CRM PARSER MANUAL

Written by Ahmed Elsawalhy (Yagasoft.com)

V2.1

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 CODE AND CONTRIBUTION

Please do not hesitate to share your opinion and ideas on GitHub repo here.

Code can be found here.

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 TERMS

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`.

6 STRUCTURES

6.1 TEXT

The text provided to the `Parse` function. It contains constructs (defined later in this guide) that will be parsed, processed, and replaced with a value based on their function.

6.2 CONSTRUCT

Code wrapped in `{` and `}` in the following format:

`{`<description>`<key>(<parameters>)%<preprocessors>(<parameters>)%|<block>|@<post-processors>(<parameters>)@<key>}`

The closing 'key' is optional.

The following tables explains each item:

Term	Description
description	Optional text that describes what the current construct is being used for. Similar to code comments/documentation.
key	Each construct is represented with a key, defined later in this guide for each construct.
parameters	A list of comma-separated values defined per item. Some items don't accept parameters.
preprocessors	A list of preprocessors (optional). Each one must be wrapped in `%`, individually. E.g. `{c%store(x)%%cache(false)%|name|c}`
block	A block of text, which is the core of most constructs. Might be optional for some constructs.
post-processors	Similar to preprocessors, but wrapped in `@`.

6.3 PREPROCESSOR

Code wrapped in `%`.

Executed, in order, before the construct performs its logic on the block. If a preprocessor is a modifier (modifies the block itself), it will replace the block with its result.

6.4 BLOCK

Code wrapped in `|`.

Similar to the 'text' structure defined previously in this section; however, the text in the block might have a specific function related to the construct – might not appear in the output.

Can contain other constructs to be processed.

To preserve spaces in the output, wrap in ` (tick), which acts as an escape character for what's inside. Constructs wrapped in ticks, are not processed.

E.g. `{e|`The account name is `{c|name|c}.|e}`

6.5 POST-PROCESSOR

Code wrapped in @.

Executed, in order, after the construct performs its logic on the block. If a post-processor is a modifier (modifies the result itself), it will replace the result with its own result and passes it to the next post-processor.

6.6 RESERVED

The following patterns are reserved, and must be escaped:

Patterns	Description
`	A tick. Escapes text that contains reserved characters.
\	Escapes the character that comes after.
{<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.
{ }	Defines constructs.
(<parameters>)	Wrapped in parenthesis, similar to methods in code.
,	Separates parameters. Only reserved in parameter definitions.
%	Defines preprocessors. Only reserved outside of blocks.
|	Defines blocks.
@	Defines post-processors. Only reserved outside of blocks.
! + - * / ? : < > ~ (same as -) || && =	Operators.

Outside a construct, only { is reserved.

7 OPERATORS

7.1 PRECEDENCE

Use `(` and `)` to control/change precedence, just like a calculator.

From highest to lowest, with operators in the same row having the same precedence:

Patterns	Description
! **~ (same as -)**	Not Negative
***** **/**	Multiply Divide
+ **-**	Add Subtract
> **>=** **<** **<=**	Greater than Greater than or equal Less than Less than or equal
!= **==**	Not equal Equals
&&	Logical and
||	Logical or
??	Null-coalescing: if left side is null, take right side `<first-clause>??<second-clause>` E.g., `null??xyz`, outputs 'xyz'.
? :	Conditional: if true, take first, else, take second `<predicate>?<true-clause>:<false-clause>` E.g., `false?1:2`, outputs '2'.

7.2 SHORT CIRCUIT

None of the operators short circuit; all operands are fully evaluated before operations take place. It's planned for a future release.

8 ALGORITHM

1. Tokenisation
    a. The first step is for the parser to try to identify all constructs and their contents. Collectively known as ‘tokens’.
    b. Tokens are represented with their respective classes.
    c. This is done to simplify the processing later.
2. Parser loop
    a. Loop over the tokens.
    b. If a text token is found
        i. Output as is.
    c. Else
        i. Matches the construct key with its logic (class decoration).
        ii. Passes the parameters, preprocessors, block, and post-processors.
        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. If the construct modifies the context, a pre-execute step is executed here.
        iii. Preprocessors are executed, in order, taking into account the block content.
        iv. The result is a single text to replace the block in the construct.
        v. Construct logic is executed using the block as input.
        vi. The result is a single text per record in the context. (usually the Reference construct returns more than one record)
        vii. Post-processors are executed, in order, on each entry resulting from the construct logic.
        viii. By default, all entries are merged into a single output string, unless an ‘aggregate’ post-processor is used.

9 HOW TO USE

9.1 TRIGGER

To start parsing, supply a text value to the following method (sample overload):

```
CrmParser.Parse(<text>, <entity-or-reference>, <crm-service>, <any-unique-id>);
```

The following table explains the function of each parameter:

Parameter	Description
input	Any text value that needs to be parsed for constructs.
context	Can be either an entity or entity reference. In case of entity, the parser will not try to retrieve the field value from CRM if missing; so, it's crucial to provide all required values in the entity object attributes. Entity values are treated like a predefined cache, essentially.
service	The CRM org service. It should only be provided when constructs that require CRM access are used. For example, if only Expression constructs are used, you don't need to pass a service to the parser.
orgId	Any unique identifier. This ID is used for separating the parser-related cache of each CRM organisation when the it is used in a plugin.
constructTypes	The type (class) that contains custom constructs in your code. For example, if you create an 'indexing' construct (exists in the Auto-Numbering solution), you can define its class as follows: `public class CustomConstructs { [Construct("j", "index")] public class IndexConstruct : DefaultContextConstruct { <logic> } }` You should pass the `CustomConstructs` class to the parser.
contextObject	An object that might be required for a custom construct to use. If you create a custom construct, you can access this object by calling `State.ContextObject` from within the construct class. All the default constructs do not require any context object.

9.2 QUICK SAMPLE

```
The budget for this project's accounts is {`Retrieve related accounts budget, sum
them, and format the result as currency.` .(this#ys_accounts_ys_project_accountid)
%distinct(accountid)%|{`Retrieve the budget column value.`c|ys_budget|c}|
@sum@@format(`$#.#`)@.}.
```

The code between the outer `{` and `}` will be replaced with the sum of the budget amount in its place in the text.

9.3 TESTING TOOL

All of the functionality of this parser can be tested using an XrmToolBox plugin:
https://www.xrmtoolbox.com/plugins/D365-CrmTextParser-Tester-Plugin

9.4 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 sharp (`#`) 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.

9.5 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 anywhere, except between ticks (`` ` ``).

Addition and subtraction have a special significance when used with a date as below.

9.5.1 TIMESPAN

Form: `<date>+<timespan>`.

Format it as defined here.

9.5.2 UNITS

Form: `<date>+<value1><unit1><value2><unit2>`.

The value and unit can be repeated as necessary to define different time granularities. The value is a number.

The unit can be one of the following:

Unit	Description
f	Millisecond
s	Second
m	Minute
h	Hour
d	Day
M	Month
y	Year

9.6 CACHING

By default, all calls to CRM's web service are cached, except if overridden by the 'cache' preprocessor or the config construct.

If the YS Common solution is installed on CRM, the cache duration defined in the Common Configuration table will be used; otherwise, an infinite fallback is used, unless it is overridden.

10 EXTENDING THE PARSER

If you want to create your own constructs, follow the example below.

For example, if you create an 'indexing' construct (exists in the Auto-Numbering solution), you can define its class as follows:

```
public class CustomConstructs
{
  [Construct("j", "index")]
  public class IndexConstruct : DefaultContextConstruct
  {
    public ActionConstruct(GlobalState state, TokenKeyword keyword,
IReadOnlyList<Preprocessor> preProcessors, IReadOnlyList<PostProcessor>
postProcessors)
      : base(state, keyword, preProcessors, postProcessors)
    { }

    protected override string ExecuteContextLogic(Entity context, string buffer)
    {
      // retrieve an index from CRM

      // increment the index and store back to CRM


      // set the output as the index by returning it
      return index;
    }
}
```

You should pass the `CustomConstructs` class to the parser as defined in the 'trigger' section of this guide.

Refer to the existing code in the repository on GitHub for more advanced scenarios.

11 CONSTRUCTS

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

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>}
```

11.1 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|`Random text.`|@store(someText)@e}
```

OUTPUT

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

11.2 COLUMN

The most important of all constructs.

Returns a column value. Supports traversing using dot and sharp (`#`) operators.

DEFINITION

Key: `c`

EXAMPLE

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

In this example, we are getting the record of the owner and then the text representation of the record (name of the record).

PARAMETERS

<table>
<tr><th>Parameter</th><th>Optional</th><th>Description</th></tr>
<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.
Supports localisation (<code>local</code> post-processor) if <code>OptionSet</code> value.</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.

11.3 PRELOAD

Ensures that the specified list of columns is loaded from CRM in one go for the current record in the context (caching mechanism). 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.

11.4 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.

Preprocessors and post-processors are executed in the context of the parent construct; only the block is executed in the context of this construct.

If you want everything (pre and post processors, and block) to execute in the context of this construct, set this reference to 'this' only (in its parameter), and wrap it in another reference with the required 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 of 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

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 sharp (`#`) 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. Execute the preprocessors.
2. Parse the parameters.
3. Extract the scope traversal from the parameters.
4. If not 'this'
    a. Try to find the stored context by name defined in the scope parameter.

    b. Set as initial context.
5. Traverse the context (`this` or otherwise) using the remaining nodes in the scope.
6. Set the resulting records as the new context.
7. If column name is given
    a. Store the whole context by that name.
8. If local variable is given
    a. Store the current record by that name.
9. Execute the block for each record in the new context.
10. If not ‘global’
    a. Reset the context to how it was before this construct.
11. Execute the post-processors

OUTPUT

The concatenated result (unless an aggregate post-processor was used) of executing the block for each record in the context.

11.5 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

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.

11.6 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, using the retrieved exam ID in the input parameters defined.

PARAMETERS

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.

11.7 ROW INFO

Returns some info about the current record in the context.

Switch	Description
raw	Output the raw value: `<logical-name>:<ID>`
name	Output the value of the `name` column.
log	Output the logical name of the row.
id	Output the ID of the row.
url	Output a hyperlink to the row.

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 parameter logic as in the `column` construct.

OUTPUT

The info value of the current record.

11.8 USER INFO

Returns some info about the user running the service, or the user with the ID given.

Switch	Description
raw	Output the raw value: `systemuser:<ID>`
name	Output the value of the `name` column.
log	Output `systemuser`.
id	Output the ID of the user.
lcid	Output the UI language of the user.
url	Output a hyperlink to the user.

DEFINITION

Key: `u`

EXAMPLE

```
{u|lcid|u}
```

In this example, we are getting the language of the current user.

PARAMETERS

Parameter	Optional	Description
ID	√	The ID of the user to retrieve info for, else, current user.

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 user.

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 has nothing to do with the parser logic; it’s the user’s way of marking text for replacement. The ‘customer name’ is the name of the field to retrieve from CRM.

The template is stored under the name: ‘salutation-template’.

PARAMETERS

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

LOGIC

1. Store the block as is in the memory under the name given.

OUTPUT

Nothing.

11.10 PLACEHOLDER

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

Text replacement will occur first, and then the template content will be parsed for constructs.

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

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. Parse the result for constructs.
5. Place the result in place of this construct.

OUTPUT

Template block, with replaced and parsed 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 below).

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

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.

If localisation is set to anything other than 1033, the value is retrieved from the column in the form: `ys_value_<lcid>`. If the value is empty, fallback to the English one.

If the specific localised field in the language you want does not exist, create it in a separate solution in CRM.

DEFINITION

Key: `v`

EXAMPLE

```
{v%local(1025)%|informationurl|v}
```

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

LOGIC

1. Retrieve from the table `ys_keyvalue` a record where key is the block content.
2. Read the value stored in the field `ys_value_1025`.

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.

11.15 INLINE CONFIG

Defines a few parameters to use when executing constructs.

DEFINITION

Key: `s`

EXAMPLE

```
{s(`{html:true,cache:'enabled:true,global:true,dur:300'}`)|{c|ys_name|c}|s}
```

In this example, we want to treat the text returned by the column construct as HTML; so any '<', for example, will be replaced with '<'. In addition, enable caching, at the global level, and fallback duration to 5 minutes.

PARAMETERS

<table>
<tr><th>Parameter</th><th>Optional</th><th>Description</th></tr>
<tr><td>JSON</td><td></td><td>Option values to use.
Options:
<table>
<tr><th>Switch</th><th>Description</th></tr>
<tr><td>html</td><td>Treat the text within the scope as HTML, which entails 'unescaping' any HTML entities back to their original form.</td></tr>
<tr><td>cache</td><td>Caching options.
<table>
<tr><th>Switch</th><th>Description</th></tr>
<tr><td>enabled</td><td>Enable or disable caching within the scope.</td></tr>
<tr><td>global</td><td>Enable or disable system-level caching.</td></tr>
<tr><td>dur</td><td>Defines a fallback cache duration to use in seconds. Used only when the Common Configuration entity does not exist in CRM.</td></tr>
</table></td></tr>
</table></td></tr>
<tr><td>Global</td><td>√</td><td>Accepts: either specify the value <code>global</code> or omit the parameter.
Default: false.
Expands the options to be applied globally (on all constructs). This is extremely useful when we want to set all those options at the whole text level.
It is best to be defined at the start of the text with an empty body: <code>{s(<json>,global)}</code></td></tr>
</table>

LOGIC

1. Sets the options in the parser 'state'.
2. Process the construct's body.
3. Reset the state again if not 'global'.

OUTPUT

Nothing.

11.16 RANDOM

Returns a randomly generated string. Supports numbers and letters.

DEFINITION

Key: `*`

EXAMPLE

```
{*(5,c,true,40)|a,b,c,1,2,3|*}
```

In this example, we are want to generate a 5-character random string using the custom pool (c): ‘a,b,c,1,2,3’. The string must start with a letter (true), and numbers should be 40% of the total character count.

PARAMETERS

Parameter	Optional	Description
Length		The total length (character count) of the generated string.
Pool		Use any of the following switches without separators: **Switch** — **Description** **u** — Upper case letters. **l** — Lower case letters. **n** — Numbers. **c** — Custom pool. Specified within the block as comma-separated values. (see example above) E.g. uln, will use upper and lower case letters, and numbers to generate the string.
Letter start	√	Either ‘true’ or ‘false’.
Number to letter ratio	√	A percentage in the full form; e.g. 43, 5, 99 … etc.

LOGIC

1. Read the parameters.
2. Set the length.
3. Define the pool based on the flags.
4. Call the Random Generator.

OUTPUT

The generated random string.

11.17 DATE

Returns ‘now’ in UTC.

DEFINITION

Key: `d`

EXAMPLE

```
{d}
```

In this example, we want the current time in UTC.

LOGIC

1. Read the current time.

OUTPUT

The current time.

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

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

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

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

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 sharp sign (`#`) before `industry`), then selecting the tax percentage value.

PARAMETERS

Parameter	Optional	Description
Columns		The list of columns to use, comma-separated. Optionally, prefix a sharp (`#`)for descending order.

LOGIC

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

12.6 LOCALISE

Sets the language of the construct and its content. Used when retrieving certain metadata and configuration values.

DEFINITION

Key: `local`

EXAMPLE

```
{c(name)%local(1025)%|ys_statuscode|c}
```

In this example, we are retrieving the Arabic label of the `Status Reason`.

PARAMETERS

Parameter	Optional	Description
LCID		The language code.
Global	√	If the text `global` is passed here, the language is set for all constructs after this point in the text.

LOGIC

1. Set the `LCID` parameter on the state object.

12.7 REPLACE

Same as the `String Replace` post-processor, but acts before the block is executed.

13 POST-PROCESSORS

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

13.1 MEMORY

13.1.1 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

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

LOGIC

1. Stores the value.

13.1.2 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)%}
```

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

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

LOGIC

1. Read the value from memory, overwriting the latest value in the construct.

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

13.3.1 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

Parameter	Optional	Description
Regex groups	√	Refer to terms section.

13.3.2 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

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).

13.3.3 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

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.

13.3.4 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

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.

13.3.5 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

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.

13.3.6 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

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., '…').

13.3.7 UPPER

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

DEFINITION

Key: `upper`

EXAMPLE

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

PARAMETERS

Parameter	Optional	Description
Regex groups	√	Refer to terms section.

13.3.8 LOWER

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

DEFINITION

Key: `lower`

EXAMPLE

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

PARAMETERS

Parameter	Optional	Description
Regex groups	√	Refer to terms section.

13.3.9 SENTENCE

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

DEFINITION

Key: `sentence`

EXAMPLE

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

PARAMETERS

Parameter	Optional	Description
Regex groups	√	Refer to terms section.

13.3.10 TITLE

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

DEFINITION

Key: `title`

EXAMPLE

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

PARAMETERS

Parameter	Optional	Description
Regex groups	√	Refer to terms section.

13.3.11 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

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).

13.3.12 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

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.

13.3.13 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

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).

13.3.14 HTML

Escapes or 'unescapes' the text as HTML.

Optionally apply only to regex groups (check terms section).

DEFINITION

Key: `html`

EXAMPLE

```
{c|ys_description|@html@c}
```

In this example, given `description` as `Test 1 & 2`, the output will be `Test 1 &amp; 2`.

PARAMETERS

Parameter	Optional	Description
Regex group	√	The pattern to find the text whose match value to escape/unescape.
Encode	√	By default, the post-processor encodes the text into HTML.
Decode	√	If `decode` is given as a parameter, decode the HTML text.

13.4 FORMAT

13.4.1 DATE

Formats output as a date. Refer to this page for supported formats. 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

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 correctly.

13.4.2 NUMBER

Formats output as a number. Refer to this page for supported formats. 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

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

13.6.1 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`.

13.6.2 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`.

13.6.3 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

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

13.6.4 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`.

13.6.5 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

Parameter	Optional	Description
Count		The number of items to return.

13.6.6 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

Parameter	Optional	Description
Regex groups	√	Refer to terms section.

13.6.7 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

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

13.6.8 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

Parameter	Optional	Description
Regex groups		Refer to terms section.

13.6.9 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

Parameter	Optional	Description
Regex groups		Refer to terms section.

13.7 AGGREGATE

13.7.1 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

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

13.7.2 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`.

13.7.3 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`.

13.7.4 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`.

13.7.5 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`.