MAC User's Guide

Version 3.03, February 12, 2014

Yves Leduc, Greg C. Warwar

MAC is based on an idea of Greg C. Warwar. The program was developed to add flexibility and interactivity to language manipulating data.

Language Purpose

MAC is a ***MAC***ro preprocessor dedicated to the preparation of ASCII data files. The program was developed to add flexibility and interactivity to language manipulating data.

MAC understands fairly simple directives: assignations, conditions, file inclusions and exits. It is interpreted. ***MAC*** is fully integrated into UNIX world as it understands the UNIX data pipe.

Unlike C preprocessor, process is strictly sequential. Input files are scanned line by line. A line beginning with the symbol '**#**' is considered as a macro directive. Macro directives modify the behavior of ***MAC***. Other lines are considered as data lines. MAC processes data lines according to the previously defined macro directives. To avoid loops, a recognized macro is replaced by its direct definition, ***MAC*** does not attempt to go further by evaluating the definition itself.

Each time it is possible, the syntax of ***MAC*** was kept identical to C preprocessors. Some features are suppressed (macro functions, pragma directives...). ***MAC*** does understand features not handled by C preprocessors (default value, indirection, messages, interactive mode, prefixes, arithmetic computation including usual mathematical functions, ...).

MAC is able to analyze inputs and to send warning and error messages. Debug is eased thanks to various integrated trace capabilities including 'verbose' mode and several levels of comments.

MAC supports two types of input: standard input "*stdin*" and ASCII files. Output is directed to standard output "*stdout*". Warning, error messages and trace are directed to standard error output "*stderr*".

MAC is fast and can be made faster and more secure thanks to the use of macro prefixes.

MAC is written in ANSI-C and is currently running on UNIX machines like HP, Sun and on DOS machines.

Glossary

MACRO: Identifier which will be replaced by its definition. All definitions are stored internally as strings.

name: left value

definition: right value

DIRECTIVE: ***MAC*** command. Directives begin with symbol **#**. They modify the behavior of ***MAC***.

TOKEN: A token is any printable ASCII text taken as a whole by ***MAC***. If punctuation marks, weird symbols or space have to be included into a single token, use single quotes to delimit the token.

These symbols are not considered as delimiters: **a...z A...Z 0...9 $ _ ( ) ! = < > | & [ ] %**

OPERATION: Usual monadic and dyadic functions as defined in C language. Operations are applied only to numbers (internal type for operations: double precision).

SEQUENCE: Set of tokens delimited by one or more delimiters.

DATA: Input line that is not recognized as a macro directive. MAC processes data according to the previous ***MAC*** directives.

BLOCK: Set of contiguous input lines delimited by **#if**, **#ifdef**, **#ifndef**, **#else** or **#endif**.

LIST: Set of tokens delimited by commas, and declared as a list by dedicated directives (this is an advanced feature).

Convention Used in this Guide

NAM:	macro identifier (must be a token beginning by a letter, $ or _).
DIG:	one digit (0..9).
VAL:	Value or macro identifier.
EXPR:	Expression containing recognized functions, macro identifiers or numbers.
COMMENT:	any text ignored by ***MAC*** (not processed nor output).
[XXX]:	at most one, i.e. xxx is optional, a sequence of xxx is not authorized.
XXX:	exactly one, i.e. xxx is mandatory, a sequence of xxx is not authorized.
[XXX..]:	0, 1 or more, i.e. xxx is optional, a sequence of xxx is authorized.
XXX [XXX..]:	1 or more, i.e. xxx is mandatory, a sequence of xxx is authorized.
➔	keyboard input.
➔	screen output.

MAC Command Line

MAC [option..] [filename..]

Options and filename can be mixed in any order. Options are processed immediately from left to right. Files are stored into a stack waiting for a sequential processing. First file to be processed is the last one.

MAC is compatible with UNIX pipes (see option "*-s[tdin]*"):

```
... UNIX_Command | MAC -s | UNIX_Command ...
... UNIX_Command | MAC -s > output_file
```

Command Line Options

-u [sage]

Usage on standard output.

This command produces a short usage on the standard output (see also appendix A).

When this option is checked, other options are ignored.

-b [uilt]

Built information on standard output.

When this option is checked, other options are ignored.

-h [elp]

Help on standard output.

This command produces a help panel on the standard output(see also appendix B).

When this option is checked, other options are ignored.

-s [tdin]

Standard input filter.

MAC is considering the standard input "*stdin"* as its only input, if no input files are provided. Keyboard entry is possible or data can be piped from other UNIX commands.

If input files are provided together with "*stdin*", all input files will be always processed before standard input.

-e [cho] file_pathname

Echoes macros from standard input into file.

It is sometimes interesting to trap the keyboard entry to record a ***MAC*** session.

If the file <file_pathname> already exists, or if ***MAC*** is unable to open it, ***MAC*** will stop, producing an error message.

It is strongly recommended to use single quotes around the pathname as UNIX parser could break its pathname into pieces.

All macro directives input through standard input "*stdin*" will be recorded into the echo file. Data will not be recorded. Echo file is directly reusable by ***MAC***, as it is.

This option is automatically disabled, if option "*-s[tdin]*" is not chosen.

-d [efine] nam val

Equivalent to directive *"#define nam val"*.

It is strongly recommended to use single quotes around the value as UNIX parser could break value into pieces.

Value <val> is mandatory for this option.

-i [nt] nam expr

Equivalent to directive *"#int nam expr"*.

It is strongly recommended to use single quotes around the expression as UNIX parser could break expression into pieces.

Expression <expr> is mandatory for this option.

-r [eal] nam expr

Equivalent to directive *#real6 nam expr"*.

It is strongly recommended to use single quotes around the expression as UNIX parser could break expression into pieces.

Expression <expr> is mandatory for this option.

-f [ix] nam expr

Equivalent to directive *"#fix2 nam expr"*.

It is strongly recommended to use single quotes around the expression as UNIX parser could break expression into pieces.

Expression <expr> is mandatory for this option.

-v [erbose]

Verbose mode enforced, trace to standard error output.

Verbose option produces extended comments on standard error output "*stderr*" about the processing of the directives and data. This option should be used with keyboard entry or during debug.

This option enforces the verbose mode during all the ***MAC*** process and cannot be turned off by macro directive "*#verbose*".

It is strongly recommended to redirect output into an output file to separate trace generated by the verbose mode and data output. It is possible to redirect trace and output into separate file thanks to UNIX redirection (see examples next chapter).

-q [uiet]

Remove warning messages.

MAC is issuing some warning messages, especially when redefining existing macros, removing not existing objects... If messages hurt you, use this option to suppress warnings. This option is not generally recommended as it can hide precious warning messages.

-l [ine]

Replace macros by blank lines in data output.

By default, **MAC** does not output a blank line when processing a macro directive. It is sometimes requested that the processed output data remain on the same line number as the input. In any cases, with or without checking this option, ***MAC*** is able to produce error messages with correct error location.

-p [refix] nam

Defines prefix for macro identifier.

It is often convenient to use prefix for the macro identifier as input file is more readable. The processing is more straightforward, ***MAC*** is faster especially when you are building huge table of macros and processing huge amount of text. There is no new constraint to the names of the macros: macros continue to be recognized inside other directives but macros inside data will not be processed if their names do not begin by the prefixes. For internal reasons, if "*-p[refix]*" option is checked, the prefixes '**$**' and '**@**' are added automatically to the list of prefixes you have defined. They are the prefixes of predefined macros (**$N, $V, $D, $F, $T**) or the prefix of macro indirection (**@**).

-c [omment]

Comment line traced on standard error output.

There are two types of macro comments. Active comment beginning with "**##**" can be traced on standard error output *"stderr*" when this option is checked. It helps to check and document the flow of data processing. Other comment, the passive comment beginning with "**#*"** is never traced on "*stderr*" and can be used to document the source files.

[filename..]

Pathname of ASCII file(s) to be expanded sequentially. Process is done with the rule:

last in, first processed,

at the exception of standard input "*stdin*" (option -s[tdin]), which is always processed after the process of these files is completed.

It is strongly recommended to use single quotes around the filename as UNIX parser could break its pathname into pieces.

Not existing files or locked files will cause ***MAC*** to issue an error message and to exit.

MAC Predefined Macros

Some macros are predefined automatically by ***MAC***. They are not protected and can be overwritten during a ***MAC*** session. They are especially useful to document output file.

$N: empty string.

$D: current time and date at the beginning of the process.

$V: current version of ***MAC***.

$F: current file name.

$T: current file tag name (see directive "*#tag*").

Macro directives

Inside processed files, directives recognized by the MACro expander begin with the symbol **#**. ***MAC*** directives MUST be the first word of the line. Unrecognized directives will cause a warning.

#include val

File to be processed. <val> can be a pathname (single quotes are recommended if pathname contains weird symbols). If <val> is itself a macro, <val> is replaced by its definition.

Including a file that cannot be accessed causes an error message and ***MAC*** exit.

Pathname <val> must be a token.

#insert val

File to be copied without processing. <val> can be a pathname (single quotes are recommended if pathname contains weird symbols). If <val> is itself a macro, <val> is replaced by its definition.

Inserting a file that cannot be accessed causes an error message and ***MAC*** exit.

Pathname <val> must be a token.

#define nam [val]

Definition without arithmetic evaluation. <nam> must be an identifier. If <val> is itself an existing macro, <val> is replaced by its definition.

If <val> is omitted, and macro was not yet defined, macro is defined as containing an empty string.

If <val> is omitted and macro already exists, its definition is reevaluated, i.e. <val> will be scanned for macro replacement.

It is allowed to redefine a macro if macro is empty. Redefining a macro with a new definition causes a warning message. <val> does not need to be a token but leading and ending spaces are dropped.

Use single quotes as delimiter if you want to include leading or ending spaces. Use back quotes to protect definition from processing.

#default nam [val]

If <nam> is already defined, the "*#default*" directive is ignored. The behavior of this directive is similar the directive "*#define*".

<val> does not need to be a token, leading and ending spaces are dropped.

#undef nam [nam..]

Undefine a list of macros. Macros are removed and memory is freed.

Removing a macro that is not existing causes a warning message.

#int nam [expr]

[Definition with] evaluation and casting to integer (truncation).

This directive is similar to "*#define*" but arithmetic evaluation is performed, data are cast to integer and stored as a string.

If no <expr> is provided, the definition of <nam> is cast to **int** and stored as a string.

If results cannot be expressed as a number, an error message is issued.

#real [dig] nam [expr]

[Definition with] evaluation and casting to floating real (scientific notation with <dig> digits for decimal part, default = 6).

This directive is similar to "*#define*" but arithmetic evaluation is performed, data are cast to float and stored as a string.

If no <expr> is provided, the definition of <nam> is cast to **double** and stored as a string.

If results cannot be expressed as a number, an error message is issued.

#fix [dig] nam [expr]

[Definition with] evaluation and casting to fixed real (decimals fix notation with precision equal to <dig>, default = 2).

This directive is similar to "*#define*" but arithmetic evaluation is performed, data are cast to fixed real and stored as a string.

If no <expr> is provided, the definition of <nam> is cast to **fixed real** and stored as a string.

If results cannot be expressed as a number, an error message is issued.

#cat nam [val..]

Concatenate a sequence of <val> to the definition of <nam>.

If <val> is itself a macro, its definition is used.

If <nam> does not exist, the macro is created and its definition contains the result of the concatenation of <val>.

<val> must be a token or a sequence of tokens.

#ask nam [val]

If <nam> is already defined, the "*#ask*" directive is ignored.

If <nam> is not yet defined, message <val> is displayed on stderr and ***MAC*** is waiting for input data from standard input "*stdin*".

If <val> contains macros, these macros are replaced by their definition.

The behavior of this directive is similar the directive "*#default*". Message <val> can be omitted, although it does not make sense.

#freeze nam [nam..]

Put single back quotes around the definition of <nam>. All previous back quotes are removed. New back quotes are placed at the begin and at the end of the definition. This directive can process a sequence of macros. Definition is then protected from further processing.

#melt nam [nam..]

Remove all single back quotes from definition of <nam>. This directive can process a sequence of macros. The definition is no more protected against processing. This directive itself does not try to reevaluate the definition.

#if expr

Execute "if block" when <expr>, evaluated and cast to integer, is not zero. If <expr> cannot be evaluated and/or cannot be cast to integer, an error message is issued.

See also "*#else*" and "*#endif*" directives.

#ifdef nam [val]

Execute "ifdef block" when name is defined equal to <val> (string comparison). If <val> is itself a macro, during test <val> is replaced by its definition. If <val> is omitted, macro is tested to be existing, its content is not checked.

See also "*#else*" and "*#endif*" directives.

<val> must be a token.

#ifndef nam [val]

Execute "ifndef" block when name is defined not equal to value (string comparison). If <val> is itself a macro, during test <val> is replaced by its definition. If <val> is omitted, macro is tested to be not existing.

See also "*#else*" and "*#endif*" directives.

<val> must be a token.

#else [comment]

"else block". Complete the "*#if*", "*#ifdef*" or "*#ifndef*" directives. "if block" must be complete inside a file (if, [else,] endif).

It is not allowed to begin with a if directives in a file and complete it by a else directive in another one. Comment is always ignored.

#endif [comment]

End of conditional block. If block must be complete inside a file (if, [else,] endif).

It is not allowed to begin with a if directives in a file and close it by a "*#endif*" in another one. If the number of "*#endif*" does not match the number of if directives, ***MAC*** will abort with an error message. Comment is always ignored.

#msg [val]

Send message <val> to standard error output "*stderr*".

If <val> is a macro, it is replaced by its definition. If <val> is omitted, a blank line is sent to standard error output "*stderr*".

#exit [val]

Exit from current file. ***MAC*** will continue to process remaining files but ***MAC*** normal termination occurs if nothing more has to be processed.

Message <val> is issued on standard error output "*stderr*".

#quit [val]

MAC normal termination whatever remaining files to be processed exist.

Message <val> on standard error output "*stderr*".

#abort [val]

Abnormal termination, <val> is sent to standard error output "*stderr*".

If <val> is a macro, it is replaced by its definition.

#stop [val]

Enforce unconditional ***MAC*** termination. It is ALWAYS processed, independently of any if condition, "*skip*" directive... This directive is useful when using ***MAC*** in interactive mode. Message <val> on standard error output "*stderr*".

#skip [expr]

Toggle Skip Mode On|Off [or set/reset if <expr> is (not) zero].

It is ALWAYS processed, independently of any if condition... When skip mode is enforced, directives and data are skipped.

#verbose [expr]

Toggle Verbose Mode On|Off [or set/reset if <expr> is (not) zero].

This directive cannot turn off the command line option "*-v[erbose]*". Every line is analyzed and processing is detailed on standard error output "*stderr*".

#macro [nam..]

List all macros on standard error output "*stderr*" [or a sequence of macros by name]. This directive is used for debug.

#status [val]

Show status of ***MAC*** variables on standard error output "*stderr*". This directive is used for debug.

<val> is sent to standard error output "*stderr*" as status title . If <val> is a macro, it is replaced by its definition.

It is ALWAYS processed independently of any if condition.

[val]

MAC single line active comment,. In comment mode, <val> is traced on standard error output "*stderr*".

#* [comment]

MAC single line passive comment. <comment> is always ignored and never traced.

#tag [val]

Define a file tag name for currently processed file. New tag is <val> and will be used in error or warning messages. Without tag directive, the default file tag name is the file name itself.

#list nam

Definition of previously defined macro <nam> is considered as an unidimensional list of elements separated by commas. A list of macros nam[0], nam[1],.. is created.

This is an advanced feature. Its use will be documented later.

Functions and expressions

Functions recognized by the MACro expander

Operations follow the C language convention (power function excepted: symbol **^**) applied to double. All operations apply ONLY to numbers or to macros that can be evaluated as numbers. All operation produces numbers.

TAKE CARE: Monadic functions MUST be enclosed between parenthesis (see below)

Grouping sub expressions

Parenthesis is used to group sub expressions. It is recommended to use parenthesis as often as possible to remove ambiguity.

✘ Parser is not complying priority rules of C (new parser is in development).

MAC Monadic functions

+	**-**
!	
(sin ())	**(asin ())**
(cos ())	**(acos ())**
(tan ())	**(atan ())**
(sinh ())	
(cosh ())	
(tanh ())	
(exp ())	**(log())**
(log10 ())	
(abs ())	**(sign ())**

(floor ())	(ceil ())	(round ())

MAC Dyadic functions

+	-	*	/	%	^
>	<	>=	<=	==	!=
||	&&				

MAC Indirection

[@..]nam

Indirection is possible using symbol(s) '@' ahead a macro names. MAC handles multiple level of indirection.

Representation of strings and numbers

MAC is storing data as strings in an internal dictionary.

The macro dictionary is updated by many directives. It is possible to obtain an image of the content of the dictionary using directive "*#macro*". Memory is allocated dynamically. As a consequence, the dictionary is reordered as soon as a directive has to remove or update a previous definition.

Text

Quotes can be used to protect text from evaluation or to determine field (space, weird symbols...). Quotes are interpreted slightly differently in directives and data:

'text' (directive): Determine field, quotes are removed in output. The text is considered as a whole (single token).

'text' (data): No effect. Quotes are considered as regular single characters. The text is processed by tokens.

`text` (data or directive): Text unprocessed by ***MAC***, back quotes are removed in output. The text is considered as a whole (single token).

"text" (data or directive): No effect. Double quotes are considered as regular single characters. The text is processed by tokens.

Strings are converted to numbers (C number of type double) when requested. All arithmetic is performed with double precision. Data are cast to integer or real at output as requested by dedicated directive.

Numbers accepted by *MAC*

Convention follows FORTRAN, Pascal or C languages.

Symbols reserved to *MAC*

Do not use back quotes or '@' in data lines except for text protection (`) or indirection (@)

Examples by topics

Example 1: usage and help (option)

-u[sage]
-h[elp]

Command Line: **`MAC -u`**

Result: The latest usage of ***MAC*** will be shown on screen (see also appendix A).

Command Line: **`MAC -h > 'foo.help'`**

Result: File 'foo.help' will contain the latest help of ***MAC*** (see also appendix B).

Command Line: **`MAC -h -s toto > 'foo.help'`**

Result: Result identical to previous example. Options other than "*-h*" are ignored.

Example 2: standard input, echo, verbose (option).

-s[tdin]
-e[cho] pathname
-v[erbose]

Command Line: **`MAC -s`**

Result: **MAC** is waiting for input from keyboard. Use exit directive ("*#exit*", "*#quit*" or "*#stop*") to quit keyboard input. Use keyboard to input line by line macro directives and data. As input and output are interleaved on screen, it is recommended to redirect output to a file. When using the keyboard as input, it is also recommended to use option "*-v[erbose]*" to track mistakes:

Command Line: **`MAC -s -v > foo.out`**

Result: ***MAC*** is waiting for input from keyboard. Verbose mode is enforced. Trace of process appears to screen. Result is redirected to file 'foo.out' and does not mix with the trace.

Command Line: **`MAC -s -v -e '/user/JDoe/foo.echo' > foo.out`**

Result: ***MAC*** is waiting for input from keyboard. Verbose mode is enforced. Trace of process appears to standard error output "*stderr*". Result is redirected to file 'foo.out' and does not mix with the trace. File 'foo.echo' stores all macro directives (and only macro directives) you have input from keyboard through standard input "*stdin*".

Command Line: **`(MAC  -v foo.in > foo.out ) >& foo.err`**

Result: ***MAC*** output is redirected towards 'foo.out'. Trace or messages are redirected towards 'foo.err' (UNIX).

Here is the transcription of a ***MAC*** session.

```
➔     %       MAC -s -v -e foo.echo > foo.out
🡐
🡐     Option -e[cho]: Standard Input will be echoed into file 'foo.echo'.
🡐     Option -v[erbose]: Verbose mode is enforced, trace on stderr.
🡐
🡐     Input files to be sequentially processed:
🡐     [1]: Standard Input
🡐
🡐     No prefix defined, all macros are global.
🡐
🡐     Macros currently defined:
🡐     [1]: (global) <$N>     <>
🡐     [2]: (global) <$V>     <APOLLO V3.01>
🡐     [3]: (global) <$D>     <Sat Oct 17 15:47:38 1992>
🡐     [4]: (global) <$F>     <Standard Input>
🡐     [5]: (global) <$T>     <Standard Input>
🡐
🡐     (FILE) -> processing 'Standard Input',       line 0
➔     #quit
🡐
🡐     (EXIT) -> from file  'Standard Input',       line 1
🡐
🡐     *** CALL FOR EXIT, MAC NORMAL TERMINATION ***
🡐     %
```

Example 3: define, int, real, fix (option).

-d[efine] nam val

-i[nt] nam expr

-r[eal] nam expr

-f[ix] nam expr

Command Line: **`MAC -s -v -i num 123`**

Result: ***MAC*** is waiting for input from keyboard. Verbose mode is enforced. Macro 'num' is defined as an integer equals to '123'.

Command Line: **`MAC -s -v -d text1 'Hello world!' -i num 123 -i c 34`**

Result: ***MAC*** is waiting for input from keyboard. Verbose mode is enforced. Macros 'text1', 'num' and 'count' are defined.

Here is the transcription of this last ***MAC*** session.

```
➔   % MAC -s -v -d text1 ' Hello world!' -i num 123 -i count 34
🡐
🡐   Option -v[erbose]:  Verbose mode is enforced, trace on stderr.
🡐
🡐   Input files to be sequentially processed:
🡐   [  1]:   Standard Input
🡐
🡐   No prefix defined, all macros are global.
🡐
🡐   Macros currently defined:
🡐   [1]: (global) <$N>     <>
🡐   [2]: (global) <$V>     <APOLLO V3.01>
🡐   [3]: (global) <$D>     <Sat Oct 17 16:00:04 1992>
🡐   [4]: (global) <text1>  < Hello world!>
🡐   [5]: (global) <num>    <123>
🡐   [6]: (global) <c>      <34>
🡐   [7]: (global) <$F>     <Standard Input>
🡐   [8]: (global) <$T>     <Standard Input>
🡐
🡐   (FILE) -> processing 'Standard Input',       line 0
➔   #quit
🡐
🡐   (EXIT) -> from file  'Standard Input',       line 1
🡐
🡐   *** CALL FOR EXIT, MAC NORMAL TERMINATION ***
🡐    %
```

Example 4: quiet mode (option).

-q[uiet]

Command Line: **`MAC -s -q`**

Result: ***MAC*** will issue no warning messages at all.

Example 5: prefix (option).

-p[refix] nam

Command Line: **`MAC -s -v -p 'RULE_' -p '$D' > foo.out`**

Here is the transcription of this ***MAC*** session.

```
➔   % MAC -s -v -p 'RULE_' -p '$D' > foo.out
🡐
🡐   Option -p[refix] :  Prefix '$' and '@' are automatically added by MAC.
🡐   Option -v[erbose]: Verbose mode is enforced, trace on stderr.
```

```
←
←     Input files to be sequentially processed:
←     [  1]: Standard Input
←
←     Prefixes currently defined:
←     [1]: (system )  <$>
←     [2]: (system)   <@>
←     [3]: (user)     <RULE_>
←     [4]: (user)     <$D>
←
←     Macros currently defined:
←     [1]: (global) <$N>     <>
←     [2]: (global) <$V>     <APOLLO V3.01>
←     [3]: (global) <$D>     <Sat Oct 17 17:32:52 1992>
←     [4]: (global) <$F>     <Standard Input>
←     [5]: (global) <$T>     <Standard Input>
←
←     (FILE) -> processing 'Standard Input',   line 0

➔     #quit

←
←     (EXIT) -> from file  'Standard Input',   line 1
←
←     *** CALL FOR EXIT, MAC NORMAL TERMINATION ***
←     %
```

In this session, all data line containing a macro beginning by 'RULE_', by '$D', by '$' or by '@' will be replaced by its definition. Other macros in data lines will be ignored. Processes of macros directives are not affected by prefixes.

Example 6: blank line (option).

-l[ine]

Command Line: **`MAC -s -l > foo.result`**

Result: Each macro directive will be replaced by a blank line. Without option "-*l*", a macro directive does not produce a blank line in the output file.

Example 7: comment mode (option).

-c[omment]

Command Line: **`MAC foo.in -c > foo.result`**

Result: line beginnings by "**##**" are traced on standard error output.

Example 8: input files (command line).

filenames

Command Line: **`MAC -v 'file1' -s 'file2' '/user/Doe' > foo.result`**

Result: Input files are processed, beginning by the last one. Standard input will be processed after all input files.

Here is the transcription of this ***MAC*** session.

```
➔     % MAC2 -v 'file1' -s 'file2' '/user/Doe' > foo.result
←
←     Option -v[erbose]: Verbose mode is enforced, trace on stderr.
←
←     Input files to be sequentially processed:
←     [1]: /user/Doe
←     [2]: file2
←     [3]: file1
←     [4]: Standard Input
←
←     No prefix defined, all macros are global.
←
←     Macros currently defined:
←     [1]: (global) <$N>     <>
←     [2]: (global) <$V>     <APOLLO V3.01>
←     [3]: (global) <$D>     <Sat Oct 17 18:04:34 1992>
←     [4]: (global) <$F>     <Standard Input>
←     [5]: (global) <$T>     <Standard Input>
←
←     (FILE) -> processing '/user/Doe',   line 0
←
etc. etc.
←     %
```

Example 9: predefined macros.

Predefined macros

Five macros are currently predefined by ***MAC***: $N, $V, $D, $F and $T. They can be used as user defined macros.

```
➔     % MAC -s -v > foo.out
←
←     Option -v[erbose]: Verbose mode is enforced, trace on stderr.
←
←     Input files to be sequentially processed:
←     [1]: Standard Input
←
←     No prefix defined, all macros are global.
←
←     Macros currently defined:
←     [1]: (global) <$N>     <>
←     [2]: (global) <$V>     <APOLLO V3.01>
←     [3]: (global) <$D>     <Sat Oct 17 15:47:38 1992>
←     [4]: (global) <$F>     <Standard Input>
←     [5]: (global) <$T>     <Standard Input>
←
←     (FILE) -> processing 'Standard Input',      line 0
➔     #quit
```

```
<-
<-      (EXIT) -> from file  'Standard Input',       line 1
<-
<-      *** CALL FOR EXIT, MAC NORMAL TERMINATION ***
<-      %
```

Example 10: define, default, undef, int, real, fix.

#define nam [val]

#default nam [val]

#undef nam [nam..]

#int nam [expr]

#real nam [expr]

#fix [dig] nam [expr]

```
->      % MAC foo -d C 123 > foo.out
<-      %
```

Where input file 'foo':

```
#define  A ' Hello World! '        (text including spaces)
#define  B val1                    (token not defined as a macro)
#default C 999                     (default not used as C is already defined)
#default D toto                    (default used as D is not already defined)
#int     E 432.1                   (number with casting to integer)
#real    F 1                       (number with casting to real (scientific))
#fix2    G 123.4567                (number with casting to real (two decimals))
#define  H B                       (using definition of another macro)
#define  I C                       (using definition of  another macro)
#int     J ((abs(-1)) + 12) / 7    (evaluation of an expression and casting)
#real    K (exp(1))                (evaluation of an expression and casting)
#fix     L 20 * (log10(1000))      (evaluation of an expression and casting)
#define  M 'my name is D'          (text where the token D is already defined)
#int     N C + 2                   (number, token C is already defined)
#define  O                         (macro with empty definition)
#define  toto qwerty               (token not defined as a macro)
macros are defined now             (data [ 1] without token defined as macro)
`A`    =    A                      (data [ 2] with token defined as macro)
`B`    =    B                      (data [ 3] with token defined as macro)
`C`    =    C                      (data [ 4] with token defined as macro)
`D`    =    D                      (data [ 5] with token defined as macro)
`E`    =    E                      (data [ 6] with token defined as macro)
`F`    =    F                      (data [ 7] with token defined as macro)
`G`    =    G                      (data [ 8] with token defined as macro)
`H`    =    H                      (data [ 9] with token defined as macro)
`I`    =    I                      (data [10] with token defined as macro)
`J`    =    J                      (data [11] with token defined as macro)
`K`    =    K                      (data [12] with token defined as macro)
`L`    =    L                      (data [13] with token defined as macro)
`M`    =    M                      (data [14] with token defined as macro)
`N`    =    N                      (data [15] with token defined as macro)
`O`    =    O                      (data [16] with token defined as macro)
`P`    =    P                      (data [17] without token defined as macro)
`toto` =    toto                   (data [18] with token defined as macro)
#undef   A C E G I                 (remove definition of macros)
#fix     F                         (casting of already defined macro)
#real    L                         (casting of already defined macro)
#define  D                         (macro D is already defined )
                                   (data [19] is a blank line)
macros are redefined now           (data [20] without token defined as macro)
```

```
`A`    =    A                    (data [21] without token defined as macro)
`B`    =    B                    (data [22] with token defined as macro)
`C`    =    C                    (data [23] without token defined as macro)
`D`    =    D                    (data [24] with token defined as macro)
`E`    =    E                    (data [25] without token defined as macro)
`F`    =    F                    (data [26] with token defined as macro)
`G`    =    G                    (data [27] without token defined as macro)
`H`    =    H                    (data [28] with token defined as macro)
`I`    =    I                    (data [29] without token defined as macro)
`J`    =    J                    (data [30] with token defined as macro)
`K`    =    K                    (data [31] with token defined as macro)
`L`    =    L                    (data [32] with token defined as macro)
`M`    =    M                    (data [33] with token defined as macro)
`N`    =    N                    (data [34] with token defined as macro)
`O`    =    O                    (data [35] with token defined as macro)
`P`    =    P                    (data [36] without token defined as macro)
`toto` =    toto                 (data [37] with token defined as macro)
```

Data lines contained into 'foo' are processed according to macro directives,

output is located into file 'foo.out':

```
macros are defined now      (corresponding to [ 1])
A    =     Hello World!     (corresponding to [ 2])
B    =    val1              (corresponding to [ 3])
C    =    123               (corresponding to [ 4])
D    =    toto              (corresponding to [ 5])
E    =    432               (corresponding to [ 6])
F    =    1.000000e+00      (corresponding to [ 7])
G    =    123.46            (corresponding to [ 8])
H    =    val1              (corresponding to [ 9])
I    =    123               (corresponding to [10])
J    =    1                 (corresponding to [11])
K    =    2.718282e+00      (corresponding to [12])
L    =    60.00             (corresponding to [13])
M    =    my name is toto   (corresponding to [14])
N    =    125               (corresponding to [15])
O    =                      (corresponding to [16])
P    =    P                 (corresponding to [17])
toto =    qwerty            (corresponding to [18])
                            (corresponding to [19])
macros are redefined now    (corresponding to [20])
A    =    A                 (corresponding to [21])
B    =    val1              (corresponding to [22])
C    =    C                 (corresponding to [23])
D    =    qwerty            (corresponding to [24])
E    =    E                 (corresponding to [25])
F    =    1.00              (corresponding to [26])
G    =    G                 (corresponding to [27])
H    =    val1              (corresponding to [28])
I    =    I                 (corresponding to [29])
J    =    F                 (corresponding to [30])
K    =    2.718282e+00      (corresponding to [31])
L    =    6.000000e+01      (corresponding to [32])
M    =    my name is toto   (corresponding to [33])
N    =    1                 (corresponding to [34])
O    =                      (corresponding to [35])
P    =    P                 (corresponding to [36])
toto =    qwerty            (corresponding to [37])
```

Example 11: include and insert.

```
#include val
#insert val
```

```
➔    % MAC foo -d fnm '/user/John/Doe/' > 'foo.out'
➔    %
```

Where input file 'foo':

```
title is included now

inserting file
#insert '/user/title'

processing file fnm
#include fnm

`A` is equal to A
`B` is equal to B

thank you for your attention
```

Inserted file: '/user/title'

```
EXAMPLE OF FILE INSERTION
#define A 123
`A` is not defined
```

Included file: '/user/John/Doe'

```
EXAMPLE OF FILE INCLUSION
#define B 234
`B` is defined
```

Data lines contained into 'foo' are processed according to macro directives, files are inserted (copied without processing) or included (copied with processing).

Output is located into file 'foo.out':

```
title is included now

inserting file
EXAMPLE OF FILE INSERTION
#define A 123
`A` is not defined

processing file /user/John/Doe
EXAMPLE OF FILE INCLUSION
B is defined

A is equal to A
B is equal to 234

thank you for your attention
```

Example 12: concatenation.

#cat nam [val..]

➔ MAC foo -d gen RULE > 'foo.out'

➔ %

Where input file 'foo':

```
#define A 1
#cat B gen '_' A
properly concatenated, the name of the rule is B
#define C ABRA
#cat C CAD C
C
```

Data lines contained into 'foo' are processed according to macro directives.

Output is located into file 'foo.out':

```
properly concatenated, the name of the rule is RULE_1
ABRACADABRA
```

Example 13: ask and message.

#ask nam [val]

#msg [val]

```
➔   % MAC foo -d A 123 > 'foo.out'
🡐   Hi dude!
🡐   Introduce number 2:
➔   234
🡐   Thank you!
🡐   %
```

Where input file 'foo':

```
#msg 'Hi dude!'
#ask A 'Introduce number 1:'          (skipped as A is already defined)
#ask B 'Introduce number 2:'
number 1 is equal to A
number 2 is equal to B
#define message  'Thank you!'
#msg message
```

Data lines contained into 'foo' are processed according to macro directives.

Output is located into file 'foo.out':

```
number 1 is equal to 123
number 2 is equal to 234
```

Example 14: freeze and melt.

#freeze nam [nam..]

#melt nam [nam..]

```
➔   % MAC foo > 'foo.out'
🡐   %
```

Where input file 'foo':

```
#define B A plus A
#define C A plus A
(B) is identical to (C)
#freeze B
#define A 123
#define B
#define C
(B) is no more identical to (C)
#melt B
#define B
(B) is identical to (C)
```

Data lines contained into 'foo' are processed according to macro directives.

Output is located into file 'foo.out':

```
(A plus A) is identical to (A plus A)
(A plus A) is no more identical to (123 plus 123)
(123 plus 123) is identical to (123 plus 123)
```

Example 15: if, else, endif, ifdef, ifndef.

#if expr

#ifdef nam [val]

#ifndef nam [val]

#else [comment]

#endif [comment]

➔ `% MAC foo -d AA something -d BB qwerty > 'foo.out'`

➔ `%`

Where input file 'foo':

```
#define AA
#ifdef AA
`AA` is defined, but empty or not (AA)
#endif
#ifdef BB qwerty
`BB` is defined (BB)
#endif
#ifdef CC
this line will be skipped
#else
this line is processed as `CC` is not defined
#endif
#ifndef DD
this line is processed as `DD` is not defined
#endif
#define var 123
#if ((var - 122) == 1)
Yes, (var - 122) == 1 !
#endif
#if (var > 0)
Yes, var is positive
#if (var > 100)
```

```
Yes, var is greater than 100
#endif
#endif
```

Data lines contained into 'foo' are processed according to macro directives.

Output is located into file 'foo.out':

```
AA is defined, but empty  (something)
BB is defined (qwerty)
this line is processed as CC is not defined
this line is processed as DD is not defined
Yes, (123 - 122) == 1 !
Yes, 123 is positive
Yes, 123 is greater than 100
```

Example 16: exit.

#exit [val]

```
➔     % MAC foo > 'foo.out'
➔     I quit as you ask it so kindly
➔     %
```

Where input file 'foo':

```
#ifdef NOTDEF
#exit 'skipped'
#else
#quit 'I quit as you ask it so kindly'
#endif
```

```
➔     % MAC foo > 'foo.out'
➔     I quit file /user/John/Doe
➔     %
```

Where input file 'foo':

```
#include '/user/John/Doe'
Welcome back to main file
```

and included file '/user/John/Doe':

```
Welcome to file '/user/John/Doe'
#exit 'I quit file '/user/John/Doe'
this part is skipped
```

Data lines contained into 'foo' are processed according to macro directives.

Output is located into file 'foo.out':

```
Welcome to file '/user/John/Doe'
Welcome back to main file
```

Example 17: quit.

#quit [val]

```
➔     % MAC foo > 'foo.out'
🡐     I quit as you ask it so kindly
🡐     %
```

Where input file 'foo':

```
#ifdef NOTDEF
#quit 'skipped'
#else
#quit 'I quit as you ask it so kindly'
#endif
```

Example 18: abort.

#abort [val]

```
➔     % MAC foo > 'foo.out'
🡐
🡐     line[1]  [foo]
🡐     MAC user error - Horror!
🡐     %
```

Where input file 'foo':

```
#ifdef NOTDEF
#quit 'skipped'
#else
#abort 'Horror!'
#endif
```

Example 19: stop.

#stop [val]

```
➔     % MAC foo > 'foo.out'
🡐     I am forced to quit
🡐     %
```

Where input file 'foo':

```
(...anything...)
#stop 'I am forced to quit '
(...anything...)
```

Example 20: skip.

#skip [expr]

```
➔     % MAC foo > 'foo.out'
➔     %
```

Where input file 'foo':

```
#define A 'Hello world!'
#define condition 1
#skip condition                          (skip mode is set to ON)
this text will be skipped
#define SKIPPED 'forget it'
#skip (condition - 1)                    (skip mode is set to OFF)
Here is the message: A
```

Data lines contained into 'foo' are processed according to macro directives.

Output is located into file 'foo.out':

```
Here is the message: Hello world!
```

"*#skip*" can be used as a toggle:

```
➔     % MAC foo > 'foo.out'
➔     %
```

Where input file 'foo':

```
#define A 'Hello world!'
#skip                                    (skip mode is toggled to ON)
this text will be skipped
#define SKIPPED 'forget it'
#skip                                    (skip mode is toggled to OFF)
Here is the message: A
```

Data lines contained into 'foo' are processed according to macro directives, output is located into file 'foo.out':

```
Here is the message: Hello world!
```

Example 21: verbose.

#verbose [expr]

```
➔     % MAC foo > 'foo.out'
➔
➔     (   3) ->  #verbose
➔     (....) <- <Verbose Mode>       <Toggled to  On>
```

```
←
←       (   4) ->  #define B 234
←       (....) <- <B>  <234>
←
←       (   5) ->  `B` is equal to B
←       (   2) <- <`B` is equal to 234 >
←
←       (   6) ->  #verbose
←       (....) <- <Verbose Mode>       <Toggled to Off>
←       %
```

Where input file 'foo':

```
#define A 123
`A` is equal to A
#verbose
#define B 234
`B` is equal to B
#verbose
#define C 345
`C` is equal to C
```

Data lines contained into 'foo' are processed according to macro directives.

Output is located into file 'foo.out':

```
 A is equal to 123
 B is equal to 234
 C is equal to 345
```

Example 22: macro listing.

#macro [nam..]

```
➔       % MAC foo > 'foo.out'
←
←       [  1]:  (  global)  <$N>     <>
←       [  2]:  (  global)  <$V>     <APOLLO V3.01>
←       [  3]:  (  global)  <$D>     <Tue Oct 20 18:31:23 1992>
←       [  4]:  (  global)  <$F>     <foo>
←       [  5]:  (  global)  <$T>     <foo>
←       [  6]:  (  global)  <A>      <123>
←       [  7]:  (  global)  <B>      <>
←       [  8]:  (  global)  <C>      <' Hello World!!! '>
←       [  9]:  (  global)  <D>      <3>
←
←       [  1]:  (  global)  <$N>     <>
←       [  2]:  (  global)  <$V>     <APOLLO V3.01>
←       [  3]:  (  global)  <$D>     <Tue Oct 20 18:31:23 1992>
←       [  4]:  (  global)  <$F>     <foo>
←       [  5]:  (  global)  <$T>     <foo>
←       [  6]:  (  global)  <B>      <>
←       [  7]:  (  global)  <C>      <' Hello World!!! '>
←       [  8]:  (  global)  <D>      <3>
←       %
```

Where input file 'foo':

```
#define A 123
#define B
#define C ' Hello World!!! '
#int    D 1 + 2
#macro
#undef A
#macro
```

Example 23: *MAC* status.

#status [val]

```
➔     % MAC foo > 'foo.out'
➔
➔     MAC Internal Status
➔
➔     File Name       :       'foo'
➔     File Tag Name   :       'foo'
➔     File Level      :       0
➔     Current Line    :       1
➔
➔     Output line     :       0
➔
➔     Evaluation Mode :       ON
➔     Skip Mode       :       OFF
➔     Prefix Mode     :       OFF
➔     Verbose Mode    :       OFF
➔     Line Mode       :       OFF
➔     Echo Mode       :       OFF
➔     Comment Mode    :       OFF
➔     Warning Mode    :       ON
➔
➔     %
```

Where input file 'foo':

```
#status
```

Example 24: active and passive comment.

[val] #* [comment]

```
➔     % MAC foo > 'foo.out'
➔     %
➔     % MAC foo -c > 'foo.out'
➔
➔     ****  comment number three
➔
➔     %
```

Where input file 'foo':

```
#ifdef AAA
## comment number one
#* comment number two
some data line
#else
## comment number three
#* comment number four
other data line
#endif
```

Data lines contained into 'foo' are processed according to macro directives.

Output is located into file 'foo.out':

```
other data line
```

Example 25: tag name.

#tag [val]

```
➔     % MAC foo > 'foo.out'
🡐
🡐     line[1]  [foo]
🡐     line[2]  [data file number 1]
🡐     MAC error - regular expression expected
🡐     %
```

Where input file 'foo':

```
#include '/user/John/Doe'
```

and include file '/user/John/Doe':

```
#tag 'data file number 1'
#int A  'not a number'
```

Example 26: indirection.

[@..]nam

```
➔     % MAC foo > 'foo.out'
🡐     %
```

Where input file 'foo':

```
#define B A
#define C A
( 1) `A` is equal to A
```

```
( 2) `B` is equal to B
( 3) `C` is equal to C
( 4) `D` is equal to D

#define @B D                          (one level of indirection)
(11) `A` is equal to A
(12) `B` is equal to B
(13) `C` is equal to C
(14) `D` is equal to D

#define @@B 123                       (two levels of indirection)
(21) `A` is equal to A
(22) `B` is equal to B
(23) `C` is equal to C
(24) `D` is equal to D
```

Data lines contained into 'foo' are processed according to macro directives.

Output is located into file 'foo.out':

```
( 1) A is equal to A
( 2) B is equal to A
( 3) C is equal to A
( 4) D is equal to D

(11) A is equal to D
(12) B is equal to A
(13) C is equal to A
(14) D is equal to D

(21) A is equal to D
(22) B is equal to A
(23) C is equal to A
(24) D is equal to 123
```

Example 27: text manipulations.

'text'	(directive)
'text'	(data)
`text`	(data or directive)
"text"	(data or directive)

➔ `% MAC foo > 'foo.out'`

➔ `%`

Where input file 'foo':

```
#define it  AH
( 1) there are several ways to say it
( 2) there are several ways to say 'it'
( 3) there are several ways to say `it`
( 4) there are several ways to say "it"
#define A  it
#define B 'it'
#define C `it`
#define D "it"

(11) there are several ways to say A
(12) there are several ways to say B
(13) there are several ways to say C
(14) there are several ways to say D
```

```
(21) there are several ways to say 'A'
(22) there are several ways to say 'B'
(23) there are several ways to say 'C'
(24) there are several ways to say 'D'

(31) there are several ways to say `A`
(32) there are several ways to say `B`
(33) there are several ways to say `C`
(34) there are several ways to say `D`

(41) there are several ways to say "A"
(42) there are several ways to say "B"
(43) there are several ways to say "C"
(44) there are several ways to say "D"
```

Data lines contained into 'foo' are processed according to macro directives.

Output is located into file 'foo.out':

```
( 1) there are several ways to say AH
( 2) there are several ways to say 'AH'
( 3) there are several ways to say it
( 4) there are several ways to say "AH"

(11) there are several ways to say AH
(12) there are several ways to say AH
(13) there are several ways to say it
(14) there are several ways to say "AH"

(21) there are several ways to say 'AH'
(22) there are several ways to say 'AH'
(23) there are several ways to say 'it'
(24) there are several ways to say '"AH"'

(31) there are several ways to say A
(32) there are several ways to say B
(33) there are several ways to say C
(34) there are several ways to say D

(41) there are several ways to say "AH"
(42) there are several ways to say "AH"
(43) there are several ways to say "it"
(44) there are several ways to say ""AH""
```

Errors and Warnings

Errors are causing immediate MAC abnormal termination. Complete error location is provided with line number and file name (hierarchical).

Warning messages may be disabled by using option "*-q[uiet]*" on command line.

Example of Command Line error

```
➔     %  MAC '/user/John/Doe' -d foo
🡐
🡐     [Command Line]
🡐     MAC error - missing parameter -d foo ?
🡐     %
```

Example of error generated by a bad definition of a macro at line 8 of file 'foo.include', called by a "*#include*" directive at line 6 of input file '/user/John/Doe'

```
➔     % MAC2 '/user/John/Doe' > foo.out
🡐
🡐     line[6] [/user/John/Doe]
🡐     line[8] [foo.include]
🡐     MAC error - regular expression expected
🡐     %
```

List of warning and error messages

```
( 1)  MAC error - XXXX identifier is missing
( 2)  MAC error - echo file already defined
( 3)  MAC error - arc cosine of number outside [-1, 1] is forbidden
( 4)  MAC error - arc sine of number outside [-1, 1] is forbidden
( 5)  MAC error - attempt to divide by 0
( 6)  MAC error - attempt to loop include files: 'XXXX'
( 7)  MAC error - attempt to rewrite existing echo file: 'XXXX'
( 8)  MAC error - casting a list is forbidden
```

```
( 9)  MAC error - dyadic operator expected
(10)  MAC error - empty value cannot be evaluated
(11)  MAC error - expected valid expression following '!='
(12)  MAC error - expected valid expression following '%'
(13)  MAC error - expected valid expression following '&&'
(14)  MAC error - expected valid expression following '*'
(15)  MAC error - expected valid expression following '+'
(16)  MAC error - expected valid expression following '-'
(17)  MAC error - expected valid expression following '/'
(18)  MAC error - expected valid expression following '<'
(19)  MAC error - expected valid expression following '<='
(20)  MAC error - expected valid expression following '=='
(21)  MAC error - expected valid expression following '>'
(22)  MAC error - expected valid expression following '>='
(23)  MAC error - expected valid expression following '^'
(24)  MAC error - expected valid expression following '||'
(25)  MAC error - file name is missing
(26)  MAC error - if type directive is missing
(27)  MAC error - illegal TYPE identifier <XXXX>
(28)  MAC error - logarithm of number lower or equal to 0 is forbidden
(29)  MAC error - maximum depth of file inclusion reached: NNNN
(30)  MAC error - maximum number of macros reached: NNNN
(31)  MAC error - memory allocation error
(32)  MAC error - missing filename for -e
(33)  MAC error - missing macro definition -d XXXX ?
(34)  MAC error - missing macro definition -f XXXX ?
(35)  MAC error - missing macro definition -i XXXX ?
(36)  MAC error - missing macro definition -r XXXX ?
(37)  MAC error - missing macro identifier -d ?
(38)  MAC error - missing macro identifier -f ?
(39)  MAC error - missing macro identifier -i ?
(40)  MAC error - missing macro identifier -r ?
(41)  MAC error - no input specified, verify command line
(42)  MAC error - power of negative number is forbidden
(43)  MAC error - problem to open echo file (MODE): 'XXXX'
(44)  MAC error - problem to open input file (MODE): 'XXXX'
(45)  MAC error - regular expression expected
(46)  MAC error - string too long, unable to concatenate (max = 512)
(47)  MAC error - too many endif
(48)  MAC error - too many prefixes
(49)  MAC error - try to break into elements an empty macro <XXXX>
(50)  MAC error - try to break into elements an undefined macro <XXXX>
(51)  MAC error - try to cast to fix an empty macro <XXXX>
(52)  MAC error - try to cast to int an empty macro <XXXX>
(53)  MAC error - try to cast to real an empty macro <XXXX>
(54)  MAC error - unable to build list definition, string too long (max = 512)
(55)  MAC error - unable to evaluate expression
(56)  MAC error - unbalanced if/endif
(57)  MAC error - unmatched parenthesis into expression <XXXX>
```

```
(58)  MAC error - unrecognized option: XXXX
(59)  MAC error - waiting for if or endif

(60)  MAC warning - disabling echo mode as standard input is not processed
(61)  MAC warning - '#XXXX' is not a legal MACro directive
(62)  MAC warning - attempt to remove an undefined macro <XXXX>
(63)  MAC warning - macro <XXXX> is not defined
(64)  MAC warning - macro <XXXX> is already a list
(65)  MAC warning - macro <XXXX> was already defined
(66)  MAC warning - try to cast to fix an undefined macro <XXXX>
(67)  MAC warning - try to cast to int an undefined macro <XXXX>
(68)  MAC warning - try to cast to real an undefined macro <XXXX>
(66)  MAC warning - try to freeze an undefined macro <XXXX>
(70)  MAC warning - try to melt an undefined macro <XXXX>
(71)  MAC warning - precision of fix number = NNN reduced to 9
```

Appendix A: MAC Usage

```
Simple MACro expander, Version DOS V3.03, WARW YLD FRTH JYCR, October 06 1992

Usage:   MAC [option..] [pathname..]

<option>:
-u[sage]            : Usage on stdout.
-h[elp]             : Help on stdout.
-s[tdin]            : Use MAC as filter accepting standard input (e.g. pipe).
-e[cho] 'pathname'  : Echo macros directives from stdin into file.  (*)
-d[efine] nam 'val' : Equivalent to directive #define nam val.      (*)
-i[nt] nam 'expr'   : Equivalent to directive #int nam expr.       (*)
-r[eal] nam 'expr'  : Equivalent to directive #real nam expr.      (*)
-f[ix] nam 'expr'   : Equivalent to directive #fix nam expr.       (*)
-p[refix] nam       : Prefix Mode,  declare prefix of macros used outside directives.
-v[erbose]          : Verbose Mode, detailed trace to stderr.
-c[omment]          : Comment Mode, send MAC comments to stderr.
-q[uiet]            : Quiet Mode,   remove warning messages.
-l[ine]             : Line Mode,    keep output and input lines numbering equal.

      (*) On command line, it is recommended to enclose value or expression between
          SINGLE quotes. They are mandatory if value or expression has more than one
          word, contains weird symbols...

<pathname>:

Pathname of ASCII file(s) to be expanded sequentially (last in is first processed).
```

Standard input is always processed after the process of these files is completed.

Appendix B: MAC Help

```
Simple MACro expander, Version DOS V3.03, WARW YLD FRTH JYCR, October 06 1992

Usage:   MAC [option..] [pathname..]

<option>:

-u[sage]            : Usage on stdout.
-b[uilt]            : Built information on stdout.
-h[elp]             : Help on stdout.
-s[tdin]            : Use MAC as filter accepting standard input (e.g. pipe).
-e[cho] 'pathname'  : Echo macros directives from stdin into file.  (*)
-d[efine] nam 'val' : Equivalent to directive #define nam val.      (*)
-i[nt] nam 'expr'   : Equivalent to directive #int nam expr.       (*)
-r[eal] nam 'expr'  : Equivalent to directive #real nam expr.      (*)
-f[ix] nam 'expr'   : Equivalent to directive #fix nam expr.       (*)
-p[refix] nam       : Prefix Mode,  declare prefix of macros used outside directives.
-v[erbose]          : Verbose Mode, detailed trace to stderr.
-c[omment]          : Comment Mode, send MAC comments to stderr.
-q[uiet]            : Quiet Mode,   remove warning messages.
-l[ine]             : Line Mode,    keep output and input lines numbering equal.

      (*) On command line, it is recommended to enclose value or expression between
          SINGLE quotes. They are mandatory if value or expression has more than one
          word, contains weird symbols...

<pathname>:

Pathname of ASCII file(s) to be expanded sequentially (last in is first processed).
Standard input is always processed after the process of these files is completed.

Operations recognized by the MACro expander during arithmetic evaluation:
```

```
+   -   *   /   %   ^   >   <   >=   <=   ==   !=   !   ||   &&
(sin())  (cos())  (tan())  (asin())  (acos())  (atan())
(exp())  (log())  (log10()) (sinh())  (cosh())  (tanh())
(abs())  (floor())  (ceil())  (round())  (sign())


Inside processed files, directives recognized by the MACro expander:

#define nam [val]    : Macro definition.
#default nam [val]   : Macro definition, if macro was not yet defined.
#undef nam [nam..]   : Undefine the macro[s].
#int nam [expr]      : [Macro definition with] arithmetic evaluation, casting to int.
#real[dig] nam [expr]: [Macro definition with] arithmetic evaluation, casting to real (scientific).
#fix[dig] nam [expr] : [Macro definition with] arithmetic evaluation, casting to real (fixed precision).
#ask nam [val]       : Macro definition by [question 'val' and] answer from user, if not yet defined.
#msg [val]           : Send message 'val' to stderr.
#cat nam [val..]     : Concatenate 'val..' into first macro.
#freeze nam [nam..]  : Protect content of macros from processing.
#melt nam [nam..]    : Unprotect content of macros from processing.
#if expr             : Execute if block when after arithmetic evaluation, 'expr' is not 0.
#ifdef nam [val]     : Execute ifdef block when 'nam' is defined [equal to 'val'].
#ifndef nam [val]    : Execute ifndef block when 'nam' is undefined [or not equal to 'val'].
#else [comment]      : Else block. ['comment' ignored.]
#endif [comment]     : End of conditional block. ['comment' ignored.]
#include val         : File to be included (pathname between single quotes).
#insert val          : Data file to copy without processing (pathname between single quotes).
#exit [val]          : Exit from current file. [Send message 'val' on stderr.]
#quit [val]          : Normal MAC termination. [Send message 'val' on stderr.]
#abort [val]         : Abnormal termination. [Send message 'val' on stderr.]
## [val]             : MAC single line comment. [In comment mode, 'val' is traced on stderr.]
#* [comment]         : MAC single line comment. ['comment' ignored.]

- MACro expander advanced features:

#list nam            : Declare existing macro as a list and split it into elements.
[@..]nam             : Indirection using symbol '@' ahead macro name 'nam'.

- Debug oriented macro directives (normally processed):
#verbose [val]       : Toggle verbose mode On|Off [or set/reset if value is (not) 0].
#macro [nam..]       : List all macros on stderr [or macros by name 'nam..'].
#tag [val]           : Define file tag name 'val' of currently processed file.

- Debug oriented macro directives (unconditionally processed):

#skip [expr]         : Toggle Skip Mode On|Off [set/reset if 'expr' is (not) 0].
#stop [val]          : MAC normal termination. [Send message 'val' on stderr.]
#status [val]        : Current status of MAC on stderr. ['val' is a title.]
```

Predefined macros:

```
$V                  : Current version of MAC (DOS V3.03).
$D                  : Current time and date at the beginning of the process.
$F                  : Current file name.
$T                  : Current file tag name (default: file name).
$N                  : Empty string.
```

Quotes in MAC:

```
'text'   (    option) : Field delimiters, quotes are removed by UNIX.
'text'   ( directive) : Determine field, quotes are removed in output.
'text'   (      data) : No effect. Quotes remain.
`text`   (dir & data) : Text unprocessed by MAC, quotes are removed at output.
"text"   (dir & data) : No effect. Quotes remain.
```

Valid macro identifier in MAC:

```
First character       : A..Z  a..z  $  or _
Other characters      : A..Z  a..z  0..9  [  ]  $  or  _
```