QLingo Language

QLingo is a scripting language developed by XMPie, to define expressions that compute ADOR values. QLingo language includes the following types of expressions:

Literal Constants

QLingo supports constant literal values of the following types:

- String Literals
- Number Literals
- Date Literals
- Boolean and Null Constants

String Literals

String Literals are used to write strings of text.

Syntax:

A string literal is enclosed in double or single quotes. A string may include both types of quotes. If you have both types of quotes, you need to escape the enclosing quote by preceding it with a backslash.

The actual backslash is ignored, as shown below:

`\' is '`

`\" is "`

`\t is tab`

`\n` or `\r` are used as line feeds (media dependent). For example, for html the line feed is <br>, therefore \n and \r will not take effect. This holds true unless the rule is defined for html production.

`\\ is \`

`\b is b, and so on`

Examples:

The following examples show the escape sequence in the strings:

`"Joan's Car"`—Joan's Car

`'Joan\'s Car'`—Joan's Car

`'\Joan'`—Joan

`"Joan\nSmith"`—Joan
 Smith

"Joan\tSmith"—`Joan    Smith`

`"text starts with \"Hello …\""`—text starts with "Hello ..."

`'text starts with "Hello …"'`—text starts with "Hello ..."

`"\\Joan\\"` - \Joan\

Number Literals

Number literals are used to write text that includes numbers.

Syntax:

A number literal can be optionally preceded by a positive or negative sign and can include a decimal point.

Examples:

```
123

+201

-2

1.34

-1898.22214
```

Date Literals

Date literals are used to include dates according to specified formats.

Syntax:

A date literal is enclosed in pound signs (#). The currently supported date formats are #dd/mm/yyyy# and #dd/mm/yy#. You can use QLingo's 'FormatDate' function to format dates in a different way.

The use of the following delimiters is also supported: `'\'`,`'/'`, `'-'` and `'.'`.

Single digit numbers must have a preceding zero.

Examples:

```
#02/03/2001#
```

```
#14\06\92#
```

```
#02-03-2001#
```

```
#14.06.92#
```

Boolean and Null Constants

Boolean constants can be used in logical expressions. There are two built-in logical constants: `TRUE` and `FALSE`.

`Null` is a built-in constant that represents the null value (the database concept of an undefined value).

`Null` also represents a non-existent value. For example, if a query returns one row, and references a value in a second row that does not exist, the result is `null`. Additionally, in the case where you have an `If` statement that does not contain `‘else'`, if the condition is not met, the expression value is `Null`.

Note that the `Null` value as an end result of an ADOR is treated as an empty string.

Syntax:

Null constants can be used in comparisons with the Equality (==) or Non-Equality (!=) tests.

Examples:

```
[MyRecordset][0].[Name] != NULL
```

```
@{startDate} == null
```

Arithmetic Expressions

QLingo currently supports several arithmetic operations with the conventional order of precedence.

The operations, in order of priority, are `mod`, `div`, `*`, `/`, `+`, `-`, and `&`.

Syntax:

Arithmetic Expression	**Description**
*, /, -, and + signs	Perform arithmetic calculations
+ sign	Concatenates strings. "a" + "b"
& sign	Concatenates any type, while the result is a string. "a" & 1 (= "a1"), 1 & 2 (= "12")

Mod or %sign	Perform modulo operations, using either “mod” or “%”, for example: ● 5 mod 2 = 1 ● 5 % 2 = 1 Any fractional part of a number is lost. 5.1 mod 2 = 1
Div	Concludes the round division answer. 5 div 2 = 2. Any fractional part of a number is lost. 5.1 div 2 = 2

Examples:

Regular arithmetic:

```
5 * 3 =15

5 / 2 =2.5

5 Div 2  =  2

5 Mod 2  =  1
```

Concatenation:

```
"Joan" & " " & "Smith"  =  Joan Smith

56 & 4  =564

#31/01/1973# & " Date"  =  31/01/1973 Date
```

The `'+'` sign between strings as concatenation:

```
"Joan" + " " + "Smith"  -  =  Joan Smith
```

The `'+'` and `'-'` signs between date and number to add/subtract days to/from a date:

```
#01/01/2002# + 2  =  03/01/2002
#03/01/2002# - 2  =  01/01/2002
```

The `'-'` sign between dates in order to get the difference in number of days:

```
#03/01/2002# - #01/01/2002#  =  2
```

Comparisons

You can use comparisons in logical expressions.

Syntax:

Comparisons use the general syntax of *compOp (comparison operator) expression*.

The compOp can be one of: `<, <=, ==,=,<>,!=, >=, or >`.

`=,==` are the same

`!=,<>` are the same

Examples:

```
@{fName} == "James"

SELECT age FROM customers WHERE id = ?; > @{ageThreshold}

@{SALARY} <= 20000

@{birthDate} > #01/01/2000#
```

This example is not applicable in uCreate Print.

The comparisons have lower precedence than the arithmetic so writing 5 + 3 > 2 + 9 is equivalent to writing (5+3) > (2+9)

To make sure that both sides of the comparison expression belong the same data type (string, number or date), it is recommended to apply AsString, AsNumber or AsDate functions to each side of the expression.

Logical Expressions

Logical expressions allow you to define a number of conditions and to make comparisons.

Syntax:

A logical expression is a logical constant, a comparison, or a logical expression made by using the following logical operators:

- `NOT`
- `!`

- `OR`
- `AND`

`Not` and `!` are the same.

The logical expressions have a lower precedence than comparisons, so writing `5 > 10` and `3 == 5` is the same as `(5>10)` and `(3==5)`

The logical expressions are evaluated in order. Therefore, when writing `if((@{a} != NULL) AND (@{a} == @{b}))`, the first expression is evaluated first, and only if it evaluates to "`True`" the second expression is evaluated as well.

The logical `'OR'` works the same as the logical `'AND'` – except that in this case, after an expression that is evaluated to `True` is found, the check is stopped, and the return value is `True`.

Examples:

```
(@{cost} > 100000) AND @{fName} == "James"
```

```
@{fName} == "James" OR @{fName} = "John"
```

```
NOT(SELECT age FROM customers WHERE id = ?; > @{ageThreshold})
```

Control Statements

If/Else Statement

If statements are used to define conditional options.

Syntax:

If statements follow syntax similar to C and C++:

```
If (condition)

{

   Expression in case of true condition

}

Else

{

Expression in case of false condition

}
```

The condition is an expression that is regarded as a Boolean value to be tested by the If statement. This can be a logical expression, a comparison, or a Boolean constant; any other expression will be converted to Boolean and be tested
(see AsBoolean Function of the Conversion Functions section).

The `Else` part is optional. If the predicate is evaluated to `False` and there is no `Else` statement, the value of the `If` statement is `Null`.

Example:

```
if(@{age} > 60)

"senior"

else if(@{age} > 20)

"adult"

else

"young"
```

Switch Statements

A `Switch` statement is a simplified way to write a multi-choice `'If'` statement.

Syntax:

The `Switch` statement uses the following syntax:

```
Switch (expression)

{

 Case literal1:

   Expression1

 Case literal2:

   Expression2

 Default:

   DefExpression

}
```

The `Default` case is optional. If no case matches the expression and there is no default case, the value of the `Switch` statement is `Null`.

Example:

```
Switch (@{category})

{

   Case "PLATINUM":

      250000

   Case "GOLD":
        70000
    Case "SILVER":
        30000
    Default:
        10000
}
```

Functions

Possible functions include:

- Numeric Functions
- Date Functions
- Barcode Function for Print Media
- String Functions
- Conversion Functions
- GetEnv Functions
- uImage Functions
- Miscellaneous Operators and Functions
- Asset Functions
- Circle Functions

- User View Functions
- Custom User-Defined Functions

Recipient Information Field Reference

You can reference the recipient information fields in expressions and SQL queries, since they are self-generated variables.

Syntax:

`|->[Field]` returns the selected Recipient Information field that was defined in the Recipient Information Schema section of the Plan.

You can also write the name without the square brackets, provided that the name starts with a letter and contains only alphanumeric characters.

You can use the character "?" to reference a special variable, which is the primary field (XMPieRecipientKey) in the Recipient Information Schema.

Example:

```
SELECT AccountId FROM Accounts WHERE CustomerId = ?;
```

Database-Related Functions

Database-related functions include:

- Variable Reference

Variable Reference

You can use Variables in QLingo expressions and SQL queries.

Syntax:

A Variable reference is defined by either the Variable name (if it starts with a letter and consists of only letters and numerals) or the ‘at’ sign ("@") followed by the Variable name in curly brackets. The second syntax option is mandatory in SQL queries that are part of the QLingo program.

Example:

@{first_name} will return the recipient’s ‘first name’. This will occur only if you defined a Variable called first_name to return the first name value.

Custom User-Defined Functions

While uPlan has many common built-in functions already, it is also possible to create custom functions using either QLingo or JavaScript to further extend uPlan's capabilities.

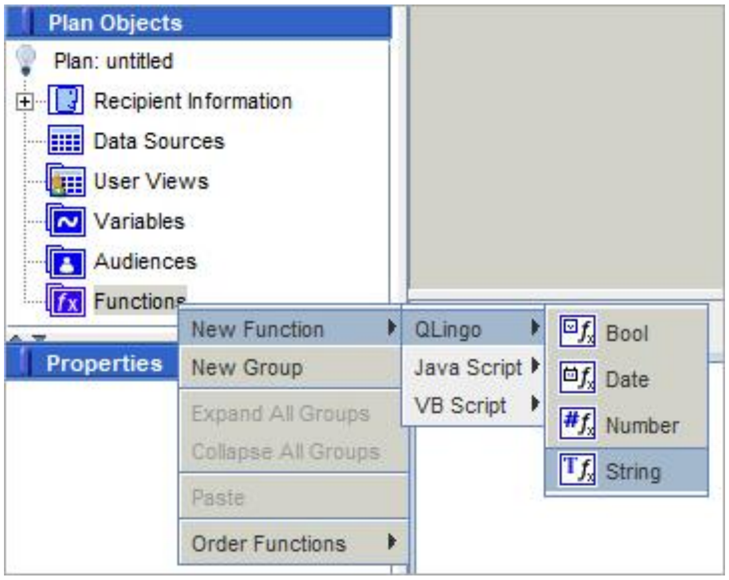
It is not possible to create custom user-defined functions in uCreate Print. However, you can link uCreate Print to a plan file that contains custom functions.

- Creating Custom Functions with QLingo
- Creating Custom Functions with JavaScript
- Using the new Custom Function in an ADOR or Variable
- Accessing other uPlan elements with custom JavaScript functions
    - Accessing Recipient record field values
    - Accessing Variable values
    - Accessing UserView record values
    - Accessing uPlan functions

Creating Custom Functions with QLingo

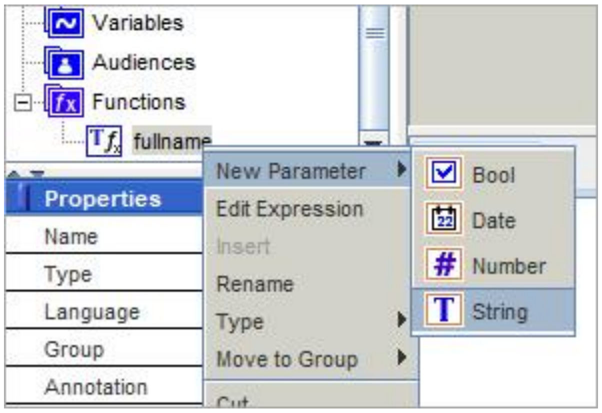
This example will create a function called "fullname" that will take firstname and lastname parameters and concatenate (join) them with a space.

1. Create a new QLingo function and select the type of data that the function will return to uPlan.



2. Name the new function.

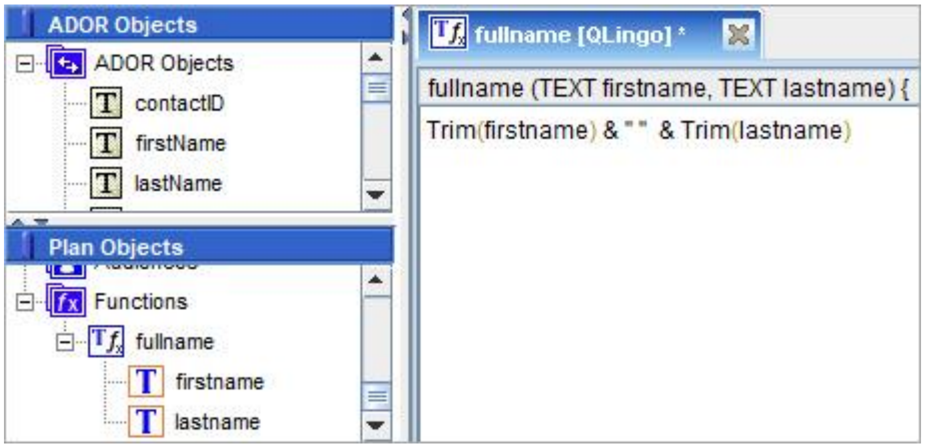
3. Right-click on the new function and add parameters for the values that you need to push into the function.



4. Select the appropriate data type for the incoming data and name the new parameters.

5. Repeat for any additional parameters needed by your function. In this example, create parameters for **firstname** (string) and **lastname** (string).

6. Double click the function name, or right-click and select Edit expression.



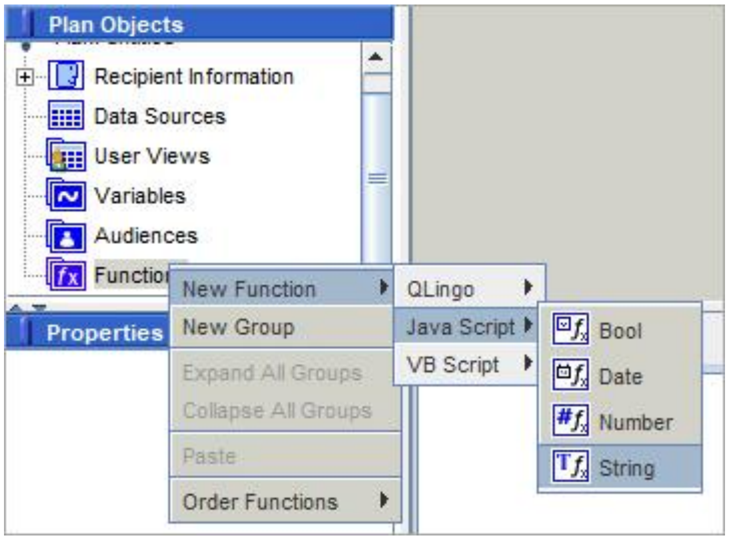
```
fullname (TEXT firstname, TEXT lastname) {
Trim(firstname) & " " & Trim(lastname)
```

7. Write your QLingo expression using normal QLingo syntax. In this example, use the **Trim()** function to remove any white space from the **firstname** and **lastname** parameters, and use the **&** (ampersand) to concatenate the **firstname**, **space** and **lastname**.

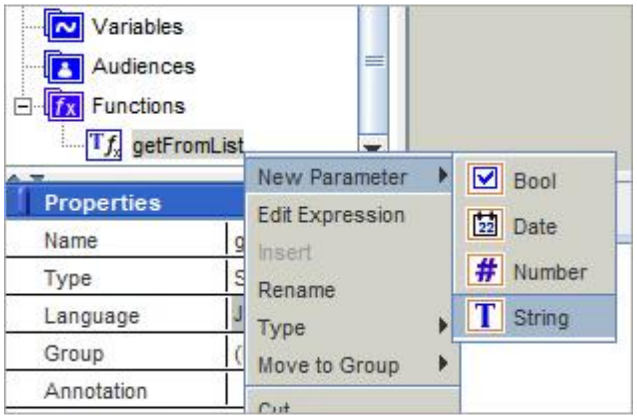
Creating Custom Functions with JavaScript

This example will create a function called "getFromList" that will receive three parameters: a list, a delimiter and an index number. And will return the value that is in the index position of the list.

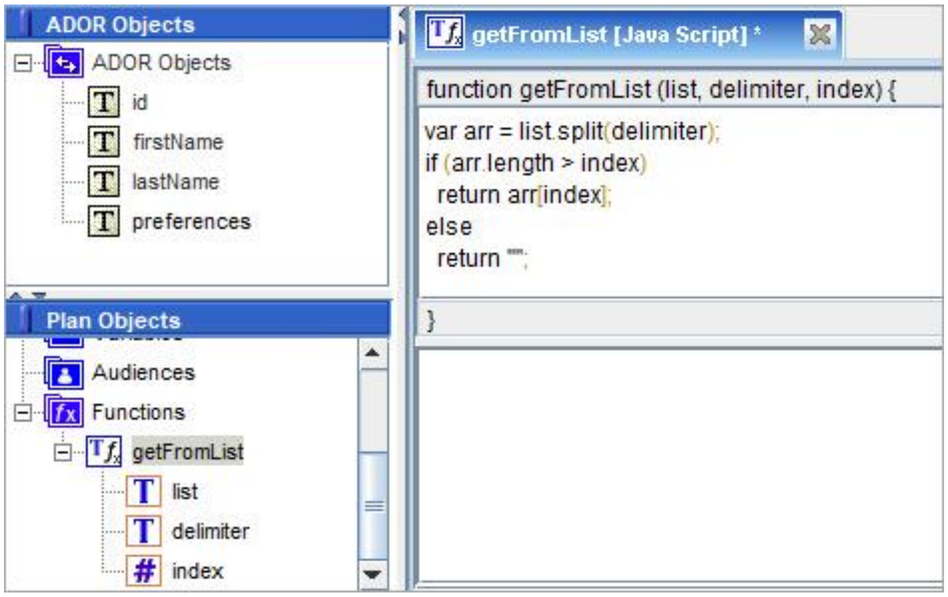
1. Create a new JavaScript function and select the type of data that the function will return to uPlan.



2. Name the new function.
3. Right-click on the new function and add parameters for the values that you need to push into the function.



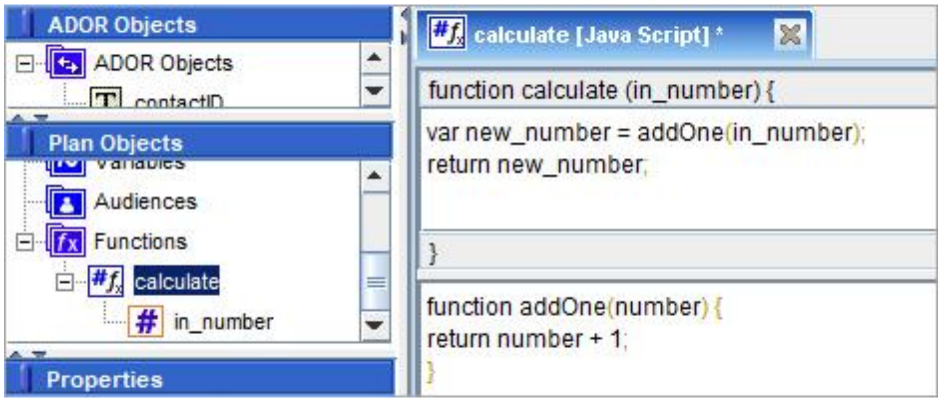
4. Select the appropriate data type for the incoming data and name the new parameters.
5. Repeat for any additional parameters needed by your function. In this example, create parameters for: **list** (string), **delimiter** (string) and **index** (number).
6. Double click the function name, or right-click and select Edit expression.



```
function getFromList (list, delimiter, index) {
var arr = list.split(delimiter);
if (arr.length > index)
  return arr[index];
else
  return "";
}
```

7. Write your JavaScript expression using normal JavaScript syntax. In this example, use the JavaScript **Split()** function to convert the list to an array. You can then simply return the value from the correct index of the array. (Remember that JavaScript is zero-based, so the first element of the array will be index 0.)

The JavaScript expression editor consists of two editable areas:



```
function calculate (in_number) {
var new_number = addOne(in_number);
return new_number;
}
```

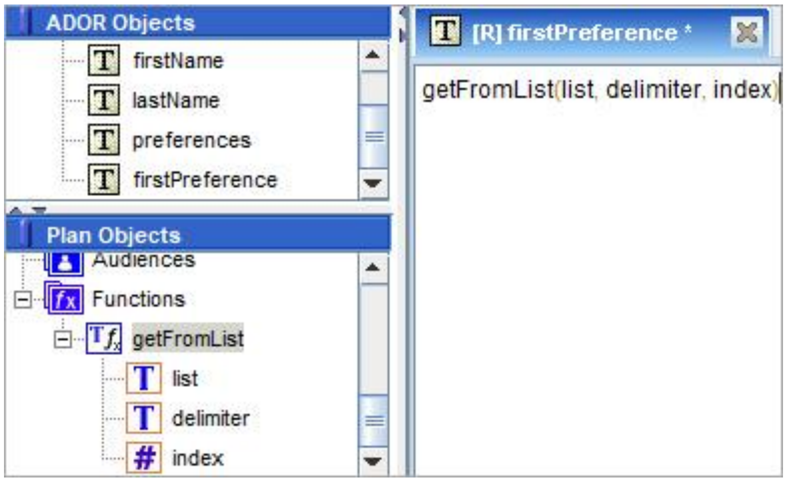
```
function addOne(number) {
return number + 1;
}
```

The upper area is for your main function that you can see is defined automatically. The lower area allows you to define any additional functions that will be called from your main function.

Using the new Custom Function in an ADOR or Variable

Now that you have created a custom function, you can use it like any other uPlan function, by dragging and dropping the function (or typing the function name) into one or more ADORs or variables and setting the required parameter values.

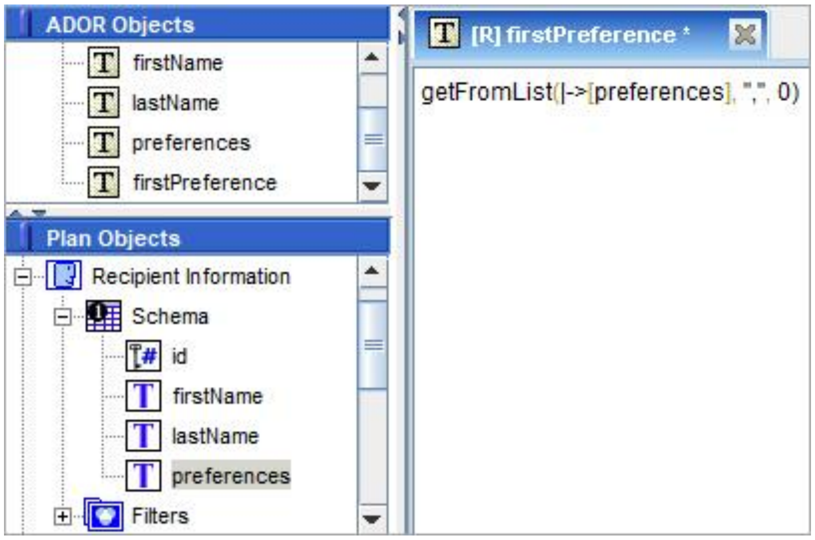
1. Open your ADOR or variable's Expression. In this case, I have created a new ADOR called "firstPreference". From the Functions list, drag the "getFromList" function into the editor.



```
getFromList(list, delimiter, index)
```

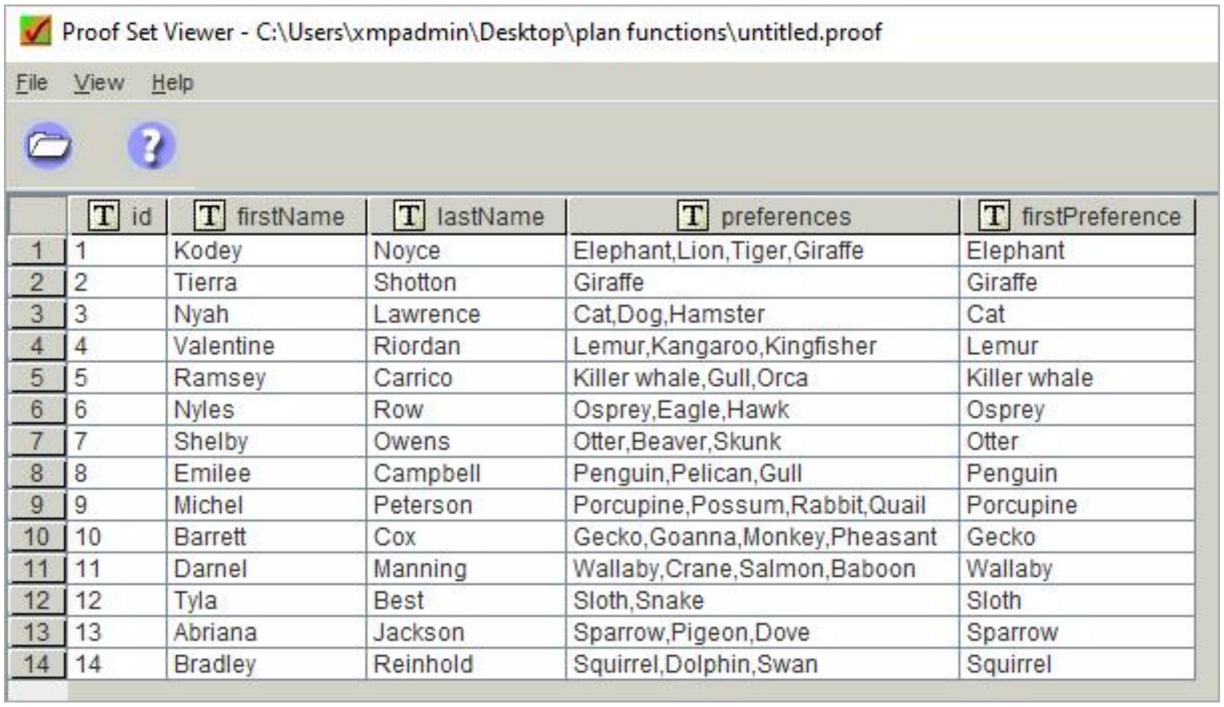
2. Replace the parameter names with the values you want to pass to the function. In this case:

   - the list is coming from the Recipient Information Schema "preferences" field,
   - the delimiter is "," (since the preferences list is comma delimited), and
   - the index is 0 (zero) to get the first item from the list.



```
getFromList(|->[preferences], ",", 0)
```

3. Generate a proofset to confirm your function is working as expected.



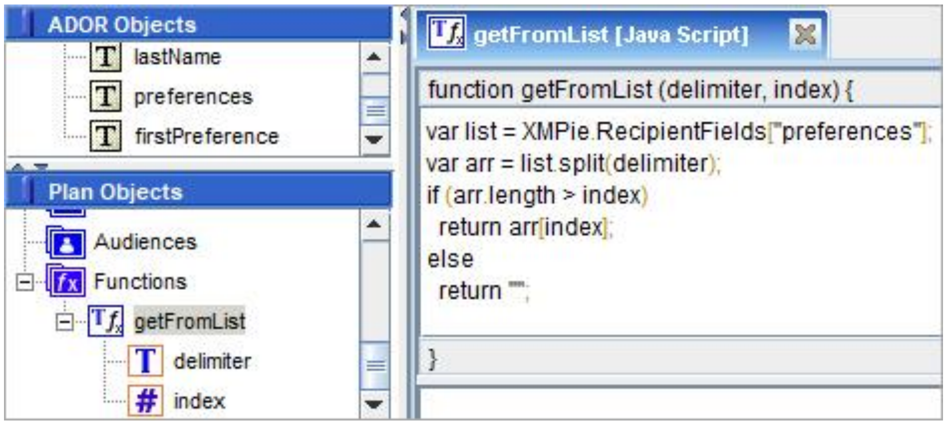
Proof Set Viewer - C:\Users\xmpadmin\Desktop\plan functions\untitled.proof

	id	firstName	lastName	preferences	firstPreference
1	1	Kodey	Noyce	Elephant,Lion,Tiger,Giraffe	Elephant
2	2	Tierra	Shotton	Giraffe	Giraffe
3	3	Nyah	Lawrence	Cat,Dog,Hamster	Cat
4	4	Valentine	Riordan	Lemur,Kangaroo,Kingfisher	Lemur
5	5	Ramsey	Carrico	Killer whale,Gull,Orca	Killer whale
6	6	Nyles	Row	Osprey,Eagle,Hawk	Osprey
7	7	Shelby	Owens	Otter,Beaver,Skunk	Otter
8	8	Emilee	Campbell	Penguin,Pelican,Gull	Penguin
9	9	Michel	Peterson	Porcupine,Possum,Rabbit,Quail	Porcupine
10	10	Barrett	Cox	Gecko,Goanna,Monkey,Pheasant	Gecko
11	11	Darnel	Manning	Wallaby,Crane,Salmon,Baboon	Wallaby
12	12	Tyla	Best	Sloth,Snake	Sloth
13	13	Abriana	Jackson	Sparrow,Pigeon,Dove	Sparrow
14	14	Bradley	Reinhold	Squirrel,Dolphin,Swan	Squirrel

Accessing other uPlan elements with custom JavaScript functions

Custom JavaScript functions can also directly access the value of other uPlan elements, for example values of a Recipient record, Variables, UserViews and Functions.

In the previous example we passed the preferences list to the function via a parameter. The function can also directly access the recipient list values, so the function could be rewritten to use just two parameters and to get the list directly from the recipient list like this:



```
function getFromList (delimiter, index) {
var list = XMPie.RecipientFields["preferences"];
var arr = list.split(delimiter);
if (arr.length > index)
  return arr[index];
else
  return "";
}
```

Accessing Recipient record field values

To access values from the Recipient record, use:

```
XMPie.RecipientFields["name of field"]
```

For example:

```
XMPie.RecipientFields["firstname"]
```

Accessing Variable values

To access uPlan Variable values, use:

```
XMPie.Variables["name of variable"]
```

For example:

```
XMPie.Variables["age_var"]
```

Accessing UserView record values

To access UserView field values, use:

```
XMPie.UserViews["userview name"][row number]["name of field"]
```

For example:

```
    XMPie.UserViews["Orders_UV"][2]["orderID"]
```

Accessing uPlan functions

Custom JavaScript functions can also directly access uPlan's built-in functions, or other custom functions that have been created in the plan file.

To access a uPlan function, use:

```
    XMPie.Functions."function name"("value")
```

For example:

```
    XMPie.Functions.UCase("This String")
```

Returns:

```
"THIS STRING"
```

Another example using a uPlan function together with a recipient record:

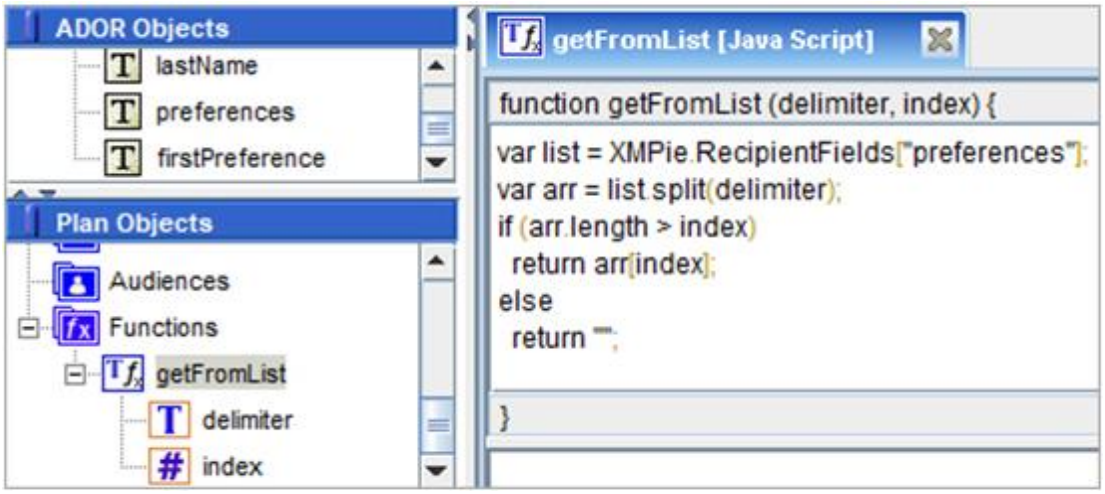
```
    XMPie.Functions.UCase(XMPie.RecipientFields["firstname"])
```

Would return, in uppercase, the value from the "firstname" field of database for the current recipient record.

Many of the uPlan built-in functions have similar functions in JavaScript. For example uPlan's UCase(value) function is the same as JavaScript's value.toUpperCase() function. However, many others like GetEnv() are unique to uPlan and offer functionality that you might want to use in your custom JavaScript functions.

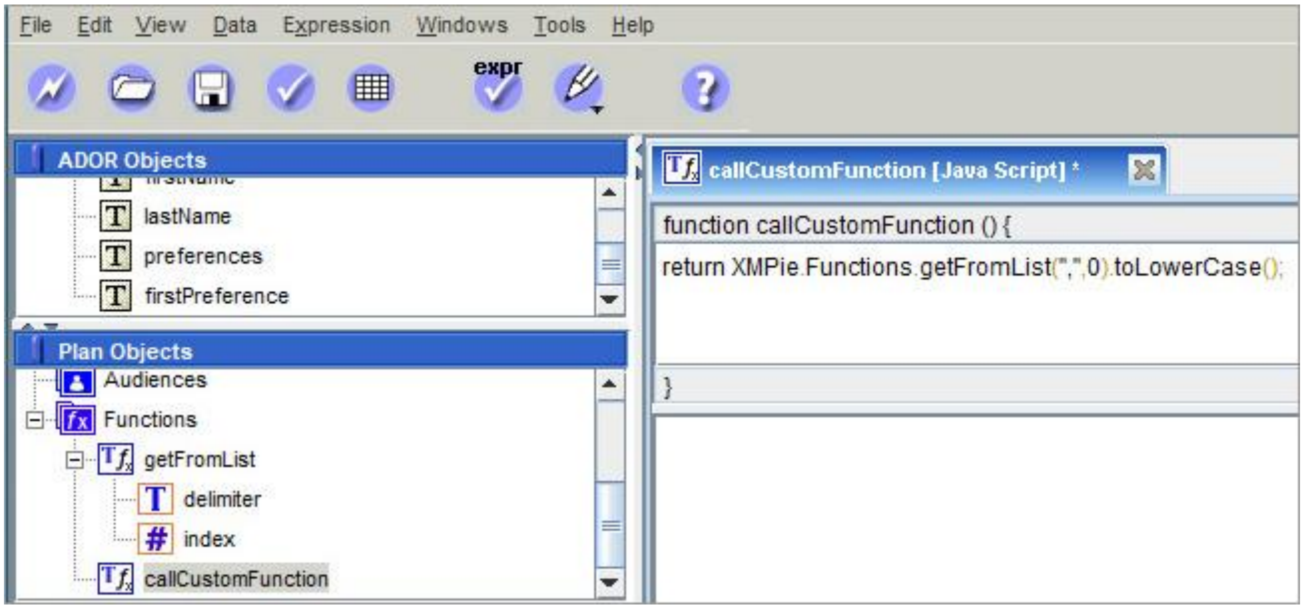
In addition to accessing the built-in uPlan functions, it is also possible for JavaScript functions to access other custom functions in the same way.

For example, if we use the getFromList() function example we created earlier:



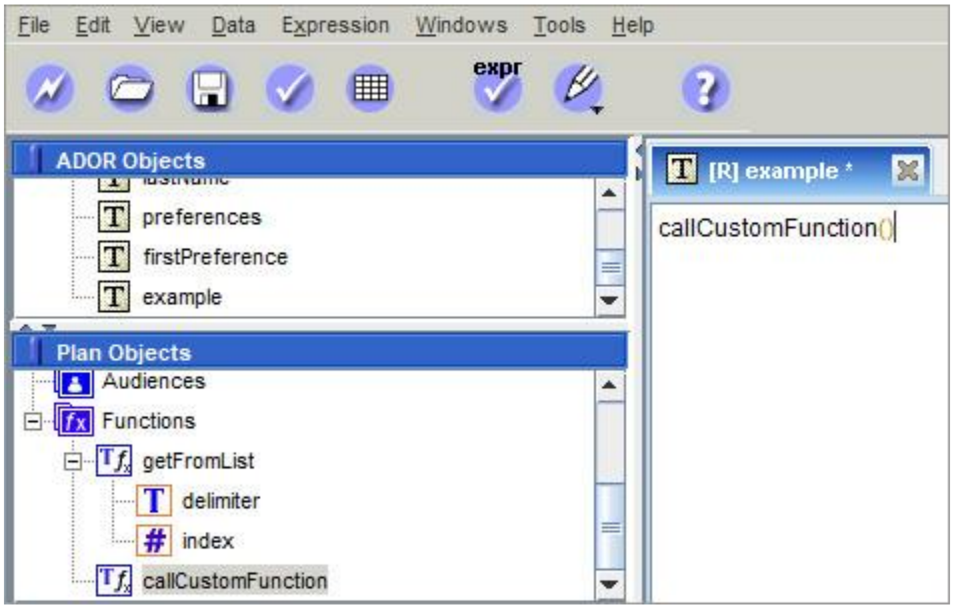
```
function getFromList (delimiter, index) {
var list = XMPie.RecipientFields["preferences"];
var arr = list.split(delimiter);
if (arr.length > index)
  return arr[index];
else
  return "";
}
```

Another JavaScript function can call the getFromList() function and do something else with the value – converting it to lowercase in this example:



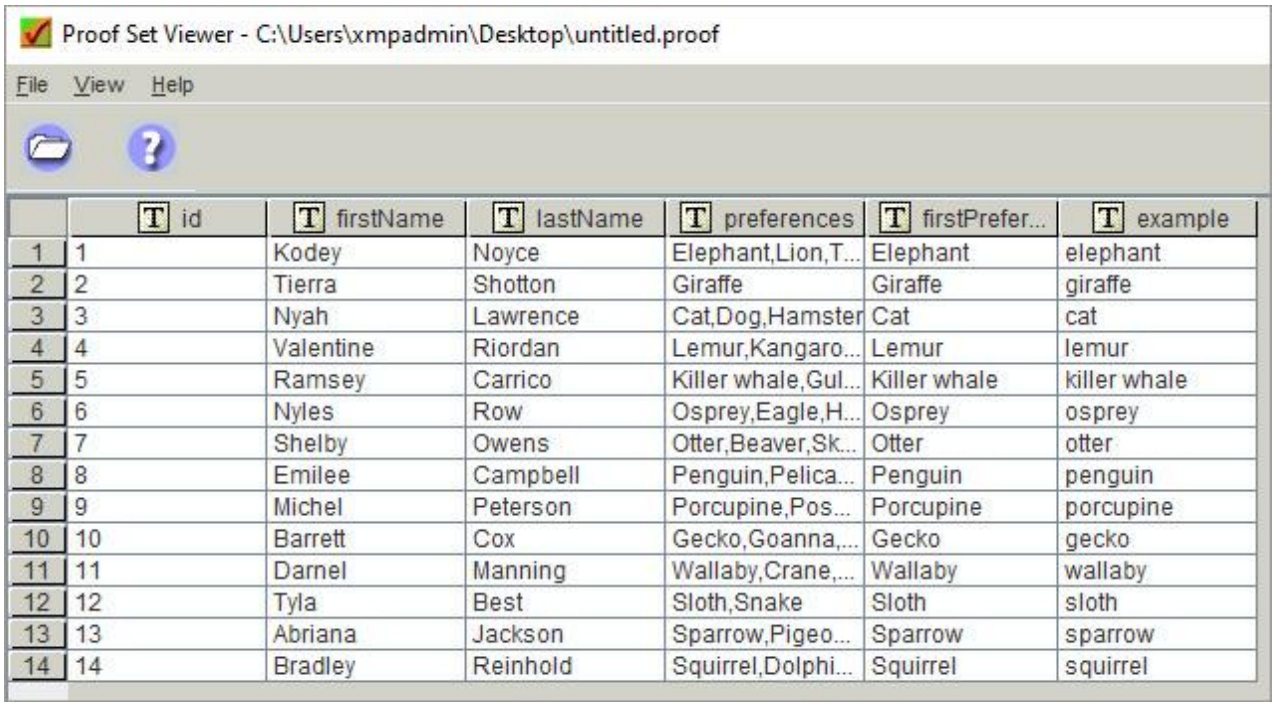
```
function callCustomFunction () {
return XMPie.Functions.getFromList(",",0).toLowerCase();
}
```

This new function can be used in an ADOR expression:



```
callCustomFunction()
```

And the result can be seen in a proofset:



	id	firstName	lastName	preferences	firstPrefer...	example
1	1	Kodey	Noyce	Elephant,Lion,T...	Elephant	elephant
2	2	Tierra	Shotton	Giraffe	Giraffe	giraffe
3	3	Nyah	Lawrence	Cat,Dog,Hamster	Cat	cat
4	4	Valentine	Riordan	Lemur,Kangaro...	Lemur	lemur
5	5	Ramsey	Carrico	Killer whale,Gul...	Killer whale	killer whale
6	6	Nyles	Row	Osprey,Eagle,H...	Osprey	osprey
7	7	Shelby	Owens	Otter,Beaver,Sk...	Otter	otter
8	8	Emilee	Campbell	Penguin,Pelica...	Penguin	penguin
9	9	Michel	Peterson	Porcupine,Pos...	Porcupine	porcupine
10	10	Barrett	Cox	Gecko,Goanna,...	Gecko	gecko
11	11	Darnel	Manning	Wallaby,Crane,...	Wallaby	wallaby
12	12	Tyla	Best	Sloth,Snake	Sloth	sloth
13	13	Abriana	Jackson	Sparrow,Pigeo...	Sparrow	sparrow
14	14	Bradley	Reinhold	Squirrel,Dolphi...	Squirrel	squirrel

SQL Queries

You can use SQL queries to retrieve data from the database or multiple databases.

SQL queries are either:

- **Select queries**, i.e. `SELECT field names FROM tables WHERE condition;`
- **Execute stored procedures**, i.e. `EXECUTE stored_procedure_name parameter values;`

XMPie is indifferent to how you implement the stored procedure. However, it does require that the end result of the stored procedure is table data. Note that any other return values or output parameters are ignored.

Syntax:

In addition to regular SQL syntax, you can include references to Plan variables in the query (making it a parameterized query).

When working with a Plan that references multiple Data Sources, you must specify the Data Source name to which the query refers, before the query. This is done using the following syntax:

For Select queries:

```
@{schema name}: SELECT field names FROM tables WHERE condition;
```

For Stored procedures :

```
@{schema name}: EXECUTE stored_procedure_name parameter values;
```

An SQL query that is part of a larger expression must end with a semi-colon (;).

Examples:

```
If(@{Movie Suggestion ID} != NULL)

{

SELECT @{Type} + [MovieName] FROM movies WHERE  MovieId = @{Movie
Suggestion ID};

}

else

{

   @{Type} + "Gladiator"

}


SELECT firstName FROM customers WHERE id = ?;

SELECT firstName FROM customers WHERE id = |->[id];

SELECT @{title} + firstName FROM customers WHERE id = ?;

EXECUTE sp_name @param = |->[id];
```

When writing SQL queries referring to Excel and CSV files-based Data Sources, the native query syntax (for example, [Sheet1$] for Excel) will not work. In this case, you must use the QLingo generic query syntax to refer to table and field names. See Table Syntax and Field Syntax topics for detailed instructions. For other Data Source types you can use either the native syntax or the QLingo generic syntax.

User View Reference

A User View reference can refer to a complete user view, a complete row, or a single field value.

Syntax:

The reference is made of three parts, each enclosed in square brackets:

`[user_view_name][row_index].[field_name]`

The row_index and field name are optional. The field name must be preceded by a period (.).

The first row in the user view is referred to by row_index value of zero.

If the row_index is omitted, and the field_name exists, the row_index is assumed to be zero. If the field_name is omitted, a complete row is referenced.

If both the field_name and row_index are omitted, the whole record set is referenced.

Note that QLingo is not case sensitive so the User View Reference can be written in a different case than the actual user view name (or field name).

Example:

A list of possible user view reference syntaxes are as follows:

`[UserView_1][2].[lastName]`– fetches the data of column "lastName" in row 3

`[CustomerUV].[Address]` – fetches the Address value of the first row

`[PurchasesUV][1]` – fetches the entire second row (first row is 0)

`[ProductsUV]` – fetches the entire user view

An example of using an entire user view reference:

```
user view 1: select MovieName, MoviePrice from ActionMovies where
MovieId = @{MovieSuggestion}

user view 2: select MovieName, MoviePrice from DramaMovies where MovieId
= @{MovieSuggestion}

user view 3:

switch(@{Customer Movie Type Liking})

{

   case "Action":

      [user view 1]

   case "Drama":

      [user view 2]

}
```

Row Subscript

The Row Subscript enables you to use an expression to represent the row_index of a user view. The Row Subscript should be an expression that evaluates to a number, and can be used instead of the row_index constant.

Example:

Say, we would like to create a QLingo function that concatenates all rows of a certain field in a user view.

In this example, we have the following table, which is in a user view UV:

Child Name	**Child Age**
John	8
Jin	12

To list of all the children's names, we create a function of the form:

```
GetChildrenList(Number i)

{

if(i >= RowCount([UV]))

""

else

{

if(i = 0)

[UV][i].[Child Name] & GetChildrenList(i+1)

else

", " & [UV][i].[Child Name] & GetChildrenList(i+1)

}

}
```

In the appropriate expression, we can call the function: GetChildrenList(0)

Assignment

The Assignment operator is available for Write expressions only. It allows you to use calculated ADOR value to update one or more database entries.

The Assignment operator can be used by signing a constant value to an ADOR, thus changing the ADOR value to the constant value.

Syntax:

```
Recipient Field Ref := QLingo Expression
```

Examples:

In the following example, the literal value ‘senior’ is written to the recipient field (database field) ‘status’ if the ‘age’ Variable value is over 40. The literal value ‘adult’ is written to the recipient field (database field) value ‘status’ if the ‘age’ Variable value is under 40

```
|->[status] := if(@{age} > 40)

         "senior"

         else

         "adult"
```

Another way for using the Assignment operator is by using the key word ‘value’ as a placeholder for input values that will update the Write ADOR through the ICP channel.

Syntax:

```
Recipient Field Ref := value
```

Examples:

```
|->[first] := value
```

In this example, the control over the update mechanism is in the hands of the application or web site programmer, who wishes to update the database through the ICP channel. For example, a web site developer can create web forms that will be used to update an ADOR value, that, in turn, will be used to update a database entry (or entries), as a part of a customer feedback mechanism.

Notes on the Evaluation of QLingo Expressions

Any QLingo expression that does not depend on individual recipients is executed (evaluated) only once during production.

For example, if an ADOR retrieves data by executing an SQL query that does not have any parameters, and hence does not depend on the individual recipient, the query is executed a single time for the entire production run. This optimization prevents uProduce from needlessly repeating the same query and evaluating expressions when the values of ADORs for each individual recipient are calculated.

It is important that you understand how this optimization affects queries that use randomized values or are time-dependent. You will need to ensure that randomized values are recalculated for each recipient and not once for the entire production run.

Take, for example, the case where a Campaign offers a random book name to each recipient:

```
SELECT TOP 1 books.name FROM books order by RAND(books.id)
```

The above query is not dependent on any recipient information whatsoever and will therefore not be evaluated separately for each recipient. As a result, all recipients will see the same book name in their Document instances. You can easily work around this limitation by making your query dependent on the recipient ID:

```
SELECT TOP 1 books.name FROM books WHERE |->[ID] <> -1 order by rand
(books.id)
```

The above query retrieves a random book name for each recipient whose ID is not equal to
(-1). Assuming there are no recipients with an ID equal to (-1), this query is executed for every recipient.

Examples of Using QLingo

In the example below, we use a Plan that has a user view named recipInfo, with the following fields:

- `FNAME` (for example, John, Stacy, Suzi)
- `INITIAL` (for example, S, G, [empty], Jesse)
- `LNAME` (for example, Smith, Brown, Green)
- `ADDR1` (for example, 253 3rd Avenue, 45O Apple St.)
- `ADDR2` (for example, Bellview, [empty], Riverdale)
- `ZIPCODE` (for example, 12345, 12345-8765, 98776-2345-63)

- `CITY` (for example, London, New York)

Middle Initials:

This example defines an ADOR called FULLNAME, which includes middle initials.

```
if ([recipInfo][0].[INITIAL] == "")

// Concatenate first name, space, last name

[recipInfo][0].[FNAME] & " " & [recipInfo][0].[LNAME]

Else

// Concatenate first name, space, FIRST CHARACTER OF INITIAL, a period
and a space, last name

[recipInfo][0].[FNAME] &" " & SubString([recipInfo][0].[INITIAL], 0, 1)
& ". " & [recipInfo][0].[LNAME]
```

Omitting ADDR2:

This example defines an ADOR named ADDR1andADDR2 (One ADOR for both).

```
if ([recipInfo][0].ADDR2 == "")

   [recipInfo][0].ADDR1

else

[recipInfo][0].ADDR1 & "\n" & [recipInfo][0].ADDR2
```

Zip code number to barcode (for a font with an "s" delimiter, such as USPSTTF):

This example defines an ADOR named BarcodedZip that uses a QLingo extension (good for 5, 5-4, 5-4-2 zip codes):

Call USPSZIP.USPSZIP([recipInfo][0].[ZIPCODE], 's')

Season calculation:

This example calculates the season (Northern Hemisphere) related to the current month.

Define a Variable called "MyMonth" (to calculate the month):

```
AsNumber(Format(|->[MyDate], "mm"))
```

Then define an ADOR called "Season":

```
If (MyMonth < 3 OR MyMonth = 12)
 "Winter"
Else If (MyMonth < 6)
 "Spring"
Else If (MyMonth < 9)
 "Summer"
Else
 "Autumn"ooc
```

Commonly Used Expressions

The Rule Editor's lists allow you to select the most commonly-used expressions:

- Control Statements
- Functions - including the following:

- Numeric Functions
- Date Functions
- Barcode Function for Print Media
- String Functions
- Conversion Functions
- GetEnv Functions
- uImage Functions
- Miscellaneous Operators and Functions

Control Statements

If/Else Statement

If statements are used to define conditional options.

Syntax:

If statements follow syntax similar to C and C++:

```
If (condition)
{
   Expression in case of true condition
}
```

```
Else

{

Expression in case of false condition

}
```

The condition is an expression that is regarded as a Boolean value to be tested by the If statement. This can be a logical expression, a comparison, or a Boolean constant; any other expression will be converted to Boolean and be tested
(see AsBoolean Function of the Conversion Functions section).

The `Else` part is optional. If the predicate is evaluated to `False` and there is no `Else` statement, the value of the `If` statement is `Null`.

Example:

```
if(@{age} > 60)

"senior"

else if(@{age} > 20)

"adult"

else

"young"
```

Switch Statements

A `Switch` statement is a simplified way to write a multi-choice `'If'` statement.

Syntax:

The `Switch` statement uses the following syntax:

```
Switch (expression)

{

 Case literal1:

   Expression1

 Case literal2:

   Expression2

 Default:

   DefExpression

}
```

The `Default` case is optional. If no case matches the expression and there is no default case, the value of the `Switch` statement is `Null`.

Example:

```
Switch (@{category})

{

   Case "PLATINUM":

      250000

   Case "GOLD":

      70000

   Case "SILVER":

      30000

   Default:

      10000

}
```

Numeric Functions

Numeric functions include:

- ABS Function
- Floor Function
- Ceil Function
- FormatNumber Function
- Rand Function
- Round Function

Numeric functions in QLingo can handle numbers with a maximum of 14 digits. Using numeric functions on numbers with 15 or more digits may cause incorrect results.

ABS Function

The `ABS` function gets the absolute value of the number expression.

Syntax:

Example:

```
ABS(-5) = 5
```

Floor Function

The `Floor` function floors the number expression.

Syntax:

```
Floor(number expression)
```

Example:

```
Floor(3.78) = 3
```

Ceil Function

The `Ceil` function ceils the number expression.

Syntax:

```
Ceil(number expression)
```

Example:

```
Ceil(3.12) = 4
```

FormatNumber Function

The `FormatNumber` function formats the number in `expression1` according to the format specification in `expression2`.

`expression2` represents the input string using three special characters: “`#`”, “`0`”and “`.`” . You can also use other characters, such as the dollar sign ($). Any character other than "`#`", "`0`"and "`.`" remains as it was in the format specification.

The dot divides the number into to parts: integral and fractional. The digits in each part are ordered as follows:

- Integral part (to the left of the dot): `expression2` adds digits from right to left (that is, from the smallest position to the largest position), depending on the number of "`#`" or "`0`" placeholders.

- Fractional part (to the right of the dot): `expression2` adds digits from left to right (that is, from the large position to the smallest position), depending on the number of "`#`" or "`0`" placeholders.

If `expression2` does not include a dot, the input number (`expression1`) is treated as an integer and the fractional part is ignored.

The following table shows the FormatNumber Function - special characters used by `expression2`.

Character	Description
#	Number sign. Used as a placeholder for digits. If there are more placeholders than digits, they will be removed.
0	Zero. Used as a placeholder for digits. If there are more placeholders than digits, they will appear as "0".

.	Dot. This character divides the number into two parts: integral (to the left of the dot) and fractional (to the right of the dot).

Syntax:

```
FormatNumber(expression1, expression2)
```

Example:

In the following example, `expression2` defines nine placeholders, divided into groups of three, separated by commas. There are no placeholders for a fraction. The format specification ends with a dollar sign ($), which remains as-is, regardless of the input string. `expression1` is an integer with five digits (`10000`), with no fractional section. In this case, there are enough placeholders for all digits and the result is “`10,000$`”:

```
FormatNumber (10000,"###,###,###$") = 10,000$
```

In the next example, `expression2` defines only two placeholders for the integer and two placeholders for the fraction. `expression1` includes both an integral part (`1234`) and a fractional part (.5). In this case, there are not enough placeholders for all digits in the integral part, only for the first two from the right: 4 and 3. Therefore, the result is “`34.50`”:

```
FormatNumber(1234.5, "00.00")= 34.50
```

Rand Function

The `Rand` function generates a random integer between 0 and the calculated integer value of the expression (not including). If the value of the expression is 1, a floating-point number between 0 and 1 (exclusive), will be returned.

Syntax:

```
Rand(expression)
```

Examples:

```
Rand(5) can return a value of 1
```

```
Rand(1) can return a value of 0.2376
```

Round Function

The `Round` function rounds off the calculated number value of `expression1` as an integer, with `expression2` as the number of precision digits.

Syntax:

```
Round(expression1,expression2)
```

Example:

```
Round(12.344, 2) returns the value of 12.34.
```

Date Functions

Date functions allow you to retrieve information on the date and manipulate the date display.

Date functions include:

- GetDay Function

- GetMonth Function
- GetYear Function
- GetDayOfWeek Function
- GetHour Function
- GetMinute Function
- GetSecond Function
- Age Function
- Now Function
- FormatDate Function

GetDay Function

The `GetDay` function gets the day (1 to 31) of the date expression.

Syntax:

```
GetDay(date expression)
```

Example:

```
GetDay("02/07/2006 12:57:20") = 2
```

GetMonth Function

The `GetMonth` function gets the month (1 to 12) of the date expression.

Syntax:

```
GetMonth(date expression)
```

Example:

```
GetMonth("02/07/2006 12:57:20") = 7
```

GetYear Function

The `GetYear` function gets the year of the date expression.

Syntax:

```
GetYear(date expression)
```

Example:

GetYear("02/07/2006 12:57:20") = 2006

GetDayOfWeek Function

The `GetDayOfWeek` function gets the day of the week (1 to 7, where 1 denotes Sunday) of the date expression.

Syntax:

GetDayOfWeek(date expression))

Example:

```
GetDayOfWeek("02/07/2006 12:57:20") = 1
```

GetHour Function

The `GetHour` function gets the hour (0 to 23) of the date expression.

Syntax:

```
GetHour(date expression)
```

Example:

```
GetHour("02/07/2006 12:57:20") = 12
```

GetMinute Function

The `GetMinute` function gets the minute (0 to 59) of the date expression.

Syntax:

```
GetMinute(date expression)
```

Example:

```
GetMinute ("02/07/2006 12:57:20") = 57
```

GetSecond Function

The `GetSecond` function gets the second (0 to 59) of the date expression.

Syntax:

```
GetSecond(date expression)
```

Example:

```
GetSecond ("02/07/2006 12:57:20") = 20
```

Age Function

The `Age` function gets the age calculates from the current date/time, including the year (for example, 31 and a half years old = 31.5) of the date expression.

Syntax:

```
Age(date expression)
```

Example:

```
Age ("31/01/1973") = 33.5
```

Now Function

The `Now` function gets the current date/time of the machine running the Plan Interpreter.

Syntax:

```
Now()
```

Example:

```
Now() = "02/07/2006 12:57:20"
```

FormatDate Function

The `FormatDate` function formats the date in `expression1` according to the format specification in `expression2`.

`expression1` should be a Date (from the data), a literal date, or a variable (data type: date).

Notes:

- The format of the date from the data is locale specific; meaning it will use the regional date setup on your system to read Month vs. Day.
- Literal dates only must be formatted as #dd/mm/yyyy# or #dd/mm/yy# and must use a leading zero when necessary for single digits. For example: July 4, 2010 would be entered as #04/07/2010#
- Dates must use a delimiter between Month, Day and Year. The following delimiters are supported: / (forward slash),   \ (back slash), – (dash) and . (period).

Syntax:

```
FormatDate(expression1, expression2)
```

Example:

The following example shows how to format a Date literal:

```
FormatDate(#27/06/2006#, "dddd, MMMM dd, yyyy") = "Tuesday, June 27,
2006"
```

The following example shows how to format a variable input (Data Source field):

```
FormatDate(|->[Birthday], "dddd, MMMM dd, yyyy")
```

The following table shows the characters that can be used to format the date and the resulting date format:

Option	Description
d	Displays the day as a number without a leading zero (for example, 1).
dd	Displays the day as a number with a leading zero (for example, 01).
ddd	Displays the day as an abbreviation (for example, Sun).
dddd	Displays the day as a full name (for example, Sunday).
M	Displays the month as a number without a leading zero (for example, January is repr
MM	Displays the month as a number with a leading zero (for example, 01/12/01).
MMM	Displays the month as an abbreviation (for example, Jan).
MMMM	Displays the month as a full month name (for example, January).

y	Displays the year number (0-9) without leading zeros.
yy	Displays the year in two-digit numeric format with a leading zero, if applicable.
yyy	Displays the year in three-digit numeric format.
yyyy	Displays the year in four-digit numeric format.
h	Displays the hour as a number without leading zeros using the 12-hour clock (for ex
hh	Displays the hour as a number with leading zeros using the 12-hour clock (for examp
H	Displays the hour as a number without leading zeros using the 24-hour clock (for ex
HH	Displays the hour as a number with leading zeros using the 24-hour clock (for examp
m	Displays the minute as a number without leading zeros (for example, 12:1:15).
mm	Displays the minute as a number with leading zeros (for example, 12:01:15).
s	Displays the second as a number without leading zeros (for example, 12:15:5).
ss	Displays the second as a number with leading zeros (for example, 12:15:05).
T	Displays an uppercase 'A' for any hour before noon; displays an uppercase 'P' for an
TT	Displays an uppercase 'AM' for any hour before noon; displays an uppercase 'PM' fo
t	Displays a lowercase 'a' for any hour before noon; displays an lowercase 'p' for any h

tt	Displays an lowercase 'am' for any hour before noon; displays an lowercase 'pm' for
Any other	Displays as is.

Barcode Function for Print Media

uPlan supports adding barcodes for print output. The barcodes are generated on the fly, based on personalized data.

Adding a barcode to your Dynamic Document is done by creating a Graphic ADOR whose rule expression contains a call to the XMPBarcode function.

The XMPBarcode function takes two mandatory parameters:

- Name: the name of the barcode
- String to be encoded: the value that is encoded by the barcode

Example of a static barcode: `XMPBarcode ("QRCode","http://www.xmpie.com/udirect")`

In this example, `QRCode` is the name of the barcode and `http://www.xmpie.com/udirect` is the string to be encoded. This function will generate a QR code barcode that, once scanned, will lead to the http://www.xmpie.com/udirect website.

Example of a dynamic barcode: `XMPBarcode("QRCode",“http://www.lioncomm.com/SummerSale/?”)`

where ? refers to the XMPieRecipientKey.

In addition to the first two mandatory parameters, you can provide a third parameter that sets up various additional options. The following options are available for further customizing the barcode:

- Module width (see Module Width Parameter)
- Code page (see CodePage Parameter)
- Binary string (see BinaryString Parameter)
- Color (see Color Parameters)

To use a barcode later in your Dynamic Document, simply place the Graphic ADOR in the design, as you would with any Graphic ADOR.

String Functions

As a rule, string functions change the input expression and return an updated string. The input expression is always regarded as a string; even if it appears as a date or number, it is interpreted as a string.

String functions include:

- LCase Function
- TCase Function
- UCase Function
- Length Function
- IsNullOrEmpty Function
- SubString Function

- Trim Function
- LTrim Function
- RTrim Function
- Find Function
- Replace Function
- FindAndReplace Function
- FindAndReplaceChars Function
- FindByRegExp
- CleanNumber
- CleanRecipientKey
- SecureID
- HexToUnicode Function
- HtmlEncode

LCase Function

The `LCase` function assigns lower case formatting. It changes the expression to lower case characters and returns a string.

Syntax:

```
Lcase(expression)
```

Example:

```
Lcase("HELLO") returns the value "hello".
```

TCase Function

The `TCase` function assigns title style formatting. It changes the expression so that every new word (starting after a non-alphabetical character) will start with an upper case character and returns a string.

Syntax:

```
Tcase(expression)
```

Example:

```
Tcase("formatting functions") returns the value "Formatting Functions".
```

UCase Function

The `UCase` function assigns upper case formatting. It changes the expression to upper case characters and returns a string.

Syntax:

```
UCase(expression)
```

Example:

```
UCase("This is it") returns the value "THIS IS IT".
```

Length Function

The `Length` function returns the number of characters in the input expression.

Syntax:

```
Length(expression)
```

Example:

```
Length("abc") returns the value of 3
```

IsNullOrEmpty Function

The `IsNullOrEmpty` function determines if the input expression is NULL (for example, a NULL Data Source field) or empty (for example, an empty string). This function returns `true` if the string is NULL or empty and `false` otherwise.

Syntax:

```
IsNullOrEmpty(expression)
```

Example:

```
IsNullOrEmpty(|->[Address2]) returns the value of false if Address2
contains a value, and true if it is either NULL or Empty.
```

SubString Function

The `SubString` function retrieves a sub-string from `expression1`,
with `expression3` characters, starting from the position `expression2`.

Syntax:

```
SubString(expression1,expression2,expression3)
```

Where:

- `expression1` is always regarded as a string; even if it appears as a date or number, it will be interpreted as a string.
- `expression2` indicates the starting position of the substring.

  For example, 0 represents the first character position in `expression1`, 1 represents the second character position, etc.
- `expression3` is the number of characters retrieved.

For example, 3 retrieves three characters, 0 retrieves an empty string, etc.
A value of –1 indicates that all characters until the end of `expression1` should be retrieved.

Examples:

```
SubString("abcd",1,2) = "bc"

SubString("James",2,2) = "me"

SubString(1973,0,2)  = "19"

SubString("James",2,-1) = "mes"
```

Trim Function

The `Trim` function trims leading and trailing white spaces in the expression.

Syntax:

```
Trim(expression)
```

Example:

```
Trim(" hello WORLD ") = "hello WORLD"
```

LTrim Function

The `LTrim` function trims leading white spaces in the expression.

Syntax:

```
LTrim(expression)
```

Example:

```
LTrim(" hello WORLD ") = "hello WORLD "
```

RTrim Function

The `RTrim` function trims trailing white spaces in the expression.

Syntax:

```
RTrim(expression)
```

Example:

```
RTrim(" hello WORLD ") = " hello WORLD"
```

Find Function

The `Find` function gets the character index (zero based) in `expression1`, where the string (`expression2`) is found. The search starts from the character index (`expression3` – zero based). The result is -1 in case the string is not found.

Syntax:

```
Find(expression1, expression2, expression3)
```

Example:

```
Find("hello WORLD", "WORLD", 0) = 6
```

Replace Function

The `Replace` function replaces a part of the `expression1` starting from character index (`expression3` – zero based) of length (`expression4`) with the string (`expression2`).

Syntax:

```
Replace(expression1, expression2, expression3, expression4)
```

Example:

```
Replace("hello WORLD", "EARTH", 6, 5) = "hello EARTH"
```

FindAndReplace Function

The `FindAndReplace` function replaces all instances of the string (`expression2`) in `expression1` with the string (`expression3`).

Syntax:

```
FindAndReplace(expression1, expression2, expression3)
```

Example:

```
FindAndReplace("hello WORLD", "WORLD", "EARTH") = "hello EARTH"
```

FindAndReplaceChars Function

The `FindAndReplaceChars` function replaces all instances of one or more characters in a string with another set of characters. For example, you can use this function to clean invalid URL characters from a string.

Syntax:

```
FindAndReplaceChars(expression1, expression2, expression3)
```

- `expression1` - a string containing the characters to be replaced
- `expression2` - list of characters to be replaced
- `expression3` - optional. Replacement string to be used instead of any of the characters listed in expression2. The string may be empty or contain multiple characters. If the string is omitted, the characters in expression2 are removed.
- Return value - the updated string, containing the replaced characters.

Example:

The following example replaces the space character with an underscore (“_”):

```
FindAndReplaceChars("John Michael Smith", " ", "_") =
"John_Michael_Smith”
```

The following example removes the space character:

```
FindAndReplaceChars("John Michael Smith", " ") = "JohnMichaelSmith"
```

FindByRegExp

The `FindByRegExp` function keeps only the string matching the definition of the regular expression.

Syntax

```
FindByRegExp(exprAsString, regExpString, firstMatchOnly)
```

- `exprAsString` - The input field to which the regular expression will be applied.
- `regExpString` - The regular expression rule.
- `firstMatchOnly` - If the value is False all results are returned. If True - only the first match is returned.

Example

```
FindByRegExp("1222 Duncan Avenue", "[A-Za-z]+",True) = Duncan
```

CleanNumber

The `CleanNumber` function allows to strip any characters from a string, leaving only digits. This is particularly useful for cleaning up telephone numbers and leaving only digits.

Syntax:

```
CleanNumber(exprAsString)
```

Example:

The following example removes the space, + and - characters from the string:

```
CleanNumber("+1 888-452-1111") = 18884521111
```

CleanRecipientKey

The `CleanRecipientKey` function allows you to handle invalid characters when adding new Recipient Keys to the Data Source (using the `INSERT` expression).

This function finds all instances of invalid URL characters (`space, ":", "?", "&", "*", "#", "<", ">", "|", quotes,"'"`) in the `RecipientKey` expression, and replaces them with the `ReplaceInvalidWith` string.

"ReplaceInvalidWith" is optional: if you do not specify it, the invalid URL characters in "RecipientKey" will be removed.

Syntax:

```
CleanRecipientKey(RecipientKey, ReplaceInvalidWith)
```

Example:

The following example replaces the space character with an underscore ("_"):

```
CleanRecipientKey("John Michael.Smith", "_") = "John_Michael.Smith"
```

The following example removes the space character:

```
CleanRecipientKey("John Michael.Smith") = "JohnMichael.Smith"
```

SecureID

Use the SecureID option to generate a unique, non-guessable and secure recipient ID (for example,   5adcf67b419a4ad796da4458d25a038e).

Syntax/Example

The following example generates a unique ID for each recipient:

```
SecureID()
```

HexToUnicode

Hexadecimal code allows you to represent special unicode characters, which cannot be represented by symbols, by using their hexadecimal values.
The `HexToUnicode` function allows you to treat a hexadecimal string as a unicode string.

The input parameter, `ExprAsHexString`, is a string of hexadecimal characters. Each sequence of four hexadecimal characters is converted into the matching unicode character (if the string cannot be divided into four, the function automatically pads the string with leading zeros).

Note that the input string value is ordered in big-endian.

Syntax:

```
HexToUnicode(ExprAsHexString)
```

Example:

In order to use the tab character, you can use its Hex value, 9, as follows:

```
"Hello" + HexToUnicode("9") + "World"
```

...would result in:

```
Hello<Tab>World
```

HtmlEncode

The `HtmlEncode(strToEncode)` function converts a string into an HTML-encoded string, and returns the encoded string. Usually used to encode a URL, so that it can be safely posted to an HTTP server (i.e. URL encoding).

Syntax

```
HtmlEncode(strToEncode)
```

Example:

```
HtmlEncode("http://my_domain/my page.html")
```

...would result in:

```
"http://my_domain/my%20page.html"
```

Conversion Functions

Conversion functions allow you to convert different types of data.

Conversion functions include:

- AsBoolean Function
- AsDate Function
- AsNumber Function
- AsString Function
- AsJsonArray Function

AsBoolean Function

The `AsBoolean` function evaluates the expression as True/False.

The `Null` value is false regardless of the data type.

For a Number expression, 0 is False; otherwise it is True.

For String expressions, Empty String is False; otherwise it is True.

A Date expression is always True.

Syntax:

```
AsBoolean(expression)
```

Example:

```
AsBoolean(1) = True
```

AsDate Function

The `AsDate` function attempts to evaluate the expression as a date, if possible.

For example, the string "23/02/1994" will be evaluated as 23/02/1994.

If the expression cannot be evaluated as a date, the function fails and an error is reported.

Null value conversion returns the current date/time.

Number value conversion assumes you are calculating the date starting from `31/12/1899` and adds the given number as the number of days elapsed (for example: `AsDate(1) = 31/12/1899`).

Syntax:

```
AsDate(expression)
```

Example:

```
AsDate("02\07\2006") = 02\07\2006
```

AsNumber Function

The `AsNumber` function evaluates the expression as a number.

Null becomes zero.

True and False are evaluated to 1 and 0, respectively.

A string beginning with a number (or leading spaces followed by a number) returns the number. Any other string is evaluated as 0.

Syntax:

```
AsNumber(expression)
```

AsNumber(23ab) is evaluated as 23.

AsNumber(ab23) is evaluated as 23, and "ab" is evaluated as 0.

Example:

```
AsNumber("5") = 5
```

AsString Function

The `AsString` function evaluates the expression as a string.

Null becomes an empty string.

Syntax:

```
AsString(expression)
```

Examples:

Expression	Outcome
AsNumber("+24")	24
AsNumber("23rd street")	23
AsNumber("James")	0
AsDate("1/1/2002")	01/01/2002
AsString(12)	the string "12"

AsJsonArray Function

From version 11.2

The `AsJsonArray` function takes multiple comma separated expressions and returns a JSON Array containing the values of the expressions.

This provides a useful way to pass multiple ADORs, variables, or calculated values to another function as a single object. The second function can simply use the JavaScript JSON.parse() function to create a JavaScript array and access the values.

Syntax:

```
AsJsonArray(expression1, expression2, …)
```

Example:

```
AsJsonArray(|->[customerid], @{firstname}, 1+3) = ["ML343","John",4]
```

GetEnv Functions

Environment constants allow you to retrieve data during production.

Syntax:

Constants are used as follows:

```
GetEnv("constant name")
```

GetEnv functions include:

- Current Record Number
- Current Record Number in Batch
- Print Media
- Proof Set
- HTML Media
- Text Media
- Host Application
- Job ID
- Parent Job ID
- Job Type

- Split Part
- Document Name
- Document ID
- Document Type
- Instance ID
- Base URL
- Base Online Document URL
- Circle Touchpoint Friendly ID
- TopMostJobId
- RecipientFilter

Current Record Number

The environment constant `CurRecordNumber` returns the number of the current record being processed.

In case of split production, each job will continue the numbering from the previous batch.

For example, if production of records 21 - 40 is split into two jobs, the numbering of the first job will be 1 - 10, and that of the second job will be 11 - 20.

Syntax:

```
GetEnv ("CurRecordNumber")
```

Example:

```
GetEnv("CurRecordNumber") = 5
```

Current Record Number in Batch

The environment constant `CurRecordNumberInBatch` returns the number of the current record being processed.

In case of split production, each job will restart its numbering.

For example, if production of records 21 - 40 is split into two jobs, the numbering of the first job will be 1 - 10, and that of the second job will be 1 - 10.

Syntax:

```
GetEnv ("CurRecordNumberInBatch")
```

Example:

```
GetEnv("CurRecordNumberInBatch") = 5
```

Print Media

The environment constant `PrintMedia` returns True if the production is of a print Document.

Syntax:

```
GetEnv("PrintMedia")
```

Proof Set

The environment constant `ProofSet` returns True if the production is of a Proof Set.

Syntax:

```
GetEnv("ProofSet")
```

HTML Media

The environment constant `HTMLMedia` returns True if the production is of html: on demand, email, or proof html.

Syntax:

```
GetEnv("HTMLMedia")
```

Text Media

The environment constant `TextMedia` returns True if the production is of text: SMS, text.

Syntax:

```
GetEnv("TextMedia")
```

Host Application

The environment constant `HostApplication` returns the name of the application that executes the Plan. Possible return values are listed below.

Return Value	Description
`uProduce`	The Plan is executed by uProduce. uProduce uses a Plan to produce cross-media outputs. ● For Print Campaigns, uProduce can produce Print and Proof jobs, as well as Proof Sets. ● For Web Campaigns, uProduce can produce ICPs and email batches. ● For Cross-Media Campaigns, uProduce can perform all of the above production types.
`InDesign`	The Plan or expression is executed by uCreate. uCreate can produce the following types of output: ● Print: by choosing the panel’s Print Document Dynamic Print... menu option. ● Proof Set: by choosing the panel’s Export... menu option.
`uPlan`	The Plan is executed by uPlan. uPlan uses a Plan to produce Proof Sets.

Syntax:

```
GetEnv("HostApplication")
```

Job ID

The environment constant `JobId` returns the job ID.

Syntax:

```
GetEnv ("JobId")
```

Example:

```
GetEnv("JobId") = 103
```

Parent Job ID

The environment constant `ParentJobId` returns, in case of a turbo job, the job ID of the parent job.

Syntax:

```
GetEnv ("ParentJobId")
```

Example:

```
GetEnv("ParentJobId") = 103
```

Job Type

The environment constant `JobType` returns the job type defined in the job ticket. Possible return values are listed below.

Value	**Description**
PRINT	A Print job. ● In uCreate Print, this value indicates a job created using the panel’s Print DocumentDynamic Print... menu option. ● In uProduce, this value indicates a job created by clicking the Process button of the Document Details page.
PROOF	A Proof job. This value indicates a job created using uProduce, by clicking the Proof button of the Document Details page.
PROOF_SET	A Proof Set job. ● In uCreate Print, this value indicates a job created using the panel’s Export... menu option. ● In uProduce, this value indicates a job created by clicking the Generate button of the Plan Details page. ● In uPlan, this value indicates a job created using one of the Data menu’s "Generate" options (for example, Generate Proof Set... option).
`ON_DEMAND`	A Dynamic HTML production job (previously known as HTML production). This value indicates a job created using uProduce, by clicking the Deploy button of the Web (HTML or TXT) Document Details page.

RECORD_SET	An Interactive Content Port (ICP) job. This value indicates that the Plan is executed by an ICP.
FLAT	A job that collects information on a future Print job. Several FLAT jobs of the same Document are later aggregated and used as the input of a single Print job. This is normally the case with job aggregation created by uStore.
EMAIL_MARKETING	An e-mail batch job. This value indicates a job created by clicking the Send button of the Email Activity Details page.
EMAIL_MARKETING_TEST	An e-mail batch test job. This value indicates a job created by clicking the Test button of the Email Activity Details page.

Syntax:

```
GetEnv("JobType")
```

Split Part

When splitting a job, you can get the current split part number.

Syntax:

```
GetEnv ("SplitPart")
```

Example:

```
GetEnv("SplitPart") = 2
```

Document Name

The environment constant `DocumentName` returns the name of the document from the job ticket.

The Document name is always defined in print jobs and in email jobs. If the document name is not defined, this function returns an empty string.

This parameter can be used in the campaign's ADOR expressions, to set a different logic for different documents within the same campaign. For example, you can set a rule that creates high resolution images for a specific document named "HighQualityPostcard" and low resolution images for documents by any other name.

Syntax:

```
GetEnv ("DocumentName")
```

Example:

```
GetEnv("DocumentName") = HighQualityPostcard
```

Document ID

The environment constant `DocumentID` returns the ID of the document from the job ticket.

The document ID is always defined in jobs created by uProduce, but is never defined in jobs created by uCreate. If the Document ID is not defined, this function returns an empty string.

This parameter can be used in the campaign's ADOR expressions, to set a different logic for different documents within the same campaign.
For example, you can set a rule that creates high resolution images for a specific document whose ID is 5 and low resolution images for document by any other ID.

Syntax:

```
GetEnv ("DocumentID")
```

Example:

```
GetEnv("DocumentID") = 5
```

Document Type

The environment constant `DocumentType` returns the Document type defined in the job ticket. Possible return values are listed in the following table.

Return Value	**Description**
`HTML`	An HTML Document
`INDD`	An Adobe InDesign Document
`TXT`	A text Document

`XLIM`	An XMPie proprietary XLIM Document

Syntax:

```
GetEnv("DocumentType")
```

This expression returns a valid value only for the following job types (other job types return an empty string):

- `PRINT`
- `PROOF`
- `ON_DEMAND`
- `EMAIL_MARKETING`
- `EMAIL_MARKETING_TEST`

Instance ID

When creating an instance from a template, the website is defined in the template and is shared by all its instances.

All instances access the same webpages and use the same website URLs. The instance ID in the URL is used to differentiate between the instances.

In order to get to the InstanceID value, you need to create an ADOR that includes the `GetEnv("InstanceID")` function.

For example, to create a URL to a specific webpage using the Instance ID ADOR, e.g. {{GetEnvInstanceID}}, the URL format should be:

- http://www.MyDomain.com/{{GetEnvInstanceID}}/{{XMPieRecipientKey}}
- http://www.MyDomain.com/MyFolder/Landing.html?iid={{GetEnvInstanceID}}&rid={{XMPieRecipientKey}}

Syntax:

```
GetEnv("InstanceID")
```

Base URL

The base URL for the website defined in the Circle campaign, template or instance.

Syntax:

```
GetEnv("BaseURL")
```

Example:

```
GetEnv("BaseURL") =
"http://www.MyDomain.com/1432/{{XMPieRecipientKey}}"
```

Base Online Document URL

The base URL for an on-demand PDF defined in the Circle campaign, template or instance.

Syntax:

```
GetEnv("BaseOnlineDocURL")
```

Example:

```
GetEnv("BaseOnlineDocURL") =
"http://www.MyDomain.com/XMPieDownloadPDF?"
```

Circle Touchpoint Friendly ID

The environment constant `CircleTouchpointFriendlyID` returns the friendly ID of the Circle touchpoint from the job ticket.

Syntax:

```
GetEnv("CircleTouchpointFriendlyID")
```

TopMostJobId

The environment constant `TopMostJobId` returns the job ID, or in case of a turbo job, the job ID of the parent job.

Syntax:

```
GetEnv ("TopMostJobId")
```

Example:

```
GetEnv("TopMostJobId") = 103
```

RecipientFilter

The environment constant `RecipientFilter` returns the recipient filter used in production, which is either the table name, plan filter name or query.

Syntax:

```
GetEnv ("RecipientFilter")
```

Example:

```
GetEnv("RecipientFilter") = "Recipient Table"
```

uImage Functions

The following uImage functions are used to create personalized images.

To learn about defining and generating personalized images in uPlan, see the uImage Help.

uImage.uImage

The `uImage.uImage()` call is used to create personalized images from a Photoshop Document Package.

uImage.CreateImage2

The `uImage.CreateImage2()` call is used to is used to create personalized images from a Photoshop Template.

uImage.CreateIllustration2

The `uImage.CreateIllustration2()` call is used to is used to create personalized illustrations from an Illustrator Template.

Miscellaneous Operators and Functions

Miscellaneous operators and functions include:

- Abort Operator
- ReportMessage Function
- Skip Operator
- Call Function
- Fetch Function

Abort Operator

The `Abort` operator serves to abort job processing (Plan execution) based on data values and Logic during the calculation of an ADOR value. The job itself is marked as aborted and a message reports that the job was aborted due to the Abort operation.

Example:

Say we wish to abort the current job if we find out that specific data is missing. For example, if the last name of a customer is missing we will abort the job. In this case, the expression for the last name ADOR will appear as follows:

```
If(|->[Last Name] = NULL or |->[Last Name] = "")

   Abort
```

```
else

   |->[ Last Name]
```

ReportMessage Function

The `ReportMessage` function inserts a message into the message list during production.

Syntax:

```
ReportMessage(message)
```

This function is different from all other functions because it does not have a value that can be used to populate the ADORs; therefore, it must be followed by an expression that has a value. You can use several calls to this function one after the other, but the last expression in the call list must be an expression that evaluates to some value (that is, not a ReportMessage function):

```
ReportMessage(msg1)

ReportMessage(msg2)

…

ReportMessage(msgn)

Expression
```

Example:

Say, for example, we want to report a message in case a record is skipped as a result of the Skip operator.

In this example, we wish to create a Campaign for customers that have credit of more than 10000. An ADOR defined for the customer's credit may appear as shown in the Skip Operator section.

```
If(|->[Credit] < 10000)

{

ReportMessage("Record number " & GetEnv("CurRecordNumber") & " was skipped")

Skip

}

else

   |->[Credit]
```

Skip Operator

The `Skip` operator skips the current record during job processing (Plan execution), without affecting the success of the job. The Skip operator enables uProduce to continue the job without creating customer communications for specific recipients based on their data values and the Logic defined for calculated ADOR values. By default, no special messages are displayed when a record is skipped. To specify that a message should be reported, you can use the ReportMessage Function.

Example:

Say we want to create a Campaign directed at customers that have credit of more than 10000, and skip those that have less than 10000. In this case, the ADOR for the customer's credit may appear as follows:

```
If(|->[Credit] < 10000)

   Skip
```

```
else

    |->[Credit]
```

Call Function

The `Call` function is used for integrating a Plan with QLingo extension modules.

Syntax:

```
Call dllName.functionName(parameters)
```

Fetch Function

From version 11.2

The `Fetch` function allows you to make HTTP requests to a server, such as Rest API calls.

Use the Fetch function to call any server side code and use it as part of the plan. For example, use Google APIs, Amazon APIs and even uProduce APIs.

The XMPie.fetch() function is based on the JavaScript fetch API.

For example, we can use the Fetch function to:

- Get current currency exchange rate from such a service.
- Get the weather information from a weather service.

Syntax

```
XMPie.fetch("<URL>")
```

Optional argument: JSON parameter that may include the following keys:

- method - GET/PUT/POST/DELETE
- headers
- body

Example

```
XMPie.fetch("http://example.com/movies.json", {method:GET,
headers:{'Content-Type': 'application/json'} ,
body:"JSON.stringify(data)"}
```

Asset Functions

Asset functions include:

- XMPBarcode
- IsAssetExist
- GetAssetPath
- GetAssetWebURL
- GetPDFAssetPageCount
- Rectangle
- Web.URLContentAsFile

IsAssetExist

The `IsAssetExist` function checks if the asset exists. Returns true if it exists, otherwise returns false.

Syntax:

```
IsAssetExist(AssetIdAsString, mediaTypeNumber)
```

The `mediaTypeNumber` parameter is optional. The default value is 8.

Asset (media) types include:

- 3 - for web graphics (graphics valid for web, such as .jpg)
- 6 - for web text
- 7 - for print text
- 8 - for print graphics, such as .pdf (default value)

Example:

```
IsAssetExist("Pic1", 3) = True
```

GetAssetPath

The `GetAssetPath` function returns the file path of the asset. If none is found, returns null.

Syntax:

```
GetAssetPath(AssetIdAsString, mediaTypeNumber)
```

Asset (media) types include:

- 3 - for web graphics (graphics valid for web, such as .jpg)
- 6 - for web text
- 7 - for print text
- 8 - for print graphics (such as .pdf)

Example:

```
GetAssetPath("Pic1", 3) = C:\\Pic1.png
```

```
GetAssetPath("Pic1", 8) = C:\\Pic1.pdf
```

GetAssetWebURL

The `GetAssetWebURL` function returns the asset's web URL. If none is found, returns null.

Syntax:

```
GetAssetWebURL(AssetIdAsString, mediaTypeNumber)
```

Asset (media) types include:

- 3 - for web graphics (graphics valid for web, such as .jpg)
- 6 - for web text

- 7 - for print text
- 8 - for print graphics (such as .pdf)

Example:

```
GetAssetWebURL("Pic1", 3) = https://www.xmpie.com/wp-content/Pic1.png
```

```
GetAssetWebURL("Pic1", 8) = https://www.xmpie.com/wp-content/Pic1.pdf
```

GetPDFAssetPageCount

The `GetPDFAssetPageCount` function returns the PDF's number of pages. If a PDF is not found, returns 0.

Syntax:

```
GetPDFAssetPageCount(assetIdAsString)
```

Example:

```
GetPDFAssetPageCount("Doc1") = 6
```

Rectangle

This function can be used to fill any shape created within InDesign (for example a circle, a polygon, etc.).

The Rectangle function generates on-the-fly a dynamic rectangle (graphic image) in the required color (CMYK).

Using this function you can set dynamic object colors using variables and campaign dials that can be used in uStore products.

Click here to learn how to create a dynamic color picker in uStore using the Rectangle function.

Syntax

```
Rectangle(cyanPercentage, magentaPercentage, yellowPercentage,
blackPercentage)
```

Example

```
Rectangle(12, 87, 71, 3) = Flame Scarlet
```

```
Rectangle(88, 67, 20, 5) = Classic Blue
```

Web.URLContentAsFile

The `Web.URLContentAsFile` function gets an image from the internet via its URL.

See Video training

Syntax

```
Call Web.URLContentAsFile(URL, timeout, defaultAssetName, contentType)
```

- URL: Required. The URL to an image, e.g. http://imagesdomain/imagesfolder/imagename.jpg
- timeout - (Optional) The default is 30 seconds, after which the request is timed out.

- defaultAssetName - (Optional) Default asset in case of error (including time out). If no asset is defined, the error will be handled as a missing asset.
- contentType - (Optional) By default it is automatically detected. You can explicitly define a IANA mime type.

Example

```
Web.URLContentAsFile("http://imagesdomain/imagesfolder/imagename.jpg
")
```

Circle Functions

Circle functions include:

- Instance ID
- GetTouchpointData

Instance ID

When creating an instance from a template, the website is defined in the template and is shared by all its instances.

All instances access the same webpages and use the same website URLs. The instance ID in the URL is used to differentiate between the instances.

In order to get to the InstanceID value, you need to create an ADOR that includes the `GetEnv("InstanceID")` function.

For example, to create a URL to a specific webpage using the Instance ID ADOR, e.g. {{ GetEnvInstanceID}}, the URL format should be:

- http://www.MyDomain.com/{{GetEnvInstanceID}}/{{XMPieRecipientKey}}
- http://www.MyDomain.com/MyFolder/Landing.html?iid={{GetEnvInstanceID}}&rid={{XMPieRecipientKey}}

Syntax:

```
GetEnv("InstanceID")
```

GetTouchpointData

The `GetTouchpointData` function gets two parameters:

- Touchpoint friendly identifier, i.e. W1, W2, P5
- Touchpoint data type, i.e. URL

The function returns the data for the touchpoint defined by these parameters.

Note: The data of the touchpoint will be available only when production is run form Circle. If not running via Circle, production will not fail.

Syntax

```
GetTouchpointData("Touchpoint friendly identifier", "Touchpoint data
type")
```

Example

```
GetTouchpointData("W1", "URL")
```

Returns the landing page URL of the web touchpoint W1 for the specific recipient and Circle project instance.

User View Functions

User View functions include:

- RowCount Function
- UV2STR Function

RowCount Function

The `RowCount` function receives a user view reference as input and returns the number of rows in the user view per recipient.

Syntax:

```
RowCount([UV Name])
```

Example:

```
RowCount([MyUV]) returns the value 2
```

UV2STR Function

The `UV2Str` function creates a formatted string from a user view.

You can also use this function to format a user view into a string and then use this string as a parameter for another function that will be exported by a QLingo extension.

Syntax:

```
UV2Str([UV Name], 'string before new row', 'string after row is ended',
'string to add between columns', 'Boolean true/false indicating whether
to add column names as first row')
```

Example:

Say, for example, we have a table of the form:

First Name	**Last Name**
John	Smith
Jin	Sonoma

And this table is in the user view UV.

If we call the function:

```
UV2Str([UV], '\"', '\" | ', '\", \"', true)
```

The resulting string will appear as follows:

```
"First Name", "Last Name" | "John", "Smith" | "Jin", "Sonoma" |
```

Custom User-Defined Functions

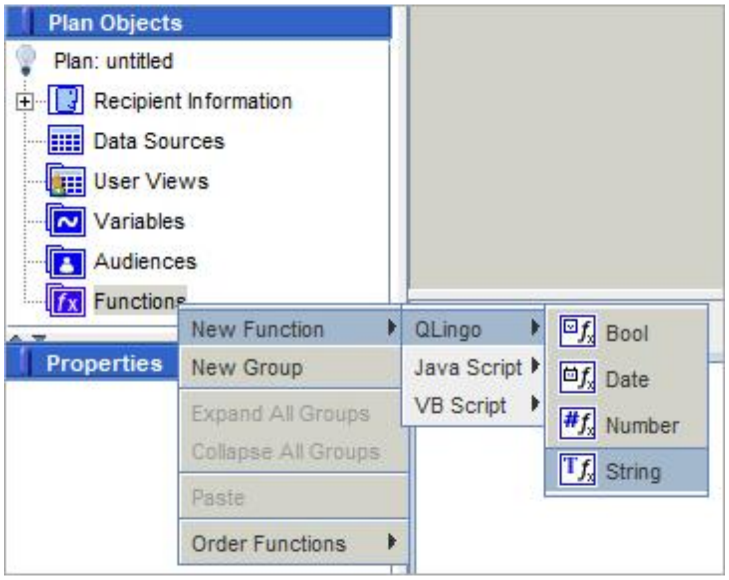
While uPlan has many common built-in functions already, it is also possible to create custom functions using either QLingo or JavaScript to further extend uPlan’s capabilities.

It is not possible to create custom user-defined functions in uCreate Print. However, you can link uCreate Print to a plan file that contains custom functions.

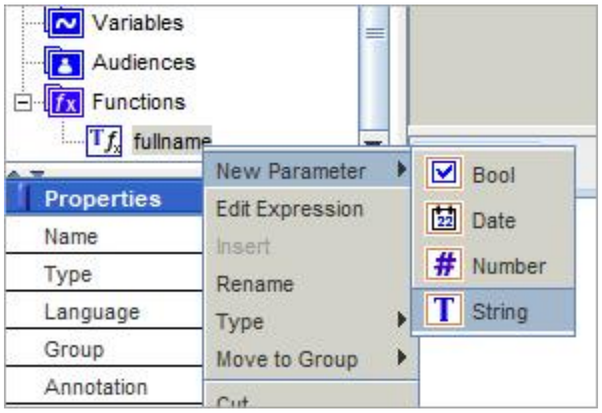
Creating Custom Functions with QLingo

This example will create a function called "fullname" that will take firstname and lastname parameters and concatenate (join) them with a space.

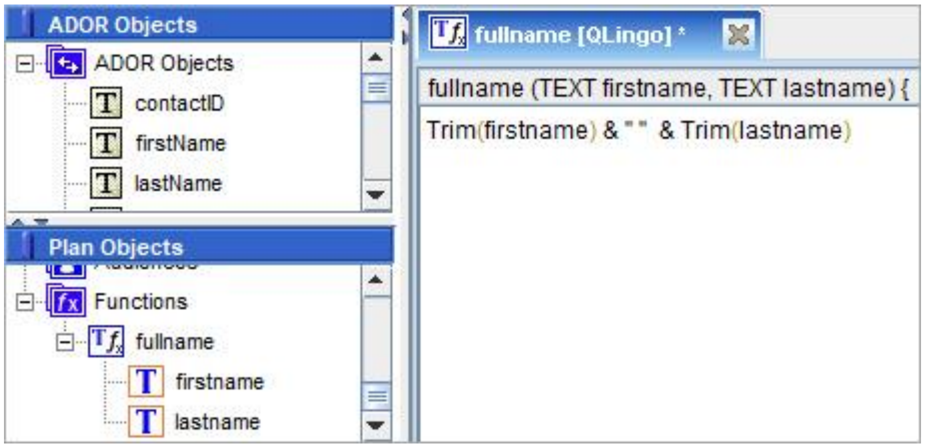
1. Create a new QLingo function and select the type of data that the function will return to uPlan.



2. Name the new function.
3. Right-click on the new function and add parameters for the values that you need to push into the function.



4. Select the appropriate data type for the incoming data and name the new parameters.
5. Repeat for any additional parameters needed by your function. In this example, create parameters for **firstname** (string) and **lastname** (string).
6. Double click the function name, or right-click and select Edit expression.



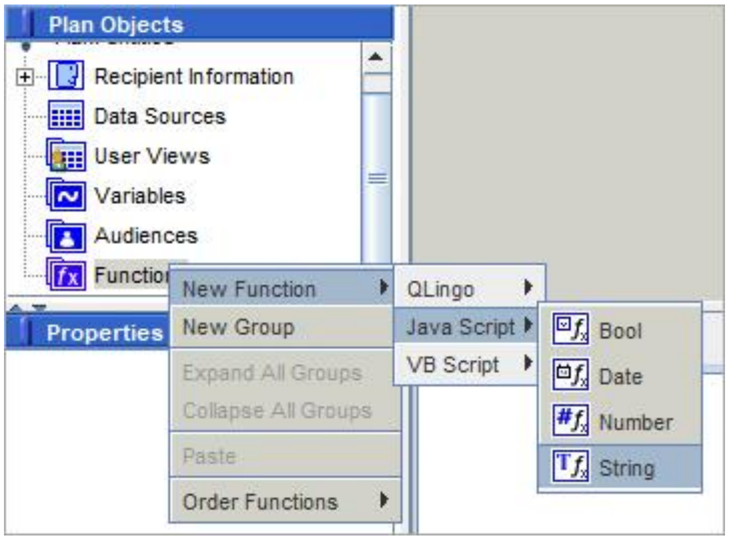
```
fullname (TEXT firstname, TEXT lastname) {
Trim(firstname) & " " & Trim(lastname)
```

7. Write your QLingo expression using normal QLingo syntax. In this example, use the **Trim()** function to remove any white space from the **firstname** and **lastname** parameters, and use the **&** (ampersand) to concatenate the **firstname**, **space** and **lastname**.

Creating Custom Functions with JavaScript

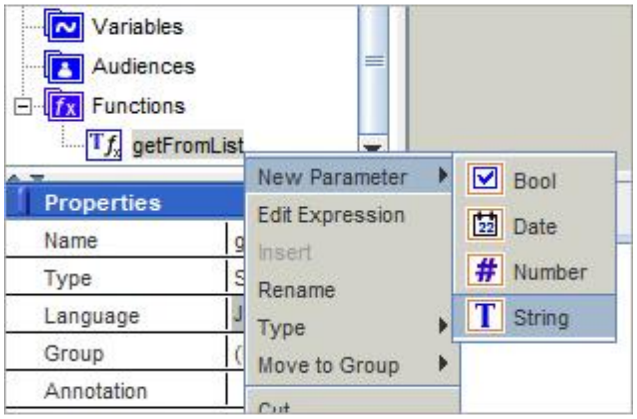
This example will create a function called "getFromList" that will receive three parameters: a list, a delimiter and an index number. And will return the value that is in the index position of the list.

1. Create a new JavaScript function and select the type of data that the function will return to uPlan.



2. Name the new function.

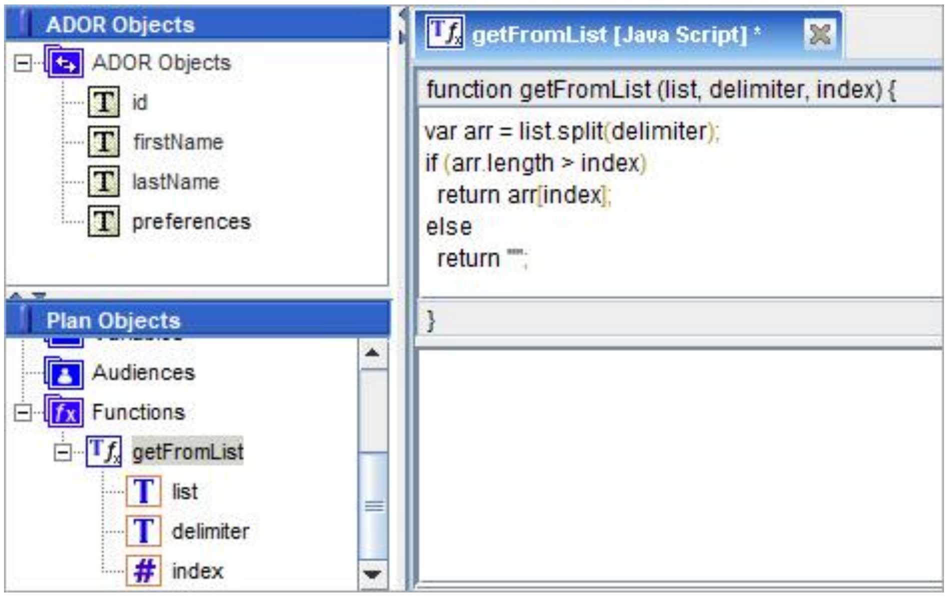
3. Right-click on the new function and add parameters for the values that you need to push into the function.



4. Select the appropriate data type for the incoming data and name the new parameters.

5. Repeat for any additional parameters needed by your function. In this example, create parameters for: **list** (string), **delimiter** (string) and **index** (number).

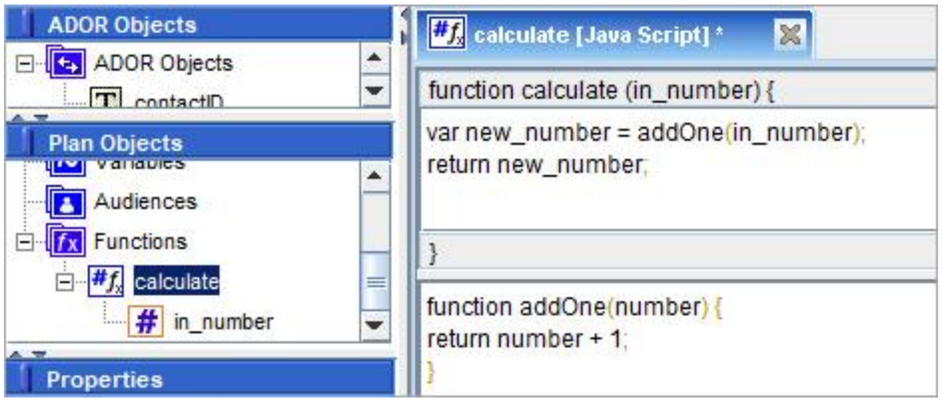
6. Double click the function name, or right-click and select Edit expression.



```
function getFromList (list, delimiter, index) {
var arr = list.split(delimiter);
if (arr.length > index)
  return arr[index];
else
  return "";
}
```

7. Write your JavaScript expression using normal JavaScript syntax. In this example, use the JavaScript **Split()** function to convert the list to an array. You can then simply return the value from the correct index of the array. (Remember that JavaScript is zero-based, so the first element of the array will be index 0.)

The JavaScript expression editor consists of two editable areas:



```
function calculate (in_number) {
var new_number = addOne(in_number);
return new_number;
}
```

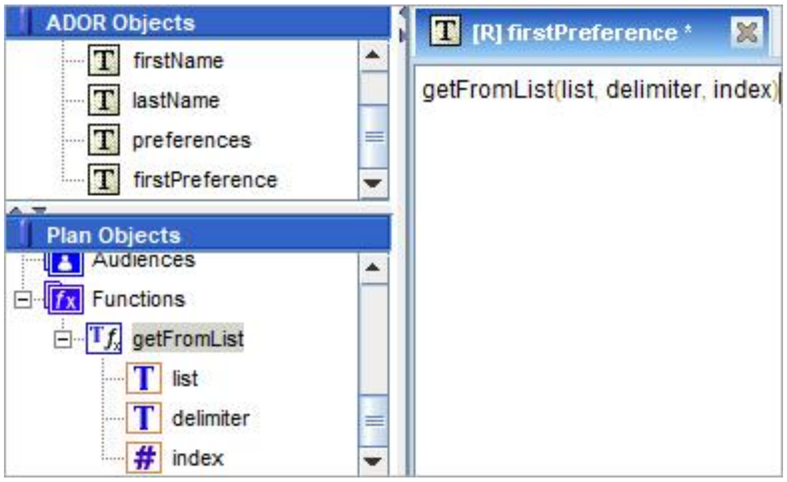
```
function addOne(number) {
return number + 1;
}
```

The upper area is for your main function that you can see is defined automatically. The lower area allows you to define any additional functions that will be called from your main function.

Using the new Custom Function in an ADOR or Variable

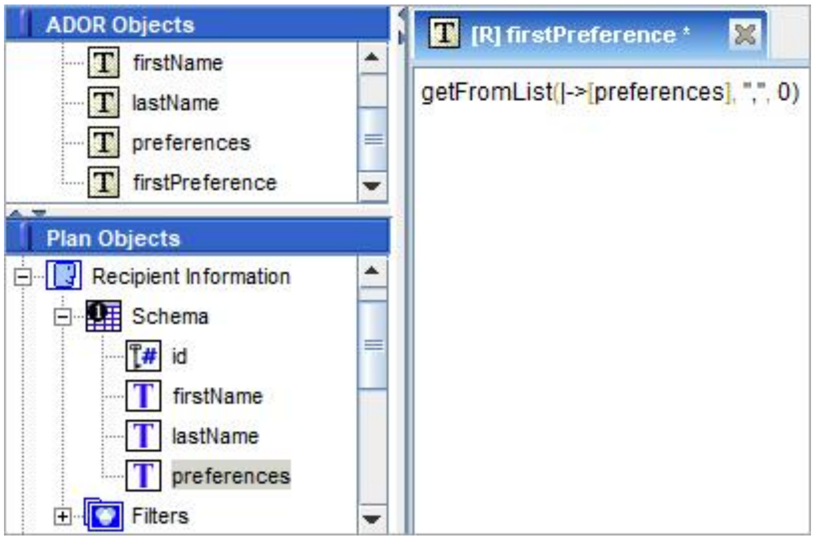
Now that you have created a custom function, you can use it like any other uPlan function, by dragging and dropping the function (or typing the function name) into one or more ADORs or variables and setting the required parameter values.

1. Open your ADOR or variable's Expression. In this case, I have created a new ADOR called "firstPreference". From the Functions list, drag the "getFromList" function into the editor.



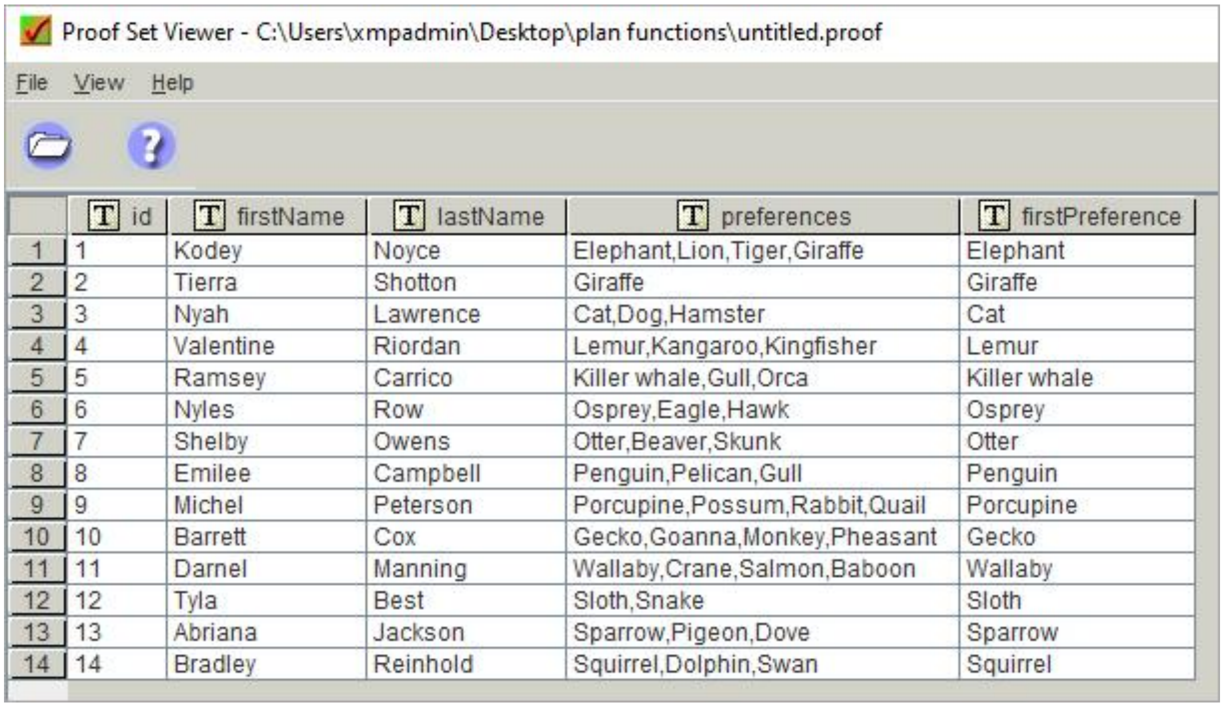
```
getFromList(list, delimiter, index)
```

2. Replace the parameter names with the values you want to pass to the function. In this case:
   - the list is coming from the Recipient Information Schema "preferences" field,
   - the delimiter is "," (since the preferences list is comma delimited), and
   - the index is 0 (zero) to get the first item from the list.



```
getFromList(|->[preferences], ",", 0)
```

3. Generate a proofset to confirm your function is working as expected.

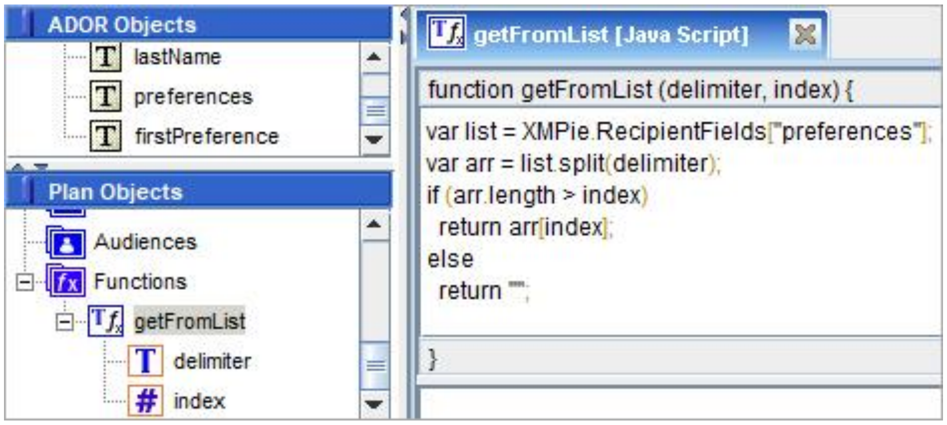

Proof Set Viewer - C:\Users\xmpadmin\Desktop\plan functions\untitled.proof
File
View
Help

	id	firstName	lastName	preferences	firstPreference
1	1	Kodey	Noyce	Elephant,Lion,Tiger,Giraffe	Elephant
2	2	Tierra	Shotton	Giraffe	Giraffe
3	3	Nyah	Lawrence	Cat,Dog,Hamster	Cat
4	4	Valentine	Riordan	Lemur,Kangaroo,Kingfisher	Lemur
5	5	Ramsey	Carrico	Killer whale,Gull,Orca	Killer whale
6	6	Nyles	Row	Osprey,Eagle,Hawk	Osprey
7	7	Shelby	Owens	Otter,Beaver,Skunk	Otter
8	8	Emilee	Campbell	Penguin,Pelican,Gull	Penguin
9	9	Michel	Peterson	Porcupine,Possum,Rabbit,Quail	Porcupine
10	10	Barrett	Cox	Gecko,Goanna,Monkey,Pheasant	Gecko
11	11	Darnel	Manning	Wallaby,Crane,Salmon,Baboon	Wallaby
12	12	Tyla	Best	Sloth,Snake	Sloth
13	13	Abriana	Jackson	Sparrow,Pigeon,Dove	Sparrow
14	14	Bradley	Reinhold	Squirrel,Dolphin,Swan	Squirrel

Accessing other uPlan elements with custom JavaScript functions

Custom JavaScript functions can also directly access the value of other uPlan elements, for example values of a Recipient record, Variables, UserViews and Functions.

In the previous example we passed the preferences list to the function via a parameter. The function can also directly access the recipient list values, so the function could be rewritten to use just two parameters and to get the list directly from the recipient list like this:



```
function getFromList (delimiter, index) {
var list = XMPie.RecipientFields["preferences"];
var arr = list.split(delimiter);
if (arr.length > index)
  return arr[index];
else
  return "";
}
```

Accessing Recipient record field values

To access values from the Recipient record, use:

```
XMPie.RecipientFields["name of field"]
```

For example:

```
XMPie.RecipientFields["firstname"]
```

Accessing Variable values

To access uPlan Variable values, use:

```
XMPie.Variables["name of variable"]
```

For example:

```
XMPie.Variables["age_var"]
```

Accessing UserView record values

To access UserView field values, use:

```
XMPie.UserViews["userview name"][row number]["name of field"]
```

For example:

```
    XMPie.UserViews["Orders_UV"][2]["orderID"]
```

Accessing uPlan functions

Custom JavaScript functions can also directly access uPlan's built-in functions, or other custom functions that have been created in the plan file.

To access a uPlan function, use:

```
    XMPie.Functions."function name"("value")
```

For example:

```
    XMPie.Functions.UCase("This String")
```

Returns:

```
"THIS STRING"
```

Another example using a uPlan function together with a recipient record:

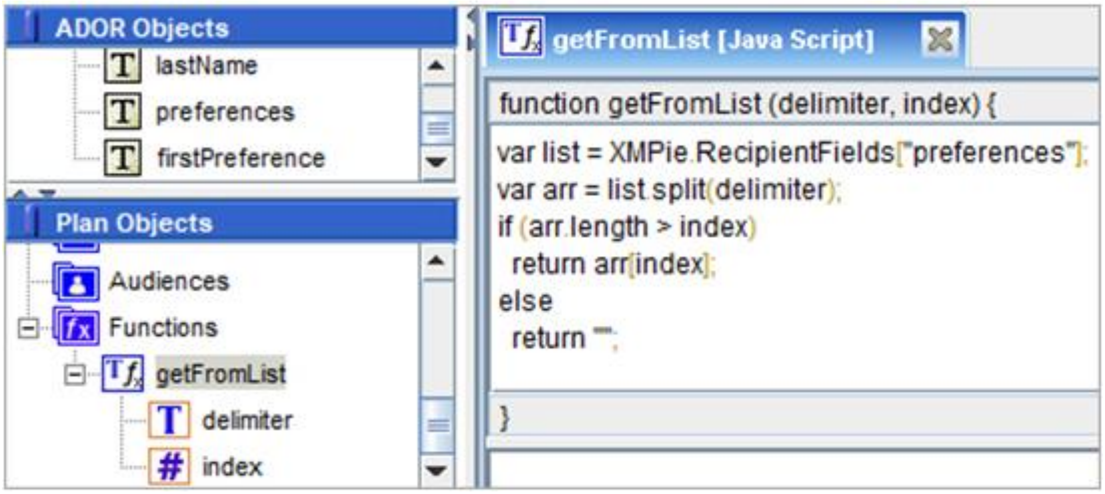
```
    XMPie.Functions.UCase(XMPie.RecipientFields["firstname"])
```

Would return, in uppercase, the value from the "firstname" field of database for the current recipient record.

Many of the uPlan built-in functions have similar functions in JavaScript. For example uPlan's UCase(value) function is the same as JavaScript's value.toUpperCase() function. However, many others like GetEnv() are unique to uPlan and offer functionality that you might want to use in your custom JavaScript functions.

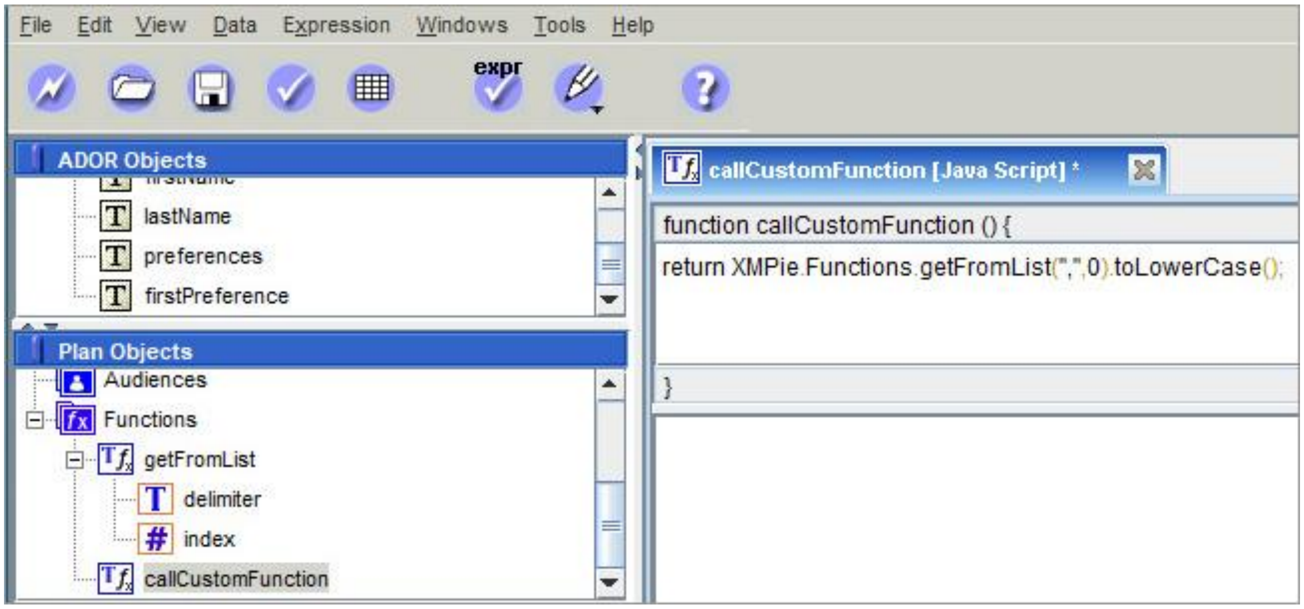
In addition to accessing the built-in uPlan functions, it is also possible for JavaScript functions to access other custom functions in the same way.

For example, if we use the getFromList() function example we created earlier:



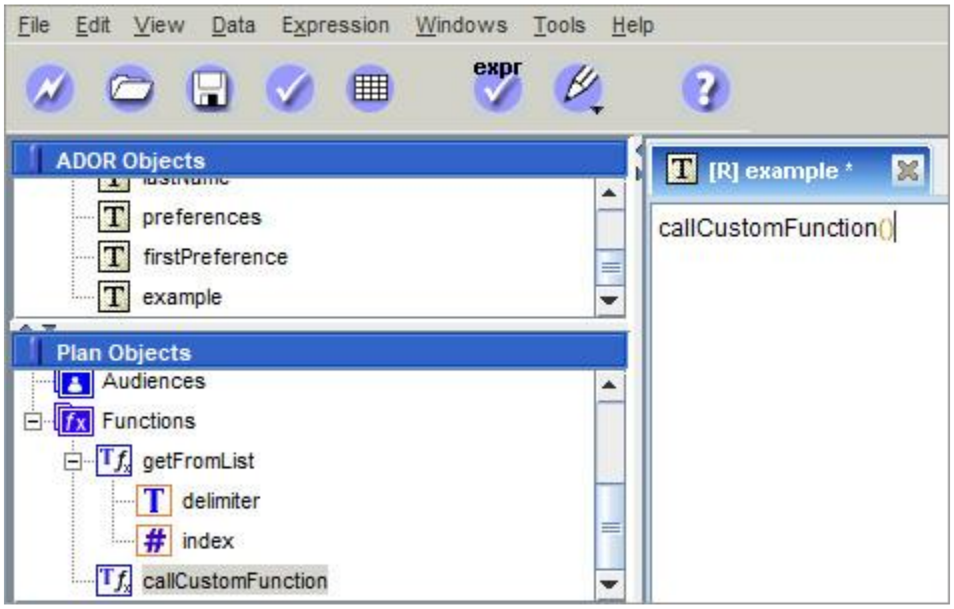
```
function getFromList (delimiter, index) {
var list = XMPie.RecipientFields["preferences"];
var arr = list.split(delimiter);
if (arr.length > index)
  return arr[index];
else
  return "";
}
```

Another JavaScript function can call the getFromList() function and do something else with the value – converting it to lowercase in this example:



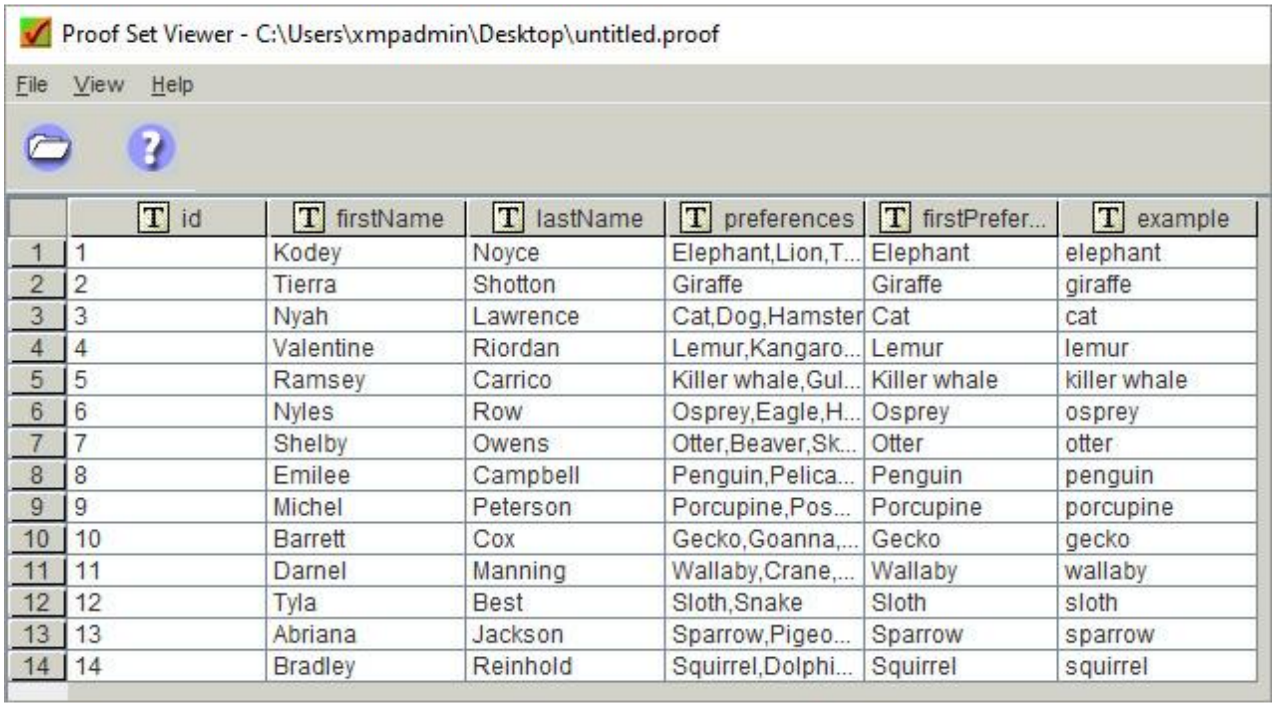
```
function callCustomFunction () {
return XMPie.Functions.getFromList(",",0).toLowerCase();
}
```

This new function can be used in an ADOR expression:



```
callCustomFunction()
```

And the result can be seen in a proofset:



	id	firstName	lastName	preferences	firstPrefer...	example
1	1	Kodey	Noyce	Elephant,Lion,T...	Elephant	elephant
2	2	Tierra	Shotton	Giraffe	Giraffe	giraffe
3	3	Nyah	Lawrence	Cat,Dog,Hamster	Cat	cat
4	4	Valentine	Riordan	Lemur,Kangaro...	Lemur	lemur
5	5	Ramsey	Carrico	Killer whale,Gul...	Killer whale	killer whale
6	6	Nyles	Row	Osprey,Eagle,H...	Osprey	osprey
7	7	Shelby	Owens	Otter,Beaver,Sk...	Otter	otter
8	8	Emilee	Campbell	Penguin,Pelica...	Penguin	penguin
9	9	Michel	Peterson	Porcupine,Pos...	Porcupine	porcupine
10	10	Barrett	Cox	Gecko,Goanna,...	Gecko	gecko
11	11	Darnel	Manning	Wallaby,Crane,...	Wallaby	wallaby
12	12	Tyla	Best	Sloth,Snake	Sloth	sloth
13	13	Abriana	Jackson	Sparrow,Pigeo...	Sparrow	sparrow
14	14	Bradley	Reinhold	Squirrel,Dolphi...	Squirrel	squirrel