IMARDA 360 –Recipe

*Maurice Verheijen*

*Document last update:* 2013-09-03

Contents

[Purpose 3](#_Toc366016954)

[Recipe File Format 3](#_Toc366016955)

[Macro Expansion 4](#_Toc366016956)

[Pseudo-macros 4](#_Toc366016957)

[Number generators 5](#_Toc366016958)

[Global macros 5](#_Toc366016959)

[Macro wildcards 5](#_Toc366016960)

[Loading the contents of files and web links 6](#_Toc366016961)

[Predefined macros 6](#_Toc366016962)

[Case sensitivity 6](#_Toc366016963)

[Variable commands 7](#_Toc366016964)

[Output 7](#_Toc366016965)

[Command List 8](#_Toc366016966)

[Syntax 8](#_Toc366016967)

[#require 8](#_Toc366016968)

[#default 9](#_Toc366016969)

[#put (alias #!) 9](#_Toc366016970)

[#putv 9](#_Toc366016971)

[#putx 10](#_Toc366016972)

[#putm 10](#_Toc366016973)

[#cdata (alias #@) 11](#_Toc366016974)

[#load 12](#_Toc366016975)

[#extract 12](#_Toc366016976)

[#exp 13](#_Toc366016977)

[#expn 13](#_Toc366016978)

[#forget 14](#_Toc366016979)

[#point 15](#_Toc366016980)

[#restore 15](#_Toc366016981)

[#seq 15](#_Toc366016982)

[#rand 16](#_Toc366016983)

[#marker 16](#_Toc366016984)

[#do 16](#_Toc366016985)

[#new 17](#_Toc366016986)

[#newer 18](#_Toc366016987)

[#before 18](#_Toc366016988)

[#after 19](#_Toc366016989)

[#subs 19](#_Toc366016990)

[#append 19](#_Toc366016991)

[#sort 19](#_Toc366016992)

[#unique 20](#_Toc366016993)

[#indentxml 21](#_Toc366016994)

[#repeat 21](#_Toc366016995)

[#encrypt 22](#_Toc366016996)

[#decrypt 22](#_Toc366016997)

[#edit 22](#_Toc366016998)

[#with 23](#_Toc366016999)

[#run 23](#_Toc366017000)

[#cmd 23](#_Toc366017001)

[#ps 24](#_Toc366017002)

[#csharp 25](#_Toc366017003)

[#git 26](#_Toc366017004)

[#recipe (alias ##) 26](#_Toc366017005)

[#return 26](#_Toc366017006)

[#stop 27](#_Toc366017007)

[#iter, #iter .. #next 27](#_Toc366017008)

[#for .. #next 29](#_Toc366017009)

[#while/#until .. #loop 29](#_Toc366017010)

[#if/#ifnot .. #else .. #endif 29](#_Toc366017011)

[#info 30](#_Toc366017012)

[#warn 30](#_Toc366017013)

[#email 30](#_Toc366017014)

[#udp 31](#_Toc366017015)

[#write 31](#_Toc366017016)

[#mbox 31](#_Toc366017017)

[#ask 31](#_Toc366017018)

[#del 31](#_Toc366017019)

[#copy 31](#_Toc366017020)

[#dir 32](#_Toc366017021)

[#mkdir 32](#_Toc366017022)

[#rmdir 32](#_Toc366017023)

[#server 32](#_Toc366017024)

[#post 32](#_Toc366017025)

[#watch 32](#_Toc366017026)

[#exit 32](#_Toc366017027)

[#pause 32](#_Toc366017028)

[Server Mode 32](#_Toc366017029)

[File Watcher Mode 33](#_Toc366017030)

[Text Tools (*#with*-scripts) 33](#_Toc366017031)

[Command Line 34](#_Toc366017032)

[GUI 34](#_Toc366017033)

[Appendix 1: #do-block markers 36](#_Toc366017034)

[.cs (C#) 36](#_Toc366017035)

[.xml, .config, .html, .htm 36](#_Toc366017036)

[.bat 36](#_Toc366017037)

[Default 36](#_Toc366017038)

[Appendix 2: The #with Command 37](#_Toc366017039)

[Appendix 3: Command Summary 38](#_Toc366017040)

[Summary Recipe 38](#_Toc366017041)

[Summary Output 38](#_Toc366017042)

[Appendix D: Examples 40](#_Toc366017043)

[Sets 40](#_Toc366017044)

[Appendix E: To be documented 40](#_Toc366017045)

# Purpose

*Recipe* is a software tool to expand and insert template text in text files, specifically C# source files, SQL files, .bat files, .config files, .xml files, and other types of text files. The program takes a recipe as input and makes changes to specified files based on that recipe.

There is a command line version and a GUI version available. The GUI version is intended for use during software development, whereas the command line version can be used to update files e.g. as part of a build script or deployment script.

Both versions can be run in a simple *HTTP* *server* mode, where they listen for requests and execute recipe code to handle the request and produce a response to the caller.

A *File System Watcher* feature allows scripts to be executed based on files being added, modified and deleted.

A recipe can invoke inline or file based Cmd shell and Windows Powershell scripts to retrieve information from its environment.

A large number of common text processing functions have been added that can be used in so called #with-*scripts* to manipulate text.

# Recipe File Format

The *Recipe* program reads a given recipe text file and interpretes instructions in that file. An instruction starts with ‘#’ as the first character on a line. The ‘#’ is followed by a command, then a space and then arguments. Some commands read so called *here*-text[[1]](#footnote-1) in the following lines until the pseudo-command ‘#end’ is found. Other text than commands and here-text is deemed comments and therfore skipped during execution.

Example of a recipe file contents:

#marker MyRecipe

#require Name

#require Email

#default Country=New Zealand

#do c:\source\Contacts.txt

#do c:\source\Log.txt

#put newfile=c:\source\details\`Name`.txt

#new `newfile`

Contact: `Name`

Country: `Country`

Details: bla bla

#end

#run {notepad.exe} `newfile`

# Macro Expansion

Each line starting with ‘#’ is split into a command and arguments. The arguments start after the space after the command. The arguments may contain *macros*, words in backquotes `…` which are replaced by entries in a dictionary that is kept while running the recipe. The dictionary is prefilled with all environment variables and is case-insensitive. More macros can be added to the dictionary during execution by the #require, #put and #default commands, to name a few. The macros in arguments are always expanded regardless of the command, before execution. There are no commands that leave macros uninterpreted. If there is a *here*-text then any macros in that text will also be expanded. Macro expansion is not recursive: if a macro would produce a text with another macro in it, then this inner-macro is not expanded. However any remaining occurrences with double quotes ``...`` will be turned into single quotes `...`. Macros can be glued together like: `a``b`, which will expand a and b without any space between them. Note that doubling the quotes only works if those double quotes occur in pairs. A single quote in a text after template expansion is produced by `bq`.

#put a=``hello``

#write `a`

#put b=``hello'

#write `b`

#put c=`bq`hello'

#write `c`

Output:

`hello`

``hello'

`hello'

### Defaults

The macro can resolve to a default value in case it is not found in the dictionary. The notation is:

`macro|default value`

If the macro is not in the dictionary, then the text between | and the final ` is used. The default value is unescaped for \r, \n, \t and \\. A | and ` cannot be used as part of the default value.

#info `xyz|two\r\nlines`

Output:

@info two

lines

Note that a macro with an empty string will not resolve to the default value but resolve to that empty string.

To return an empty string for a non-existent macro *m*, write: `m|`

### Pseudo-macros

These look like macros but they *are not stored* in the dictionary because their value changes depending on time or context. Their values cannot be changed by using #put or other dictionary commands

* `newguid` which will be expanded as a random GUID in dash-format without braces or parentheses,
* `newseqguid` which returns a sequential GUID for use in databases,
* `recipe` returns the path of the recipe currently being executed and will change when subrecipes are called
* `filename` the name of the file being created or modified, without extension
* `filename.ext` the name of the file being created or modified
* `fullpath` the full path of the file being created or modified
* `ts:*datetime-format*` a timestamp in UTC using a .NET date/time format, e.g.   
  `ts:d-MMM-yy HH:mm`, `ts:t`, `ts:s` - the last one is the same as `timestamp`
* `timestamp` UTC timestamp in ISO sortable format “2010-11-24T22:07:00”
* `ts:sql` and `ts:iso` UTC timestamp for use with T-SQL: "2015-07-31 23:59:08"
* `version` the version of the recipe tool
* `rxTrue` a regex for *true-like* values:(?i)^true$|^y(?:es)?$|^1$
* `rxFalse` a regex for *false-like* values :(?i)^false$|^no?$|^0$

### Number generators

There are two types of number generators:

* sequence number generators
* random number generators

To get the next value from a number generator, a notation is used similar to a macro, but the name must start with an ampersand &, e.g.:

`&smallrandom`

`&evennum`

The generators are set up by a #-command, see below.

### Global macros

Macros starting with a $ sign will be stored in a s special global dictionary. This is only of importance when running in server mode, where each request gets a copy of the normal dictionary for the duration of the request. Global macros can be used in this case to keep state between requests. The $ is part of the macro name, therefore global macro names cannot clash with local macros. There are no predefined global macros, the dictionary is empty on start-up.

`$RequestCounter`

`$Language`

#put $start=`timestamp`

### Macro wildcards

Incomplete macro names in back-quotes that end with an asterisk select all macros that start with the character before the asterisk. E.g. if the dictionary contains:

name, street, suburb, state, country

and a template contains

`st\*`

then a text with two lines is returned:

street

state

Note that you *cannot* make a command execute multiple times once for each macro:

#warn `st\*`

will just once execute #warn (pop up an alert box) with all the matching macro names in it.

❖ Tip: *to get the values of all the macros, surround the macros with `…` and expand:*

#put a=`st\*`

#with a=enclose ``[]``

#exp a

### Loading the contents of files and web links

A file path can be given between back-quotes; it will be expanded as theliteral text of the file. If macro expansion has to happen, then #exp can be used for that:

#put template=`c:\templates\myfile.txt`

#exp template

If part of the file path does not exist, then an empty string will be assigned.

Note that one can use #default like in:

#default template=`c:\templates\myfile.txt`

but since command lines are always expanded before the command is interpreted, the file will be loaded regardless of the key already having a value. Of course the content is not assigned if the key already has a value.

Likewise, an http(s) website can be accessed and its literal text is loaded into the dictionary.

#put pageContents=`http://myintranet/`

### Predefined macros

In addition to the *environment variables* there are some useful extra macros. Unlike pseudo-macros, these are stored in the dictionary and can be changed, but this is not recommended.

|  |  |
| --- | --- |
| empty | Empty string |
| *(1 space)* | Empty string |
| nl | New line (CR LF) |
| tab | Tab stop |
| paste | The content of the clipboard before the recipe is executed. Does not change during execution, even when using #copy. However, it can be changed by using #put. |
| bq | ` the backquote character |
| sp | a single space |

### Case sensitivity

All macros are *case-insensitive*, however the ts: macro will not ignore the case after the colon.

### Variable commands

The command itself can be an expandable macro. This looks like:

#`*macro*`

E.g.:

#require cmd

#`cmd` marker c:\temp\test.txt

text

#end

The user could select before or after or subs as command. See also the examples (appendix D).

### Variable arguments

The arguments line is always expanded before it is passed to the command. This allow constructions as:

#put a=b=hello

#put `a`

#info `b`

Output:

@set a=b=hello

@set b=hello

@info hello

# Output

During execution the progress is logged. If the recipe fails then one can see how much has been done and why it failed. The messages are printed in the log text box of the GUI. If run on command line then the messages are printed to standard output and can be redirected into a file.

The output uses certain tokens starting with @:

@name – recipe name change

@set – set a macro; i.e. using #put, #putv, #default or #require

@do – finished a do-block replacement

@skip – file was skipped because it already existed (#new) or it didn’t exist (#do, #before, #after)

@nomatch before/after – the #before or #afer command could not find a message

@new – created a new file

@run - launched a program asynchronously

@call – ran a program

@info – just a message

@warn – popped up a message dialog

@done – finished processing

@email – email sent

@clip – contents copied to clipboard

@del – file deleted

@recipe – execution of subrecipe

@append – text appended to file or macro

@iter – iterate and invoke recipe

@extract – extract command executed

@numg - number generator defined

(to do)

# Command List

There are several groups of commands:

* Dictionary commands
* Number generator commands
* Text creation/modification commands
* Execution commands
* I/O commands
* File and clipboard
* Server
* Sets

## Syntax

In the following sections the green *italic font* indicates parameters. Where it says *paths* it can mean a single path like:

c:\temp\file.txt

or multiple paths generated by a wildcard:

c:\temp\book\*.txt

or a recursive path like:

-r c:\tree\log\*.txt

where -r indicates recursion through the folder structure.

Purple text is used for examples.

## #require

Syntax: #require *macro* *optional\_comment*

Use #require to indicate a required parameter whose identifier is given immediately after #require. The parameter is either read from command line (command line version of Recipe) or obtained from a dialog box that is filled in by the user (GUI version of Recipe). Failure to provide this parameter will cause an error and abort execution. The parameter and its value are put as a macro/value mapping in the macro dictionary.

The identifier can be followed by an optional comment, which is ignored by the Recipe interpreter.

## #default

Syntax: #default *macro*=*value*

Syntax: #default *macro  
 here-text* #end

This puts the given identifier and its value in the macro dictionary *only if* there is no such identifier already.

There is another form of this command that only works in the GUI:

Syntax: #default *macro*=*option1|option2|option3...*

The options are presented in a dropdown when in input mode (F4). You can then select one of the options and it will get put in the dictionary as value of the key (identifier).

## #put (alias #!)

Syntax: #put *macro*=*value*

Syntax: #put *macro  
 here-text* #end

This puts the given identifier and its value in the macro dictionary. If the identifier already exists in the dictionary, then its value will be replaced by the new value.

## #putv

Syntax: #putv *macro*=*value*

This puts an entry in the dictionary using the first identifier as key and interpreting the second identifier as an existing key, using *its value* to assign to the first key. If the first identifier already exists in the dictionary, then its value will be replaced by the new value.

#put a=hello

#putv b=a

output:

@set a=hello

@set b=hello

This is identical to:

#put a=hello

#put b=`a`

but in practice #putv will be used over two levels: first let the user select one of multiple options

#default ColourName=Red|White|Blue

#put Red=5b912783-7c49-437e-8e56-3301ca25058a

#put White=a0aa36b2-0e3d-4a64-b068-04fbf1bef015

#put Blue=4fb45d6b-e71e-421f-93f0-052ca3ded9fa

#putv ColourCode=`ColourName`

The last line will first expand `ColourName` to one of the strings Red, White or Blue and #putv will then grab the associated value of the selected string and assign it to ColourCode.

## #putx

Syntax: #putx *macro*=*template*-*macro*

This will expand the text in the template and assign it to the given macro. This function works the same as #putv but does an additional expansion of the text assigned by the template.

Example 1

#put templ=Hello ``name``

#put name=Alice  
#putx greeting=templ

#put name=Bob  
#putx greeting2=templ

Output:

@set templ=Hello `name`  
@set name=Alice  
@set greeting=Hello Alice  
@set name=Bob  
@set greeting2=Hello Bob

After execution, greeting=Hello Alice and greeting2=Hello Bob

## #putm

Syntax: #putm *kv\_macro*

This can be used to quickly import a configuration file with key=value lines into the dictionary.

## #cdata (alias #@)

Syntax: #cdata *macro  
 here-text* #end *macro*

Normally all here-texts will be expanded immediately, but there may be cases where this is not wanted. That is where #cdata comes in. It assigns the unparsed here-text in the dictionary, under the given macro name. Unlike a normal here-text, the end marker of this here-text is followed by one space and the macro name. This allows recipes and templates - including occurrences of #end markers - to be stored unchanged in the dictionary. In case of a recipe, it can be executed later by the #recipe command. In case of a template, it can be expanded later by a #exp command or by #putx.

Example 1 (inline recipe)

#cdata **makefile**  
#new c:\temp\`name`.txt  
hello `name`  
#end  
#end **makefile**  
  
#put name=Alice  
#recipe makefile  
#put name=Bob  
#recipe makefile

Example 2 (inline template)

#cdata template1  
INSERT INTO MyTbl VALUES ('`newseqguid`', '`name`')  
GO  
#end template1  
  
#put name=Charlie  
#putx stmt1=template1  
#put name=Jane  
#putx stmt2=template1

After this, stmt1 will have a value for the id and the first name filled in:

INSERT INTO MyTbl VALUES ('cba8a4a4-dc3b-4eaf-805b-08cf2ae94b80', 'Charlie')  
GO

and stmt2 is

INSERT INTO MyTbl VALUES ('d04957da-de4c-4098-aa36-08cf2ae94b80', 'Jane')  
GO

❖ Tip: *if you want to execute the recipe up to a certain point, you can temporarily change it by inserting at that point a* #cdata <name> *line without a corresponding* #end <name>. *The interpreter will not run anything from that* #cdata *line up to the end of the file because it cannot find its end marker.*

## #load

Syntax: #load *macro*=*path*

This will put in the dictionary the content of the file with the given path. Note that #put can do something similar if the file path is a literal:

#put macro=`c:\temp\xyz.txt`

but if the path is not a literal then #put cannot be used this way and #load should be used:

#put file=c:\temp\xyz  
#load macro=`file`

*❖ Note: #load will crash if part of the path is not found, #put …=`…` will not crash but assign empty string.*

#extract

Syntax: #extract *regex* *macro  
 here-text* #end

Syntax: #extract *regex* *path  
here-text*#end

Finds the regular expression in the given macro or files and substitutes them with the replacement text, which is the expanded *here-*text*.*

❖ Tip: *To extract information from a file, first load the contents of the file into the dictionary as a new macro and then run* #extract *on the macro or else it would overwrite the file with the result!*

Example:

The file c:\temp\myfile.log contains

1 error in file “foo.txt”  
2 errors in file “bar.txt”  
Compilation ends

We execute the following script:

#load log=c:\temp\myfile.log  
#extract (?i)errors?\ in\ file\s\*([^\r]+) log  
Error $1#end  
#info `log`

This will change the macro “log” to

Error “foo.txt”  
Error “bar.txt”

The *here*-text of #find can contain placeholders for the groups captured by the regular expression. In the above example $1 represents the text captured by ([^\r]+)

## #exp

Syntax: #exp *macro*

This looks up the *macro* as key in the dictionary. The associated value is then expanded: all `... ` occurrences are replaced by their dictionary values. All remaining double ``...`` will be changed into `...`.

Note that #put a=`a` is not the same: this will simply assign the value of x to itself, a useless operation. However #putx a=a accomplishes the same as #exp a

Example:

#put arg1=blue

#cdata template

the sea is `arg1`

#end template

#exp template

output:

@set arg1=blue

@set template=the sea is `arg1`

@set template=the sea is blue

## #expn

Syntax: #expn *regex* *macro  
 key1=value1a|key2=value2a|key3=value3a  
 key1=value1b|key2=value2b|key3=value3b  
 …* #end

Syntax: #expn *regex* *macro  
 value1a|value2a|value3a  
 value1b|value2b|value3b  
 …* #end

Use this to do multiple expansions of a template. The argument of the #expn command is the macro that contains the template. Ths will usually be defined as a #cdata (#@) because it contains macro references that should be expanded during execution of #expn. The here-text is a list of multiple key=value pairs, separated by

#@ sql

insert int Contact (ID, Name, Phone)

values (`id`, '`name`', '`phone`')

#end sql

#expn sql

id=1498|name=Joe|phone=123456

id=1499|name=Zoe|phone=9494949

id=8811|name=Ann|phone=330033

#end

#info `sql`

Output:

insert int Contact (ID, Name, Phone)\r\nvalues (`id`, '`name`', '`phon...

@info insert int Contact (ID, Name, Phone)

values (1498, 'Joe', '123456')

insert int Contact (ID, Name, Phone)

values (1499, 'Zoe', '9494949')

insert int Contact (ID, Name, Phone)

values (8811, 'Ann', '330033')

Instead of using named keys, the second form of #expn allows values only, separated by |. These values can be referenced in the template as `1`, `2`, etc. respectively. So the output of the example above can be achieved also by:

#@ sql

insert int Contact (ID, Name, Phone)

values (`1`, '`2`', '`3`')

#end sql

#expn sql

1498|Joe|123456

1499|Zoe|9494949

8811|Ann|330033

#end

In the first example, it is also possible to use `1`, `2`, `3` etc. instead of `id`, `name`, `phone`.

There are two special macros that can be used in the template

`0` This expands to the current line in the here-text

`#` Is the line number in the here-text, starting from 1.

## #forget

Syntax: #forget *macro*

Remove a macro from the dictionary. This works for local and global macros.

#put $a=hello

#put a=world

#info `$a` `a`

#forget $a

#info `$a` `a`

#forget a

#info `$a` `a`

output:

@set $a=hello

@set a=world

@info hello world

@info world

@info

## #point

Syntax: #point

Set a restore point. This stores the current state of the dictionary somewhere for later retrieval. See #restore.

## #restore

Syntax: #restore

This restores the dictionary state saved with #point, undoing all changes to the dictionary since #point.

## #seq

Syntax: #seq *name* *fmt min step max*

Define a sequence number generator using the given format string, minimum (default 1), increment (default 1) and maximum (default int.MaxValue). All parameters except the name are optional: they can be omitted from right to left in which case their default values are used. The pseudo-macro of the format `&name` will expand to the next number of the sequence. Each time the max is reached the generator will restart at min.

Examples:

#seq even 0 2 2

#info `&even` `&even` `&even` `&even` `&even`

result:

@info 2 4 6 8 10

#seq sixplusfive 000 6 5

#info `&sixplusfive` `&sixplusfive` `&sixplusfive` `&sixplusfive`

result:

@info 006 011 016 021

#seq cycle 0 1 1 3

#info `&cycle` `&cycle` `&cycle` `&cycle` `&cycle` `&cycle` `&cycle`

result:

@info 1 2 3 1 2 3 1

## #rand

Syntax: #rand *name fmt min max seed*

Define a random number generator using the given format string. Generate pseudo-random numbers between min and max inclusive. The same sequence can be produced by using a fixe seed other than 0. A seed of 0 will use the clock to generate a sequence. All parameters except the *name* are optional: they can be omitted from right to left in which case their default values are used. The defaults are fmt 0, min 0, max 9, seed 0. The pseudo-macro of the format `&name` will expand to the next number of the random sequence.

Examples:

#rand bit 0 0 1

#info `&bit``&bit``&bit``&bit``&bit``&bit``&bit``&bit`

result example:

@info 10001100

#rand r 000 200 400 1

#info `&r` `&r` `&r` `&r` `&r` `&r`

result:

@info 249 222 293 355 332 286

## #marker

Syntax: #marker *identifier*

It sets the name of the markers for the #do command. The marker name can be changed as often as required. The default is “Recipe”, i.e. if no #marker has been executed before a #do, then the #do will use “Recipe” as a marker. The identifier is case-sensitive.

## #do

Syntax: #do *path*

The #do command takes a file path as its argument. It opens the file and searches for a block labeled by the identifier set by the last executed #marker command. The block is marked with special begin and end tokens depending on the file type. The whole block without markers is inserted before the marked block and any macros in the text are expanded.

For example blocks inside a SQL file start with /\*\*\* and end with \*\*\*/, where the start marker is followed by the block identifier:

CREATE TABLE [Customer] ( ID uniqueidentifier NOT NULL

,[Name] varchar(50) NOT NULL

,[City] varchar(50) NOT NULL

/\*\*\* AddField

,[`**FieldName**`] [`**Type**`] `**Constraint**`

\*\*\*/

)

After running this recipe …

#marker AddField

#put FieldName=CountryCode

#put Type=char(2)

#put Constraint=NOT NULL DEFAULT 'XX'

#do file1

... the result is a changed file contents:

CREATE TABLE [Customer] ( ID uniqueidentifier NOT NULL

,[Name] varchar(50) NOT NULL

,[City] varchar(50) NOT NULL

,[CountryCode] [char(2)] NOT NULL DEFAULT 'XX'

/\*\*\* AddField

,[`FieldName`] [`Type`] `Constraint`

\*\*\*/

)

The block has been duplicated and its macros expanded. If the file does not exist then a @skip-message i printed but execution of the recipe continues.

Block markers are defined by the file extension. **Appendix A** lists them all.

❖ Tip: *it is often better to put a do-block in the source code rather than using the #after/#before commands because the do-block is visible to the programmer when changes are required. A template defined outside the source file may be forgotten and become obsolete.*

## #new

Syntax: #new *path*

*here-text*

#end

The given file (path) is created (including any non-existing directories in the path), and the *here-*text is expanded and copied into the new file.

❖Tip: *in the case of versioned source code, follow the #new command by a #cmd, #run or #git command to add the new source to the code repository.*

## #newer

Syntax: #newer *path*

*here-text*

#end

The same as #new, but overwrites the file if it exists and creates a new file if it does not exist.

This is the same as the sequence:

#del *file*

#new *file*

## #before

Syntax: #before *regex paths*

*here-text*

#end

Syntax: #before *regex key*

*here-text*

#end

In the first syntax, the given file is opened and its contents matched against the regular expression. The expanded *here*-text is then inserted **before** each match. This way changes can be made to files without polluting them with templates, e.g. if they have to be interpreted by software that cannot handle the do-blocks.

In the second syntax, a macro in the dictionary is modified. The key is looked up and its value is changed. The program will first try to find the macro in the dictionary, if not found, it tries to interpret the argument as a file path.

The regex follow normal .NET rules. If a space needs to be included in the regex, it should be escaped by a backslash. This is optional in .NET regex, but it’s the only way here to make sure the space belongs to the regex and is not the separator between the regex and the path.

Example:

#before CREATE\ PROCEDURE c:\temp files\my file.txt

-- `username`

#end

The space after the \ belongs to the regex. The space after PROCEDURE is the separator between the regex and the path. All subsequent spaces belong to the path.

❖ Note: *although the #end marker is on a new line, the CR LF just before it will be stripped from the here-text. The remaining text will be inserted directly before the matching text. If the insertion has to occur on a new line then a blank line should be left just before #end.This is a feature of all here-blocks (#new, #before, #after), but most notably with #before and #after.*

❖ Tip: *put the regex in the dictionary as a macro value with #put and then reference it in backquotes like `macroname` to make it easier to read. Especially when the regex is long and complicated or contains spaces.*

❖ Tip: *for ignore-case, single line and multiline regex options prefix the regex by a option group, e.g.* (?is)*regex : ignore case and single line.*

## #after

Syntax: #after *regex paths*

*here-text*

#end

Syntax: #after *regex macro*

*here-text*

#end

Like #before, but inserts the expanded *here*-text **after** each match.

## #subs

Syntax: #subs *regex paths*

*here-text*

#end

Syntax: #subs *regex macro*

*here-text*

#end

Like #before, but substitutes the matching texts with the expanded *here*-text.

## #append

Syntax: #append *paths*

*here-text*

#end

Syntax: #append *macro*

*here-text*

#end

This appends the *here-*text to the specified file or macro. It is similar in functionality to:

#after \z *paths\_or\_macro  
here-text*#end

The regular expression \z finds the end of the file and #after will append from there. However, #append is easier to understand.

## #sort

Syntax: #sort *regex paths*

Syntax: #sort *regex macro*

In both forms, regex can be replaced by

date to sort on sortable date

! to use the whole line for sorting

This will sort the lines in each file addressed by the given path, using a regular expression. The regex should have one group in parentheses which defines the string within the line to sort on. The path can be a single file, or a wildcard. If preceded by –r then the subdirectories are included in the file search.

Example:

#sort (\d+)\ NZD -r c:\lists\\*.csv  
-cmd-|--regex---|----paths--------

Note that the regex must escape any spaces in it. This example will work on all csv files found in c:\lists or any of its subdirectories. It will use the first whole number on the line that is followed by “ NZD” as a sort key. No new files are created; the files are replaced by their sorted versions.

#sort date -r c:\datafiles\log\*.txt

#sort ! linesOfText

## #unique

Syntax: #unique *regex paths*

Syntax: #unique *regex macro*

Remove lines from a text that have a given pattern that occurred on a previous line. This will loop through the lines in the given files or macro and try to match each line with the given regular expression. The first matching group is looked up in a dictionary and if it exists, the line will be skipped, otherwise the line will be in the result.

Example:

#put a

colour=red

shape=circle

size=50

colour=green

size=100

name=F1

#end

#unique ^(\w+)= a

#info `a`

output:

@set a=colour=red\r\nshape=circle\r\nsize=50\r\ncolour=green\r\nsize=100…

@unique a

@info colour=red

shape=circle

size=50

name=F1

The regex finds the first word before the *equals*-sign and checks if it has seen it before, therefore it will skip the line with colour=green and size=100 because colour=red and size=50 were found before.

Like in #sort, we can substitute an exclamation mark ! for the regex to match the whole line:

#unique ! var1

## #indentxml

Syntax: #indentxml *paths*

Will interpret all the files specified by *paths* as xml and reformat them by indenting with tab characters. Also a header <?xml version=”1.0”?> will be inserted as the first line if not already there.

Example:

#indentxml c:\data\\*.xml

This will reformat all the files that match \*.xml in the c:\data directory.

## #repeat

Syntax: #repeat *macro count*

Looks up the macro in the dictionary, takes its contents , repeats it the specified number of times and replaces the macro in the dictionary with the repeated content.

Example

#put a

one

two

#end

#repeat a 3

#info `a`

Result:

one

two

one

two

one

two

❖ Tip: *when working with multi-line text, it is often good to* *put an empty line before #end or else the next repeated item will be concatenated directly to the last item. E.g. above:* twoone

## #encrypt

Syntax: #encrypt *macro password*

Syntax: #encrypt *macro*

Encrypts the text in the macro with the given password. If the password is not given as the 2nd parameter then the value is taken from a macro with the name password. Encryption uses the Rijndael algorithm with a 256 bit key hash based on the given password. To decrypt the cipher text, use the #decrypt command.

Example

#put password=M!88-aUwF

#put text

Hello

world

#end

#encrypt text

#info `text`

Result

@info T8ARcS1nY0iRgh++YFkONQ==

## #decrypt

Syntax: #decrypt *macro password*

Syntax: #decrypt *macro*

Decrypts the cipher text assigned to the given macro using the provided password. If the password is missing as 2nd parameter then the value is taken from a macro with the name password. If the password is wrong, the result will be an empty string.

Example

#put text=T8ARcS1nY0iRgh++YFkONQ==

#decrypt text M!88-aUwF

#info `text`

Result

@info Hello

world

## #edit

Syntax: #edit *macro*

Only works with the GUI version of the Recipe program (RecipeGUI.exe). The #edit command takes the value of the given macro and replaces the edit window contents with it.

❖ Tip: *You can run a recipe that generates another recipe. Use #edit to put the generated recipe in the edit box ready to run.*

## #with (alias #$)

Syntax: #with *macro*

…*script…*

#end

The here-text is interpreted as a text manipulation script, operating on the given macro, and assigning the result back to that macro.

The script language has about 50 commands and is described in the chapter *Text Tools*. This turns the recipe program into quite a powerful tool.

❖ Tip: *If even more complex operations on a text have to be performed, invoke a full scripting language like Ruby, Python with* #cmd *(Windows old style cmd shell) or call PowerShell with* #ps*.*

## #run

Syntax: #run {*exe-or-doc*} *command-line*-*arguments*

This starts a program and passes arguments to it. The braces are mandatory, the command line arguments are optional. The program-or-doc can be an executable or an associated document, e.g.:

{notepad} c:\temp\my file.txt

{notepad++} c:\temp\my file.txt

{c:\temp\my file.txt}

{`myFavEditor`} c:\temp\`newfile`

{http://www.wiursaoieroiqeuwq.com}

❖ Tip: *use this to start up text editors to view or manually edit files after they have been changed. In the case of source version repository, use this to check in a changed file or add a new file to the repository (if you trust your recipe).*

## #cmd

Syntax: #cmd {*program-path*} *command-line*-*arguments*

Syntax: #cmd *macro*  
*here-text*  
#end

The first form works the same as #run, but the execution of the recipe is suspended until the program has exited. If the called program has not finished in 30 seconds, the recipe program will no longer wait for it, print a message and continue with the next command. The second form has an inline Windows command batch file as here-text that gets executed and its output (stdout) assigned to a macro.

Example:

#require drive  
  
#cmd dir-listing  
@echo off  
`drive`:  
cd `drive`:\  
dir /b \*.pdf | sort  
#end  
  
#info `dir-listing`

## #ps

Syntax: #ps *macro=PowerShellOneLiner*Syntax: #ps *macro  
 …PowerShell script…* #end

Invokes an inline Windows Powershell script and assigns the output to the given macro. In the first format, the text " | Out-String" is appended to the one-liner to make sure the output is a string. In the second format, the script itself has to make sure it outputs a string.

Example

#ps list

$folders = [system.Enum]::GetValues([System.Environment+SpecialFolder])

"Folder Name Path"

"----------- -----------------------------------------------"

foreach ($folder in $folders) {

"{0,-22} {1,-15}" -f $folder,[System.Environment]::GetFolderPath($folder)

}

#end

#write `list`

Result

Folder Name Path

----------- -----------------------------------------------

Desktop C:\Users\user\Desktop

Programs C:\Users\user\AppData\Roaming\Microsoft\Windows\Start Menu\Programs

Personal C:\Users\user\Documents

Personal C:\Users\user\Documents

Favorites C:\Users\user\Favorites

Startup C:\Users\user\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\ … Recent C:\Users\user\AppData\Roaming\Microsoft\Windows\Recent

SendTo C:\Users\user\AppData\Roaming\Microsoft\Windows\SendTo

StartMenu C:\Users\user\AppData\Roaming\Microsoft\Windows\Start Menu

MyMusic C:\Users\user\Music

DesktopDirectory C:\Users\user\Desktop

MyComputer

Templates C:\Users\user\AppData\Roaming\Microsoft\Windows\Templates

ApplicationData C:\Users\user\AppData\Roaming

LocalApplicationData C:\Users\user\AppData\Local

InternetCache C:\Users\user\AppData\Local\Microsoft\Windows\Temporary Internet Files

Cookies C:\Users\user\AppData\Roaming\Microsoft\Windows\Cookies

History C:\Users\user\AppData\Local\Microsoft\Windows\History

CommonApplicationData C:\ProgramData

System C:\WINDOWS\system32

ProgramFiles C:\Program Files

MyPictures C:\Users\user\Pictures

CommonProgramFiles C:\Program Files\Common Files

## #csharp

Syntax: #csharp *macro  
 …C# code returning a string…* #end

Compiles C# source code in the here-text as a body of a method and then invokes that method. The return value of the method must be a string. The body can contain macros, see `method` in the example below.

#put method=Now

#csharp a

var dt = DateTime.`method` + TimeSpan.FromDays(-7.0);

return dt.ToString("dddd dd/MM HH:mm");

#end

#info `a`

Result, e.g.

@info Tuesday 27/08 20:49

The here-text is inserted into this code before compilation:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public class Program

{

public static void Main(string[] args) {}

public static string Get()

{

<<here-text>>

}

};

The Get() method is then called through reflection. Make sure to include namespaces in the types if they not found in the ones provided in the 'using' section above. You an only use code in the basic library.

## #git

Source control

(to do)

## #recipe (alias ##)

Syntax: #recipe *path*

Syntax: #recipe *retval*=*path*Syntax: #recipe *macro*

Syntax: #recipe *retval*=*macro*

This will execute a (sub)recipe located at the given path or contained in the given macro. The current dictionary of macros is used with the subrecipe. The subrecipe can alter the dictionary; these changes will be available to the calling recipe. Multiple levels of recipe calls are allowed, but the system does not allow direct or indirect recursion.

The subrecipe can contain #return statements that returns a value that can be assigned to a macro to the left of the equals sign. Alternatively the subrecipe can just add an entry to the dictionary that is retrieved by the calling recipe after the #recipe command. See below.

## #return

Syntax: #return  
Syntax: #return *text*

Returns from a recipe call immediately, optionally passing the text to the caller. When the end of a recipe is reached, #return (without a value) is implicit.

Example:

#cdata r1

#if input ^[Yy]

#return yes

#endif

#return no

#end r1

#require input

#recipe answer=r1

#write `answer`

❖ Tip: *use return values so you don't have to remember which macro contains the value produced by the recipe.*

## #stop

Syntax: #stop  
Syntax: #stop *macro regex*

Stop execution of the whole recipe stack. This is different from a #return, which only returns from execution of the current recipe on the call stack.

#stop can be used inside an #if statement to conditionally stop execution, but it is also has a second form that takes the same parameters as #if: a macro name and a regex. If the macro matches the regex, then execution stops.

#if input `rxTrue`

#stop

#endif

is functionally the same as

#stop input `rxTrue`

There is no form for a negated match, so in that case #ifnot has to be used.

## #iter, #iter .. #next

Syntax: #iter *list\_macro recipe\_macro*

Syntax: #iter *list\_macro recipe\_macro loop\_var*

Syntax: #iter *list\_macro  
 … recipe commands containing `\_` to represent next line …* #next

This is an iteration construct in three different forms. The **first form** has two arguments which are macro names. The second macro must contain a recipe. The command iterates over the lines of text contained in the first macro, assigns them one by one to a macro with the same name (*list\_macro*) while executing the recipe for each line. Empty lines are skipped. An example will make this clear:

The file *c:\temp\subroutine.txt* contains:

#append result  
line `i`  
  
#end  
  
#new c:\temp\file\_`i`.txt  
file `i`  
#end

We execute this recipe that uses the subroutine recipe:

#put result=  
#load rcp1=c:\temp\subroutine.txt  
  
#put i  
one  
two  
three  
#end  
  
#iter i rcp1  
  
#info `result`

The main recipe will create an empty “result” macro, loads the subroutine into “rcp1”, creates a list of items called “i”, and then iter will first assign “one” to “i” and invoke the subroutine, then it will assign “two” to “i” and again invoke the subroutine, and finally the same for “three”. The subroutine will first append the text “line one” to the value of “result” and then create a file *c:\temp\file\_one.txt* with the contents “file one”, when i=one, then the same for i=two etc. The end result is that three new files have been created and the value of “result” is

line one  
line two  
line three

The blank line before #end makes sure a new line is inserted for each time #append is called.

After #iter, the macro ‘i’ has been removed from the dictionary, so if the value of i is still needed, then one could assign it first to another macro before using it in the #iter command:

#put list  
one  
two  
three  
#end  
  
#putv i list  
#iter i rcp1

The #iter command has no form with a recipe path that can be used directly. The recipe has to be loaded into the dictionary for fast repeated execution. Quite often the list of values to use will be in a file. #load can be used to load those values into memory, or #find can be used to extract certain information from a file.

The **second form** makes it unnecessary to assign the list to a loop variable explicitly with #putv. Instead it can just be provided as a third parameter.

Instead of

#putv i list  
#iter i rcp1

one can write

#iter list rcp1 i

After the iterations, list is still valid, but the loop variable i is undefined.

The **third form** contains a recipe in the here text and uses a special macro `\_` to represent each line in the list:

#put list  
one  
two  
three  
#end  
  
#iter list

#write counting: `\_`

#next

Output:

@set list=one\r\ntwo\r\nthree

@iter \_ = one

counting: one

@eor

@iter \_ = two

counting: two

@eor

@iter \_ = three

counting: three

Note that in the third form it is not possible to nest an inner- #iter in the here-text. Also there can be only one `\_` active at a time.

## #for .. #next

(to do)

## #while/#until .. #loop

(to do)

## #if/#ifnot .. #else .. #endif

Syntax 1: #if macro regex  
...then-commands...  
#endif

Syntax 2: #if macro regex  
...then-commands...  
#else  
...else-commands...  
#endif

With #if it is possible to branch the recipe code execution based on the contents of a macro and a regular expression. In the first format, the commands between #if and #endif are executed only if a match of the regex is found within the macro’s contents. In the second format, the same is true, but in addition if no match is found only the part between #else and #endif is executed.

***Note****:* #if *commands cannot be nested lexically, but the nested parts can be moved into new macros which can be called using* #recipe*, see below*

#require x

#cdata one

#if x ^-

#info negative 1

#else

#info positive 1

#endif

#end one

#if x ^-?1$

#recipe one

#else

#info other

#endif

## #info

Syntax: #info *message*

Simply writes @info with the message to the output.

## #warn

Syntax: #info message

Writes @warn with the message to the output and in the GUI also pops up a message box ans waits for the user to click on OK.

## #email

Syntax: #email *recipient-list file-list  
 here-text* #end

This will send an email to the listed recipients, using the here-text as body and attaching the listed files. The first line and the here-text will be expanded. Recipients are separated by a semi-colon, as are the attachments. Between #email and *recipients* should be a single space, and between *recipients* and *attachments* should be a single space if there are attachments. The subject line is the value set by #marker, use it to set a different subject if needed.

#put smtp-sender=myemail@test

#put smtp-host=smtp.myserver.test

#email `name`@xyz.test;admin@xyz.test c:\temp\file1.rcp;c:\temp\file2.rcp

Hello `name`

The recipe has been executed. It is attached.

#end

The Smtp Host and the Sender email address have to be set up in the dictionary in the entries:

smtp-sender

and

smtp-host

## #udp

(to do)

## #write

Syntax: #write *message*Syntax: #write  
 *message* #end

Write is like #info, but it does not write the @info label. Command #info is meant for debugging and can be switched off in the *Window* menu by unchecking *Log* whereas #write still outputs in that case. The #write command can be switched off by unchecking the *Write* menu item.

*Note: when using the first format, make sure that the message is not empty or the system will get confused and think it’s the second form and start looking for an #end tag.*

In the command line version of Recipe, #write writes to console which can be redirected into a file. (to be implemented)

## #mbox

Syntax: #mbox *message*This pops up a window with the given message in the RecipeGUI program. The user has to click OK to continue execution.

## #ask

(to do)

## #del

Syntax: #del *path*

Deletes the file with the given path.

## #copy

Syntax: #copy text

Syntax: #copy

*here-text*

#end

Copies the argument text or here-text to the clipboard after expansion of any macros. This will *not* change the value of the paste marcro. Note that paste is only put in the dictionary before recipe start.

## #dir

(to do)

## #mkdir

(to do)

## #rmdir

(to do)

## #server

(to do)

## #post

(to do)

## #watch

(to do)

## #exit

(to do)

## #pause

Syntax: #pause *milliseconds*

Pauses execution of the recipe for the given number of milliseconds. This can sometimes be useful if an asynchronous operations, typically launched with #run, needs some time to complete.

Example

#run {myprog1.exe}

#run {myprog2.exe}

#run {myprog3.exe}

#pause 10000

#info Done

# Server Mode

(to do)

# File Watcher Mode

(to do)

# Text Tools (*#with*-scripts)

(to do) (see Text Tools document)

Append, ChkSum, Chomp, Count, DelBlank, Enclose, ETrim, First, FromBase64, FromUni, FromXX, IndentXml, Info, Insert, Join, JoinN, JoinX, LineNum, Match, MD5Ascii, MD5Utf8, mkguid, Modify, NoMatch, Num, Pad, Prefix, Repeat, Reverse, Sha256Ascii, Sha256Utf8, Sort, Split, Store, STrim, Subs, Suffix, Table, ToBase64, ToUni, ToXX, Trim, TwoCol, Unique, Validate, Wrap, XGuid, XMatch

# Command Line

A command line version is provided for calling the recipe tool from other batch files.

Usage:

recipe.exe recipe-file key1=val1 key2=val2...

Use double quotes around *key=value* if the value contains spaces. All required parameters must be given on command line. The program will not stop to prompt the user for values.

The key=value pairs put entries into the dictionary as if #put were used. When a #require is encountered during the execution of the recipe, the value must already be in the dictionary or an error will be printed and the executions cancelled. A #default will assign a value to the key if not provided on command line.

To make sure the option command like #default a=x|y|z in the recipe executes correctly, set the key on the command line. Failure to do that will assign "x|y|z" to a.

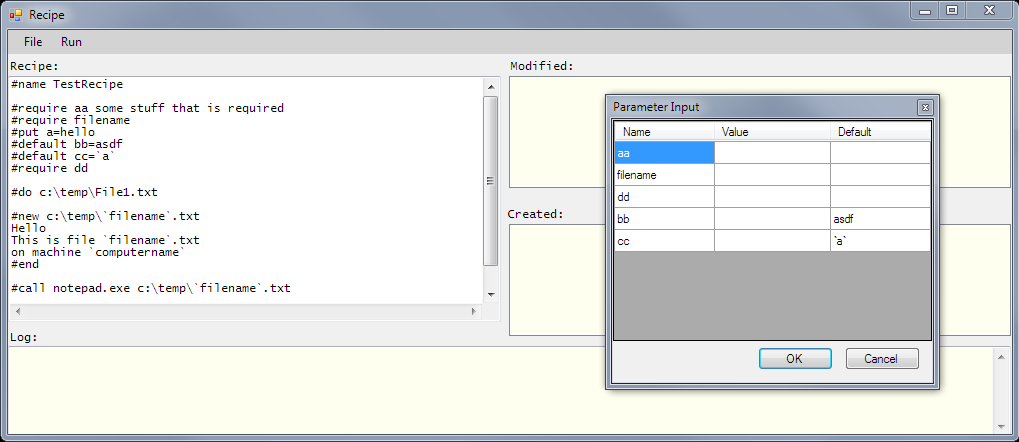
Note that clipboard features will not work (#copy and `paste`) in the command line version.

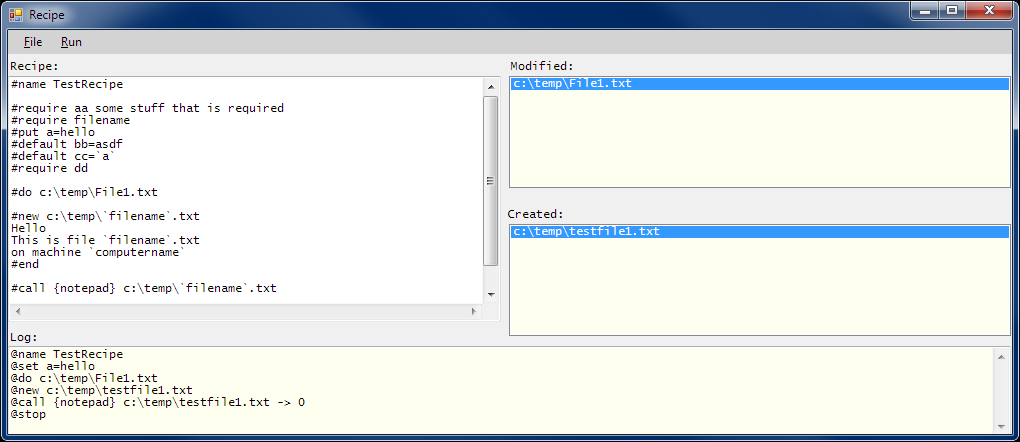
The command line cannot be run in test-mode.

# GUI

(to do)

- note: selected text is run only. If no text selected then all text is run





# Appendix 1: #do-block markers

The file extension determines which block markers are used.

## .cs (C#)

An #if block is used. One has to make sure that the block name is not set with #define or on command line to make sure the compiler skips it.

Example:

#if AddField

`cstype` `field`;

#endif

The compiler will skip the block. It is possible to use /\* \*/ comments inside the block. Do not nest #if. Do not use #else inside a block meant to be handled by the Recipe tool.

## .xml, .config, .html, .htm

A comment starting with the word "template" is used[[2]](#footnote-2). Example below shows how to append elements to an XML file.

<Log Name=’Ideas’>

<Idea t=’2010-11-01T12:13:14’>bla bla</Idea>

<!--template AppendIdea

<Idea t=’`timestamp`’>`text`</Idea>

-->

</Log>

After the keyword template and the block name the actualy template text starts on a new line.

## .bat

(to do)

GOTO :AddFile

copy "`newfile`" %folder%

:AddFile

This will expand the macro `newfile` and insert a line just before the block, e.g.:

copy "myfile.txt" %folder%

## Default

For all other types /\*\*\* and \*\*\*/ are used as block markers. This works well with e.g. with SQL.

/\*\*\* AddField

,[`FieldName`] [`Type`] `Constraint`

\*\*\*/

# Appendix 2: The #with Command

# Appendix 3: Command Summary

## Summary Recipe

The following recipe parses the C# source that contains the commands and creates a summary from it.

Please change the recipe path if required

#load m=C:\Util\Recipe\RecipeLib\Recipe.cs

#find case\ "(\p{Ll}+)"[^/]\*//\s\*([^\r\n]+) m

$1^$2

#end

#with m

table /^/ | /

sort

#end

#info `m`

## Summary Output

The output of the above recipe is printed below. Indentation has been adjusted for paper width.

after | insert after regex in macro or files

append | add to the end of a macro or files

ask | pop up form to ask user for input; input is text or multiple

choice

before | insert before regex in macro or files

call | run a program, wait until it finishes; or run here-text in cmd

shell; assign output

cdata | treat value as unparsed character data and store in dictionary,

read until #end + key name

copy | copy to clipboard

decrypt | decrypt text in macro with given key

default | put macro in dictionary only if not exists

del | delete file

dir | get files in directory; assign to macro

do | modify file, expand macros

edit | put contents of macro in editor window

email | send email, requires smtp-\* macros

encrypt | encrypt text in macro with given key

exit | exit recipe

exp | expand macros inside given macro

find | find regex in macro or files and generate here-text for each

match

forget | remove macro from dictionary

if | if macro regex // else // endif; don't next

ifnot | ifnot macro regex // else // endif; don't next

indentxml | #indentxml [-r] path\\*.ext

info | print text in log or console

intersect | #intersect m=m1 m2 m3... | intersection of lines

iter | iterate each line of given macro and invoke recipe

load | load content of file and store in dictionary

map | apply dictionary to macro or files; dictionary is macro with

key|value lines

mbox | pop up text in box

name | marker name to search for when using #do

new | create new file and copy following lines upto #end into it,

expand macros

pause | sleep for given milliseconds

point | restore point: remember all the macros

post | http post of here-text to given server

product | #product m=m1 m2 | carthesian product of two lists

ps | run a powershell program, wait till it finishes; assign output

put | put macro in dictionary

putv | put macro in dictionary, treat value as a lookup key and put the

value of that one in dict

putx | like putv, but expand the value before storing

rand | define a random number generator: key fmt min max seed

recipe | call given recipe

repeat | #repeat macro n

require | throw exception if macro does not exist

restore | restore all macros remembered with #point

run | run a program, don't wait for it to finish

seq | define a sequence number generator: key fmt min step max

server | server mode: run recipe commands for each incoming http request

sort | #sort rx macro-or-files

subs | substitute regex in macro or files

subtract | #subtract m=m1 m2 m3... | subtraction of lines

udp | send here-text as udp to addr port

union | #union m=m1 m2 m3... | union of lines, remove duplicates

unique | skip lines that already have the regex

warn | pop up warning and ask whether to continue or cancel

watch | file system watcher

with | apply script operations on the macro

write | write to output

# Appendix D: Examples

## Sets

This interactive recipe demonstrates *set operations* with the use of #ask in combination with a variable command (#`*macro*`)

#ask cmd

Which SET operation?

-

union|Union

intersect|Intersection

subtract|Subtraction

product|Product

#end

#put fruit1

apple

pear

orange

#end

#put fruit2

orange

lemon

#end

#`cmd` result=fruit1 fruit2

#info `result`

# Appendix E: To be documented

#auto

#count

#date

#esc

#exists

#map

#mode

#tags

#version

1. Term used in several shell script interpreters in Unix [↑](#footnote-ref-1)
2. A <?template … ?> construction was tried, but some xml interpreters complain about this. [↑](#footnote-ref-2)