Welcome to Nicl

Nicl is *NEON's Ingest Conversion Language,* a scripting language used to map and filter one set of incoming values into another set of values. Nicl is meant to handle these values in a single batch.

The Nicl Engine

The main driver to process data through the Nicl language is the ***Nicl Engine***. The Nicl Engine works on a set of registers called the Nicl ***Register Frame***.

Each register starts with some initial value. All registers in the Register frame are evaluated left-to-right as a batch, applying to each register its customized script of Nicl functions to arrive at each output value.

Each register is named. The results of these registers are further processed and persisted to a database or otherwise put into some document and displayed according to the context of that register name.

Register		Initial value		Nicl Script	
`loc`	`←(`	`3`	`)[`	`IDENT`	`];`
`tax`	`←(`	`4`	`)[`	`IDENT`	`];`
`col`	`←(`		`)[`	`loc + tax`	`];`

Table 1 Example Nicl Register Frame

The figure above shows a sample Register Frame. This frame has three named registers: `loc`, `tax`, and `col`. `loc` and `tax` are given the initial values `3` and `4` respectively. `Col` does not have an initial value. Instead, `col` is a calculated as the sum of `loc` and `tax`, as its Nicl Script shows. `loc` and `tax` have only the call to the `IDENT` function. The output result set of this small example is expected to be:

```
(loc:3|tax:4|col:7).
```

The evaluations of all registers in a frame are done as a batch. The ordering in which these registers are written are not important. The example register frame above could have been in the order: `tax, col, loc`, and the output would have been the same. This despite that the equation for `col` uses values that are defined both before and after it.

Os L0 Parser Application

The Os L0 Parser Application (OsParser) uses the Nicl Engine to process Level 0 data from the field, perform some transform and validation on the data, and then save the output as Activity information in Neon's PDR database.

OsParser gathers records from a set of Fulcrum UI query requests, or from rows of a set of comma-separated spreadsheet (CSV) files, or from some other resource capable of providing a stream of flat records of field data.

Each record of field data is evaluated in its own Nicl Register Frame. The record's data is used to populate the initial values of the Register Frame.

Example

I have an input CSV file containing four rows: One header row and three rows of data. I want to process this CSV file using t*he Table 1 Example Nicl Register Frame* from the previous section.

	A	B
1	loc	tax
2	30	7
3	31	5
4	28	4

Table 2 Input 0 data as CSV

That the first row is the presumed header row, and the names in each header cell (loc, tax) matches the name of the registers in the Register Frame that are to be initialized. The Register Frame entries can be in arbitrary order, and so too the header columns. The above CSV columns could have been reversed.

When the Nicl Engine runs through, we expect the following three result sets, one for each data row:

```
1   (loc:30|tax:7|col:37)
2   (loc:31|tax:5|col:36)
3   (loc:28|tax:4|col:32)
```

From here OsParser will do further processing and data threading of this output and ultimately generate Activity database records having this data.

OsParser Ingest Worksheets

Bundled with the raw Level 0 field data are *metadata* that identify which Register Frame to use to process its particular payload. There are potentially hundreds of different types of CSV form data, and there is a customized Register Frame for each.

For OsParser, each Register Frame is defined within an ***Ingest Worksheet***. Within each Ingest Worksheet defines one particular Register Frame, and each frame comprises:

- The list of registers that compose the frame. This includes both the input fields and calculated fields.
- The name of each register, which is the Ingest Worksheet field's ***Field Name.***
- Custom Nicl script for each register.
- Mapping constructs that tell for example how to match csv header names to register names in case they don't match one for one.
- Mapping constructs that tell how to gather records from the Fulcrum UI's Json document.

The Ingest Worksheet also contains workflow information on how to process batches of information and how that information is grouped, data types and underlying context types for each field, and more information that won't be covered in this section.

For OsParser, an Ingest Worksheet (and therefore a particular Register Frame) is defined by a *Data Product ID/Table Name* combination, for example:

```
NEON.DOM.SITE.DP0.20066.001/apl_clipHarvest_in
```

See [HERE] for more details on the Ingest Worksheet and how it is processed.

Nicl Script Introduction

The basic Nicl construct is the ***Statement***. A *Statement* should be thought of a pipeline of expressions that transform from one value into another, from left to right. A Statement can generate an *output value* most easily by just supplying a literal value or literal expression such as:

```
4 * 3
```

This will be evaluated and will generate an output value of `12`. A Statement can otherwise be an *If Statement* or a *Case Statement*, which will be discussed later. All other Statements, like `4*3` are *Expressions*.

Expressions

An expression can be some simple value, such as

```
'George'
```

The literal string of characters "George".

An Nicl expression may be an *Identifier*, which may be the name of a 0-argument function such as `TRUE`, or the name of some value in the Nicl environment such as

```
systemDate
```

An expression may be some more elaborate construct, a combination of identifiers, literal data, data structures, functions with or without parameters:

```
myAgeYears / 2 + 7 >= THRESHOLD(yourAgeYears)
```

By the usual naming convention `myAgeYears` and `yourAgeYears` look to be value identifiers whose values come from registers of the same name in a Register Frame. `THRESHOLD` is a function taking one parameter. This function may be a "canned" function that is available in general, or may be a function written by *you*! More on that later.

From the expression above `/`, `+` and `>=` are each *operators*. They represent the familiar divide, plus, and greater-than-or-equal-to operations. These are binary infix left-to-right (L-R) operators, meaning the operator takes two operands, evaluates the left-hand operand before the right-hand side, and sits typographically between its two operands. The usual operator precedence with which you are most familiar applies to Nicl expressions. From the expression above divide (`/`) takes a higher precedence than both `+` and `>=` and so `myAgeYears / 2` is evaluated first. Similarly the addition is performed before the comparison is made. Incidentally, the parentheses in the term `THRESHOLD(yourAgeYears)` can also be considered an operator, and

it is bound most tightly, and will evaluate and bind its operators to the function and evaluate that function ahead of any of the other operators around it.

Control Flow

To move beyond a single expression Nicl allows you to execute a simple series of expressions by putting your expression or other statement (remember that an *Expression* is a kind of *Statement* in Nicl) inside a ***Block*** and then stringing these blocks together forming a ***Block Chain.*** Blocks are formed by encasing expressions in square brackets ('`[`' and '`]`') and then chaining them together left to right like this:

```
[ Statement1 ] [ Statement2 ] [ Statement3 ]
```

At the head of the Nicl script, an implicit ***Incoming Value*** (***inValue***) may be provided from the left side. This is the initial *register value* as described in The Nicl Engine section above. In the case of the Os L0 Parser, this would be the cell of raw data from a CSV file from the field, for example. The *Block Chain* statement is executed left to right, with the result from one Block fed to the block on to its right, becoming the right's *inValue*. The result from the last block of the entire chain in the script will become the outcome of the *Register*.

Consider the example from above:

```
myAgeYears / 2 + 7 >= THRESHOLD(yourAgeYears)
```

This can also be enclosed in a single block like this:

```
[myAgeYears / 2 + 7 >= THRESHOLD(yourAgeYears)]
```

The value coming out of a block is by default the result of the (last) expression's result, and so this has the same outcome as the first expression. So far this hasn't bought us anything. This can be split up into a *Block Chain* like so:

```
[myAgeYears][ / 2][+7][ >= THRESHOLD(yourAgeYears)]
```

On the face of it, this doesn't look much different, and in fact looks to have merely added extra characters. But this construct has some deep consequences on how expressions are evaluated in Nicl as will become apparent.

Data Types

Strings

Numbers

Numbers in Nicl are represented as BigDecimals, carrying a default of 12 digits of precision. They can be represented in Nicl in a variety of ways, as both signed and unsigned *Integers*, floating point numbers, including exponents. Nicl numbers may also carry *Units*.

Note: Nicl's number representation is not the same as the formats of incoming data from ingest workbooks used by the OS L0 Parser and other such ancient tools.

Numeric literals are represented as a series of digits, with an optional decimal point, without enclosing the sequence with quotes or any other delimiters. The following are each valid numeric literal values:

```
0
1
9.070-
0.3
```

The biggest thing to notice here is that negative numbers are represented with a *postfix* minus sign following the digits rather than the usual prefix notation (before the digits) to which you may be used. When using the decimal point, the digits to the left and right of the dot are optional. The following are equivalent:

```
.3-
0.3-
```

As are the following. Note that leading and trailing zeroes have no special meaning. All numbers are treated as decimal base-10 except when using the hexadecimal notation discussed below.

```
9.0
9.
009.00.
```

You can express numbers as exponents using the familiar 'E' notation. The following literals each represent the value 55.340×10^3

```
55.34E3
55.340e3
55.34E03
```

Note: 'E' (or 'e') must be surrounded by digits. Also, no embedded spaces are allowed. Extraneous embedded characters are also disallowed. None of these will work:

```
55.34 E3
```

```
55 .34
3.4z4.
.
```
<= WRONG!

You can use a Hexadecimal format to represent integer values. Note that decimal notation and exponential ('E') notation does not mix with the hexadecimal representation. The following is the decimal value 65535:

```
0xffff
```

You can embed underscores ('_') in your numeric literal in order to group digits for better readability. Note that underscores used in this manner must be both preceded and followed by a digit. The underscores have no effect on the meaning of the value. All of the following have the same value:

```
1900089E-45.002103
1_900_089E-45.002_103
1_900_089e-45.002_103
```

Boolean (TRUE and FALSE) expressions

Some Neon functions and many NICL expressions evaluate to either `TRUE` or `FALSE`. This section gives several examples of how that happens.

Boolean expressions typically are not *Transformed* into a final output value, but rather are used in Nicl conditionial statements (`IF` and `CASE` statements) directing the script to perform some *Validation* or *Transform* function.

True and false constant values are represented in Nicl as the literal values `TRUE` and `FALSE` respectively. These should not be put into quotes. Boolean functions evaluate to `TRUE` and `FALSE`. `TRUE` and `FALSE` can be used in expressions directly. The equal sign `=`is Nicl's operator to check for equality between two boolean values. `!=` is Nicl's *Not equals* operator, which returns `TRUE` if its two arguments (left and right hand side expressions) are not the same. The exclamation sign `!` is Nicl's *NOT* operator, which toggles between `TRUE` and `FALSE` values. The following list of examples illustrate this.

Input	Nicl Script	Remarks

-	`[TRUE = !FALSE]`
	`[!FALSE]`
	`[TRUE]`
	`[TRUE = TRUE]`
	`[TRUE != FALSE]`
	`[TRUE = !FALSE]`
	`[FALSE != TRUE]`
	`[FALSE = FALSE]`
		These each evaluate to TR

`NOT` is a keyword in Nicl, and it is an operator that is synonymous to the `!` operator.

Input	Nicl Script	Remarks
-	`[TRUE = NOT FALSE]` `[NOT FALSE]`	These both evaluate to `TRUE` as well. Note though that `NOT=` is not a thing: it is not a substitue for Nicl's *Not equa*

Input	Nicl Script	Remark

-	`[FALSE]`
	`[TRUE != TRUE]`
	`[TRUE = FALSE]`
	`[FALSE = TRUE]`
	`[FALSE != FALSE]`
	`[TRUE != TRUE]`
	`[TRUE = !TRUE]`
	`[TRUE = NOT TRUE]`
	`[!TRUE]`
	`[NOT TRUE]`	

Lists

Lists are collections of other primitive values in Nicl, and a list can be passed around and operated on by functions as if it were a primitive value. A list is an *ordered* collection of 0 or

more values. The values within one list typically share a common base type, but that isn't a requirement.

Lists are formed in the Nicl language by listing the list elements surrounded by parentheses and separated by commas like so:

```
(a, b, c)
```

As in other places in the Nicl language, spaces after commas and whitespace in general is ignored.

Note again that the order of the list is relevant: The following lists are not equal due to the ordering of their elements:

```
(a, b, c)   <>   (a, c, b)
```

Sets

Sets in Nicl are a purer, more restricted form of list. Sets are an *unordered* collection of *unique* primitive values and/or *tuples*. Like lists, sets can be operated upon by functions that take collections to operate upon.

Sets are formed in the Nicl language by listing the elements in any order, separated by the *OR* operator like so:

```
(a OR b OR c)
```

A few things to note about sets vs lists.

- The elements of a set are *unique*. Trying to place two of the same value in a set will cause the element to collapse down to a single element.
- The parentheses in the set are *optional*, and are used here merely for expression grouping in the same way as parentheses are used to group algebraic terms.
- The vertical-bar operator (`|`) is synonymous with `OR`.

Therefore, all of the following Set expressions are equivalent:

```
(a OR b OR c)
(a | b | c)
a | b | c
a | c | b | b
a OR b | c
```

Nicl Operators

This is a list of the operators available in the Nicl language. These are listed in what is more or less highest to lowest *precedence* order. This determines how tightly bound one operator is compared to another, in the absence of grouping parentheses.

For example, because multiplication is of a *higher precedent* than addition, the following equalities hold:

```
5 * 3 + 2 = 2 + 5 * 3 = (5 * 3) + 2 = 17
```

Use parentheses to change the order of evaluation:

```
5 * (3 + 2)  = 25
```

Operator Name	Symbol	Alternate Symbol	Remarks
Function Arguments	`f()`		Highest Precedence. A function tightly binds to its argument list, if it has one.
Negate	*x* `-`		Negates a number. [4-] yields negative four. This is not to be confused with
Power	`^`		Raises one number to the power of another. [9 ^ (1/2)] yields 3.0.
Multiply	`*`		Multiplies two numbers together. [8 * 2] yields 16.0
Divide	`/`		Divides two numbers. [8 / 2] yields 4.0
Add	`+`		Adds two numbers together. [38.9 + 1.1] evaluates to 40.0
Subtract	*a* `-` *b*		Subtracts two numbers. [45 - 23] evaluates to 22.0
In	`<:`	IN	Determines if one value is in the set of another. ['grass' IN ('twigs', 'grass')] returns TRUE

Operator Name	Symbol	Alternate Symbol	Remarks
Contains	:>	CONTAINS	Determines if one set contains some value (the inverse of IN). [('twigs', 'grass') CONTAINS 'weeds'] returns FALSE
Less Than	<		Compares two numbers. [3 < 5] yields TRUE, for example.
Less Than or Equal	<=		Compares two numbers. [5 <= 5] returns TRUE.
Greater Than	>		Compares two numbers. [3 > 5] returns FALSE.
Greater Than or Equal	>=		Compares two numbers. [5 >= 5] returns TRUE. Not to be confused with the forwarding arrow of the case statement, which is =>
Equal	=		Compares any two things for equality.
Not Equal	!=		Compares any two things for inequality.
Almost Equal	~=		Compares any two numbers, and returns true if their absolute difference is within 1/10000th.
Logical Not	!	NOT	returns the inverse of a logical (true/false) value, so that TRUE becomes FALSE and vice versa.
Logical And	&	AND	Compares two logical (true/false) values and returns TRUE if both are TRUE
Logical Or	|	OR	Compares two logical (true/false) values and returns TRUE if either is TRUE
List Or	|	OR	Builds a list of values, ['Me' OR 'You'] yields the set ('Me', 'You')
Bind	->		Binds a value to a temporary variable in one field's script

Operator Name	Symbol	Alternate Symbol	Remarks
Assign	`<-`		Assigns a value to a form-level identifier.
If statement	`IF` *`cond`* `,` *`stmt`*		Evaluates the boolean conditional expression `cond`, and if `TRUE` executes the statement.
Case statement	`?` `cond` `=>` *`stmt`*	`CASE` *`cond`* `=>` *`stmt`*	Evaluates the boolean conditional expression `cond`, and if `TRUE` executes the statement

Nicl Algebra Examples

The following rows contain the results from various Nicl statements and expressions.

The first column is the NICL script that will be evaluated. The second column are the results, followed by any remarks.

Nicl Script	Results	Remarks
[1 + 3]	4.0	'+' adds two numbers
[(5-) * 4] [5- * 4] [(0-5) * 4] [4 * (5-)] [4 * 5-]	-20.0	'*' multiplies two numbers, '-' subtracts. '-' is also a "unary" negation operator Each of these lines are equivalent
[8 * 3 - 1 / 2 ^ 2]	23.75	'/' divides. '^' is the power function. The usual *Operator Precedence* applies.
[(8 * 3) - (1 / (2 ^ 2))]	23.75	Parentheses are used for grouping expressions. This grouping is equivalent to the one above.
[8 * (3 - 1) / 2 ^ 2]	4.0	
[8 * (3 - 1 / 2) ^ 2]	50.0	
[(8 * 3 - 1 / 2) ^ 2]	552.25	
[8 * ((3 - 1) / 2) ^ 2]	8.0	

Nicl Script	Results	Remarks
[(8 * ((3 - 1) / 2)) ^ 2] [(8 * (3 - 1) / 2) ^ 2]	64.0	

Nicl as a Desktop Calculator

TODO

Negation and the Minus Operator

TODO

Transform Functions

This section provides the *Transform Functions* from both the Nicl core library and the Neon custom library. Transform functions are those that convert its input value(s) to some other value. A Transform function fails only on bad input. Boolean functions also convert its input values to some other value, namely `TRUE` or `FALSE`, but that is a large enough slice of functionality that they are listed separately. Also, Boolean functions typically are not used to transform data to output values, but are used as the conditional parameter to `IF` and `CASE` statements.

Nicl Validation Functions

This lists the Validation Functions that are part of the Nicl core library and the Neon custom library.

Validation Functions differ from Boolean Functions. Boolean functions work on its arguments and return either `TRUE` or `FALSE`. Boolean Functions fail only on bad input.

A Validation function on the other hand determines whether its input is valid or not. When valid, the Validation function "passes through" its input value (coming in from the left) to its output result (going out of the right). When invalid, the Validation function short-circuits the field evaluation, and delivers a FAIL state.

Most of the following functions allow an "implied" first parameter, which is the value that is coming in from the immediate left of the expression. This is called the **invalue**. A function that takes a single "implied" parameter means that it will take no parameters when the invalue is to be used, otherwise it takes a single explicit value, in which case the invalue is ignored. Similarly, a function that takes two parameters, where the first one is "implied" means that it will take 1 parameter when the invalue is to be used, otherwise it takes two explicit values, in which case the invalue is ignored. And so on.

In the listing below, the implied functions are marked with a special bullet (•) as its first parameter.

Nicl "Canned" Functions

COUNT

COUNT's function footprint	
Signature:	Explicit 0-arg
Validating:	No
Exceptions:	None
Result:	Integer

Function COUNT counts all of the non-blank elements in the given parameter list. The parameters can be any type. If a parameter is a list of items (such as ('grass', 'leaves')) then COUNT does a deep dive into these items and counts each of those elements.

Each of the parameters, whether in comes from the explicit parameter list or from the implicit value (the **invalue**), is assumed to be a tuple of elements. Each of the elements of that tuple is counted separately and *deeply*. If the value is not a tuple, but is instead VOID or an Error, it counts as 0. All other non-tuple values count as 1.

COUNT does not fail. COUNT is part of the Nicl Canned Library.

COUNT Signature

```
COUNT
COUNT( @ )
```

Counts the list of elements in the implicit value (the **invalue**), treated as a tuple of values. The second form COUNT(@) does the same thing explicitly. Note that the at-sign symbol (@, "the sponge") indicates the **invalue**.

```
COUNT( param, ...)
```

The first form (without any parameters) counts the implicit value (the **invalue**). The second form COUNT(@) does the same thing explicitly. Note that the at-sign symbol (@) indicates the **invalue**.

COUNT Example

Given these variable assignments:

```
empty <- ''
plotId <- 'ltr.01'
sampleType <- 'soil'
after <- 10
```

The following expressions are equivalent. Each evaluate to 5, because nope is VOID, and not counted:

```
[COUNT(empty | nope | plotId | sampleType | after | 'foo')]
[(empty | nope | plotId | sampleType | after | 'foo')][COUNT]
[(empty | nope | plotId | sampleType | after | 'foo');COUNT]
[(empty | nope | plotId | sampleType | after | 'foo');COUNT(@)]
```

IDENT

FALSE

TRUE

IS_BLANK

Function IS_BLANK returns `TRUE` if and only if its parameter is null, empty or blank.

`IS_BLANK( ● )`

The function takes a single, implied input value of any type. It does not fail. `IS_BLANK` is part of the Neon Custom Library.

Input	Nicl Script	Result	Remarks
-	`[IS_BLANK('')]`	`TRUE`	Explicit blank string argument
-	`[IS_BLANK(nope)]`	`TRUE`	Assuming the `nope` identifier isn't d anywhere
`empty <- ' '`	`[IS_BLANK(empty)]`	`TRUE`	Explicit parameter's value is blank.
invalue <- 4.8	`[IS_BLANK]`	`FALSE`	Implicit `invalue` is *not* blank.
invalue <- ''	`[IS_BLANK]`	`TRUE`	Implicit `invalue` is blank.
-	`[42.3][IS_BLANK]`	`FALSE`	Implicit version

Input	Nicl Script	Result	Remarks
-	`[42.3;IS_BLANK]`	`FALSE`	Implicit version
-	`[' '; IS_BLANK]`	`TRUE`	Implicit version

IS_NOT_BLANK

Function IS_NOT_BLANK returns `TRUE` if and only if its parameter is *not* null, empty or blank.

`IS_NOT_BLANK( ● )`

The function takes a single, implied input value of any type. It does not fail. `IS_NOT_BLANK` is part of the Neon Custom Library.

Input	Nicl Script	Result	Remarks
-	`[IS_NOT_BLANK('')]`	`FALSE`	Explicit blank string argument
-	`[IS_NOT_BLANK(nope)]`	`FALSE`	Assuming the `nope` identifier is anywhere
`empty <- ' '`	`[IS_NOT_BLANK(empty)]`	`FALSE`	Explicit parameter's value is bla
invalue <- 'Hi'	`[IS_NOT_BLANK]`	`TRUE`	Implicit `invalue` is *not* blank.
invalue <- ''	`[IS_NOT_BLANK]`	`FALSE`	Implicit `invalue` is blank.

MATCHES
MATCHES_EXACT
STARTS_WITH
ENDS_WITH
VOID
INT
NUM
BOOL
STRING
DATE
LIST
SET
MICROSECONDS
MILLISECONDS
SECONDS
MINUTES
HOURS
DAYS
PI
NEG
ABS
MIN
MAX
SUM
MEAN
DECR
INCR
SQRT
VARIANCE
STDDEV
PVARIANCE
PSTDDEV

Neon Custom Functions

This table describes the set of canned NEON functions and their usage. When Nicl evaluates any function there shall always be an implicitly provided *upstreamValue*. The *upstreamValue* comes either from the result of the previous expression in this statement, or comes from the value of a *Tablet Form Field*. Many functions have an *Implicit Form* which assumes that it will use the *upstreamValue* as its argument, as well as a secont *Explicit Form* in which an explicit value is provided as the first argument to be processed instead of the *upstreamValue*. Each form of each Neon function is detailed below.

NA

Alias for IDENT

CREATE_UID

This *Transform Function* generates a unique universal user id (UUID) string of the form `"7e4f528c-7ee3-4f80-a54e-0cb623fa3893"`.

Syntax

```
[CREATE_UID]
```

This function takes no parameters and returns a UID string. The *upstreamValue* is ignored.

Usage

```
(_) -> [CREATE_UID] -> ('9ac412e4-ff23-0234-ff67-ff65a2b01cd4')
```

DEFAULT_TO

Transform function provides a default value to this statement block when the *upstreamValue* is missing or empty. This is an *implicit* 2-argument function: When only one argument is provided, it is assumed that the first argument should be the *upstreamValue*.

Implicit Syntax

`[DEFAULT_TO(` ***default-value*** `)]`

Explicit Syntax

`[DEFAULT_TO(` ***in-value*** `,` ***default-value*** `)]`

Here *default-value* and *in-value* can be any expressions. For the *Implicit Syntax* form the *upstreamValue* is used as the *in-value*. The *in-value* is evaluated. If it is `VOID` or undefined, or

if its trimmed string value is empty, then *default-value* is used as this function's result. Otherwise *in-value* is the result.

Usage

```
() -> [DEFAULT_TO(45.6)] -> (45.6)

('active') -> [DEFAULT_TO('inactive')] -> ('active')

('2006-03-15 14:00') -> [AS_DATE][DEFAULT_TO(NOW)] -> (2006-03-15T14:00)

(_) -> [DEFAULT_TO('good', 'bad')] -> ('good')

(_) -> [DEFAULT_TO((), 'inactive')] -> ('inactive')
```

SPLIT_BY

Transform function splits a value's string into an ordered tuple of values. This resultant tuple is suitable for further processing downstream.

Implicit Syntax

```
[SPLIT_BY( default-value )]
```

Explicit Syntax

```
[SPLIT_BY( in-value , default-value )]
```

Here *default-value* and *in-value* can be any valid string expressions. For the *Implicit Syntax* form the *upstreamValue* is used as the *in-value*. The string value of *in-value* is split into a sequence of individual string values at the occurrence of the *default-value* separator, which is typically a comma or a vertical bar. The separators are not included in the output. The split strings are trimmed when put into the output. Empty strings, resulting from two adjacent separators, are allowed in the output.

Usage

```
('a|b|c') -> [SPLIT_BY('|')] -> ('a','b','c')

(_) -> [SPLIT_BY('start by | ending-by | who', '|')] -> ('start by','ending-
by','who')

(_) -> [SPLIT_BY(' ; start by ; ending-by ; who;', ';')] -> ('','start by','ending-
by','who','')
```

REQUIRE

Validating Function REQUIRE determines if its parameter is not blank and not an error. The function "passes through" the input value to the output result on success. The function FAILs if there the parameter is an error or is blank.

```
REQUIRE( ● )
```

`REQUIRE` is part of the Neon Custom Library.

REQUIRE_NULL

Validating Function REQUIRE_NULL determines if its parameter is blank or is an error. The function "passes through" the input value to the output result on success. The function FAILs if there the parameter is not null.

```
REQUIRE_NULL( ● )
```

`REQUIRE_NULL` is part of the Neon Custom Library.

DICT

LOV

CONVERT_TO_UTC

NAMED_LOCATION_TYPE

Validating Function NAMED_LOCATION_TYPE

```
NAMED_LOCATION_TYPE( ● , list-of-named-location-types)
```

`NAMED_LOCATION_TYPE` is part of the Neon Custom Library.

ELEMENT_OF

Validating Function ELEMENT_OF

```
ELEMENT_OF( ●, arg-list ... )
```

`ELEMENT_OF` is part of the Neon Custom Library.

GREATER_THAN

GREATER_THAN_OR_EQUAL_TO

LESS_THAN

LESS_THAN_OR_EQUAL_TO

DEFAULT_TO

DERIVE_FROM_SAMPLE_TREE

MATCH_REGULAR_EXPRESSION

Validating Function MATCH_REGULAR_EXPRESSION passes through its first implicit parameter if that first parameter is found anywhere within the Regular expression pattern given by the second parameter. The function "passes through" the input value to the output result on success. The function FAILs if there is not a match. Both parameters are treated as strings.

```
MATCH_REGULAR_EXPRESSION( ● , pattern)
```

`MATCH_REGULAR_EXPRESSION` is part of the Neon Custom Library.

NOT_MATCH_REGULAR_EXPRESSION

Validating Function NOT_MATCH_REGULAR_EXPRESSION passes through its first implicit parameter if that first parameter is *not* found anywhere within the Regular expression pattern given by the second parameter. The function "passes through" the input value to the output result on success. The function FAILs if there is there is no match. Both parameters are treated as strings.

```
NOT_MATCH_REGULAR_EXPRESSION( ● , pattern)
```

`NOT_MATCH_REGULAR_EXPRESSION` is part of the Neon Custom Library.

MATCH_EXACT_REGULAR_EXPRESSION

Validating Function MATCH_EXACT_REGULAR_EXPRESSION passes through its first implicit parameter if that first parameter exactly matches the Regular expression pattern given by the second parameter. The function "passes through" the input value to the output result on success. The function FAILs if there is not an exact match. Both parameters are treated as strings.

```
MATCH_EXACT_REGULAR_EXPRESSION( ● , pattern)
```

`MATCH_EXACT_REGULAR_EXPRESSION` is part of the Neon Custom Library.

NOT_MATCH_EXACT_REGULAR_EXPRESSION

Validating Function NOT_MATCH_EXACT_REGULAR_EXPRESSION passes through its first implicit parameter if that first parameter does *not* foundexactly match the Regular expression pattern given by the second parameter. The function "passes through" the input value to the output result on success. The function FAILs if there is there is an exact match. Both parameters are treated as strings.

```
NOT_MATCH_EXACT_REGULAR_EXPRESSION( ● , pattern)
```

`NOT_MATCH_EXACT_REGULAR_EXPRESSION` is part of the Neon Custom Library.

DEFAULT_TO_LAB_LOGGED_IN

DOES_NOT_EXIST

EXISTS

MIGHT_EXIST

UPLOAD_DATE