CRM PARSER MANUAL

Written by Ahmed Elsawalhy (Yagasoft.com)

V1.1

1. CONTENTS

2. INTRODUCTION

This solution tries to solve the issue of the rigidity of text in CRM.

For example, notifications are limited to including only columns at a single level or double level down. This can be limiting in cases where deep references, relationship parsing, or non-related references are required. In addition, there is no option to manipulate text retrieved from the fields.

For advanced scenarios, a custom solution is required; hence, creating the CRM Parser.

3. TERMS AND STRUCTURES

The following terms are used throughout the manual.

Term or form	Description
<>	Anything between `<` and `>` should be manually replaced with what it describes when used. E.g. `Retrieve("<entity-name>", "<id>")`, when used could be `Retrieve("account", "000-0001")`.
Row/record	A CRM record or row in a table or entity.
Context	The list of records the parser is currently using to execute constructs – a reference point.
Global state	An object that includes the context, CRM service, cache, and local memory for the parser
Processor	A preprocessor, construct, or post-processor.
Construct	The main structure or placeholder that is placed in the to-be-parsed text. Takes the form `{_`<description>`_<key>(<parameters>)|%<preprocessors>%<block>@<post-processors>@|<key>}`.
Block	What is going to be used to run the construct logic. E.g., the field name to retrieve, or the name of the action to execute.
Preprocessor	Logic applied before the main construct logic is executed using the block. Wrapped in `%`.
Post-processor	Logic applied after the main construct logic is executed using the block. Wrapped in `@`.
Parameters	A list of inputs to a construct or processor. Parameters listed are all required, in order, unless otherwise mentioned.
Replacement map	A simplified JSON of a capture group name and a replacement text. The JSON should only contain a key and value; nested objects are not supported.
Regex groups	A feature where if the first parameter is defined as below, the regex will be executed and the result processed. If the regex contains captures, the processor will be applied to each and the result replacing the capture. Form: `$`<regex>`` E.g., given the text 'this is a test', if the upper post-processor is applied using the regex group `$`\b([a-z])``, the processor will be applied to the first letter of each word only. Output: `This Is A Test`.

4. INSTALLATION

The YS Common solution is required for the configuration entities. It can be skipped if the `dictionary` or `configuration` constructs are not needed.

Install either `Yagasoft.Libraries.Common` (DLL installed) or `Yagasoft.Libraries.Common.File` (the parser class itself is embedded in the project itself) NuGet package, and then reference the `CrmParser` class.

https://github.com/yagasoft/Dynamics365-YsCommonSolution

https://www.nuget.org/packages/Yagasoft.Libraries.Common/

https://www.nuget.org/packages/Yagasoft.Libraries.Common.File/

https://github.com/yagasoft/Dynamics365-CrmTextParser

5. QUICK SAMPLE

```
{_`Retrieve related accounts budget, sum them, and format the result as currency.`_.(this!ys_accounts_project_accountid)|%distinct(accountid)%{_`Retrieve the budget column value.`_c|ys_budget|c}@sum@format(`$#.#`)@|.}
```

6. ALGORITHM

1. Tokenisation
    a. The first step is for the parser to try to extract all constructs from the inside out (nested first).
    b. Constructs are replaced with a token.
    c. This is done to simplify the Regex and to allow for defining a context that is applied to all nested constructs if necessary.
2. Parser loop
    a. If no constructs are found in the text
        i. Tries to detokenize.
    b. Else
        i. Matches the construct key with its logic (decoration).
        ii. Passes the parameters, preprocessors, block, and postprocessors.
        iii. The global state is passed as well.
3. Construct execution
    a. Depends mainly on the construct.
    b. In general, the basic logic is as follows:
        i. The context is retrieved from the global state.
        ii. Preprocessors are executed, in order, taking into account the block content.
        iii. The result is a single text to replace the block in the construct.
        iv. Construct logic is executed using the block as input.
        v. The result is a single text per record in the context.
        vi. Post-processors are executed, in order, on each entry resulting from the construct logic.
        vii. By default, all entries are merged into a single output string, unless an ‘aggregate’ post-processor is used.

7. RESERVED

The following patterns are reserved:

Patterns	Description
`	A tick. Used to escape text that contains reserved characters.
{<key>|<block>|<key>}	This is the basic form of a construct. The keys on the sides are a single-character marker, used for defining which construct to use. This form is very unique and shouldn't occur naturally in any text.
(<parameters>)	Parenthesis. Should be escaped. Used for parameters inside a construct.
%	Should be escaped. Used for defining preprocessors.
@	Should be escaped. Used for defining post-processors.
|	A pipe. Should be escaped. Causes issues if not escaped.

8. RELATIONSHIP TRAVERSAL

Some constructs support traversing relationships. A relation could be a lookup or grid.

A lookup can be referenced by using the dot operator; e.g., `this.ownerid.managerid`.

A grid can be referenced by using the bang operator; e.g. `this.ownerid!ys_servicerequests_OwnerId`. Use the schema name of the relation to traverse it. You can still traverse a lookup after the relation.

9. EXPRESSIONS

The parser supports evaluating expressions. Below are the rules in order of precedence. The higher entries in the table are evaluated first unless wrapped in parenthesis.

Expressions can be used in a block anywhere, except between ticks (`).

<table>
<tr><th>Pattern</th><th>Description</th></tr>
<tr><td>*,/</td><td>Multiple and divide.</td></tr>
<tr><td>+,-</td><td>Add and subtract.
Can be a number or date.
If date, it must be in the following form: <code><date>+<value><unit></code>
The value is a number.
The unit is one of the following:
<table>
<tr><th>Unit</th><th>Description</th></tr>
<tr><td>m</td><td>Minute</td></tr>
<tr><td>h</td><td>Hour</td></tr>
<tr><td>d</td><td>Day</td></tr>
<tr><td>M</td><td>Month</td></tr>
<tr><td>y</td><td>Year</td></tr>
</table>
</td></tr>
<tr><td><,>,<=,>=</td><td>Greater and less than.</td></tr>
<tr><td>==,!=</td><td>Equality.</td></tr>
<tr><td>&&</td><td>And.</td></tr>
<tr><td>||</td><td>Or.</td></tr>
<tr><td>??</td><td>If the left side is empty, take the right side.</td></tr>
<tr><td>?:</td><td>Ternary conditional: <code><predicate>?<true-clause>:<false-clause></code>.
E.g., <code>true?1:2</code>, outputs <code>1</code>.</td></tr>
</table>

10. CACHING

By default, all calls to CRM's web service are cached, except in the following cases:

-

11. CONSTRUCTS

This section lists all the constructs that can be used in the text.

11.1. COMPLETE FORM

The complete form of a construct with all its optional parts:

```
{_`<description>`_<key>(<parameters>)|%<preprocessor>(<parameters>)%<preproc2>%<preproc3…>%<block>@<postprocessor>(<parameters>)@<postproc2>@<postproc3…>@|<key>}
```

11.2. EXPRESSION

Does nothing except place its block in its place in the text. It is just like any other construct, except that it does nothing with the block.

Useful when you only want the effect of pre and post processors, or just evaluate an expression (check respective section).

If the block is not intended to be treated as an expression, escape it by wrapping in ticks `.

DEFINITION

Key: `e`

EXAMPLE

```
{e|`This is a repeatable text.`@store(repeatableText)@|e}
```

OUTPUT

The block after expression evaluation and applying the pre and post processors.

11.3. COLUMN

The most important of all constructs.

Returns a column value. Supports traversing using dot and bang operators.

DEFINITION

Key: `c`

EXAMPLE

```
{c(name)|ownerid|c}
```

In this example, we are getting the record of the owner and then his name.

PARAMETERS

Inputs:

Parameter	Optional	Description

<table>
<tr><td>Transform</td><td>√</td><td>Transform the result using one of the following switches:<table><tr><th>Switch</th><th>Description</th></tr><tr><td>raw</td><td>Output the raw value based on its type as follows:
Date: sortable ISO
Option-set: numeric value
Lookup: <code><logical-name>:<ID></code></td></tr><tr><td>name</td><td>Output the default formatted representation provided by CRM's web service. Falls back to <code>raw</code> if missing.</td></tr><tr><td>log</td><td>Output the logical name of the lookup. Defaults to empty if wrong type.</td></tr><tr><td>id</td><td>Output the ID of the lookup. Defaults to empty if wrong type.</td></tr><tr><td>url</td><td>Output a hyperlink of the lookup. Defaults to empty if wrong type.</td></tr></table>The default behaviour is to try to represent result in as user-friendly pattern as possible.</td></tr>
</table>

LOGIC

1. Traverse the block.
2. If one of the nodes is empty
    a. Output nothing.
3. Apply the parameter that is given in the table above.

OUTPUT

The column value after the traversal.

11.4. PRELOAD

Ensures that the specified list of columns is loaded from CRM in one go for the current record in the context. It improves performance when multiple columns are accessed in the text.

DEFINITION

Key: `<`

EXAMPLE

```
{<|ownerid,createdon|<}
```

LOGIC

1. Cache the column values in the current record in the context by retrieving the list of columns given from CRM.

OUTPUT

Nothing.

11.5. REFERENCE

The most interesting of all constructs.

Either reference a query, action, or a relation. The reference is then set as the context for all nested constructs. The block is executed for each record in the context.

DEFINITION

Key: `.`

EXAMPLE

```
{.(this.owner)|Owner's name: {c|fullname|c}|.}
```

In this example, we are setting the context of execution to the owner record set in the current record. Effectively, we traversed into the `owner` lookup. The full name will be retrieved from the owner record. The output could be `Owner's name: Test User`.

PARAMETERS

Inputs:

Parameter	Optional	Description
Scope		Can be a query, action, or relation. The first two are actual constructs defined elsewhere in the parsed text before the reference. The latter will be defined below. Use `this` to reference the current context instead of a query or action. A relation could be a lookup or grid. A lookup can be referenced by using the dot operator. A grid can be referenced by using the bang operator.
Context name	√	Stores the resulting context in memory for later reference.
Local variable name	√	Store each record in the context in this variable in memory to be referenced in the nested constructs. Similar to a `foreach` in C#.
Global	√	The text `global` could be added as a parameter to indicate that the context should not be reset after execution the reference.

LOGIC

1. Parse the parameters.
2. Extract the scope traversal from the parameters.
3. If not 'this'
    a. Try to find the stored context by name defined in the scope parameter
    b. Set as initial context
4. Traverse the context (`this` or otherwise) using the remaining nodes in the scope.
5. Set the resulting records as the new context.

6. If column name is given
    a. Store the whole context by that name.
7. If local variable is given
    a. Store the current record by that name.
8. Execute the block for each record in the new context.
9. If 'global'
    a. Reset the context to how it was before this construct.

OUTPUT

The result of executing the block for each record in the context.

11.6. FETCH

Stores the FetchXML given in memory for later execution when referenced (reference-construct).

DEFINITION

Key: `f`

EXAMPLE

```
{f(anyAccount)|`<fetch no-lock='true' top='1'>
  <entity name='account' >
    <attribute name='name' />
  </entity>
</fetch>`|f}
```

In this example, we are defining a query that will retrieve any account from CRM upon execution.

PARAMETERS

Inputs:

Parameter	Optional	Description
Store name		The name under which the query will be stored.

LOGIC

1. Store the query in an executable wrapper in the state's memory under the given name.

OUTPUT

Nothing.

11.7. ACTION

Stores a reference to the given action in memory for later execution when referenced (reference-construct).

DEFINITION

Key: `a`

EXAMPLE

```
{a(getLiveExamResult,`{examId:'{c(id)|ys_examid|c}'}`)|ys_getexamresultfromwebservice|a}
```

In this example, we are defining an action reference that retrieves an exam result from an external web service upon execution, given the retrieved exam ID in the input parameters defined.

PARAMETERS

Inputs:

Parameter	Optional	Description
Store name		The name under which the query will be stored.
Input params	√	A simplified JSON of input parameters.
Global	√	Accepts: either specify the value `global` or omit the parameter. Default: false. Defines the action as a `global` action, which does not pass the context as a `Target`.

LOGIC

1. Store the action in an executable wrapper in the state's memory under the given name.
2. Include in the definition
    a. The input params.
    b. The `global` flag.

OUTPUT

Nothing.

11.8. INFO

Returns some info about the current record in the context.

DEFINITION

Key: `i`

EXAMPLE

```
{i|name|i}
```

In this example, we are getting the record of the owner and then his name.

LOGIC

1. Treat the current record in the context as a lookup.
2. Apply the same logic as in the Column construct.

OUTPUT

The info value of the current record.

11.9. TEMPLATE

Define any text, including constructs, given a name that can be referenced later. Think of it like a 'constant' defined in code.

DEFINITION

Key: `t`

EXAMPLE

```
{t(salutation-template)|[salutation] {c|ys_customername|c}|t}
```

In this example, we marked the 'salutation' to be replaced later. This marking is nothing to do with the parser logic. It's a user-defined way to mark text for replacement; do your thing here. The 'customer name' is the name of the field to retrieve from CRM; it's a construct defined below.

The template is stored under the name: 'salutation-template'.

PARAMETERS

Inputs:

Parameter	Optional	Description
Name		The name to store the template. Can be used later to reference the template.

LOGIC

1. Detokenise the block completely.
2. Store the result in the memory under the name given.

OUTPUT

Nothing.

11.10. PLACEHOLDER

Reference a template by name to insert into this position, optionally, replace certain text in the template.

DEFINITION

Key: `p`

EXAMPLE

```
{p(`[salutation]`,`Dear`)|salutation-template|p}
```

In this example, we are retrieving the template called ‘salutation-template’. The text ‘[salutation]’ in the template will be replace with the word ‘Dear’.

PARAMETERS

Inputs, defined in pairs (e.g., `` `text1`,`replacement1`,`text2`,`replacement2` ``):

Parameter	Optional	Description
Text	√	Text to find in the template.
Replacement	√	Text to replace it with.

LOGIC

1. Retrieve the template by the name defined in the block.
2. Parse the parameters.
3. If replacement pairs are defined
    a. Take each pair.
    b. Replace first item in the pair with the second item.
4. Place the result in place of the construct.

OUTPUT

Template block, with replaced text.

11.11.DISCARD

Executes the block, but outputs nothing in the end. Can be used to do some operations in memory in preparation for later actions.

DEFINITION

Key: `_`

EXAMPLE

```
{_|{c|ys_name@store(accountName)@|c}|_}
```

In this example, retrieve the name column, store it as `accountName` in memory, and then output nothing.

LOGIC

1. Parse the block and execute.
2. Discard the result.

OUTPUT

Nothing.

11.12.REPLACE

Returns the block with the matching pattern replaced. It supports capture groups (explained in the logic).

DEFINITION

Key: `r`

EXAMPLE

```
{r($`\s\d\s`,`_`)|Test 2 programs|r}
```

In this example, we are replacing all spaces followed by a digit and a space by an underscore.
Output: `Test_programs`.

PARAMETERS

Inputs:

Parameter	Optional	Description
Regex group		The pattern to use to match the text to replace.
Replacement		Either a text or a replacement map. If a replacement map (check terms table) is given, replace the capture group by name with the value given in the map; otherwise, the text will be inserted in place of the match.

LOGIC

1. If replacement map given
    a. Search for capture groups.
    b. Match the capture group name or index with one in the map.
    c. If found
        i. Replace with corresponding value
2. Else
    a. Replace all the matches (regardless of captures) with the given text.

OUTPUT

The text with the matches replaced.

11.13.DICTIONARY

Returns the text value stored in the `ys_keyvalue` table in CRM.

DEFINITION

Key: `v`

EXAMPLE

```
{v|informationurl|v}
```

In this example, we are retrieving a value from the dictionary table with key: `informationurl`.

LOGIC

1. Retrieve from the table `ys_keyvalue` a record where key is the block content.

OUTPUT

The value retrieved.

11.14.COMMON CONFIG

Returns the value stored in the Common Configuration table in CRM.

DEFINITION

Key: `g`

EXAMPLE

```
{g|ys_informationurl|g}
```

In this example, we are retrieving the value of the column `ys_informationurl` from the Common Config table.

LOGIC

1. Retrieve from the table `ldv_genericconfiguration` the value of the column specified in the block content.
2. Output the value in the most user-friendly format possible.

OUTPUT

The value retrieved.

12. PREPROCESSORS

This section lists all the preprocessors that can be applied on blocks.

12.1. STORE

Stores the value so far in memory. If preprocessors are chained, the block value processed by preceding processors up to this point is stored.

DEFINITION

Key: `store`

EXAMPLE

```
{c|%store(x)%ys_name|c}
```

In this example, we are storing the value `ys_name` in memory under the name `x`.

PARAMETERS

Inputs:

Parameter	Optional	Description
Store name		The name under which the value is stored in memory.

LOGIC

1. Stores the value.

12.2. READ

Read the value stored from memory. Overwrites the value of preceding processors.

DEFINITION

Key: `read`

EXAMPLE

```
{c|%read(y)%ys_name|c}
```

In this example, we are reading the value stored in `y`. If `y` had value `ownerid` then the column-construct will retrieve the owner of the record instead of the name.

PARAMETERS

Inputs:

Parameter	Optional	Description
Store name		The name under which the value is stored in memory.

LOGIC

1. Read the value from memory.
2. If the read value is a collection (from a post-processor store for example)
    a. Concatenate the elements of the collection into a single value.
3. Overwrite preceding processors value.

12.3. CACHE

Sets the cache mode of the construct. Not inherited by nested constructs.

DEFINITION

Key: `cache`

EXAMPLE

```
{.(this!departments)|%cache(true,global)%{c|ys_name|c}|.}
```

In this example, we are retrieving related departments, and then caching the result of this query in CRM's memory – not the parser.

PARAMETERS

Inputs:

Parameter	Optional	Description
Cache		Flag to enable or disable caching. Default: `true`.
Global	√	If the text `global` is passed here, the caching action happens on the executing process memory – not the parser state memory; so, it persists after the parser has finished.

LOGIC

1. Set the 'cache' parameter on the construct object.

12.4. DISTINCT

Used with the reference-construct to retrieve only unique records.

DEFINITION

Key: `distinct`

EXAMPLE

```
{.(this!accounts)|%distinct(accounttype,industry)%{c|ys_taxpercent|c}|.}
```

In this example, we are retrieving related accounts and then making them distinct over the account type and industry (combined), then selecting the tax percentage value.

PARAMETERS

Inputs:

Parameter	Optional	Description
Columns		The list of columns to use, comma-separated.

LOGIC

1. Set the `distinct` parameter on the construct object to be used after running the query.

12.5. ORDER

Used with the reference-construct to sort records.

DEFINITION

Key: `order`

EXAMPLE

```
{.(this!accounts)|%order(accounttype,!industry)%{c|ys_taxpercent|c}|.}
```

In this example, we are retrieving related accounts, ordering them by account type first, and then by industry in descending order (notice the bang sign `!` before `industry`), then selecting the tax percentage value.

PARAMETERS

Inputs:

Parameter	Optional	Description
Columns		The list of columns to use, comma-separated. Optionally, prefix a bang `!` for descending order.

LOGIC

1. Set the `order` parameter on the construct object to be used after running the query.

13. POST-PROCESSORS

This section lists all the post-processors that can be applied on the construct result.

13.1. MEMORY

STORE

Stores the value so far in memory. If post-processors are chained, the result processed by preceding processors up to this point is stored.

DEFINITION

Key: `store`

EXAMPLE

```
{c|ys_name@store(x)@|c}
```

In this example, we are storing the value stored in `ys_name` in CRM in memory under the name `x`.

PARAMETERS

Inputs:

Parameter	Optional	Description
Store name		The name under which the value is stored in memory.

LOGIC

1. Stores the value.

READ

Read the value stored from memory. Overwrites the value of preceding processors.

Useful in combination with the store post-processor. It can be used to store a construct result, do some post-processing, store again under a different name, read the previously stored result, do more processing, store again, and so on. Effectively, using the same value to transform it into different stored values that can be referenced later.

DEFINITION

Key: `read`

EXAMPLE

```
{c|ys_name@store(y)@trim( )@store(trimmed)@read(y)@|c} cleaned = {e|%read(trimmed)%|e}
```

In this example, …

1. Store the value retrieved by the column construct in `y`
    a. E.g., `John Doe` .
2. Remove spaces around the name.
3. Store the trimmed text in `trimmed`.

4. Restore the original text in the pipeline from memory (`y`).
5. Output the read value.
6. Append `cleaned =` to the output.
7. Parse the expression-construct.
8. Read `trimmed` from memory.
9. Append the read value to the output.

PARAMETERS

Inputs:

Parameter	Optional	Description
Store name		The name under which the value is stored in memory.

LOGIC

1. Read the value from memory.
2. Overwrite preceding processors value.

13.2. DISCARD

Ignore the result of the construct pipeline so far.

DEFINITION

Key: `discard`

EXAMPLE

```
{c|ys_name@discard@|c}
```

In this example, we output nothing.

LOGIC

1. Ignore the value returned by the construct after this point.

13.3. STRING

LENGTH

Outputs the length of the text. Supports regex groups (check terms section), but only 1 capture is recommended.

DEFINITION

Key: `length`

EXAMPLE

```
{c|ys_count@length@|c}
```

In this example, the output is the length of the name.

PARAMETERS

Inputs:

Parameter	Optional	Description
Regex groups	√	Refer to terms section.

INDEX

Returns the indices of the matches in the output. Supports regex groups (check terms section).

If a 'capture group' is detected in the pattern, all capture group values indexes are returned as comma-separated values (example below).

DEFINITION

Key: `index`

EXAMPLE

```
{c|ys_description@index($`\d`,last)@join(`|`)@|c}
```

In this example, given `description` as `test 1 and 234, test 5 and 678`, the output will be `11,12,13|27,28,29`.

PARAMETERS

Inputs:

Parameter	Optional	Description
Regex group		The pattern to find the text whose index is required.
Last	√	If `last` is given as a parameter, retrieve the index of the last match only, and last capture as well if multiple.
Single	√	If `single` is given as a parameter, only output one capture per match in the text (discards 2, 3, 6, and 7 indices in the example above).

SUBSTRING

Takes part of the string specified by the index and length. Supports regex groups (check terms section).

DEFINITION

Key: `sub`

EXAMPLE

```
{c|ys_name@sub(0,3)@|c}...
```

In this example, output the first 3 characters of the name, followed by 3 dots.

PARAMETERS

Inputs:

Parameter	Optional	Description
Regex groups	√	Refer to terms section.
Start		The starting index. Zero-based.
Length	√	The length of the string starting at the start index.

TRIM

Removes the specified characters from the edges of the output. Supports regex groups (check terms section).

DEFINITION

Key: `trim`

EXAMPLE

```
{c|ys_name@trim( )@|c}
```

In this example, spaces are removed from around the name.

PARAMETERS

Inputs:

Parameter	Optional	Description
Regex groups	√	Refer to terms section.
Characters		Characters to remove.
Start	√	If `start` is given as a parameter, only trim the output start. Default: `start` and `end`, both enabled
End	√	If `end` is given as a parameter, only trim the output end.

PAD

Fills the text with the given character up to the specified length. Supports regex groups (check terms section).

DEFINITION

Key: `pad`

EXAMPLE

```
{c|ys_count@pad(0,4)@|c}
```

In this example, the output is filled with zeros on the left until it fits the length specified. If it exceeds the length, nothing happens.

PARAMETERS

Inputs:

Parameter	Optional	Description

Regex groups	√	Refer to terms section.
Character		Character to fill with.
Length		Length of the output.
Right	√	If `right` is given as a parameter, pad the right side of the output.

TRUNCATE

Shortens the output to the specified length, and then append the given replacement. Supports regex groups (check terms section).

DEFINITION

Key: `truncate`

EXAMPLE

```
{c|ys_description@truncate(10,`...`)@|c}
```

PARAMETERS

Inputs:

Parameter	Optional	Description
Regex groups	√	Refer to terms section.
Length		Length of the new output.
Replacement	√	Text to append to the new output (e.g., '…').

UPPER

Converts the text to upper case. Supports regex groups (check terms section).

DEFINITION

Key: `upper`

EXAMPLE

```
{c|ys_description@upper@|c}
```

PARAMETERS

Inputs:

Parameter	Optional	Description
Regex groups	√	Refer to terms section.

LOWER

Converts the text to lower case. Supports regex groups (check terms section).

DEFINITION

Key: `lower`

EXAMPLE

```
{c|ys_description@lower@|c}
```

PARAMETERS

Inputs:

Parameter	Optional	Description
Regex groups	√	Refer to terms section.

SENTENCE

Converts the text to sentence case. Supports regex groups (check terms section).

DEFINITION

Key: `sentence`

EXAMPLE

```
{c|ys_description@sentence@|c}
```

PARAMETERS

Inputs:

Parameter	Optional	Description
Regex groups	√	Refer to terms section.

TITLE

Converts the text to title case. Supports regex groups (check terms section).

DEFINITION

Key: `title`

EXAMPLE

```
{c|ys_description@title@|c}
```

PARAMETERS

Inputs:

Parameter	Optional	Description
Regex groups	√	Refer to terms section.

EXTRACT

Returns the matches in the output. Supports regex groups (check terms section).

If a 'capture group' is detected in the pattern, all capture group values are returned as comma-separated values (example below).

DEFINITION

Key: `extract`

EXAMPLE

```
{c|ys_description@extract($`\d`,last)@join(`|`)@|c}
```

In this example, given `description` as `test 1 and 234, test 5 and 678`, the output will be `2,3,4|6,7,8`.

PARAMETERS

Inputs:

Parameter	Optional	Description
Regex group		The pattern to find the text whose match value is required.
Last	√	If `last` is given as a parameter, retrieve the last match only.
Single	√	If `single` is given as a parameter, only output one capture per match in the text (discards 2, 3, 6, and 7 in the example above).

REPLACE

Replaces text matching the pattern given. Supports regex groups (check terms section).

DEFINITION

Key: `replace`

EXAMPLE

```
{c|ys_description@replace($`\d`,_)@|c}
```

In this example, given `description` as `test 1 and 234, test 5 and 678`, the output will be `test _ and ___, test _ and ___`.

PARAMETERS

Inputs:

Parameter	Optional	Description
Regex group		The pattern to use to match the text to replace.
Replacement		Either a text or a replacement map. If a replacement map (check terms table) is given, replace the capture group by name with the value given in the map; otherwise, the text will be inserted in place of the match.

SPLIT

Finds the matches in the text, and then splits them by the given text. Supports regex groups (check terms section).

All splits are blown and returned as separate items. E.g., if the output from the previous processor returned 2 items and the split caused each item to expand by 3, then the output of this processor is 6 items.

DEFINITION

Key: `split`

EXAMPLE

```
{c|ys_description@split($`(?:(.*?)(?: and )?)*`,`,`)@join(`|`)@|c}
```

In this example, given `description` as `test,1 and test,2`, the output will be `test|1|test|2`.

PARAMETERS

Inputs:

Parameter	Optional	Description
Regex group		The pattern to find the text whose match value is required.
Last	√	If `last` is given as a parameter, retrieve the last match only, and last capture as well if multiple.
Single	√	If `single` is given as a parameter, only output one capture per match in the text (discards 2, 3, 6, and 7 in the example above).

13.4. FORMAT

DATE

Formats output as a date. Supports regex groups (check terms section).

If regex groups are used, the captures are the ones to be formatted, with the rest of the text intact.

DEFINITION

Key: `date`

EXAMPLE

```
{c|ys_description@date(`yyyy`)@|c}
```

In this example, given `description` as `2021-01-11`, output: `2021`.

PARAMETERS

Inputs:

Parameter	Optional	Description
Regex groups	√	Refer to terms section.
Output format		The date format to output the date.
Kind	√	Either `utc` or `local` to define the kind of input date.
Input format	√	The date format to process the input date. Useful in assisting the processor in processing the input text incorrectly.

NUMBER

Formats output as a number. Supports regex groups (check terms section).

If regex groups are used, the captures are the ones to be formatted, with the rest of the text intact.

DEFINITION

Key: `number`

EXAMPLE

```
{c|ys_description@number(`Cost: (\d*)`,`$#.000`)@|c}
```

In this example, given `description` as `Cost: 123.1`, output: `Cost: $123.100`.

PARAMETERS

Inputs:

Parameter	Optional	Description
Regex groups	√	Refer to terms section.
Output format		The date format to output the date.

13.5. CLEAR

Removes all empty text from the output.

DEFINITION

Key: `clear`

EXAMPLE

```
{.(this!accounts)|{c|ys_taxpercent|c}@clear@|.}
```

In this example, if the system has 3 accounts stored with the tax percentages: 15, empty tax, and 20; the output will be `15,20`.

13.6. COLLECTION

COUNT

Returns the count of items in the output.

DEFINITION

Key: `count`

EXAMPLE

```
{.(this!accounts)|{c|ys_taxpercent|c}@count@|.}
```

In this example, if the system has 3 accounts, the output will be `3`.

FIRST

Returns only the first item in the output.

DEFINITION

Key: `first`

EXAMPLE

```
{.(this!accounts)|{c|ys_taxpercent|c}@first@|.}
```

In this example, if the system has 3 accounts stored with the tax percentages: 15, 10, and 20; the output will be `15`.

NTH

Returns only the last item in the output.

DEFINITION

Key: `nth`

EXAMPLE

```
{.(this!accounts)|{c|ys_taxpercent|c}@nth(2)@|.}
```

In this example, if the system has 3 accounts stored with the tax percentages: 15, 10, and 20; the output will be `10`.

PARAMETERS

Inputs:

Parameter	Optional	Description
Index		The index of the item to return. Not zero-based.

LAST

Returns only the last item in the output.

DEFINITION

Key: `last`

EXAMPLE

```
{.(this!accounts)|{c|ys_taxpercent|c}@last@|.}
```

In this example, if the system has 3 accounts stored with the tax percentages: 15, 10, and 20; the output will be `20`.

TOP

Returns the first x items in the output.

DEFINITION

Key: `top`

EXAMPLE

```
{.(this!accounts)|{c|ys_taxpercent|c}@top(2)@|.}
```

In this example, if the system has 3 accounts stored with the tax percentages: 15, 10, and 20; the output will be `15,10`.

PARAMETERS

Inputs:

Parameter	Optional	Description
Count		The number of items to return.

DISTINCT

Return unique values from the output. Supports regex groups (check terms section).

DEFINITION

Key: `distinct`

EXAMPLE

```
{.(this!accounts)|{c|ys_name|c}@distinct($`(?:.*?)(\d*)`)@|.}
```

In this example, if the system has 3 accounts stored with the categories: `Company1-Site1`, `Company2-Site1`, and `Company1-Site2`; the output will be `Company1-Site1,Company2-Site1`.

PARAMETERS

Inputs:

Parameter	Optional	Description
Regex groups	√	Refer to terms section.

ORDER

Return unique values from the output. Supports regex groups (check terms section).

DEFINITION

Key: `order`

EXAMPLE

```
{.(this!accounts)|{c|ys_name|c}@order($`(?:.*?)(\d*)`)@|.}
```

In this example, if the system has 3 accounts stored with the categories: `Company1-Site1`, `Company2-Site1`, and `Company1-Site2`; the output will be `Company1-Site1,Company1-Site2,Company2-Site1`.

PARAMETERS

Inputs:

Parameter	Optional	Description

Regex groups	√	Refer to terms section.
Descending	√	If `true` is given, the order will be reversed.

WHERE

Return values that pass the test from the output. Supports regex groups (check terms section).

DEFINITION

Key: `where`

EXAMPLE

```
{.(this!accounts)|{c|ys_name|c}@where($`Site\d+`)@|.}
```

In this example, if the system has 3 accounts stored with the categories: `Company1-Site1`, `Company2-Site1`, and `Company1-Site2`; the output will be `Company1-Site1,Company2-Site1`.

PARAMETERS

Inputs:

Parameter	Optional	Description
Regex groups		Refer to terms section.

FILTER

Remove values that pass the test from the output. Supports regex groups (check terms section).

DEFINITION

Key: `filter`

EXAMPLE

```
{.(this!accounts)|{c|ys_name|c}@filter($`Site\d+`)@|.}
```

In this example, if the system has 3 accounts stored with the categories: `Company1-Site1`, `Company2-Site1`, and `Company1`; the output will be `Company1`.

PARAMETERS

Inputs:

Parameter	Optional	Description
Regex groups		Refer to terms section.

13.7. AGGREGATE

JOIN

Returns only the last item in the output.

DEFINITION

Key: `join`

EXAMPLE

```
{.(this!accounts)|{c|ys_taxpercent|c}@join(`|`)@|.}
```

In this example, if the system has 3 accounts stored with the tax percentages: 15, 10, and 20; the output will be `15|10|20`.

PARAMETERS

Inputs:

Parameter	Optional	Description
Join pattern		The text to insert between the items when joining.

MINIMUM

Returns the minimum value in the output.

DEFINITION

Key: `min`

EXAMPLE

```
{.(this!accounts)|{c|ys_taxpercent|c}@min@|.}
```

In this example, if the system has 3 accounts stored with the tax percentages: 15, 10, and 20; the output will be `10`.

MAXIMUM

Returns the maximum value in the output.

DEFINITION

Key: `max`

EXAMPLE

```
{.(this!accounts)|{c|ys_taxpercent|c}@max@|.}
```

In this example, if the system has 3 accounts stored with the tax percentages: 15, 10, and 20; the output will be `20`.

AVERAGE

Returns the average value of all items in the output.

DEFINITION

Key: `avg`

EXAMPLE

```
{.(this!accounts)|{c|ys_taxpercent|c}@avg@|.}
```

In this example, if the system has 3 accounts stored with the tax percentages: 15, 10, and 20; the output will be `15`.

SUM

Returns the sum of all items in the output.

DEFINITION

Key: `sum`

EXAMPLE

```
{.(this!accounts)|{c|ys_taxpercent|c}@sum@|.}
```

In this example, if the system has 3 accounts stored with the tax percentages: 15, 10, and 20; the output will be `45`.