GPP 2.27 — Generic Preprocessor

Denis Auroux, Tristan Miller

1 DESCRIPTION

GPP is a general-purpose preprocessor with customizable syntax, suitable for a wide range of preprocessing tasks. Its independence from any programming language makes it much more versatile than cpp, while its syntax is lighter and more flexible than that of m4.

GPP is targeted at all common preprocessing tasks where cpp is not suitable and where no very sophisticated features are needed. In order to be able to process equally efficiently text files or source code in a variety of languages, the syntax used by GPP is fully customizable. The handling of comments and strings is especially advanced.

Initially, GPP only understands a minimal set of built-in macros, called *meta-macros*. These meta-macros allow the definition of *user macros* as well as some basic operations forming the core of the preprocessing system, including conditional tests, arithmetic evaluation, wildcard matching (globbing), and syntax specification. All user macro definitions are global—*i.e.*, they remain valid until explicitly removed; meta-macros cannot be redefined. With each user macro definition GPP keeps track of the corresponding syntax specification so that a macro can be safely invoked regardless of any subsequent change in operating mode.

In addition to macros, GPP understands comments and strings, whose syntax and behavior can be widely customized to fit any particular purpose. Internally comments and strings are the same construction, so everything that applies to comments applies to strings as well.

2 SYNTAX

```
gpp [-{}{o|O} outfile] [-{}I/include/path ...]
    [-{}Dname=val ...] [-{}z|+z] [-{}x] [-{}m]
    [-{}C|-{}T|-{}H|-{}X|-{}P|-{}U ... [-{}M ...]]
    [-{}n|+n] [+c<n> str1 str2] [+s<n> str1 str2 c]
    [-{}c str1] [-{}-{}nostdinc] [-{}-{}nocurinc]
    [-{}-{}curdirinclast] [-{}-{}warninglevel n]
    [-{}-{}includemarker str] [-{}-{}include file]
    [infile]

gpp -{}-{}help
```

3 OPTIONS

GPP recognizes the following command-line switches and options. Note that the -nostdinc, -nocurinc, -curdirinclast, -warninglevel, and -includemarker options from version 2.1 and earlier are deprecated and should not be used. Use the "long option" variants instead (--nostdinc, *etc.*).

- **-h --help**
  Print a short help message.
- **--version**
  Print version information.
- **-o** outfile
  Specify a file to which all output should be sent (by default, everything is sent to standard output).
- **-O** outfile
  Specify a file to which all output should be sent; output is simultaneously sent to stdout.
- **-I** /include/path
  Specify a path where the *#include* meta-macro will look for include files if they are not present in the current directory. The default is /usr/include if no -I option is specified. Multiple -I options may be specified to look in several directories.

- **-D** name=val
  Define the user macro *name* as equal to *val*. This is strictly equivalent to using the *#define* meta-macro, but makes it possible to define macros from the command-line. If *val* makes references to arguments or other macros, it should conform to the syntax of the mode specified on the command-line. Starting with version 2.1, macro argument naming is allowed on the command-line. The syntax is as follows: -D*macro*(*arg1*,. . .)=*definition*. The arguments are specified in C-style syntax, without any whitespace, but the definition should still conform to the syntax of the mode specified on the command-line.

- **+z**
  Set text mode to Unix mode (LF terminator). Any CR character in the input is systematically discarded. This is the default under Unix systems.

- **-z**
  Set text mode to DOS mode (CR–LF terminator). In this mode all CR characters are removed from the input, and all output LF characters are converted to CR–LF. This is the default if GPP is compiled with the WIN_NT option.

- **-x**
  Enable the use of the *#exec* meta-macro. Since *#exec* includes the output of an arbitrary shell command line, it may cause a potential security threat, and is thus disabled unless this option is specified.

- **-m**
  Enable automatic mode switching to the cpp compatibility mode if the name of an included file ends in '.h' or '.c'. This makes it possible to include C header files with only minor modifications.

- **-n**
  Prevent newline or whitespace characters from being removed from the input when they occur as the end of a macro call or of a comment. By default, when a newline or whitespace character forms the end of a macro or a comment it is parsed as part of the macro call or comment and therefore removed from output. Use the -n option to keep the last character in the input stream if it was whites-

pace or a newline. This is activated in cpp and Prolog modes.

- **+n**
  The opposite of -n. This is the default in all modes except cpp and Prolog. Note that +n must be placed *after* -C or -P in order to have any effect.

- **-U** arg1 . . . arg9
  User-defined mode. The nine following command-line arguments are taken to be respectively the macro start sequence, the macro end sequence for a call without arguments, the argument start sequence, the argument separator, the argument end sequence, the list of characters to stack for argument balancing, the list of characters to unstack, the string to be used for referring to an argument by number, and finally the quote character (if there is none an empty string should be provided). These settings apply both to user macros and to meta-macros, unless the -M option is used to define other settings for meta-macros. See the section on syntax specification for more details.

- **-M** arg1 . . . arg7
  User-defined mode specifications for meta-macros. This option can only be used together with -U. The seven following command-line arguments are taken to be respectively the macro start sequence, the macro end sequence for a call without arguments, the argument start sequence, the argument separator, the argument end sequence, the list of characters to stack for argument balancing, and the list of characters to unstack. See below for more details.

- **(default mode)**
  The default mode is a vaguely cpp-like mode, but it does not handle comments, and presents various incompatibilities with cpp. Typical meta-macros and user macros look like this:

  ```
  #define x y
  macro(arg,...)
  ```

  This mode is equivalent to

  ```
  -U "" "" "(" "," ")" "(" ")" "#" "\\"
  -M "#" "\n" " " " " " "\n" "(" ")"
  ```

- **-C**
  cpp compatibility mode. This is the mode where GPP's behavior is the closest to that of cpp. Unlike in the default mode, meta-macro expansion occurs only at the beginning of lines, and C comments and strings are understood. This mode is equivalent to

  ```
  -n -U "" "" "(" "," ")" "(" ")" "#" ""
  -M "\n#\w" "\n" " " " " " "\n" "" ""
  +c "/*" "*/" +c "//" "\n" +c "\\\n" ""
  +s "\"" "\"" "\\" +s "'" "'" "\\"
  ```

- **-T**
  TEX-like mode. In this mode, typical meta-macros and user macros look like this:

  ```
  \define{x}{y}
  \macro{arg}{...}
  ```

  No comments are understood. This mode is equivalent to

  ```
  -U "\\" "" "{" "}{" "}" "{" "}" "#" "@"
  ```

- **-H**
  HTML-like mode. In this mode, typical meta-macros and user macros look like this:

  ```
  <#define x|y>
  <#macro arg|...>
  ```

  No comments are understood. This mode is equivalent to

  ```
  -U "<#" ">" "\B" "|" ">" "<" ">" "#" "\\"
  ```

- **-X**
  XHTML-like mode. In this mode, typical meta-macros and user macros look like this:

  ```
  <#define x|y/>
  <#macro arg|.../>
  ```

  No comments are understood. This mode is equivalent to

  ```
  -U "<#" "/>" "\B" "|" "/>" "<" ">" "#" "\\"
  ```

- **-P**
  Prolog-compatible cpp-like mode. This mode differs from the cpp compatibility mode by its handling of comments, and is equivalent to

```
-n -U "" "" "(" "," ")" "(" ")" "#" ""
-M "\n#\w" "\n" " " " " " "\n" "" ""
+ccss "\!o/*" "*/" +ccss "%" "\n" +ccii "\\\n" ""
+s "\"" "\"" "" +s "\!#'" "'" ""
```

- **+c** <n> str1 str2
  Specify comments. Any unquoted occurrence of *str1* will be interpreted as the beginning of a comment. All input up to the first following occurrence of *str2* will be discarded. This option may be used multiple times to specify different types of comment delimiters. The optional parameter <*n*> can be specified to alter the behavior of the comment and, *e.g.*, turn it into a string or make it ignored under certain circumstances, see below.

- **-c** str1
  Un-specify comments or strings. The comment/string specification whose start sequence is *str1* is removed. This is useful to alter the built-in comment specifications of a standard mode—*e.g.*, the cpp compatibility mode.

- **+s** <n> str1 str2 c
  Specify strings. Any unquoted occurrence of *str1* will be interpreted as the beginning of a string. All input up to the first following occurrence of *str2* will be output as is without any evaluation. The delimiters themselves are output. If *c* is non-empty, its first character is used as a *string-quote character—i.e.*, a character whose presence immediately before an occurrence of *str2* prevents it from terminating the string. The optional parameter <*n*> can be specified to alter the behavior of the string and, *e.g.*, turn it into a comment, enable macro evaluation inside the string, or make the string specification ignored under certain circumstances. See below.

- **-s** str1
  Un-specify comments or strings. Identical to -c.

- **--include** file

Process *file* before *infile*

- **--nostdinc**
  Do not look for include files in the standard directory /usr/include.
- **--nocurinc**
  Do not look for include files in the current directory.
- **--curdirinclast**
  Look for include files in the current directory *after* the directories specified by *-I* rather than before them.
- **--warninglevel** n
  Set warning level to *n* (0, 1 or 2). Default is 2 (most verbose).
- **--includemarker** str
  keep track of *#include* directives by inserting a marker in the output stream. The format of the marker is determined by *str*, which must contain three occurrences of the character *%* (or equivalently *?*). The first occurrence is replaced with the line number, the second with the file name, and the third with 1, 2 or blank. When this option is specified in default, cpp or Prolog mode, GPP does its best to ensure that line numbers are the same in the output as in the input by inserting blank lines in the place of definitions or comments.
- **infile**
  Specify an input file from which GPP reads its input. If no input file is specified, input is read from standard input.

4 SYNTAX SPECIFICATION

The syntax of a macro call is as follows: it must start with a sequence of characters matching the *macro start sequence* as specified in the current mode, followed immediately by the name of the macro, which must be a valid *identifier—i.e.*, a sequence of letters, digits, or underscores ("_"). The macro name must be followed by a *short macro end sequence* if the macro has no arguments, or by a sequence of arguments initiated by an *argument start sequence*. The various arguments are then separated by an *argument separator*, and the macro ends with a *long macro end sequence*.

In all cases, the parameters of the current context—*i.e.*, the arguments passed to the body being evaluated—can be referred to by using an *argument reference sequence* followed by a digit between 1 and 9. Alternatively, macro parameters may be named (see below). Furthermore, to avoid interference between the GPP syntax and the contents of the input file, a *quote character* is provided. The quote character can be used to prevent the interpretation of a macro call, comment, or string as anything but plain text. The quote character "protects" the following character, and always gets removed during evaluation. Two consecutive quote characters evaluate as a single quote character.

Finally, to facilitate proper argument delimitation, certain characters can be "stacked" when they occur in a macro argument, so that the argument separator or macro end sequence are not parsed if the argument body is not balanced. This allows nesting macro calls without using quotes. If an improperly balanced argument is needed, quote characters should be added in front of some stacked characters to make it balanced.

The macro construction sequences described above can be different for meta-macros and for user macros: this is the case in cpp mode, for example. Note that, since meta-macros can only have up to two arguments, the delimitation rules for the second argument are somewhat sloppier, and unquoted argument separator sequences are allowed in the second argument of a meta-macro.

Unless one of the standard operating modes is selected, the above syntax sequences can be specified either on the command-line, using the -M and -U options respectively for meta-macros and user macros, or inside an input file via the *#mode meta* and *#mode user* meta-macro calls. In both cases the mode description consists of nine parameters for user macro specifications, namely the macro start sequence, the short macro end sequence, the argument start sequence, the argument separator, the long macro end sequence, the string listing characters to stack, the string listing characters to unstack, the argument reference sequence, and finally the quote character. As explained below, these sequences should be supplied using the syntax of C strings; they must start with a non-alphanumeric character, and in the first five strings special matching sequences can be used (see below). If the argument corresponding to the quote character is the empty string, that argument's functional-

ity is disabled. For meta-macro specifications there are only seven parameters, as the argument reference sequence and quote character are shared with the user macro syntax.

The structure of a comment/string is as follows: it must start with a sequence of characters matching the given *comment/string start sequence*, and always ends at the first occurrence of the *comment/string end sequence*, unless it is preceded by an odd number of occurrences of the *string-quote character* (if such a character has been specified). In certain cases comment/strings can be specified to enable macro evaluation inside the comment/string; in that case, if a quote character has been defined for macros it can be used as well to prevent the comment/string from ending, with the difference that the macro quote character is always removed from output whereas the string-quote character is always output. Also note that under certain circumstances a comment/string specification can be *disabled*, in which case the comment/string start sequence is simply ignored. Finally, it is possible to specify a *string warning character* whose presence inside a comment/string will cause GPP to output a warning (this is useful to locate unterminated strings in cpp mode). Note that input files are not allowed to contain unterminated comments/strings.

A comment/string specification can be declared from within the input file using the *#mode comment* meta-macro call (or equivalently *#mode string*), in which case the number of C strings to be given as arguments to describe the comment/string can be anywhere between two and four: the first two arguments (mandatory) are the start sequence and the end sequence, and can make use of the special matching sequences (see below). They may not start with alphanumeric characters. The first character of the third argument, if there is one, is used as the string-quote character (use an empty string to disable the functionality), and the first character of the fourth argument, if there is one, is used as the string-warning character. A specification may also be given from the command-line, in which case there must be two arguments if using the +c option and three if using the +s option.

The behavior of a comment/string is specified by a three-character modifier string, which may be passed as an optional argument either to the +c/+s command-line options or to the *#mode comment*/*#mode string*

meta-macros. If no modifier string is specified, the default value is “ccc” for comments and “sss” for strings. The first character corresponds to the behavior inside meta-macro calls (including user-macro definitions since these come inside a *#define* meta-macro call), the second character corresponds to the behavior inside user-macro parameters, and the third character corresponds to the behavior outside of any macro call. Each of these characters can take the following values:

- **i**: disable the comment/string specification.
- **c**: comment (neither evaluated nor output).
- **s**: string (the string and its delimiter sequences are output as-is).
- **q**: quoted string (the string is output as-is, without the delimiter sequences).
- **C**: evaluated comment (macros are evaluated, but output is discarded).
- **S**: evaluated string (macros are evaluated, delimiters are output).
- **Q**: evaluated quoted string (macros are evaluated, delimiters are not output).

Important note: any occurrence of a comment/string start sequence inside another comment/string is always ignored, even if macro evaluation is enabled. In other words, comments/strings cannot be nested. In particular, the ‘Q’ modifier can be a convenient way of defining a syntax for temporarily disabling all comment and string specifications.

Syntax specification strings should always be provided as C strings, whether they are given as arguments to a *#mode* meta-macro call or on the command-line of a Unix shell. If command-line arguments are given via another method than a standard Unix shell, then the shell behavior must be emulated—*i.e.*, the surrounding "" quotes should be removed, all occurrences of ‘\\’ should be replaced by a single backslash, and similarly ‘\"’ should be replaced by ‘"’. Sequences like ‘\n’ are recognized by GPP and should be left as is.

Special sequences matching certain subsets of the character set can be used. They are of the form ‘*x*’, where *x* is one of:

- **b**: matches any sequence of one or more spaces or tab characters (‘\b’ is identical to ‘ ’).
- **w**: matches any sequence of zero or more spaces or tab characters.
- **B**: matches any sequence of one or more spaces, tabs or newline characters.
- **W**: matches any sequence of zero or more spaces, tabs or newline characters.
- **a**: an alphabetic character (‘a’ to ‘z’ and ‘A’ to ‘Z’).
- **A**: an alphabetic character, or a space, tab or newline.
- **#**: a digit (‘0’ to ‘9’).
- **i**: an identifier character. The set of matched characters is customizable using the *#mode charset id* command. The default setting matches alphanumeric characters and underscores (‘a’ to ‘z’, ‘A’ to ‘Z’, ‘0’ to ‘9’ and ‘_’).
- **t**: a tab character.
- **n**: a newline character.
- **o**: an operator character. The set of matched characters is customizable using the *#mode charset op* command. The default setting matches all characters in “+-*/\^<>=‘∼:.?@#&!%|”, except in Prolog mode where ‘!’, ‘%’ and ‘|’ are not matched.
- **O**: an operator character or a parenthesis character. The set of additional matched characters in comparison with ‘\o’ is customizable using the *#mode charset par* command. The default setting is to have the characters in “()[]{}” as parentheses.

Moreover, all of these matching subsets except ‘\w’ and ‘\W’ can be negated by inserting a ‘!’—*i.e.*, by writing ‘\!*x*’ instead of ‘*x*’.

Note an important distinctive feature of *start sequences*: when the first character of a macro or comment/string start sequence is ‘ ’ or one of the above special sequences, it is not taken to be part of the sequence itself but is used instead as a context check: for example a start sequence beginning with ‘\n’ matches only at the beginning of a line, but the match-

ing newline character is not taken to be part of the sequence. Similarly a start sequence beginning with ‘ ’ matches only if some whitespace is present, but the matching whitespace is not considered to be part of the start sequence and is therefore sent to output. If a context check is performed at the very beginning of a file (or more generally of any body to be evaluated), the result is the same as matching with a newline character (this makes it possible for a cpp-mode file to start with a meta-macro call).

Two special syntax rules were added in version 2.1. First, argument references (*#n*) are no longer evaluated when they are outside of macro calls and definitions. However, they are no longer allowed to appear (unless protected by quote characters) inside a call to a defined user macro; the current behavior (backwards compatible) is to remove them silently from the input if that happens.

Second, if the end sequence (either for macros or comments) consists of a single newline character, and if delimitation rules lead to evaluation in a context where the final newline character is absent, GPP silently ignores the missing newline instead of producing an error. The main consequence is that meta-macro calls can now be nested in a simple way in standard, cpp and Prolog modes.

5 EVALUATION RULES

Input is read sequentially and interpreted according to the rules of the current mode. All input text is first matched against the specified comment/string start sequences of the current mode (except those which are disabled by the ‘i’ modifier), unless the body being evaluated is the contents of a comment/string whose modifier enables macro evaluation. The most recently defined comment/string specifications are checked for first. Important note: comments may not appear between the name of a macro and its arguments (doing so results in undefined behavior).

Anything that is not a comment/string is then matched against a possible meta-macro call, and if that fails too, against a possible user-macro call. All remaining text undergoes substitution of argument reference sequences by the relevant argument text (empty unless the body being evaluated is the definition of a user macro) and removal of the quote

character if there is one.

Note that meta-macro arguments are passed to the meta-macro prior to any evaluation (although the meta-macro may choose to evaluate them, see meta-macro descriptions below). In the case of the *#mode* meta-macro, GPP temporarily adds a comment/string specification to enable recognition of C strings ("...") and prevent any evaluation inside them, so no interference of the characters being put in the C string arguments to *#mode* with the current syntax is to be feared.

On the other hand, the arguments to a user macro are systematically evaluated, and then passed as context parameters to the macro definition body, which gets evaluated with that environment. The only exception is when the macro definition is empty, in which case its arguments are not evaluated. Note that GPP temporarily switches back to the mode in which the macro was defined in order to evaluate it, so it is perfectly safe to change the operating mode between the time a macro is defined and the time when it is called. Conversely, if a user macro wishes to work with the current mode instead of the one that was used to define it it needs to start with a *#mode restore* call and end with a *#mode save* call.

A user macro may be defined with named arguments (see *#define* description below). In that case, when the macro definition is being evaluated, each named parameter causes a temporary virtual user-macro definition to be created; such a macro may be called only without arguments and simply returns the text of the corresponding argument.

Note that, since macros are evaluated when they are called rather than when they are defined, any attempt to call a recursive macro causes undefined behavior except in the very specific case when the macro uses *#undef* to erase itself after finitely many loop iterations.

Finally, a special case occurs when a user macro whose definition does not involve any arguments (neither named arguments nor the argument reference sequence) is called in a mode where the short user-macro end sequence is empty (*e.g.*, cpp or TEX mode). In that case it is assumed to be an *alias macro*: its arguments are first evaluated in the current mode as usual, but instead of being passed to the macro definition as parameters (which would cause them to be discarded) they are actually

appended to the macro definition, using the syntax rules of the mode in which the macro was defined, and the resulting text is evaluated again. It is therefore important to note that, in the case of a macro alias, the arguments actually get evaluated twice in two potentially different modes.

6 META-MACROS

These macros are always predefined. Their actual calling sequence depends on the current mode; here we use cpp-like notation.

- **#define** x y
  This defines the user macro *x* as *y*. *y* can be any valid GPP input, and may for example refer to other macros. *x* must be an identifier (*i.e.*, a sequence of alphanumeric characters and '_'), unless named arguments are specified. If *x* is already defined, the previous definition is overwritten. If no second argument is given, *x* will be defined as a macro that outputs nothing. Neither *x* nor *y* are evaluated; the macro definition is only evaluated when it is called, not when it is declared.

  It is also possible to name the arguments in a macro definition: in that case, the argument *x* should be a user-macro call whose arguments are all identifiers. These identifiers become available as user-macros inside the macro definition; these virtual macros must be called without arguments, and evaluate to the corresponding macro parameter.

- **#defeval** x y
  This acts in a similar way to *#define*, but the second argument *y* is evaluated immediately. Since user macro definitions are also evaluated each time they are called, this means that the macro *y* will undergo *two* successive evaluations. The usefulness of *#defeval* is considerable as it is the only way to evaluate something more than once, which may be needed to force evaluation of the arguments of a meta-macro that normally doesn't perform any evaluation. However since all argument references evaluated at define-time are understood as the arguments of the body in which the macro is being defined and not as the arguments of the macro itself, usually one has to use the quote character to prevent immediate evaluation of

argument references.

- **#undef** x
  This removes any existing definition of the user macro *x*.

- **#ifdef** x
  This begins a conditional block. Everything that follows is evaluated only if the identifier *x* is defined, and until either a *#else* or a *#endif* statement is reached. Note, however, that the commented text is still scanned thoroughly, so its syntax must be valid. It is in particular legal to have the *#else* or *#endif* statement ending the conditional block appear only as the result of a user-macro expansion and not explicitly in the input.

- **#ifndef** x
  This begins a conditional block. Everything that follows is evaluated only if the identifier *x* is not defined.

- **#ifeq** x y
  This begins a conditional block. Everything that follows is evaluated only if the results of the evaluations of *x* and *y* are identical as character strings. Any leading or trailing whitespace is ignored for the comparison. Note that in cpp-mode any unquoted whitespace character is understood as the end of the first argument, so it is necessary to be careful.

- **#ifneq** x y
  This begins a conditional block. Everything that follows is evaluated only if the results of the evaluations of *x* and *y* are not identical (even up to leading or trailing whitespace).

- **#else**
  This toggles the logical value of the current conditional block. What follows is evaluated if and only if the preceding input was commented out.

- **#endif**
  This ends a conditional block started by a *#if...* meta-macro.

- **#include** file
  This causes GPP to open the specified file and evaluate its contents, inserting the resulting text in the current output. All defined user

macros are still available in the included file, and reciprocally all macros defined in the included file will be available in everything that follows. The include file is looked for first in the current directory, and then, if not found, in one of the directories specified by the *-I* command-line option (or */usr/include* if no directory was specified). Note that, for compatibility reasons, it is possible to put the file name between "" or <>.

The order in which the various directories are searched for include files is affected by the *-nostdinc*, *-nocurinc* and *-curdirinclast* command-line options.

Upon including a file, GPP immediately saves a copy of the current operating mode onto the mode stack, and restores the operating mode at the end of the included file. The included file may override this behavior by starting with a *#mode restore* call and ending with a *#mode push* call. Additionally, when the *-m* command line option is specified, GPP will automatically switch to the cpp compatibility mode upon including a file whose name ends with either '.c' or '.h'.

- **#exec** command
  This causes GPP to execute the specified command line and include its standard output in the current output. Note that, for security reasons, this meta-macro is disabled unless the *-x* command line flag was specified. If use of *#exec* is not allowed, a warning message is printed and the output is left blank. Note that the specified command line is evaluated before being executed, thus allowing the use of macros in the command-line. However, the output of the command is included verbatim and not evaluated. If you need the output to be evaluated, you must use *#defeval* (see above) to cause a double evaluation.

- **#eval** expr
  The *#eval* meta-macro attempts to evaluate *expr* first by expanding macros (normal GPP evaluation) and then by performing arithmetic evaluation and/or wildcard matching. The syntax and operator precedence for arithmetic expressions are the same as in C; the only missing operators are <<, >>, ?:, and the assignment operators.

POSIX-style wildcard matching ('globbing') is available only on POSIX implementations and can be invoked with the =~ operator. In brief, a '?' matches any single character, a '*' matches any string (including the empty string), and '[...]' matches any one of the characters enclosed in brackets. A '[...]' class is complemented when the first character in the brackets is '!'. The characters in a '[...]' class can also be specified as a range using the '-' character—*e.g.*, '[F-N]' is equivalent to '[FGHIJKLMN]'.

If unable to assign a numerical value to the result, the returned text is simply the result of macro expansion without any arithmetic evaluation. The only exceptions to this rule are the comparison operators ==, !=, <, >, <=, and >= which, if one of the sides does not evaluate to a number, perform string comparison instead (ignoring trailing and leading spaces). Additionally, the *length(...)* arithmetic operator returns the length in characters of its evaluated argument.

Inside arithmetic expressions, the *defined(...)* special user macro is also available: it takes only one argument, which is not evaluated, and returns 1 if it is the name of a user macro and 0 otherwise.

- **#if** expr
  This meta-macro invokes the arithmetic/globbing evaluator in the same manner as *#eval* and compares the result of evaluation with the string "0" in order to begin a conditional block. In particular note that the logical value of *expr* is always true when it cannot be evaluated to a number.

- **#elif** expr
  This meta-macro can be used to avoid nested *#if* conditions. *#if ... #elif ... #endif* is equivalent to *#if ... #else #if ... #endif #endif*.

- **#mode** keyword ...
  This meta-macro controls GPP's operating mode. See below for a list of *#mode* commands.

- **#line**
  This meta-macro evaluates to the line number of the current input

file.

- **#file**
  This meta-macro evaluates to the filename of the current input file as it appears on the command line or in the argument to *#include*. If GPP is reading its input from stdin, then *#file* evaluates to ‘stdin’.

- **#date** fmt
  This meta-macro evaluates to the current date and time as formatted by the specified format string *fmt*. See the section *DATE AND TIME CONVERSION SPECIFIERS* below.

- **#error** msg
  This meta-macro causes an error message with the current filename and line number, and with the text *msg*, to be printed to the standard error device. Subsequent processing is then aborted.

- **#warning** msg
  This meta-macro causes a warning message with the current filename and line number, and with the text *msg*, to be printed to the standard error device. Subsequent processing is then resumed.

The key to GPP’s flexibility is the *#mode* meta-macro. Its first argument is always one of a list of available keywords (see below); its second argument is always a sequence of words separated by whitespace. Apart from possibly the first of them, each of these words is always a delimiter or syntax specifier, and should be provided as a C string delimited by double quotes (`" "`). The various special matching sequences listed in the section on syntax specification are available. Any *#mode* command is parsed in a mode where `"..."` is understood to be a C-style string, so it is safe to put any character inside these strings. Also note that the first argument of *#mode* (the keyword) is never evaluated, while the second argument is evaluated (except of course for the contents of C strings), so that the syntax specification may be obtained as the result of a macro evaluation.

The available *#mode* commands are:

- **#mode save / #mode push**
  Push the current mode specification onto the mode stack.

- **#mode restore / #mode pop**

Pop mode specification from the mode stack.

- **#mode standard** name
  Select one of the standard modes. The only argument must be one of: default (default mode); cpp, C (cpp mode); tex, TeX (TEX mode); html, HTML (html mode); xhtml, XHTML (xhtml mode); prolog, Prolog (prolog mode). The mode name must be given directly, not as a C string.

- **#mode user** “s1” . . . “s9”
  Specify user macro syntax. The 9 arguments, all of them C strings, are the mode specification for user macros (see the -U command-line option and the section on syntax specification). The meta-macro specification is not affected.

- **#mode meta** {user | “s1” . . . “s7”}
  Specify meta-macro syntax. Either the only argument is *user* (not as a string), and the user-macro mode specifications are copied into the meta-macro mode specifications, or there must be seven string arguments, whose significance is the same as for the -M command-line option (see section on syntax specification).

- **#mode quote** [“c”]
  With no argument or `""` as argument, removes the quote character specification and disables the quoting functionality. With one string argument, the first character of the string is taken to be the new quote character. The quote character can be neither alphanumeric nor ‘_’, nor can it be one of the special matching sequences.

- **#mode comment** [xxx] “start” “end” [“c” [“c”]]
  Add a comment specification. Optionally a first argument consisting of three characters not enclosed in `" "` can be used to specify a comment/string modifier (see the section on syntax specification). The default modifier is *ccc*. The first two string arguments are used as comment start and end sequences respectively. The third string argument is optional and can be used to specify a string-quote character. (If it is `""`, the functionality is disabled.) The fourth string argument is optional and can be used to specify a string delimitation warning character. (If it is `""`, the functionality is disabled.)

- **#mode string** [xxx] “start” “end” [“c” [“c”]]

Add a string specification. Identical to *#mode comment* except that the default modifier is *sss*.

- **#mode nocomment / #mode nostring** ["start"]
  With no argument, remove all comment/string specifications. With one string argument, delete the comment/string specification whose start sequence is the argument.

- **#mode preservelf** { on | off | 1 | 0 }
  Equivalent to the *-n* command-line switch. If the argument is *on* or *1*, any newline or whitespace character terminating a macro call or a comment/string is left in the input stream for further processing. If the argument is *off* or *0* this feature is disabled.

- **#mode charset** { id | op | par } "string"
  Specify the character sets to be used for matching the \o, \O and \i special sequences. The first argument must be one of *id* (the set matched by \i), *op* (the set matched by \o) or *par* (the set matched by \O in addition to the one matched by \o). *"string"* is a C string which lists all characters to put in the set. It may contain only the special matching sequences \a, \A, \b, \B, and \# (the other sequences and the negated sequences are not allowed). When a '-' is found inbetween two non-special characters this adds all characters inbetween (e.g. "A-Z" corresponds to all uppercase characters). To have '-' in the matched set, either put it in first or last position or place it next to a \x sequence.

7 DATE AND TIME CONVERSION SPECIFIERS

Ordinary characters placed in the format string are copied without conversion. Conversion specifiers are introduced by a '%' character, and are replaced as follows:

- **%a**
  The abbreviated weekday name according to the current locale.

- **%A**
  The full weekday name according to the current locale.

- **%b**
  The abbreviated month name according to the current locale.
- **%B**
  The full month name according to the current locale.
- **%c**
  The preferred date and time representation for the current locale.
- **%d**
  The day of the month as a decimal number (range 01 to 31).
- **%F**
  Equivalent to %Y-%m-%d (the ISO 8601 date format).
- **%H**
  The hour as a decimal number using a 24-hour clock (range 00 to 23).
- **%I**
  The hour as a decimal number using a 12-hour clock (range 01 to 12).
- **%j**
  The day of the year as a decimal number (range 001 to 366).
- **%m**
  The month as a decimal number (range 01 to 12).
- **%M**
  The minute as a decimal number (range 00 to 59).
- **%p**
  Either ‘AM’ or ‘PM’ according to the given time value, or the corresponding strings for the current locale. Noon is treated as ‘PM’ and midnight as ‘AM’.
- **%R**
  The time in 24-hour notation (%H:%M).
- **%S**
  The second as a decimal number (range 00 to 61).

- **%U**
  The week number of the current year as a decimal number, range 00 to 53, starting with the first Sunday as the first day of week 01.
- **%w**
  The day of the week as a decimal, range 0 to 6, Sunday being 0.
- **%W**
  The week number of the current year as a decimal number, range 00 to 53, starting with the first Monday as the first day of week 01.
- **%x**
  The preferred date representation for the current locale without the time.
- **%X**
  The preferred time representation for the current locale without the date.
- **%y**
  The year as a decimal number without a century (range 00 to 99).
- **%Y**
  The year as a decimal number including the century.
- **%Z**
  The time zone or name or abbreviation.
- **%%**
  A literal '%' character.

Depending on the C compiler and library used to compile GPP, there may be more conversion specifiers available. Consult your compiler's documentation for the *strftime()* function. Note, however, that any conversion specifiers not listed above may not be portable across installations of GPP.

8 EXAMPLES

Here is a basic self-explanatory example in standard or cpp mode:

```
#define FOO This is
#define BAR a message.
#define concat #1 #2
concat(FOO,BAR)
#ifeq (concat(foo,bar)) (foo bar)
This is output.
#else
This is not output.
#endif
```

Using argument naming, the *concat* macro could alternatively be defined as

```
#define concat(x,y) x y
```

In TEX mode and using argument naming, the same example becomes:

```
\define{FOO}{This is}
\define{BAR}{a message.}
\define{\concat{x}{y}}{\x \y}
\concat{\FOO}{\BAR}
\ifeq{\concat{foo}{bar}}{foo bar}
This is output.
\else
This is not output.
\endif
```

In HTML mode and without argument naming, one gets similarly:

```
<#define FOO|This is>
<#define BAR|a message.>
<#define concat|#1 #2>
<#concat <#FOO>|<#BAR>>
<#ifeq <#concat foo|bar>|foo bar>
This is output.
<#else>
This is not output.
<#endif>
```

The following example (in standard mode) illustrates the use of the quote character:

```
#define FOO This is \
   a multiline definition.
#define BLAH(x) My argument is x
BLAH(urf)
\BLAH(urf)
```

Note that the multiline definition is also valid in cpp and Prolog modes despite the absence of quote character, because ‘\’ followed by a newline is then interpreted as a comment and discarded.

In cpp mode, C strings and comments are understood as such, as illustrated by the following example:

```
#define BLAH foo
BLAH "BLAH" /* BLAH */
‘It\’s a /*string*/ !’
```

The main difference between Prolog mode and cpp mode is the handling of strings and comments: in Prolog, a ‘...’ string may not begin immediately after a digit, and a /*...*/ comment may not begin immediately after an operator character. Furthermore, comments are not removed from the output unless they occur in a #command.

The differences between cpp mode and default mode are deeper: in default mode #commands may start anywhere, while in cpp mode they must be at the beginning of a line; the default mode has no knowledge of comments and strings, but has a quote character (‘\’), while cpp mode has extensive comment/string specifications but no quote character. Moreover, the arguments to meta-macros need to be correctly parenthesized in default mode, while no such checking is performed in cpp mode.

This makes it easier to nest meta-macro calls in default mode than in cpp mode. For example, consider the following HTML mode input, which tests for the availability of the *#exec* command:

```
<#ifeq <#exec echo blah>|blah
> #exec allowed <#else> #exec not allowed <#endif>
```

There is no cpp mode equivalent, while in default mode it can be easily translated as

```
#ifeq (#exec echo blah
) (blah
)
\#exec allowed
#else
\#exec not allowed
#endif
```

In order to nest meta-macro calls in cpp mode it is necessary to modify the mode description, either by changing the meta-macro call syntax, or more elegantly by defining a silent string and using the fact that the context at the beginning of an evaluated string is a newline character:

```
#mode string QQQ "$" "$"
#ifeq $#exec echo blah
$ $blah
$
\#exec allowed
#else
\#exec not allowed
#endif
```

Note, however, that comments/strings cannot be nested ("..." inside $...$ would go undetected), so one needs to be careful about what to include inside such a silent evaluated string. In this example, the loose meta-macro nesting introduced in version 2.1 makes it possible to use the following simpler version:

```
#ifeq blah #exec echo -n blah
\#exec allowed
#else
\#exec not allowed
#endif
```

Remember that macros without arguments are actually understood to be aliases when they are called with arguments, as illustrated by the following example (default or cpp mode):

```
#define DUP(x) x x
#define FOO and I said: DUP
FOO(blah)
```

The usefulness of the *#defeval* meta-macro is shown by the following example in HTML mode:

```
<#define APPLY|<#defeval TEMP|<\##1 \#1>><#TEMP #2>>
<#define <#foo x>|<#x> and <#x>>
<#APPLY foo|BLAH>
```

The reason why *#defeval* is needed is that, since everything is evaluated in a single pass, the input that will result in the desired macro call needs to be generated by a first evaluation of the arguments passed to APPLY before being evaluated a second time.

To translate this example in default mode, one needs to resort to parenthesizing in order to nest the #defeval call inside the definition of APPLY, but need to do so without outputting the parentheses. The easiest solution is

```
#define BALANCE(x) x
#define APPLY(f,v) BALANCE(#defeval TEMP f
TEMP(v))
#define foo(x) x and x
APPLY(\foo,BLAH)
```

As explained above the simplest version in cpp mode relies on defining a silent evaluated string to play the role of the BALANCE macro.

The following example (default or cpp mode) demonstrates arithmetic evaluation:

```
#define x 4
The answer is:
#eval x*x + 2*(16-x) + 1998%x

#if defined(x)&&!(3*x+5>17)
This should be output.
#endif
```

To finish, here are some examples involving mode switching. The following example is self-explanatory (starting in default mode):

```
#mode push
#define f(x) x x
```

```
#mode standard tex
\f{blah}
\mode{string}{"$" "$"}
\mode{comment}{"/*" "*/"}
$\f{urf}$ /* blah */
\define{FOO}{bar/* and some more */}
\mode{pop}
f($FOO$)
```

A good example where a user-defined mode becomes useful is the GPP source of this document (available with GPP's source code distribution).

Another interesting application is selectively forcing evaluation of macros in C strings when in cpp mode. For example, consider the following input:

```
#define blah(x) "and he said: x"
blah(foo)
```

Obviously one would want the parameter *x* to be expanded inside the string. There are several ways around this problem:

```
#mode push
#mode nostring "\""
#define blah(x) "and he said: x"
#mode pop

#mode quote "‘"
#define blah(x) ‘"and he said: x‘"

#mode string QQQ "$$" "$$"
#define blah(x) $$"and he said: x"$$
```

The first method is very natural, but has the inconvenience of being lengthy and neutralizing string semantics, so that having an unevaluated instance of 'x' in the string, or an occurrence of '/*', would be impossible without resorting to further contortions.

The second method is slightly more efficient because the local presence of a quote character makes it easier to control what is evaluated and what isn't, but has the drawback that it is sometimes impossible to find

a reasonable quote character without having to either significantly alter the source file or enclose it inside a *#mode push/pop* construct. For example, any occurrence of '/*' in the string would have to be quoted.

The last method demonstrates the efficiency of evaluated strings in the context of selective evaluation: since comments/strings cannot be nested, any occurrence of '"' or '/*' inside the '$$' gets output as plain text, as expected inside a string, and only macro evaluation is enabled. Also note that there is much more freedom in the choice of a string delimiter than in the choice of a quote character.

Starting with version 2.1, meta-macro calls can be nested more efficiently in default, cpp and Prolog modes. This makes it easy to make a user version of a meta-macro, or to increment a counter:

```
#define myeval #eval #1

#define x 1
#defeval x #eval x+1
```

9 ADVANCED EXAMPLES

Here are some examples of advanced constructions using GPP. They tend to be pretty awkward and should be considered as evidence of GPP's limitations.

The first example is a recursive macro. The main problem is that (since GPP evaluates everything) a recursive macro must be very careful about the way in which recursion is terminated in order to avoid undefined behavior (most of the time GPP will simply crash). In particular, relying on a *#if/#else/#endif* construct to end recursion is not possible and results in an infinite loop, because GPP scans user macro calls even in the unevaluated branch of the conditional block. A safe way to proceed is for example as follows (we give the example in TEX mode):

```
\define{countdown}{
  \if{#1}
  #1...
  \define{loop}{\countdown}
  \else
```

```
  Done.
  \define{loop}{}
  \endif
  \loop{\eval{#1-1}}
}
\countdown{10}
```

Another example, in cpp mode:

```
#mode string QQQ "$" "$"
#define triangle(x,y) y \
 $#if length(y)<x$ $#define iter triangle$ $#else$ \
 $#define iter$ $#endif
$ iter(x,*y)
triangle(20)
```

The following is an (unfortunately very weak) attempt at implementing functional abstraction in GPP (in standard mode). Understanding this example and why it can't be made much simpler is an exercise left to the curious reader.

```
#mode string "‘" "‘" "\\"
#define ASIS(x) x
#define SILENT(x) ASIS()
#define EVAL(x,f,v) SILENT(
  #mode string QQQ "‘" "‘" "\\"
  #defeval TEMP0 x
  #defeval TEMP1 (
    \#define \TEMP2(TEMP0) f
  )
  TEMP1
  )TEMP2(v)
#define LAMBDA(x,f,v) SILENT(
  #ifneq (v) ()
  #define TEMP3(a,b,c) EVAL(a,b,c)
  #else
  #define TEMP3(a,b,c) \LAMBDA(a,b)
  #endif
  )TEMP3(x,f,v)
#define EVALAMBDA(x,y) SILENT(
```

```
  #defeval TEMP4 x
  #defeval TEMP5 y
  )
#define APPLY(f,v) SILENT(
  #defeval TEMP6 ASIS(\EVA)f
  TEMP6
  )EVAL(TEMP4,TEMP5,v)
```

This yields the following results:

```
LAMBDA(z,z+z)
  => LAMBDA(z,z+z)

LAMBDA(z,z+z,2)
  => 2+2

#define f LAMBDA(y,y*y)
f
  => LAMBDA(y,y*y)

APPLY(f,blah)
  => blah*blah

APPLY(LAMBDA(t,t t),(t t))
  => (t t) (t t)

LAMBDA(x,APPLY(f,(x+x)),urf)
  => (urf+urf)*(urf+urf)

APPLY(APPLY(LAMBDA(x,LAMBDA(y,x*y)),foo),bar)
  => foo*bar

#define test LAMBDA(y,‘#ifeq y urf
y is urf#else
y is not urf#endif
‘)
APPLY(test,urf)
  => urf is urf
```

```
APPLY(test,foo)
  => foo is not urf
```

10 AUTHOR

GPP was written by Denis Auroux <auroux@math.mit.edu>. Since version 2.12 it has been maintained by Tristan Miller <tristan@logological.org>.

11 COPYRIGHT