CTL

This chapter describes the syntax and the use of CTL. For detailed information on language reference or built-in functions see:

- Language Reference
- Functions Reference

Example 55.1. Example of CTL syntax (Rollup)

```
//#CTL  string[] customers; integer Length;   function void
initGroup(VoidMetadata groupAccumulator) { }  function boolean
updateGroup(VoidMetadata groupAccumulator) {      customers =
split($0.customers," - ");      Length = length(customers);
return true; }  function boolean finishGroup(VoidMetadata
groupAccumulator) {      return true; }  function integer
updateTransform(integer counter, VoidMetadata groupAccumulator) {
if (counter >= Length) {           clear(customers);             return
SKIP;      }       $0.customers = customers[counter];
$0.EmployeeID = $0.EmployeeID;       return ALL; }  function integer
transform(integer counter, VoidMetadata groupAccumulator) {      return
ALL; }
```

Language Reference

This section describes the following areas:

- Program Structure
- Comments
- Import **Where?**
- Data Types in CTL
- Literals
- Variables
- Dictionary in CTL
- Operators
- Simple Statement and Block of Statements
- Control Statements
- Functions
- Conditional Fail Expression
- Accessing Data Records and Fields
- Mapping
- Parameters

Program Structure

Each program written in CTL must contain the following parts:

```
ImportStatements VariableDeclarations FunctionDeclarations Statements
Mappings
```

All of them may be interspersed, however there are some principles that are valid for them:

- If an import statement is defined, it must be situated at the beginning of the code.
- Variables and functions must first be declared and only then they can be used.
- Declarations of variables and functions, statements, and mappings may also be mutually interspersed.

Important
In CTL declaration of variables and functions may be in any place of the transformation code and may be preceded by other code. However, remember that each variable and each function must always be declared before it is used.

Comments

Throughout the program you can use comments. These comments are not processed, they only serve to describe what happens within the program.

There are two types of comments. They can be one-line comments or multiline comments. See the following two options:

- `// This is an one-line comment.`
- `/* This is a multiline comment. */`

Data Types in CTL

For basic information about data types used in metadata see Data Types and Record Types. **What? (see where?)**

In any program, you can use some variables. Data types in CTL are the following:

boolean

Its declaration look like this: `boolean` ***`identifier`***`;`

byte

This data type is an array of bytes of a length that can be up to `Integer.MAX_VALUE` as a maximum. It behaves similarly to the list data type (see below).

Its declaration looks like this: `byte` ***`identifier`***`;`

cbyte

This data type is a compressed array of bytes of a length that can be up to `Integer.MAX_VALUE` as a maximum. It behaves similarly to the list data type (see below).

Its declaration looks like this: `cbyte` ***`identifier`***`;`

date

Its declaration look like this: `date` *`identifier`*`;`

decimal

Its declaration looks like this: `decimal` *`identifier`*`;`

By default, any decimal may have up to 32 significant digits. If you want to have different **Length** or **Scale**, you need to set these properties of `decimal` field in metadata.

Example 55.2. Example of usage of decimal data type in CTL

If you assign 100.0 /3 to a decimal variable, its value might for example be `33.333333333333335701809119200333`. Assigning it to a decimal field (with default **Length** and **Scale**, which are 8 and 2, respectively), it will be converted to `33.33D`.

You can cast any float number to the decimal data type by apending the `d` letter to its end.

integer

Its declaration looks like this: `integer` *`identifier`*`;`

If you apend a `l` letter to the end of any integer number, you can cast it to the long data type.

long

Its declaration looks like this: `long` *`identifier`*`;`

Any integer number can be cast to this data type by appending a `l` letter to its end.

number (double)

Its declaration looks like this: `number` *`identifier`*`;`

string

Its declaration looks like this: `string` *`identifier`*`;`

list

Each `list` is a container of one the following data types: `boolean`, `byte`, `date`, `decimal`, `integer`, `long`, `number`, `string`, or `record`.

The list data type is indexed by integers starting from 0.

Its declaration can look like this: `string[ ]` *`identifier`*`;`

Lists cannot be created as a list of `lists` or `maps`.

The default list is an empty list.

Examples:

```
integer[] myIntegerList; myIntegerList[5] = 123;
```

```
Customer JohnSmith;
```

```
Customer PeterBrown;
```

```
Customer[] CompanyCustomers;
```

```
CompanyCustomers[0] = JohnSmith;
```

```
CompanyCustomers[1] = PeterBrown
```

Assignments:

- `myStringList[3] = "abc";`

  It means that the specified string is put to the fourth position in the string list. The other values are fille with `null` as follows:

  `myStringList` is `[null,null,null,"abc"]`

- `myList1 = myList2;`

  It means that both lists reference the same elements.

- `myList1 = myList1 + myList2;`

  It adds all elements of `myList2` to the end of `myList1`.

  Both lists must be based on the same primitive data type.

- `myList1 = myList1 + "abc";`

  It adds the `"abc"`string to the `myList1` as its new last element.

  `myList1` must be based on string data type.

- `myList1 = null;`

  It destroys the `myList1`.

Be careful when performing list operations (such as append). See Warning.

map

This data type is a container of pairs of a key and a value.

Its declaration looks like this: `map[<type of key>, <type of value>] identifier;`

Both the `Key` and the `Value` can be of the following primitive data types: `boolean`, `byte`, `date`, `decimal`, `integer`, `long`, `number`, `string`. `Value` can be also of `record` type.

Map cannot be created as a map of `lists` or other `maps`.

The default map is an empty map.

Examples:

```
map[string, boolean] map1; map1["abc"]=true;

Customer JohnSmith;

Customer PeterBrown;

map[integer, Customer] CompanyCustomersMap;

CompanyCustomersMap[JohnSmith.ID] = JohnSmith;

CompanyCustomersMap[PeterBrown.ID] = PeterBrown
```

The assignments are similar to those valid for a list.

Literals

Literals serve to write values of any data type.

Table 55.1. Literals

Literal	Description	Declaration syntax	Example
integer	digits representing integer number	[0-9]+	95623
long integer	digits representing integer number with absolute value even greater than 2^{31}, but less than 2^{63}	[0-9]+L?	257L, or 9562307813123123
hexadecimal integer	digits and letters representing integer number in hexadecimal form	0x[0-9A-F]+	0xA7B0
octal integer	digits representing integer number in octal form	0[0-7]*	0644
number (double)	floating point number represented by 64bits in double precision format	[0-9]+.[0-9]+	456.123

Literal	Description	Declaration syntax	Example
decimal	digits representing a decimal number	[0-9]+.[0-9]+D	123.456D
double quoted string	string value/literal enclosed in double quotes; escaped characters [\n,\r,\t, \\, \", \b] get translated into corresponding control chars	"...anything except ["]..."	"hello\tworld\n\r"
single quoted string	string value/literal enclosed in single quotes; only one escaped character [\'] gets translated into corresponding char [']	'...anything except [']...'	'hello\tworld\n\r'
list of literals	list of literals where individual literals can also be other lists/maps/records	[<any literal> (, <any literal>)*]	[10, 'hello', "world", 0x1A, 2008-01-01], [[1 , 2]] , [3 , 4]]
date	date value	this mask is expected: yyyy-MM-dd	2008-01-01
datetime	datetime value	this mask is expected: yyyy-MM-dd HH:mm:ss	2008-01-01 18:55:00

Important

You cannot use any literal for `byte` data type. If you want to write a `byte` value, you must use any of the conversion functions that return `byte` and aply it on an argument value.

For information on these conversion functions see Conversion Functions

Important

Remember that if you need to assign decimal value to a decimal field, you should use decimal literal. Otherwise, such a number would not be decimal, it would be a double number!

For example:

1. **Decimal value to a decimal field (correct and accurate)**

   ```
   // correct - assign decimal value to decimal field

   myRecord.decimalField = 123.56d;
   ```

2. **Double value to a decimal field (possibly inaccurate)**

   ```
   // possibly inaccurate - assign double value to decimal
   ```

```
        field

        myRecord.decimalField = 123.56;
```

The latter might produce inaccurate results!

Variables

If you define some variable, you must do it by typing the data type of the variable, white space, the name of the variable, and a semicolon.

Such a variable can be initialized later, but it can also be initialized in the declaration itself. Of course, the value of the expression must be of the same data type as the variable.

Both cases of variable declaration and initialization are shown below:

-
  ```
  dataType variable;
  ...
  variable = expression;
  ```
- `dataType variable = expression;`

Dictionary in CTL

If you want to have a dictionary in your graph and access an entry from CTL, you must define it in the graph as shown in **What? (no link/table)**

To access the entries from CTL, use the dot syntax as follows:

```
dictionary.<dictionary entry>
```

This expression can be used to

- define the value of the entry:

  ```
  dictionary.customer = "John Smith";
  ```

- get the value of the entry:

  ```
  myCustomer = dictionary.customer;
  ```

- map the value of the entry to an output field:

  ```
  $0.myCustomerField = dictionary.customer;
  ```

- serve as the argument of a function:

  ```
  myCustomerID = isInteger(dictionary.customer);
  ```

Operators

The operators serve to create more complicated expressions within the program. They can be arithmetic, relational, and logical. The relational and logical operators serve to create expressions with resulting boolean value. The arithmetic operators can be used in all expressions, not only the logical ones.

All operators can be grouped into three categories:

- Arithmetic Operators
- Relational Operators
- Logical Operators

Arithmetic Operators

The following operators serve to put together values of different expressions (except those of boolean values). These signs can be used more times in one expression. In such a case, you can express priority of operations by parentheses. The result depends on the order of the expressions.

- Addition

  +

  The operator above serves to sum the values of two expressions.

  But the addition of two boolean values or two date data types is not possible. To create a new value from two boolean values, you must use logical operators instead.

  Nevertheless, if you want to add any data type to a string, the second data type is converted to a string automatically and it is concatenated with the first (string) summand. But remember that the string must be on the first place **What?**! Naturally, two strings can be summed in the same way. Note also that the `concat()` function is faster, and you should use this function instead of adding any summand to a string.

  You can also add any numeric data type to a date. The result is a date in which the number of days is increased by the whole part of the number. Again, here is also necessary to have the date on the first place.

  The sum of two numeric data types depends on the order of the data types. The resulting data type is the same as that of the first summand. The second summand is converted to the first data type automatically.

- Subtraction and Unitary minus

  -

  The operator serves to subtract one numeric data type from another. Again the resulting data type is the same as that of the minuend. The subtrahend is converted to the minuend data type automatically.

  But it can also serve to subtract numeric data type from a date data type. The result is a date in which the number of days is reduced by the whole part of the subtrahend.

- Multiplication

  `*`

  The operator serves only to multiplicate two numeric data types.

  Remember that during multiplication the first multiplicand determines the resulting data type of the operation. If the first multiplicand is an integer number and the second is a decimal, the result will be an integer number. On the other hand, if the first multiplicand is a decimal and the second is an integer number, the result will be a decimal data type. In other words, the order of multiplicands is important.

- Division

  `/`

  The operator serves only to divide two numeric data types. Remember that you must not divide by zero. Dividing by zero throws `TransformLangExecutorRuntimeException` or gives `Infinity` (in case of a number data type).

  Remember that during division the numerator determines the resulting data type of the operation. If the nominator is an integer number and the denominator is a decimal, the result will be an integer number. On the other hand, if the nominator is a decimal and the denominator is an integer number, the result will be of decimal data type. In other words, data types of nominator and denominator are of importance.

- Modulus

  `%`

  The operator can be used for both floating-point data types and integer data types. It returns the remainder of division.

- Incrementing

  `++`

  The operator serves to increment numeric data type by one. The operator can be used for both floating-point data types and integer data types.

  If it is used as a prefix, the number is incremented first and then it is used in the expression.

  If it is used as a postfix, first, the number is used in the expression and then it is incremented.

  Important
  Remember that the incrementing operator cannot be applied on literals, record fields, map, or list values of integer data type. It can only be used with integer variables.

- Decrementing

  `--`

The operator serves to decrement numeric data type by one. The operator can be used for both floating-point data types and integer data types.

If it is used as a prefix, the number is decremented first, and then used in the expression.

If it is used as a postfix, the number is first used in the expression, and then decremented.

Important

Remember that the decrementing operator cannot be applied on literals, record fields, map, or list values of integer data type. It can only be used with integer variables.

Relational Operators

The following operators serve to compare some subexpressions when you want to obtain a boolean value result. Each of the mentioned signs can be used. If you choose the `.operator.` signs, they must be surrounded by white spaces. These signs can be used more than once in one expression. In such a case you can express priority of comparisons by parentheses.

- Greater than

  Each of the two signs below can be used to compare expressions consisting of numeric, date, and string data types. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data type.

  - `>`
  - `.gt.`

- Greater than or equal to

  Each of the three signs below can be used to compare expressions consisting of numeric, date, and string data types. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data type.

  - `>=`
  - `=>`
  - `.ge.`

- Less than

  Each of the two signs below can be used to compare expressions consisting of numeric, date, and string data types. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data type.

  - `<`
  - `.lt.`

- Less than or equal to

Each of the three signs below can be used to compare expressions consisting of numeric, date, and string data types. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data type.

  - `<=`
  - `=<`
  - `.le.`

- Equal to

  Each of the two signs below can be used to compare expressions of any data type. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data type.

  - `==`
  - `.eq.`

- Not equal to

  Each of the three signs below can be used to compare expressions of any data type. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data type.

  - `!=`
  - `<>`
  - `.ne.`

- Matches regular expression

  The operator serves to compare string and some regular expression. It returns `true` if the whole string matches the regular expression, otherwise returns `false`.

  - `~=`
  - `.regex.`

- Contains regular expression

  The operator serves to compare string and some regular expression. It returns `true` if the string contains a substring that matches the regular expression, otherwise returns `false`.

  - `?=`

- Contained in

  This operator serves to specify whether some value is contained in the list or in the map of other values.

The operator has double syntax. See following examples:

```
1.      boolean myBool; string myString;  string[] = myList; ... myBool
= myString.in(myList);
2.      boolean myBool; string myString;  string[] = myList; ... myBool
= in(myString,myList);
```

Logical Operators

If the expression whose value must be of boolean data type is complicated, it can consist of some subexpressions (see above) that are put together by logical conjunctions (AND, OR, NOT, EQUAL TO, NOT EQUAL TO). If you want to express priority in such an expression, you can use parentheses. From the conjunctions mentioned below you can choose either form (for example, `&&` or `and`, etc.).

Every sign of the form `.operator.` must be surrounded by white space.

- Logical AND
  - `&&`
  - `and`
- Logical OR
  - `||`
  - `or`
- Logical NOT
  - `!`
  - `not`
- Logical EQUAL TO
  - `==`
  - `.eq.`
- Logical NOT EQUAL TO
  - `!=`
  - `<>`
  - `.ne.`

Assignment Operator

Example 55.3. Clearing a copied list

```
        integer[] list1 = [1, 2, 3];
        integer[] list2;
        list2 = list1;
        list1.clear(); //  only list1 is cleared (old implementation:
list2 was cleared, too)
```

Simple Statement and Block of Statements

All statements can be divided into two groups:

- **Simple statement** is an expression terminated by semicolon.

  For example:

  ```
  integer MyVariable;
  ```

- **Block of statements** is a series of simple statements (each of them is terminated by semicolon). The statements in a block can follow each other in one line or they can be written in more lines. They are surrounded by curled braces. No semicolon is used after the closing curled brace.

  For example:

  ```
  while (MyInteger<100) {      Sum = Sum + MyInteger;
  MyInteger++;      }
  ```

Control Statements

Some statements serve to control the process of the program.

All control statements can be grouped into the following categories:

- Conditional Statements
- Iteration Statements
- Jump Statements

Conditional Statements

These statements serve to branch out the process of the program.

If Statement

On the basis of the `Condition` value this statement decides whether the `Statement` should be executed. If the `Condition` is true, `Statement` is executed. If it is false, the `Statement` is ignored and process continues next after the `if` statement. `Statement` is either simple statement or a block of statements

- `if (Condition) Statement`

Unlike the previous version of the `if` statement (in which the `Statement` is executed only if the `Condition` is true), other `Statements` that should be executed even if the `Condition` value is false can be added to the `if` statement. Thus, if the `Condition` is true, `Statement1` is executed, if it is false, `Statement2` is executed. See below:

- `if (Condition) Statement1 else Statement2`

The `Statement2` can even be another `if` statement and also with `else` branch:

- `if (Condition1) Statement1      else if (Condition2) Statement3 else Statement4`

Switch Statement

Sometimes you would have a very complicated statement if you created the statement out of more branched out `if` statements. In such a case, it is better to use the `switch` statement.

Now, instead of the `Condition` as in the `if` statement with only two values (true or false), an `Expression` is evaluated and its value is compared with the `Constants` specified in the `switch` statement.

Only the `Constant` that equals to the value of the `Expression` decides which of the `Statements` is executed.

If the `Expression` value is `Constant1`, the `Statement1` will be executed, etc.

Important

Remember that literals must be unique in the `Switch statement`.

- `switch (Expression) {      case Constant1 : Statement1 StatementA [break;]      case Constant2 : Statement2 StatementB [break;]      ...      case ConstantN : StatementN StatementW [break;] }`

The optional `break;` statements ensure that only the statements correspoding to a constant will be executed. Otherwise, all below them would be executed as well.

In the following case, even if the value of the `Expression` does not equal to the values of the `Constant1,...,ConstantN`, the default statement (`StatementN+1`) is executed.

- `switch (Expression) {      case Constant1 : Statement1 StatementA [break;]      case Constant2 : Statement2 StatementB [break;]      ...      case ConstantN : StatementN StatementW [break;]      default : StatementN+1 StatementZ }`

Iteration Statements

These iteration statements repeat some processes during which some inner `Statements` are executed cyclically until the `Condition` that limits the execution cycle becomes false or they are executed for all values of the same data type.

For Loop

First, the Initialization is set up. After that, the `Condition` is evaluated, and if its value is true, the `Statement` is executed and finally the `Iteration` is made.

During the next cycle of the loop, the `Condition` is evaluated again. If it's true, the `Statement` is executed and an `Iteration` is made. This process repeats until the `Condition` becomes false. The loop is then terminated and the process continues with the other part of the program.

If the `Condition` is false at the beginning, the process jumps over the `Statement` out of the loop.

- `for (Initialization;Condition;Iteration)      Statement`

Important

Remember that the `Initialization` part of the `For Loop` may also contain the declaration of the variable that is used in the loop.

`Initialization`, `Condition`, and `Iteration` are optional.

Do-While Loop

First, the `Statement` is executed; then, the process depends on the value of the `Condition`. If its value is true, the `Statement` is executed again, and the `Condition` is then evaluated again and the subprocess either continues (if it is true again) or stops and jumps to the next or higher level subprocesses (if it is false). Since the `Condition` is at the end of the loop, even if it is false at the beginning of the subprocess, the `Statement` is executed at least once.

- `do Statement while (Condition)`

While Loop

This process depends on the value of `Condition`. If its value is true, the `Statement` is executed; then, the `Condition` is evaluated again and the subprocess either continues (if it is true again), or stops and jumps to the next or higher level subprocesses (if it is false). Since the `Condition` is at the start of the loop, if it is false at the beginning of the subprocess, the `Statement` is not executed at all and the loop is jumped over.

- `while (Condition) Statement`

For-Each Loop

The `foreach` statement is executed on all fields of the same data type within a container. Its syntax is as follows:

- `foreach (<data type> myVariable : iterableVariable) Statement`

All elements of the same data type (data type is declared in this statement) are searched in the `iterableVariable` container. The `iterableVariable` can be a list or a record. For each variable of the same data type, specified `Statement` is executed. It can be either a simple statement or a block of statements.

Thus, for example, the same `Statement` can be executed for all `string` fields of a record, etc.

Note

It is possible to iterate over *values* of a map (i.e. not whole `<entries>`). Remember the type of the loop variable has to match the type of map's values:

```
                              map[string, integer] myMap;
        myMap["first"] = 1; myMap["second"] = 2;
        foreach(integer value: myMap) {   printErr(value); //
prints 1 and 2 }
```

To obtain map's keys as a `list[]`, use the getKeys() function.

Jump Statements

Sometimes you need to control the process in a different way than by a decision based on the `Condition` value. To do that, you have the following options:

Break Statement

If you want to stop some subprocess, you can use the following statement in the program:

- `break;`

The subprocess breaks and the process jumps to the higher level or to the next `Statement`.

Continue Statement

If you want to stop some iteration subprocess, you can use the following statement in the program:

- `continue;`

The subprocess breaks and the process jumps to the next iteration step.

Return Statement

In the functions you can use the `return` word either alone or along with an `expression`. (See the following two options below.) The `return` statement can be in any place within the function. There may also be multiple `return` statements among which a specific one is executed depending on a condition, etc.

- `return;`
- `return expression;`

Error Handling

CTL also provides a simple mechanism for catching and handling possible errors.

It only uses a set of optional `OnError()` functions that exist to each required transformation function.

For example, for required functions (e.g., `append()`, `transform()`, etc.), there exist the following optional functions:

`appendOnError()`, `transformOnError()`, etc.

Each of these required functions may have its (optional) couterpart whose name differs from the original (required) by adding the `OnError` suffix.

Moreover, every `<required ctl template function>OnError()` function returns the same values as the original required function.

This way, any exception that is thrown by the original required function causes call of its `<required ctl template function>OnError()` counterpart (e.g., `transform()` fail may call `transformOnError()`, etc.).

In this `transformOnError()`, any incorrect code can be fixed, an error message can be printed to Console, etc.

Important

Remember that these `OnError()` functions are not called when the original required functions return **Error codes** (values less then -1)! If you want that some `OnError()` function is called, you need to use a `raiseError(string arg)` function. Or also, any exception thrown by original required function calls its `OnError()` counterpart.

Functions

You can define your own functions in the following way:

```
function returnType functionName (type1 arg1, type2 arg2,..., typeN
argN) {
variableDeclarations
    otherFunctionDeclarations
Statements     Mappings
return [expression];
}
```

You must put the return statement at the end. For more information about the return statement see Return Statement. Inside some functions, there can be `Mappings`. These may be in any place inside the function.

In addition to any other data type mentioned above, the function can also return `void`.

Message Function

```
function string getMessage() {          return message; }
```

This `message` variable should be declared as a global string variable and defined anywhere in the code so as to be used in the place where the `getMessage()` function is located. The `message` will be written to console.

Conditional Fail Expression

You can also use conditional fail expressions.

They look like this:

```
expression1 : expression2 : expression3 : ... : expressionN;
```

This conditional fail expression may be used for mapping, assignment to a variable, and as an argument of a function too.

The expressions are evaluated one by one, starting from the first expression and going from left to right.

1. As soon as one of these expressions may be successfully assigned to a variable, mapped to an output field, or used as the argument of the function, it is used and the other expressions are not evaluated.

2. If none of these expressions may be used (assigned to a variable, mapped to the output field, or used as an argument), graph fails.

Important

Remember that in CTL this expression may be used in multiple ways: for assigning to a variable, mapping to an output field, or as an argument of the function.

Remember also that this expression can only be used in an interpreted mode of CTL.

Accessing Data Records and Fields

This section describes the way that the record fields should be worked with. As you know, each component may have ports. Both input and output ports are numbered starting from 0.

Metadata of connected edges must be identified by their names. Different metadata must have different names.

Working with Records and Variables

Important

Since v. 3.2, the syntax has changed to:

`$in.portID.fieldID` and `$out.portID.fieldID`

e.g. `$in.0.* = $out.0.*;`

That way, you can clearly distinguish input and output metadata.

Transformations you have written before will be compatible with the old syntax.

Now we suppose that `Customers` is the ID of metadata, their name is `customers`, and their third field (field 2) is `firstname`.

Following expressions represent the value of the third field (field 2) of the specified metadata:

- `$<port number>.<field number>`

  Example: `$0.2`

  `$0.*` means all fields on the first port (port 0).

- `$<port number>.<field name>`

  Example: `$0.firstname`

- `$<metadata name>.<field number>`

  Example: `$customers.2`

  `$customers.*` means all fields on the first port (port 0).

- `$<metadata name>.<field name>`

  Example: `$customers.firstname`

You can also define records in CTL code. Such defitions can look like these:

- `<metadata name> MyCTLRecord;`

  Example: `customers myCustomers;`

- After that, you can use the following expressions:

  `<record variable name>.<field name>`

  Example: `myCustomers.firstname;`

Mapping of records to variables looks like this:

- `myVariable = $<port number>.<field number>;`

  Example: `FirstName = $0.2;`

- `myVariable = $<port number>.<field name>;`

  Example: `FirstName = $0.firstname;`

- `myVariable = $<metadata name>.<field number>;`

  Example: `FirstName = $customers.2;`

- `myVariable = $<metadata name>.<field name>;`

  Example: `FirstName = $customers.firstname;`

- `myVariable = <record variable name>.<field name>;`

  Example: `FirstName = myCustomers.firstname;`

Mapping of variables to records can look like this:

- `$<port number>.<field number> = myVariable;`

  Example: `$0.2 = FirstName;`

- `$<port number>.<field name> = myVariable;`

  Example: `$0.firstname = FirstName;`

- `$<metadata name>.<field number> = myVariable;`

Example: `$customers.2 = FirstName;`

- `$<metadata name>.<field name> = myVariable;`

  Example: `$customers.firstname = FirstName;`

- `<record variable name>.<field name> = myVariable;`

  Example: `myCustomers.firstname = FirstName;`

Important

Remember that if component has single input port or single output port, you can use the syntax as follows:

```
$firstname
```

Generally, the syntax is:

```
$<field name>
```

Important

You can assign input to an internal CTL record using following syntax:

```
MyCTLRecord.* = $0.*;
```

Also, you can map values of an internal record to the output using following syntax:

```
$0.* = MyCTLRecord.*;
```

Mapping

Calculated or generated values—or values of input fields—are assigned (mapped) to output fields.

1. Mapping assigns a value to an output field.
2. Mapping operator is the following:

   ```
   =
   ```

3. Mapping must always be defined inside a function.
4. Mapping may be defined in any place inside a function.
5. Remember that you can also wrap a mapping in a user-defined function, which would be subsequently used inside another function.
6. You can also map different input metadata to different output metadata by field names or by field positions. See examples below.

Mapping of Different Metadata (by Name)

When you map input to output like this:

```
$0.* = $0.*;
```

input metadata may even differ from those on the output.

In the expression above, fields of the input are mapped to the fields of the output that have the same name and type as those of the input. The order in which they are contained in respective metadata and the number of all fields in either metadata is of no importance.

When you have input metadata in which the first two fields are `firstname` and `lastname`, each of these two fields is mapped to its counterpart on the output. Such output `firstname` field may even be the fifth and `lastname` field be the third, but those two fields of the input will be mapped to these two output fields.

Even if input and output metadata had more fields, such fields would not be mapped to each other if there did not exist an output field with the same name as one of the input fields (independently on the mutual position of the fields in corresponding metadata).

In addition to the simple mapping as shown above (`$0.* = $0.*;`) you can also use the following function:

`void` **`copyByName`**`(`record *to*, record *from*`);`

Example 55.4. Mapping of Metadata by Name (using the copyByName() function)

```
recordName2 myOutputRecord; copyByName(myOutputRecord.*,$0.*); $0.* =
myOutputRecord.*;
```

Important

Metadata fields are mapped from input to output by name and data type independently on their order and on the number of all fields!

The following syntax may also be used: `myOutputRecord.copyByName($0.*);`

Mapping of Different Metadata (by Position)

Sometimes you need to map input to ouput, but names of input fields are different from those of output fields. In such a case, you can map input to output by position.

To achieve this, you *must* use the following function:

`void` **`copyByPosition`**`(`record *to*, record *from*`);`

Example 55.5. Mapping of Metadata by Position

```
recordName2 myOutputRecord; copyByPosition(myOutputRecord,$0.*); $0.* =
myOutputRecord.*;
```

Important

Metadata fields may be mapped from input to output by position (as shown in the example above)!

Following syntax may also be used: `myOutputRecord.copyByPosition($0.*);`

Use Case 1 - One String Field to Upper Case

To show in more details how mapping works, here are a few examples of mappings.

We have a graph with a **Reformat** component. Metadata on its input and output are identical. First two fields (`field1` and `field2`) are of string data type, the third (`field3`) is of integer data type.

1. We want to change the letters of `field1` values to upper case while passing the other two fields unchanged to the output.
2. We also want to distribute records according to the value of `field3`. Those records in which the value of `field3` is less than 5 should be sent to the output port 0, the others to the output port 1.

Examples of Mapping

The first possibility is having the mapping for both ports and all fields defined inside the `transform()` function of CTL template.

Example 55.6. Example of Mapping with Individual Fields

Note that the mappings will be performed for all records. In other words, even when the record will go to the output port 1, also the mapping for output port 0 will be performed, and vice versa.

Moreover, mapping consists of individual fields, which may be complicated in case there are many fields in a record. In the next examples, we will see how this can be solved in a better way.

```
function integer transform() {
   // mapping input port records to output port records
// each field is mapped separately
 $0.field1 = upperCase($0.field1);
   $0.field2 = $0.field2;
    $0.field3 = $0.field3;
              $1.field1 = upperCase($0.field1);
              $1.field2 = $0.field2;
     $1.field3 = $0.field3;
     // output port number returned
    if ($0.field3 < 5)
     return 0;
   else
     return 1;
 }
```

Note

As CTL allows to use any code *after* the mapping, here we have used the `if` statement with two `return` statements after the mapping.

In CTL mapping may be in any place of the transformation code and may be followed by any code!

As the second possibility, we also have the mapping for both ports and all fields defined inside the `transform()` function of CTL template. But now there are wild cards used in the mapping. These passes the records unchanged to the outputs and after this wildcard mapping the fields that should be changed are specified.

Example 55.7. Example of Mapping with Wild Cards

Note that mappings will be performed for all records. In other words, even when the record will go to the output port 1, the mapping for output port 0 will also be performed, and vice versa.

However, the mapping now uses wild cards first, which passes the records unchanged to the output. The first field is now changed *below* the mapping with wild cards.

This is useful when there are many unchanged fields and a few that will be changed.

```
function integer transform() {
     // mapping input port records to output port records
   // wild cards for mapping unchanged records
    // transformed records mapped aditionally
   $0.* = $0.*;
   $0.field1 = upperCase($0.field1);
    $1.* = $0.*;
   $1.field1 = upperCase($0.field1);
   // return the number of output port
     if ($0.field3 < 5)
         return 0;
     else
         return 1;
}
```

Note

As CTL allows to use any code *after* the mapping, here we have used the `if` statement with two `return` statements after the mapping.

In CTL mapping may be in any place of the transformation code and may be followed by any code!

As the third possibility, we have the mapping for both ports and all fields defined outside the `transform()` function of CTL template. Each output port has its own mapping.

Wild cards are also used here.

The mapping that is defined in separate functions for each output port allows the following improvements:

- Mapping is performed only for the respective output port! In other words, there is no need to map record to the port 1 when it will go to the port 0, and vice versa.

Example 55.8. Example of Mapping with Wild Cards in Separate User-Defined Functions

Moreover, mapping uses wild cards at first, which passes the records unchanged to the output, but the first field is changed below the mapping with wild card. This useful when there are many unchanged fields and a few that will be changed.

```
          // mapping input port records to output port records  //
inside separate functions // wild cards for mapping unchanged records
// transformed records mapped aditionally function void mapToPort0 () {
$0.* = $0.*;      $0.field1 = upperCase($0.field1); }  function void
mapToPort1 () {       $1.* = $0.*;      $1.field1 =
upperCase($0.field1); }  // use mapping functions for all ports in the
if statement function integer transform() {      if ($0.field3 < 5) {
mapToPort0();            return 0;      }      else {
mapToPort1();           return 1;      } }
```

Parameters

The parameters can be used in Clover transformation language in the following way: `${nameOfTheParameter}`. If you want such a parameter to be considered a string data type, you must surround it by single or double quotes like this: `'${nameOfTheParameter}'` or `"${nameOfTheParameter}"`.

Important

1. Remember that escape sequences are always resolved as soon as they are assigned to parameters. For this reason, if you want that they are not resolved, type double backslashes in these strings instead of single ones.
2. Remember also that you can get the values of environment variables using parameters. To learn how to do it, see Environment Variables.

Functions Reference

Clover transformation language has at its disposal a set of functions you can use. We describe them here.

All functions can be grouped into following categories:

- Conversion Functions
- Date Functions
- Mathematical Functions
- String Functions
- Container Functions
- Record functions (dynamic field access)
- Miscellaneous Functions
- Lookup Table Functions
- Sequence Functions

Built-in functions

- `substring(upperCase(getAplhanumericChars($0.field1))1,3)`
- `$0.field1.getAlphanumericChars().upperCase().substring(1,3)`

The two expressions above are equivalent. The second option with the first argument preceding the function itself is sometimes referred to as **object notation**. Do not forget to use the "$port.field.function()" syntax. Thus, `arg.substring(1,3)` is equal to `substring(arg,1,3).`

You can also declare your own function with a set of arguments of any data type, ex.:

```
function integer myFunction(integer arg1, string arg2, boolean
arg3) {  <function body> }
```

User-defined functions

- `myFunction($0.integerField,$0.stringField,$0.booleanField)`
- `$0.integerField.myFunction($0.stringField,$0.booleanField)`

Warning

Remember that the object notation (<first argument>.function(<other arguments>) cannot be used in **Miscellaneous** functions. See Miscellaneous Functions.

Important

Remember that if you set the **Null value** property in metadata for any `string` data field to any non-empty string, any function that accept `string` data field as an argument and throws NPE when applied on `null` (e.g., `length()`), it will throw NPE when applied on such specific string.

For example, if `field1` has **Null value** property set to `"<null>"`, `length($0.field1)` will fail on the records in which the value of `field1` is `"<null>"` and it will be 0 for empty field.

See Null value for detailed information.

Conversion Functions

Sometimes you need to convert values from one data type to another.

In the functions that convert one data type to another, sometimes a format pattern of a date or a number must be defined. Also, locale can have an influence to their formatting.

- For detailed information about date formatting and/or parsing see Data and Time Format. **What? (where are these?)**
- For detailed information about formatting and/or parsing of any numeric data type see Numeric Format.
- For detailed information about locale see Locale.

Note

Remember that numeric and date formats are displayed using system value **Locale** or **Locale** specified in the `defaultProperties` file, unless other **Locale** is explicitly specified.

Here we provide the list of these functions:

- `byte` **`base64byte`**`(`string *arg*`);`

  The `base64byte(string)` function takes one string argument in `base64` representation and converts it to an array of bytes. Its counterpart is the `byte2base64(byte)` function.

- `string` **`bits2str`**`(`byte *arg*`);`

  The `bits2str(byte)` function takes an array of bytes and converts it to a string consisting of two characters: `"0"` or `"1"`. Each byte is represented by eight characters ("0" or "1"). For each byte, the lowest bit is at the beginning of these eight characters. The counterpart is the `str2bits(string)` function.

- `integer` **`bool2num`**`(`boolean *arg*`);`

  The `bool2num(boolean)` function takes one boolean argument and converts it to either integer `1` (if the argument is true) or integer `0` (if the argument is false). Its counterpart is the `num2bool(<numeric type>)` function.

- `string` **`byte2base64`**`(`byte *arg*`);`

  The `byte2base64(byte)` function takes an array of bytes and converts it to a string in `base64` representation. Its counterpart is the `base64byte(string)` function.

- `string` **`byte2hex`**`(`byte *arg*`);`

  The `byte2hex(byte)` function takes an array of bytes and converts it to a string in `hexadecimal` representation. Its counterpart is the `hex2byte(string)` function.

- `string` **`byte2str`**`(`byte *payload*, string *charset*`);`

  Returns bytes converted to `string` using a given charset.

- `long` **`date2long`**`(`date *arg*`);`

  The `date2long(date)` function takes one date argument and converts it to a long type. Its value is equal to the number of milliseconds elapsed from `January 1, 1970, 00:00:00 GMT` to the date specified as the argument. Its counterpart is the `long2date(long)` function.

- `integer` **`date2num`**`(`date *arg*, unit *timeunit*`);`

  The `date2num(date, unit)` function accepts two arguments: the first is date and the other is any time unit. The unit can be one of the following: `year`, `month`, `week`, `day`, `hour`, `minute`, `second`, `millisec`. The unit must be specified as a constant. It can neither be received through an edge nor set as variable. The function takes these two arguments and converts them to an integer using system locale. If the time unit is contained in the date, it is returned as an integer number. If it is not contained, the function returns `0`. Remember that months are numbered starting from `1`. Thus, `date2num(2008-06-12, month)` returns `6`. And `date2num(2008-06-12, hour)` returns `0`.

- `integer` **`date2num`**`(`date *arg*, unit *timeunit*, string *locale*`);`

  The `date2num(date, unit, string)` function accepts three arguments: the first is date, the second is any time unit, the third is a locale. The unit can be one of the following: `year`, `month`, `week`, `day`, `hour`, `minute`, `second`, `millisec`. The unit must be specified as a constant. It can neither be received through an edge nor set as variable. The function takes these two arguments and converts them to an integer using the specified locale. If the time unit is contained in the date, it is returned as an integer number. If it is not contained, the function returns `0`. Remember that months are numbered starting from `1`. Thus, `date2num(2008-06-12, month)` returns `6`. And `date2num(2008-06-12, hour)` returns `0`.

- `string` **`date2str`**`(`date *arg*, string *pattern*`);`

  The `date2str(date, string)` function accepts two arguments: date and string. The function takes them and converts the date according to the `pattern` specified as the second argument. Thus, `date2str(2008-06-12, "dd.MM.yyyy")` returns the following string: `"12.6.2008"`. Its counterpart is the `str2date(string, string)` function.

- `string` **`date2str`**`(`date *arg*, string *pattern*, string *locale*`);`

  Converts the `date` field type into a date of the `string` data type according to the `pattern` (describing the date and time format) and `locale` (defining what date format symbols should be used). Thus, `date2str(2009/01/04,"yyyy-MMM-d","fr.CA")` returns `2009-janv.-4`. See Locale for more info about locale settings.

- `number` **`decimal2double`**`(`decimal *arg*`);`

  The `decimal2double(decimal)` function takes one argument of decimal data type and converts it to a double value.

  The conversion is narrowing. And, if a `decimal` value cannot be converted into a `double` (as the ranges of `double` data type do not cover all `decimal` values), the function throws exception. Thus, `decimal2double(92378352147483647.23)` returns `9.2378352147483648E16`.

  On the other hand, any `double` can be converted into a `decimal`. Both **Length** and **Scale** of a decimal can be adjusted for it.

- `integer` **`decimal2integer`**`(`decimal *arg*`);`

  The `decimal2integer(decimal)` function takes one argument of decimal data type and converts it to an integer.

  The conversion is narrowing. And, if a `decimal` value cannot be converted into an `integer` (as the range of `integer` data type does not cover the range of `decimal` values), the function throws exception. Thus, `decimal2integer(352147483647.23)` throws exception, whereas `decimal2integer(25.95)` returns `25`.

On the other hand, any `integer` can be converted into a `decimal` without loss of precision. **Length** of a decimal can be adjusted for it.

- `long` **`decimal2long`**`(`decimal *arg*`);`

  The `decimal2long(decimal)` function takes one argument of decimal data type and converts it to a long value.

  The conversion is narrowing. And, if a `decimal` value cannot be converted into a `long` (as the range of `long` data type does not cover all `decimal` values), the function throws exception. Thus, `decimal2long(9759223372036854775807.25)` throws exception, whereas `decimal2long(72036854775807.79)` returns `72036854775807`.

  On the other hand, any `long` can be converted into a `decimal` without loss of precision. **Length** of a decimal can be adjusted for it.

- `integer` **`double2integer`**`(`number *arg*`);`

  The `double2integer(number)` function takes one argument of double data type and converts it to an integer.

  The conversion is narrowing. And, if a `double` value cannot be converted into an `integer` (as the range of `double` data type does not cover all `integer` values), the function throws exception. Thus, `double2integer(352147483647.1)` throws exception, whereas `double2integer(25.757197)` returns `25`.

  On the other hand, any `integer` can be converted into a `double` without loss of precision.

- `long` **`double2long`**`(`number *arg*`);`

  The `double2long(number)` function takes one argument of double data type and converts it to a long.

  The conversion is narrowing. And, if a `double` value cannot be converted into a `long` (as the range of `double` data type does not cover all `long` values), the function throws exception. Thus, `double2long(1.3759739E23)` throws exception, whereas `double2long(25.8579)` returns `25`.

  On the other hand, any `long` can always be converted into a `double`, however, user should take into account that loss of precision may occur.

- `byte` **`hex2byte`**`(`string *arg*`);`

  The `hex2byte(string)` function takes one string argument in `hexadecimal` representation and converts it to an array of bytes. Its counterpart is the `byte2hex(byte)` function.

- `string` **`json2xml`**`(`string *arg*`);`

  The `json2xml(string)` function takes one string argument that is `JSON` formatted and converts it to an `XML` formatted string. Its counterpart is the `xml2json(string)` function.

- `date` **`long2date`**`(`long *arg*`);`

  The `long2date(long)` function takes one long argument and converts it to a date. It adds the argument number of milliseconds to `January 1, 1970, 00:00:00 GMT` and returns the result as a date. Its counterpart is the `date2long(date)` function.

- `integer` **`long2integer`**`(`long *arg*`);`

  The `long2integer(decimal)` function takes one argument of long data type and converts it to an integer value. The conversion is successful only if it is possible without any loss of information, otherwise the function throws exception. Thus, `long2integer(352147483647)` throws exception, whereas `long2integer(25)` returns `25`.

  On the other hand, any `integer` value can be converted into a `long` number without loss of precision.

- `byte` **`long2packDecimal`**`(`long *arg*`);`

  The `long2packDecimal(long)` function takes one argument of long data type and returns its value in the representation of packed decimal number. It is the counterpart of the `packDecimal2long(byte)` function.

- `byte` **`md5`**`(`byte *arg*`);`

  The `md5(byte)` function accepts one argument consisting of an array of bytes. It takes this argument and calculates its `MD5` hash value.

- `byte` **`md5`**`(`string *arg*`);`

  The `md5(string)` function accepts one argument of string data type. It takes this argument and calculates its `MD5` hash value.

- `boolean` **`num2bool`**`(`<numeric type> *arg*`);`

  The `num2bool(<numeric type>)` function takes one argument of any numeric data type (`integer`, `long`, `number`, or `decimal`) and returns boolean `false` for 0 and `true` for any other value.

- `string` **`num2str`**`(`<numeric type> *arg*`);`

  The `num2str(<numeric type>)` function takes one argument of any numeric data type (`integer`, `long`, `number`, or `decimal`) and converts it to a string in decimal representation. Locale is system value. Thus, `num2str(20.52)` returns `"20.52"`.

- `string` **`num2str`**`(`<numeric type> *arg*, integer *radix*`);`

  The `num2str(<numeric type>,integer)` function accepts two arguments: the first is of any of three numeric data types (`integer`, `long`, `number`) and the second is integer. It takes these two arguments and converts the first to its string representation in the `radix` based numeric system. Thus, `num2str(31, 16)` returns `"1F"`. Locale is system value.

For both `integer` and `long` data types, any integer number can be used as radix. For double (`number`) only 10 or 16 can be used as radix.

- `string` **`num2str`**`(`<numeric type> *arg*, string *format*`);`

  The `num2str(<numeric type>, string)` function accepts two arguments: the first is of any numeric data type (`integer`, `long`, `number`, or `decimal`) and the second is string. It takes these two arguments and converts the first to a string in decimal representation using the format specified as the second argument. Locale has system value.

- `string` **`num2str`**`(`<numeric type> *arg*, string *format*, string *locale*`);`

  The `num2str(<numeric type>, string, string)` function accepts three arguments: the first is of any numeric data type (`integer`, `long`, `number`, or `decimal`) and two are strings. It takes these arguments and converts the first to its string representation using the format specified as the second argument and the locale specified as the third argument.

- `long` **`packDecimal2long`**`(`byte *arg*`);`

  The `packDecimal2long(byte)` function takes one argument of an array of bytes whose meaning is the packed decimal representation of a long number. It returns its value as long data type. It is the counterpart of the `long2packDecimal(long)` function.

- `byte` **`sha`**`(`byte *arg*`);`

  The `sha(byte)` function accepts one argument consisting of an array of bytes. It takes this argument and calculates its `SHA` hash value.

- `byte` **`sha`**`(`string *arg*`);`

  The `sha(string)` function accepts one argument of string data type. It takes this argument and calculates its `SHA` hash value.

- `byte` **`str2bits`**`(`string *arg*`);`

  The `str2bits(string)` function takes one string argument and converts it to an array of bytes. Its counterpart is the `bits2str(byte)` function. The string consists of the following characters: Each of them can be either `"1"` or it can be any other character. In the string, each character `"1"` is converted to the bit `1`, all other characters (not only `"0"`, but also `"a"`, `"z"`, `"/"`, etc.) are converted to the bit `0`. If the number of characters in the string is not an integral multiple of eight, the string is completed by "0" characters from the right. Then, the string is converted to an array of bytes as if the number of its characters were integral multiple of eight.

  The first character represents the lowest bit.

- `boolean` **`str2bool`**`(`string *arg*`);`

  The `str2bool(string)` function takes one string argument and converts it to the corresponding boolean value. The string can be one of the following: `"TRUE"`, `"true"`, `"T"`, `"t"`, `"YES"`, `"yes"`, `"Y"`, `"y"`, `"1"`, `"FALSE"`, `"false"`, `"F"`, `"f"`, `"NO"`, `"no"`, `"N"`, `"n"`, `"0"`. The strings are converted to boolean `true` or boolean `false`.

- `string` **`str2byte`**`(`string *payload*, string *charset* `);`

  Returns a string converted from input bytes using a given charset encoder.

- `date` **`str2date`**`(`string *arg*, string *pattern*`);`

  The `str2date(string, string)` function accepts two string arguments. It takes them and converts the first string to the date according to the `pattern` specified as the second argument. The `pattern` must correspond to the structure of the first argument. Thus, `str2date("12.6.2008", "dd.MM.yyyy")` returns the following date: `2008-06-12`.

- `date` **`str2date`**`(`string *arg*, string *pattern*, string *locale*`);`

  The `str2date(string, string, string)` function accepts three string arguments and one boolean. It takes the arguments and converts the first string to the date according to the `pattern` and `locale` specified as the second and the third argument, respectively. The `pattern` must correspond to the structure of the first argument. Thus, `str2date("12.6.2008", "dd.MM.yyyy",cs.CZ)` returns the following date: `2008-06-12` . The third argument defines the locale for the date.

- `decimal` **`str2decimal`**`(`string *arg*`);`

  The `str2decimal(string)` function takes one string argument and converts it to the corresponding decimal value.

- `decimal` **`str2decimal`**`(`string *arg*, string *format*`);`

  The `str2decimal(string, string)` function takes the first string argument and converts it to the corresponding decimal value according to the format specified as the second argument. Locale has system value.

- `decimal` **`str2decimal`**`(`string *arg*, string *format*, string *locale*`);`

  The `str2decimal(string, string, string)` function takes the first string argument and converts it to the corresponding decimal value according to the format specified as the second argument and the locale specified as the third argument.

- `number` **`str2double`**`(`string *arg*`);`

  The `str2double(string)` function takes one string argument and converts it to the corresponding double value.

- `number` **`str2double`**`(`string *arg*, string *format*`);`

  The `str2double(string, string)` function takes the first string argument and converts it to the corresponding double value according to the format specified as the second argument. Locale has system value.

- `number` **`str2double`**`(`string *arg*, string *format*, string *locale*`);`

The `str2decimal(string, string, string)` function takes the first string argument and converts it to the corresponding double value according to the format specified as the second argument and the locale specified as the third argument.

- `integer` **`str2integer`**`(`string *arg*`);`

  The `str2integer(string)` function takes one string argument and converts it to the corresponding integer value.

- `integer` **`str2integer`**`(`string *arg*, integer *radix*`);`

  The `str2integer(string, integer)` function accepts two arguments: string and integer. It takes the first argument as if it were expressed in the `radix` based numeric system representation and returns its corresponding integer decimal value.

- `integer` **`str2integer`**`(`string *arg*, string *format*`);`

  The `str2integer(string, string)` function takes the first string argument as decimal string representation of an integer number corresponding to the format specified as the second argument and the system locale and converts it to the corresponding integer value.

- `integer` **`str2integer`**`(`string *arg*, string *format*, string *locale*`);`

  The `str2integer(string, string, string)` function takes the first string argument as decimal string representation of an integer number corresponding to the format specified as the second argument and the locale specified as the third argument and converts it to the corresponding integer value.

- `long` **`str2long`**`(`string *arg*, integer *radix*`);`

  The `str2long(string, integer)` function accepts two arguments: string and integer. It takes the first argument as if it were expressed in the `radix` based numeric system representation and returns its corresponding long decimal value.

- `long` **`str2long`**`(`string *arg*, string *format*`);`

  The `str2long(string, string)` function takes the first string argument as decimal string representation of a long number corresponding to the format specified as the second argument and the system locale and converts it to the corresponding long value.

- `long` **`str2long`**`(`string *arg*, string *format*, string *locale*`);`

  The `str2long(string, string, string)` function takes the first string argument as decimal string representation of a long number corresponding to the format specified as the second argument and the locale specified as the third argument and converts it to the corresponding long value.

- `string` **`toString`**`(`<numeric|list|map type> *arg*`);`

  The `toString(<numeric|list|map type>)` function takes one argument and converts it to its string representation. It accepts any numeric data type, list of any data type, as well as map of any data types.

- `string` **`xml2json`**`(`string *arg*`);`

  The `xml2josn(string)` function takes one string argument that is `XML` formatted and converts it to a `JSON` formatted string. Its counterpart is the `json2xml(string)` function.

Date Functions

When you work with dates, you may use the functions that process dates.

In these functions, sometimes a format pattern of a date or any number must be defined. Locale can also have an influence on their formatting.

- For detailed information about date formatting and/or parsing see Data and Time Format. **What? (where?)**
- For detailed information about locale see Locale.

Note

Remember that numeric and date formats are displayed using system value **Locale** or **Locale** specified in the `defaultProperties` file, unless another **Locale** is explicitly specified.

Here we provide the list of the functions:

- `date` **`dateAdd`**`(`date *arg*, long *amount*, unit *timeunit*`);`

  The `dateAdd(date, long, unit)` function accepts three arguments: the first is date, the second is of long data type and the last is any time unit. The unit can be one of the following: `year`, `month`, `week`, `day`, `hour`, `minute`, `second`, `millisec`. The unit must be specified as a constant. It can neither be received through an edge nor set as variable. The function takes the first argument, adds the amount of time units to it and returns the result as a date. The `amount` and `time` units are specified as the second and third arguments, respectively.

- `long` **`dateDiff`**`(`date *later*, date *earlier*, unit *timeunit*`);`

  The `dateDiff(date, date, unit)` function accepts three arguments: two `dates` and one `time` unit. It takes these arguments and subtracts the second argument from the first argument. The unit can be one of the following: `year`, `month`, `week`, `day`, `hour`, `minute`, `second`, `millisec`. The unit must be specified as a constant. It can be neither received through an edge nor set as variable. The function returns the resulting time difference expressed in time units specified as the third argument. Thus, the difference of two dates is expressed in defined time units. The result is expressed as an integer number. Thus, `dateDiff(2008-06-18, 2001-02-03, year)` returns `7`. But, `dateDiff(2001-02-03, 2008-06-18, year)` returns `-7`!

- `date` **`extractDate`**`(`date *arg*`);`

  The `extractDate(date)` function takes one date argument and returns only the information containing year, month, and day values. The function's argument is not modified by the return value.

- `date` **`extractTime`**`(`date *arg*`);`

The `extractTime(date)` function takes one date argument and returns only the information containing hours, minutes, seconds, and milliseconds. The function's argument is not modified by the return value.

- `date` **`randomDate`**`(`date *startDate*, date *endDate*`);`

  The `randomDate(date, date)` function accepts two date arguments and returns a random date between `startDate` and `endDate`. These resulting dates are generated at random for different records and different fields. They can be different for both records and fields. The return value can also be `startDate` or `endDate`. However, it cannot be the date before `startDate` nor after `endDate`. Remember that dates represent 0 hours and 0 minutes and 0 seconds and 0 milliseconds of the specified day, thus, if you want that `endDate` could be returned, enter the next date as `endDate`. As `locale`, system value is used. The default `format` is specified in the `defaultProperties` file.

- `date` **`randomDate`**`(`long *startDate*, long *endDate*`);`

  The `randomDate(long, long)` function accepts two arguments of long data type - each of them represents a date - and returns a random date between `startDate` and `endDate`. These resulting dates are generated at random for different records and different fields. They can be different for both records and fields. The return value can also be `startDate` or `endDate`. However, it cannot be the date before `startDate` nor after `endDate`. Remember that dates represent 0 hours and 0 minutes and 0 seconds and 0 milliseconds of the specified day, thus, if you want that `endDate` could be returned, enter the next date as `endDate`. As `locale`, system value is used. The default `format` is specified in the `defaultProperties` file.

- `date` **`randomDate`**`(`string *startDate*, string *endDate*, string *format*`);`

  The `randomDate(string, string, string)` function accepts three stringarguments. Two first represent dates, the third represents a format. The function returns a random date between `startDate` and `endDate` corresponding to the `format` specified by the third argument. These resulting dates are generated at random for different records and different fields. They can be different for both records and fields. The return value can also be `startDate` or `endDate`. However, it cannot be the date before `startDate` nor after `endDate`. Remember that dates represent 0 hours and 0 minutes and 0 seconds and 0 milliseconds of the specified day, thus, if you want that `endDate` could be returned, enter the next date as `endDate`. As `locale`, system value is used.

- `date` **`randomDate`**`(`string *startDate*, string *endDate*, string *format*, string *locale*`);`

  The `randomDate(string, string, string, string)` function accepts four string arguments. The first and the second argument represent dates. The third is a format and the fourt is locale. The function returns a random date between `startDate` and `endDate`. These resulting dates are generated at random for different records and different fields. They can be different for both records and fields. The return value can also be `startDate` or `endDate` corresponding to the `format` and the `locale` specified by the third and the fourth argument, respectively. However, it cannot be the date before `startDate` nor after `endDate`. Remember that dates represent 0 hours and 0 minutes and 0 seconds and 0 milliseconds of the specified day, thus, if you want that `endDate` could be returned, enter the next date as `endDate`.

- `date` **`today`**`();`

  The `today()` function accepts no argument and returns current date and time.

- `date` **`zeroDate`**`();`

  The `zeroDate()` function accepts no argument and returns 1.1.1970.

Note

The following two functions are **deprecated**. Their return value modifies the argument at the same time.

- `date` **`trunc`**`(`date *arg*`);`

  The `trunc(date)` function takes one date argument and returns the date with the same year, month and day, but hour, minute, second and millisecond are set to 0 values.

- `date` **`truncDate`**`(`date *arg*`);`

  The `truncDate(date)` function takes one date argument and returns the date with the same hour, minute, second and millisecond, but year, month and day are set to 0 values. The 0 date is 0001-01-01.

Mathematical Functions

You may also want to use some mathematical functions:

- `<numeric type>` **`abs`**`(`<numeric type> *arg*`);`

  The `abs(<numeric type>)` function takes one argument of any numeric data type (`integer`, `long`, `number`, or `decimal`) and returns its absolute value in the same data type.

- `integer` **`bitAnd`**`(`integer *arg1*, integer *arg2*`);`

  The `bitAnd(integer, integer)` function accepts two arguments of integer data type. It takes them and returns the number corresponding to the bitwise `and`. (For example, `bitAnd(11,7)` returns `3`.) As decimal `11` can be expressed as bitwise `1011`, decimal `7` can be expressed as `111`, thus the result is `11` what corresponds to decimal `3`.

- `long` **`bitAnd`**`(`long *arg1*, long *arg2*`);`

  The `bitAnd(long, long)` function accepts two arguments of long data type. It takes them and returns the number corresponding to the bitwise `and`. (For example, `bitAnd(11,7)` returns `3`.) As decimal `11` can be expressed as bitwise `1011`, decimal `7` can be expressed as `111`, thus the result is `11` what corresponds to decimal `3`.

- `boolean` **`bitIsSet`**`(`integer *arg*, integer *Index*`);`

  The `bitIsSet(integer, integer)` function accepts two arguments of integer data type. It takes them, determines the value of the bit of the first argument located on the `Index` and returns `true` or `false`, if the bit is `1` or `0`, respectively. (For example, `bitIsSet(11,3)` returns `true`.) As decimal `11` can be expressed as bitwise `1011`, the bit whose index is 3 (the fourth from the right) is `1`, thus the result is `true`. And `bitIsSet(11,2)` would return `false`.

- `boolean` **`bitIsSet`**`(`long *arg*, integer *Index*`);`

The `bitIsSet(long, integer)` function accepts one argument of long data type and one integer. It takes these arguments, determines the value of the bit of the first argument located on the `Index` and returns `true` or `false`, if the bit is `1` or `0`, respectively. (For example, `bitIsSet(11,3)` returns `true`.) As decimal `11` can be expressed as bitwise `1011`, the bit whose index is 3 (the fourth from the right) is `1`, thus the result is `true`. And `bitIsSet(11,2)` would return `false`.

- `integer` **`bitLShift`**`(`integer *arg*, integer *Shift*`);`

  The `bitLShift(integer, integer)` function accepts two arguments of integer data type. It takes them and returns the number corresponding to the original number with some bits added (`Shift` number of bits on the left side are added and set to `0`.) (For example, `bitLShift(11,2)` returns `44`.) As decimal `11` can be expressed as bitwise `1011`, thus the two bits on the right side (`10`) are added and the result is `101100` which corresponds to decimal `44`.

- `long` **`bitLShift`**`(`long *arg*, long *Shift*`);`

  The `bitLShift(long, long)` function accepts two arguments of long data type. It takes them and returns the number corresponding to the original number with some bits added (`Shift` number of bits on the left side are added and set to `0`.) (For example, `bitLShift(11,2)` returns `44`.) As decimal `11` can be expressed as bitwise `1011`, thus the two bits on the right side (`10`) are added and the result is `101100` which corresponds to decimal `44`.

- `integer` **`bitNegate`**`(`integer *arg*`);`

  The `bitNegate(integer)` function accepts one argument of integer data type. It returns the number corresponding to its bitwise `inverted number`. (For example, `bitNegate(11)` returns `-12`.) The function inverts all bits in an argument.

- `long` **`bitNegate`**`(`long *arg*`);`

  The `bitNegate(long)` function accepts one argument of long data type. It returns the number corresponding to its bitwise `inverted number`. (For example, `bitNegate(11)` returns `-12`.) The function inverts all bits in an argument.

- `integer` **`bitOr`**`(`integer *arg1*, integer *arg2*`);`

  The `bitOr(integer, integer)` function accepts two arguments of integer data type. It takes them and returns the number corresponding to the bitwise `or`. (For example, `bitOr(11,7)` returns `15`.) As decimal `11` can be expressed as bitwise `1011`, decimal `7` can be expressed as `111`, thus the result is `1111` what corresponds to decimal `15`.

- `long` **`bitOr`**`(`long *arg1*, long *arg2*`);`

  The `bitOr(long, long)` function accepts two arguments of long data type. It takes them and returns the number corresponding to the bitwise `or`. (For example, `bitOr(11,7)` returns `15`.) As decimal `11` can be expressed as bitwise `1011`, decimal `7` can be expressed as `111`, thus the result is `1111` what corresponds to decimal `15`.

- `integer` **`bitRShift`**`(`integer *arg*, integer *Shift*`);`

The `bitRShift(integer, integer)` function accepts two arguments of integer data type. It takes them and returns the number corresponding to the original number with some bits removed (`Shift` number of bits on the right side are removed.) (For example, `bitRShift(11,2)` returns `2`.) As decimal `11` can be expressed as bitwise `1011`, thus the two bits on the right side are removed and the result is `10` what corresponds to decimal `2`.

- `long` **`bitRShift`**`(`long *arg*, long *Shift*`);`

  The `bitRShift(long, long)` function accepts two arguments of long data type. It takes them and returns the number corresponding to the original number with some bits removed (`Shift` number of bits on the right side are removed.) (For example, `bitRShift(11,2)` returns `2`.) As decimal `11` can be expressed as bitwise `1011`, thus the two bits on the right side are removed and the result is `10` what corresponds to decimal `2`.

- `integer` **`bitSet`**`(`integer *arg1*, integer *Index*, boolean *SetBitTo1*`);`

  The `bitSet(integer, integer, boolean)` function accepts three arguments. The first two are of integer data type and the third is boolean. It takes them, sets the value of the bit of the first argument located on the `Index` specified as the second argument to `1` or `0`, if the third argument is `true` or `false`, respectively, and returns the result as an integer. (For example, `bitSet(11,3,false)` returns `3`.) As decimal `11` can be expressed as bitwise `1011`, the bit whose index is 3 (the fourth from the right) is set to `0`, thus the result is `11` what corresponds to decimal `3`. And `bitSet(11,2,true)` would return `1111` what corresponds to decimal `15`.

- `long` **`bitSet`**`(`long *arg1*, integer *Index*, boolean *SetBitTo1*`);`

  The `bitSet(long, integer, boolean)` function accepts three arguments. The first is long, the second is integer, and the third is boolean. It takes them, sets the value of the bit of the first argument located on the `Index` specified as the second argument to `1` or `0`, if the third argument is `true` or `false`, respectively, and returns the result as an integer. (For example, `bitSet(11,3,false)` returns `3`.) As decimal `11` can be expressed as bitwise `1011`, the bit whose index is 3 (the fourth from the right) is set to `0`, thus the result is `11` what corresponds to decimal `3`. And `bitSet(11,2,true)` would return `1111` what corresponds to decimal `15`.

- `integer` **`bitXor`**`(`integer *arg*, integer *arg*`);`

  The `bitXor(integer, integer)` function accepts two arguments of integer data type. It takes them and returns the number corresponding to the bitwise `exclusive or`. (For example, `bitXor(11,7)` returns `12`.) As decimal `11` can be expressed as bitwise `1011`, decimal `7` can be expressed as `111`, thus the result is `1100` what corresponds to decimal `15`.

- `long` **`bitXor`**`(`long *arg*, long *arg*`);`

  The `bitXor(long, long)` function accepts two arguments of long data type. It takes them and returns the number corresponding to the bitwise `exclusive or`. (For example, `bitXor(11,7)` returns `12`.) As decimal `11` can be expressed as bitwise `1011`, decimal `7` can be expressed as `111`, thus the result is `1100` what corresponds to decimal `15`.

- `number` **`ceil`**`(`decimal *arg*`);`

  The `ceil(decimal)` function takes one argument of decimal data type and returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument and is equal to a mathematical integer.

- `number` **`ceil`**`(`number *arg*`);`

  The `ceil(number)` function takes one argument of double data type and returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument and is equal to a mathematical integer.

- `number` **`e`**`();`

  The `e()` function accepts no argument and returns the Euler number.

- `number` **`exp`**`(`<numeric type> *arg*`);`

  The `exp(<numeric type>)` function takes one argument of any numeric data type (`integer`, `long`, `number`, or `decimal`) and returns the result of the exponential function of this argument.

- `number` **`floor`**`(`decimal *arg*`);`

  The `floor(decimal)` function takes one argument of decimal data type and returns the largest (closest to positive infinity) double value that is less than or equal to the argument and is equal to a mathematical integer.

- `number` **`floor`**`(`number *arg*`);`

  The `floor(number)` function takes one argument of double data type and returns the largest (closest to positive infinity) double value that is less than or equal to the argument and is equal to a mathematical integer.

- `void` **`setRandomSeed`**`(`long *arg*`);`

  The `setRandomSeed(long)` takes one long argument and generates the seed for all functions that generate values at random.

  This function should be used in the `preExecute()` function or method.

  In such a case, all values generated at random do not change on different runs of the graph, they even remain the same after the graph is resetted.

- `number` **`log`**`(`<numeric type> *arg*`);`

  The `log(<numeric type>)` takes one argument of any numeric data type (`integer`, `long`, `number`, or `decimal`) and returns the result of the natural logarithm of this argument.

- `number` **`log10`**`(`<numeric type> *arg*`);`

  The `log10(<numeric type>)` function takes one argument of any numeric data type (`integer`, `long`, `number`, or `decimal`) and returns the result of the logarithm of this argument to the base 10.

- `number` **`pi`**`();`

The `pi()` function accepts no argument and returns the pi number.

- `number` **`pow`**`(`<numeric type> *base*, <numeric type> *exp*`);`

  The `pow(<numeric type>, <numeric type>)` function takes two arguments of any numeric data types (that do not need to be the same, `integer`, `long`, `number`, or `decimal`) and returns the exponential function of the first argument as the exponent with the second as the base.

- `number` **`random`**`();`

  The `random()` function accepts no argument and returns a random positive double greater than or equal to `0.0` and less than `1.0`.

- `boolean` **`randomBoolean`**`();`

  The `randomBoolean()` function accepts no argument and generates at random boolean values `true` or `false`. If these values are sent to any numeric data type field, they are converted to their numeric representation automatically (`1` or `0`, respectively).

- `number` **`randomGaussian`**`();`

  The `randomGaussian()` function accepts no argument and generates at random both positive and negative values of number data type in a Gaussian distribution.

- `integer` **`randomInteger`**`();`

  The `randomInteger()` function accepts no argument and generates at random both positive and negative integer values.

- `integer` **`randomInteger`**`(`integer *Minimum*, integer *Maximum*`);`

  The `randomInteger(integer, integer)` function accepts two argument of integer data types and returns a random integer value greater than or equal to `Minimum` and less than or equal to `Maximum`.

- `long` **`randomLong`**`();`

  The `randomLong()` function accepts no argument and generates at random both positive and negative long values.

- `long` **`randomLong`**`(`long *Minimum*, long *Maximum*`);`

  The `randomLong(long, long)` function accepts two argument of long data types and returns a random long value greater than or equal to `Minimum` and less than or equal to `Maximum`.

- `long` **`round`**`(`decimal *arg*`);`

  The `round(decimal)` function takes one argument of decimal data type and returns the long that is closest to this argument. Decimal is converted to number prior to the operation.

- `long` **`round`**`(`number *arg*`);`

  The `round(number)` function takes one argument of number data type and returns the long that is closest to this argument.

- `number` **`sqrt`**`(`<numeric type> *arg*`);`

  The `sqrt(mumerictype)` function takes one argument of any numeric data type (`integer`, `long`, `number`, or `decimal`) and returns the square root of this argument.

String Functions

Some functions work with strings.

In the functions that work with strings, sometimes a format pattern of a date or any number must be defined.

- For detailed information about date formatting and/or parsing see Data and Time Format. **What? (where)**
- For detailed information about formatting and/or parsing of any numeric data type see Numeric Format.
- For detailed information about locale see Locale.

Note

Remember that numeric and date formats are displayed using system value **Locale** or **Locale** specified in the `defaultProperties` file, unless other **Locale** is explicitly specified.

Here we provide the list of the functions:

- `string` **`charAt`**`(`string *arg*, integer *index*`);`

  The `charAt(string, integer)` function accepts two arguments: the first is string and the second is integer. It takes the string and returns the character that is located at the position specified by the `index`.

- `string` **`chop`**`(`string *arg*`);`

  The `chop(string)` function accepts one string argument. The function takes this argument, removes the line feed and the carriage return characters from the end of the string specified as the argument and returns the new string without these characters.

- `string` **`chop`**`(`string *arg1*, string *arg2*`);`

  The `chop(string, string)` function accepts two string arguments. It takes the first argument, removes the string specified as the second argument from the end of the first argument and returns the first string argument without the string specified as the second argument.

- `string` **`concat`**`(`string *arg1*, string ..., string *argN*`);`

  The `concat(string, ..., string)` function accepts unlimited number of arguments of string data type. It takes these arguments and returns their concatenation. You can also concatenate these arguments using plus signs, but this function is faster for more than two arguments.

- `integer` **`countChar`**`(`string *arg*, string *character*`);`

  The `countChar(string, string)` function accepts two arguments: the first is string and the second is one character. It takes them and returns the number of occurrences of the character specified as the second argument in the string specified as the first argument.

- `string[]` **`cut`**`(`string *arg*, integer[] *indeces*`);`

  The `cut(string, integer[])` function accepts two arguments: the first is string and the second is list of integers. The function returns a list of strings. The number of elements of the list specified as the second argument must be even. The integers in the list serve as position (each number in the odd position) and length (each number in the even position). Substrings of the specified length are taken from the string specified as the first argument starting from the specified position (excluding the character at the specified position). The resulting substrings are returned as list of strings. For example, `cut("somestringasanexample",[2,3,1,5])` returns `["mes","omest"]`.

- `integer` **`editDistance`**`(`string *arg1*, string *arg2*`);`

  The `editDistance(string, string)` function accepts two string arguments. These strings will be compared to each other. The strength of comparison is 4 by default, the default value of locale for comparison is the system value and the maximum difference is 3 by default.

  The function returns the number of letters that should be changed to transform one of the two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

  For more details, see another version of the `editDistance()` function below - the editDistance (string, string, integer, string, integer) function.

- `integer` **`editDistance`**`(`string *arg1*, string *arg2*, string *locale*`);`

  The `editDistance(string, string, string)` function accepts three arguments. The first two are strings that will be compared to each other and the third (string) is the locale that will be used for comparison. The default strength of comparison is 4. The maximum difference is 3 by default.

  The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

  For more details, see another version of the `editDistance()` function below - the editDistance (string, string, integer, string, integer) function.

- `integer` **`editDistance`**`(`string *arg1*, string *arg2*, integer *strength*`);`

  The `editDistance(string, string, integer)` function accepts three arguments. The first two are strings that will be compared to each other and the third (integer) is the strength of comparison. The default locale that will be used for comparison is the system value. The maximum difference is 3 by default.

  The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

  For more details, see another version of the `editDistance()` function below - the editDistance (string, string, integer, string, integer) function.

- `integer` **`editDistance`**`(`string *arg1*, string *arg2*, integer *strength*, string *locale*`);`

  The `editDistance(string, string, integer, string)` function accepts four arguments. The first two are strings that will be compared to each other, the third (integer) is the strength of comparison and the fourth (string) is the locale that will be used for comparison. The maximum difference is 3 by default.

  The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

  For more details, see another version of the `editDistance()` function below - the editDistance (string, string, integer, string, integer) function.

- `integer` **`editDistance`**`(`string *arg1*, string *arg2*, string *locale*, integer *maxDifference*`);`

  The `editDistance(string, string, string, integer)` function accepts four arguments. The first two are strings that will be compared to each other, the third (string) is the locale that will be used for comparison and the fourth (integer) is the maximum difference. The strength of comparison is 4 by default.

  The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

  For more details, see another version of the `editDistance()` function below - the editDistance (string, string, integer, string, integer) function.

- `integer` **`editDistance`**`(`string *arg1*, string *arg2*, integer *strength*, integer *maxDifference*`);`

  The `editDistance(string, string, integer, integer)` function accepts four arguments. The first two are strings that will be compared to each other and the two others are both integers. These are the strength of comparison (third argument) and the maximum difference (fourth argument). The locale is the default system value.

The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

For more details, see another version of the `editDistance()` function below - the editDistance (string, string, integer, string, integer) function.

- `integer` **`editDistance`**`(`string *arg1*, string *arg2*, integer *strength*, string *locale*, integer *maxDifference*`);`

  The `editDistance(string, string, integer, string, integer)` function accepts five arguments. The first two are strings, the three others are integer, string and integer, respectively. The function takes the first two arguments and compares them to each other using the other three arguments.

  The third argument (integer number) specifies the strength of comparison. It can have any value from 1 to 4.

  If it is 4 (identical comparison), that means that only identical letters are considered equal. In case of 3 (tertiary comparison), that means that upper and lower cases are considered equal. If it is 2 (secondary comparison), that means that letters with diacritical marks are considered equal. Last, if the strength of comparison is 1 (primary comparison), that means that even the letters with some specific signs are considered equal. In other versions of the `editDistance()` function where this strength of comparison is not specified, the number 4 is used as the default strength (see above).

  The fourth argument is the string data type. It is the locale that serves for comparison. If no locale is specified in other versions of the `editDistance()` function, its default value is the system value (see above).

  The fifth argument (integer number) means the number of letters that should be changed to transform one of the first two arguments to the other. If another version of the `editDistance()` function does not specify this maximum difference, the default maximum difference is number 3 (see above).

  The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

  Actually the function is implemented for the following locales: CA, CZ, ES, DA, DE, ET, FI, FR, HR, HU, IS, IT, LT, LV, NL, NO, PL, PT, RO, SK, SL, SQ, SV, TR. These locales have one thing in common: they all contain language-specific characters. A complete list of these characters can be examined in CTL Appendix - List of National-specific Characters

- `string` **`escapeUrl`**`(`string *arg*`);`

  The function escapes illegal characters within components of specified URL (see isUrl() CTL function for the URL component description). Illegal characters must be escaped by a percent character `%` symbol, followed by the two-digit hexadecimal representation (case-insensitive) of the ISO-Latin code point for the character, e.g., `%20` is the escaped encoding for the US-ASCII space character.

- `string[]` **`find`**`(`string *arg*, string *regex*`);`

  The `find(string, string)` function accepts two string arguments. The second one is a regular expression. The function takes them and returns a list of substrings corresponding to the regex pattern that are found in the string specified as the first argument.

- `string` **`getAlphanumericChars`**`(`string *arg*`);`

  The `getAlphanumericChars(string)` function takes one string argument and returns only letters and digits contained in the string argument in the order of their appearance in the string. The other characters are removed.

- `string` **`getAlphanumericChars`**`(`string *arg*, boolean *takeAlpha*, boolean *takeNumeric*`);`

  The `getAlphanumericChars(string, boolean, boolean)` function accepts three arguments: one string and two booleans. It takes them and returns letters and/or digits if the second and/or the third arguments, respectively, are set to true.

- `string` **`getFieldLabel`**`(`reference *record*, string *arg*`);`

  The function returns a label of a field whose name is specified in `arg`. The fields are taken from `record`.

- `string` **`getFieldLabel`**`(`reference *record*, integer *arg*`);`

  The function returns a label of a field whose index is specified in `arg`. The fields are taken from `record`.

- `string` **`getUrlHost`**`(`string *arg*`);`

  The function parses out host name from specified URL (see isUrl() CTL function for the scheme). If the hostname part is not present in the URL argument, an empty string is returned. If the URL is not valid, `null` is returned.

- `string` **`getUrlPath`**`(`string *arg*`);`

  The function parses out path from specified URL (see isUrl() CTL function for the scheme). If the path part is not present in the URL argument, an empty string is returned. If the URL is not valid, `null` is returned.

- `integer` **`getUrlPort`**`(`string *arg*`);`

  The function parses out port number from specified URL (see isUrl() CTL function for the scheme). If the port part is not present in the URL argument, -1 is returned. If the URL has invalid syntax, -2 is returned.

- `string` **`getUrlProtocol`**`(`string *arg*`);`

  The function parses out protocol name from specified URL (see isUrl() CTL function for the scheme). If the protocol part is not present in the URL argument, an empty string is returned. If the URL is not valid, `null` is returned.

- `string` **`getUrlQuery`**`(`string *arg*`);`

  The function parses out query (parameters) from specified URL (see isUrl() CTL function for the scheme). If the query part is not present in the URL argument, an empty string is returned. If the URL syntax is invalid, `null` is returned.

- `string` **`getUrlUserInfo`**`(`string *arg*`);`

  The function parses out username and password from specified URL (see isUrl() CTL function for the scheme). If the userinfo part is not present in the URL argument, an empty string is returned. If the URL syntax is invalid, `null` is returned.

- `string` **`getUrlRef`**`(`string *arg*`);`

  The function parses out fragment after # character, also known as ref, reference or anchor, from specified URL (see isUrl() CTL function for the scheme). If the fragment part is not present in the URL argument, an empty string is returned. If the URL syntax is invalid, `null` is returned.

- `integer` **`indexOf`**`(`string *arg*, string *substring*`);`

  The `indexOf(string, string)` function accepts two strings. It takes them and returns the index of the first appearance of `substring` in the string specified as the first argument.

- `integer` **`indexOf`**`(`string *arg*, string *substring*, integer *fromIndex*`);`

  The `indexOf(string, string, integer)` function accepts three arguments: two strings and one integer. It takes them and returns the index of the first appearance of `substring` counted from the character located at the position specified by the third argument.

- `boolean` **`isAscii`**`(`string *arg*`);`

  The `isAscii(string)` function takes one string argument and returns a boolean value depending on whether the string can be encoded as an ASCII string (true) or not (false).

- `boolean` **`isBlank`**`(`string *arg*`);`

  The `isBlank(string)` function takes one string argument and returns a boolean value depending on whether the string contains only white space characters (true) or not (false).

- `boolean` **`isDate`**`(`string *arg*, string *pattern*`);`

  The `isDate(string, string)` function accepts two string arguments. It takes them, compares the first argument with the second as a pattern and, if the first string can be converted to a date which is valid within system value of `locale`, according to the specified `pattern`, the function returns true. If it is not possible, it returns false.

  (For more details, see another version of the `isDate()` function below - the `isDate(string, string, string, boolean)` function.)

  This function is a variant of the mentioned `isDate(string, string, string)` function in which the default value of the third argument is set to system value.

- `boolean` **`isDate`**`(`string *arg*, string *pattern*, string *locale*`);`

  The `isDate(string, string, string)` function accepts three string arguments. It takes them, compares the first argument with the second as a pattern, use the third argument (`locale`) and, if the first string can be converted to a date which is valid within specified `locale`, according to the specified `pattern`, the function returns true. If it is not possible, it returns false.

  (For more details, see another version of the `isDate()` function below - the `isDate(string, string, string, boolean)` function.)

- `boolean` **`isInteger`**`(`string *arg*`);`

  The `isInteger(string)` function takes one string argument and returns a boolean value depending on whether the string can be converted to an integer number (true) or not (false).

- `boolean` **`isLong`**`(`string *arg*`);`

  The `isLong(string)` function takes one string argument and returns a boolean value depending on whether the string can be converted to a long number (true) or not (false).

- `boolean` **`isNumber`**`(`string *arg*`);`

  The `isNumber(string)` function takes one string argument and returns a boolean value depending on whether the string can be converted to a double (true) or not (false).

- `boolean` **`isUrl`**`(`string *arg*`);`

  The function checks whether specified string is a valid URL of the following syntax

  ```
  foo://username:passw@host.com:8042/there/index.dtb?type=animal;na
  me=cat#nose \_/   \____________/ \______/ \__/\______________/
  \___________________/ \__/  |            |           |      |
  |                   |               | protocol  userinfo    host
  port      path                query          ref
  ```

- `string` **`join`**`(`string *delimiter*, <element type>[] *arg*`);`

  The `join(string, <element type>[])` function accepts two arguments. The first is string, the second is a list of elements of any data type. The elements that are not strings are converted to their string representation and put together with the first argument as delimiter.

- `string` **`join`**`(`string *delimiter*, map[<type of key>,<type of value>] *arg*`);`

  The `join(string, map[<type of key>,<type of value>])` function accepts two arguments. The first is string, the second is a map of any data types. The map elements are displayed as `key=value` strings. These are put together with the first argument as delimiter.

- `string` **`left`**`(`string *arg*, integer *length*`);`

  The `left(string, integer)` function accepts two arguments: the first is string and the second is integer. It takes them and returns the substring of the length specified as the second argument counted from the start of the string specified as the first argument. If the input string is shorter than the *`length`* parameter, an exception is thrown and the graph fails. To avoid such failure, use the `left(string, integer, boolean)` function described below.

- `string` **`left`**`(`string *arg*, integer *length*, boolean *spacePad*`);`

  The function returns prefix of the specified length. If the input string is longer or equally long as the *`length`* parameter, the function behaves the same way as the `left(string, integer)` function. There is different behaviour if the input string is shorter than the specified length. If the 3th argument is `true`, the right side of the result is padded with blank spaces so that the result has specified length beeing left justified. Whereas if `false`, the input string is returned as the result with no space added.

- `integer` **`length`**`(`structuredtype *arg*`);`

  The `length(structuredtype)` function accepts a structured data type as its argument: `string`, `<element type>[]`, `map[<type of key>,<type of value>]` or `record`. It takes the argument and returns a number of elements forming the structured data type.

- `string` **`lowerCase`**`(`string *arg*`);`

  The `lowerCase(string)` function takes one string argument and returns another string with cases converted to lower cases only.

- `boolean` **`matches`**`(`string *arg*, string *regex*`);`

  The `matches(string, string)` function takes two string arguments. The second argument is some regular expression. If the first argument can be expressed with such regular expression, the function returns `true`, otherwise it is `false`.

- `string` **`metaphone`**`(`string *arg*, integer *maxLength*`);`

  The `metaphone(string, integer)` function accepts one string argument and one integer meaning the maximum length. The function takes these arguments and returns the metaphone code of the first argument of the specified maximum length. The default maximum length is 4.

- `string` **`NYSIIS`**`(`string *arg*`);`

  The `NYSIIS(string)` function takes one string argument and returns the New York State Identification and Intelligence System Phonetic Code of the argument.

- `string` **`randomString`**`(`integer *minLength*, integer *maxLength*`);`

  The `randomString(integer, integer)` function takes two integer arguments and returns strings composed of lowercase letters whose length varies between `minLength` and `maxLength`. These resulting strings are generated at random for records and fields. They can be different for both different records and different fields. Their length can also be equal to `minLength` or `maxLength`, however, they can be neither shorter than `minLength` nor longer than `maxLength`.

- `string` **`randomUUID`**`();`

Generates a random but undoubtedly unique string identifier. The generated string has this format:

`hhhhhhhh-hhhh-hhhh-hhhh-hhhhhhhhhhhh`

where `h` belongs to `[0-9a-f]`. In other words, you generate a hexadecimal code of a random 128bit number.

Example generated string: cee188a3-aa67-4a68-bcd2-52f3ec0329e6

- `string` **`removeBlankSpace`**`(`string *arg*`);`

  The `removeBlankSpace(string)` function takes one string argument and returns another string with white spaces removed.

- `string` **`removeDiacritic`**`(`string *arg*`);`

  The `removeDiacritic(string)` function takes one string argument and returns another string with diacritical marks removed.

- `string` **`removeNonAscii`**`(`string *arg*`);`

  The `removeNonAscii(string)` function takes one string argument and returns another string with non-ascii characters removed.

- `string` **`removeNonPrintable`**`(`string *arg*`);`

  The `removeNonPrintable(string)` function takes one string argument and returns another string with non-printable characters removed.

- `string` **`replace`**`(`string *arg*, string *regex*, string *replacement*`);`

  The `replace(string, string, string)` function takes three string arguments - a string, a regular expression, and a replacement - and replaces all regex matches inside the string with the replacement string you specified. All parts of the string that match the regex are replaced. You can also reference the matched text using a backreference in the replacement string. A backreference to the entire match is indicated as $0. If there are capturing parentheses, you can reference specifics groups as $1, $2, $3, etc.

  `replace("Hello","[Ll]","t")` returns `"Hetto"`

  `replace("Hello", "e(l+)", "a$1")` returns `"Hallo"`

  Important - please beware of similar syntax of $0, $1 etc. While used inside the replacement string it refers to matching regular expression parenthesis (in order). If used outside a string, it means a reference to an input field. See other example:

  `replace("Hello", "e(l+)", $0.name)` returns `HJohno` if input field "name" on port 0 contains the name "John".

You can also use modifier in the start of the regular expression: `(?i)` for case-insensitive search, `(?m)` for multiline mode or `(?s)` for "dotall" mode where a dot (".") matches even a newline character

`replace("Hello", "(?i)L", "t")` will produce `Hetto` while `replace("Hello", "L", "t")` will just produce `Hello`

- `string` **`right`**`(`string *arg*, integer *length*`);`

  The `right(string, integer)` function accepts two arguments: the first is string and the second is integer. It takes them and returns the substring of the length specified as the second argument counted from the end of the string specified as the first argument. If the input string is shorter than the *`length`* parameter, an exception is thrown and the graph fails. To avoid such failure, use the `right(string, integer, boolean)` function described below.

- `string` **`right`**`(`string *arg*, integer *length*, boolean *spacePad*`);`

  The function returns suffix of the specified length. If the input string is longer or equally long as the *`length`* parameter, the function behaves the same way as the `right(string, integer)` function. There is different behaviour if the input string is shorter than the specified length. If the 3th argument is `true`, the left side of the result is padded with blank spaces so that the result has specified length beeing right justified. Whereas if `false`, the input string is returned as the result with no space added.

- `string` **`soundex`**`(`string *arg*`);`

  The `soundex(string)` function takes one string argument and converts the string to another. The resulting string consists of the first letter of the string specified as the argument and three digits. The three digits are based on the consonants contained in the string when similar numbers correspond to similarly sounding consonants. Thus, `soundex("word")` returns `"w600"`.

- `string[]` **`split`**`(`string *arg*, string *regex*`);`

  The `split(string, string)` function accepts two string arguments. The second is some regular expression. It is searched in the first string argument and if it is found, the string is split into the parts located between the characters or substrings of such a regular expression. The resulting parts of the string are returned as a list of strings. Thus, `split("abcdefg", "[ce]")` returns `["ab", "d", "fg"]`.

- `string` **`substring`**`(`string *arg*, integer *fromIndex*, integer *length*`);`

  The `substring(string, integer, integer)` function accepts three arguments: the first is string and the other two are integers. The function takes the arguments and returns a substring of the defined length obtained from the original string by getting the `length` number of characters starting from the position defined by the second argument. Thus, `substring("text", 1, 2)` returns `"ex"`.

- `string` **`translate`**`(`string *arg*, string *searchingSet*, string *replaceSet*`);`

  The `translate(string, string, string)` function accepts three string arguments. The number of characters must be equal in both the second and the third arguments. If some character from the string specified as the second argument is found in the string specified as the first argument, it is replaced by a character taken from the string specified as the third argument. The character from the third string must be at the same position as the character in the second string. Thus, `translate("hello", "leo", "pii")` returns `"hippi"`.

- `string` **`trim`**`(`string *arg*`);`

  The `trim(string)` function takes one string argument and returns another string with leading and trailing white spaces removed.

- `string` **`unescapeUrl`**`(`string *arg*`);`

  The function decodes escape sequences of illegal characters within components of specified URL (see isUrl() CTL function for the URL component description). Escape sequences consist of a percent character `%` symbol, followed by the two-digit hexadecimal representation (case-insensitive) of the ISO-Latin code point for the character, e.g., `%20` is the escaped encoding for the US-ASCII space character.

- `string` **`upperCase`**`(`string *arg*`);`

  The `upperCase(string)` function takes one string argument and returns another string with cases converted to upper cases only.

Container Functions

When you work with containers (`list`, `map`, `record`), you may use the following functions:

- `elemenettype[]` **`append`**`(`<element type>[] *arg*, <element type> *list_element*`);`

  The `append(<element type>[], <element type>)` function accepts two arguments: the first is a list of any element data type and the second is of the element data type. The function takes the second argument and adds it to the end of the first argument. The function returns the new value of list specified as the first argument.

  This function is alias of the `push(<element type>[], <element type>)` function. From the list point of view, `append()` is much more natural.

- `void` **`clear`**`(`<element type>[] *arg*`);`

  The `clear(<element type>[])`function accepts one list argument of any element data type. The function takes this argument and empties the list. It returns void.

- `void` **`clear`**`(`map[<type of key>,<type of value>] *arg*`);`

  The `clear(map[<type of key>,<type of value>])`function accepts one map argument. The function takes this argument and empties the map. It returns void.

- `<element type>[]` **`copy`**`(`<element type>[] *arg*, <element type>[] *arg*`);`

The `copy(<element type>[], <element type>[])` function accepts two arguments, each of them is a list. Elements of both lists must be of the same data type. The function takes the second argument, adds it to the end of the first list and returns the new value of the list specified as the first argument.

- `void` **`copyByName`**`(`record *to*, record *from*`);`

  Copies data from the input record to the output record based on field names. Enables mapping of equally named fields only.

- `void` **`copyByPosition`**`(`record *to*, record *from*`);`

  Copies data from the input record to the output record based on fields order. The number of fields in output metadata decides which input fields (beginning the first one) are mapped.

- `map[<type of key>, <type of value>]` **`copy`**`(`map[<type of key>, <type of value>] *arg*, map[<type of key>, <type of value>] *arg*`);`

  The `copy(map[<type of key>, <type of value>], map[<type of key>, <type of value>])` function accepts two arguments, each of them is a map. Elements of both maps must be of the same data type. The function takes the second argument, adds it to the end of the first map replacing existing key mappings and returns the new value of the map specified as the first argument.

- `list[]` **`getKeys`**`(`map[<type of key>, <type of value>] *arg*`);`

  The function returns a `list` of your map's *keys*. Remember the list has to be the same type as map's keys, e.g.:

```
map[string, integer] myMap;
```

- `<element type>[]` **`insert`**`(`<element type>[] *arg*, integer *position*, <element type> *newelement*`);`

  The `insert(<element type>[], integer, <element type> )`function accepts the following arguments: the first is a list of any element data type, the second is integer, and the other is of the element data type. The function takes the third argument and inserts it to the list at the position defined by the second argument. The list specified as the first argument changes to this new value and it is returned by the function. Remember that the list element are indexed starting from 0.

- `boolean` **`isEmpty`**`(`<element type>[] *arg*`);`

  The `isEmpty(<element type>[])`function accepts one argument of list of any element data type. It takes this argument, checks whether the list is empty and returns `true`, or `false`.

- `boolean` **`isEmpty`**`(`map[<type of key>,<type of value>] *arg*`);`

  The `isEmpty(map[<type of key>,<type of value>])`function accepts one argument of a map of any value data types. It takes this argument, checks whether the map is empty and returns `true`, or `false`.

- `integer` **`length`**`(`structuredtype *arg*`);`

  The `length(structuredtype)` function accepts a structured data type as its argument: `string`, `<element type>[]`, `map[<type of key>,<type of value>]` or `record`. It takes the argument and returns a number of elements forming the structured data type.

- `<element type>` **`poll`**`(`<element type>[] *arg*`);`

  The `poll(<element type>[])`function accepts one argument of list of any element data type. It takes this argument, removes the first element from the list and returns this element. The list specified as the argument changes to this new value (without the removed first element).

- `<element type>` **`pop`**`(`<element type>[] *arg*`);`

  The `pop(<element type>[])`function accepts one argument of list of any element data type. It takes this argument, removes the last element from the list and returns this element. The list specified as the argument changes to this new value (without the removed last element).

- `<element type>[]` **`push`**`(`<element type>[] *arg*, <element type> *list_element*`);`

  The `push(<element type>[], <element type>)`function accepts two arguments: the first is a list of any data type and the second is the data type of list element. The function takes the second argument and adds it to the end of the first argument. The function returns the new value of the list specified as the first argument.

  This function is alias of the `append(<element type>[], <element type>)` function. From the stack/queue point of view, `push()` is much more natural.

- `<element type>` **`remove`**`(`<element type>[] *arg*, integer *position*`);`

  The `remove(<element type>[], integer)`function accepts two arguments: the first is a list of any element data type and the second is integer. The function removes the element at the specified position and returns the removed element. The list specified as the first argument changes its value to the new one. (List elements are indexed starting from 0.)

- `<element type>[]` **`reverse`**`(`<element type>[] *arg*`);`

  The `reverse(<element type>[])`function accepts one argument of a list of any element data type. It takes this argument, reverses the order of elements of the list and returns such new value of the list specified as the first argument.

- `<element type>[]` **`sort`**`(`<element type>[] *arg*`);`

  The `sort(<element type>[])`function accepts one argument of a list of any element data type. It takes this argument, sorts the elements of the list in ascending order according to their values and returns such new value of the list specified as the first argument.

Record functions (dynamic field access)

These functions are to be found in the **Functions** tab, section **Dynamic field access library** inside the Transform Editor.

- `integer` **`length`**`( );`

  Returns the number of fields of a record the function is called on.

- `integer` **`compare`**`(`reference *record1*, string *field1*, reference *record2*, string *field2*`);`

  Compares two fields of given records. The fields are identified by their name. The function returns an integer value which is either:

  1. < 0 ... `field2` is greater than `field1`
  2. > 0 ... `field2` is lower than `field1`
  3. 0 ... fields are equal

- `integer` **`compare`**`(`reference *record1*, integer *field1*, reference *record2*, integer *field2*`);`

  Compares two fields of given records. The fields are identified by their index (0 is the first field). The function returns an integer value which is either:

  1. < 0 ... `field2` is greater than `field1`
  2. > 0 ... `field2` is lower than `field1`
  3. 0 ... fields are equal

- `boolean` **`getBoolValue`**`(`reference *record*, integer *field*`);`

  Returns the value of a boolean field. The field is identified by its index.

- `boolean` **`getBoolValue`**`(`reference *record*, string *field*`);`

  Returns the value of a boolean field. The field is identified by its name.

- `byte` **`getByteValue`**`(`reference *record*, integer *field*`);`

  Returns the value of a byte field. The field is identified by its index.

- `byte` **`getByteValue`**`(`reference *record*, string *field*`);`

  Returns the value of a byte field. The field is identified by its name.

- `date` **`getDateValue`**`(`reference *record*, integer *field*`);`

  Returns the value of a date field. The field is identified by its index.

- `date` **`getDateValue`**`(`reference *record*, string *field*`);`

  Returns the value of a date field. The field is identified by its name.

- `decimal` **`getDecimalValue`**`(`reference *record*, integer *field*`);`

  Returns the value of a decimal field. The field is identified by its index.

- `decimal` **`getDecimalValue`**`(`reference *record*, string *field*`);`

  Returns the value of a decimal field. The field is identified by its name.

- `integer` **`getFieldIndex`**`(`reference *record*, string *field*`);`

  Returns the index (zero-based) of a field which is identified by its name. If the field name is not found in the record, the function returns -1.

- `string` **`getFieldLabel`**`(`reference *record*, integer *field*`);`

  Returns the label of a field which is identified by its index. Please note a label is not a field's name, see Field Name vs. Label vs. Description.

- `string` **`getFieldName`**`(`record *argRecord*, integer *index*`);`

  The `getFieldName(record, integer)` function accepts two arguments: record and integer. The function takes them and returns the name of the field with the specified index. Fields are numbered starting from 0.

  Important

  The `argRecord` may have any of the following forms:

  - `$<port number>.*`

    E.g., `$0.*`

  - `$<metadata name>.*`

    E.g., `$customers.*`

  - `<record variable name>[.*]`

    E.g., `Customers` or `Customers.*` (both cases, if `Customers` was declared as record in CTL.)

  - `lookup(<lookup table name>).get(<key value>)[.*]`

    E.g., `lookup(Comp).get("JohnSmith")` or `lookup(Comp).get("JohnSmith").*`

  - `lookup(<lookup table name>).next()[.*]`

    E.g., `lookup(Comp).next()` or `lookup(Comp).next().*`

- `string` **`getFieldType`**`(`record *argRecord*, integer *index*`);`

  The `getFieldType(record, integer)` function accepts two arguments: record and integer. The function takes them and returns the type of the field with the specified index. Fields are numbered starting from 0.

Important

Records as arguments look like the records for the `getFieldName()` function. See above.

- `integer` **`getIntegerValue`**`(`reference *record*, integer *field*`);`

  Returns the value of an integer field. The field is identified by its index.

- `integer` **`getIntegerValue`**`(`reference *record*, string *field*`);`

  Returns the value of an integer field. The field is identified by its name.

- `long` **`getLongValue`**`(`reference *record*, integer *field*`);`

  Returns the value of a `long` field. The field is identified by its index.

- `long` **`getLongValue`**`(`reference *record*, string *field*`);`

  Returns the value of a `long` field. The field is identified by its name.

- `number` **`getNumValue`**`(`reference *record*, integer *field*`);`

  Returns the value of a `number` field. The field is identified by its index.

- `number` **`getNumValue`**`(`reference *record*, string *field*`);`

  Returns the value of a `number` field. The field is identified by its name.

- `string` **`getStringValue`**`(`reference *record*, integer *field*`);`

  Returns the value of a `string` field. The field is identified by its index.

- `string` **`getStringValue`**`(`reference *record*, string *field*`);`

  Returns the value of a `string` field. The field is identified by its name.

- `string` **`getValueAsString`**`(`reference *record*, string *field*`);`

  Attempts to return the value of a field (no matter its type) as a common `string`. The field is identified by its name.

- `string` **`getValueAsString`**`(`reference *record*, integer *field*`);`

  Attempts to return the value of a field (no matter its type) as a common `string`. The field is identified by its index.

- `boolean` **`isNull`**`(`reference *record*, string *field*`);`

  Checks whether a given field is `null`. The field is identified by its name.

- `boolean` **`isNull`**`(`reference *record*, integer *field*`);`

  Checks whether a given field is `null`. The field is identified by its index.

- `void` **`setBoolValue`**`(`reference *record*, integer *field*, boolean *value*`);`

  Sets a `boolean` value to a field. The field is identified by its index.

- `void` **`setBoolValue`**`(`reference *record*, string *field*, boolean *value*`);`

  Sets a `boolean` value to a field. The field is identified by its name.

- `void` **`setByteValue`**`(`reference *record*, integer *field*, byte *value*`);`

  Sets a `byte` value to a field. The field is identified by its index.

- `void` **`setByteValue`**`(`reference *record*, string *field*, byte *field*`);`

  Sets a `byte` value to a field. The field is identified by its name.

- `void` **`setDateValue`**`(`reference *record*, integer *field*, date *value*`);`

  Sets a `date` value to a field. The field is identified by its index.

- `void` **`setDateValue`**`(`reference *record*, string *field*, date *value*`);`

  Sets a `date` value to a field. The field is identified by its name.

- `void` **`setDecimalValue`**`(`reference *record*, integer *field*, decimal *value*`);`

  Sets a `decimal` value to a field. The field is identified by its index.

- `void` **`setDecimalValue`**`(`reference *record*, string *field*, decimal *value*`);`

  Sets a `decimal` value to a field. The field is identified by its name.

- `void` **`setIntValue`**`(`reference *record*, integer *field*, integer *value*`);`

  Sets an `integer` value to a field. The field is identified by its index.

- `void` **`setIntValue`**`(`reference *record*, string *field*, integer *value*`);`

  Sets an `integer` value to a field. The field is identified by its name.

- `void` **`setLongValue`**`(`reference *record*, integer *field*, long *value*`);`

  Sets a `long` value to a field. The field is identified by its index.

- `void` **`setLongValue`**`(`reference *record*, string *field*, long *value*`);`

Sets a `long` value to a field. The field is identified by its name.

- `void` **`setNumValue`**`(`reference *record*, integer *field*, number *value*`);`

  Sets a `number` value to a field. The field is identified by its index.

- `void` **`setNumValue`**`(`reference *record*, string *field*, number *value*`);`

  Sets a `number` value to a field. The field is identified by its name.

- `void` **`setStringValue`**`(`reference *record*, integer *field*, string *value*`);`

  Sets a `string` value to a field. The field is identified by its index.

- `void` **`setStringValue`**`(`reference *record*, string *field*, string *value*`);`

  Sets a `string` value to a field. The field is identified by its name.

Miscellaneous Functions

The rest of the functions can be denominated as miscellaneous. They are the functions listed below.

Important

Remember that the object notation (e.g., `arg.isnull()`) cannot be used for these **Miscellaneous** functions!

For more information about object notation see Functions Reference.

- `<any type>` **`iif`**`(`boolean *con*, <any type> *iftruevalue*, <any type> *iffalsevalue*`);`

  The `iif(boolean, <any type>, <any type>)` function accepts three arguments: one is boolean and two are of any data type. Both argument data types and return type are the same.

  The function takes the first argument and returns the second if the first is true or the third if the first is false.

- `boolean` **`isnull`**`(`<any type> *arg*`);`

  The `isnull(<any type>)` function takes one argument and returns a boolean value depending on whether the argument is null (true) or not (false). The argument may be of any data type.

  Important

  If you set the **Null value** property in metadata for any `string` data field to any non-empty string, the `isnull()` function will return `true` when applied on such string. And return `false` when applied on an empty field.

For example, if `field1` has **Null value** property set to `"<null>"`, `isnull($0.field1)` will return `true` on the records in which the value of `field1` is `"<null>"` and `false` on the others, even on those that are empty.

See Null value for detailed information.

- `<any type>` **`nvl`**`(`<any type> *arg*, <any type> *default*`);`

  The `nvl(<any type>, <any type>)` function accepts two arguments of any data type. Both arguments must be of the same type. If the first argument is not null, the function returns its value. If it is null, the function returns the default value specified as the second argument.

- `<any type>` **`nvl2`**`(`<any type> *arg*, <any type> *arg_for_non_null*, <any type> *arg_for_null*`);`

  The `nvl2(<any type>, <any type>, <any type>)` function accepts three arguments of any data type. This data type must be the same for all arguments and return value. If the first argument is not null, the function returns the value of the second argument. If the first argument is null, the function returns the value of the third argument.

- `void` **`printErr`**`(`<any type> *message*`);`

  The `printErr(<any type>)` function accepts one argument of any data type. It takes this argument and prints out the `message` on the error output.

  This function works as `void printErr(<any type> arg, boolean printLocation)` with `printLocation` set to `false`.

- `void` **`printErr`**`(`<any type> *message*, boolean *printLocation*`);`

  The `printErr(type, boolean)` function accepts two arguments: the first is of any data type and the second is boolean. It takes them and prints out the `message` and the location of the error (if the second argument is `true`).

- `void` **`printLog`**`(`level *loglevel*, <any type> *message*`);`

  The `printLog(level, <any type>)` function accepts two arguments: the first is a log level of the `message` specified as the second argument, which is of any data type. The first argument is one of the following: `debug`, `info`, `warn`, `error`, `fatal`. The log level must be specified as a constant. It can be neither received through an edge nor set as variable. The function takes the arguments and sends out the `message` to a logger.

- `void` **`raiseError`**`(`string *message*`);`

  The `raiseError(string)` function takes one string argument and throws out error with the `message` specified as the argument.

- `void` **`sleep`**`(`long *duration*`);`

  The function pauses the execution for specified milliseconds.

Lookup Table Functions

In your graphs you are also using lookup tables. You need to use them in CTL by specifying the name of the lookup table and placing it as an argument in the `lookup()` function.

Warning

Remember that you should not use the functions shown below in the `init()`, `preExecute()`, or `postExecute()` functions of CTL template.

Now, the key in the function below is a sequence of values of the field names separated by comma (not semicolon!). Thus, the key is of the following form: `keyValuePart1,keyValuePart2,...,keyValuePartN`.

See the following options:

- `lookup(<lookup name>).get(keyValue)[.<field name>|.*]`

  This function searches the first record whose key value is equal to the value specified in the `get(keyValue)` function.

  It returns the record of the lookup table. You can map it to other records in CTL (with the same metadata). If you want to get the value of the field, you can add the `.<field name>` part to the expression or `.*` to get the values of all fields.

- `lookup(<lookup name>).count(keyValue)`

  If you want to get the number of records whose key value equals to `keyValue`, use the syntax above.

- `lookup(<lookup name>).next()[.<field name>|.*]`

  After getting the number of duplicate records in lookup table using the `lookup().count()` function, and getting the first record with specified key value using the `lookup().get()` function, you can work (one by one) with all records of lookup table with the same key value.

  You need to use the syntax shown here in a loop and work with all records from lookup table. Each record will be processed in one loop step.

  The mentioned syntax returns the record of the lookup table. You can map it to other records in CTL (with the same metadata). If you want to get the value of the field, you can add the `.<field name>` part to the expression or `.*` to get the values of all fields.

Example 55.9. Usage of Lookup Table Functions

```
//#CTL  // record with the same metadata as those of lookup table
recordName1 myRecord;  // variable for storing number of duplicates
integer count;  // Transforms input record into output record. function
integer transform() {       // if lookup table contains duplicate
records,       // their number is returned by the following expression
// and assigned to the count variable      count =
lookup(simpleLookup0).count($0.Field2);       // getting the first
record whose key value equals to $0.Field2      myRecord =
lookup(simpleLookup0).get($0.Field2);       // loop for searching the
last record in lookup table      while ((count-1) > 0) {       //
searching the next record with the key specified above      myRecord =
lookup(simpleLookup0).next();       // incrementing counter      count-
-;      }       // mapping to the output       // last record from
lookup table      $0.Field1 = myRecord.Field1;      $0.Field2 =
myRecord.Field2;       // corresponding record from the edge
$0.Field3 = $0.Field1;      $0.Field4 = $0.Field2;         return 0; }
```

Warning

In the example above we have shown you the usage of all lookup table functions. However, we suggest you using other syntax for lookup tables.

The reason is that the following expression of CTL:

```
lookup(Lookup0).count($0.Field2);
```

searches the records through the whole lookup table which may contain a great number of records.

The syntax shown above may be replaced with the following loop:

```
myRecord = lookup(<name of lookup table>).get(<key value>);
while(myRecord != null) {   process(myRecord);   myRecord =
lookup(<name of lookup table>).next(); }
```

Especially DB lookup tables can return -1 instead of real count of records with specified key value (if you do not set **Max cached size** to a non-zero value).

Important

Remember that DB lookup tables cannot be used in compiled mode! (code starts with the following header: `//#CTL:COMPILE`)

You need to switch to interpreted mode (with the header: `//#CTL`) to be able to access DB lookup tables from CTL.

Sequence Functions

In your graphs you are also using sequences. You can use them in CTL by specifying the name of the sequence and placing it as an argument in the `sequence()` function.

Warning

Remember that you should not use the functions shown below in the `init()`, `preExecute()`, or `postExecute()` functions of CTL template.

You have three options depending on what you want to do with the sequence. You can get the current number of the sequence, or get the next number of the sequence, or you may want to reset the sequence numbers to the initial number value.

See the following options:

- `sequence(<sequence name>).current()`
- `sequence(<sequence name>).next()`
- `sequence(<sequence name>).reset()`

Although these expressions return integer values, you may also want to get long or string values. This can be done in one of the following ways:

- `sequence(<sequence name>,long).current()`
- `sequence(<sequence name>,long).next()`
- `sequence(<sequence name>,string).current()`
- `sequence(<sequence name>,string).next()`

CTL Appendix - List of National-specific Characters

Several functions, e.g. editDistance (string, string, integer, string, integer) need to work with special national characters. These are important especially when sorting items with a defined comparison strength.

The list below shows first the locale and then a list of its national-specific derivatives for each letter. These may be treated either as equal or different characters depending on the comparison stregth you define.

Table 55.2. National Characters

Locale	National Characters

CA - Catalan	`"a=à=A=À",`
	`"e=è=é=E=È=É",`
	`"i=í=ï=I=Í=Ï",`
	`"o=ò=ó=O=Ò=Ó",`
	`"u=ú=ü=U=Ú=Ü",`
	`"c=ç=C=Ç"`

Locale	National Characters
CZ - Czech	"a=á=A=Á", "c=č=C=Č", "d=ď=D=Ď", "e=é=ě=E=É=Ě", "i=í=I=Í", "n=ň=N=Ň", "o=ó=O=Ó", "r=ř=R=Ř", "s=š=S=Š", "t=ť=T=Ť", "u=ů=ú=U=Ů=Ú", "y=ý=Y=Ý", "z=ž=Z=Ž"
DA - Danish and Norwegian	"a=æ=å=A=Æ=Å", "o=ø=O=Ø"
DE - German	"a=ä=A=Ä", "o=ö=O=Ö", "u=ü=U=Ü"
ES - Spanish	"a=á=A=Á", "e=é=E=É", "i=í=I=Í", "o=ó=O=Ó", "u=ú=ü=U=Ú=Ü", "n=ñ=N=Ñ"
ET - Estonian	"a=ä=A=Ä", "o=ö=õ=O=Ö=Õ", "u=ü=U=Ü", "s=š=S=Š", "z=ž=Z=Ž"
FI - Finnish	"a=ä=A=Ä", "o=ö=O=Ö"

Locale	National Characters
FR - French	"a=à=â=A=À=Â", "e=è=é=ê=ë=E=È=É=Ê=Ë", "i=ï=I=Ï", "o=ô=O=Ô", "u=ù=û=ü=U=Ù=Û=Ü", "c=ç=C=Ç"
HR - Croatian	"c=ć=č=C=Ć=Č", "d=đ=D=Đ", "s=š=S=Š", "z=ž=Z=Ž"
HU - Hungarian	"a=á=A=Á", "e=é=E=É", "i=í=I=Í", "o=ó=ö=ő=O=Ó=Ö=Ő", "u=ú=ü=ű=U=Ú=Ü=Ű"
IS - Icelandic	"a=á=æ=A=Á=Æ", "e=é=E=É", "i=í=I=Í", "o=ó=ö=O=Ó=Ö", "u=ú=U=Ú", "y=ý=Y=Ý", "d=ð=D=Ð"
IT - Italian	"a=à=A=À", "e=é=è=E=É=È", "i=ì=I=Ì", "o=ò=O=Ò", "u=ù=U=Ù"

Locale	National Characters
LV - Latvian	"a=ā=A=Ā", "e=ē=E=Ē", "i=ī=I=Ī", "u=ū=U=Ū", "c=č=C=Č", "g=ģ=G=Ģ", "k=ķ=K=Ķ", "l=ļ=L=Ļ", "n=ņ=N=Ņ", "s=š=S=Š", "z=ž=Z=Ž"
PL - Polish	"a=ą=A=Ą", "c=ć=C=Ć", "e=ę=E=Ę", "l=ł=L=Ł", "n=ń=N=Ń", "o=ó=O=Ó", "s=ś=S=Ś", "z=ż=ź=Z=Ż=Ź"
PT - Portuguese	"a=ã=á=à=â=A=Ã=Á=À=Â", "e=é=ê=E=É=Ê", "i=í=I=Í", "o=õ=ó=ô=O=Õ=Ó=Ô", "u=ú=U=Ú", "c=ç=C=Ç"
RO - Romanian	"a=ă=â=A=Ă=Â", "i=î=I=Î", "s=ş=S=Ş", "t=ţ=T=Ţ"

Locale	National Characters
RU - Russian	"Е=е=Ё=ё=Е́=е́", "И=и=Й=й=И'=и́", "О=о=О́=о́", "А=а=А́=а́", "Ы=ы=Ы'=ы'", "Я=я=Я'=я́", "Ю=ю=Ю'=ю'", "У=у=У'=у́", "Э=э=Э'=э́"
SK - Slovak	"a=á=ä=A=Á=Ä", "c=č=C=Č", "d=ď=D=Ď", "e=é=E=É", "i=í=I=Í", "l=ĺ=I=L=Ĺ=L", "n=ň=N=Ň", "o=ó=ô=O=Ó=Ô", "r=ŕ=R=Ŕ", "s=š=S=Š", "t=ť=T=Ť", "u=ú=U=Ú", "y=ý=Y=Ý", "z=ž=Z=Ž"
SL - Slovenian	"c=č=C=Č", "s=š=S=Š", "z=ž=Z=Ž"
SQ - Albanian	"e=ë=E=Ë", "c=ç=C=Ç"
SV - Swedish	"a=ä=å=A=Ä=Å", "o=ö=O=Ö"

Chapter 56. Regular Expressions

A *regular expression* is a formalism used to specify a set of strings with a single expression. Since the implementation of regular expressions comes from the Java standard library, the syntax of expressions is the same as in Java.

Example 56.1. Regular Expressions Examples

`[p-s]{5}`

- means the string has to be exactly five characters long and it can only contain the `p`, `q`, `r` and `s` characters

`[^a-d].*`

- this example expression matches any string which starts with a character other than `a`, `b`, `c`, `d` because
    - the `^` sign means exception
    - `a-d` means characters from `a` to `d`
    - these characters can be followed by zero or more (`*`) other characters
    - the dot stands for an arbitrary character

For more detailed explanation of how to use regular expressions see the Java documentation for `java.util.regex.Pattern`.

The meaning of regular expressions can be modified using embedded flag expressions. The expressions include the following:

`(?i)` – `Pattern.CASE_INSENSITIVE`
  Enables case-insensitive matching.
`(?s)` – `Pattern.DOTALL`
  In dotall mode, the dot `.` matches any character, including line terminators.
`(?m)` – `Pattern.MULTILINE`
  In multiline mode you can use `^` and `$` to mean the beginning and end othe line, respectively (that includes at the beginning and end of the entire expression).

Further reading and description of other flags can be found at http://docs.oracle.com/javase/tutorial/essential/regex/pattern.html.