XFL

Extended Formula Language

Entwickelt 2005-2009
von Bert Häßler
www.nappz.de/xfl
mailto:xfl@nappz.de

Dokumentation XFL Version 2.85
15. Februar 2009

Vorbemerkung

Die Formelsprache von Notes ist eine sehr einfach gehaltene Programmiersprache, die dabei jedoch eine große Mächtigkeit besitzt. Darin birgt sie gegenüber LotusScript einige Vorteile, z.B. bei der Listenbehandlung. Formeln lassen sich dadurch nicht nur gut zur Implementierung von Agenten, Aktionen usw. einsetzen, sondern auch zur Konfiguration komplexerer Anwendungen. So nutzen z.B. notesbasierte Workflowsysteme die Formelsprache zur Definition von Prozessen oder Aktionen. Leider gibt es aber beim Einsatz der Formelsprache auch Grenzen. Einige Unzulänglichkeiten der Formelsprache bzw. des Zusammenspiels mit LotusScript sind:

- fehlende Möglichkeit, eigene Formeln zu definieren*
- keine echtes Debugging für Formeln
- keine Einbindung eigener Skriptbibliotheken möglich
- keine Nutzung von Unterfunktionen möglich
- kein direkter Datenaustausch zwischen Script- und Formelcode
- keine Sprünge/Schleifen programmierbar**
- Neuzuweisung von Variablen umständlich**
- *Evaluate()* mit Einschränkungen bei Nutzerinteraktion, z.B. kein *@Prompt* möglich

* zumindest jenseits der DLL-Programmierung
** teilweise ab Notes R6 möglich

So entstand die Idee, die Formelsprache an einigen Stellen zu erweitern. Ziel war es einerseits, die genannten Defizite zu beheben und andererseits dem Entwickler Einflussmöglichkeiten auf die Gestaltung der Sprache zu geben. Der Code sollte zudem auch unter Notes R4/R5 lauffähig sein. Im Ergebnis entstand der neue Sprachdialekt *Extended Formula Language* (XFL). XFL ist eine Sprache, die an die Notes Formelsprache angelehnt ist. Sie ist in ihrer Syntax abwärtskompatibel zur Standard-Notes-Formelsprache (SFL). Es sind in XFL jedoch einige neue und abweichende Sprachkonstrukte enthalten, die SFL nicht kennt.

XFL-Sprachkonstrukte

Alle SFL-Konstrukte können auch unter XFL verwendet werden. Die Möglichkeiten von SFL können Sie in der Hilfe des Domino Designers nachlesen. Im Folgenden sollen nur jene Sprachkonstrukte genannt werden, die XFL von SFL unterscheidet. Einige Features gehören ab Notes R6 bereits zur Standard. XFL stellt diese Funktionalitäten jedoch auch schon für Notes R4/R5 bereit! Teilweise ist die Bedeutung unter XFL auch eine etwas andere.

Schlüsselworte

DEFINE

Erlaubt es, während der Laufzeit, Funktionen zu definieren. In den meisten Fällen ist es sinnvoll, solche anwendungsspezifischen Funktionen bei der Initialisierung über die globale LotusScript-Variable *XFLInit* zu definieren (siehe Abschnitt *Global deklarierte LotusScript-Variablen*).

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

Es können Funktionen mit oder ohne Parametern definiert werden. *FunctionBody* ist ein XFL-Ausdruck, der bei Aufruf der Funktion *FunctionName* ausgeführt werden soll. Die optionalen Parameter *Param1-N* liegen dabei als lokale Variablen vor. Eine Funktionsdefinition wirkt global, d.h. sie bleibt erhalten, solange die Scriptbibliothek *XFLEngine* geladen ist.

Beispiel 1
Das ist die Implementierung eines Sortieralgorithmus. Da es ab Notes R6 die @Funktion *@Sort()* gibt, können wir diese bei R6-Clientes gleich verwenden. Für ältere Versionen wird hier *@Sort* als Bubblesort definiert.

```
@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 {testen wir es aus};
_l := 4:2:3:1;
@Print(@Sort(_l))
```

Beispiel 2
Es ist auch möglich, bestehende @Funktionen neu zu definieren. Mit folgender Zeile lässt sich beispielsweise ein datumsabhängiger Code auf Lauffähigkeit zu anderen Zeiten testen.

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

Beispiel 3
@Formeln können „überladen“, d.h. mit verschiedenen Parameteranzahlen definiert werden.

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

UNDEFINE

Löscht eine mit *DEFINE* definierte Funktion.

Syntax

UNDEFINE *FunctionName*

Beispiel

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

ORIGINAL

Wenn eine bestehende @Funktion durch *DEFINE* neu definiert wurde, so bewirkt *ORIGINAL* den Aufruf der ursprünglichen Funktion.

Syntax

ORIGINAL *FunctionName*

Beispiel

Mit *DEFINE* in Verbindung mit *ORIGINAL* kann die Syntax von @Funktionen erweitert werden. *@Prompt()* erfordert normalerweise mindestens drei Parameter. In diesem Beispiel werden *@Prompt()*-Varianten mit nur einem oder zwei Parametern definiert.

```
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

Kann vor einer Variablen stehen und zeigt an, dass diese zum globalen Variablenbereich gehört. Globale Variablen bleiben im Gegensatz zu lokalen Variablen auch nach Ausführung einer XFL-Formel erhalten und können damit auch von nachfolgenden *XFLExecute()*-Aufrufen weiterverwendet werden. Auf den globalen Variablenbereich kann zudem per LotusScript über *XFLGetGlobalVar()* und *XFLSetGlobalVar()* zugegriffen werden. Dadurch eignen sich globale Variablen gut zum Datensustausch zwischen XFL-Code und LotusScript.
Auch innerhalb einer XFL-Formel kann eine klare Separierung von lokalen und globalen Variablen sinnvoll sein, siehe Beispiel.

Beispiel

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

Während der Ausführung des Ausdrucks *Sqr(4)* wird eine lokale Variable *x* angelegt, die nur innerhalb der Funktionsdefinition von *Sqr(x)* existiert. Mit einer lokale Variablen *x* in der Hauptschleife würde es keinen Konflikt geben. Veranschaulichen kann man sich das mit dem Debugger:

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

Ein völlig anderes Ergebnis hingegen ergibt folgende Formel:

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

Hier greifen die Hauptfunktion und die Unterfunktion auf eine globale Variable zu.

Beim lesenden Zugriff auf eine globale Variable kann das Schlüsselwort *GLOBAL* entfallen, sofern es keine lokale Variable gleichen Namens gibt.

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

Der Interpreter führt bei Auftreten eines Bezeichners, z.B. "*X*" die Ermittlung, was damit bezeichnet wird, nach folgender Reihenfolge durch:

1. Gibt es eine lokale Variable *X*?
2. Gibt es ein lokales Objekt *X*?
3. Gibt es eine globale Variable *X*?
4. Gibt es ein globales Objekt *X*?
5. Gibt es ein Feld namens *X*?
6. Gibt es eine selbst definierte Funktion *X*?

Beim ersten positiven Ergebnis wird die Suche abgebrochen. Wenn auch die letzte Abfrage kein Ergebnis bringt, wird *UNAVAILABLE* zurückgegeben. Daher führt im obigen Beispiel das erste *@Print(x)* zur Ausgabe der globalen Variable, das zweite *@Print(x)* zur Ausgabe der lokalen Variablen *x*.

LABEL

Definiert eine Sprungmarke. Verwendung im Zusammenhang mit *@Goto()* oder *@Gosub()*.

Syntax

LABEL *LabelName*

Beispiel

```
LABEL Start;
@Prompt([OK] ; "Start" ; "Das ist der Anfang");
@If(@Prompt([yesno] ; "endlos"; "Noch einmal?") ; @Goto(Start) ; "");
```

DEFAULT

Syntax

DEFAULT *fieldName* := *value*

Wenn das aktuelle Dokument das Feld *fieldName* nicht besitzt, wird es mit dem Wert *value* gesetzt. Anders als bei SFL kann ein solcher Ausdruck auch verkettet werden.

Beispiel

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

Wenn das Feld *Qty* noch nicht existiert, wird es zunächst auf *1* gesetzt. Danach wird das Produkt berechnet. Wenn das Feld *Qty* schon existiert, wird mit dessen Feldwert multipliziert.

OBJECT

Zeigt an, dass der folgende Bezeichner eine Objektvariable darstellt.
Syntax
OBJECT *oName* := *Expression*
Beispiel 1
Dieser Code erstellt ein Objekt der Klasse *Lamp*.

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

Beispiel 2
Das Schlüsselwort *OBJECT* kann auch in Verbindung mit dem Schlüsselwort *GLOBAL* verwendet werden. Dies ermöglicht einen gemeinsamen Zugriff auf das Objekt über Formelcode und LotusScript (über die Funktion *XFLGetGlobalObject()*).

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

Weitere Informationen im Abschnitt *Objektorientierte Programmierung mit XFL*.

CALL

Zeigt an, dass der folgende Methodenaufruf keinen Rückgabewert liefert (wie *Call* in LotusScript).
Syntax
CALL *oName.Method*
Beispiel 1
Dieser Code definiert eine @Formel für eine Objektmethode.

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

Weitere Informationen im Abschnitt *Objektorientierte Programmierung mit XFL*.

ALIAS

Definiert ein Synonym für eine Funktion oder einen Bezeichner. In den meisten Fällen ist es sinnvoll, die Aliase bei der Initialisierung über die globale LotusScript-Variable *XFLInit* zu definieren (siehe Abschnitt *Global deklarierte LotusScript-Variablen*).
Syntax
ALIAS *newName* := *oldName*
Beispiel 1

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

Ist identisch mit:

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

Auf diese Weise kann z.B. der englische Wortschatz von XFL in eine andere Sprache übersetzt werden.

Beispiel 2

ALIAS lässt sich auch auf Feld- oder Variablennamen anwenden. Folgender Code verändert das Feld *Name*, welches über den Alias *FullName* angesprochen wird. Über dieses Prinzip können Felder physikalisch unter anderen Namen gespeichert werden als sie in Formeln bezeichnet werden.

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

Funktionen

FOR

Führt eine oder mehrere Anweisungen so lange aus, solange eine Bedingung wahr bleibt. Prüft diese Bedingung, vor Ausführung der Anweisungen. Führt zudem eine Initialisierung und eine Inkrementierung aus.

Syntax

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

Parameter

Init: Anweisung, die üblicherweise einer Laufvariablen einen Anfangswert zuweist.

Condition: Ein Ausdruck, der *TRUE* oder *FALSE* zurückgibt.

Increment: Eine Anweisung, die üblicherweise den Wert der Laufvariablen erhöht.

Instruction: Eine Anweisung in Formelsprache. Sie können beliebig viele Formeln angeben.

Beispiel

Bubblesort-Algorithmus

```
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

Führt eine oder mehrere Anweisungen so lange aus, solange eine Bedingung wahr bleibt. Prüft diese Bedingung, vor Ausführung der Anweisungen.

Syntax

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

Parameter

Condition: Ein Ausdruck, der *TRUE* oder *FALSE* zurückgibt.

Instruction: beliebig viele XFL-Formeln.

Beispiel

Dieser Code schreibt alle Elemente des Felds *Names* in die Statuszeile.

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

DOWHILE

Führt eine oder mehrere Anweisungen so lange aus, solange eine Bedingung wahr bleibt. Prüft diese Bedingung, nach Ausführung der Anweisungen.

Syntax

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

Parameter

Condition: Ein Ausdruck, der *TRUE* oder *FALSE* zurückgibt.

Instruction: beliebig viele XFL-Formeln.

Beispiel
Dieser Code schreibt alle Elemente des Felds *Names* in die Statuszeile.

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

GOTO

Setzt die Abarbeitung des Codes an einer Sprungmarke fort. Sprungmarken werden mit dem Schlüsselwort *LABEL* definiert.
Syntax
@Goto(*Label*)
Parameter
Label: Bezeichnung einer Sprungmarke

Die Sprungmarke muss innerhalb der selben Routine definiert sein, wie der Aufruf *@Goto()* oder in einer äußeren Funktion.

Beispiel
Eine Schleife über *@Goto()* realisiert.

```
LABEL Start;
@Prompt([OK] ; "Start" ; "Das ist der Anfang");
@If(@Prompt([yesno] ; "endlos"; "Noch einmal?") ; @Goto(Start) ; "");
```

GOSUB

Unterbricht die Abarbeitung des Codes, um an einer Sprungmarke fortzufahren. Nach *@Return()* wird die Abarbeitung an der Stelle nach *@Gosub()* fortgesetzt.
Syntax
@Gosub(*Label*)
Parameter
Label: Bezeichnung einer Sprungmarke
Rückgabewert
Der Wert, der von @Return() zurückgegeben wird

Beispiel

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

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

Die Sprungmarke muss innerhalb der selben Routine definiert sein, wie der Aufruf *@Gosub()* oder in einer äußeren Funktion. Folgende Konstruktion ist möglich:

```
@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));
```

Folgende Konstruktion hingegen ist nicht erlaubt:

```
@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

Diese Funktion beendet eine Prozedur. In einer Unterfunktion wird nur diese Unterfunktion beendet und zur aufrufenden Funktion zurückgekehrt. Nach *@Gosub()* wird ein Rücksprung zur Anweisung nach *@Gosub()* bewirkt.

Syntax

@Return(*Value*)

Parameter

Value: Wert, der an den Aufrufer zurückgegeben wird.

EVAL

Führt XFL-Code aus. Diese Funktion erlaubt es, XFL-Code dynamisch einzubinden.

Syntax

@Eval(*Code*)

Parameter

Code: Text, XFL-Code.

Rückgabe

Wert des letzten Ausdrucks des XFL-Codes.

Beispiel

Ausführung einer in einem Profildokument gespeicherten Formel

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

EVALONSERVER

Führt XFL-Code auf dem Server aus. Damit können Aktionen ggfs. mit höheren Zugriffsrechten ausgeführt werden. Diese Funktion benötigt den Agenten *(XFLOnServer).* Alle globalen Variablen sind im Code verwendbar. Zum Aufruf dieser @Formel, muss der Nutzer zumindest das Recht haben, öffentliche Dokumente zu schreiben.

Syntax

@EvalOnServer(*Code*)

Parameter
Code: Text, XFL-Code.
Rückgabe
Wert des letzten Ausdrucks des XFL-Codes.
Beispiel
Leseprotokoll in einer Datenbank, auf der die Nutzer nur Leserechte besitzen.
Im PostOpen der Maske:

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

Der Nutzer löst das Schreiben des Feldes *Log* aus, ohne dass er Autorenrechte auf das Dokument haben muss.

ISDEFINED

Prüft, ob es sich bei einem Funktionsnamen um eine über *DEFINE* definierte Funktion handelt. Eine Prüfung, ob die Funktion eine gültige Notes-@Funktion ist, wird nicht durchgeführt.
Syntax
@IsDefined(*Fname*)
Parameter
Fname: Text, Name der Funktion
Rückgabe
TRUE, wenn eine Funktion *Fname* über *DEFINE* definiert wurde, sonst *FALSE*
Beispiel

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

DEBUG

Schaltet den Debugmodus ein oder aus.
Syntax
@Debug(*Mode*)
Parameter
Mode: Bei *1* wird der Debugger aktiviert, bei *0* deaktiviert.

Der Debugger erscheint in Form einer Dialogbox. Er kann jedoch bei Bedarf umprogrammiert werden in der Funktion *XFLDebug()* in der Skriptbibliothek *XFLExtension*.

PRINT

Gibt Werte in der Statuszeile aus.
Syntax
@Print(*Value1*[; ... [; *ValueN*]])
Parameter
Value1..N: Werte beliebiger Datentypen.

Die Ausgabe der Werte erfolgt untereinander. Enthält ein Parameter eine Liste, so werden auch die Elemente dieser Liste untereinander ausgegeben. Alternativ zu *@Print()* kann auch ***@StatusBar()*** (wie SFL ab R6) verwendet werden.

EXECUTE

Führt LotusScript aus. Sie können dabei alle global deklarierten Variablen der Skriptbibliothek *XFLExtension* verwenden, auch auf die, welche Sie dort zusätzlich einbauen. Auf das aktuelle Dokument kann z.B. über die Variable *XFLRefDoc* zugegriffen werden.

Syntax

@Execute(*Code*)

Parameter

Code: LotusScript.

Rückgabe

Returncode eines End-Statements, sofern angegeben, sonst *0*.

Beispiel

Setzt das IsReaders-Flag für ein Feld.

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

GETFIELD

Gibt den Wert eines Feldes im aktuellen Dokument zurück.

Syntax

@GetField(*Fieldname*)

Parameter

Fieldname: Der Name des Feldes, dessen Wert ausgelesen werden soll.

Beispiel

```
Print(GetField(Prompt([OKCancelEdit] ; "Feld auslesen" ; "Welchen Feldwert möchten Sie wissen?" ; "")))
```

SETFIELD

Setzt ein Feld im aktuellen Dokument. Verwendung wie in SFL. Anders als in SFL ist in XFL vorher keine Deklaration des Feldes notwendig.

Syntax

@SetField(*Fieldname* ; *Value*)

Parameter

Fieldname: Der Name des Feldes, dem ein Wert zugewiesen werden soll.

Value: Wert, der zugewiesen wird.

GETDOCFIELD

Gibt einen Feldwert eines bestimmten Feldes in einem bestimmten Dokument zurück. Das Dokument muss in der gleichen Datenbank wie das aktuelle Dokument gespeichert sein. Anders als in SFL darf hier auch auf das aktuelle Dokument zugegriffen werden.

Syntax

@GetDocField(*UNID* ; *Fieldname*)

Parameter

UNID: UniversalID des Dokuments.

Fieldname: Der Name des Feldes, dessen Wert ausgelesen werden soll.

Beispiel

Diese Formel funktioniert in XFL, ist jedoch in SFL unter Notes R4/R5 noch nicht erlaubt:

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

SETDOCFIELD

Setzt ein Feld eines bestimmten Dokuments auf einen bestimmten Wert. Das Dokument muss in der gleichen Datenbank wie das aktuelle Dokument gespeichert sein. Anders als in SFL darf hier auch auf das aktuelle Dokument zugegriffen werden.

Syntax

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

Parameter

UNID: UniversalID des Dokuments.

Fieldname: Der Name des Feldes, dessen Wert gesetzt werden soll.

Value: Wert, der dem Feld zugewiesen werden soll.

GET

Gibt den Wert einer lokalen Variablen zurück.

Syntax

@Get(*Varname*)

Parameter

Varname: Name der Variablen

Beispiel

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

SET

Weist einer lokalen Variablen einen Wert zu. Es ist nicht nötig, die Variable vorher zu initialisieren, wie es in SFL bis R5 zu tun ist.

Syntax

@Set(*Varname* ; *Value*)

Parameter

Varname: Der Name einer lokalen Variablen.

Value: Wert, der der Variablen zugewiesen werden soll.

Beispiel

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

GETGLOBAL

Gibt den Wert einer globalen Variablen zurück.

Syntax

@GetGlobal(*Varname*)

Parameter

Varname: Name der globalen Variablen.

Beispiel

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

SETGLOBAL

Weist einer globalen Variablen einen Wert zu. Gleiche Bedeutung wie *@Set()* für lokale Variablen.

Syntax

@SetGlobal(*Varname* ; *Value*)

Parameter

Vaname: Der Name einer globalen Variablen.

Value: Wert, der der Variablen zugewiesen werden soll.

Beispiel

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

XFLVARIABLES

Gibt eine Liste der Namen aller gesetzten lokalen Variablen zurück.

Syntax

@XFLVariables

Beispiel

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

XFLGLOBALVARIABLES

Gibt eine Liste der Namen aller gesetzten globalen Variablen zurück.

Syntax

@XFLGlobalVariables

Beispiel

```
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

Wie in SFL, jedoch gibt es unter XFL die Beschränkung auf maximal 99 Bedingungen/Aktionen nicht mehr.

XFLVERSION

Gibt die Versionsnummer des XFL-Interpreters zurück.

Syntax

@XFLVersion

PROMPT

Öffnet ein Dialogfeld. Funktioniert wie in SFL, ist jedoch intern mittels LotusScript umgesetzt, weil leider nicht über *Evaluate()* nutzbar. Geändertes Verhalten durch Setzen der Variablen XFLDoNotQuitOnCancel (siehe Abschnitt *Global deklarierte LotusScript-Variablen*).

PICKLIST

Öffnet ein modales Fenster. Funktioniert wie in SFL, ist jedoch intern mittels LotusScript umgesetzt, weil leider nicht über *Evaluate()* nutzbar. Geändertes Verhalten durch Setzen der Variablen XFLDoNotQuitOnCancel (siehe Abschnitt *Global deklarierte LotusScript-Variablen*).

DIALOGBOX

Öffnet ein modales Fenster. Funktioniert wie in SFL, ist jedoch intern mittels LotusScript umgesetzt, weil leider nicht über *Evaluate()* nutzbar.

SAVEDOCUMENT

Speichert das aktuelle Dokument. Wird eine Formel über *Evaluate()* oder *XFLExecute()* auf ein Dokument ausgeführt, so ist es leider nicht möglich zu erkennen, ob die Ausführung Änderungen am Dokument bewirkt hat. Um dies zu ermitteln, müssen daher nach *Evaluate()* seit jeher zunächst ein entsprechender Test und danach ggfs. das Speichern in LotusScript erfolgen. XFL bietet hier eine zusätzliche Möglichkeit.

Syntax

@SaveDocument

Beispiel

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

Wenn sich das Feld *Name* ändert, wird hier das Dokument gespeichert.

CREATEOBJECT

Erzeugt ein Objekt. Weitere Informationen im Abschnitt *Objektorientierte Programmierung mit XFL*.

Syntax

@CreateObject(*ClassName* [; *args*])

Parameter

ClassName: Bezeichnung der Klasse.

Der Klassenname kann entweder einer nativen LotusScript-Klasse entsprechen, z.B. "NotesSession" oder einer eigenen Klasse. Eigene Klassen sind in der Scriptbibliothek *XFLExtension* zu definieren. Wenn Klassen in anderen Bibliotheken definiert sind, so müssen diese Bibliotheken per *"USE ..."* in *XFLExtension* eingebunden werden.

args: Initialisierungsparameter, optional.

Diese Parameter werden der Methode *NEW* übergeben.

Beispiel

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

SETOBJECT

Setzt eine Objektvariable. Weitere Informationen im Abschnitt *Objektorientierte Programmierung mit XFL*.

Syntax

@SetObject(*Oname* ; *Object*)

Parameter

Oname: Text. Objektbezeichner.

Object: Objekt, welches zugewiesen wird.

Beispiel

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

Der Code ist identisch mit

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

SETGLOBALOBJECT

Setzt eine globale Objektvariable. Weitere Informationen im Abschnitt *Objektorientierte Programmierung mit XFL*.

Syntax

@SetGlobalObject(*Oname* ; *Object*)

Parameter
Oname: Text. Objektbezeichner.
Object: Objekt, welches zugewiesen wird.

GETGLOBALOBJECT

Gibt eine globale Objektvariable zurück. Wenn die Variable nicht existiert, wird *@Nothing* zurückgegeben. Weitere Informationen im Abschnitt *Objektorientierte Programmierung mit XFL*.
Syntax
@GetGlobalObject(*Oname*)
Parameter
Oname: Text. Objektbezeichner.
Beispiel

```
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

Gibt eine Liste der Namen aller gesetzten globalen Objektvariablen zurück. Weitere Informationen im Abschnitt *Objektorientierte Programmierung mit XFL*.
Syntax
@XFLGlobalObjects
Beispiel

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

NOTHING

Initialwert für eine Objektvariable (wie *Nothing* in LotusScript). Weitere Informationen im Abschnitt *Objektorientierte Programmierung mit XFL*.
Syntax
@Nothing
Beispiel

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

ISNOTHING

Prüft, ob eine Objektvariable den Initialwert Nothing besitzt. Weitere Informationen im Abschnitt *Objektorientierte Programmierung mit XFL*.
Syntax
@IsNothing(*Object*)
Beispiel

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

ERROR

Erweiterung der SFL-@Funktion *@Error* um die Möglichkeit, einen LotusScript-Laufzeitfehler auszulösen. Funktionsweise entspricht dem *ERROR*-Statement von LotusScript.

Syntax 1

@Error

Funktionsweise wie in SFL.

Syntax 2

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

Parameter

errNumber: Fehlernummer.

errMessage: Optional. Fehlertext.

XFL-spezifische Fehlertexte sind in der Funktion *XFLErrorText()* der Scriptbibliothek *XFLExtension* definiert. Dort können auch weitere nutzerspezifische Fehlertexte hinterlegt werden. Wird *@Error()* ohne *errMessage* aufgerufen, versucht der Interpreter, einen Fehlertext über den Aufruf der Funktion *XFLErrorText()* zu ermitteln.

Beispiel

```
        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$ + ")"
        ' ...
```

Operatoren

And- und Or-Operator ("&" und "|")

Bei einer *AND*-Verknüpfung zweier Ausdrücke werden unter SFL zunächst beide Ausdrücke geprüft und danach die beiden Ergebnisse mit *AND* verknüpft. Dies dürfte in vielen Fällen ineffizient sein, da bereits nach Prüfung nur einer der beiden Ausdrücke das Ergebnis feststehen kann, nämlich dann, wenn dieser Ausdruck *FALSE* ist und damit auch die Verknüpfung *FALSE* wird. Analog ergibt eine *OR*-Verknüpfung schon bei einem „wahren" ersten Ausdruck *TRUE*. In diesen Fällen kann auf die Ausführung des zweiten Ausdrucks verzichtet werden. XFL verwendet diese optimierte Logik.

Beispiel

Hier wird eine doppelte Sicherheitsabfrage durchgeführt.

```
@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");
```

Unter SFL wird nutzloser Weise die zweite Frage auch bei Verneinung der ersten Frage angezeigt. Unter XFL nicht.

Index Operator "[ ]"

Die Elemente einer Liste können über einen Index angesprochen werden. Dieser wird in eckige Klammern hinter die Variable geschrieben.

Beispiel

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

SFL kennt ab Notes R6 diesen Operator zwar auch, jedoch mit der Einschränkung, dass er nur den lesenden Zugriff auf Listenelemente erlaubt. So kann folgender Code in SFL unter Notes R6 nicht ausgeführt werden:

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

Erweiterte Feldsuche durch Operator "?"

Der Operator ? bietet eine einfache Möglichkeit, auf Felder anderer Dokumente zurückzugreifen. Dieser Operator wird hinter einen Feldnamen geschrieben. Dies bewirkt bei Fehlen dieses Feldes im aktuellen Dokument eine erweiterte Suche nach diesem Feld in alternativen Dokumenten. Welche alternativen Dokumente durchsucht werden, wird über die Funktion *XFLGetAltRefDoc()* in der Skriptbibliothek *XFLExtension* gesteuert. In der Standardfassung ist in dieser Funktion die Anwortdokumenthierarchie umgesetzt, d.h. die Suche im nächst übergeordneten Elterndokument. Es lassen sich hier aber auch andere Mechanismen entsprechend den Strukturen Ihrer Applikation realisieren, wie z.B. Zugriff auf separate Konfigurationsdatenbanken.

Beispiel

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

Wenn das aktuelle das Feld *Categories* nicht hat, wird das Feld in den übergeordneten Elterndokumenten gesucht. Danach wird der Feldwert in das aktuelle Dokument geschrieben.

Weitere Besonderheiten

Stringbegrenzer { } und ' '

Strings können nicht nur in Gänsefüsschen "" eingeschlossen werden, sondern auch in geschweifte Klammern {} und einfache Hochkommas. Funktional besteht kein Unterschied. Man kann sich dadurch die Maskierung von Gänsefüßchen innerhalb von Stringkonstanten sparen.

Verwendung von Schlüsselworten

Im Unterschied zu SFL können Schlüsselworte nicht nur vor Hauptausdrücken stehen, sondern auch innerhalb von Ausdrücken.

Beispiel

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

Unter Notes R6 ist diese starke Einschränkung teilweise aufgehoben. So etwas kann aber auch R6 noch nicht:

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

Präfix @ bei Notesfunktionen ist optional

Es können alle @Funktionen von Notes verwendet werden. Dabei kann das Zeichen @ auch weggelassen werden.

Beispiel

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

ist identisch mit

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

Ausnahme bilden die @Funktionen, die keine Parameter besitzen, da dort das @ als Unterscheidung zwischen Funktionsnamen und Bezeichnern von Feldern oder Variablen dient. *@Today* ergibt das heutige Datum, *Today* ergibt den Wert der Variablen oder des Feldes *"Today"*.

Einfache Klammer ersetzt @Do

In SFL müssen gruppierte Anweisungen von *@Do()* umschlossen sein. Das ist mit XFL nicht nötig. Eine einfache Klammer reicht aus.

Beispiel

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

ist identisch mit

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

Eingschänkte Verschachtelungstiefe

Der XFL-Interpreter arbeitet verschachtelte Ausdrücke rekursiv ab. Rekursion in LotusScript-Code ist allerdings nicht endlos erlaubt. LotusScript generiert einen „Out of stack space“-Fehler, sobald die Tiefe überschritten wird. Unter SFL ist die Grenze höher, so dass extrem verschachtelte Ausdrücke, die unter SFL laufen, unter XFL möglicherweise zu diesem Fehler führen. In diesen Fällen sollte man den Ausdruck teilen und mit Zwischenergebnissen arbeiten oder auf eine neuere Notesversion hoffen, die eine tiefere Verschachtelung erlaubt.

Beispiel
Diese geklammerte Summierung von 50 Einsen funktioniert unter SFL, unter XFL nicht. Ohne Klammern tritt das Problem nicht auf.

```
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 Error
```

Ohne Klammern gibt es jedoch auch bei XFL kein Problem.

Feld- und Variablennamen mit Punkten "."

Da der Punkt "." in XFL für die objektorientierte Programmierung verwendet wird (siehe Abschnitt *Objektorientierte Programmierung mit XFL*), sind Punkte im Namen von Bezeichnern besonders zu behandeln. Nicht erlaubt sind Anweisungen wie:

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

Es muss hier auf die @Funktionen *@Set()*, *@Get()*, *@SetField()* bzw. *@GetField()* ausgewichen werden, z.B.:

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

Die XFL-Laufzeitumbegung

Ausführung von XFL-Code

Ausgeführt wird XFL-Programmcode über einen Interpreter. Dieser wurde mit LotusScript programmiert und wird in Form zweier Scriptbibliotheken (*XFLEngine* und *XFLExtension*) bereitgestellt. Die Ausführung von XFL-Code erfolgt über den Aufruf einer Scriptfunktion, z.B. *XFLExecute()*. Der Interpreter läuft bereits ab Notes 4.6x und kann überall dort eingesetzt werden, wo Scriptcode möglich ist, sowohl im Client als auch in Serveragenten. Nicht einsetzbar ist XFL z.B. in Ansichtsformeln. Um Feldformeln in XFL abzubilden, müssen Ereignisroutinen genutzt werden, in denen LotusScript erlaubt ist, z.B. *PostOpen* oder *QuerySave*.

Die Ausführung von XFL-Code erfolgt in zwei Stufen:

1. Umformung des Quellcodes in eine Funktionsbaumstruktur.
   Wenn syntaktische Fehler im Quellcode vorhanden sind, führen diese zu einer Fehlermeldung mit konkreter Fehlerursache und Position.
2. Durchlaufen des Funktionsbaums.
   Die Funktionen und Operationen werden in drei Typen unterschieden:
   1. Standard-Notes-@Funktionen und -Operatoren (SFL)
      Diese werden über *Evaluate()* ausgeführt.
   2. Standard-XFL-Funktionen und -Operatoren
      Diese sind in LotusScript in der Bibliothek XFLEngine implementiert.
   3. selbst definierte Funktionen
      Wenn fest in Scriptbibliothek *XFLExtension* kodiert, erfolgt Ausführung über den Aufruf von *XFLExecuteFunction()*.
      Wenn über einen *DEFINE*-Ausdruck definiert, erfolgt die Ausführung durch Ausführung dieser Definition.

Skriptbibliothek XFLEngine

Kern des Interpreters. Stellt folgende Funktionalitäten bereit:
- Funktionen zur Ausführung und Prüfung von XFL-Code
- Funktionen zum Zugriff auf globale XFL-Variablen und Objekte

Diese Bibliothek verwendet wiederum Funktionen und Variablen der Skriptbibliothek *XFLExtension*.

Function XFLExecute

Führt eine XFL-Formel aus.

Syntax

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

Parameter

Formula: XFL-Code.

doc: Referenzdokument, auf das die Formel angewandt wird.

Rückgabe

Ergebnis des zuletzt ausgeführten Ausdrucks der Formel. Während die LotusScript-Funktion *Evaluate()* immer ein Array zurückgibt, tut *XFLExecute()* dies nur dann, wenn das Ergebnis eine Liste mit mehreren Werten ist.

Beispiel

Beispiel für eine Maskenschaltfläche. Berechnet für zwei Datumswerte das Datum des nächsten Jubiläums. Das Beispiel zeigt, wie leicht sich Funktionen definieren lassen.

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

Const formula = |
REM "Algorithmus als Funktion definieren";
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

Führt eine XFL-Formel auf das angezeigte Dokument aus. Diese Funktion kann in Maskenschaltflächen oder Agenten verwendet werden. Man erspart sich dadurch etwas Schreibarbeit im Vergleich zur Nutzung von *XFLExecute()*. Intern ist diese Funktion folgendermaßen programmiert:

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

Function XFLExecuteOnUIView

Führt eine XFL-Formel auf die in einer Ansicht markierten Dokumente aus. Diese Funktion kann in Ansichtsschaltflächen oder Agenten verwendet werden. Man erspart sich dadurch etwas Schreibarbeit im Vergleich zur Nutzung von *XFLExecute()*. Intern ist diese Funktion folgendermaßen programmiert:

```
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

Führt XFL-Code auf dem Server aus. Damit können Aktionen ggfs. mit höheren Zugriffsrechten ausgeführt werden. Diese Funktion benötigt den Agenten *(XFLOnServer)*. Alle globalen Variablen sind im Code verwendbar. Zum Aufruf von *XFLExecuteOnServer()* muss der Nutzer zumindest das Recht haben, öffentliche Dokumente zu schreiben.

Syntax

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

Parameter

Formula: XFL-Code.

doc: Referenzdokument, auf das die Formel angewandt wird.

Rückgabe

Wert des letzten Ausdrucks des XFL-Codes.

Beispiel

Leseprotokoll in einer Datenbank, auf der die Nutzer nur Leserechte besitzen.

Im PostOpen der Maske:

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

Der Nutzer löst das Schreiben des Feldes *Log* aus, ohne dass er Autorenrechte auf das Dokument haben muss.

Sub XFLCheckSyntax

Untersucht einen Code auf Fehler in seiner Struktur. Dabei wird z.B. untersucht, ob Klammern richtig gesetzt sind. Bei Auftreten eines Fehlers wird ein Laufzeitfehler generiert, der detaillierte Informationen zum Fehler enthält. Die Prüfung bezieht sich nur auf die Struktur, und nicht darauf, ob die Funktionsnamen richtig geschrieben sind. Nur bei einigen Funktionen, z.B. *@If()*, wird auch auf die korrekte Anzahl der Parameter geachtet.

Syntax

Sub XFLExecute(*Formula* as String)

Parameter

Formula: XFL-Code.

Beispiel

In diesem Fall wird ein Fehler wegen falscher Klammersetzung ausgelöst.

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

Sub XFLSetGlobalVar

Setzt eine globale XFL-Variable. Diese kann in nachgelagertem XFL-Code weiterverwendet werden.

Syntax

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

Parameter

VarName: Name der XFL-Variablen.

Value: Wert, der der globalen Variablen zugewiesen werden soll.

Beispiel

Hier wird im Debugmode gut deutlich, wie der Interpreter arbeitet.

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

Function XFLGetGlobalVar

Liest eine globale XFL-Variable aus. Globale Variablen können in XFL-Code oder über die Funktion *XFLSetGlobalVar()* gesetzt werden..

Syntax

Function XFLGetGlobalVar(*VarName* As String) As Variant

Parameter

VarName: Name der XFL-Variablen.

Rückgabe

Wert der XFL-Variable

Wenn die Variable nicht existiert, wird *EMPTY* zurückgegeben.

Beispiel

Datenaustausch zwischen LotusScript und XFL.

```
Use "XFLEngine"
Const Formula = |GLOBAL NewText := @ReplaceSubstring(GLOBAL Text ; " " ; "")|
Call XFLSetGlobalVar("Text" , "Dies ist ein Beispiel")
Call XFLExecute(Formula, Nothing)
Print XFLGetGlobalVar("Text") & " -> " & XFLGetGlobalVar("NewText")
```

Sub XFLDeleteGlobalVar

Löscht globale XFL-Variablen.

Syntax

Sub XFLDeleteGlobalVar(*VarNames* As Variant)

Parameter

VarNames: Name oder Array von Namen globaler XFL-Variablen.

Beispiel

Datenaustausch zwischen Script und XFL.

```
Const Formula = |GLOBAL NewText := @ReplaceSubstring(GLOBAL Text ; " " ; "")|
Call XFLSetGlobalVar("Text" , "Dies ist ein Beispiel")
Call XFLExecute(Formula, Nothing)
Print XFLGetGlobalVar("Text") & " -> " & XFLGetGlobalVar("NewText")
```

```
Call XFLDeleteGlobalVar("Text")
Print XFLGetGlobalVar("Text") & " -> " & XFLGetGlobalVar("NewText") ' jetzt fehlt der erste Text
```

Function XFLGlobalVariables

Liefert eine Liste aller gesetzten globalen XFL-Variablen.

Syntax

Function XFLGlobalVariables As Variant

Rückgabe

Stringarray mit den Variablennamen.

Beispiel

Datenaustausch zwischen Script und XFL.

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

Function XFLCreateGlobalObject

Erzeugt eine globale Objektvariable, die in nachgelagertem XFL-Code verwendet werden kann.

Syntax

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

Parameter

oname: Name der Objektvariablen.

ClassName: Bezeichnung der Klasse.

Der Klassenname kann entweder einer nativen LotusScript-Klasse entsprechen, z.B. "NotesDocument" oder einer eigenen Klasse. Eigene Klassen sind in der Scriptbibliothek *XFLExtension* zu definieren. Wenn Klassen in anderen Bibliotheken definiert sind, so müssen diese Bibliotheken per *"USE ..."* in *XFLExtension* eingebunden werden.

args: Initialisierungsparameter

Diese Parameter werden der Methode *NEW* übergeben.

Rückgabe

Erzeugtes Objekt.

Beispiel

Austausch von Objekten zwischen LotusScript und XFL.

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

Siehe auch Abschnitt *Objektorientierte Programmierung mit XFL*.

Function XFLSetGlobalObject

Setzt eine globale Objektvariable.

Syntax

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

Parameter
oname: Name der Objektvariablen.
obj: Objekt, das der Variablen zugewiesen werden soll.
Beispiel
Austausch von Objekten zwischen LotusScript und XFL.

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

Siehe auch Abschnitt *Objektorientierte Programmierung mit XFL*.

Function XFLGetGlobalObject

Liefert eine globale Objektvariable.
Syntax
Function XFLGetGlobalObject(*oname* As String) as Variant
Parameter
oname: Name der Objektvariablen.
Rückgabe
Globales XFLObjekt, sofern ein globales Objekt dieses Namens existiert, sonst *Nothing*.
Beispiel
Austausch von Objekten zwischen LotusScript und XFL.

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

Siehe auch Abschnitt *Objektorientierte Programmierung mit XFL*.

Sub XFLDeleteGlobalObject

Löscht globale Objektvariablen.
Syntax
Sub XFLDeleteGlobalObject(*ObjectNames* As Variant)
Parameter
ObjectNames: Name oder Array von Namen globaler Objektvariablen.
Beispiel
Austausch von Objekten zwischen LotusScript und XFL.

```
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")
```

Siehe auch Abschnitt *Objektorientierte Programmierung mit XFL*.

Function XFLFormatCode

Formatiert einen Formelausdruck. Dabei wird die Struktur der Formel aufgelöst und eingerückt dargestellt. Überflüssige Leerzeichen werden entfernt

Syntax

Function XFLFormatCode(*Code* As String) as String

Parameter

Code: Formel, die formatiert werden soll.

Rückgabe

Formel in neuem Format.

Beispiel

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

Ausgabe:

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

Skriptbibliothek XFLExtension

Die Bibliothek *XFLExtension* enthält Funktionen, die von aus *XFLEngine* aufgerufen werden. Hier hat der Programmierer vielseitige Möglichkeiten der Anpassung und Erweiterung der Sprache.

Global deklarierte LotusScript-Variablen

Folgende Variablen sind von Bedeutung:
XFLRefDoc as NotesDocument
Wird durch die Funktion *XFLExecute()* auf das übergebene NotesDocument-Objekt gesetzt. Kann in den Funktionen dieser Bibliothek verwendet werden.
XFLDebugMode as Integer
Kann zum Ein- oder Ausschalten des Debuggers verwendet werden. Diese Variable ändert sich bei Aufruf der @Funktion *@Debug()*.
XFLExtendedFunctions as String
Hier werden selbst definierte Funktionen angegeben. Diese Variable muss im *Initialize* gesetzt werden und wird bei der Laufzeit von XFL-Code verwendet, um zu ermitteln, welche XFL-Funktionen in der Bibliothek *XFLExtension* definiert wurden und daher über *XFLExecuteFunction()* auszuführen sind. Wenn Sie mehrere Funktionsnamen angeben, sind diese mit Komma zu trennen.
XFLInit as String
Hier kann Formelcode hinterlegt werden, der beim Initialisieren der Bibliothek *XFLEngine* ausgeführt wird. Diese Variable muss im *Initialize* gesetzt werden. Diese Variable eignet sich besonders dazu, *ALIAS*- und *DEFINE*-Statements vorab zu laden oder globale Variablen zu setzen.
XFLDoNotQuitOnCancel As Integer
Wird in einem *@Prompt()*- oder *@Picklist()*-Dialog „Abbrechen“ gewählt, so wird in SFL dadurch der gesamte Formelcode abgebrochen. Eine Reaktion auf dieses Ereignis ist dadurch nicht ohne weiteres programmierbar. Das Setzen der Variable *XFLDoNotQuitOnCancel* auf *1* bewirkt, dass *@Prompt()* im Abbruchfall *-1* zurückgibt und der Code fortgesetzt wird.

Function XFLExecuteFunction

Diese Routine wird immer dann während der Ausführung einer Formel aufgerufen, wenn eine Funktion ausgeführt wird, deren Name über *XFLFunctions* angemeldet wurde.
Syntax
Function XFLExecuteFunction(*fname* As String, *Params* As Variant, *IsXFL* As Integer) As Variant
Parameter
Fname: Name der Funktion.
Params: übergebene Parameter.
IsXFL: Rückgabewert. Wenn der Wert innerhalb dieser Routine auf *TRUE* gesetzt wird, wird das Ergebnis wiederum als XFL-Formel interpretiert und danach ausgeführt.
Rückgabe
Wert der Funktion.

Eine eigene XFL-Funktion können Sie folgendermaßen programmieren:
1. Funktionsnamen in Variable *XFLExtendedFunctions* hinterlegen.
2. In Sub *XFLExecuteFunctions()* ausprogrammieren.

Beispiel 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": ' <- immer in Großbuchstaben!
                Messagebox "Hallo Welt! " + Params(0) ' Diese @Funktion soll mit einem Parameter aufgerufen werden
                XFLExecuteExtendedFunction = True
        End Select
End Function
```

Das ist schon alles. Testen wir das über einen einfachen Agenten:

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

Es ist zu beachten, dass Funktions- und Variablennamen **nicht** casesensitiv behandelt werden, sondern intern in Großbuchstaben verwaltet werden. Es ist daher unerheblich, ob der XFL-Code @Funktionen in Groß- oder Kleinschreibung aufweist. Im obigen Beispiel etwa können Sie im Aufruf *XFLExecute()* auch *" ... @mYfUnCtIoN(...) ... "* schreiben. Der @Funktionsname wird der Funktion *XFLExecuteFunction()* immer in **Großbuchstaben** übergeben.

Beispiel 2

Das Ergebnis einer selbst definierten Funktion kann auch wiederum XFL-Code sein, der danach ausgeführt werden soll. In diesem Fall ist der Parameter *IsXFL* auf *TRUE* zu setzen. Hier die Definition der Funktion *@2dnRight*, welche nach dem zweiten Vorkommen eines Suchstrings in einem String sucht.

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

Testen wir die Funktion:

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

Sub XFLDebug

Diese Routine wird während der Ausführung eines XFL-Codes nach jeder Funktion oder Operation vom Interpreter aufgerufen, wenn der XFL-Debugger aktiviert ist. Das aktuelle Zwischenergbnis und Variablen werden dargestellt. Der Debugmodus kann entweder über die globale LotusScript-Variable *XFLDebugMode* oder die XFL-Funktion *@Debug()* geändert werden.

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)

Parameter
Code: Formelcode, der ausgeführt wird.
StartPos: Position des ersten Zeichens des ausgeführten Ausdrucks in der Codezeichenfolge.
EndPos: Position des letzten Zeichens des ausgeführten Ausdrucks in der Codezeichenfolge.
Value: Zwischenergebnis.
Locals: Lokale Variablen.
Globals: Globale Variablen.
LocalObjects: Lokale Objekte.
GlobalObjects: Globale Objekte.

Die Variablen werden über Felder in temporären Dokumenten übergeben, da LotusScript nicht alle Formel-Datentypen darstellen kann, z.B. *@Error* oder Referenzen. Die ausgelieferte Routine zeigt die Werte lediglich an. Bei Bedarf können Sie die Routine auch um Manipulationsmöglichkeiten der Variablen erweitern, wie man es auch von gängigen Debuggern kennt. Dazu müsste ein Mechanismus programmiert werden, der die Felder in den übergebenen Dokumenten ändert.

Sub XFLPrepareCode

Diese Routine wird vor Ausführung jedes Codes intern aufgerufen. Hier können noch Änderungen am Code vorgenommen werden oder Aktionen, z.B. zur Protokollierung ausgeführt werden. Diese Funktion kann z.B. auch in Verbindung eines SmartIcons verwendet werden, das den Debugger aktiviert oder deaktiviert.
Syntax
Sub XFLPrepareCode(*Code* as String)
Parameter
Code: Code, der ausgeführt werden soll.

Sub XFLCommand

Diese Routine wird während der Ausführung einer Formel vom Interpreter aufgerufen, sobald ein *@Command([])* auftritt.
Syntax
Sub XFLCommand(*cname* As String, *params* As Variant)
Parameter
Cname: Name des Befehls.
Params: Parameter des @Commands.

Implementieren Sie hier die @Commands, die Sie in den XFL-Formeln verwenden möchten. Diese Routine dient der Herstellung der Kompatibilität zu SFL. Zunächst sind nur *@Command([FileSave])* und *@Command([ViewRefreshFields])* umgesetzt. Nativer Notes-Formelcode mit @Commands ist damit über den XFL-Interpreter ausführbar. @Commands, die nicht an dieser Stelle implementiert sind, bleiben aber wirkungslos.

Function XFLGetAltRefDoc

XFL bietet den Operator "?" um auf Felder anderer Dokumente zuzugreifen. Der Operator wird hinter einen Feldnamen geschrieben. Dies bewirkt bei Fehlen dieses Feldes im aktuellen Dokument eine erweiterte Suche in alternativen Dokumenten. Diese Suche wird über die Funktion *XFLGetAltRefDoc()* gesteuert. In der Standardfassung ist in dieser Funktion die

Anwortdokumenthierarchie umgesetzt, d.h. die Suche im nächstübergeordneten Dokument (verlinkt über das Feld *$Ref*). Es lassen sich hier aber auch andere Mechanismen definieren.

Syntax

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

Parameter

doc: Dokument, welches das Feld nicht enthalten hat.

fieldname: Name des gesuchten Feldes.

Rückgabe

Dokument, in dem weiter nach dem Feld gesucht werden soll.

Die Funktion wird so lange ausgeführt, bis sie *NOTHING* zurückgibt oder das Feld gefunden wurde.

Beispiel

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

Wenn *doc* das Feld *Categories* nicht hat, wird zunächst das Feld in den übergeordenten Dokumenten gesucht und dieser Feldwert in *doc* gesetzt.

Agent (XFLOnServer)

Dieser Agent wird von der Formel *@EvalOnServer()* und der Script-Funktion *XFLExecuteOnServer()* verwendet. Der Code, der an *@EvalOnServer()* oder *XFLExecuteOnServer()* übergeben wird, wird im Namen des Unterzeichners dieses Agenten ausgeführt.

Teilmaske XFLExtensionR4

Diese Teilmaske ist notwendig, wenn Ihre Anwendungen unter Notes R4 lauffähig sein sollen. Sie wird verwendet von den @Funktionen *@Prompt()* und *@Picklist()*, da LotusScript in R4 kein Äquivalent zu diesen Funktionen unterstützt. Wenn Sie ausschließlich NotesR5 oder höher einsetzen, ist die Teilmaske nicht nötig.

Objektorientierte Programmierung mit XFL

SFL ist keine objektorientierte Sprache. Mit XFL hingegen lassen sich Formelsprache und Objektorientierung kombinieren.

Es können alle vorhandenen LotusScript-Klassen ohne weiteres im XFL-Formelcode verwendet werden. Eigene Klassen sind in der Scriptbibliothek *XFLExtension* zu definieren bzw. dort per *"USE ..."* einzubinden, sofern sie in anderen Bibliotheken definiert sind.

Ein Objekt wird in XFL über die Funktion *@CreateObject()* oder in LotusScript über die Funktion *XFLCreateGlobalObject()* erzeugt. Bereits instantiierte LotusScript-Objekte können per *XFLSetGlobalObject()* einer Formel bereitgestellt werden.

In XFL sind Objektvariablen über das Schlüsselwort *OBJECT* als solche zu kennzeichnen.

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

Ein Objekt kann über die Methoden seiner Klasse angesprochen werden. Diese werden in klassischer Punktnotation angegeben. Dabei sind auch Verkettungen mehrerer Methoden möglich.

```
REM {Beispiel zum Setzen des IsAuthors-Flags eines Feldes. Das ist mit einfachen Formeln nicht möglich};
OBJECT session := @CreateObject("NotesSession");
OBJECT mydoc := session.currentdatabase.AllDocuments.GetFirstDocument; REM {Verkettung};
mydoc.Editors := @Username;
OBJECT it := mydoc.GetFirstItem("Editors");
It.IsAuthors := @True; REM {Ändern von Objekteigenschaften durch Zuweisungsoperator :=}
```

Wie auch bei Variablen gibt es in XFL die Möglichkeit, lokale und globale Objekte zu unterscheiden. Dazu stehen die Schlüsselworte *OBJECT*, *OBJECT GLOBAL* und die @Funktionen *@SetObject()* und *@SetGlobalObject()* zur Verfügung.

Globale Objekte können wiederum auch in LotusScript über die Funktionen *XFLSetGlobalObject()* und *XFLGetGlobalObject()* angesprochen werden.

Das aktuelle Dokument ist standardmäßig in der globalen Objektvariablen *doc* gespeichert. Folgende Formel ist daher ohne zusätzliche Variablenzuweisungen lauffähig:

```
doc.GetFirstItem("Editors").IsAuthors := @True; REM {Nutzung der globalen Objektvariable DOC};
```