Extending Ninox with NX

Introduction

NX, the Ninox query language, is a powerful programming language which allows you to quickly extend Ninox databases with calculations and trigger actions.
While Ninox provides a visual function editor as described in the online manual, it also sports a text-mode for entering more complex expressions. This manual covers the text mode.
There's an example database — covering most of the topics of the tutorial chapter — available at:

http://templates.ninoxdb.de/en/NX-Language-Tutorial.ninox

Contents

NX Tutorial

Basic Arithmetics

NX provides arithmetic operations comparable to most other programming languages:

```
1 + 2          = 3
3 - 2          = 1
2 * 3          = 6
10 / 5         = 2
```

Arithmetic operations respect the typical *operator precedence*:

```
1 + 2 * 3      = 1 + (2 * 3) = 7
```

Expressions may be enclosed in parentheses:

```
(1 + 2) * 3    = 9
```

Note, that NX will automatically remove insignificant parentheses, i. e. if you enter 1 + (2 * 3) it will store this expression as 1 + 2 * 3.

String Operations

A string, i. e. a sequence of characters, may be entered enclosed in double-quotes:

```
"Hello World!"
"This is a string."
```

Strings may be concatenated using the + operator:

```
"Hello World!" + " " + "This is a string."      = "Hello World! This is a String."
```

Strings may also be concatenated with other values:

```
"Hi " + 42                                      = "Hi 42"

2 + " fast "+ 4 + " U"                          = "2 fast 4 U"
```

To enclose a double-quote within a string, it has to be escaped as "":

```
"Hi, my name is ""Sally""!"
```

Working with Fields

Most NX functions operate in the context of a specific record. NX gives you access to all fields of that record. Consider a table *Customer* consisting of fields *First Name*, *Last Name*, *Street*, *ZIP*, *City*. To generate the full address line, you may enter:

```
Street + ", " + ZIP + " " + City
```

Some field names may contain special characters like spaces, dots, colons and so on. Also they may collide with a reserved keyword of NX, like **if**, **then**, **else**. To work around this, field names can be enclosed in single quotes:

```
'First Name' + " " + 'Last Name'
```

If a field name contains a single quote, this has to be escapes as '':

```
'Last year''s total revenue'
```

Note, that NX is agnostic to field name changes. That is, if you've entered an expression referring to a field and you change that field's name afterwards it will not have an impact to the expression. This is due to the fact that NX internally stores the expression with a reference to the field's ID (which cannot me modified). Actually, if you open the function editor again, it will reflect the field's new name.

Working with Table References

NX expressions can navigate along table references as well. Table references are always one-to-may relationships, e.g. *one* Customer may have *many* Invoices — where the table reference is defined as *from* Invoice *to* Customer (Invoice => Customer).

To display the Customer's name in an Invoice, you'd write:

```
Customer.Name
Customer.'First Name' + " " + Customer.'Last Name'
```

Expressions may even jump along multiple hops, consider the case where a Customer has a reference to a Company. You could then display the Company's in the Invoice as well:

```
Customer.Company.Name
```

Vice-versa, it's also possible to get Invoice information for a Customer. Remember, that a Customer may have multiple Invoice's. Thus, an expression referring to the Customer's invoices will return multiple values. In order to display those values, some kind of *aggregation* has to be applied. The most common one is to sum up values:

```
sum(Invoice.Amount)
```

But there are other aggregations as well:

```
avg(Invoice.Amount)     the average, ignoring empty fields
cnt(Invoice.Amount)     the count of non-empty fields
min(Invoice.Amount)     the minimum value, ignoring empty fields
```

max(Invoice.Amount)	the maximum value, ignoring empty fields
first(Invoice.Amount)	the first value, according to Ninox' internal sorting
last(Invoice.Amount)	the first value, according to Ninox' internal sorting
concat(Invoice.Amount)	lists all values, separated by ", "

Logical Expressions

Logical expressions check if something is the case or not. Most often you'll need them to create specific filter rules or to make decisions within a calculation. A logical expression either returns true or false. The most common form of a logical expression is a comparison of two values, like:

Amount > 100	greater than
Amount < 100	less than
Amount >= 100	greater than or equal to
Amount <= 100	less than or equal to
Amount = 100	equal
Amount != 100	not equal

Logical expressions may be combined by **and**, **or**, **not**.

Age > 12 **and** Age < 18
Status = 1 **or** Status = 2 **or** Status = 3
not (Status = 4 **or** Status = 5)
Status != 4 **and** Status != 5

Note, the operator precedence is **not** > **and** > **or**.

Making Decisions

In the previous chapter you've learned about logical expressions. The result of such an expression may be used in a an **if / then / else** expression:

```
if Age < 18 then "Child" else "Grown-up"

if Age < 18 then
        if Age < 13  then
                "Child"
        else
                "Teenager"
else
        "Grown-up"
```

Please note, that the **then** and **else** part have to return values of the same *type* — see chapter "Understanding Types".

Filtering Values

In the chapter "Working with Table References" you've already learned how to access multiple values from a table referring the current one — like sum(Invoice.Amount). With logical expressions, these results may also be filtered. A filter expression has to be enclosed in braces:

```
sum(Invoice[Status = 2].Amount)
```

This will only sum up the Invoice Amounts of Invoices with Status = 2.

Calling Functions

NX provides a range of built-in functions allowing you to transform values. A function call has the form:

```
function ( argument1, argument2, … )
```

Some examples:

age(Birthdate)	The current age of a person with given Birthdate in years
cos(5)	Cosinus of 5

Using Variables

Sometimes it can be useful to store the result of a intermediate calculation and do further calculations based on that result. Consider the case where you want to check the age of a person like:

```
if age(Birthdate) > 18 then "Grown-up"
else if age(Birthdate) > 12 then "Teenager"
else "Child"
```

Since age(Birthday) is used multiple times, things may be simplified with a variable which stores the result of the age calculation:

```
let a := age(Birthdate);
if a > 18 then "Grown-up" else if a > 12 then "Teenager" else "Child"
```

A variable is declared with a **let** statement:

```
let variable := expression;
```

Any expression following that let statement can make use of that variable.

Modifying Data

Some expressions in NX may also modify data. Up to the current release 1.5 of Ninox, this is only allowed for trigger expressions (field option **Trigger after update**). While further enhancements are planned, modifying data is currently restricted to change the value of record fields.

As an example, consider a table *Article* with a field *Price* (number) and another table *Invoice Item* with fields *Article* (reference to table Article) and *Price* (number). After assigning the Article, Ninox shall copy the Article's Price to the Invoice Item's Price. This can be achieved with a **Trigger after update** on Article containing following expression:

```
Price := Article.Price
```

It is also possible, to update multiple fields, using a semicolon:

```
Price := Article.Price;
Description := Article.'Article No' + " " + Article.Name
```

Function Reference

Type Conversion

number(value)

Tries to interpret the given value as a number.

- If value stems from a choice field, this will be the choice's internal id. Use number(text(choice)) to get a numeric representation of the choice's text.
- If value is a date, time or timestamp, this will be the the number of milliseconds between midnight of January 1, 1970 and the specified date.
- If value is an appointment, it is treated like the appointment's begin date.

```
number("10")        => 10
number(5)           => 5
number(now())       => 1440681777046
```

string(value)

Converts any value to a string representation possibly reflecting the format option of its field settings. If there's no value, the result will be the empty string: "".

```
text("Hello")               => "Hello"
text(2.34)                  => "2.34"
text(MyCurrencyField)       => "1,234.56 $"
text(today())               => "08/27/2015"
```

Mathematical Functions

round(x) Rounds a number to the nearest integer.

floor(x) Rounds a number DOWNWARDS to the nearest integer.

ceil(x) Rounds a number UPWARDS to the nearest integer.

sqrt(x) The square root of x.

sqr(x) The square of x: x^2

sign(x) The signum of x:

```
sign(-2.5)      => -1
sign(2.5)       => 1
sign(0)         => 1
```

abs(x) The absolute value of x: abs(-5) = abs(5) = 5

sin(x) Sinus of x (in radians).

cos(x) Cosinus of x (in radians).

tan(x) Tangens of x (in radians).

asin(x) Arcus sinus of x (in radians).

acos(x) Arcus cosinus of x (in radians).

atan(x) Arcus tangens of x (in radians).

atan2(x) Arcus tangens of x (in radians), squared.

random() A random number between 0 (incl.) and 1 (excl.)

pow(x, y) x to the power of y: x^y

exp(x) 10 to the power of x: 10^x

log(x) Logarithm of x to the base of 10.

log(x, y) Logarithm of x to the base of y.

ln(x) Natural logarithm of x.

Text Functions

trim(string) Removes leading and trailing white-space of string.

lower(string) Converts a string to lower case .

upper(s) Converts a string to upper case.

lpad(string, length, padding)

If string's length is smaller than the given length, the missing space is filled up with the given padding at the start of the string.

rpad(string, length, padding)

If string's length is smaller than the given length, the missing space is filled up with the given padding at the end of the string.

substring(string, start, end)

Extracts a part of the string. Start and end are zero-based.
substr("Hello World!", 0, 5) => "Hello"
substr("Hello World!", 6, 10) => "World"

substr(string, start, length)

Extracts a part of the string. Start is zero-based.
substring("Hello World!", 0, 5) => "Hello"
substring("Hello World!", 6, 5) => "World"

contains(string, match)

Checks if string contains the given match string by exact comparison.
contains("Hello World!", "World") => true
contains("Hello World!", "world") => false

index(string, match)

Finds the start index of the given match string within string.
index("Hello World!", "World") => 6
index("Hello World!", "world") => -1 (not found)

Date Functions

year(date) Full year of the given date.

month(date) Month of the given date (1 = January, … 12 = December).

day(date) Day of month of the given date (between 1 and 31).

weekday(date)

Weekday of the given date (0 = Sunday, 1 = Monday, … 6 = Saturday)

today() The current date (without time).

now() The current timestamp.

age(date) Number of full year's between now and the given date (e.g. a person's age).

format(date, format)

Formats a date as a string. The format expression is a string which may contain following tokens (example for 9th of August 2015, 01:02:03 am):

Token	Description	Example
YY	two digit year	15
YYYY	four digit year	2015
M	one or two digit month	8
MM	two digit month	08
MMM	abbreviated month name	Aug
MMMM	full month name	August
D	one digit day	9
DD	two digit day	09
Do	day ordinal	9th
h	one digit hour	1
hh	tow digit hour	01
m	one digit minute	2
mm	two digit minute	02
s	one digit second	3
ss	two digit second	03
a	am or pm	am

yearmonth(date)

Year and month of a date as a string, e.g. "2015/08". Useful for grouping records per month.

yearquater(date)

Year and quarter of a date as a string, e.g. "2015/03". Useful for grouping records per quarter.

yearweek(date)

Year and week of year of a date as a string, e.g. "2015/32". Useful for grouping records per calendar week.

week(date) Calendar week of a date as a number.

start(appointment)

Start timestamp of an appointment.

end(appointment)

End timestamp of an appointment.

duration(appointment)

Duration of an appointment in milliseconds.

days(start, end)

Number of days between two dates.

workdays(start, end)

Number of working days between two dates. This function does consider any Monday to Friday to be working days, it does not respect holidays.