XFL

Extended Formula Language

2005-2009
Written by Bert Haessler
www.nappz.de/xfl
mailto:xfl@nappz.de

Documentation XFL Version 2.85
15th of February 2009

Preliminary remark

The formula language of Lotus Notes is a very simple designed programming language. It is easy to learn but nevertheless possesses a large extend. In this respect it has some benefits compared with LotusScript, e.g. handling lists. Formulas are not only suited for writing agents, actions etc. but also for configuring complex applications. Therefore you can find Notes based workflow management systems which uses the formula language to define processes or actions. Unfortunately you are often confronted with several limits of the formula language. Some of the insufficiencies of the language respectively of its interplay with LotusScript are:

- lack of a possibility to define own formulas*
- no real formula debugging
- not possible to integrate own LotusScript libraries
- subfunctions not programmable
- no direct data interchange between script and formula code
- no jumps and loops programmable**
- complicated reassignment of variables**
- *Evaluate()* with restrictions, e.g. no *@Prompt* possible

* apart from coding dll
** partly possible since Notes R6

Thus the idea occurred to extend the given formula language in some points. The aim was on the one hand to eliminate the insufficiencies mentioned above and on the other hand to provide useful possibilities of influence on the language and its runtime environment to the developer. Furthermore the code should be executable under Notes R4/5. Finally the new language dialect *Extended Formula Language* (XFL) was created. XFL is a language in imitation of the *Standard Formula Language* (SFL) provided by Lotus Notes. XFL is downward compatible to SFL. However there are some new or differing language constructs which are not available in SFL.

XFL language constructs

All SFL constructs can also be used in XFL. You can get the whole scope of SFL in detail reading the help of the Domino Designer. Therefore the following documentation only contains those XFL language constructs which differ from SFL. Some of the features belong to SFL since Notes R6. XFL provides them even for clients with Notes R4/5! Partly the meaning of some constructs is a little different to SFL.

Keywords

DEFINE

Allows you to define functions during runtime. Mostly it is useful to define such application specific functions at initialization time using the global LotusScript variable *XFLInit* (see chapter *Global declared LotusScript variables*).

Syntax
DEFINE *FunctionName* [(*Param1* [; ... [; *ParamN*]])] := *FunctionBody*

Functions can be defined with or without parameters. *FunctionBody* is an XFL expression which is executed when function *FunctionName* is called. *Param1-N* are optional parameters of the function which are available as local variables in *FunctionBody*. The definition of a function via *DEFINE* effects global, i.e. it is kept as long as the script library *XFLEngine* is loaded.

Example 1
Implementation of a sort algorithm. Since Notes R6 there is the function *@Sort()* so that R6 clients can use it. For older clients the function *@Sort()* will be defined as a bubble sort.

```
@If(@TextToNumber(@Version) >= 190 ; "" ;
DEFINE @Sort(list) := (max := @Elements(list);
        @For(i := 1 ; i < max ; i := i + 1;
          @For(k := max ; k>i ; k := k - 1;
               @If(list[k] < list[k-1] ; (t:=list[k];list[k]:=list[k-1];list[k-1] := t) ; "")
                 )
              );
         list)
        );

REM {let's test it};
_l := 4:2:3:1;
@Print(@Sort(_l))
```

Example 2
It is also possible to redefine native @Functions. This statement lets you test a time dependent code to run at different dates.

```
DEFINE @Today := [01.01.2020] ;
@Print(@Today)
```

Example 3
@Functions can be defined with different parameters. Thus XFL supports a kind of function overloading.

```
DEFINE @Today := [01.01.2020] ;
DEFINE @Today(days) := @Adjust(@Today; 0;0;days;0;0;0);
```

UNDEFINE

Deletes a function defined by *DEFINE*.

Syntax

UNDEFINE *FunctionName*

Example

```
DEFINE @Today := [01.01.2020] ;
@Print(@Today);
UNDEFINE @Today;
@Print(@Today)
```

ORIGINAL

If an native @Function is redefined by *DEFINE* the original function can be called using the keyword *ORIGINAL*.

Syntax

ORIGINAL *FunctionName*

Example

Combining *DEFINE* and *ORIGINAL* can extend native @Functions. *@Prompt()* requires at least parameters. This example creates variantions of *@Prompt()* with one or two parameters.

```
DEFINE @Prompt(par1 ; par2 ; par3 ; par4 ; par5) := @If(
 par3 != @Unavailable ;  ORIGINAL @Prompt(par1 ; par2 ; par3 ; par4  ; par5);
 Par2 = "" ; ORIGINAL @Prompt([OK] ; "" ; par1) ;
 ORIGINAL @Prompt([OK] ; par1 ; par2)
);

REM {let's try it out...};
@Prompt([ok] ; "Title" ; "Test"); REM {how boring...};
@Prompt("Title" ; "Test"); REM {[OK] is added automatically};
@Prompt("Test"); REM {This can help you saving characters}
```

GLOBAL

Can stand ahead of a variable and indicates that this is a global variable. Global variables remain in memory after execution of an XFL code in contrast to local variables and can be reused in subsequent *XFLExecute()* calls. Global variables can be accessed via the LotusScript functions *XFLGetGlobalVar()* and *XFLSetGlobalVar()* as well. Because of that global variables are suitable for data exchange between XFL code and LotusScript. Even inside an XFL formula a clear separation of local and global variables can be useful (see example).

Example

```
DEFINE Sqr(x) := x*x;
@Print(Sqr(4));
```

When the expression *Sqr(4)* is executed a local variable *X* is generated which exists only inside the function definition of *Sqr(x)*. This variable *X* would not come into conflict with a local variable *X* inside the main routine. You can easily test this using the debugger.

```
DEFINE Sqr(x) := x*x;
x := 3;
@Debug(1);
@Print(Sqr(4));
```

A completely different result would give the following formula:

```
DEFINE Sqr(x) := x* (GLOBAL x);
GLOBAL x := 3;
@Print(Sqr(4));                 REM {returns 12}
```

Main routine and subfunction access the same global variable.

When you just read a global variable the keyword *GLOBAL* can be omitted unless a local variable of the same name exists.

```
GLOBAL x := 3;
@Print(x);     REM {3};
x := 5;
@Print(x);     REM {5};
@Print(GLOBAL x);     REM {3};
```

When there is an identifier e.g. *X* in the XFL code the interpreter tries to determine the meaning of this identifier at runtime using the following algorithm:

1. Is there a local variable named *X*?
2. Is there a local object named *X*?
3. Is there a global variable named *X*?
4. Is there a global object *X*?
5. Is there a field named *X* in the current document?
6. Is there a user defined function named *X*?

The algorithm stops at the first positive test. If the last test is negative *UNAVAILABLE* is returned. Therefore the first *@Print(x)* in the example above returns the global variable *X*, the second *@Print(x)* returns the local *X*.

LABEL

Defines a label. Can be used in combination with *@Goto()* or *@Gosub()*.

Syntax

LABEL *LabelName*

Example

```
LABEL Start;
@Prompt([OK] ; "Start" ; "This is the beginning");
@If(@Prompt([yesno] ; "endless"; "Again?") ; @Goto(Start) ; "");
```

DEFAULT

Syntax

DEFAULT *fieldName* := *value*

If the current document does not contains the field *fieldName* it is created with the value *value*. Differently from SFL such a expression can be chained.

Example

```
FIELD Prod := Price * DEFAULT Qty := 1
```

If the Field *Qty* does not exists it will created as *1*. After that the product is calculated. If the Field *Qty* already exists its value is taken for multiplication.

OBJECT

Indicates that the following identifier is an object variable.

Syntax

OBJECT *oName* := *Expression*

Example 1

This code creates an object of the class *Lamp*.

```
OBJECT l1 := @CreateObject("Lamp"; "")
```

Example 2

The keyword *OBJECT* can also be used in combination with the keyword *GLOBAL*. So you access a global object which is as well accessible via the LotusScript function *XFLGetGlobalObject()*.

```
Call XFLExecute({GLOBAL OBJECT ses := @CreateObject("NotesSession")}, Nothing)
Dim o As Variant
Set o = XFLGetGlobalObject("ses")
```

For more details see chapter *Object oriented programming with XFL*.

CALL

Indicates that the following method has no return value.

Syntax

CALL *oName.Method*

Example 1

Defining an @Formula to work with Notes classes.

```
DEFINE @Refresh := (OBJECT uidoc := @CreateObject("NotesUIWorkspace").CurrentDocument;
                                CALL uidoc.Refresh);
@Refresh;
```

For more details see chapter *Object oriented programming with XFL*.

ALIAS

Defines a synonym for a function or an identifier. Mostly it is useful to define aliases at initialization time using the global LotusScript variable *XFLInit* (see also chapter *Global declared LotusScript variables*).

Syntax

ALIAS *newName* := *oldName*

Example 1

```
ALIAS MyFunction := @Left;
MyFunction(Name ; " ")
```

Means the same as:

```
@Left(Name ; " ")
```

This way e.g. the English vocabulary of XFL can be translated into an other language.

Example 2

ALIAS can be applied to names of fields or variables. The following code changes the field *Name* which is accessed by its alias *FullName*. Using this principle you can store fields under different names than they are named in formulas.

```
ALIAS FullName := Name;
FIELD FullName := @UpperCase(FullName)
```

Functions

FOR

Executes one or more instructions as long as a condition stays true. The condition is tested before the execution of the instructions. Furthermore an initialization and incremental code is executed.

Syntax

@For(*Init* ; *Condition* ; *Increment* ; *Instruction* [; ...])

Parameters

Init: Instruction, usually sets an initial value to a loop variable.

Condition: Expression returning value *TRUE* or *FALSE*.

Increment: Instruction, usually incrementing the value of the loop variable.

Instruction: You can write as many formulas as you want.

Example

Bubblesort algorithm

```
list := 3:2:4:8:1;
max := Elements(list);
For(i := 1 ; i < max ; i := i + 1;
    For(k := max ; k>i ; k := k - 1;
        If(list[k] < list[k-1] ; (t:=list[k];list[k]:=list[k-1];list[k-1] := t) ; "")
     )
);
print(list);
```

WHILE

Executes one or more instructions as long as a condition stays true. The condition is tested before the execution of the instructions.

Syntax

@While(*Condition* ; *Instruction* [; ...])

Parameters

Condition: Expression returning value *TRUE* or *FALSE*.

Instruction: One or more formulas.

Example

This code prints all elements of the field *Names* to the status bar.

```
n := 1;
While(n <= Elements(Names);
          Print("Name " + Text(n) + ": " + Names[n]);
          n := n + 1)
```

DOWHILE

Executes one or more instructions as long as a condition stays true. The condition is tested after the execution of the instructions.

Syntax

@DoWhile(*Instruction* [; ...] ; *Condition*)

Parameters

Condition: Expression returning value *TRUE* or *FALSE*.

Instruction: One or more formulas.

Example

This code prints all elements of the field *Names* to the status bar.

```
If(Elements(Names) = 0; Return(0); "");
n := 1;
DoWhile(
        Print("Name " + Text(n) + ": " + Names[n]);
        n := n + 1;
        n <= Elements(Names))
```

GOTO

Continues running a routine at a label. Labels are defined using the keyword *LABEL*.

Syntax

@Goto(*Label*)

Parameters

Label: Name of a label.

The label must be defined inside the same routine as the function *@Goto()* or inside a outer routine.

Example

A loop using *@Goto()*.

```
LABEL Start;
@Prompt([OK] ; "Start" ; "This is the start");
@If(@Prompt([yesno] ; "endless"; "Again?") ; @Goto(Start) ; "");
```

GOSUB

Interrupts running a routine to continue at a label. After *@Return()* the interrupted routine will be continued at the instruction after *@Gosub()*.

Syntax

@Gosub(*Label*)

Parameters

Label: Name of a label.

Return value

Value returned by the *@Return()* statement.

Example

```
a := 0;
Print (GoSub(NextA) + " One");
Print (GoSub(NextA) + " Two");
Print (GoSub(NextA) + " Three");
Return("");

LABEL NextA;
Return(@Text(a := a + 1))
```

The label must be defined inside the same routine as the function *@Goto()* or inside a outer routine. A following construction is possible:

```
@For(i:=1; i<10 ; i:=i+1;
        @Gosub(Time); REM {jump out of @For};
        @Gosub(Inner); REM {jump not so far};
```

```
        @Print(y);
        @Goto(next);
        LABEL inner; REM{label in inner procedure};
        @Return(y := i*i);
Label next
);
@Return("");
LABEL Time; REM {label in outer procedure};
@Return(@Print(@now));
```

The following construction however is not allowed:

```
@For(i:=1; i<10 ; i:=i+1;
        @Gosub(Inner); REM {jump not so far};
        @Print(y); @Goto(next);
        LABEL inner; REM{label in inner procedure};
        @Return(y := i*i);
Label next
);
@Gosub(inner); REM {<-- not allowed}
```

RETURN

This function terminates a procedure. Called from a subfunction that very subfunction is terminated and the execution of the outer function continues. Called after *@Gosub()*, it causes a return to the instruction after *@Gosub()*.

Syntax

@Return(*Value*)

Parameters

Value: value which is returned to the caller.

EVAL

Executes XFL code. This function makes it possible to link XFL code dynamically.

Syntax

@Eval(*Code*)

Parameters

Code: Text, XFL code to be executed.

Return value

Value of the last expression of the executed code.

Example

Execution of a formula stored in a field of a profile document

```
@Eval(@GetProfileField("Config" ; "Formula"))
```

EVALONSERVER

Executes XFL code on the server. Can be used for executing several actions with a higher access level than the user possesses. This @Formula uses the agent *(XFLOnServer)*. All global variables are passed between client and server code. The user at least needs the access to write public documents in the database.

Syntax

@EvalOnServer(*Code*)

Parameters

Code: Text, XFL code to be executed.

Return value
Value of the last expression of the executed code.
Example
Writing an access log to a document. The user does not need author access to the document. Put this to the PostOpen event of the form:

```
Call XFLExecuteOnUIDoc(|
GLOBAL User := @Username;
@EvalOnServer({FIELD Log := @Trim(Log : @Text(@Now) + " " + GLOBAL User) ; @SaveDocument})
|)
```

The user triggers writing the log by the server without having author access to the document.

ISDEFINED

Tests if a given function name is a function formerly defined by *DEFINE*. It is not tested if the function is a valid Notes @Function.
Syntax
@IsDefined(*Fname*)
Parameters
Fname: Text, name of the function
Return value
TRUE if a function named *Fname* is defined by *DEFINE* else *FALSE*
Example

```
@If(@IsDefined("MyFunction"); ""; DEFINE MyFunction(x) := x*x);
a := MyFunction(2);
```

DEBUG

Switches debug mode on or off.
Syntax
@Debug(*Mode*)
Parameters
Mode: *1* to activate the debugger, *0* to deactivate it.

The debugger appears as a dialog box. If necessary you can implement your own debugger changing the code of the function *XFLDebug()* in script library *XFLExtension*.

PRINT

Prints values to the status bar.
Syntax
@Print(*Value1*[; ... [; *ValueN*]])
Parameters
Value1..N: Values of any data type.

Every value is printed to a separate line. If a value itself contains an array then every element of this array is printed separately. Alternatively you can use instead of *@Print()* the @Function ***@StatusBar()*** as it is standard since Notes R6.

EXECUTE

Executes LotusScript code. All global variables declared in the LotusScript library *XFLExtension* can be used within the code even the ones you added there. To access the current document e.g. use the variable *XFLRefDoc*.

Syntax

@Execute(*Code*)

Parameters

Code: Text, LotusScript.

Return value

Return code of the *End* statement if the code contains one, otherwise *0*.

Example

Setting the *IsReaders* flag of a field.

```
FIELD Readers := @UserName : "[Admin]";
@Execute({Dim it as NotesItem
Set it = XFLRefDoc.GetFirstItem("Readers")
it.IsReaders = True});
```

GETFIELD

Returns the value of a field of the current document.

Syntax

@GetField(*Fieldname*)

Parameters

Fieldname: Name of the field which is to be read.

Example

```
Print(GetField(Prompt([OKCancelEdit] ; "Accessing a field" ; "Which field do you want to read?" ; "")))
```

SETFIELD

Sets a field in the current document. This function is similar to @SetField() in SFL. A difference to SFL (up to Notes R5) is that you do not need to declare the field before.

Syntax

@SetField(*Fieldname* ; *Value*)

Parameters

Fieldname: Name of the field to be set.

Value: Value which is to be assigned to the field.

GETDOCFIELD

Returns the value of a certain field of a certain document. The document has to be stored in the same database as the current document. Differently form SFL the current document can be accessed too.

Syntax

@GetDocField(*UNID* ; *Fieldname*)

Parameter

UNID: UniversalID of the target document.

Fieldname: Name of the field whose value is to be read.

Example

This formula works in XFL but it is not allowed under Notes R4/R5:

```
@GetDocField(@DocumentUniqueID ; "Subject")
```

SETDOCFIELD

Sets the value of a certain field of a certain document. The document has to be stored in the same database as the current document. Differently form SFL the current document can be accessed too.

Syntax

@SetDocField(*UNID* ; *Fieldname* ; *Value*)

Parameters

UNID: UniversalID of the target document.

Fieldname: Name of the field whose value is to be set.

Value: Value which is to be assigned to the field.

GET

Returns the value of a local variable.

Syntax

@Get(*Varname*)

Parameters

Varname: Name of the variable.

Example

```
_a := "TEST";
Print(Get("_a"));
```

SET

Assigns a value to a local variable. It is not necessary to initialize the variable before as in SFL under Notes R4/5.

Syntax

@Set(*Varname* ; *Value*)

Parameters

Varname: Name of a local variable.

Value: Value which is to be assigned to the variable.

Example

```
@Set("Fullname" ; Firstname + " " + Lastname)
```

GETGLOBAL

Returns the value of a global variable.

Syntax

@GetGlobal(*Varname*)

Parameters

Varname: Name of a global variable.

Example

```
Global _a := "TEST";
Print(GetGlobal("_a"));
```

SETGLOBAL

Assigns a value to a global variable. Same meaning as *@Set()* for local variables.

Syntax

@SetGlobal(*Varname* ; *Value*)

Parameters

Varname: Name of a global Valiable.

Value: Value which is to be assigned to the global variable.

Example

```
@SetGlobal("Fullname" ; Firstname + " " + Lastname)
```

XFLVARIABLES

Returns an array of the names of all set local variables.

Syntax

@XFLVariables

Example

```
_a := "TEST"; _b := 123;
_vars := @XFLVariables;
For(i := 1 ; i <= @Elements(_vars) ; i := i + 1;
        Print(_Vars[i] + ": " + Text(Get(_Vars[i]))
);
```

XFLGLOBALVARIABLES

Returns an array of the names of all set global variables.

Syntax

@XFLGlobalVariables

Example

```
GLOBAL _a := "TEST"; GLOBAL _b := 123;
_vars := @XFLGlobalVariables;
For(i := 1 ; i <= @Elements(_vars) ; i := i + 1;
        Print(_Vars[i] + ": " + Text(Get(_Vars[i]))
);
```

IF

Similar to *@If()* in SFL. In XFL there is not a limit of the number of conditions/actions.

XFLVERSION

Returns the version number of the XFL interpreter.

Syntax

@XFLVersion

PROMPT

Opens a dialog box. Same syntax like in SFL. This function had to be reimplemented in LotusScript because it does not work in *Evaluate()*. Modified behaviour if *XFLDoNotQuitOnCancel* is set to *1* (see chapter *Global declared LotusScript variables*).

PICKLIST

Opens a modal window. Same syntax like in SFL. This function had to be reimplemented in LotusScript because it does not work in *Evaluate()*. Modified behaviour if *XFLDoNotQuitOnCancel* is set to *1* (see chapter *Global declared LotusScript variables*).

DIALOGBOX

Opens a modal window. Same syntax like in SFL. This function had to be reimplemented in LotusScript because it does not work in *Evaluate()*.

SAVEDOCUMENT

Saves the current document physically. If a formula is executed via *Evaluate()* or *XFLExecute()* it is unfortunately not possible to make out if the execution caused a change of the document. In SFL an

according check and the saving has to be performed in LotusScript after *Evaluate()*. XFL provides *@SaveDocument* as an other way.

Syntax

@SaveDocument

Example

```
_NewName := @Prompt([OKCancelEdit] ; "Change name" ; "Enter a new name" ; Name);
@If(Name = (FIELD Name := _NewName) ; "" ; @SaveDocument)
```

If the field Name is changed the document will be saved.

CREATEOBJECT

Creates an object. For more details see chapter *Object oriented programming with XFL*.

Syntax

@CreateObject(*ClassName* ; *args*)

Parameters

ClassName: Name of the class.

The class name can be the name of a native LotusScript class e.g. “NotesSession” or an user defined class. User defined classes have to be defined in the LotusScript library *XFLExtension*. Classes defined in other libraries can be embedded via *"USE ..."* in *XFLExtension*.

args: Initial parameters.

These parameters are passed to the method *NEW*.

Example

```
OBJECT d := @CreateObject("NotesDocument"; OBJECT DB)
```

SETOBJECT

Sets an object variable. For more details see chapter *Object oriented programming with XFL*.

Syntax

@SetObject(*Oname* ; *Object*)

Parameters

Oname: Text. Identifier of an object.

Object: Object which is assigned.

Example

```
@SetObject("Item" ; doc.GetFirstItem("Readers"));
```

This code does exactly the same:

```
OBJECT Item := doc.GetFirstItem("Readers");
```

SETGLOBALOBJECT

Sets a global object variable. For more details see chapter *Object oriented programming with XFL*.

Syntax

@SetGlobalObject(*Oname* ; *Object*)

Parameters

Oname: Text. Identifier of an object.

Object: Object which is assigned.

GETGLOBALOBJECT

Returns the value of a global object variable. If there is no object variable of this name *@Nothing* is returned. For more details see chapter *Object oriented programming with XFL*.

Syntax

@GetGlobalObject(*Oname*)

Parameters

Oname: Text. Identifier of an object.

Example

```
GLOBAL OBJECT db := @CreateObject("NotesSession").CurrentDatabase;
GLOBAL OBJECT doc1 := db.CreateDocument;
GLOBAL OBJECT doc2 := db.CreateDocument;
...
For(i := 1 ; i <= 2 ; i := i + 1;
        @GetGlobalObject("doc" + @Text(i)).Save(@True; @True)
);
```

XFLGLOBALOBJECTS

Returns an array of the names of all set global object variables. For more details see chapter *Object oriented programming with XFL*.

Syntax

@XFLGlobalObjects

Example

```
GLOBAL OBJECT ses := @CreateObject("NotesSession");
GLOBAL OBJECT db := ses.CurrentDatabase;
_olist := @XFLGlobalObjects;
Print(_olist);
```

NOTHING

Initial value of an object variable (as *Nothing* in LotusScript. For more details see chapter *Object oriented programming with XFL*.

Syntax

@Nothing

Example

```
OBJECT Doc := @Nothing;
```

ISNOTHING

Checks if an object has the initial value *Nothing*. For more details see chapter *Object oriented programming with XFL*.

Syntax

@IsNothing(*Object*)

Example

```
@If(@IsNothing(Doc) ; @Return("") ; "");
Doc.From := @UserName;
```

ERROR

Extends the SFL @Function *@Error* by the possibility of raising LotusScript runtime errors. Works like the ERROR statement of LotusScript.

Syntax 1

@Error

SLF @Function.

Syntax 2

@Error(*errNumer* [; *errMessage*])

Parameters

errNumber: Error number.

errMessage: Optional. Error message.
XFL specific error messages are defined in function *XFLErrorText()* of the script library *XFLExtension*. In this function other user defined error messages can be added. If *@Error()* is called without the second parameter the interpreter tries to get the error message text by calling *XFLErrorText()*.

Example

```
        On Error Goto labError
        Call xflexecute({
@If(@Prompt([YesNo] ; "Abort?"; "Question") ; @Error(5500 ; "Function aborted"); "")
} , Nothing)

        ' ...

        Exit Sub
labError:
        Messagebox "LotusScript error handling (" + Cstr(Err) + ", " + Error$ + ")"
        '...
```

Operators

Operators "&" and "|"

If two expressions as logically ANDed the native formula processor of Notes at first solves both expressions and after that ANDs the both results. In many cases this may be inefficient because if the first expression is *FALSE* the total result is *FALSE* and solving the second expression would not be necessary. Analogous two logically ORed expressions return *TRUE* even after solving the first to *TRUE*. In these cases solving the second expression is also not necessary. XFL uses this optimized algorithm to handle *AND* and *OR*.

Example

This formula combines two security questions in a simple way.

```
@If(@Prompt([yesno] ; "Question" ; "Do you want to do this?") & @Prompt([yesno] ; "Question" ; "Do you REALLY want to do this?") ; "" ; @Return(""));
@Prompt([ok] ; "Info" ; "Done");
```

In SFL the second question is asked uselessly even if the first was answered in negative. Not in XFL.

List subscript operator "[ ]"

The elements of an array can be accessed by its index. This index number is written behind the variable in square brackets.

Example

```
_list := "A" : "B" : "C";
Print(_list[2]);
```

Since Notes R6 this operator is available in SFL too, but only to **read** the elements not to change them. This code will not run in SFL under Notes R6:

```
_list := "A" : "B" : "C";
_list[2] := "Z"; REM {problem for SFL, no problem for XFL};
@StatusBar(_list);
```

Extended field search by operator "?"

The search operator *?* provides an easy way to access fields of other documents. This operator is written behind a field name. The operator causes in case of the current document does not contain the field a search for this field in alternative documents. The documents to be searched are determined by the function *XFLGetAltRefDoc()* in the LotusScript library *XFLExtension*. The original version of this function implements the hierarchy of response documents linked to each other by the *$Ref* field. Thus the next document to search is the parent document. It is possible to implement other algorithms according to the conditions and structures of your application e.g. accessing separate configuration databases.

Example

```
FIELD Categories := Categories?
```

If the current document does not contain the field *Categories* then the parent documents of the document are searched. After that the value is written to the field *Categories* in the current document.

Additional features

String delimiter { } and ' '

Strings constants can be enclosed not only in quotation marks " " but also in braces { } and single quotation marks. Functional there is no difference. It helps you to avoid unattractive masking of quotation marks inside of string constants.

Use of keywords

Opposed to SFL, keywords need not stand before main expressions, but can be used inside of expressions.

Example

```
@If(Subject = "" ; FIELD Subject := "<no Subject>" ; "")
```

Notes R6 tries to remove this restriction, which did not succeed completely. Code like the following works fine in XFL but it would not run even in Notes R6:

```
@If(_cond = @True ; DEFAULT Subject := "True" ; "")
```

Prefix @ of Notes @Functions is optional

All @Functions of Notes can be used in XFL. The starting character @ can be omitted.

Example

```
@Explode(@Left(Subject ; ".") ; " ")
```

is identical to

```
Explode(Left(Subject ; ".") ; " ")
```

Exceptions are Notes functions without parameters because in these cases the @ is necessary to distinguish function names from identifiers of fields or variables. So *@Today* means today's date but *Today* means a variable or field named *"Today"*.

Parentheses can replace @Do

In SFL grouped instructions have to be enclosed in *@Do()*. This is not necessary in XFL. Simple parentheses are sufficient.

Example

```
@If(_var1 = "1" ; @Do(FIELD abc := "Test"; FIELD xyz := 1) ; "")
```

is identical to

```
If(_var1 = "1" ; (FIELD abc := "Test"; FIELD xyz := 1) ; "")
```

Limited nesting levels

The XFL interpreter processes nested expressions in a recursive way. Recursion in LotusScript however is allowed only up to a certain level. Notes generates an "Out of stack space" error in case of overrunning this limit. In SFL the limit is higher so that extremely nested terms running well in SFL may cause an error in XFL runtime environment. To avoid this error you can divide the large term into smaller ones or wait for better Notes versions which support more nesting.

Example

The summation of 50 ones works in SFL, not in XFL. Deleting the brackets removes the problem.

```
formula =
"1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+
(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+1)))))))))))))))))))))))))))))))))))))))))))))))))"
v = Evaluate(formula)                   'works fine
v = XFLExecute(formula, Nothing)        'Out of stack space
```

Handling field and variable names containing dots "."

The dot "." is used in XFL for object oriented programming (see chapter *Object oriented programming with XFL*). Therefore it is not possible to use instructions like this:

```
FIELD DB.Title := @DbTitle;
```

You have to use the @Functions *@Set()*, *@Get()*, *@SetField()* and *@GetField()* instead:

```
@SetField("DB.Title"; @DbTitle);
```

XFL runtime environment

The way of processing XFL code

XFL code is executed by an interpreter. This interpreter is implemented in LotusScript and consists of the two LotusScript libraries *XFLEngine* and *XFLExtension*. To run XFL code just call a script function e.g. *XFLExecute()*. The interpreter can be used in Notes R4.6x or higher in every place where LotusScript is available in Notes clients or background agents. It is not possible to use XFL in view selection formulas. To implement field formulas use events that support LotusScript e.g. *PostOpen* or *QuerySave*.

Execution of XFL code takes place in two steps:

1. Parsing the code and transforming it into an internal function tree structure.
   Syntax errors are detected at this point and are reported by generating a LotusScript error with detailed error description and position.
2. Stepping through the function tree.
   Three types of functions and operations are differentiated:
   1. Native Notes operators and @Functions (SFL)
      These are executed via *Evaluate()*.
   2. Native XFL operators and functions
      These are implemented in LotusScript in library *XFLEngine*.
   3. User defined functions
      If the function is hard coded in library *XFLExtension* it is executed calling *XFLExecuteFunction()*.
      If the function is defined by a *DEFINE* expression it is executed interpreting this statement.

LotusScript library XFLEngine

Kernel of the interpreter. It provides several functionalities:

- Routines to test and execute XFL code
- Routines to access global XFL variables

This library itself uses variables and routines of the LotusScript library *XFLExtension*.

Function XFLExecute

Executes XFL code.

Syntax

Function XFLExecute(*Formula* as String, *doc* As notesdocument) as Variant

Parameters

Formula: XFL code.

doc: reference document on which the formula is applied.

Return value

Result of the last expression of the code. Whereas the LotusScript function *Evaluate()* always returns an array *XFLExecute()* does so just in case of the result is really an array with more than one value.

Example

This is an example for a form button. It calculates for each of two dates the next anniversary. It shows how easily new functions can be defined in XFL.

```
Use "XFLEngine"
Dim wks As New NotesUIWorkspace, uidoc As NotesUIDocument, v As Variant

Const formula = |
REM "define an algorithm as an @Function";
DEFINE @NextAnniversary(date) := @Do(
        _thisAnniversary := @Adjust(date ; @Year(@Today) - @Year(date);0;0;0;0;0);
        @Adjust(_thisAnniversary ; @If(_thisAnniversary > @Today ; 0 ; 1);0;0;0;0;0));

FIELD NextDate1 := @NextAnniversary(Date1);
FIELD NextDate2 := @NextAnniversary(Date2);
|

Set uidoc = wks.currentdocument
v = XFLExecute(formula , uidoc.document)
```

Function XFLExecuteOnUIDoc

Executes XFL code on the current UI document. Use this function in form buttons or agents. Using *XFLExecuteOnUIDoc()* instead of *XFLExecute()* can save some lines of code. Internally it is implemented this way:

```
Public Function XFLExecuteOnUIDoc(code As String) As Variant
        Dim wks As New NotesUIWorkspace
        XFLExecuteOnUIDoc = XFLExecute(code, wks.CurrentDocument.Document)
End Function
```

Function XFLExecuteOnUIView

Executes XFL code on selected documents of a view. Use this function in view buttons or agents. Using *XFLExecuteOnUIView()* instead of *XFLExecute()* can save some lines of code. Internally it is implemented this way:

```
Public Function XFLExecuteOnUIView(code As String) As Variant
	Dim ses As New NotesSession, db As NotesDatabase, col As NotesDocumentCollection, doc As NotesDocument
	Set db = ses.CurrentDatabase
	Set col = db.UnprocessedDocuments
	Set doc = col.GetFirstDocument
	While Not doc Is Nothing
		XFLExecuteOnUIView = XFLExecute(code , doc)
		Set doc = col.GetNextDocument(doc)
	Wend
End Function
```

Function XFLExecuteOnServer

Executes XFL code on server. Can be used for executing several actions with a higher access level than the user possesses. This function uses the agent *(XFLOnServer)*. All global variables are passed between client and server code. The user at least needs the access to write public documents in the database.

Syntax

Function XFLExecuteOnServer(*Formula* as String, *doc* As notesdocument) as Variant

Parameters

Formula: Text, XFL code to be executed.

doc: reference document on which the formula is applied.

Return value

Value of the last expression of the executed code.

Example

Writing an access log to a document. The user does not need author access to the document. Put this to the PostOpen event of the form:

```
Call XFLSetGlobalVar("User", ses.Username)
Call XFLExecuteOnServer({FIELD Log := @Trim(Log : @Text(@Now) + " " + GLOBAL User);
@SaveDocument}, source.document)
```

The user triggers writing the log by the server without having author access to the document.

Sub XFLCheckSyntax

Checks formula code for errors in its structure. The code is e.g. searched for wrong parentheses. When an error is found this function generates a error containing detailed information about it. Only the structure of the code is analysed not the validity of function names. Only some functions e.g. *@If()* are checked for the correct number of parameters.

Syntax

Sub XFLExecute(*Formula* as String)

Parameters

Formula: XFL code.

Example

This raises an error because of wrong parentheses.

```
Use "XFLEngine"
Const Formula = |@If(a=0; "0"; a>0 ; "positive" ; "negative"))|
Call XFLCheckSyntax(Formula)
```

Sub XFLSetGlobalVar

Sets a global variable. It can be reused in later XFL code.

Syntax

Sub XFLSetGlobalVar(*VarName* As String, *value* As Variant)

Parameters

VarName: Name of an XFL variable.

Value: Value to be assigned to the variable.

Example

The debugger shows well how the function works.

```
Use "XFLEngine"
Call XFLSetGlobalVar("Text" , "This is an example")
Print XFLExecute(|@Debug(1);@ReplaceSubstring(GLOBAL Text ; " " ; "")| , Nothing)
```

Function XFLGetGlobalVar

Reads a global variable. Global variables can be set by XFL code or by the function *XFLSetGlobalVar()*.

Syntax

Function XFLGetGlobalVar(*VarName* As String) As Variant

Parameters

VarName: Name of an XFL variable.

Return value

Value of the variable.

If the variable does not exist *EMPTY* is returned.

Example

Data exchange between XFL and LotusScript.

```
Use "XFLEngine"
Const Formula = |GLOBAL NewText := @ReplaceSubstring(GLOBAL Text ; " " ; "")|
Call XFLSetGlobalVar("Text" , "This is an example")
Call XFLExecute(Formula, Nothing)
Print XFLGetGlobalVar("Text") & " -> " & XFLGetGlobalVar("NewText")
```

Sub XFLDeleteGlobalVar

Deletes one or more global variables.

Syntax

Sub XFLDeleteGlobalVar(*VarNames* As Variant)

Parameters

VarNames: Name or array of names of global variables.

Example

Data exchange between XFL and LotusScript.

```
Const Formula = |GLOBAL NewText := @ReplaceSubstring(GLOBAL Text ; " " ; "")|
Call XFLSetGlobalVar("Text" , "This is an example")
Call XFLExecute(Formula, Nothing)
Print XFLGetGlobalVar("Text") & " -> " & XFLGetGlobalVar("NewText")
Call XFLDeleteGlobalVar("Text")
Print XFLGetGlobalVar("Text") & " -> " & XFLGetGlobalVar("NewText") ' now the first string is empty
```

Function XFLGlobalVariables

Lists the names of all global variables.

Syntax

Function XFLGlobalVariables As Variant

Return value

Array of the variable names.

Example

Data exchange between XFL and LotusScript.

```
Dim vnames As Variant
Const Formula = |GLOBAL NewText := @ReplaceSubstring(GLOBAL Text ; " " ; "")|
Call XFLSetGlobalVar("Text" , "This is an example")
Call XFLExecute(Formula, Nothing)
vnames = XFLGlobalVariables
Forall v In vnames
        Print v, XFLGetGlobalVar(Cstr(v))
End Forall
```

Function XFLCreateGlobalObject

Creates a global object variable. It can be reused in later XFL code.

Syntax

Function XFLCreateGlobalObject(*oname* As String, *classname* As String, *args* As Variant) as Variant

Parameters

oname: Name of the new object variable.

ClassName: Name of the class.

The class name can be the name of a native LotusScript class e.g. “NotesSession” or an user defined class. User defined classes have to be defined in the LotusScript library *XFLExtension*. Classes defined in other libraries can be embedded via *"USE ..."* in *XFLExtension*.

args: Initial parameters.

These parameters are passed to method *NEW*.

Return value

Object created.

Example

Exchanging objects between XFL and LotusScript.

```
Use "XFLEngine"
Call XFLCreateGlobalObject("session", "NotesSession", Null)
Const Formula = {OBJECT _agent := session.CurrentAgent ; @Print(_agent.Name)}
Call XFLExecute(Formula, Nothing)
```

For more information see chapter *Object oriented programming with XFL*.

Function XFLSetGlobalObject

Sets a global object variable.

Syntax

Sub XFLSetGlobalObject(*oname* As String, *obj* as Variant)

Parameters

oname: Name of the object variable.

obj: Object to be assigned to the variable.

Example
Exchanging objects between XFL and LotusScript.

```
Dim ses As New NotesSession
Call XFLSetGlobalObject("session", ses)
Call XFLExecute({
        OBJECT agent := session.CurrentAgent;
        @Print(agent.Name)
}, Nothing)
```

For more information see chapter *Object oriented programming with XFL*.

Function XFLGetGlobalObject

Returns a global object variable.
Syntax
Function XFLGetGlobalObject(*oname* As String) as Variant
Parameters
oname: Name of the object variable.
Return value
If an object of this name exists it is returned else *Nothing* is returned.
Example
Exchanging objects between XFL and LotusScript.

```
Dim ag As Variant
Const Formula = {GLOBAL OBJECT agent := @CreateObject("NotesSession").CurrentAgent ; @Print(agent.Name)}
Call XFLExecute(Formula, Nothing)
Set ag = XFLGetGlobalObject("agent")
```

For more information see chapter *Object oriented programming with XFL*.

Sub XFLDeleteGlobalObject

Deletes one or more global object variables.
Syntax
Sub XFLDeleteGlobalObjects(Object*Names* As Variant)
Parameters
ObjectNames: Name or array of names of global object variables.
Example
Exchanging objects between XFL and LotusScript.

```
Dim ag As Variant
Const Formula = {GLOBAL OBJECT agent := @CreateObject("NotesSession").CurrentAgent ; @Print(agent.Name)}
Call XFLExecute(Formula, Nothing)
Set ag = XFLGetGlobalObject("agent")
Call XFLDeleteGlobalObject("agent")
```

For more information see chapter *Object oriented programming with XFL*.

Function XFLFormatCode

Formats a XFL expression. The structure of the formula is solved and presented inserted. Unnecessary spaces are removed.

Syntax

Function XFLFormatCode(*Code* As String) as String

Parameters

Code: Formula to be formatted.

Return value

Formula in new format.

Example

```
Use "XFLEngine"
Const f = |@If(b<100; 100;b)|
Messagebox XFLFormatCode(f)
```

Output:

```
@IF
   (
         B
      <
         100
   ;
      100
   ;
      B
   )
```

LotusScript library XFLExtension

The library *XFLExtension* contains functions which are called by functions of the library *XFLEngine*. Here the programmer gets the possibility to implement own @Functions and classes or adapt the language otherwise.

Global declared LotusScript variables

The following variables are important:

XFLRefDoc as NotesDocument

Is set by the function *XFLExecute()* to the passed NotesDocument object. Can be used in subs or functions of this library.

XFLDebugMode as Integer

Represents the current debugging mode. This variable changes executing the @Function *@Debug()*. By setting it to *1* you can enable debugging or to *0* to disable debugging.

XFLExtendedFunctions as String

Here the names of user defined functions are declared. Separate the names by comma. This variable has to be set in *Initialize*. It is used by the engine to decide if a function is user defined and has to be executed by calling *XFLExecuteFunction()*.

XFLInit as String

Here you can declare XFL code which is to be executed at the time of loading the library *XFLEngine*. This variable has to be set in *Initialize*. It is suitable for *ALIAS* or *DEFINE* statements or for setting global variables.

XFLDoNotQuitOnCancel As Integer

If a *@Prompt()* or *@Picklist()* dialog is cancelled in SFL the whole formula code is aborted. So it is not possible to react to this event by formulas. By setting the global variable *XFLDoNotQuitOnCancel* to *1* you can cause continuing the formula after canceling *@Prompt()*. In this case *@Prompt()* returns *-1*.

Function XFLExecuteExtendedFunction

This routine is called every time a user defined function occurs in XFL code whose name is declared in *XFLExecuteFunctions*.

Syntax

Function XFLExecuteFunction(*fname* As String, *Params* As Variant, *IsXFL* As Integer) As Variant

Parameters

Fname: Name of the called function.

Params: passed parameters for the function.

IsXFL: Is a return value. If you set it within this routine to *TRUE* the result is interpreted as XFL code as well and will be executed afterwards.

Return value

Result of the user defined function.

An own XFL function is programmed in two steps:

1. Declare the nameof the function in variable *XFLExtendedFunctions*.
2. Write the function code into Sub *XFLExecuteFunctions()*.

Example 1

```
Sub Initialize
```

```
        XFLExtendedFunctions = "... , @MyFunction"
End Sub
Function XFLExecuteExtendedFunction(fname As String, Params As Variant, IsXFL As Integer) As Variant
        Select Case fname
        ...
        Case "@MYFUNCTION": ' <- allways capital letters!
                Messagebox "Hello World! " + Params(0) ' This @Function shall have at least one parameter
                XFLExecuteExtendedFunction = True
        End Select
End Function
```

That is all. Test it by a little agent:

```
Use "XFLEngine"
Call XFLExecute(|@For(i := 1 ; i<4 ; i := i + 1 ; @MyFunction("Little test " + @Text(i)))|, Nothing)
```

Please note that XFL function or variable names are case insensitive. Internally they are always handled in capital letters. So in the example above you also could write

```
...@For(i := 1 ; i<4 ; i := i + 1 ; @mYfUnCtIoN("Little test " +...
```

However, the name of the XFL Function is passed to sub ***XFLExecuteFunction()*** always in **capital letters** (see example in LotusScript libary *XFLExtension*)!

Example 2

The result of an user defined function can be XFL code again being executed afterwards. In this case the parameter IsXFL has to be set to *TRUE*. This example defines the function *@2ndRight()* which searches for the second occurrence of a given string.

```
Sub Initialize
        XFLExtendedFunctions = "... , @2ndRight"
End Sub
Function XFLExecuteExtendedFunction(fname As String, Params As Variant, IsXFL As Integer) As Variant
        Select Case fname
        ...
        Case "@2NDRIGHT": ' <- always capital letters!
        XFLExecuteExtendedFunction = |@Right(@Right("| + params(0) + |" ; "| + params(1) + |") ; "| + params(1) + |")|
        IsXFL = True
        End Select
```

Test it:

```
Use "XFLEngine"
Print XFLExecute(|_text := "these are some words"; @2ndRight(_text ; " ")|, Nothing)      ' prints "some words"
```

Sub XFLPrepareCode

This sub is called internally before execution of XFL code. Here you can make changes on the code or add actions e.g. for logging. You also can use this sub in conjunction with a SmartIcon to toggle the debug mode.

Syntax

Sub XFLPrepareCode(Code as String)

Parameter

Code: Code to be executed.

Sub XFLDebug

This routine is called by the interpreter during execution of XFL code after every function or operation, if the XFL debugger is activated. The intermediate result and set variables are displayed. The debugger can be activated by the XFL function *@Debug(1)* or by setting the global LotusScript variable *XFLDebugMode = True*.

Syntax

Function XFLDebug(*code* As String, *StartPos* As Integer, *EndPos* As Integer, *Value* As NotesDocument, *locals* As NotesDocument, *globals* As NotesDocument, *localObjects* As Variant, *globalObjects* As Variant)

Parameters

Code: Code being executed.
StartPos: Position of the first character of the current expression in code string.
EndPos: Position of the last character of the current expression in code string.
Value: Intermediate result.
Locals: Local variables.
Globals: Global variables.
LocalObjects: Local objects.
GlobalObjects: Global objects.

The variables are delivered as fields of temporary documents because LotusScript does not support all formula data types, e.g. @Error or links. The standard version of this sub only displays the values. If necessary you can change the code even to manipulate the values as it is usual in other debuggers. To do so just implement methods to change the fields of the temporary documents.

Sub XFLCommand

This sub is called by the interpreter during execution of XFL code every time an *@Command([])* occurs.

Syntax

Sub XFLCommand(*cname* As String, *params* As Variant)

Parameters

Cname: Name of the command.
Params: Parameters of the @Command.

Use this sub to implement @Commands you want to use in XFL code. This sub is necessary for achieving compatibility to SFL. For the moment only *@Command([FileSave])* and *@Command([ViewRefreshFields])* are implemented. This way SFL code with @Commands can be executed by the XFL interpreter. Unsupported @Commands will have no effect.

Function XFLGetAltRefDoc

XFL provides a special operator "?" to search fields in other documents. The operator is written behind a field name. If the current document would not contain the field other documents can be searched. The function *XFLGetAltRefDoc()* defines which alternative documents are to be searched. In the standard version of this sub the response hierarchy of documents linked by *$Ref* is implemented. This means the search in each parent document. Other algorithms can be programmed here.

Syntax

Function XFLGetAltRefDoc(*doc* As notesdocument, *fieldname* As String) As notesdocument

Parameters
doc: Document not containing the field.
fieldname: Name desired field.
Return value
Document to be searched next

This function is called as long as it returns *NOTHING* or the field is found.

Example

```
Call XFLExecute(|FIELD Categories := Categories?| , doc)
```

If *doc* does not contains the field *Categories* then at first the parent documents are searched and after that the fields value is written to the field *Categories* in *doc*.

Agent (XFLOnServer)

This agent is used by @formula *@EvalOnServer()* and function *XFLExecuteOnServer()*. The code passed to *@EvalOnServer()* and *XFLExecuteOnServer()* is executed in the name of the signer of this agent.

Subform XFLExtensionR4

This subform is necessary only is your application runs on Notes R4 clients. It is used by the functions *@Prompt()* and *@Picklist()* for LotusScript of Notes R4 does not provide an equivalent for these functions. If there are no R4 clients any longer this subform would be unnecessary.

Object oriented programming with XFL

SFL dos not support object oriented programming whereas XFL allows combining formula and object oriented programming.

All available LotusScript classes can be used. User defined classes has to be defined in the LotusScript library *XFLExtension* or has to be embedded via *"USE ..."* in *XFLExtension* if they are defined in other libraries.

An object is created in XFL by the function *@CreateObject()* or in LotusScript by the function *XFLCreateGlobalObject()*.

In XFL formulas an object variable has to be identified by the keyword *OBJECT*.

```
OBJECT session := @CreateObject("NotesSession");
```

An object is accessed by the methods of its class. Use the common dot notation. Chaining of methods is possible.

```
REM {Example to set the IsAuthor flag of a field. This is not possible using simple formulas};
OBJECT session := @CreateObject("NotesSession");
OBJECT mydoc := session.currentdatabase.AllDocuments.GetFirstDocument; REM {chaining};
OBJECT it := mydoc.GetFirstItem("Editors");
It.IsAuthors := @True; REM {changing properties of objects by assignment operator :=}
```

It is possible to distinguish between local and global objects as it is possible for common variables. To do so use the keywords *OBJECT* and *OBJECT GLOBAL* or the @functions *@SetObject()* and *@SetGlobalObject()*.

Global objects can also be accessed in LotusScript by the functions *XFLSetGlobalObject()* and *XFLGetGlobalObject()*.

The current document is stored in the global object variable *doc* by default. The following formula line would work without assignment of additional variables:

```
doc.GetFirstItem("Editors").IsAuthors := @True; REM {use of object variable DOC};
```