NAME

gema − general purpose macro processor

SYNOPSIS

gema *options...* [*input_file* [*output_file*]]

gema *options...* **−out** *output_file input_file...*

gema *options...* **−odir** *output_directory* [**−otyp** *output_suffix*] *input_file...*

Where *options* is:
−f *pattern_file* | [**−p**] '*patterns*' | **−b** | **−i** | **−k** | **−line** | **−match** | **−t** | **−w** | **−idchars** *charset* | **−filechars** *charset* | **−literal** *charset* | **−ml** | **−trace** | **−arglen** *num*

DESCRIPTION

gema is a general purpose text processing utility based on the concept of pattern matching. In general, it reads an input file and copies it to an output file, while performing certain transformations to the data as specified by a set of patterns defined by the user. It can be used to do the sorts of things that are done by Unix utilities such as **cpp**, **grep**, **sed**, **awk**, or **strings**. It can be used as a macro processor, but it is much more general than **cpp** or **m4** because it does not impose any particular syntax for what a macro call looks like. Unlike utilities like **sed** or **awk**, **gema** can deal with patterns that span multiple lines and with nested constructs. It is also distinguished by being able to use multiple sets of rules to be used in different contexts.

This **man** page provides a tutorial introduction and a brief reference summary. See the user manual for a more detailed specification.

If no file names are given on the command line, the program reads from standard input and writes to standard output. With two file names provided, it reads from the first and writes to the second. If the output file previously existed, the old file is saved with a "**.bak**" suffix. The input file and output file may be the same, in which case the program actually reads from the backup file. With the **−out** option, it will write to the designated file, concatenating the results of reading from any number of input files, which may be specified by wild cards. With the **−odir** option, it reads any number of input files, creating a corresponding output file for each in the designated output directory, with the output file suffix optionally specified by the **−otyp** option.

TUTORIAL EXAMPLES

For a simple example, consider first the following command line:

```
gema 'Abram=Abraham;Sarai=Sarah' old.text new.text
```

This copies file "old.text" to "new.text", replacing each occurrence of the string "Abram" with the string "Abraham" and replacing "Sarai" with "Sarah". The first command line argument consists of two transformation rules separated by '**;**'. Each rule consists of a *template* followed by '**=**' and an *action*. Any text that matches a template is replaced by the result of the corresponding action. (Here the action is just a literal replacement string, but it is called an *action* instead of a *replacement* because it is more general than that.) The presence of the equal sign serves to distinguish a pattern argument from a file name. Note that pattern arguments will usually need to be enclosed in single quotes on Unix (as shown in the example), or double quotes on MS-DOS or Windows.

Patterns can also be defined in one or more files loaded with the **−f** option. In a pattern file, new lines separate rules, blank lines are ignored, and a '**!**' causes the rest of the line to be ignored as a comment. The following characters have special meaning in patterns:

: * ? # / < > \ ^ = $ @ { } ; ! Space NewLine

Usually when doing string replacement we need to be a little more careful. For example, suppose we want to replace the variable name "x" with the name "horizontal". We don't want to replace instances of "x" that appear as parts of words, only those that are a word by themselves. This can easily be done like this:

```
gema -t 'x=horizontal' ...
```

where the **−t** option (token mode) requires all identifiers in templates to match a complete identifier in the input data. If you don't want this behavior everywhere, it can instead be specified locally in the template like this:

  gema '\Ix\I=horizontal' ...
Here the notation "**\I**" denotes an identifier delimiter; it matches an empty string but only if one of the adjacent characters can not be part of an identifier. There are a number of other such operators listed in the reference section below. Another that is particularly useful is "**\W**", which says that any number of whitespace characters will be accepted and skipped over if they appear at that point. Thus a template like "**x\W+\Wy**" will match regardless of whether the input text contains optional spaces. The **−w** option may be used to ignore spaces everywhere (except within identifiers or numbers).

A template such as "**first down**" will not only match these two words separated by a space, it will also match if they are separated by multiple spaces or even by a newline. This is because the space character actually has special meaning, which is to match one or more whitespace characters. In the less likely event that you really want to match exactly one space character, you can use "\ " or "**\s**".

Templates can also have *arguments* -- i.e. portions which match variable text. There are in fact several different kinds of arguments supported. The first kind of argument is the *wild card* characters "*", which matches any number of characters (up to a limit that can be adjusted by the **−arglen** option), and "**?**", which matches any one character. For example, consider the C macro definition:
  #define frob(p,m) do(Frob,p[m])
A first approximation to doing this with **gema** is to use a rule like this: "**\Ifrob\W(*,*)=do(Frob,*[*])**" Here the asterisks in the template match any text up to the delimiting comma or parenthesis, and the same text is substituted where the asterisks appear in the action. Note though that in **gema**, the comma and parentheses do not have any special meaning; a template may delimit the arguments with whatever characters you want. There may be as many as 20 arguments, and the "*" or "**?**" in the action will be substituted in the same order as in the template. In cases where the arguments need to be used in some different order or an argument needs to be used more than once, a different notation may be used that designates the arguments by number. For example:
  Rule: **ADD * TO *.=$2 \:\= $2 + $1\;**
  Input: **ADD ITEM TO SUM.**
  Output: **SUM := SUM + ITEM;**
Note that some of the characters are escaped so that they will be treated as literals. A backslash preceding a special character always means that the character is to be taken literally, so it won't hurt to escape all special character literals if you aren't sure which ones have special meaning. (If there are more than nine arguments, the number needs to be enclosed in braces, like: "**${10}**")

One problem to watch out for with "*" arguments is that since **gema** allows patterns to span line boundaries, the argument may read many lines ahead, gobbling up much more text than intended. In cases where you want to match only on a single line, the operator "**\L**" can be used in a template to specify that the following arguments will not be allowed to match on a newline character. This mode may also be selected globally with the command line option **−line**.

When a template is to match an entire line, the first inclination might be to write a template such as: "**\nCommand *\n**" where "**\n**" designates the new line character. But there are two problems with this: it won't match on the first line of the file (because there isn't a preceding new line) and it won't match on lines where the preceding new line has already been read as part of the match for the previous line. Both of these are solved by using "**\N**" which matches if it is at the beginning or end of a line, but doesn't actually read the new line character. In an action, "**\N**" outputs a new line character if the output stream is not already at the beginning of a line.

Another way to further constrain argument matching is to use the next kind of argument, which is called a *predefined recognizer domain*. (We'll see user-defined domains later.) For example, in a template the string "**<D>**" designates an argument that will match one or more digits. Similarly, "**<L>**" matches one or more letters, "**<I>**" matches an identifier (letters, digits, and underscore), and "**<N>**" matches a number (digits with optional sign and decimal point). Other kinds are listed in the reference section below. With a lower case letter, the argument becomes optional. Thus, "**<d>**" matches zero or more digits. Preceding the letter with a minus sign inverts the test, so that, for example, "**<−D>**" matches one or more characters that are *not* digits. The letter may be followed by a number to limit how many characters are matched. For example, "**<D3>**" matches exactly three digits and "**<d3>**" will match from zero up to a maximum of three digits. A *universal* argument such as "**<U10>**" matches exactly 10 characters of any kind; this may be useful for splitting an input record into fixed-length fields. Note that "**<U1>**" is equivalent to "**?**". (For these arguments, the action must access the value using "**$**" instead of repeating the argument designator. That short cut is only supported for arguments designated by a single character.)

An even more general way of specifying the set of characters to be accepted is to use a *regular expression*. A regular expression argument is designated by slashes before and after. For example, the template "**c/[ad]+/r**" will match input text "cadar" with "**$1**" having the value "ada". Note that between the slashes, the special characters have the meaning defined by the usual regular expression syntax, not their meaning elsewhere in templates.

The last kind of argument is a *recursive argument*. To show the need for this, first consider this example that is trying to convert Lisp s-expressions to function call notation:

  Rule: **(* * *)=*(*,*)**
  Input: **(fn xyz 34)**
  Output: **fn(xyz,34)**

So far, so good, but look what happens here:

  Input: **(fn (g a b) z)**
  Output: **fn((g,a b)**

What is needed is a way to properly associate matching nested parentheses and to translate nested constructs. Both of these are done by simply modifying the rule like this:

  Rule: **(# # #)=#(#,#)**
  Input: **(fn (g a b) z)**
  Output: **fn(g(a,b),z)**

The "**#**" designates a *recursive argument*, which means that the argument value is the result of translating input text until reaching the terminator character(s) following it. The space following "g" does not terminate the outer-level argument because it is read as part of the inner-level template match. Similarly, the inner "(" is read by the inner match which also reads the matching inner ")".

Actions can also perform a wide variety of activities by using the large set of built-in functions that are available. A function call is designated by "@" followed by the name of the function, followed by any arguments enclosed in curly braces and separated by semicolons. The "functions" section below lists all of the functions that are available. For example, you can define a default rule like this: "**\N.*\N=@err{@file line @line\: Unrecognized\: $1}**" The **err** function writes its argument to the error output stream. The **file** and **line** functions (which have no arguments) return the name and line number of the input file.

gema supports defining multiple sets of rules, each of which is called a *domain.* By default, rules are defined in the default domain, which is also the one used to translate the input file. Alternate domains are created by prefixing a rule with a domain name followed by "**:**". The domain name applies until the end of the line. The name of the default domain is the empty string, so a rule line beginning with a colon is the same as one without any colon. Alternate domains are used for several purposes, including defining new kinds of arguments for use in templates, defining new functions for use in actions, and for translations that require different rules for various contexts in the input data.

To illustrate using an alternate domain as a new argument type, suppose we want a template argument that will match on either "yes" or "no", so that we can write a rule like:

  done\? <yesno>=Finished \= $1

All that is needed is to define the following:

  yesno:yes=yes@end;no=no@end;=@fail

This says that if you see "yes" or "no", output it to the argument value and then return, and if anything is seen that doesn't match either of those, then the argument match fails. Note that the last rule has an empty template; this matches as a last resort if no other template in the domain can be matched. Since it doesn't advance the input stream, this makes sense only if the action is to exit. Note that domain names should have at least two letters in order to not conflict with predefined recognizers.

Domain names can also be used as functions of one argument, which means that the function returns the result of translating the argument value with the patterns of the domain. This is typically used in a two-step translation process where the first pattern match is used to split the input into fields, and then other domains are used to process individual fields. Remember that the default domain has an empty name, so "@{*arg*}" means to translate the argument with the default domain.

For an example of the use of alternate domains for different contexts, suppose that we are doing name substitution in C source code and we don't want to make any changes inside of character strings. We could add a rule "**"*"="*"**" to match on string constants and pass them through. However, string constants can contain "**\"**" and we don't want the argument to be terminated by that quote. To handle this, we can use a separate domain for processing the contents of a string. Then the rule becomes "**"<sbody>"="$1"**" and we add a rule: "**sbody:\\"=\\"**"

OPTIONS

−f *pattern_file*
: Reads pattern definitions from a file.

−p *patterns*
: Patterns defined directly as a command line argument. The **−p** can usually be omitted since an argument containing "**=**" or beginning with "**@**" is automatically recognized as a pattern argument. Any number of **−f** and/or **−p** options may be given.

−in *file*
: Explicitly specifies the input file pathname. If the file name is "**-**" then standard input is used. Usually the **−in** is not necessary since the first file name on the command line is understood to be the input file.

−out *file*
: Specifies the pathname of the output file. If the name is "**-**" then standard output is used. After an explicit **−out** option has been used, the remainder of the command line can have any number of input file names (without **−in**) which will be read in sequence, with the concatenated result going to the single output file. For example, a command such as the following can be used to do a **grep**-like search of a group of files:

```
gema -match -p 'Copyright *\n=@file\: $0' -out - *.c
```

(The special notation "**$0**" copies all of the matched text into the output.)

−odir *directory*
: Specifies the output directory. For each input file that follows, a corresponding output file will be written in the designated directory.

−otyp *suffix*
: When used with the **−odir** option, this specifies that each output file will have the designated suffix replacing the suffix of the input file. For example, given the command line:

```
gema −f patterns −odir /stuff −otyp .list *.text
```

then if the current directory contains a file named "**foo.text**", it will be translated to an output file named "**/stuff/foo.list**".

−backup *suffix*
: The argument will be used as the backup file suffix in place of the default "**.bak**".

−nobackup
: Output files will be overwritten instead of saving the old file as a backup file.

−line
: Places the program in line mode, which means that all pattern matching is limited to single lines. Arguments and template operators will never cross a line boundary except where the template explicitly specifies so with "**\n**".

−b
: Binary. With this option, all input and output files are opened in binary mode instead of text mode. This makes no difference on most Unix systems, but on Windows it changes the meaning of the new line character and doesn't treat Control-Z as the end of the file.

−k
: Keep going. With this option, the program will try to continue execution after certain errors that would normally cause it to abort. This may be useful when you want to see everything that is wrong before starting to fix the errors. Errors will still cause a non-zero exit status despite this option.

–match
: Matches only mode. Input text that doesn't match any template will be discarded instead of being copied to the output. This would be used when you want to extract selected information (like with **grep**) instead of doing a translation. This option applies only to the default domain. Another way to discard otherwise unmatched text is with the default rule "**?=**" while the rule "**?=?**" can be used to explicitly copy.

–i
: Case insensitive mode. All letters in templates will be matched without regard to distinctions of upper case or lower case. This also makes the names of domains and built-in functions case insensitive.

–w
: Whitespace insensitive mode. Space and tab characters in rules will be ignored except where they separate identifiers. Template matching will ignore whitespace characters in the input data as though templates had an implicit "**\W**" everywhere except within identifiers. Templates can use "**\J**" to indicate where space is *not* allowed.

–t
: Token mode. All identifiers appearing in templates will match only against a complete identifier, as though each identifier was implicitly surrounded by "**\I**" except where counter-acted by "**\J**".

–idchars *charset*
: Identifier characters. The argument value specifies the set of characters that will be considered to be identifier constituents, in addition to letters and digits. The default value is "_". This affects the behavior of "**\I**", "**<I>**", and "**<Y>**", and the **–w** and **–t** options. For example, if you were processing COBOL source code, you would need "**–idchars '-'**". For Lisp code, you would probably want something like: "**–idchars '-+=*/_<>'**"

–filechars *charset*
: File name characters. The argument value specifies the set of characters which are accepted by "**<F>**" as being file name constituents, in addition to letters and digits. The default value is "**./–_˜#@%+=**" for Unix. On Windows, colon and backslash are also included in the default set.

–literal *charset*
: This option specifies that each of the characters in the argument value will be treated as an ordinary literal character in patterns, instead of whatever special meaning it might normally have. For example, rather than saying something like:
  gema '\/usr\/foo\/<F>=\/usr\/bar\/$1' ...
  you could instead say:
  gema -literal / '/usr/foo/<F>=/usr/bar/$1' ...

–ml
: For convenience in processing Markup Languages (HTML, XML, etc.), this option (which is new in version 1.4) changes the syntax so that the characters "**<**", "**>**" and "**/**" are taken as literals, using "**[**", "**]**" and "**|**" respectively in their places. For example:
  gema -ml -p '<i>[T]</i>=<em>$1</em>' ...
  has the same effect as:
  gema -p '\<i\><T>\<\/i\>=\<em\>$1\<\/em\>' ...
  The "**-ml**" option is an abbreviation for "**@set-syntax{</>LLL;[|]</>}**".

–arglen *number*
: Specify the maximum length of a "*" argument. The default is 4096.

–prim *pattern_file*
: Primitive mode (for advanced users only). Like the **–f** option, this loads patterns from a file. It also suppresses loading of the built-in patterns for command line processing. This option is meaningful only when it appears as the first argument, and then it becomes the only argument that has any predefined meaning. The designated pattern file must define **ARGV** domain rules sufficient to specify what to do with the remainder of the command line.

–help
: Display brief usage message on the standard error output.

−version
Display program version identification on the standard error output.

−trace If the program was compiled with **−DTRACE,** then this option can be used to enable a report of template matches and failures to be written to the error output. This may be helpful for diagnosing obscure cases of unexpected results, but this is a crude experimental feature, so don't expect too much. Each line describes one event, possibly showing the line and column number at the beginning of the current template, the line and column of the current position, and a description of the event, indented for recursive arguments.

PATTERNS REFERENCE SUMMARY

The following characters have special meaning:

*	matches any number of characters
?	matches any one character
#	argument recursively translated in the current domain
=	end of template, beginning of action
$0	copies the template into the action to show all matched text
$*digit*	inserts argument value
$*letter*	inserts value of a variable with single-letter name
${*name***}**	value of named variable (only in action)
${*name***;***default***}**	variable with default value if not defined (action only)
\	escape character; see the section on "escape sequences" below.
^*x*	combine control key with the following character
Space	matches one or more whitespace characters (same as "**\S**").
NewLine	end of action
;	end of action, or separator between function arguments
@*name***{***args***}**	invoke built-in function or user-defined translation domain (action only)
@*special-character*	has the default meaning of the special character by itself, as documented here; this can be used to access the original functionality of a character that has been changed by the **−literal** option or **@set-syntax** function.
:	separates domain name from rule
<*name***>**	recursive argument, translated according to the named domain, or pre-defined recognizer argument. (template only)
/*regexp***/**	regular expression argument (template only)
!	the rest of the line is a comment

Also, as a special case, the first line of a pattern file is ignored if the first two characters are "**#!**". This allows a pattern file to be made directly executable on Unix by putting something like "**#!/usr/local/bin/gema -f**" as the first line.

ESCAPE SEQUENCES

The backslash character denotes special handling for the character that follows it. When followed by a lower-case letter or a digit, it represents a particular control character. When followed by an upper case letter, it is a pattern match operator. A backslash at the end of a line designates continuation by causing the newline to be ignored along with any leading white space on the following line. Before any other character, the backslash quotes the character so that it simply represents itself. In particular, a literal backslash is represented by two backslashes.

Following are the defined escape sequences:

\a	Alert (a.k.a. bell) character
\b	Backspace character
\c*x*	Control key combined with the following character
\d	Delete character
\e	Escape character (i.e. ESC, not backslash)
\f	Form feed character
\n	New line character
\r	carriage Return character
\s	Space character
\t	horizontal Tab character
\v	Vertical tab character
\x*xx*	character specified by its heXadecimal code
\digits	character specified by its octal code
\A	matches beginning of input data
\B	matches Beginning of file
\C	Case-insensitive comparison for the rest of the template
\E	matches End of file
\G	Goal -- complete preceding argument before considering rest of template
\I	Identifier separator
\J	Join -- locally counteracts the **−w** and/or **−t** option by saying that spaces in the input will not be ignored at this position, and an identifier delimiter is not required here. If neither of these options is being used, then it has no effect.
\L	Line mode -- following arguments can't cross line boundary
\N	New line; matches beginning or end of line
\P	Position -- leave input stream here after the template matches
\S	Space -- matches one or more whitespace characters
\W	Whitespace -- skips over any optional whitespace characters
\X	word separator
\Z	matches end of input data

RECOGNIZERS

The following argument designators, consisting of a single letter between angle brackets, can be used in templates to match on various kinds of characters. Preceding the letter with "**–**" inverts the test. The argument requires at least one matching character if the letter is uppercase, or is optional if the letter is lowercase. The letter may be followed by a number to match on that many characters, or up to that maximum for an optional argument. If the number is **0**, the argument matches if the next character is of the indicated kind, but the input stream is not advanced past it; in other words, this acts as a one-character look-ahead.

<A>	Alphanumeric (letters and digits)
<C>	Control characters
<D>	Digits
<F>	File pathname
<G>	Graphic characters, i.e. any non-space printable character
<I>	Identifier
<J>	lower case letters
<K>	upper case letters
<L>	Letters (either upper or lower case)
<N>	Number, i.e. digits with optional sign and decimal point
<O>	Octal digits
<P>	Printing characters, including space
<S>	white Space characters (space, tab, newline, FF, VT)
<T>	Text characters, including all printing characters and white space
<U>	Universal (matches anything except end-of-file)
<W>	Word (letters, apostrophe, and hyphen)
<X>	hexadecimal digits
<Y>	punctuation (graphic characters that are not identifiers)

FUNCTIONS

The following built-in functions may be used in actions, either in the action portion of a rule, or appearing by itself as an immediate action. When a line in a pattern file begins with "@", the actions are executed before reading the next line.

Function calls have the form "@*name*{*args*}", with arguments separated by "**;**". For functions without arguments, the argument delimiters "**{}**" may be omitted if not needed to separate the name from the following character. All arguments are evaluated, so all of the special characters available in actions apply within the arguments also. (In a few cases, arguments that are not used are skipped instead of evaluated, but arguments are never used literally.) Arguments shown as *number* or *length* must have a value which is a valid decimal representation of an integer, with optional leading whitespace and optional sign. All numbers are considered to be 32 bit signed integers. The descriptions given here for the functions is just a terse reference summary; refer to the user manual for more detailed information.

@abort{}
Immediately terminate execution.

@add{*number***;***number***}**
Return the sum of the two numbers.

@and{*number***;***number***}**
Return the bit-wise *and* of the two numbers.

@append{_var_**;**_string_**}**
Append the string to the end of the named variable's value. No return value.

@bind{_var_**;**_string_**}**
Bind named variable to a value. No return value.

@center{_length_**;**_string_**}**
Center the string within a field of the designated length.

@char-int{_character_**}**
Returns decimal number representation of internal character code.

@close{ _pathname_**}**
Closes a file previously opened by **@write{** _pathname_**}**

@cmpi{_string_**;**_string_**;**_less-value_**;**_equal-value_**;**_greater-value_**}**
Compare, case-insensitive. Return either the third, fourth, or fifth argument value depending on whether the first argument is less than, equal to, or greater than the second. The two unused arguments are not evaluated.

@cmpn{_number_**;**_number_**;**_less-value_**;**_equal-value_**;**_greater-value_**}**
Compare numbers.

@cmps{_string_**;**_string_**;**_less-value_**;**_equal-value_**;**_greater-value_**}**
Compare, case-sensitive.

@column{}
Returns the current column number in the input stream.

@date{}
Returns the current date, in the form _mm/dd/yyyy_

@datime{}
Returns the current date and time, formatted by the C function **ctime**(3).

@decr{_var_**}**
Decrement value of variable. No return value.

@define{ _patterns_**}**
Run-time definition of additional rules. No return value.

@div{_number_**;**_number_**}**
Return result of dividing the first argument by the second.

@downcase{_string_**}**
Convert any letters from upper case to lower case.

@end{}
End translation. No return value.

@err{_string_**}**
Write the argument value to the error output stream. No return value.

@exit-status{_number_**}**
Specify exit code to return when the program terminates. No return value.

@expand-wild{ _path_**}**
Expand wild card pathname on MS-DOS or Windows.

@fail{}
Signal translation failure; causes failed match of recursive argument.

@file{} Returns the name of the input file.

@file-time{}
Returns the modification time and date of the input file, formatted by the C function **ctime**(3).

@fill-center{*background***;***value***}**
: Center the value on top of the background string.

@fill-left{*background***;***value***}**
: Left-justify the value on top of the background string.

@fill-right{*background***;***value***}**
: Right-justify the value on top of the background string.

@getenv{*name***;***default***}**
: Return the value of an environment variable. Returns the optional second argument if the environment variable is not defined.

@get-switch{*name***}**
: Return value of switch (see **@set-switch**)

@incr{*var***}**
: Increments the value of a variable. No value returned.

@inpath{}
: Returns the pathname of the input file.

@int-char{*number***}**
: Returns the character whose internal code is given by the argument.

@line{}
: Returns the current line number in the input file.

@left{*length***;***string***}**
: Left-justify the string, padding with spaces to the designated length.

@length{*string***}**
: Returns the length of the argument.

@makepath{*directory***;***name***;***suffix***}**
: Returns the file pathname formed by merging the second argument with the default directory in the first argument and replacing the suffix from the third argument, if not empty.

@mergepath{*pathname***;***name***;***suffix***}**
: Returns the file pathname formed by merging the second argument with a default directory extracted from the first argument and replacing the suffix from the third argument, if not empty.

@mul{*number***;***number***}**
: Returns the result of multiplying the two numbers.

@mod{*number***;***number***}**
: Returns the first argument modulo the second.

@not{*number***}**
: Returns the bit-wise inverse of the argument.

@or{*number***;***number***}**
: Returns the bit-wise *or* of the two numbers.

@out{*string***}**
: Writes the argument value directly to the current output file. No return value.

@out-column{}
: Returns the current column number in the output file.

@outpath{}
: Returns the pathname of the output file.

@push{*var***;***value***}**
: Set the value of a variable while remembering the previous value. Same as **@bind**.

@**pop{**_var_**}**
: Restore the variable to the value it had before the most recent @**push**. Same as @**unbind**.

@**probe{**_pathname_**}**
: Return "F" if the argument names a file, "D" if a directory, "V" if a device, or "U" if undefined.

@**quote{**_string_**}**
: Returns a copy of the argument with backslashes inserted where necessary so that @**define** will treat all of the characters as literals.

@**radix{**_from_**;**_to_**;**_value_**}**
: Radix conversion. The first two arguments must be decimal integers. The third argument is interpreted as a number whose base is specified by the first argument. The result value is that number represented in the base specified by the second argument.

@**read{**_pathname_**}**
: Return an input stream that reads the contents of the named file. Note that this just specifies where the input comes from; it is usually used as an argument to another function that specifies what to do with the data.

@**relative-path{**_pathname_**;**_pathname_**}**
: If the two pathnames have the same directory portion, return the second argument with the common directory removed; else return the whole second argument.

@**repeat{**_number_**;**_action_**}**
: The second argument is executed the number of times specified by the first argument. If the number is less than or equal to zero, the second argument is not evaluated at all.

@**reset-syntax{}**
: Re-initializes the syntax tables to undo the effects of @**set-syntax** or the **-literal** option.

@**reverse{**_string_**}**
: Return the characters of the argument in reverse order.

@**right{**_length_**;**_string_**}**
: Right-justify the string, padding with spaces to the designated length.

@**set{**_var_**;**_value_**}**
: Set the named variable to the designated value. No return value.

@**set-locale{**_name_**}**
: Set internationalization locale, using **setlocale**(3). This may affect which characters are considered to be letters, and the format of times and dates. No result value.

@**set-parm{**_name_**;**_value_**}**
: Set a string-valued option, either "idchars", "filechars", or "backup". No result value.

@**set-switch{**_name_**;**_value_**}**
: Set one of the following options to 1 for true or 0 for false: "line" for line mode, "b" for binary mode, "k" to keep going after errors, "match" for match-only mode, "i" for case-insensitive mode, "w" for whitespace insensitive mode, "t" for token mode, or "trace".

@**set-syntax{**_type_**;**_charset_**}**
: The characters in the second argument will have the same meaning as the corresponding special character(s) in the first argument, or use one of the alphabetic type codes: "L" for literal, "I" for ignore, etc. No result value.

@**set-wrap{**_number_**;**_string_**}**
: For @**wrap**, the first argument is the number of columns, and the second argument is the indentation string. No result value.

@**shell{**_string_**}**
: The argument is executed as a shell command. No return value.

@show-help{}
Display usage message on the standard error stream.

@sub{*number***;***number***}**
Subtract.

@subst{*patterns***;***operand***}**
Substitution. Return the result of translating the operand according to the patterns temporarily defined by the first argument.

@substring{*skip***;***length***;***string***}**
Return substring of the third argument by skipping the number of characters indicated by the first argument and then taking the number of characters indicated by the second argument.

@tab{*number***}**
Output spaces until the output stream reaches the designated column.

@terminate{}
End translation of a recursive argument, with success if any characters have been accepted, or failure if the argument value is empty.

@time{}
Return the current time, in the form *hh*:*mm*:*ss*

@unbind{*var***}**
Restore the variable to the value, if any, it had before the most recent **@bind**.

@undefine{*patterns***}**
Delete pattern definitions.

@upcase{*string***}**
Convert any letters from lower case to upper case.

@var{*var***;***default***}**
Return the value of the named variable. If the variable is not defined, return the optional second argument, if supplied, else report an error. **@var** has the same effect as **$** when the name is not a number.

@version{}
Return the program version identification string.

@wrap{*string***}**
Output the string, after starting a new line if necessary according to the parameters set by **@set-wrap**. The default is 80 columns and no indentation.

@write{*pathname***;***string***}**
The second argument is evaluated with its result value being written to the file named by the first argument. Subsequent calls to **@write** with the same pathname will append to the file, until a **@close**.

BUGS

The implementation of the **−t** and **−w** options is a little sloppy and may produce unexpected results requiring use of explicit **\J**, **\I**, or **\W** to work around in certain contexts.

SEE ALSO

The **gema** user manual and the example pattern files provided.

The source files and documentation can be downloaded from "**http://sourceforge.net/projects/gema/**" or "**http://www.cbayona.com/pub/Macro%20Languages/Gema/**". The documentation can be read at "**http://gema.sourceforge.net/**".

AUTHORS

gema was written by David N. Gray <DGray@acm.org>.