure to match provided names with the given members. The same feature does also apply to base variables and function names. The search duration increases logarithmically with the number of members. For a variable with 128 members, a maximum of 8 checks are needed. For 1024 members, a maximum of 11 checks are needed, and for ca. 1 million members, a maximum of 21 checks are needed.*

Some key rules:

* Numbering starts with 0. For *n* members, *n*-1 is the highest index.
* Numeric references is only possible with numerals (regardless if constant or expression)
* Total number of elements can be retrieved with the function **member count**(…)
* Exceptions are asserted if referencing a member variable outside the range
* Negative numbers allow referencing from the last to the first entry.  
  If the negative number is smaller than accessing to the first entry, then an exception  
  will be asserted.
* Numeric references are also allowed for sub-members (aka grandchildren), etc.
* The relationship of index number to actual members is not fixed and changes when members are added or removed.
* In case a default member variable is defined (with a blank literal), then it will always assume the top position, referenced with index zero.

Example 1: Output 1:

{

names[Wolfang] = Wolfgang Amadeus Mozart; 0: Anne Sophie Mutter

names[Martin] = Martin Luther King; 1: George Washington

names[Anne] = Anne Sophie Mutter; 2: Jack Daniels

names[Jack] = Jack Daniels; 3: Martin Luther King

names[George] = George Washington; 4: Napoleon Bonaparte

names[Napoleon] = Napoleon Bonaparte; 5: Wolfang Amadeus Mozart

Last entry is: Wolfgang Amadeus Mozart

**for** (i[] = 0; i[] < **member count**(names[]); i[]+=1)

{

**echo**(i[], ": ", names[i[]]);

}

**echo**("Last entry is: ", names[-1] );

}

Example 2 (continuation of example 1): Output 2:

{

names[Justin] = Justin Biber; 0: George Washington

**delete**( names[Anne] ); 1: Jack Daniels

2: Justin Biber

**for all variables**( names[], member[], i[] ) 3: Martin Luther King

{ 4: Napoleon Bonaparte

**echo**(i[], ": ", member[]); 5: Wolfgang Amadeus Mozart

}

}

In the two code examples, you may have seen two different forms of **for**-loops which provide the same output.

## Arrays

Arrays are an alternative to structures. The main difference is that arrays do only allow referencing by numbers (index) and not by names.

Arrays can be defined with following functions: **dim**(…), **dim protect**(…), **redim**(…) and **redim protect**(…).

Regarding numeric referencing to arrays, the same key rules as described in section 0 apply.

Additional rules regarding arrays:

* Attempting to access a member with a name will assert exceptions (all array members are unnamed)
* Attempting to create a member with a name will also assert exceptions (mix of array members with and without names is forbidden)
* Structures and further arrays may be assigned to array members as well as structure members (multi-dimensional arrays, nested structures)

Example: Output:

{

**dim**( a[], 5, "-" ); // Define an array with 5 elements 0: -

a[1] = "Hello"; 1: Hello

**for all variables**(a[], value[], index[]) 2: -

{ 3: -

**echo**(index[], ": ", value[] ); 4: -

}

}

Multi-dimensional arrays:

**dim**( a[], {10,10}, 0 ); // A 10 x 10 array, all initialized with 0

**dim**( a[9], 15, 1); // The last "row" in this array will exceptionally contain 15 instead of

// 10 items, and the 5 additional entries are initialized with 1.

Note: The 4th parameter in "**for all variables**" would normally reveal the member name. For arrays, the text "# Array Member #" will be issued.

You may have noticed that the array definition functions (**dim**(…), etc.) do not affect the base variable.

In addition, you may have noticed in 2- and multi-dimensional arrays that the first level members created from the base variable (and further if 3- or more-dimensional array) are also available. If you create a 5 x 4 array with *dim( a[ ], {5,4}, 0 );*, then 5 additional array members (a[0] … a[4]) are also initialized and available. They may become useful for intermediate sums and other convenient features.

{

a[] = The root;

**dim**( a[], {6,3}, Hi); // Defines 6x3 array, which is actually a 6 x 3 + 6

a[1,1] = He; // An element in the 3 x 3 array

a[2] = Ho; // First level member still available

}

Following code is also valid:

{

**dim**( a[], {10, 10}, 0 );

**redim**( a[3], 20, 1 ); // Additional 10 entries initialized with 1

**dim**( a[1,1], {5, 5}, 2 );

}

In this example, a[0] to a[2] as well as a[4] till a[9] contain 10 sub-members each, but a[3] contains 20 submembers. Array element a[1,1] in turn contains an additional 5 x 5 array. You can reference an element with *a[1,1,4,4] = 3*;.

## Indirect Variable References

The intricateness of adding brackets behind variable references has its unique advantage allowing any expression which finally return a literal result which can be used as a variable name.

Example 1:

{

a[] = Name;

(a[])[] = Jim; // Parentheses? Why are they needed here. See below.

**echo**( Name[] );

**echo**( a[][]);

}

On left-hand assignments, expressions containing a variable reference to be read need to be inside parentheses to avoid confusion with the final variable write access. When reading variables (like in the 'echo' shown above, then the parentheses is not needed.

Example 2:

{

member name 1[] = animal;

member name 2[] = plant;

a[member name 1[]] = "rat";

a[member name 2[]] = "dahlia";

// ...

}

## Dynamic Referencing of Variable Members

Indirect variable references are fine when the nesting depth is the same all time. However, in some application cases, you may want to dynamically assign values to members of different depths where the member and submember names are provided in a parameter set.

Beyond4P solves this problem by allowing parameter sets containing literals and numerals to reference with a dynamic number of member / submember names and array indices.

Important to know:

* The parameter set must contain literals (= member names) or numerals (numeric references, required for arrays, permitted for structures)
* Following are equivalent: a[ b, c ] = [ { b }, c ] = [ b, { c } ] = [ { b, c } ]
* Empty parameter sets denotes to no member. E.g. a[ ] = a[ {} ] = a[ {}, {} ].
* Given member level applies if following parameter set is empty, e.g. a[ b, { }] = a[ { }, b] = a[ b ]
* Multiple parameter sets may be included, e.g. a[ {b,c}, d, {d, f} ].
* Parameter elements must be literal or numerals (numeric references) or combination  
  of both, e.g. a[ {1,2,a} ]
* Variables containing parameter sets are allowed, e.g. p[ ] = {x,y}; a[ p[ ] ] equals to a[ x, y ].
* Nested parameter sets, e.g. a[ {b, {c,d}} ] are not allowed.

Example:

{

a1[] = { son 1, granddaugher 1, greatgrandson 1 };

a2[] = { son 1, granddaugher 2 };

a3[] = { grandson 1, greatgranddaughter 1 ];

family[ a1[] ] = Mike;

family[ a2[] ] = Jane;

family[ daughter 1, a3[]] = Tina;

family[ { son 1, grandson 1 } ] = Dave;

family[ {} ] = Mom;

}

The outcome of the above is equivalent to following assignments:

family[ son 1, granddaughter 1, greatgrandson 1 ] = Mike;

family[ son 1, granddaugther 2] = Jane;

family[ daughter 1, grandson 1, greatgranddaughter 1] = Tina;

family[ son 1, grandson 1 ] = Dave;

family[ ] = Mom;

## Referencing Variables: Summary

* Global variables can only be defined in the main application. Called functions and other called applications can access them, but access rights are limited (no deleting).
* The variables may contain structures (1 or more named members) or arrays (1 or more numbered members), but not a combination of both.
* Nested structures and arrays (incl. arrays of structures, structure of arrays) are supported.
* Protection settings can be defined to every member individually.
* The base variable as well as the members may be of different type and the types may change with new values assigned, unless restrictions or protection settings apply (for example on system variables).
* Assignments (for example *a[ ] = b [ ];* ) will affect the actual value in this variable, regardless of presence of structure or array members.  
  - Only the value of b [ ] is copied into a[ ].  
  - Possible members in a[ ] remain unaffected.
* Transactions will affect the member variables. For more details, please see section 6

## References to Variables

In several cases, complex variable with multiple levels, e.g. a[b,c,d,e] need to be referenced repeatedly where every reference checks for the existence of variable and member names a through e. To address this problem, Beyond4P provides a novel features to define *references to variables*. Modern programming languages such as C++, C# and Java provide similar features.

Do not confuse references with other object types such as numerals, literals, parameter sets, etc. The contents referred with references to variables are always the contents in the destination variables.

### Defining and Accessing Simple References to Variables

Code example 1: Simple definition to a reference to a variable.

|  |  |
| --- | --- |
| a[] = Hello;  ref1[] =^ a[];  a[] = Hi;  **echo**( a[],", ",ref1[] );  // Outputs "Hi, Hi"  ref1[] = He;  **echo**( a[],", ",ref1[] );  // Outputs "He, He" |  |

Note that assigning reference to variables to particular variables are only allowed if the new variable name is not yet existing or referring to a variable which has no own members. This is insofar relevant since you can access further members and sub-members of the referenced variable.

Code example 2: Accessing via reference to variable through to a member

|  |  |
| --- | --- |
| // Continued from code  // example 1:  ref1[b] = Ho;  **echo**( a[b],", ",ref1[b] );  // Outputs "Ho, Ho"  a[b] = Ha;  **echo**( a[b],", ",ref1[b] );  // Outputs "Ha, Ha" |  |

Following code example demonstrates how a linkage (reference to variable) can be released and that references to variables can be made on members as well as from members.

Code example 3: Reassigning references to different target variables

|  |  |
| --- | --- |
| // Continued from code  // example 2:  ref1[] = ^a[b];  ref1[] = Hi;  **echo**( a[b],", ",ref1[] );  // Outputs "Hi, Hi" |  |

If a simple copy is made from one reference variable to a new one using the = ^ sequence, then the new reference will also point directly to the target variable.

Alternatively, references can also be created on existing references to variables using the =^ ^ sequence.

Code example 4: Demonstrating 2 ways to copy references

|  |  |
| --- | --- |
| // Continued from code  // example 3:  ref2[] =^ ref1[];  ref3[] =^^ ref2[];    ref3[] = Hu;  **echo**( a[b],", ",ref1[] );  **echo**( ref2[],", ",ref3[] );  // Outputs "Hu, Hu" twice  // Not illustrated: // ref3[] points to a[]  ref3[] =^ a[];  **echo**( a[],", ", ref3[] );  // Outputs "He, He"  **echo**( ref1[],", ",ref2[] );  // Outputs "Hu, Hu"  **release all**;  // Releases all references |  |

### Releasing References to Variables

Following methods are available to release references:

* Function call **release** (…)  
  Specify references to variables directly. Example: release ( ref1[ ], ref2[ ], ref3[ ] );
* Function call **release all**; // Without parameters   
  All locally defined references to variables will be released. When called from the main program (neither function nor other program called), then all globally defined references will be released.
* Function call **release all**( … ) // With parameters  
  Specify references to variables directly, or variables which contain one or more members / sub-members which are references to variables. All of them will be deleted.
* Leaving a user-defined procedure or function, or program code called with **start**(…):  
  All locally defined references to variables will be released.
* Reassigning a reference to variable to a new target variable (as already shown in code example 3 in the previous section).

References to variables released with the function calls **release** and **release all** will not be deleted. The targeted variable contents remain unaffected. Once a target variable is detached from all references, then the target variable may be deleted. The released references to variables will not be deleted. However they become regular variables containing void values.

Code Example 5: Releasing references

|  |  |
| --- | --- |
| // Define reference  ref1[] =^ a[];  // Release reference  **release**( ref1[] ); |  |

### References in Member Variables

Beyond4P allows definition of references in member variables of structures and arrays or combination of both. The only requirement is that the base (parent) variable must not be a reference

Code example 1: References in Member Variables in Structures

|  |  |
| --- | --- |
| // Black part of illustration  a[] = Ha;  a[b] = He;  c[] = Hi;  c[d] = ^a[];  c[e] = ^a[b];  c[f] = ^c[e]; // 🡪 to a[b]  **echo**( c[d],", ", c[e] );  // Outputs "Ha, He"  // Blue part of illustration  a[g] = ^c[];  **echo**( a[g],", ", a[g,d] );  // Outputs "Hi, Ha";  // You can loop around.  **echo**( a[g,d,b] ); // "He".  **echo**( a[g,d,g],", ",  a[g,d,g,d] );  // Outputs "Hi, Ha". |  |

Code example 2: References in Member Variables in Arrays

|  |  |
| --- | --- |
| // Black part of illustration  // Define array with 4 members  array( a[ ],{Ha,He,Hi,Ho} );  a[0] =^ a[3];  a[1] =^^ a[0]; // Ref. to Ref.  **echo**( a[0],", ", a[1] );  // Outputs "Ho, Ho";  // Blue part of illustration  a[0] =^ a[2];  **echo**( a[0],", ", a[1] );  // Outputs "Hi, Hi"; |  |

### Invalid Use

Invalid examples (will cause error messages)

a[] =^ 1; // Right hand side must always refer to a variable.

a[] =^ [Name]; // No constants, no table references, no calculations allowed.

a[] =^ -b[];

a[] =^ b[]{4}; // E.g. linking to an element of a parameter set

Attention to following example:

b[] = Ha; // Define b[]

a[] = ^b[]; // a[] refers to b[]

b[] =^ a[]; // and b[] refers to a[]

**echo**(a[]); // The program will hang up (infinite loop as result of 2 mutual

// references to variables.

### Summary

Key rules on using references to variables:

* References to variables can only be defined on other existing variables, regardless if they are global or local.
* References to variables can be defined in base variables and member variables (arrays and structures) and combination of both.
* Multiple references may be created on the same target variable.
* Referenced variables will be temporarily locked, i.e. access is possible except deleting them. Only if all references on one particular variable are released, then it can be deleted again.
* On variables which do already contain members, references to other variables can only be created on the end points (members which do not have any further members behind). A member variable may point to a more parental position under the same base variable.
* Nested references to variables are allowed, even circular references. However, avoid creating mutual references as they will run permanently in infinite loops.
* References on table entries, constants, calculations or elements in parameter sets (e.g. a[ ] =^ b[ ] {1}) are forbidden.
* Defining a reference is only possible if the new variable name is not existing, or, if existing, it has no own members (in form of a structure or array).
* All variables on which references have been defined will be protected from accidental deleting.
* The functions **release** (…) and **release all** (…) are available to release references. The reference to variable will be deleted, but the target variable stays alive.
* All local references to variables defined inside a called user function or program code will be released when the user function or program has been completed and returns to the calling code.
* Beyond4P forbids any form of "pointer arithmetic" or other manipulations on references intending to access other memory locations.

## Roundup on Variables

The following table summarizes all methods to access variables:

|  |  |  |
| --- | --- | --- |
| **Families** |  | **Description** |
| Simple variable references with literals | abc [ ]  two words [ ]  "Hello World" [ ] | Simple access to variable.  Multiple words, softquoted and quoted literals are allowed.  Numerals, dates, Booleans and voids are not allowed. |
| Simple variable references with expressions | ( "Hello " + World ) [ ] ⬄ "Hello World" [ ] | Calculated expression results must be literal.. |
| Numeric References | Forbidden, e.g. 1 [ ], 1.2 [ ], (1+2) [ ]  Allowed: e.g. "1" [ ], ( Literal(1) ) [ ] | Numeric references are forbidden. Convert to literals first. Boolean and dates as references will be converted to literals automatically. |
| Simple member variable references with literals | abc [ def ]  "Hello World" [ "Hi Folks" ]  Array [ "1" ] | Member variables are also referenced with literals. Dates and Booleans are converted to literals, but numbers are not. See below. |
| Nested member variable references | abc [ def, ghi ] | abc is base variable. def is a member of abc. ghi is a member of def (and sub-member of abc). |
| Numeric references to member variables | abc [ 0 ]  def [ ghi, 2 ]  jkl [ 1, 3 ]  mno [ 1+2, 3+4 ]  jkl[ -1 ]  // Negative counting: Access last entry | A base variable may contain n member variables, numbered 0 … n-1. As long they are existing, the member variables can be accessed with index numbers from 0 to n-1. The member variables are always kept in alphabetical (case sensitive) order and this even applies if further members are added. Rely on the member names and not numbers.  Out of bounds will flag error and stop execution.  *Attention:* It is not possible to reference member variables with numbers directly if they do not exist. |
| Retrieve number of members | member count( abc[ ] ) | This function retrieves the number of members. |
| Indirect Reference | Reading: abc [ ] [ ]  Writing: ( abc [ ] ) [ ]  abc [ def ] [ ghi ] | Retrieve value in variable "abc" and use this one to access target variable.  2nd example applies to write: The first reference is a read and the whole function must be packed into parentheses. The final [ ] or […] initiates the write access.  The 3rd example requires that referenced variable in "abc" contains a branch member named "ghi". |
| Nested Reference | abc [ def [ ] ] | Access "abc" with branch member variable name stored in "def". |
| Default Value | abc [ '' ] = 0;  abc [ "" ] = "Empty";  Read: abc[ 'nonexisting member'] gives 0 | Use an empty literal (use single or double quotation marks, insert no spaces inbetween) to create default variables.  Failed read accesses to non-existing member variables at that level returns zero. |
| Nested Default Value | abc [ def, '' ]  abc [ '', '' ] | Default for particular subscript  Default applicable to 2 levels (if first level has already failed) |
| References to variables | abc[ ] = ^def[ ];  ghi[ ] = ^abc[ ]; | Varible "abc" and "ghi" point to same contents as "def" |
| References to references | abc[ ] = ^def[ ];  ghi[ ] = ^^abc[ ]; | Here, ghi[ ] points to abc[ ] which will then immediately refer to def[ ]. |

Figure: Typical creation of variables.

## Protecting Variables

The function **protect**. (…) is available to change protection settings. This function can only be applied on locally declared variables. For example, protection settings of system variables and global variables (if not at global level) cannot be changed.

All variables created have no protection applied.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Protection setting** | **Full Access** (default) | **Prevent Deleting** | **Limited Access** | **Read Only** | **Locked** | **Tightly Locked** |
| Read variables | ⚫ | ⚫ | ⚫ | ⚫ |  |  |
| Input variable to functions | ⚫ | ⚫ | ⚫ | ⚫ |  |  |
| Write variable | ⚫ | ⚫ | Must be of same type |  |  |  |
| I/O variable to functions | ⚫ | ⚫ |  |  |  |  |
| Delete variables | local only |  |  |  |  |  |
| Read member variables | ⚫ | ⚫ | ⚫ | ⚫ | ⚫ |  |
| Write member variables | ⚫ | ⚫ | ⚫ | ⚫ | ⚫ |  |
| Create member variables | ⚫ | ⚫ |  |  |  |  |
| Modifying members (see 1) | ⚫ |  |  |  |  |  |
| Create arrays | ⚫ | ⚫ see 2 |  |  |  |  |
| Delete member variables | ⚫ |  |  |  |  |  |
| Convert structures to arrays | ⚫ |  |  |  |  |  |
| Transactions | ⚫ |  |  |  |  |  |
| Change protection | local only | local only | local only | local only | local only | local only |
| Check protection with **protect**( check, … ) | ⚫ | ⚫ | ⚫ | ⚫ | ⚫ | ⚫ |

1 Applicable to functions like **array to structure**(…) and **structure to array**(…).

2 Only possible if variable contains no other members so far.

Existing member variables are not covered by this protection since they have their own protection parameters. For example if a member variable has a protection and you apply a transaction on the base variable, the protection setting of the member variable is ignored. To avoid this, follow the simple rule: "*To protect the children, protect their parents, too*".

Only the 'tightly locked' prevents accessing members, regardless what protection settings they have.

Note on "**limited access**" protection setting: The values may be modified, but the type may not. E.g. an existing literal may be replaced by a literal (regardless if quoted, softquoted or not), but not by a Boolean, a numeral, date, parameter set or void value. If the protected variable contains a variable set, then the elements inside the parameter set may be changed freely, e.g. adding, removing and replacing data and changing their types.

Note that in addition to the protection settings, the variables may be temporarily locked, i.e. access is possible except deleting. This applies in following cases:

* An assignment to a variable is made. While the expression on the right hand side is calculated, the target variable is locked.
* A transaction is made. Similar behavior as above.
* The variable is referenced as an I/O parameter to a function
* The variable is locked by a reference from a different variable.

The locking mechanism is based on a counter. In case multiple locks apply, then the variable is released when the last lock is removed.

# Tables

Tables are the main structural objects on which this programming language has actually been developed for, namely processing tables of any size effectively and efficiently with minimum amount of code to be written.

Tables are 2-dimensional data structures which hold all data as literals (text), including numeric data and dates.

Table anatomy:

* All tables are global, i.e. accessible by called procedure and functions, no matter in which code file they are running, and retained if created inside procedure and functions and then returned.
* The table may contain any number of rows, including 0 (= no) row.
* Row counting starts with 0 (row 0 is typically the header row), 1, 2, etc.
* Blank rows are allowed
* Column counting starts with 0, 1, 2, etc.
* The width of every row (i.e. number of columns) may vary.
* The number of entries in a table row may exceed the number of header entries, but in this case these entries cannot be identified by column header names as they are missing. They can only be accessed with column numbers.
* Multiple identical column header names are allowed, but referencing them by names will always match with the first occurrence (from left to right).
* Tables loaded, e.g. from HTML, XML, excel, contain no formatting attributes (e.g. font size, frame and cell background color, alignments, typefaces like bold, underline, etc.). However, before saving tables, you may want to append formatting attributes which will then be parts of the data contents. (See section 11.1 on formatting attributes)

If a table loaded from a file contains header data (column names) on a different row other than row 0, then simple function calls such as **table delete rows** (…) can be used to move the header data to the top row.

In this chapter, all examples refer to the table below:



Example of a table being used throughout this document (Cities.csv).

## Creating Tables

Various functions are available to create new tables:

|  |  |
| --- | --- |
| **Function Name (Selection)** | **Action** |
| **table create** (…)  **table create if not existing** (…) | Creates blank tables (zero rows, zero columns). Existing table will be destroyed.  Here, no initialization if table already exists. |
| **table load** (…) | Loads tables (HTML, MHTML, or Text / CSV format) |
| **table load excel file** (…) | Loads tables from Excel file (.xlsx, .xlsm). Must include "Office Library" to use this function. |
| **table initialize** (…) | Creates a table with initial contents |
| **table configure** (…) | Configures rules how to deal with row contents or column header names not found |

## Accessing Tables



In the introductory part, only **Simple Accesses** will be explained, then followed by other accesses

## Iterators

Whenever partial table specification is allowed, cases apply where the row number is not known. This is typical in function calls where you need to specify a expression to select rows or use functions like **table process** (…). A typical example is if you want to compare the current row with the next one up or down.

The function **row**(…) will provide the current row number.

**row** ( ) and **row** ( 0 ) provide the current iterator. **row** ( 1 ) and higher numbers provide next higher iterators if nesting is applied on partial table specifications.

Example: **table insert above selected rows** ( Table, [Name] != [Name, **row**()-1], 1 );

Compares Name in current and previous row and inserts blank rows whenever the name has changed.

If no partial table specification is possible, or a nested one does not exist, then an exception will happen.

## Automatic data conversions

Data written to tables will always be converted to literals, including Boolean values (translates to English terms 'true' and 'false'), dates (with local or applicable date format) and numerals.

Note: These automatic data conversions are also applied in the function **tokenize** (…)

Following automatic data conversion rule applies if data is read from tables:

* If the data looks like a number (digits, one decimal point, no scientific notation, no  
  thousand separation symbol), then it will be converted to a numeral. The literal representation will be preserved.

The data conversion rules for reading data from tables can be modified by changing the Boolean values in the system variable **table conversion** [ … ].

* table conversion [ numeral ] Default: true
* table conversion [ scientific ] Default: false   
  ( Converts numbers with and without scientific notation (exponents) )
* table conversion [ date ] Default: false   
  ( if true: Converts dates in strict format YYYY-MM-DD or YYYY-MM-DD hh:mm:ss or hh:mm:ss )
* table conversion [ Boolean ] Default: false  
  ( if true: Converts lower-case 'true' and 'false' to Boolean values )
* table conversion [ blank to zero ] Default: false  
  ( if true: Converts blank literals to zero )

These settings apply to all tables. Limiting settings to a particular table is not possible.

See appendix for further details

## Rules on Column Header Names

As mentioned before, the first table row is treated as header row containing column header names. Table specifications using column names checked from left to right until a first match is found. Header names must match precisely. They are case sensitive, sensitive to blank symbols, non-character symbols and foreign characters.

If a column header contains a number, then reference them with literals (e.g. '3', not 3). Otherwise, column 3 from the left (actually 4th column because 1st column is column 0) is accessed instead

Avoiding compilation errors, use single or double quotation marks to reference column headers if they contain special symbols such as "Currency [EUR]", "Speed (km/h)" and "First & Last Name".

## Referencing Table Cells – Special Rules

**Negative row numbers**

Counts form last table row (-1) upwards. Negative numbers *bigger* than total row count will assert errors.

**Negative column numbers**

Counts form last table column (-1) leftwards. Negative numbers *bigger* than total number of columns with headers will assert errors.

**Accessing data beyond last row**

A blank will be read out.

Write case: No exception will occur. The table will be extended accordingly. For the example table used throughout this manual (Cities.csv), the statement **[table:City,1000] = Moscow;** will insert a large number of blank rows so the entire table will contain 1001 rows in total, with "Moscow" in the last row.

**Accessing data beyond last column**

A blank will be read out.

Write case: The table will be extended accordingly.

**Case "Row not found"**

This case applies if a row is identified by contents but could not be found. With the example table, accessing [ table : City, Rome ] returns -1, or when writing, an exception.

The function **table configure**( table name, row not found, *rule* ) provides alternative rules as listed below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Example** | **Rule** | **Output** | **Explanation** |
| [ table : City, Rome ]  (Reading case) | exception  header row  first row  last row  new row | -1  0  1  14  15 | Indicates 'missing row'  Row number of header row  Row number of 1st row  Row number of last row  New row added with column "City" containing "Rome", all other fields are blank |
| [ table : City, Rome ] = Roma;  (Writing case) | exception  header row  first row  last row  new row | ( Exception )  Header row overwritten !  1st row (New Work City) &  Last row (Davos) overwritten  New row, set to "Roma" | Be careful with these write accesses if specific rules are set. Not being careful will spoil the table contents. |
| [ table : City, Rome,  Famous cultural place ] | exception  header row  first row  last row  new row | ( Exception )  Famous cultural place  Guggenheim  Kirchner Museum  ( blank ) | Be careful with these write accesses if specific rules are set. Not being careful will spoil the table contents.  New row added with column "City" containing Rome. |
| [ table : City, Rome,  Famous attraction ] = Coliseum | exception  header row  first row  last row  new row | ( Exception ) | Be careful with these write accesses if specific rules since you assign the "Famous attraction" for Rome to wrong cities.  New row added with column "City" containing Rome as well as its new attraction (2 columns written with a single assignment). |

Meaningful usage of this rule:

* You may want to add bottom "default" row which contains default data, such as "don't know".
* Gently extend the table while working with it
* Get row number 0 if not found, see example below.

Nice example:

if ([table: City, Cologne] = 0) **echo**( "Cologne is not listed" );

**Case "Column not found"**

This case applies if a column is identified by column header name but could not be found. With the example table, accessing [ table : City, Vienna, Famous Person ] generates an exception.

The function **table configure**( table name, column not found, *rule* ) provides alternative rules as listed below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Example** | **Rule** | **Output** | **Explanation** |
| [ table:Famous Person ]  (Reading case ) | exception  first column  last column  new column | ( Exception )  Washington  Delivery drone  Blank | New column called "Famous person" created, but nothing inside yet. |
| [ table : Famous Person, 3 ] = Benjamin Franklin; | exception  first column  last column  new column | ( Exception ) | Indicates 'column not found'  Dangerous. City name overwritten.  Dangerous. 'Moving along' overwritten  Useful. A new column header "Famous Person" will be added on the right hand side of the table and "Benjamin Franklin" added to row 3 (Philadelphia).  Attention: If rows below the new header names contain data (more data entries than headers), then they will be retained. |
| [ table : City, New York City, River ]   = Hudson River | exception  first column  last column  new column | ( Exception ) | Indicates 'column not found'  Overwrites 'City' by river name  Overwrites 'Moving along' by river name  Creates new column called "River" and adds "Hudson River" to the row for New York City. |

**Combination "new rows" / "new column"**

You have the possibility to configure both "row not found" and "column not found" settings. Set them to "new row" and "new column" accordingly. Then you can easily create and/or extend tables in a structured approach.

Program example:

**table load**( table, "Cities.csv");

**table** configure( table, row not found, new row,

column not found, new column);

[table:City,Düsseldorf,Favorite Beer] = Alt;

[table:City,Plzen,Favorite Beer] = Pils;

[table:City,Boston,Favorite Beer] = Samuel Adams;

Output: The table now contains one additional row at the bottom as well as one additional column at the right. A total of 3 cells have been added:

* *Düsseldorf* in the new row, column *City*
* *Favorite Beer* is the new column header name added
* For *Düsseldorf*, the *Alt* is assigned to *Favorite Beer*.

The 2nd statement adds another row for the Czech town *Plzen* and adds its *Favorite Beer* called *Pils*.

The 3rd statement neither adds a new row nor a new column. *Samuel Adams* is added into the existing column for the already existing city *Boston*.

## Memorizing Table Column Numbers (Secretly) for Performance

For the first time a table column is referenced with a constant literal value, then the corresponding column number will be memorized for future references. This is useful in order to accelerate subsequent references, particularly if the tables contain a large number of columns.

|  |  |
| --- | --- |
| **Examples where column numbers are memorized** | **Comments** |
| [ Table 1, Last Name, 5 ] = Thomson  **table process**( Table 1, [Full Name] = [First Name] + " " + [Last Name] ); | The column numbers are resolved only once in this program and re-used for all other table rows. In case this function is called again, then the same column numbers will be re-used (no repeated resolving) |

|  |  |
| --- | --- |
| **Examples where column numbers are not memorized** | **Comments** |
| **table consolidate** ( Table 1, Full Name ); | Column header names provided as parameters to functions will be resolved to column numbers at every call. |
| [ Table 1, (Last Name), 5 ] = Thomson **table process**( Table 1, [( Full Name ) ] = [ (First Name) ] + " " + [)Last Name= ] );  a[ ] = Last Name; [ Table 1, a[ ], 5 ] = Thomson **table process**( Table 1, **echo**( [ a[ ] ] ) );  [ Table 1, Last Name + '', 5] = Thomson; | The literal "Last Name" is part of an expression. Beyond4P assumes that values from expressions may change and the column number will be identified at every function call.  Same applies for variables  Addition of an empty string is an alternative way to form an expression. |

**Possible measures to suppress memorizing column numbers:**

**Option 1: Use expressions instead of literal constants**

Example a: Putting the value into parentheses Example: [ ( Last Name ) ] )  
Example b: Add useless blank literal Example: [ Last Name + '' ]  
Example c: Use a variable Example: [ a [ ] ]  
Example d: Any other form of expression you want to use

This feature is beneficial if only few accesses to the table are made but will slow performance on tables with large number of columns.

**Option 2: Force Beyond4P to forget all memorized column numbers in the currently running program.**

Example: Call **forget memorized table columns** before starting to reference table columns.  
 This procedure call will walk through the currently running program to identify and forget all memorized  
 column numbers.

This features applies on the program file currently executed. Other program files, for example a superior program which started the current program, or included library files are not affected. This feature is useful if you write a function library on your own with various smart table functions. Make sure this function is called in the library file every time you need it, and not just once during code initialization. Calling this procedure make take additional time if the current program file is very large.

**Option 3: Disable memorizing column numbers**

Set the system "variable runtime settings[memorize table columns]" to false to prevent memorizing column numbers, and back to true to enable it. This rule applies on any program file (included called other Beyond4P programs and function libraries) until disabled again. This feature is very convenient and idiot-proof, but will show slightly slower performance on tables with large number of columns because every referenced header name is checked from left to right every time.

Code example:

**table initialize** ( t1, {{ Last Name, First Name }, { Miller, Abel }, { Fuller, Brigitte }} );

**table initialize** ( t2, {{ First Name, Last Name }, { Charlotte, Tanner }, { Danny, Turner }} );

**define procedure**( 1st try, {{ table name, literal }} )

{

**echo**( "1: ", table name[],": ", [table name[]:Last Name,..] );

}

**define procedure**( 2nd try, {{ table name, literal }} )

{

**echo**( "2: ", table name[],": ", [table name[]:(Last Name),..] );

}

**define procedure**( 3rd try, {{ table name, literal }} )

{

**forget memorized table columns**;

**echo**( "3: ", table name[],": ", [table name[]:Last Name,..] );

}

**define procedure**( 4th try, {{ table name, literal }} )

{

runtime settings[memorize table columns] = false;

**echo**( "4: ", table name[],": ", [table name[]:Last Name,..] );

runtime settings[memorize table columns] = true;

}

...

**1st try** ( t1 ); **1st try** ( t2 ); // Function call on t2 lists first names

**2nd try** ( t1 ); **2nd try** ( t2 ); // This one is OK

**3rd try** ( t1 ); **3rd try** ( t2 ); // This one, too

**4th try** ( t1 ); **4th try** ( t2 ); // This one, too

Output: Note the mistake on the 2nd call of **1st try** (…):

1: t1: {Miller,Fuller}

1: t2: {Charlotte,Danny}

2: t1: {Miller,Fuller}

2: t2: {Tanner,Turner}

3: t1: {Miller,Fuller}

3: t2: {Tanner,Turner}

4: t1: {Miller,Fuller}

5: t2: {Tanner,Turner}

If no precautions are taken, you may be wondering why the table column specified was correct but the wrong value has been returned. Taking the right precautionary measures will prevent surprising bugs.

## Horizontal Access (Multiple Cells in a Row)

This novel feature allows reading multiple cells from one row where a range or set of columns need to be specified. The information read is a **parameter set** containing the values from the selected columns. Write access is also supported with parameter sets as well as with scalars (other types).

Full and partial table specification.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **[** | **table name** | **:** | **Column Specifier** | **,** | **Simple Row Specifier** | **,** | **Column Specifier 2** | **]** |
|  | Mandatory  Partial table speci-fication: (Optional) | | Mandatory | Mandatory  Partial table specification: (Optional) | | Optional | |  |
|  | Literal  Note the colon (:) symbol behind table name, do not use a comma. | | Simple Column Specifier  or  Advanced Column Specifier | Literal (looks for 1st match)  or  Positive row number  or  Negative row number | | Simple Column Specifier  or  Advanced Column Specifier | |  |

Following Column Specifiers are supported:

(For partial table specification, the context assumes table named "table" and row 1 )

In the following table, capital letters A and B, etc. refer to simple column specifiers.

|  |  |  |  |
| --- | --- | --- | --- |
| **Category** | **Type** | **Explanation** | **Example (Full and Partial Table Spec.)** |
| **Advanced Column Specifier**  **(ACS)** | : Literal | Set of all columns which match with the literal expression. Don't forget the additional colon!  Wildcards (asterisks, commas and question marks) are supported as long the content is not a hard quoted literal (e.g. inside double quotation marks)  No match case: Empty set is returned | [ table: **: 'C\*'**, 1 ] Partial: [**:'C\*'** ] returns parameter set *{New York City,USA}*  Why? The header names *City* and *Country* begin with a capital C. Wildcard symbols ?, \* and , (comma) are supported.  Following modifiers are available to laxen comparison rules: + Ignore case  [ table  **:+'C\*'** , 1 ]  ~ Ignore blanks  [ table  **:~'C\*'** , 1 ]  and combinations of the above. |
| {A, B, …} | Parameter set containing simple column specifiers (column header names, positive and negative column numbers).  This rule applies even if the parameter set contains only 1 element.  Empty sets are returned if empty sets specified. | [ table: **{ Famous attraction, City}**, 1 ] Partial: [ **{ Famous attraction, City}** ] returns parameter set *{St. of Liberty,New York City}* |
| A..B | Range from column A to B (both are simple column specifiers). If B is before A, then empty set is returned. | [ table: **City..2**, 1 ] Partial: [ **City..2** ] *{New York City,New York,USA}* |
| A.. | Range from column A to last column with header name | [ table: **-2..**, 1] Partial: [ **-2..** ]  *{Guggenheim,Taxi}* |
|  | ..A | Range from column 0 to A | [ table: **..State**, 1 ] Partial: [ **..State** ]  *{New York City,New York }* |
|  | .. | Range from column 0 to last column with header name. If the current row contains additional fields whereas the header row is shorter, then the excess fields will not be included. | [ table: **..**,1 ] Partial: [ **..** ] *{New York City,New York,USA,83000000, St. of Liberty,10,Guggenheim,Taxi}*  Assuming the Favorite Beer column has been added to Boston, then the parameter set for New York would contain an additional blank element at the end: *…,Taxi,}* |
|  | (nothing) | Entire row according to the full width of the actual row (ignoring the header row) | [ table:,1 ] Partial: [ ] *{New York City,New York,USA,83000000, St. of Liberty,10,Guggenheim,Taxi}*  Assuming the Favorite Beer column has been added to Boston, then the parameter set for New York would not show the additional blank field. |

Program example 1:

**table load**( table, "Cities.csv");

**echo**( [table:,0] ); // Lists the header row

**table process**( t, **echo**([]) ); // Lists all other rows below

Program example 2:

**table load**( table, "Cities.csv");

**echo**( [table:City, New York City ] ); // Reveals row number

**echo**( [table:City, {New York City} ] ); // Reveals Contents in parameter set

**echo**( [table:City, {} ] ); // Reveals an empty set

**echo**( [table:City, New York, { "State / Province", Country } ] );

// In this lookup case, the first column must be a simple column specifier

// to identify one city. Using multiple columns as search criteria is not

// supported here.

Output:

1

{New York City}

{}

{New York,USA}

Horizontal write access rules:

* If a scalar value (any type except parameter set) is written with horizontal access, then the same value will be applied on all specified columns. All other values will be converted to literals.
* If a parameter set is written, then the respective elements from left to right will be written.  
  - If the parameter set contains fewer elements than the number of cells specified in the table,  
   then the remaining cells will not be overwritten.  
  - If the parameter set contains more elements than the number of cells specified in the table,  
   then not all parameter elements will be written into the table.
* If an element in the parameter set is in turn a parameter set, then it will be converted to a  
  literal in order to visualize the parameter set contents (with braces around, elements separated with commas).

Code Example:

[table:..,1] = {Big Apple, NY}; // Overwrites "New York City" & "New York".

[table:0..1] = {NYC, NY, U.S.}; // NYC and NY will overwrite,

// but U.S. does not overwrite USA

[table:..,1] = {}; // Empty sets result in no overwrites.

[table:1..3,1] = Hi; // Columns 1..3 / Row 1 will be overwritten by "Hi".

## Vertical Access (Multiple Cells in a Column)

This novel feature allows reading multiple cells from one column where a range or set of rows need to be specified. The information read is a **parameter set** containing the values from the selected columns. Write access is also supported with parameter sets as well as with scalars (other types).

Full and partial table specification.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **[** | **table name** | **:** | **Simple Column Specifier** | **,** | **Row Specifier** | **,** | **Simple Column Specifier 2** | **]** |
|  | Mandatory  Partial table speci-fication: (Optional) | | Mandatory | Mandatory  Partial table specification: (Optional) | | Optional | |  |
|  | Literal  Note the colon (:) symbol behind table name, do not use a comma. | | Literal (looks for 1st match)  or  Positive row number  or  Negative row number | Simple Column Specifier  or  Advanced Column Specifier | | Literal (looks for 1st match)  or  Positive row number  or  Negative row number | |  |

Following Row Specifiers are supported:

(For partial table specification, the context assumes table named "table" )

In the following table, capital letters A and B, etc. refer to simple column specifiers.

|  |  |  |  |
| --- | --- | --- | --- |
| **Category** | **Type** | **Explanation** | **Example (Full and Partial Table Spec.)** |
| **Advanced Column Specifier**  **(ACS)** | : Literal | Set of all rows which match with the literal expression. Don't forget the additional colon!  Wildcards (asterisk, commas and question marks) are supported as long the content is not a hard quoted literal (e.g. inside double quotation marks)  Row 0 (header row) will not be included in the search. This is to avoid unwanted inclusion of header names when matching some patterns in the contents below. | Full: [ table: City, **: 'V\*'** ] Partial: [ City, **: 'V\*'** ] returns parameter set *{Venice,Vienna}*  Why? The header names *City* and *Country* begin with a capital C.  Wildcard symbols ?, \* and , (comma) are supported.  Following modifiers are available to laxen comparison rules: + Ignore case  [ table: City, **: +'V\*'** ]  ~ Ignore blanks  [ table: City, **: ~'V\*'** ]  and combinations of the above. |
| {A, B, …} | Parameter set containing simple row specifiers (row contents, positive and negative row numbers).  This rule applies even if the parameter set contains only 1 element.  If B is before A, then empty set is returned. | Full: [ table: 0, **{ Paris, Venice }** ] Partial: [ 0, **{ Paris, Venice }** ] returns parameter set *{Paris, Venice}*  Full: [ table: 0, **{ Paris, Venice }**, 2 ] Partial: [ 0, **{ Paris, Venice }**, 2 ] returns parameter set *{FRA,ITA}* |
| : {A, B, …} | Set of all rows which match with the literal expressions specified here which must correspond with the column headers.  Important: 1st and 2nd parameter must reflect the same number of elements. The 1st parameter may refer to single cell, a range or a set.  Row 0 (header row) will not be included in the search. This is to avoid unwanted inclusion of header names when matching some patterns in the contents below. | Full: [ table, {0,2], **:**{ 'V\*', ITA } ] returns parameter set *{{Venice,ITA}} which is actually a matrix access*  Full: [ table, {0,2], **:**{ 'V\*', ITA }, -1 ] returns parameter set *{Gondola}*  Following modifiers are available to laxen comparison rules: + Ignore case  [ table, {0,2], **:+**{ 'V\*', ITA } ]  ~ Ignore blanks  [ table, {0,2], **:~**{ 'V\*', ITA } ]  and combinations of the above. |
| A..B | Range from row A to B (both are simple row specifiers) | [ table: 0, **2..3** ] Partial: [ 0, **2..3** ] *{Washington,Philadelphia}* |
| A.. | Range from row A to last row | [ table: 1, **-2..**] Partial: [ 1, **-2..** ]  *{Île de France,Grishun}* |
|  | ..A | Range from row 1 to A (excludes header row) | [ table: 1, **..2** ] Partial: [ 1, **..2** ] *{New York,D.C.}* |
|  | .. | Range from column 0 to last column with header name. If the current row contains additional fields whereas the header row is shorter, then the excess fields will not be included.  The number of elements read is the same, regardless which column is chosen. | [ table: 2, **..** ] Partial: [ 2, **..** ] *{USA,USA,USA,USA,USA,Can,Dan,ITA,USA,AUT,THA,SWI,FRA,SWI}*  Assuming the Favorite Beer column has been added to a new city, then the parameter set for New York would contain an additional blank element at the end: *…,FRA,SWI,,}* |
|  | (nothing) | Entire column from row 0 (including header) last existing row.  The number of elements read may differ, depending which column is chosen.  Beyond end of longest row: empty set returned. | [ table:2, ] Partial: [ 2, ] *Same output as above.*  Assuming an additional city has been added, but no country, etc., then the result will not differ. |

Program example 1:

**table load**( table, "Cities.csv");

**echo**( **sum**( [table: Inhabitants, .. ] ) ); // Sums all inhabitants

**echo**( **sum**( [table: Country, :USA, Inhabitants ] ) ); // USA only

// The last function makes use of the lookup feature to identify the

// rows to include in the calculation. In short: A conditional sum

Output:

36415000

14870000

Vertical write access rules:

* If a scalar value (any type except parameter set) is written with vertical access, then the same value will be applied on all specified rows. All other values will be converted to literals.
* If a parameter set is written, then the respective elements from left to right will be written into the successive rows.  
  - If the parameter set contains fewer elements than the number of cells specified in the table,  
   then the remaining cells will not be overwritten.  
  - If the parameter set contains more elements than the number of cells specified in the table,  
   then not all parameter elements will be written into the table.
* If an element in the parameter set is in turn a parameter set, then it will be converted to a  
  literal in order to visualize the parameter set contents (with braces around, elements separated with commas).

Code Example:

[table:Country,:USA] = "U.S.";

// Renames all country names from USA to GB

[table:0,1..] = {Big Apple,Georgetown,Phily};

// Replaces New York City, Washington and Philadelphia with new names

[table:8,..] = "..."; // Adds a new column with ... inside

[table:9,] = Blabla; // No action, because of 0 existing rows beyond table

[table:0,3..2] = Blabla; // No action because 1st row is below 2nd row.

## Matrix Access (Multiple Rows and Columns)

Matrix accesses are a combination of horizontal and vertical accesses. The output are nested (i.e. 2-dimensional) parameter sets. This access method allows to selectively retrieve from and update specific columns in multiple rows.

Full and partial table specification.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **[** | **table name** | **:** | **Column Specifier** | **,** | **Row Specifier** | **,** | **Column Specifier 2** | **]** |
|  | Mandatory  Partial table speci-fication: (Optional) | | Mandatory | Mandatory  Partial table specification: (Optional) | | Optional | |  |
|  | Literal  Note the colon (:) symbol behind table name, do not use a comma. | | Simple Column Specifier  or  Advanced Column Specifier | Simple Column Specifier  or  Advanced Column Specifier | | Simple Column Specifier  or  Advanced Column Specifier | |  |

Programming example 1:

**table load**( table, "Cities.csv");

**echo**( [table: { Country, City }, { 1, 4 } ] );

Output:

{{USA,New York City},{USA,Boston}}

Programming example 2:

**table load**( table, "Cities.csv");

**echo**( [table: Country, :SWI, {City,Inhabitants} ] );

// Looks up everything in Switzerland and retrieves 2 columns

Output:

{{Zürich,404000},{Davos,11000]]

Matrix write access rules:

* If a scalar value (any type except parameter set) is written with vertical access, then the same value will be applied on all specified cells (in multiple rows and columns). All other values will be converted to literals.
* If a simple parameter set containing scalar values (no further parameter sets inside) is written, then the parameter set will be applied repeatedly on each row
* If a parameter set is written, then the respective elements from left to right will be written into the successive rows and columns.  
  - If the parameter set contains fewer elements than the number of cells specified in the table,  
   then the remaining cells will not be overwritten.  
  - If the parameter set contains more elements than the number of cells specified in the table,  
   then not all parameter elements will be written into the table.
* If an element in the parameter set is in turn a parameter set, then it will be converted to a  
  literal in order to visualize the parameter set contents (with braces around, elements separated with commas).

Programming example 1: Overwriting (parts of) several rows with same contents

[table:.."State / Province",1..2]= { San José, CA, U.S. };

Output:

The first 3 columns in rows 1 and 2 (New York City and Washington's rows) are overwritten  
 by *San José, CA,* and *U.S.*

Programming example 2: Replacing some contents

[table:.."State / Province",1..2]= {{ NYC, NY, U.S. }, {Wash, DC, U.S.}};

**echo**( [table:..2,1], " and ", [table:..2,2] );

Output:

{NYC,NY,USA} and {Wash,DC,USA}

Programming example 3: Swapping or rotating columns

[table:0..2,] = [table:{1,2,0},];

or following statement:

[table:{City,"State / Province",Country},]

= [table:{ "State / Province",Country,City},]

Output: Column 0 (City) is moved to column 2. Original columns 1 and 2 (State / Province, Country) are moved one step to the left. The rotation includes the column header. If you want to move data without the column headers, then add 2 points after the last comma, as shown below

[table:0..2,..] = [table:{1,2,0},..];

Programming example 4: Swapping or rotating rows

[table:,1..2] = [table:,{2,1}];

Output: Rows 1 and 2 have been exchanged.

Programming example 5: Doing some destructions

[table:,] = deleted; // Overwrites the entire table with "deleted".

// including the header names !!!

## Special Cases

When a read access reveals an empty set (regardless if horizontal, vertical or matrix access), then write accesses to the same destination have no effect.

Example 1

**echo**( [table:City,5..4] ); // Output: {} because row 5 is before row 4

[table:City,5..4] = Ignored; // Nothing will be written

Example 2

**echo**( [t:City,{}] ); // Returns empty set because no search is done

**echo**( [t:City,{Lugano}]); // Returns an error message (Lugano not in list)

**echo**( [t:{},{} ] ); // Returns an empty set for same reason, too.

**echo**( [t:{},Lugano] ); // Error message: No column specified to compare

// contents below with Lugano.

## Partial Table Specifications – Next Level Up

As mentioned before, partial table specifications allow for limiting table specifications to the column header name(s) whereas the current context information such as table name and row number are available.

In case you use nested partial table specifications (e.g. **for all table rows**(…) with **table process**(…) inside), the table context information does always reflect the innermost level where the program is running, which is the context of **table process**(…) in the example stated above.

Assuming you have a customer and order list, both have columns for customer names 'Name' as well as 'Amount' in the table 'orders' and 'Total Amount' in the customer table. The following four statements calculate the total amount every customer has spent and writes them into the column 'Total Amount'.

{

**table process**( customers,

{

total[]= 0;

name[] = [Name];

**table process selected rows**( orders, [Name]==name[], total[] += [Amount] );

[Total Amount] = total[];

}

}

With the modifier symbol accent circumflex ( ^ ), also used as deep operator symbols for unary and binary operations as well as transactions added immediately after the opening bracket referencing tables, it indicates to use the next outer table context information. We call this symbol " ^ " "Next level up".

Code example with same functionality as above:

{

**table process**( customers,

{

**table process selected rows** ( orders, [**^**Name] == [Name], [**^**Total Amount] += [Amount] );

}

}

You see the big difference where the number of statements has shrunk from 5 to 2.

Four statements are condensed into one simple statement and without need to use temporary variables. In the inner context (statements of the 2nd **table** **process selected rows**(…) in this case),

[ Name ] refers, as usual, to the table 'orders' and row number provided by  
 **table process selected rows** (…).

[ ^ Name ] refers to the table 'customers' and row number provided by  
 **table process** (…).

In brief, [^Name]==[Name] looks for orders with current name matching in the customer list, and  
[^Total Amount] += [Amount] adds the amounts of all orders into the 'Total Amount' column of the customers.

Important properties:

* The "Next level up" can be applied regardless if a simple, horizontal, vertical or matrix access is made.
* The "Next level up" symbol will be ignored if a full table specification follows because present table name as well as row number supersede the context information. If a table name is mentioned but not a row number, then the row number applies. If a row number is specified but not the table name, then the table in the selected context applies.
* Multiple consecutive **^** symbols (repeated "Next level up'') are supported as long the context information is available. In the code example above, [ **^^** Name ] would assert an error because only two contexts are provided. However it works fine if you have one additional nested level.

## Formatting Attributes

**ATTENTION – Feature declared obsolete starting with Release 5.00: These features are still running, but will not be supported any further and are subject to be removed in due time. Please refer to section 11.1 instead where a novel and comprehensive library of style and formatting functions are provided. These functions are much more powerful and independent from the target file format.**

A novelty is saving Beyond4P tables into a HTML file:  
**table save ( table name, "file.html", HTML );**

This feature allows table contents to include formatting attributes which are supported in the HTML language. Below you see a selection of formatting features. Please refer to HTML handbooks to see a complete list of formatting options.

* Text color
* Field color
* Left / center / right alignment
* Excel numbering format
* Force Excel to recognize numeric, date and Boolean contents as text  
  ( 00012 is 00012 and not 12).
* Font, boldface, italic, underlining, font size
* and many more.

The formatting attributes are added to the end of the text contents like carry-on luggage. The formatting information begins with the **escape** symbol (ANSI / UNICODE 27) and further optional formatting entries are separated with **escape** symbols, too.

1. Data without formatting, as practiced all time in the past

|  |
| --- |
| **Table contents** |
| Hello |

2. Data with field formatting

|  |  |  |
| --- | --- | --- |
| **Table contents** | **escape** | **Data field formatting option** |
| Hello | escape | bgcolor = "yellow" |

In order to force Excel not to interpret data which looks like numbers, prices, dates and boolean values, use the formatting option **style="vnd.ms-excel.numberformat:@"**  
  
Example assignment:

force\_text\_format[ ] = escape + '**style="vnd.ms-excel.numberformat:@"**';

[ table : Phone No, 1] = '0800800800' + force\_text\_format[];

// Excel will treat phone number as text. Leading zero is preserved.

3. Data with field contents formatting. Example: Red text.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Table contents** | **escape** | **Data field formatting option** | **escape** | **Formatting before field contents** | **escape** | **Formatting after field contents** |
| Hello | escape | ( keep blank ) | escape | <font color = "red"> | escape | </font> |

Example assignment:

red\_text[] = escape + escape + '**<font color = "red">**' + escape + '**</font>**';

[ table : Status, 1] = Alarming + red\_text[];

4. Combination of field and field contents formatting

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **able contents** | **escape** | **Data field formatting option** | **escape** | **Formatting before field contents** | **escape** | **Formatting after field contents** |
| Hello | escape | bgcolor = "yellow" | escape | <font color = "red"> | escape | </font> |

red\_on\_yellow[] = escape + '**bgcolor = "yellow"**'

+escape + '**<font color = "red">**' + escape + '**</font>**';

[ table : Status, 1] = Alarming + red\_text[];

**Attention:**

Beyond4P will not check or process any attribute contents. They are recognizes as regular text contents where the escape character is treated similarly like any other visible character. The code will be inserted into the generated HTML files. Erroneous code may probably (but this is not guaranteed) be highlighted when viewing the resulting HTML file with a web browser or Excel.

You can easily remove all formatting attributes with the function call **left** (…), e.g.

a[] = **left**( a[], escape );

**Formatting attributes are discarded when saving files in the conventional way** using comma or other separation symbols.

**Formatting attributes or data with formatting attributes attached will not be discarded** with any other operations or function calls applied, e.g. a[ ] = b[ ] + c[ ]; .

# Formulas and Operators

Complete overhaul and enhancements of operators, including comparing parameter sets and dates with time information.

## Formula Basics

Formulas are expression which contains a combination of the following:

* Constants (numerals, literals, Booleans, dates etc.)
* Parameter sets
* Variables
* Table references
* Unary operators (e.g. minus sign)
* Binary operators (Arithmetic operators, comparison operators, boolean operators)
* Parentheses (to influence the calculation precedence)

Following precedence rules apply (starting with highest, ending with lowest)

1. Unary operators
2. Arithmetic operators \* and /
3. Arithmetic operators + and –
4. Comparison operators: =, ==, <>, !=, >, >=, <, <=
5. Logical binary operators & (and) and | (or)

In expressions, calculations are first made at highest precedence. As an example, multiplications are done first before additions if no parentheses are used to influence the calculation precedence.

a[] = 2\*3 + 4\*5; // Yields 26. Multiplications are done before addition.

a[] = 2\*(3+4)\*5; // Yields 70. Addition is done first.

## Logical Binary Operators

The logical binary operators **&** (*and*) and **|** (*or*) are available for Boolean values (*and*, *or*) as well as for parameter sets: & (*intersection*) and | (*union set*) applies. Logical operations on all other types will result in exceptions.

**Calculation with Logical Binary Operators:**

d

Processing void values will result in exceptions (error messages).

## Comparison Operators

Beyond4P provides a total of 8 different comparison operators. A distinction of Beyond4P compared to other programming languages are two different *equal* (=, ==) and *not equal* (<>, !=) symbols which have different behavior. The symbols (==, !=) are referred as *strictly equal* and *strictly not equal*.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning** | **Explanation** |
| = | Equal | **Comparing values of different types:**  **Smart comparison** applied on selected combinations, otherwise false returned  Example:  true = 1 false ( Boolean compared with numeral )  Smart comparison on literals with numerals:  1 = 01 true 1 = '1' true 1 = '01' false  '1'='01' false 01 = '1' false   01 = '01' true 01+0='01' false 01+0 = '1' true  In the last 2 example, the original literal representation '01' in the first term  is destroyed by adding 0 before comparing, sot '1' is compared against numeric 1.  123 = '1\*' true (wildcard allowed)  Smart comparison on literals with dates:  date("01.10.2016") = 01.10.2016 true  Smart comparisons on literals with booleans:  true = 'true' true  true = TRUE false ( value to right is a literal, capital letters)  true = +TRUE true ( comparison made case insensitive )  false = 'f\*' true ( wildcards allowed)  **Comparing void values:**  Void with void values return always true, regardless how they have been  generated and text messages if converyed. Void with valid types return false.  v[ w ] = 0; echo( v[ ] = 0 ) false  echo( v[ ] = v [ ] ) true  **Supports comparison modifiers:**  Abc = +aBC true (Ignore case)  AB C =~A BC true (Ignore blanks) Combination of both is allowed.  Combinations of + and ~ are allowed  (Do not confuse these symbols with unary operators, If you intend to use them as  unary operators, use additional parentheses, e.g. Abc = (+abc) returns false, and  ABC = (+abc) returns true.  **Supports** **wildcards if right hand variable is a literal**  Hello = 'He\*' true (This is a softquoted literal)  Hello = "He\*" false (This is a quoted literal)  Hello = 'Hi,Hello' true  Hello = 'H?llo' true  **Relaxed parameter set comparison:** Contents must be same, but ordering does not matter  {1,2,3} = {1,2,3} true  {1,2,3} = {1,3,2} true  **Supports selections and ranges:**  3 = 2,3,5 true (Equals to 2, 3 or 5)  3 = 4 .. 6 false (Lies within and including 4..6)  {a,c} = {a,b}..{a,d} true (Also valid on parameter sets)  **Attention** when using this comparison symbol in comma separated lists such as  function parameters, parameter sets, etc:  **echo**( "Result is ", a[ ] = 3, 4, " and is OK" );  // The term " and is OK" is seen as a 3rd parameter to compare with a[ ].  // Use parentheses as shown below to compare with 3 and 4:  **echo**( "Result is ", (a[ ] = 3, 4), " and is OK" );  // The statement below shows a comparison with 'strictly equal' where comparison  // is only made with 3. The values 4 and "and is OK" are echoed as usual  **echo**( "Result is ", a[ ] == 3, 4, " and is OK" ); |

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning** | **Explanation** |
| == | Strictly equal | **Comparing values of different types:** Returns 'false' all time  1 == '1' false  'true' == true false  **Comparing void values:**  Same behavior as with '=' operator  **Supports comparison modifiers:** Like with the = symbol.  Abc ==+ aBC true (ignore case)  Hell o ==~He llo true (ignore blanks). Combination of both is allowed.  **No wildcards are supported:**  Hello == 'He\*' false  'He\*' == 'He\*' true (The two literals, 3 letters each, are identical)  **Strict parameter set comparison:** Contents and ordering must be identical  {1,2,3} = {1,2,3} true  {1,2,3} = {1,3,2} false  **No Selections and Ranges are supported:**  Only 1 expression on right hand side of the operator is allowed. |
| <> | Not equal | Like "equal" but outcome is inverted |
| != | Not strictly equal | Like "strictly equal" but outcome is inverted. |
| > | Greater than | **Comparing values of same types:**  Numerals: Values of numbers, e.g. 3 > 2 returns true.  Dates Later date resp. later time is greater. Special rules apply and  are described further down.  Literals Alphabetic ordering, e.g. "Café" > "Apéro"  Booleans true > false returns true  **Comparing void values:**  Exceptions (error messages) will occur. If you suspect some values compared  to be void, use the '=', '==', '<>' or '!=' operator first to extract void values and  treat them separately.  **Comparing values of different types:**  false will be returned all time, regardless of the values, including those shown below:  '0' >= 0 false '0' <= 0 false  '' >= 0 false '' <= 0 false  true >= 0 false false <= 1 false  **Supports comparison modifiers:**  BCD > abc false  BCD > +abc true (ignore case)  BCD > BC D true  BCD > ~BC D false (ignore blanks)  Combination of + and ~ are allowed.  When comparing parameter sets, the modifiers apply to the values contained.  **No wildcards are supported:**  Hello <= 'He\*' false  'He\*' <= 'He\*' true (The two literals, 3 letters each, are identical)  **Relational comparison of parameter sets:**  Rule 1: The elements in both sets are compared from left to right.  Rule 2: If the elements are of same type, the values are compared  (corresponding nested parameter sets are compared recursively)  Rule 3: If the elements are of different types, then following ruling is used:  boolean < numeral < date < literal < parameter set  Rule 4: If 2 parameter set have different lengths, but common contents are  the same, then the longer parameter set is 'greater'.  Special case: Empty set is always smaller than a set with elements inside:  Both { } < { 1 } and { } < { { } } return true.  **No Selections and Ranges are supported:**  Only 1 expression on right hand side of the operator is allowed. |
| >= | Greater than or equal |
| < | Less than |
| <= | Less than or equal |

**Comparison with operators = (equal) and <> (not equal):**



**Special case: Dates compared with = and <>:**



Examples:

echo( 123 == 123, ' and ', 123 = 123 ); // Output: true and true

echo( 123 == 0123, ' and ', 123 = 0123 ); // Output: true and true

echo( 123 == '123',' and ', 123 = '123' ); // Output: false and true

echo( 0123== '123',' and ',0123 = '123' ); // Output: false and false

echo( 123 == "123",' and ', 123 = "123" ); // Output: false and true

echo( 123 == "12\*",' and ', 123 = "12\*" ); // Output: false and false

echo( 123 == '12\*',' and ', 123 = '12\*' ); // Output: false and true

echo( date(today) == '09.04.2016', ' and ', date(today) = '09.04.2016' ); // false and true

echo( date(today) == '09.04.2016', ' and ', date(today) = '9.04.2016' ); // false and false

**Comparison with operators == (strictly equal) and != (strictly not equal):**



**Special case: Dates compared with == and !=:**



Examples:

echo( 123 == 123, ' and ', 123 = 123 ); // Output: true and true

echo( 123 == 0123, ' and ', 123 = 0123 ); // Output: true and true

echo( 123 == '123',' and ', 123 = '123' ); // Output: false and true

echo( 0123== '123',' and ',0123 = '123' ); // Output: false and false

echo( 123 == "123",' and ', 123 = "123" ); // Output: false and true

echo( 123 == "12\*",' and ', 123 = "12\*" ); // Output: false and false

echo( 123 == '12\*',' and ', 123 = '12\*' ); // Output: false and true

echo( date(today) == '09.04.2016', ' and ', date(today) = '09.04.2016' ); // false and true

echo( date(today) == '09.04.2016', ' and ', date(today) = '9.04.2016' ); // false and false

**Comparison with relational comparison operators >, >=, < and <=:**



Literals are compared by alphabetical order, e.g. 'B' is greater than 'A'. Boolean 'true' is greater than 'false'. Parameter sets cannot be compared with these relational comparison operators. If you want to compare the number of elements inside the set, then use the { } ending, see example below:

Comparing differing types will always return false, regardless of value contained. Examples:

echo( "" >= 0, ", ", "" <= 0 ) // Output: false, false

a[] = text; // a[] = 123; // Try with text and numeral

if ((a[]>= 0) | (a[]<=0)) echo( "a[] contains a numeral" ); else: echo( "Not a numeral");

**Special case: Dates compared with >, >=, < and <=:**



### Wildcards

For selected operations, wildcards provide some powerful means to compare strings against specific patterns. With simple symbols, you can specify a check like "starts with …", "contains …","ends with …" or any free combinations. Wildcard symbols will only be considered as such as long they are inside (unquoted) literals (e.g. literals without quotation marks, literals directly read from tables) or softquoted literals (inside single quotation marks 'abc'). In quoted literals ("abc"), wildcard usage is disabled and all characters inside are considered as actual text contents.

Supported wildcard symbols:

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbol** | **Meaning** | **Example** | **Explanation** |
| ? | Placeholder for exactly 1 character | H?  ???  ?e | Text must be 2 characters long and start with *H*, e.g. *Ha, He, Hi*  Text must be exactly 3 characters long  Text must be 2 characters long and end with *e* |
| \* | Placeholder for any number of characters, including 0. | A\*  \*s  \*-\*  \*AB\*CD | Begins with *A*  Ends with *s*  Contains a hyphen  Contains *AB* and further on ends with *CD*. |
| , | Divides pattern text into substrings and compares them individually | Ha,He,Hi  Ha,  ,Ha | Must equal to *Ha*, *He* or *Hi*. *Ho* will not match.  Must equal to *Ha* or blank literal (note the extra comma)  Same as above |
| Combi-nations | The wildcard symbols may be combined freely | A\*,B\*,C\*  \*1\*,\*0\*  A??\*  ??:??:??  \*.\*.20?? | Must begin with *A*, *B* or *C*.  Must contain digit *1* or *0* (or both).  Must contain at least 3 characters and begin with *A*  3 x two characters separated with colons, like a time format.  Checks for something which looks somehow like a date |
| **Extended Wildcard Set** | | | |
| # | Placeholder for exactly 1 numeric digit | (###) ###-#### | Must look like a telephone number (example for N. America) |
| & | Placeholder for any number of numeric digits (0 or more) | +&-&-&\*& | An international telephone number |
| ^ | Placeholder for exactly 1 alphanumeric character | ^^ | 2 letters  Characters: A-Z, a-z, all non-ANSI characters |
| ~ | Placeholder for any number of alphanumeric characters (0 or more) | A~ | Number followed by test  Characters: A-Z, a-z, all non-ANSI characters |

Wildcards are supported in following features:

* Comparing literals using the = (equal) or <> (not equal) symbol. Wildcards are used on  
  the right hand side of the comparison symbol only.  
  Note that the comparison symbols == and != symbols do not support wildcards.
* Seeking tables with all matches, e.g.  
  [ table : City, :'A\*' ] Find all rows with cities beginning with A.  
  [ table : 0, :'???' ] Find all column header names made of 3 characters
* Functions **directory listing** (…) and **directory listing recursive** (…).  
  In this case, wildcard checking on file names is taken care by the operating system and not by the Beyond4P language. In this case, asterisk and question marks are OK, but the comma separator will not work.

### Ranges and Selections

The comparison operators = and <> (but not the remaining ones) support ranges and selections on the right hand side.

b[] = a[] = 2 | a[] = 5 | a[] = 9 | a[] = 13; // The conventional way

b[] = a[] = 2,5,9,13; // Using selections

b[] = !(a[] >= 4 & a[] <= 8); // The conventional way

b[] = a[]<> 4..8; // Using ranges

Combinations of selections and ranges

b[] = a[]= 4..8,14..18,20,23,None,Invalid; // Using a combination of ranges and  
 // selections, and different types

Hint: Use ranges and selections in functions like **if** (…), **while** (…), **case** (...) functions, etc.

### Comparison Modifier Symbols

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Operator** | **Numeral** | **Date** | **Literal** | **Boolean** | **Parameters** | **Void** |
| **+** *expr …* | No effect | No effect | Ignore case | Exception | Ignore case 1 | Void |
| **~** *expr …* | Tolerance 2 | No effect | Ignore blanks | Exception | Ignore blanks1 | Void |

1 Parameter set members are affected by the modifiers.

2 An error tolerance value (epsilon **ε**, located as system variable named **runtime settings [epsilon]**, initialized as 1.0E-9, or e.g. 0.1 cent in currency values in millions) will be considered when comparing. See figure below.

The upper schemes refer to precise comparison, i.e. only a perfect difference of zero is a match. The schemes on the bottom side apply if the tilde operator (~ = tolerance) is applied on numeric comparisons. This feature may be useful to avoid trouble with residual arithmetic errors resulting from complicated mathematics operations.



The tolerance value may be changed. Please note that the value will not be reset when leaving called procedures, functions and program files. E.g. **runtime settings [ epsilon ] = 0.001;**

## Comparison Expressions

Comparison expressions are a combination of comparison operator followed by the value(s) on the right hand side. If no operator is provided, then equal (=) is assumed implicitly.

Comparison Expressions are used in following places:

* In **case** (…) function located inside **switch** and **check** blocks
* Function **table lookup smart** (…) where 1 or more table columns contain comparison expresions.

|  |  |  |  |
| --- | --- | --- | --- |
| **Pattern** | **Numeric** | **Alphanumeric** | **Function** |
| Equal | 1  12.3  =089 | '1'  "12.3"  ='089'  France  Saudi Arabia " United States" = United Kingdom | No operator defaults to equal. Compares if equal  Leading and ending blanks will be ignored unless the alphanumeric text is inside single or double quotation marks (soft and hard quotation)  No quotes: Numeric or literal, ignores leading and final blanks (e.g. for United Kingdom, Saudi Arabia)  ' ' = Soft quotes (allows wildcards \*, ?) " " = Hard quotes (forbids wildcards) |
| Absolutely Equal | ==1 | =='A'  =='A\*' | Comparison of precisely two values.  No type conversions between numerals and literals E.g. numeral 123 is not equal to literal "123".  No wildcards, no selections, no ranges. Example compares with actual text contents A\* |
| Not equal | <>1 | <>ABC, <>"5.43" | Compares if not equal |
| Absolutely Not Equal | !=1 | !='A' | Comparison of precisely two values. No type conversions between numerals and literals No wildcards, no selections, no ranges. |
| Relational comparison | >1 >= 1.2 <-1 <=-1.1 | >AB >= 'A B' <" X " <= X | Supported: <, <=, >, >= Attention alphanumeric: '9' > '11' yields *true*, |
| Range  (only with Equal or Not equal) | 1..10 01..10 =2..11 <>3..6 | '1' .. 9 'A'..'Z' ='A'..'E' <>F..X | Numeric range only if target value and both range parameters are numeric. If one of them is alphanumeric, then the range is based on alphanumeric comparison.  Boundary values are included, e.g. testing 2 for 2..4 yields true. |
| Ranges | 1..5 <>4..6 | =BA...BZ <>A..E | No wildcards allowed. <, <=, >, >=, == and != forbidden |
| Selection  (only with **=** and **<>** symbol) | 2,3,5,7,11  =5,10  <>3,6,9 | DE,FR,IT  ='1','2',3.14  <>"DE","FR"  =A\*,B\*,C\* | Compares selection. No relational operators allowed  No or equal operator: True if >=1 element matches Not equal operator: False if >=1 element matches  Numeric elements will be compared as numbers if target is also numeric, e.g. number compared with 3.14.  Wildcards are allowed. |
| Both of above | -1…3,7..9 | <>A\*..C\*,F,M\*..Z\* | Combination of both. Wildcards allowed if not in a range. |
| Wildcards | Not applicable | A\* H?llo =' B\*' \* (always true)  >"C\*" (no wildcard) | Equal and not-equal comparisons allow wildcards in alphanumeric text except if hard-quoted (double quotation marks: " C\*" checks contents literally. \* = 0 or 1 or more characters of choice ? = Exactly 1 character of choice \* alone always yields *true* <>\* always yields *false* |
| Blank | Not applicable | (blank) '' "" | True only if blank. Blanks inside a number, e.g. - 1 .2 converts into a literal "- 1 .2". |
| Ignore cases | Not applicable (ignored) | +abc +=" hello " +köln | Cases are ignored in both comparison and target string. + must precede operator, incl. W. European diacritics. |
| Ignore blanks |  | ~Goodmorning <>~Good morning' ~ Good Morn\* | True for both "Good morning", "goodmorning", "good morn ing". May combine with + (ignore case). |
| Arithmetic tolerance | ~0  <>~4 | See "Ignore blanks" | Epsilon є = 0.000 000 001 (0.1 ct for M EUR range) Equal: Within +/- є Not equal: Outside +/- є <=, >= Equal section within +/- є <, > Value must differ from taget at least by є |

## Arithmetic Operators

Following symbols are used for arithmetic operations: **+** (*addition*), **-** (*subtraction*), **\*** (*multiplication*) and **/** (*division*). See table below for supported operations:

|  |  |  |  |
| --- | --- | --- | --- |
| **Types** | **Description** | **Example** | **Result to example** |
| **Numerals** | Arithmetic applied | 5 + 4 | 9 |
| **Boolean** | Converted to 0 and 1, then arithmetic applied | 5 + false 4 \* true true + true | 5 4 2 |
| **Date** | Date and time specific arithmetic manipulations (+ and – only) | date("31.12.2016")+2 date("31.12.2016")- date("24.12.2016") | 02.01.2017 7 |
| **Literal** | String manipulations: Catenation: Remove 1 pattern: Remove all patterns: | ABC + DEF ABCBCDCDE – BC ABCBCDCDE / BC | ABCDEF ABCDCDE ADCDE |
| **Parameter set** | Set operations: Catenation: Remove elements once Remove all elements | { 1, 3, 5, 7, A } + { 2, 3, 4, A } { 1, 3, 5, 7, 2, 3, 4, 5 } - { 5, 3 } { 1, 3, 5, 7, 2, 3, 4, 5 } / { 5, 3 } | { 1, 3, 5, 7, 2, 3, 4, A, A } { 1, 7, 2, 3, 4, 5 } { 1, 7, 2, 4 } |
| **Blank literals with numerals, dates, booleans** | Only blank literals accepted. Treated as 0. | 5 + “” date("31.12.2016") + "" true + "" | 5 31.12.2016 1 |
| **void** | Exception (error message) |  |  |

**Additions with +:**



**Subtractions with -:**



**Multiplication with \*:**



Regarding multiplication of numerals with parameter sets (and vice versa): The number of elements is multiplied (actually repeated) in order to achieve the applied factor. Examples:

{a,b,c,d} \* 2 // returns { a,b,c,d,a,b,c,d }

{a,b,c,d} \* 1.5 // returns { a,b,c,d,a,b }

1.4 \* {a,b,c,d} // Same outcome (rounded up)

1.6 \* {a,b,c,d} // Same outcome (rounded down)

{a,b,c,d} \* 1 // returns { a,b,c,d }

{a,b,c,d} \* 0.25 // returns { a }

{a,b,c,d} \* 0.1 // returns { } (empty set) (rounded down)

{a,b,c,d} \* 0 // returns { } (empty set)

{a,b,c,d} \* -1 // returns { } (empty set) (negative numbers keep result empty)

true \* {a,b,c,d} // returns {a,b,c,d} since true is treated as 1

{a,b,c,d} \* false // returns { } (empty set) since false is treated as 0

**Division with /:**



### Arithmetic operations with Dates (Time of day)

Special rules apply on arithmetic with dates. Only the addition (+) and subtraction (-) operators are allowed. Different approaches are applied whether the operands contain no date (e.g. date("")), just a date (e.g. date(today)), a time (e.g. time(now)) or a combination of both (e.g. date(now)).

Wrong use of certain operations may result in negative dates or time operations reaching beyond the 00:00:00 to 23:59:59 range.

Two dates can only be added if at most one of the dates contain a date value, and the other at most a time value. Numerals (and Booleans, which translate into 0 and 1) can be added to dates and the outcome are of type dates, too.

**Additions with +:**



Fractional values are values behind the decimal comma to describe time of day:  
1 / (24\*60\*60) corresponds to 1 second,1/12 to 1 hour and 0.5 to half day.

Example:

1. date(today) + 1.5 ignores the fractional part behind decimal point.

2 date(now) + 1.5 adds 1 and half days

3 time(now) + 1.5 asserts error as it passes beyond the 24 hour line

4 date(today) + time("10:00") adds 10:00h to the date and result contains both date and time.

5. date( 1/(24\*60\*60) ) returns a 00:00:01

|  |  |  |  |
| --- | --- | --- | --- |
| **Types** | **Description** | **Example** | **Result to example** |
| **Blank Dates** | Adding blank dates | date("") + date("") date("10:00") + date("") date("05.01.2016") + date("")date("05.01.2016 10:00") + date("") | (blank date) 10:00:00 05.01.2016 05.01.2016 10:00:00 |
| **Times** | Adding times | date("10:00") + date("13:00") date("10:00") + date("13:00") | 14:00:00 Exception (time out of range) |
| **Dates** | Adding dates not allowed, even if time info is included in the dates. | date("05.01.2016")+date("03.01.2016") | Exception (invalid combination) |
| **Dates with Times** | Combining dates and times into one | date("05.01.2016")+date("13:00") date("05.01.2016 13:00") + date("13:00") | 05.01.2016 13:00:00 06.01.2016 02:00:00 (note the next day.) |
| **Dates with numerals** | Adding days (whole numbers) and time (fractional parts) | date("05.01.2016 03:00") + 1.5 date("20:00") + 0.125 date("20:00") + 0.25  date("05.01.2016") + 1 date("05.01.2016") + 1.5 | 06.01.2016 15:00 23:00 Exception (time out of range)  06.01.2016 06.01.2016 (Note: no time available to change, too) |

Subtraction: Subtracting two dates will, in valid cases, give a numeral. Subtracting numerals from dates will result in dates. However, dates cannot be subtracted from numbers because date would turn out negative.



|  |  |  |  |
| --- | --- | --- | --- |
| **Types** | **Description** | **Example** | **Result to example** |
| **Blank Dates** | Subtracting blank dates | date("") - date("")  date("10:00") –date("") | 0  Exception (invalid combination) |
| **Times** | Subtracting times | date("10:00") - date("04:00")  date("04:00") - date("10:00") | 0.25 (eqv. to 6 hours)  -0.25 |
| **Dates** | Subtracting dates | date("05.01.2016")-date("03.01.2016")  date("03.01.2016")-date("05.01.2016") | 2 (eqv. to 2 days)  -2 |
| **Dates with Times** | Subtracting dates and times | date("05.01.2016 04:00") - date("03.01.2016 10:00") | 1.75 (1 day and 18 hours) |
| **Dates with numerals** | Adding days (whole numbers) and time (fractional parts) | date("05.01.2016 03:00") - 1.5 date("04:00") - 0.125 date("20:00") - 0.25  date("05.01.2016") - 1 date("05.01.2016") - 1.5 | 03.01.2016 15:00 01:00 Exception (time out of range)  04.01.2016 04.01.2016 (Note: no time available to change, too) |

## Unary Operators



Processing void values will result in exceptions (error messages).

## Parentheses

Parentheses are available to encapsulate parts of the expression which are calculated first within the parentheses.

Example expression: (1+2) \* (3+4) will calculate 21. And 1+2 \* 3+4 returns 11.

## Assignment Operators

The language supports assignment operators listed in the table below. An operator such as a[ ]+=1; works the same way as a[ ] = a[ ] + 1;, utilizing the same type checking and conversion rules as described for the operators in the previous section.

Spaces between operator and the concluding equal sign are allowed but not mandatory.

Assignment operators are allowed and useful in loops, e.g. (for a[ ] = 1, a[ ] <= 10, a[ ] += 1) {…};

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Example** | **Output** | **Description** |
| += | a[ ] += 1;  s[ ] = 'a'; s[ ] += 'b'; | Adds 1  Adds b behind a | Numeral: Addition  Literal: String catenation  Date: Adding days  Parameter sets: catenation |
| - = | a[ ] -= 1;  s[ ] = 'aal'; s[ ] -= 'a'; | Subtracts 1  Removes one a | Numeral: Subtraction  Literal: String removal  Date: Subtracting days.  Parameter sets: removal |
| \* = | a[ ] = 1; a[ ] \*= 3; | Multiplies by 3 | Numeral: Multiplication |
| / = | a[ ] = 10; a[ ] /= 2; | Divides by 2 | Numeral: Division  Literal: String removals  Parameter sets: removals |
| & = | a[ ] &= b[ ]; | AND combination | Boolean: AND |
| | = | a [ ] | = b [ ]; | OR combination | Boolean: OR |
| != = | a[ ] != = b[ ]; | XOR combination | Boolean: Exclusive OR |
| == = | a[ ] == = b[ ]; | EQV combination | Boolean: Equivalent, also known as inverted XOR |

In case assignment operators are applied on non-existing variables, then they will be initialized automatically, depending on the type applied:

* Numeral: Zero
* Literal: Empty string
* Date: First assignment
* Parameter set: empty set
* Boolean: False

**Assignment Operators on Tables:**

Using assignment operators on tables is supported to a limited extent since the type of the destination is either a literal or numeral (legitimate positive or negative number with or without decimal point recognized).

Exception to the rule: If the table entry is blank (no characters inside, not even spaces) and a numeric assignment is made, then blank will be treated as zero.

Booleans, dates and parameter sets are not supported.

Example:

[table: 0, 0] = "First Name";

[table: 0, 0] += ' and Last Name'; // Do string catenation, outcome: 'First and Last Name'

[table: 0, 1] = ''; [table: 0, 1] += 3; // Blank is treated as numeral when applying assignment  
 operator with numeral on right hand side: Do an addition to zero

## Ad hoc Operations

Beyond4P received some inspirations from the common programming language C/C++, particularly the increment and decrement operators. Here we have developed it further to provide a broader range of ad hoc operations.

### Basics

What are ad hoc operations? In brief, they are simple means to modify the associated variables and table references as part of a statements quickly and efficiently. You are actually able to modify more than one variable inside a single assignment.

Ad-hoc operations are possible in following setups

* As standalone use e.g. a[ ]++;
* As part of left-hand assignments e.g. a[ ]++ = b[ ];
* As part of right hand expression e.g. a[ ] = b[ ]++;
* and function call parameters e.g. echo( a[ ]++ );

As an example, following three operations are equivalent:

c[] = c[] + 1; a[] = b[] \* c[]; a[] = a[] – 1; // Conventional way of coding

c[] += 1; a[] = b[] \* c[]; a[] -= 1; // Possible since release 3.06

a[]-- = b[] \* ++c[]; // Possible with release 4.06

Because less code text is interpreted, the operations will be carried out more efficiently, and you will perceive a higher overall performance.

In the examples shown above, the two consecutive plus and minus signs (++ and −− must be written together without white space between) and they indicate increment-by-1 and decrement-by-1 operations respectively.

### Restrictions

Ad hoc operations are not allowed in transactions, e.g. a[ ]++ <== b[ ]; and a[ ] <<= b[ ]--. Violations will create exceptions (syntax error or wrong usage of ad hoc operations).

They are also not allowed in function parameters where an expression (actually a code piece and not just the value) is requested, for example in **dim**(…), **array**(…), **protect**(…), etc.

### Ad hoc Operators

Beyond4P supports the following three ad hoc operators:

++ Increment

−− Decrement

\*\* Scaling operator (\*\* always in combination with a scaling factor in parentheses)

Note that the two characters must be written together, without white spaces between. Otherwise they are recognized as individual plus (addition operator), minus (subtraction) and multiplication symbols. See below:

{

a[] = 10; b[] = 10;

c[] = --a[];

d[] = - -b[]; // Attention: Double negation makes number positive again

echo("a: ",a[]," b: ",b[]," c: ",c[]," d: ",d[]);

}

Output would be:

a: 9 b: 10 c: 9 d: 10

A related division symbol is not included. Calculate the reciprocal and use it for scaling.

### Target Objects for ad hoc operations

Ad hoc are possible on the following:

|  |  |  |
| --- | --- | --- |
| **Type** | **Example** | **Description** |
| **Variables** | a[ ]++ | The value a[ ] is incremented. Member variables, if existing, are not affected. |
| a[ b ] ++ | The member variable b of a[ ] is incremented. |
| a[ 1 ] ++ | The array member (or numerically referenced member variable) is incremented |
| **Tables** | [ table: Inhabitants, 1 ]++ | One particular field in the table (column named Inhabitants, row 1) is incremented |
| [ table: Inhabitants, ..]++ | Incrementing affects horizontal, vertical and matrix access. This example refers to a vertical access covering all values in the column "Inhabitants". All values will be incremented. |

### Defining the right moment to do ad hoc operation

You can freely define whether the ad hoc operation needs to be executed before (pre ad hoc) or after the access to the specified variable or table. Even combinations of pre and post ad hoc are allowed.

|  |  |  |
| --- | --- | --- |
| **Type** | **Example** | **Description** |
| **pre ad hoc** | Standalone case:  ++ a[ ];  ++ [ table: Age, 1 ]; | It does not matter if you use pre or post ad hoc operations here.  The value in a[ ] is incremented.  The field in the table is incremented |
| Inside an expression (right hand):  b[ ] = ++ a[ ];  b[ ] = ++ [ table: Age, 1 ]; | The value in a[ ] is incremented before it is used.  The field in the table is incremented before it is used. |
| During an assignment (left hand):  ++b[ ] = a[ ];  ++[ table: Age, 1 ] = a[ ]; | These operations make no sense because result are overwritten.  The value in b[ ] is incremented before being overwritten.  The value in the table is incremented before being overwritten. |
| During an assignment (left hand):  ++b[ ] += a[ ];  ++[ table: Age, 1 ] += a[ ]; | If used in combination with assignment operators, then these pre ad hoc operations make sense: Increment, then add value of a[ ]. |
| Inside function calls (input parameters)  b[ ] = abs( ++a[ ] );  b[ ] = abs( ++[ table: Age, 1 ] ); | The operation takes place before the function call starts.  The value in a[ ] is incremented before it is used.  The field in the table is incremented before it is used. |
| Inside function calls (I/O parameters)  exchange( ++a[ ], b[ ] );  exchange( ++[ table: Age, 1 ], b[ ] ); | The operation takes place before the function call starts.  The value in a[ ] is incremented before it is used.  The field in the table is incremented before it is used. |
| **post ad hoc** | Standalone case:  a[ ] ++;  [ table: Age, 1 ]++; | It does not matter if you use pre or post ad hoc operations here.  The value in a[ ] is incremented.  The field in the table is incremented |
| Inside an expression (right hand):  b[ ] = a[ ] ++;  b[ ] = [ table: Age, 1 ]++; | The value in a[ ] is incremented after it is used.  The field in the table is incremented after it is used. |
| During an assignment (left hand):  b[ ]++ = a[ ];  [ table: Age, 1 ]++ = a[ ]; | The value in b[ ] is incremented after being overwritten.  The value in the table is incremented after being overwritten. |
| Inside function calls (input parameters)  b[ ] = abs( a[ ]++ );  b[ ] = abs( [ table: Age, 1 ]++ ); | The operation takes place before the function call starts.  The value in a[ ] is incremented after it is read out for the function.  The field in the table is incremented after it is read out, too.. |
| Inside function calls (I/O parameters)  exchange( ++a[ ], b[ ] );  exchange( ++[ table: Age, 1 ], b[ ] ); | The operation takes place after function call is completed.  The value in a[ ] is incremented after value is written back.  The field in the table is incremented after the value is written back. |

### Advanced Ad hoc operations

Beyond4P has some more features in the treasure box. They include:

* Combination of pre and post ad hoc
* Cascaded pre and post ad hoc operations
* Parameterized ad hoc operation
* Multiple ad hoc operations combined

Combination of pre and post ad hoc:

You can freely apply booth pre and post ad hoc operators on variables and table specifications. Valid examples:

++a[]++; // Add 2 to a[]

--a[]++; // Subtracts 1 and adds 1 again (zero sum game)

a[] = ++b[]++; // Increments b[], assigns to a[], then increments b[] again

Cascaded ad hoc operations:

You can freely combine ad hoc operations. From the operational point of view, the total sum to increment or decrement will be identified, then one single operation is applied on the variables or tables.

a[]++ ++; // Add 2 to a[]

a[]++ --; // Zero sum game

b[] = ----a[]; // Subtracts 2 from a[] before assigning to b[]

b[] = -----a[]; // 5 symbols: Error. Last – is recognized as negation operator.

b[] = - ----a[]; // This makes more sense: Decrement, then negate expression

Parameterized ad hoc operations:

Instead of using large number of cascaded ad hoc operator symbols, parameterized ad hoc operations make more sense. They follow by a numeric expression (constant or calculation) inside parentheses.

a[]++(4); // Increment by 4

--(5)a[]; // Decrement by 5

c[] = a[]\*\*(2); // Scale a[] by 2 (double) after using it for assignment

You will also recognize that the scaling operator (\*\*) will only work in combination with a parameter. Otherwise, an error will be flagged.

Combinations:

If you use combination of scaling and ++/−− operations, you must start with scaling first.

Correct example:

a[] = \*\*(3)++b[]; // Multiply by 3, then add by 1

a[]\*\*(1/2)--(5); // Divide by 2, then subtract 5

a[]++\*\*(2); // Not allowed. Must scale first.

++a[]\*\*(2); // But this is OK Increment, then double.

[table:Balance,..]\*\*(1.015); // Apply 1.5% interest rate on all values.

a[]++(2)--(3); // Same as -- or -- (1)

### Summary of Ad Hoc Operations

Note: Relaxed ad hoc operations are applied on literals. Blank literals are assumed zero and will be incremented or decremented. However non-blank literals will not be touched at all. This is particularly useful when processing tables with ad hoc operations where occasional text values like "not applicable" are ignored.

Dates are also treated differently: No scaling allowed. Additional of non-integer values will affect time values only if dates contain time values. If dates contain time values only, then incrementing / decrementing beyond the 00:00:00 – 23:59:59 range is not possible and will assert exceptions.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Type** | **Oper.** | **Description** | **Oper.** | **Description** | **Oper.** | **Description** |
| Numeral (See 1) | ++  ++ ++  ++(5) | Increment by 1  Increment by 2  Increment by 5 | --  -- --  --(5) | Decrement by 1  Decrement by 2  Decrement by 5 | \*\*  \*\* \*\*  \*\*(5) | Exception (missing factor)  "  Multiply by 5 |
| Literal  (if blank) | ++  ++ ++  ++(5) | Sets to 1  Sets to 2  Sets to 5 | --  -- --  --(5) | Sets to -1  Sets to -2  Sets to -5 | \*\*  \*\* \*\*  \*\*(5) | Exception (missing factor)  "  Sets to 0 |
| Literal  (if not blank) | ++  ++ ++  ++(5) | Unchanged  Unchanged  Unchanged | --  -- --  --(5) | Unchanged  Unchanged  Unchanged | \*\*  \*\* \*\*  \*\*(5) | Unchanged  Unchanged  Unchanged |
| Blank Date  (See 1) | ++  ++ ++  ++(5) | Blank date  "  " | --  -- --  --(5) | Blank date  "  " | \*\*  \*\* \*\*  \*\*(0)  \*\*(1)  \*\*(5) | Blank date  "  "  "  " |
| Date  (See 1) | ++  ++ ++  ++(5)  ++(0.5) | Tomorrow  Day after tomorrow  5 days later  Adds 12 hours only if date also contains a time, otherwise fractional part of value is ignored. | --  -- --  --(5)    --(0.5) | Yesterday  Day before yesterday  5 days before  Jan 1, 1900 is absolute minimum.  Subtractcs 12 hours only if date also contains a time, otherwise fractional part of value is ignored | \*\*  \*\* \*\*  \*\*(0)  \*\*(1)  \*\*(5) | Exception (missing factor)  "  Factor 0: Blank date  Factor 1: same value  Exception (date multiplication not allowed) |
| Date (Time only) | ++  ++ ++  ++(5)  ++(0.5) | Exception (beyond 24h)  "  "  Adds 12 hours only if date also contains a time | --  -- --  --(5)  --(0.5) | Exception (beyond 24h)  "  ".  Subtractcs 12 hours only if date also contains a time | \*\*  \*\* \*\*  \*\*(0)  \*\*(1)  \*\*(5) | Exception (missing factor)  "  Factor 0: Blank date  Factor 1: same value  Exception (date multiplication not allowed) |
| Boolean | ++  ++ ++  ++(5)  ++(0) | Sets to *true*  "  "  No impact (unchanged) | --  -- --  --(5)  --(0) | Sets to *false*  "  "  No impact (unchanged) | \*\*  \*\* \*\*  \*\*(5)  \*\*(0) | Exception (missing factor)  "  No impact (unchanged)  Zero Sets to *false* |
| Void | ++  ++ ++  ++(5) | Exception (not allowed on void values) | --  -- --  --(5) | Exception (not allowed on void values) | \*\*  \*\* \*\*  \*\*(5) | Exception (not allowed on void values) |
| Parameter set | All elements will be affected by this operation, including elements in nested parameter sets. This is a clear difference from all other arithmetic operations where you need to distinguish between regular operators (affects one individual value) and deep operators (affects elements in a parameter set at a specified depth).  Example:  a[ ] = { 1, 2, 3, 4, { 5, 6 } }; a[ ] \*\*(1.5); // a[ ] now contains { 1.5, 3, 4.5, 6, { 7.5, 9 } }.  No impact on empty sets. | | | | | |
|  |  | | | | | |

1 Numeral and dates will change to *plain numeral* and *plain date* since applied calculations destroy the text representation.

Standalone usage example:

{

a[]++; // Increments a[] by 1

a[]++(5); // Adds 5 to a[]

++a[]; // Same functionality

++(5)a[]; // Adds 5 to a[]-

a[]++ ++; // Two increments

a[]++ --(5); // Subtracts of 4 (1-5) from a[]

a[]++(a[]); // Doubles a[]

a[]--(a[]); // A meaningless feature: Sets a[] to zero.

++a[]++; // Two increments

[table:Name,Lisa,Age]--; // Decrement number in one field in the table (age of Lisa)

[table:Age,..]++; // Increment all numbers in column under "Age"

}

Usage in assignments (left hand side):

{

a[]++ = 10; // Assigns 10 and increments by 1,. a[] contains 11

++a[] = 10; // Asignment is after increment, so a[] contains 10

++a[] += 10; // Value incremented, then added by 10.

a[]++ += 10; // Value added, then incremented by 10 (same output).

[table:Age,..]++ = 30; // Write 30 into all fields below "Age" and increments to 31

}

Usage in references (right hand side):

{

a[] = 10;

echo( --a[] ); // Output 9

b[] = a[]--; // b[] contains 9, a[] contains 8

b[] = --a[]; // Both b[] and a[] contain 7

echo( a[]++ ); // Outputs 7, a[] becomes 8

echo( --a[]++ ); // Reduce a[] to 7, output, then increement to 8.

}

Usage in function calls with I/O parameters

{

a[] = 5;

b[] = 15;

exchange( ++a[], b[]++ ); // Step 1: a[] becomes 6

// Step 2: a[] becomes 15 and b[] becomes 6

// Step 3: b[] becomes 7.

echo( --a[] ); // Output 9

b[] = a[]--; // b[] contains 9, a[] contains 8

b[] = --a[]; // Both b[] and a[] contain 7

echo( a[]++ ); // Outputs 7, a[] becomes 8

}

Do it yourself loop index in table process(…)

{

table process selected rows( table, ([Value]>1000) & (++i[]>0),

echo( i[], " and ", ++j[] );

}

The example above, i[ ] reflects the current row as it is incremented for every row being checked (see 2nd parameter), and j[ ] is only incremented if the conditions in the 2nd parameter are met. You can easily use these variables to refer to previous and next rows (e.g. i[ ]+1 and i[ ]-1 ). You also don't need to worry about initializing to zero if the variable des not yet exist yet.

For loops made convenient, like in "C/C++" programs:

{

for (a[]=1, a[]<=10, a[]++) echo(a[]);

}

Apply VAT to all prices in the table (ignore those fields which are blank) and postpone all delivery dates by 30 days (assumption: Table access is configured for automatic date recognition).

{

[table:Price,..] \*\*(vat[]);

[table:Delivery date,..] ++(30);

}

Alternatively, the following statements provide similar results, but require all fields with prices and dates filled unless table is configured to interpret blank fields as zero and do date recognition.

{

table process( table, [Price] \*= vat[]; [Delivery date]+=30 );

}

{

[table:Price,..] \*^= vat[]; // Deep operator

[table:Delivery date,..] +^= 30; // Deep operator

}

Performance-wise, the ad hoc operation, despite its limited functionality, is the fastest one as it does not interpret program code during execution in every row (1st example) nor does it generate (large) parameter sets as intermediate variables before doing the calculation before copying the parameter sets back into the table.

## Deep Operators

Deep Operators are a very unique feature specifically introduced to this language. Deep operators applied on parameter sets will apply the operator on the elements inside rather on the actual parameter set. Adding an accent circumflex symbol ' ^ ' behind operators will declare them as deep operators. Deep operators provide very effective and powerful means to do vector and even matrix calculations on each of the parameter elements. In order to have deep operators work correctly, the corresponding operands, which are corresponding elements in the parameter sets, must be of matching type.

Case 1: One parameter set with unary operators:

All valid unary operators are supported.

Examples: Assuming a[ ] = { 1, 2, 3, 4 } and b[ ] = { true, false }:

c[ ] = –^a[ ]; // Calculates { -1, -2, -3, -4 }

c[ ] = !^ b[ ]; // Calculates { false, true }

Case 2: Two parameter sets of equal length applied with binary operators:

All binary operators (arithmetic, Boolean and comparison) are supported. Following restrictions apply to comparison operators: No modifiers '+' to ignore case, '~' to laxen comparison, ',' to compare with more than 1 value, and '..' to compare with a range.

Examples: Assuming a[ ] = { 1, 2, 3, 4 } and b[ ] = { 5, 6, 7, 8 }:

c[ ] = a[ ] + b [ ]; // Calculates { 1, 2, 3, 4, 5, 6, 7, 8 }

c[ ] = a[ ] +^ b[ ]; // Calculates { 6, 8, 10, 12 }

c[ ] = b[ ] –^ a[ ]; // Calculates { 4, 4, 4, 4 }

c[ ] = { 1, H, date(today) } +^ { 2, i, 7 }; // Calculates { 3, Hi, (date of next week) }

c[ ] = { 1, abc } +^ { def, 2 }; // Exception – Two type mismatches

c[ ] = { 1, 2, 3, 4 } >=^ { 2, 2, 4, 4 } // Calculates { false, true, false, true }

c[ ] = sum( a[ ] \*^ b[ ] ) // Calculates sum of products, totaling 70

d[ ] = { true, true, false, false } !=^ { true, false, true, false}; // Example of Exclusive OR.

// Calculates { false, true, true, false }

Case 3: One parameter set and one scalar applied with binary operators:

Examples: Assuming a[ ] = { 1, 2, 3, 4 } and b[ ] = { 5, 6, 7, 8 }:

c[ ] = a[ ] –^ 3; // Calculates { -2, -1, 0, 1 }

c[ ] = 3 \*^ b[ ]; // Calculates { 15, 18, 21, 24 }

c[ ] = "The " +^ { house, dog, tree }; // Calculates { The house, The dog, The tree }

c[ ] = true !=^ { true, false }; // Calculates { false, true }

Case 4: Two parameter sets of different lengths applied with binary operators:

Subcase 4.1: Stuffing shorter parameter set with neutral values according to rules listed in the table below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Binary Operator** | **Value** | **Remarks** |
| literal | Any allowed operator | Blank literal |  |
| numeral | +, -  \*, / | 0  1 |  |
| date | Any allowed operator | 0 | For binary operators on dates, the other operand must be a number: |
| boolean | and  All other operators | 1  0 |  |
| parameter set | Any allowed operator | Empty set { } |  |

Examples: Assuming a[ ] = { 2, 3 } and b[ ] = { 5, 6, 7, 8 }:

c[ ] = a[ ] +^ b[ ]; // Calculates { 7, 9, 7, 8 } ( a[ ] extends to {2,3,0,0} )

c[ ] = a[ ] \*^ b[ ]; // Calculates { 10, 18, 7, 8 } ( a[ ] extends to {2,3,1,1} )

c[ ] = a[ ] +^ { }; // Empty set = 0 for addition. c[ ] is same as a[ ]

c[ ] = { } -^ a[ ]; // Calculates { -2, -3 } ( {} extends to {0,0,0,0} )

Subcase 4.2: Repeating parameter values in shorter parameter set:

Alternative to subcase 4.1, the colon suffix ' : ' added to the (last) circumflex symbol applies a different rule for shorter parameter sets: The existing pattern will be repeated until the size of the opposite parameter set has been reached. This suffix applies to binary operators only. If the shorter parameter set is empty, then subcase 4.1 applies.

c[ ] = a[ ] +^: b[ ]; // Calculates { 7, 9, 9, 11 } ( a[ ] extends to {2,3,2,3} )

c[ ] = a[ ] \*^: b[ ]; // Calculates { 10, 18, 14, 24 }

c[ ] = { Jim, Sam, Jane, Tom, Lisa } +^: {' says ',' said ',' meant '} +^: { Hi, Ho }

// Calculates {Jim says Hi,Sam said Ho, Jane meant Hi,Tom says Ho,Lisa said Hi}

Case 5: Nested deep operators – Enabler for matrix arithmetics:

Use of 2 or more succeeding circumflex symbols (e.g. '^^') allows deeper use of operators in nested parameters. This feature becomes useful when doing matrix operations. All data types with their supported operators are supported. The example below shows the different results with additions on numeric parameter sets, i.e. matrices:

Illustrated example on scalar multiplication (left) and matrix multiplication (right)



Further examples, assuming a[ ] = { {1,2},{3,4} } and b[ ] = { {5,6},{7,8} }:

c[ ] = a[ ] + b[ ] ; // Calcula tes { {1,2},{3,4},{5,6},{7,8} }

c[ ] = a[ ] +^ b[ ] ; // Calculates { {1,2,5,6},{3,4,7,8} }

c[ ] = a[ ] +^^ b[ ] ; // Calculates { {6,8},{10,12} }

d[ ] = a[ ] +^^ 10; // Calculates { {11,12},{13,14} }

d[ ] = a[ ] +^^: {10}; // Same result

d[ ] = a[ ] +^^: {{10}}; // Same result

d[ ] = a[ ] +^^ {10}; // Calculates { {11,12},{3,4} }

d[ ] = a[ ] +^^ {{10}}; // Calculates { {11,2},{3,4} }

d[ ] = a[ ] +^^ {10, 20}; // Calculates { {11,12},{23,24} }

d[ ] = a[ ] +^^: {{10},{20}}; // Same result

d[ ] = a[ ] +^^ {{10,20}}; // Calculates { {11, 22},{3,4} }

d[ ] = a[ ] +^^ {{10,20},{10,20}}; // Calculates { {11,22},{13,24} }

d[ ] = a[ ] +^^: {{10,20}}; // Same result

d[ ] = a[ ] +^^ {{10},{20}}; // Calculates { {11, 2},{23,4} }

**Accessing Tables:**

Table values are handled like literals or numerals. Deep operators are only possible if the other operand is a parameter set.

**Deep Assignment Operators:**

Deep assignment operators are only allowed on destination variables, but not on tables.

Examples:

a[ ] = { 1, 2, 3, 4 };

a[ ] -^= 2; // Calculates { -1, 0, 1, 2}

a[ ] \*= { 2, 3, 4, 5 }; // Calculates { -2, 0, 4, 10 }, using result from previous line

a[ ] +^:= { 0, 10 }; // Calculates { -2, 10, 4, 20}.

**Deep Function Call:**

A dedicated function call named 'deep' will be introduced which allows function calls in deep operations.

a[ ] = deep( abs, {{ -1, 2, -3, 4 }} );

## Type Conversions

Implicit type conversions take place under following circumstances:

* Writing data back into tables: All data will be converted to literal.
* Reading data from tables (applicable to all forms: assignments, function calls, transactions):  
  Type conversion is ruled by Boolean settings of the system variable family   
  **table conversion** [ ]. Default setting: only **table conversion**[ **numeral** ] is true, the rest is false. This means that everything, which looks like number (regardless if with or without decimal point), will be converted into a numeral. You are allowed to change the settings, allowing for converting dates, Boolean values and forcing blank entries to zero. See section 16 for details.
* Specific function calls (parameter types may be checked strictly or be converted to required type, e.g. numeral to literal).
* Arithmetic operations with Boolean values: false converts to 0 and true to 1.  
  Examples: true + true returns a 2, 3 \* false returns a zero. Unary operators: +true returns 1,   
  -true returns -1, and +false / -false return 0.
* Arithmetic operations with dates:  
  Adding days: Date + Numeral results in a date (postponed by n days)  
   Date – Numeral results in a date, too (preponed by n days)  
   Date – Date results in a number (number of days between these dates)  
   Date + Date is invalid. Exception will be asserted.

Explicit type conversions with available functions:

* numeral ( ) Converts boolean and literals to numerals.
* literal ( ) (will also convert void values into literal with explanatory message text inside)
* date ( ) Converts numeral and literal into date.
* boolean ( ) Converts 0, 1, No, Yes, True, False (not case sensitive) into Boolean
* clean numeral ( ) Converts literal with commercial style number representations,  
   e.g. "EUR (1'000.50)" to correct number, which is -1000.5 in this case.

# Transactions

## Transactions Basics

Transactions are elegantly structured assignment and manipulation features for exchanging information between variables (with their structure and array members) and tables. Transactions make use of dedicated assignment symbols called *transaction operators* which include copying, moving and swapping contents

|  |  |  |  |
| --- | --- | --- | --- |
| **Left hand destination** | **Transaction prefix** | **Transaction operator** | **Right hand destination** |
| Variables like a[ ], b[ c ], etc.  Tables like [ table1, 0, 1 ] | Simple replacement: No operator  And combination: **&**  Or combination: **|**  Add entry: **+** | Copy: **<== <==^**  Move: **<<= <<=^**  Swap: **<=> <=>^** | Variables like a[ ];, b[ c ];, etc.  Tables like [ table1, 0, 1 ]; |

**Suffix ^: Members only:** Applicable to variable-variable transactions

If this suffix '^' is appended to the transaction operator, then the transaction will only affect the member variables and array members, but the base variable contents remain unaffected on both sides of the operator. This operator applies only for transaction between variables (and no tables).

a[Hi] = He;

d1[] <== a[]; // Transaction

d2[] <== b[]; // Exception

d3[] <== a[Ho]; // Exception

d4[] <== a[Hi]; // Transaction initiated, but no members found.

## Transactions between Variables

Transactions are possible with simple variables (base variables without members), structures and arrays. Note that on both left and right hand side, variables need to be referenced directly. No arithmetic or logic operations, function calls and parentheses are allowed.

|  |  |  |  |
| --- | --- | --- | --- |
| **Correct Specifications** | **Explanation** | **Incorrect Specifications** | **Explanation** |
| a[ ]  a[1], a[b]  a[1,2], a[b,c], a[b,2], a[1,c] | Base variable  Chosen Member, will be treated as given base variable for the transaction  Chosen sub-member, same approach as above | a[ ] + 1  a[ ] + b[ ]  abs( a[ ] )  - a[ ]  ( a[ ] ) | Arithmetic  Arithmetic  Function call  Arithmetic  Parentheses  All these features use the value in the base value only and actual variable specification is lost. |

It does not matter if the variables on both sides are arrays, structures of a combination of both.

|  |  |  |  |
| --- | --- | --- | --- |
| **Transactions from variable to variable** | | | |
| **Transaction** | **Symbol** | **Example** | **Description of transaction sequence** |
| **Copy** | **<==** | a[ ] <== b[ ]; | Step 1a: If a[ ] does not yet exist, then it will be created  Step 1b: If a[ ] does already exist, then the variable will be initialized, including discarding all members and sub-members.  Step 2: Copies the value of the base value from b[ ] to a[ ] like in a regular assignment: a[ ] = b[ ];  Step 3: Copies all members and sub-members from b[ ] to a[ ] |
| **Copy Members** | **<==^** | a[ ] <==^ b[ ] | Step 1a: Like above  Step 1b: If a[ ] does already exist, then the members and sub-members will be discarded. The base variable will be preserved.  Step 2: Skipped because base variables shall not be touched  Step 3: Like above |
| **Move** | **<<=** | a[ ] <<= b[ ] | Steps 1-3: Same as for symbol <==  Step 4: Deletes variable b[ ] entirely |
| **Move Members** | **<<=^** | a[ ] <<=^ b[ ] | Steps 1-3: Same as for symbol <==^  Step 4: Deletes all members and sub-members of variable b[ ], but the base variable remains unaffected. |
| **Swap** | **<=>** | a[ ] <=> b[ ] | Step 1: All contents (base variable, members, sub-members) will be exchanged between a[ ] and b[ ] |
| **Swap Members** | **<=>^** | a[ ] <=>^ b[ ] | Step 1: All contents in members, sub-members will be exchanged between a[ ] and b[ ], but the base variables remain unchanged.  As an example, if a[ ] is a structure and b[ ] an array, then a[ ] will be an array and b[ ] a structure. |

Code example 1:

{

structure( animals1[], { mammal, bird, fish }, { dog, duck, trout } );

animals1[] = Mikes collection;

a[] <== animals1[];

b[] <==^ animals1[];

echo( a[], ", member names: ", set names( a[] ), " contents: ", set ( b[] ));

echo( b[], ", member names: ", set names( b[] ), " contents: ", set ( b[] ));

}

Output:

Mikes collection,member names: {bird,fish,mammal} contents: {duck,trout,dog}

# void value #,member names: {bird,fish,mammal} contents: {duck,trout,dog}

Code example 2:

{

structure( a[], { mammal, bird, fish }, { dog, duck, trout } );

a[] = Mike;

structure( b[], { bird, reptile, fish }, { swallow, crocodile, shark } );

b[] = Jane;

a[] <=> b[]; // The two owners just change position, but keep their pets.

echo( a[], " member names: ", set names( a[] ), " contents: ", set ( a[] ));

echo( b[], " member names: ", set names( b[] ), " contents: ", set ( b[] ));

echo(new line, "Exchange the pets ...");

a[] <=>^ b[];

echo(new line, "Jane gives the animals to Nick");

c[] = "Nick";

c[] <<=^ a[]; // But Jane inside base variable stays inside

echo( a[], ", member names: ", set names( a[] ), " contents: ", set ( a[] ));

echo( c[], ", member names: ", set names( c[] ), " contents: ", set ( c[] ));

echo(new line, "Nick gives all his animals to Mike's crocodile");

b[reptile] <<=^ c[]; // Use ^ symbol because Nick should not be eaten

echo( b[], ", member names: ", set names( b[] ), " contents: ", set ( b[] ));

echo(new line, "Lynn gets the same animals as Mike.");

d[] = Lynn;

d[] <==^ b[]; // Duplicate all members

echo( d[], ", member names: ", set names( d[] ), " contents: ", set ( d[] ));

}

Output:

Jane, member names: {bird,fish,reptile} contents: {swallow,shark,crocodile}

Mike, member names: {bird,fish,mammal} contents: {duck,trout,dog}

Exchange the pets ...

Jane, member names: {bird,fish,mammal} contents: {duck,trout,dog}

Mike, member names: {bird,fish,reptile} contents: {swallow,shark,crocodile}

Jane gives the animals to Nick

Jane, member names: {} contents: {}

Nick, member names: {bird,fish,mammal} contents: {duck,trout,dog}

Nick gives all his animals to Mike's crocodile

Mike, member names: {bird,fish,reptile,{bird,fish,mammal}} contents: {swallow, shark,crocodile,{duck,trout,dog}}

Lynn gets the same animals as Mike.

Lynn, member names: {bird,fish,reptile,{bird,fish,mammal}} contents: {swallow, shark,crocodile,{duck,trout,dog}}

## Transactions from Tables to Variables

Transactions provide a powerful way to read one row or specified part of the row and store the contents into a structure. For every item read from the table, a new member will be created, given the name of the corresponding column header name and the contents placed into the variable. Similar Implicit type conversions apply to reading the contents into the members, e.g. what looks like numbers will be converted into numerals unless configured otherwise.

Rules:

* Only simple and horizontal accesses on tables are allowed. Vertical and matrix accesses will be rejected and cause exceptions.
* Partial table specifications are allowed, provided the context information for the table and row number is available (e.g. inside the **table process**(…) call)
* Transactions from tables to variables can generate members, but no submembers
* Empty accesses create no members
* Void value is assigned to the base variable unless the **^** suffix is used behind the transction operator.
* Make sure every table header you want to work with is unique. Otherwise, the same member variable will be overwritten since the same name is used repeatedly.
* Blank table header names translate into a [""] (blank member name).

In the table below, the example table with the cities is used.

|  |  |  |  |
| --- | --- | --- | --- |
| **Correct Specifications** | **Explanation** | **Incorrect Specifications** | **Explanation** |
| Any simple and horizontal accesses are allowed:  [ table: , 1 ]  [ table: .., 1 ]  [ table: City, 1]  [ table: 1..3, 1]  [ table: ..2, 1 ]  [ table: -2.., 1]  [ table: {City, Country}, 1]  [ table: :'C\*' only)  [ table, .., {1} ] | Entire row 1, all items  "  Row 1, column "City" only  Columns selected by range  "  "  Columns chosen by a set  Columns chosen by a filter  Designates 1 row, also OK | Vertical and matrix accesses  [ table: City, 2..3 ]  [ table: .., .. ] | Vertical access  Matrix access |
| Partial table specifications:  [ ], [..]  [1..3], [..6], [5..]  [ {City,Country} ] | Entire row  Columns selected by range  Columns | Vertical and matrix accesses  [ City, 2..3 ]  [ .., .. ] | Vertical access  Matrix access |
| Lookups  { table: City, Boston, .. } | Entire row with Boston inside |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Transactions from table to variable** | | | |
| **Transaction** | **Symbol** | **Example** | **Description of transaction sequence** |
| **Copy** | **<==** | a[ ] <== [t: .., 1]; | Step 1a: If a[ ] does not yet exist, then it will be created  Step 1b: If a[ ] does already exist, then the variable will be initialized, including discarding all members and sub-members.  Step 2: Initializes the base variable a[ ] to a void value  Step 3: For every field retrieved from the table, a dedicated member will be created, named by corresponding table header name. Members are overwritten in case a header name is encountered for which a member has already been created. |
| **Copy Members** | **<==^** | a[ ] <==^ [t: ..,1] | Step 1a: Like above  Step 1b: If a[ ] does already exist, then the members and sub-members will be discarded. The base variable will be preserved.  Step 2: Skipped because base variables shall not be touched  Step 3: Like above |
| **Move** | **<<=** | a[ ] <<= [t: ..,1] | Steps 1-3: Same as for symbol <==  Step 4: Deletes the table row entirely. The rows below shift upwards. |
| **Move Members** | **<<=^** | a[ ] <<=^ [t: ..,1] | Steps 1-3: Same as for symbol <==^  Step 4: Deletes the table row entirely. The rows below shift upwards. |
| **Swap** | **<=>** | a[ ] <=> [t: ..,1] | Step 1: All contents (base variable, members, sub-members) will be exchanged between a[ ] and the table. |
| **Swap Members** | **<=>^** | a[ ] <=>^ [t: ..,1] | Step 1: All contents in members, sub-members will be exchanged between a[ ] and b[ ], but the base variables remain unchanged. |

If you want to use the transaction to create an array instead of a structure, then consider following two steps:

1. Delete the variable if already existing
2. Use the operator + in front of the transaction, e.g. a[ ] +<== [t:..1];  
   The size of the array will be adjusted automatically.
3. Attention: Further operations with this transaction will add further elements to the variable.

## Transactions from Variables to Tables

This is the opposite direction where member contents of structures as well as arrays are transferred into tables. As long the transaction operator has no assignment operator (discussed in later sections), then the destination row will be cleared entirely before writing.

Structures to tables: The member names will be matched with the existing column header names, provided they are part of the table specification (where whole row, a set of columns, a range or just one column is defined). If a member name cannot be matched with any column header, then an additional header with that name will be introduced and the data will be written into the designated row.

Arrays to tables: The array members will be written into the table as defined in the table specification (whole row, set of columns, a range or just one column). Normally, the members are written to the table from left to right except if a set of columns is specified. In this case, the sequence inside the set will apply. If the array contains more elements than the number of columns provided, then the remaining elements will not be written into the table. Only exception: Open-ended ranges, for example: **[t:5..,1]**, **[t:..,1]**, **[t:, 1]**.

|  |  |  |  |
| --- | --- | --- | --- |
| **Transactions from table to variable** | | | |
| **Transaction** | **Symbol** | **Example** | **Description of transaction sequence** |
| **Copy** | **<==** | [t: .., 1] <== a[ ]; | Step 1: Checks that exactly one table row and one or more or all columns are specified.  Step 2: Clears the contents in the specified table row  Step 3a: if a[ ] is a structure, then all members are checked for existing header names and, if found, the contents are written to the table. This applies to those columns explicitly specified in the table specification.  If the column header does not exist and the table specification does not restrict column selection, then additional columns are created.  Step 3b: If a[ ] is an array, then member 0 through the last ones are copied into the corresponding table columns as specified in the table specification.  Additional columns are created if the table specification denotes to an open end range, e.g. [t: 5.., 1], [t: .., 1], [t:, 1]. |
| **Copy Members** | **<==^** | [t: .., 1] <==^ a[ ]; | Same as above since base variable contents are not copied. |
| **Move** | **<<=** | [t: .., 1] <<= a[ ]; | Steps 1-3: Same as for symbol <==  Step 4: Deletes the table row entirely. The rows below shift upwards. |
| **Move Members** | **<<=^** | [t: .., 1] <<=^ a[ ]; | Same as above since base variable contents are not copied. |
| **Swap** | **<=>** | [t: .., 1] <=> a[ ]; | Step 1: All contents (base variable, members, sub-members) will be exchanged between a[ ] and the table. |
| **Swap Members** | **<=>^** | [t: .., 1] <=>^ a[ ]; | Step 1: All contents in members, sub-members will be exchanged between a[ ] and the table, but the base variable remains unchanged. |

Code example on transactions from and to tables:

{

table load( table, "Cities.csv" );

a[] <== [table:..Country,1];

echo( "member names: ", set names( a[] ), " contents: ", set ( a[] ));

c[City] = Köln;

c["State / Province"] = NRW;

c[Famous attraction] = Dome;

c[Famous cultural place] = Walraff Richards Museum;

c[Country] = GER;

echo(new line, Exchange Cologne with NYC);

c[] <=> [table:,1]; // Cologne replaces New York City

echo("Whereabouts of orig. variable for cologne: ", c[City]);

echo([table:,0], new line, [table:,1], new line, [table:,2]);

echo(new line,Remove Cologne);

c[] <<= [table:,1];

echo([table:,0], new line, [table:,1], new line, [table:,2]);

echo("Whereabouts of orig. variable for cologne: ", c[City]);

echo(new line,Put Düsseldorf into table);

d[City] = Düsseldorf;

d["State / Province"] = NRW;

c[Famous attraction] = Altstadt;

d[Inhabitants] = 604000;

d[Famous cultural place] = Tonhalle;

d[Country] = GER;

d[Famous Person] = Robert Schumann;

d[Drink] = Altbier;

[table:..3,2] <== d[];

echo([table:,0], new line, [table:,2]);

echo(new line,Add Montreux where data is stored in an array);

array( m[], { Jazz Festival, SWI, Montreux, 390, sunshine } );

[table:{Famous attraction,Country,City,"Alt. (m)"},3] <<= m[];

echo([table:,3], " and the array is ", identify(m[]));

}

Output:

member names: {City,Country,State / Province} contents: {New York City,USA,New York}

Exchange Cologne with NYC

Whereabouts of orig. variable for cologne: New York City

{City,State / Province,Country,Inhabitants,Famous attraction,Alt. (m),Famous cultural place,  
Moving along}

{Köln,NRW,GER,,Dome,,Walraff Richards Museum}

{Washington,D.C.,USA,650000,Lincoln Statue,7,Smithsonian Institute,Delivery drone}

Exchange Cologne with NYC

Whereabouts of orig. variable for cologne: Köln

{Washington,D.C.,USA,650000,Lincoln Statue,7,Smithsonian Institute,Delivery drone}

{Philadelphia,Pennsylvania,USA,1500000,Independence hll,12,,PPC streetcar}

{City,State / Province,Country,Inhabitants,Famous attraction,Alt. (m),Famous cultural place,  
Moving along,Drink,Famous Person}

{Düsseldorf,,GER,,,,,,Altbier,Robert Schumann}

Add Montreux where data is stored in an array

{Montreux,,SWI,,Jazz Festival,390} and the array is not found.

Düsseldorf: The table specification describes a horizontal access covering the first four columns. For this reason, the *Inhabitants*, *Famous attraction* and *Famous cultural place* are not updated. However the two new columns have (*Drink*, *Famous Person*) have been added.

Montreux: A set specifies the right columns and sequence in order to update the table correctly. The weather condition (sunshine) was not added because the column specification is not open ended. The original array **m[ ]** is no longer found because the **<<=** operator deleted the source afterwards.

## Transactions between Tables

Transactions are possible between two table rows inside the same table or between two different tables. The **^** suffix has no impact into these operations since no variables are involved.

|  |  |  |  |
| --- | --- | --- | --- |
| **Transactions from table to table** | | | |
| **Transaction** | **Symbol** | **Example** | **Description of transaction sequence** |
| **Copy** | **<==** | [t: .., 1] <== [u: .., 1]  [t: .., 1] <== [t: .., 2] | Step 1: Checks that exactly one table row and one or more or all columns are specified in both source and destination tables  Step 2: Clears the contents in the specified table row in the destination table.  Step 3: Copies all selected items from the source table to the destination table as long they are also selected in the destination table. It's like an intersection of the column headers selected in the source and the destination table. If no overlap exists, then nothing will be copied, for example: [t: {City, Country}, 3] <== [t: {State, Street}, 4]; |
| **Copy Members** | **<==^** | [t: .., 1] <==^ a[ ]; | Same functionality because variables are not involved. |
| **Move** | **<<=** | [t: .., 1] <<= a[ ]; | Steps 1-3: Same as for symbol <==  Step 4: Deletes the row in the source table entirely. The rows below shift upwards. |
| **Move Members** | **<<=^** | [t: .., 1] <<=^ a[ ]; | Same functionality because variables are not involved. |
| **Swap** | **<=>** | [t: .., 1] <=> a[ ]; | Step 1: All contents (base variable, members, sub-members) will be exchanged between the two table rows. |
| **Swap Members** | **<=>^** | [t: .., 1] <=>^ a[ ]; | Same functionality because variables are not involved. |

Code example:

{

table load( t1, "Cities.csv" );

table initialize( t2,

{ { Country, Moving along, City, Famous attraction, Drink },

{ GER , Hoverboard, Düsseldorf, Tonhalle, Altbier },

{ GER , By foot, Köln, Dome, Kölsch },

{ SWI , Mountain bike, Montreux, Châtelard castle }

} );

[t2:,4] <== [t1:{Country,Moving along,City,Famous attraction,"State / Province"},1];

[t2:,5] <<= [t1:{Country,Moving along,City,Famous attraction,"State / Province"},2];

echo([t2:,0]);

table process( t2, echo([]) );

[t1:,1] <== [t2:,1]; // Note: It creates a new column named "Drink".

echo(new line,"Row 1 should contain Düsseldorf, Row 2 should be Phily.");

echo([t1:,1], new line, [t1:,2]);

}

Output:

{Country,Moving along,City,Famous attraction,Drink,State / Province}

{GER,Hoverboard,Düsseldorf,Tonhalle,Altbier}

{GER,By foot,Köln,Dome,Kölsch},

{SWI,Mountain bike,Montreux,Châtelard castle}

[USA,Taxi,New York City,St. of Liberty,,New York}

{USA,Delivery drone,Washington,Lincoln Statue,,D.C.}

Row 1 should contain Düsseldorf, Row 2 should be Phily.

{Düsseldorf,,GER,,Tonhalle,,,Hoverboard,Altbier}

{Philadelphia,Pennsylvania,USA,1500000,Independence hall,12,,PCC streetcar}

## Transaction Assignment Operators

Transaction assignment operators provide additional powerful features which allow existing contents in the destination tables and variables to be preserved and only selectively overwritten where intended to do so

### No Assignment Operator

This is a repetition of the basic table access functionality

|  |  |  |
| --- | --- | --- |
| **Assignment Operator:** | | **None Combinations: <== <<= <=>** |
| **Destination** | **Source** | **Description** |
| Nonexistent Simple Structure Array  *Changes to:* Structure | Structure | Step 1: The destination is initialized as specified, i.e. old members will be deleted entirely.  Step 2: The base variable is copied if the assignment operator has no ^ suffix.  Step 3: The members are transferred from source to destination. Missing members will be created. |
| Nonexistent Simple Structure Array  *Changes to:* Array | Array | Step 1: The destination is initialized as specified, i.e. old members will be deleted entirely.  Step 2: The base variable is copied if the assignment operator has no ^ suffix.  Step 3: The members are transferred in alphabetical order from source to destination. The destination contains no header names. |
| Nonexistent Simple Structure Array  *Changes to:* Structure | Table | Step 1: The destination is initialized as specified, i.e. old members will be deleted entirely.  Step 2: The base variable is initialized if the assignment operator has no ^ suffix.  Step 3: The selected table rows are read and added as named elements to the destination structure. Missing members will be created. |
| Table | Simple | Step 1: The specified table row contents will be cleared.  No further actions because the source variable contains no members. |
| Table | Structure | Step 1: The specified table row contents will be cleared.  Step 2: For each member, the member name will be matched with the corresponding table header, provided the header is part of the table specification. - If a matching table header exists, use that column. - If no match found, then add a new table header at the end of the table  and use that new column  Step 3: Write the contents in the member variable into the corresponding field in the date. |
| Table | Array | Step 1: The specified table row contents will be cleared.  Step 2: Each member will be copied into the columns as given in the table specification. - If the table specifies fewer number of columns the number of elements, then the remaining elements will not be transferred.   Example: [t:1..2,1] = a[ ]; // a[ ] contains 3 members. - If the table specifies open end range (e.g. [t:3..,1], t[..,1], [t:,1]), then the  contents will be written into further columns to the right, even if these  columns have no column header names. - If the table specifies more columns, then the excessive columns will not  be updated. |
| Table | Table | Step 1: The specified table row contents will be cleared.  Step 2: Each column as specified in both table specifications and provided if existing will be copied from one table row to the other. |

### The & (And) Assignment Operator

The & (and) operator considers following simple rules to carry out modified transactions:

* Contents in the destination will not be cleared
* Type of destination (array or structure) will be preserved. If destination is not yet existing, then a structure will be built up.
* Contents may be overwritten
* No new members, or if destination is a table, new table columns will be created.

|  |  |  |
| --- | --- | --- |
| **Assignment Operator:** | | **& Combinations: &<== &<<= &<=>** |
| **Destination** | **Source** | **Description** |
| Nonexistent Simple  *Changes to:* Structure | Structure Array | Step 1: The destination is initialized as specified.  Step 2: The base variable is copied if the assignment operator has no ^ suffix.  No further actions because destination has no prior members to match. |
| Structure | Structure | Step 1: The base variable is copied if the assignment operator has no ^ suffix.  Step 2: Members with matching member names will be copied. All other members in the source will not be copied. All other members in the destination remain unaffected. |
| Array | Structure | Step 1: The base variable is copied if the assignment operator has no ^ suffix.  Step 2: The existing array members will be replaced by members from the source variable, sorted in alphabetic order by member name. - Excess destination members remain unaffected. - If the destination has fewer members than the source, then not all  members will be copied. |
| Structure | Array | Step 1: The base variable is copied if the assignment operator has no ^ suffix.  No further actions because no member names are available to match. |
| Array | Array | Step 1: The base variable is copied if the assignment operator has no ^ suffix.  Step 2: The existing array members will be replaced from the members in the source variable in unchanged sequential order. - Excess destination members remain unaffected. - If the array is too small, then the remaining members will not be copied. |
| Structure | Table | Step 1: The base variable is initialized if the assignment operator has no ^ suffix.  Step 2: Data will be copied from the specified table columns into the structure as long as existing member names match with the corresponding column header names. - No further members are created - All other members in the destination remain unaffected. |
| Array | Table | Step 1: The base variable is initialized if the assignment operator has no ^ suffix.  Step 2: Data will be copied from the specified table columns into the existing array members starting with the first one and in the sequential order as specified in the table specification. - Excess destination members remain unaffected. - If the array is too small, then the remaining members will not be copied. |
| Table | Simple | No actions because the source variable contains no members. |
| Table | Structure | Step 1: Only the members which names match with the table header names will be copied into the corresponding columns. - No additional table columns will be created. - All other contents in the table row remain unaffected. |
| Table | Array | Step 1: The columns specified in the table specification will be overwritten in sequential order. - No additional table columns will be created if the array is bigger - If the array is smaller, then remaining table columns remain unchanged. |
| Table | Table | Step 1: The columns with matching header names in both tables, provided they are specified in the table specification, will be copied. - No additional table columns will be created. |

### The | (Or) Assignment Operator

The | (or) operator considers following simple rules to carry out modified transactions:

* Contents in the destination will not be cleared
* Type of destination (array or structure) will be preserved. If destination is not yet existing, then a structure will be built up.
* Contents may be overwritten
* New members, or if destination is a table, new table columns will be created if needed.

|  |  |  |
| --- | --- | --- |
| **Assignment Operator:** | | **| Combinations: |<== |<<= |<=>** |
| **Destination** | **Source** | **Description** |
| Nonexistent Simple Structure  *Changes to:* Structure | Structure | Step 1: The base variable is copied if the assignment operator has no ^ suffix.  Step 2: All members will be copied. Existing destination members will be overwritten. New members are added if the member names do not yet exist. |
| Array | Structure | Step 1: The base variable is copied if the assignment operator has no ^ suffix.  Step 2: The existing array members will be replaced by members from the source variable, sorted in alphabetic order by member name. - Excess destination members remain unaffected. - The array will be extended if needed. |
| Structure | Array | Step 1: The base variable is copied if the assignment operator has no ^ suffix.  Step 2: Member names will be created for each array member: Literal representation of index numbers with 8 digits ("00000000", "00000001", "00000002"), etc. The leading zeros will preserve the sorting order in order to avoid the "1", "10", "11", … "2" sequence.  Step 3: Continue with Step 2 described for structures in the section above. |
| Array | Array | Step 1: The base variable is copied if the assignment operator has no ^ suffix.  Step 2: The existing array members will be replaced from the members in the source variable in unchanged sequential order. - Excess destination members remain unaffected. - The array will be extended if needed. |
| Structure | Table | Step 1: The base variable is initialized if the assignment operator has no ^ suffix.  Step 2: Data will be copied from the specified table columns into the structure in the sequence specified as long as existing member names match with the corresponding column header names. - Missing members will be created. |
| Array | Table | Step 1: The base variable is initialized if the assignment operator has no ^ suffix.  Step 2: Data will be copied from the specified table columns into the array in the sequence specified. - Excess destination members remain unaffected. - The array will be extended if needed. |
| Table | Simple | No actions because the source variable contains no members. |
| Table | Structure | Step 1: Only the members which names match with the table header names will be copied into the corresponding columns. - If header names are missing, then they will be added. - All other contents in the table row remain unaffected. |
| Table | Array | Step 1: The columns specified in the table specification will be overwritten in sequential order. - Additional table columns will be created if the array is bigger, but  without header names! - If the array is smaller, then remaining table columns remain unchanged. |
| Table | Table | Step 1: The columns with matching header names in both tables, provided they are specified in the table specification, will be copied. - Additional table columns will be created. |

### The + (Plus) Assignment Operator

The + (plus) operator considers following simple rules to carry out modified transactions:

* Contents in the destination will not be cleared
* If the destination is an array, then it will be extended to fit the additional data. Existing members are not overwritten.
* If the destination is a structure, then it behaves similar the "or" assignment operator, but values in already existing target members will not be overwritten.
* If the destination is a simple variable or nonexistent, then it will be initialized as an array.
* If the destination is a table row, then a new blank row will be inserted and the remaining rows below will shift down accordingly before the transaction starts.

|  |  |  |
| --- | --- | --- |
| **Assignment Operator:** | | **+ Combinations: +<== +<<= +<=>** |
| **Destination** | **Source** | **Description** |
| Nonexistent Simple  *Changes to:* Array | Structure  Array | Step 1: The base variable is copied if the assignment operator has no ^ suffix.  Step 2: An array will be created and data copied from the source, sorted in alphabetic order by member name if the source is a structure, or by the given order of the source array. |
| Structure Array  *Changes to:* Array | Structure  Array | Step 1: The base variable is copied if the assignment operator has no ^ suffix.  Step 2: If the destination is a structure, then it will be converted to an array. The ordering corresponds with the previous alphabetic order of the member names.  Step 3: The array will be extended with the data copied from the source. |
| Structure Array  *Changes to:* Array | Table | Step 1: The base variable is initialized if the assignment operator has no ^ suffix.  Step 2: See Step 2 in the previous section.  Step 3: The array will be extended with the data copied from the specified table row and selected table columns. |
| Table | Simple | Step 1: A new empty row will be inserted, the rows below move downwards.  No further actions because the source variable contains no members. |
| Table | Structure | Step 1: A new empty row will be inserted, the rows below move downwards.  Step 2: For each member, the member name will be matched with the corresponding table header, provided the header is part of the table specification. - If a matching table header exists, use that column. - If no match found, then add a new table header at the end of the table  and use that new column  Step 3: Write the contents in the member variable into the corresponding field in the date. |
| Table | Array | Step 1: A new empty row will be inserted, the rows below move downwards.  Step 2: Each member will be copied into the columns as given in the table specification. - If the table specifies fewer number of columns the number of elements, then the remaining elements will not be transferred.   Example: [t:1..2,1] = a[ ]; // a[ ] contains 3 members. - If the table specifies open end range (e.g. [t:3..,1], t[..,1], [t:,1]), then the  contents will be written into further columns to the right, even if these  columns have no column header names. - If the table specifies more columns, then the excessive columns will not  be updated. |
| Table | Table | Step 1: A new empty row will be inserted, the rows below move downwards.  Step 2: Each column as specified in both table specifications and provided if existing will be copied from one table row to the other. |

### Transaction Assignment Operators: Examples

Code examples targeting structures:

{

structure( a[], { Name, Street, Town }, { Jane, Main St., Naples } );

structure( b[], { Town, Name, State }, { Florence, Linda, Tuscany } );

echo(set(a[])); // Outputs {Jane,Main St.,Naples}

echo(set(b[])); // Outputs {Linda,Tuscany,Florence} Name is first in alphabetic order

c[] <== a[]; // Make a copy

c[] &<== b[]; // Overwrites Name and Town.

c[] |<== b[]; // Overwrites Name and Town, and adds State

c[] <== a[];

c[] +<== b[]; // Adds State, but leaves existing items unchanged.

echo(set(c[])); // Outputs {Jane,Tuscany,Main St.,Naples}

}

Code examples targeting arrays:

{

array( a[], { Ha, He, Hi, Ho, Hu } );

array( b[], { Hi, Hello, Ahoi } );

c[] <== b[]; // Make a copy

c[] &<== a[]; // Hi, Hello and Ahoi are replaced by Ha, He, Hi

echo( set(c[])); // Outputs {Ha,He,Hi}

c[] <== a[];

c[] |<== b[];

echo( set(c[])); // Outputs {Hi,Hello,Ahoi,Ho,Hu}

c[] <== a[];

c[] +<== b[]; // Plus operator will always convert to array

echo( identify(c[]),":", set(c[])); // Output: array {Ha,He,Hi,Ho,Hu,Hi,Hello,Ahoi}

}

Code examples targeting tables:

{

table load( t1, "Cities.csv" );

ny[City] = "Big Apple";

ny[Country] = "U.S.";

ny[Famous person] = Frank Sinatra;

// This assignment replaces city and country, but preserves remaining contents

[t:,1] &<== ny[];

// And this operator also adds the missing column

[t:,1] |<== ny[];

dus[City] = Düsseldorf;

dus[Famous Person] = Robert Schumann;

// Moves NYC and other rows down by 1 row, row 1 becomes vacant.

// Adds Düssledorf (City and new column Famous Person) to row 1.

[t:,1] +<== dus[];

}

Again, all these transaction operators with assignment operator prefixes may also carry the ^ suffix to avoid manipulating the base variable. This happens independently from whether a and which operator is chosen.

## Parameter Sets, Arrays, Structures, Tables: Summary

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Data conversions and transactions** | | | | |
| from below | to Parameter set | to Array | to Structure | to Table |
| Parameter set | Assignment  a[ ] = b[ ]; | **array (…)** function  array( a[ ]. {…} ) | **structure (…)** function  structure( a[ ], {…}, {…} ) | horizontal, vertical and matrix write access to table  Alternatively with functions like: **table write row(…)** |
| Array | **set (…)** function  s[ ] = set( a[ ] ); | Transaction  a[ ] <== b[ ]; | Function **array to structure(…)**  array to structure( a[ ] );  Transactions with assignment operators & or | (or).  a[ ] |<== b[ ]; | Transaction  [t: .., 1] <== a[ ]; |
| Structure | **set (…)** function For member names: **set names (…)**  s[ ] = set( a[ ] ); n[ ] = set names( a[ ] ); | Function **structure to array** (…)  structure to array( a[ ] );  Transaction with assignment operator plus:  a[ ] +<== b[ ]; | Transaction  a[ ] <== b[ ]; | Transaction  [t: .., 1] <== a[ ]; |
| Table | horizontal, vertical and matrix read access to table  Alternatively with functions like: **table read row(…)** | Transaction with assignment operator plus  delete( a[ ] ); // if existing a[ ] +<== [t: .., 1]; | Transaction  a[ ] <== [t: .., 1]; | Transaction  [t: .., 1] <== [u: .., 1]; |
| **Further actions:** | | | | |
| Initialization | Assignment  a[ ] = { … }; | Functions including:  **dim(…), redim(…), array(…),** etc. | Assignments:  a[a1] = Hi; a[a2] =Ho;  Function **structure**(…) | Functions including:  table create (…) table initialize (…) table load (…) |
| Elimination | delete( a[ ] ); | delete members( a[ ] ); | members( a[ ] ); | table delete (…); |

# Special Features

## Exceptions

The language interpreter has an exception handling model which handles start and termination of application programs as well as exceptions resulting from errors. Following exceptions are distinguished:

|  |  |  |
| --- | --- | --- |
| **Exception** | **Invoked by** | **Description** |
| start | Program start  **start**() function called | A program will be loaded, compiled and started. Calling programs can also start other programs. When they have ended, the calling program continues with the next instructions following the **start**() function.  The start exception passes through interactive mode without stopping there. |
| stop | **stop**() function called  Most error exception (e.g. file not found) | The program will be stopped and returns to interactive mode, allowing users to debug and restart the program.  When typing 'return', control resumes to calling program or ends if the main program was left before. |
| interactive | **interactive**() function called | The interactive mode is called. Returning from interactive mode will resume program execution with the next instructions falling the **interactive**() function. |
| return | **return** command in interactive mode  Program execution: Leaving function | Return to calling program. |
| end | **end**() function called.  End of program reached | The current program (or interactive mode) will be ended and control resumes to calling program or ends if the main program was ended.. |
| abort | **abort**() function called  Some error exceptions (e.g. memory overflow) | Immediate program stop. |
| continue | Some error exceptions (not severe ones, are more a kind of warnings) | Issues exception to document an error or warning message, then continues program execution immediately. |
| pause | **pause**() function called | Does "Press ENTER to continue". |
| exit | **exit()** function called | Normal program termination. Execution will end entirely, even if program is called by a superior program. |
| throw | **throw**() function called | User defined exception. Execution stops at this place. Further code is "flown over" without entering nested program blocks or conditional statements until **catch**() is encountered or end of program has been reached. In the latter case, **throw**() behaves like **exit**(). |
| catch | **catch**() function called | Catches throw exceptions encountered earlier in the program and continues following the **catch**() function. |



## JavaScript Open Notation (JSON) Data Format

Beyond4P supports the JavaScript Open Notation format to load and save both table and variables. The JSON format is based on the standard described in [www.json.org](http://www.json.org). The JSON standard has no special ruling for comments. However, Beyond4P accepts comments specified in C/C++ format as comments: // for line comments and /\* … \*/ for commented sections. Others use comments as part of data in objects, e.g. { "" : "a comment?" , "\_comment" : "another comment?" } . In these cases, both will be fully read into Beyond4P variables and you need to ignore them afterwards: a[''] = "a comment?", and a['\_comment'] = "another comment?".

Beyond4P translates all JSON escape sequences as ruled, including. \\, \" ,\n, \u20ac. The last one is the Euro sign.

When saving variables into JSON text, or loading JSON text into variables, Beyond4P is smart enough to detect dates and parameter sets.

Dates must be in a strict format (no extra spaces tolerated) in order to be translated into dates automatically. Format must be "YYYY-MM-DD", "YYYY-MM-DD hh:mm:ss", "hh:mm:ss", or "0000-00-00" for blank dates. All values must be numeric.

Parameter sets are also properly handled. They are parsed and the parameter set structure is built up accordingly. Example: "{a,b,{1,2}}" will automatically build up the parameter set. Prerequisite: Must begin and end with braces.

Following functions support JSON:

* **table load** (…, JSON …)
* **table save** (…, JSON …)
* **variable load** (…)
* **variable save** (…)
* **variable to json** (…)
* **json to variable** (…)
* **attribute …** (…) function

Restriction specific to Beyond4P: JSON numbers with scientific notations are not yet supported, e.g. 3.14E+03. Will be enabled in the near future.

# Standard Function and Procedure Library

### Registry Functions (Windows only)

**WARNING!**

**Beyond4P's registry functions are considered depreciated and will no longer be supported for user programming. To store user settings and other data or variables you want to make resident, consider attribute functions or JSON data format instead.**

The following functions accesses the **Microsoft Windows Registry**. Please note that the reliable operation of your computer, the operating system and applications relies on valid contents in the registry. Any form of manipulations without knowledge of the impacts may severely affect the system. **For this reason, functions doing manipulations on the registry** (creating folders, writing, deleting data and folders) **are restricted to the following isolated location:**

**HKEY\_CURRENT\_USER: Software\Beyond4P\My Space**

Inside this dedicated location, you can freely add and remove further subkeys (folders), write and read data without causing harm elsewhere. This space is specific to your personal computer and your user name. Other users sharing your machine will not see your data. However, read accesses are possible in any registry location as long user access privileges are given.

The path is also available in the system variable (read-only) called **runtime settings [ registry my space ]**

Applications which require extended write access must use an **activation code** (20 bytes in hexadecimal format) or with a license providing higher privileges (e.g. **supervisor** privileges) which can be provided upon legitimate request. Such a code is provided in the installation script. The code will only be valid with the given script. Any manipulations in the script will void the activation and a new activation code is required.

**$$** **activation code** = 11 12 13 14 21 22 23 24 31 32 33 34 41 42 43 44 51 52 53 54; // Example

**Referencing the registry: Not case-sensitive:**

In contrast to variables, function names, header names, etc., which are case sensitive, the registry keys (folders) as well as the entry names are not case sensitive.

The registry stores data in following types:

|  |  |  |  |
| --- | --- | --- | --- |
| **Registry Data Type** | **Beyond4P Data Type** | **Representation when reading the value** | **Example** |
| REG\_BINARY | Literal | Hexadecimal contents. Bytes (2 Hex values) separated with spaces. Every line contains 16 values.  For write access: Use valid hexadecimal symbols (0-9,A-F or a-f; blanks, tabs and new lines are allowed). Separation allowed between even number of hex digits. | 10 5A 92 3E 15 14 E4 FF 91 A2 B3 CE D6 76 F1 00 03 12 |
| REG\_DWORD | Plain Numeral | 32 bit integer is converted into numeral format and vice versa | 10002000 |
| REG\_DWORD\_ BIG\_ENDIAN | Plain Numeral | Same as above, but byte ordering is reverse: Lowest byte address begins with most significant 8 bits. | 10002000 |
| REG\_EXAND\_SZ | Literal | Literal contents, contains references to environment variables, e.g. %PATH% or %USERPROFILE%  Write access: Must always be literal. | %USERPROFILE%\Own Documents |
| REG\_LINK | Literal | Registry links, stored as literals. | Contains cross references to other registry entries |
| REG\_MULTI\_SZ | Parameter set containing literals | Multiple strings.  Contents will be converted into literal if they are not literals. | { Hello, Hi, He, Ho } |
| REG\_NONE | Boolean | No data in this entry.  *False* will be returned all time.  Write access: Value of type Boolean expected (e.g. *true* or *false*), actual value will be ignored.  Write access: Must always be literal. |  |
| REG\_QWORD | Numeral  Write access: Numeral (double precision value); Literal (hex code) | Quadruple word (64 bits) will be provided as literal containing hexadecimal representation and a numeral which is a 64-bit double precision floating point number (size is same as size of Qword).  Note that the value read contains both numeric and literal representation.  Use **literal**(…) to isolate the literal contents in hexadecimal format. First symbol is most significant.  Use **numeral** (…) to isolate the numeric contents.  Write access: You can either provide a literal containing a hexadecimal value (max. 16 hex digits) or a numeral (equivalent to double precision floating point value). | Numeric representation: 2.5  Literal representation: 4004000000000000 (hex code) |
| REG\_SZ | Literal  Write access: Any type | Literal contents.  Write access: The value provided will be converted into literal format if not already being of literal format. This is particularly useful when writing dates. | Hello World |

All registry functions can be either used with the suffix "**silently**" or without. If "**silently**" is not specified (e.g. **registry read value** (…), then any error results in an exception and program execution will stop. If "**silently**" is added to the name (e.g. registry read value **silently** (…)), then short text error messages of *literal* type are returned.

Registry functions: Return values and error messages

|  |  |
| --- | --- |
| **Short message (Literal)** | **Full error message and description** |
| OK | No error has occurred. |
| Registry bad HKEY | The HKEY to access the registry is invalid or does not exist.  Check the spelling. The following HKEYs are supported: HEKY\_CLASSES\_ROOT, HKEY\_LOCAL\_MACHINE, HKEY\_CURENT\_USER, HKEY\_USERS, HKEY\_CURRENT\_CONFIG |
| Registry can't open | The registry cannot be opened with the HKEY provided.  You may probably have insufficient access privileges. |
| Registry entry not found | Access to registry entry failed (not existing).  The entry is probably not existing in the folder (key) specified. Check the spelling. |
| Registry can't read | Read access to registry not possible or allowed.  Check path and entry name as well as access rights. |
| Registry can't write | Write access to registry not possible or allowed.  Check path and entry name as well as access rights. |
| Registry unsupported format | The value read from the registry has a data format not supported by Beyond4P.  This error should normally not happen since all existing data formats in the present and past Windows versions are supported. |
| Registry invalid data type | The registry specific data type of the value to write to the registry is invalid.  Following data types are valid:  REG\_BINARY, REG\_DWORD, REG\_DWORD\_BIG\_ENDIAN, REG\_EXPAND\_SZ, REG\_LINK, REG\_MULTI\_SZ, REG\_NONE, REG\_QWORD, REG\_SZ |
| Registry incompatible types | The type of the value to write is incompatible with the registry data type specified.  E.g. writing a literal value to a registry entry of type REG\_DWORD would cause this error. |
| Registry improper hex code | The data provided in hexadecimal format is improper.  When providing hex code to data of type REG\_BINARY: 1) Total number of hex digits must be even 2) Blanks may be put between even number of hex digits (e.g. CA FE AF FE, or CAFE AFFE. Invalid: CAF E A FFE) 3) Invalid characters (Only **0**-**9** **A**-**F** and **a**-**f** as well as space ' ', new line and tabs are allowed)  When providing hex code to data of type REG\_QWORD: 1) Total number of hex digits must be 16 or less. Odd number of digits are allowed. 2) Invalid characters (see above) |
| Registry key already exists | Attempted to create a registry key (folder) which is already existing. |
| Registry other error | An undocumented error has occurred while accessing the registry  This error message should normally not occur. |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Function** | | **registry read value registry read value silently** | | | | |
| Description | | Reads a dedicated Windows registry entry  The ending "silently" will not issue an exception if the read access fails. | | | | |
| Call as | | **X** | Procedure call | | **X** | Function call, providing a return value |
| Parameter count | | 4 - 5 | | | | |
| Parameters | 1, … | Literal | | HKEY  Takes one of the following values: HKEY\_CLASSES\_ROOT, HKEY\_LOCAL\_MACHINE, HKEY\_CURENT\_USER, HKEY\_USERS, HKEY\_CURRENT\_CONFIG | | |
|  | 2. | Literal | | Registry Folder Path (keys) | | |
|  | 3. | Literal | | Registry Entry Name | | |
|  | 4. Out | Depends on data type | | Value  See next table below on the data type used. It depends on the data type of the value stored in the registry. | | |
|  | 5. Out, Opt. | Literal | | Registry Data Type  See next table. Example: REG\_BINARY, REG\_DWORD, etc. | | |
| Return value | | Literal | | Short error messages as described in the introductory part of this section | | |
| Exception | | Unsuccessful access (if function name does not end with 'silently') | | | | |
| Example | | registry read value ( HKEY\_CURRENT\_USER, Control Panel\International, sCountry, country name[ ] ) | | | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Function** | | **registry write value registry write value silently** | | | | |
| Description | | Writes a dedicated Windows registry entry  The ending "silently" will not issue an exception if the write access fails. | | | | |
| Call as | | **X** | Procedure call | | **X** | Function call, providing a return value |
| Parameter count | | 5 | | | | |
| Parameters | 1, … | Literal | | HKEY  Takes one of the following values: HKEY\_CLASSES\_ROOT, HKEY\_LOCAL\_MACHINE, HKEY\_CURENT\_USER, HKEY\_USERS, HKEY\_CURRENT\_CONFIG | | |
|  | 2. | Literal | | Registry Folder Path (keys) | | |
|  | 3. | Literal | | Registry Entry Name | | |
|  | 4. | Depends on data type | | Value  See next table below on the data type used. It depends on the data type of the value stored in the registry. | | |
|  | 5. | Literal | | Registry Data Type  See next table. Example: REG\_BINARY, REG\_DWORD, etc. | | |
| Return value | | Literal | | Short error messages as described in the introductory part of this section | | |
| Exception | | Unsuccessful access (if function name does not end with 'silently') | | | | |
| Example | | registry write value ( HKEY\_CURRENT\_USER, Control Panel\International, sCountry, country name[ ], REG\_SZ ) | | | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Function** | | **registry delete value registry delete value silently** | | | | |
| Description | | Deletes a dedicated Windows registry entry  The ending "silently" will not issue an exception if the delete access fails. | | | | |
| Call as | | **X** | Procedure call | | **X** | Function call, providing a return value |
| Parameter count | | 3 | | | | |
| Parameters | 1 | Literal | | HKEY  Takes one of the following values: HKEY\_CLASSES\_ROOT, HKEY\_LOCAL\_MACHINE, HKEY\_CURENT\_USER, HKEY\_USERS, HKEY\_CURRENT\_CONFIG | | |
|  | 2. | Literal | | Registry Folder Path (keys) | | |
|  | 3. | Literal | | Registry Entry Name | | |
| Return value | | Literal | | Short error messages as described in the introductory part of this section | | |
| Exception | | Unsuccessful access (if function name does not end with 'silently') | | | | |
| Example | | registry delete value ( HKEY\_CURRENT\_USER, Beyond4P Demo, My Entry) | | | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Function** | | **registry list names registry list keys registry list names silently registry list keys silently** | | | | |
| Description | | Lists the names of all entries resp. keys (subfolders) in a specific registry location | | | | |
| Call as | | **X** | Procedure call | | **X** | Function call, providing a return value |
| Parameter count | | 3 | | | | |
| Parameters | 1 | Literal | | HKEY  Takes one of the following values: HKEY\_CLASSES\_ROOT, HKEY\_LOCAL\_MACHINE, HKEY\_CURENT\_USER, HKEY\_USERS, HKEY\_CURRENT\_CONFIG | | |
|  | 2. | Literal | | Registry Folder Path | | |
|  | 3. Out | Parameter set containing literals | | **registry list names** … (…): List of all entry names **registry list keys** … (…): List of all keys (subfolders) | | |
| Return value | | Literal | | Short error messages as described in the introductory part of this section | | |
| Exception | | Unsuccessful access (if function name does not end with 'silently') | | | | |
| Example | | **registry list names** ( HKEY\_CURRENT\_USER, Control Panel\International, entry Names[ ] )  **registry list keys** ( HKEY\_CURRENT\_USER, Control Panel, folder names[] ); | | | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Function** | | **registry value existing registry key existing** | | | | |
| Description | | Checks if the specified data entry name or registry key (folder name) is existing | | | | |
| Call as | | **X** | Procedure call | | **X** | Function call, providing a return value |
| Parameter count | | 2, 3 | | | | |
| Parameters | 1. | Literal | | HKEY  Takes one of the following values: HKEY\_CLASSES\_ROOT, HKEY\_LOCAL\_MACHINE, HKEY\_CURENT\_USER, HKEY\_USERS, HKEY\_CURRENT\_CONFIG | | |
|  | 2. | Literal | | Registry Folder Path (keys)  Specify the registry folder path name but excluding the final key to delete | | |
|  | 3. Opt. | Literal | | Key or entry name  required for **registry value existing**. | | |
| Return value | | Literal | | true if existing, otherwise false | | |
| Exception | | Only if wrong HKEY is specified.  If access privileges are not sufficient, Boolean false will be returned | | | | |
| Example | | Following three function calls return **true**:  **registry value existing**( HKEY\_CURRENT\_USER, control panel\international, sCountry)  **registry key existing**( HKEY\_CURRENT\_USER, control panel\international ) | | | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Function** | | **registry read accessible registry write accessible** | | | | |
| Description | | Checks if access rights are given for the specified registry folder path (key). | | | | |
| Call as | | **X** | Procedure call | | **X** | Function call, providing a return value |
| Parameter count | | 2 | | | | |
| Parameters | 1. | Literal | | HKEY  Takes one of the following values: HKEY\_CLASSES\_ROOT, HKEY\_LOCAL\_MACHINE, HKEY\_CURENT\_USER, HKEY\_USERS, HKEY\_CURRENT\_CONFIG | | |
|  | 2. | Literal | | Registry Folder Path (keys)  Specify the registry folder path name but excluding the final key to delete | | |
| Return value | | Literal | | true if accessible, otherwise false | | |
| Exception | | Only if wrong HKEY or a non-existing registry key is specified.  If access privileges are not sufficient, Boolean false will be returned | | | | |
| Example | | Following three function calls return **true**:  **registry name existing**( HKEY\_CURRENT\_USER, control panel\international, sCountry)  **registry key existing**( HKEY\_CURRENT\_USER, control panel, international )  **registry key existing**( HKEY\_CURRENT\_USER, control panel\international ) | | | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Function** | | **registry create key registry create key silently** | | | | |
| Description | | Creates a key (registry folder).  Multiple nested keys (folders) can be created with one function call. | | | | |
| Call as | | **X** | Procedure call | | **X** | Function call, providing a return value |
| Parameter count | | 2 | | | | |
| Parameters | 1. | Literal | | HKEY  Takes one of the following values: HKEY\_CLASSES\_ROOT, HKEY\_LOCAL\_MACHINE, HKEY\_CURENT\_USER, HKEY\_USERS, HKEY\_CURRENT\_CONFIG | | |
|  | 2. | Literal | | Registry Folder Path (keys)  It must be the complete path, e.g. "Software\Beyond4P\Demo" where only the missing keys (folders) are created. | | |
| Return value | | Literal | | Short error messages as described in the introductory part of this section | | |
| Exception | | Unsuccessful access (if function name does not end with 'silently') | | | | |
| Example | | registry delete value ( HKEY\_CURRENT\_USER, Beyond4P Demo, My Entry) | | | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Function** | | **registry delete key registry delete tree registry delete key silently registry delete tree silently** | | | | |
| Description | | Deletes a registry key (folder).  With **registry delete key** … (…), the key must not contain any further sub-keys. If they do, then they need to be deleted first.  The **registry delete tree** … (…) is more aggressive: It will eliminate all values and subkeys recursively before concluding with deleting the specified key. | | | | |
| Call as | | **X** | Procedure call | | **X** | Function call, providing a return value |
| Parameter count | | 3 | | | | |
| Parameters | 1. | Literal | | HKEY  Takes one of the following values: HKEY\_CLASSES\_ROOT, HKEY\_LOCAL\_MACHINE, HKEY\_CURENT\_USER, HKEY\_USERS, HKEY\_CURRENT\_CONFIG | | |
|  | 2. | Literal | | Registry Folder Path (keys)  Specify the registry folder path name but excluding the final key to delete | | |
|  | 3. | Literal | | Key to delete | | |
| Return value | | Literal | | Short error messages as described in the introductory part of this section | | |
| Exception | | Unsuccessful access (if function name does not end with 'silently') | | | | |
| Example | | registry delete value ( HKEY\_CURRENT\_USER, Beyond4P Demo, My Entry) | | | | |

# Supplementary Function and Procedure Libraries

Beyond4P now offers additional functions which provide powerful features in table processing and formatting. These functions are not implemented in the Beyond4P machine but are actually written in the language Beyond4P and need to be included using the **include** (…) function.

## Style and Formatting Functions

### Introduction

Tables in Beyond4P may be saved in CSV, HTML or EXCEL (actually XML) data formats. All three formats are readable by Microsoft Excel. The HTML format is also suitable for web browsers. The EXCEL data format is in fact an XML format specifically designed for database systems creating formatted output which can be viewed in Excel.

The typical approach to format a table is:

1. **Finalize the table with all necessary contents**. You may want to have some helper columns on the right hand side which you can delete after adding formatting features, but the formatted table must not move in any direction.
2. If needed: **Define special colors** so you can reference them conveniently by color name.  
   Available functions:  
    **add color** (…) Add user-defined colors  
    **lighten colors** (…) Lighten colors  
    **darken colors** (…) Darken colors  
    **weaken colors** (…) Lower color intensities of selected colors
3. **Add style and formatting** with the functions listed below:  
    **table style table** (…) to format the entire sheet, table or table body  
    **table style rows** (…) to format table rows  
    **table style columns** (…) to format table columns  
    **table style cells** (…) to format individual cells or areas  
   The actual table contents remain unaffected. No special markup symbols
4. **If needed, remove redundant contents** in rows at the bottom and columns at the right.  
   Attention: The table you want to keep must not shift. The formats you have defined keep their positions where they are. For example, If you delete (or insert) a row, the formats in the row below will not shift up (or down) accordingly. Same applies to columns, sorting, rearranging, etc.
5. **Generate the final formatting automatically** for the intended type (HTML or EXCEL) by calling  
   one of the two functions below:  
    **translate style attributes for excel** (…)  
    **translate style attributes for html** (…)
6. **And finally, save your work** using one of the functions below:  
    **table save** (…),  
    **table save multiple** (…),  
   with the formats **EXCEL** or **HTML**. Other formats, such as the default CSV, will ignore the formatting. These two functions will actually spot the formatting data structures generated in a global variable and use the contents while writing contents to the files.
7. **To reset all format and styles in your table,** call the following function  
    **table style reset** (…)  
   so all formatting have been removed and you can start from scratch again.

The following function library provides generic functions which allows you to apply specific styles and formatting on tables, rows, columns and individual cells. During the formatting process, a global variable with underlying structures will be built up which needs to be converted and is then spotted by the functions table save (…) and table save multiple (…).

Important to watch out for:

* You must call the file "Style Library.txt" as follows  
  **include** ("Style Library.txt"). If not located in the same directory, then include the path in front of the file name.
* The library generates and works with the following global variables listed below. UNDER ANY CIRCUMSTANES, DO NOT USE OR MANIPULATE THE CONTENTS OF THESE VARIABLES INCLUDING THEIR MEMBERS. DOING SO WILL RESULT TO UNSTABLE AND MALFUNCTIONING BEHAVIOR.  
   **table style descriptions** [ ] Contains all formatting relevant information for all tables formatted  
   **hex** [ ] Conversion to hexadecimal values  
   **table style rules** [ ] Defines rules to properly interpret formatting codes  
   **table style html comments**[ ] Template to put comments into HTML file format
* The table **global color table** is created. WE STRONGLY ADVISE NOT TO MODIFY THE CONTENTS IN THIS FILE DIRECTLY, BUT FEEL FREE READING THE TABLE. Use the documented functions to add or modify colors. Feel free to add further columns to create your own categorizations.  
    
  The table contains following columns:  
   **Color Name** Obvious, like 'green', 'turquoise', etc.  
   **Cat 1**  First category (standard, lightened, darkened, weakened, user defined)  
   **Cat 2** 2nd category (primary, saturated, gray scales, excel color, red tones,   
   yellow tones, green tones, blue tones, purple tones, pink tones, brown tones)  
   **Red** Red color channel intensity, 0 … 255  
   **Green** Green color channel intensity, 0 … 255  
   **Blue** Blue color channel intensity, 0 … 255  
   **HTML Color** HTML RGB color coding, e.g. #FFC000 for orange

### Defining User specific Colors

Colors can either be specified by color name, the RGB value in a parameter set (e.g. { 255, 192, 0 } for orange), or the RGB value in HTML format (#FFC000 for orange).

Beyond4P provides more than 150 different standard colors, including the six primary colors (red, yellow, green, cyan, blue, magenta), gray scales (from black to white), fully saturated colors along the color circle (a total of 32 including the six primary colors), excel 2000+ colors (excel light green, excel green, excel light blue, excel blue, excel dark blue, excel violet; the other ones are primary colors) and a nice fashion and interior designer's collection of different shades of red, yellow, green, blue, purple, pink and brown colors.

Functions are available to extend the named colors by applying functions to derive lightened, darkened and weakened (reduction of color saturation) varieties. The function **add color**(…) allows you to add colors.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Function** | | **add color** | | | | |
| Library File | | Style Library.b4p Use **include** ( Style Library ) to include the library | | | | |
| Description | | Adds a further color to the color table | | | | |
| Call as | | **X** | Procedure call | |  | Function call, returning a return value |
| Parameter count | | 5 | | | | |
| Parameters | 1. | Literal | | Color name | | |
|  | 2. | Literal | | Category 2 (written to column Cat 2) Note: Category 1 will be set to "user defined" | | |
|  | 3. | Numeral | | Red color intensity, 0..255 (values beyond will be corrected) | | |
|  | 4 | Numeral | | Green color intensity, see above | | |
|  | 5. | Numeral | | Blue color intensity, see above | | |
| Return value | | n/a | | Procedure call | | |
| Example | | **add color**( corporate red, logo, 213, 0, 50 );  **add color**( taxi beige, surface color, 229, 219, 182 ); | | | | |

Next page: Overview of standard color names

 

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Function** | | **lighten colors** | | | | |
| Library File | | Style Library.b4p Use **include** ( Style Library ) to include the library | | | | |
| Description | | Derives lightened shades of selected colors | | | | |
| Call as | | **X** | Procedure call | |  | Function call, returning a return value |
| Parameter count | | 5 | | | | |
| Parameters | 1. | Numerals (Parameter set of numerals) | | Row numbers  The row numbers must correspond to the rows in the **global color table**. | | |
|  | 2. | Literal or numeral | | Suffix (can be a number or literal, will always be converted to literal) | | |
|  | 3. | Numeral | | Lightening up factor. e.g. factor 2 is the color precisely between white and specified color E.g. mandarin (255,128,0) translates to (255,192,128)  Use lower factors (e.g. 1.5) for less lighter shades, and bigger factors (e.g. 4) for significantly lighter shades. | | |
| Return value | | n/a | | Procedure call | | |
| Example | | **lighten colors**( **table selected rows**( global color table, [Cat 2]=primary), 1, 2 ); Generates lighter color variants of the six primary colors: red 🡪 light red 1 yellow 🡪 light yellow 1 etc. | | | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Function** | | **darken colors** | | | | |
| Library File | | Style Library.b4p Use **include** ( Style Library ) to include the library | | | | |
| Description | | Derives darkened shades of selected colors | | | | |
| Call as | | **X** | Procedure call | |  | Function call, returning a return value |
| Parameter count | | 5 | | | | |
| Parameters | 1. | Numerals (Parameter set of numerals) | | Row numbers  The row numbers must correspond to the rows in the **global color table**. | | |
|  | 2. | Literal or numeral | | Suffix (can be a number or literal, will always be converted to literal) | | |
|  | 3. | Numeral | | Darkening down factor. e.g. factor 2 is the color precisely between black and specified color E.g. mandarin (255,128,0) translates to (128,65,0)  Use lower factors (e.g. 1.5) for lesser darkening, and bigger factors (e.g. 4) for significantly darker shades. | | |
| Return value | | n/a | | Procedure call | | |
| Example | | **darken colors**( **table selected rows**( global color table, [Cat 2]=primary), 1, 2 ); Generates darker color variants of the six primary colors: red 🡪 dark red 1 yellow 🡪 dark yellow 1, etc. | | | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Function** | | **weaken colors** | | | | |
| Library File | | Style Library.b4p Use **include** ( Style Library ) to include the library | | | | |
| Description | | Derives color shades of weakened intensity | | | | |
| Call as | | **X** | Procedure call | |  | Function call, returning a return value |
| Parameter count | | 5 | | | | |
| Parameters | 1. | Numerals (Parameter set of numerals) | | Row numbers  The row numbers must correspond to the rows in the **global color table**. | | |
|  | 2. | Literal or numeral | | Suffix (can be a number or literal, will always be converted to literal) | | |
|  | 3. | Numeral | | Weakening factor. e.g. factor 2 is the color precisely between gray (gray level depends on color tone, e.g. light gray when weakening yellow). The corresponding luminance of gray is (0.299\*Red+0.587\*Green+0.114\*Blue) as applied in color television.  Use lower factors (e.g. 1.5) little weakening, and bigger factors (e.g. 4) for significantly weaker intensities. | | |
| Return value | | n/a | | Procedure call | | |
| Example | | **weaken colors**( **table selected rows**( global color table, [Cat 2]=primary), 1, 2 ); Generates weaker color variants of the six primary colors: red 🡪 weak red 1 yellow 🡪 weak yellow 1, etc. | | | | |

### Add Style and Formatting

After you have defined the colors you need, start with style and formatting work. The following four functions are available:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Function** | | **table style table** | | | | |
| Library File | | Style Library.b4p Use **include** ( Style Library ) to include the library | | | | |
| Description | | Applies styles and formatting on the whole table | | | | |
| Call as | | **X** | Procedure call | |  | Function call, returning a return value |
| Parameter count | | 4, 6, 8, etc. | | | | |
| Parameters | 1. | Literal | | Table name | | |
|  | 2. | Literal | | Coverage, takes one of following:  sheet Applies formatting to entire sheet, even beyond the table.  (When saving as HTML, rows below the table remains  unaffected)  table Applies formatting to entire table, but not beyond last row  and last column  body Like 'table', but except the top row. | | |
|  | 3, 5, … | Literal | | Generic Attribute Name (described later) | | |
|  | 4, 6, … | Valid parameter | | Generic Attribute Value (described later) | | |
| Return value | | n/a | | Procedure call | | |
| Example | | table style table( cities, sheet, fill color, yellow):    If saved in HTML, then vacant rows below end of table will not be colored yellow.  table style table( cities, table, fill color, lemon):    table style table( cities, body, fill color, green): | | | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Function** | | **table style rows** | | | | |
| Library File | | Style Library.b4p Use **include** ( Style Library ) to include the library | | | | |
| Description | | Applies styles and formatting on selected rows | | | | |
| Call as | | **X** | Procedure call | |  | Function call, returning a return value |
| Parameter count | | 5, 7, 9, etc. | | | | |
| Parameters | 1. | Literal | | Table name | | |
|  | 2. | Numerals (parameter set of numerals) | | Row numbers | | |
|  | 3. | Literal | | Coverage, takes one of following:  sheet Applies formatting to entire sheet, even beyond the table.  table Applies formatting to entire table. Width determined by  width of header row  body Like 'table', width in actual table row (which may deviate  from number of columns in the header row | | |
|  | 4, 6, … | Literal | | Generic Attribute Name (described later) | | |
|  | 5, 7, … | Valid parameter | | Generic Attribute Value (described later) | | |
| Return value | | n/a | | Procedure call | | |
| Example | | table style rows( cities, 2, sheet, fill color, yellow ); // Entire Excel row is yellow  table style rows( cities, 3, table, fill color, lemon );  table style rows( cities, 4, body, fill color, green ); // Given: row 4 has fewer columns inside | | | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Function** | | **table style columns** | | | | |
| Library File | | Style Library.b4p Use **include** ( Style Library ) to include the library | | | | |
| Description | | Applies styles and formatting on selected rows | | | | |
| Call as | | **X** | Procedure call | |  | Function call, returning a return value |
| Parameter count | | 5, 7, 9, etc. | | | | |
| Parameters | 1. | Literal | | Table name | | |
|  | 2. | *Table columns specification*  (See section 9.3) | | Select one or multiple columns by column numbers or column header names. | | |
|  | 3. | Literal | | Coverage, takes one of following:  sheet Applies formatting to entire sheet, even beyond the table.  (When saving as HTML, rows below the table remains  unaffected)  table Applies formatting to entire table. Width determed by  width of header row  body Like 'table', but except header row | | |
|  | 4, 6, … | Literal | | Generic Attribute Name (described later) | | |
|  | 5, 7, … | Valid parameter | | Generic Attribute Value (described later) | | |
| Return value | | n/a | | Procedure call | | |
| Example | | table style columns( cities, 1, sheet, fill color, yellow );  table style columns( cities, 2, table, fill color, lemon );  table style columns( cities, Dining, body, fill color, green );    If saved in HTML, then vacant rows below end of table will not be colored yellow. | | | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Function** | | **table style cells** | | | | |
| Library File | | Style Library.b4p Use **include** ( Style Library ) to include the library | | | | |
| Description | | Applies styles and formatting on selected cells | | | | |
| Call as | | **X** | Procedure call | |  | Function call, returning a return value |
| Parameter count | | 6, 8, 10, etc. | | | | |
| Parameters | 1. | Literal | | Table name | | |
|  | 2. | *Table columns specification*  (See section 9.3) | | Select one or multiple columns by column numbers or column header names. | | |
|  | 3. | Numerals (parameter set of numerals) | | Row numbers | | |
|  | 4. | Literal | | Coverage, takes one of following:  single Single cell addressed. If parameter set provided, then both  parameter set must contain same number of elements. Each  element pair corresponds to coordinates for one cell matrix Intersections of all specified rows and columns will be  formatted field Pairs of 2 parameter set elements represent a field  to format. | | |
|  | 5, 7, … | Literal | | Generic Attribute Name (described later) | | |
|  | 6, 8, … | Valid parameter | | Generic Attribute Value (described later) | | |
| Return value | | n/a | | Procedure call | | |
| Example | | table style cells( cities, Daytime 1, 1, single, fill color, yellow ); table style cells( cities, {0,1}, {3,5}, single, fill color, lemon ); table style cells( cities, {3,5}, {3,5}, matrix, fill color, green ); table style cells( cities, {3,5}, {0,1}, field, fill color, cyan ); | | | | |

Available formatting options:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Generic Attribute Name** | **Generic Attribute Value** | **Scope** | **Description** | **Restrictions, compatibility** |
| **text color** | color specification  (see separate table below) | table, row, column, cell | Text color | - |
| **fill color** | color specification | " | Background color of the cell | - |
| **top border bottom border left border right border** | border specification | " | Applies colored border lines above / below / left to / right to the cell | HTML: Thickness 'hairline' is same as 'thin' |
| **underscore boldface italic** | Boolean true / false | " | Applies text formatting | - |
| **wrap text** | Boolean true / false | " | Wrap text forces text to continue on the next line inside the same cell if the entire line does not fit inside. | - |
| **comment** | Literal | cell | Adds comments to specified cells. In Excel, cells with comments are tagged with red triangles at the top right corner. Hold the mouse above to see the comments. | Excel: Will only function properly in 1st table in case multiple tables are saved in one HTML file. |
| **font name** | Literal | table, row, column, cell | Sets the font name |  |
| **font size** | Numeral | " | Sets font size in units of typographical points |  |
| **number format** | Excel number format | " | Applies a number format according to Excel specification.  E.g. #,###0.00, DD-MMM-YYY, etc. | Not compatible to Beyond4P's number format as used by the function **literal** (…) |
| **type** | default, literal, numeral, boolean | " | default No special rules  literal All contents are literal, including numbers  numeral Shown as numerals  boolean Shows 0 and 1 as TRUE and  FALSE in Excel  Attention: Forcing boolean and numerals while literals are inside may be rejected by Excel. Loading the file would fail. | Functionality depends on output format EXCEL or HTML. |
| **horizontal align** | automatic left middle right | " | Alignment of contents to desired horizontal direction |  |
| **vertical align** | automatic top center bottom | " | Alignment of contents to desired vertical direction |  |
| **column width** | 1 … 255  (Numeral) | column | Sets the column width. Values are compatible with Excel column width units |  |
| **row height** | 1 … 255  (Numeral) | row | Sets the row height. Values are compatible with Excel row height units. | Not 100% accurate with HTML exports (+/- 1% deviation) |
| **autofilter** | Row number  (Numeral) | table | Applies the Excel autofilter feature on a specific row. Only 1 row can be specified.  The header row is row 0.  Attention: No autofilter settings are applied in order to prefilter selected rows. |  |
| **freeze rows** | Row number | table  Use coverage: 'sheet' | Freeze pane at row number (0 = No freeze)  E.g. specify 1 to freeze header row | Works with EXCEL only. HTML format ignores this. |
| **freeze columns** | Column number | table  Use coverage: 'sheet' | Freeze pane at column number (0 = No freeze)  E.g. specify 2 to freeze first two columns | Works with EXCEL only. HTML format ignores this. |
| **hidden**  Also allowed: **hidden row hidden column** | Boolean true / false | row, column  row column | Hides selected rows and columns  Attention: Combination with some other styles may prevent this function. | In HTML, column width is set to 0 |

Color Specification

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method** | **Type** | **Example** | **Description** | **Restrictions, compatibility** |
| By color name | Literal | red | Existing name found in the table called '**global color table**', column 'Color Name'  Any standard, customized or custom added color name added to that table is valid. | - |
| HTML color code | Literal | #FF0000  (for red) | Color values for red, green and blue are coded with 2 hexadecimal digits each. Range: 00 … FF |  |
| RGB color values | parameter set containing 3 numerals | { 255, 0, 0 }  (for red) | Color values for red, green and blue are coded from 0 to 255. |  |

Border Specification

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method** | **Type** | **Example** | **Description** | **Restrictions, compatibility** |
| Direct (Thickness name and color specification) | Parameter set containing 1 literal and 1 color specification | { thin, red }  (for thin line in red color) | Thickness names may be one of:  hairline, thin, medium, thick | - |

Note that HTML as well as XML for Excel have different characteristics and functional scopes and limitations which are listed below:

|  |  |  |
| --- | --- | --- |
| **Category** | **Excel** | **HTML** |
| Default character set | Arial, 10 pt  Font name may change when modifying text color. To prevent this, I suggest to assign a default font name for the entire table. | Calibri, 11 pt. |
| Spreadsheet grid visible | Yes  To make invisible, apply following function: **table style table** ( table name, sheet, fill color, white ); | No  If required, imitate them with drawing borders. |
| Column width | Standard 10.71 | Automatic (to fit contents) |
| Dates | No automatic date recognition.  You need to put numbers into the fields (e.g. with function **abs** ( *date value* ) and then apply a number format with values like "DD.MM.YYYY". | Excel recognizes dates in an intelligent manner. Dates in local format or specified as YYYY-MM-DD are detected automatically and loaded as dates. |
| Horizontal align | Default: left | Default: left, except header row: middle |
| Vertical align | Default: bottom | Default: center |
| Header row: | Standard text | Boldface text |
| Wrap text | Default: false | Default: true |
| Text colors | Some text colors appear slightly modified, e.g. crimson is displayed as plain red. | All colors appear to be shown as specified |
| Formulas | Not supported | Supported, but language specific. Formulate as you work in Excel. All formulas must begin with an equal sign.  Function names must be specified in the local language set. E.g. if Excel is configured for German, use a function name like "Summewenn" instead of "Sumif". |
| Outputting Booleans properly | Not supported | Boolean values in local language (which is not necessarily English), e.g. Wahr and Falsch (not case sensitive). Other languages (e.g. English if German is chosen as Excel language) are shown as text. |
| Forcing data to literal format, including numbers | use attribute name 'type' and value 'text'. e.g. table style table( test, sheet, type, literal); | Same. |

Code example with number formatting:

echo("Test various date, numeric and boolean representations");

format[] = input quick ignore case("Choose H for HTML, X for EXCEL: ", HX );

table initialize( test,

{ { Dates Lit 1,Dates Lit 2,Dates Num,Boolean EN,Boolean DE,Boolean Num,Numbers,Formulas },

{ "2017-01-30","30.01.2017",abs(date("2017-01-30")),TRUE,WAHR,1,123.45,"=Wurzel(4)" },

{ "2017-01-30","30.01.2017",abs(date("2017-01-30")),TRUE,WAHR,1,123.45,"=G2\*2" },

{ "2017-01-30","30.01.2017",abs(date("2017-01-30")),TRUE,WAHR,1,123.45,"=G2\*2" },

{ "2017-01-30","30.01.2017",abs(date("2017-01-30")),TRUE,WAHR,1,123.45,"=Root(4)" } } ) ;

table style columns( test, {0..7}, table, column width, 14 );

table style cells( test, Dates Num, 2, single, number format, "DD.MMM.YYYY" );

table style cells( test, Dates Lit 1, 2, single, number format, "DD.MMM.YYYY" );

table style cells( test, Dates Lit 2, 2, single, number format, "DD.MMM.YYYY" );

table style rows( test, 3, sheet, type, literal );

table style cells( test, Boolean Num, 4, single, type, boolean );

if (format[] == H)

{

translate style attributes for html( test );

table save( test , working directory[] + "Numbers out in html.xls", HTML );

}

else

{

translate style attributes for excel( test );

table save( test, working directory[] + "Numbers out in excel.xls", EXCEL );

}

Following HTML output is generated:



Note: It recognizes Boolean values as well as formula names (Wurzel instead of Root) in current Excel language and not necessarily in English. Note the last row in "Boolean Num" is blank.

Following EXCEL output is generated:



Formulas are not recognized in this format. Implementation is being considered in a future release. Note the last row under "Boolean Num" is the only field containing an authentic Boolean value. Formulas are ignored and treated as plain text instead.

### Translate Generic Type to Specific File Format

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Function** | | **translate style attributes for excel**  **translate style attributes for html** | | | | |
| Library File | | Style Library.b4p Use **include** ( Style Library ) to include the library | | | | |
| Description | | All generic style descriptions will be translated to a desired target format which is then used by the **table save** / **table save multiple** (…) function. | | | | |
| Call as | | **X** | Procedure call | |  | Function call, returning a return value |
| Parameter count | | 1 | | | | |
| Parameters | 1. | Literal | | Table name | | |
| Return value | | n/a | | Procedure call | | |
| Example | | **translate style attributes for html** ( customer list ); | | | | |

### Reset Style and Formatting

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Function** | | **table style reset** | | | | |
| Library File | | Style Library.b4p Use **include** ( Style Library ) to include the library | | | | |
| Description | | Removes any style and formatting on the specified tables.  If no formatting exists, then there will be no error.. | | | | |
| Call as | | **X** | Procedure call | |  | Function call, returning a return value |
| Parameter count | | min 0 | | | | |
| Parameters | 1. | Literal | | Table name | | |
| Return value | | n/a | | Procedure call | | |
| Example | | **table style reset**( customers, orders, products ); | | | | |

# Compile-Time Directives

Compile-Time Directives are special-purpose instructions recognized during initial code loading subsequent compilation into internal data structures. These directives become invisible during run-time before code execution begins. Directives only apply for the program file where specified, but not in other files referenced, e.g. with the **start**(…) function. The intention of these directives is to influence the code compilation process, but not to influence run-time behavior.

Compile-Time Directives consist of the following sequence:

1. **$$** Two subsequent dollar signs, without a space in-between

2. **literal** Name of directive: One literal (without quotation marks)

3. **=** One equal sign

4. **value** Value to assign: Choice of literal (with or without quotation marks),   
 numeral or Boolean, depending on the name of directive

5. **;** Semicolon to delimit the Compile-Time Directive.

Arithmtics, function calls and other algorithmic features, even calculations with constants only (for example $$ abc = 3+5 ; ), are not allowed and will give out error messagees.

Compile-Time Directives inside comments are ignored, e.g. // $$ abc = 123; has no impact.

Compile-Time Directives may be used in any place of the program code, as long they are outside quoted literals and comments. This means that they do also work if inserted somewhere inside code blocks and evens statements.

Presently, following directives are supported:

|  |  |  |
| --- | --- | --- |
| **Directive** | **conventional coding style** | |
| Description | Enables conventional coding style where literals without quotation marks are automatically treated as variables unless they are used as function and procedure names.  Benefit: Code sections using lots of variables, e.g. while doing calculations, will look leaner and cleaner.  Attention: If enabled, make sure you use single or double quotation marks to declare literal values. This also applies to parameters passed into functions as well as table names and table header names.  E.g. a = a + b[ ] + c['a'] + 'd' + e[a]  is automatically converted into a[ ] = a[ ] + b[ ] + c['a'] + 'd' + e[ a[ ] ]  Attention: To avoid confusions with procedure calls (without own parameters) submitted into function calls such as **table process**…(), delimit these statements with semicolon. Otherwise they are interpreted as simple variables. e.g. **table process**( 'table1', **echo;** );  Note the table name has been put inside quotation marks. | |
| Values | false (default) | Not enabled |
| true | Enabled |
| Example | **$$ conventional coding style = true;**  for (a = 1, a < 10, a+=1) { echo; echo( "a is ", a ); echo( "a is ", a[ ] ); }  **$$ conventional coding style = false;** // Reset directive.  This example shows that *a* is treated as a variable and *echo* remains a procedure call. | |

|  |  |  |
| --- | --- | --- |
| **Directive** | **allow entity references** | |
| Description | Allows conversion of HTML like character entity references in **softquoted literals** into their effective characters. Example:  If enabled: a[ ] = '**&euro;** 20.00'; // is "€ 20.00"  a[ ] = "&euro; 20.00"; // is "&euro; 20.00"  a[ ] = '&' + 'euro; 20.00'; // is "&euro; 20.00"  In the 3rd entry, the entity reference is not recognized in a single  string. Use function call **decode entities** (…) when needed during  run time.  If not enabled: No conversion. Entity references remain unaffected.  Result is always "&euro; 20.00". | |
| Values | false | Not enabled |
| true (default) | Enabled |
| Example | echo('10 &euro; ');  **$$ allow entity references = false;**  echo('10 &euro; ');  **$$ conventional coding style = true;** // Reset directive.  echo('10 &euro; ');  Resulting output:  10 € 10 &euro; 10 € | |
| See also | Function **decode entities** | |

# Appendix: Programming Language Symbols

|  |  |  |  |
| --- | --- | --- | --- |
| **Character** | **Usage** | **Context** | **Description** |
| Constants | | |  |
| **"** | " … " | Quoted literals | Contents as enclosed in double quotation marks are (hard-) quoted literals. |
| **'** | ' … ' | Softquoted literals | Contents as enclosed in double quotation marks are softquoted literals. Comma used as separator in some parameters to functions, ? and \* inside used as wildcard symbols for comparisons. |
| **.** | . | Decimal point | Only applicable if correctly used in a numeral, e.g. 1.234 |
| Comments | | |  |
| **//** | // | Line comment | Ignores rest of line. |
| **/\* \*/** | /\* … \*/ | Code comment | Ignores contents between these two symbols. |
| Assignment operator | | |  |
| **=** | *dest* = *expr* | Assignment |  |
| Arithmetic operators and Boolean binary operators | | |  |
| **+** | *expr* + … *expr* | Addition |  |
| **−** | *expr* − *expr* | Subtraction |  |
| **\*** | *expr* \* *expr* | Multiplication |  |
| **/** | *expr* / *expr* | Division |  |
| **&** | *expr* & *expr* | Boolean AND |  |
| **|** | *expr* | *expr* | Boolean OR |  |
| **==** | *expr* == *expr* | Boolean Equivalent | See also comparison operator (if expressions are not Boolean) |
| **!=** | *expr* ! = *expr* | Boolean XOR | See also comparison operator (if expressions are not Boolean) |
| Arithmetic assignment operators | | | Symbols may be written with or without space in-between. |
| **+=** | *dest* + = *expr* | Addition |  |
| **-=** | *dest* − = *expr* | Subtraction |  |
| **\*=** | *dest* \*= *expr* | Multiplication |  |
| **/=** | *expr* / = *expr* | Division |  |
| **&=** | *dest* & = *expr* | Boolean AND |  |
| **|=** | *dest* | = *expr* | Boolean OR |  |
| **== =** | *dest* == = *expr* | Boolean Equivalent |  |
| **!= =** | *dest* != = *expr* | Boolean XOR |  |
| Parentheses in formulas | | |  |
| **(, )** | ( *expr* ) | Expression | Contents inside parentheses are calculated first |
| Unary operators | | |  |
| **+** | + *expr* | Upper case |  |
| **-** | − *expr* | Negation |  |
| **!** | ! *expr* | Boolean NOT |  |
| Comparison operators | | |  |
| < | *expr* < *expr* | Less than |  |
| <= | *expr* <= *expr* | Less than or equal |  |
| > | *expr* > *expr* | Greater than |  |
| >= | *expr* >= *expr* | Greater than or equal |  |
| = | *expr* = *expr* | Equal | Supports selection (a,b,c,…) and ranges on right hand side |
| == | *expr* == *expr* | Equal to | Supports no selection (a,b,c…) and ranges (a..c) |
| <> | *expr* <> *expr* | Not equal to | Supports selection (a,b,c,…) and ranges on right hand side |
| != | *expr* == *expr* | Not equal to | Supports no selection (a,b,c…) and ranges (a..c) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Character** | **Usage** | **Context** | **Description** |
| Comparison suffix operators | | | To be used after = and <> |
| **+** | + *expr* | Ignore case |  |
| **~** | ~ *expr* | Ignore blanks / epsilon tolerance | Comparing texts: Ignore blank characters  Comparing numbers: Tolerate residual error epsilon. |
| Ad hoc operators | | |  |
| **++** | ++ expr, expr++ | Increment | On variables & tables. A delta value in parenthesis is optional |
| **--** | -- expr, expr - | Decrement | On variables & tables. A delta value in parenthesis is optional |
| **\*\*** | \*\*(expr) expr expr \*\*(expr) | Apply scaling factor | On variables & tables. A scaling factor inside parenthesis is required |
| **(, )** | ++/--/\*\* (*expr*) | Value to apply | Value to applied in ad hoc operation, e.g. ++(3) adds 3. |
| Transactions | | |  |
| **<==** | *dest* <== *dest* | Copy transaction |  |
| **<<=** | *dest* <<= *dest* | Move transaction |  |
| **<=>** | *dest* <=> *dest* | Swap transaction |  |
| Transaction prefix | | | Applicable to all 3 transacation operators |
| **&** | *dest* &<== *dest* | Overwrite existing element only |  |
| **|** | *dest* |<== *dest* | Overwrite + new elements |  |
| **+** | *dest* +<== *dest* | Add new elements | Add / append items (no overwriting) |
| Operator suffix | | |  |
| **^** | *operator* ^ | Deep operator suffix | Added behind selected operators to apply the operator not on the parameter set but on their elements (1 level down) |
| **^** | ^ *transaction* | Base variable unchanged | Do the 'deep' part of the transaction |
| **^** | = ^ | Create reference to variable |  |
| **^^** | = ^ ^ | Create reference to referece |  |
| Wildcard symbols | | | Inside literals and softquoted literals, used in comparisons and selecting table rows and columns |
| **\*** | \* | Characters of choice | Stands for 0, 1, or more characters of choice between |
| **?** | ? | 1 character | Stands for exactly 1 character of choice |
| **,** | , | OR Separator | Choice of multiple texts to compare. OK if one is matching |
| Selections in comparisons | | | Combined use of commas and .. is allowed |
| **,** | *= expr , expr <> expr, expr* | Comma separator | Multiple Selections |
| **..** | *= expr* .. *expr <> expr .. expr* | From .. to range | Alphabetic and numeric range |
| Selections in picking details | | | Combined use of commas and .. is allowed |
| , | *= expr , expr* | Comma separator | Multiple Selections |
| .. | *= expr* .. *expr* | From .. to range | Numeric range |
| Selections in table specifications | | | Combined use of commas and .. is *not* allowed |
| **..** | *expr* .. *expr* | From A to B |  |
| **..** | *expr ..* | From A to end |  |
| **..** | *.. expr* | From begin to A |  |
| **..** | *..* | From begin to end |  |
| **:** | : *expr* | Match all | Following expression may use wildcard symbols |
| Parameter sets | | |  |
| **{ , }** | { *expr* , *expr* } { } | Parameter sets | Can contain 0 or 1 or multiple elements, comma separated |
| Parameters in function and procedure calls | | |  |
| **( , )** | *expr* ( *expr, expr* ) | Expression | Contains 1 or multiple parameters, comma separated |

|  |  |  |  |
| --- | --- | --- | --- |
| Code blocks | | |  |
| **{ ; }** | { *stmt* ; *stmt* } { } | Code block | Contains 0 or 1 or multiple code statements |
| **;** | *statement* ; … | Statement separator | All statements shall end with semicolon (Exception: Last statement when passed as parameter to function or procedure) |
| Picking details | | |  |
| **{ , }** | *expr* { *expr*, *expr* } *expr* { } | Pick details | Contains 0 or 1 or multiple expression providing numeric index to select parameter set elements or characters from literals |
| Accessing variables | | |  |
| **[ , ]** | *expr* [ ] expr [ expr, expr ] | Access base variable Access var. member | Contains 0 or 1 or multiple expressions describing a variable access. Expressions inside selected intended member. |
| Accessing tables | | |  |
| **[ : , ]** | [ *expr* : *expr,expr* ] | Full table specification | Must contain table name before colon |
| **[ , ]** | [ *expr* ] [ *expr*, *expr* ] | Partial table specification | Table name not required |
| **[ ^ , }** | [ ^ *expr* ] etc. | Partial table specification "Next level up" | Use 1 or more accent circumflex symbols to select the next outer table context information in order access their table contents easily. |
| Miscellany | | |  |
| **:** | else : *statement* | Literal separator | If single statement follows the 'else', then a colon is required to separate. |
| **$$** | $$ literal = value ; | Compile-Time directive | Processed during compile time. Directives will be invisible during run time. |

# Language Syntax Summary

Style: Only the symbols in bold typestyle are actual programming language symbols.

| **Building Block** | **Summary** | **Description** |
| --- | --- | --- |
| Program | *Choice or combination of 1 or more:*  block  statement **;** … | The entire program may consist of one or more blocks and/or one or more statements. |
| Block | *Choice or combination of 1 or more:* **{**  statement **;**  block …  **}** | The program starts with an open brace symbol **{**, contains at least one statement or nested block and is finalized with an close brace symbol **}**.  The initial brackets are not required |
| Statement | *choice of:*  assignment  procedure call  transaction | Statements are the key instructions interpreted and carried out during run time. Statements consist of assignments and procedure calls. |
| Assignment | *Sequence of:*  destination expression  assignment operator  expression; | In assignments, values obtained / calculated in the expression are assigned to LH (left-hand) expressions. |
| Transaction | *Sequence of:*  destination expression  transaction operator  destination expression | Transactions allow for smart data movement inside and among table rows and structured variables. |
| Transaction operator | *Sequence of:*  *optional* transaction prefix  transaction operator |  |
| Transaction prefix | *Choice of:*  **+**, **&**, **|** |  |
| Transaction operator | *Choice of:*  **<==**, **<<=**, **<=>** |  |
| Destination expression | *Choice of:*  variable reference  table reference | Destination expressions do either reference variables (main variables or one of its members) or tables. |
| Variable reference | *Choice of:*  variable name **[** **]**  variable name **[** member spec  ( … **,** member spec) **]** | Variable name must be an expression providing a literal.  1 or more member specs separated with commas can be used to specify members, sub-members, etc.  member spec is either a literal expression (refer member variable by name) or numeric expression (members are automatically kept in alphabetic sequence, 0 points to first member in alphabetic order) |
| Table reference | *Choice of:*  full table specification*,*  partial table specification | Full table specification always includes table name and a reference to a row number. For partial table specifier, implicit knowledge of table name and current row (typically an iterator) must be known. Applicable in combination with specific functions, e.g. **with table** (…), **for all table rows** (…), **table process** (…). |
| Full table specification | *Choice of:*  **[** table name **:** column spec **,**  row spec **]**,  **[** table name **:** column spec **,**  row spec **,**  column spec **]** | Table name: Expression returning a literal  Column spec: Expression returning a literal (column name) or numeral (column number).  Row spec: Expression returning a literal (matching row content) or numeral (row number). |
| Partial table specification | *Choice of:*  **[** column spec **]**,  **[** column spec, row spec **]**  **[** column spec **,** row spec **,**  column spec **]** | Table name and row spec are not mandatory.  Column spec and Row spec: See above. |
| Assignment Operator | *Choice of:*  **=**  binary operator **=** | Simple assignment: Use equal sign. Also allowed in combination with certain allowed arithmetic and Boolean operators, e.g. **+=**. as well as deep operators, e.g. **+^=** |
| Binary operator | *Choice of:*  **+**, **−**, **\***, **/**, **&**, **|**, **==**, **!=**,  **=**, **<>**, **>**, **>=**, **<**, **<=**  *Optional: Operator suffix 1 or more occurrence of*  **^** | Arithmetic operators Boolean operators Simple comparative operators Comparative operators (Not as part of assignment operator) Relational operators (Not as part of assignment operator)  Circumflex symbol is a suffix which indicates deep operator usage, applicable on calculations with parameter sets. |
| Expression | *Combination of:*  constants  variable references  table references  formulas  function calls  parameter sets  *Optionally followed by 1 or more:*  parameter elements | Expression can be any constant value (e.g. 123, Hello, "Price [EUR]"), variables (e.g. variable name[ ]), tables (e.g.[table1:…]), formulae (combination of values and references with unary and/or binary operators) and function calls.  Parameter elements may be specified if the expression represents or returns a parameter set. Otherwise, error will be issued. |
| Function calls | *Sequence of:*  function name  **(** parameter**,**  parameter**,** … **)**  **or**  function name **(** **)** | Function name is a literal. Not allowed: Any other form of expression, softquoted and quoted literals.  A function with provided parameters is called and returns a value. Selected functions may be called as functions and procedure calls (return value ignored in this case). |
| Procedure calls | *Sequence of:*  procedure name  **(** parameter**,**  parameter**,** … **)**  *or:* procedure name **;** | Function name is a literal. Not allowed: Any other form of expression, softquoted and quoted literals.  Same syntax as function calls, but in procedure calls possible return values will be discarded.  No parentheses required here if no parameters are passed.  Some procedures provide additional control flow structures which are described in the function library. Examples: **if**(…), **while**(…). |
| Parameter (in function and procedure calls) | *Choice of:*  expression  destination expression  code piece | 🡪 Applicable for most parameters (= input parameters)  🡪 Where data is written back to variable during function call  e.g. in **exchange**(…)  🡪 Code piece, e.g. in **for** (…), **table delete selected rows** (…)  Not to confuse with 'parameter set'. |
| Code piece | *Choice of:*  expression  comparison expression  destination expression  statement ( **;** statement )  **:** literal expression ) | Hint: If the expression contains a comparison with **=** or **<>** (but not any of the other comparison operators), then apply parentheses around them if the expression passed as parameter into a procedure or function call and is not the last parameter. Otherwise, the succeeding comma may be interpreted as a selection separator symbol (🡪 see 'Selection').  Instead of code pieces, expressions returning literals can be supplied, but must precede with a colon symbol. |
| Constants | *Choice of:* numerals  literals  softquoted literals  quoted literals  **true**, **false**, **tab**, **new line** | Numerals are positive and negative numbers which must start with a minus sign, then a digit and may contain a decimal point with digits behind. Spaces, thousand-separators and scientific notations are not supported.  Softquoted literals: 'literal' Quoted literals: **"**literal**"**  **true, false**: Boolean values **tab**, **new line**; Literal values |
| Formulas | Combination of expressions with unary operators and binary operators as well as parentheses.  Expressions combined with preceding symbols **=**, **<>**, **<**, **<=**, **=>** and **<** are comparison expressions (including the symbol), but not applicable to **==** and **!=**. | Pay attention to precedence rules.  Parentheses |
| Comparison expression | *Choice of:*  expression  comparison spec, expression  comparison spec, range  comparison spec, selection … | Ranges and selections are only applicable in combination with = and <> or = assumed implicitly in the absence of an equal sign. |
| Comparison spec(ification) | Sequence of:  comparison operator  comparison modifier |  |
| Comparison operator | *Choice of:* **<**, **<=**, **=**, **=>**, **>**, **<>** |  |
| Comparison modifier | *Choice or combination of:* **~**, **+** | **~** = Literal: Ignore case, Numeric: epsilon error tolerance **+** = Literal: Ignore blanks |
| Range | expression **..** expression |  |
| Selection | expression ( **,** expression … ) |  |
| Parameter set | Sequence of: **{**  expression,  expression,  …  expression **}** | Contains zero or more expressions (so called elements) inside braces.  (Not to confuse with 'parameter') |
| Parameter element | Sequence of: **{**  numeric expression, **}**  *or* **{ }** | Parameter elements may succeed expressions which represent or return  parameter sets.  Empty set will return the number of elements. |

# Reserved System Variables

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable Name** | **Type** | **Description** | **Restrictions** |
| time stamp [ ] | quoted literal | Time stamp taken when program has started | Read-only |
| version info [ ] | quoted literal | Version info headline | Read only |
| local settings [ ]  … [ country name ]  … [ language ] \*  … [ decimal separator ]  … [ thousand separator ]  … [ list separator ]  … [ date format ]  … [ date separator ]  … [ time format ]  … [ time separator ]  … [ currency symbol ] | void  literal  literal  literal  literal  literal  literal  literal  literal  literal  literal | Not to be used.  E.g. "Germany".  E.g. "english". **Important:** See Remark 1  Typically containing a point or coma  Typically containing a point, comma or apostrophe  Typically containing comma or semicolon  Typically DD.MM.YYYY  Typically a point, hyphen, blank.  e.g. HH:MM:SS  e.g. colon symbol  e.g. $, €, SFr., etc. | locked  limited access  limited access  limited access  limited access  limited access  limited access  limited access  limited access  limited access  limited access |
| system info [ ]  … [ user name ]  … [ user domain ]  … [ user directory ]  [ app data directory ]  [ app data directory beyond4p ]  [ user app data directory ]  [ user app data directory beyond4p]  [ program directory ]  [ bin directory ]  [ program directory beyond4p]  [ operating system ]  [ operating system product]  [ operating system version]  [ temp directory ]  [ directory separator ]  [ privileges ] | void  literal  literal  literal  literal  literal  literal  literal  literal  literal  literal  literal  literal  literal  literal  literal | Not to be used  Your user name  E.g. XX.XX.NET (Local user domain is read if DNS domain name not found)  C:\users\*username,* LINUX: /home/*username*  C:\ProgramData, LINUX: /home/*username*/.config  C:\ProgramData\Beyond4P, LINUX: /home/*username*/.config/Beyond4P  C:\users\*username*\AppData\Local LINUX: /home/*username*/.config  C:\users\*username*\AppData\Local\Beyond4P LINUX: /home/*username*/.config/Beyond4P  C:\program files, LINUX: /etc (Location of Beyond4P executable)  C:\program files, LINUX: /bin (Location of related files to program)  C:\program files\Beyond4P, LINUX: /etc/Beyond4P  "Windows", "Linux", "MACOS"  E.g. "Windows 10 Pro", LINUX: distribution, e.g. "Ubuntu"  Windows: "6.3". See <https://www.arclab.com/en/kb/cppmfc/get-windows-version-from-registry.html> for details. LINUX: depends on distribution.  C:\users\*username*\AppData\Temp LINUX: \tmp  Windows: \ LINUX: /  One of the following: **standard** Standard user privileges (not sufficient to install SW) **elevated** Sufficient privileges to install programs, etc. **administrator** Local admin (or "Superuser") privileges  (LINUX: if started with "sudo" command prefix) | locked  read only  read only  read only  read only  read only  read only  read only  read only  read only  read only  read only  read only  read only  read only  read only |
| runtime settings [ ]  [ input file character set ]  [ output file character set ]  [ epsilon ] \*  [ blank to zero ]  [ memorize table columns ]  [ registry my space ]  [ verbose ]  [ exit directly ]  [ crlf ]  [ file search ignore case ] | void  literal  literal  numeral  boolean  boolean  literal  literal  boolean  boolean  boolean | Not to be used  WINDOWS: Initial value: "win1252". **Important:** See Remark 2 LINUX: Initial value: "utf-8"  Initial value: "utf-8" **Important:** See Remark 2  Tolerance value for comparing (if tilde symbol applied)  True (default): For arithmetic operations (+, -, \*, /) with numerals on one side and blank literals on the other (absolutely blank, no space symbol tolerated), blanks are treated as zero. If false, exception will be flagged. Example. 5 + "" returns 5.  True (default): Column numbers will be memorized if table references with brackets, e.g. [Directory:Last Name] are used. This accelerates table access, but messes up if the same call is repeated with a different table. False prevents this. See Section 5.8 for details  Available location in registry to write data which is not volatile and causes no harm in the system elsewhere.  Verbose, can take following values: quiet, warnings, standard, loud, debug.  If true: Leaves Beyond4P directly without asking to press Enter key. This is useful for batch jobs. Direct exit will not happen in case of exceptions requiring program discontinuation.  if true: Saves files with CR+LF line breaks. if false: Saves files with LF line breaks  Under LINUX, this value is *false* by default Under WINDOWS, it's *true*.  Path search and match criteria for directory listing, searching files and resolving file names. Note: Opening files with file names is still case sensitive under LINUX. | locked  limited access  limited access  limited access  limited access  limited access  read only  limited access  limited access  limited access |
| command line arguments [ ]  [ 0 ], [ 1 ], [ 2 ], … | numeral  literal | Number of arguments passed. It is at least 1.  Actual command line arguments (array members)  If the current program has been started with the **start**(…) function used by a different program, then the command line arguments 1, 2, etc. will be replaced and take the data type as passed, i.e. numerals, Booleans, dates, parameter sets, etc. are allowed. | Read only  Read only |

Limited access: Values can be read and modified, but the type cannot be changed, e.g. replacing literal by a numeral.

**Remark 1: Language settings.**

The following values for **local settings [ language ]** are allowed. Blank values are forbidden.

This value is used by the function **literal** ( … ) if formatting relies on a language setting and no language has been specified explicitly.

Supported languages:

|  |  |
| --- | --- |
| **language** | **Remarks** |
| English | This is the default setting. |
| german |  |
| French  french without accents | Written weekdays and months contain no accents, e.g. Fevrier instead of Février. |
| Italian |  |
| Spanish |  |
| Swedish |  |

**Remark 3: File Character Sets.**

The following values for **local settings [ input file character set ]** and **local settings [ output file character set ]** are allowed. Blank values are forbidden.

This value is used by the function **table load** ( … ) and **table save** ( … ) if no character sets have been specified explicitly in the parameters passed.

|  |  |
| --- | --- |
| **input file character set**  **output file character set** |  |
| ansi |  |
| iso8859-1 |  |
| win1252 | This is the default setting. |
| utf-8 |  |
| utf-16 |  |
| utf-16 big endian |  |

**Remark 3: verbose.**

The following values for **runtime settings [ verbose ]** are allowed. Blank values are forbidden.

This value is used by throughout the entire system to control the extent of commenting.

|  |  |  |
| --- | --- | --- |
| **verbose** | **Suppressed** | **Not suppressed** |
| quiet | Automatically generated comments  Warning messages | Application-intended I/O, e.g. calling **echo** (…)  Error messages |
| warnings | Automatically generated comments | Application-intended I/O, e.g. calling **echo** (…)  Error and warning messages |
| standard | This is the default setting. | Application-intended I/O, e.g. calling **echo** (…)  Error and warning messages  Automatically generated comments |
| loud |  | Like 'standard', but further details |
| debug |  | Like 'loud', but further details useful for debugging code. |