


Adobe XML Architecture

FormCalc Specification

Version 2.0

Adobe Systems Incorporated

October 2003

This page is left intentionally blank.

Intellectual Property

Adobe will enforce its intellectual property rights. Adobe's intention is to maintain the integrity of the Adobe XML Architecture standard. This enables the public to distinguish between the Adobe XML Architecture and other interchange formats for electronic documents, transactions and information. However, Adobe desires to promote the use of the Adobe XML Architecture for information interchange among diverse products and applications. Accordingly, Adobe gives anyone permission to use Adobe's intellectual property, subject to the conditions stated below, to:

- Prepare files whose content conforms to the Adobe XML Architecture
- Write drivers and applications that produce output represented in the Adobe XML Architecture
- Write software that accepts input in the form of the Adobe XML Architecture specifications and displays, prints, or otherwise interprets the contents
- Copy Adobe's intellectual property, as well as the example code to the extent necessary to use the Adobe XML Architecture for the purposes above

The condition of such intellectual property usage is:

- Anyone who uses the Adobe intellectual property, as stated above, must include the appropriate intellectual property and patent notices.

This page is left intentionally blank.

Table of Contents

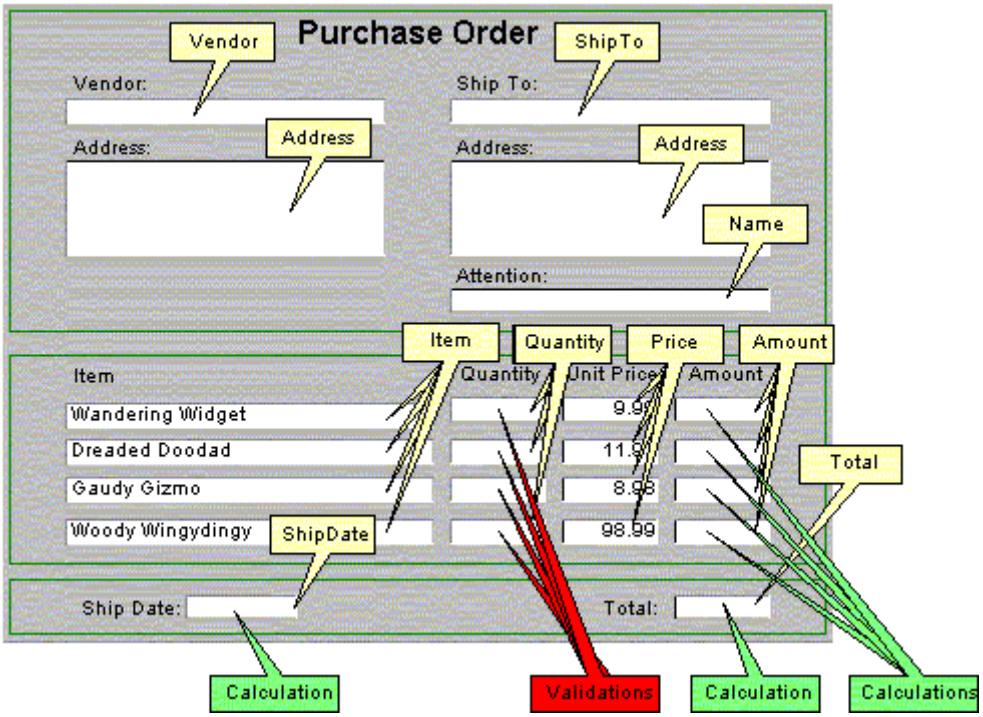
1 Language Overview

FormCalc is a simple calculation language whose roots lie in electronic form software from Adobe, and common spreadsheet software. It is an expression-based language. It is also a type less language, where values of type string or type number can be promoted to strings, numbers or booleans to suit the context.

FormCalc is tailored to the skills of the nonprogrammer who is comfortable with spreadsheet-class application software. This user can, with the addition of a few expressions, validate user input and/or unburden the form user from the spreadsheet-like calculations.

To that aim, the language provides a large set of built-in functions to perform arithmetic, and financial tasks. Locale-sensitive date and time functions are provided, as are string manipulation functions.

To better illustrate the capabilities of the **FormCalc** language, we present a simple purchase order application, and focus on those spreadsheet like calculations and validations typically required of such forms.



Down-pointing call-outs indicate all the field names on this form. In the tabular area of the form are four fields called Item, four fields called Quantity, four fields called Price, and four fields called Amount. We will focus on these shortly.

Green up-pointing call-outs indicate fields with embedded calculations, and the red up-pointing call-outs indicate fields with embedded validations.

A subset of the XFA template syntax used to define this purchase order form might be as follows:

```
<xfa>
  <template name="FormCalc Example">
    <subform name="PO">
      <subform name="Table">
        <field name="Item"> ... </field>
        <field name="Quantity">
          <validate>
```

```
        <script>Within($, 0, 19)</script>
      </validate>
    </field>
    <field name="Price"> ... </field>
    <field name="Amount">
      <calculate>
        <script>Quantity * Price</script>
      </calculate>
    </field>
    <field name="Item"> ... </field>
    <field name="Quantity">
      <validate>
        <script>Within($, 0, 19)</script>
      </validate>
    </field>
    <field name="Price"> ... </field>
    <field name="Amount">
      <calculate>
        <script>Quantity * Price</script>
      </calculate>
    </field>
    <field name="Item"> ... </field>
    <field name="Quantity">
      <validate>
        <script>Within($, 0, 19)</script>
      </validate>
    </field>
    <field name="Price"> ... </field>
    <field name="Amount">
      <calculate>
        <script>Quantity * Price</script>
      </calculate>
    </field>
    <field name="Item"> ... </field>
    <field name="Quantity">
      <validate>
       <script>Within($, 0, 19)</script>
      </validate>
    </field>
    <field name="Price"> ... </field>
    <field name="Amount">
      <calculate>
      <script>Quantity * Price</script>
      </calculate>
    </field>
  </subform>
  <subform Name="Summary" ...>
    <field name="ShipDate">
      <calculate>
      <script>Num2Date(Date() + 2, DateFmt())</script>
      </calculate>
    </field>
    <field name="Total">
      <calculate>
      <script>Str(Sum(Amount[*]), 10, 2)</script>
      </calculate>
    </field>
  </subform>
</subform>
```

```
      </template>
    </xfa>
```
Focusing our attention on the contents of the <script> elements, we see text like

```
Within($, 0, 19)
Quantity * Price
Num2Date(Date() + 2, DateFmt())
```

and

```
Str(Sum(Amount[*]), 10, 2)
```

all of which are real-world examples of form calculations.

The first of these ensures that the entered field value is within the range of 0 to 19. The second computes the product of two fields, while the third simply displays a date that is two days hence from the current date. Some of these expressions are continually being re-executed as the user interacts with the form and enters new data.

On each of the four Quantity fields is the validation:

```
Within($, 0, 19)
```

This is used to limit the user's input to between 0 and 19 items. Any other value entered in these fields will cause a validation error, requiring the user to modify his input. Here the symbol $ is an identifier that refers to the value of the field to which this form calculation is bound; in this case, the Quantity field.

On each of the four Amount fields is the calculation:

```
Quantity * Price
```

which multiplies the value of the Quantity field by the value of the Price field on that row, and stores the resulting product in the Amount field. Whenever the user changes any of the quantity fields, this calculation is re-executed and the new value is displayed in the corresponding Amount field.

Below the column of Amounts is the Total field. It contains the calculation:

```
Str(Sum(Amount[*]), 10, 2)
```

This sums all four occurrences of the field Amount, and formats the resulting number to two decimal places in a string, 10 characters wide. Whenever any of the amount fields change, this calculation is re-executed and a new value is displayed in the Total field.

Finally, the field named ShipDate also contains a calculation, specifically, a date calculation

```
Num2Date(Date() + 2, DateFmt())
```

This calculation gets the value of the current date (in days), adds 2 days to it and then formats this date value into a locale-sensitive date string. Were that user to be in the United States, in the year 2000, and on the ides of March, the result that would be displayed in the ShipDate field, is:

```
Mar 17, 2000
```

A user in Germany, on that same day, would see the value

```
17.03.2000
```

displayed in the same field. The latter is an illustration of the built-in internationalization capabilities of **FormCalc**'s date and time functions.

Admittedly, this is a very simple application. A real-world purchase order form would be significantly more complex, with perhaps several dozen calculations and validations. Hopefully this example will suffice to introduce some of the capabilities of the **FormCalc** language.

We will now proceed to formalize the definition of this language. More complex language examples will be presented throughout.

1.1 Grammar

The **FormCalc** language is defined in terms of a context-free grammar. This is a specification of the lexical and syntactic structure of **FormCalc** calculations.

A context-free grammar is defined as a number of productions. Each production has an abstract symbol called a nonterminal as its left-hand side, and a sequence of one or more nonterminal and terminal symbols as its right-hand side. The grammar specifies the set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production in which the nonterminal is the left-hand side.

1.2 Notational Conventions

The following convention in notation is used to describe the grammar of **FormCalc**:

Metasymbol	Description	Example
::=	start of the definition of a nonterminal symbol.	the syntax `FormCalculation ::= ExpressionList` defines the production FormCalculation as an ExpressionList symbol.
| *symbol*	alternative symbol.	the syntax `'+' | '-'` allows alternate additive operator symbols.
[*symbol*]	one from the set of enclosed symbol(s).	the syntax `['E' 'e']` allows one symbol from the set 'E', 'e' of symbols.
[*symbol* –*symbol*]	range of symbols.	the syntax `['0'–'9']` allows one symbol from the consecutive set '0', '1', ..., '9' of symbols.
symbol \– *symbol*	set difference of symbols.	the syntax `Character \– LineTerminator` allows one symbol from the set of Characters that is not a LineTerminator symbol.
(*symbol*)	one occurrence of the enclosed symbol(s).	the syntax `( '+' | '-' )` allows for one occurrance of either alternative symbol.
(*symbol*)*	zero or more occurrences of the enclosed symbol(s).	the syntax `( ',' SimpleExpression )*` allows for zero or more occurrence of the ',' symbol followed by a SimpleExpression symbol.
(*symbol*)?	at most one occurrence of the enclosed symbol(s).	the syntax `( ArgumentList )?` allows for zero or one occurrence of the ArgumentList symbol.

The nonterminal symbols of the grammar are always in normal print, often sub scripted by a production key, e.g., ProductionName$_{[ProductionKey]}$, as in LogicalAndExpression$_{[25]}$. The terminal symbols of the grammar are always enclosed in single quotes, as in '=' and 'then'.

2 Lexical Grammar

This section describes the lexical grammar of the **FormCalc** language. It defines a set of productions, starting from the nonterminal symbol Input[1], to describe how sequences of Unicode characters are translated into a sequence of input elements.

The grammar has as its terminal symbols the characters of the Basic Multilingual Plane (BMP) of the [UNICODE] 2.1 character set; this limitation allows us to hold onto the *"one character = one storage unit"* paradigm the original Unicode standard promised, a bit longer.

Input elements other than white spaces, line terminators and comments form the terminal symbols for the syntactic grammar of **FormCalc**, and are called tokens. These tokens are the literals, identifiers, keywords, separators and operators of the **FormCalc** language.

```
[1] Input ::= WhiteSpace | LineTerminator | Comment | Token
```

The source text for a **FormCalc** calculation is a sequence of characters using the Unicode character encoding. These Unicode characters are scanned from left to right, repeatedly taking the longest possible sequence of characters as the next input element.

```
[2] Character ::=  [#x9-#xD] | [#x20-#xD7FF] | [#xE000-#xFFFD]
```

Note: not all **FormCalc** hosting environments recognize these characters, e.g., XML does not allow the vertical tab (#xB) and form feed (#xC) characters as input.

2.1 White Space

White space characters are used to separate tokens from each other and improve readability but are otherwise insignificant.

```
[3] WhiteSpace ::=  #x9 | #xB | #xC | #x20
```

These are the horizontal tab (#x9), vertical tab (#xB), form feed (#xC), and space (#x20) characters.

2.2 Line Terminators

Line terminators, like white spaces are used to separate tokens and improve readability but are otherwise insignificant.

```
[4] LineTerminator ::=  #xA | #xD
```

These are the linefeed (#xA), and carriage return (#xD) characters.

2.3 Comments

Comments are used to improve readability but are otherwise insignificant.

A comment is introduced with a semi-colon (;) character, or a pair of slash (/) characters, and continues until a line terminator is encountered.

```
[5] Comment ::= ';' ( Character \- LineTerminator )*
              | '/' '/' ( Character \- LineTerminator )*
```

2.4 String Literals

A string literal is a sequence of Unicode characters enclosed within double quote characters, e.g., "the cat jumped over the fence." The string literal "" defines an empty sequence of text characters called the empty string.

To embed a double quote within a string literal, specify two double quote characters, as in "He said ""She said.""". Moreover within string literals, any Unicode character may be expressed as a Unicode escape sequence of 6 characters consisting of \u followed by four hexadecimal digits, e.g.,

```
"\u0047\u006f \u0066\u0069\u0073\u0068\u0021"
```

To embed a control character with a string literal, specify its Unicode escape sequence, e.g., specify \u000d for a carriage return, and \u000a for a newline character.

```
[6] HexDigit ::= ['0'-'9'] | ['a'-'f'] | ['A'-'F']
[7] EscapedCharacter ::= '"' '"'
                       | '\' 'u' HexDigit HexDigit HexDigit HexDigit
[8] StringLiteral ::= '"' ( Character \- ['"'] | EscapedCharacter )* '"'
```

2.5 Number Literals

A number literal is a sequence of mostly digits consisting of an integral part, a decimal point, a fractional part, an e (or E) and an optionally signed exponent part. Either the integral part or the fractional part may be missing, but not both. In the fractional part, either the decimal point or the e and exponent part may be missing, but not both.

```
[9] Integer ::= ['0'-'9']+ ( ['0'-'9'] )*
[10] Exponent ::= ['E' 'e'] ['+' '-']? Integer
[11] NumberLiteral ::= Integer '.' ['0'-'9']* Exponent?
                     | '.' Integer Exponent?
                     | Integer Exponent?
                     | 'nan'
                     | 'inf'
```

Examples of number literals include 12, 1.2345, .12, 1e-2, and 1.2E+3.

All number literals are internally converted to [IEEE754] 64-bit binary values. However, IEEE 754 values can only represent a finite quantity of numbers. Just as some numbers, such as 1/3, are not representable precisely as decimal fractions, other numbers are not precisely representable as binary fractions. Specifically, but not limited to, number literals having more than 16 significant digits in the non-exponent part will be the rounded to the nearest representable IEEE 754 64-bit value using a round-to-nearest mechanism. This means that a number literal like

123456789.012345678

will be rounded to the (nearest) value

123456789.01234567

and a number literal like

9999999999999999

will be rounded to the (nearest) value

100000000000000000

This behaviour is conformant to the IEEE 754 standard.

This behaviour can sometimes lead to surprising results. **FormCalc** provides a function, Round(), which returns a given number rounded to a given number of decimal places. When the given number is exactly halfway between two representable numbers, it is rounded away from zero; up if positive, and down if negative. So, e.g.,

round(.124, 2)

returns 0.12, and

round(.125, 2)
returns 0.13. Given this convention, one might expect then that
round(0.045, 2)
returns 0.05. It doesn't. IEEE 754 dictates that the number literal
0.045
be approximated to
0.0449999999999999
This approximation is closer to 0.04 than to 0.05, so
Round(0.045, 2)
returns 0.04. This too is conformant to the IEEE 754 standard.

IEEE 754 64-bit values also support representations like NaN (not a number), +Inf (positive infinity), and -Inf (negative infinity). **FormCalc** does not support these; currently, any intermediate expression that evaluates to NaN, +Inf, or -Inf results in an error exception which is propagated in the remainder of the expression. This behaviour is expected to change soon.

2.6 Literals

```
[12] NullLiteral ::= 'null'
[13] Literal ::= StringLiteral | NumberLiteral  | NullLiteral
```

The NullLiteral equates to the null value

2.7 Keywords

Keywords in **FormCalc** are reserved words and are case insensitive. Of these, the 'if', 'then', 'elseif', 'else', 'endif' keywords delimit the parts of an IfExpression[39]. The 'nan' and 'inf' keywords denote special number literals, whereas the 'null' keyword denotes the null literal. The 'this' keyword denotes a specific accessor. The remaining keywords are keyword operators.

The following are keywords and may not be used as identifiers:

```
[14] Keyword ::= 'if' | 'then' | 'elseif' | 'else' | 'endif'
               | 'or' | 'and' | 'not' | 'eq' | 'ne'
               | 'le' | 'ge' | 'lt' | 'gt'
               | 'this' | 'null' | 'nan' | 'infinity'


[1A] ReservedKeyword ::= | 'while' | 'do' | 'endwhile' | 'end'
               | 'for' | 'upto' | 'downto' | 'step' | 'endfor'
               | 'foreach | 'in'
               | 'break' | 'continue'
               | 'var' | 'func' | 'endfunc'
               | 'throw' | 'return' | 'exit'
```

2.8 Identifiers

An identifier is a sequence of characters of unlimited length but always beginning with an alphabetic character, or an underscore (_) character, or a dollar sign ($) character, or an exclamation mark (!) character.

FormCalc identifiers are case sensitive, i.e., identifiers whose characters only differ in case, are considered distinct. Case sensitivity is mandated by **FormCalc**'s hosting environments.

```
[15] Identifier ::= ( AlphabeticCharacter | '_' | '$' | '!' )
                    ( AlphaNumericCharacter | '_' | '$' )*
```

An alphabetic character is any Unicode character classified as a letter. An alphanumeric character is any Unicode character classified as either a letter, or a digit.

2.9 Operators

FormCalc defines a number of operators; they include unary operators, multiplicative operators, additive operators, relational operators, equality operators, logical operators, and the assignment operator.

FormCalc operators are symbols common to most other scripting languages:

```
[16] Operator ::= '=' | '|' | '&' | '==' | '<>' | '<='
                | '>=' | '<' | '>' | '+' | '-' | '*' | '/'
```

Several of the **FormCalc** operators have an equivalent mnemonic operator keyword. These keyword operators are useful whenever **FormCalc** expressions are embedded in HTML and XML source text, where symbols <, >, and & have predefined meanings and must be escaped. Here's an enumeration of all **FormCalc** operators, illustrating the symbolic and mnemonic forms of various operators.

```
[31] LogicalOrOperator ::= '|' | 'or'
[32] LogicalAndOperator ::= '&' | 'and'
[33] EqualityOperator ::= '==' | '<>' | 'eq' | 'ne'
[34] RelationalOperator ::= '<=' | '>=' | '<' | '>' | 'le' | 'ge' | 'lt' | 'gt'
[35] AdditiveOperator ::= '+' | '-'
[36] MultiplicativeOperator ::= '*' | '/'
[37] UnaryOperator ::= '-' | '+' | 'not'
```

2.10 Tokens

```
[17] Separator ::= '(' | ')' | '[' | ']' | ',' | '.' | '..' | '.#' | '.*'
[18] Token ::= Literal | Keyword | Identifier | Operator | Separator
```

3 Syntactic Grammar

The syntactic grammar for **FormCalc** has the tokens defined in the preceding lexical grammar as its terminal symbols, and defines the set of productions, starting from the nonterminal symbol FormCalculation[20], to describe how sequences of tokens can form a syntactically valid calculation.

```
[20] FormCalculation ::= ExpressionList

[21] ExpressionList ::=
        Expression |
        ExpressionList Expression

[22] Expression ::=
        IfExpression |
        AssignmentExpression |
        SimpleExpression

[23] SimpleExpression ::=
        LogicalOrExpression

[24] LogicalOrExpression ::=
        LogicalAndExpression |
        LogicalOrExpression LogicalOrOperator LogicalAndExpression

[25] LogicalAndExpression ::=
        EqualityExpression |
        LogicalAndExpression LogicalAndOperator EqualityExpression

[26] EqualityExpression ::=
        RelationalExpression |
        EqualityExpression EqualityOperator RelationalExpression

[27] RelationalExpression ::=
        AdditiveExpression |
        RelationalExpression RelationalOperator AdditiveExpression

[28] AdditiveExpression ::=
        MultiplicativeExpression |
        AdditiveExpression AdditiveOperator MultiplicativeExpression

[29] MultiplicativeExpression ::=
        UnaryExpression |
        MultiplicativeExpression MultiplicativeOperator UnaryExpression

[30] UnaryExpression ::=
        PrimaryExpression |
        UnaryOperator UnaryExpression

[31] LogicalOrOperator ::= '|' | 'or'

[32] LogicalAndOperator ::= '&' | 'and'

[33] EqualityOperator ::= '==' | '<>' | 'eq' | 'ne'

[34] RelationalOperator ::= '<=' | '>=' | '<' | '>'
                          | 'le' | 'ge' | 'lt' | 'gt'
```

```
[35] AdditiveOperator ::= '+' | '-'

[36] MultiplicativeOperator ::= '*' | '/'

[37] UnaryOperator ::= '-' | '+' | 'not'

[38] PrimaryExpression ::=
       Literal |
       FunctionCall |
       Accessor ( '.*' )? |
       '(' SimpleExpression ')'

[39] IfExpression ::=
       'if' '(' SimpleExpression ')' 'then'
           ExpressionList
     ( 'elseif' '(' SimpleExpression ')' 'then'
           ExpressionList )*
     ( 'else'
           ExpressionList )?
       'endif'

[40] AssignmentExpression ::=
       Accessor '=' SimpleExpression

[4G] FunctionCall ::= Function '(' ( ArgumentList )? ')'

[4E] Function ::= Identifier

[41] Accessor ::=
       Container |
       Accessor [ '.' '..' '.#' ] Container

[42] Container ::=
       Identifier |
       Identifier '[' '*' ']' |
       Identifier '[' SimpleExpression ']' |
       MethodCall

[43] MethodCall ::= Method '(' ( ArgumentList )? ')'

[44] Method ::= Identifier

[45] ArgumentList ::= SimpleExpression (',' SimpleExpression )*
```

4 Expressions

4.1 Expressions Lists

```
[20] FormCalculation ::= ExpressionList

[21] ExpressionList ::= Expression
                      | ExpressionList Expression
```

A FormCalculation[20] is just a list of expressions. Under normal circumstances, each Expression[22] evaluates to a value, and the value of an ExpressionsList[21] is the value of last expression in the list. For example, if the FormCalculation[20]

```
5 + Abs(Price)
"Hello World"
10 * 3 + 5 * 4
```

is associated with field `Amount`, then after the evaluation of this FormCalculation[20], the value of field `Amount` would be 50.

4.2 Simple Expressions

```
[22] Expression ::= SimpleExpression | ...
```

The grammar presented above defining a SimpleExpression[23] is common to conventional languages. Operator precedence rules behave as expected, so, e.g.,

```
10 * 3 + 5 * 4
```

is equivalent to

```
(10 * 3) + (5 * 4)
```

and evaluates to 50. Similarly

```
0 and 1 or 2 > 1
```

is equivalent to

```
(0 and 1) or (2 > 1)
```

and evaluates to 1. Enumerating all the **FormCalc** operators in order, from high precedence to lowest precedence yields:

```
=
(unary) - + not
*  /
+  -
<  <=  >  >=  lt  le  gt  ge
==  <>  eq  ne
&  and
|  or
```

When performing numeric operations involving non-numeric operands, the non-numeric operands are first promoted to numbers; if the non-numeric operand can be fully converted to a numeric value then that is its value; otherwise its value is zero (0). When promoting null-valued operands to numbers, their value is always zero. As examples, the expression

```
(5 - "abc") * 3
```

evaluates to 15, i.e., (5 - 0) * 3 = 15, whereas

```
"100" / 10e1
```

evaluates to 1, and

```
5 + null + 3
```

evaluates to 8.

When performing boolean operations on non-boolean operands, the non-boolean operands are first promoted to booleans; if the non-boolean operand can be fully converted to a nonzero value then its value is true (1); otherwise its value is false (0). When promoting null-valued operands to booleans, their value is always false. As examples, the expression

```
"abc" | 2
```

evaluates to 1, i.e., false | true = true, whereas

```
if ("abc") then
    10
else
    20
endif
```

evaluates to 20.

When performing string operations on non-string operands, the non-string operands are first promoted to strings by using their value as a string. When promoting null-valued operands to strings, their value is always the empty string. As examples, the expression

```
concat("The total is ", 2, " dollars and ", 57, " cents.")
```

evaluates to "The total is 2 dollars and 57 cents."

All the intermediate results of numeric expressions are evaluated as double precision IEEE 754 64 bit values. The final result is displayed with up to 11 fractional digits of precision. Should an intermediate expression yield an NaN, +Inf or -Inf, **FormCalc** will currently generate an error exception and propagate that error for the remainder of that expression, and the expression's value will always be zero, e.g.,

```
3 / 0 + 1
```

Handling of error exceptions returned from the evaluation of **FormCalc** expressions is application defined.

4.3 Logical Or Expressions

```
[24] LogicalOrExpression ::= LogicalAndExpression
       | LogicalOrExpression LogicalOrOperator LogicalAndExpression

[31] LogicalOrOperator ::= '|' | 'or'
```

A LogicalOrExpression[24] returns the result of a logical disjunction of its operands, or null if both operands are null. If not both null, the operands are promoted to numeric values, and a numeric operation is performed.

The LogicalOrOperators '|' and 'or', both represent the same logical-or operator. The logical-or operator returns the boolean result true, represented by the numeric value 1, whenever either operand is not 0 and returns the boolean result false, represented by the numeric value 0, otherwise.

4.4 Logical And Expressions

```
[25] LogicalAndExpression ::= EqualityExpression
       | LogicalAndExpression LogicalAndOperator EqualityExpression

[32] LogicalAndOperator ::= '&' | 'and'
```

A LogicalAndExpression[25] returns the result of a logical conjunction of its operands, or null if both operands are null. If not both null, the operands are promoted to numeric values, and a numeric operation is performed.

The LogicalAndOperators '&' and 'and', both represent the same logical-and operator. The logical-and operator returns the boolean result true, represented by the numeric value 1, whenever both operands are not 0 and returns the boolean result false, represented by the numeric value 0, otherwise.

4.5 Equality Expressions

```
[26] EqualityExpression ::= RelationalExpression
         | EqualityExpression EqualityOperator RelationalExpression

[33] EqualityOperator ::= '==' | '<>' | 'eq' | 'ne'
```

An EqualityExpression[26] returns the result of an equality comparison of its operands.

If either operand is null, then a null comparison is performed. Null valued operands compare identically whenever both operands are null, and compare differently whenever one operand is not null.

If both operands are references, then both operands compare identically when they both refer to the same object, and compare differently when they don't refer to the same object.

If both operands are string valued, then a locale-sensitive lexicographic string comparison is performed on the operands. Otherwise, if not both null, the operands are promoted to numeric values, and a numeric comparison is performed.

The EqualityOperators '==' and 'eq', both denote the equality operator. The equality operator returns the boolean result true, represented by the numeric value 1, whenever both operands compare identically and returns the boolean result false, represented by the numeric value 0, otherwise.

The EqualityOperators '<>' and 'ne', both denote the inequality operator. The inequality operator returns the boolean result true, represented by the numeric value 1, whenever both operands compare differently and returns the boolean result false, represented by the numeric value 0, otherwise.

4.6 Relational Expressions

```
[27] RelationalExpression ::= AdditiveExpression
         | RelationalExpression RelationalOperator AdditiveExpression

[34] RelationalOperator ::= '<=' | '>=' | '<' | '>' | 'le' | 'ge' | 'lt' | 'gt'
```

A RelationalExpression[27] returns the result of a relational comparison of its operands.

If either operand is null valued, then a null comparison is performed. Null valued operands compare identically whenever both operands are null and the relational operator is less-than-or-equal or greater-than-or-equal, and compare differently otherwise.

If both operands are string valued, then a locale-sensitive lexicographic string comparison is performed on the operands. Otherwise, if not both null, the operands are promoted to numeric values, and a numeric comparison is performed.

The RelationalOperators '<' and 'lt', both denote the same less-than operator. The less-than-or-equal relational operator returns the boolean result true, represented by the numeric value 1, whenever the first operand is less than the second operand, and returns the boolean result false, represented by the numeric value 0, otherwise.

The RelationalOperators '<=' and 'le', both denote the less-than-or-equal operator. The less-than-or-equal relational operator returns the boolean result true, represented by the numeric value 1, whenever the first operand is less than or equal to the second operand, and returns the boolean result false, represented by the numeric value 0, otherwise.

The RelationalOperators '>' and 'gt', both denote the same greater-than operator. The greater-than relational operator returns the boolean result true, represented by the numeric value 1, whenever the first operand is greater than the second operand, and returns the boolean result false, represented by the numeric value 0, otherwise.

The RelationalOperators '>=' and 'ge', both denote the greater-than-or-equal operator. The greater-than-or-equal relational operator returns the boolean result true, represented by the numeric value 1, whenever the first operand is greater than or equal to the second operand, and returns the boolean result false, represented by the numeric value 0, otherwise.

4.7 Additive Expressions

```
[28] AdditiveExpression ::= MultiplicativeExpression
        | AdditiveExpression AdditiveOperator MultiplicativeExpression

[35] AdditiveOperator ::= '+' | '-'
```

An AdditiveExpression[28] returns the result of an addition (or subtraction) of its operands, or null if both operands are null. If not both null, the operands are promoted to numeric values, and a numeric operation is performed.

The AdditiveOperator '+', is the addition operator; it returns the sum of its operands.

The AdditiveOperator '-', is the subtraction operator; it returns the difference of its operands.

4.8 Multiplicative Expressions

```
[29] MultiplicativeExpression ::= UnaryExpression
         | MultiplicativeExpression MultiplicativeOperator UnaryExpression

[36] MultiplicativeOperator ::= '*' | '/'
```

A MultiplicativeExpression[29] returns the result of a multiplication (or division) of its operands, or null if both operands are null. If not both null, the operands are promoted to numeric values, and a numeric operation is performed.

The MultiplicativeOperator '*', is the multiplication operator; it returns the product of its operands.

The MultiplicativeOperator '/', is the division operator; it returns the quotient of its operands.

4.9 Unary Expressions

```
[30] UnaryExpression ::= PrimaryExpression
         | UnaryOperator UnaryExpression

[37] UnaryOperator ::= '-' | '+' | 'not'
```

A UnaryExpression[30] returns the result of a unary operation of its operand.

The UnaryOperator '-' denotes the unary minus operator; it returns the arithmetic negation of its operand, or null if its operand is null. If its operand is not null, it is promoted to a numeric value, and the unary minus operation is performed.

The UnaryOperator '+' denotes the unary plus operator; it returns the arithmetic value of its operand, or null if its operand is null. If its operand is not null, it is promoted to a numeric value, and the unary plus operation is performed.

The UnaryOperator 'not' denotes the logical negation operator. it returns the logical negation of its operand. Its operand is promoted to a boolean value, and the logical operation is performed.

The logical negation operation returns the boolean result true, represented by the numeric value 1, whenever its operand is 0, and returns the boolean result false, represented by the numeric value 0, otherwise.

Note: the arithmetic negation of a null operand yields the result null, whereas the logical negation of a null operand yields the boolean result true. This is justified by the common sense statement: If null means nothing then "not nothing" should be something.

4.10 Primary Expressions

```
[38] PrimaryExpression ::=
          Literal
        | Accessor
        | MethodCall
        | '(' SimpleExpression ')'
```

A PrimaryExpression[38] is the building block of all simple expressions. It consists of literals, variables, accessors, function calls, and parenthesised simple expressions.

The value of the PrimaryExpression[38] is the value of its constituent Literal[13], Accessor[41], SimpleExpression[23].

4.11 If Expressions

```
[22] Expression ::= IfExpression | ...

[39] IfExpression ::=
          'if' '(' SimpleExpression ')' 'then'
               ExpressionList
        ( 'elseif' '(' SimpleExpression ')' 'then'
               ExpressionList )*
        ( 'else'
               ExpressionList )?
          'endif'
```

An IfExpression[39] is a conditional expression, which, depending upon the value of the SimpleExpression[23] in the if-part, will either evaluate and return the value of the ExpressionList[21] in its then-part or, if present, evaluate and return the value of the ExpressionList[21] in its elseif-part or else-part.

The value of the SimpleExpression[23] in the if-part is promoted to a boolean value and a logical boolean operation is performed. If this boolean operation evaluates to true (1), the value of the ExpressionList[21] in the then-part is returned. Otherwise, if there's an elseif-part present, and the value of the SimpleExpression[23] in the elseif-part evaluates to true (1), then the value of its ExpressionList[21] is returned. If there are several elseif-parts, the SimpleExpression[23] of each elseif-part, is evaluated, in order, and if true(1), then the value of its corresponding ExpressionList[21] is returned. Otherwise, the value of the ExpressionList[21] in the else-part is returned; if there is no else-part, the value 0 is returned.

In any circumstance, only one of the expression lists is ever evaluated.

5 Assignment Expressions

```
[22] Expression ::= AssignmentExpression  | ...

[40] AssignmentExpression ::= Accessor '=' SimpleExpression
```

An AssignmentExpression[40] sets the property identified by the Accessor[41] to the value of the SimpleExpression[23].

The value of the AssignmentExpression[40] is the value of the SimpleExpression[23]

6 Accessors

```
[38] PrimaryExpression ::= Accessor ( '.*' )?
                         | ...

[40] AssignmentExpression ::= Accessor '=' SimpleExpression
                            | ...

[41] Accessor ::= Container
                | Accessor [ '.' '..' '.#' ] Container

[42] Container ::= Identifier
                 | Identifier '[' '*' ']'
                 | Identifier '[' SimpleExpression ']'
                 | MethodCall
```

FormCalc provides access to object properties and values, which are all described in the [XFASOM] specification. An Accessor[41] is the syntactic element through which object values and properties are assigned, when used on the left-hand side of an AssignmentExpression[40], or retrieved, when used in a SimpleExpression[23], as in:

```
Invoice.VAT = Invoice.Total * (8 / 100)
```

Accessors may consist of a fully qualified hierarchy of objects, as in:

```
$form.subform.subform.field[10].Price = "255.99"
```

and optionally followed by an object property, as in:

```
Invoice.border.edge[1].color.#value = "255,9,9"
```

The object property is indicated by the use of the '.#' separator.

Accessors may equally consist of a partially qualified hierarchy of objects, again optionally followed by an object property, as in:

```
Invoice..edge[1].color.#value = "255,9,9"
```

The hierarchy is indicated by the use of the '..' separator.

When terminated with the '.*' separator, instead, what is referred to is the collection of sub-objects of the object identified by the accessor.

A container is simply the name of an object or object property.

A hierarchy of objects presupposes an architectural model, and the complete description of that model is outside the scope of this document -- see the [XFASOM] specification. However, it suffices to say that because there can be multiple instances of objects with the same name on a form, each instance gets assigned an occurrence number, starting from zero. To refer to a specific instance of an object which bears the same ambiguous name as other objects, it is required that the name be qualified by an occurrence number corresponding to the desired ordinal instance of the object.

Aside from a referral to the absolute occurrence of an object, there also exists the need to refer to the relative occurrence of an object, and to all occurrences of an object. To that end, **FormCalc** uses the notation:

`Identifier`	to refer to an occurrence of the object that bears the same ordinal

	occurrence number as the referencing object.
`Identifier[ SimpleExpression ]`	to refer to the occurrence of the object identified by the runtime value of the expression.
`Identifier[+ ( SimpleExpression )]`	to refer to the n'th **succeeding** occurrence of the object identified by the runtime value of the expression, relative to the referencing object's occurrence number.
`Identifier[- ( SimpleExpression )]`	to refer to the n'th **preceding** occurrence of the object identified by the runtime value of the expression, relative to the referencing object's occurrence number.
`Identifier[*]`	to refer to every occurrence of the identified object.

Thus, `Identifier[0]` refers to the first occurrence of the identified object, and by convention, `Identifier[+0]` and `Identifier[-0]` refer to the object whose occurrence number is the same as the referencing object.

The notation `Identifier [ SimpleExpression ]` involves an indexing operation, which must yield a numeric result. If the SimpleExpression[23] operand is non-numeric, then it will be promoted to a number using the rule for a SimpleExpression[23]; if the non-numeric indexing operand can be fully converted to a numeric value then that is its value; otherwise its value is zero (0), and, when promoting a null-valued indexing operand to a number, its value is always zero.

Some accessor expressions can often evaluate to a set of values, and some built-in functions like: Avg(), Count(), Max(), Min(), Sum(), and, Concat() are designed to accept a set of values. However, it is not always possible to determine the exact number of arguments passed to a function at time of compilation. For example, consider the following form calculation:

```
Max(Price[*])
```

If there are no occurrences of object `Price`, then the function Max() will generate an error exception. If there is a single occurrence of object `Price`, then the function Max() will return the value of that object occurrence. If there are multiple occurrences of object `Price`, then the function Max() will return the maximum value of all those object occurrences.

For all occurrences of a given object to be included in a calculation, the object must be specified using the [*]-style of accessor referral, e.g.,

```
Sum(Price[*])
```

will sum all occurrences of object `Price`. Specifying

```
Sum(Price)
```

will only sum a single occurrence of object `Price`.

In most other built-in functions, the description of the formal arguments stipulates that it must be a single value, but it may be that the passed argument evaluates to a set of values. In such circumstances, the function will generate an error exception. For example, the following form calculation:

```
Abs(Quantity[*])
```

will always generate an error exception, irrespectively of the number of occurrences of object `Quantity`. This rule applies to all binary and unary operands involving accessors the [*]-style of referral;

```
Quantity[*] + 10
```

will always generate an error exception.

As noted earlier, the dollar sign ($) character is a valid character in Identifier names. However, this specification recommends that processing applications forbid including the dollar sign ($) character in the names of objects and

properties -- object names and properties containing this character can thus be reserved for special application-defined tasks.

7 Method Calls

```
[43] MethodCall ::= Method '(' ( ArgumentList )? ')'

[44] Method ::= Identifier

[45] ArgumentList ::= SimpleExpression (',' SimpleExpression )*
```

FormCalc also provides access to object methods, not just objects and object properties. The syntax for Accessors[41] permits this. Object methods are all described in the [XFASOM] specification.

Methods are application-defined operators that act upon objects and their properties; these operators are invoked like a function call, in that arguments may be passed to methods exactly like function calls. The number and type of arguments in each method are prescribed by each object type. Objects of different types will support different methods.

Here are examples of method calls:

```
var v = $.getValue();   // retrieve this referencing object's value.
$host.MessageBox(1, v); // display it in this host's dialog box.
```

8 Function Calls

```
[22] Expression ::= FunctionCall | ...

[4G] FunctionCall ::= Function '(' ( ArgumentList )? ')'

[4E] Function ::= Identifier

[45] ArgumentList ::= SimpleExpression (',' SimpleExpression )*
```

FormCalc supports a large set of built-in functions to do arithmetic, financial, logic, date, time, and string operations.

The names of the **FormCalc** built-in functions are case insensitive, but are not reserved. This means that calculations on forms with objects whose names coincide with the names of built-in functions do not conflict; any object method or built-in function can be equally called. Case insensitivity of built-in function names is mandated by **FormCalc**'s historical legacy.

All built-in functions take an ArgumentList[45]. The number and type of arguments varies with each function. Some, like Date() and Time() take no arguments. Others, like Num2Date() take 1, 2 or 3 arguments, the first argument being a number, with the remaining arguments being strings. Many functions accept a variable number of arguments. Leading arguments are mandatory, and trailing arguments are often optional. This maintains the complexity of most functions at a low level. Increased functionality is provided to those users who need it by requiring them to supply the additional arguments.

All arguments in an ArgumentList[45] are evaluated in order, leading arguments first. If the number of mandatory arguments passed to a function is less than the number required, the function generates an error exception.

Many functions require numeric arguments. If any of the passed arguments are non-numeric, they are promoted to numbers. Some function arguments only require integral values; in such cases, the passed arguments are always promoted to integers by truncating the fractional part.

Here's a summary of the key properties of built-in functions:

- built-in function names are case-insensitive.
- the built-in functions are predefined, but their names are not reserved words: this means that the built-in function Max() will never conflict with an object, object property, or object method named Max.
- many of the built-in functions have a mandatory number of arguments which can be followed by a optional number of arguments.
- a few built-in functions, Avg(), Count(), Max(), Min(), Sum(), and, Concat() will accept an indefinite number of arguments.

9 Arithmetic Built-in Functions

9.1 Abs()

This function returns the absolute value of a given number.

9.1.1 Definition

```
Abs(n1)
```

9.1.2 Parameters

n1
is the number to evaluate.

9.1.3 Returns

The absolute value or null if its parameter is null.

9.1.4 Examples

```
Abs(1.03)
```

returns 1.03.

```
Abs(-1.03)
```

returns 1.03.

```
Abs(0)
```

returns 0.

9.2 Avg()

This function returns the average of the non-null elements of a given set of numbers.

9.2.1 Definition

```
Avg(n1 [, n2...])
```

9.2.2 Parameters

n1
is the first number in the set.

n2, ...
are optional additional numbers in the set.

9.2.3 Returns

The average of its non-null parameters, or null if its parameter are all null.

9.2.4 Examples

```
Avg(Price[0], Price[1], Price[2], Price[3])
```

returns 9 if `Price[0]` has a value of 8, `Price[1]` has value 10, and `Price[2]` and `Price[3]` are null.

```
Avg(Quantity[*])
```

returns 9 if `Quantity` has two occurrences with values of 8 and 10, and returns null if all occurrences of `Quantity` are null.

9.3 Ceil()

This function returns the whole number greater than or equal to a given number.

9.3.1 Definition

```
Ceil(n1)
```

9.3.2 Parameters

n1
is the number to evaluate.

9.3.3 Returns

The ceiling or null if its parameter is null.

9.3.4 Examples

```
Ceil(1.9)
```

returns 2.

```
Ceil(-1.9)
```

returns -1.

```
Ceil(A)
```

is 100 if the value A is 99.999

9.4 Count()

This function returns the count of the non-null elements of a given set of numbers.

9.4.1 Definition

```
Count(n1 [, n2...])
```

9.4.2 Parameters

n1
: is the first argument to count.

n2, ...
: are optional additional arguments in the set.

9.4.3 Returns

The count.

9.4.4 Examples

```
Count(5, "ABCD", "", null)
```

returns 3.

```
Count(Quantity[*])
```

returns the number of occurrences of `Quantity` that are non-null, and returns 0 if all of occurrences of `Quantity` are null.

9.5 Floor()

This function returns the largest whole number that is less than or equal to a given value.

9.5.1 Definition

```
Floor(n1)
```

9.5.2 Parameters

n1
: is the number to evaluate.

9.5.3 Returns

The floor or null if its parameter is null.

9.5.4 Examples

```
Floor(6.5)
```

returns 6.

```
Floor(7.0)
```

returns 7.

```
Floor(Price)
```

returns 99 if the value of `Price` is 99.999.

9.6 Max()

This function returns the maximum value of the non-null elements of a given set of numbers.

9.6.1 Definition

```
Max(n1 [, n2...])
```

9.6.2 Parameters

n1
: is the first number in the set.

n2, ...
: are optional additional numbers in the set.

9.6.3 Returns

The maximum of its non-null parameters, or null if all its parameters are null.

9.6.4 Examples

```
Max(Price[*], 100)
```

returns the maximum value of all occurrences of the object Price or 100, whichever is greater.

```
Max(7, 10, null, -4, 6)
```

returns 10.

```
Max(null)
```

returns null.

9.7 Min()

This function returns the minimum value of the non-null elements of a given set of numbers.

9.7.1 Definition

```
Min(n1 [, n2...])
```

9.7.2 Parameters

n1
is the first number in the set.

n2, ...
are optional additional numbers in the set.

9.7.3 Returns

The minimum of its non-null parameters, or null if all its parameters are null.

9.7.4 Examples

```
Min(7, 10, null, -4, 6)
```

returns -4.

```
Min(Price[*], 100)
```

returns the minimum value of all occurrences of the object Price or 100, whichever is less.

```
Min(null)
```

returns null.

9.8 Mod()

This function returns the modulus of one number divided by another.

9.8.1 Definition

```
Mod(n1, n2)
```

9.8.2 Parameters

n1
is the dividend number.

n2
is the divisor number.

9.8.3 Returns

The modulus or null if any of its parameter are null.

The modulus is the remainder of the implied division of the dividend and the divisor. The sign of the remainder always equals the sign of the dividend.

For integral operands, this is simple enough. For floating point operands, the floating point remainder `r` of `Mod(n1, n2)` is defined as `r = n1 - (n2 * q)` where `q` is an integer whose sign is negative when `n1 / n2` is negative, and positive when `n1 / n2` is positive, and whose magnitude is the largest integer less than the quotient `n1 / n2`.

If the divisor is zero, the function generates an error exception.

9.8.4 Examples

```
Mod(64, 2)
```

returns 0.

```
Mod(-13, 3)
```

returns -1.

```
Mod(13, -3)
```

returns 1.

```
Mod(-13.6, 2.2)
```

returns -0.4.

9.9 Round()

This function returns a number rounded to a given number of decimal places.

9.9.1 Definition

```
Round(n1 [, n2])
```

9.9.2 Parameters

n1
is the number to be evaluated.

n2
is the number of decimal places.
If n2 is omitted, 0 is used as the default.
If n2 is greater than 12, 12 is used as the maximal precision.

9.9.3 Returns

The rounded value or null if any of its parameters are null.

9.9.4 Examples

```
Round(33.2345, 3)
```

returns 33.235.

```
Round(20/3, 2)
```

returns 6.67.

```
Round(-1.3)
```

returns -1.

```
Round(Price, 2)
```

returns 2.33 if the value of the object Price is 2.3333

9.10 Sum()

This function returns the sum of the non-null elements of a given set of numbers.

9.10.1 Definition

```
Sum(n1 [, n2...])
```

9.10.2 Parameters

n1
is the first number to sum.

n2, ...
are optional additional numbers in the set.

9.10.3 Returns

The sum of its non-null parameters, or null if all of its parameters are null.

9.10.4 Examples

```
Sum(1, 2, 3, 4)
```

returns 10.

```
Sum(Amount[*])
```

returns the sum of all occurrences of the object Amount.

```
Sum(Amount[2], Amount[3])
```

returns the sum of two occurrences of the object Amount.

10 Date And Time Built-in Functions

A number of built-in date and time functions are provided, to allow the form designer to:

- select locale-specific date formats (`DateFmt()` and `LocalDateFmt()`),
- parse strings into numbers according to locale-specific date formats (`Date2Num()`),
- get the current date (`Date()`),
- do basic arithmetic on dates,
- format numbers into strings according to locale-specific date formats (`Num2Date()`).
- select locale-specific time formats (`TimeFmt()` and `LocalTimeFmt()`),
- parse strings into numbers according to locale-specific time formats (`Time2Num()`),
- get the current time (`Time()`),
- do basic arithmetic on times,
- format numbers into strings according to locale-specific time formats (`Num2Time()`).
- parse ISO-8601 date strings and time strings into numbers (`IsoDate2Num()` and `IsoTime2Num()`).

To properly parse and format a date, we need to know what format it is in, and what locale it comes from. As we all know, dates are represented differently, even within the same country. So we need to define the concepts of locale and date format more precisely.

10.1 Locales

When developing internationalized applications, a **locale** is the standard term used to identify a particular nation (language and/or country). A locale defines (but is not limited to) the format of dates, times, numeric and currency punctuation that are culturally relevant to a specific nation. A properly internationalized application will always rely on the locale to supply it with the format of dates, and times. This way, users operating in their locale will always be presented with the date and time formats they are accustomed to.

A locale is identified by a language code and/or a country code. Usually, both elements of a locale are important. For example, the names of weekdays and months in English Canada and in the UK are formatted identically, but dates are formatted differently. So, specifying an English language locale would not suffice. Conversely, specifying only a country as the locale may not suffice either -- for example, Canada, has different date formats for English and French.

A **locale identifier** is a unique string representing a locale. The structure and meaning of locale identifiers are defined in [RFC1766].

In general, every application operates in an environment where a locale is present; this is the **ambient locale**. In the rare circumstance where the application is operating on a system or within an environment where a locale is not present, the ambient locale will default to English United States (en-US); this is the **default locale**.

10.2 Date Formats

The format of dates is governed by an ISO standards body whereby each nation gets to specify the form of its default, short, medium, long, and full date formats. Specifically, the locale is responsible for identifying the format of dates that conform to the standards of that nation.

Short date formats tend to be purely numeric, e.g.,
10/2/70.
Medium date formats specify use of abbreviated month names, e.g.,
10-Feb-70,
and long date formats specify use of full month names, e.g.,

February 10, 1970.
Full date formats tend to include the weekday name, as in:
Thursday, February 10, 1970.
The default format tends to coincide with the medium date format.

Properly internationalized applications then, will always query the locale for a date format. The form designer has the option of choosing from either the default, short, medium, long or full formats, and will never present to the user a hand-crafted date format. Except for the need of a common format for data interchange, use of hand-crafted date formats are best avoided.

A **date format** is a shorthand specification to format a date. It consists of punctuations, literals, and pattern symbols, e.g., "D/M/YY" is a date format.

For a specification of how to construct date formats, refer to the section Date Pictures within the XFA-Picture Clause Specification for more information.

Examples of date formats include:

- MM/DD/YY
  MM/DD/YY
  DD.MM.YYYY
  DD MMM YYYY
  MMMM DD, YYYY
  EEEE,' le 'D MMMM, YYYY

Specifically, in the default en_US locale, the **default date format** is
MMM D, YYYY.

Date formats are used to both format and parse date strings, using the built-in functions Num2Date() and Date2Num(). All formatting and parsing is strict; when formatting, all literals and punctuations are included, and when parsing, all literals and punctuations must be matched exactly. If the date format is meaningless, no formatting nor parsing is attempted.

10.3 Localized Date Formats

Properly internationalized e-forms need to prompt the user for a date in a particular format, e.g., if the caption on a date entry field reads:
YY/MM/DD,
then that is a compelling hint to the user that a date needs to be entered in the
"YY/MM/DD"
format. That is, to an English user. But to a user from the city of Montreal, that caption may be unrecognizable; the user may more likely be able to understand the caption:
aa-nn-jj.
The above string is an example of a localized date format -- it's the localized French Canadian date format equivalent to the
"YY/MM/DD"
date format.

As another example, the full date format for the German-speaking part of Switzerland is
"EEEE, D. MMMM YYYY".
The equivalent localized date format is
"EEEE, t. MMMM uuuu",
which is hopefully more meaningful to these users than it is to us, and that

"EEEE, D. MMMM YYYY"
is to them.

The built-in functions Num2Date() and Date2Num() do not understand localized date formats; they only understand the date formats defined above. Use of these localized date formats is for UI presentation only.

10.4 Time Formats

In much the same way that date formats are governed by an ISO standards body, so are time formats. Again, each nation gets to specify the form of its default, short, medium, long, and full time formats. The locale is responsible for identifying the format of times that conform to the standards of that nation.

The default time format tends to coincide with the medium time format.

Specifically, in the default en_US locale, the **default time format** is
h:MM:SS A.

Just as with a date format, a **time format** is a shorthand specification to format a time. It consists of punctuations, literals, and pattern symbols, e.g., "HH:MM:SS" is a time format.

For a specification of how to construct time formats, refer to the section Time Pictures within the XFA-Picture Clause Specification for more information.

Examples of time formats include:

- h:MM A
  HH:MM:SS
  HH:MM:SS 'o''clock' A Z

Any time format containing incorrectly specified pattern symbols, e.g., HHH are invalid. When parsing, time formats with multiple instances of the same pattern symbols, e.g., HH:MM:HH are invalid, as are time formats with conflicting pattern symbols, e.g., h:HH:MM:SS. Time formats with adjacent one letter pattern symbols, e.g., HMS, are inherently ambiguous and should be avoided.

10.5 Date and Time Values

To do basic arithmetic on dates and times, we introduce the concept of date values and time values. Both of these are of numeric type, but their actual numeric value is implementation defined and thus meaningless in any context other than a date or time function. In other words, a form calculation obtains a date value from a date function, performs some arithmetic on that date value, and only passes that value to another date function. These same rules apply to time values.

Both date values and time values have an associated origin or **epoch** -- a moment in time when things began. Any date value prior to its epoch is invalid, as is, any time value prior to its epoch.

The unit of value for all date function is the number of days since the epoch. The unit of value for all time functions is the number of milliseconds since the epoch.

The reference implementation defines the epoch for all date functions such that day 1 is Jan 1, 1900, and defines the epoch for all time functions such that millisecond 1 is midnight, 00:00:00, GMT. This means negative time values may be returned to users in timezones east of Greenwich Mean Time.

10.6 Date()

This function returns the current system date as the number of days since the epoch.

10.6.1 Definition

```
Date()
```

10.6.2 Returns

The number of days for the current date.

10.6.3 Examples

```
Date()
```

returns 35733 on Oct 31 1998.

10.7 Date2Num()

This function returns the number of days since the epoch, given a date string.

10.7.1 Definition

```
Date2Num(d1[, f1[, k1]])
```

10.7.2 Parameters

d1
: is a date string in the format given by f1, governed by the locale given by k1.

f1
: is a date format string.
If f1 is omitted, the default date format is used.

k1
: is a locale identifier string conforming to the locale naming standards defined above.
If k1 is omitted, the ambient locale is used.

10.7.3 Returns

The days since the epoch or null if any of its parameters are null.

If the given date is not in the format given, or the format is invalid, or the locale is invalid, the function returns 0.

Sufficient information must be provided to determine a unique day since the epoch: if any of the day of the year and year of the era are missing, or any of the day of the month, month of the year and year of the era are missing, the function returns 0.

10.7.4 Examples

```
Date2Num("Mar 15, 1996")
```

returns 35138>.

```
Date2Num("1/1/1900", "D/M/YYYY")
```

returns 1.

```
Date2Num("03/15/96", "MM/DD/YY")
```

returns 35138.

```
Date2Num("Aug 1,1996", "MMM D,YYYY")
```

returns 35277.

```
Date2Num("31-ago-96", "DD-MMM-YY", "es_ES")
```

returns 35307.

```
Date2Num("1/3/00", "D/M/YY") - Date2Num("1/2/00", "D/M/YY")
```

returns 29, year 2000 being a leap year!

10.7.5 See Also

- Num2Date(), and DateFmt().

10.8 Num2Date()

This function returns a date string, given a number of days since the epoch.

10.8.1 Definition

```
Num2Date(n1 [,f1 [, k1]])
```

10.8.2 Parameters

n1
: is the number of days.

f1
: is a date format string.
If f1 is omitted, the default date format is used.

k1
: is a locale identifier string conforming to the locale naming standards defined above.
If k1 is omitted, the ambient locale is used.

10.8.3 Returns

The date string or null if any of its parameters are null.

The formatted date is in the format given in f1, governed by the locale given in k1.

If the given date is invalid, the function returns an empty string.

10.8.4 Examples

```
Num2Date(1, "DD/MM/YYYY")
```

returns "01/01/1900".

```
Num2Date(35139, "DD-MMM-YYYY", "de_CH")
```

returns "16-Mrz-1996".

```
Num2Date(Date2Num("31-ago-98", "DD-MMM-YY", "es_ES") - 31, "D' de 'MMMM' de
'YYYY", "pt_BR")
```

returns "31 de Julho de 1998".

10.8.5 See Also

- `Date2Num()`, `DateFmt()`, and `Date()`.

10.9 DateFmt()

This function returns a date format string, given a date format style.

10.9.1 Definition

```
DateFmt([n1[, k1]])
```

10.9.2 Parameters

n1
: is an integer identifying the date format style:
- if the value is 0, the locale specific default-style date format is requested.
- if the value is 1, the locale specific short-style date format is requested.
- if the value is 2, the locale specific medium-style date format is requested.
- if the value is 3, the locale specific long-style date format is requested.
- if the value is 4, the locale specific full-style for date format is requested.

If n1 is omitted, the default style value 0 is used.

k1
: is a locale identifier string conforming to the locale naming standards defined above.
If k1 is omitted, the default locale is used.

10.9.3 Returns

The date format string or null if any of its mandatory parameters are null.

If the given format style is invalid, the function returns default-style date format.

10.9.4 Examples

```
DateFmt()
```

returns "MMM D, YYYY". This is the default date format .

```
DateFmt(1)
```

returns "M/D/YY".

```
DateFmt(2, "fr_CA")
```

returns "YY-MM-DD".

```
DateFmt(3, "de_DE")
```

returns "D. MMMM YYYY".

```
DateFmt(4, "es_ES")
```

returns "EEEE D' de 'MMMM' de 'YYYY".

10.9.5 See Also

- `Num2Date()`, `Date2Num()`, and `LocalDateFmt()`.

10.10 LocalDateFmt()

This function returns a localized date format string, given a date format style.

10.10.1 Definition

```
LocalDateFmt([n1[, k1]])
```

10.10.2 Parameters

n1

is an integer identifying the date format style:
- if the value is 0, the locale specific default-style localized date format is requested.
- if the value is 1, the locale specific short-style localized date format is requested.
- if the value is 2, the locale specific medium-style localized date format is requested.
- if the value is 3, the locale specific long-style localized date format is requested.

- if the value is 4, the locale specific full-style localized date format is requested.
If n1 is omitted, the default style value 0 is used.

k1

is a locale identifier string conforming to the locale naming standards defined above.
If k1 is omitted, the ambient locale is used.

10.10.3 Returns

The localized date format or null if any of its parameters are null.

If the given format style is invalid, the function returns default-style localized date format.

The date format strings returned by this function are not usable in the functions `Date2Num()` and `Num2Date()`.

10.10.4 Examples

```
LocalDateFmt(1, "de_DE")
```

returns "tt.MM.uu".

```
LocalDateFmt(2, "fr_CA")
```

returns "aa-nn-jj".

```
LocalDateFmt(3, "de_CH")
```

returns "t. MMMM uuuu".

```
LocalDateFmt(4, "es_ES")
```

returns "EEEE t' de 'MMMM' de 'uuuu".

10.10.5 See Also

- `DateFmt()`.

10.11 Time()

This function returns the current system time as the number of milliseconds since the epoch.

10.11.1 Definition

```
Time()
```

10.11.2 Returns

The number of milliseconds for the current time.

10.11.3 Examples

```
Time()
```

returns 61200001 at precisely noon to a user in Boston.

10.12 Time2Num()

This function returns the number of milliseconds since the epoch, given a time string.

10.12.1 Definition

```
Time2Num(d1[, f1[, k1]])
```

10.12.2 Parameters

d1
: is a time string in the format given by f1, governed by the locale given by k1.

f1
: is a time format string, as defined above.
If f1 is omitted, the default time format is used.

k1
: is a locale identifier string conforming to the locale naming standards defined above.
If k1 is omitted, the ambient locale is used.

10.12.3 Returns

The milliseconds from the epoch or null if any of its parameters are null.

If the time string does not include a timezone, the current timezone is used.

The locale is used to parse any timezone names.

If the given time is not in the format given, or the format is invalid, or the locale is invalid, the function returns 0.

Sufficient information must be provided to determine a second since the epoch: if any of the hour of the meridiem, minute of the hour, second of the minute, and meridiem are missing, or any of the hour of the day, minute of the hour, and second of the minute are missing, the function returns 0.

10.12.4 Examples

```
Time2Num("00:00:00 GMT", "HH:MM:SS Z")
```

returns 1.

```
Time2Num("1:13:13 PM")
```

returns 76393001 to a user in California on Standard Time, and 76033001 when that same user is on Daylight Savings Time.

```
    (Time2Num("13:13:13", "HH:MM:SS") - Time2Num("13:13:13 GMT", "HH:MM:SS Z")) /
(60 * 60 * 1000)
```

returns 8 to a user in Vancouver and returns 5 to a user in Ottawa when on Standard Time. On Daylight Savings Time, the returned values are returns 7 and 4, respectively.

```
Time2Num("1.13.13 dC GMT+01:00", "h.MM.SS A Z", "it_IT")
```

returns 43993001.

10.12.5 See Also

- `Num2Time()`, and `TimeFmt()`.

10.13 Num2GMTime()

This function returns a GMT time string, given a number of milliseconds from the epoch.

10.13.1 Definition

```
Num2GMTime(n1 [,f1 [, k1]])
```

10.13.2 Parameters

n1
: is the number of milliseconds.

f1
: is a time format string, as defined above.
If f1 is omitted, the default time format is used.

k1
: is a locale identifier string conforming to the locale naming standards defined above.
If k1 is omitted, the ambient locale is used.

10.13.3 Returns

The GMT time string or null if any of its parameters are null.

The formatted time is in the format given in f1, governed by the locale given in k1.

The locale is used to format any timezone names.

If the given time is invalid, the function returns an empty string.

10.13.4 Examples

```
Num2GMTime(1, "HH:MM:SS")
```

returns "00:00:00".

```
Num2GMTime(65593001, "HH:MM:SS Z")
```

returns "18:13:13 GMT".

```
Num2GMTime(43993001, TimeFmt(4, "de_CH"), "de_CH")
```

returns "12.13 Uhr GMT".

10.13.5 See Also

- Num2Time().

10.14 Num2Time()

This function returns a time string, given a number of milliseconds from the epoch.

10.14.1 Definition

```
Num2Time(n1 [,f1 [, k1]])
```

10.14.2 Parameters

n1
: is the number of milliseconds.

f1
: is a time format string, as defined above.
If f1 is omitted, the default time format is used.

k1
: is a locale identifier string conforming to the locale naming standards defined above.
If k1 is omitted, the ambient locale is used.

10.14.3 Returns

The time string or null if any of its parameters are null.

The formatted time is in the format given in f1, governed by the locale given in k1.

The locale is used to format any timezone names.

If the given time is invalid, the function returns an empty string.

10.14.4 Examples

```
Num2Time(1, "HH:MM:SS")
```

returns "00:00:00" in Greenwich, England and "09:00:00" in Tokyo.

```
Num2Time(65593001, "HH:MM:SS Z")
```

returns "13:13:13 EST" in Boston.

```
Num2Time(65593001, "HH:MM:SS Z", "de_CH")
```

returns "13:13:13 GMT-05:00" to a German Swiss user in Boston.

```
Num2Time(43993001, TimeFmt(4, "de_CH"), "de_CH")
```

returns "13.13 Uhr GMT+01:00" to a user in Zurich.

```
Num2Time(43993001, "HH:MM:SSzz")
```

returns "13:13+01:00" to that same user in Zurich.

10.14.5 See Also

- Date2Num(), DateFmt(), and Date().

10.15 TimeFmt()

This function returns a time format given a time format style.

10.15.1 Definition

```
TimeFmt([n1[, k1]])
```

10.15.2 Parameters

n1
is an integer identifying the time format style:
- if the value is 0, the locale specific default-style time format is requested.
- if the value is 1, the locale specific short-style time format is requested.
- if the value is 2, the locale specific medium-style time format is requested.
- if the value is 3, the locale specific long-style time format is requested.
- if the value is 4, the locale specific full-style time format is requested.

If n1 is omitted, the default style value 0 is used.

k1
is a locale identifier string conforming to the locale naming standards defined above.
If k1 is omitted, the ambient locale is used.

10.15.3 Returns

The time format or null if any of its parameters are null.

If the given format style is invalid, the function returns default-style time format.

10.15.4 Examples

```
TimeFmt()
```

returns "h:MM:SS A".

```
TimeFmt(1)
```

returns "h:MM A".

```
TimeFmt(2, "fr_CA")
```

returns "HH:MM:SS".

```
TimeFmt(4, "de_DE")
```

returns "H.MM' Uhr 'Z".

10.15.5 See Also

- `Time()`.

10.16 LocalTimeFmt()

This function returns a localized time format string, given a time format style.

10.16.1 Definition

```
LocalTimeFmt([n1[, k1]])
```

10.16.2 Parameters

n1
is an integer identifying the time format style:
- if the value is 0, the locale specific default-style localized time format is requested.
- if the value is 1, the locale specific short-style localized time format is requested.
- if the value is 2, the locale specific medium-style localized time format is requested.
- if the value is 3, the locale specific long-style localized time format is requested.
- if the value is 4, the locale specific full-style localized time format is requested.

If n1 is omitted, the default style value 0 is used.

k1
is a locale identifier string conforming to the locale naming standards defined above.
If k1 is omitted, the ambient locale is used.

10.16.3 Returns

The localized time format or null if any of its parameters are null.

If the given format style is invalid, the function returns default-style localized time format.

The time format strings returned by this function are not usable in the functions `Time2Num()` and `Num2Time()`.

10.16.4 Examples

```
LocalTimeFmt(1, "de_DE")
```

returns "HH:mm".

```
LocalTimeFmt(2, "fr_CA")
```

returns "HH:mm:ss".

```
LocalTimeFmt(3, "de_DE")
```

returns "HH:mm:ss z".

```
LocalTimeFmt(4, "es_ES")
```

returns "HH'H'mm'"' z".

10.16.5 See Also

- `TimeFmt()`.

10.17 IsoDate2Num()

This function returns the number of days since the epoch, given an [ISO8601] date string.

10.17.1 Definition

```
IsoDate2Num(d1)
```

10.17.2 Parameters

d1

is a date string in one of the following two formats:
YYYY[MM[DD]]
YYYY[-MM[-DD]]
or, is an ISO-8601 date-time string -- the concatenation of an ISO-8601 date string with an ISO-8601 time string, separated by the character T, as in:
1997-07-16T20:20:20

10.17.3 Returns

The days from the epoch or null if its parameter is null.

If the given date is not in one of the accepted formats, the function returns 0.

10.17.4 Examples

```
IsoDate2Num("1900")
```

returns 1.

```
IsoDate2Num("1900-01")
```

returns 1.

```
IsoDate2Num("1900-01-01")
```

returns 1.

```
IsoDate2Num("19960315T20:20:20")
```

returns 35138.

```
IsoDate2Num("2000-03-01") - IsoDate2Num("20000201")
```

returns 29.

10.17.5 See Also

- `IsoTime2Num()`, and `Num2Date()`.

10.18 IsoTime2Num()

This function returns the number of milliseconds since the epoch, given an [ISO8601] time string.

10.18.1 Definition

```
IsoTime2Num(d1)
```

10.18.2 Parameters

d1

is a time string in one of the following formats:
HH[MM[SS[.FFF][z]]]
HH[MM[SS[.FFF][+HH[MM]]]]
HH[MM[SS[.FFF][-HH[MM]]]]
HH[:MM[:SS[.FFF][z]
HH[:MM[:SS[.FFF][-HH[:MM]]]]
HH[:MM[:SS[.FFF][+HH[:MM]]]]
or, is an ISO-8601 date-time string -- the concatenation of an ISO-8601 date string with an ISO-8601 time string, separated by the character T, as in:
1997-07-16T20:20:20

10.18.3 Returns

The number of milliseconds from the epoch or null if its parameter is null.

If the time string does not include a timezone, the current timezone is used.

If the given time is not in a valid format, the function returns 0.

10.18.4 Examples

```
IsoTime2Num("00:00:00Z")
```

returns 1.

```
IsoTime2Num("13")
```

returns 64800001 to a user in Boston.

```
IsoTime2Num("13:13:13")
```

returns 76393001 to a user in California.

```
IsoTime2Num("19111111T131313+01")
```

returns 43993001.

10.18.5 See Also

- `IsoDate2Num()`, and `Num2Time()`.

11 Financial Built-in Functions

Note: the value of the results in the examples of this section have all been rounded for presentation purposes. A number followed by an * indicates a rounded return value.

11.1 Apr()

This function returns the annual percentage rate for a loan.

11.1.1 Definition

```
Apr(n1, n2, n3)
```

11.1.2 Parameters

n1
: is the principal amount of the loan.

n2
: is the payment on the loan.

n3
: is the number of periods.

11.1.3 Returns

The annual percentage rate or null if any of its parameters are null.

If any of n1, n2, or n3 are non-positive, the function generates an error exception.

11.1.4 Examples

```
Apr(35000, 269.50, 30 * 12)
```

returns 0.085* (8.5%) which is the annual interest rate on a loan of $35,000 being repaid at $269.50 per month over 30 years.

11.2 CTerm()

This function returns the number of periods needed for an investment earning a fixed, but compounded, interest rate to grow to a future value.

11.2.1 Definition

```
CTerm(n1, n2, n3)
```

11.2.2 Returns

The number of periods or null if any of its parameters are null.

If any of n1, n2, or n3 are non-positive, the function generates an error exception.

11.2.3 Parameters

n1
: is the interest rate per period.

n2
: is the future value of the investment.

n3
: is the amount of the initial investment.

11.2.4 Examples

```
CTerm(.02, 200, 100)
```

returns 35*, which is the required period for $100 invested at 2% to grow to $200.

11.3 FV()

This function returns the future value of periodic constant payments at a constant interest rate.

11.3.1 Definition

```
FV(n1, n2, n3)
```

11.3.2 Parameters

n1
: is the amount of each equal payment.

n2
: is the interest rate per period.

n3
: is the total number of periods.

11.3.3 Returns

The future value or null if any of its parameters are null.

If n1 or n3 are non-positive, or if n2 is negative, the function generates an error exception.

If n2 is 0, the function returns the product of n1 and n3, i.e., the payment amount multiplied by the number of payments.

11.3.4 Examples

```
FV(100, .075 / 12, 10 * 12)
```

returns 17793.03*, which is the amount present after paying $100 a month for 10 years in an account bearing an annual interest of 7.5%.

```
FV(1000, 0.01, 12)
```

returns 12682.50*.

11.4 IPmt()

This function returns the amount of interest paid on a loan over a period of time.

11.4.1 Definition

```
IPmt(n1, n2, n3, n4, n5)
```

11.4.2 Parameters

n1
: is the principal amount of the loan.

n2
: is the annual interest rate.

n3
: is the monthly payment.

n4
: is the first month of the computation.

n5
: is the number of months to be computed.

11.4.3 Returns

The interest amount or null if any of its parameters are null.

If any of n1, n2, or n3 are non-positive, the function generates an error exception.

If n4 or n5 are negative, the function generates an error exception.

If the payment is less than the monthly interest load, the function returns 0.

11.4.4 Examples

```
IPmt(30000, .085, 295.50, 7, 3)
```

returns 624.88* which is the amount of interest paid starting in July (month 7) for 3 months on a loan of $30,000.00 at an annual interest rate of 8.5% being repaid at a rate of $295.50 per month.

11.5 NPV()

This function returns the net present value of an investment based on a discount rate, and a series of periodic future cash flows.

11.5.1 Definition

```
NPV(n1, n2 [, ...])
```

11.5.2 Parameters

n1
: is the discount rate over one period.

n2, ...
: are the cash flow values which must be equally spaced in time and occur at the end of each period.

11.5.3 Returns

The net present value rate or null if any of its parameters are null.

The function uses the order of the values n2, ... to interpret the order of the cash flows. Ensure payments and incomes are specified in the correct sequence.

If n1 is non-positive, the function generates an error exception.

11.5.4 Examples

```
NPV(0.15, 100000, 120000, 130000, 140000, 50000)
```

returns 368075.16* which is the net present value of an investment projected to generate $100,000, $120,000, $130,000, $140,000 and $50,000 over each of the next five years and the rate is 15% per annum.

```
NPV(0.10, -10000, 3000, 4200, 6800)
```

returns 1188.44*.

```
NPV(0.08, 8000, 9200, 10000, 12000, 14500)
```

returns 41922.06*.

11.6 Pmt()

This function returns the payment for a loan based on constant payments and a constant interest rate.

11.6.1 Definition

```
Pmt(n1, n2, n3)
```

11.6.2 Parameters

n1
: is the principal amount of the loan.

n2
: is the interest rate per period.

n3
is the number of payment periods.

11.6.3 Returns

The loan payment or null if any of its parameters are null.

If any of n1, n2, or n3 are non-positive, the function generates an error exception.

11.6.4 Examples

```
Pmt(30000.00, .085 / 12, 12 * 12)
```

returns 333.01*, which is the monthly payment for a loan of a $30,000, borrowed at a yearly interest rate of 8.5%, repayable over 12 years (144 months).

```
Pmt(10000, .08 / 12, 10)
```

returns 1037.03*, which is the monthly payment for a loan of a $10,000 loan, borrowed at a yearly interest rate of 8.0%, repayable over 10 months.

11.7 PPmt()

This function returns the amount of principal paid on a loan over a period of time.

11.7.1 Definition

```
PPmt(n1, n2, n3, n4, n5)
```

11.7.2 Parameters

n1
is the principal amount of the loan.
n2
is the annual interest rate.
n3
is the monthly payment.
n4
is the first month of the computation.
n5
is the number of months to be computed

11.7.3 Returns

The principal paid or null if any of its parameters are null.

If any of n1 , n2 , or n3 are non-positive, the function generates an error exception.

If n4 or n5 are negative, the function generates an error exception.

If payment is less than the monthly interest load, the function generates an error exception.

11.7.4 Examples

```
PPmt(30000, .085, 295.50, 7, 3)
```

returns 261.62*, which is the amount of principal paid starting in July (month 7) for 3 months on a loan of $30,000 at an annual interest rate of 8.5%, being repaid at $295.50 per month. The annual interest rate is used in the function because of the need to calculate a range within the entire year.

11.8 PV()

This function returns the present value of an investment of periodic constant payments at a constant interest rate.

11.8.1 Definition

```
PV(n1, n2, n3)
```

11.8.2 Parameters

n1
: is the amount of each equal payment.

n2
: is the interest rate per period.

n3
: is the total number of periods.

11.8.3 Returns

The present value or null if any of its parameters are null.

If any of n1 and n3 are non-positive, the function generates an error exception.

11.8.4 Examples

```
PV(1000, .08 / 12, 5 * 12)
```

returns 49318.43* which is the present value of $1000.00 invested at 8% for 5 years.

```
PV(500, .08 / 12, 20 * 12)
```

returns 59777.15*.

11.9 Rate()

This function returns the compound interest rate per period required for an investment to grow from present to future value in a given period.

11.9.1 Definition

```
Rate(n1, n2, n3)
```

11.9.2 Parameters

n1
: is the future value.

n2
: is the present value.

n3
: is the total number of periods.

11.9.3 Returns

The compound rate or null if any of its parameters are null.

If any of n1, n2 , or n3 are non-positive, the function generates an error exception.

11.9.4 Examples

```
Rate(110, 100, 1)
```

returns 0.10 which is what the rate of interest must be for and investment of $100 to grow to $110 if invested for 1 term.

11.10 Term()

This function returns the number of periods needed to reach a given future value from periodic constant payments into an interest bearing account.

11.10.1 Definition

```
Term(n1, n2, n3)
```

11.10.2 Parameters

n1
: is the payment amount made at the end of each period.

n2
: is the interest rate per period.

n3
: is the future value.

11.10.3 Returns

The number of periods or null if any of its parameters are null.

If any of n1, n2, or n3 are non-positive, the function generates an error exception.

11.10.4 Examples

```
Term(475, .05, 1500)
```

returns 3* which is the number of periods for an investment of $475, deposited at the end of each period into an account bearing 5% compound interest, to grow to $1500.00.

12 Logical Built-in Functions

Most of these logical functions return the boolean results true or false, represented by the numeric values of 1 and 0, respectively.

Some of the following built-in function examples make use of the identifier $ to mean a reference to the value of the object to which the form calculation is bound; this object is typically called the referencing object.

12.1 Choose()

This function selects a value from a given set of parameters.

12.1.1 Definition

```
Choose(n1, s1 [, s2...])
```

12.1.2 Parameters

n1
: is n'th value to select from the set.

s1
: is the first value of the set.

s2, ...
: are optional additional value of the set.

12.1.3 Returns

The selected argument or null if its first parameter is null.

If n1 is less than 1 or greater than the number of arguments in the set, the function returns an empty string.

12.1.4 Examples

```
Choose(3, "Accounting", "Administration", "Personnel", "Purchasing")
```

returns "Personnel".

```
Choose(Quantity, "A", "B", "C")
```

returns B if the value in Quantity is 2.

12.2 Oneof()

This logical function returns true if a value is in a given set.

12.2.1 Definition

```
Oneof(s1, s2 [, s3...])
```

12.2.2 Parameters

s1
: is the value to match.

s2
: is the first value in the set.

s3, ...
: are optional additional values in the set.

12.2.3 Returns

True (1) if the first parameter is in the set, false (0) if it is not in the set.

12.2.4 Examples

```
Oneof($, 4, 13, 24)
```

returns true (1) if the current object has a value of 4, 13 or 24; otherwise it returns false (0).

```
Oneof(Item, null, "A", "B", "C")
```

returns true (1) if the value in the object Item is null, "A", "B" or "C"; otherwise it returns false (0).

12.3 Within()

This logical function returns true if a value is within a given range.

12.3.1 Definition

```
Within(s1, s2, s3)
```

12.3.2 Parameters

s1
: is the value to test.

s2
: is the lower bound of the range.

s3
: is the upper bound of the range.

12.3.3 Returns

True (1) if the first parameter is within range, false (0) if it is not in range, or null if the first parameter is null.

If the first value is numeric then the ordering comparison is numeric.

If the first value is non-numeric then the ordering comparison uses the collating sequence for the current locale.

12.3.4 Examples

```
Within("C", "A", "D")
```

returns true (1).

```
Within(1.5, 0, 2)
```

returns true (1).

```
Within(-1, 0, 2)
```

returns false (0).

```
Within($, 1, 10)
```

returns true (1) if the value of the current object is between 1 and 10.

12.4 Exists()

Determines if the given parameter is an accessor to an existing object.

12.4.1 Definition

```
Exists(v1)
```

12.4.2 Parameters

v1
is the accessor.

12.4.3 Returns

True (1) if the given parameter is an accessor to (a property of) an object that exists, and false (0), if it does not.

If the given parameter is not an accessor, the function returns false (0).

12.4.4 Examples

```
Exists(Item)
```

returns true (1) if the object `Item` exists, false (0) otherwise.

```
Exists("hello world")
```

returns false (0) -- the string is not an accessor.

```
Exists(Invoice.Border.Edge[1].Color)
```

returns true (1) if the object `Invoice` exists and has a `Border` property, which in turn, has at least one `Edge` property, which in turn, has a `Color` property. Otherwise, it returns false (0).

12.5 HasValue()

Determines if the given parameter is an accessor with a non-null, nonempty, non-blank value.

12.5.1 Definition

```
HasValue(v1)
```

12.5.2 Parameters

v1
is the accessor.

12.5.3 Returns

True (1) if the given parameter is an accessor with a non-null, nonempty, non-blank value. A non-blank value will contain characters other than white spaces.

If the given parameter is not an accessor, the function returns true (1), if its a non-null, non-empty, non-blank value.

12.5.4 Examples

```
HasValue(Item)
```

returns true (1), if the object `Item` exists, and has a non-null, nonempty, non-blank value. Otherwise, it returns false (0).

```
HasValue(" ")
```

returns false (0).

```
HasValue(0)
```

returns true (1).

13 Miscellaneous Built-in Functions

This list of functions may grow over time, for instance to include to retrieve system or application specific properties, but for now, its limited to the following:

13.1 UnitValue()

Returns the value of a unitspan after an optional unit conversion. A unitspan string consist of a number immediately followed by a unit name. Recognized unit names include:

"in", "inches",
"mm", "millimeters",
"cm", "centimeters",
"pt", "picas", "points",
"mp", "millipoints".

13.1.1 Definition

```
UnitValue(s1 [, s2])
```

13.1.2 Parameters

s1
is a unitspan string.

s2
is an optional string containing a unit name. The unitspan's value will be converted to the given units. If s2 is omitted, the unitspan's units are used.

13.1.3 Returns

The unitspan's value.

13.1.4 Examples

```
UnitValue("1in", "cm")
```

returns 2.54.

```
UnitValue("72pt", "in")
```

returns 1.

13.2 UnitType()

Returns the units of a unitspan.

13.2.1 Definition

```
UnitType(s1)
```

13.2.2 Parameters

s1
is a unitspan string.

13.2.3 Returns

The unitspan's units. Unit names are canonized to one of the following:

"in",
"mm",
"cm",
"pt",
"mp".

13.2.4 Examples

```
UnitType("36in")
```

returns "in".

```
UnitValue("2.54centimeters")
```

returns "cm".

14 String Built-in Functions

FormCalc provides a large number of functions to operate on the content of strings, including the ability to:

- retrieve parts of a string
- insert parts of a string
- delete parts of a string

Many of these functions require a numeric position argument. All strings are indexed starting at character position one; i.e., character position 1 is the first character of the array. The last character position coincides with the length of the string.

Any character position less than one refers to the first character string, and any character position greater than the length of the string refers to the last character of the string.

14.1 At()

This function locates the starting character position of string s2 within string s1.

14.1.1 Definition

```
At(s1, s2)
```

14.1.2 Parameters

s1
is the source string.

s2
is the string to search for.

14.1.3 Returns

The character position of the start of s2 within s1 or null if any of its parameters are null.

If string s2 is not in s1, the function returns 0.

If string s2 is empty, the function returns 1.

14.1.4 Examples

```
At("ABC", "AB")
```

returns 1.

```
At("ABCDE", "DE")
```

returns 4.

```
At("WXYZ", "YZ")
```

returns 3.

```
At("123999456", "999")
```

returns 4.

14.2 Concat()

This function returns the string concatenation of a given set of strings.

14.2.1 Definition

```
Concat(s1 [, s2...])
```

14.2.2 Parameters

s1
: is the first string in the set.

s2,...
: are additional strings to append from the set.

14.2.3 Returns

The concatenated string or null if all of its parameters are null.

14.2.4 Examples

```
Concat("ABC", "CDE")
```

returns "ABCCDE".

```
Concat("XX", Item, "-01")
```

returns "XXABC-01" if the value of `Item` is "ABC".

14.3 Decode()

This function returns the decoded version of a given string.

14.3.1 Definition

```
Decode(s1 [, s2])
```

14.3.2 Parameters

s1
: is the string to be decoded.

s2
: is a string identifying the type of decoding to perform:
- if the value is "url", the string will be URL decoded.
- if the value is "html", the string will be HTML decoded.
- if the value is "xml", the string will be XML decoded.
If s2 is omitted, the string will be URL decoded.

14.3.3 Returns

The decoded string.

14.3.4 Examples

```
Decode("%ABhello,%20world!%BB", "url")
```

returns "«hello, world!»".

```
Decode("&AElig;&Aacute;&Acirc;&Aacute;&Acirc;", "html")
```

returns "ÆÁÂÁÂ".

```
Decode("~!@#$%^&amp;*()_+|`{&quot;}[]&lt;&gt;?,./;&apos;:", "xml")
```

returns "~!@#$%^&*()_+|`{""}[]<>?,./;':".

14.3.5 See Also

- Encode().

14.4 Encode()

This function returns the encoded version of a given string.

14.4.1 Definition

```
Encode(s1 [, s2])
```

14.4.2 Parameters

s1
: is the string to be encoded.

s2
: is a string identifying the type of encoding to perform:
- if the value is "url", the string will be URL encoded.
- if the value is "html", the string will be HTML encoded.

- if the value is "xml", the string will be XML encoded.
If s2 is omitted, the string will be URL encoded.

14.4.3 Returns

The encoded string.

14.4.4 Examples

```
Encode("""hello, world!""", "url")
```

returns "%22hello,%20world!%22".

```
Encode("ÁÂÃÄÅÆ", "html")
```

returns the HTML encoding "ÁÂÃÄÅÆ".

14.4.5 See Also

- Decode().

14.5 Format()

This function formats the given data according to the given picture.

14.5.1 Definition

```
Format(s1, s2[, s3...])
```

14.5.2 Parameters

s1
is the picture format string, which may be a locale-sensitive picture clause. For a complete specification of picture formats, refer to the XFA-Picture Clause Specification.

s2
is the source data being formatted.

s3, ...
is any additional source data being formatted.

For date picture formats, the source data must be an ISO date string in one of two formats:

YYYY[MM[DD]]
YYYY[-MM[-DD]]
or, be an ISO date-time string.

For time picture formats, the source data must be an ISO time string in one of the following formats:

HH[MM[SS[.FFF][z]]]
HH[MM[SS[.FFF][+HH[MM]]]]

HH[MM[SS[.FFF][-HH[MM]]]]
HH[:MM[:SS[.FFF][z]
HH[:MM[:SS[.FFF][-HH[:MM]]]]
HH[:MM[:SS[.FFF][+HH[:MM]]]]

or, be an ISO date-time string.

For date-time picture formats, the source data must be an ISO date-time string.

For numeric picture formats, the source data must be numeric.

For text picture formats, the source data must be textual.

For compound picture formats, the number of source data arguments must match the number of sub elements in the picture.

14.5.3 Returns

The formatted data as a string, or an empty string if unable to format the data.

14.5.4 Examples

```
Format("MMM D, YYYY", "20020901")
```

returns Sep 1, 2002.

```
Format("$9,999,999.99", 1234567.89)
```

returns $1,234,567.89 in the US and €1 234 567,89 in France.

14.5.5 See Also

- IsoDate2Num(),
- IsoTime2Num(), and
- Parse().

14.6 Left()

This function extracts a number of characters from a given string, starting with the first character on the left.

14.6.1 Definition

```
Left(s1, n1)
```

14.6.2 Parameters

s1
is the string to extract from.

n1

is the number of characters to extract.

14.6.3 Returns

The extracted string or null if any of its parameters are null.

If the number of characters to extract is greater than the length of the string, the function returns the whole string.

If the number of characters to extract is 0 or less, the function returns the empty string.

14.6.4 Examples

```
Left("ABCD", 2)
```

returns "AB".

```
Left("ABCD", 10)
```

returns "ABCD".

```
Left("XYZ-3031", 3)
```

returns "XYZ".

14.7 Len()

This function returns the number of characters in a given string.

14.7.1 Definition

```
Len(s1)
```

14.7.2 Parameters

s1
is the string to be evaluated.

14.7.3 Returns

The length or null if its parameter is null.

14.7.4 Examples

```
Len("ABC")
```

returns 3.

```
Len("ABCDEFG")
```

returns 7.

14.8 Lower()

This function returns a string where all given uppercase characters are converted to lowercase.

14.8.1 Definition

```
Lower(s1[, k1])
```

14.8.2 Parameters

s1
: is the string to be converted.

k1
: is a locale identifier string conforming to the locale naming standards.
If k1 is omitted, the ambient locale is used.

14.8.3 Returns

The lowercased string or null if any of its mandatory parameters are null.

In some locales, there are alphabetic characters that do not have an lowercase equivalent.

14.8.4 Bugs

The current implementation limits the operation of this function to the Latin1 subrange of the Unicode 2.1 character set. Characters outside this subrange are never converted.

14.8.5 Examples

```
Lower("Abc123X")
```

returns "abc123x".

```
Lower("ÀBÇDÉ")
```

returns "àbçdé".

14.9 Ltrim()

This function returns a string with all leading white space characters removed.

14.9.1 Definition

```
Ltrim(s1)
```

14.9.2 Parameters

s1
is the string to be trimmed.

14.9.3 Returns

The trimmed string or null if its parameter is null.

White space characters includes the ASCII space, horizontal tab, line feed, vertical tab, form feed and carriage return, as well as, the Unicode space characters (Unicode category Zs).

14.9.4 Examples

```
Ltrim(" ABC")
```

returns "ABC".

```
Ltrim(" XY ABC")
```

returns "XY ABC".

14.10 Parse()

This function parses the given data according to the given picture.

14.10.1 Definition

```
Parse(s1, s2)
```

14.10.2 Parameters

s1
is a picture format string. For a specification of picture formats refer to the XFA-Picture Clause Specification.

s2
is the string data being parsed.

14.10.3 Returns

The parsed data as string, or the empty string if unable to parse the data.

A successfully parsed date picture format is returned as an ISO date string of the form YYYY-MM-DD.

A successfully parsed time picture format is returned as an ISO time string of the form: HH:MM:SS.

A successfully parsed date-time picture format is returned as an ISO date-time string of the form: YYYY-MM-DDTHH:MM:SS.

A successfully parsed numeric picture format is returned as a number.

A successfully parsed text pictures is format returned as text.

14.10.4 Examples

```
Parse("MMM D, YYYY", "Sep 1, 2002")
```

returns 2002-09-01.

```
Parse("$9,999,999.99", "$1,234,567.89")
```

returns 1234567.89 in the US.

14.10.5 See Also

- Format().

14.11 Replace()

This function replaces all occurrences of one string with another within a given string.

14.11.1 Definition

```
Replace(s1, s2[, s3])
```

14.11.2 Parameters

s1
: is the source string.

s2
: is the string to be replaced.

s3
: is the replacement string.
If s3 is omitted or null, the empty string is used.

14.11.3 Returns

The replaced string or null if any of its mandatory parameters are null.

14.11.4 Examples

```
Replace("it's a dog's life", "dog", "cat")
```

returns the string "it's a cat's life".

```
Replace("it's a dog's life", "dog's ")
```

returns the string "it's a life".

14.12 Right()

This function extracts a number of characters from a given string, beginning with the last character on the right.

14.12.1 Definition

```
Right(s1, n1)
```

14.12.2 Parameters

s1
: is the string to be extract from.

n1
: is the number of characters to extract.

14.12.3 Returns

The extracted string or null if any of its parameters are null.

If the number of characters to extract is greater than the length of the string, the function returns the whole string.

If the number of characters to extract is 0 or less, the function returns the empty string.

14.12.4 Examples

```
Right("ABC", 2)
```

returns "BC".

```
Right("ABC", 10)
```

returns "ABC".

```
Right("XYZ-3031", 4)
```

returns "3031".

14.13 Rtrim()

This function returns a string with all trailing white space characters removed.

14.13.1 Definition

```
Rtrim(s1)
```

14.13.2 Parameters

s1
is the string to be trimmed.

14.13.3 Returns

The trimmed string or null if any of its parameters are null.

White space characters includes the ASCII space, horizontal tab, line feed, vertical tab, form feed, and carriage return, as well as, the Unicode space characters (Unicode category Zs).

14.13.4 Examples

```
Rtrim("ABC ")
```

returns "ABC".

```
Rtrim("XYZ ABC ")
```

returns "XYZ ABC".

14.14 Space()

This function returns a string consisting of a given number of blank spaces.

14.14.1 Definition

```
Space(n1)
```

14.14.2 Parameters

n1
is the number of spaces to generate.

14.14.3 Returns

The blank string or null if its parameter is null.

14.14.4 Examples

```
Concat("Hello ", null, "world.")
```

returns "Hello world.".

```
Concat(FIRST, Space(1), LAST)
```

returns "Gerry Pearl" when the value of the object FIRST is Gerry, and the value of the object LAST is Pearl.

14.15 Str()

This function converts a number to a character string.

14.15.1 Definition

```
Str(n1 [, n2 [, n3]])
```

14.15.2 Parameters

n1
: is the number to convert.

n2
: is the maximal width of the string; if omitted, a value of 10 is used as the default width.

n3
: is the precision -- the number of digits to appear after the decimal point; if omitted, or negative, 0 is used as the default precision.

14.15.3 Returns

The formatted number or null if any of its mandatory parameters are null.

The number is formatted to the specified width and rounded to the specified precision; the number may have been zero-padded on the left of the decimal to the specified precision. The decimal radix character used is the dot (.) character; it is always independent of the ambient locale .

If the resulting string is longer than the maximal width of the string, as defined by n2, then the function returns a string of '*' (asterisk) characters of the specified width.

14.15.4 Examples

```
Str(2.456) returns "       2".
Str(4.532, 6, 4) returns "4.5320".
Str(31.2345, 4, 2) returns "****".
```

14.15.5 See Also

- `Format()`.

14.16 Stuff()

This function inserts a string into another string.

14.16.1 Definition

```
Stuff(s1, n1, n2[, s2])
```

14.16.2 Parameters

s1
is the source string.

n1
is the character position in string s1 to start stuffing.
If n1 is less than one, the first character position is assumed.
If n1 is greater than then length of s1, the last character position is assumed

n2
is the number of characters to delete from string s1, starting at character position n1.
If n2 is less than or equal to 0, 0 characters are assumed.

s2
is the string to insert into s1.
If s2 is omitted or null, the empty string is used.

14.16.3 Returns

The stuffed string or null if any of its mandatory parameters are null.

14.16.4 Examples

```
Stuff("ABCDE", 3, 2, "XYZ")
```

returns "ABXYZE".

```
Stuff("abcde", 4, 1, "wxyz")
```

returns "abcwxyze".

```
Stuff("ABCDE", 2, 0, "XYZ")
```

returns "AXYZBCDE".

```
Stuff("ABCDE", 2, 3)
```

returns "AE".

14.17 Substr()

This function extracts a portion of a given string.

14.17.1 Definition

```
Substr(s1, n1, n2)
```

14.17.2 Parameters

s1
is the string to be evaluated.

n1
is the character position in string s1 to start extracting.
If n1 is less than one, the first character position is assumed.
If n1is greater than then length of s1, the last character position is assumed

n2
is the number of characters to extract.
If n2 is less than or equal to 0, 0 characters are assumed.

14.17.3 Returns

The sub string or null if any of its parameters are null.

If n1 + n2 is greater than the length of s1 then the function returns the sub string starting a position n1 to the end of s1 .

14.17.4 Examples

```
Substr("ABCDEFG", 3, 4)
```

returns "CDEF".

```
Substr("abcdefghi", 5, 3)
```

returns "efg".

14.18 Uuid()

This function returns a Universally Unique Identifier (UUID) string which is guaranteed (or at least extremely likely) to be different from all other UUIDs generated until the year 3400 A.D.

14.18.1 Definition

```
Uuid([n1])
```

14.18.2 Parameters

n1
identifies the format of UUID string requested:
- if the value is 0, the returned UUID string will only contain hex octets.
- if the value is 1, the returned UUID string will contain dash characters separating the sequences of hex octets, at fixed positions.
If n1 is omitted, the default value of 0 will be used.

14.18.3 Returns

The string representation of a UUID, which is an optionally dash-separated sequence of 16 hex octets.

14.18.4 Examples

```
Uuid()
```

returns "3c3400001037be8996c400a0c9c86dd5" on some system at some point in time.

```
Uuid(1)
```

returns "1a3ac000-3dde-f352-96c4-00a0c9c86dd5" on that same system at some other point in time.

14.19 Upper()

This function returns a string with all given lowercase characters converted to uppercase.

14.19.1 Definition

```
Upper(s1[, k1])
```

14.19.2 Parameters

s1
: is the string to convert.

k1
: is a locale identifier string conforming to the locale naming standards.
If k1 is omitted, the ambient locale is used.

14.19.3 Returns

The uppercased string or null if any of its mandatory parameters are null.

In some locales, there are alphabetic characters that do not have a lowercase equivalent.

14.19.4 Bugs

The current implementation limits the operation of this function to the Latin1 subrange of the Unicode 2.1 character set. Characters outside this subrange are never converted.

14.19.5 Examples

```
Upper("abc")
```

returns "ABC".

```
Upper("àbCdé")
```

returns "ÀBCDÉ".

14.20 WordNum()

This function returns the English text equivalent of a given number.

14.20.1 Definition

```
WordNum(n1 [, n2 [, k1]])
```

14.20.2 Parameters

n1
is the number to be converted.

n2
identifies the format option as one of the following:
- if the value is 0, the number is converted into text representing the simple number.
- if the value is 1, the number is converted into text representing the monetary value with no fractional digits.
- if the value is 2, the number is converted into text representing the monetary value with fractional digits.
If n2 is omitted, the default value of 0 will be used.

k1
is a locale identifier string conforming to the locale naming standard defined above.
If k1 is omitted, the default en_US locale is used.

14.20.3 Returns

The English text, or null if any of its parameters are null.

If n1 is not numeric or the integral value of n1 is negative or greater than 922,337,203,685,477,550 the function returns "*" (asterisk) characters to indicate an error condition.

14.20.4 Bugs

By specifying an locale identifier other than the default, it should be possible to have this function return something other than English text. However the language rules used to implement this function are inherently English. Thus, for now, the locale identifier is ignored.

14.20.5 Examples

```
WordNum(123.54)
```

returns "One Hundred Twenty-three".

```
WordNum(1011.54, 1)
```

returns "One Thousand Eleven Dollars".

```
WordNum(73.54, 2)
```

returns "Seventy-three Dollars And Fifty-four Cents".

15 URL Built-in Functions

FormCalc provides a number of functions to manipulate the content of URLs, including the ability to:

- download data from a URL,
- upload data to a URL, and
- post data to a URL

These functions are only operational when a protocol host has been provided to the **FormCalc** engine. The list of supported URL protocols (http, https, ftp, file) may thus vary with each protocol hosting environment.

15.1 Get()

This function downloads the contents of the given URL.

15.1.1 Definition

```
Get(s1)
```

15.1.2 Parameters

s1
is the URL being downloaded.

15.1.3 Returns

The downloaded data as a string, or an error exception if unable to download the URL's contents.

15.1.4 Examples

```
Get("http://xtg.can.adobe.com/projects/xfa/formcalc/2.0/")
```

returns this HTML document.

```
Get("ftp://ftp.gnu.org/gnu/GPL")
```

returns a document our Legal Department studies carefully.

```
Get("http://coretech?sql=SELECT+*+FROM+projects+FOR+XML+AUTO,+ELEMENTS")
```

returns the result of an SQL query as an XML document.

15.1.5 See Also

- `Post()`, and `Put()`.

15.2 Post()

This function posts the given data to the given URL.

15.2.1 Definition

```
Post(s1, s2[, s3[, s4[, s5]]])
```

15.2.2 Parameters

s1
is the URL being posted.

s2
is the data being posted.

s3
is an optional string containing the name of the content type of the data being posted. Valid content types include:
- text/html,
- text/xml,
- text/plain,
- multipart/form-data,
- application/x-www-form-urlencoded,
- application/octet-stream, or
- any valid MIME type.

If s3 is omitted, the content type defaults to "application/octet-stream". Note that the application is responsible for ensuring that the posted data is formatted according to the given content type.

s4
is an optional string containing the name of the code page that was used to encode the data being posted. Valid code page names include:
- UTF-8,
- UTF-16,
- ISO8859-1, or
- any recognized [IANA] character encoding.

If s4 is omitted, the code page defaults to "UTF-8". Note that the application is responsible for ensuring that the posted data is encoded according to the given code page.

s5
is an optional string containing any additional HTTP headers to be included in the post. If s5 is omitted, no additional HTTP header is included in the post. Note that when posting to SOAP servers, a "SOAPAction" header is usually required.

15.2.3 Returns

The post response as a string, or an error exception if unable to post the data. The response string will be decoded according to the response's content type. For example, if the server indicates the response is UTF-8 encoded, then this function will UTF-8 decode the response data before returning to the application.

15.2.4 Examples

```
Post("http://tools_build/scripts/jfecho.cgi",
         "user=joe&passwd=xxxxx&date=27/08/2002",
                 "application/x-www-form-urlencoded")
```

posts some urlencoded login data to a server and returns that server's acknowledgement page.

```
var Req = "<?xml version='1.0' encoding='UTF-8'?>"
Req = concat(Req, "<soap:Envelope>")
Req = concat(Req, " <soap:Body>")
Req = concat(Req, "  <getLocalTime/>")
Req = concat(Req, " </soap:Body>")
Req = concat(Req, "</soap:Envelope>")
var Head
Head = "SOAPAction: ""http://www.Nanonull.com/TimeService/getLocalTime"""
var Url
Url = "http://www.nanonull.com/TimeService/TimeService.asmx/getLocalTime"
var Resp = post(Url, Req, "text/xml", "utf-8", Head)
```

posts a SOAP request for the local time to some server, expecting an XML response back.

- Get(), and Put().

15.3 Put()

This function uploads the given data into the given URL.

15.3.1 Definition

```
Put(s1, s2[, s3])
```

15.3.2 Parameters

s1
: is the URL being uploaded.

s2
: is the data being uploaded.

s3
: is an optional string containing the name of the code page that is to be used to encode the data before uploading it. Valid code page names include:
- UTF-8,
- UTF-16,
- ISO8859-1, or
- any recognized [IANA] character encoding.
If s3 is omitted, the code page defaults to "UTF-8".

15.3.3 Returns

The empty string, or an error exception if unable to upload the data.

15.3.4 Examples

```
Put("ftp://www.example.com/pub/fubu.xml",
    "<?xml version='1.0' encoding='UTF-8'?><msg>hello world!</msg>")
```

returns nothing if the ftp server has permitted the user to upload some xml data to the file pub/fubu.xml.

- `Get()`, and `Post()`.

16 Alphabetical Function Listing

17 Bibliography

THIS BIBLIOGRAPHY PROVIDES details on books and documents, from both Adobe Systems and other sources, that are referred to in this specification.

Resources from Adobe Systems Incorporated

All of these resources from Adobe Systems are available on the Adobe Solutions Network (ASN) Developer Program site on the World Wide Web, located at

http://partners.adobe.com/asn/developer/

Document version numbers and dates given in this Bibliography are the latest at the time of publication; more recent versions may be found on the Web site.

The ASN can also be contacted as follows:

Adobe Solutions Network
Adobe Systems Incorporated
345 Park Avenue
San Jose, CA 95110-2704

(800) 685-3510 (from North America)
(206) 675-6145 (from other areas)

acrodevsup@adobe.com

[**Adobe Patent Clarification Notice**]
"*Adobe Patent Clarification Notice*". Available on the Legal Notices page of the ASN Developer Program Web site.

[**XFA-SOM**]
"*Scripting Object Model Expression Specification 2.0*", Adobe Systems Incorporated, October 2003. Available from the http://partner.adobe.com site.

Resources from other sources

[**IANA**]
"Character Sets".
The list is available at http://www.iana.org/assignments/character-sets.

[**IEEE754**]
"IEEE 754: Standard for Binary Floating-Point Arithmetic".
The standard may be purchase through http://grouper.ieee.org/groups/754.

[**ISO639**]
"Code for the representation of names of languages".
The list is available at http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt.

[**ISO3166**]
"ISO 3166 Code (Countries)".
The list is available at http://www.ics.uci.edu/pub/ietf/http/related/iso3166.txt.

[**ISO8601**]
"Data elements and interchange formats -- Information interchange -- Representation of dates and times", ISO 8601:1988.
Available at http://www.iso.ch/markete/8601.pdf.

[**UNICODE**]
"The Unicode Standard, Version 2.1".
Available from http://www.unicode.org/unicode/reports/tr8/
[**RFC1766**]
"Tags for the Identification of Languages", RFC 1766:1995.
Available at http://ietfreport.isoc.org/rfc/rfc1766.txt.