**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

# Text, File Format and I/O Conventions

## 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.

# Formulas and Operators

### 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.

## 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.

# Special Features

## Exceptions



## 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) | | | | |

# 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. |