Open Document Format for Office Applications (OpenDocument) Version 1.4. Part 4: Recalculated Formula (OpenFormula) Format

Committee Specification Draft 01

11 March 2024

This stage:
https://docs.oasis-open.org/office/v1.4/csd01/part4-formula/OpenDocument-v1.4-csd01-part4-formula.odt (Authoritative)
https://docs.oasis-open.org/office/v1.4/csd01/part4-formula/OpenDocument-v1.4-csd01-part4-formula.html
https://docs.oasis-open.org/office/v1.4/csd01/part4-formula/OpenDocument-v1.4-csd01-part4-formula.pdf

Previous stage:

Latest stage:
https://docs.oasis-open.org/office/v1.4/OpenDocument-v1.4-part4-formula.odt (Authoritative)
https://docs.oasis-open.org/office/v1.4/OpenDocument-v1.4-part4-formula.html
https://docs.oasis-open.org/office/v1.4/OpenDocument-v1.4-part4-formula.pdf

Technical Committee:
OASIS Open Document Format for Office Applications (OpenDocument) TC

Chairs:
Patrick Durusau (patrick@durusau.net), Individual
Svante Schubert (svante.schubert@gmail.com), Individual

Editors:
Francis Cave (francis@franciscave.com), Individual
Patrick Durusau (patrick@durusau.net), Individual
Svante Schubert (svante.schubert@gmail.com), Individual
Michael Stahl (michael.stahl@allotropia.de), allotropia software GmbH

Additional artifacts:
This prose specification is one component of a Work Product which includes:

- *Open Document Format for Office Applications (OpenDocument) Version 1.4. Part 1: Introduction*. https://docs.oasis-open.org/office/v1.4/csd01/part1-introduction/OpenDocument-v1.4-csd01-part1-introduction.html.
- *Open Document Format for Office Applications (OpenDocument) Version 1.4. Part 2: Packages*. https://docs.oasis-open.org/office/v1.4/csd01/part2-packages/OpenDocument-v1.4-csd01-part2-packages.html.

- *Open Document Format for Office Applications (OpenDocument) Version 1.4. Part 3: OpenDocument Schema*. https://docs.oasis-open.org/office/v1.4/csd01/part3-schema/OpenDocument-v1.4-csd01-part3-schema.html.
- *Open Document Format for Office Applications (OpenDocument) Version 1.4. Part 4: Recalculated Formula (OpenFormula) Format*. (this part) https://docs.oasis-open.org/office/v1.4/csd01/part4-formula/OpenDocument-v1.4-csd01-part4-formula.html.
- XML/RNG schemas and OWL ontologies. https://docs.oasis-open.org/office/v1.4/csd01/schemas/.

Related work:

This specification replaces or supersedes:

- *OASIS Open Document Format for Office Applications (OpenDocument) Version 1.3*. Part 4: Recalculated Formula (OpenFormula) Format. Edited by Francis Cave, Patrick Durusau, Svante Schubert and Michael Stahl. 27 April 2021. OASIS Standard. https://docs.oasis-open.org/office/OpenDocument/v1.3/os/part4-formula/OpenDocument-v1.3-os-part4-formula.html. Latest stage: https://docs.oasis-open.org/office/OpenDocument/v1.3/OpenDocument-v1.3-part4-formula.html.

Abstract:

This document is Part 4 of the Open Document Format for Office Applications (OpenDocument) Version 1.4 specification.

Status:

This document was last revised or approved by the OASIS Open Document Format for Office Applications (OpenDocument) TC on the above date. The level of approval is also listed above. Check the "Latest stage" location noted above for possible later revisions of this document. Any other numbered Versions and other technical work produced by the Technical Committee (TC) are listed at https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=office#technical.

TC members should send comments on this specification to the TC's email list. Others should send comments to the TC's public comment list, after subscribing to it by following the instructions at the "Send A Comment" button on the TC's web page at https://www.oasis-open.org/committees/office/.

This specification is provided under the RF on Limited Terms Mode of the OASIS IPR Policy, the mode chosen when the Technical Committee was established. For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the TC's web page (https://www.oasis-open.org/committees/office/ipr.php).

Note that any machine-readable content (Computer Language Definitions) declared Normative for this Work Product is provided in separate plain text files. In the event of a discrepancy between any such plain text file and display content in the Work Product's prose narrative document(s), the content in the separate plain text file prevails.

Citation format:

When referencing this specification the following citation format should be used:

[OpenDocument-v1.4-part4]

Open Document Format for Office Applications (OpenDocument) Version 1.4. Part 4: Recalculated Formula (OpenFormula) Format. Edited by Francis Cave, Patrick Durusau, Svante Schubert and Michael Stahl. 11 March 2024. OASIS Committee Specification Draft 01. https://docs.oasis-open.org/office/v1.4/csd01/part4-formula/OpenDocument-v1.4-csd01-part4-formula.html. Latest stage: https://docs.oasis-open.org/office/v1.4/OpenDocument-v1.4-part4-formula.html.

Notices

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full Policy may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

As stated in the OASIS IPR Policy, the following three paragraphs in brackets apply to OASIS Standards Final Deliverable documents (Committee Specification, OASIS Standard, or Approved Errata).

[OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Standards Final Deliverable, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this deliverable.]

[OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this OASIS Standards Final Deliverable by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this OASIS Standards Final Deliverable. OASIS may include such claims on its website, but disclaims any obligation to do so.]

[OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this OASIS Standards Final Deliverable or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Standards Final Deliverable, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.]

The name "OASIS" is a trademark of OASIS, the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see https://www.oasis-open.org/policies-guidelines/trademark/ for above guidance.

Table of Contents

1 Introduction

1.1 Introduction

This document is part of the Open Document Format for Office Applications (OpenDocument) Version 1.3 specification. It defines a formula language for OpenDocument documents, which is also called OpenFormula.

OpenFormula is a specification of an open format for exchanging recalculated formulas between office applications, in particular, formulas in spreadsheet documents. OpenFormula defines data types, syntax, and semantics for recalculated formulas, including predefined functions and operations.

OpenFormula is intended to be a supporting document to the Open Document Format for Office Applications (OpenDocument) format, particularly for defining its attributes `table:formula` and `text:formula`. It can also be used in other circumstances where a simple, easy-to-read infix text notation is desired for exchanging recalculated formulas.

Note: Using OpenFormula allows document creators to change the office application they use, exchange formulas with others (who may use a different application), and access formulas far in the future, with confidence that the recalculated formulas in their documents will produce equivalent results if given equivalent inputs.

1.2 Terminology

All text is normative unless otherwise labeled.

Within the normative text of this specification, the terms "shall", "shall not", "should", "should not", "may" and “need not” are to be interpreted as described in Annex H of [ISO/IEC Directives].

1.3 Purpose

OpenFormula defines:

1. data types
2. syntax
3. semantics

for recalculated formulas. 3.5

OpenFormula also defines functions.

OpenFormula does not define:

1. a user interface
2. a general notation for mathematical expressions

1.4 Normative References

[CharModel] Martin J. Dürst, et. al., Character Model for the World Wide Web 1.0: Fundamentals, http://www.w3.org/TR/2005/REC-charmod-20050215/, W3C, 2005.

[ISO/IEC Directives] ISO/IEC Directives, Part 2 (Fifth Edition) Rules for the structure and drafting of International Standards, International Organization for Standardization and International Electrotechnical Commission, 2004.

[ISO4217] ISO 4217:2008 Codes for the representation of currencies and funds, International Organization for Standardization and International Electrotechnical Commission, 2008.

[ISO8601] ISO 8601:2019 ISO 8601:2019 Date and time — Representations for information interchange — Part 1: Basic rules, International Organization for Standardization, 2019.

[RFC3986] T. Berners-Lee, R. Fielding, L. Masinter, Uniform Resource Identifier (URI): Generic Syntax, http://www.ietf.org/rfc/rfc3986.txt, IETF, 2005.

[RFC3987] M. Duerst, M. Suignard, Internationalized Resource Identifiers (IRIs), http://www.ietf.org/rfc/rfc3987.txt, IETF, 2005.

[UNICODE] The Unicode Consortium. The Unicode Standard, Version 5.2.0, defined by: The Unicode Standard, Version 5.2 (Mountain View, CA, The Unicode Consortium, 2009. ISBN 978-1-936213-00-9). (http://www.unicode.org/versions/Unicode5.2.0/).

[UTR15] Mark Davis, Martin Dürst, Unicode Normalization Forms, Unicode Technical Report #15, http://www.unicode.org/reports/tr15/tr15-25.html, 2005.

[XML1.0] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau , Extensible Markup Language (XML) 1.0 (Fourth Edition), http://www.w3.org/TR/2006/REC-xml-20060816/, W3C, 2006.

1.5 Non-Normative References

[JISX0201] The Unicode Consortium., JIS X 0201 (1976) to Unicode 1.1 Table, 1994, http://www.unicode.org/Public/MAPPINGS/OBSOLETE/EASTASIA/JIS/JIS0201.TXT.

[JISX0208] The Unicode Consortium., JIS X 0208 (1990) to Unicode, 1994, http://www.unicode.org/Public/MAPPINGS/OBSOLETE/EASTASIA/JIS/JIS0208.TXT.

[UAX11] Asmus Freytag, East Asian Width, Unicode Standard Annex #11, http://www.unicode.org/reports/tr11/tr11-19.html, 2009.

2 Expressions and Evaluators

2.1 Introduction

The OpenDocument specification defines conformance for formula expressions and evaluators. For evaluators, there are three groups of features that an evaluator may support. This chapter defines the basic requirements for the individual conformance targets.

2.2 OpenDocument Formula Expression

An *OpenDocument formula expression* shall adhere to the expression syntax defined in chapter 5. It may use subsets or supersets of OpenFormula.

2.3 Evaluator Conformance

2.3.1 OpenDocument Formula Evaluator

An *OpenDocument Formula Evaluator* is a program that can parse and recalculate OpenDocument formula expressions, and that meets the following additional requirements:

A) It may implement subsets or supersets of this specification.

B) It shall conform to one of: (C16) OpenDocument Formula Small Group Evaluator, (C17) OpenDocument Formula Medium Group Evaluator or (C18) OpenDocument Formula Large Group Evaluator

C) It may implement additional functions beyond those defined in this specification. It may further implement additional formula syntax, additional operations, additional optional parameters for functions, or may consider function parameters to be optional when they are required by this specification.

D) Applications should clearly document their extensions in their user documentation, both online and paper, in a manner so users would be likely to be aware when they are using a non-standard extension.

> **Note 1:** An expression may reference a function not defined by this specification by name, or depend on implementation-defined behavior, or on semantics not guaranteed by this specification. Reference to or dependence upon functions or behavior not defined by this standard may impair the interoperability of the resulting expression(s).
>
> **Note 2:** This specification defines formulas in terms of a canonical text representation used for exchange. If formulas are contained in XML attributes some characters shall be escaped as required by the XML specification (e.g., the character & shall be escaped in XML attributes using notations such as &). All string and character literals references by this specification are in the value space defined by [UNICODE] thus, “A” is U+0041, “Z” is U+005A, and the range of characters “A-Z” is the range U+0041 through U+005A inclusive.

2.3.2 OpenDocument Formula Small Group Evaluator

An *OpenDocument Formula Small Group Evaluator* is an OpenDocument Formula Evaluator that meets the following additional requirements:

A) It shall implement at least the limits defined in the “Basic Limits” section. 3.7

B) It shall implement the syntax defined in these sections on syntax: Criteria 4.11.11; Basic Expressions 5.2; Constant Numbers 5.3; Constant Strings 5.4; Operators 5.5; Functions and Function Parameters 5.6; Nonstandard Function Names Error: Reference source not found; References 5.8; Simple Named Expressions ; Errors 5.12; Whitespace 5.14.

C) It shall implement all implicit conversions for the types it implements, at least Text 6.3.14, Conversion to Number Error: Reference source not found, Reference , Conversion to Logical 6.3.12, and when an expression returns an Error.

D) It shall implement the following operators (which are all the operators except reference union (~)): Infix Operator Ordered Comparison ("<", "<=", ">", ">=") Error: Reference source not found; Infix Operator "&" 6.4.10; Infix Operator "+" 6.4.2; Infix Operator "-" 6.4.3; Infix Operator "*" 6.4.4; Infix Operator "/" 6.4.5; Infix Operator "^" 6.4.6; Infix Operator "=" 6.4.7; Infix Operator "<>" 6.4.8; Postfix Operator "%" 6.4.14; Prefix Operator "+" 6.4.15; Prefix Operator "-" 6.4.16; Infix Operator Reference Intersection ("!") 6.4.12; Infix Operator Range (":") 6.4.11.

E) It shall implement at least the following functions as defined in this specification: ABS 6.16.2 ; ACOS 6.16.3 ; AND 6.15.2 ; ASIN 6.16.7 ; ATAN 6.16.9 ; ATAN2 6.16.10 ; AVERAGE 6.18.3 ; AVERAGEIF 6.18.5 ; CHOOSE 6.14.3 ; COLUMNS 6.13.5 ; COS 6.16.19 ; COUNT 6.13.6 ; COUNTA 6.13.7 ; COUNTBLANK 6.13.8 ; COUNTIF 6.13.9 ; DATE 6.10.2 ; DAVERAGE 6.9.2 ; DAY 6.10.5 ; DCOUNT 6.9.3 ; DCOUNTA 6.9.4 ; DDB 6.12.14 ; DEGREES 6.16.25 ; DGET 6.9.5 ; DMAX 6.9.6 ; DMIN 6.9.7 ; DPRODUCT 6.9.8 ; DSTDEV 6.9.9 ; DSTDEVP 6.9.10 ; DSUM 6.9.11 ; DVAR 6.9.12 ; DVARP 6.9.13 ; EVEN 6.16.30 ; EXACT 6.20.8 ; EXP 6.16.31 ; FACT 6.16.32 ; FALSE 6.15.3 ; FIND 6.20.9 ; FV 6.12.20 ; HLOOKUP 6.14.5 ; HOUR 6.10.11 ; IF 6.15.4 ; INDEX 6.14.6 ; INT 6.17.2 ; IRR 6.12.24 ; ISBLANK 6.13.14 ; ISERR 6.13.15 ; ISERROR 6.13.16 ; ISLOGICAL 6.13.19 ; ISNA 6.13.20 ; ISNONTEXT 6.13.21 ; ISNUMBER 6.13.22 ; ISTEXT 6.13.25 ; LEFT 6.20.12 ; LEN 6.20.13 ; LN 6.16.39 ; LOG 6.16.40 ; LOG10 6.16.41 ; LOWER 6.20.14 ; MATCH 6.14.9 ; MAX 6.18.45 ; MID 6.20.15 ; MIN 6.18.48 ; MINUTE 6.10.13 ; MOD 6.16.42 ; MONTH 6.10.14 ; N 6.13.26 ; NA 6.13.27 ; NOT 6.15.7 ; NOW 6.10.16 ; NPER 6.12.29 ; NPV 6.12.30 ; ODD 6.16.44 ; OR 6.15.8 ; PI 6.16.45 ; PMT 6.12.36 ; POWER 6.16.46 ; PRODUCT 6.16.47 ; PROPER 6.20.16 ; PV 6.12.41 ; RADIANS 6.16.49 ; RATE 6.12.42 ; REPLACE 6.20.17 ; REPT 6.20.18 ; RIGHT 6.20.19 ; ROUND 6.17.5 ; ROWS 6.13.30 ; SECOND 6.10.17 ; SIN 6.16.55 ; SLN 6.12.45 ; SQRT 6.16.58 ; STDEV 6.18.72 ; STDEVP 6.18.74 ; SUBSTITUTE 6.20.21 ; SUM 6.16.61 ; SUMIF 6.16.62 ; SYD 6.12.46 ; T 6.20.22 ; TAN 6.16.69 ; TIME 6.10.18 ; TODAY 6.10.20 ; TRIM 6.20.24 ; TRUE 6.15.9 ; TRUNC 6.17.8 ; UPPER 6.20.27 ; VALUE 6.13.34 ; VAR 6.18.82 ; VARP 6.18.84 ; VLOOKUP 6.14.12 ; WEEKDAY 6.10.21 ; YEAR 6.10.24

F) It need not evaluate references that contain more than one area.

G) It need not implement inline arrays 5.13, complex numbers 4.4, and the reference union operator 6.4.13.

Note: This specification does not mandate a user interface for international characters, so a resource-constrained application may choose to not show the traditional glyph (e.g., it may show the [UNICODE] numeric code instead).

2.3.3 OpenDocument Formula Medium Group Evaluator

An *OpenDocument Formula Medium Group Evaluator* is an OpenDocument Small Group Formula Evaluator that meets the following additional requirements:

A) It shall implement the following functions as defined in this specification: ACCRINT 6.12.2 ; ACCRINTM 6.12.3 ; ACOSH 6.16.4 ; ACOT 6.16.5 ; ACOTH 6.16.6 ; ADDRESS 6.14.2 ; ASINH 6.16.8 ; ATANH 6.16.11 ; AVEDEV 6.18.2 ; BESSELI 6.16.12 ; BESSELJ 6.16.13 ; BESSELK 6.16.14 ; BESSELY 6.16.15 ; BETADIST 6.18.7 ; BETAINV 6.18.8 ; BINOMDIST 6.18.10 ; CEILING 6.17.1 ; CHAR 6.20.3 ; CLEAN 6.20.4 ; CODE 6.20.5 ; COLUMN 6.13.4 ; COMBIN 6.16.16 ; CONCATENATE 6.20.6 ; CONFIDENCE 6.18.16 ; CONVERT 6.16.18 ; CORREL 6.18.17 ; COSH 6.16.20 ; COT 6.16.21 ; COTH 6.16.22 ; COUPDAYBS 6.12.5 ; COUPDAYS 6.12.6 ; COUPDAYSNC 6.12.7 ; COUPNCD 6.12.7 ; COUPNUM 6.12.9 ; COUPPCD 6.12.10 ; COVAR 6.18.18 ; CRITBINOM 6.18.19 ; CUMIPMT 6.12.11 ; CUMPRINC 6.12.12 ; DATEVALUE 6.10.4 ; DAYS360 6.10.7 ; DB 6.12.13 ; DEVSQ 6.18.20 ; DISC 6.12.15 ; DOLLARDE 6.12.16 ; DOLLARFR 6.12.17 ; DURATION 6.12.18 ; EFFECT

6.12.19 ; EOMONTH 6.10.10 ; ERF 6.16.27 ; ERFC 6.16.28 ; EXPONDIST 6.18.21 ; FISHER 6.18.26 ; FISHERINV 6.18.27 ; FIXED 6.20.10 ; FLOOR 6.17.3 ; FORECAST 6.18.28 ; FTEST 6.18.30 ; GAMMADIST 6.18.31 ; GAMMAINV 6.18.32 ; GAMMALN 6.16.35 ; GCD 6.16.36 ; GEOMEAN 6.18.34 ; HARMEAN 6.18.36 ; HYPGEOMDIST 6.18.37 ; INTERCEPT 6.18.38 ; INTRATE 6.12.22 ; ISEVEN 6.13.17 ; ISODD 6.13.23 ; ISOWEEKNUM 6.10.12 ; KURT 6.18.39 ; LARGE 6.18.40 ; LCM 6.16.38 ; LEGACY.CHIDIST 6.18.11 ; LEGACY.CHIINV 6.18.13 ; LEGACY.CHITEST 6.18.15 ; LEGACY.FDIST 6.18.23 ; LEGACY.FINV 6.18.25 ; LEGACY.NORMSDIST 6.18.54 ; LEGACY.NORMSINV 6.18.55 ; LEGACY.TDIST 6.18.77 ; LINEST 6.18.41 ; LOGEST 6.18.42 ; LOGINV 6.18.43 ; LOGNORMDIST 6.18.44 ; LOOKUP 6.14.8 ; MDURATION 6.12.26 ; MEDIAN 6.18.47 ; MINVERSE 6.5.3 ; MIRR 6.12.27 ; MMULT 6.5.4 ; MODE 6.18.50 ; MROUND 6.17.4 ; MULTINOMIAL 6.16.43 ; NEGBINOMDIST 6.18.51 ; NETWORKDAYS 6.10.15 ; NOMINAL 6.12.28 ; ODDFPRICE 6.12.31 ; ODDFYIELD 6.12.32 ; ODDLPRICE 6.12.33 ; ODDLYIELD 6.12.34 ; OFFSET 6.14.11 ; PEARSON 6.18.56 ; PERCENTILE 6.18.57 ; PERCENTRANK 6.18.58 ; PERMUT 6.18.59 ; POISSON 6.18.62 ; PRICE 6.12.38 ; PRICEMAT 6.12.40 ; PROB 6.18.63 ; QUARTILE 6.18.64 ; QUOTIENT 6.16.48 ; RAND 6.16.50 ; RANDBETWEEN 6.16.51 ; RANK 6.18.65 ; RECEIVED 6.12.43 ; ROMAN 6.19.17 ; ROUNDDOWN 6.17.6 ; ROUNDUP 6.17.7 ; ROW 6.13.29 ; RSQ 6.18.66 ; SERIESSUM 6.16.53 ; SIGN 6.16.54 ; SINH 6.16.56 ; SKEW 6.18.67 ; SKEWP 6.18.68 ; SLOPE 6.18.69 ; SMALL 6.18.70 ; SQRTPI 6.16.59 ; STANDARDIZE 6.18.71 ; STDEVA 6.18.73 ; STDEVPA 6.18.75 ; STEYX 6.18.76 ; SUBTOTAL 6.16.60 ; SUMPRODUCT 6.16.64 ; SUMSQ 6.16.65 ; SUMX2MY2 6.16.66 ; SUMX2PY2 6.16.67 ; SUMXMY2 6.16.68 ; TANH 6.16.70 ; TBILLEQ 6.12.47 ; TBILLPRICE 6.12.48 ; TBILLYIELD 6.12.49 ; TIMEVALUE 6.10.19 ; TINV 6.18.78 ; TRANSPOSE 6.5.6 ; TREND 6.18.79 ; TRIMMEAN 6.18.80 ; TTEST 6.18.81 ; TYPE 6.13.33 ; VARA 6.18.83 ; VDB 6.12.50 ; WEEKNUM 6.10.22 ; WEIBULL 6.18.86 ; WORKDAY 6.10.23 ; XIRR 6.12.51 ; XNPV 6.12.52 ; YEARFRAC 6.10.25 ; YIELD 6.12.53 ; YIELDDISC 6.12.54 ; YIELDMAT 6.12.55 ; ZTEST 6.18.87

B) It shall implement the Infix Operator Reference Union ("~") 6.4.13

C) It shall evaluate references with more than one area.

2.3.4 OpenDocument Formula Large Group Evaluator

An *OpenDocument Formula Large Group Evaluator* is an OpenDocument Medium Group Formula Evaluator that meets the following additional requirements:

A) It shall implement the syntax defined in these sections on syntax: Inline Arrays 5.13; Automatic Intersection 5.10.6; External Named Expressions 5.11.

B) It shall implement the complex number type as discussed in the section on Complex Number 4.4, array formulas, and Sheet-local Named Expressions.

It shall implement the following functions as defined in this specification: AMORLINC 6.12.4 ; ARABIC 6.19.2 ; AREAS 6.13.2 ; ASC 6.20.2 ; AVERAGEA 6.18.4 ; AVERAGEIFS 6.18.6 ; BASE 6.19.3 ; BIN2DEC 6.19.4 ; BIN2HEX 6.19.5 ; BIN2OCT 6.19.6 ; BINOM.DIST.RANGE 6.18.9 ; BITAND 6.6.2 ; BITLSHIFT 6.6.3 ; BITOR 6.6.4 ; BITRSHIFT 6.6.5 ; BITXOR 6.6.6 ; CHISQDIST 6.18.12 ; CHISQINV 6.18.14 ; COMBINA 6.16.17 ; COMPLEX 6.8.2 ; COUNTIFS 6.13.10 ; CSC 6.16.23 ; 6.16.23CSCH 6.16.24 ; DATEDIF 6.10.3 ; DAYS 6.10.6 ; DDE 6.11.2 ; DEC2BIN 6.19.7 ; DEC2HEX 6.19.8 ; DEC2OCT 6.19.9 ; DECIMAL 6.19.10 ; DELTA 6.16.26 ; EDATE 6.10.9 ; ERROR.TYPE 6.13.11; EUROCONVERT 6.16.29 ; FACTDOUBLE 6.16.33 ; FDIST 6.18.22 ; FINDB 6.7.2 ; FINV 6.18.24 ; FORMULA 6.13.12 ; FREQUENCY 6.18.29 ; FVSCHEDULE 6.12.21 ; GAMMA 6.16.34 ; GAUSS 6.18.33 ; GESTEP 6.16.37 ; GETPIVOTDATA 6.14.4 ; GROWTH 6.18.35 ; HEX2BIN 6.19.11 ; HEX2DEC 6.19.12 ; HEX2OCT 6.19.13 ; HYPERLINK 6.11.3 ; IFERROR 6.15.5 ; IFNA 6.15.6 ; IMABS 6.8.3 ; IMAGINARY 6.8.4 ; IMARGUMENT 6.8.5 ; IMCONJUGATE 6.8.6 ; IMCOS 6.8.7 ; IMCOT 6.8.9 ; IMCSC 6.8.10 ; IMCSCH 6.8.11 ; IMDIV 6.8.12 ; IMEXP 6.8.13 ; IMLN 6.8.14 ; IMLOG10 6.8.15 ; IMLOG2 6.8.16 ; IMPOWER 6.8.17 ; IMPRODUCT 6.8.18 ; IMREAL 6.8.19 ; IMSEC 6.8.22 ; IMSECH 6.8.23 ; IMSIN 6.8.20 ; IMSQRT 6.8.24 ; IMSUB 6.8.25 ; IMSUM 6.8.26 ; IMTAN 6.8.27; INDIRECT 6.14.7 ; INFO 6.13.13 ; IPMT 6.12.23 ; ISFORMULA 6.13.18 ; ISPMT 6.12.25 ; ISREF 6.13.24 ; JIS 6.20.11 ; LEFTB 6.7.3 ; LENB 6.7.4 ; MAXA 6.18.46 ; MDETERM 6.5.2 ; MULTIPLE.OPERATIONS 6.14.10 ; MUNIT 6.5.5 ; MIDB 6.7.5 ; MINA 6.18.49 ; NORMDIST 6.18.52 ; NORMINV 6.18.53 ; NUMBERVALUE 6.13.28 ;

OCT2BIN 6.19.14 ; OCT2DEC 6.19.15 ; OCT2HEX 6.19.16 ; PDURATION 6.12.35 ; PERMUTATIONA 6.18.60 ; PHI 6.18.61 ; PPMT 6.12.37 ; PRICEDISC 6.12.39 ; REPLACEB 6.7.6 ; RIGHTB 6.7.7 ; RRI 6.12.44 ; SEARCH 6.20.20 ; SEARCHB 6.7.8 ; SEC 6.16.52 ; SECH 6.16.57 ; SHEET 6.13.31 ; SHEETS 6.13.32 ; SUMIFS 6.16.63 ; TEXT 6.20.23 ; UNICHAR 6.20.25 ; UNICODE 6.20.26 ; VARPA 6.18.85 ; XOR 6.15.10

Note: The following functions are documented by this specification, but not included even in the Large group:CELL 6.13.3 ; DOLLAR 6.20.7

2.4 Variances (Implementation-defined, Unspecified, and Behavioral Changes)

Applications should document all implementation-defined variances from this standard in a manner that enables application users to obtain the information.

In a few cases a specific approach is required (e.g., string indexes begin at one), which may vary in the user interfaces of different implementations.

> **Note 1:** In practice, for nearly all documents the differences are irrelevant. The primary variances and differences from OpenFormula and some existing applications are:

- Some conversions between types are not required to be automatic. In particular, applications may, but need not, perform automatic conversion of text in a cell when it is to be used as a number.
- There need not be a distinguishable Logical type. Applications may have a Logical type distinct from Number and Text (see Distinct Logical 8.2), but Logical values may also be represented by the Number type using the values 1 (TRUE) and 0 (FALSE). This means that functions that take number sequences (such as SUM) may but need not include true and false values in the sequence.
- Applications vary on the set of Errors they support. In this specification the only distinguished Error is #N/A; all other errors are simply errors, allowing applications to choose the Error set that best meets their needs.
- In this specification, string index positions start from 1. Users of applications with string index positions starting from 0 shall add and subtract 1 on import/export of this format, as appropriate.
- Database criteria match patterns (such as the pattern matching language for text) have historically varied: Some support glob syntax (e.g., a*b is a, followed by 0 or more characters, followed by b), while others support traditional regular expression syntax (e.g., a*b is zero or more a’s, followed by b). This specification supports both pattern languages.

> **Note 2:** Interoperability is improved by the use of the DATE 6.10.2 and TIME 6.10.18 functions or the textual [ISO8601] date representation because dates in that format do not rely upon epoch or locale-specific settings.

In an OpenDocument file, calculation settings impact formula recalculation, which can be the same or different from a particular application's defaults. These include whether or not text comparisons are case-sensitive, or if search criteria apply to the whole cell.

3 Formula Processing Model

3.1 General

This section describes the basic formula processing model: how expressions are calculated, when recalculation occurs, and limits on formulas.

3.2 Expression Evaluation

3.2.1 General

OpenFormula defines rules for the evaluation of expressions as well as the functions and operators that appear in expressions.

3.2.2 Expression Calculation

Expressions in OpenFormula shall be evaluated by application of the following rules:

1) If an expression consists of a constant Number (5.3), a constant String (5.4), a Reference (5.8), constant Error (per section 5.12), the value of that type is returned.
2) If an expression consists of one or more operations, apply the operators in order of precedence and associativity as defined by Table 1 in 5.5 (Operators). Precedence of operators may be altered by the use of "(" (LEFT PARENTHESES, U+0028) and ")" (RIGHT PARENTHESES, U+0029) to group operators. Evaluate the operator as described in Operator and Function Evaluation, 3.2.3.
3) If an expression consists of a function call (Error: Reference source not found, Error: Reference source not found), evaluate the function as described in Operator and Function Evaluation, 3.2.3.
4) If an expression consists of a named expression (Error: Reference source not found), the result of evaluating the named expression is returned.
5) If an expression consists of a QuotedLabel (5.10), AutomaticIntersection (5.10.6), or Array (Error: Reference source not found), its value is returned. Expression Syntax 5

Once evaluation has completed:

1. If the result is a Reference and a single non-reference value is needed, it is converted to the referenced value, using the rules of Non-Scalar Evaluation, Error: Reference source not found, 1.2.
2. If the result an Array, for the display area, apply the rules of Non-Scalar Evaluation, Error: Reference source not found, 1.1.

3.2.3 Operator and Function Evaluation

Operators and functions in OpenFormula **shall** be evaluated according to their definitions by applying the following rules:

1. The value of all expression arguments are computed. Exceptions to computation of all arguments are noted in a function's specification.

   > **Note:** The practice of computing all argument expressions is known as "eager" evaluation. The IF function is an example of a function that does not require computation of all arguments.

2. If an argument expression evaluates to Error, calculation of the operator or function may short-circuit and return the Error if the function does not suppress error propagation as noted in the function's specification.

3. If an operator or function is passed a value of incorrect type, call the appropriate implicit conversion function to convert the value to the correct type. If conversion is not possible, generate an Error.
4. The function or operation is called with its argument expressions' results, and the result of the function or operation is the evaluation of the expression.

3.3 Non-Scalar Evaluation (aka 'Array expressions')

Non-scalar values passed as arguments to functions are evaluated by intersection or iteration.

1) Evaluation as an implicit intersection of the argument with the expression's evaluation position.

1.1) Inline Arrays
Element (0;0) of the array is used in place of the array.

Note 1:

```
=ABS({-3;-4})       => ABS(-3)      // row vector
=ABS({-3|-4})       => ABS(-3)      // column vector
=ABS({-3;-4|-6;-8}) => ABS(-3)      // matrix
={1;2;3|4;5;6}      => 1            // simple display
```

1.2) References

1.2.1) If the target reference is a row-vector (Nx1), use the value at the intersection of the evaluation position's column and the reference's row.

Note 2:
in cell B2 : =ABS(A1:C1) => ABS(B1)
If there is no intersection the result is #N/A or a more specific Error value.

Note 3: in cell D4 : =ABS(A1:C1) => #N/A or a more specific Error value.

1.2.2) If the target reference is a column-vector (1xM), the value at the intersection of the evaluation position's row and the reference's column.

Note 4:
in cell B2 : =ABS(A1:A3) => ABS(A2)
in cell D4 : =ABS(A1:A3) => #N/A or a more specific Error value.

2) Matrix evaluation.

If an expression is being evaluated in a cell flagged as a being part of a 'Matrix' (OpenDocument Part 3, 19.683 `table:number-matrix-columns-spanned`):

2.1) The portion of a non-scalar result to be displayed need not be co-extensive with a specified display area. The portion of the non-scalar result to be displayed is determined by:

2.1.1) If the position to be displayed exists in the result, display that position.

2.1.2) If the non-scalar result is 1 column wide, subsequent columns in the display area display the value in the first column. This applies to
- scalars '3'
- singletons '{3}'
- column vectors '{1|2|3}'

2.1.3) If the non-scalar result is 1 row high, subsequent rows in the display area use the value of the first row. This applies to
- scalars '3'
- singletons '{3}'
- row vectors '{1;2;3}'

2.1.4) If none of the other rules apply #N/A

Note 5:
in matrix A1:B3 with ={1;2|3;4|5;6} : cell B2 contains 4. [Rule 2.1.1]

in matrix A1:B3 with ={1|3|5} : cell B2 contains 3. [Rule 2.1.1 for row, and Rule 2.1.2 column]
in matrix A1:B3 with ={2;4} : cell B2 contains 4. [Rule 2.1.3 for row, and Rule 2.1.1 column]
in matrix A1:C4 with ={1;2|3;4|5;6} : cell C1,A4 contain #N/A. [Rule 2.1.4]

Note 6: if a value is not requested it is not displayed
in matrix A1:B2 with ={1;2|3;4|5;6} : the value '6' is not displayed because B3 is not part of the display matrix.

2.2) Calculations with non-scalar inputs are a generalization of (2.1).

When evaluating a formula in 'matrix' mode, and a non-scalar value is passed to a function argument that expects a scalar, the function is evaluated multiple times, iterating over the non-scalar input(s) and putting the function result into a matrix at the position corresponding to the input. Unary/Binary operators, other than range and union, follow the rules for scalar functions when passed non-scalar values.

Inline arrays and references are interchangeable.

2.2.1) Functions returning arrays are not eligible for implicit iteration. When evaluated in 'matrix' mode the {0;0}th element is used.

Note 7:
e.g. =SUM(INDIRECT({"A1";"A2")) would produce the value in A1 when evaluated in array mode.

2.2.2) The result matrix is rectangular, sized with the maximum number of rows and columns from all non-scalar arguments.

Note 8:
={1;2}+{3;4;5} => {4;6;#N/A}
={1}+{1;2} => {2;3}

2.2.3) The result matrix is populated by extracting the corresponding value from each of the non-scalar arguments based on the following rules, and evaluating the function with that set of arguments.

2.2.3.1) If the argument data is a singleton array or a scalar the value is repeated for each evaluation.

Note 9:
= 1 + {1;2;3|4;5;6} => {2;3;4|5;6;7}
= {1} + {1;2;3|4;5;6} => {2;3;4|5;6;7}

2.2.3.2) If the argument data is 1 column wide the value in the corresponding row is used to evaluate all columns in the result matrix.

Note 10:
= {1|2} + {10;20|30;40} => {11;21|32;42}

2.2.3.3) If the argument data is 1 row height the value in the corresponding column is used to evaluate all rows in the result matrix.

Note 11:
= {1;2} + {10;20|30;40} => {11;22|31;42}

2.2.3.4) If one argument data is 1 column wide and another argument data is 1 row height the value of the corresponding row respectively column is used to evaluate all elements in the result matrix.

Note 12:
={1;2} + {10|20} => {11;12|21;22}

2.2.3.5) If an argument is a 2d matrix the argument value in the position corresponding to the current output position is used if it is within range of the supplied argument, otherwise #N/A is used in the calculation.

Note 13:
=MID("abcd";{1;2};{1;2;3}) => {"a";"bc";#N/A}

3.4 Host-Defined Behaviors

A Formula Evaluator operates in an execution environment (a "host"). The behavior of the Formula Evaluator is parametrized by host-defined properties and functions.

The following properties are host-defined:

1. HOST-CASE-SENSITIVE: if true, text comparisons are case-sensitive. This influences the operators =, <>, <, <=, >, and >=, as well as database query functions that use them. Note that the EXACT function is always case-sensitive, regardless of this calculation setting.
2. HOST-PRECISION-AS-SHOWN: If true, calculations are performed using rounded values of those displayed; otherwise, calculations are performed using the precision of the underlying numeric representation.

   Note: This does not impose a particular numeric model. Since implementations may use binary representations, this rounding may be inexact for decimal values.
3. HOST-SEARCH-CRITERIA-MUST-APPLY-TO-WHOLE-CELL If true, the specified search criteria shall apply to the entire cell contents if it is a text match using = or <>; if not, only a subpart of the cell content needs to match the text.
4. HOST-AUTOMATIC-FIND-LABELS: if true, row and column labels are automatically found.
5. HOST-USE-REGULAR-EXPRESSIONS: If true, regular expressions are used for character string comparisons and when searching.
6. HOST-USE-WILDCARDS: If true, wildcards question mark '?' and asterisk '*' are used for character-string comparisons and when searching. Wildcards may be escaped with a tilde '~' character.
7. HOST-NULL-YEAR: This defines how to convert a two-digit year into a four-digit year. Each two-digit year value is interpreted as a year that equals or follows this year.
8. HOST-NULL-DATE: Defines the beginning of the epoch; a numeric date of 0 equals this date.
9. HOST-LOCALE: The locale to be used for locale-dependent operations, such as conversion of text to dates, or text to numbers.
10. HOST-ITERATION-STATUS: If enabled, iterative calculations of cyclic references are performed.
11. HOST-ITERATION-MAXIMUM-DIFFERENCE: If iterative calculations of cyclic references are enabled, the maximum absolute difference between calculation steps that all involved formula cells must yield for the iteration to end and yield a result.
12. HOST-ITERATION-STEPS: If iterative calculations of cyclic references are enabled, the maximum number of steps iterations that are performed if the results are not within HOST-ITERATION-MAXIMUM-DIFFERENCE.

The function HOST-REFERENCE-RESOLVER(Reference) is implementation-defined. This function takes as input a Unicode string containing a Reference according to section 4.8 and returns a resolved value.

3.5 When recalculation occurs

Implementations of OpenFormula typically recalculate formulas when its information is needed. Typical implementations will note what values a formula depends on, and when those dependent values are changed and the formula's results are displayed, it will re-execute the formulas that depend on them to produce the new results (choosing the formulas in the right order based on their dependencies). Implementations may recalculate when a value changes

(this is termed *automatic recalculation*) or on user command (this is termed *manual recalculation*).

Some functions' dependencies are difficult to determine and/or should be recalculated more frequently. These include functions that return today's date or time, random number generator functions (such as RAND 6.16.50), or ones that indirectly determine the cells to act on. Many implementations *always* recalculate formulas including such functions whenever a recalculation occurs. Functions that are *always* recalculated whenever a recalculation occurs are termed *volatile* functions. Functions that are often volatile functions include CELL 6.13.3, HYPERLINK 6.11.3, INDIRECT 6.14.7, INFO 6.13.13, NOW 6.10.16, OFFSET 6.14.11, RAND 6.16.50 and TODAY 6.10.20. Functions that depend on the cell position of the formula they are contained in or the position of a cell they reference need to be recalculated whenever that cell is moved, such functions are COLUMN 6.13.4, ROW 6.13.29 and SHEET 6.13.31. In addition, formulas may indicate that they should *always* be recalculated during a recalculation process by including a forced recalculation marker, as described in the syntax below.

3.6 Numerical Models

This specification does not, by itself, specify a numerical implementation model, though it does imply some minimal levels of accuracy for most functions. For example, an application cannot say that it implements the infix operator “/” as specified in this document if it implements integer-only arithmetic.

3.7 Basic Limits

Evaluators which claim to support “basic limits” shall support at least the following limits:

1. formulas up to at least 1024 characters long, as measured when in OpenDocument interchange format not counting the square brackets around cell addresses, or the “.” in a cell address when the sheet name is omitted.
2. at least 30 parameters per function when the function prototype permits a list of parameters.
3. permit strings of ASCII characters of up to 32,767 (2^15-1) characters.
4. support at least 7 nesting levels of functions.

4 Types

4.1 General

All values defined by OpenFormula have a type. OpenFormula defines Text Error: Reference source not found, Number 4.3, Complex Number 4.4, Logical Error: Reference source not found, Error 4.6, Reference 4.8, ReferenceList 4.9 and Array 4.10 types.

4.2 Text (String)

A Text value (also called a string value) is a Character string as specified in [CharModel].

A text value of length zero is termed the empty string.

Index positions in a text value begin at 1.

Whether or not Unicode Normalization [UTR15] is performed on formulas, formula results or user inputs is implementation-defined. Some functions defined in OpenFormula are labeled as "normalization-sensitive", meaning that the results of the formula evaluation may differ depending on whether normalization occurs, and which normalization form is used. Mixing operands of different normalization forms in the same calculation is undefined.

4.3 Number

4.3.1 General

A number is a numeric value.

Numbers shall be able to represent fractional values (they shall not be limited to only integers). Evaluators may implement Number with a fixed or with a variable bit length. A cell with a constant numeric value has the Number type.

Implementations typically support many subtypes of Number, including Date, Time, DateTime, Percentage, fixed-point arithmetic, and arithmetic supporting arbitrarily long integers, and determine the display format from this.

> **Note:** This specification does not require that specific subtypes be distinguishable from each other, or that the subtype be tracked, but in practice most implementations do such tracking because requiring users to manually format every cell appropriately becomes tedious very quickly. Automatically determining the most likely subtype is especially important for a good user interface when generating OpenDocument format, since some subtypes (such as date, time, and currency) are stored in a different manner depending on their subtype. Thus, this specification identifies some common subtypes and identifies those subtypes where relevant in some function definitions, as an aid to implementing good user interfaces. Many applications vary in the subtype produced when combining subtypes (e.g., what is the result when percentages are multiplied together), so unless otherwise noted these are unspecified. Typical implementations try to heuristically determine the "right" format for a cell when a formula is first created, based on the operations in the formula. Users can then override this format, so as a result the heuristics are not important for data exchange (and thus outside the scope of this specification).

All such Number subtypes shall yield TRUE for the ISNUMBER 6.13.22 function.

4.3.2 Time

Time is a subtype of Number.

Time is represented as a fraction of a day.

4.3.3 Date

Date is a subtype of Number.

Date is represented by an integer value.

A serial date is the expression of a date as the number of days elapsed from a start date called the epoch.

Evaluators shall support all dates from 1904-01-01 through 9999-12-31 (inclusive) in calculations, should support dates from 1899-12-30 through 9999-12-31 (inclusive) and may support a wider date range.

> **Note 1:** Using expressions that assume serial numbers are based on a particular epoch may cause interoperability issues.

Evaluators shall support positive serial numbers. Evaluators may support negative serial numbers to represent dates before an epoch.

> **Note 2:** It is implementation-defined if the year 1900 is treated as a leap year.
>
> **Note 3:** Evaluators that treat 1900 as a non-leap year can use the epoch date 1899-12-30 to compensate for serial numbers that originate from evaluators that treat 1900 as a leap year and use 1899-12-31 as an epoch date.

4.3.4 DateTime

DateTime is a subtype of Number. It is a Date plus Time.

4.3.5 Percentage

A percentage is a subtype of Number that may be displayed by multiplying the number by 100 and adding the sign “%” or with other formatting depending upon the number format assigned to the cell where it appears.

4.3.6 Currency

A currency is a subtype of Number that may appear with or without a currency symbol or with other formatting depending upon the number format assigned to the cell where it appears.

4.3.7 Logical (Number)

Applications may have a Logical type distinct from both Number and Text (see Logical (Boolean)4.5 Logical (Boolean)), but Logical values may also be represented by the Number type using the values 1 (True) and 0 (False). (see 8.2 Distinct Logical for details)

4.4 Complex Number

A complex number (sometimes also called an imaginary number) is a pair of real numbers including a *real part* and an *imaginary part.* In mathematics, complex numbers are often written as $x + iy$, where x (the real part) and y (the imaginary part) are real numbers and i is $\sqrt{-1}$. A complex number can also be written as $re^{i\theta} = r\cos(\theta) + ir\sin(\theta)$, where r is the *modulus* of the complex number (a real number) and θ is the *argument* or *phase* (a real number representing an angle in radians).

A complex number may, but need not be, represented as a Number or Text. The results of the functions ISNUMBER() 6.13.22 and ISTEXT() 6.13.25 are implementation-defined when applied to a complex number.

Functions and operators that accept complex numbers shall accept Text values as complex numbers (Conversion to Complex Number 6.3.10), as well as Numbers that are not complex numbers.

> **Note 1:** IMSUM("3i";4) will produce the same result as COMPLEX(4;3).
>
> **Note 2:** Expression authors should be aware that use of functions that are not defined by OpenFormula as accepting complex numbers as input may impair interoperability.

Equality can be tested using IMSUB to compute the difference, use IMABS to find the absolute difference, and then ensure the absolute difference is smaller than or equal to some nonnegative value (for exact equality, compare for equality with 0).

4.5 Logical (Boolean)

Applications may have a Logical type distinct from both Number and Text, but Logical values may also be represented by the Number type using the values 1 (True) and 0 (False) (see 4.3.7 Logical (Number)). (see 8.2 Distinct Logical for details)

4.6 Error

An Error is one of a set of possible error values. Implementations may have many different error values (see 5.12), but one error value in particular is distinct: #N/A, the result of the NA() function. Users may choose to enter some data values as #N/A, so that this error value propagates to any other formula that uses it, and may test for this using the function ISNA().

Functions and operators that receive one or more error values as an input shall produce one of those input error values as their result, except when the formula or operator is specifically defined to do otherwise.

In an OpenDocument document, if an error value is the result of a cell computation it shall be stored as if it was a string. That is, the `office:value-type` (OpenDocument Part 3, 19.389) of an error value is `string`; if the computed value is stored, it is stored in the attribute `office:string-value` (OpenDocument Part 3, 19.383).

Note: This does not change an Error into a string type (since the Error will be restored on recalculation); this enables applications which cannot recalculate values to display the error information.

4.7 Empty Cell

An empty cell is neither zero nor the empty string, and an empty cell can be distinguished from cells containing values (including zero and the empty string). An empty cell is not the same as an Error, in particular, it is distinguishable from the Error #N/A (not available).

4.8 Reference

A cell position is the location of a single cell at the intersection of a column and a row.

A cell strip consists of cell positions in the same row and in one or more contiguous columns.

A cell rectangle consists of cell positions in the same cell strips of one or more contiguous rows.

A cell cuboid consists of cell positions in the same cell rectangles of one or more contiguous sheets.

A reference is the smallest cuboid that (1) contains specifically-identified cell positions and/or specifically-identified complete columns/rows such that (2) removal of any cell positions either violates condition (1) or does not leave a cuboid.

Cell positions in a cell cuboid/rectangle/strip can resolve to empty cells (section 4.7).

The definitions of specific operations and functions that allow references as operands and parameters stipulate any particular limitations there are on forms of references and how empty cells, when permitted, are interpreted.

4.9 ReferenceList

A reference list contains 1 or more references, in order. A reference list can be passed as an argument to functions where passing one reference results in an identical computation as an arbitrary sequence of single references occupying the identical cell range.

> **Note 1:** For example, SUM([.A1:.B2]) is identical to SUM([.A1]~[.B2]~[.A2]~[.B1]), but COLUMNS([.A1:.B2]), resulting in 2 columns, is not identical to COLUMNS([.A1]~[.B2]~[.A2]~[.B1]), where iterating over the reference list would result in 4 columns.

A reference list cannot be converted to an array.

> **Note 2:** For example, in array context {ABS([.A1]~[.B2]~[.A2]~[.B1])} is an invalid expression, whereas {ABS([.A1:.B2])} is not.

Passing a reference list where a function does not expect one shall generate an Error. Passing a reference list in array iteration context to a function expecting a scalar value shall generate an Error.

4.10 Array

An array is a set of rows each with the same number of columns that contain one or more values. There is a maximum of one value per intersection of row and column. The intersection of a row and column may contain no value.

4.11 Pseudotypes

4.11.1 General

Many functions require a type or a set of types with special properties, and/or process them specially. For example, a "Database" requires headers that are the field names. These specialized types are called *pseudotypes*.

4.11.2 Scalar

A *Scalar* value is a value that has a *single* value. A reference to more than one cell is *not* a scalar (by itself), and is converted to one as described below. An array with more than one element is not a scalar. The types Number (including a complex number), Logical, and Text are scalars.

4.11.3 DateParam

A DateParam is a value that is either a Number (interpreted as a serial number; 4.3.3) or Text; text is automatically converted to a date value. Error: Reference source not found

4.11.4 TimeParam

A TimeParam is a value that is either a Number (interpreted as a serial number; 4.3.2) or Text; text is automatically converted to a time value (fraction of a day). Error: Reference source not found

4.11.5 Integer

An integer is a subtype of Number that has no fractional value. An integer X is equal to INT(X). Division of one integer by another integer may produce a non-integer.

4.11.6 TextOrNumber

TextOrNumber is a value that is either a Number or Text.

4.11.7 Basis

4.11.7.1 General

A basis is a subtype of Integer that specifies the day-count convention to be used in a calculation.

This standard defines five day-count conventions, corresponding to widely used current and historical accounting conventions. Each of these five bases defines two things:

1. How to calculate the number of days between two dates, date1 and date2.
2. How to calculate the number of days in each year between two dates, date1 and date2.

Historically day-count bases used the naming convention x/y, which indicated that the convention assumed x days per month and y days per year. These names are given for reference purposes.

Date Basis	***Historical Name***	***Day Count***	***Days in Year***
0	US (NASD) 30/360	Procedure A, 4.11.7.3	Procedure D, 4.11.7.6
1	Actual/Actual	Procedure B, 4.11.7.4	Procedure E, 4.11.7.7
2	Actual/360	Procedure B, 4.11.7.4	Procedure D, 4.11.7.6
3	Actual/365	Procedure B, 4.11.7.4	Procedure F, 4.11.7.8
4	European 30/360	Procedure C, 4.11.7.5	Procedure D, 4.11.7.6

4.11.7.2 Procedural Notation

The day-count procedures are expressed using notations defined as:

- day(date) returns the day of the month for the given date value, an integer from 1 to 31
- month(date) returns the month of a given date value, an integer from 1-12
- year(date) returns the year of the given date value
- truncate(date) truncates any fractional (hours, minutes, seconds) of a date value and returns the whole date portion.
- Binary comparison operators date1>date2 and date1 == date2
- is-leap-year(year) returns true if year is a leap year, otherwise false.

  Note: Some of the day count procedures use intermediate results that contain counter-factual dates, such as February 30th. This is not an error. The above functions work on such dates as well, e.g., day(February 30th) == 30.

4.11.7.3 Procedure A

1. truncate(date1), truncate(date2)
2. If date1==date2 return 0
3. If date1> date2, then swap the values of date1 and date2.
4. If day(date1)==31 then subtract 1 day from date1
5. If day(date1)==30 and day(date2)==31 then subtract 1 day from date2

6. If both date1 and date2 are the last day of February, change date2 to the 30th of the month.
7. If date1 is the last day of February, change it to the 30th of the month.
8. Return (year(date2)*360 + month(date2)*30 + day(date2)) - (year(date1)*360 + month(date1)*30 + day(date1)).

4.11.7.4 Procedure B

1. truncate(date1), truncate(date2)
2. If date1> date2, then swap the values of date1 and date2.
3. Return the actual numbers of days between date1 and date2, inclusive of date1, but not inclusive of date2.

4.11.7.5 Procedure C

1. truncate(date1), truncate(date2)
2. If date1==date2 return 0
3. If date1> date2, then swap the values of date1 and date2.
4. If day(date1)==31 then subtract 1 from date1
5. If day(date2)==31 then subtract 1 from date 2
6. Return (year(date2)*360 + month(date2)*30 + day(date2)) - (year(date1)*360 + month(date1)*30 + day(date1)).

4.11.7.6 Procedure D

1. Return 360

4.11.7.7 Procedure E

1. Evaluate A: year(date1) != year(date2)
2. Evaluate B: year(date2)!=year(date1)+1
3. Evaluate C: month(date1) < month(date2)
4. Evaluate D: month(date1) == month(date2)
5. Evaluate E: day(date1) < day(date2)
6. Evaluate F: (A and B) or (A and C) or (A and D and E)
7. If F is true then return the average of the number of days in each year between date1 and date2, inclusive.
8. Otherwise, if A and is-leap-year(year(date1)) then return 366
9. Otherwise, if a February 29 occurs between date1 and date2 then return 366
10. Otherwise, if date2 is a February 29, then return 366
11. Otherwise return 365

4.11.7.8 Procedure F

1. Return 365

4.11.8 Criterion

A *criterion* is a single cell Reference, Number or Text. It is used in comparisons with cell contents.

A reference to an empty cell is interpreted as the numeric value 0.

A matching expression can be:

- A Number or Logical value. A matching cell content equals the Number or Logical value.
- A value beginning with a comparator (<, <=, >, >=) Error: Reference source not found or an infix operator (=, <>). = 6.4.7, <> 6.4.8

  For =, if the value is empty it matches empty cells. Empty cell 4.7, = 6.4.7

  For <>, if the value is empty it matches non-empty cells. <> 6.4.8

  For <>, if the value is not empty it matches any cell content except the value, including empty cells.

  > **Note:** "=0" does not match empty cells.

  For = and <>, if the value is not empty and can not be interpreted as a Number type or one of its subtypes and the host-defined property HOST-SEARCH-CRITERIA-MUST-APPLY-TO-WHOLE-CELL is true, comparison is against the entire cell contents, if false, comparison is against any subpart of the field that matches the criteria. For = and <>, if the value is not empty and can not be interpreted as a Number type or one of its subtypes 3.4 applies.
- Other Text value. If the host-defined property HOST-SEARCH-CRITERIA-MUST-APPLY-TO-WHOLE-CELL is true, the comparison is against the entire cell contents, if false, comparison is against any subpart of the field that matches the criteria.

4.11.9 Database

A *database* is a rectangular organized set of data. Any database has a set of one or more *fields* that determine the structure of the database. A database has a set of zero or more *records* with data, and each record contains data for every field (though that field may be empty).

Evaluators shall support the use of ranges as databases if they support any database functions. The first row of a range is interpreted as a set of field names.

> **Note:** Field names of type Text and unique without regard to case enhance the interoperability of data. It is also a common expectation that rows following the first row of data are data records that correspond to field names in the first row.

A single cell containing text can be used as a database; if it is, it is a database with a single field and no data records.

4.11.10 Field

A *field* is a value that selects a field in a database; it is also called a *field selector*. If the field selector is Text, it selects the field in the database with the same name.

Evaluators should match the database field name case-insensitively.

If a field selector is a Number, it is a positive integer and used to select the fields. Fields are numbered from left to right beginning with the number 1.

All functions that accept a field parameter shall, when evaluated, return an Error if the selected field does not exist.

4.11.11 Criteria

A *criteria* is a rectangular set of values, with at least one column and two rows, that selects matching records from a database. The first row lists fields against which expressions will be matched. Error: Reference source not found Rows after the first row contain fields with expressions for matching against database records.

For a record to be selected from a database, all of the expressions in a criteria row shall match.

A reference to an empty cell is interpreted as the numeric value 0.

- Expressions are matched as per 4.11.8 Criterion.

4.11.12 Sequences (NumberSequence, NumberSequenceList, DateSequence, LogicalSequence, and ComplexSequence)

Some functions accept a sequence, i.e., a value that is to be treated as a sequential series of values. The following are sequences: NumberSequence, NumberSequenceList, DateSequence, LogicalSequence, and ComplexSequence.

When evaluating a function that accepts a sequence, the evaluator shall follow the rules for that sequence as defined in section Error: Reference source not found. When processing a ReferenceList, the references are processed in order (first, second if any, and so on). In a cuboid, the first sheet is first processed, followed by later sheets (if any) in order. Inside a sheet, it is implementation-defined as to whether the values are processed row-at-a-time or column-at-a-time, but it shall be one of these two processing orders. If processing row-at-a-time, the sequence shall be produced by processing each row in turn, from smallest to largest column value (e.g., A1, B1, C1). If processing column-at-a-time, the sequence shall be produced by processing each column at a time, from the smallest to the largest row value (e.g., A1, A2, A3).

4.11.13 Any

Any represents a value of any type defined in this standard, including Error values.

5 Expression Syntax

5.1 General

The OpenFormula syntax is defined using the BNF notation of the XML specification, chapter 6 [XML1.0]. Each syntax rule is defined using "::=".

Note: Formulas are typically embedded inside an XML document. When this occurs, characters (such as "<", ">", "", and "&") shall be escaped, as described in section 2.4 of the XML specification [XML1.0]. In particular, the less-than symbol "<" is typically represented as “<”, the double-quote symbol as “"”, and the ampersand symbol as “&” (alternatively, a numeric character reference can be used).

5.2 Basic Expressions

Formulas may start with a '=' (EQUALS SIGN, U+003D), which if present may be followed by a “forced recalculate” marker '=' (EQUALS SIGN, U+003D), followed by an expression. If the second '=' (EQUALS SIGN, U+003D) is present, this formula is a "forced recalculation" formula. If a formula is marked as a "forced recalculation" formula, then it should be recalculated whenever one of its predecessors it depends on changes.

Expressed as a grammar in BNF notation, a formula is specified:

```
Formula ::= Intro? Expression
Intro ::= '=' ForceRecalc?
ForceRecalc ::= '='
```

The primary component of a formula is an `Expression`. Formulas are composed of `Expression`s, which may in turn be composed from other `Expression`s.

```
Expression ::=
      Whitespace* (
      Number |
      String |
      Array |
      PrefixOp Expression |
       Expression PostfixOp |
       Expression InfixOp Expression |
       '(' Expression ')' |
       FunctionName '(' ParameterList ')' |
      Reference |
      QuotedLabel |
      AutomaticIntersection |
      NamedExpression |
       Error
      ) Whitespace*
SingleQuoted ::= "'" ([^'] | "''")+ "'"
```

5.3 Constant Numbers

Constant numbers are written using '.' (FULL STOP, U+002E) dot as the decimal separator. Optional "E" or "e" denotes scientific notation. Syntactically, negative numbers are positive numbers with a prefix "-" (HYPHEN-MINUS, U+002D) operator. A constant number is of type Number.

```
Number ::= StandardNumber |
   '.' [0-9]+ ([eE] [-+]? [0-9]+)?
StandardNumber ::= [0-9]+ ('.' [0-9]+)? ([eE] [-+]? [0-9]+)?
```


Evaluators should be able to read the `Number` format, which accepts a decimal fraction that starts with decimal point '.' (FULL STOP, U+002E), without a leading zero. Evaluators shall write numbers only using the `StandardNumber` format, which requires a leading digit, and shall not write numbers with a leading '.' (FULL STOP, U+002E).

5.4 Constant Strings

Constant strings are surrounded by double-quote characters (QUOTATION MARK, U+0022); a literal double-quote character "" (QUOTATION MARK, U+0022) as string content is escaped by duplicating it. A constant string is of type Text.

```
String ::= '"' ([^"#x00] | '""')* '"'
```

5.5 Operators

Operators are functions with one or more parameters.

```
PrefixOp ::= '+' | '-'
PostfixOp ::= '%'
InfixOp ::= ArithmeticOp | ComparisonOp | StringOp | ReferenceOp
ArithmeticOp ::= '+' | '-' | '*' | '/' | '^'
ComparisonOp ::= '=' | '<>' | '<' | '>' | '<=' | '>='
StringOp ::= '&'
```

There are three predefined reference operators: reference intersection, reference concatenation, and range. The result of these operators may be a 3-dimensional range, with front-upper-left and back-lower-right corners, or even a list of such ranges in the case of cell concatenation.

```
ReferenceOp ::= IntersectionOp | ReferenceConcatenationOp |
RangeOp
IntersectionOp ::= '!'
ReferenceConcatenationOp ::= '~'
RangeOp ::= ':'
```

Table 1 - Operators defines the associativity and precedence of operators, from highest to lowest precedence.

Table 1 - Operators

Associativity	***Operator(s)***	***Comments***
left	:	Range.
left	!	Reference intersection ([.A1:.C4]![.B1:.B5] is [.B1:.B4]). Displayed as the space character in some implementations.
left	~	Reference union. **Note:** Displayed as the function parameter separator in some implementations.
right	+,-	Prefix unary operators, e.g., -5 or -[.A1]. Note that these have a different precedence than add and subtract.
left	%	Postfix unary operator % (divide by 100). Note that this is legal with expressions (e.g., [.B1]%).
left	^	Power (2 ^ 3 is 8).
left	*,/	Multiply, divide.
left	+,-	Binary operations add, subtract. Note that unary (prefix) + and - have a different precedence.
left	&	Binary operation string concatenation. Note that unary (prefix) +

		and - have a different precedence. Note that "&" shall be escaped when included in an XML document
left	=, <>, <, <=, >, >=	Comparison operators equal to, not equal to, less than, less than or equal to, greater than, greater than or equal to

Note 1: Prefix “-” has a higher precedence than “^”, “^” is left-associative, and reference intersection has a higher precedence than reference union.

Note 2: Prefix “+” and “-“ are defined to be right-associative. However, note that typical applications which implement at most the operators defined in this specification (as specified) may implement them as left-associative, because the calculated results will be identical.

Note 3: Precedence can be overridden by using parentheses, so "=2+3*4" computes to 14 but "=(2+3)*4" computes 20. Implementations *should* retain "unnecessary" parentheses and white space, since these are added by people to improve readability.

5.6 Functions and Function Parameters

Functions are called by name, followed by parentheses surrounding a list of parameters. Parameters are separated using the semicolon ';' (SEMICOLON, U+003B) character:

```
FunctionName ::=  LetterXML (LetterXML | DigitXML |
       '_' | '.' | CombiningCharXML)*
```

Where LetterXML, DigitXML, and CombiningCharXML are Letter, Digit, and CombiningChar as they are defined in [XML1.0].

Function names are case-insensitive.

Function calls shall be given a parameter list, though it may be empty. An empty list of parameters is considered a call with 0 parameters, not a call with one parameter that happens to be empty. TRUE() is syntactically a function call with 0 parameters. It is syntactically legitimate to provide empty parameters, though functions need not accept empty parameters unless otherwise noted:

```
ParameterList ::= /* empty */ |
       Parameter (Separator EmptyOrParameter )* |
       Separator EmptyOrParameter /* First param empty */
       (Separator EmptyOrParameter )*
EmptyOrParameter ::= /* empty */ Whitespace* | Parameter
Parameter ::= Expression
Separator ::= ';'
```

5.7 Nonstandard Function Names

When writing a document using function(s) not defined in this specification, an evaluator shall include a prefix in such function names to identify the original definer of the function's semantics. When the origin of a function cannot be determined, producers may omit a prefix. Producers may use the prefix to differentiate between different definition types. Evaluators that do not support a function should compute its result as some Error value other than #N/A.

> **Note**: Examples of implementation-defined functions include extension functions included with an implementation, user-defined functions written by users, and 3rd party functions distributed in libraries.
>
> **Note:** Examples of such names include COM.MICROSOFT.CUBEMEMBER, ORG.OPENOFFICE.STYLE, ORG.GNUMERIC.RANDRAYLEIGH, and COM.LOTUS.V98.FOO.

Evaluators should avoid defining evaluator-unique functions beginning with a top-level domain name followed by a period. Evaluators should avoid defining application functions beginning with “NET.”, “COM.”, “ORG.”, or a country code followed by a period.

Evaluators that do not support a function should compute its result as some Error value other than #N/A.

5.8 References

References refer to a specific cell or set of cells. The syntax for a constant reference:

```
Reference ::= '[' (Source? RangeAddress) | ReferenceError ']'
RangeAddress ::=
 SheetLocatorOrEmpty '.' Column Row (':' '.' Column Row )? |
 SheetLocatorOrEmpty '.' Column ':' '.' Column |
 SheetLocatorOrEmpty '.' Row ':' '.' Row |
 SheetLocator '.' Column Row ':' SheetLocator '.' Column Row |
 SheetLocator '.' Column ':' SheetLocator '.' Column |
 SheetLocator '.' Row ':' SheetLocator '.' Row
SheetLocatorOrEmpty ::= SheetLocator | /* empty */
SheetLocator ::= SheetName ('.' SubtableCell)*
SheetName ::= QuotedSheetName | '$'? [^\]\. #$']+
QuotedSheetName ::= '$'? SingleQuoted
SubtableCell ::= ( Column Row ) | QuotedSheetName
ReferenceError ::= "#REF!"
Column ::= '$'? [A-Z]+
Row ::= '$'? [1-9] [0-9]*
Source ::= "'" IRI "'" "#"
CellAddress ::= SheetLocatorOrEmpty '.' Column Row /* Not used
directly */
```

References always begin with '[' (LEFT SQUARE BRACKET, U+005B); this disambiguates cell addresses from function names and named expressions. `SheetName`s include single-quote“'” (APOSTROPHE, U+0027) characters by doubling them and having the entire name surrounded by single-quotes. Column labels shall be in uppercase. The syntax supports whole-row and whole-column references. A reference is of type Reference.

A ReferenceError provides information that a formula evaluates to an Error because of a particular reference having been invalidated by actions that occurred after the formula was validly created.

Columns are named by a sequence of one or more uppercase letters A-Z (U+0041 through U+005A). Columns are named A, B, C, ... X, Y, Z, AA, AB, AC, ... AY, AZ, BA, BB, BC, ... ZX, ZY, ZZ, AAA, AAB, AAC, AAZ, ABA, ABB, and so on.

If a `RangeAddress` does not contain a `Column` element or does not contain a `Row` element, it specifies a cell rectangle (4.8 Reference). If it contains `Row` elements, the cell rectangle starts on the first column and ends on the last column the evaluator supports. If it contains `Column` elements, the cell rectangle starts on the first row and ends on the last row the evaluator supports.

If in a `RangeAddress` the first part (left of ':' colon) contains a `SheetLocator` and the second part (right of ':' colon) does not contain a `SheetLocator`, the second part inherits the `SheetLocator` from the first part.

If a `RangeAddress` contains two different `SheetLocator`s, it specifies a cell cuboid (4.8 Reference).

If a `RangeAddress` contains no SheetLocator, the current sheet local to the position where the expression is evaluated is referred.

A reference with an explicit row or column value beyond the capabilities of an evaluator shall be computed as an Error, and not as a reference.

Note that references can include a single embedded “:” separator. Evaluators should use references with embedded “:” separators inside the [..] markers, instead of the general-purpose “:” operator, when saving files, and, where there is a choice of cells to join, evaluators should choose the leftmost pair.

The optional Source expresses that the reference is to sheets and/or cells in a different location (possibly in a same-document fragment) from that for the formula in which the reference occurs. The optional Source is also used for locating Named Expressions (section 5.11).

The IRI portion of Source shall be an IRI reference [RFC3987] conforming to the general syntax IRI-reference rule (section 2.2 of [RFC3987]) after each pair of consecutive single-quote characters (APOSTROPHE, U+0027) is replaced by one single single-quote character.

> **Note**: The escaping of single-quotes as paired single-quotes is because the IRI is enclosed in single quote characters of the Source.

Resolution of the [RFC3987] IRI reference is host-defined behavior. 3.4

5.9 Reference List

A reference list is the result of the Infix Operator Reference Concatenation 6.4.13 '~', the syntax is:

```
ReferenceList ::= Reference (Whitespace* ReferenceConcatenationOp
Whitespace* Reference)*
```

A reference list can be passed as an argument to functions expecting a reference parameter where passing one reference results in an identical computation as an arbitrary sequence of single references occupying the identical cell range. A reference list cannot be converted to an array.

5.10 Quoted Label

5.10.1 General

A quoted label is Text contained in a table as cell content, either literally or as a formula result.

```
QuotedLabel ::= SingleQuoted
```

A quoted label identifies a column or a row, depending on the label range in which its text appears.

5.10.2 Lookup of Defined Labels

For a `QuotedLabel`, first the cells defined in column label ranges (cell ranges of the `table:label-cell-range-address` attribute (OpenDocument Part 3, 19.660) in the elements `<table:label-range>` (OpenDocument Part 3, 9.4.9) with attribute `table:orientation` (OpenDocument Part 3, 19.690.4) set to `column`) are searched for the string content of `QuotedLabel` (without the quotes). If found, the corresponding column's range of the data cell range of the `table:data-cell-range-address` attribute (OpenDocument Part 3, 19.612) is taken as a reference. If not found, the cells defined in row label ranges (attribute `table:orientation` set to `row`) are searched and if found the corresponding row's range of the data cell range is taken. Label ranges of the current formula's sheet take precedence over label ranges of other sheets if a name occurs in both.

5.10.3 Automatic Lookup of Labels

For a `QuotedLabel` where no defined label is found, an automatic lookup is performed on the sheet where the formula cell resides, if that document setting is enabled (HOST-AUTOMATIC-FIND-LABELS value true).

Matches to the upper left of the formula cell are preferred over other matches, followed by matches with a smaller distance. The following algorithm is used:

Cells on the same sheet as the formula cell are examined **column-wise** from left to right whether they contain the text of `QuotedLabel` (without the quotes). If more than one cell match, the distance and direction from the formula cell's position is taken into account. The distance is calculated by `Distance= ColumnDifference*ColumnDifference+`

`RowDifference*Row Difference` using an idealized layout of square cells. For the direction, during the run two independent match positions are remembered each time Distance is smaller than a previous Distance: *Match2* for positions right of and/or below the formula position (FormulaColumn < MatchColumn || FormulaRow < MatchRow), *Match1* for all others (not right of and not below). Match1 also holds the very first match, in case there is only one match or all matches are somewhere below or right of the formula cell. After having found the smallest distances the conditions are:

1. If Match1 has the smallest distance, that match is taken.
2. Else, Match2 (right and/or below) has the smallest or an equal distance:

    2.1 A match to the upper left (FormulaColumn >= Match1Column && FormulaRow >= Match1Row) takes precedence over matches to other directions.

    2.2 Else, if there is no match to the upper left:

        2.2.1 If Match1 is somewhere right of the formula cell (FormulaColumn < Match1Column) it was the first match found in column-wise lookup.

            2.2.1.1 If Match2 is above the formula cell (FormulaRow >= Match2Row) it is to the upper right of the formula cell and either nearer than Match1 or Match1 is below. Match2 is taken.

            2.2.1.2 Else Match2 is below the formula cell and Match1 is taken.

        2.2.2 Else (Match1 not right of the formula cell => two matches below or below and right) the match with the smallest distance is taken.

If the resulting cell is below or above another cell containing Text a row label is assumed, else a column label is assumed.

> **Note:** Use of automatically looked up column or row labels in expressions impairs interoperability.

5.10.4 Implicit Intersection

For the reference resulting from a single `QuotedLabel` an implicit intersection is generated if the operator or function with which it is used expects a scalar value. The intersection is generated with the current formula's cell position, thus for a column label an intersection is generated with the formula cell's row, for a row label with the formula cell's column.

5.10.5 Automatic Range

When passed as a non-scalar argument (e.g. Array or NumberSequence) to a function, an automatically-looked-up column or row label (not defined label range) is converted to an automatic range reference that is adjusted each time the formula is interpreted. The range is generated from the column below a column label, or the row to the right of a row label, constructed by encompassing contiguous non-empty cells. An empty cell interrupts contiguousness, one empty cell directly below a column label cell or to the right of a row label cell is ignored.

Example:

Table 2 - Automatic Range

Row	Data	Expression	Result	Comment
1	Label	=SUM('Label')	3	Empty cell in row 2 is skipped (two empty cells in row 2 and 3 would not be skipped and would stop the automatic range), empty cell in row 5 stops the automatic range.
2				
3	1			
4	2			
5				
6	8			
7				
8	32			

If any cell content is entered in row 5 the range is regenerated as follows:

Table 3 - Automatic Range

Row	Data	Expression	Result	Comment
1	Label	=SUM('Label')	15	Empty cell in row 2 is skipped, empty cell in row 7 stops the automatic range.
2				
3	1			
4	2			
5	4			
6	8			
7				
8	32			

5.10.6 Automatic Intersection

An automatic intersection may be used to identify the intersection of two quoted labels. Note that this is different from the `IntersectionOp`, which takes two references instead of two labels:

```
AutomaticIntersection ::= QuotedLabel Whitespace* '!!' Whitespace* QuotedLabel
```

In an automatic intersection, one of the labels identifies a row, the other a column; they may be in either order. Each `QuotedLabel` is looked up as defined above under "Lookup of Defined Labels" and "Automatic Lookup of Labels". If two data cell ranges are found, the intersection of the column's data cell range and the row's data cell range is generated. If the intersection result is not exactly one cell, an Error is generated.

5.11 Named Expressions

A NamedExpression references another expression, possibly in a completely different spreadsheet or any other type of document that can be imported into a spreadsheet.

```
NamedExpression ::= SimpleNamedExpression |
      SheetLocalNamedExpression | ExternalNamedExpression
```



```
SimpleNamedExpression ::= Identifier |
      '$$' (Identifier | SingleQuoted)
SheetLocalNamedExpression ::=
      QuotedSheetName '.' SimpleNamedExpression
ExternalNamedExpression ::=
      Source (SimpleNamedExpression | SheetLocalNamedExpression)
```

Evaluators supporting named expressions shall support Simple Named Expressions that are global to all the sheets in a (spreadsheet) document in the current document. This is a named expression without a `Source`, `QuotedSheetName`, or `SubtableCell`. The type of a named expression is the type of the value that the named expression returns.

Named expressions are case-consistent, meaning that matching is done case-insensitive and identifiers can not differ ONLY in their case. Evaluators should write identifiers with identical case in all locations.

Evaluators may support Sheet-local Named Expressions that are local (attached) to individual sheets. In that case, a non-empty `QuotedSheetName` can be used to reference a sheet-specific named expression. The most specific named expression for a given expression is used. If the `QuotedSheetName` is empty, the search for the named expression begins with the current sheet, then up through the container(s) of the sheet (the same is true if the QuotedSheetName rule fragment is not included at all). If there is a non-empty `QuotedSheetName`, search begins with that named sheet, then up through its container(s) for the given name.

Note: There is no syntax for referencing a named expression without first looking at the current sheet's named expressions; where this is a problem, a user can define a blank sheet and reference that sheet as the starting location for finding the named expression.

If a sheetname is not empty, it shall be quoted using the single-quote character “'” (APOSTROPHE, U+0027). While both `Source` and `QuotedSheetName` can begin with the single-quote character, they are distinguished: after the closing single-quote character, a non-empty source shall have the '#' (NUMBER SIGN, U+0023) character as the next non-whitespace character; a non-empty sheetname shall be followed by the '.' (FULL STOP, U+002E) character as the next non-whitespace character.

Expressions should limit the names of their identifiers to only ([UNICODE]) letters, underscores, and digits, not including patterns that look like cell references or the words True or False.

```
Identifier ::= ( LetterXML
     (LetterXML | DigitXML | '_' | CombiningCharXML)* )
   - ( [A-Za-z]+[0-9]+ )
   - ([Tt][Rr][Uu][Ee]) - ([Ff][Aa][Ll][Ss][Ee])
```

5.12 Constant Errors

Evaluators shall support the Error named #N/A. Evaluators may support other Errors. Evaluators may allow entry of errors directly, parse them and recognize them as Errors. Functions shall propagate Errors unless stated otherwise.

Error names shall have the following syntax:

```
Error ::= '#' [A-Z0-9]+ ([!?] | ('/' ([A-Z] | ([0-9] [!?]))))
```

Specific Errors are indicated by their corresponding names.

Table 4 is a list of Errors that are used by several existing implementations.

Table 4 - Possible Other Constant Error Names

Name	***Comments***
#DIV/0!	Attempt to divide by zero, including division by an empty cell. ERROR.TYPE of 2 6.13.11

#NAME?	Unrecognized/deleted name. ERROR.TYPE of 5.
#N/A	Not available. ISNA() applied to this value will return TRUE. Lookup functions which failed, and NA(), return this value. ERROR.TYPE of 7.
#NULL!	Intersection of ranges produced zero cells. ERROR.TYPE of 1.
#NUM!	Failed to meet domain constraints (e.g., input was too large or too small). ERROR.-TYPE of 6.
#REF!	Reference to invalid cell (e.g., beyond the application's abilities). ERROR.TYPE of 4.
#VALUE!	Parameter is wrong type. ERROR.TYPE of 3.

Evaluators may implement other Errors.

An unknown Error name shall be mapped into an Error supported by the evaluator when read (e.g., the application's equivalent of #NAME?), though an evaluator may warn the user if this has or will take place. It is desirable to preserve the original specific Error name when writing an Error back out, where possible, but for Errors other than #N/A evaluators may write a different Error for a formula than they did when reading it. Whitespace shall not be included in an Error name.

Evaluators should use a human-comprehensible name, not a numeric id, for Error names they write.

5.13 Inline Arrays

Inline arrays are enclosed with curly braces. Inside, they contain one or more rows, with each row separated by a row separator:

```
Array ::= '{' MatrixRow ( RowSeparator MatrixRow )* '}'
MatrixRow ::= Expression ( ';' Expression )*
RowSeparator ::= '|'
```

Evaluators that support inline arrays shall accept a matrix with one or more rows, each with one or more columns, with the same number of columns in each row, with constant values for each expression. Evaluators that do not support inline arrays, or cannot support a particular use permitted by this syntax, should compute an Error value for such arrays. An inline array is of type Array.

> **Note:** Expression authors should be aware that use of `Expression` other than constant `Number` or constant `String` may impair interoperability.

5.14 Whitespace

```
Whitespace ::= #x20 | #x09 | #x0a | #x0d
```

For calculation purposes, whitespace is ignored unless it is inside the contents of string constants or text surrounded by single quotes. Evaluators shall ignore any whitespace characters before and/or after any operators, constant numbers, constant strings, constant errors, inline arrays, parentheses used for controlling precedence, and the closing parenthesis of a function call. Whitespace shall be ignored following the initial equal sign(s). Whitespace shall be ignored just before a function name, but whitespace shall not separate a function name from its initial opening parenthesis. Whitespace shall not be used in the interior of a terminating grammar rule (a rule that references no other rule other than character sets, internally or externally-defined), unless specifically permitted by the terminating grammar rule, since these rules define the lexical properties of a component. Evaluators shall not write formulas with whitespace embedded in any unquoted identifier, constant Number, or constant Error. Evaluators shall treat SPACE (U+0020), CHARACTER TABULATION (U+0009), LINE FEED (U+000A), and CARRIAGE RETURN (U+000D) as whitespace characters.

An embedded line break shall be represented by a single LINE FEED character (U+000A), *not* by a carriage return-linefeed pair. When embedded in an XML attribute the linefeed character is represented as "
".

Evaluators should retain whitespace entered by the original formula creator and use it when saving or presenting the formula, and should not add additional whitespace unless directed to do so during the process of editing a formula.

6 Standard Operators and Functions

6.1 General

OpenFormula defines commonly used operators and functions.

Function names ignore case. Evaluators should write function names in all uppercase letters when writing OpenFormula formulas.

Unless otherwise noted, if any value being provided is an Error, the result is an Error; if more than one Error is provided, one of them is returned (evaluators should return the leftmost Error result).

6.2 Common Template for Functions and Operators

For every function or operator, the following are defined in this specification:

- **Name:** The function/operator name.
- **Summary:** One sentence briefly describing the function or operator.
- **Syntax:**
  - Parameter names are shown in order, with each parameter prefixed by the type or pseudotype of that parameter. If the type has multiple names separated by “|”, then any of those types are permitted.
  - A { ... } indicates a list of zero or more parameters, separated by the function parameter separator character.
  - A { ... } followed by a superscripted + indicates a list of one or more parameters, separated by the function parameter separator character.
  - Components surrounded by [...] are optional. Optional components may be omitted.
  - An optional parameter followed by the = symbol has the default value given after the equal sign.
  - Parameters are separated with a semicolon (";"), as per the OpenFormula expression syntax 5.6.

    When a function is given a value of a different type, the parameters are first converted using the implicit conversion rules before the function operates on its parameters.

Evaluators may extend functions by permitting fewer or additional parameters, which documents may use. Extended functions may result in a lack of interoperability.

- **Returns:** Return type (e.g., Number, Text, Logical, Reference).
- **Constraints:** A description of constraints, in addition to the constraints imposed by the parameter types. If there are no additional constraints beyond those imposed by the parameter types, this is "None". If a constraint is not met, the function/operator shall return an Error unless otherwise noted.
- **Semantics:** This text describes what the function/operator does.

  If a parameter is a pseudotype, but the provided value fails to meet the requirements for that type, the behavior is implementation-defined.

  > **Note:** Functions and operators are defined by mathematical formulas or by an OpenFormula formula. Formulas define the *correct result,* and *not* the algorithm for calculation. Since computing systems have limited precision and range of numbers, some functions *cannot* or *should not* be

naively implemented as their formulas suggest. This specification defines the mathematically correct answer, and allows implementors to choose the best algorithm that will meet that definition.

- **Comment:** Explanatory comment.
- **See also** A list of related operators and functions.

The implicit conversion operators omit many of these items, e.g., the syntax (since there is none).

6.3 Implicit Conversion Operators

6.3.1 General

Any given function or operand takes 0 or more parameters, and each of those parameters has an *expected type*. The expected type can be one of the base types, identified above. It can also be of some *conversion type* that controls conversion, e.g., *Any* means that no conversion is done (it can be of any type); *NumberSequence* causes a conversion to an ordered sequence of zero or more numbers. If the passed-in type does not match the expected type, an attempt is made to automatically convert the value to the expected type. An Error is returned if the type cannot be converted (this can never happen if the expected type is Any). Unless otherwise noted, any conversion operation applied to a value of type Error returns the same value.

6.3.2 Conversion to Scalar

To convert to a scalar, if the value is of type:

- Number, Logical, or Text, return the value.
- reference to a single cell: obtain the value of the referenced cell, and return that value.
- reference to more than one cell: do an implied intersection, Error: Reference source not found, to determine which single cell to use, then handle as a reference to a single cell.

6.3.3 Implied intersection

In some cases a reference to a single cell is needed, but a reference to multiple cells is provided. In this case an "implied intersection" is performed. To perform an implied intersection:

- Compute the union of cells contained in the current row and current column of the formula being computed.
- Intersect this with the provided reference to multiple cells
- If a single cell is referenced; return it; otherwise, return an Error.

6.3.4 Force to array context (ForceArray)

A *ForceArray* attribute forces calculation of the argument's expression into non-scalar array mode. This means that no implied intersection is performed, instead where a reference to a single cell is expected and multiple cells are provided, iteration over the multiple cells is performed and results are stored in an array that is passed on.

See also Non-Scalar Evaluation 3.3

6.3.5 Conversion to Number

If the expected type is Number, then if the value is of type:

- Number, return it.
- Logical, return 0 if FALSE, 1 if TRUE.

- Text: The specific conversion is implementation-defined; an evaluator may return 0, an Error value, or the results of its attempt to convert the Text value to a Number (and fall back to 0 or Error if it fails to do so). Evaluators may apply VALUE 6.13.34 or some other function to do this conversion, should they choose to do so. Conversion depends on the actual locale the application runs in, especially if group or decimal separators are involved.
- Reference: If the reference covers more than one cell, do an implied intersection to determine which cell to use. Then obtain the value of the single cell and perform the rules as above. If the calculation setting “precision-as-shown” is true, then convert the number to the closest possible representation of the displayed number. If the cell is empty (blank), use 0 (zero) as the value. Evaluators may choose to convert references to Text in a different manner than they handle converting embedded Text to a Number.

6.3.6 Conversion to Integer

If the expected type is Integer for a function or operator, apply the “Conversion to Number” operation. Error: Reference source not found Then, if the result is a Number but not an integer, apply the specific conversion from Number to integer specified by that particular function/operator. If the function or operator does not specify any particular conversion operation, then the conversion from a non-integer Number into an integer is implementation-defined.

Many different conversions from a non-integer number into an integer are possible. The conversion direction may be towards negative infinity, towards positive infinity, towards zero, away from zero, towards the nearest even number, or towards the nearest odd number. A conversion can select the nearest integer, the nearest even or odd integer, or the “next” integer in the given direction if it is not already an integer. If a conversion selects the nearest integer, a direction is still needed (for when a value is halfway between two integers). In this specification, this conversion is referred to as “rounding” or “truncation”; these terms by themselves do not specify any specific operation.

If a function specifies its rounding operation using a series of capital letters, the function defined in this specification for that function is used to do the conversion to integer. Common such functions are:

- INT, which if given non-integer rounds down to the next integer towards negative infinity, regardless of whether or not it is the closest integer.
- ROUND, which if given non-integer rounds to the nearest integer. If the input number is halfway between integers, it rounds away from zero.
- TRUNC, which if given non-integer rounds towards zero, regardless of whether or not that integer is the closest integer.

6.3.7 Conversion to NumberSequence

If the expected type is NumberSequence, then if value is of type:

- Number, Text, or Logical, handle as Conversion to Number 6.3.5 (creating a sequence of length 1).
- reference, create a sequence of numbers from the values of the referenced cells that *only* includes the values of type Number or Error. Thus, Empty cells and Text that *could* be converted into a value are *not* included in a number sequence. If the Logical type is a distinguished type from the Number type, it should not be included in the sequence of numbers; if the Logical type is not a distinguished type, then such values will (of course) be included in the number sequence.

6.3.8 Conversion to NumberSequenceList

Identical to Conversion to NumberSequence 6.3.7, with the addition that instead of a *Reference* also a *ReferenceList* is accepted as argument. Each *Reference* in the list is converted to a *NumberSequence* in the order of occurrence.

6.3.9 Conversion to DateSequence

Identical to Conversion to NumberSequence 6.3.7 except that each element in the list represents a serial date value of subtype Date.

6.3.10 Conversion to Complex Number

An evaluator may accept complex numbers as Text, Number, or a different distinguishable type.

If the value is:

- Number that is not complex, use the Number with 0 as the imaginary part.
- Text, attempt to convert to complex number using VALUE 6.13.34. If it is a number that is not complex, use it. If the text matches one of these patterns, accept it:

  ```
  ([+-]?Number [+-])?Number[ij]
  ```

  ```
  [+-]?Number[ij]
  ```

- Logical, convert to Number and then handle as Number.
- reference: Convert to Scalar 6.3.2, then use the rules above. If the reference is to an empty cell, consider it equal to 0.

6.3.11 Conversion to ComplexSequence

If the expected type is ComplexSequence, then if value is of type:

- Number, Text, or Logical, handle as Conversion to Complex Number (creating a sequence of length 1).
- Reference, create a sequence of complex numbers from the values of the referenced cells that only includes the values of type Number, Text, and Error. Empty cells are not included in a complex number sequence. If the Logical type is a distinguished type from the Number type, it should not be included in the sequence of numbers; if the Logical type is not a distinguished type, then such values will (of course) be included in the number sequence.

6.3.12 Conversion to Logical

If the expected type is Logical, then if value is of type:

- Number, return TRUE for nonzero and FALSE for 0.
- Text: The specific conversion is implementation-defined; an evaluator may return FALSE, an Error value, or the results of its attempt to convert the Text value (ignoring case) to a Logical value (and fall back to FALSE or Error if it fails to do so). Conversion depends on the actual locale the evaluator runs in.
- Logical, return it.
- Reference, convert to scalar and then perform as above. If the reference is to an empty cell, consider it FALSE.

6.3.13 Conversion to LogicalSequence

If the expected type is LogicalSequence, then if value is of type:

- Number or Logical, handle as Conversion to Logical (creating a sequence of length 1).
- Reference, create a sequence of logical values from the values of the referenced cells that only includes the values of type Logical and Error. If the Logical type is not a distinguished type, then include values of type Number, converting each to a Logical value as described in Conversion to Logical. Empty cells are not included in a LogicalSequence.

6.3.14 Conversion to Text

If the expected type is Text, then if value is of type:

- Number, transform into Text (with no whitespace).
- Text, return it.
- Logical, return "TRUE" if it is true and "FALSE" if it is false.
- Reference: perform conversion to scalar. If the referenced cell is empty, treat as an empty string (a text value with length 0). Then perform as above.

6.3.15 Conversion to DateParam

If the expected type is the pseudotype DateParam, then if value is of type:

- Number, return it.
- Text, pass to DATEVALUE 6.10.4, and if non-Error, return it. If DATEVALUE would return an Error, an evaluator may attempt to convert to a Number in other ways (such as by calling VALUE 6.13.34); this is implementation-defined. If the evaluator cannot convert to Number, it returns an Error.
- Logical, the result is implementation-defined, either a Number or Error
- Reference: perform conversion to scalar, then perform as above. If the cell is empty, return 0.

6.3.16 Conversion to TimeParam

If the expected type is the pseudotype TimeParam, then if value is of type:

- Number, return it.
- Text, pass to TIMEVALUE 6.10.19, and if non-Error, return it. If TIMEVALUE would return an Error, an evaluator may attempt to convert to a Number in other ways (such as by calling VALUE 6.13.34); this is implementation-defined. If the evaluator cannot convert to Number, it returns an Error.
- Logical, the result is implementation-defined, either a Number or Error
- Reference: perform conversion to scalar, then perform as above. If the cell is empty, return 0.

6.4 Standard Operators

6.4.1 General

The functions defined under *standard operators* differ from other functions only by their frequency of use. That frequency of use has lead to the colloquial terminology, standard operators.

6.4.2 Infix Operator "+"

Summary: Add two numbers.

Syntax: *Number* ***Left*** + *Number* ***Right***

Returns: *Number*

Constraints: None

Semantics: Adds numbers together.

See also Infix Operator "-" 6.4.3, Prefix Operator "+" 6.4.15

6.4.3 Infix Operator "-"

Summary: Subtract the second number from the first.

Syntax: *Number **Left*** - *Number **Right***

Returns: *Number*

Constraints: None

Semantics: Subtracts one number from another number.

See also Infix Operator "+" 6.4.2, Prefix Operator "-" 6.4.16

6.4.4 Infix Operator "*"

Summary: Multiply two numbers.

Syntax: *Number **Left*** * *Number **Right***

Returns: *Number*

Constraints: None

Semantics: Multiplies numbers together.

See also Infix Operator "+" 6.4.2, Infix Operator "/" 6.4.5

6.4.5 Infix Operator "/"

Summary: Divide the first number by the second.

Syntax: *Number **Left*** / *Number **Right***

Returns: *Number*

Constraints: None

Semantics: Divides numbers. Dividing by zero returns an Error.

See also Infix Operator "-" 6.4.3, Infix Operator "*" 6.4.4

6.4.6 Infix Operator "^"

Summary: Exponentiation (Power).

Syntax: *Number **Left*** ^ *Number **Right***

Returns: *Number*

Constraints: NOT(AND(***Left***=0; ***Right***=0)); Evaluators may evaluate expressions where OR(***Left*** != 0; ***Right*** != 0) evaluates to a non-Error value.

Semantics: Returns POWER(***Left***, ***Right***).

See also Infix Operator "*" 6.4.4, AND 6.15.2, NOT 6.15.7, POWER 6.16.46

6.4.7 Infix Operator "="

Summary: Report if two values are equal

Syntax: *Scalar **Left*** = *Scalar **Right***

Returns: *Logical*

Constraints: None

Semantics: Returns TRUE if two values are equal. If the values differ in type, return FALSE. If the values are both Number, return TRUE if they are considered equal, else return FALSE. If they are both Text, return TRUE if the two values match, else return FALSE. For Text values, if the calculation setting `HOST-CASE-SENSITIVE` is `false`, text is compared but characters differencing only in case are considered equal. If they are both Logicals, return TRUE if they

are identical, else return FALSE. Error values *cannot* be compared to a constant Error value to determine if that is the same Error value.

Evaluators may approximate and test equality of two numeric values with an accuracy of the magnitude of the given values scaled by the number of available bits in the mantissa, ignoring some least significant bits and thus providing compensation for not exactly representable values.

The result of “1=TRUE()” is FALSE for evaluators that implement a distinct Logical type and TRUE if they don't.

See also Infix Operator "<>" 6.4.8

6.4.8 Infix Operator "<>"

Summary: Report if two values are not equal

Syntax: *Any **Left*** <> *Any **Right***

Returns: *Logical*

Constraints: None

Semantics: Returns NOT(***Left*** = ***Right***) if ***Left*** and ***Right*** are not Error. For Text values, if the calculation setting `HOST-CASE-SENSITIVE` is `false`, text is compared but characters differencing only in case are considered equal.

If either ***Left*** and ***Right*** are an Error, the result is an Error; this operator cannot be used to determine if two Errors are the same kind of Error.

Note: In some user interfaces the infix operator “<>” is displayed (or accepted) as “!=” or “≠”.

See also Infix Operator "=" 6.4.7, NOT 6.15.7

6.4.9 Infix Operator Ordered Comparison ("<", "<=", ">", ">=")

Summary: Report if two values have the given order

Syntax: *Scalar **Left** op Scalar **Right***

where *op* is one of: "<", "<=", ">", or ">="

Returns: *Logical*

Constraints: None

Semantics: Returns TRUE if the two values are less than, less than or equal, greater than, or greater than or equal (respectively). If both ***Left*** and ***Right*** are Numbers, compare them as numbers. If both ***Left*** and ***Right*** are Text, compare them as text; if the calculation setting `HOST-CASE-SENSITIVE` is `false`, text is compared but characters are compared ignoring case. If the values are both Logical, convert both to Number and then compare as Number.

These functions return one of TRUE, FALSE, or an Error if ***Left*** and ***Right*** have different types, but it is implementation-defined which of these results will be returned when the types differ.

See also Infix Operator "<>" 6.4.8, Infix Operator "=" 6.4.7

6.4.10 Infix Operator "&"

Summary: Concatenate two strings.

Syntax: *Text **Left*** & *Text **Right***

Returns: *Text*

Constraints: None

Semantics: Concatenates two text (string) values.

Note: The infix operator “&” is equivalent to CONCATENATE(***Left***,***Right***).

See also Infix Operator "+" 6.4.2, CONCATENATE 6.20.6

6.4.11 Infix Operator Reference Range (":")

Summary: Computes an inclusive range given two references

Syntax: *Reference* ***Left*** : *Reference* ***Right***

Returns: *Reference*

Constraints: None

Semantics: Takes two references and computes the range, that is, a reference to the smallest 3-dimensional cube of cells that include both ***Left*** and ***Right*** including the cells on sheets positioned between ***Left*** and ***Right***. ***Left*** and ***Right*** need not be a single cell. For an expression such as [.B4:.B5]:[.C5] the resulting range is B4:C5. In case ***Left*** and/or ***Right*** involve a reference list (result of operator reference union), the range is computed and extended for each element of the list(s).

> **Note:** For example, (a,b,c,d denoting one reference each) (a~b):(c~d) computes a:b:c:d determining the outermost front-top-left and rear-bottom-right corners.

Left and ***Right*** may also be defined names or the result of a function returning a reference, such as INDIRECT.

See also Infix Operator Reference Union 6.4.13, Infix Operator Reference Intersection 6.4.12, INDIRECT 6.14.7

6.4.12 Infix Operator Reference Intersection ("!")

Summary: Compute the intersection of two references

Syntax: *Reference* ***Left*** ! *Reference* ***Right***

Returns: *Reference*

Constraints: None

Semantics: Takes two references and computes the intersection - a reference to the intersection of cells in both ***Left*** and ***Right***. If there are no cells in common, returns an Error.

If ***Left*** or ***Right*** are not of type Reference or ReferenceList, an Error shall be returned.

If ***Left*** and/or ***Right*** are reference lists (result of infix operator reference concatenation), the intersection is computed for each combination of ***Left*** and ***Right***, producing a reference list of intersections.

> **Note 1:** For example (a,b,c,d denoting one reference each):
> (a~b)!(c~d) will compute (a!c)~(a!d)~(b!c)~(b!d)

If for a resulting intersection there are no cells in common, the element is ignored and omitted from the result list. If for all intersections there are no cells in common and the result list is empty, an Error is returned.

Note 2: Intersection is notated as "!" in OpenFormula format, but as a space character in some user interfaces.

See also Infix Operator Reference Union 6.4.13

6.4.13 Infix Operator Reference Concatenation ("~") (aka Union)

Summary: Concatenate two references

Syntax: *Reference* ***Left*** ~ *Reference* ***Right***

Returns: *ReferenceList*

Constraints: None

Semantics: Takes two references and computes the "cell union", which is a concatenation of the reference ***Left*** followed by the reference ***Right***. This is *not* the same as a union in set theory; duplicate references to cells are *not* removed. The resulting reference will have the number of areas, as reported by AREAS, as AREAS(***Left***)+AREAS(***Right***).

Note: Concatenation is notated as "~" in OpenFormula format, but as a comma or “+” in some user interfaces.

If ***Left*** or ***Right*** are not of type Reference or ReferenceList, an Error shall be returned.

Test Cases:

See also Infix Operator Reference Range 6.4.11, Infix Operator Reference Intersection 6.4.12, AREAS 6.13.2

6.4.14 Postfix Operator "%"

Summary: Divide the operand by 100

Syntax: *Number* ***Left*** %

Returns: *Number*

Constraints: None

Semantics: Computes ***Left*** / 100.

See also Prefix Operator "-" 6.4.16, Prefix Operator "+" 6.4.15

6.4.15 Prefix Operator "+"

Summary: No operation; returns its one argument.

Syntax: + *Any* ***Right***

Returns: *Any*

Constraints: None

Semantics: Returns the value given to it. Note that this does **not** convert a value to the Number type. In fact, it does *no* conversion at all of a Number, Logical, or Text value - it returns the same Number, Logical, or Text value (respectively). The "+" applied to a reference may return the reference, or an Error.

See also Infix Operator "+" 6.4.2

6.4.16 Prefix Operator "-"

Summary: Negate its one argument.

Syntax: - *Number* ***Right***

Returns: *Number*

Constraints: None

Semantics: Computes 0 - ***Right***.

See also Infix Operator "-" 6.4.3

6.5 Matrix Functions

6.5.1 General

Matrix functions operate on matrices.

A matrix with *M* rows and *N* columns is defined by

$$A_{M\times N}=\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{M1} & a_{M2} & \dots & a_{MN} \end{pmatrix}$$

The dimension subscript may be omitted, if the context allows it, i.e. $A_{M\times N}=A$. Matrices are represented by upper-case letters. The elements of a matrix are denoted by the corresponding lower case letter and subscripts, which defines the row and column number.

Square matrices have the same number of rows and columns, i.e. $M=N$.

6.5.2 MDETERM

Summary: Calculates the determinant of a matrix.

Syntax: MDETERM(*ForceArray Array* ***A***)

Returns: *Number*

Constraints: Only square matrices are allowed.

Semantics: Returns the determinant of matrix ***A***. The determinant is defined by

$$det\left(A_{N\times N}\right)=\sum_{P} sgn(P)\prod_{i=1}^{N} a_{ip_i}$$

where *P* denotes a permutation of the numbers 1, 2, ..., n and $sgn(P)$ is the sign of the permutation, which is +1 for an even amount of permutations (i.e., permutations that can be written as the composition of an even number of transpositions), -1 otherwise. ***A*** transposition on 1, ..., n is a permutation of 1, ..., n with exactly (n - 2) numbers fixed.

See also MINVERSE 6.5.3

6.5.3 MINVERSE

Summary: Returns the inverse of a matrix.

Syntax: MINVERSE(*ForceArray Array* ***A***)

Returns: *Array*

Constraints: Only square matrices are allowed.

Semantics: Calculates the inverse A^{-1} of matrix ***A***. The matrix ***A*** multiplied with its inverse A^{-1} results in the unity matrix of the same dimension as ***A***:

$$A_{N\times N}\,A^{-1}_{N\times N}=A^{-1}_{N\times N}\,A_{N\times N}=\mathbf{1}_{N\times N}$$

Invertible matrices have a non-zero determinant. If the matrix is not invertible, this function should return an Error value.

See also MDETERM 6.5.2

6.5.4 MMULT

Summary: Multiplies the matrices ***A*** and ***B***.

Syntax: MMULT(*ForceArray Array* ***A*** ; *ForceArray Array* ***B***)

Returns: *Array*

Constraints: COLUMNS(***A***) = ROWS(***B***)

Semantics: Returns the matrix product of the two matrices. The elements c_{mn} of the resulting matrix $C_{M\times N} = A_{M\times K}\, B_{K\times N}$, are defined by:

$$c_{mn} = \sum_{k=1}^{K} a_{m\,k}\, b_{k\,n}$$

See also COLUMNS 6.13.5, ROWS 6.13.30

6.5.5 MUNIT

Summary: Creates a unit matrix of a specified dimension ***N***.

Syntax: MUNIT(*Integer* ***N***)

Returns: *Array*

Constraints: The dimension has to be greater than zero.

Semantics: Creates the unit matrix (identity matrix) of dimension ***N***.

$$\mathbf{1}_{N\times N} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

6.5.6 TRANSPOSE

Summary: Returns the transpose of a matrix.

Syntax: TRANSPOSE(*Array* ***A***)

Returns: *Array*

Constraints: None

Semantics: Returns the transpose $\boldsymbol{A}^T$ of a matrix ***A***, i.e. rows and columns of the matrix are exchanged.

$$A^{T}_{M\times N} = \begin{pmatrix} a_{11} & a_{21} & \dots & a_{M1} \\ a_{12} & a_{22} & \dots & a_{M2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1N} & a_{2N} & \dots & a_{MN} \end{pmatrix}_{N\times M}$$

6.6 Bit operation functions

6.6.1 General

Evaluators shall support unsigned integer values and results of at least 48 bits (values from 0 to 2^48-1 inclusive). Operations that receive or result in a value that cannot be represented within 48 bits are implementation-defined.

6.6.2 BITAND

Summary: Returns bitwise “and” of its parameters

Syntax: BITAND(*Integer* ***X*** ; *Integer* ***Y***)

Returns: *Number*

Constraints: ***X*** ≥ 0, ***Y*** ≥ 0

Semantics: Returns bitwise “and” of its parameters. In the result, each bit position is 1 if and only if all parameters' bits at that position are also 1; else it is 0.

See also BITOR 6.6.4, BITXOR 6.6.6, AND 6.15.2

6.6.3 BITLSHIFT

Summary: Returns left shift of value ***X*** by ***N*** bits (“<<”)

Syntax: BITLSHIFT(*Integer* ***X*** ; *Integer* ***N***)

Returns: *Number*

Constraints: ***X*** ≥ 0

Semantics: Returns left shift of value ***X*** by ***N*** bit positions:

- If ***N*** < 0, return BITRSHIFT(***X***,-***N***)
- if ***N*** = 0, return ***X***
- if ***N*** > 0, return ***X*** * 2^***N***

See also BITAND 6.6.2, BITXOR 6.6.6, BITRSHIFT 6.6.5

6.6.4 BITOR

Summary: Returns bitwise “or” of its parameters

Syntax: BITOR(*Integer* ***X*** ; *Integer* ***Y***)

Returns: *Number*

Constraints: ***X*** ≥ 0, ***Y*** ≥ 0

Semantics: Returns bitwise “or” of its parameters. In the result, each bit position is 1 if any of its parameters' bits at that position are also 1; else it is 0.

See also BITAND 6.6.2, BITXOR 6.6.6, AND 6.15.2

6.6.5 BITRSHIFT

Summary: Returns right shift of value ***X*** by ***N*** bits (“>>”)

Syntax: BITRSHIFT(*Integer* ***X*** ; *Integer* ***N***)

Returns: *Number*

Constraints: ***X*** ≥ 0

Semantics: Returns right shift of value ***X*** by ***N*** bit positions:

- If ***N*** < 0, return BITLSHIFT(***X***,-***N***)
- if ***N*** = 0, return ***X***
- if ***N*** > 0, return INT(***X*** / 2^***N***)

See also BITAND 6.6.2, BITXOR 6.6.6, BITLSHIFT 6.6.3, INT 6.17.2

6.6.6 BITXOR

Summary: Returns bitwise “exclusive or” of its parameters

Syntax: BITXOR(*Integer* ***X*** ; *Integer* ***Y***)

Returns: *Number*

Constraints: ***X*** ≥ 0, ***Y*** ≥ 0

Semantics: Returns bitwise “exclusive or” (xor) of its parameters. In the result, each bit position is 1 if one or the other parameters' bits at that position are 1; else it is 0.

See also BITAND 6.6.2, BITOR 6.6.4, OR 6.15.8

6.7 Byte-position text functions

6.7.1 General

Byte-position text functions are like their equivalent ordinary text functions, but manipulate byte positions rather than a count of the number of characters. Byte positions are integers that may depend on the specific text representation used by the implementation. Byte positions are by definition implementation-dependent and reliance upon them reduces interoperability.

The pseudotypes ByteLength and BytePosition are Integers, but their exact meanings and values are not further defined by this specification.

6.7.2 FINDB

Summary: Returns the starting position of a given text, using byte positions.

Syntax: FINDB(*Text* ***Search*** ; *Text* ***T*** [; *BytePosition* ***Start***])

Returns: *BytePosition*

Semantics: The same as FIND, but using byte positions.

See also FIND 6.20.9 , LEFTB 6.7.3 , RIGHTB 6.7.7

6.7.3 LEFTB

Summary: Returns a selected number of text characters from the left, using a byte position.

Syntax: LEFTB(*Text* ***T*** [; *ByteLength* ***Length***])

Returns: *Text*

Semantics: As LEFT, but using a byte position.

See also LEFT 6.20.12, RIGHT 6.20.19, RIGHTB 6.7.7

6.7.4 LENB

Summary: Returns the length of given text in units compatible with byte positions

Syntax: LENB(*Text* ***T***)

Returns: *ByteLength*

Constraints: None.

Semantics: As LEN, but compatible with byte position values.

See also LEN 6.20.13, LEFTB 6.7.3, RIGHTB 6.7.7

6.7.5 MIDB

Summary: Returns extracted text, given an original text, starting position using a byte position, and length.

Syntax: MIDB(*Text* ***T*** ; *BytePosition* ***Start*** ; *ByteLength* ***Length***)

Returns: *Text*

Constraints: ***Length*** ≥ 0.

Semantics: As MID, but using byte positions.

See also MID 6.20.15, LEFTB 6.7.3, RIGHTB 6.7.7, REPLACEB 6.7.6

6.7.6 REPLACEB

Summary: Returns text where an old text is replaced with a new text, using byte positions.

Syntax: REPLACEB(*Text* ***T*** ; *BytePosition* ***Start*** ; *ByteLength* ***Len*** ; *Text* ***New***)

Returns: *Text*

Semantics: As REPLACE, but using byte positions.

See also REPLACE 6.20.17, LEFTB 6.7.3, RIGHTB 6.7.7, MIDB 6.7.5, SUBSTITUTE 6.20.21

6.7.7 RIGHTB

Summary: Returns a selected number of text characters from the right, using byte position.

Syntax: RIGHTB(*Text* ***T*** [; *ByteLength* ***Length***])

Returns: *Text*

Semantics: As RIGHT, but using byte positions.

See also RIGHT 6.20.19, LEFTB 6.7.3

6.7.8 SEARCHB

Summary: Returns the starting position of a given text, using byte positions.

Syntax: SEARCHB(*Text* ***Search*** ; *Text* ***T*** [; *BytePosition* ***Start***])

Returns: *BytePosition*

Semantics: As SEARCH, but using byte positions.

See also SEARCH 6.20.20, EXACT 6.20.8, FIND 6.20.9, FINDB 6.7.2

6.8 Complex Number Functions

6.8.1 General

Functions for complex numbers.

6.8.2 COMPLEX

Summary: Creates a complex number from a given real coefficient and imaginary coefficient.

Syntax: COMPLEX(*Number* ***Real*** ; *Number* ***Imaginary*** [; *Text* ***Suffix***])

Returns: *Complex*

Constraints: None

Semantics: Constructs a complex number from the given coefficients. The third parameter ***Suffix*** is optional, and should be either “i” or “j”. Upper case “I” or “J” are not accepted for the suffix parameter.

6.8.3 IMABS

Summary: Returns the absolute value of a complex number.

Syntax: IMABS(*Complex* ***X***)

Returns: *Number*

Constraints: None

Semantics: If ***X*** = a + bi or ***X*** = a + bj, the absolute value = $\sqrt{a^2+b^2}$; if ***X*** = r(cosφ + isinφ), the absolute value = r.

See also IMARGUMENT 6.8.5

6.8.4 IMAGINARY

Summary: Returns the imaginary coefficient of a complex number.

Syntax: IMAGINARY(*Complex* ***X***)

Returns: *Number*

Constraints: None

Semantics: If ***X*** = a + bi or ***X*** = a + bj, then the imaginary coefficient is b.

See also IMREAL 6.8.19

6.8.5 IMARGUMENT

Summary: Returns the complex argument of a complex number.

Syntax: IMARGUMENT(*Complex* ***X***)

Returns: *Number*

Constraints: None

Semantics: If ***X*** = a + bi = r(cosφ + isinφ), a or b is not 0 and $-\pi < \varphi \leq \pi$, then the complex argument is φ. φ is expressed by radians. If ***X*** = 0, then IMARGUMENT(***X***) is implementation-defined and either 0 or an error.

See also IMABS 6.8.3

6.8.6 IMCONJUGATE

Summary: Returns the complex conjugate of a complex number.

Syntax: IMCONJUGATE(*Complex* ***X***)

Returns: *Complex*

Constraints: None

Semantics: If ***X*** = a + bi, then the complex conjugate is a - bi.

6.8.7 IMCOS

Summary: Returns the cosine of a complex number.

Syntax: IMCOS(*Complex* ***X***)

Returns: *Complex*

Constraints: None

Semantics: If ***X*** = a + bi, then cos(***X***) = cos(a)cosh(b) - sin(a)sinh(b)i.

See also IMSIN 6.8.20

6.8.8 IMCOSH

Summary: Returns the hyperbolic cosine of a complex number.

Syntax: IMCOSH(*Complex* ***N***)

Returns: *Complex*

Constraints: None

Semantics: If ***N*** = a + bi, then cosh(***N***) = cosh(a)cos(b) + sinh(a)sin(b)i.

6.8.9 IMCOT

Summary: Returns the cotangent of a complex number.

Syntax: IMCOT(*Complex* ***N***)

Returns: *Complex*

Constraints: None

Semantics: Equivalent to the following (except ***N*** is computed only once):

IMDIV(IMCOS(***N***);IMSIN(***N***))

See also IMCOS 6.8.7, IMDIV 6.8.12, IMSIN 6.8.20, IMTAN 6.8.27

6.8.10 IMCSC

Summary: Returns the cosecant of a complex number.

Syntax: IMCSC(*Complex* ***N***)

Returns: *Complex*

Constraints: None

Semantics: Equivalent to the following:

IMDIV(1;IMSIN(***N***))

See also IMDIV 6.8.12, IMSIN 6.8.20

6.8.11 IMCSCH

Summary: Returns the hyperbolic cosecant of a complex number.

Syntax: IMCSCH(*Complex* ***N***)

Returns: *Complex*

Constraints: None

Semantics: Computes the hyperbolic cosecant. This is equivalent to:

IMDIV(1;IMSINH(***N***))

See also IMSINH 6.8.21, CSCH 6.16.24

6.8.12 IMDIV

Summary: Divides the first number by the second.

Syntax: IMDIV(*Complex* ***X*** ; *Complex* ***Y***)

Returns: *Complex*

Constraints: None

Semantics: Given ***X*** = a + bi and ***Y*** = c + di, return the quotient

$$\frac{(ac+bd)+(bc-ad)i}{(c^2+d^2)}$$

Division by zero returns an Error.

See also IMDIV 6.8.12

6.8.13 IMEXP

Summary: Returns the exponent of e and a complex number.

Syntax: IMEXP(*Complex* ***X***)

Returns: *Complex*

Constraints: None

Semantics: If ***X*** = a + bi, the result is $e^{a}(\cos b + i \sin b)$.

See also IMLN 6.8.14

6.8.14 IMLN

Summary: Returns the natural logarithm of a complex number.

Syntax: IMLN(*Complex* ***X***)

Returns: *Complex*

Constraints: ***X*** ≠ 0

Semantics: COMPLEX(LN(IMABS(***X***)); IMARGUMENT(***X***)) .

See also COMPLEX 6.8.2, IMABS 6.8.3, IMARGUMENT 6.8.5, IMEXP 6.8.13 , IMLOG10 6.8.15, LN 6.16.39

6.8.15 IMLOG10

Summary: Returns the common logarithm of a complex number.

Syntax: IMLOG10(*Complex* ***X***)

Returns: *Complex*

Constraints: ***X*** ≠ 0

Semantics: IMLOG10(***X***) is IMDIV(IMLN(***X***);COMPLEX(LN(10);0)) .

See also COMPLEX 6.8.2, IMDIV 6.8.12, IMLN 6.8.14 , IMPOWER 6.8.17, LN 6.16.39

6.8.16 IMLOG2

Summary: Returns the binary logarithm of a complex number.

Syntax: IMLOG2(*Complex* ***X***)

Returns: *Complex*

Constraints: ***X*** ≠ 0

Semantics: IMLOG2(***X***) is IMDIV(IMLN(***X***);COMPLEX(LN(2);0)) .

See also COMPLEX 6.8.2, IMDIV 6.8.12, IMLN 6.8.14 , IMPOWER 6.8.17, LN 6.16.39

6.8.17 IMPOWER

Summary: Returns the complex number ***X*** raised to the Yth power.

Syntax: IMPOWER(*Complex* ***X*** ; *Complex* ***Y***) or IMPOWER(*Complex* ***X*** ; *Number* ***Y***)

Returns: *Complex*

Constraints: ***X*** ≠ 0

Semantics: IMPOWER(***X***;***Y***) is IMEXP(IMPRODUCT(***Y***; IMLN(***X***)))

An evaluator implementing this function shall permit any Number ***Y*** but may also allow any Complex ***Y***.

See also IMEXP 6.8.13, IMLN 6.8.14, IMPOWER 6.8.17, IMPRODUCT 6.8.18

6.8.18 IMPRODUCT

Summary: Returns the product of complex numbers.

Syntax: IMPRODUCT({ *ComplexSequence* ***N*** }$^{+}$)

Returns: *Complex*

Constraints: None

Semantics: Multiply the complex numbers together. Given two complex numbers X = a + bi and
Y = c + di, the product X * Y = (ac - bd) + (ad + bc)i

See also IMDIV 6.8.12

6.8.19 IMREAL

Summary: Returns the real coefficient of a complex number.

Syntax: IMREAL(*Complex* ***N***)

Returns: *Number*

Constraints: None

Semantics: If ***N*** = a + bi or ***N*** = a + bj, then the real coefficient is a.

See also IMAGINARY 6.8.4

6.8.20 IMSIN

Summary: Returns the sine of a complex number.

Syntax: IMSIN(*Complex* ***N***)

Returns: *Complex*

Constraints: None

Semantics: If ***N*** = a + bi, then sin(***N***) = sin(a)cosh(b) + cos(a)sinh(b)i.

See also IMCOS 6.8.7

6.8.21 IMSINH

Summary: Returns the hyperbolic sine of a complex number.

Syntax: IMSINH(*Complex* ***N***)

Returns: *Complex*

Constraints: None

Semantics: If ***N*** = a + bi, then sinh(***N***) = sinh(a)cos(b) + cosh(a)sin(b)i.

6.8.22 IMSEC

Summary: Returns the secant of a complex number.

Syntax: IMSEC(*Complex* ***N***)

Returns: *Complex*

Constraints: None

Semantics: Equivalent to the following:

IMDIV(1;IMCOS(***N***))

See also IMCOS 6.8.7, IMDIV 6.8.12

6.8.23 IMSECH

Summary: Returns the hyperbolic secant of a complex number.

Syntax: IMSECH(*Complex* ***N***)

Returns: *Number*

Constraints: None

Semantics: Computes the hyperbolic secant. This is equivalent to:

IMDIV(1;IMCOSH(***N***))

See also IMCOSH 6.8.8, IMDIV 6.8.12, SECH 6.16.57

6.8.24 IMSQRT

Summary: Returns the square root of a complex number.

Syntax: IMSQRT(*Complex* ***N***)

Returns: *Complex*

Constraints: None

Semantics: If ***N*** = 0 + 0i, then IMSQRT(***N***) = 0. Otherwise IMSQRT(***N***) is SQRT(IMABS(***N***)) * sin(IMARGUMENT(***N***) / 2) + SQRT(IMABS(***N***)) * cos(IMARGUMENT(***N***) / 2)i.

See also IMABS 6.8.3, IMARGUMENT 6.8.5, IMPOWER 6.8.17, SQRT 6.16.58

6.8.25 IMSUB

Summary: Subtracts the second complex number from the first.

Syntax: IMSUB(*Complex* ***X*** ; *Complex* ***Y***)

Returns: *Complex*

Constraints: None

Semantics: Subtract complex number ***Y*** from ***X***.

See also IMSUM 6.8.26

6.8.26 IMSUM

Summary: Sums (add) a set of complex numbers, including all numbers in ranges.

Syntax: IMSUM({ *ComplexSequence* ***N*** }$^{+}$)

Returns: *Complex*

Constraints: None

Semantics: Adds complex numbers together. Text that cannot be converted to a complex number is ignored.

It is implementation-defined what happens if this function is given zero parameters; an evaluator may either produce an Error or the Number 0 if it is given zero parameters.

See also IMSUB 6.8.25

6.8.27 IMTAN

Summary: Returns the tangent of a complex number

Syntax: IMTAN(*Complex* ***N***)

Returns: *Complex*

Constraints: None

Semantics: Equivalent to the following (except ***N*** is computed only once):

IMDIV(IMSIN(***N***);IMCOS(***N***))

See also IMDIV 6.8.12, IMSIN 6.8.20, IMCOS 6.8.7, IMCOT 6.8.25

6.9 Database Functions

6.9.1 General

Database functions use the variables, Database 4.11.9, Field Error: Reference source not found, and Criteria Error: Reference source not found.

The results of database functions may change when the values of the HOST-USE-REGULAR-EXPRESSIONS or HOST-USE-WILDCARDS or HOST-SEARCH-CRITERIA-MUST-APPLY-TO-WHOLE-CELL properties change. 3.4

6.9.2 DAVERAGE

Summary: Finds the average of values in a given field from the records (rows) in a database that match a search criteria.

Syntax: DAVERAGE(*Database **D*** ; *Field **F*** ; *Criteria **C***)

Returns: *Number*

Constraints: None

Semantics: Perform AVERAGE on data records in database ***D*** field ***F*** that match criteria ***C***.

See also AVERAGE 6.18.3, COUNT 6.13.6, DSUM 6.9.11, DCOUNT 6.9.3, SUM 6.16.61

6.9.3 DCOUNT

Summary: Counts the number of records (rows) in a database that match a search criteria and contain numerical values.

Syntax: DCOUNT(*Database **D*** ; [*Field **F***] ; *Criteria **C***)

Returns: *Number*

Constraints: None

Semantics: Perform COUNT on data records in database ***D*** field ***F*** that match criteria ***C***. If the Field argument is omitted, DCOUNT returns the count of all records that satisfy Criteria ***C***.

See also COUNT 6.13.6, COUNTA 6.13.7, DCOUNTA 6.9.4, DSUM 6.9.11

6.9.4 DCOUNTA

Summary: Counts the number of records (rows) in a database that match a search criteria and contain values (as COUNTA).

Syntax: DCOUNTA(*Database **D*** ; [*Field **F***] ; *Criteria **C***)

Returns: *Number*

Constraints: None

Semantics: Perform COUNTA on data records in database ***D*** field ***F*** that match criteria ***C***. If the Field argument is omitted, DCOUNTA returns the count of all records that satisfy criteria ***C***.

See also COUNT 6.13.6, COUNTA 6.13.7, DCOUNT 6.9.3, DSUM 6.9.11

6.9.5 DGET

Summary: Gets the single value in the field from the single record (row) in a database that matches a search criteria.

Syntax: DGET(*Database **D*** ; *Field **F*** ; *Criteria **C***)

Returns: *Number*

Constraints: None

Semantics: Extracts the value in field ***F*** of the one data record in database ***D*** that matches criteria ***C***. If no records match, or more than one matches, it returns an Error.

See also DMAX 6.9.6, DMIN 6.9.7, DSUM 6.9.11

6.9.6 DMAX

Summary: Finds the maximum value in a given field from the records (rows) in a database that match a search criteria.

Syntax: DMAX(*Database* ***D*** ; *Field* ***F*** ; *Criteria* ***C***)

Returns: *Number*

Constraints: None

Semantics: Perform MAX on only the data records in database ***D*** field ***F*** that match criteria ***C***.

See also MAX 6.18.45, DMIN 6.9.7, MIN 6.18.48

6.9.7 DMIN

Summary: Finds the minimum value in a given field from the records (rows) in a database that match a search criteria.

Syntax: DMIN(*Database* ***D*** ; *Field* ***F*** ; *Criteria* ***C***)

Returns: *Number*

Constraints: None

Semantics: Perform MIN on only the data records in database ***D*** field ***F*** that match criteria ***C***.

See also MIN 6.18.48, DMAX 6.9.6, MAX 6.18.45

6.9.8 DPRODUCT

Summary: Finds the product of values in a given field from the records (rows) in a database that match a search criteria.

Syntax: DPRODUCT(*Database* ***D*** ; *Field* ***F*** ; *Criteria* ***C***)

Returns: *Number*

Constraints: None

Semantics: Multiply together only the data records in database ***D*** field ***F*** that match criteria ***C***.

See also SUM 6.16.61, DSUM 6.9.11

6.9.9 DSTDEV

Summary: Finds the sample standard deviation in a given field from the records (rows) in a database that match a search criteria.

Syntax: DSTDEV(*Database* ***D*** ; *Field* ***F*** ; *Criteria* ***C***)

Returns: *Number*

Constraints: None

Semantics: Perform STDEV on only the data records in database ***D*** field ***F*** that match criteria ***C***.

See also STDEV 6.18.72, DSTDEVP 6.9.10

6.9.10 DSTDEVP

Summary: Finds the population standard deviation in a given field from the records (rows) in a database that match a search criteria.

Syntax: DSTDEVP(*Database **D*** ; *Field **F*** ; *Criteria **C***)

Returns: *Number*

Constraints: None

Semantics: Perform STDEVP on only the data records in database ***D*** field ***F*** that match criteria ***C***.

See also STDEVP 6.18.74, DSTDEV 6.9.9

6.9.11 DSUM

Summary: Finds the sum of values in a given field from the records (rows) in a database that match a search criteria.

Syntax: DSUM(*Database **D*** ; *Field **F*** ; *Criteria **C***)

Returns: *Number*

Constraints: None

Semantics: Perform SUM on only the data records in database ***D*** field ***F*** that match criteria ***C***.

See also SUM 6.16.61, DMIN 6.9.7, DMAX 6.9.6

6.9.12 DVAR

Summary: Finds the sample variance in a given field from the records (rows) in a database that match a search criteria.

Syntax: DVAR(*Database **D*** ; *Field **F*** ; *Criteria **C***)

Returns: *Number*

Constraints: None

Semantics: Perform VAR on only the data records in database ***D*** field ***F*** that match criteria ***C***.

See also VAR 6.18.82, DVARP 6.9.13

6.9.13 DVARP

Summary: Finds the population variance in a given field from the records (rows) in a database that match a search criteria.

Syntax: DVARP(*Database **D*** ; *Field **F*** ; *Criteria **C***)

Returns: *Number*

Constraints: None

Semantics: Perform VARP on only the data records in database ***D*** field ***F*** that match criteria ***C***.

See also VARP 6.18.84, DVAR 6.9.12

6.10 Date and Time Functions

6.10.1 General

6.10.2 DATE

Summary: Constructs a date from year, month, and day of month.

Syntax: DATE(*Integer **Year*** ; *Integer **Month*** ; *Integer **Day***)

Returns: *Date*

Constraints: 1904 ≤ ***Year*** ≤ 9956; 1 ≤ ***Month*** ≤ 12; 1 ≤ ***Day*** ≤ 31; Evaluators may evaluate expressions that do no meet this constraint.

Semantics: This computes the date's serial number given ***Year***, ***Month***, and ***Day*** of the Gregorian calendar. Fractional values are truncated. ***Month*** > 12 and ***Day*** > days of ***Month*** will roll over the date, computing the result by adding months and days as necessary. The value of the serial number depends on the current epoch.

See also TIME 6.10.18, DATEVALUE 6.10.4

6.10.3 DATEDIF

Summary: Returns the difference in years, months, or days of two date numbers.

Syntax: DATEDIF(*DateParam* ***StartDate*** ; *DateParam* ***EndDate*** ; *Text* ***Format***)

Returns: *Number*

Constraints: None

Semantics: Compute difference of ***StartDate*** and ***EndDate***, in the units given by ***Format***.

The ***Format*** is a code from the following table, entered as text, that specifies the format you want the result of DATEDIF to have.

Table 5 - DATEDIF

Format	***Returns the number of***
y	Years
m	Months. If there is not a complete month between the dates, 0 will be returned.
d	Days
md	Days, ignoring months and years
ym	Months, ignoring years
yd	Days, ignoring years

See also DAYS360 6.10.7, DAYS 6.10.6, Infix Operator "-" 6.4.3

6.10.4 DATEVALUE

Summary: Returns the date serial number from given text.

Syntax: DATEVALUE(*Text* ***D***)

Returns: *Date*

Constraints: None

Semantics: This computes the serial number of the text string ***D***, using the current locale. This function shall accept ISO date format (YYYY-MM-DD), which is locale-independent. It is semantically equal to VALUE(Date), if Date has a date format, since text matching a date format is automatically converted to a serial number when used as a Number. If the text of ***D*** has a combined date and time format, e.g. YYYY-MM-DD HH:MM:SS, the integer part of the date serial number is returned. If the text of ***D*** does not have a date or time format, an evaluator *may* return an Error. See VALUE for more information on date formats. The value of the serial number depends on the current epoch.

See also TIME 6.10.18, DATE 6.10.2, TIMEVALUE 6.10.19, VALUE 6.13.34

6.10.5 DAY

Summary: Returns the day from a date.

Syntax: DAY(*DateParam* ***D***)

Returns: *Number*

Constraints: None

Semantics: Returns the day portion of ***D***.

See also MONTH 6.10.14, YEAR 6.10.24

6.10.6 DAYS

Summary: Returns the number of days between two dates

Syntax: DAYS(*DateParam* ***EndDate*** ; *DateParam* ***StartDate***)

Returns: *Number*

Constraints: None

Semantics: Returns the number of days between two dates. If ***StartDate*** and ***EndDate*** are Numbers, this is ***EndDate*** – ***StartDate***. If they are both Text, this is DATEVALUE(***EndDate***) – DATEVALUE(***StartDate***).

See also DATEDIF 6.10.3, DATEVALUE 6.10.4, DAYS360 6.10.7, MONTH 6.10.14, YEAR 6.10.24, Infix Operator “-” 6.4.3

6.10.7 DAYS360

Summary: Returns the number of days between two dates using the 360-day year

Syntax: DAYS360(*DateParam* ***StartDate*** ; *DateParam* ***EndDate*** [; *Logical* ***Method*** = FALSE])

Returns: *Number*

Constraints: None

Semantics: If ***Method*** is FALSE, it uses the National Association of Securities Dealers (NASD) method, also known as the U.S. method. If ***Method*** is TRUE, the European method is used.

The US/NASD method (30US/360):

1. Truncate date values, set sign = 1.
2. If ***StartDate***'s day-of-month is 31, it is changed to 30.
3. Otherwise, if ***StartDate***'s day-of-month is the last day of February, it is changed to 30.
4. If ***EndDate***'s day-of-month is 31 and ***StartDate***'s day-of-month is 30 (after having applied a change for #2 or #3, if necessary), ***EndDate***'s day-of-month is changed to 30.

   Note 1: This calculation is slightly different from Basis 0 (4.11.7 Basis). Dates are never swapped.

The European method (30E/360):

1. Truncate date values, set sign = 1.
2. If ***StartDate*** is after ***EndDate*** then swap dates and set sign = -1.
3. If ***StartDate***'s day-of-month is 31, it is changed to 30.
4. If ***EndDate***'s day-of-month is 31, it is changed to 30.

   Note 2: Days in February are never changed.

Note 3: This calculation is identical to Basis 4 (4.11.7 Basis)

For both methods the value then returned is
sign * ((***EndDate***.year * 360 + ***EndDate***.month * 30 + ***EndDate***.day) – (***StartDate***.year * 360 + ***StartDate***.month * 30 + ***StartDate***.day))

See also DAYS 6.10.6, DATEDIF 6.10.3

6.10.8 EASTERSUNDAY

Summary: Returns the date of Easter Sunday in the Gregorian Calendar in ***Year*** if given, otherwise returns the date of the next Easter Sunday on or after TODAY().

> **Note:** The use of the Gregorian calendar was introduced in different locations at different times.

Syntax: EASTERSUNDAY([*Integer* ***Year***])

Returns: *Date*

Constraints: 1583 ≤ ***Year*** ≤ 9956

Semantics: Given an integer value for Year in the allowed range, the Easter Sunday date is calculated according to the following algorithm:

```
A1 = INT(Year/100)
A2 = Year-19*INT(Year/19)
A3 = INT((A1-17)/25)
A4 = MOD(A1-INT(A1/4)-INT((A1-A3)/3)+19*A2+15;30)
A5 = A4-INT(A4/28)*(1-INT(A4/28)*INT(29/(A4+1))*INT((21-A2)/11))
A6 = MOD(Year+INT(Year/4)+A5+2-A1+INT(A1/4);7)
A7 = A5-A6
month = 3+INT((A7+40)/44)
day = A7+28-31*INT(month/4)
```

The resulting date is: DATE(**`Year`**; **`month`**; **`day`**).

If ***Year*** is not given, the above calculation is performed for the current and the next year, and the smaller of the two dates that is not less than TODAY() is returned.

6.10.9 EDATE

Summary: Returns the serial number of a given date when ***MonthAdd*** months is added

Syntax: EDATE(*DateParam* ***StartDate*** ; *Number* ***MonthAdd***)

Returns: *Number*

Constraints: None

Semantics: First truncate ***StartDate*** and ***MonthAdd***, then add ***MonthAdd*** number of months. ***MonthAdd*** can be positive, negative, or 0; if zero, the number representing ***StartDate*** (in the current epoch) is returned.

If after adding the given number of months, the day of month in the new month is larger than the number of days in the given month, the day of month is adjusted to the last day of the new month. Then the serial number of that date is returned.

See also DAYS 6.10.6, DATEDIF 6.10.3, EOMONTH 6.10.10

6.10.10 EOMONTH

Summary: Returns the serial number of the end of a month, given date plus ***MonthAdd*** months

Syntax: EOMONTH(*DateParam* ***StartDate*** ; *Integer* ***MonthAdd***

Returns: *Number*

Constraints: None

Semantics: First truncate ***StartDate*** and ***MonthAdd***, then add ***MonthAdd*** number of months. ***MonthAdd*** can be positive, negative, or 0. Then return the serial number representing the end of that month. Due to the semantics of this function, the value of DAY(***StartDate***) is irrelevant.

See also DAY 6.10.5, EDATE 6.10.9

6.10.11 HOUR

Summary: Extracts the hour (0 through 23) from a time.

Syntax: HOUR(*TimeParam* ***T***)

Returns: *Number*

Constraints: None

Semantics: Extract from ***T*** the hour value, 0 through 23, as per a 24-hour clock. This is equal to:

DayFraction = (***T*** - INT(***T***))

Hour = INT(DayFraction * 24)

See also MONTH 6.10.14, DAY 6.10.5, MINUTE 6.10.13, SECOND 6.10.17, INT 6.17.2

6.10.12 ISOWEEKNUM

Summary: Determines the ISO week number of the year for a given date.

Syntax: ISOWEEKNUM(*DateParam* ***D***)

Returns: *Number*

Constraints: None

Semantics: Returns the ordinal number of the [ISO8601] calendar week in the year for the given date ***D***. ISO 8601 defines the calendar week as a time interval of seven calendar days starting with a Monday, and the first calendar week of a year as the one that includes the first Thursday of that year.

See also DAY 6.10.5, MONTH 6.10.14, YEAR 6.10.24, WEEKDAY 6.10.21, WEEKNUM 6.10.22

6.10.13 MINUTE

Summary: Extracts the minute (0 through 59) from a time.

Syntax: MINUTE(*TimeParam* ***T***)

Returns: *Number*

Constraints: None

Semantics: Extract from ***T*** the minute value, 0 through 59, as per a clock. This is equal to:

Second = MOD(ROUND(***T*** * 86400) ; 60)

Minute = (MOD(ROUND(***T*** * 86400) ; 3600) - Second) / 60

DayFraction = (***T*** - INT(***T***))

HourFraction = (DayFraction * 24 - INT(DayFraction * 24))

Minute = INT(HourFraction * 60)

See also MONTH 6.10.14, DAY 6.10.5, HOUR 6.10.11, SECOND 6.10.17, INT 6.17.2

6.10.14 MONTH

Summary: Extracts the month from a date.

Syntax: MONTH(*DateParam* ***Date***)

Returns: *Number*

Constraints: None

Semantics: Takes ***Date*** and returns the month portion.

See also YEAR 6.10.24, DAY 6.10.5

6.10.15 NETWORKDAYS

Summary: Returns the whole number of work days between two dates.

Syntax: NETWORKDAYS(*DateParam* ***Date1*** ; *DateParam* ***Date2*** [; [*DateSequence* ***Holidays***] [; *LogicalSequence* ***Workdays***]])

Returns: *Number*

Constraints: None

Semantics: Returns the whole number of work days between two dates.

Work days are defined as non-weekend, non-holiday days. By default, weekends are Saturdays and Sundays and there are no holidays.

The optional 3rd parameter ***Holidays*** can be used to specify a list of dates to be treated as holidays. Note that this parameter can be omitted as an empty parameter (two consecutive ;; semicolons) to be able to pass the set of ***Workdays*** without ***Holidays***.

The optional 4th parameter ***Workdays*** can be used to specify a different definition for the standard work week by passing in a list of numbers which define which days of the week are workdays (indicated by 0) or not (indicated by non-zero) in order Sunday, Monday,...,Saturday. So, the default definition of the work week excludes Saturday and Sunday and is: {1;0;0;0;0;0;1}. To define the work week as excluding Friday and Saturday, the third parameter would be: {0;0;0;0;0;1;1}.

6.10.16 NOW

Summary: Returns the serial number of the current date and time.

Syntax: NOW()

Returns: *DateTime*

Constraints: None

Semantics: This returns the current day and time serial number, using the current locale. If you want only the serial number of the current day, use TODAY 6.10.20.

See also DATE 6.10.2, TIME 6.10.18, TODAY 6.10.20

6.10.17 SECOND

Summary: Extracts the second (the integer 0 through 59) from a time. This function presumes that leap seconds never exist.

Syntax: SECOND(*TimeParam* ***T***)

Returns: *Number*

Constraints: None

Semantics: Extract from ***T*** the second value, 0 through 59, as per a clock, after first *rounding* to the nearest second. Note that this returns an *integer*, without a fractional part. Note also that this *rounds* to the nearest second, instead of returning the integer part of the seconds. This is equal to:

Second = MOD(ROUND(***T*** * 86400) ; 60)

DayFraction = (***T*** - INT(***T***))

HourFraction = (DayFraction * 24 - INT(DayFraction * 24))

MinuteFraction = (HourFraction * 60 - INT(HourFraction * 60))

Second = ROUND(MinuteFraction * 60)

See also MONTH 6.10.14, DAY 6.10.5, HOUR 6.10.11, MINUTE 6.10.13, INT 6.17.2

6.10.18 TIME

Summary: Constructs a time value from hours, minutes, and seconds.

Syntax: TIME(*Number* ***Hours*** ; *Number* ***Minutes*** ; *Number* ***Seconds***)

Returns: *Time*

Constraints: None. Evaluators may first perform INT() on the hour, minute, and second before doing the calculation.

Semantics: Returns the fraction of the day consumed by the given time, i.e.:

((***Hours*** * 60 * 60) ***+ (Minutes*** * 60) ***+ Seconds***) / (24 * 60 * 60)

Time is a subtype of Number, where a time value of 1 = 1 day = 24 hours.

Hours, Minutes, and Seconds may be any number (they shall not be limited to the ranges 0..24, 0..59, or 0..60 respectively).

See also DATE 6.10.2, INT 6.17.2

6.10.19 TIMEVALUE

Summary: Returns a time serial number from given text.

Syntax: TIMEVALUE(*Text* ***T***)

Returns: *Time*

Constraints: None

Semantics: This computes the serial number of the text string ***T***, which is a time, using the current locale. This function shall accept ISO time format (HH:MM:SS), which is locale-independent. If the text of ***T*** has a combined date and time format, e.g. YYYY-MM-DD HH:MM:SS, the fractional part of the date serial number is returned. If the text of ***T*** does not have a time format, an evaluator may attempt to convert the number another way (e.g., using VALUE), or it may return an Error (this is implementation-dependent).

See also TIME 6.10.18, DATE 6.10.2, DATEVALUE 6.10.4, VALUE 6.13.34

6.10.20 TODAY

Summary: Returns the serial number of today.

Syntax: TODAY()

Returns: *Date*

Constraints: None

Semantics: This returns the current day's serial number, using current locale. This only returns the date, not the datetime value. For the specific time of day as well, use NOW 6.10.16.

See also TIME 6.10.18, NOW 6.10.16

6.10.21 WEEKDAY

Summary: Extracts the day of the week from a date; if text, uses current locale to convert to a date.

Syntax: WEEKDAY(*DateParam* ***D*** [; *Integer* ***Type*** = 1])

Returns: *Number*

Constraints: None

Semantics: Returns the day of the week from a date ***D***, as a number from 0 through 7. The exact meaning depends on the value of ***Type***:

1. When ***Type*** is 1, Sunday is the first day of the week, with value 1; Saturday has value 7.
2. When ***Type*** is 2, Monday is the first day of the week, with value 1; Sunday has value 7.
3. When ***Type*** is 3, Monday is the first day of the week, with value 0; Sunday has value 6.
4. When ***Type*** is 11, Monday is the first day of the week, with value 1; Sunday has value 7.
5. When ***Type*** is 12, Tuesday is the first day of the week, with value 1; Monday has value 7.
6. When ***Type*** is 13, Wednesday is the first day of the week, with value 1; Tuesday has value 7.
7. When ***Type*** is 14, Thursday is the first day of the week, with value 1; Wednesday has value 7.
8. When ***Type*** is 15, Friday is the first day of the week, with value 1; Thursday has value 7.
9. When ***Type*** is 16, Saturday is the first day of the week, with value 1; Friday has value 7.
10. When ***Type*** is 17, Sunday is the first day of the week, with value 1; Saturday has value 7.

Table 6 - WEEKDAY

Weekday Type	1	2	3	11	12	13	14	15	16	17
Sunday	1	7	6	7	6	5	4	3	2	1
Monday	2	1	0	1	7	6	5	4	3	2
Tuesday	3	2	1	2	1	7	6	5	4	3
Wednesday	4	3	2	3	2	1	7	6	5	4
Thursday	5	4	3	4	3	2	1	7	6	5
Friday	6	5	4	5	4	3	2	1	7	6
Saturday	7	6	5	6	5	4	3	2	1	7

See also DAY 6.10.5, MONTH 6.10.14, YEAR 6.10.24

6.10.22 WEEKNUM

Summary: Determines the week number of the year for a given date.

Syntax: WEEKNUM(*DateParam* ***D*** [; *Number* ***Mode*** = 1])

Returns: *Number*

Constraints: 1 ≤ ***Mode*** ≤ 2, or 11 ≤ ***Mode*** ≤ 17, or ***Mode*** = 21, or ***Mode*** = 150

Semantics: Returns the number of the week in the year for the given date.

For ***Mode*** = {1, 2, 11, 12, ..., 17} the week containing January 1 is the first week of the year, and is numbered week 1. The week starts on {Sunday, Monday, Monday, Tuesday, ..., Sunday}.

Mode 21 and ***Mode*** 150 are both [ISO8601], the week starts on Monday and the week containing the first Thursday of the year is the first week of the year, and is numbered week 1.

See also DAY 6.10.5, MONTH 6.10.14, YEAR 6.10.24, WEEKDAY 6.10.21, ISOWEEKNUM 6.10.12

6.10.23 WORKDAY

Summary: Returns the date serial number which is a specified number of work days before or after an input date.

Syntax: WORKDAY(*DateParam Date* ; *Number Offset* [; [*DateSequence* ***Holidays***] [; *LogicalSequence* ***Workdays***]])

Returns: *DateTime*

Constraints: None

Semantics: Returns the date serial number for the day that is offset from the input ***Date*** parameter by the number of work days specified in the *Offset* parameter. If *Offset* is negative, the offset will return a date prior to ***Date***. If *Offset* is positive, a date later ***Date*** is returned. If *Offset* is zero, then ***Date*** is returned.

Work days are defined as non-weekend, non-holiday days. By default, weekends are Saturdays and Sundays and there are no holidays.

The optional 3rd parameter ***Holidays*** can be used to specify a list of dates to be treated as holidays. Note that this parameter can be omitted as an empty parameter (two consecutive ;; semicolons) to be able to pass the set of ***Workdays*** without ***Holidays***.

The optional 4th parameter ***Workdays*** can be used to specify a different definition for the standard work week by passing in a list of numbers which define which days of the week are workdays (indicated by 0) or not (indicated by non-zero) in order Sunday, Monday,...,Saturday. If all seven numbers in ***Workdays*** are non-zero and Offset is also non-zero, WORKDAY returns an error.

Note: The default definition of the work week that excludes Saturday and Sunday and is: {1;0;0;0;0;0;1}. To define the workweek as excluding Friday and Saturday, the third parameter would be: {0;0;0;0;0;1;1}.

6.10.24 YEAR

Summary: Extracts the year from a date given in the current locale of the evaluator.

Syntax: YEAR(*DateParam* ***D***)

Returns: *Number*

Constraints: None

Semantics: Parses a date-formatted string in the current locale's format and returns the year portion.

If a year is given as a two-digit number, as in "05-21-15", then the year returned is either 1915 or 2015, depending upon the break point in the calculation context. In an OpenDocument document, this break point is determined by `HOST-NULL-YEAR`.

Evaluators shall support extracting the year from a date beginning in 1900. Three-digit year numbers precede adoption of the Gregorian calendar, and may return either an Error or the

year number. Four-digit year numbers preceding 1582 (inception of the Gregorian Calendar) may return either an Error or the year number. Four-digit year numbers following 1582 should return the year number.

See also MONTH 6.10.14, DAY 6.10.5, VALUE 6.13.34

6.10.25 YEARFRAC

Summary: Extracts the number of years (including fractional part) between two dates

Syntax: YEARFRAC(*DateParam* ***StartDate*** ; *DateParam* ***EndDate*** [; *Basis* ***B*** = 0])

Returns: *Number*

Constraints: None

Semantics: Computes the fraction of the number of years between a ***StartDate*** and ***EndDate***.

B indicates the day-count convention to use in the calculation. 4.11.7

See also DATEDIF 6.10.3

6.11 External Access Functions

6.11.1 General

OpenFormula defines two functions, DDE and HYPERLINK, for accessing external data.

6.11.2 DDE

Summary: Returns data from a DDE request

Syntax: DDE(*Text* ***Server*** ; *Text* ***Topic*** ; *Text* ***Item*** [; *Integer* ***Mode*** = 0])

Returns: *Number|Text*

Constraints: None

Semantics: Performs a DDE request and returns its result. The request invokes the service ***Server*** on the topic named as ***Topic***, requesting that it reply with the information on ***Item***.

Evaluators may choose to not perform this function on every recalculation, but instead cache an answer and require a separate action to re-perform these requests. Evaluators shall perform this request on initial load when their security policies permit it.

Mode is an optional parameter that determines how the results are returned:

Table 7 - DDE

Mode	Effect
0 or missing	Data converted to number using VALUE in the number style's locale of the default table cell style
1	Data converted to number using VALUE in the English-US (en_US) locale
2	Data retrieved as text (not converted to number)

In an OpenDocument spreadsheet document the default table cell style is specified with `table:default-cell-style-name`. Its `number:number-style` specified by `style:data-style-name` specifies the locale to use in the conversion.

The DDE function is non-portable because it depends on availability of external programs (server parameter) and their interpretation of the topic and item parameters.

6.11.3 HYPERLINK

Summary: Creation of a hyperlink involving an evaluated expression.

Syntax: HYPERLINK(*Text* ***IRI*** [; *Text|Number* ***FunctionResult***])

Returns: *Text* or *Number*

Constraints: None

Semantics: The default for the second argument is the value of the first argument. The second argument value is returned.

In addition, hosting environments may interpret expressions containing HYPERLINK function calls as calling for an implementation-dependent creation of a hypertext link based on the expression containing the HYPERLINK function calls.

6.12 Financial Functions

6.12.1 General

The financial functions are defined for use in financial calculations.

An annuity is a recurring series of payments. A "simple annuity" is one where equal payments are made at equal intervals, and the compounding of interest occurs at those same intervals. The time between payments is called the "payment interval". Where payments are made at the end of the payment interval, it is called an "ordinary annuity". Where payments are made at the beginning of the payment interval, it is called an "annuity due". Periods are numbered starting at 1.

Financial functions defined in this standard use a cash flow sign convention where outgoing cash flows are negative and incoming cash flows are positive.

6.12.2 ACCRINT

Summary: Calculates the accrued interest for securities with periodic interest payments.

Syntax: ACCRINT(*DateParam* ***Issue*** ; *DateParam* ***First*** ; *DateParam* ***Settlement*** ; *Number* ***Coupon*** ; *Number* ***Par*** ; *Integer* ***Frequency*** [; *Basis* ***B*** = 0 [; *Logical* ***CalcMethod*** = TRUE]])

Returns: *Currency*

Constraints: ***Issue*** < ***First*** < ***Settlement*** ; ***Coupon*** > 0; ***Par*** > 0

Frequency is one of the following values:

Table 8 - ACCRINT

Frequency	***Frequency of coupon payments***
1	Annual
2	Semiannual
4	Quarterly
12	Monthly

Semantics: Calculates the accrued interest for securities with periodic interest payments. ACCRINT supports short, standard, and long ***Coupon*** periods.

If ***CalcMethod*** is TRUE (the default) then ACCRINT returns the sum of the accrued interest in each coupon period from issue date until settlement date. If ***CalcMethod*** is FALSE then ACCRINT returns the sum of the accrued interest in each coupon period from first interest date until settlement date. For each coupon period, the interest is ***Par*** * ***Coupon*** * YEARFRAC(start-of-period;end-of-period; ***B***)

- ***Issue***: The security's issue or dated date.
- ***First***: The security's first interest date.
- ***Settlement***: The security's settlement date.
- ***Coupon***: The security's annual coupon rate.
- ***Par***: The security's par value, that is, the principal to be paid at maturity.
- ***Frequency****:* The number of coupon payments per year.
- ***B***: Indicates the day-count convention to use in the calculation. 4.11.7
- ***CalcMethod***: A logical value that specifies how to treat the case where ***Settlement*** > ***First***.

See also ACCRINTM 6.12.3, YEARFRAC 6.10.25

6.12.3 ACCRINTM

Summary: Calculates the accrued interest for securities that pay at maturity.

Syntax: ACCRINT(*DateParam* ***Issue*** ; *DateParam* ***Settlement*** ; *Number* ***Coupon*** ; *Number* ***Par*** [; *Basis* ***B*** = 0])

Returns: *Currency*

Constraints: ***Coupon*** > 0; ***Par*** > 0

Semantics: Calculates the accrued interest for securities that pay at maturity.

- ***Issue***: The security's issue or dated date.
- ***Settlement***: The security's maturity date.
- ***Coupon***: The security's annual coupon rate.
- ***Par***: The security's par value, that is, the principal to be paid at maturity.
- ***B***: Indicates the day-count convention to use in the calculation. 4.11.7

See also ACCRINT 6.12.2

6.12.4 AMORLINC

Summary: Calculates the amortization value for the French accounting system using linear depreciation (l'amortissement linéaire comptable) .

Syntax: AMORLINC(*Number* ***Cost*** ; *DateParam* ***PurchaseDate*** ; *DateParam* ***FirstPeriodEndDate*** ; *Number* ***Salvage*** ; *Integer* ***Period*** ; *Number* ***Rate*** [; *Basis* ***B*** = 0])

Returns: *Currency*

Constraints: ***Cost*** > 0; ***PurchaseDate*** ≤ ***FirstPeriodEndDate***; ***Salvage*** ≥ 0; ***Period*** ≥ 0; ***Rate*** > 0

Semantics: *Calculates the amortization value for the French accounting system using linear depreciation.*

- ***Cost***: The value of the asset at the date of aquisition.
- ***PurchaseDate***: The date of aquisition.
- ***FirstPeriodEndDate****:* The end date of the first depreciation period.
- ***Salvage***: The value of the asset at the end of the depreciation life time.
- ***Period***: Which period the depreciation should be calculated for.
- ***Rate***: The rate of depreciation.
- ***B***: Indicates the day-count convention to use in the calculation. 4.11.7

When ***Period*** = 0:

$$AMORLINC = Cost \cdot Rate \cdot YEARFRAC\left(PurchaseDate, FirstPeriodEndDate, Basis\right)$$

For full periods, where ***Period*** > 0, the depreciation is ***Cost*** * ***Rate***

$$t = \frac{Cost - Salvage}{Cost \cdot Rate}$$

For the last period, possibly a partial period: the depreciation = ***Cost*** - ***Salvage*** - accumulated-depreciation, where accumulated-depreciation is the sum of the depreciation in period 0 plus any full period depreciations.

$$AMORLINC = Cost \cdot Rate$$

When ***Period*** > depreciated life of the asset, i.e., when ***Period*** > (***Cost - Salvage***) ***/ (Cost * Rate***) then the depreciation is 0.

$$AMORLINC = 0$$

> **Note:** The behavior of this function is implementation-defined in cases where ***PurchaseDate*** = ***FirstPeriodEndDate***.

See also DB 6.12.13, DDB 6.12.14, YEARFRAC 6.10.25

6.12.5 COUPDAYBS

Summary: Calculates the number of days between the beginning of the coupon period that contains the settlement date and the settlement date.

Syntax: COUPDAYBS(*DateParam* ***Settlement*** ; *DateParam* ***Maturity*** ; *Integer* ***Frequency*** [; *Basis* ***B*** = 0])

Returns: *Number*

Constraints: ***Settlement*** < ***Maturity***

Frequency is one of the following values:

Table 9 - COUPDAYBS

Frequency	***Frequency of coupon payments***
1	Annual
2	Semiannual
4	Quarterly

Semantics: Calculate the number of days from the beginning of the coupon period to the settlement date.

- ***Settlement***: The settlement date.
- ***Maturity***: The maturity date.
- ***Frequency***: The number of coupon payments per year.
- ***B***: Indicates the day-count convention to use in the calculation. 4.11.7

See also COUPDAYS 6.12.6 , COUPDAYSNC 6.12.7 , COUPNCD 6.12.7 , COUPNUM 6.12.9 , COUPPCD 6.12.10

6.12.6 COUPDAYS

Summary: Calculates the number of days in a coupon period that contains the settlement date.

Syntax: COUPDAYS(*DateParam* ***Settlement*** ; *DateParam* ***Maturity*** ; *Integer* ***Frequency*** [; *Basis* ***B*** = 0])

Returns: *Number*

Constraints: ***Settlement*** < ***Maturity***

Frequency is one of the following values:

Table 10 - COUPDAYS

Frequency	***Frequency of coupon payments***
1	Annual
2	Semiannual
4	Quarterly

Semantics: Calculates the number of days in the coupon period containing the settlement date.

- ***Settlement***: The settlement date.
- ***Maturity***: The maturity date.
- ***Frequency***: The number of coupon payments per year.
- ***B***: Indicates the day-count convention to use in the calculation. 4.11.7

See also *COUPDAYBS 6.12.5 , COUPDAYSNC 6.12.7 , COUPNCD 6.12.7 ,* COUPNUM *6.12.9 , COUPPCD 6.12.10*

6.12.7 COUPDAYSNC

Summary: Calculates the number of days between a settlement date and the next coupon date.

Syntax: COUPDAYNC(*DateParam* ***Settlement*** ; *DateParam* ***Maturity*** ; *Integer* ***Frequency*** [; *Basis* ***B*** = 0])

Returns: *Number*

Constraints: ***Settlement*** < ***Maturity***

Frequency is one of the following values:

Table 11 - COUPDAYSNC

Frequency	***Frequency of coupon payments***
1	Annual
2	Semiannual
4	Quarterly

Semantics: Calculates the number of days between the settlement date and the next coupon date.

- ***Settlement***: The settlement date.
- ***Maturity***: The maturity date.
- ***Frequency***: The number of coupon payments per year.
- ***B***: Indicates the day-count convention to use in the calculation. 4.11.7

See also COUPDAYBS 6.12.5 , COUPDAYS 6.12.6 , COUPNCD 6.12.7 , COUPNUM 6.12.9 , COUPPCD 6.12.10

6.12.8 COUPNCD

Summary: Calculates the next coupon date following a settlement.

Syntax: COUPNCD(*DateParam* ***Settlement*** ; *DateParam* ***Maturity*** ; *Integer* ***Frequency*** [; *Basis* ***B*** = 0])

Returns: *Date*

Constraints: ***Settlement*** < ***Maturity***

Frequency is the number of coupon payments per year. ***Frequency*** is one of the following values:

Table 12 - COUPNCD

Frequency	***Frequency of coupon payments***
1	Annual
2	Semiannual
4	Quarterly

Semantics: Calculates the next coupon date after the ***Settlement*** date based on the ***Maturity*** (expiration) date of the asset, the ***Frequency*** of coupon payments and the day-count ***B***.

B indicates the day-count convention to use in the calculation. 4.11.7

See also: COUPDAYSNC 6.12.7

6.12.9 COUPNUM

Summary: Calculates the number of outstanding coupons between settlement and maturity dates.

Syntax: COUPNUM(*DateParam* ***Settlement*** ; *DateParam* ***Maturity*** ; *Integer* ***Frequency*** [; *Basis* ***B*** = 0])

Returns: *Number*

Constraints: ***Frequency*** is the number of coupon payments per year. ***Frequency*** is one of the following values:

Table 13 - COUPNUM

Frequency	***Frequency of coupon payments***
1	Annual
2	Semiannual
4	Quarterly

Semantics: Calculates the number of coupons in the interval between the ***Settlement*** and the ***Maturity*** (expiration) date of the asset, the ***Frequency*** of coupon payments and the day-count ***B***.

B indicates the day-count convention to use in the calculation. 4.11.7

See also COUPDAYBS 6.12.5, COUPDAYS 6.12.6, COUPDAYSNC 6.12.7, COUPNCD 6.12.7, COUPPCD 6.12.10

6.12.10 COUPPCD

Summary: Calculates the next coupon date prior a settlement.

Syntax: COUPPCD(*DateParam* ***Settlement*** ; *DateParam* ***Maturity*** ; *Integer* ***Frequency*** [; *Basis* ***B*** = 0])

Returns: *Date*

Constraints: ***Settlement*** < ***Maturity***

Frequency is the number of coupon payments per year. ***Frequency*** is one of the following values:

Table 14 - COUPPCD

Frequency	***Frequency of coupon payments***
1	Annual
2	Semiannual
4	Quarterly

Semantics: Calculates the next coupon date prior to the ***Settlement*** date based on the ***Maturity*** (expiration) date of the asset, the ***Frequency*** of coupon payments and the day-count ***B***.

B indicates the day-count convention to use in the calculation. 4.11.7

See also COUPDAYBS 6.12.5, COUPDAYS 6.12.6, COUPDAYSNC 6.12.7, COUPNCD 6.12.7, COUPNUM 6.12.9

6.12.11 CUMIPMT

Summary: Calculates a cumulative interest payment.

Syntax: CUMIPMT(*Number* ***Rate*** ; *Number* ***Periods*** ; *Number* ***Value*** ; *Integer* ***Start*** ; *Integer* ***End*** ; *Integer* ***Type***)

Returns: *Currency*

Constraints: ***Rate*** > 0; ***Value*** > 0; 1 ≤ ***Start*** ≤ ***End*** ≤ ***Periods***

Type is one of the following values:

Table 15 - CUMIPMT

Type	***Maturity date***
0	due at the end
1	due at the beginning

Semantics: Calculates the cumulative interest payment.

- ***Rate***: The interest rate per period.
- ***Periods***: The number of periods.
- ***Value***: The current value of the loan.
- ***Start***: The starting period.
- ***End***: The end period.
- ***Type***: The maturity date, the beginning or the end of a period.

$$CUMIPMT = \sum_{p=Start}^{End} IPMT\left(Rate\,,\,p\,,\,Periods\,,\,Value\,,\,0,Type\right)$$

See also IPMT 6.12.23, CUMPRINC 6.12.12

6.12.12 CUMPRINC

Summary: Calculates a cumulative principal payment.

Syntax: CUMPRINC(*Number* ***Rate*** ; *Number* ***Periods*** ; *Number* ***Value*** ; *Integer* ***Start*** ; *Integer* ***End*** ; *Integer* ***Type***)

Returns: *Currency*

Constraints: ***Type*** is one of the following values:

Table 16 - CUMPRINC

Type	***Maturity date***
0	due at the end
1	due at the beginning

Type	***Maturity date***
0	due at the end
1	due at the beginning

Semantics: Calculates the cumulative principal payment.

- ***Rate***: The interest rate per period.
- ***Periods***: The number of periods.
- ***Value***: The current value of the loan.
- ***Start***: The starting period.
- ***End***: The end period.
- ***Type***: The maturity date, the beginning or the end of a period.

$$CUMPRINC = \sum_{p=Start}^{End} PPMT\left(Rate\,,\,p\,,\,Periods\,,\,Value\,,\,0,\,Type\right)$$

See also PPMT 6.12.37 , CUMIPMT 6.12.11

6.12.13 DB

Summary: Compute the depreciation allowance of an asset.

Syntax: DB(*Number* ***Cost*** ; *Number* ***Salvage*** ; *Integer* ***LifeTime*** ; *Number* ***Period*** [; *Number* ***Month*** = 12])

Returns: *Currency*

Constraints: ***Cost*** > 0, ***Salvage*** ≥ 0, ***LifeTime*** > 0; ***Period*** > 0; 0 < ***Month*** < 13

Semantics: Calculate the depreciation allowance of an asset with an initial value of ***Cost***, an expected useful ***LifeTime***, and a final ***Salvage*** value at a specified ***Period*** of time, using the fixed-declining balance method. The parameters are:

- ***Cost***: the total amount paid for the asset.
- ***Salvage***: the salvage value at the end of the ***LifeTime***.
- ***LifeTime***: the number of periods that the depreciation will occur over. A positive integer.

- ***Period***: the time period for which you want to find the depreciation allowance, in the same units as ***LifeTime***.
- ***Month***: (optional) the number of months in the first year of depreciation, assumed to be 12, if not specified. If a value is specified for ***Month***, ***LifeTime*** and ***Period*** are assumed to be measured in years.

The rate is calculated as follows:

$$rate = 1 - \left(\frac{Salvage}{Cost} \right)^{\frac{1}{LifeTime}}$$

and is rounded to 3 decimals.

For the first period the residual value is

$$value_1 = Cost \left(1 - \frac{Month}{12} \cdot rate \right)$$

For all periods, where ***Period*** ≤ ***LifeTime***, the residual value is calculated by

$$value_{Period} = value_{Period-1} \cdot (1 - rate)$$

If ***Month*** was specified, the residual value for the period after ***LifeTime*** becomes

$$value_{LifeTime+1} = value_{LifeTime} \cdot \left(1 - \left(1 - \frac{Month}{12} \right) \cdot rate \right)$$

The depreciation allowance for the first period is

$$DB_1 = Cost - value_1$$

For all other periods the allowance is calculated by

$$DB_{period} = value_{period} - value_{period-1}$$

For all periods, where ***Period*** > ***LifeTime*** + 1 – INT(***Month*** / 12), the depreciation allowance is zero.

See also DDB 6.12.14, SLN 6.12.45, INT 6.17.2

6.12.14 DDB

Summary: Compute the amount of depreciation at a given period of time.

Syntax: DDB(*Number* ***Cost*** ; *Number* ***Salvage*** ; *Number* ***LifeTime*** ; *Number* ***Period*** [; *Number* ***DeclinationFactor*** = 2])

Returns: *Currency*

Constraints: ***Cost*** ≥ 0, ***Salvage*** ≥ 0, ***Salvage*** ≤ ***Cost***, 1 ≤ ***Period*** ≤ ***LifeTime***, ***DeclinationFactor*** > 0

Semantics: Compute the amount of depreciation of an asset at a given period of time. The parameters are:

- ***Cost***: the total amount paid for the asset.
- ***Salvage***: the salvage value at the end of the ***LifeTime***
- ***LifeTime***: the number of periods that the depreciation will occur over.
- ***Period***: the period for which a depreciation value is specified.
- ***DeclinationFactor***: the method of calculating depreciation, the rate at which the balance declines. Defaults to 2. If 2, double-declining balance is used.

To calculate depreciation, DDB uses a fixed rate. When ***DeclinationFactor*** = 2 this is the double-declining-balance method (because it is double the straight-line rate that would depreciate the asset to zero). The rate is given by:

$$rate = \frac{DeclinationFactor}{LifeTime}$$

The depreciation each period is calculated as

depreciation_of_period = MIN(book_value_at_start_of_ period * *rate*;
book_value_at_start_of_ period - ***Salvage***)

Thus the asset depreciates at *rate* until the book value is ***Salvage*** value.

$$BookValueAtStartOfPeriod_{Period} = Cost - \sum_{i=1}^{Period-1} DepreciationOfPeriod_i$$

To allow also non-integer ***Period*** values this algorithm may be used:

$$rate = \frac{DeclinationFactor}{LifeTime}$$
if $rate \geq 1$ then
{
  $rate = 1$
  if $Period = 1$ then
    $oldValue = Cost$
  else
    $oldValue = 0$
  endif
}
else
{
  $oldValue = Cost \cdot (1 - rate)^{Period-1}$
}
endif
$newValue = Cost \cdot (1 - rate)^{Period}$
if $newValue < Salvage$ then
  $DDB = oldValue - Salvage$
else
  $DDB = oldValue - newValue$
endif
if $DDB < 0$ then
  $DDB = 0$
endif

If ***Period*** is an Integer number, the relation between DDB and VDB is:
DDB(***Cost*** ; ***Salvage*** ; ***LifeTime*** ; ***Period*** ; ***DeclinationFactor***)
equals
VDB(***Cost*** ; ***Salvage*** ; ***LifeTime*** ; ***Period*** - 1 ; ***Period*** ; ***DeclinationFactor*** ; TRUE)

See also SLN 6.12.45, VDB 6.12.50, MIN 6.18.48

6.12.15 DISC

Summary: Returns the discount rate of a security.

Syntax: DISC(*DateParam* ***Settlement*** ; *DateParam* ***Maturity*** ; *Number* ***Price*** ; *Number* ***Redemption*** [; *Basis* ***B*** = 0])

Returns: *Percentage*

Constraints: ***Settlement*** < ***Maturity***

Semantics: *Calculates the discount rate of a security.*

- ***Settlement***: The settlement date of the security.
- ***Maturity***: The maturity date.
- ***Price***: The price of the security.
- ***Redemption***: The redemption value of the security.
- ***B***: Indicates the day-count convention to use in the calculation. 4.11.7

$$DISC = \frac{\frac{Redemption - Price}{Redemption}}{YEARFRAC\left(Settlement\ ,\ Maturity\ ,\ B\right)}$$

See also YEARFRAC 6.10.25

6.12.16 DOLLARDE

Summary: Converts a fractional dollar representation into a decimal representation.

Syntax: DOLLARDE(*Number* ***Fractional*** ; *Integer* ***Denominator***)

Returns: *Number*

Constraints: ***Denominator*** > 0

Semantics: Converts a fractional dollar representation into a decimal representation.

- ***Fractional***: Decimal fraction.
- ***Denominator***: The denominator of the fraction.

$$DOLLARDE = TRUNC\left(Fractional\right) + \frac{Fractional - TRUNC\left(Fractional\right)}{Denominator}$$

See also DOLLARFR 6.12.17 , TRUNC 6.17.8

6.12.17 DOLLARFR

Summary: Converts a decimal dollar representation into a fractional representation.

Syntax: DOLLARFR(*Number* ***Decimal*** ; *Integer* ***Denominator***)

Returns: *Number*

Constraints: ***Denominator*** > 0

Semantics: Converts a decimal dollar representation into a fractional representation.

- ***Decimal***: A decimal number.
- ***Denominator***: The denominator of the fraction.

$$DOLLARFR = TRUNC\left(Decimal\right) + \left(Decimal - TRUNC\left(Decimal\right)\right) \cdot Denominator$$

See also DOLLARDE 6.12.16, TRUNC 6.17.8

6.12.18 DURATION

Summary: Returns the Macaulay duration of a fixed interest security in years

Syntax: DURATION(*Date* ***Settlement*** ; *Date* ***Maturity*** ; *Number* ***Coupon*** ; *Number* ***Yield*** ; *Number* ***Frequency*** [; *Basis* ***B*** = 0])

Returns: *Number*

Constraints: ***Yield*** ≥0, ***Coupon*** ≥ 0, ***Settlement*** ≤ ***Maturity***; ***Frequency*** = 1, 2, 4

Semantics: Computes the Macaulay duration, given:

- ***Settlement****:* the date of purchase of the security
- ***Maturity***: the date when the security matures
- ***Coupon***: the annual nominal rate of interest
- ***Yield***: the annual yield of the security
- ***Frequency***: number of interest payments per year
- ***B***: Indicates the day-count convention to use in the calculation. 4.11.7

See also MDURATION 6.12.26

6.12.19 EFFECT

Summary: Returns the net annual interest rate for a nominal interest rate.

Syntax: EFFECT(*Number* ***Rate*** ; *Integer* ***Payments***)

Returns: *Number*

Constraints: ***Rate*** ≥ 0; ***Payments*** > 0

Semantics: Nominal interest refers to the amount of interest due at the end of a calculation period. Effective interest increases with the number of payments made. In other words, interest is often paid in installments (for example, monthly or quarterly) before the end of the calculation period.

- ***Rate***: The interest rate per period.
- ***Payments***: The number of payments per period.

$$EFFECT = \left(1 + \frac{Rate}{Payments}\right)^{Payments} - 1$$

See also NOMINAL 6.12.28

6.12.20 FV

Summary: Compute the future value (FV) of an investment.

Syntax: FV(*Number* ***Rate*** ; *Number* ***Nper*** ; *Number* ***Payment*** [; [*Number* ***Pv*** = 0] [; *Number* ***PayType*** = 0]])

Returns: *Currency*

Constraints: None.

Semantics: Computes the future value of an investment. The parameters are:

- ***Rate***: the interest rate per period.
- ***Nper***: the total number of payment periods.
- ***Payment***: the payment made in each period.
- ***Pv***: the present value; default is 0.
- ***PayType***: the type of payment, defaults to 0. It is 0 if payments are due at the end of the period; 1 if they are due at the beginning of the period.

See PV 6.12.41 for the equation this solves.

See also PV 6.12.41, NPER 6.12.29, PMT 6.12.36, RATE 6.12.42

6.12.21 FVSCHEDULE

Summary: Returns the accumulated value given starting capital and a series of interest rates.

Syntax: FVSCHEDULE(*Number* ***Principal*** ; *NumberSequence* ***Schedule***)

Returns: *Currency*

Constraints: None.

Semantics: Returns the accumulated value given starting capital and a series of interest rates, as follows:

$$Principle \cdot \prod_{i=1}^{N} (1 + Schedule[i])$$

See also PV 6.12.41, NPER 6.12.29, PMT 6.12.36, RATE 6.12.42

6.12.22 INTRATE

Summary: Computes the interest rate of a fully vested security.

Syntax: INTRATE(*Date* ***Settlement*** ; *Date* ***Maturity*** ; *Number* ***Investment*** ; *Number* ***Redemption*** [; *Basis Basis* = 0])

Returns: *Number*

Constraints: ***Settlement*** < ***Maturity***

Semantics: Calculates the annual interest rate that results when an item is purchased at the investment price and sold at the redemption price. No interest is paid on the investment. The parameters are:

- ***Settlement***: the date of purchase of the security.
- ***Maturity***: the date on which the security is sold.
- ***Investment***: the purchase price.
- ***Redemption***: the selling price.
- ***Basis***: indicates the day-count convention to use in the calculation. 4.11.7

The return value for this function is:

$$INTRATE = \frac{\frac{Redemption - Investment}{Investment}}{YEARFRAC(Settlement\ ;\ Maturity\ ;\ B)}$$

See also RECEIVED 6.12.43, YEARFRAC 6.10.25

6.12.23 IPMT

Summary: Returns the amount of an annuity payment going towards interest.

Syntax: IPMT(*Number* ***Rate*** ; *Number* ***Period*** ; *Number* ***Nper*** ; *Number* ***PV*** [; *Number* ***FV*** = 0 [; *Number* ***Type*** = 0]])

Returns: *Currency*

Constraints: None.

Semantics: Computes the interest portion of an amortized payment for a constant interest rate and regular payments. The interest payment is the interest rate multiplied by the balance at the beginning of the period. The parameters are:

- ***Rate***: The periodic interest rate.

- ***Period***: The period for which the interest payment is computed.
- ***Nper***: The total number of periods for which the payments are made
- ***PV***: The present value (e.g. The initial loan amount).
- ***FV***: The future value (optional) at the end of the periods. Zero if omitted.
- ***Type***: the due date for the payments (optional). Zero if omitted. If ***Type*** is 1, then payments are made at the beginning of each period. If ***Type*** is 0, then payments are made at the end of each period.

See also PPMT 6.12.37, PMT 6.12.36

6.12.24 IRR

Summary: Compute the internal rate of return for a series of cash flows.

Syntax: IRR(*NumberSequence* ***Values*** [; *Number* ***Guess*** = 0.1])

Returns: *Percentage*

Constraints: None.

Semantics: Compute the internal rate of return for a series of cash flows.

If provided, ***Guess*** is an estimate of the interest rate to start the iterative computation. If omitted, the value 0.1 (10%) is assumed.

The result of IRR is the rate at which the NPV() function will return zero with the given values.

There is no closed form for IRR. Evaluators may return an approximate solution using an iterative method, in which case the ***Guess*** parameter may be used to initialize the iteration. If the evaluator is unable to converge on a solution given a particular ***Guess***, it may return an Error.

See also NPV 6.12.30, RATE 6.12.42

6.12.25 ISPMT

Summary: Compute the interest payment of an amortized loan for a given period.

Syntax: ISPMT(*Number* ***Rate*** ; *Number* ***Period*** ; *Number* ***Nper*** ; *Number* ***Pv***)

Returns: *Currency*

Constraints: None.

Semantics: Computes the interest payment of an amortized loan for a given period. The parameters are:

- ***Rate***: the interest rate per period.
- ***Period***: the period for which the interest is computed
- ***Nper***: the total number of payment periods.
- ***Pv***: the amount of the investment

See also PV 6.12.41, FV 6.12.20, NPER 6.12.29, PMT 6.12.36, RATE 6.12.42

6.12.26 MDURATION

Summary: Returns the modified Macaulay duration of a fixed interest security in years.

Syntax: MDURATION(*Date* ***Settlement*** ; *Date* ***Maturity*** ; *Number* ***Coupon*** ; *Number* ***Yield*** ; *Number* ***Frequency*** [; *Basis* ***B*** = 0])

Returns: *Number*

Constraints: ***Yield*** ≥ 0, ***Coupon*** ≥ 0, ***Settlement*** ≤ ***Maturity***; ***Frequency*** = 1, 2, 4

Semantics: Computes the modified Macaulay duration, given:

- ***Settlement***: the date of purchase of the security
- ***Maturity***: the date when the security matures
- ***Coupon***: the annual nominal rate of interest
- ***Yield***: the annual yield of the security
- ***Frequency***: number of interest payments per year
- ***B***: Indicates the day-count convention to use in the calculation. 4.11.7

The modified duration is computed as follows:

$$duration = DURATION(Settlement, Maturity, Coupon, Yield, Frequency, B)$$

$$MDURATION = \frac{duration}{1 + \left(\frac{Yield}{Frequency}\right)}$$

See also DURATION 6.12.18

6.12.27 MIRR

Summary: Returns the modified internal rate of return (IRR) of a series of periodic investments.

Syntax: MIRR(*Array* ***Values*** ; *Number* ***Investment*** ; *Number* ***ReinvestRate***)

Returns: *Percentage*

Constraints: ***Values*** shall contain at least one positive value and at least one negative value.

Semantics: ***Values*** is a series of periodic income (positive values) and payments (negative values) at regular intervals (Text and Empty cells are ignored). ***Investment*** is the rate of interest of the payments (negative values); ***ReinvestRate*** is the rate of interest of the reinvestment (positive values).

Computes the modified internal rate of return, which is:

$$\left(\frac{-NPV(ReinvestRate, Values>0)*(1+ReinvestRate)^{n}}{NPV(Investment; Values<0)*(1+Investment)}\right)^{\left(\frac{1}{n-1}\right)} - 1$$

where N is the number of incomes and payments in ***Values*** (total).

See also IRR 6.12.24, NPV 6.12.30

6.12.28 NOMINAL

Summary: Compute the annual nominal interest rate.

Syntax: NOMINAL(*Number* ***EffectiveRate*** ; *Integer* ***CompoundingPeriods***)

Returns: *Number*

Constraints: ***EffectiveRate*** > 0 , ***CompoundingPeriods*** > 0

Semantics: Returns the annual nominal interest rate based on the effective rate and the number of compounding periods in one year. The parameters are:

- ***EffectiveRate***: effective rate
- ***CompoundingPeriods***: the compounding periods per year

Suppose that P is the present value, m is the compounding periods per year, the future value after one year is

$$P*\left(1+\frac{NOMINAL}{m}\right)^{m}$$

The mapping between nominal rate and effective rate is

$$EFFECT = \left(1 + \frac{NOMINAL}{m}\right)^{m} - 1$$

See also EFFECT 6.12.19

6.12.29 NPER

Summary: Compute the number of payment periods for an investment.

Syntax: NPER(*Number* ***Rate*** ; *Number* ***Payment*** ; *Number* ***Pv*** [; [*Number* ***Fv*** = 0] [; *Number* ***PayType*** = 0]])

Returns: *Number*

Constraints: None.

Semantics: Computes the number of payment periods for an investment. The parameters are:

- ***Rate***: the constant interest rate.
- ***Payment***: the payment made in each period.
- ***Pv***: the present value of the investment.
- ***Fv***: the future value; default is 0.
- ***PayType***: the type of payment, defaults to 0. It is 0 if payments are due at the end of the period; 1 if they are due at the beginning of the period.

If ***Rate*** is 0, then NPER solves this equation:

$$Pv = -Fv - (Payment * NPER)$$

If ***Rate*** is non-zero, then NPER solves this equation:

$$0 = Pv \cdot (1 + Rate)^{NPER} + \frac{Payment \cdot (1 + Rate \cdot PaymentType) \cdot ((1 + Rate)^{NPER} - 1)}{Rate} + Fv$$

Evaluators claiming to support the “Medium” or “Large” set shall support negative rates; evaluators only claiming to support the “Small” set need not.

See also FV 6.12.20, RATE 6.12.42, PMT 6.12.36, PV 6.12.41

6.12.30 NPV

Summary: Compute the net present value (NPV) for a series of periodic cash flows.

Syntax: NPV(*Number* ***Rate*** ; { *NumberSequenceList* ***Values*** }$^{+}$)

Returns: *Currency*

Constraints: None.

Semantics: Computes the net present value for a series of periodic cash flows with the discount rate ***Rate***. ***Values*** should be positive if they are received as income, and negative if the amounts are paid as outgo. Because the result is affected by the order of values, evaluators shall evaluate arguments in the order given and range reference and array arguments row-wise starting from top left.

If N is the number of values in ***Values***, the formula for NPV is:

$$NPV = \sum_{i=1}^{N} \frac{Values_i}{(1 + Rate)^i}$$

See also FV 6.12.20, IRR 6.12.24, NPER 6.12.29, PMT 6.12.36, PV 6.12.41, XNPV 6.12.52

6.12.31 ODDFPRICE

Summary: Compute the value of a security per 100 currency units of face value. The security has an irregular first interest date.

Syntax: ODDFPRICE(*DateParam* ***Settlement*** ; *DateParam* ***Maturity*** ; *DateParam* ***Issue*** ; *DateParam* ***First*** ; *Number* ***Rate*** ; *Number* ***Yield*** ; *Number* ***Redemption*** ; *Number* ***Frequency*** [; *Basis* ***B*** = 0])

Returns: *Number*

Constraints: ***Rate***, ***Yield***, and ***Redemption*** should be greater than 0.

Semantics: The parameters are

- ***Settlement***: the settlement/purchase date of the security
- ***Maturity***: the maturity/expiry date of the security
- ***Issue***: the issue date of the security
- ***First***: the first coupon date of the security
- ***Rate***: the interest rate of the security
- ***Yield***: the annual yield of the security
- ***Redemption***: the redemption value per 100 currency units face value
- ***Frequency***: the number of interest payments per year. 1 = annual; 2 = semiannual; 4 = quarterly.
- ***B***: indicates the day-count convention to use in the calculation. 4.11.7

See also ODDLPRICE 6.12.33 , ODDFYIELD 6.12.32

6.12.32 ODDFYIELD

Summary: Compute the yield of a security per 100 currency units of face value. The security has an irregular first interest date.

Syntax: ODDFYIELD(*DateParam* ***Settlement*** ; *DateParam* ***Maturity*** ; *DateParam* ***Issue*** ; *DateParam* ***First*** ; *Number* ***Rate*** ; *Number* ***Price*** ; *Number* ***Redemption*** ; *Number* ***Frequency*** [; *Basis* ***B*** = 0])

Returns: *Number*

Constraints: ***Rate***, ***Price***, and ***Redemption*** should be greater than 0. ***Maturity*** > ***First*** > ***Settlement*** > ***Issue***.

Semantics: The parameters are

- ***Settlement***: the settlement/purchase date of the security
- ***Maturity***: the maturity/expiry date of the security
- ***Issue***: the issue date of the security
- ***First***: the first coupon date of the security
- ***Rate***: the interest rate of the security
- ***Price***: the price of the security
- ***Redemption***: the redemption value per 100 currency units face value
- ***Frequency***: the number of interest payments per year. 1 = annual; 2 = semiannual; 4 = quarterly.
- ***B***: indicates the day-count convention to use in the calculation. 4.11.7

See also ODDLYIELD 6.12.34 , ODDFPRICE 6.12.31

6.12.33 ODDLPRICE

Summary: Compute the value of a security per 100 currency units of face value. The security has an irregular last interest date.

Syntax: ODDLPRICE(*DateParam* ***Settlement*** ; *DateParam* ***Maturity*** ; *DateParam* ***Last*** ; *Number* ***Rate*** ; *Number* ***AnnualYield*** ; *Number* ***Redemption*** ; *Number* ***Frequency*** [; *Basis* ***B*** = 0])

Returns: *Number*

Constraints: ***Rate***, ***AnnualYield***, and ***Redemption*** should be greater than 0. The ***Maturity*** date should be greater than the ***Settlement*** date, and the ***Settlement*** should be greater than the last interest date.

Semantics: The parameters are

- ***Settlement***: the settlement/purchase date of the security
- ***Maturity***: the maturity/expiry date of the security
- ***Last***: the last interest date of the security
- ***Rate***: the interest rate of the security
- ***AnnualYield***: the annual yield of the security
- ***Redemption***: the redemption value per 100 currency units face value
- ***Frequency***: the number of interest payments per year. 1 = annual; 2 = semiannual; 4 = quarterly
- ***B***: indicates the day-count convention to use in the calculation. 4.11.7

See also ODDFPRICE 6.12.31

6.12.34 ODDLYIELD

Summary: Compute the yield of a security which has an irregular last interest date.

Syntax: ODDLYIELD(*DateParam* ***Settlement*** ; *DateParam* ***Maturity*** ; *DateParam* ***Last*** ; *Number* ***Rate*** ; *Number* ***Price*** ; *Number* ***Redemption*** ; *Number* ***Frequency*** [; *Basis* ***B*** = 0])

Returns: *Number*

Constraints: ***Rate***, ***Price***, and ***Redemption*** should be greater than 0.

Semantics: The parameters are

- ***Settlement***: the settlement/purchase date of the security
- ***Maturity***: the maturity/expiry date of the security
- ***Last***: the last interest date of the security
- ***Rate***: the interest rate of the security
- ***Price***: the price of the security
- ***Redemption***: the redemption value per 100 currency units face value
- ***Frequency***: the number of interest payments per year. 1 = annual; 2 = semiannual; 4 = quarterly.
- ***B***: indicates the day-count convention to use in the calculation. 4.11.7

See also ODDLPRICE 6.12.33 , ODDFYIELD 6.12.32

6.12.35 PDURATION

Summary: Returns the number of periods required by an investment to realize a specified value.

Syntax: PDURATION(*Number* ***Rate*** ; *Number* ***CurrentValue*** ; *Number* ***SpecifiedValue***)

Returns: *Number*

Constraints: ***Rate*** > 0; ***CurrentValue*** > 0; ***SpecifiedValue*** > 0

Semantics: *Calculates the number of periods for attaining a certain value* ***SpecifiedValue****, starting from* ***CurrentValue*** *and using the interest rate* ***Rate****.*

- ***Rate***: The interest rate per period.
- ***CurrentValue***: The current value of the investment.
- ***SpecifiedValue***: The value, that should be reached.

$$PDURATION = \frac{\log(SpecifiedValue) - \log(CurrentValue)}{\log(Rate + 1)}$$

See also DURATION 6.12.18

6.12.36 PMT

Summary: Compute the payment made each period for an investment.

Syntax: PMT(*Number* ***Rate*** ; *Integer* ***Nper*** ; *Number* ***Pv*** [; [*Number* ***Fv*** = 0] [; *Number* ***PayType*** = 0]])

Returns: *Currency*

Constraints: ***Nper*** > 0

Semantics: Computes the payment made each period for an investment. The parameters are:

- ***Rate***: the interest rate per period.
- ***Nper***: the total number of payment periods.
- ***Pv***: the present value of the investment.
- ***Fv***: the future value of the investment; default is 0.
- ***PayType***: the type of payment, defaults to 0. It is 0 if payments are due at the end of the period; 1 if they are due at the beginning of the period.

If ***Rate*** is 0, the following equation is solved:

$$Pv = -Fv - (PMT * Nper)$$

If ***Rate*** is nonzero, then PMT solves this equation:

$$0 = Pv \cdot (1 + Rate)^{Nper} + \frac{PMT \cdot (1 + Rate \cdot PayType) \cdot ((1 + Rate)^{Nper} - 1)}{Rate} + Fv$$

See also FV 6.12.20, NPER 6.12.29, PV 6.12.41, RATE 6.12.42

6.12.37 PPMT

Summary: Calculate the payment for a given period on the principal for an investment at a given interest rate and constant payments.

Syntax: PPMT(*Number* ***Rate*** ; *Integer* ***Period*** ; *Integer* ***Nper*** ; *Number* ***Present*** [; *Number* ***Future*** = 0 [; *Number* ***Type*** = 0]])

Returns: *Number*

Constraints: ***Rate*** and ***Present*** should be greater than 0. 0 < ***Period*** < ***Nper***.

Semantics: The parameters are:

- ***Rate***: the interest rate.

- ***Period***: the given period that the payment returned is for.
- ***Nper***: the total number of periods.
- ***Present***: the present value.
- ***Future***: optional, the future value specified after ***Nper*** periods. The default value is 0.
- ***Type***: optional, 0 or 1, respectively for payment at the end or at the beginning of a period. The default value is 0.

See also PMT 6.12.36

6.12.38 PRICE

Summary: Calculates a quoted price for an interest paying security, per 100 currency units of face value.

Syntax: PRICE(*DateParam* ***Settlement*** ; *DateParam* ***Maturity*** ; *Number* ***Rate*** ; *Number* ***AnnualYield*** ; *Number* ***Redemption*** ; *Number* ***Frequency*** [; *Basis* ***Bas*** = 0])

Returns: *Number*

Constraints: ***Rate***, ***AnnualYield***, and ***Redemption*** should be greater than 0; ***Frequency*** = 1, 2 or 4.

Semantics: If A is the number of days from the ***Settlement*** date to next coupon date, B is the number of days of the coupon period that the ***Settlement*** is in, C is the number of coupons between ***Settlement*** date and ***Redemption*** date, D is the number of days from beginning of coupon period to ***Settlement*** date, then PRICE is calculated as

$$PRICE = \frac{Redemption}{\left(1+\frac{Yield}{Frequency}\right)^{C-1+\frac{A}{B}}} + \sum_{k=1}^{C} \frac{\frac{100*Rate}{Frequency}}{\left(1+\frac{Yield}{Frequency}\right)^{k-1+\frac{A}{B}}} - 100*\frac{Rate}{Frequency}*\frac{D}{B}$$

The parameters are:

- ***Settlement***: the settlement/purchase date of the security.
- ***Maturity***: the maturity/expiry date of the security.
- ***Rate***: the interest rate of the security.
- ***AnnualYield***: a measure of the annual yield of a security (compounded at each interest payment).
- ***Redemption***: the redemption value per 100 currency units face value.
- ***Frequency***: the number of interest payments per year. 1 = annual; 2 = semiannual; 4 = quarterly.
- ***Bas***: indicates the day-count convention to use in the calculation. 4.11.7

See also PRICEDISC 6.12.39, PRICEMAT 6.12.40

6.12.39 PRICEDISC

Summary: Calculate the price of a security with a discount per 100 currency units of face value.

Syntax: PRICEDISC(*DateParam* ***Settlement*** ; *DateParam* ***Maturity*** ; *Number* ***Discount*** ; *Number* ***Redemption*** [; *Basis* ***B*** = 0])

Returns: *Number*

Constraints: ***Discount*** and ***Redemption*** should be greater than 0.

Semantics: The parameters are:

- ***Settlement***: the settlement/purchase date of the security.
- ***Maturity***: the maturity/expiry date of the security.
- ***Discount***: the discount rate of the security.
- ***Redemption***: the redemption value per 100 currency units face value.
- ***B***: indicates the day-count convention to use in the calculation. 4.11.7

See also PRICE 6.12.38, PRICEMAT 6.12.40, YIELDDISC 6.12.54

6.12.40 PRICEMAT

Summary: Calculate the price per 100 currency units of face value of the security that pays interest on the maturity date.

Syntax: PRICEMAT(*DateParam* ***Settlement*** ; *DateParam* ***Maturity*** ; *DateParam* ***Issue*** ; *Number* ***Rate*** ; *Number* ***AnnualYield*** [; *Basis* ***B*** = 0])

Returns: *Number*

Constraints: ***Settlement*** < ***Maturity***, ***Rate*** ≥ 0, ***AnnualYield*** ≥ 0

Semantics: The parameters are:

- ***Settlement***: the settlement/purchase date of the security.
- ***Maturity***: the maturity/expiry date of the security.
- ***Issue***: the issue date of the security.
- ***Rate***: the interest rate of the security.
- ***AnnualYield***: the annual yield of the security.
- ***B***: indicates the day-count convention to use in the calculation. 4.11.7

If both, ***Rate*** and ***AnnualYield***, are 0, the return value is 100.

See also PRICEDISC 6.12.39, PRICEMAT 6.12.40

6.12.41 PV

Summary: Compute the present value (PV) of an investment.

Syntax: PV(*Number* ***Rate*** ; *Number* ***Nper*** ; *Number* ***Payment*** [; [*Number* ***Fv*** = 0] [; *Number* ***PayType*** = 0]])

Returns: *Currency*

Constraints: None.

Semantics: Computes the present value of an investment. The parameters are:

- ***Rate***: the interest rate per period.
- ***Nper***: the total number of payment periods.
- ***Payment***: the payment made in each period.
- ***Fv***: the future value; default is 0.
- ***PayType***: the type of payment, defaults to 0. It is 0 if payments are due at the end of the period; 1 if they are due at the beginning of the period.

If ***Rate*** is 0, then:

$$PV = -Fv - (Payment * Nper)$$

If ***Rate*** is nonzero, then PV solves this equation:

$$0 = PV \cdot (1 + Rate)^{Nper} + \frac{Payment \cdot (1 + Rate \cdot PayType) \cdot ((1 + Rate)^{Nper} - 1)}{Rate} + Fv$$

See also FV 6.12.20, NPER 6.12.29, PMT 6.12.36, RATE 6.12.42

6.12.42 RATE

Summary: Compute the interest rate per period of an investment.

Syntax: RATE(*Number* ***Nper*** ; *Number* ***Payment*** ; *Number* ***Pv*** [; [*Number* ***Fv*** = 0] [; [*Number* ***PayType*** = 0] [; *Number* ***Guess*** = 0.1]]])

Returns: *Percentage*

Constraints: If ***Nper*** is 0 or less than 0, the result is an Error.

Semantics: Computes the interest rate of an investment. The parameters are:

- ***Nper***: the total number of payment periods.
- ***Payment***: the payment made in each period.
- ***Pv***: the present value of the investment.
- ***Fv***: the future value; default is 0.
- ***PayType***: the type of payment, defaults to 0. It is 0 if payments are due at the end of the period; 1 if they are due at the beginning of the period.
- ***Guess***: An estimate of the interest rate to start the iterative computation. If omitted, 0.1 (10%) is assumed.

RATE solves this equation:

$$\begin{aligned} 0 &= Fv + Pv \cdot (1 + Rate)^{Nper} \\ &+ \frac{Payment \cdot (1 + Rate \cdot PayType) \cdot \left((1 + Rate)^{Nper} - 1\right)}{Rate} \end{aligned}$$

See also FV 6.12.20, NPER 6.12.29, PMT 6.12.36, PV 6.12.41

6.12.43 RECEIVED

Summary: Calculates the amount received at maturity for a zero coupon bond.

Syntax: RECEIVED(*DateParam* ***Settlement*** ; *DateParam* ***Maturity*** ; *Number* ***Investment*** ; *Number* ***Discount*** [; *Basis* ***B*** = 0])

Returns: *Number*

Constraints: ***Investment*** and ***Discount*** should be greater than 0, ***Settlement*** < ***Maturity***

Semantics: The parameters are:

Settlement: the settlement/purchase date of the security

- ***Maturity***: the maturity/expiry date of the security
- ***Investment***: the amount of investment in the security
- ***Discount***: the discount rate of the security
- ***B***: indicates the day-count convention to use in the calculation. 4.11.7

The returned value is:

$$RECEIVED = \frac{Investment}{1 - Discount \cdot YEARFRAC\left(Settlement\ ;\ Maturity\ ;\ B\right)}$$

See also YEARFRAC 6.10.25

6.12.44 RRI

Summary: Returns an equivalent interest rate when an investment increases in value.

Syntax: RRI(*Number* ***Nper*** ; *Number* ***Pv*** ; *Number* ***Fv***)

Returns: *Percentage*

Constraints: ***Nper*** > 0

Semantics: Returns the interest rate given ***Nper*** (the number of periods), ***Pv*** (present value), and ***Fv*** (future value), calculated as follows:

$$\left(\frac{Fv}{Pv}\right)^{(1/Nper)} - 1$$

See also FV 6.12.20, NPER 6.12.29, PMT 6.12.36, PV 6.12.41, RATE 6.12.42

6.12.45 SLN

Summary: Compute the amount of depreciation at a given period of time using the straight-line depreciation method.

Syntax: DDB(*Number* ***Cost*** ; *Number* ***Salvage*** ; *Number* ***LifeTime***)

Returns: *Currency*

Constraints: None.

Semantics: Compute the amount of depreciation of an asset at a given period of time using straight-line depreciation. The parameters are:

- ***Cost***: the total amount paid for the asset.
- ***Salvage***: the salvage value at the end of the ***LifeTime*** (often 0)
- ***LifeTime***: the number of periods that the depreciation will occur over. A positive integer.

For alternative methods to compute depreciation, see DDB 6.12.14.

6.12.46 SYD

Summary: Compute the amount of depreciation at a given period of time using the Sum-of-the-Years'-Digits method.

Syntax: SYD(*Number* ***Cost*** ; *Number* ***Salvage*** ; *Number* ***LifeTime*** ; *Number* ***Period***)

Returns: *Currency*

Constraints: None.

Semantics: Compute the amount of depreciation of an asset at a given period of time using the Sum-of-the-Years'-Digits method. The parameters are:

- ***Cost***: the total amount paid for the asset.
- ***Salvage***: the salvage value at the end of the ***LifeTime*** (often 0).
- ***LifeTime***: the number of periods that the depreciation will occur over. A positive integer.
- ***Period***: the period for which the depreciation value is specified.

$$SYD = \frac{(Cost - Salvage) \cdot (LifeTime + 1 - Period) \cdot 2}{(LifeTime + 1) \cdot LifeTime}$$

For other methods of computing depreciation, see DDB 6.12.14.

See also SLN 6.12.45

6.12.47 TBILLEQ

Summary: Compute the bond-equivalent yield for a treasury bill.

Syntax: TBILLEQ(*DateParam* ***Settlement*** ; *DateParam* ***Maturity*** ; *Number* ***Discount***)

Returns: *Number*

Constraints: The maturity date should be less than one year beyond settlement date. ***Discount*** is any positive value.

Semantics: The parameters are defined as:

- ***Settlement***: the settlement/purchase date of the treasury bill.
- ***Maturity***: the maturity/expiry date of the treasury bill.
- ***Discount***: the discount rate of the treasury bill.

TBILLEQ is calculated as

$$TBILLEQ = \frac{365 \cdot rate}{360 - (rate \cdot DSM)}$$

where *DSM* is the number of days between settlement and maturity computed according to the 360 days per year basis (basis 2, 4.11.7)

See also TBILLPRICE 6.12.48, TBILLYIELD 6.12.49

6.12.48 TBILLPRICE

Summary: Compute the price per 100 face value for a treasury bill.

Syntax: TBILLPRICE(*DateParam* ***Settlement*** ; *DateParam* ***Maturity*** ; *Number* ***Discount***)

Returns: *Number*

Constraints: The maturity date should be less than one year beyond settlement. ***Discount*** is any positive value.

Semantics: The parameters are:

- ***Settlement***: the settlement/purchase date of the treasury bill.
- ***Maturity***: the maturity/expiry date of the treasury bill.
- ***Discount***: the discount rate of the treasury bill.

See also TBILLEQ 6.12.47, TBILLYIELD 6.12.49

6.12.49 TBILLYIELD

Summary: Compute the yield for a treasury bill.

Syntax: TBILLYIELD(*DateParam* ***Settlement*** ; *DateParam* ***Maturity*** ; *Number* ***Price***)

Returns: *Number*

Constraints: The maturity date should be less than one year beyond settlement. ***Price*** is any positive value.

Semantics: The parameters are:

- ***Settlement***: the settlement/purchase date of the treasury bill.
- ***Maturity***: the maturity/expiry date of the treasury bill.
- ***Price***: the price of the treasury bill per 100 face value

See also TBILLEQ 6.12.47, TBILLPRICE 6.12.48

6.12.50 VDB

Summary: Calculates the depreciation allowance of an asset with an initial value, an expected useful life, and a final value of salvage for a period specified, using the variable-rate declining balance method..

Syntax: VDB(*Number* ***Cost*** ; *Number* ***Salvage*** ; *Number* ***LifeTime*** ; *Number* ***StartPeriod*** ; *Number* ***EndPeriod*** [; *Number* ***DepreciationFactor*** = 2 [; *Logical* ***NoSwitch*** = FALSE]])

Returns: *Number*

Constraints: ***Salvage*** < ***Cost***, ***LifeTime*** > 0, 0 ≤ ***StartPeriod*** ≤ ***LifeTime***, ***StartPeriod*** ≤ ***EndPeriod*** ≤ ***LifeTime***, ***DepreciationFactor*** ≥ 0

Semantics: The parameters are:

- ***Cost*** is the amount paid for the asset. ***Cost*** can be any value greater than ***Salvage***.
- ***Salvage*** is the value of the asset at the end of its life. ***Salvage*** can be any value.
- ***LifeTime*** is the number of periods the asset takes to depreciate to its salvage value. ***LifeTime*** can be any value greater than 0.
- ***StartPeriod*** is the point in the asset's life when you want to begin calculating depreciation. ***StartPeriod*** can be any value greater than or equal to 0, but cannot be greater than ***LifeTime***.
- ***EndPeriod*** is the point in the asset's life when you want to stop calculating depreciation. ***EndPeriod*** can be any value greater than ***StartPeriod***.
- ***StartPeriod*** and ***EndPeriod*** correspond to the asset's life, relative to the fiscal period. For example, if you want to find the first year's depreciation of an asset purchased at the beginning of the second quarter of a fiscal year, ***StartPeriod*** would be 0 and ***EndPeriod*** would be 0.75 (1 minus 0.25 of a year).

VDB allows for the use of an initialPeriod option to calculate depreciation for the period the asset is placed in service. VDB uses the fractional part of ***StartPeriod*** and ***EndPeriod*** to determine the initialPeriod option. If both ***StartPeriod*** and ***EndPeriod*** have fractional parts, then VDB uses the fractional part of ***StartPeriod***.

DepreciationFactor is an optional argument that specifies the percentage of straight-line depreciation you want to use as the depreciation rate. If you omit this argument, VDB uses 2, which is the double-declining balance rate. ***DepreciationFactor*** can be any value greater than or equal to 0; commonly used rates are 1.25, 1.50, 1.75, and 2.

NoSwitch is an optional argument that you include if you do not want VDB to switch to straight-line depreciation for the remaining useful life. Normally, declining-balance switches to such a straight-line calculation when it is greater than the declining-balance calculation.

If ***NoSwitch*** is FALSE or omitted, VDB automatically switches to straight-line depreciation when that is greater than declining-balance depreciation. If ***NoSwitch*** is TRUE, VDB never switches to straight-line depreciation.

See also DDB 6.12.14, SLN 6.12.45

6.12.51 XIRR

Summary: Compute the internal rate of return for a non-periodic series of cash flows.

Syntax: XIRR(*NumberSequence* ***Values*** ; *DateSequence* ***Dates*** [; *Number* ***Guess*** = 0.1])

Returns: *Number*

Constraints: The size of ***Values*** and ***Dates*** are equal. ***Values*** contains at least one positive and one negative cash flow.

Semantics: Compute the internal rate of return for a series of cash flows which is not necessarily periodic. The parameters are:

- ***Values***: a series of cash flows. The first cash-flow amount is a negative number that represents the investment. The later cash flows are discounted based on the annual discount rate and the timing of the flow. The series of cash flow should contain at least one positive and one negative value.
- ***Dates***: a series of dates that corresponds to values. The first date indicates the start of the cash flows. The range of ***Values*** and ***Dates*** shall be the same size.
- ***Guess***: If provided, ***Guess*** is an estimate of the interest rate to start the iterative computation. If omitted, the value 0.1 (10%) is assumed. The result of XIRR is the rate at which the XNPV() function will return zero with the given cash flows. There is no closed form for XIRR. Implementations may return an approximate solution using an iterative method, in which case the ***Guess*** parameter may be used to initialize the iteration. If the implementation is unable to converge on a solution given a particular ***Guess***, it may return an error.

See also IRR 6.12.24, XNPV 6.12.52

6.12.52 XNPV

Summary: Compute the net present value of a series of cash flows.

Syntax: XNPV(*Number* ***Rate*** ; *Reference*|*Array* ***Values*** ; *Reference*|*Array* ***Dates***)

Returns: *Number*

Constraints:

Number of elements in ***Values*** equals number of elements in ***Dates***.

All elements of ***Values*** are of type Number.

All elements of ***Dates*** are of type Number.

All elements of ***Dates*** ≥ ***Dates***[1]

Semantics: Compute the net present value for a series of cash flows which is not necessarily periodic. The parameters are:

- ***Rate***: discount rate. The value should be greater than -1.
- ***Values***: a series of cash flows. The first cash-flow amount is a negative number that represents the investment. The later cash flows are discounted based on the annual discount rate and the timing of the flow. The series of cash flow should contain at least one positive and one negative value.
- ***Dates***: a series of dates that corresponds to values. The first date indicates the start of the cash flows. If the dimensions of the ***Values*** and ***Dates*** arrays differ, evaluators shall match value and date pairs row-wise starting from top left.

With N being the number of elements in ***Values*** and ***Dates*** each, the formula is:

$$XNPV = \sum_{i=1}^{N} \frac{Values_i}{(1+Rate)^{\frac{Dates_i - Dates_1}{365}}}$$

See also NPV 6.12.30

6.12.53 YIELD

Summary: Calculate the yield of a bond.

Syntax: YIELD(*DateParam* ***Settlement*** ; *DateParam* ***Maturity*** ; *Number* ***Rate*** ; *Number* ***Price*** ; *Number* ***Redemption*** ; *Number* ***Frequency*** [; *Basis* ***B*** = 0])

Returns: *Number*

Constraints: ***Rate***, ***Price***, and ***Redemption*** should be greater than 0.

Semantics: The parameters are:

- ***Settlement***: the settlement/purchase date of the bond.
- ***Maturity***: the maturity/expiry date of the bond.
- ***Rate***: the interest rate of the bond.
- ***Price***: the price of the bond per 100 currency units face value.
- ***Redemption***: the redemption value of the bond per 100 currency units face value.
- ***Frequency***: the number of interest payments per year. 1 = annual; 2 = semiannual; 4 = quarterly.
- ***B***: indicates the day-count convention to use in the calculation. 4.11.7

See also PRICE 6.12.38, YIELDDISC 6.12.54, YIELDMAT 6.12.55

6.12.54 YIELDDISC

Summary: Calculate the yield of a discounted security per 100 currency units of face value.

Syntax: YIELDDISC(*DateParam* ***Settlement*** ; *DateParam* ***Maturity*** ; *Number* ***Price*** ; *Number* ***Redemption*** [; *Basis* ***B*** = 0])

Returns: *Number*

Constraints: ***Price*** and ***Redemption*** should be greater than 0.

Semantics: The parameters are:

- ***Settlement***: the settlement/purchase date of the security.
- ***Maturity***: the maturity/expiry date of the security.
- ***Price***: the price of the security per 100 currency units face value.
- ***Redemption***: the redemption value per 100 currency units face value.
- ***B***: indicates the day-count convention to use in the calculation. 4.11.7

The return value is

$$YIELDDISC = \frac{\frac{Redemption}{Price} - 1}{YEARFRAC(Settlement\ ;\ Maturity\ ;\ Basis)}$$

See also PRICEDISC 6.12.39, YEARFRAC 6.10.25

6.12.55 YIELDMAT

Summary: Calculate the yield of the security that pays interest on the maturity date.

Syntax: YIELDMAT(*DateParam* ***Settlement*** ; *DateParam* ***Maturity*** ; *DateParam* ***Issue*** ; *Number* ***Rate*** ; *Number* ***Price*** [; *Basis* ***B*** = 0])

Returns: *Number*

Constraints: ***Rate*** and ***Price*** should be greater than 0.

Semantics: The parameters are:

- ***Settlement***: the settlement/purchase date of the security.
- ***Maturity***: the maturity/expiry date of the security.
- ***Issue***: the issue date of the security.
- ***Rate***: the interest rate of the security.
- ***Price***: the price of the security per 100 currency units face value.

- ***B***: indicates the day-count convention to use in the calculation. 4.11.7

See also PRICE 6.12.38, YIELD 6.12.53, YIELDDISC 6.12.54

6.13 Information Functions

6.13.1 General

Information functions provide information about a data value, the spreadsheet, or underlying environment, including special functions for converting between data types.

6.13.2 AREAS

Summary: Returns the number of areas in a given list of references.

Syntax: AREAS(*ReferenceList* ***R***)

Returns: *Number*

Constraints: None

Semantics: Returns the number of areas in the reference list ***R***.

See also Infix Operator Reference Concatenation 6.4.13, INDEX 6.14.6

6.13.3 CELL

Summary: Returns information about position, formatting or contents in a reference.

Syntax: CELL(*Text* ***Info_Type*** [; *Reference* ***R***])

Returns: Information about position, formatting properties or content

Constraints: None

Semantics: The parameters are

- ***Info_Type***: the text string which specifies the type of information. Please refer to Table 17 - CELL.
- ***R*** : if ***R*** is a reference to a cell, it is the cell whose information will be returned; if ***R*** is a reference to a range, the top-left cell in the range is the selected one; if ***R*** is omitted, the current cell is used.

Table 17 - CELL

Info_Type	***Comment***
COL	Returns the column number of the cell.
ROW	Returns the row number of the cell.
SHEET	Returns the sheet number of the cell.
ADDRESS	Returns the absolute address of the cell. The sheet name is included if given in the reference and does not reference the same sheet as the sheet the expression is evaluated upon. For an external reference a `Source` as specified in the syntax rules for `References` 5.8 is included.
FILENAME	Returns the file name of the file that contains the cell as an IRI. If the file is newly created and has not yet been saved, the file name is empty text “”.
CONTENTS	Returns the contents of the cell, without formatting properties.
COLOR	Returns 1 if color formatting is set for negative value in this cell; oth-

	erwise returns 0
FORMAT	Returns a text string which shows the number format of the cell. ,(comma) = number with thousands separator F = number without thousands separator C = currency format S = exponential representation P = percentage To indicate the number of decimal places after the decimal separator, a number is given right after the above characters. D1 = MMM-D-YY, MM-D-YY and similar formats D2 = DD-MM D3 = MM-YY D4 = DD-MM-YYYY HH:MM:SS D5 = MM-DD D6 = HH:MM:SS AM/PM D7 = HH:MM AM/PM D8 = HH:MM:SS D9 = HH:MM G = All other formats - (Minus) at the end = negative numbers in the cell have color setting () (brackets) at the end = this cell has the format settings with parentheses for positive or all values
TYPE	Returns the text value corresponding to the type of content in the cell: “b” : blank or empty cell content “l” : label or text cell content “v” : number value cell content
WIDTH	Returns the column width of the cell. The unit is the width of one zero (0) character in default font size.
PROTECT	Returns the protection status of the cell: 1 = cell is protected 0 = cell is unprotected
PARENTHESES	Returns 1 if the cell has the format settings with parentheses for positive or all values, otherwise returns 0
PREFIX	Returns single character text strings corresponding to the alignment of the cell. “‘” (APOSTROPHE, U+0027) = left alignment "" (QUOTATION MARK, U+0022) = right alignment “^” (CIRCUMFLEX ACCENT, U+005E) = centered alignment

	“\” (REVERSE SOLIDUS, U+005C) = filled alignment otherwise, returns empty string "".

•

6.13.4 COLUMN

Summary: Returns the column number(s) of a reference.

Syntax: COLUMN([*Reference* ***R***])

Returns: *Number*

Constraints: AREAS(***R***) = 1

Semantics: Returns the column number of a reference, where “A” is 1, “B” is 2, and so on. If no parameter is given, the current cell is used. If a reference has multiple columns, an array of numbers is returned with all of the columns in the reference.

See also AREAS 6.13.2, ROW 6.13.29, SHEET 6.13.31

6.13.5 COLUMNS

Summary: Returns the number of columns in a given range.

Syntax: COLUMNS(*Reference|Array* ***R***)

Returns: *Number*

Constraints: None

Semantics: Returns the number of columns in the range or array specified. The result is not dependent on the cell content in the range.

See also ROWS 6.13.30

6.13.6 COUNT

Summary: Count the number of Numbers provided.

Syntax: COUNT({ *NumberSequenceList* ***N*** }$^{+}$)

Returns: *Number*

Constraints: One or more parameters.

Semantics: Counts the numbers in the list ***N***. Only numbers in references are counted; all other types are ignored. This function does *not* propagate Error values. It is implementation-defined what happens if 0 parameters are passed, but it should be an Error or 0.

See also COUNTA 6.13.7

6.13.7 COUNTA

Summary: Count the number of non-empty values.

Syntax: COUNTA({ *Any* ***AnyValue*** }$^{+}$)

Returns: *Number*

Constraints: None.

Semantics: Counts the number of non-blank values. A value is non-blank if it contains any content of any type, including an Error. In a reference, every cell that is not empty is included in the count. An empty string value ("") is *not* considered blank. Errors contained in a range are considered a non-blank value for purposes of the count. Constant expressions or formulas are allowed; these are evaluated and if they produce an Error value the Error value is counted as one non-blank value. It is implementation-defined what happens if 0 parameters are passed, but it should be an Error or 0.

See also COUNT 6.13.6, ISBLANK 6.13.14

6.13.8 COUNTBLANK

Summary: Count the number of blank cells.

Syntax: COUNTBLANK(*ReferenceList* ***R***)

Returns: *Number*

Constraints: None.

Semantics: Counts the number of blank cells in ***R***. A cell is blank if the cell is empty for purposes of COUNTBLANK. If ISBLANK(***R***) is TRUE, then it is blank. A cell with numeric value zero ('0') is not blank. It is implementation-defined whether or not a cell returning the empty string ("") is considered blank; because of this, there is a (potential) subtle difference between COUNTBLANK and ISBLANK.

Evaluators shall support one Reference as a parameter and may support a ReferenceList as a parameter.

See also COUNT 6.13.6, COUNTA 6.13.7, COUNTIF 6.13.9, ISBLANK 6.13.14

6.13.9 COUNTIF

Summary: Count the number of cells in a range that meet a criteria.

Syntax: COUNTIF(*ReferenceList* ***R*** ; *Criterion* ***C***)

Returns: *Number*

Constraints: Does not accept constant values as the reference parameter.

Semantics: Counts the number of cells in the reference range ***R*** that meet the Criterion ***C*** (4.11.8).

The values returned may vary depending upon the HOST-USE-REGULAR-EXPRESSIONS or HOST-USE-WILDCARDS or HOST-SEARCH-CRITERIA-MUST-APPLY-TO-WHOLE-CELL properties. 3.4

See also COUNT 6.13.6, COUNTA 6.13.7, COUNTBLANK 6.13.8, COUNTIFS 6.13.10, SUMIF 6.16.62, Infix Operator "=" 6.4.7, Infix Operator "<>" 6.4.8, Infix Operator Ordered Comparison ("<", "<=", ">", ">=") 6.4.9

6.13.10 COUNTIFS

Summary: Count the number of cells that meet multiple criteria in multiple ranges.

Syntax: COUNTIFS(*Reference* ***R1*** ; *Criterion* ***C1*** [; *Reference* ***R2*** ; *Criterion* ***C2***]...)

Returns: *Number*

Constraints: Does not accept constant values as the reference parameter.

Semantics: Counts the number of cells that meet the Criterion ***C1*** in the reference range ***R1*** and the Criterion ***C2*** in the reference range ***R2***, and so on (4.11.8). All reference ranges shall have the same dimension and size, else an Error is returned. A logical AND is applied between each array result of each selection; an entry is counted only if the same position in each array is the result of a Criterion match.

The values returned may vary depending upon the HOST-USE-REGULAR-EXPRESSIONS or HOST-USE-WILDCARDS or HOST-SEARCH-CRITERIA-MUST-APPLY-TO-WHOLE-CELL properties. 3.4

See also AVERAGEIFS 6.18.6, COUNT 6.13.6, COUNTA 6.13.7, COUNTBLANK 6.13.8, COUNTIF 6.13.9, SUMIF 6.16.62, SUMIFS 6.16.63, Infix Operator "=" 6.4.7, Infix Operator "<>" 6.4.8, Infix Operator Ordered Comparison ("<", "<=", ">", ">=") 6.4.9

6.13.11 ERROR.TYPE

Summary: Returns Number representing the specific Error type.

Syntax: ERROR.TYPE(*Error **E***)

Returns: *Number*

Constraints: None.

Semantics: Returns a number representing what kind of Error has occurred. This function does *not* propagate Error values. Receiving a *non*-Error value returns an Error. In particular, ERROR.TYPE(NA()) returns 7, and ERROR.TYPE applied to a non-Error returns an Error.

See also NA 6.13.27

6.13.12 FORMULA

Summary: Returns formula at given reference as text.

Syntax: FORMULA(*Reference **X***)

Returns: *String*

Constraints: Reference ***X*** shall contain a formula.

Semantics: Returns the formula in reference ***X*** as a string. The specific syntax of this returned string is implementation-defined. This function is intended to aid debugging by simplifying display of formulas in other cells. This function does *not* propagate Error values.

See also ISFORMULA 6.13.18

6.13.13 INFO

Summary: Returns information about the environment.

Syntax: INFO(*Text **Category***)

Returns: *Any* (see below)

Constraints: ***Category*** shall be valid.

Semantics: Returns information about the environment in the given category.

Evaluators shall support at least the following categories:

Table 18 - INFO

Category	***Meaning***	***Type***
"directory"	Current directory. This shall be formatted so file names can be appended to the result (e.g., on POSIX and Windows systems it shall end with the separator “/” or “\” respectively).	Text
"memavail"	Amount of memory “available”, in bytes. On many modern (virtual memory) systems this value is not really available, but a system should return 0 if it is known that there is no more memory available, and greater than 0 otherwise	Number
"memused"	Amount of memory used, in bytes, by the data	Number
"numfile"	Number of active worksheets in files	Number
"osversion"	Operating system version	Text
"origin"	The top leftmost visible cell's absolute reference pre-fixed with “$A:”. In locales where cells are ordered	Text

	right-to-left, the top rightmost visible cell is used instead.	
"recalc"	Current recalculation mode. If the locale is English, this is either "Automatic" or "Manual" (the exact text depends on the locale)	Text
"release"	The version of the implementation.	Text
"system"	The type of the operating system.	Text
"totmem"	Total memory available in bytes, including the memory already used.	Number

Evaluators may support other categories.

See also CELL *6.13.3*

6.13.14 ISBLANK

Summary: Return TRUE if the referenced cell is blank, else return FALSE.

Syntax: ISBLANK(*Scalar* ***X***)

Returns: *Logical*

Constraints: None

Semantics: If ***X*** is of type Number, Text, or Logical, return FALSE. If ***X*** is a reference to a cell, examine the cell; if it is blank (has no value), return TRUE, but if it has a value, return FALSE. A cell with the empty string is *not* considered blank. This function does *not* propagate Error values, but returns FALSE in such cases.

See also ISNUMBER 6.13.22, ISTEXT 6.13.25

6.13.15 ISERR

Summary: Return TRUE if the parameter has type Error and is not #N/A, else return FALSE.

Syntax: ISERR(*Scalar* ***X***)

Returns: *Logical*

Constraints: None

Semantics: If ***X*** is of type Error, and ISNA(***X***) is not true, returns TRUE. Otherwise it returns FALSE. Note that this function returns FALSE if given #N/A; if this is not desired, use ISERROR 6.13.16. This function does *not* propagate Error values.

ISERR(***X***) is the same as:

IF(ISNA(***X***),FALSE(),ISERROR(***X***))

See also ERROR.TYPE 6.13.11, ISERROR 6.13.16, ISNA 6.13.20, ISNUMBER 6.13.22, ISTEXT 6.13.25, NA 6.13.27

6.13.16 ISERROR

Summary: Return TRUE if the parameter has type Error, else return FALSE.

Syntax: ISERROR(*Scalar* ***X***)

Returns: *Logical*

Constraints: None

Semantics: If ***X*** is of type Error, returns TRUE, else returns FALSE. Note that this function returns TRUE if given #N/A; if this is not desired, use ISERR 6.13.15. This function does *not* propagate Error values.

See also ERROR.TYPE 6.13.11, ISERR 6.13.15, ISNA 6.13.20, ISNUMBER 6.13.22, ISTEXT 6.13.25, NA 6.13.27

6.13.17 ISEVEN

Summary: Return TRUE if the value is even, else return FALSE.

Syntax: ISEVEN(*Number* ***X***)

Returns: *Logical*

Constraints: None

Semantics: First, compute X1 = TRUNC(***X***). Then, if X1 is even (a division by 2 has a remainder of 0), return TRUE, else return FALSE. The result is implementation-defined if given a Logical value; an evaluator may return either an Error or the result of converting the Logical value to a Number (per Conversion to Number 6.3.5).

See also ISODD 6.13.23, TRUNC 6.17.8

6.13.18 ISFORMULA

Summary: Return TRUE if the reference refers to a formula, else return FALSE.

Syntax: ISFORMULA(*Reference* ***X***)

Returns: *Logical*

Constraints: None

Semantics: If ***X*** refers to a cell whose value is computed by a formula, return TRUE, else return FALSE. A formula itself may compute a constant; in that case it will still return TRUE since it is still a formula. Passing a non-reference, or a reference to more than one cell, is implementation-defined. This function does *not* propagate Error values.

See also ISTEXT 6.13.25, ISNUMBER 6.13.22

6.13.19 ISLOGICAL

Summary: Return TRUE if the parameter has type Logical, else return FALSE.

Syntax: ISLOGICAL(*Scalar* ***X***)

Returns: *Logical*

Constraints: None

Semantics: If ***X*** is of type Logical, returns TRUE, else FALSE. Evaluators that do not have a distinct Logical type will return the same value ISNUMBER(***X***) would return. This function does *not* propagate Error values, but returns FALSE in such cases.

See also ISTEXT 6.13.25, ISNUMBER 6.13.22

6.13.20 ISNA

Summary: Return TRUE if the parameter has type Error and is #N/A, else return FALSE.

Syntax: ISERR(*Scalar* ***X***)

Returns: *Logical*

Constraints: None

Semantics: If ***X*** is #N/A, return TRUE, else return FALSE. Note that if ***X*** is a reference, the value being referenced is considered. This function does *not* propagate Error values.

See also ERROR.TYPE 6.13.11, ISERROR 6.13.16, ISERR 6.13.15, ISNUMBER 6.13.22, ISTEXT 6.13.25, NA 6.13.27

6.13.21 ISNONTEXT

Summary: Return TRUE if the parameter does *not* have type Text, else return FALSE.

Syntax: ISNONTEXT(*Scalar* ***X***)

Returns: *Logical*

Constraints: None

Semantics: If ***X*** is of type Text, ISNONTEXT returns FALSE, else TRUE. If ***X*** is a reference, it examines what ***X*** references. References to empty cells are not considered text, so for reference to an empty cell ISNONTEXT will return TRUE. Empty Cell 4.7 This function does *not* propagate Error values, but returns TRUE in such cases.

ISNONTEXT(***X***) is equivalent to NOT(ISTEXT(***X***))

See also ISNUMBER 6.13.22, ISLOGICAL 6.13.19, ISTEXT 6.13.25, NOT 6.15.7

6.13.22 ISNUMBER

Summary: Return TRUE if the parameter has type Number, else return FALSE.

Syntax: ISNUMBER(*Scalar* ***X***)

Returns: *Logical*

Constraints: None

Semantics: If ***X*** is of type Number, returns TRUE, else FALSE. Evaluators need not have a distinguished Logical type; in such evaluators, ISNUMBER(TRUE()) is TRUE. This function does *not* propagate Error values, but returns FALSE in such cases.

See also ISTEXT 6.13.25, ISLOGICAL 6.13.19

6.13.23 ISODD

Summary: Return TRUE if the value is even, else return FALSE.

Syntax: ISODD(*Number* ***X***)

Returns: *Logical*

Constraints: None

Semantics: First, compute X1 = TRUNC(***X***). Then, if X1 is odd (a division by 2 has a remainder of 1), return TRUE, else return FALSE. The result is implementation-defined if given a Logical value; an evaluator may return either an Error or the result of converting the Logical value to a Number (per Conversion to Number 6.3.5).

See also ISEVEN 6.13.17, TRUNC 6.17.8

6.13.24 ISREF

Summary: Return TRUE if the parameter is of type reference, else return FALSE.

Syntax: ISREF(*Any* ***X***)

Returns: *Logical*

Constraints: None

Semantics: If ***X*** is of type Reference or ReferenceList, return TRUE, else return FALSE. Note that unlike nearly all other functions, when given a reference this function does *not* then examine the value being referenced. Some functions and operators return references, and thus ISREF will return TRUE when given their results. ***X*** may be a ReferenceList, in which case ISREF returns TRUE. This function does *not* propagate Error values.

See also ISNUMBER 6.13.22, ISTEXT 6.13.25

6.13.25 ISTEXT

Summary: Return TRUE if the parameter has type Text, else return FALSE.

ISTEXT(***X***) is equivalent to NOT(ISNONTEXT(***X***)).

Syntax: ISTEXT(*Scalar* ***X***)

Returns: *Logical*

Constraints: None

Semantics: If ***X*** is of type Text, returns TRUE, else FALSE. References to empty cells are NOT considered Text. If ***X*** is a reference, examines what ***X*** references. References to empty cells are NOT considered Text, so a reference to an empty cell will return FALSE. Empty Cell 4.7 This function does *not* propagate Error values, but returns FALSE in such cases.

See also ISNONTEXT 6.13.21, ISNUMBER 6.13.22, ISLOGICAL 6.13.19

6.13.26 N

Summary: Return the number of a value.

Syntax: N(*Any* ***X***)

Returns: *Number*

Constraints: None

Semantics: If ***X*** is a Reference, it is first dereferenced to a scalar. Then its type is examined. If it is of type Number, it is returned. If it is of type Logical, 1 is returned if TRUE else 0 is returned. It is implementation-defined what happens if it is provided a Text value.

See also T 6.20.22, VALUE 6.13.34

6.13.27 NA

Summary: Return the constant Error value #N/A.

Syntax: NA()

Returns: *Error*

Constraints: Shall have 0 parameters

Semantics: This function takes no arguments and returns the Error #N/A.

See also ERROR.TYPE 6.13.11, ISERROR 6.13.16

6.13.28 NUMBERVALUE

Summary: Convert text to number, in a locale-independent way.

Syntax: NUMBERVALUE(*Text* ***X*** [; *Text* ***DecimalSeparator*** [; *Text* ***GroupSeparator***]])

Returns: *Number*

Constraints: LEN(***DecimalSeparator***) = 1, ***DecimalSeparator*** shall not appear in ***GroupSeparator***

Semantics: Converts given Text value ***X*** into Number. If ***X*** is a Reference, it is first dereferenced.

X is transformed according to the following rules:

1. Starting from the beginning, remove all occurrences of the group separator before any decimal separator

2. Starting from the beginning, replace the first occurrence in the text of the decimal separator character with the FULL STOP (U+002E) character
3. Remove all whitespace characters (5.14).
4. If the first character of the resulting string is a period FULL STOP (U+002E) then prepend a zero
5. If the string ends in one or more instances of PERCENT SIGN (U+0025) , remove the percent sign(s)

If percent signs were removed in step 5, divide the value of the returned number by 100 for each percent sign removed. If the resulting string is a valid xsd:float, then return the number corresponding to that string, according to the definition provided in XML Schema, Part 2, Section 3.2.4.

If the string is not a valid xsd:float then return an error.

See also N 6.13.26, T 6.20.22, DATEVALUE 6.10.4, TIMEVALUE 6.10.19, VALUE 6.13.34

6.13.29 ROW

Summary: Returns the row number(s) of a reference.

Syntax: ROW([*Reference* ***R***])

Returns: *Number*

Constraints: AREAS(***R***) = 1

Semantics: Returns the row number of a reference. If no parameter is given, the current cell is used. If a reference has multiple rows, an array of numbers is returned with all of the rows in the reference.

See also AREAS 6.13.2, COLUMN 6.13.4, SHEET 6.13.31

6.13.30 ROWS

Summary: Returns the number of rows in a given range.

Syntax: ROWS(*Reference|Array* ***R***)

Returns: *Number*

Constraints: None

Semantics: The result is not dependent on the cell content in the range.

See also COLUMNS 6.13.5

6.13.31 SHEET

Summary: Returns the sheet number of the reference or the string representing a sheet name.

Syntax: SHEET([*Text|Reference* ***R***])

Returns: *Number* ≥ 1

Constraints: ***R*** shall not contain a Source Location (5.8 References)

Semantics: Returns the 1-based sheet number of the given reference or sheet name.

Hidden sheets are not excluded from the sheet count.

If no parameter is given, the result is the sheet number of the sheet containing the formula.

If a Reference is given it is not dereferenced.

If the reference encompasses more than one sheet, the result is the number of the first sheet in the range.

If a reference does not contain a sheet reference, the result is the sheet number of the sheet containing the formula.

If the function is not evaluated within a table cell, an error is returned.

See also COLUMN 6.13.4, ROW 6.13.29, SHEETS 6.13.32

6.13.32 SHEETS

Summary: Returns the number of sheets in a reference or current document.

Syntax: SHEETS([*Reference* ***R***])

Returns: *Number* ≥ 1

Constraints: ***R*** shall not contain a Source Location (5.8 References)

Semantics: Returns the number of sheets in the given reference.

If no parameter is given, the number of sheets in the document is returned.

Hidden sheets are not excluded from the sheet count.

See also COLUMNS 6.13.5, ROWS 6.13.30, SHEET 6.13.31

6.13.33 TYPE

Summary: Returns a number indicating the type of the provided value.

Syntax: TYPE(*Any* ***Value***)

Returns: *Number*

Constraints: None

Semantics: Returns a number indicating the type of the value given:

Table 19 - TYPE

***Value*'s Type**	***TYPE* Return**
Number	1
Text	2
Logical	4
Error	16
Array	64

If a Reference is provided, the reference is first dereferenced, and any formulas are evaluated. This function does *not* propagate Error values.

> **Note:** Reliance on the return of 4 for TYPE will impair the interoperability of a document containing an expression that relies on that value.

See also ERROR.TYPE 6.13.11

6.13.34 VALUE

Summary: Convert text to number.

Syntax: VALUE(*Text* ***X***)

Returns: *Number*

Constraints: None

Semantics: Converts given text value ***X*** into Number. If ***X*** is a Reference, it is first dereferenced. It is implementation-defined what happens if VALUE is given neither a Text value nor a Reference to a Text value. If the Text has a date, time, or datetime format, it is converted into a serial Number. In many cases the conversion of a date or datetime format is locale-dependent.

If the supplied text ***X*** cannot be converted into a Number, an Error is returned.

Regardless of the current locale, an evaluator shall accept numbers matching this regular expression (which does not include a decimal point character) and convert it into a Number. If the value ends in %, it shall divide the number by 100:

```
[+-]? [0-9]+([eE][+-]?[0-9]+)?)%?
```

VALUE shall accept text representations of numbers in the current locale. In the en_US locale, an evaluator shall accept decimal numbers matching this regular expression and convert it into a Number (the leading “$” is ignored; commas are ignored if they match the rule of a thousands separator; if the value ends in %, it shall divide the number by 100):

```
[+-]?\$?([0-9]+(,[0-9]{3})*)?(\.[0-9]+)?(([eE][+-]?[0-9]+)|%)?
```

Evaluators shall accept fractional values matching the regular expression:

```
[+-]? [0-9]+ \ [0-9]+/[1-9][0-9]?
```

A leading minus sign shall be interpreted as identifying a negative number for the entire value. There is a space between the integer and the fractional portion; values between 0 and 1 can be represented by using 0 for the integer part.

Evaluators shall support time values in at least the HH:MM and HH:MM:SS formats, where HH is a 1-2 digit value from 0 to 23, MM is a one- or two-digit value from 0 to 59, and SS is a one- or two-digit value from 0 to 59. The hour may be one or two digits when it is less than 10. VALUE converts time values into Numbers ranging from 0 to 1, which is percentage of day that has elapsed by that time. Thus, VALUE("2:00") is the same as 2/24. Evaluators should accept times with fractional seconds as well when expressed in the form HH:MM:SS.ssss...

Evaluators shall accept textual dates in [ISO8601] format (YYYY-MM-DD), converting them into serial numbers based on the current epoch. Evaluators shall, when running in the en_US locale, accept the format MM/DD/YYYY .

In addition, in locale en_US, evaluators shall support the following formats (where YYYY is a 4-digit year, YY a 2-digit year, MM a numerical month, DD a numerical day, mmm a 3-character abbreviated alphabetical name, and mmmmm a full name):

Table 20 - VALUE

Format	***Example***	***Comment***
MM/DD/ YYYY	5/21/2006	LOCALE-DEPENDENT; Long year format with slashes.
MM/DD/YY	5/21/06	LOCALE-DEPENDENT; Short year format with slashes
MM-DD-YYYY	5-21-2006	LOCALE-DEPENDENT; Long year format with dashes (short year may be supported, but it may also be used for years less than 100 .
mmm DD, YYYY	Oct 29, 2006	LOCALE-DEPENDENT; Short alphabetic month day, year. **Note:** mmm depends on the locale's language.
DD mmm YYYY	29 Oct 2006	LOCALE-DEPENDENT; Short alphabetic day month year
mmmmm DD, YYYY	October 29, 2006	LOCALE-DEPENDENT; Long alphabetic month day, year

DD mmmmm YYYY	29 October 2006	LOCALE-DEPENDENT; Long alphabetic day month year

Evaluators should support other locales. Many conversions will vary by locale, including the decimal point (comma or period), names of months, date formats (MM/DD vs. DD/MM), and so on. Dates in particular vary by locale.

Evaluators shall support the datetime format, which is a date followed by a time, using *either* the space character or the literal “T” character as the separator (the “T” is from ISO 8601). Evaluators shall support at least the ISO date format in a datetime format; they may support other date formats in a datetime format as well. Formats such as “YYYY-MM-DD HH:MM” and “YYYY-MM-DDTHH:MM:SS” (where “T” is the literal character T) shall be accepted. The result of accepting a datetime format shall be a representation of that specific time (without removing either the date or the time of day, unlike DATEVALUE or TIMEVALUE).

Evaluators may accept other formats that will convert to numbers, and those conversions may be locale-dependent, as long as they do not conflict with the above. Where no conversion is determined, an Error is returned.

See also N 6.13.26, T 6.20.22, DATEVALUE 6.10.4, TIMEVALUE 6.10.19, NUMBERVALUE 6.13.28

6.14 Lookup Functions

6.14.1 General

These functions look up information. Note that IF() 6.15.4 can be considered a trivial lookup function, but it is listed as a logical function instead.

6.14.2 ADDRESS

Summary: Returns a cell address (reference) as text.

Syntax: ADDRESS(*Integer* ***Row*** ; *Integer* ***Column*** [; *Integer* ***Abs*** = 1 [; *Logical* ***A1Style*** = TRUE [; *Text* ***Sheet***]]])

Returns: *Text*

Constraints: ***Row*** ≥ 1, ***Column*** ≥ 1, 1 ≤ A*bs* ≤ 4; ***A1Style*** = TRUE. Evaluators may evaluate expressions that do not meet the constraint ***A1Style*** = TRUE.

Semantics: Returns a cell address (reference) as text. The text does *not* include the surrounding [...] of a reference. If a ***Sheet*** name is given, the sheet name in the text returned is followed by a “.” and the column/row reference if ***A1Style*** is TRUE, or a “!” and the column/row reference if ***A1Style*** is FALSE; otherwise no “.” respectively “!” is included. Columns are identified using uppercase letters. The value of ***Abs*** determines if the column and/or row is absolute or relative. The value of ***A1Style*** determines if A1 reference style or R1C1 reference style is used.

Table 21 - ADDRESS

Abs	***Meaning***	***A1Style* = *TRUE***	***A1Style* = *FALSE***
1	fully absolute	A1	R1C1
2	row absolute, column relative	A$1	R1C[1]
3	row relative, column absolute	$A1	R[1]C1
4	fully relative	A1	R[1]C[1]

Note that the INDIRECT function accepts this format.

See also INDIRECT 6.14.7

6.14.3 CHOOSE

Summary: Uses an index to return a value from a list of values.

Syntax: CHOOSE(*Integer* ***Index*** ; { *Any* ***Value*** }$^{+}$)

Returns: *Any*

Constraints: Returns an Error if ***Index*** < 1 or if there is no corresponding value in the list of Values.

Semantics: Uses ***Index*** to determine which value, from a list of values, to return. If ***Index*** is 1, CHOOSE returns the first ***Value***; if ***Index*** is 2, CHOOSE returns the second value, and so on. Note that the Values may be formula expressions. Expression paths of parameters other than the one chosen are not calculated or evaluated for side effects.

See also IF 6.15.4

6.14.4 GETPIVOTDATA

6.14.4.1 General

Summary: Return a value from a data pilot table.

Note: This function has two syntaxes. The first of these is the preferred syntax, while the second, alternative syntax is provided for compatibility reasons.

6.14.4.2 Preferred Syntax

Syntax: GETPIVOTDATA(*Text* ***DataField*** ; *Reference* ***Table*** { ; *Text* ***FieldName*** ; *Scalar* ***Member*** }*)

Note: This version of the syntax is distinguished by the parameter ***Table*** being the *second* parameter.

Returns: *Any*

Semantics: Returns a single result from the calculation of a data pilot table.

The data pilot table is selected by ***Table***, which is a reference to a cell or cell range that's within a data pilot table or contains a data pilot table. If the cell range contains several data pilot tables, the last one in the order of `<table:data-pilot-table>` elements (OpenDocument, Part 3, 9.6.3) in the file is used.

DataField selects one of the data pilot table's data fields. It can be the name of the source column, or the given name of the data field (such as “Sum of Sales”).

If no ***FieldName***/***Member*** pairs are given, the grand total is returned. Otherwise, each pair adds a constraint that the result shall satisfy. ***FieldName*** is the name of a field from the data pilot table. ***Member*** is the name of a member (item) from that field. If a member is a number, ***Member*** can alternatively be its numerical value.

If the data pilot table contains only a single result value that fulfills all of the constraints, or a subtotal result that summarizes all matching values, that result is returned. If there is no matching result, or several ones without a subtotal for them, an Error is returned. These conditions apply to results that are included in the data pilot table. If the source data contains entries that are hidden by settings of the data pilot table, they are ignored. The order of the ***FieldName***/***Member*** pairs is not significant. Field and member names are case-insensitive.

If no constraint for a page field is given, the field's selected value is implicitly used. If a constraint for a page field is given, it shall match the field's selected value, or an Error is returned.

Subtotal values from the data pilot table are only used if they use the function “auto” (except when specified in the constraint, see below).

6.14.4.3 Alternative Syntax

Syntax: GETPIVOTDATA(*Reference* ***Table*** ; *Text* ***Constraints***)

Note: This version of the syntax is distinguished by the parameter ***Table*** being the *first* parameter.

Returns: *Any*

Semantics: Returns a single result from the calculation of a data pilot table.

Table has the same meaning as in the preferred syntax.

Constraints is a space-separated list. Entries can be quoted (single quotes). One of the entries can be the data field name. The data field name can be left out if the data pilot table contains only one data field, otherwise it shall be present. Each of the other entries specifies a constraint in the form *FieldName*[*Member*] (with literal characters "[" (LEFT SQUARE BRACKET, U+005B) and "]" (RIGHT SQUARE BRACKET U+005D)), or only Member if the member name is unique within all fields that are used in the data pilot table. A function name can be added in the form *FieldName*[*Member*;*Function*], which will cause the constraint to match only subtotal values which use that function. The possible function names are the same as in the `table:function` attribute of the `<table:data-pilot-subtotal>` element (OpenDocument, Part 3, 19.647.4), case-insensitive.

6.14.5 HLOOKUP

Summary: Look for a matching value in the first row of the given table, and return the value of the indicated row.

Syntax: HLOOKUP(*Any* ***Lookup*** ; *ForceArray Reference|Array* ***DataSource*** ; *Integer* ***Row*** [; *Logical* ***RangeLookup*** = TRUE])

Returns: *Any*

Constraints: ***Row*** ≥ 1; Searched portion of ***DataSource*** shall not include Logical values. Evaluators may evaluate expressions that do not meet the constraint that the searched portion of a ***DataSource*** not include Logical values.

Semantics:

If ***RangeLookup*** is omitted or TRUE or not 0, the first row of ***DataSource*** is assumed to be sorted in ascending order, with smaller numbers before larger ones, smaller text values before larger ones (e.g., "A" before "B", and "B" before "BA"), and FALSE before TRUE. If the types are mixed, Numbers are sorted before Text, and Text before Logicals; evaluators without a separate Logical type may include a Logical as a Number. The lookup will try to match an entry of value ***Lookup***. If none is found the largest entry less than ***Lookup*** is taken as a match. From a sequence of identical values ≤ ***Lookup*** the last entry is taken. If there is no data less than or equal to ***Lookup***, the #N/A Error is returned. If ***Lookup*** is of type Text and the value found is of type Number, the #N/A Error is returned. If ***DataSource*** is not sorted, the result is undetermined and implementation-dependent.

If ***RangeLookup*** is FALSE or 0, ***DataSource*** does not need to be sorted and an exact match is searched. Each value in the first row of ***DataSource*** is examined in order (starting at the left) until its value matches ***Lookup***.

Both methods, if there is a match, return the corresponding value in row ***Row***, relative to the ***DataSource***, where the topmost row in ***DataSource*** is 1.

The values returned may vary depending upon the HOST-USE-REGULAR-EXPRESSIONS or HOST-USE-WILDCARDS or HOST-SEARCH-CRITERIA-MUST-APPLY-TO-WHOLE-CELL properties. 3.4

See also INDEX 6.14.6, MATCH 6.14.9, OFFSET 6.14.11, VLOOKUP 6.14.12

6.14.6 INDEX

Summary: Returns a value using a row and column index value (and optionally an area index).

Syntax: INDEX(*ReferenceList*|*Array* ***DataSource*** ; [*Integer* ***Row***] [; [*Integer* ***Column***]] [; *Integer* ***AreaNumber*** = 1])

Returns: *Any*

Constraints: ***Row*** ≥ 0, ***Column*** ≥ 0,
1 ≤ ***AreaNumber*** ≤ number of references in ***DataSource*** if that is a ReferenceList, else ***AreaNumber*** = 1

Semantics:

Given a ***DataSource***, returns the value at the given ***Row*** and ***Column*** (starting numbering at 1, relative to the top-left of the ***DataSource***) of the given area ***AreaNumber***. If ***AreaNumber*** is not given, it defaults to 1 (the first and possibly only area). This function is essentially a two-dimensional version of CHOOSE, which does not accept range parameters.

If ***Row*** is omitted or an empty parameter (two consecutive ;; semicolons) or 0, an entire column of the given area ***AreaNumber*** in ***DataSource*** is returned. If ***Column*** is omitted or an empty parameter (two consecutive ;; semicolons) or 0, an entire row of the given area ***AreaNumber*** in ***DataSource*** is returned. If both, ***Row*** and ***Column***, are omitted or empty or 0, the entire given area ***AreaNumber*** is returned.

If ***Row*** or ***Column*** have a value greater than the dimension of the corresponding given area ***AreaNumber***, an Error is returned.

See also AREAS 6.13.2, CHOOSE 6.14.3

6.14.7 INDIRECT

Summary: Return a reference given a string representation of a reference.

Syntax: INDIRECT(*Text* ***Ref*** [; *Logical* ***A1*** = TRUE])

Returns: *Reference*

Constraints: ***Ref*** is valid reference

Semantics: Given text for a reference (such as “A3”), returns a reference. If ***A1*** is False, it is interpreted as an R1C1 reference style. For interoperability, if the ***Ref*** text includes a sheet name, evaluators should be able to parse both, the “.” dot and the “!” exclamation mark, as the sheet name separator. If evaluators support the ***A1*** = FALSE case of the ADDRESS 6.14.2 function and include the “!” exclamation mark as the sheet name separator, evaluators shall correctly parse that in the ***A1*** = FALSE case of this INDIRECT function. Evaluators shall correctly parse the “.” dot as the sheet name separator in the ***A1*** = TRUE case.

See also ADDRESS 6.14.2

6.14.8 LOOKUP

Summary: Look for criterion in an already-sorted array, and return a corresponding result.

Syntax: LOOKUP(*Any* ***Find*** ; *ForceArray Reference*|*Array* ***Searched*** [; *ForceArray Reference*|*Array* ***Results***])

Returns: *Any*

Constraints: The searched portion of ***Searched*** shall be sorted in ascending order; if provided, ***Results*** shall have the same length as ***Searched***. The searched portion of ***Searched*** shall not include Logical values. Evaluators may evaluate expressions that do not meet the constraint that the searched portion of a ***Searched*** not include Logical values.

Semantics: This function searches for ***Find*** in a row or column of the previously-sorted array ***Searched*** and returns a corresponding value. The match is the largest value in the row/column of ***Searched*** that is less than or equal to ***Find*** (so an exact match is always

preferred over inexact ones). From a sequence of identical values ≤ ***Find*** the last entry is taken. If ***Find*** is smaller than the smallest value in the first row or column (depending on the array dimensions), LOOKUP returns the #N/A Error. If ***Find*** is of type Text and the value found is of type Number, the #N/A Error is returned.

The searched portion of ***Searched*** shall be sorted in ascending order, and so that values of type Number precede values of type Text if both types are included (e.g., -2, 0, 2, “A”, “B”).

There are two major uses for this function; the 3-parameter version (vector) and the 2-parameter version (non-vector array).

> **Note:** Interoperability is improved by use of HLOOKUP or VLOOKUP in expressions over LOOKUP.

When given two parameters, ***Searched*** is first examined:

- If ***Searched*** is square or is taller than it is wide (more rows than columns), LOOKUP searches in the first column (similar to VLOOKUP), and returns the corresponding value in the last column.
- If ***Searched*** covers an area that is wider than it is tall (more columns than rows), LOOKUP searches in the first row (similar to HLOOKUP), and returns the corresponding value in the last row.

When given 3 parameters, ***Results*** shall be a vector (either a row or a column) or an Error is raised. The function determines the index of the match in the first column respectively row of ***Searched***, and returns the value in ***Results*** with the same index.

Searched is first examined:

- If ***Searched*** is square or is taller than it is wide (more rows than columns), LOOKUP searches in the first column (similar to VLOOKUP).
- If ***Searched*** covers an area that is wider than it is tall (more columns than rows), LOOKUP searches in the first row (similar to HLOOKUP).

The lengths of the search vector and the result vector do not need to be identical. When the match position falls outside the length of the result vector, an Error is returned if the result vector is given as an array object. If it is a cell range, it gets automatically extended to the length of the searched vector, but in the direction of the result vector. If just a single cell reference was passed, a column vector is generated. If the cell range cannot be extended due to the sheet's size limit, then the #N/A Error is returned.

The values returned may vary depending upon the HOST-USE-REGULAR-EXPRESSIONS or HOST-USE-WILDCARDS or HOST-SEARCH-CRITERIA-MUST-APPLY-TO-WHOLE-CELL properties. 3.4

See also HLOOKUP 6.14.5, INDEX 6.14.6, MATCH 6.14.9, OFFSET 6.14.11, VLOOKUP 6.14.12

6.14.9 MATCH

Summary: Finds a ***Search*** item in a sequence, and returns its position (starting from 1).

Syntax: MATCH(*Scalar* ***Search*** ; *ForceArray Reference|Array* ***SearchRegion*** [; *Integer* ***MatchType*** = 1])

Returns: *Any*

Constraints: -1 ≤ ***MatchType*** ≤ 1; The searched portion of ***SearchRegion*** shall not include Logical values. Evaluators may evaluate expressions that do not meet the constraint that the searched portion of a ***SearchRegion*** not include Logical values.

SearchRegion shall be a vector (a single row or column)

Semantics:

- ***MatchType*** = -1 finds the smallest value that is greater than or equal to ***Search*** in a ***SearchRegion*** where values are sorted in descending order. From a sequence of identical values ≥ ***Search*** the last value is taken. If no value ≥ ***Search*** exists, the #N/A Error is returned. If ***Search*** is of type Number and the value found is of type Text, the #N/A Error is returned.
- ***MatchType*** = 0 finds the first value that is equal to ***Search***. Values in ***SearchRegion*** do not need to be sorted. If no value equal to ***Search*** exists, the #N/A Error is returned.
- ***MatchType*** = 1 or omitted finds the largest value that is less than or equal to ***Search*** in a ***SearchRegion*** where values are sorted in ascending order. From a sequence of identical values ≤ ***Search*** the last value is taken. If no value ≤ ***Search*** exists, the #N/A Error is returned. If ***Search*** is of type Text and the value found is of type Number, the #N/A Error is returned.

If a match is found, MATCH returns the relative position (starting from 1). For Text the comparison is case-insensitive. ***MatchType*** determines the type of search; if ***MatchType*** is 0, the ***SearchRegion*** shall be considered unsorted, and the first match is returned. If ***MatchType*** is 1, the ***SearchRegion*** may be assumed to be sorted in ascending order, with smaller Numbers before larger ones, smaller Text values before larger ones (e.g., "A" before "B", and "B" before "BA"), and FALSE before TRUE. If the types are mixed, Numbers are sorted before Text, and Text before Logicals; evaluators without a separate Logical type may include a Logical as a Number. If ***MatchType*** is -1, then ***SearchRegion*** may be assumed to be sorted in descending order (the opposite of the above). If ***MatchType*** is 1 or -1, evaluators may use binary search or other techniques so that they do *not* need to examine every value in linear order. ***MatchType*** defaults to 1.

The values returned may vary depending upon the HOST-USE-REGULAR-EXPRESSIONS or HOST-USE-WILDCARDS or HOST-SEARCH-CRITERIA-MUST-APPLY-TO-WHOLE-CELL properties. 3.4

See also HLOOKUP 6.14.5, OFFSET 6.14.11, VLOOKUP 6.14.12

6.14.10 MULTIPLE.OPERATIONS

Summary: Executes a formula expression while substituting a row reference and a column reference.

Syntax: MULTIPLE.OPERATIONS(*Reference* ***FormulaCell*** ; *Reference* ***RowCell*** ; *Reference* ***RowReplacement*** [; *Reference* ***ColumnCell*** ; *Reference* ***ColumnReplacement***])

Returns: *Any*

Semantics:

- ***FormulaCell***: reference to the cell that contains the formula expression to calculate.
- ***RowCell***: reference that is to be replaced by ***RowReplacement***.
- ***RowReplacement***: reference that replaces ***RowCell***.
- ***ColumnCell***: reference that is to be replaced by ***ColumnReplacement***.
- ***ColumnReplacement***: reference that replaces ***ColumnCell***.

MULTIPLE.OPERATIONS executes the formula expression pointed to by ***FormulaCell*** and all formula expressions it depends on while replacing all references to ***RowCell*** with references to ***RowReplacement*** respectively *all* references to ***ColumnCell*** with references to ***ColumnReplacement***.

If calls to MULTIPLE.OPERATIONS are encountered in dependencies, replacements of target cells shall occur in queued order, with each replacement using the result of the previous replacement.

> **Note:** The function may be used to create tables of expressions that depend on two input parameters.

Example: ***FormulaCell*** is B5, ***RowCell*** is B3, ***ColumnCell*** is B2

Table 22 - MULTIPLE.OPERATIONS

	col_B	col_C	col_D	col_E	col_F
row_2	1		1	2	3
row_3	1	1	=MULTIPLE.OP-ERATIONS(B5;B3;$C3;$B$2;D$2)	=MULTIPLE.OP-ERATIONS(B5;B3;$C3;$B$2;E$2)	=MULTIPLE.OP-ERATIONS(B5;B3;$C3;$B$2;F$2)
row_4	=B2+B3	2	=MULTIPLE.OP-ERATIONS(B5;B3;$C4;$B$2;D$2)	=MULTIPLE.OP-ERATIONS(B5;B3;$C4;$B$2;E$2)	=MULTIPLE.OP-ERATIONS(B5;B3;$C4;$B$2;F$2)
row_5	=B2*B3+B4	3	=MULTIPLE.OP-ERATIONS(B5;B3;$C5;$B$2;D$2)	=MULTIPLE.OP-ERATIONS(B5;B3;$C5;$B$2;E$2)	=MULTIPLE.OP-ERATIONS(B5;B3;$C5;$B$2;F$2)
		4	=MULTIPLE.OP-ERATIONS(B5;B3;$C6;$B$2;D$2)	=MULTIPLE.OP-ERATIONS(B5;B3;$C6;$B$2;E$2)	=MULTIPLE.OP-ERATIONS(B5;B3;$C6;$B$2;F$2)

Result:

Table 23 - MULTIPLE.OPERATIONS

	col_B	col_C	col_D	col_E	col_F
row_2	1		1	2	3
row_3	1	1	3	5	7
row_4	2	2	5	8	11
row_5	3	3	7	11	15
		4	9	14	19

Note that although only cell B5 is passed as the ***FormulaCell*** parameter, also the references to B2 and B3 of the formula in cell B4 are replaced, because B5 depends on B4.

6.14.11 OFFSET

Summary: Modifies a reference's position and dimension.

Syntax: OFFSET(*Reference* ***R*** ; *Integer* ***RowOffset*** ; *Integer* ***ColumnOffset*** [; [*Integer* ***NewHeight***] [; [*Integer* ***NewWidth***]]])

Returns: *Reference*

Constraints: ***NewWidth*** > 0; ***NewHeight*** > 0
The modified reference shall be in a valid range, starting from column/row one to the maximum column/row.

Semantics: Shifts reference by ***RowOffset*** rows and by ***ColumnOffset*** columns. Optionally, the dimension can be set to ***NewWidth*** and/or ***NewHeight***, if omitted the width/height of the original reference is taken. Note that ***NewHeight*** may be empty (two consecutive semicolons ;;) followed by a given ***NewWidth*** argument. Returns the modified reference.

See also COLUMN 6.13.4, COLUMNS 6.13.5, ROW 6.13.29, ROWS 6.13.30

6.14.12 VLOOKUP

Summary: Look for a matching value in the first column of the given table, and return the value of the indicated column.

Syntax: VLOOKUP(*Any* ***Lookup*** ; *ForceArray Reference|Array* ***DataSource*** ; *Integer* ***Column*** [; *Logical* ***RangeLookup*** = TRUE()])

Returns: *Any*

Constraints: ***Column*** ≥ 1; The searched portion of ***DataSource*** shall not include Logical values. Evaluators may evaluate expressions that do not meet the constraint that the searched portion of a ***DataSource*** not include Logical values.

Semantics:

If ***RangeLookup*** is omitted or TRUE or not 0, the first column of ***DataSource*** is assumed to be sorted in ascending order, with smaller Numbers before larger ones, smaller Text values before larger ones (e.g., "A" before "B", and "B" before "BA"), and FALSE before TRUE. If the types are mixed, Numbers are sorted before Text, and Text before Logicals; evaluators without a separate Logical type may include a Logical as a Number. The lookup will try to match an entry of value ***Lookup***. From a sequence of identical values ≤ ***Lookup*** the last entry is taken. If none is found the largest entry less than ***Lookup*** is taken as a match. If there is no data less than or equal to ***Lookup***, the #N/A Error is returned. If ***Lookup*** is of type Text and the value found is of type Number, the #N/A Error is returned. If ***DataSource*** is not sorted, the result is undetermined and implementation-dependent.

If ***RangeLookup*** is FALSE or 0, ***DataSource*** does not need to be sorted and an exact match is searched. Each value in the first column of ***DataSource*** is examined in order (starting at the top) until its value matches ***Lookup***. If no value matches, the #N/A Error is returned.

Both methods, if there is a match, return the corresponding value in column ***Column***, relative to the ***DataSource***, where the leftmost column in ***DataSource*** is 1.

The values returned may vary depending upon the HOST-USE-REGULAR-EXPRESSIONS or HOST-USE-WILDCARDS or HOST-SEARCH-CRITERIA-MUST-APPLY-TO-WHOLE-CELL properties. 3.4

See also HLOOKUP 6.14.5, INDEX 6.14.6, MATCH 6.14.9, OFFSET 6.14.11

6.15 Logical Functions

6.15.1 General

The logical functions are: TRUE() and FALSE(); the functions that compute Logical values NOT(), AND(), and OR(); and the conditional function IF(). The OpenDocument specification mentions "logical operators"; these are another name for the logical functions.

Note that because of Error values, any logical function that accepts parameters can actually produce TRUE, FALSE, or an Error value instead of TRUE or FALSE.

These are not bitwise operations, e.g., AND(12;10) produces TRUE, not 8. See the bit operation functions for bitwise operations.

6.15.2 AND

Summary: Compute logical AND of all parameters.

Syntax: AND({ *Logical|NumberSequenceList* ***L*** }$^{+}$)

Returns: *Logical*

Constraints: Shall have 1 or more parameters

Semantics: Computes the logical AND of the parameters. If all parameters are TRUE, returns TRUE; if any are FALSE, returns FALSE. When given one parameter, this has the effect of

converting that one parameter into a Logical value. When given zero parameters, evaluators may return a Logical value or an Error.

Also in array context a logical AND of all arguments is computed, range or array parameters are not evaluated as a matrix and no array is returned. This behavior is consistent with functions like SUM. To compute a logical AND of arrays per element use the * operator in array context.

See also OR 6.15.8, IF 6.15.4

6.15.3 FALSE

Summary: Returns constant FALSE.

Syntax: FALSE()

Returns: *Logical*

Constraints: Shall have 0 parameters

Semantics: Returns logical constant FALSE. This may be a Number or a distinct type.

See also TRUE 6.15.9, IF 6.15.4

6.15.4 IF

Summary: Return one of two values, depending on a condition.

Syntax: IF(*Logical* ***Condition*** [; [*Any* ***IfTrue***] [; [*Any* ***IfFalse***]]])

Returns: *Any*

Constraints: None.

Semantics: Computes ***Condition***. If it is TRUE, it returns ***IfTrue***, else it returns ***IfFalse***. This function only evaluates ***IfTrue***, or ***IfFalse***, and never both; that is to say, it short-circuits.

Seven versions are possible:

One parameter:

a) IF(***Condition***)

Two parameters:

b) IF(***Condition***;)

c) IF(***Condition***;***IfTrue***)

Three parameters:

d) IF(***Condition***;;)

e) IF(***Condition***;;***IfFalse***)

f) IF(***Condition***;***IfTrue***;)

g) IF(***Condition***;***IfTrue***;***IfFalse***)

If there is only 1 parameter (case a), ***IfTrue*** is considered to be TRUE and ***IfFalse*** is considered to be FALSE. Thus the 1 parameter version converts ***Condition*** into a Logical value.

If there are 2 parameters (cases b and c), ***IfFalse*** is considered to be FALSE. If there are 2 parameters and the second parameter is null (semicolon but no ***IfTrue***, case b), ***IfTrue*** is considered to be 0.

If there are 3 parameters but the second parameter is null (two consecutive ;; semicolons, cases d and e), ***IfTrue*** is considered to be 0.

If there are 3 parameters but the third parameter is null (semicolon but no ***IfFalse***, cases d and f), ***IfFalse*** is considered to be 0.

See also AND 6.15.2, OR 6.15.8

6.15.5 IFERROR

Summary: Return ***X*** unless it is an Error, in which case return an alternative value.

Syntax: IFERROR(*Any* ***X*** ; *Any* ***Alternative***)

Returns: *Any*

Constraints: None.

Semantics: Computes ***X***. If ISERROR(***X***) is TRUE, return ***Alternative***, else return ***X***.

> **Note:** This is semantically equivalent to IF(ISERROR(***X***); ***Alternative***; ***X***), except that ***X*** is only computed once.

See also IF 6.15.4, ISERROR 6.13.16

6.15.6 IFNA

Summary: Return ***X*** unless it is #N/A, in which case return an alternative value.

Syntax: IFNA(*Any* ***X*** ; *Any* ***Alternative***)

Returns: *Any*

Constraints: None.

Semantics: Computes ***X***. If ISNA(***X***) is TRUE, return ***Alternative***, else return ***X***.

> **Note:** This is semantically equivalent to IF(ISNA(***X***); ***Alternative***; ***X***), except that ***X*** is only computed once.

See also IF 6.15.4, ISNA 6.13.20

6.15.7 NOT

Summary: Compute logical NOT.

Syntax: NOT(*Logical* ***L***)

Returns: *Logical*

Constraints: Shall have 1 parameter.

Semantics: Computes the logical NOT. If given TRUE, returns FALSE; if given FALSE, returns TRUE.

See also AND 6.15.2, IF 6.15.4

6.15.8 OR

Summary: Compute logical OR of all parameters.

Syntax: OR({ *Logical*|*NumberSequenceList* ***L*** }$^{+}$)

Returns: *Logical*

Constraints: Shall have 1 or more parameters

Semantics: Computes the logical OR of the parameters. If all parameters are FALSE, it shall return FALSE; if any are TRUE, it shall returns TRUE. When given one parameter, this has the effect of converting that one parameter into a Logical value. When given zero parameters, evaluators may return a Logical value or an Error.

Also in array context a logical OR of all arguments is computed, range or array parameters are not evaluated as a matrix and no array is returned. This behavior is consistent with functions like SUM. To compute a logical OR of arrays per element use the + operator in array context.

See also AND 6.15.2, IF 6.15.4

6.15.9 TRUE

Summary: Returns constant TRUE

Syntax: TRUE()

Returns: *Logical*

Constraints: Shall have 0 parameters

Semantics: Returns logical constant TRUE. The result of this function may but need not be equal to 1 when compared using "=". It always has the value of 1 if used in a context requiring Number (because of the automatic conversions), so if ISNUMBER(TRUE()) is TRUE, then it shall have the value 1.

See also FALSE 6.15.3, IF 6.15.4, ISNUMBER 6.13.22

6.15.10 XOR

Summary: Compute a logical XOR of all parameters.

Syntax: XOR({ *Logical* ***L*** }$^{+}$)

Returns: *Logical*

Constraints: Shall have 1 or more parameters.

Semantics: Computes the logical XOR of the parameters such that the result is an addition modulo 2. If an even number of parameters is TRUE it returns FALSE, if an odd number of parameters is TRUE it returns TRUE. When given one parameter, this has the effect of converting that one parameter into a Logical value.

See also AND 6.15.2, OR 6.15.8

6.16 Mathematical Functions

6.16.1 General

This section describes functions for various mathematical functions, including trigonometric functions like SIN 6.16.55). Note that the constraint text presumes that a value of type Number is a real number (no imaginary component). Unless noted otherwise, all angle measurements are in radians.

6.16.2 ABS

Summary: Return the absolute (nonnegative) value.

Syntax: ABS(*Number* ***N***)

Returns: *Number*

Constraints: None

Semantics: If ***N*** < 0, returns -***N***, otherwise returns ***N***.

See also Prefix Operator "-" 6.4.16

6.16.3 ACOS

Summary: Returns the principal value of the arc cosine of a number. The angle is returned in radians.

Syntax: ACOS(*Number* ***N***)

Returns: *Number*

Constraints: -1.0 ≤ ***N*** ≤ 1.0.

Semantics: Computes the arc cosine of a number, in radians.

$$ACOS(N)=\frac{\pi}{2}-\left[N+\frac{1}{2\cdot 3}N^3+\frac{1\cdot 3}{2\cdot 4\cdot 5}N^5+\frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6\cdot 7}N^7+...\right]$$

Returns a principal value 0 ≤ result ≤ π.

See also COS 6.16.19, RADIANS 6.16.49, DEGREES 6.16.25

6.16.4 ACOSH

Summary: Return the principal value of the inverse hyperbolic cosine.

Syntax: ACOSH(*Number* ***N***)

Returns: *Number*

Constraints: ***N*** ≥ 1

Semantics: Computes the principal value of the inverse hyperbolic cosine.

$$ACOSH(N)=\ln\left(N+\sqrt{N^2-1}\right)$$

See also COSH 6.16.20, ASINH 6.16.8

6.16.5 ACOT

Summary: Return the principal value of the arc cotangent of a number. The angle is returned in radians.

Syntax: ACOT(*Number* ***N***)

Returns: *Number*

Semantics: Computes the arc cotangent of a number, in radians.

Returns a principal value 0 < result < π.

See also COT 6.16.21, ATAN 6.16.9, TAN 6.16.69, RADIANS 6.16.49, DEGREES 6.16.25

6.16.6 ACOTH

Summary: Return the hyperbolic arc cotangent

Syntax: ACOTH(*Number* ***N***)

Returns: *Number*

Constraints: ABS(***N***) > 1

Semantics: Computes the hyperbolic arc cotangent. The hyperbolic arc cotangent is an analog of the ordinary (circular) arc cotangent.

$$ACOTH(N)=\frac{1}{2}\ln\left(\frac{x+1}{x-1}\right)$$

See also COSH 6.16.20, ASINH 6.16.8

6.16.7 ASIN

Summary: Return the principal value of the arc sine of a number. The angle is returned in radians.

Syntax: ASIN(*Number* ***N***)

Returns: *Number*

Constraints: -1 ≤ ***N*** ≤ 1.

Semantics: Computes the arc sine of a number, in radians.

$$ASIN\left(N\right)=N+\frac{1}{2\cdot3}N^{3}+\frac{1\cdot3}{2\cdot4\cdot5}N^{5}+\frac{1\cdot3\cdot5}{2\cdot4\cdot6\cdot7}N^{7}+\ldots$$

Returns a principal value -π/2 ≤ result ≤ π/2.

See also SIN 6.16.55, RADIANS 6.16.49, DEGREES 6.16.25

6.16.8 ASINH

Summary: Return the principal value of the inverse hyperbolic sine

Syntax: ASINH(*Number* ***N***)

Returns: *Number*

Constraints: None

Semantics: Computes the principal value of the inverse hyperbolic sine.

$$ASINH\left(N\right)=\ln\left(N+\sqrt{N^{2}+1}\right)$$

See also SINH 6.16.56, ACOSH 6.16.4

6.16.9 ATAN

Summary: Return the principal value of the arc tangent of a number. The angle is returned in radians.

Syntax: ATAN(*Number* ***N***)

Returns: *Number*

Semantics: Computes the arc tangent of a number, in radians.

Returns a principal value -π/2 < result < π/2.

See also ATAN2 6.16.10, TAN 6.16.69, RADIANS 6.16.49, DEGREES 6.16.25

6.16.10 ATAN2

Summary: Returns the principal value of the arc tangent given a coordinate of two numbers.

The angle is returned in radians.

Syntax: ATAN2(*Number* ***x*** ; *Number* ***y***)

Returns: *Number*

Constraints: x ≠ 0 or y ≠ 0

Semantics: Computes the arc tangent of two numbers (the ***x*** and ***y*** coordinates of a point), in radians. This is similar to ATAN(***y***/***x***), but the signs of the two numbers are taken into account so that the result covers the full range from -π to +π. ATAN2(0;0) is implementation-defined, evaluators may return 0 or an Error.

Returns a principal value -π < result ≤ π.

See also ATAN 6.16.9, TAN 6.16.69, RADIANS 6.16.49, DEGREES 6.16.25

6.16.11 ATANH

Summary: Return the principal value of the inverse hyperbolic tangent

Syntax: ATANH(*Number* ***N***)

Returns: *Number*

Constraints: -1 < ***N*** < 1

Semantics: Computes the principal value of the inverse hyperbolic tangent.

$$ATANH(N) = \frac{1}{2}\ln\left(\frac{1+N}{1-N}\right)$$

See also COSH 6.16.20, SINH 6.16.56, ASINH 6.16.8, ACOSH 6.16.4, ATAN 6.16.9, ATAN2 6.16.10, FISHER 6.18.26

6.16.12 BESSELI

Summary: Returns the modified Bessel function of integer order In(***X***).

Syntax: BESSELI(*Number* ***X*** ; *Number* ***N***)

Returns: *Number*

Constraints: ***N*** ≥ 0, INT(***N***) = ***N***; Evaluators may evaluate expressions where ***N*** ≥ 0 returns a non-error value.

Semantics: Computes the modified Bessel function of integer order In(***X***). ***N*** is also known as the *order*.

See also BESSELJ 6.16.13, BESSELK 6.16.14, BESSELY 6.16.15, INT 6.17.2

6.16.13 BESSELJ

Summary: Returns the Bessel function of integer order Jn(***X***) (cylinder function)

Syntax: BESSELJ(*Number* ***X*** ; *Number* ***N***)

Returns: *Number*

Constraints: ***N*** ≥ 0, INT(***N***) = ***N***; Evaluators may evaluate expressions where ***N*** ≥ 0 returns a non-error value.

Semantics: Computes the Bessel function of integer order Jn(***X***). ***N*** is also known as the *order*.

See also BESSELI 6.16.12, BESSELK 6.16.14, BESSELY 6.16.15, INT 6.17.2

6.16.14 BESSELK

Summary: Returns the modified Bessel function of integer order Kn(x).

Syntax: BESSELK(*Number* ***X*** ; *Number* ***N***)

Returns: *Number*

Constraints: ***X*** ≠ 0, ***N*** ≥ 0, INT(***N***) = ***N***; Evaluators may evaluate expressions where ***N*** ≥ 0 returns a non-error value.

Semantics: Computes the Bessel function of integer order Kn(x). ***N*** is also known as the *order*.

See also BESSELI 6.16.12, BESSELJ 6.16.13, BESSELY 6.16.15, INT 6.17.2

6.16.15 BESSELY

Summary: Returns the Bessel function of integer order Yn(***X***), also known as the Neumann function.

Syntax: BESSELY(*Number* ***X*** ; *Number* ***N***)

Returns: *Number*

Constraints: ***X*** ≠ 0, ***N*** ≥ 0, INT(***N***) = ***N***; Evaluators may evaluate expressions where ***N*** ≥ 0 returns a non-error value.

Semantics: Computes Bessel function of integer order Yn(***X***), also known as the Neumann function. ***N*** is also known as the *order*.

See also BESSELI 6.16.12, BESSELJ 6.16.13, BESSELK 6.16.14, INT 6.17.2

6.16.16 COMBIN

Summary: Returns the number of different ***R***-length sets that can be selected from ***N*** items.

Syntax: COMBIN(*Integer* ***N*** ; *Integer* ***R***)

Returns: *Number*

Constraints: ***N*** ≥ 0, ***R*** ≥ 0, ***R*** ≤ ***N***

Semantics: COMBIN returns the binomial coefficient, which is the number of different ***R***-length sets that can be selected from ***N*** items. Since they are sets, order in the sets is not relevant. The parameters are truncated (using INT) before use. For example, if a jar contains five marbles, each one a distinct color, the number of different three-marble groups COMBIN(5;3) = 10. The result is

$$\binom{N}{R} = \frac{PERMUT}{R!} = \frac{N!}{R!(N-R)!}$$

Note that if order is important, use PERMUT instead.

See also INT 6.17.2, PERMUT 6.18.59

6.16.17 COMBINA

Summary: Returns the number of combinations with repetitions.

Syntax: COMBINA(*Integer* ***N*** ; *Integer* ***M***)

Returns: *Number*

Constraints: ***N*** ≥ 0, ***M*** ≥ 0, ***N*** ≥ ***M***; Evaluators may evaluate expressions where ***N*** ≥ 0, ***M*** ≥ 0 returns a non-error value.

Semantics: Returns the number of possible combinations of ***M*** objects out of ***N*** possible ones, with repetitions allowed. Actual arguments that are not integers are truncated (using INT) before use. The result is

$$\binom{N+M-1}{N-1}$$

See also COMBIN 6.16.16

6.16.18 CONVERT

Summary: Returns a number converted from one unit system into another.

Syntax: CONVERT(*Number* ***N*** ; *Text* ***From*** ; *Text* ***Into***)

Returns: *Number*

Constraints: ***From*** and ***Into*** shall be legal units, and shall be in the same unit group.

Semantics: Returns the number converted from the unit identified by ***From*** into the unit identified by ***Into***. A unit is a *unit symbol* , optionally preceded by a *unit prefix* (either a decimal prefix or a binary prefix, as specified in Table 25 - Decimal Prefixes for use in CONVERT and Table 26 - Binary prefixes for use in CONVERT respectively). Units (including both the unit symbol and the optional unit prefix) are case-sensitive.

Evaluators claiming to implement this function shall support at least the following unit symbols (with conversions between them and other units in the same group):

Table 24 - Unit names

Unit group	*Unit symbol*	*Description*
Area	**"uk_acre"**	International acre (using international feet), exactly 4046.8564224 m^2; normally *not* used for U.S. land areas
	"us_acre"	U.S. survey/statute acre (using U.S. survey/statute feet), exactly 4046+13525426/15499969 m^2
	"ang2" or **"ang^2"** *	Square angstrom (an Angstrom is exactly 10^{-10} m)
	"ar" *	are, 100 m^2 (not abbreviated as "a")
	"ft2" or **"ft^2"**	Square international feet (1 foot is exactly 0.3048 m)
	"ha"	hectare, 10000 m^2
	"in2" or **"in^2"**	Square international inches (1 inch is exactly 2.54 cm)
	"ly2" or **"ly^2"**	Square light-year (where year=365.25 days)
	"m2" or **"m^2"** *	Square meters
	"Morgen"	Morgen, 2500 m^2
	"mi2" or **"mi^2"**	Square international miles
	"Nmi2" or **"Nmi^2"**	Square nautical miles (1 nautical mile is 1852 m)
	"Pica2" or **"Pica^2"** or **"picapt2"** or **"picapt^2"**	Square Pica Point (one Pica point is 1/72 inch)
	"pica2" or **"pica^2"**	Square Pica (one Pica is 1/6 inch)
	"yd2" or **"yd^2"**	Square international yards (1 yard is 0.9144 m)
Distance (Length)	**"ang"** *	Angstrom, exactly 10^{-10} m
	"ell"	Ell, exactly 45 international inches
	"ft"	International Foot, exactly 0.3048 m and also exactly 12 international inches.
	"in"	International Inch, exactly 2.54 cm.
	"ly" *	Light-year, (299792458 m/s) (3600 s/hr) (24 hr/day) (365.25 day)
	"m" *	Meter
	"mi"	International Mile, exactly 1609.344 m and exactly 5280 international feet. This is *not* a U.S. survey/statute mile (see "survey_mi") nor a nautical mile (see "Nmi"), but this *is* the mile normally used in the U.S. customary

Unit group	*Unit symbol*	*Description*
		system
	"Nmi"	International nautical mile, exactly 1852 m. Note that this is not the obsolete U.S. nautical mile nor the Admiralty mile.
	"parsec" or **"pc"** *	Distance from sun to a point having heliocentric parallax of one second (used for stellar distance), AU/ tan(1/3600 degree) where an AU is exactly 149,597,870.691 kilometers. *
	"Pica" or **"pi-capt"**	Pica point (1/72 inch)
	"pica"	Pica (1/6 inch)
	"survey_mi"	U.S. survey "mile, aka U.S. statute mile, exactly 6336000/3937 m; used in some U.S. maps. This is *not* the mile (see “mi”) normally used in the U.S.
	"yd"	International yard, exactly 0.9144 m and exactly 3 international feet.
Energy	**"BTU"** (**"btu"**)	International Table British Thermal Unit
	"c" *	Thermodynamic calorie, 4.184 J. This is not a dietary Calorie (kilocalorie). For high accuracy, use Joule, due to the many conflicting definitions of calorie.
	"cal" *	International Table (IT) calorie, 4.1868 J. This is not a dietary Calorie (kilocalorie). For high accuracy, use Joule, due to the many conflicting definitions of calorie.
	"e" *	Erg
	"eV" (**"ev"**) *	Electron volt (eV preferred)
	"flb"	Foot-pound (international foot, avoirdupois pound)
	"HPh" (**"hh"**)	Horsepower-hour (HPh preferred)
	"J" *	Joule
	"Wh" (**"wh"**) *	Watt-hour
Force (Weight)	**"dyn"** (**"dy"**) *	Dyne
	"N" *	Newton
	"lbf"	Pound force (see “lbm” for pound mass)
	"pond" *	Pond, gravitational force on a mass of one gram, 9.80665E-3 N.
Information	**"bit"** * †	bit

Unit group	*Unit symbol*	*Description*
	"byte" * †	byte = 8 bits
Magnetic Flux Density	**"ga"** *	Gauss
	"T" *	Tesla
Mass	**"g"** *	Gram
	"grain"	Grain, 1/7000 international pound mass (avoirdupois) (U.S. usage).
	"cwt" (**"shweight"**)	U.S. (short) hundredweight, 100 lbm
	"uk_cwt" or **"lcwt"** (**"hweight"**)	Imperial hundredweight, aka long hundredweight; 112 lbm
	"lbm"	International pound mass (avoirdupois), exactly 453.59237 g (see “lbf” for pound force)
	"stone"	14 international pound mass (avoirdupois)
	"ton"	2000 international pound mass (avoirdupois) (U.S. usage). Note that there are many other measures also called “ton”; in particular, this is not a metric ton (tonne).
	"ozm"	Ounce mass (avoirdupois), exactly 1/16 of an international pound mass (avoirdupois) (see “oz” for fluid ounce)
	"sg"	Slug; 32.174 international pound mass (avoirdupois)
	"u" *	U (atomic mass unit)
	"uk_ton" or **"LTON"** (**"brton"**)	Imperial ton, aka “long ton”, "deadweight ton", or "weight ton". 2240 lbm.
Power	**"HP"** (**"h"**)	Mechanical horsepower aka Imperial horsepower. 550 foot-pounds per second. The unit “h” is deprecated and should be replaced with “HP”.
	"PS"	Pferdestärke (German “horse strength”, close but not identical to “HP”), the amount of power to lift a mass of 75 kilograms in one second against the earth gravitation between a distance of one meter, approximately 735.49875 W.
	"W" (**"w"**) *	Watt
Pressure	(**"at"**) *	**Note:** "at" has been implemented inconsistently to mean either Standard atmosphere or Technical atmosphere and is now deprecated. Use either "atm" or

Unit group	*Unit symbol*	*Description*
		"SI_at".
	"atm" *	Standard atmosphere = 1.0132510E+5 Pa
	"mmHg" *	mm of Mercury
	"Pa" *	Pascal **Note:** “P” or “p” in user input as abbreviations for Pascal may be accepted by implementations, but should be stored as “Pa”..
	"psi"	Pounds per square inch, using avoirdupois pounds and international inches.
	"SI_at" *	Technical atmosphere = 9.8066510E+4 Pa
	"Torr"	Torr, exactly 101325/760 Pa (this is close but not equal to mmHg)
Speed	**"admkn"**	Admiralty knot, exactly 6080 international feet/hour.
	"kn"	Knot, exactly one Nautical mile per hour or exactly 1852/3600 m/s. Note that this is not an Admiralty knot (“admkn”).
	"m/h" or **"m/hr"** *	Meters per hour
	"m/s" or **"m/sec"** *	Meters per second
	"mph"	Miles per hour (international miles)
Temperature	**"C"** (**"cel"**)	degrees Celsius
	"F" (**"fah"**)	degrees Fahrenheit
	"K" (**"kel"**) *	Kelvin
	"Rank"	degrees Rankine
	"Reau"	degrees Réaumur; °C = °Ré · 5/4.
Time	**"day"** or **"d"**	Day (exactly 24 hours)
	"hr"	Hour (exactly 60 minutes)
	"mn" or **"min"**	Minute (exactly 60 seconds)
	"sec" or **"s"** *	Second (“s” is the official abbreviation of this SI base unit, while “sec” is a widely-recognized abbreviation in the CONVERT function) *
	"yr"	Year (exactly 365.25 days, for purposes of this function)

Unit group	*Unit symbol*	*Description*
Volume	**"ang3"** or **"ang^3"** *	Cubic angstrom
	"barrel"	U.S. oil barrel, exactly 42 U.S. customary gallons (liquid). Note that many other units are also called barrels (e.g., a beer barrel in the U.K. is 36 Imperial gallons)
	"bushel"	U.S. bushel (*not* Imperial bushel), interpreted as volume
	"cup"	Cup (U.S. customary liquid measure)
	"ft3" or **"ft^3"**	Cubic international feet
	"gal"	Gallon (U.S. customary liquid measure), 3.785411784 liters.
	"GRT" (**"regton"**)	Gross Registered Ton, 100 cubic (international) feet
	"in3" or **"in^3"**	Cubic international inch
	"l" or **"L"** (**"lt"**) *	Liter
	"ly3" or **"ly^3"**	Cubic light-year
	"m3" or **"m^3"** *	Cubic meter
	"mi3" or **"mi^3"**	Cubic international mile
	"MTON"	Measurement ton aka “freight ton”, 40 cubic feet
	"Nmi3" or **"Nmi^3"**	Cubic nautical mile
	"oz"	Fluid ounce (U.S. customary liquid measure; see “ozm” for ounce mass)
	"Pica3" or **"Pica^3"** **"picapt3"** or **"picapt^3"**	Cubic Pica Point (one Pica point is 1/72 inch)
	"pica3" or **"pica^3"**	Cubic Pica (one Pica is 1/6 inch)
	"pt" or **"us_pt"**	U.S. Pint (liquid measure)
	"qt"	Quart (U.S. customary liquid measure). This is 0.946352946 liters, and thus *not* the same as the U.S. dry quart (1.101220 liters), *nor* is this the same as the Imperial quart (as used in the U.K. and Canada, which is 1.1365225 liters exactly)
	"tbs"	Tablespoon (U.S. customary, traditional meaning). This shall be 0.5 U.S. fluid ounce, not 15mL (common in U.S.) or 20mL (common in Australia).
	"tsp"	Teaspoon (U.S. customary, traditional meaning), 1/6

Unit group	*Unit symbol*	*Description*
		fluid ounce in U.S. customary measure. This is not the 1/8 Imperial fl. oz. per Imperial units nor the modern teaspoon of 5 mL currently used in the U.S.; see “tspm”
	"tspm"	Modern teaspoon, 5mL
	"uk_gal"	U.K. / Imperial gallon, 4.54609 liters.
	"uk_pt"	U.K. / Imperial pint,1/8 of a UK gallon.
	"uk_qt"	U.K. / Imperial quart,1/4 of a UK gallon.
	"yd3" or **"yd^3"**	Cubic international yard

If a conversion factor (as listed above) is not exact, an implementation may use a more accurate conversion factor instead.

Implementation-defined unit names should contain a 'FULL STOP' (U+002E) character.

Evaluators shall support decimal prefixes for unit symbols marked with * and binary prefixes for unit symbols marked with †. Evaluators should not support prefixes for other unit symbols.

The unit symbols in parentheses are deprecated unit symbols; evaluators shall support these unit symbols.

Evaluators should use internationally-standardized unit name abbreviations for such additions where possible. Evaluators may support the obsolete symbols “p” and “P” as unit names for Pascals.

For purposes of this function, a year is exactly 365.25 days long.

Evaluators claiming to support this function shall permit the unit decimal prefixes specified in Table 25 - Decimal Prefixes for use in CONVERT to be prepended to any unit symbol marked with * in Table 24 - Unit names. Adding a unit prefix indicates multiplication of the (scalar) unit by the given prefix value; for example km indicates kilometres, and km2 or km^2 indicate square kilometres.

Table 25 - Decimal Prefixes for use in CONVERT

Unit Prefix	*Description*	*Prefix Value*
"Y"	yotta	1E+24
"Z"	zetta	1E+21
"E"	exa	1E+18
"P"	peta	1E+15
"T"	tera	1E+12
"G"	giga	1E+09
"M"	mega	1E+06
"k"	kilo	1E+03
"h"	hecto	1E+02
"da" or **"e"**	deka (**Note:** “e” is not a standard SI prefix	1E+01

Unit Prefix	*Description*	*Prefix Value*
`"d"`	deci	1E-01
`"c"`	centi	1E-02
`"m"`	milli	1E-03
`"u"`	micro N**ote:** this is “u”, not the standard SI µ	1E-06
`"n"`	nano	1E-09
`"p"`	pico	1E-12
`"f"`	femto	1E-15
`"a"`	atto	1E-18
`"z"`	zepto	1E-21
`"y"`	yocto	1E-24

Note: The prefix “e” for 10 [1] is nonstandard and included for backward compatibility with legacy applications and documents.

The unit names marked with † in Table 24 - Unit names (see the Information Unit group) shall also support the following binary prefixes per IEC 60027-2:

Table 26 - Binary prefixes for use in CONVERT

Binary Unit Prefix	*Description*	*Prefix Value*	*Derived from*
`"Yi"`	yobi	2^80 = 1 208 925 819 614 629 174 706 176	yotta
`"Zi"`	zebi	2^70 = 1 180 591 620 717 411 303 424	zetta
`"Ei"`	exbi	2^60 = 1 152 921 504 606 846 976	exa
`"Pi"`	pebi	2^50 = 1 125 899 906 842 624	peta
`"Ti"`	tebi	2^40 = 1 099 511 627 776	tera
`"Gi"`	gibi	2^30 = 1 073 741 824	giga
`"Mi"`	mebi	2^20 = 1 048 576	mega
`"Ki"`	kibi	2^10 = 1024	kilo

In the case where there is a naming conflict (a unit name with a prefix is the same as an unprefixed name), the unprefixed name shall take precedence.

Evaluators may implement this conversion by first converting to some SI unit (e.g., meter and kilogram), and then convert again to the final unit.

See also EUROCONVERT 6.16.29

6.16.19 COS

Summary: Return the cosine of an angle specified in radians.

Syntax: COS(*Number* ***N***)

Returns: *Number*

Constraints: None

Semantics: Computes the cosine of an angle specified in radians.

$$\mathrm{COS}(N) = 1 - \frac{N^2}{2!} + \frac{N^4}{4!} - \frac{N^6}{6!} + \ldots$$

See also ACOS 6.16.3, RADIANS 6.16.49, DEGREES 6.16.25

6.16.20 COSH

Summary: Return the hyperbolic cosine of the given hyperbolic angle.

Syntax: COSH(*Number* ***N***)

Returns: *Number*

Constraints: None

Semantics: Computes the hyperbolic cosine of a hyperbolic angle. The hyperbolic cosine is an analog of the ordinary (circular) cosine. The points (cosh t, sinh t) define the right half of the equilateral hyperbola, just as the points (cos t, sin t) define the points of a circle.

$$\mathrm{COSH}(N) = \frac{e^N + e^{-N}}{2}$$

See also ACOSH 6.16.4, SINH 6.16.56, TANH 6.16.70

6.16.21 COT

Summary: Return the cotangent of an angle specified in radians.

Syntax: COT(*Number* ***N***)

Returns: *Number*

Constraints: None

Semantics: Computes the cotangent of an angle specified in radians.

COT(x) = 1 / TAN(x)

See also ACOT 6.16.5, TAN 6.16.69, RADIANS 6.16.49, DEGREES 6.16.25, SIN 6.16.55, COS 6.16.19

6.16.22 COTH

Summary: Return the hyperbolic cotangent of the given hyperbolic angle.

Syntax: COTH(*Number* ***N***)

Returns: *Number*

Constraints: ***N*** ≠ 0

Semantics: Computes the hyperbolic cotangent of a hyperbolic angle. The hyperbolic cotangent is an analog of the ordinary (circular) cotangent.

$$\coth(N) = \frac{1}{\tanh(N)} = \frac{\cosh(N)}{\sinh(N)} = \frac{e^N + e^{-N}}{e^N - e^{-N}}$$

See also ACOSH 6.16.4, COSH 6.16.20, SINH 6.16.56, TANH 6.16.70

6.16.23 CSC

Summary: Return the cosecant of an angle specified in radians.

Syntax: CSC(*Number* ***N***)

Returns: *Number*

Constraints: None

Semantics: Computes the cosecant cosine of an angle specified in radians. Equivalent to:

1 / SIN(***N***)

See also SIN 6.16.55

6.16.24 CSCH

Summary: Return the hyperbolic cosecant of the given angle specified in radians.

Syntax: CSCH(*Number* ***N***)

Returns: *Number*

Constraints: None

Semantics: Computes the hyperbolic cosecant of a hyperbolic angle. This is equivalent to:

1 / SINH(***N***)

See also SINH 6.16.56

6.16.25 DEGREES

Summary: Convert radians to degrees.

Syntax: DEGREES(*Number* ***N***)

Returns: *Number*

Constraints: None

Semantics: Converts a number in radians into a number in degrees. DEGREES(***N***) is equal to ***N*** * 180 / π.

See also RADIANS 6.16.49, PI 6.16.45

6.16.26 DELTA

Summary: Report if two numbers are equal, returns 1 if they are equal.

Syntax: DELTA(*Number* ***X*** [; *Number* ***Y*** = 0])

Returns: *Number*

Constraints: None

Semantics: If ***X*** and ***Y*** are equal, return 1, else 0. ***Y*** is set to 0 if omitted.

See also Infix operator “=” 6.4.7

6.16.27 ERF

Summary: Calculates the error function.

Syntax: ERF(*Number* ***Z0*** [; *Number* ***Z1***])

Returns: *Number*

Constraints: None

Semantics: With a single argument, returns the error function of ***Z0***:

$$ERF(Z0) = \frac{2}{\sqrt{(\pi)}} \int_0^{Z0} e^{-t^2} dt$$

With two arguments, returns

$$ERF(Z0;Z1)=\frac{2}{\sqrt{(\pi)}}\int_{Z0}^{Z1} e^{-t^2}\,dt$$

See also ERFC 6.16.28

6.16.28 ERFC

Summary: Calculates the complementary error function.

Syntax: ERFC(*Number* ***Z***)

Returns: *Number*

Constraints: None

Semantics: returns the complementary error function of ***Z***: ERFC(***Z***) = 1 – ERF(***Z***)

See also ERF 6.16.27

6.16.29 EUROCONVERT

Summary: Converts a Number, representing a value in one European currency, to an equivalent value in another European currency, according to the fixed conversion rates defined by the Council of the European Union.

Syntax: EUROCONVERT(*Number* ***N*** ; *Text* ***From*** ; *Text* ***To*** [; *Logical* ***FullPrecision*** = FALSE [; *Integer* ***TriangulationPrecision***]])

Returns: *Currency*

Constraints: ***From*** and ***To*** shall be known to the evaluator. ***TriangulationPrecision*** shall be ≥ 3, if not omitted.

If an evaluator does not support the parameters ***FullPrecision*** and ***TriangulationPrecision***, ***FullPrecision*** should be assumed to be false.

Semantics: Returns the given money value of a conversion from ***From*** currency into ***To*** currency. Both ***From*** and ***To*** shall be the official [ISO4217] abbreviation for the given currency; note that these are in upper case, but the function accepts lower case or mixed case as well. If ***From*** and ***To*** are equal currencies, the value ***N*** is returned, no precision or triangulation is applied.

The function shall use the rates of exchange as set by the European Commission, as follows:

Table 27 - EUROCONVERT

From	***To***	***Rate***	***Currency***	***Decimals***
"EUR"	"ATS"	13.7603	Austrian Schilling	2
"EUR"	"BEF"	40.3399	Belgian Franc	0
"EUR"	"DEM"	1.95583	German Mark	2
"EUR"	"ESP"	166.386	Spanish Peseta	0
"EUR"	"FIM"	5.94573	Finnish Markka	2
"EUR"	"FRF"	6.55957	French Franc	2
"EUR"	"IEP"	0.787564	Irish Pound	2
"EUR"	"ITL"	1936.27	Italian Lira	0
"EUR"	"LUF"	40.3399	Luxembourg Franc	0
"EUR"	"NLG"	2.20371	Dutch Guilder	2

From	*To*	*Rate*	*Currency*	*Decimals*
"EUR"	"PTE"	200.482	Portuguese Escudo	2
"EUR"	"GRD"	340.750	Greek Drachma	2
"EUR"	"SIT"	239.640	Slovenian Tolar	2
“EUR”	“MTL”	0.429300	Maltese Lira	2
“EUR”	“CYP”	0.585274	Cypriot Pound	2
"EUR"	"SKK"	30.1260	Slovak Koruna	2

As new member countries adopt the Euro, new conversion rates will become active and evaluators may add them using the respective [ISO4217] codes and fixed rates as defined by the European Council, on the basis of a European Commission proposal.

Note:
The European Commission's Euro entry page is http://ec.europa.eu/euro/
The conversion rates and triangulation rules are available at http://ec.europa.eu/economy_finance/euro/adoption/conversion/index_en.htm with links to the European Council Regulation legal documents at the http://eur-lex.europa.eu/ European Union law database server.

If ***FullPrecision*** is omitted or FALSE, the result is rounded according to the decimals of the ***To*** currency. If ***FullPrecision*** is TRUE the result is not rounded.

If ***TriangulationPrecision*** is given and ≥ 3, the intermediate result of a triangular conversion (currency1,EUR,currency2) is rounded to that precision. If ***TriangulationPrecision*** is omitted, the intermediate result is not rounded. Also if ***To*** currency is “EUR”, ***TriangulationPrecision*** precision is used as if triangulation was needed and conversion from EUR to EUR was applied.

See also CONVERT

6.16.30 EVEN

Summary: Rounds a number up to the nearest even integer. Rounding is away from zero.

Syntax: EVEN(*Number* ***N***)

Returns: *Number*

Constraints: None

Semantics: Returns the even integer whose sign is the same as ***N***'s and whose absolute value is greater than or equal to the absolute value of ***N***.

See also ODD 6.16.44

6.16.31 EXP

Summary: Returns e raised by the given number.

Syntax: EXP(*Number* ***X***)

Returns: *Number*

Constraints: None

Semantics: Computes

$$e^{X} = 1 + \frac{X}{1!} + \frac{X^2}{2!} + \frac{X^3}{3!} + \frac{X^n}{n!} + \ldots$$

See also LOG 6.16.40, LN 6.16.39

6.16.32 FACT

Summary: Return factorial (!).

Syntax: FACT(*Integer* ***F***)

Returns: *Number*

Constraints: ***F*** ≥ 0

Semantics: Return the factorial

$$F\,! = F \cdot (F-1) \cdot (F-2) \cdot \ldots 1$$

F(0) = ***F***(1) = 1.

See also Infix Operator "*" 6.4.4, GAMMA 6.16.34

6.16.33 FACTDOUBLE

Summary: Returns double factorial (!!).

Syntax: FACTDOUBLE(*Integer* ***F***)

Returns: *Number*

Constraints: ***F*** ≥ 0

Semantics: Return

$$F\,! = F \cdot (F-2) \cdot (F-4) \cdot \ldots 1$$

Double factorial is computed by multiplying every other number in the 1..N range, with N always being included.

See also Infix Operator "*" 6.4.4, GAMMA 6.16.34, FACT 6.16.32

6.16.34 GAMMA

Summary: Return gamma function value.

Syntax: GAMMA(*Number* ***N***)

Returns: *Number*

Constraints: ***N*** ≠ 0 and ***N*** not a negative integer.

Semantics: Return

$$\Gamma(\mathrm{N}) = \int_0^\infty t^{N-1} e^{-t}\, dt$$

with Γ(***N*** + 1) = ***N*** * Γ(***N***). Note that for non-negative integers ***N***, Γ(***N*** + 1) = ***N***! = FACT(***N***). Note that GAMMA can accept non-integers.

See also FACT 6.16.32

6.16.35 GAMMALN

Summary: Returns the natural logarithm of the GAMMA function.

Syntax: GAMMALN(*Number* ***X***)

Returns: *Number*

Constraints: For each ***X***, ***X*** > 0

Semantics: Returns the same value as LN(GAMMA(***X***))

See also GAMMA 6.16.34, FACT 6.16.32

6.16.36 GCD

Summary: Returns the greatest common divisor (GCD)

Syntax: GCD({ *NumberSequenceList* ***X*** }$^{+}$)

Returns: *Number*

Constraints: For all a in ***X***: INT(a) ≥ 0 and for at least one a in ***X***: INT(a) > 0

Semantics: Return the largest integer N such that for every a in ***X***: INT(a) is a multiple of N.

Note: If for all a in ***X***: INT(a) = 0 the return value is implementation-defined but is either an Error or 0.

See also LCM 6.16.38, INT 6.17.2

6.16.37 GESTEP

Summary: Returns 1 if a number is greater than or equal to another number, else returns 0.

Syntax: GESTEP(*Number* ***X*** [; *Number* ***Step*** = 0])

Returns: *Number*

Semantics: Number ***X*** is tested against number ***Step***. If greater or equal 1 is returned, else 0. The second parameter is assumed 0 if omitted. If one of the parameters is not a Number, the function results in an Error.

6.16.38 LCM

Summary: Returns the least common multiplier

Syntax: LCM({ *NumberSequenceList* ***X*** }$^{+}$)

Returns: *Number*

Constraints: For all in ***X***: INT(***X***) = ***X***, ***X*** ≥ 0

Semantics: Return the smallest integer that is the multiple of the given values. Each value has INT applied to it first. Note that if given two numbers, ABS(a * b) = LCM(a;b) * GCD(a;b).

See also GCD 6.16.36, INT 6.17.2

6.16.39 LN

Summary: Return the natural logarithm of a number.

Syntax: LN(*Number* ***X***)

Returns: *Number*

Constraints: ***X*** > 0

Semantics: Computes the natural logarithm (base e) of the given number.

$$\mathrm{LN}(X)=2\left[\frac{x-1}{x+1}+\frac{1}{3}\left(\frac{x-1}{x+1}\right)^{3}+\frac{1}{5}\left(\frac{x-1}{x+1}\right)^{5}+\ldots\right]$$

See also LOG 6.16.40, LOG10 6.16.41, POWER 6.16.46, EXP 6.16.31

6.16.40 LOG

Summary: Return the logarithm of a number in a specified base.

Syntax: LOG(*Number* ***N*** [; *Number* ***Base*** = 10])

Returns: *Number*

Constraints: ***N*** > 0

Semantics: Computes the logarithm of a number in the specified base. Note that if the base is not specified, the logarithm base 10 is returned.

See also LOG10 6.16.41, LN 6.16.39, POWER 6.16.46, EXP 6.16.31

6.16.41 LOG10

Summary: Return the base 10 logarithm of a number.

Syntax: LOG10(*Number* ***N***)

Returns: *Number*

Constraints: ***N*** > 0

Semantics: Computes the base 10 logarithm of a number.

See also LOG 6.16.40, LN 6.16.39, POWER 6.16.46, EXP 6.16.31

6.16.42 MOD

Summary: Return the remainder when one number is divided by another number.

Syntax: MOD(*Number* ***A*** ; *Number* ***B***)

Returns: *Number*

Constraints: ***B*** != 0

Semantics: Computes the remainder of ***A*** / ***B***. The remainder has the same sign as ***B***.

See also Infix Operator "/" 6.4.5, QUOTIENT 6.16.48

6.16.43 MULTINOMIAL

Summary: Returns the multinomial for the given values.

Syntax: MULTINOMIAL({ *NumberSequence* ***A*** }$^{+}$)

Returns: *Number*

Constraints: None

Semantics: Returns the multinomial of the sequence ***A*** = (a_1, a_2, ..., a_n). Multinomial is defined as FACT(a_1 + a_2 +...+ a_n) / (FACT(a_1) * FACT(a_2) *...* FACT(a_n))

See also FACT 6.16.32

6.16.44 ODD

Summary: Rounds a number up to the nearest odd integer, where "up" means "away from 0".

Syntax: ODD(*Number* ***N***)

Returns: *Number*

Constraints: None

Semantics: Returns the odd integer whose sign is the same as ***N***'s and whose absolute value is greater than or equal to the absolute value of ***N***. In other words, any "rounding" is away from zero. By definition, ODD(0) is 1.

See also EVEN 6.16.30

6.16.45 PI

Summary: Return the approximate value of π.

Syntax: PI()

Returns: *Number*

Constraints: None.

Semantics: This function takes no arguments and returns the (approximate) value of π (pi). Evaluators should use the closest possible numerical representation that is possible in their representation of numbers.

See also SIN 6.16.55, COS 6.16.19

6.16.46 POWER

Summary: Return the value of one number raised to the power of another number.

Syntax: POWER(*Number* ***A*** ; *Number* ***B***)

Returns: *Number*

Constraints: None

Semantics: Computes ***A*** raised to the power ***B***.

- POWER(0,0) is implementation-defined, but shall be one of 0,1, or an Error.
- POWER(0,***B***), where ***B*** < 0, shall return an Error.
- POWER(***A***,***B***), where ***A*** ≤ 0 and INT(***B***) != ***B***, is implementation-defined.

See also LOG 6.16.40, LOG10 6.16.41, LN 6.16.39, EXP 6.16.31

6.16.47 PRODUCT

Summary: Multiply the set of numbers, including all numbers inside ranges.

Syntax: PRODUCT({ *NumberSequenceList* ***N*** }$^{+}$)

Returns: *Number*

Constraints: None

Semantics: Returns the product of the Numbers (and only the Numbers, i.e., not Text inside ranges).

See also SUM 6.16.61

6.16.48 QUOTIENT

Summary: Return the integer portion of a division.

Syntax: QUOTIENT(*Number* ***A*** ; *Number* ***B***)

Returns: *Number*

Constraints: ***B*** ≠ 0

Semantics: Return the integer portion of a division.

See also MOD 6.16.42

6.16.49 RADIANS

Summary: Convert degrees to radians.

Syntax: RADIANS(*Number* ***N***)

Returns: *Number*

Constraints: None

Semantics: Converts a number in degrees into a number in radians. RADIANS(***N***) is equal to ***N*** * PI() / 180.

See also DEGREES 6.16.25, PI 6.16.45

6.16.50 RAND

Summary: Return a random number between 0 (inclusive) and 1 (exclusive).

Syntax: RAND()

Returns: *Number*

Semantics: This function takes no arguments and returns a random number between 0 (inclusive) and 1 (exclusive). Note that unlike most functions, this function will typically return *different* values when called each time with the same (empty set of) parameters.

See also RANDBETWEEN 6.16.51

6.16.51 RANDBETWEEN

Summary: Return a random integer number between ***A*** and ***B***.

Syntax: RANDBETWEEN(*Integer* ***A*** ; *Integer* ***B***)

Returns: *Integer*

Constraints: ***A*** ≤ ***B***

Semantics: The function returns a random integer number between ***A*** and ***B*** inclusive. Note that unlike most functions, this function will often return *different* values when called each time with the same parameters.

See also RAND 6.16.50

6.16.52 SEC

Summary: Return the secant of an angle specified in radians.

Syntax: SEC(*Number* ***N***)

Returns: *Number*

Constraints: None

Semantics: Computes the secant cosine of an angle specified in radians. Equivalent to:

1 / COS(***N***)

See also SIN 6.16.55

6.16.53 SERIESSUM

Summary: Returns the sum of a power series.

Syntax: SERIESSUM(*Number* ***X*** ; *Number* ***N*** ; *Number* ***M*** ; *Array* ***Coefficients***)

- ***X***: the independent variable of the power series.
- ***N***: the initial power to which ***X*** is to be raised.
- ***M***: the increment by which to increase ***N*** for each term in the series.
- ***Coefficients***: a set of coefficients by which each successive power of the variable ***X*** is multiplied.

Returns: *Number*

Constraints:

All elements of ***Coefficients*** are of type Number.

X ≠ 0 if any of the exponents, which are generated from ***N*** and ***M***, are negative.

Semantics: Returns a sum of powers of the number ***X***.

With C being the number of coefficients the function is computed as:

$$SERIESSUM = \sum_{i=1}^{C} Coefficient_i \cdot X^{(N+(i-1)M)}$$

If ***X*** = 0 and all of the exponents are non-negative then 0^0 shall be set to 1 and $0^{(exponent>0)}$ shall be set to 0.

6.16.54 SIGN

Summary: Return the sign of a number.

Syntax: SIGN(*Number* ***N***)

Returns: *Number*

Constraints: None

Semantics: If ***N*** < 0, returns -1; if ***N*** > 0, returns +1; if ***N*** = 0, returns 0.

See also ABS 6.16.2

6.16.55 SIN

Summary: Return the sine of an angle specified in radians.

Syntax: SIN(*Number* ***N***)

Returns: *Number*

Constraints: None

Semantics: Computes the sine of an angle specified in radians.

$$\mathrm{SIN}(N) = N - \frac{N^3}{3!} + \frac{N^5}{5!} - \frac{N^7}{7!} + \ldots$$

See also ASIN 6.16.7, RADIANS 6.16.49, DEGREES 6.16.25

6.16.56 SINH

Summary: Return the hyperbolic sine of the given hyperbolic angle.

Syntax: SINH(*Number* ***N***)

Returns: *Number*

Constraints: None

Semantics: Computes the hyperbolic sine of a hyperbolic angle. The hyperbolic sine is an analog of the ordinary (circular) sine. The points (cosh t, sinh t) define the right half of the equilateral hyperbola, just as the points (cos t, sin t) define the points of a circle.

$$\mathrm{SINH}(N) = \frac{e^N - e^{-N}}{2}$$

See also ASINH 6.16.8, COSH 6.16.20, TANH 6.16.70

6.16.57 SECH

Summary: Return the hyperbolic secant of the given angle specified in radians.

Syntax: SECH(*Number* ***N***)

Returns: *Number*

Constraints: None

Semantics: Computes the hyperbolic secant of a hyperbolic angle. This is equivalent to:

1 / COSH(***N***)

See also SINH 6.16.56, COSH 6.16.20, CSCH 6.16.24

6.16.58 SQRT

Summary: Return the square root of a number.

Syntax: SQRT(*Number* ***N***)

Returns: *Number*

Constraints: ***N*** ≥ 0

Semantics: Returns the square root of a non-negative number. This function shall produce an Error if given a negative number; for producing complex numbers, see IMSQRT.

See also POWER 6.16.46, IMSQRT 6.8.24, SQRTPI 6.16.59

6.16.59 SQRTPI

Summary: Return the square root of a number multiplied by π (pi).

Syntax: SQRTPI(*Number* ***N***)

Returns: *Number*

Constraints: ***N*** ≥ 0

Semantics: Returns the square root of a non-negative number after it was first multiplied by π, that is, SQRT(***N*** * PI()). This function shall produce an Error if given a negative number; for producing complex numbers, see IMSQRT.

See also POWER 6.16.46, SQRT 6.16.58, PI 6.16.45, IMSQRT 6.8.24

6.16.60 SUBTOTAL

Summary: Evaluates a function on a range.

Syntax: SUBTOTAL(*Integer* ***Function*** ; *NumberSequence* ***Sequence***)

Returns: *Number*

Constraints: None

Semantics: Computes a given function on a number sequence. The function is denoted by the first parameter: The difference from standard functions is that all members of the sequence are excluded which:

- include a call to SUBTOTAL in their formula
- are in a row that is hidden by a `table:visibility="filter"` attribute of the `<table:table-row>` element (OpenDocument, Part 3, 19.754).
- are in a row that is hidden by a `table:visibility="collapse"` attribute of the `<table:table-row>` element if the function ID is one of 101...111.

Table 28 - SUBTOTAL

Function	Exclude hidden by filter	Exclude hidden by filter or collapsed
AVERAGE	1	101
COUNT	2	102
COUNTA	3	103
MAX	4	104

MIN	5	105
PRODUCT	6	106
STDEV	7	107
STDEVP	8	108
SUM	9	109
VAR	10	110
VARP	11	111

See also SUM 6.16.61, AVERAGE 6.18.3

6.16.61 SUM

Summary: Sum (add) the set of numbers, including all numbers in ranges.

Syntax: SUM({ *NumberSequenceList* ***N*** }$^{+}$)

Returns: *Number*

Constraints: ***N*** != {}; Evaluators may evaluate expressions that do not meet this constraint.

Semantics: Adds Numbers (and only Numbers) together (see the text on conversions).

See also AVERAGE 6.18.3

6.16.62 SUMIF

Summary: Sum the values of cells in a range that meet a criteria.

Syntax: SUMIF(*ReferenceList|Reference* ***R*** ; *Criterion* ***C*** [; *Reference* ***S***])

Returns: *Number*

Constraints: Does not accept constant values as the range parameter.

Semantics: Sums the values of type Number in the range ***R*** or ***S*** that meet the Criterion ***C*** (4.11.8).

If ***S*** is not given, ***R*** may be a reference list. If ***S*** is given, ***R*** shall not be a reference list with more than 1 references and an Error be generated if it was.

If the optional range ***S*** is included, then the values of ***S*** starting from the top left cell and matching the geometry of ***R*** (same number of rows and columns) are summed if the corresponding value in ***R*** meets the Criterion. The actual range ***S*** is not considered. If the resulting range exceeds the sheet bounds, column numbers larger than the maximum column and row numbers larger than the maximum row are silently ignored, no Error is generated for this case.

The values returned may vary depending upon the HOST-USE-REGULAR-EXPRESSIONS or HOST-USE-WILDCARDS or HOST-SEARCH-CRITERIA-MUST-APPLY-TO-WHOLE-CELL properties. 3.4

See also COUNTIF 6.13.9, SUM 6.16.61, Infix Operator "=" 6.4.7, Infix Operator "<>" 6.4.8, Infix Operator Ordered Comparison ("<", "<=", ">", ">=") 6.4.9

6.16.63 SUMIFS

Summary: Sum the values of cells in a range that meet multiple criteria in multiple ranges.

Syntax: SUMIFS(*Reference* ***R*** ; *Reference* ***R1*** ; *Criterion* ***C1*** [; *Reference* ***R2*** ; *Criterion* ***C2***]...)

Returns: *Number*

Constraints: Does not accept constant values as the reference parameter.

Semantics: Sums the value of cells in range ***R*** that meet the Criterion ***C1*** in the reference range ***R1*** and the Criterion ***C2*** in the reference range ***R2***, and so on (4.11.8). All reference ranges shall have the same dimension and size, else an Error is returned. A logical AND is applied between each array result of each selection; an entry is counted only if the same position in each array is the result of a criteria match.

The values returned may vary depending upon the HOST-USE-REGULAR-EXPRESSIONS or HOST-USE-WILDCARDS or HOST-SEARCH-CRITERIA-MUST-APPLY-TO-WHOLE-CELL properties. 3.4

See also AVERAGEIFS 6.18.6, COUNTIFS 6.13.10, SUMIF 6.16.62, Infix Operator "=" 6.4.7, Infix Operator "<>" 6.4.8, Infix Operator Ordered Comparison ("<", "<=", ">", ">=") 6.4.9

6.16.64 SUMPRODUCT

Summary: Returns the sum of the products of the matrix elements.

Syntax: SUMPRODUCT({ *ForceArray Array* ***A*** }$^{+}$)

Returns: *Number*

Constraints: All matrices shall have the same dimensions.

Semantics: Multiplies the corresponding elements of all matrices and returns the sum of them.

$$SUMPRODUCT\left(A_1, A_2, \ldots, A_K\right) = \sum_{m=1}^{M}\sum_{n=1}^{N}\left(\prod_{k=1}^{K} a_{k,mn}\right)$$

where $a_{k,mn}$ denotes an element of the matrix A_K.

6.16.65 SUMSQ

Summary: Sum (add) the set of squares of numbers, including all numbers in ranges

Syntax: SUMSQ({ *NumberSequence* ***N*** }$^{+}$)

Returns: *Number*

Constraints: ***N*** != {}; Evaluators may evaluate expressions that do not meet this constraint.

Semantics: Adds squares of Numbers (and only Numbers) together. See the text on conversions.

6.16.66 SUMX2MY2

Summary: Returns the sum of the difference between the squares of the matrices ***A*** and ***B***.

Syntax: SUMX2MY2(*ForceArray Array* ***A*** ; *ForceArray Array* ***B***)

Returns: *Number*

Constraints: Both matrices shall have the same dimensions.

Semantics: Sums up the differences of the corresponding elements squares for two matrices.

$$SUMX2MY2\left(A, B\right) = \sum_{m=1}^{M}\sum_{n=1}^{N}\left(a_{mn}^{2} - b_{mn}^{2}\right)$$

6.16.67 SUMX2PY2

Summary: Returns the total sum of the squares of the matrices ***A*** and ***B***.

Syntax: SUMX2PY2(*ForceArray Array* ***A*** ; *ForceArray Array* ***B***)

Returns: *Number*

Constraints: Both matrices shall have the same dimensions.

Semantics: Sums up the squares of each element of the two matrices.

$$SUMX2PY2(A, B) = \sum_{m=1}^{M} \sum_{n=1}^{N} \left(a_{mn}^2 + b_{mn}^2 \right)$$

6.16.68 SUMXMY2

Summary: Returns the sum of the squares of the differences between matrix ***A*** and ***B***.

Syntax: SUMXMY2(*ForceArray Array* ***A*** ; *ForceArray Array* ***B***)

Returns: *Number*

Constraints: Both matrices shall have the same dimensions.

Semantics: Sums up the squares of the differences of the corresponding elements for two matrices.

$$SUMXMY2(A, B) = \sum_{m=1}^{M} \sum_{n=1}^{N} \left(a_{mn} - b_{mn} \right)^2$$

6.16.69 TAN

Summary: Return the tangent of an angle specified in radians

Syntax: TAN(*Number* ***N***)

Returns: *Number*

Constraints: None

Semantics: Computes the tangent of an angle specified in radians.

TAN(x) = SIN(x) / COS(x)

See also ATAN 6.16.9, ATAN2 6.16.10, RADIANS 6.16.49, DEGREES 6.16.25, SIN 6.16.55, COS 6.16.19, COT 6.16.21

6.16.70 TANH

Summary: Return the hyperbolic tangent of the given hyperbolic angle

Syntax: TANH(*Number* ***N***)

Returns: *Number*

Constraints: None

Semantics: Computes the hyperbolic tangent of a hyperbolic angle. The hyperbolic tangent is an analog of the ordinary (circular) tangent. The points (cosh t, sinh t) define the right half of the equilateral hyperbola, just as the points (cos t, sin t) define the points of a circle.

$$\mathrm{TANH}(N) = \frac{\mathrm{SINH}(N)}{\mathrm{COSH}(N)} = \frac{e^N - e^{-N}}{e^N + e^{-N}}$$

See also ATANH 6.16.11, SINH 6.16.56, COSH 6.16.20, FISHERINV 6.18.27

6.17 Rounding Functions

6.17.1 CEILING

Summary: Round a number ***N*** **up** to the nearest multiple of the second parameter, *significance*.

Syntax: CEILING(*Number* ***N*** [; [*Number* ***Significance***] [; *Number* ***Mode***]])

Returns: *Number*

Constraints: Both ***N*** and ***Significance*** shall be numeric and have the same sign if not 0.

Semantics: Rounds a number up to a multiple of the second number. If ***Significance*** is omitted or an empty parameter (two consecutive ;; semicolons) it is assumed to be -1 if ***N*** is negative and +1 if ***N*** is non-negative, making the function act like the normal mathematical *ceiling* function if ***Mode*** is not given or zero. If ***Mode*** is given and not equal to zero, the absolute value of ***N*** is rounded away from zero to a multiple of the absolute value of ***Significance*** and then the sign applied . If ***Mode*** is omitted or zero, rounding is toward positive infinity; the number is rounded to the smallest multiple of significance that is equal-to or greater than ***N***. If any of the two parameters ***N*** or ***Significance*** is zero, the result is zero.

Note: Many application user interfaces have a CEILING function with only two parameters, and somewhat different semantics than given here (e.g., they operate as if there was a non-zero ***Mode*** value). These CEILING functions are inconsistent with the standard mathematical definition of CEILING.

See also FLOOR 6.17.3, INT 6.17.2

6.17.2 INT

Summary: Rounds a number down to the nearest integer.

Syntax: INT(*Number* ***N***)

Returns: *Number*

Constraints: None

Semantics: Returns the nearest integer whose value is less than or equal to ***N***. Rounding is towards negative infinity.

See also ROUND 6.17.5, TRUNC 6.17.8

6.17.3 FLOOR

Summary: Round a number ***N*** **down** to the nearest multiple of the second parameter, *significance*.

Syntax: FLOOR(*Number* ***N*** [; [*Number* ***Significance***] [; *Number* ***Mode***]])

Returns: *Number*

Constraints: Both ***N*** and ***Significance*** shall be numeric and have the same sign.

Semantics: Rounds a number down to a multiple of the second number. If ***Significance*** is omitted or an empty parameter (two consecutive ;; semicolons) it is assumed to be -1 if ***N*** is negative and +1 if ***N*** is non-negative, making the function act like the normal mathematical *floor* function if ***Mode*** is not given or zero. If ***Mode*** is given and not equal to zero, the absolute value of ***N*** is rounded toward zero to a multiple of the absolute value of ***Significance*** and then the sign applied . Otherwise, it rounds toward negative infinity, and the result is the largest multiple of ***Significance*** that is less than or equal to ***N***. If any of the two parameters ***N*** or ***Significance*** is zero, the result is zero.

Note: Many application user interfaces have a FLOOR function with only two parameters, and somewhat different semantics than given here (e.g., they operate as if there was a non-zero ***Mode*** value). These FLOOR functions are inconsistent with the standard mathematical definition of FLOOR.

See also CEILING 6.17.1, INT 6.17.2

6.17.4 MROUND

Summary: Rounds the number to given multiple.

Syntax: MROUND(*Number* ***A*** ; *Number* ***B***)

Returns: *Number*

Constraints: None

Semantics: Returns the number X, for which the following holds: X/***B*** = INT(X / ***B***) (***B*** divides X), and for any other Y with the same property, ABS(Y – ***A***) ≥ ABS(X - ***A***). In case that two such X exist, the greater one is the result. In less formal language, this function rounds the number ***A*** to multiples of ***B***.

See also ABS 6.16.2, INT 6.17.2, ROUND 6.17.5

6.17.5 ROUND

Summary: Rounds the value ***X*** to the nearest multiple of the power of 10 specified by ***Digits***.

Syntax: ROUND(*Number* ***X*** [; *Number* ***Digits*** = 0])

Returns: *Number*

Constraints: None

Semantics: Round number ***X*** to the precision specified by ***Digits***. The number ***X*** is rounded to the nearest power of 10 given by $10^{-Digits}$. If ***Digits*** is zero, or absent, round to the nearest decimal integer. If ***Digits*** is non-negative, round to the specified number of decimal places. If ***Digits*** is negative, round to the left of the decimal point by -***Digits*** places. If ***X*** is halfway between the two nearest values, the result shall round away from zero. Note that if ***X*** is a Number, and ***Digits*** ≤ 0, the results will always be an integer (without a fractional component).

See also TRUNC 6.17.8, INT 6.17.2

6.17.6 ROUNDDOWN

Summary: Rounds the value ***X*** towards zero to the number of digits specified by ***Digits***.

Syntax: ROUNDDOWN(*Number* ***X*** [; *Integer* ***Digits*** = 0])

Returns: *Number*

Constraints: None

Semantics: Round ***X*** towards zero, to the precision specified by ***Digits***. The number returned is a multiple of $10^{-Digits}$. If ***Digits*** is zero, or absent, round to the largest decimal integer whose absolute value is smaller or equal to the absolute value of ***X***. If ***Digits*** is positive, round towards zero to the specified number of decimal places. If ***Digits*** is negative, round towards zero to the left of the decimal point by -***Digits*** places.

See also TRUNC 6.17.8, INT 6.17.2, ROUND 6.17.5, ROUNDUP 6.17.7

6.17.7 ROUNDUP

Summary: Rounds the value ***X*** away from zero to the number of digits specified by ***Digits***

Syntax: ROUNDUP(*Number* ***X*** [; *Integer* ***Digits*** = 0])

Returns: *Number*

Constraints: None

Semantics: Round ***X*** away from zero, to the precision specified by ***Digits***. The number returned is a multiple of $10^{-Digits}$. If ***Digits*** is zero, or absent, round to the smallest decimal

integer whose absolute value is larger or equal to the absolute value of ***X***. If ***Digits*** is positive, round away from zero to the specified number of decimal places. If ***Digits*** is negative, round away from zero to the left of the decimal point by -***Digits*** places.

See also TRUNC 6.17.8, INT 6.17.2, ROUND 6.17.5, ROUNDDOWN 6.17.6

6.17.8 TRUNC

Summary: Truncate a number to a specified number of digits.

Syntax: TRUNC(*Number* ***A*** ; *Integer* ***B***)

Returns: *Number*

Constraints: None

Semantics: Truncate number ***A*** to the number of digits specified by ***B***. If ***B*** is zero, or absent, truncate to an integer. If ***B*** is positive, truncate to the specified number of decimal places. If ***B*** is negative, truncate to the left of the decimal point.

See also ROUND 6.17.5, INT 6.17.2

6.18 Statistical Functions

6.18.1 General

The following are statistical functions (functions that report information on a set of numbers). Some functions that could also be considered statistical functions, such as SUM, are listed elsewhere.

6.18.2 AVEDEV

Summary: Calculates the average of the absolute deviations of the values in list.

Syntax: AVEDEV({ *NumberSequenceList* ***N*** }$^{+}$)

Returns: *Number*

Constraints: None.

Semantics: For a list ***N*** containing n numbers x_1 to x_n, with average $\bar{x}$, AVEDEV(***N***) is equal to:

$$\frac{1}{n}\sum_{i=1}^{n}\left|\left(x_i-\bar{x}\right)\right|$$

See also SUM 6.16.61, AVERAGE 6.18.3

6.18.3 AVERAGE

Summary: Average the set of numbers

Syntax: AVERAGE({ *NumberSequence* ***N*** }$^{+}$)

Returns: *Number*

Constraints: At least one Number included. Returns an Error if no Numbers provided.

Semantics: Computes SUM(***N***) / COUNT(***N***).

See also SUM 6.16.61, COUNT 6.13.6

6.18.4 AVERAGEA

Summary: Average values, including values of type Text and Logical.

Syntax: AVERAGEA({ *Any* ***N*** }$^{+}$)

Returns: *Number*

Constraints: At least one value included. Returns an Error if no value provided.

Semantics: A variant of the AVERAGE function that includes values of type Text and Logical. Text values are treated as number 0. Logical TRUE is treated as 1 and FALSE is treated as 0. Empty cells are not included. Any ***N*** may be of type ReferenceList.

See also AVERAGE 6.18.3

6.18.5 AVERAGEIF

Summary: Average the values of cells in a range that meet a criteria.

Syntax: AVERAGEIF(*Reference* ***R*** ; *Criterion* ***C*** [; *Reference* ***A***])

Returns: *Number*

Constraints: Does not accept constant values as reference parameters.

Semantics: If reference ***A*** is omitted, averages the values of cells in the reference range ***R*** that meet the Criterion ***C*** *(4.11.8)*. If reference ***A*** is given, averages the values of cells of a range that is constructed using the top left cell of reference ***A*** and applying the dimensions, shape and size, of reference ***R***. If no cell in range ***R*** matches the Criterion ***C***, an Error is returned. If no Numbers are in the range to be averaged, an Error is returned.

The values returned may vary depending upon the HOST-USE-REGULAR-EXPRESSIONS or HOST-USE-WILDCARDS or HOST-SEARCH-CRITERIA-MUST-APPLY-TO-WHOLE-CELL properties. 3.4

See also AVERAGEIFS 6.18.6, COUNTIF 6.13.9, SUMIF 6.16.62, Infix Operator "=" 6.4.7, Infix Operator "<>" 6.4.8, Infix Operator Ordered Comparison ("<", "<=", ">", ">=") 6.4.9

6.18.6 AVERAGEIFS

Summary: Average the values of cells that meet multiple criteria in multiple ranges.

Syntax: AVERAGEIFS(*Reference* ***A*** ; *Reference* ***R1*** ; *Criterion* ***C1*** [; *Reference* ***R2*** ; *Criterion* ***C2***]...)

Returns: *Number*

Constraints: Does not accept constant values as reference parameters.

Semantics: Averages the values of cells in the reference range ***A*** that meet the Criterion ***C1*** in the reference range ***R1*** and the Criterion ***C2*** in the reference range ***R2***, and so on (4.11.8). All reference ranges shall have the same dimension and size, else an Error is returned. ***A*** logical AND is applied between each array result of each selection; a cell of reference range ***A*** is evaluated only if the same position in each array is the result of a Criterion match. If no numbers are in the result set to be averaged, an Error is returned.

The values returned may vary depending upon the HOST-USE-REGULAR-EXPRESSIONS or HOST-USE-WILDCARDS or HOST-SEARCH-CRITERIA-MUST-APPLY-TO-WHOLE-CELL properties. 3.4

See also AVERAGEIF 6.18.5, COUNTIFS 6.13.10, SUMIFS 6.16.63, Infix Operator "=" 6.4.7, Infix Operator "<>" 6.4.8, Infix Operator Ordered Comparison ("<", "<=", ">", ">=") 6.4.9

6.18.7 BETADIST

Summary: returns the value of the probability density function or the cumulative distribution function for the beta distribution.

Syntax: BETADIST(*Number* ***x*** ; *Number* ***α*** ; *Number* ***β*** [; *Number* ***a*** = 0 [; *Number* ***b*** = 1 [; *Logical* ***Cumulative*** = TRUE]]])

Returns: *Number*

Constraints: ***α*** > 0, ***β*** > 0, ***a*** < ***b***,
If ***α*** < 1, then the density function has ***a*** pole at ***x*** = ***a***.

If ***β*** < 1, then the density function has ***a*** pole at ***x*** = ***b***.
In both cases, if ***x*** = ***a*** respectively ***x*** = ***b*** and ***Cumulative*** = FALSE, an Error is returned.

Semantics: If ***Cumulative*** is FALSE, BETADIST returns 0 if ***x*** < ***a*** or ***x*** > ***b*** and the value

$$\frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}\cdot\left(\frac{x-a}{b-a}\right)^{\alpha-1}\cdot\left(1-\frac{x-a}{b-a}\right)^{\beta-1}\cdot\frac{1}{b-a}$$

otherwise.

If ***Cumulative*** is TRUE, BETADIST returns 0 if ***x*** < ***a***, 1 if ***x*** > ***b***, and the value

$$\int_a^x \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}\cdot\left(\frac{t-a}{b-a}\right)^{\alpha-1}\cdot\left(1-\frac{t-a}{b-a}\right)^{\beta-1}\cdot\frac{1}{b-a}dt$$

otherwise.

Note: With substitution

$$z \stackrel{\text{def}}{=} \frac{t-a}{b-a}$$

the term can be written as

$$\int_0^{\frac{x-a}{b-a}} \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}\cdot z^{\alpha-1}\cdot(1-z)^{\beta-1}dz$$

See also BETAINV 6.18.8

6.18.8 BETAINV

Summary: returns the inverse of BETADIST(x;***α***;***β***;***A***;***B***;TRUE()).

Syntax: BETAINV(*Number* ***P*** ; *Number* ***α*** ; *Number* ***β*** [; *Number* ***A*** = 0 [; *Number* ***B*** = 1]])

Returns: *Number*

Constraints: 0 ≤ ***P*** ≤ 1, ***α*** > 0, ***β*** > 0, ***A*** < ***B***

Semantics: BETAINV returns the unique number x in the closed interval from ***A*** to ***B*** such that BETADIST(x;***α***;***β***;***A***;***B***) = ***P***.

See also BETADIST 6.18.7

6.18.9 BINOM.DIST.RANGE

Summary: Returns the probability of a trial result using binomial distribution.

Syntax: BINOM.DIST.RANGE(*Integer* ***N*** ; *Number* ***P*** ; *Integer* ***S*** [; *Integer* ***S2***])

Returns: *Number*

Constraints: 0 ≤ ***P*** ≤ 1, 0 ≤ ***S*** ≤ ***S2*** ≤ ***N***

Semantics: Let ***N*** be a total number of independent trials, and ***P*** be a probability of success for each trial. This function returns the probability that the number of successful trials shall be exactly ***S***. If the optional parameter ***S2*** is provided, this function returns the probability that the number of successful trials shall lie between ***S*** and ***S2*** inclusive.

This function is computed as follows:

If ***S2*** is not given, let ***S2*** = ***S***. Then the function returns the value of

$$\sum_{k=S}^{S2}\binom{N}{k}P^k(1-P)^{N-k}$$

See also BINOMDIST 6.18.10

6.18.10 BINOMDIST

Summary: Returns the binomial distribution.

Syntax: BINOMDIST(*Integer* ***S*** ; *Integer* ***N*** ; *Number* ***P*** ; *Logical* ***Cumulative***)

Returns: *Number*

Constraints: 0 ≤ ***P*** ≤ 1; 0 ≤ ***S*** ≤ ***N***

Semantics: If ***Cumulative*** is FALSE, this function returns the same result as BINOM.DIST.RANGE(***N***;***P***;***S***). If ***Cumulative*** is TRUE, it is equivalent to calling BINOM.DIST.RANGE(***N***;***P***;0;***S***).

See also BINOM.DIST.RANGE 6.18.9

6.18.11 LEGACY.CHIDIST

Summary: returns the right-tail probability for the χ2-distribution.

Syntax: LEGACY.CHIDIST(*Number* ***X*** ; *Number* ***DegreesOfFreedom***)

Returns: *Number*

Constraints: ***DegreesOfFreedom*** is a positive integer.

Semantics: In the following n is ***DegreesOfFreedom***. LEGACY.CHIDIST returns 1 for ***X*** ≤ 0 and the value

$$\int_{X}^{\infty} \frac{t^{\frac{n}{2}-1} e^{-\frac{t}{2}}}{2^{\frac{n}{2}} \Gamma(n/2)} dt$$

for ***X*** > 0.

See also CHISQDIST 6.18.12, LEGACY.CHITEST 6.18.15

6.18.12 CHISQDIST

Summary: returns the value of the probability density function or the cumulative distribution function for the χ2-distribution.

Syntax: CHISQDIST(*Number* ***X*** ; *Number* ***DegreesOfFreedom*** [; *Logical* ***Cumulative*** = TRUE])

Returns: *Number*

Constraints: ***DegreesOfFreedom*** is a positive integer.

Semantics: In the following n is ***DegreesOfFreedom***.

If ***Cumulative*** is FALSE, CHISQDIST returns 0 for ***X*** ≤ 0 and the value

$$\frac{X^{\frac{n}{2}-1} e^{-\frac{X}{2}}}{2^{\frac{n}{2}} \Gamma(n/2)}$$

for ***X*** > 0.

If ***Cumulative*** is TRUE, CHISQDIST returns 0 for ***X*** ≤ 0 and the value

$$\int_{0}^{X} \frac{t^{\frac{n}{2}-1} e^{-\frac{t}{2}}}{2^{\frac{n}{2}} \Gamma(n/2)} dt$$

for ***X*** > 0.

See also LEGACY.CHIDIST 6.18.11

6.18.13 LEGACY.CHIINV

Summary: returns the inverse of LEGACY.CHIDIST(x; ***DegreesOfFreedom***).

Syntax: LEGACY.CHIINV(*Number* ***P*** ; *Number* ***DegreesOfFreedom***)

Returns: *Number*

Constraints: ***DegreesOfFreedom*** is a positive integer and 0 < ***P*** ≤ 1.

Semantics: LEGACY.CHIINV returns the unique number x such that LEGACY.CHIDIST(x; ***DegreesOfFreedom***) = ***P***.

See also LEGACY.CHIDIST 6.18.11

6.18.14 CHISQINV

Summary: returns the inverse of CHISQDIST(x; ***DegreesOfFreedom***; TRUE()).

Syntax: CHISQINV(*Number* ***P*** ; *Number* ***DegreesOfFreedom***)

Returns: *Number*

Constraints: ***DegreesOfFreedom*** is a positive integer and 0 < ***P*** ≤ 1 .

Semantics: CHISQINV returns the unique number x ≥ 0 such that CHISQDIST(x; ***DegreesOfFreedom***;TRUE()) = ***P***.

See also CHISQDIST 6.18.12

6.18.15 LEGACY.CHITEST

Summary: Returns some Chi square goodness-for-fit test.

Syntax: LEGACY.CHITEST(*ForceArray Array* ***A*** ; *ForceArray Array* ***E***)

Returns: *Number*

Constraints:
ROWS(***A***) = ROWS(***E***)
COLUMNS(***A***) = COLUMNS(***E***)
COLUMNS(***A***) * ROWS(***A***) > 1

Semantics:

For an empty element or an element of type Text or Boolean in ***A*** the element at the corresponding position of ***E*** is ignored, and vice versa.

- ***A****:* actual observation data.
- ***E****:* expected values.

First a Chi square statistic is calculated:

$$\chi^2 = \sum_{i=1}^{r} \sum_{j=1}^{c} \frac{(A_{ij} - E_{ij})^2}{E_{ij}}$$

with

r = number of rows
c = number of columns
A_{ij} = element of actual data
E_{ij} = element of expected values

Then LEGACY.CHIDIST is called with the Chi-square value and a degree of freedom (df):

if $r > 1$ and $c > 1$
$df = (r-1) \cdot (c-1)$
else
$df = r \cdot c - 1$

$$LEGACY.CHITEST = LEGACY.CHIDIST\ (\chi^2; df)$$

See also COLUMNS 6.13.5, ROWS 6.13.30, LEGACY.CHIDIST 6.18.11

6.18.16 CONFIDENCE

Summary: Returns the confidence interval for a population mean.

Syntax: CONFIDENCE(*Number* ***Alpha*** ; *Number* ***Stddev*** ; *Number* ***Size***)

Returns: *Number*

Constraints: 0 < ***Alpha*** < 1; ***Stddev*** > 0, ***Size*** ≥ 1

Semantics: Calling this function is equivalent to calling
NORMINV(1 - ***Alpha*** / 2; 0; 1) * ***Stddev*** / SQRT (***Size***)

See also NORMINV 6.18.53, SQRT 6.16.58

6.18.17 CORREL

Summary: Calculates the correlation coefficient of values in ***N1*** and ***N2***.

Syntax: CORREL(*ForceArray Array* ***N1*** ; *ForceArray Array* ***N2***)

Returns: *Number*

Constraints: COLUMNS(***N1***) = COLUMNS(***N2***), ROWS(***N1***) = ROWS(***N2***), both sequences shall contain at least one number at corresponding positions each.

Semantics: Has the same value as COVAR(***N1***;***N2***) / STDEVP(***N1***) * (STDEVP(***N2***)). The CORREL function actually is identical to the PEARSON function.

For an empty element or an element of type Text or Boolean in ***N1*** the element at the corresponding position of ***N2*** is ignored, and vice versa.

See also COLUMNS 6.13.5, ROWS 6.13.30, COVAR 6.18.18, STDEVP 6.18.74, PEARSON 6.18.56

6.18.18 COVAR

Summary: Calculates covariance of two cell ranges.

Syntax: COVAR(*ForceArray Array* ***N1*** ; *ForceArray Array* ***N2***)

Returns: *Number*

Constraints: COLUMNS(***N1***) = COLUMNS(***N2***), ROWS(***N1***) = ROWS(***N2***), both sequences shall contain at least one number at corresponding positions each.

Semantics: returns

$$\frac{1}{N}\cdot\left(\sum_{a\in N1,\, b\in N2}(a-\overline{N1})\cdot(b-\overline{N2})\right)$$

where $\overline{N1}$ is the result of calling AVERAGE(***N1***), and $\overline{N2}$ is the result of calling AVERAGE(***N2***), and N is the number of terms in the sum.

For an empty element or an element of type Text or Boolean in ***N1*** the element at the corresponding position of ***N2*** is ignored, and vice versa.

See also COLUMNS 6.13.5, ROWS 6.13.30, AVERAGE 6.18.3

6.18.19 CRITBINOM

Summary: Returns the smallest value for which the cumulative binomial distribution is greater than or equal to a criterion value.

Syntax: CRITBINOM(*Number* ***Trials*** ; *Number* ***SP*** ; *Number* ***Alpha***)

Returns: *Number*

Constraints: ***Trials*** ≥ 0, 0 ≤ ***SP*** ≤ 1, 0 ≤ ***Alpha*** ≤ 1

Semantics:

- ***Trials***: the total number of trials.
- ***SP***: the probability of success for one trial.
- ***Alpha***: the threshold probability to be reached or exceeded.

6.18.20 DEVSQ

Summary: Calculates sum of squares of deviations.

Syntax: DEVSQ({ *NumberSequence* ***N*** }$^{+}$)

Returns: *Number*

Semantics: returns

$$\sum_{x\in N}(x-a)^2$$

where a is the result of calling AVERAGE(***N***).

6.18.21 EXPONDIST

Summary: returns the value of the probability density function or the cumulative distribution function for the exponential distribution.

Syntax: EXPONDIST(*Number* ***X*** ; *Number* ***λ*** [; *Logical* ***Cumulative*** = TRUE])

Returns: *Number*

Constraints: ***λ*** > 0

Semantics: If ***Cumulative*** is FALSE, EXPONDIST returns 0 if ***X*** < 0 and the value

$$\lambda e^{-\lambda X}$$

otherwise.

If ***Cumulative*** is TRUE, EXPONDIST returns 0 if ***X*** < 0 and the value

$$\int_0^X \lambda e^{-\lambda t} dt = 1 - e^{-\lambda X}$$

otherwise.

6.18.22 FDIST

Summary: returns the value of the probability density function or the cumulative distribution function for the F-distribution.

Syntax: FDIST(*Number* ***X*** ; *Number* $\boldsymbol{R}_1$; *Number* $\boldsymbol{R}_2$ [; *Logical* ***Cumulative*** = TRUE])

Returns: *Number*

Constraints: $\boldsymbol{R}_1$ and $\boldsymbol{R}_2$ are positive integers

Semantics:

- $\boldsymbol{R}_1$: the degrees of freedom in the numerator of the F distribution.
- $\boldsymbol{R}_2$: the degrees of freedom in the denominator of the F distribution.

If ***Cumulative*** is FALSE, FDIST returns 0 if ***X*** < 0, an Error if the numerator degrees of freedom $\boldsymbol{R}_1$ = 1 and ***X*** = 0, and the value

$$\frac{\Gamma\left(\frac{R_1+R_2}{2}\right)\left(\frac{R_1}{R_2}\right)^{\frac{R_1}{2}}}{\Gamma\left(\frac{R_1}{2}\right)\Gamma\left(\frac{R_2}{2}\right)} \cdot \frac{x^{\frac{R_1}{2}-1}}{\left(1+\frac{R_1}{R_2}x\right)^{\frac{R_1+R_2}{2}}}$$

otherwise.

If the numerator degrees of freedom $\boldsymbol{R}_1$ = 1, then the density function has a pole at ***X*** = 0, the subterm $x^{\frac{R_1}{2}-1} = 0^{-0.5}$ is not defined.

If ***Cumulative*** is TRUE, FDIST returns 0 if ***X*** < 0 and the value

$$\frac{\Gamma\left(\frac{R_1+R_2}{2}\right)\left(\frac{R_1}{R_2}\right)^{\frac{R_1}{2}}}{\Gamma\left(\frac{R_1}{2}\right)\Gamma\left(\frac{R_2}{2}\right)} \cdot \int_0^X \frac{t^{\frac{R_1}{2}-1}}{\left(1+\frac{R_1}{R_2}t\right)^{\frac{R_1+R_2}{2}}} dt$$

otherwise.

See also LEGACY.FDIST 6.18.23

6.18.23 LEGACY.FDIST

Summary: returns the area of the right tail of the probability density function for the F-distribution.

Syntax: LEGACY.FDIST(*Number* ***X*** ; *Number* $\boldsymbol{R}_1$; *Number* $\boldsymbol{R}_2$)

Returns: *Number*

Constraints: $\boldsymbol{R}_1$ and $\boldsymbol{R}_2$ are positive integers

Semantics:

LEGACY.FDIST returns Error if x < 0 and the value

$$\frac{\Gamma\left(\frac{r_1+r_2}{2}\right)\left(\frac{r_1}{r_2}\right)^{\frac{r_1}{2}}}{\Gamma\left(\frac{r_1}{2}\right)\Gamma\left(\frac{r_2}{2}\right)} \cdot \int_x^{\infty} \frac{t^{\frac{r_1}{2}-1}}{\left(1+\frac{r_1}{r_2}t\right)^{\frac{r_1+r_2}{2}}} dt$$

otherwise.

Note that the latter is (1-FDIST(x; r_1; r_2;TRUE())).

See also FDIST 6.18.22

6.18.24 FINV

Summary: returns the inverse of FDIST(x;***R***$_1$;***R***$_2$;TRUE()).

Syntax: FINV(*Number* ***P*** ; *Number* ***R***$_1$; *Number* ***R***$_2$)

Returns: *Number*

Constraints: 0 ≤ ***P*** < 1, ***R***$_1$ and ***R***$_2$ are positive integers

Semantics: FINV returns the unique non-negative number x such that FDIST(x;***R***$_1$;***R***$_2$) = ***P***.

See also FDIST 6.18.22, LEGACY.FDIST 6.18.23, LEGACY.FINV 6.18.25

6.18.25 LEGACY.FINV

Summary: returns the inverse of LEGACY.FDIST(x;***R***$_1$;***R***$_2$).

Syntax: LEGACY.FINV(*Number* ***P*** ; *Number* ***R***$_1$; *Number* ***R***$_2$)

Returns: *Number*

Constraints: 0 < ***P*** ≤ 1, ***R***$_1$ and ***R***$_2$ are positive integers

Semantics: LEGACY.FINV returns the unique non-negative number x such that LEGACY.FDIST(x;***R***$_1$;***R***$_2$) = ***P***.

See also FDIST 6.18.22, LEGACY.FDIST 6.18.23, FINV 6.18.24

6.18.26 FISHER

Summary: returns the Fisher transformation.

Syntax: FISHER(*Number* ***R***)

Returns: *Number*

Constraints: -1 < ***R*** < 1

Semantics: Returns the Fisher transformation with a sample correlation ***R***. This function computes

$$\frac{1}{2}\ln\left(\frac{1+R}{1-R}\right)$$

where *ln* is the natural logarithm function.

FISHER is a synonym for ATANH.

See also ATANH 6.16.11

6.18.27 FISHERINV

Summary: returns the inverse Fisher transformation.

Syntax: FISHERINV(*Number* ***R***)

Returns: *Number*

Constraints: none

Semantics: Returns the inverse Fisher transformation. This function computes

$$\frac{e^{2R}-1}{e^{2R}+1}$$

FISHERINV is a synonym for TANH.

See also TANH 6.16.70

6.18.28 FORECAST

Summary: Extrapolates future values based on existing x and y values.

Syntax: FORECAST(*Number* ***Value*** ; *ForceArray Array* ***Data_Y*** ; *ForceArray Array* ***Data_X***)

Returns: *Number*

Constraints: COLUMNS(***Data_Y***) = COLUMNS(***Data_X***), ROWS(***Data_Y***) = ROWS(***Data_X***)

Semantics:

- ***Value***: the x-value, for which the y-value on the linear regression is to be returned.
- ***Data_Y***: the array or range of known y-values.
- ***Data_X***: the array or range of known x-values.

For an empty element or an element of type Text or Boolean in ***Data_Y*** the element at the corresponding position of ***Data_X*** is ignored, and vice versa.

See also COLUMNS 6.13.5, ROWS 6.13.30

6.18.29 FREQUENCY

Summary: Categorizes values into intervals and counts the number of values in each interval.

Syntax: FREQUENCY(*NumberSequenceList* ***Data*** ; *NumberSequenceList* ***Bins***)

Returns: *Array*

Constraints: Values in ***Bins*** shall be sorted in ascending order and ***Bins*** shall be a column vector. Evaluators may accept unsorted values in bins.

Semantics: Counts the number of values for each interval given by the border values in ***Bins*** .
The values in ***Bins*** determine the upper boundaries of the intervals. The intervals include the upper boundarie. The returned array is a column vector and has one more element than ***Bins*** ; the last element represents the number of all elements greater than the last value in ***Bins*** . If ***Bins*** is empty, all values in ***Data*** are counted. The values in the result array are ordered matching the original order of ***Bins*** . If the values in ***Bins*** are not sorted in ascending order, they are sorted internally to form category intervals and the counts of ***Data*** values are "unsorted" to the original order of ***Bins***. If ***Data*** is empty, the value of all elements in the returned array is 0.

Data: The data, that should be categorized and counted according to the given intervals.
Bins: The upper boundaries determining the intervals the values in *data* should be grouped by.

6.18.30 FTEST

Summary: Calculates the probability of an F-test.

Syntax: FTEST(*ForceArray NumberSequence* ***Data_1*** ; *ForceArray NumberSequence* ***Data_2***)

Returns: *Number*

Constraints: ***Data_1*** and ***Data_2*** shall both contain at least 2 numbers and shall both have nonzero variances

Semantics:

Calculates a two-sided P-value to decide, whether the difference in the variances of the two data sets are significant enough to reject the hypothesise, that both sets come from normally distributed populations with the same variances.

Suppose the data set Data_1 is a sample of size n_1 from a normal distribution and has the sample variance s_1^2, and the data set Data_2 is a sample of size n_2 from a normal distribution and has the sample variance s_2^2.

Get the value P_{tail} as the area of the right tail beyond s_1^2/s_2^2 of the F-distribution with numerator degrees of freedom n_1-1 and denominator degrees of freedom n_2-1.

FTEST returns twice the minimum of the values P_{tail} and $1-P_{tail}$ **.See also** TTEST 6.18.81

6.18.31 GAMMADIST

Summary: returns the value of the probability density function or the cumulative distribution function for the Gamma distribution.

Syntax: GAMMADIST(*Number* ***X*** ; *Number* ***α*** ; *Number* ***β*** [; *Logical* ***Cumulative*** = TRUE])

Returns: *Number*

Constraints: ***α*** > 0, ***β*** > 0

Semantics: If ***Cumulative*** is FALSE, GAMMADIST returns 0 if ***X*** < 0 and the value

$$\frac{1}{\beta^{\alpha}\cdot\Gamma(\alpha)}\cdot X^{\alpha-1}\cdot e^{-\frac{X}{\beta}}$$

otherwise.

If ***Cumulative*** is TRUE(), GAMMADIST returns 0 if ***X*** < 0 and the value

$$\int_0^X \frac{1}{\beta^{\alpha}\cdot\Gamma(\alpha)}\cdot t^{\alpha-1}\cdot e^{-\frac{t}{\beta}}dt$$

otherwise.

See also GAMMA 6.16.34, GAMMAINV 6.18.32

6.18.32 GAMMAINV

Summary: returns the inverse of GAMMADIST(X;***α***;***β***;TRUE).

Syntax: GAMMAINV(*Number* ***P*** ; *Number* ***α*** ; *Number* ***β***)

Returns: *Number*

Constraints: 0 ≤ ***P*** < 1, ***α*** > 0, ***β*** > 0

Semantics: GAMMAINV returns the unique number X ≥ 0 such that GAMMAINV(X;***α***;***β***) = ***P***.

See also GAMMADIST 6.18.31

6.18.33 GAUSS

Summary: Returns 0.5 less than the standard normal cumulative distribution

Syntax: GAUSS(*Number* ***X***)

Returns: *Number*

Semantics: Returns NORMDIST(***X***;0;1;TRUE())-0.5

See also NORMDIST 6.18.52

6.18.34 GEOMEAN

Summary: returns the geometric mean of a sequence

Syntax: GEOMEAN({ *NumberSequenceList* ***N*** }$^{+}$)

Returns: *Number*

Semantics: Returns the geometric mean of a given sequence. That means

$$\left(\prod_{a \in N} a\right)^{1/n}$$

where *n* is a result of calling COUNT(***N***).

See also COUNT 6.13.6

6.18.35 GROWTH

Summary: Returns predicted values based on an exponential regression.

Syntax: GROWTH(*Array* ***KnownY*** [; [*Array* ***KnownX***] [; [*Array* ***NewX***] [; *Logical* ***Const*** = TRUE]]])

Returns: *Array*

Constraints: (COLUMNS(***KnownY***) = COLUMNS(***KnownX***) and ROWS(***KnownY***) = ROWS(***KnownX***)) or (COLUMNS(***KnownY***) = 1 and ROWS(***KnownY***) = ROWS(***KnownX***) and COLUMNS(***KnownX***) = COLUMNS(***NewX***)) or (COLUMNS(***KnownY***) = COLUMNS(***KnownX***) and ROWS(***KnownY***) = 1 and ROWS(***KnownX***) = ROWS(***NewX***))

Semantics:

- ***KnownY***: The set of known y-values to be used to determine the regression equation
- ***KnownX***: The set of known x-values to be used to determine the regression equation. If omitted or an empty parameter, it is set to the sequence 1,2,3,…,*k* , where

  k = ROWS(***KnownY***) · COLUMNS(***KnownY***)
- ***NewX***: The set of x-values for which predicted y-values are to be calculated. If omitted or an empty parameter, it is set to ***KnownX***.

Const: If set to FALSE, the model constant *a* is equal to 0.

LOGEST(***KnownY*** ; ***KnownX***; ***Const***; FALSE) either returns an error or an array with 1 row and *n*+1 columns. If it returns an error then so does GROWTH. If it returns an array, we call the entries in that array $b_n, b_{n-1}, \ldots, b_1, a$.

Let z_{ij} denote the entry in the *i*th row and *j*th column of ***NewX***.

If COLUMNS(***KnownY***) = COLUMNS(***KnownX***) and ROWS(***KnownY***) = ROWS(***KnownX***), then GROWTH returns an array with ROWS(***NewX***) rows and COLUMNS(***NewX***) column, such that the entry in its *i*th row and *j*th column is $a \times b_1^{z_{ij}}$.

Otherwise, if COLUMNS(***KnownY***) = 1 and ROWS(***KnownY***) = ROWS(*KnownX*) and COLUMNS(***KnownX***) = COLUMNS(***NewX***), then GROWTH returns an array with ROWS(***NewX***) rows and 1 column, such that the entry in the *i*th row is $a \times \prod_{j=1}^{n} b_j^{z_{ij}}$.

Otherwise, if COLUMNS(***KnownY***) = COLUMNS(***KnownX***) and ROWS(***KnownY***) = 1 and ROWS(***KnownX***) = ROWS(***NewX***), then GROWTH returns an array with 1 row and COLUMNS(***NewX***) columns, such that the entry in the *j*th column is $a \times \prod_{i=1}^{n} b_i^{z_{ij}}$.

See also COLUMNS 6.13.5, ROWS 6.13.30, LOGEST 6.18.42, TREND 6.18.79

6.18.36 HARMEAN

Summary: returns the harmonic mean of a sequence

Syntax: HARMEAN({ *NumberSequenceList N* }$^{+}$)

Returns: *Number*

Semantics: Returns the harmonic mean of a given sequence. That means

$$\frac{n}{\sum_{i=1}^{n} \frac{1}{a_i}}$$

where $a_1, a_2, \ldots, a_n$ are the numbers of the sequence *N* and *n* is a result of calling COUNT(*N*).

See also COUNT 6.13.6

6.18.37 HYPGEOMDIST

Summary: The hypergeometric distribution returns the number of successes in a sequence of *n* draws from a finite population without replacement.

Syntax: HYPGEOMDIST(*Integer* ***X*** ; *Integer* ***T*** ; *Integer* ***M*** ; *Integer* ***N*** [; *Logical* ***Cumulative*** = FALSE])

Returns: *Number*

Constraints: 0 ≤ ***X*** ≤ ***T*** ≤ ***N***, 0 ≤ ***M*** ≤ ***N***

Semantics:

- ***X***: the number of successes in ***T*** trials
- ***T***: the number of trials
- ***M***: the number of successes in the population
- ***N***: the total population
- ***Cumulative*** : a Logical parameter.

If ***Cumulative*** is FALSE, return the probability of exactly ***X*** successes. If ***Cumulative*** is TRUE, return the probability of at most ***X*** successes. If omitted, FALSE is assumed.

If ***Cumulative*** is FALSE, HYPGEOMDIST returns

$$\frac{\binom{M}{X}\binom{N-M}{T-X}}{\binom{N}{T}}$$

If ***Cumulative*** is TRUE, HYPGEOMDIST returns

$$\sum_{i=0}^{X} \frac{\binom{M}{i}\binom{N-M}{T-i}}{\binom{N}{T}}$$

Note:

$$\binom{x}{y} = 0 \ for \ y > x$$

6.18.38 INTERCEPT

Summary: Returns the y-intercept of the linear regression line for the given data.

Syntax: INTERCEPT(*ForceArray Array* ***Data_Y*** ; *ForceArray Array* ***Data_X***)

Returns: *Number*

Constraints: COLUMNS(***Data_X***) = COLUMNS(***Data_Y***), ROWS(***Data_X***) = ROWS(***Data_Y***)

Semantics:

INTERCEPT returns the intercept (a) calculated as described in 6.18.41 for the function call LINEST(***Data_Y***,***Data_X***,FALSE()).

For an empty element or an element of type Text or Boolean in ***Data_Y*** the element at the corresponding position of ***Data_X*** is ignored, and vice versa.

See also COLUMNS 6.13.5, ROWS 6.13.30

6.18.39 KURT

Summary: Return the kurtosis (“peakedness”) of a data set.

Syntax: KURT({ *NumberSequenceList* ***X*** }$^{+}$)

Returns: *Number*

Constraints: COUNT(***X***) ≥ 4, STDEV(***X***) ≠ 0

Semantics:

Kurtosis characterizes the relative peakedness or flatness of a distribution compared with the normal distribution. Positive kurtosis indicates a relatively peaked distribution (compared to the normal distribution), while negative kurtosis indicates a relatively flat distribution.

$$kurtosis = \left(\frac{n(n+1)}{(n-1)(n-2)(n-3)} \sum_{i=1}^{n} \left(\frac{x_i - \overline{x}}{s} \right)^4 \right) - \frac{3(n-1)^2}{(n-2)(n-3)}$$

where *s* is the sample standard deviation, and *n* is the number of numbers.

See also STDEV 6.18.72

6.18.40 LARGE

Summary: Finds the nth largest value in a list.

Syntax: LARGE(*NumberSequenceList* ***List*** ; *Number|Array* ***N***)

Returns: *Number* or *Array*

Constraints: ROUNDUP(***N***;0) = ***N***. If the resulting ***N*** is <1 or larger than the size of ***List***, Error is returned

Semantics: If ***N*** is an array of numbers, an array of largest values is returned.

See also SMALL 6.18.70, ROUNDUP 6.17.7

6.18.41 LINEST

Summary: Returns the parameters of the (simple or multiple) linear regression equation for the given data and, optionally, statistics on this regression.

Syntax: LINEST(*ForceArray Array* ***KnownY*** [; [*ForceArray Array* ***KnownX***] [; *Logical* ***Const*** = TRUE [; *Logical* ***Stats*** = FALSE]]])

Returns: *Array*

Constraints: (COLUMNS(***KnownY***) = COLUMNS(***KnownX***) and ROWS(***KnownY***) = ROWS(***KnownX***)) or (COLUMNS(***KnownY***) = 1 and ROWS(***KnownY***) = ROWS(***KnownX***)) or (COLUMNS(***KnownY***) = COLUMNS(***KnownX***) and ROWS(***KnownY***) = 1)

Semantics:

- ***KnownY***: The set of y-values for the equation
- ***KnownX***: The set of x-values for the equation. If omitted or an empty parameter, it is set to the sequence 1,2,3,…,*k* , where *k* = ROWS(***KnownY***) · COLUMNS(***KnownY***).

Const: If set to FALSE, the model constant *a* is equal to 0.

- ***Stats***: If FALSE, only the regression coefficient is to be calculated. If set to TRUE, the result will include other statistical data.

If any of the entries in ***KnownY*** and ***KnownX*** do not convert to Number, LINEST returns an error.

The result created by LINEST if STATS is TRUE is given in Table 29 - LINEST. If STATS is FALSE it is just the first row of Table 29 - LINEST. The empty cells in this table are returned as empty or as containing an error.

Table 29 - LINEST

b_n	b_{n-1}	…	b_1	a
s_{b_n}	$s_{b_{n-1}}$	…	s_{b_1}	s_a
R^2	s_e			
F	df			
SS_{reg}	SS_{resid}			

If COLUMNS(***KnownY***) = COLUMNS(***KnownX***) and ROWS(***KnownY***) = ROWS(***KnownX***) then *n* = 1, *k* = ROWS(***KnownY***) · COLUMNS(***KnownY***), the entries of ***KnownX*** in column major order are denoted with $x_{1n}, x_{2n}, \ldots, x_{kn}$ and the entries of ***KnownY*** in column major order are denoted with $y_1, y_2, \ldots, y_k$.

Otherwise but if COLUMNS(***KnownY***) = 1, then *n* = COLUMNS(***KnownX***), *k* = ROWS(***KnownY***), the entry in the *j*th column and *i*th row of ***KnownX*** is denoted x_{ij} and the entry in the *i*th row of ***KnownY*** is denoted y_i .

Otherwise but if ROWS(***KnownY***) = 1, then *n* = ROWS(***KnownX***), *k* = COLUMNS(***KnownY***), the entry in the *j*th column and *i*th row of ***KnownX*** is denoted x_{ji} and the entry in the *j*th column of ***KnownY*** is denoted y_j .

If ***Const*** is TRUE and *k* ≤ *n* + 1, LINEST returns an error. Similarly, if ***Const*** is FALSE and *k* ≤ *n,* LINEST returns an error.

We denote

$$\overline{x_i} = \frac{1}{k} \cdot \sum_{j=1}^{k} x_{ij}$$

and

$$\overline{y}=\frac{1}{k}\cdot\sum_{j=1}^{k}y_j$$,

and define the following matrices:

$$Y=\begin{pmatrix}y_1\\ \vdots\\ y_k\end{pmatrix}$$ and $$X=\begin{pmatrix}1 & x_{11} & \dots & x_{1n}\\ 1 & x_{12} & \dots & x_{2n}\\ \vdots & \vdots & \ddots & \vdots\\ 1 & x_{k1} & \dots & x_{kn}\end{pmatrix}$$ for ***Const*** being TRUE, and $$X=\begin{pmatrix}x_{11} & \dots & x_{1n}\\ x_{12} & \dots & x_{2n}\\ \vdots & \ddots & \vdots\\ x_{k1} & \dots & x_{kn}\end{pmatrix}$$ for ***Const*** being FALSE().

Let X^T denote the transpose of *X*, see TRANSPOSE 6.5.6. Then the matrix product $X^T\cdot X$ is a square matrix. If $X^T\cdot X$ is not invertible, then LINEST shall either return an error or calculate a result as described below.

If $X^T\cdot X$ is invertible, then $(X^T\cdot X)^{-1}\cdot X^T\cdot Y$ is a matrix *B* with a single column. If ***Const*** is TRUE, the entries of *B* are denoted $B=\begin{pmatrix}a\\ b_1\\ \vdots\\ b_n\end{pmatrix}$; if ***Const*** is FALSE, the entries of *B* are denoted $B=\begin{pmatrix}b_1\\ \vdots\\ b_n\end{pmatrix}$ and *a* = 0.

These $a,b_1,\dots,b_n$ are the values returned by LINEST in the first row of its result array in the order given in Table 29 - LINEST.

The statistics in the 2nd to 5th rows of Table 29 - LINEST are as follows:

If ***Const*** is TRUE:

$$df=k-n-1$$.

$$SS_{resid}=\sum_{i=1}^{k}\left(\left(a+\sum_{j=1}^{n}b_j x_{ij}\right)-y_i\right)^2$$, $$SS_{reg}=\sum_{i=1}^{k}\left(\left(a+\sum_{j=1}^{n}b_j x_{ij}\right)-\overline{y}\right)^2$$, $$s_a=s_e\sqrt{d_1}$$

and

$$s_{b_i}=s_e\sqrt{d_{i+1}}$$

where d_i is the element in the *i*th row and *i*th column of

$$(X^T\cdot X)^{-1}$$, $$s_e=\sqrt{\frac{SS_{resid}}{df}}$$, $$R^2=\frac{SS_{reg}}{\sum_{i=1}^{k}(y_i-\overline{y})^2}$$ and $$F=\frac{SS_{reg}/n}{SS_{resid}/df}$$.

If ***Const*** is FALSE:

$$df=k-n$$, $$SS_{resid}=\sum_{i=1}^{k}\left(\left(\sum_{j=1}^{n}b_j x_{ij}\right)-y_i\right)^2$$, $$SS_{reg}=\sum_{i=1}^{k}\left(\sum_{j=1}^{n}b_j x_{ij}\right)^2$$, $$s_{b_i}=s_e\sqrt{d_i}$$

where d_i is the element in the *i*th row and *i*th column of

$$(X^T \cdot X)^{-1}, \quad s_e = \sqrt{\frac{SS_{resid}}{df}}, \quad R^2 = \frac{SS_{reg}}{\sum_{i=1}^{k} y_i^2} \text{ and } F = \frac{SS_{reg}/n}{SS_{resid}/df}.$$

In this case s_a is undefined and is returned as either 0, blank or an error.

If $X^T \cdot X$ is not invertible, then the columns of X are linearly dependent. In this case an evaluator shall return an error or select any maximal linearly independent subset of these columns that if ***Const*** is TRUE includes the first column and perform the above calculations with that subset. In the latter case the coefficients b_i of omitted columns are returned as 0.

See also COLUMNS 6.13.5, ROWS 6.13.30

6.18.42 LOGEST

Summary: Returns the parameters of an exponential regression equation for the given data obtained by linearizing this intrinsically linear response function and returns, optionally, statistics on this regression.

Syntax: LOGEST(*ForceArray Array* ***KnownY*** [; [*ForceArray Array* ***KnownX***] [; *Logical* ***Const*** = TRUE [; *Logical* ***Stats*** = FALSE]]])

Returns: *Array*

Constraints: (COLUMNS(***KnownY***) = COLUMNS(***KnownX***) and ROWS(***KnownY***) = ROWS(***KnownX***)) or (COLUMNS(***KnownY***) = 1 and ROWS(***KnownY***) = ROWS(***KnownX***)) or (COLUMNS(***KnownY***) = COLUMNS(***KnownX***) and ROWS(***KnownY***) = 1)

Semantics:

- ***KnownY***: The set of y-values for the equation
- ***KnownX***: The set of x-values for the equation. If omitted or an empty parameter, it is set to the sequence 1,2,3,…,*k*, where *k* = ROWS(***KnownY***) · COLUMNS(***KnownY***).

Const: If set to FALSE, the model constant a is equal to 0.

- ***Stats***: If FALSE, only the regression coefficient is to be calculated. If set to TRUE, the result will include other statistical data.

If any of the entries in ***KnownY*** and ***KnownX*** do not convert to Number or if any of the entries in ***KnownY*** is negative, LOGEST returns an error.

The result created by LOGEST if STATS is TRUE is given in Table 30 - LOGEST. If STATS is FALSE it is just the first row of Table 30 - LOGEST. The empty cells in this table are returned as empty or as containing an error.

Table 30 - LOGEST

e^{b_n}	$e^{b_{n-1}}$	…	e^{b_1}	e^{a}
s_{b_n}	$s_{b_{n-1}}$	…	s_{b_1}	s_a
R^2	s_e			
F	*df*			
SS_{reg}	SS_{resid}			

If COLUMNS(***KnownY***) = COLUMNS(***KnownX***) and ROWS(***KnownY***) = ROWS(***KnownX***) then *n* = 1, *k* = ROWS(***KnownY***) · COLUMNS(***KnownY***), the entries of ***KnownX*** in column major order are denoted with $x_{1n}, x_{2n}, \ldots, x_{kn}$ and the entries of ***KnownY*** in column major order are denoted with $y_1, y_2, \ldots, y_k$.

Otherwise but if COLUMNS(***KnownY***) = 1, then *n* = COLUMNS(***KnownX***), *k* = ROWS(***KnownY***), the entry in the *j*th column and *i*th row of ***KnownX*** is denoted x_{ij} and the entry in the *i*th row of ***KnownY*** is denoted y_i.

Otherwise but if ROWS(***KnownY***) = 1, then *n* = ROWS(***KnownX***), *k* = COLUMNS(***KnownY***), the entry in the *j*th column and *i*th row of ***KnownX*** is denoted x_{ji} and the entry in the *j*th column of ***KnownY*** is denoted y_j.

If ***Const*** is TRUE and $k \leq n + 1$, LOGEST returns an error. Similarly, if ***Const*** is FALSE and $k \leq n$, LOGEST returns an error.

We denote

$$\overline{x_i} = \frac{1}{k} \cdot \sum_{j=1}^{k} x_{ij}$$

and

$$\overline{\ln(y)} = \frac{1}{k} \cdot \sum_{j=1}^{k} \ln(y_j)$$

,

and define the following matrices:

$$Y = \begin{pmatrix} \ln(y_1) \\ \vdots \\ \ln(y_k) \end{pmatrix}$$ and $$X = \begin{pmatrix} 1 & x_{11} & \dots & x_{1n} \\ 1 & x_{12} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{k1} & \dots & x_{kn} \end{pmatrix}$$ for ***Const*** being TRUE, and

$$X = \begin{pmatrix} x_{11} & \dots & x_{1n} \\ x_{12} & \dots & x_{2n} \\ \vdots & \ddots & \vdots \\ x_{k1} & \dots & x_{kn} \end{pmatrix}$$ for ***Const*** being FALSE().

Let X^T denote the transpose of *X*, see TRANSPOSE 6.5.6. Then the matrix product $X^T \cdot X$ is a square matrix. If $X^T \cdot X$ is not invertible, then LOGEST shall either return an error or calculate a result as described below.

If $X^T \cdot X$ is invertible, then $(X^T \cdot X)^{-1} \cdot X^T \cdot Y$ is a matrix *B* with a single column. If ***Const*** is TRUE, the entries of *B* are denoted $B = \begin{pmatrix} a \\ b_1 \\ \vdots \\ b_n \end{pmatrix}$; if ***Const*** is FALSE, the entries of *B* are denoted

$$B = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$ and $a = 0$.

Then $e^a, e^{b_1}, \dots, e^{b_n}$ are the values returned by LOGEST in the first row of its result array in the order given in Table 1 - Operators.

The statistics in the 2nd to 5th rows of Table 1 - Operators are as follows:

If ***Const*** is TRUE():

$$df = k - n - 1$$.

$$SS_{resid}=\sum_{j=1}^{k}\left(\left(a+\sum_{i=1}^{n}b_j x_{ij}\right)-\ln\left(y_i\right)\right)^2 \quad , \quad SS_{reg}=\sum_{j=1}^{k}\left(\left(a+\sum_{i=1}^{n}b_j x_{ij}\right)-\overline{\ln\left(y\right)}\right)^2 \quad , \quad s_a=s_e\sqrt{d_1}$$

and

$$s_{b_i}=s_e\sqrt{d_{i+1}}$$

where d_i is the element in the *i*th row and *i*th column of

$$\left(X^T\cdot X\right)^{-1} \quad , \quad s_e=\sqrt{\frac{SS_{resid}}{df}} \quad , \quad R^2=\frac{SS_{reg}}{\sum_{j=1}^{k}\left(\ln\left(y_k\right)-\overline{\ln\left(y\right)}\right)^2} \quad \text{and} \quad F=\frac{SS_{reg}/n}{SS_{resid}/df} \; .$$

If ***Const*** is FALSE:

$$df=k-n \quad , \quad SS_{resid}=\sum_{j=1}^{k}\left(\left(\sum_{i=1}^{n}b_j x_{ij}\right)-\ln\left(y_i\right)\right)^2 \quad , \quad SS_{reg}=\sum_{j=1}^{k}\left(\sum_{i=1}^{n}b_j x_{ij}\right)^2 \quad , \quad s_{b_i}=s_e\sqrt{d_i}$$

where d_i is the element in the *i*th row and *i*th column of

$$\left(X^T\cdot X\right)^{-1} \quad , \quad s_e=\sqrt{\frac{SS_{resid}}{df}} \quad , \quad R^2=\frac{SS_{reg}}{\sum_{j=1}^{k}\ln\left(y_k\right)^2} \quad \text{and} \quad F=\frac{SS_{reg}/n}{SS_{resid}/df} \; .$$

In this case s_a is undefined and is returned as either 0, blank or an error.

If $X^T\cdot X$ is not invertible, then the columns of X are linearly dependent. In this case an evaluator shall return an error or select any maximal linearly independent subset of these columns that if ***Const*** is TRUE includes the first column and perform the above calculations with that subset. In the latter case the coefficients e^{b_i} of omitted columns are returned as 1.

See also COLUMNS 6.13.5, ROWS 6.13.30

6.18.43 LOGINV

Summary: returns the inverse of LOGNORMDIST(x;***Mean***;***StandardDeviation***,TRUE()).

Syntax: LOGINV(*Number* ***P*** [; *Number* ***Mean*** = 0 [; *Number* ***StandardDeviation*** = 1]])

Returns: *Number*

Constraints: ***StandardDeviation*** > 0 and 0 < ***P*** < 1.

Semantics: LOGINV returns the unique number x such that LOGNORMDIST(x;***Mean***;***StandardDeviation***;TRUE()) = ***P***.

See also LOGNORMDIST 6.18.44

6.18.44 LOGNORMDIST

Summary: returns the value of the probability density function or the cumulative distribution function for the lognormal distribution with the mean and standard deviation given.

Syntax: LOGNORMDIST(*Number* ***X*** [; *Number* ***μ*** = 0 [; *Number* ***σ*** = 1 [; *Logical* ***Cumulative*** = TRUE]]])

Returns: *Number*

Constraints: ***σ*** > 0; ***X*** > 0 if ***Cumulative*** is FALSE

Semantics: If ***Cumulative*** is FALSE, LOGNORMDIST returns the value

$$\frac{e^{-\frac{1}{2}\cdot\left(\frac{\ln(X)-\mu}{\sigma}\right)^2}}{X\sqrt{2\pi}\,\sigma}$$

If ***Cumulative*** is TRUE, LOGNORMDIST returns the value

$$\int_0^X \frac{e^{-\frac{1}{2}\cdot\left(\frac{\ln(t)-\mu}{\sigma}\right)^2}}{t\sqrt{2\pi}\,\sigma}dt$$

if ***X*** > 0 and 0 otherwise.

6.18.45 MAX

Summary: Return the maximum from a set of numbers.

Syntax: MAX({ *NumberSequenceList* ***N*** }$^+$)

Returns: *Number*

Constraints: None.

Semantics: Returns the value of the maximum number in the list passed in. Non-numbers are ignored. Note that if Logical types are a distinct type, they are not included.

See also MAXA 6.18.46, MIN 6.18.48

6.18.46 MAXA

Summary: Return the maximum from a set of values, including values of type Text and Logical.

Syntax: MAXA({ *Any* ***N*** }$^+$)

Returns: *Number*

Constraints: None.

Semantics: A variation of the MAX function that includes values of type Text and Logical. Text values are treated as number 0. Logical TRUE is treated as 1, and FALSE is treated as 0. Empty cells are not included. Any ***N*** may be of type ReferenceList.

See also MAX 6.18.45, MIN 6.18.48, MINA 6.18.49

6.18.47 MEDIAN

Summary: Returns the median (middle) value in the list.

Syntax: MEDIAN({ *NumberSequenceList* ***X*** }$^+$)

Returns: *Number*

Semantics:

MEDIAN logically ranks the numbers (lowest to highest). If given an odd number of values, MEDIAN returns the middle value. If given an even number of values, MEDIAN returns the arithmetic average of the two middle values.

$$n = is\,the\,count\,of\ the\,ranked\ numbersequence$$

$$\tilde{x} = x_{\left(\frac{(n+1)}{2}\right)}$$
$$for\,n = odd$$

$$\tilde{x} = \frac{1}{2}\left(x_{\left(\frac{n}{2}\right)} + x_{\left(\frac{n}{2}+1\right)}\right)$$
$$for\,n = even$$

6.18.48 MIN

Summary: Return the minimum from a set of numbers.

Syntax: MIN({ *NumberSequenceList* ***N*** }$^{+}$)

Returns: *Number*

Constraints: None.

Semantics: Returns the value of the minimum number in the list passed in. Returns zero if no numbers are provided in the list. What happens when MIN is provided 0 parameters is implementation-defined, but MIN() with no parameters should return 0.

See also MAX 6.18.45, MINA 6.18.49

6.18.49 MINA

Summary: Return the minimum from a set of values, including values of type Text and Logical.

Syntax: MINA({ *Any* ***N*** }$^{+}$)

Returns: *Number*

Constraints: None.

Semantics: A variation of the MIN function that includes values of type Text and Logical. Text values are treated as number 0. Logical TRUE is treated as 1, and FALSE is treated as 0. Empty cells are not included. What happens when MINA is provided 0 parameters is implementation-defined. Any ***N*** may be of type ReferenceList.

See also MIN 6.18.48, MAXA 6.18.46

6.18.50 MODE

Summary: Returns the most common value in a data set.

Syntax: MODE({ *ForceArray NumberSequence* ***N*** }$^{+}$)

Semantics: Returns the most common value in a data set. If there are more than one values with the same largest frequency, returns the smallest value. If the number sequence does no contain at least two equal values, the MODE is not defined, as no most common value can be found, and an Error is returned.

6.18.51 NEGBINOMDIST

Summary: Returns the negative binomial distribution.

Syntax: NEGBINOMDIST(*Integer* ***X*** ; *Integer* ***R*** ; *Number* ***Prob***)

- ***X***: The number of failures.

- ***R***: The threshold number of successes.
- ***Prob****:* The probability of a success.

Returns: *Number*

Constraints:

- If (***X*** + ***R*** - 1) ≤ 0, NEGBINOMDIST returns an Error.
- If ***Prob*** < 0 or ***Prob*** > 1, NEGBINOMDIST returns an Error.

Semantics:

NEGBINOMDIST returns the probability that there will be ***X*** failures before the ***R***-th success, when the constant probability of a success is ***Prob***.

> **Note:** This function is similar to the binomial distribution, except that the number of successes is fixed, and the number of trials is variable. Like the binomial, trials are assumed to be independent.

$$P_{R,Prob}(X) = \binom{X+R-1}{R-1} Prob^{R}(1-Prob)^{X}$$

$$\binom{X+R-1}{R-1} \text{ is a binomial coefficient}$$

6.18.52 NORMDIST

Summary: returns the value of the probability density function or the cumulative distribution function for the normal distribution with the mean and standard deviation given.

Syntax: NORMDIST(*Number* ***X*** ; *Number* ***Mean*** ; *Number* ***StandardDeviation*** [; *Logical* ***Cumulative*** = TRUE()])

Returns: *Number*

Constraints: ***StandardDeviation*** > 0.

Semantics: In the following μ is ***Mean*** and σ is ***StandardDeviation***.

If ***Cumulative*** is FALSE, NORMDIST returns the value

$$\frac{e^{-\frac{1}{2}\cdot\left(\frac{X-\mu}{\sigma}\right)^{2}}}{\sqrt{2\pi}\,\sigma}$$

If ***Cumulative*** is TRUE, NORMDIST returns the value

$$\int_{-\infty}^{X} \frac{e^{-\frac{1}{2}\cdot\left(\frac{t-\mu}{\sigma}\right)^{2}}}{\sqrt{2\pi}\,\sigma}\,dt$$

See also LEGACY.NORMSDIST 6.18.54

6.18.53 NORMINV

Summary: returns the inverse of NORMDIST(x;***Mean***;***StandardDeviation***,TRUE()).

Syntax: NORMINV(*Number* ***P*** ; *Number* ***Mean*** ; *Number* ***StandardDeviation***)

Returns: *Number*

Constraints: ***StandardDeviation*** > 0 and 0 < ***P*** < 1.

Semantics: NORMINV returns the unique number x such that NORMDIST(x;***Mean***;***StandardDeviation***;TRUE()) = ***P***.

See also NORMDIST 6.18.52

6.18.54 LEGACY.NORMSDIST

Summary: returns the value of the cumulative distribution function for the standard normal distribution.

Syntax: LEGACY.NORMSDIST(*Number* ***X***)

Returns: *Number*

Constraints: None

Semantics: LEGACY.NORMSDIST returns the value

$$\int_{-\infty}^{X} \frac{e^{-\frac{1}{2}\cdot t^2}}{\sqrt{2\,\pi}}\,dt$$

This is exactly NORMDIST(***X***;0;1;TRUE()).

See also NORMDIST 6.18.52, LEGACY.NORMSINV 6.18.55

6.18.55 LEGACY.NORMSINV

Summary: returns the inverse of LEGACY.NORMSDIST(X).

Syntax: LEGACY.NORMSINV(*Number* ***P***)

Returns: *Number*

Constraints: 0 < ***P*** < 1.

Semantics: LEGACY.NORMSINV returns NORMINV (***P***).

See also NORMINV 6.18.53, LEGACY.NORMSDIST 6.18.54

6.18.56 PEARSON

Summary: PEARSON returns the Pearson correlation coefficient of two data sets

Syntax: PEARSON(*ForceArray Array* ***IndependentValues*** ; *ForceArray Array* ***DependentValues***)

Returns: *Number*

Constraints: COLUMNS(***IndependentValues***) = COLUMNS(***DependentValues***), ROWS(***IndependentValues***) = ROWS(***DependentValues***), both sequences shall contain at least one number at corresponding positions each.

Semantics:

- ***IndependentValues***: represents the array of the first data set. (X-Values)
- ***DependentValues***: represents the array of the second data set. (Y-Values)

$$r = \frac{\sum_{i=1}^{N} (x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{N} (x_i - \overline{x})^2 \sum_{i=1}^{N} (y_i - \overline{y})^2}}$$

$$\overline{x}\,, \overline{y} \quad \text{are the averages of the given } x\,, y \text{ data}$$

For an empty element or an element of type Text or Boolean in ***IndependentValues*** the element at the corresponding position of ***DependentValues*** is ignored, and vice versa.

See also COLUMNS 6.13.5, ROWS 6.13.30

6.18.57 PERCENTILE

Summary: Calculates the x-th sample percentile among the values in range.

Syntax: PERCENTILE(*NumberSequenceList* ***Data*** ; *Number* ***X***)

Returns: *Number*

Constraints:

- COUNT(***Data***) > 0
- 0 ≤ ***X*** ≤ 1
- **Semantics:**
- ***Data***: The array or range of values to get the percentile from.
- ***X***: The percentile value between 0 and 1, inclusive. If ***X*** is not a multiple of $\frac{1}{n-1}$, PERCENTILE interpolates to obtain the value between two data points.

Returns the ***X***-th sample percentile of data values in ***Data***. A percentile returns the scale value for a data series which goes from the smallest (Alpha = 0) to the largest value (Alpha = 1) of a data series. For Alpha = 25%, the percentile means the first quartile; Alpha = 50% is the MEDIAN.

Step 1:
Sort the list of numbers given by array $Data$.

Step 2:
Calculate the ranking $\{1,\ldots,n\}$, split into integer and decimal part

$$r = 1 + X \cdot (n-1) = I + D$$

with
X = the percentile you want to find
n = the count of values
I = the integer part of the ranking = $\lfloor r \rfloor$
D = the decimal part of the ranking = $r - \lfloor r \rfloor$

Step 3:
Interpolate between the necessary two numbers

$$PERCENTILE = Y_I + D \cdot (Y_{I+1} - Y_I)$$

with Y_I being the data point ranked at position I

See also COUNT 6.13.6, MAX 6.18.45, MAX 6.18.45, MEDIAN 6.18.47, MIN 6.18.48, PERCENTRANK 6.18.58, QUARTILE 6.18.64, RANK 6.18.65

6.18.58 PERCENTRANK

Summary: Returns the percentage rank of a value in a sample.

Syntax: PERCENTRANK(*NumberSequenceList* ***Data*** ; *Number* ***X*** [; *Integer* ***Significance*** = 3])

Returns: *Number*

Constraints:

- COUNT(***Data***) > 0
- MIN(***Data***) ≤ ***X*** ≤ MAX(***Data***)
- INT(***Significance***) = ***Significance***; ***Significance*** ≥ 1

Semantics:

- ***Data***: the array or range of data with numeric values.
- ***X***: the value whose rank is to be determined.
- ***Significance***: an optional value that identifies the number of significant digits for the returned percentage value. If omitted, a value of 3 is used (0.xxx).

Returns the rank of a value in a data set ***Data*** as a percentage of the data set, a value between 0 and 1, inclusive. This function can be used to evaluate the relative standing of a value within a data set.

For COUNT(***Data***) > 1, PERCENTRANK returns r / (COUNT(***Data***) -1), where r is the rank of ***X*** in ***Data***. The rank of the lowest number in ***Data*** is 0, and of the next lowest number 1, and so on. If ***X*** is not in ***Data***, it is assigned a fractional rank proportionately between the rank of the numbers on either side. Specifically, if ***X*** lies between Y and Z = Y + 1 (Y < ***X*** < Z) with Y being the largest number smaller than ***X*** and Z the smallest number larger than ***X***, and where Y has rank ry, the rank of ***X*** is calculated as

$$rx = ry + \frac{X - Y}{Z - Y}$$

In the special case where COUNT(***Data***) = 1, the only valid value for ***X*** is the single value in ***Data***, in which case PERCENTRANK returns 1.

See also COUNT 6.13.6, INT 6.17.2, MAX 6.18.45, MIN 6.18.48, PERCENTILE 6.18.57, RANK 6.18.65

6.18.59 PERMUT

Summary: returns the number of permutations of *k* objects taken from *n* objects.

Syntax: PERMUT(*Integer* ***N*** ; *Integer* ***K***)

Returns: *Number*

Constraints: ***N*** ≥ 0; ***K*** ≥ 0; ***N*** ≥ ***K***

Semantics: PERMUT returns

$$\frac{N!}{(N-K)!}$$

6.18.60 PERMUTATIONA

Summary: Returns the number of permutations for a given number of objects (repetition allowed).

Syntax: PERMUTATIONA(*Integer* ***Total*** ; *Integer* ***Chosen***)

Returns: *Number*

Constraints: ***Total*** ≥ 0, ***Chosen*** ≥ 0

Semantics: *Given* ***Total*** *number of objects, return the number of permutations containing* ***Chosen*** *number of objects, with repetition permitted. The result is 1 if* ***Total*** *= 0 and* ***Chosen*** *= 0, otherwise the result is*

$$PERMUTATIONA = Total^{Chosen}$$

6.18.61 PHI

Summary: Returns the values of the density function for a standard normal distribution.

Syntax: PHI(*Number* ***N***)

Returns: *Number*

Semantics: PHI(***N***) is a synonym for NORMDIST(***N***,0,1,FALSE()).

See also NORMDIST 6.18.52

6.18.62 POISSON

Summary: returns the probability or the cumulative distribution function for the Poisson distribution

Syntax: POISSON(*Integer* ***X*** ; *Number* ***λ*** [; *Logical* ***Cumulative*** = TRUE])

Returns: *Number*

Constraints: ***λ*** > 0, ***X*** ≥ 0

Semantics: If ***Cumulative*** is FALSE, POISSON returns the value

$$\frac{e^{-\lambda}\lambda^{\lfloor X \rfloor}}{\lfloor X \rfloor !}$$

If ***Cumulative*** is TRUE, POISSON returns the value

$$\sum_{k=0}^{k=\lfloor X \rfloor} \frac{e^{-\lambda}\lambda^{k}}{k\,!}$$

6.18.63 PROB

Summary: Returns the probability that a discrete random variable lies between two limits.

Syntax: PROB(*ForceArray Array* ***Data*** ; *ForceArray Array* ***Probability*** ; *Number* ***Start*** [; *Number* ***End***])

Returns: *Number*

Constraints:

- The sum of the probabilities in ***Probability*** shall equal 1.
- All values in ***Probability*** shall be > 0 and ≤ 1.
- COUNT(***Data***) = COUNT(***Probability***)

Semantics:

- ***Data***: the array or range of data in the sample (the Number values in this array or range are referred to below as $d_1, d_2, \ldots, d_n$).
- ***Probability***: the array or range of the corresponding probabilities (the Number values in this array or range are referred to below as $p_1, p_2, \ldots, p_n$).
- ***Start***: the start value (lower bound) of the interval whose probabilities are to be summed.
- ***End***: (optional) the end value (upper bound) of the interval whose probabilities are to be summed. If omitted, ***End*** = ***Start*** is used.

Suppose that $I(x, a, b)$ denotes the indicator function that is 1 if $a \le x \le b$ and 0 otherwise.

Then PROB returns

$$\sum_{i=1}^{n} (I(d_i, Start, End) \times p_i)$$

i.e. the sum of all probabilities p_i whose corresponding data value d_i satisfies $Start \le d_i \le End$. Note that if $End < Start$ then PROB returns 0 since in this case $I(d_i, Start, End) = 0$ for all *i*.

See also COUNT 6.13.6

6.18.64 QUARTILE

Summary: Returns a quartile of a set of data points.

Syntax: QUARTILE(*NumberSequence* ***Data*** ; *Integer* ***Quart***)

Returns: *Number*

Constraints:

- COUNT(***Data***) > 0
- 0 ≤ ***Quart*** ≤ 4

Semantics:

- ***Data***: The cell range or data array of numeric values.
- ***Quart***: The number of the quartile to return.

  If ***Quart*** = 0, the minimum value is returned, which is equivalent to the MIN() function.

  If ***Quart*** = 1, the value of the 25th percentile is returned.

  If ***Quart*** = 2, the value of the 50th percentile is returned, which is equivalent to the MEDIAN() function.

  If ***Quart*** = 3, the value of the 75th percentile is returned.

  If ***Quart*** = 4, the maximum value is returned, which is equivalent to the MAX() function.

Based on the statistical rank of the data points in ***Data***, QUARTILE returns the percentile value indicated by ***Quart***. The percentile is calculated as ***Quart*** divided by 4. An algorithm to calculate the percentile for a set of data points is given in the definition of PERCENTILE.

See also COUNT 6.13.6, MAX 6.18.45, MEDIAN 6.18.47, MIN 6.18.48, PERCENTILE 6.18.57, PERCENTRANK 6.18.58, RANK 6.18.65

6.18.65 RANK

Summary: *R*eturns the rank of a number in a list of numbers.

Syntax: RANK(*Number* ***Value*** ; *NumberSequenceList* ***Data*** [; *Number* ***Order*** = 0])

Returns: *Number*

Constraints: ***Value*** shall exist in ***Data***.

Semantics: *The RANK function returns the rank of a value within a list.*

- ***Value***: the number for which to determine the rank.
- ***Data***: numbers used to determine the ranking.
- ***Order***: specifies how to rank the numbers:
  If 0 or omitted, ***Data*** is ranked in descending order.
  If not 0, ***Data*** is ranked in ascending order.

If a number in ***Data*** occurs more than once it is given the same rank, but increments the rank for subsequent different numbers. If ***Value*** does not exist in ***Data*** an Error is returned.

6.18.66 RSQ

Summary: Returns the square of the Pearson product moment correlation coefficient.

Syntax: RSQ(*ForceArray Array* ***ArrayY*** ; *ForceArray Array* ***ArrayX***)

Returns: *Number*

Constraints:

The arguments shall be either numbers or names, arrays, or references that contain numbers.

If an array or reference argument contains Text, Logical values, or empty cells, those values are ignored; however, cells with the value zero are included.

If ***ArrayY*** *and* ***ArrayX*** *are empty or have a different number of data points, then #N/A is returned.*

COLUMNS(***ArrayY***) = COLUMNS(***ArrayX***), ROWS(***ArrayY***) = ROWS(***ArrayX***)***Semantics:*** *The r-squared value can be interpreted as the proportion of the variance in y attributable to the variance in x.*

$$r^2 = \frac{\sum_{i=1}^{N}(y_i - \overline{y})^2 - \sum_{i=1}^{N}(y_i - y_{calc})^2}{\sum_{i=1}^{N}(y_i - \overline{y})^2}$$

$$y_{calc} = a + bx$$

and

$$a = \frac{\left(\sum_{i=1}^{N}(x_i^2)\right)\left(\sum_{i=1}^{N}(y_i)\right) - \left(\sum_{i=1}^{N}(x_i)\right)\left(\sum_{I=1}^{N}(x_i y_i)\right)}{N\left(\sum_{i=1}^{N}(x_i^2)\right) - \left(\sum_{i=1}^{N}(x_i)\right)^2}$$

$$b = \frac{N\left(\sum_{i=1}^{N}(x_i y_i)\right) - \left(\sum_{i=1}^{N}(x_i)\right)\left(\sum_{i=1}^{N}(y_i)\right)}{N\left(\sum_{i=1}^{N}(x_i^2)\right) - \left(\sum_{i=1}^{N}(x_i)\right)^2}$$

The result of the RSQ function is the same as PEARSON * PEARSON.

For an empty element or an element of type Text or Boolean in ***ArrayY*** the element at the corresponding position of ***ArrayX*** is ignored, and vice versa.

See also COLUMNS 6.13.5, ROWS 6.13.30, PEARSON 6.18.56

6.18.67 SKEW

Summary: Estimates the skewness of a distribution using a sample set of numbers.

Syntax: SKEW({ *NumberSequenceList* ***Sample*** }$^{+}$)

Returns: *Number*

Constraints: The sequence shall contain three numbers at least.

Semantics: Estimates the skewness of a distribution using a sample set of numbers.

Given the expectation value $\overline{x}$ and the standard deviation estimate s, the skewness becomes

$$v=\frac{N}{(N-1)(N-2)}\sum_{i=1}^{N}\left(\frac{x_i-\overline{x}}{s}\right)^3$$

See also SKEWP 6.18.68

6.18.68 SKEWP

Summary: Calculates the skewness of a distribution using the population of a random variable.

Syntax: SKEWP({ *NumberSequence* ***Population*** }+)

Returns: *Number*

Constraints: The sequence shall contain three numbers at least.

Semantics: Calculates the skewness of a distribution using the population, i.e. the possible outcomes, of a random variable.

Given the expectation value $\overline{x}$ and the standard deviation σ,the skewness becomes

$$v=\frac{1}{N}\sum_{i=1}^{N}\left(\frac{x_i-\overline{x}}{\sigma}\right)^3$$

See also SKEW 6.18.67

6.18.69 SLOPE

Summary: Calculates the slope of the linear regression line.

Syntax: SLOPE(*ForceArray Array* ***Y*** ; *ForceArray Array* ***X***)

Returns: *Number*

Constraints: COLUMNS(***Y***) = COLUMNS(***X***), ROWS(***Y***) = ROWS(***X***), both sequences shall contain at least one number at corresponding positions each.

Semantics: Calculates the slope of the linear regression line.

$$a=\frac{\sum_{i=1}^{N}(x_i-\overline{x})(y_i-\overline{y})}{\sum_{i=1}^{N}(x_i-\overline{x})^2}$$

For an empty element or an element of type Text or Boolean in ***Y*** the element at the corresponding position of ***X*** is ignored, and vice versa.

See also COLUMNS 6.13.5, ROWS 6.13.30, INTERCEPT 6.18.38, STEYX 6.18.76

6.18.70 SMALL

Summary: Finds the nth smallest value in a list.

Syntax: SMALL(*NumberSequenceList* ***List*** ; *Integer|Array* ***N***)

Returns: *Number* or *Array*

Constraints: ROUNDDOWN(***N***;0) = ***N***, effectively being INT(***N***) = ***N*** for positive numbers. If the resulting ***N*** is <1 or larger than the size of ***List***, Error is returned.

Semantics: If ***N*** is an array of numbers, an array of smallest values is returned.

See also INT 6.17.2, LARGE 6.18.40, ROUNDDOWN 6.17.6

6.18.71 STANDARDIZE

Summary: Calculates a normalized value of a random variable.

Syntax: STANDARDIZE(*Number* ***Value*** ; *Number* ***Mean*** ; *Number* ***Sigma***)

Returns: *Number*

Constraints: ***Sigma*** > 0

Semantics: Calculates a normalized value of a random variable.

$$\text{STANDARDIZE} = \frac{(Value - Mean)}{Sigma}$$

See also GAUSS 6.18.33

6.18.72 STDEV

Summary: Compute the sample standard deviation of a set of numbers.

Syntax: STDEV({ *NumberSequenceList* ***N*** }$^{+}$)

Returns: *Number*

Constraints: At least two numbers shall be included. Returns an Error if less than two Numbers are provided.

Semantics: Computes the sample standard deviation *s*, where

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2$$

with

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Note that *s* is not the same as the standard deviation of the set, σ, which uses *n* rather than *n* − 1.

See also STDEVP 6.18.74, AVERAGE 6.18.3

6.18.73 STDEVA

Summary: Calculate the standard deviation using a sample set of values, including values of type Text and Logical.

Syntax: STDEVA({ *Any* ***Sample*** }$^{+}$)

Returns: *Number*

Constraints: COUNTA(***Sample***) > 1.

Semantics: Unlike the STDEV function, includes values of type Text and Logical. Text values are treated as number 0. Logical TRUE is treated as 1, and FALSE is treated as 0. Empty cells are not included.

The handling of string constants as parameters is implementation-defined. Either, string constants are converted to numbers, if possible and otherwise, they are treated as 0, or string constants are always treated as 0.

Suppose the resulting sequence of values is x_1, x_2, …, x_n. Then let

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

STDEVA returns

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2}$$

See also COUNTA 6.13.7, STDEV 6.18.72

6.18.74 STDEVP

Summary: Calculates the standard deviation using the population of a random variable, including values of type Text and Logical.

Syntax: STDEVP({ *NumberSequence* ***N*** }$^{+}$)

Returns: *Number*

Constraints: COUNT(***N***) ≥ 1.

Semantics: Computes the standard deviation of the set σ, where

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2$$

Note that σ is not the same as the sample standard deviation, *s*, which uses *n* − 1 rather than *n*.

See also COUNT 6.13.6, STDEV 6.18.72, AVERAGE 6.18.3

6.18.75 STDEVPA

Summary: Calculates the standard deviation using the population of a random variable, including values of type Text and Logical.

Syntax: STDEVPA({ *Any* ***Sample*** }$^{+}$)

Returns: *Number*

Constraints: COUNTA(***Sample***) ≥ 1.

Semantics: Unlike the STDEV function, includes values of type Text and Logical. Text values are treated as number 0. Logical TRUE is treated as 1, and FALSE is treated as 0. Empty cells are not included.

Given the expectation value $\overline{x}$ the standard deviation becomes

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2$$

In the sequence, only Numbers and Logical types are considered; cells with Text are converted to 0; other types are ignored. If Logical types are a distinct type, they are still included, with TRUE considered 1 and FALSE considered 0. Any ***Sample*** may be of type ReferenceList.

The handling of string constants as parameters is implementation-defined. Either, string constants are converted to numbers, if possible and otherwise, they are treated as zero, or string constants are always treated as zero.

See also COUNTA 6.13.7, STDEVP 6.18.74

6.18.76 STEYX

Summary: Calculates the standard error of the predicted y value for each x in the regression.

Syntax: STEYX(*ForceArray Array* ***MeasuredY*** ; *ForceArray Array* ***X***)

Returns: *Number*

Constraints: COLUMNS(***MeasuredY***) = COLUMNS(***X***), ROWS(***MeasuredY***) = ROWS(***X***), both sequences shall contain at least three numbers at corresponding positions each.

Semantics: Calculates the standard error of the predicted y value for each x in the regression.

$$\mathrm{STEYX} = \sqrt{\frac{1}{n(n-2)}\left(n\sum_n y_i^2 - \left(\sum_n y_i\right)^2 - \frac{\left(n\sum_n x_i y_i - \sum_n x_i \sum_n y_i\right)^2}{n\sum_n x^2 - \left(\sum_n x\right)^2}\right)}$$

For an empty element or an element of type Text or Boolean in ***MeasuredY*** the element at the corresponding position of ***X*** is ignored, and vice versa.

See also COLUMNS 6.13.5, ROWS 6.13.30, INTERCEPT 6.18.38, SLOPE 6.18.69

6.18.77 LEGACY.TDIST

Summary: Returns the area to the tail or tails of the probability density function of the t-distribution.

Syntax: LEGACY.TDIST(*Number* ***X*** ; *Integer* ***Df*** ; *Integer* ***Tails***)

Returns: *Number*

Constraints: ***X*** ≥ 0, ***Df*** ≥ 1, ***Tails*** = 1 or 2

Semantics: Then LEGACY.TDIST returns

$$Tails \cdot \int_x^{\infty} f(t)\,dt$$

where

$$f(t) = \frac{\Gamma\left(\frac{Df+1}{2}\right)}{\sqrt{\pi\,Df}\,\Gamma\left(\frac{Df}{2}\right)}\left(1+\frac{t^2}{Df}\right)^{-\frac{(Df+1)}{2}}$$

Note that ***Df*** denotes the degrees of freedom of the t-distribution and Γ is the Gamma function.

See also GAMMA 6.16.34, BETADIST 6.18.7, BINOMDIST 6.18.10, CHISQDIST 6.18.12, EXPONDIST 6.18.21, FDIST 6.18.22, GAMMADIST 6.18.31, GAUSS 6.18.33, HYPGEOMDIST 6.18.37, LOGNORMDIST 6.18.44, NEGBINOMDIST 6.18.51, NORMDIST 6.18.52, POISSON 6.18.62, WEIBULL 6.18.86

6.18.78 TINV

Summary: Calculates the inverse of the two-tailed t-distribution.

Syntax: TINV(*Number* ***Probability*** ; *Integer* ***DegreeOfFreedom***)

Returns: *Number*

Constraints: 0 < ***Probability*** ≤ 1, ***DegreeOfFreedom*** ≥ 1

Semantics: Calculates the inverse of the two-tailed t-distribution.

See also LEGACY.TDIST 6.18.77

6.18.79 TREND

Summary: Returns predicted values based on a simple or multiple linear regression.

Syntax: TREND(*Array* ***KnownY*** [; [*Array* ***KnownX***] [; [*Array* ***NewX***] [; *Logical* ***Const*** = TRUE]]])

Returns: *Array*

Constraints: (COLUMNS(***KnownY***) = COLUMNS(***KnownX***) and ROWS(***KnownY***) = ROWS(***KnownX***)) or (COLUMNS(***KnownY***) = 1 and ROWS(***KnownY***) = ROWS(***KnownX***) and COLUMNS(***KnownX***) = COLUMNS(***NewX***)) or (COLUMNS(***KnownY***) = COLUMNS(***KnownX***) and ROWS(***KnownY***) = 1 and ROWS(***KnownX***) = ROWS(***NewX***))

Semantics:

KnownY: The set of known y-values to be used to determine the regression equation

KnownX: The set of known x-values to be used to determine the regression equation. If omitted or an empty parameter, it is set to the sequence 1,2,3,…,*k*, where *k* = ROWS(***KnownY***) · COLUMNS(***KnownY***).

NewX: The set of x-values for which predicted y-values are to be calculated. If omitted or an empty parameter, it is set to ***KnownX***.

Const: If set to FALSE, the model constant *a* is equal to 0.

LINEST(***KnownY***; ***KnownX***; ***Const***; FALSE()) either returns an error an array with 1 row and *n* + 1 columns. If it returns an error then so does TREND. If it returns an array, we call the entries in that array $b_n, b_{n-1}, \ldots, b_1, a$.

Let z_{ij} denote the entry in the *i*th row and *j*th column of ***NewX***.

If COLUMNS(***KnownY***) = COLUMNS(***KnownX***) and ROWS(***KnownY***) = ROWS(***KnownX***), then TREND returns an array with ROWS(***NewX***) rows and COLUMNS(***NewX***) column, such that the entry in its *i*th row and *j*th column is $a + b_1 \cdot z_{ij}$.

Otherwise, if COLUMNS(***KnownY***) = 1 and ROWS(***KnownY***) = ROWS(***KnownX***) and COLUMNS(***KnownX***) = COLUMNS(***NewX***), then TREND returns an array with ROWS(***NewX***) rows and 1 column, such that the entry in the *i*th row is $a + \sum_{j=1}^{n} b_j \cdot z_{ij}$.

Otherwise, if COLUMNS(***KnownY***) = COLUMNS(***KnownX***) and ROWS(***KnownY***) = 1 and ROWS(***KnownX***) = ROWS(***NewX***), then TREND returns an array with 1 row and COLUMNS(***NewX***) columns, such that the entry in the *j*th column is $a + \sum_{i=1}^{n} b_i \cdot z_{ij}$.

See also COLUMNS 6.13.5, ROWS 6.13.30, INTERCEPT 6.18.38, LINEST 6.18.41, SLOPE 6.18.69 , STEYX 6.18.76

6.18.80 TRIMMEAN

Summary: Returns the mean of a data set, ignoring a proportion of high and low values.

Syntax: TRIMMEAN(*NumberSequenceList* ***DataSet*** ; *Number* ***CutOffFraction***)

Returns: *Number*

Constraints: 0 ≤ ***CutOffFraction*** < 1

Semantics: Returns the mean of a data set, ignoring a proportion of high and low values.

Let *n* denote the number of elements in the data set and let

$$SortedDataSet_1, SortedDataSet_2, SortedDataSet_3, \ldots, SortedDataSet_n$$

be the values in the data set sorted in ascending order. Moreover let

$$CutOff = \mathrm{INT}\left(\frac{n \cdot CutOffFraction}{2}\right)$$

Then TRIMMEAN returns the value

$$\frac{1}{n-2\cdot CutOff}\sum_{i=CutOff+1}^{n-CutOff} SortedDataSet_i$$

See also AVERAGE 6.18.3 , GEOMEAN 6.18.34 , HARMEAN 6.18.36

6.18.81 TTEST

Summary: Calculates the p-value of a 2-sample t-test.

Syntax: TTEST(*ForceArray Array* ***X*** ; *ForceArray Array* ***Y*** ; *Integer* ***Tails*** ; *Integer* ***Type***)

Returns: *Number*

Constraints: COUNT(***X***) > 1, COUNT(***Y***) > 1, ***Tails*** = 1 or 2, ***Type*** = 1,2, or 3, (COUNT(***X***) = COUNT(***Y***) or ***Type*** ≠ 1)

COLUMNS(***X***) = COLUMNS(***Y***), ROWS(***X***) = ROWS(***Y***)

Semantics: Let $\boldsymbol{X}_1$, $\boldsymbol{X}_2$, …,$\boldsymbol{X}_n$ be the numbers in the sequence ***X*** and $\boldsymbol{Y}_1$, $\boldsymbol{Y}_2$, …,$\boldsymbol{Y}_m$ be the numbers in the sequence ***Y***. Then

$$\overline{X}=\frac{1}{n}\sum_{i=1}^{n} X_i$$

and

$$\overline{Y}=\frac{1}{m}\sum_{i=1}^{m} Y_i$$

Moreover let

$$s_X^2=\frac{1}{n-1}\sum_{i=1}^{n}(X_i-\overline{X})^2$$

$$s_Y^2=\frac{1}{m-1}\sum_{i=1}^{m}(Y_i-\overline{Y})^2$$

and

$$f(x,df)=\frac{\Gamma\left(\frac{df+1}{2}\right)}{\sqrt{\pi\, df}\,\Gamma\left(\frac{df}{2}\right)}\left(1+\frac{x^2}{df}\right)^{-\frac{(df+1)}{2}}$$

where Γ is the Gamma function.

(1) If type = 1, TTEST calculates the p-value for a paired-sample comparison of means test. Note that in this case due to the above constraints n = m. With

$$s_{X-Y}^2=\frac{1}{n-1}\sum_{i=1}^{n}((X_i-Y_i)-(\overline{X}-\overline{Y}))^2$$

and

$$t=\left|\frac{\overline{X}-\overline{Y}}{\sqrt{s_{X-Y}^2}}\sqrt{n}\right|$$

TTEST returns

$$Tails\cdot\int_t^{\infty} f(x,n-1)\,dx$$

(2) If ***Type*** = 2, TTEST calculates the p-value of a comparison of means for independent samples from populations with equal variance. With

(1) $$s_p^2 = \frac{(n-1)s_X^2 + (m-1)s_Y^2}{n+m-2}$$

and

$$t = \left| \frac{\overline{X} - \overline{Y}}{\sqrt{s_p^2 \left(\frac{1}{n} + \frac{1}{m}\right)}} \right|$$

TTEST returns

$$tails \cdot \int_t^{\infty} f(x, n+m-2)\,dx$$

(3) If ***Type*** = 3, TTEST calculates the p-value of a comparison of means for independent samples from populations with not necessarily equal variances. With

(2) $$t = \left| \frac{\overline{X} - \overline{Y}}{\sqrt{\frac{s_X^2}{n} + \frac{s_Y^2}{m}}} \right|$$

and

$$v = \frac{\left(\frac{S_X^2}{n} + \frac{S_Y^2}{m}\right)^2}{\frac{\left(\frac{S_X^2}{n}\right)^2}{(n-1)} + \frac{\left(\frac{S_Y^2}{m}\right)^2}{(m-1)}}$$

TTEST returns

$$tails \cdot \int_t^{\infty} f(x, v)\,dx$$

For an empty element or an element of type Text or Boolean in ***X*** the element at the corresponding position of ***Y*** is ignored, and vice versa.

See also COLUMNS 6.13.5, COUNT 6.13.6, ROWS 6.13.30, FTEST 6.18.30, LEGACY.TDIST 6.18.77, ZTEST 6.18.87

6.18.82 VAR

Summary: Compute the sample variance of a set of numbers.

Syntax: VAR({ *NumberSequence* ***N*** }$^{+}$)

Returns: *Number*

Constraints: At least two numbers shall be included. Returns an Error if less than two Numbers are provided.

Semantics: Computes the sample variance s^2, where

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2 = \frac{1}{n-1}\left(\left(\sum_{i=1}^{n} x_i^{\,2}\right) - n\,\overline{x}^2\right)$$

Note that s^2 is not the same as the variance of the set, σ^2, which uses n rather than $n - 1$.

See also VARP 6.18.84, STDEV 6.18.72, AVERAGE 6.18.3

6.18.83 VARA

Summary: Estimates the variance using a sample set of values, including values of type Text and Logical.

Syntax: VARA({ *Any* ***Sample*** }$^{+}$)

Returns: *Number*

Constraints: The sequence shall contain two numbers at least.

Semantics: Unlike the VAR function, includes values of type Text and Logical. Text values are treated as number 0. Logical TRUE is treated as 1, and FALSE is treated as 0. Empty cells are not included.

Given the expectation value $\overline{x}$ the estimated variance becomes

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2 = \frac{1}{n-1}\left(\left(\sum_{i=1}^{n} x_i^{\,2}\right) - n\,\overline{x}^2\right)$$

In the sequence, only Numbers and Logical types are considered; cells with Text are converted to 0; other types are ignored. If Logical types are a distinct type, they are still included, with TRUE considered 1 and FALSE considered 0. Any ***Sample*** may be of type ReferenceList.

The handling of string constants as parameters is implementation-defined. Either, string constants are converted to numbers, if possible and otherwise, they are treated as zero, or string constants are always treated as zero.

See also VAR 6.18.82

6.18.84 VARP

Summary: Compute the variance of the set for a set of numbers.

Syntax: VARP({ *NumberSequence* ***N*** }$^{+}$)

Returns: *Number*

Constraints: COUNT(***N***) ≥ 1

Semantics: Computes the variance of the set σ^2, where

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2 = \frac{1}{n}\left(\left(\sum_{i=1}^{n} x_i^{\,2}\right) - n\,\overline{x}^2\right)$$

Note that σ^2 is not the same as the sample variance, s^2, which uses $n - 1$ rather than n.

If only one number is provided, returns 0.

See also COUNT 6.13.6, VAR 6.18.82, STDEVP 6.18.74, AVERAGE 6.18.3

6.18.85 VARPA

Summary: Calculates the variance using the population of the distribution, including values of type Text and Logical.

Syntax: VARPA({ *Any* ***Sample*** }$^{+}$)

Returns: *Number*

Constraints: COUNTA(***Sample***) ≥ 1.

Semantics: Unlike the VARP function, includes values of type Text and Logical. Text values are treated as number 0. Logical TRUE is treated as 1, and FALSE is treated as 0. Empty cells are not included.

Given the expectation value $\overline{x}$ the variance becomes

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2 = \frac{1}{n}\left(\left(\sum_{i=1}^{n} x_i^2\right) - n\overline{x}^2\right)$$

In the sequence, only Numbers and Logical types are considered; cells with Text are converted to 0; other types are ignored. If Logical types are a distinct type, they are still included, with TRUE considered 1 and FALSE considered 0. Any ***Sample*** may be of type ReferenceList.

The handling of string constants as parameters is implementation-defined. Either, string constants are converted to numbers, if possible and otherwise, they are treated as zero, or string constants are always treated as zero.

See also COUNTA 6.13.7, VARP 6.18.84

6.18.86 WEIBULL

Summary: Calculates the Weibull distribution.

Syntax: WEIBULL(*Number* ***Value*** ; *Number* ***Shape*** ; *Number* ***Scale*** ; *Logical* ***Cumulative***)

Returns: *Number*

Constraints: ***Value*** ≥ 0; ***Shape*** > 0; ***Scale*** > 0

Semantics: Calculates the Weibull distribution at the position ***Value***.

If ***Cumulative*** is FALSE, the probability density function is calculated:

$$\frac{Shape}{Scale}\left(\frac{Value}{Scale}\right)^{Shape-1} e^{-\left(\frac{Value}{Scale}\right)^{Shape}}$$

If ***Cumulative*** is TRUE, the cumulative distribution function is calculated:

$$1 - e^{-\left(\frac{Value}{Scale}\right)^{Shape}}$$

See also BETADIST 6.18.7, BINOMDIST 6.18.10, CHISQDIST 6.18.12, EXPONDIST 6.18.21, FDIST 6.18.22, GAMMADIST 6.18.31, GAUSS 6.18.33, HYPGEOMDIST 6.18.37, LOGNORMDIST 6.18.44, NEGBINOMDIST 6.18.51, NORMDIST 6.18.52, POISSON 6.18.62, LEGACY.TDIST 6.18.77

6.18.87 ZTEST

Summary: Calculates the probability of observing a sample mean as large or larger than the mean of the given sample for samples drawn from a normal distribution.

Syntax: ZTEST(*NumberSequenceList* ***Sample*** ; *Number* ***Mean*** [; *Number* ***Sigma***])

Returns: *Number*

Constraints: The sequence ***Sample*** shall contain at least two numbers.

Semantics: Calculates the probability of observing a sample mean as large or larger than the mean of the given ***Sample*** for samples drawn from a normal distribution with the given mean ***Mean*** and the given standard deviation ***Sigma***. If ***Sigma*** is omitted, it is estimated from ***Sample***, using STDEV. With ***Sample*** being the mean of ***Sample*** and

$$z = \frac{\overline{Sample} - Mean}{Sigma}\sqrt{n}$$

ZTEST returns

$$P(z \le Z) = \frac{1}{\sqrt{2\pi}} \int_{z}^{\infty} e^{-\frac{x^2}{2}} dx$$

See also FTEST 6.18.30, TTEST 6.18.81

6.19 Number Representation Conversion Functions

6.19.1 General

These functions convert between different representations of numbers, such as between different bases and Roman numerals.

The base conversion functions xxx2BIN (such as DEC2BIN), xxx2OCT, and xxx2HEX functions return Text, while the xxx2DEC functions return Number. All of the xxx2yyy functions accept either Text or Number, though a Number is interpreted as the digits when printed in base 10. These are intended to support relatively small numbers, and have a somewhat convoluted interface and semantics, as described in their specifications. General base conversion capabilities are provided by BASE and DECIMAL.

As an argument for the HEX2xxx functions, a hexadecimal number is any string consisting solely of the characters "0","1" to "9", "a" to "f" and "A" to "F". The hexadecimal output of an xxx2HEX function **shall** be a string consisting solely of the characters "0","1" to "9" (U+0030 through U+0039), "a" to "f" (U+0061 through U+0066) and "A" to "F" (U+0041 through U+0046), and should be a string consisting solely of the characters "0","1" to "9" and "A" to "F". In both cases, the 40th bit (from the right) is considered a sign bit.

6.19.2 ARABIC

Summary: Convert Roman numerals to Number.

Syntax: ARABIC(*Text* ***X***)

Returns: *Number*

Constraints: ***X*** shall contain Roman numerals, or an empty string.

Semantics: Converts the Roman numeral to Number. This is the reverse of ROMAN; see ROMAN for the values of individual Roman numeral symbols. A Roman symbol to the left of a larger symbol (directly or indirectly) reduces the final value by the symbol amount, otherwise, it increases the final amount by the symbol's amount. Case is ignored.

The characters accepted are U+004D "M", U+0044 "D", U+0043 "C", U+004C "L", U+0058 "***X***", U+0056 "V", U+0049 "I", U+006D "m", U+0064 "d", U+0063 "c", U+006C "l", U+0078 "x", U+0076 "v", U+0069 "i" .

The following identity shall hold: ARABIC(ROMAN(x; any)) = x, when ROMAN(x; any) is not an Error.

If ***X*** is an empty string, 0 is returned.

See also Infix Operator "&" 6.4.10, ROMAN 6.19.17

6.19.3 BASE

Summary: Converts a number into a text representation with the given base.

Syntax: BASE(*Integer* ***X*** ; *Integer* ***Radix*** [; *Integer* ***MinimumLength***])

Returns: *Text*

Constraints: ***X*** ≥ 0, 2 ≤ ***Radix*** ≤ 36, ***MinimumLength*** ≥ 0

Semantics: Converts number ***X*** into text that represents the value of ***X*** in base ***Radix***. The symbols 0-9 (U+0030 through U+0039), then upper case A-Z (U+0041 through U+005A) are used as digits. Thus, BASE(45745;36) returns “ZAP”.

If ***MinimumLength*** is not supplied, the generated text uses the smallest number of characters (i.e., it does not add leading 0s). If ***MinimumLength*** is supplied, and the resulting text would normally be smaller than ***MinimumLength***, leading 0s are added to produce text exactly ***MinimumLength*** characters long. If the text is longer than the ***MinimumLength*** argument, the ***MinimumLength*** parameter is ignored.

See also DECIMAL 6.19.10

6.19.4 BIN2DEC

Summary: Converts a binary (base 2) number (up to 10 digits) to its decimal equivalent

Syntax: BIN2DEC(*TextOrNumber* ***X***)

Returns: *Number*

Constraints: ***X*** shall contain only binary digits (no space or other characters), and shall contain at least one binary digit. When considered as a Number, INT(***X***) = ***X***. Evaluators may evaluate expressions where the digits in ***X*** are only 0 or 1, no more than 10 digits.

Semantics: Converts given binary number into decimal equivalent, with the topmost 10^{th} digit being the sign bit (using a two's-complement representation). If given Text, the text is considered a binary number representation. If given a Number, the digits of the number when printed as base 10 are considered the digits of the equivalently-represented binary number. It is implementation-defined what happens if given a Logical value; an evaluator may produce an Error, or it may convert the Logical to Number (per “Convert to Number”) and then process as a Number.

If any digits are 2 through 9, an evaluator shall return an Error. It is implementation-defined what happens if an evaluator is given an empty string; evaluators may return an Error or 0 in such cases.

See also INT 6.17.2

6.19.5 BIN2HEX

Summary: Converts a binary (base 2) number (10^{th} bit is sign) to its hexadecimal equivalent

Syntax: BIN2HEX(*TextOrNumber* ***X*** [; *Number* ***Digits***])

Returns: *Text*

Constraints: ***X*** shall contain only binary digits (no space or other characters), and shall contain at least one binary digit. When considered as a Number, INT(***X***) = ***X***. Evaluators may evaluate expressions where the digits in ***X*** are only 0 or 1, no more than 10 digits.

Semantics: Converts given binary number into hexadecimal (base 16) equivalent. For input value ***X***, the topmost 10^{th} digit is considered the sign bit (using a two's-complement representation). If given Text, the text is considered a binary number representation. If given a Number, the digits of the number when printed as base 10 are considered the digits of the equivalently-represented binary number. It is implementation-defined what happens if given a Logical value; an evaluator may produce an Error, or it may convert the Logical to Number (per “Convert to Number”) and then process as a Number.

If any digits in ***X*** are 2 through 9, an evaluator shall return an Error. It is implementation-defined what happens if an evaluator is given an empty string; evaluators may return an Error or 0 in such cases.

The resulting value is a hexadecimal value, up to 10 hexadecimal digits, with the topmost bit (40^{th} bit) being the sign bit and in two's-complement form. The digits A through F are in uppercase. If the input has its 10^{th} bit on, the ***Digits*** argument is ignored; otherwise, the ***Digits*** indicates the number of digits in the output, with leading 0 digits added as necessary to bring it up to that number of digits. If there are more digits required than the ***Digits*** parameter specifies, the results are implementation-defined.

See also INT 6.17.2

6.19.6 BIN2OCT

Summary: Converts a binary (base 2) number (10th bit is sign) to its octal (base 8) equivalent

Syntax: BIN2OCT(*TextOrNumber* ***X*** [; *Number* ***Digits***])

Returns: *Text*

Constraints: ***X*** shall contain only binary digits (no space or other characters), and shall contain at least one binary digit. When considered as a Number, INT(***X***) = ***X***. Evaluators may evaluate expressions where the digits in ***X*** are only 0 or 1, no more than 10 digits.

Semantics: Converts given binary number into octal (base 8) equivalent. For input value ***X***, the topmost 10th digit is considered the sign bit (using a two's-complement representation). If given Text, the text is considered a binary number representation. If given a Number, the digits of the number when printed as base 10 are considered the digits of the equivalently-represented binary number. It is implementation-defined what happens if given a Logical value; an evaluator may produce an Error, or it may convert the Logical to Number (per "Convert to Number") and then process as a Number.

If any digits in ***X*** are 2 through 9, an evaluator shall return an Error. It is implementation-defined what happens if an evaluator is given an empty string; evaluators may return an Error or 0 in such cases.

The resulting value is an octal value, up to 10 octal digits, with the topmost bit (30th bit) being the sign bit and in two's-complement form. If the input has its 10th bit on, the ***Digits*** argument is ignored; otherwise, the ***Digits*** indicates the number of digits in the output, with leading 0 digits added as necessary to bring it up to that number of digits. If there are more digits than specified by the ***Digits*** parameter, its results are implementation-defined.

See also INT 6.17.2

6.19.7 DEC2BIN

Summary: Converts a decimal number to base 2 (whose 10th bit is sign)

Syntax: DEC2BIN(*TextOrNumber* ***X*** [; *Number* ***Digits***])

Returns: *Text*

Constraints: ***X*** shall contain only decimal digits (no space or other characters), and shall contain at least one decimal digit. When considered as a Number, INT(***X***) = ***X***. Evaluators may evaluate expressions where -512 ≤ ***X*** ≤ 511.

Semantics: Converts given number into binary (base 2) equivalent. If given Text, the text is considered a decimal number representation, and may have a leading minus sign. It is implementation-defined what happens if given a Logical value; an evaluator may produce an Error, or it may convert the Logical to Number (per "Convert to Number") and then process as a Number.

The resulting value is a binary value, up to 10 digits, with the topmost bit (10th bit) being the sign bit and in two's-complement form. If the input is negative, the ***Digits*** argument is ignored; otherwise, the ***Digits*** indicates the number of digits in the output, with leading 0 digits added as necessary to bring it up to that number of digits. If there are more digits than specified by the ***Digits*** parameter, the results are implementation-defined.

See also INT 6.17.2

6.19.8 DEC2HEX

Summary: Converts a decimal number to base 16 (whose 40th bit is sign)

Syntax: DEC2HEX(*TextOrNumber* ***X*** [; *Number* ***Digits***])

Returns: *Text*

Constraints: ***X*** shall contain only decimal digits (no space or other characters), and shall contain at least one decimal digit. When considered as a Number, INT(***X***) = ***X***. Evaluators may evaluate expressions where $-2^{39} \leq X \leq 2^{39}-1$.

Semantics: Converts given number into hexadecimal (base 16) equivalent. If given Text, the text is considered a decimal number representation, and may have a leading minus sign. It is implementation-defined what happens if given a Logical value; an evaluator may produce an Error, or it may convert the Logical to Number (per "Convert to Number") and then process as a Number.

The resulting value is a hexadecimal value, up to 10 digits, with the topmost bit (40^{th} bit) being the sign bit and in two's-complement form. If the input is negative, the ***Digits*** argument is ignored; otherwise, the ***Digits*** indicates the number of digits in the output, with leading 0 digits added as necessary to bring it up to that number of digits. If there are more digits than specified by the ***Digits*** parameter, the results are implementation-defined.

See also INT 6.17.2

6.19.9 DEC2OCT

Summary: Converts a decimal number to base 8 (whose 30^{th} bit is sign)

Syntax: DEC2OCT(*TextOrNumber* ***X*** [; *Number* ***Digits***])

Returns: *Text*

Constraints: ***X*** shall contain only decimal digits (no space or other characters), and shall contain at least one decimal digit. When considered as a Number, INT(***X***) = ***X***. Evaluators may evaluate expressions where $-2^{29} \leq X \leq 2^{29}-1$.

Semantics: Converts given number into octal (base 8) equivalent. If given Text, the text is considered a decimal number representation, and may have a leading minus sign. It is implementation-defined what happens if given a Logical value; an evaluator may produce an Error, or it may convert the Logical to Number (per "Convert to Number") and then process as a Number.

The resulting value is a octal value, up to 10 digits, with the topmost bit (30^{th} bit) being the sign bit and in two's-complement form. If the input is negative, the ***Digits*** argument is ignored; otherwise, the ***Digits*** indicates the number of digits in the output, with leading 0 digits added as necessary to bring it up to that number of digits. If there are more digits than specified by the ***Digits*** parameter, the results are implementation-defined.

See also INT 6.17.2, OCT2DEC 6.19.15

6.19.10 DECIMAL

Summary: Converts text representing a number in a given base into a base 10 number.

Syntax: DECIMAL(*Text* ***X*** ; *Integer* ***Radix***)

Returns: *Number*

Constraints: $2 \leq$ ***Radix*** ≤ 36

Semantics: Converts text ***X*** in base ***Radix*** to a Number. Uppercase letters (U+0041 through U+005A) and lowercase letters (U+0061 through U+007A) are both accepted as equivalent if ***Radix*** > 10. Thus, DECIMAL("zap";36) and DECIMAL("ZAP";36) both compute 45745.

An Error is returned if ***X*** has characters that do not belong in base ***Radix***. However, leading spaces and tabs in ***X*** are always ignored. If ***Radix*** is 16, a leading regular expression "0?[Xx]" is ignored, as is a trailing letter H or h. If ***Radix*** is 2, the letter b or B at the end is ignored (if present).

See also BASE 6.19.3

6.19.11 HEX2BIN

Summary: Converts a hexadecimal number (40th bit is sign) to base 2 (whose 10th bit is sign)

Syntax: HEX2BIN(*TextOrNumber* ***X*** [; *Number* ***Digits***])

Returns: *Text*

Constraints: ***X*** shall contain only hexadecimal digits (no space or other characters), and shall contain at least one hexadecimal digit. When considered as a Number, INT(***X***) = ***X***. Evaluators may evaluate expressions where ***X*** is considered in base 10, -512 ≤ ***X*** ≤ 511.

Semantics: Converts given hexadecimal number into binary (base 2) equivalent. If given Text, the text is considered a hexadecimal number representation; if its 40th bit is 1, it is considered a negative number. It is implementation-defined what happens if given a Logical value; an evaluator may produce an Error, or it may convert the Logical to Number (per "Convert to Number") and then process as a Number.

The resulting value is a binary value, up to 10 digits, with the topmost bit (10th bit) being the sign bit and in two's-complement form. If the input is negative (40th bit is 1), the ***Digits*** argument is ignored; otherwise, the ***Digits*** indicates the number of digits in the output, with leading 0 digits added as necessary to bring it up to that number of digits. If there are more digits than specified by the ***Digits*** parameter, the results are implementation-defined.

See also INT 6.17.2

6.19.12 HEX2DEC

Summary: Converts a hexadecimal number (40th bit is sign) to decimal

Syntax: HEX2DEC(*TextOrNumber* ***X***)

Returns: *Number*

Constraints: ***X*** shall contain only hexadecimal digits (no space or other characters), and shall contain at least one hexadecimal digit. When considered as a Number, INT(***X***) = ***X***. Evaluators may evaluate expressions where ***X*** shall have 1 though 10 (inclusive) hexadecimal digits.

Semantics: Converts given hexadecimal number into decimal equivalent. If given Text, the text is considered a hexadecimal number representation. If ***X***'s 40th bit is 1, it is considered a negative number. It is implementation-defined what happens if given a Logical value; an evaluator may produce an Error, or it may convert the Logical to Number (per "Convert to Number") and then process as a Number.

The resulting value is a decimal number.

See also INT 6.17.2

6.19.13 HEX2OCT

Summary: Converts a hexadecimal number (40th bit is sign) to base 8 (whose 30th bit is sign)

Syntax: HEX2OCT(*TextOrNumber* ***X*** [; *Number* ***Digits***])

Returns: *Text*

Constraints: ***X*** shall contain hexadecimal digits (no spaces or other characters), and shall contain at least one hexadecimal digit. When considered as Number, INT(***X***) = ***X***. Evaluators may evaluate expressions where ***X*** has 1 to 10 (inclusive) hexadecimal digits, base 10 value of ***X*** is $-2^{29} < X < 2^{29} - 1$.

Semantics: Converts given hexadecimal number into octal (base 8) equivalent. If given Text, the text is considered a hexadecimal number representation; if its 40th bit is 1, it is considered a negative number. It is implementation-defined what happens if given a Logical value; an evaluator may produce an Error, or it may convert the Logical to Number (per "Convert to Number") and then process as a Number.

The resulting value is an octal value, up to 10 digits, with the topmost bit (10th bit) being the sign bit and in two's-complement form. If the input is negative (40th bit is 1), the ***Digits***

argument is ignored; otherwise, the ***Digits*** indicates the number of digits in the output, with leading 0 digits added as necessary to bring it up to that number of digits. If there are more digits than specified by the ***Digits*** parameter, the results are implementation-defined.

See also INT 6.17.2

6.19.14 OCT2BIN

Summary: Converts an octal number (30th bit is sign) to base 2 (whose 10th bit is sign)

Syntax: OCT2BIN(*TextOrNumber* ***X*** [; *Number* ***Digits***])

Returns: *Text*

Constraints: ***X*** shall contain only octal digits (no space or other characters), and shall contain at least one octal digit. When considered as a Number, INT(***X***) = ***X***. Evaluators may evaluate expressions where ***X*** is considered in base 10, -512 ≤ ***X*** ≤ 511.

Semantics: Converts given octal (base 8) number into binary (base 2) equivalent. If given Text, the text is considered an octal number representation; if its 30th bit is 1, it is considered a negative number. It is implementation-defined what happens if given a Logical value; an evaluator may produce an Error, or it may convert the Logical to Number (per “Convert to Number”) and then process as a Number.

The resulting value is a binary value, up to 10 digits, with the topmost bit (10th bit) being the sign bit and in two's-complement form. If the input is negative (30th bit is 1), the ***Digits*** argument is ignored; otherwise, the ***Digits*** indicates the number of digits in the output, with leading 0 digits added as necessary to bring it up to that number of digits. If there are more digits than specified by the ***Digits*** parameter, the results are implementation-defined.

See also INT 6.17.2

6.19.15 OCT2DEC

Syntax: OCT2DEC(*TextOrNumber* ***X***)

Summary: Converts an octal number (30th bit is sign) to decimal

Returns: *Number*

Constraints: ***X*** shall contain only octal digits (no space or other characters), and shall contain at least one octal digit. When considered as a Number, INT(***X***) = ***X***. Evaluators may evaluate expressions where ***X*** shall have 1 though 10 (inclusive) octal digits.

Semantics: Converts given octal number into decimal equivalent. If given Text, the text is considered a octal number representation. If ***X***'s 30th bit is 1, it is considered a negative number. It is implementation-defined what happens if given a Logical value; an evaluator may produce an Error, or it may convert the Logical to Number (per “Convert to Number”) and then process as a Number.

The resulting value is a decimal number.

See also INT 6.17.2

6.19.16 OCT2HEX

Summary: Converts an octal number (30th bit is sign) to hexadecimal (whose 40th bit is sign)

Syntax: OCT2HEX(*TextOrNumber* ***X*** [; *Number* ***Digits***])

Returns: *Text*

Constraints: ***X*** shall contain only octal digits (no space or other characters), and shall contain at least one octal digit. When considered as a Number, INT(***X***) = ***X***. Evaluators may evaluate expressions where ***X*** shall have 1 to 10 (inclusive) octal digits.

Semantics: Converts given octal (base 8) number into hexadecimal (base 16) equivalent. If given Text, the text is considered an octal number representation; if its 30th bit is 1, it is

considered a negative number. It is implementation-defined what happens if given a Logical value; an evaluator may produce an Error, or it may convert the Logical to Number (per “Convert to Number”) and then process as a Number.

The resulting value is a hexadecimal value, up to 10 digits, with the topmost bit (40th bit) being the sign bit and in two's-complement form. If the input is negative (30th bit is 1), the ***Digits*** argument is ignored; otherwise, the ***Digits*** indicates the number of digits in the output, with leading 0 digits added as necessary to bring it up to that number of digits. If there are more digits than specified by the ***Digits*** parameter, the results are implementation-defined.

See also INT 6.17.2

6.19.17 ROMAN

Summary: Convert to Roman numerals

Syntax: ROMAN(*Integer* ***N*** [; *Integer* ***Format*** = 0])

Returns: *Text*

Constraints: ***N*** ≥ 0, ***N*** < 4000, 0 ≤ ***Format*** ≤ 4, ISLOGICAL(1) or NOT(ISLOGICAL(***Format***))

Semantics: Return the Roman numeral representation of ***N***. ***Format*** specifies the level of conciseness, and defaults to 0, the classic representation, with larger numbers requiring increasing conciseness.

To support legacy documents, evaluators with Logical types that are distinct from Number may accept the format parameter as a scalar, and accept TRUE specifying format 0, and FALSE specifying format 4.

The following identity shall hold: ARABIC(ROMAN(x; any)) = x, when ROMAN(x; any) is not an Error.

If ***N*** is 0, an empty string is returned.

Table 31 - ROMAN lists the values of individual roman numerals; roman numerals that precede (directly or indirectly) a larger-valued roman number *subtract* their value from the final value.

Table 31 - ROMAN

Roman Numeral	***Value***	***Unicode Code Point***
I	1	U+0049
V	5	U+0056
X	10	U+0058
L	50	U+004C
C	100	U+0043
D	500	U+0044
M	1000	U+004D

Evaluators that accept 0 as a value of ***N*** should return the string “0”. Evaluators that accept negative values of ***N*** should include a negative sign (“-”) as the first character.

The ***Format*** levels are:

Table 32 - ROMAN

Format	***Meaning***
0 or omitted (or TRUE)	Only subtract powers of 10, not L or V, and only if the next number is not more than 10 times greater. A number following the larger one shall be smaller than the subtracted number. Also known as *classic*.
1	Powers of 10, and L and V may be subtracted, only if the next number is not more than 10 times greater. A number following the larger one shall be smaller than the subtracted number.
2	Powers of 10 and L, but not V, may be subtracted, also if the next number is more than 10 times greater. A number following the larger one shall be smaller than the subtracted number.
3	Powers of 10, and L and V may be subtracted, also if the next number is more than 10 times greater. A number following the larger one shall be smaller than the subtracted number.
4 (or FALSE)	Produce the fewest Roman digits possible. Also known as *simplified*.

See also Infix Operator "&" 6.4.10, ISLOGICAL 6.13.19, ARABIC 6.19.2

6.20 Text Functions

6.20.1 General

6.20.2 ASC

Summary: Converts full-width to half-width ASCII and katakana characters.

Syntax: ASC(*Text* ***T***)

Returns: *Text*

Constraints: None

Semantics: Conversion is done for full-width ASCII and [UNICODE] katakana characters, some characters are converted in a special way, see table below. Other characters are copied from ***T*** to the result. This is the complementary function to JIS.

The percent sign % in the conversion table below denotes the modulo operation. A *followed by* means that a character is converted to two consecutive characters.

Table 33 - ASC

From Unicode Character (c)	***To Unicode Character***	***Comment***
U+30a1 ≤ c ≤ U+30aa if c%2==0	(c - 0x30a2) / 2 + 0xff71	katakana a-o
U+30a1 ≤ c ≤ U+30aa if c%2==1	(c - 0x30a1) / 2 + 0xff67	katakana small a-o
U+30ab ≤ c ≤ U+30c2 if c%2==1	(c - 0x30ab) / 2 + 0xff76	katakana ka-chi

From Unicode Character (c)	*To Unicode Character*	*Comment*
U+30ab ≤ c ≤ U+30c2 if c%2==0	(c - 0x30ac) / 2 + 0xff76 followed by 0xff9e	katakana ga-dhi
U+30c3	0xff6f	katakana small tsu
U+30c4 ≤ c ≤ U+30c9 if c%2==0	(c - 0x30c4) / 2 + 0xff82	katakana tsu-to
U+30c4 ≤ c ≤ U+30c9 if c%2==1	(c - 0x30c5) / 2 + 0xff82 followed by 0xff9e	katakana du-do
U+30ca ≤ c ≤ U+30ce	c - 0x30ca + 0xff85	katakana na-no
U+30cf ≤ c ≤ U+30dd if c%3==0	(c - 0x30cf) / 3 + 0xff8a	katakana ha-ho
U+30cf ≤ c ≤ U+30dd if c%3==1	(c - 0x30d0) / 3 + 0xff8a followed by 0xff9e	katakana ba-bo
U+30cf ≤ c ≤ U+30dd if c%3==2	(c - 0x30d1) / 3 + 0xff8a followed by 0xff9f	katakana pa-po
U+30de ≤ c ≤ U+30e2	c - 0x30de + 0xff8f	katakana ma-mo
U+30e3 ≤ c ≤ U+30e8 if c%2==0	(c - 0x30e4) / 2 + 0xff94)	katakana ya-yo
U+30e3 ≤ c ≤ U+30e8 if c%2==1	(c - 0x30e3) / 2 + 0xff6c	katakana small ya-yo
U+30e9 ≤ c ≤ U+30ed	c - 0x30e9 + 0xff97	katakana ra-ro
U+30ef	U+ff9c	katakana wa
U+30f2	U+ff66	katakana wo
U+30f3	U+ff9d	katakana nn
U+ff01 <= c <= U+ff5e	c - 0xff01 + 0x0021	ASCII characters
U+2015	U+ff70	HORIZONTAL BAR => HALFWIDTH KATAKANA-HI-RAGANA PROLONGED SOUND MARK
U+2018	U+0060	LEFT SINGLE QUOTATION MARK => GRAVE ACCENT
U+2019	U+0027	RIGHT SINGLE QUOTATION MARK => APOSTROPHE
U+201d	U+0022	RIGHT DOUBLE QUOTA-TION MARK => QUOTATION MARK
U+3001	U+ff64	IDEOGRAPHIC COMMA
U+3002	U+ff61	IDEOGRAPHIC FULL STOP
U+300c	U+ff62	LEFT CORNER BRACKET
U+300d	U+ff63	RIGHT CORNER BRACKET

From Unicode Character (c)	*To Unicode Character*	*Comment*
U+309b	U+ff9e	KATAKANA-HIRAGANA VOICED SOUND MARK
U+309c	U+ff9f	KATAKANA-HIRAGANA SEMI-VOICED SOUND MARK
U+30fb	U+ff65	KATAKANA MIDDLE DOT
U+30fc	U+ff70	KATAKANA-HIRAGANA PROLONGED SOUND MARK
U+ffe5	U+005c	FULLWIDTH YEN SIGN => REVERSE SOLIDUS "\"
Note 1: The "\" (REVERSE SOLIDUS, U+005C) is a specialty that gets displayed as a Yen sign with some Japanese fonts, which is a legacy of code-page 932.		

Note 2: For references regarding halfwidth and fullwidth characters see [UAX11] and the Halfwidth and Fullwidth Code Chart of [UNICODE].

Note 3: For information about the mapping of JIS X 0201 and JIS X 0208 to Unicode characters see [JISX0201] and [JISX0208].

See also JIS 6.20.11

6.20.3 CHAR

Summary: Return character represented by the given numeric value

Syntax: CHAR(*Number* ***N***)

Returns: *Text*

Constraints: ***N*** ≤ 127; Evaluators may evaluate expressions where ***N*** ≥ 1, ***N*** ≤ 255.

Semantics:

Returns character represented by the given numeric value.

Evaluators should return an Error if ***N*** > 255.

Evaluators should implement CHAR such that CODE(CHAR(***N***)) returns ***N*** for any 1 ≤ ***N*** ≤ 255.

Note 1: Beyond 127, some evaluators return a character from a system-specific code page, while others return the [UNICODE] character. Most evaluators do not allow values greater than 255.

Note 2: Where interoperability is a concern, expressions should use the UNICHAR function. 6.20.25

See also CODE 6.20.5, UNICHAR 6.20.25, UNICODE 6.20.26

6.20.4 CLEAN

Summary: Remove all non-printable characters from the string and return the result.

Syntax: CLEAN(*Text* ***T***)

Returns: *Text*

Semantics:

Removes all non-printable characters from the string ***T*** and returns the resulting string. Evaluators should remove each particular character from the string, if and only if the character belongs to [UNICODE] class Cc (Other - Control), or to Unicode class Cn (Other - Not Assigned). The resulting string shall contain all printable characters from the original string, in the same order. The space character is considered a printable character.

6.20.5 CODE

Summary: Return numeric value corresponding to the first character of the text value.

Syntax: CODE(*Text* ***T***)

Returns: *Number*

Constraints: code point ≤ 127. Evaluators may evaluate expressions where Length(***T***) > 0.

Semantics:

Returns a numeric value which represents the first letter of the given text ***T***.

Behavior for code points ≥ 128 is implementation-defined. Evaluators may use the underlying system's code page. Evaluators should implement CODE such that CODE(CHAR(N)) returns N for 1 ≤ N ≤ 255.

> **Note:** Where interoperability is a concern, expressions should use the UNICODE function. 6.20.26

See also CHAR 6.20.3, UNICHAR 6.20.25, UNICODE 6.20.26

6.20.6 CONCATENATE

Summary: Concatenate the text strings

Syntax: CONCATENATE({ *Text* ***T*** }$^{+}$)

Returns: *Text*

Constraints: None

Semantics: Concatenate each text value, in order, into a single text result.

See also Infix Operator "&" 6.4.10

6.20.7 DOLLAR

Summary: Convert the parameters to Text formatted as currency.

Syntax: DOLLAR(*Number* ***N*** [; *Integer* ***D***])

Returns: *Text*

Constraints: None

Semantics: Returns the value formatted as a currency, using locale-specific data. ***D*** is the number of decimal places used in the result string, a negative ***D*** rounds number ***N***. If ***D*** is omitted, locale information may be used to determine the currency's decimal places, or a value of 2 shall be assumed.

6.20.8 EXACT

Summary: Report if two text values are equal using a case-sensitive comparison .

Syntax: EXACT(*Text* ***T1*** ; *Text* ***T2***)

Returns: *Logical*

Constraints: None

Semantics: Converts both sides to Text, and then returns TRUE if the two text values are equal, including case, otherwise it returns FALSE.

See also FIND 6.20.9, SEARCH 6.20.20, Infix Operator "<>" 6.4.8, Infix Operator "=" 6.4.7

6.20.9 FIND

Summary: Return the starting position of a given text.

Syntax: FIND(*Text* ***Search*** ; *Text* ***T*** [; *Integer* ***Start*** = 1])

Returns: *Number*

Constraints: ***Start*** ≥ 1

Semantics: Returns the character position where ***Search*** is first found in ***T***, when the search is started from character position ***Start***. The match is case-sensitive, and no wildcards or other instructions are considered in ***Search***. Returns an Error if text not found.

See also EXACT 6.20.8, SEARCH 6.20.20

6.20.10 FIXED

Summary: Round the number to a specified number of decimals and format the result as a text.

Syntax: FIXED(*Number* ***N*** [; *Integer* ***D*** = 2 [; *Logical* ***OmitSeparators*** = FALSE]])

Returns: *Text*

Constraints: None

Semantics: Rounds value ***N*** to ***D*** decimal places (after the decimal point) and returns the result formatted as text, using locale-specific settings. If ***D*** is negative, the number is rounded to ABS(***D***) places to the left from the decimal point. If the optional parameter ***OmitSeparators*** is TRUE, then group separators are omitted from the resulting string. Group separators are included in the absence of this parameter. If ***D*** is a fraction, it is rounded towards 0 as an integer (ignoring what is the closest integer).

See also ABS 6.16.2

6.20.11 JIS

Summary: Converts half-width to full-width ASCII and katakana characters.

Syntax: JIS(*Text* ***T***)

Returns: *Text*

Constraints: None

Semantics: Conversion is done for half-width ASCII and [UNICODE] katakana characters, some characters are converted in a special way, see table below. Other characters are copied from ***T*** to the result. This is the complementary function to ASC.

A *followed by* means that there are two consecutive characters to convert from.

Table 34 - JIS

From Unicode Character (c)	***To Unicode Character***	***Comment***
U+0022	0x201d	QUOTATION MARK => RIGHT DOUBLE QUOTATION MARK This is an exception to the ASCII range that follows below.
U+005c	0xffe5	REVERSE SOLIDUS "\" => FULLWIDTH YEN SIGN

From Unicode Character (c)	*To Unicode Character*	*Comment*
		(code-page 932 legacy, for details see ASC function) This is an exception to the ASCII range that follows below.
U+0060	0x2018	GRAVE ACCENT => LEFT SINGLE QUOTATION MARK This is an exception to the ASCII range that follows below.
U+0027	0x2019	APOSTROPHE => RIGHT SINGLE QUOTATION MARK This is an exception to the ASCII range that follows below.
U+0021 ≤ c ≤= U+007e	c - 0x0021 + 0xff01	ASCII characters
U+ff66	0x30f2	katakana wo
U+ff67 ≤ c ≤ U+ff6b	(c - 0xff67) * 2 + 0x30a1	katakana small a-o
U+ff6c ≤ c ≤ U+ff6e	(c - 0xff6c) * 2 + 0x30e3	katakana small ya-yo
U+ff6f	0x30c3	katakana small tsu
U+ff71 ≤ c ≤ U+ff75	(c - 0xff71) * 2 + 0x30a2	katakana a-o
U+ff76 ≤ c ≤ U+ff81 followed by U+ff9e	(c - 0xff76) * 2 + 0x30ac	katakana ga-dsu
U+ff76 ≤ c ≤ U+ff81 not followed by U+ff9e	(c - 0xff76) * 2 + 0x30ab	katakana ka-chi
U+ff82 ≤ c ≤ U+ff84 followed by U+ff9e	(c - 0xff82) * 2 + 0x30c5	katakana du-do
U+ff82 ≤ c ≤ U+ff84 not followed by U+ff9e	(c - 0xff82) * 2 + 0x30c4	katakana tsu-to
U+ff85 ≤ c ≤ U+ff89	c - 0xff85 + 0x30ca	katakana na-no
U+ff8a ≤ c ≤ U+ff8e followed by U+ff9e	(c - 0xff8a) * 3 + 0x30d0	katakana ba-bo
U+ff8a ≤ c ≤ U+ff8e followed by U+ff9f	(c - 0xff8a) * 3 + 0x30d1	katakana pa-po
U+ff8a ≤ c ≤ U+ff8e neither followed by U+ff9e nor U+ff9f	(c - 0xff8a) * 3 + 0x30cf	katakana ha-ho
U+ff8f ≤ c ≤ U+ff93	c - 0xff8f + 0x30de	katakana ma-mo
U+ff94 ≤ c ≤ U+ff96	(c - 0xff94) * 2 + 0x30e4	katakana ya-yo
U+ff97 ≤ c ≤ U+ff9b	c - 0xff97 + 0x30e9	katakana ra-ro
U+ff9c	U+30ef	katakana wa

From Unicode Character (c)	*To Unicode Character*	*Comment*
U+ff9d	U+30f3	katakana nn
U+ff9e	U+309b	HALFWIDTH KATAKANA VOICED SOUND MARK => FULLWIDTH
U+ff9f	U+309c	HALFWIDTH KATAKANA SEMI-VOICED SOUND MARK => FULLWIDTH
U+ff70	U+30fc	HALFWIDTH KATAKANA-HIRAGANA PROLONGED SOUND MARK => FULLWIDTH
U+ff61	U+3002	HALFWIDTH IDEOGRAPHIC FULL STOP => FULLWIDTH
U+ff62	U+300c	HALFWIDTH LEFT CORNER BRACKET => FULLWIDTH
U+ff63	U+300d	HALFWIDTH RIGHT CORNER BRACKET => FULLWIDTH
U+ff64	U+3001	HALFWIDTH IDEOGRAPHIC COMMA => FULLWIDTH
U+ff65	U+30fb	HALFWIDTH KATAKANA MIDDLE DOT => FULLWIDTH

Note 1: For references regarding halfwidth and fullwidth characters see [UAX11] and the Halfwidth and Fullwidth Code Chart of [UNICODE].

Note 2: For information about the mapping of JIS X 0201 and JIS X 0208 to Unicode characters see [JISX0201] and [JISX0208].

See also ASC 6.20.2

6.20.12 LEFT

Summary: Return a selected number of text characters from the left.

Syntax: LEFT(*Text* ***T*** [; *Integer* ***Length***])

Returns: *Text*

Constraints: ***Length*** ≥ 0

Semantics: Returns the INT(***Length***) number of characters of text ***T***, starting from the left. If ***Length*** is omitted, it defaults to 1; otherwise, it computes ***Length*** = INT(***Length***). If ***T*** has fewer than ***Length*** characters, it returns ***T***. This means that if ***T*** is an empty string (which has length 0) or the parameter ***Length*** is 0, LEFT() will always return an empty string. Note that if ***Length*** < 0, an Error is returned. This function shall return the same string as MID(***T***; 1; ***Length***).

The results of this function may be normalization-sensitive. 4.2

See also INT 6.17.2, LEN 6.20.13, MID 6.20.15, RIGHT 6.20.19

6.20.13 LEN

Summary: Return the length, in characters, of given text

Syntax: LEN(*Text* ***T***)

Returns: *Integer*

Constraints: None.

Semantics: Computes number of characters (*not* the number of bytes) in ***T***. If ***T*** is of type Number, it is automatically converted to Text, including a fractional part and decimal separator if necessary.

The results of this function may be normalization-sensitive. 4.2

See also TEXT 6.20.23, ISTEXT 6.13.25, LEFT 6.20.12, MID 6.20.15, RIGHT 6.20.19

6.20.14 LOWER

Summary: Return input string, but with all uppercase letters converted to lowercase letters.

Syntax: LOWER(*Text* ***T***)

Returns: *Text*

Constraints: None

Semantics: Return input string, but with all uppercase letters converted to lowercase letters, as defined by §3.13 Default Case Algorithms, §4.2 Case-Normative and §5.18 Case Mappings of [UNICODE]. As with most functions, it is side-effect free (it does *not* modify the source values). All Evaluators shall convert A-Z to a-z.

Note: As this function can be locale aware, results may be unexpected in certain cases. For example in a Turkish locale an upper case "I without dot" (LATIN CAPITAL LETTER I, U+0049) is converted to a lower case "i without dot" (LATIN SMALL LETTER DOTLESS I, U+0131).

See also UPPER 6.20.27, PROPER 6.20.16

6.20.15 MID

Summary: Returns extracted text, given an original text, starting position, and length.

Syntax: MID(*Text* ***T*** ; *Integer* ***Start*** ; *Integer* ***Length***)

Returns: *Text*

Constraints: ***Start*** ≥ 1, ***Length*** ≥ 0.

Semantics: Returns the characters from ***T***, starting at character position ***Start***, for up to ***Length*** characters. For the integer conversions, ***Start*** = INT(***Start***), and ***Length*** = INT(***Length***). If there are less than ***Length*** characters starting at start, it returns as many characters as it can beginning with ***Start***. In particular, if ***Start*** > LEN(***T***), it returns the empty string (""). If ***Start*** < 0, it returns an Error. If ***Start*** ≥ 0, and ***Length*** = 0, it returns the empty string. Note that MID(***T***;1;***Length***) produces the same results as LEFT(***T***;***Length***).

The results of this function may be normalization-sensitive. 4.2

See also INT 6.17.2, LEFT 6.20.12, LEN 6.20.13, RIGHT 6.20.19, REPLACE 6.20.17, SUBSTITUTE 6.20.21

6.20.16 PROPER

Summary: Return the input string with the first letter of each word converted to an uppercase letter and the rest of the letters in the word converted to lowercase.

Syntax: PROPER(*Text* ***T***)

Returns: *Text*

Constraints: None

Semantics: Return input string, but modified as follows:

- If the first character is a letter, it is converted to its uppercase equivalent; otherwise, the original character is returned
- If a letter is preceded by a non-letter, it is converted to its uppercase equivalent
- If a letter is preceded by a letter, it is converted to its lowercase equivalent.

Evaluators shall implement this for at least the Latin letters A-Z and a-z.

As with most functions, it is side-effect free, that is, it does *not* modify the source values.

See also LOWER 6.20.14, UPPER 6.20.27

6.20.17 REPLACE

Summary: Returns text where an old text is substituted with a new text.

Syntax: REPLACE(*Text* ***T*** ; *Number* ***Start*** ; *Number* ***Count*** ; *Text* ***New***)

Returns: *Text*

Constraints: ***Start*** ≥ 1.

Semantics: Returns text ***T***, but remove the characters starting at character position ***Start*** for ***Count*** characters, and instead replace them with ***New***. Character positions defined by ***Start*** begin at 1 (for the leftmost character). If ***Count***=0, the text ***New*** is inserted before character position ***Start***, and all the text before and after ***Start*** is retained. If ***Start*** > length of text ***T*** (TLen) then ***Start*** is set to TLen. If ***Count*** > TLen - ***Start*** then ***Count*** is set to TLen - ***Start***.

REPLACE(***T***;***Start***;Len;***New***) is the same as LEFT(***T***;***Start*** - 1) & ***New*** & MID(***T***; ***Start*** + Len; LEN(***T***)))

See also LEFT 6.20.12, LEN 6.20.13, MID 6.20.15, RIGHT 6.20.19, SUBSTITUTE 6.20.21

6.20.18 REPT

Summary: Return text repeated ***Count*** times.

Syntax: ***T***(*Text* ***T*** ; *Integer* ***Count***)

Returns: *Text*

Constraints: ***Count*** ≥ 0

Semantics: Returns text ***T*** repeated ***Count*** number of times; if ***Count*** is zero, an empty string is returned. If ***Count*** < 0, the result is Error.

See also LEFT 6.20.12, MID 6.20.15, RIGHT 6.20.19, SUBSTITUTE 6.20.21

6.20.19 RIGHT

Summary: Return a selected number of text characters from the right.

Syntax: RIGHT(*Text* ***T*** [; *Integer* ***Length***])

Returns: *Text*

Constraints: ***Length*** ≥ 0

Semantics: Returns the ***Length*** number of characters of text ***T***, starting from the right. If ***Length*** is omitted, it defaults to 1; otherwise, it computes ***Length*** = INT(***Length***). If ***T*** has fewer than ***Length*** characters, it returns ***T*** (unchanged). This means that if ***T*** is an empty string (which has length 0) or the parameter ***Length*** is 0, RIGHT() will always return an empty string. Note that if ***Length*** < 0, an Error is returned.

The results of this function may be normalization-sensitive. 4.2

See also INT 6.17.2, LEFT 6.20.12, LEN 6.20.13, MID 6.20.15

6.20.20 SEARCH

Summary: Return the starting position of a given text.

Syntax: SEARCH(*Text* ***Search*** ; *Text* ***T*** [; *Integer* ***Start*** = 1])

Returns: *Integer*

Constraints: ***Start*** ≥ 1

Semantics: Returns the character position where ***Search*** is first found in ***T***, when the search is started from character position ***Start***. The match is **not** case-sensitive. ***Start*** is 1 if omitted. Returns an Error if text not found.

The values returned may vary depending upon the HOST-USE-REGULAR-EXPRESSIONS or HOST-USE-WILDCARDS properties. 3.4

See also EXACT 6.20.8, FIND 6.20.9

6.20.21 SUBSTITUTE

Summary: Returns text where an old text is substituted with a new text.

Syntax: SUBSTITUTE(*Text* ***T*** ; *Text* ***Old*** ; *Text* ***New*** [; *Integer* ***Which***])

Returns: *Text*

Constraints: ***Which*** ≥ 1 (when provided)

Semantics: Returns text ***T***, but with text ***Old*** replaced by text ***New*** (when searching from the left). If ***Which*** is omitted, every occurrence of ***Old*** is replaced with ***New***; if ***Which*** is provided, only that occurrence of ***Old*** is replaced by ***New*** (starting the count from 1). If there is no match, or if ***Old*** has length 0, the value of ***T*** is returned. Note that ***Old*** and ***New*** may have different lengths. If ***Which*** is present and ***Which*** < 1, returns Error.

See also LEFT 6.20.12, LEN 6.20.13, MID 6.20.15, REPLACE 6.20.17, RIGHT 6.20.19

6.20.22 T

Summary: Return the text (if Text), else return 0-length Text value

Syntax: T(*Any* ***X***)

Returns: *Text*

Constraints: None

Semantics: The type of (a dereferenced) ***X*** is examined; if it is of type Text, it is returned, else an empty string (Text value of zero length) is returned. This is not a type-conversion function; T(5) produces an empty string, not "5".

See also N 6.13.26

6.20.23 TEXT

Summary: Return the value converted to a text.

Syntax: TEXT(*Scalar* ***X*** ; *Text* ***FormatCode***)

Returns: *Text*

Constraints: The ***FormatCode*** is a sequence of characters with an implementation-defined meaning.

Semantics: Converts the value ***X*** to a Text according to the rules of a number format code passed as ***FormatCode*** and returns it.

See also N 6.13.26, T 6.20.22

6.20.24 TRIM

Summary: Remove leading and trailing spaces, and replace all internal multiple spaces with a single space.

Syntax: TRIM(*Text* ***T***)

Returns: *Text*

Constraints: None.

Semantics: Takes ***T*** and removes all leading and trailing space. Any other sequence of 2 or more spaces is replaced with a single space.

A space is one or more, HORIZONTAL TABULATION (U+0009), LINE FEED (U+000A), CARRIAGE RETURN (U+000D) or SPACE (U+0020) characters.

See also LEFT 6.20.12, RIGHT 6.20.19

6.20.25 UNICHAR

Summary: Return the character represented by the given numeric value according to the [UNICODE] Standard.

Syntax: UNICHAR(*Integer* ***N***)

Returns: *Text*

Constraints: ***N*** ≥ 0, ***N*** ≤ 1114111 (U+10FFFF)

Semantics: Returns the character having the given numeric value as [UNICODE] code point. Evaluators shall support values between 1 and 0xFFFF. Evaluators should allow ***N*** to be any [UNICODE] code point of type Graphic, Format or Control. Evaluators should implement UNICHAR such that UNICODE(UNICHAR(***N***)) returns ***N*** for any [UNICODE] code point ***N*** of type Graphic, Format or Control.

See also UNICODE 6.20.26

6.20.26 UNICODE

Summary: Return the [UNICODE] code point corresponding to the first character of the text value.

Syntax: UNICODE(*Text* ***T***)

Returns: *Number*

Constraints: Length(***T***) > 0.

Semantics: Returns the numeric value of the [UNICODE] code point of the first character of the given text ***T***.

The results of this function may be normalization-sensitive. 4.2

See also UNICHAR 6.20.25

6.20.27 UPPER

Summary: Return input string, but with all lowercase letters converted to uppercase letters.

Syntax: UPPER(*Text* ***T***)

Returns: *Text*

Constraints: None

Semantics: Return input string, but with all lowercase letters converted to uppercase letters, as defined by §3.13 Default Case Algorithms, §4.2 Case-Normative and §5.18 Case Mappings of [UNICODE]. As with most functions, it is side-effect free (it does *not* modify the source values). All Evaluators shall convert a-z to A-Z.

Note: As this function can be locale aware, results may be unexpected in certain cases, for example in a Turkish locale a lower case "i with dot" (LATIN SMALL LETTER I) U+0069 is converted to an upper case "I with dot" (LATIN CAPITAL LETTER I WITH DOT ABOVE, U+0130).

See also LOWER 6.20.14, PROPER 6.20.16

7 Other Capabilities

7.1 General

Evaluators may implement capabilities in addition to the functions they support. The following sections describe some specific additional capabilities; evaluators may implement them, and documents may require them (though such documents need not be correctly recalculated on applications which do not implement them). Documents that depend on these other capabilities can still be considered “portable documents”, but only if these additional capabilities are clearly noted (since not all applications implement these additional capabilities).

7.2 Inline constant arrays

Evaluators claiming to implement “Inline constant arrays” shall support inline arrays with one matrix, with one or more rows, and one or more columns. Such evaluators shall support these 2-dimensional arrays as long as the number of expressions in each row is identical; evaluators may but need not support arrays with a different number of expressions in each row. They shall support *at least* the following syntactic rules in the Expression values for the inline array:

- Number, optionally preceded with the prefix “-” operator (for negative numbers)
- Text
- Logical constants TRUE and FALSE
- Error

7.3 Inline non-constant arrays

Evaluators claiming to implement “Inline non-constant arrays” shall support the full Expression syntax in each component of an array (and not just constants).

7.4 Year 1583

Evaluators claiming to implement “Year 1583” can calculate dates correctly starting from the January 1 of the (ISO) year 1583. This means that the evaluator correctly determines that 1900 was not a leap year, and can handle year values for dates back to *at least* 1583.

These calculations use the ISO (Gregorian) calendar, that is, the calculations use the usual rules for the ISO (Gregorian) calendar, regardless of locale. This calendar began official use in some locales in 1582, but other locales used other calendars (such as the Julian calendar) and switched to the Gregorian calendar at different times in history, if they switched at all. Evaluators may choose to support dates even earlier than this; such evaluators may use a proleptic Gregorian system (continuing the dates backwards as if the calendar existed in those dates), but the use of such a system is implementation-defined. Note that the ISO (Gregorian) calendar has never been and is still not currently in universal use .

Correct date calculations in this calendar system require that leap years be handled correctly. In this calendar system, leap years include 29 days in February (which otherwise has 28 days), for 366 total days in a leap year. In general, all years evenly divisible by 4 are leap years. However, years that are divisible by 100 shall also be divisible by 400 to be a leap year; otherwise, they are common (non-leap) years.

8 Non-portable Features

8.1 General

Expressions may depend upon features that are not implemented by all evaluators. This section identifies and defines some features not commonly implemented to enable expressions to indicate their reliance on these features.

8.2 Distinct Logical

Applications may have a Logical type distinct from both Number and Text (see 4.5 Logical (Boolean)), but Logical values may also be represented by the Number type using the values 1 (True) and 0 (False) (see 4.3.7 Logical (Number)).

In this standard, TRUE represents the logical value (true) and FALSE represents the logical value (false), independent of the concrete representation used by an evaluator.

For an evaluator that represents Logical values using the Number type with values 1 (TRUE) and 0 (FALSE): the implicit conversion operator “Convert to Logical” Error: Reference source not found, when a Number is passed as a condition, 0 is considered FALSE and all other numeric values are considered TRUE.

An evaluator that has a Logical type distinct from both Number and Text shall have the following properties:

- ISNUMBER() applied to a Logical value (constant or calculated) will return FALSE, and ISLOGICAL() applied to a Number will be FALSE, either directly or via a reference.
- TRUE will not be equal to 1, and FALSE will not be equal to 0, when they are compared using “=”,
- In a NumberSequence (such as when using SUM), Logical values are skipped when inside a range, but are included and automatically converted to a Number if provided as the NumberSequence itself.

Appendix A Changes from “Open Document Format for Office Applications (OpenDocument) v1.3”

This appendix describes changes that are related to Part 4 of this specification.

The following is a list of major features that have been added or changed. For minor features please see the lists of new and changed elements and attributes.

The following descriptions have changed:

Year 1583 7.4 Office-4112

The following function is new:

EASTERSUNDAY 6.10.8 Office-2164

The definitions of the following functions changed:

CONVERT 6.16.18 (Table 24) Office-3851

COUNTA 6.13.7 Office-4113

INDEX 6.14.6 Office-4143

ISBLANK 6.13.14 Office-4001

ISFORMULA 6.13.18 Office-4001

ISLOGICAL 6.13.19 Office-4001

ISNONTEXT 6.13.21 Office-4001

ISNUMBER 6.13.22 Office-4001

ISREF 6.13.24 Office-4001

ISTEXT 6.13.25 Office-4001

NPER 6.12.29 Office-3915

PMT 6.12.36 Office-3915

