GNU m4

A macro processor

by Rene' Seindal

Edition 1.0.3.

last updated 15 November 1992,

for GNU `m4`, Version 1.0.3

This is Edition 1.0.3 of the *GNU* `m4` *Manual*,
last updated 15 November 1992,
for `m4` Version 1.0.3.

1 Introduction to m4

`m4` is a macro processor, in the sense that in copies its input to the output, expanding macros as it goes. Macros are either built-in or user-defined, and can take any number of arguments. Besides just doing macro expansion, `m4` has built-in functions for including named files, running Unix commands, doing integer arithmetic, manipulating text in various ways, recursion, etc. . .

`m4` can be used either as a front-end to a compiler, or as a macro processor in its own right.

GNU `m4` is mostly compatible with the System V, Release 3 version, except for some minor differences. See Chapter 17 [Compatibility], page 38, for more details.

2 Using this manual

This manual contains a number of examples of `m4` input and output, and a simple notation is used to distinguish input, output and error messages from `m4`. Examples are set out from the normal text, and shown in a fixed width font, like this

```
This is an example of an example!
```

To distinguish input from output, all output from `m4` is prefixed by the string '⇒', and all error messages by the string ' error '. Thus

```
Example of input line
⇒Output line from m4
error and an error message
```

As each of the predefined macros in `m4` is described, a prototype call of the macro will be shown, giving descriptive names to the arguments, e.g.,

```
regexp(string, regexp, opt replacement)
```

All macro arguments in `m4` are strings, but some are given special interpretation, e.g., as numbers, filenames, regular expressions, etc.

The '`opt`' before the third argument shows that this argument is optional—if it is left out, it is taken to be the empty string. An ellipsis ('`...`') last in the argument list indicates that any number of arguments may follow.

3 Problems and bugs

If you have problems with GNU `m4` or think you've found a bug, please report it. Before reporting a bug, make sure you've actually found a real bug. Carefully reread the documentation and see if it really says you can do what you're trying to do. If it's not clear whether you should be able to do something or not, report that too; it's a bug in the documentation!

Before reporting a bug or trying to fix it yourself, try to isolate it to the smallest possible input file that reproduces the problem. Then send us the input file and the exact results `m4` gave you. Also say what you expected to occur; this will help us decide whether the problem was really in the documentation.

Once you've got a precise problem, send e-mail to (Internet) `bug-gnu-utils@prep.ai.mit.edu` or (UUCP) `mit-eddie!prep.ai.mit.edu!bug-gnu-utils`. Please include the version number of `m4` you are using. You can get this information with the command '`m4 -V /dev/null`'.

Non-bug suggestions are always welcome as well. If you have questions about things that are unclear in the documentation or are just obscure features, please report them too.

4 Invoking m4

The format of the `m4` command is:

```
m4 [options] [macro-definitions] [input-files]
```

All options begin with '`-`', or if long option names are used, with a '`--`'. A long option name need not be written completely, and unambigous prefix is sufficient. `m4` understands the following options:

`-V`
`--version`
: Print the version number of the program. To see only the version number, use the command '`m4 -V /dev/null`'.

`-G`
`--no-gnu-extensions`
: Suppress all the extensions made in this implementation, compared to the System V version. For a list of these, see Chapter 17 [Compatibility], page 38.

`-d`*flags*
`--debug` *flags*
: Set the debug-level according to the flags *flags*. The debug-level controls the format and amount of information presented by the debugging functions. See Section 9.3 [Debug Levels], page 19, for more details on the format and meaning of *flags*.

`-l`*num*
`--arglength` *num*
: Restrict the size of the output generated by macro tracing. See Section 9.3 [Debug Levels], page 19, for more details.

`-o`*file*
`--erroroutput` *file*
: Redirect debug and trace output to the named file. Error messages are still printed on the standard error output. See Section 9.4 [Debug Output], page 20, for more details.

`-I`*dir*
`--include` *dir*
: Make `m4` search *dir* for included files that are not found in the current working directory. See Section 11.2 [Search Path], page 25, for more details.

`-e`
`--interactive`
: Makes this invocation of `m4` interactive. This means that all output will be unbuffered, and interrupts will be ignored.

`-s`
`--synclines`
: Generate synchronisation lines, for use by the C preprocessor or other similar tools. This is useful, for example, when `m4` is used as a front end to a compiler. Source file name and line number information is conveyed by lines of the form

'`#line` *linenum* `"`*filename*`"`', which are inserted as needed into the middle of the input (but always on complete lines per themselves). Such lines mean that the following line originated or was expanded from the contents of input file *filename* at line *linenum*. The '`"`*filename*`"`' part is often omitted when the file name did not change from the previous synchronisation line.

`-H`*n*
`--hashsize` *n*
: Make the internal hash table for symbol lookup be *n* entries big. The number should be prime. The default is 509 entries. It should not be necessary to increase this value, unless you define an excessive number of macros.

`-N`*n*
`--diversions` *n*
: Allow for up to *n* diversions to be used at the same time. The default is 10 diversions.

`-Q`
`--quiet`
`--silent`
: Suppress warnings about missing or superflous arguments in macro calls.

`-B`
`-S`
`-T`
: These options are present for compatibility with System V `m4`, but do nothing in this implementation.

Macro definitions and deletions can be made on the command line, by using the '`-D`' and '`-U`' options. They have the following format:

`-D`*name*
`-D`*name*`=`*value*
`--define` *name*
`--define` *name*`=`*value*
: This enters *name* into the symbol table, before any input files are read. If '`=`*value*' is missing, the value is taken to be the empty string. The *value* can be any string, and the macro can be defined to take arguments, just as if it was defined from within the input.

`-U`*name*
`--undefine` *name*
: This deletes any predefined meaning *name* might have. Obviously, only predefined macros can be deleted in this way.

`-t`*name*
`--trace` *name*
: This enters *name* into the symbol table, as undefined but traced. The macro will consequently be traced from the point it is defined.

The remaining arguments on the command line are taken to be input file names. If no names are present, the standard input is read. A file name of `-` is taken to mean the standard input.

The input files are read in the sequence given. The standard input can only be read once, so the filename `-` should only appear once on the command line.

5 Lexical and syntactic conventions

As `m4` reads its input, it separates it into *tokens*. A token is either a name, a quoted string, or any single character, that is not a part of either a name or a string. Input to `m4` can also contain comments.

5.1 Names

A name is any sequence of letters, digits, and the character `_` (underscore), where the first character is not a digit. If a name has a macro definition, it will be subject to macro expansion (see Chapter 6 [Macros], page 8, for more details).

Examples of legal names are: '`foo`', '`_tmp`', and '`name01`'.

5.2 Quoted strings

A quoted string is a sequence of characters surrounded by the quotes `‘` and `’`, where the number of start and end quotes within the string balances. The value of a string token is the text, with one level of quotes stripped off. Thus

```
‘’
```

is the empty string, and

```
‘‘quoted’’
```

is the string

```
‘quoted’
```

The quote characters can be changed at any time, using the built-in macro `changequote`. See Section 10.2 [Changequote], page 21, for more information.

5.3 Other tokens

Any character, that is neither a part of a name, nor of a quoted string, is a token by itself.

5.4 Comments

Comments in `m4` are normally delimited by the characters '`#`' and newline. All characters between the comment delimiters are ignored, but the entire comment (including the delimiters) is passed through to the output—comments are *not* discarded by `m4`.

Comments cannot be nested, so the first newline after a '`#`' ends the comment. The begin comment character can be included in the input by quoting it.

The comment delimiters can be changed to any string at any time, using the built-in macro `changecom`. See Section 10.3 [Changecom], page 22, for more information.

6 How to invoke macros

This chapter covers macro invocation, macro arguments and how macro expansion is treated.

6.1 Macro invocation

Macro invocations has one of the forms

```
name
```

which is a macro invocation without any arguments, or

```
name(arg1, arg2, ..., argn)
```

which is a macro invocation with *n* arguments. Macros can have any number of arguments. All arguments are strings, but different macros might interpret the arguments in different ways.

The opening parenthesis *must* follow the *name* directly, with no spaces in between. If it does not, the macro is called with no arguments at all.

For a macro call to have no arguments, the parentheses *must* be left out. The macro call

```
name()
```

is a macro call with one argument, which is the empty string, not a call with no arguments.

6.2 Macro arguments

When a name is seen, and it has a macro definition, it will be expanded as a macro.

If the name is followed by an opening parenthesis, the arguments will be collected before the macro is called. If too few arguments are supplied, the missing arguments are taken to be the empty string. If there are too many arguments, the excess arguments are ignored.

Normally `m4` will issue warnings if a built-in macro is called with an inappropriate number of arguments, but it can be suppressed with the '`-Q`' command line option. For user defined macros, there is no check of the number of arguments given.

Macros are expanded normally during argument collection, and whatever commas, quotes and parentheses that might show up in the resulting expanded text will serve to define the arguments as well. Thus, if *foo* expands to '`,b,c`', the macro call

```
bar(a foo,d)
```

is a macro call with four arguments, which are '`a `', '`b`', '`c`' and '`d`'.

6.3 Quoting macro arguments

Each argument has leading unquoted whitespace removed. Within each argument, all unquoted parentheses must match. For example, if *foo* is a macro,

```
foo(() (‘(’) ‘(’)
```

is a macro call, with one argument, whose value is '`() (() (`'.

It is common practice to quote all arguments to macros, unless you are sure you want the arguments expanded. Thus, in the above example with the parentheses, the 'right' way to do it is like this:

```
foo(‘() (() (’)
```

It is, however, in certain cases necessary to leave out quotes for some arguments, and there is nothing wrong in doing it. It just makes life a bit harder, if you are not careful.

6.4 Macro expansion

When the arguments, if any, to a macro call have been collected, the macro is expanded, and the expansion text is pushed back onto the input (unquoted), and reread. The expansion text from one macro call might therefore result in more macros being called, if the calls are included, completely or partially, in the first macro calls' expansion.

Taking a very simple example, if *foo* expands to '`bar`', and *bar* expands to '`Hello world`', the input

```
foo
```

will expand first to '`bar`', and when this is reread and expanded, into '`Hello world`'.

7 How to define new macros

Macros can be defined, redefined and deleted in several different ways. Also, it is possible to redefine a macro, without losing a previous value, which can be brought back at a later time.

7.1 Defining a macro

The normal way to define or redefine macros is to use the built-in `define`:

```
define(name, expansion)
```

which defines *name* to expand to *expansion*.

The expansion of `define` is void.

The following example defines the macro *foo* to expand to the text '`Hello World.`'.

```
define(‘foo’, ‘Hello world.’)
⇒
foo
⇒Hello world.
```

The empty line in the output is there because the newline is not a part of the macro definition, and it is consequently copied to the output. This can be avoided by use of the macro `dnl` (see Section 10.1 [Dnl], page 21, for details).

7.2 Arguments to macros

Macros can have arguments. The *n*th argument is denoted by `$n` in the expansion text, and is replaced by the *n*th actual argument, when the macro is expanded. Here is a example of a macro with two arguments. It simply exchanges the order of the two arguments.

```
define(‘exch’, ‘$2, $1’)
⇒
exch(arg1, arg2)
⇒arg2, arg1
```

This can be used, for example, if you like the arguments to `define` to be reversed.

```
define(‘exch’, ‘$2, $1’)
⇒
define(exch(‘‘expansion text’’, ‘‘macro’’))
⇒
macro
⇒expansion text
```

For an explanation of the double quotes, see Section 6.3 [Quoting Arguments], page 8.

GNU `m4` allows the number following the '`$`' to consist of one or more digits, allowing macros to have any number of arguments. This is not so in Unix implementations of `m4`, which only recognize one digit.

As a special case, the zero'th argument, `$0`, is always the name of the macro being expanded.

```
define(‘test’, ‘‘Macro name: $0’’)
⇒
```

```
test
⇒Macro name: test
```

If you want quoted text to appear as part of the expansion text, remember that quotes can be nested in quoted strings. Thus, in

```
define(‘foo’, ‘This is macro ‘foo’.’)
⇒
foo
⇒This is macro foo.
```

The ‘`foo`’ in the expansion text is *not* expanded, since it is a quoted string, and not a name.

7.3 Special arguments to macros

There is a special notation for the number of actual arguments supplied, and for all the actual arguments.

The number of actual arguments in a macro call is denoted by `$#` in the expansion text. Thus, a macro to display the number of arguments given can be

```
define(‘nargs’, ‘$#’)
⇒
nargs
⇒0
nargs()
⇒1
nargs(arg1, arg2, arg3)
⇒3
```

The notation `$*` can be used in the expansion text to denote all the actual arguments, unquoted, with commas in between. For example

```
define(‘echo’, ‘$*’)
⇒
echo(arg1,    arg2, arg3 , arg4)
⇒arg1,arg2,arg3 ,arg4
```

Often each argument should be quoted, and the notation `$@` handles that. It is just like `$*`, except that it quotes each argument. A simple example of that is:

```
define(‘echo’, ‘$@’)
⇒
echo(arg1,    arg2, arg3 , arg4)
⇒arg1,arg2,arg3 ,arg4
```

Where did the quotes go? Of course, they were eaten, when the expanded text were reread by `m4`. To show the difference, try

```
define(‘echo1’, ‘$*’)
⇒
define(‘echo2’, ‘$@’)
⇒
define(‘foo’, ‘This is macro ‘foo’.’)
⇒
echo1(foo)
```

```
⇒This is macro This is macro foo..
echo2(foo)
⇒This is macro foo.
```

If you don't understand this, see Section 9.2 [Trace], page 18.

A '`$`' sign in the expansion text, that is not followed by anything `m4` understands, is simply copied to the macro expansion, as any other text is.

```
define(‘foo’, ‘$$$ hello $$$’)
⇒
foo
⇒$$$ hello $$$
```

If you want a macro to expand to something like '`$12`', put a pair of quotes after the `$`. This will prevent `m4` from interpreting the `$` sign as a reference to an argument.

7.4 Deleting a macro

A macro definition can be removed with `undefine`:

```
undefine(name)
```

which removes the macro *name*. The macro name must necessarily be quoted, since it will be expanded otherwise.

The expansion of `undefine` is void.

```
foo
⇒foo
define(‘foo’, ‘expansion text’)
⇒
foo
⇒expansion text
undefine(‘foo’)
⇒
foo
⇒foo
```

It is not an error for *name* to have no macro definition. In that case, `undefine` does nothing.

7.5 Renaming macros

It is possible to rename an already defined macro. To do this, you need the built-in `defn`:

```
defn(name)
```

which expands to the *quoted definition* of *name*. If the argument is not a defined macro, the expansion is void.

If *name* is a user-defined macro, the quoted definition is simply the quoted expansion text. If, instead, *name* is a built-in, the expansion is a special token, which points to the built-in's internal definition. This token is only meaningful as the second argument to `define` (and `pushdef`), and is ignored in any other context.

Its normal use is best understood through an example, which shows how to rename `undefine` to `zap`:

```
define(‘zap’, defn(‘undefine’))
⇒
zap(‘undefine’)
⇒
undefine(‘zap’)
⇒undefine(zap)
```

In this way, `defn` can be used to copy macro definitions, and also definitions of built-in macros. Even if the original macro is removed, the other name can still be used to access the definition.

7.6 Temporarily redefining macros

It is possible to redefine a macro temporarily, reverting to the previous definition at a later time. This is done with the built-ins `pushdef` and `popdef`:

```
pushdef(name, expansion)
popdef(name)
```

which are quite analogous to `define` and `undefine`.

These macros work in a stack-like fashion. A macro is temporarily redefined with `pushdef`, which replaces an existing definition of *name*, while saving the previous definition, before the new one is installed. If there is no previous definition, `pushdef` behaves exactly like `define`.

If a macro has several definitions (of which only one is accessible), the topmost definition can be removed with `popdef`. If there is no previous definition, `popdef` does nothing.

```
define(‘foo’, ‘Expansion one.’)
⇒
foo
⇒Expansion one.
pushdef(‘foo’, ‘Expansion two.’)
⇒
foo
⇒Expansion two.
popdef(‘foo’)
⇒
foo
⇒Expansion one.
popdef(‘foo’)
⇒
foo
⇒foo
```

If a macro with several definitions is redefined with `define`, the topmost definition is *replaced* with the new definition. If it is removed with `undefine`, *all* the definitions are removed, and not only the topmost one.

```
define(‘foo’, ‘Expansion one.’)
```

```
⇒
foo
⇒Expansion one.
pushdef(‘foo’, ‘Expansion two.’)
⇒
foo
⇒Expansion two.
define(‘foo’, ‘Second expansion two.’)
⇒
foo
⇒Second expansion two.
undefine(‘foo’)
⇒
foo
⇒foo
```

It is possible to temporarily redefine a built-in with `pushdef` and `defn`.

7.7 Indirect call of macros

Any macro can be called indirectly with `indir`:

```
indir(name, ...)
```

which results in a call to the macro *name*, which is passed the rest of the arguments. This can be used to call macros with "illegal" names (`define` allows such names to be defined):

```
define(‘$$internal$macro’, ‘Internal macro (name ‘$0’)’)
⇒
$$internal$macro
⇒$$internal$macro
indir(‘$$internal$macro’)
⇒Internal macro (name $$internal$macro)
```

The point is, here, that larger macro packages can have private macros defined, that will not be called by accident. They can *only* be called through the built-in `indir`.

7.8 Indirect call of built-ins

Built-in macros can be called indirectly with `built-in`:

```
builtin(name, ...)
```

which results in a call to the built-in *name*, which is passed the rest of the arguments. This can be used, if *name* has been given another definition that has covered the original.

8 Conditionals, loops and recursion

Macros, expanding to plain text, perhaps with arguments, are not quite enough. We would like to have macros expand to different things, based on decisions taken at run-time. E.g., we need some kind of conditionals. Also, we would like to have some kind of loop construct, so we could do something a number of times, or while some condition is true.

8.1 Testing macro definitions

There are two different built-in conditionals in `m4`. The first is `ifdef`:

```
ifdef(name, string-1, opt string-2)
```

which makes it possible to test whether a macro is defined or not. If *name* is defined as a macro, `ifdef` expands to *string-1*, otherwise to *string-2*. If *string-2* is omitted, it is taken to be the empty string (according to the normal rules).

```
ifdef(‘foo’, ‘‘foo’ is defined’, ‘‘foo’ is not defined’)
⇒foo is not defined
define(‘foo’, ‘’)
⇒
ifdef(‘foo’, ‘‘foo’ is defined’, ‘‘foo’ is not defined’)
⇒foo is defined
```

8.2 Comparing strings

The other conditional, `ifelse`, is much more powerful. It can be used as a way to introduce a long comment, as an if-else construct, or as a multibranch, depending on the number of arguments supplied:

```
ifelse(comment)
ifelse(string-1, string-2, equal, opt not-equal)
ifelse(string-1, string-2, equal, ...)
```

Used with only one argument, the `ifelse` simply discards it and produces no output. This is a common `m4` idiom for introducing a block comment, as an alternative to repeatedly using `dnl`. This special usage is recognized by GNU `m4`, so that in this case, the warning about missing arguments is never triggered.

If called with three or four arguments, `ifelse` expands into *equal*, if *string-1* and *string-2* are equal (character for character), otherwise it expands to *not-equal*.

```
ifelse(foo, bar, ‘true’)
⇒
ifelse(foo, foo, ‘true’)
⇒true
ifelse(foo, bar, ‘true’, ‘false’)
⇒false
ifelse(foo, foo, ‘true’, ‘false’)
⇒true
```

However, `ifelse` can take more than four arguments. If given more than four arguments, `ifelse` works like a `case` or `switch` statement in traditional programming languages. If

string-1 and *string-2* are equal, `ifelse` expands into *equal*, otherwise the procedure is repeated with the first three arguments discarded. This calls for an example:

```
ifelse(foo, bar, ‘third’, gnu, gnats, ‘sixth’, ‘seventh’)
⇒seventh
```

Naturally, the normal case will be slightly more advanced than these examples. A common use of `ifelse` is in macros implementing loops of various kinds.

8.3 Loops and recursion

There is no direct support for loops in `m4`, but macros can be recursive. There is no limit on the number of recursion levels, other than those enforced by your hardware and operating system.

Loops can be programmed using recursion and the conditionals described previously.

There is a built-in macro, `shift`, which can, among other things, be used for iterating through the actual arguments to a macro:

```
shift(...)
```

It takes any number of arguments, and expands to all but the first argument, separated by commas, with each argument quoted.

```
shift(bar)
⇒
shift(foo, bar, baz)
⇒bar,baz
```

An example of the use of `shift` is this macro, which reverses the order of its arguments:

```
define(‘reverse’, ‘ifelse($#, 0, , $#, 1, ‘‘$1’’,
  ‘reverse(shift($@)), ‘$1’’)’)
⇒
reverse
⇒
reverse(foo)
⇒foo
reverse(foo, bar, gnats,and gnus)
⇒and gnus, gnats, bar, foo
```

While not a very interesting macro, it does show how simple loops can be made with `shift`, `ifelse` and recursion.

Here is an example of a loop macro that implements a simple forloop. It can, for example, be used for simple counting:

```
forloop(‘i’, 1, 8, ‘i ’)
⇒1 2 3 4 5 6 7 8
```

The arguments are a name for the iteration variable, the starting value, the final value, and the text to be expanded for each iteration. With this macro, the macro `i` is defined only within the loop. After the loop, it retains whatever value it might have had before.

For-loops can be nested, like

```
forloop(‘i’, 1, 4, ‘forloop(‘j’, 1, 8, ‘(i, j) ’)
’)
```

```
⇒(1, 1) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7) (1, 8)
⇒(2, 1) (2, 2) (2, 3) (2, 4) (2, 5) (2, 6) (2, 7) (2, 8)
⇒(3, 1) (3, 2) (3, 3) (3, 4) (3, 5) (3, 6) (3, 7) (3, 8)
⇒(4, 1) (4, 2) (4, 3) (4, 4) (4, 5) (4, 6) (4, 7) (4, 8)
⇒
```

The implementation of the `forloop` macro is fairly straightforward. The `forloop` macro itself is simply a wrapper, which saves the previous definition of the first argument, calls the internal macro `_forloop`, and re-establishes the saved definition of the first argument.

The macro `_forloop` expands the fourth argument once, and tests to see if it is finished. If it has not finished, it increments the iteration variable (using the predefined macro `incr` (see Section 14.1 [Incr], page 33, for details)), and recurses.

Here is the actual implementation of `forloop`:

```
define(‘forloop’,
       ‘pushdef(‘$1’, ‘$2’)_forloop(‘$1’, ‘$2’, ‘$3’, ‘$4’)popdef(‘$1’)’)
define(‘_forloop’,
       ‘$4‘’ifelse($1, ‘$3’, ,
   ‘define(‘$1’, incr($1))_forloop(‘$1’, ‘$2’, ‘$3’, ‘$4’)’)’)
```

Notice the careful use of quotes. Only three macro arguments are unquoted, each for its own reason. Try to find out *why* these three arguments are left unquoted, and see what happens if they are quoted.

Now, even though these two macros are useful, they are still not robust enough for general use. They lack even basic error handling of cases like start value less than final value, and the first argument not being a name. Correcting these errors are left as an exercise to the reader.

9 How to debug macros and input

When writing macros for `m4`, most of the time they won't work as intended (as is the case with most programming languages). There is a little support for macro debugging in `m4`.

9.1 Displaying macro definitions

If you want to see what a name expands into, you can use the built-in `dumpdef`:

```
dumpdef(...)
```

which accepts any number of arguments. If called without any arguments, it displays the definitions of all known names, otherwise it displays the definitions of the names given. The output is printed directly on the standard error output.

The expansion of `dumpdef` is void.

```
define(‘foo’, ‘Hello world.’)
⇒
dumpdef(‘foo’)
error foo:  ‘Hello world.’
⇒
dumpdef(‘define’)
error define:  <define>
⇒
```

The last example shows how built-in macros definitions are displayed.

See Section 9.3 [Debug Levels], page 19, for information on controlling the details of the display.

9.2 Tracing macro calls

It is possible to trace macro calls and expansions through the built-ins `traceon` and `traceoff`:

```
traceon(...)
traceoff(...)
```

When called without any arguments, `traceon` and `traceoff` will turn tracing on and off, respectively, for all defined macros. When called with arguments, only the named macros are affected.

The expansion of `traceon` and `traceoff` is void.

Whenever a traced macro is called and the arguments have been collected, the call is displayed. If the expansion of the macro call is not void, the expansion can be displayed after the call. The output is printed directly on the standard error output.

```
define(‘foo’, ‘Hello World.’)
⇒
define(‘echo’, ‘$@’)
⇒
traceon(‘foo’, ‘echo’)
⇒
foo
```

```
error m4trace: -1- foo -> ‘Hello World.’
⇒Hello World.
echo(gnus, and gnats)
error m4trace: -1- echo(‘gnus’, ‘and gnats’) -> ‘‘gnus’,‘and gnats’’
⇒gnus,and gnats
```

The number between dashes is the depth of the expansion. It is one most of the time, signifying an expansion at the outermost level, but it increases when macro arguments contain unquoted macro calls.

See Section 9.3 [Debug Levels], page 19, for information on controlling the details of the display.

9.3 Controlling debugging output

The ‘`-d`’ option to `m4` controls the amount of details presented, when using the macros described in the preceding sections.

The *flags* following the option can be one or more of the following:

`t` Trace all macro calls made in this invocation of `m4`.

`a` Show the actual arguments in each macro call. This applies to all macro calls if the ‘`t`’ flag is used, otherwise only the macros covered by calls of `traceon`.

`e` Show the expansion of each macro call, if it is not void. This applies to all macro calls if the ‘`t`’ flag is used, otherwise only the macros covered by calls of `traceon`.

`q` Quote actual arguments and macro expansions in the display with the current quotes.

`c` Show several trace lines for each macro call. A line is shown when the macro is seen, but before the arguments are collected; a second line when the arguments have been collected and a third line after the call has completed.

`x` Add a unique ‘macro call id’ to each line of the trace output. This is useful in connection with the ‘`c`’ flag above.

`f` Show the name of the current input file in each trace output line.

`l` Show the the current input line number in each trace output line.

`p` Print a message when a named file is found through the path search mecanism (see Section 11.2 [Search Path], page 25), giving the actual filename used.

`i` Print a message each time the current input file is changed, giving file name and input line number.

`V` A shorthand for all of the above flags.

If no flags are specified with the ‘`-d`’ option, the default is ‘`aeq`’. The examples in the previous two sections assumed the default flags.

There is a built-in macro `debugmode`, which allows on-the-fly control of the debugging output format:

```
debugmode(opt flags)
```

The argument *flags* should be a subset of the letters listed above. As special cases, if the argument starts with a '`+`', the flags are added to the current debug flags, and if it starts with a '`-`', they are removed. If no argument is present, the debugging flags are set to zero (as if no '`-d`' was given), and with an empty argument the flags are reset to the default.

9.4 Saving debugging output

Debug and tracing output can be redirected to files using either the '`-o`' option to `m4`, or with the built-in macro `debugfile`:

```
debugfile(opt filename)
```

will send all further debug and trace output to *filename*. If *filename* is empty, debug and trace output are discarded and if `debugfile` is called without any arguments, debug and trace output are sent to the standard error output.

10 Input control

This chapter describes various built-in macros for controlling the input to `m4`.

10.1 Deleting whitespace in input

The built-in `dnl` reads and discards all characters, up to and including the first newline:

```
dnl
```

and it is often used in connection with `define`, to remove the newline that follow the call to `define`. Thus

```
define(‘foo’, ‘Macro ‘foo’.’)dnl A very simple macro, indeed.
foo
⇒Macro foo.
```

The input up to and including the next newline is discarded, as opposed to the way comments are treated (see Section 5.4 [Comments], page 7).

Usually, `dnl` is immediately followed by an end of line or some other whitespace. GNU `m4` will produce a warning diagnostic if `dnl` is followed by an open parenthesis. In this case, `dnl` will collect and process all arguments, looking for a matching close parenthesis. All predictable side effects resulting from this collection will take place. `dnl` will return no output. The input following the matching close parenthesis up to and including the next newline, on whatever line containing it, will still be discarded.

10.2 Changing the quote characters

The default quote delimiters can be changed with the built-in `changequote`:

```
changequote(opt start, opt end)
```

where *start* is the new start-quote delimiter and *end* is the new end-quote delimiter. If any of the arguments are missing, the default quotes (‘ and ’) are used instead of the void arguments.

The expansion of `changequote` is void.

```
changequote([,])
⇒
define([foo], [Macro [foo].])
⇒
foo
⇒Macro foo.
```

If no single character is appropriate, *start* and *end* can be of any length.

```
changequote([[,]])
⇒
define([[foo]], [[Macro [[[foo]]].]])
⇒
foo
⇒Macro [foo].
```

Changing the quotes to the empty strings will effectively disable the quoting mechanism, leaving no way to quote text.

```
define(‘foo’, ‘Macro ‘FOO’.’)
⇒
changequote(,)
⇒
foo
⇒Macro ‘FOO’.
‘foo’
⇒‘Macro ‘FOO’.’
```

There is no way in `m4` to quote a string containing an unmatched left quote, except using `changequote` to change the current quotes.

Neither quote string should start with a letter or ‘_’ (underscore), as they will be confused with names in the input. Doing so disables the quoting mechanism.

10.3 Changing comment delimiters

The default comment delimiters can be changed with the built-in macro `changecom`:

`changecom(opt` *start*`, opt` *end*`)`

where *start* is the new start-comment delimiter and *end* is the new end-comment delimiter. If any of the arguments are void, the default comment delimiters (`#` and newline) are used instead of the void arguments. The comment delimiters can be of any length.

The expansion of `changecom` is void.

```
define(‘comment’, ‘COMMENT’)
⇒
# A normal comment
⇒# A normal comment
changecom(‘/*’, ‘*/’)
⇒
# Not a comment anymore
⇒# Not a COMMENT anymore
But: /* this is a comment now */ while this is not a comment
⇒But: /* this is a comment now */ while this is not a COMMENT
```

Note how comments are copied to the output, much as it they were quoted strings. If you want the text inside a comment expanded, quote the start comment delimiter.

Calling `changecom` without any arguments disables the commenting mechanism completely.

```
define(‘comment’, ‘COMMENT’)
⇒
changecom
⇒
# Not a comment anymore
⇒# Not a COMMENT anymore
```

10.4 Saving input

It is possible to 'save' some text until the end of the normal input has been seen. Text can be saved, to be read again by `m4` when the normal input has been exhausted. This feature is normally used to initiate cleanup actions before normal exit, e.g., deleting temporary files.

To save input text, use the built-in `m4wrap`:

`m4wrap(`*`string`*`, ...)`

which stores *string* and the rest of the arguments in a safe place, to be reread when end of input is reached.

```
define(‘cleanup’, ‘This is the ‘cleanup’ actions.
’)
⇒
m4wrap(‘cleanup’)
⇒
This is the first and last normal input line.
⇒This is the first and last normal input line.
^D
⇒This is the cleanup actions.
```

The saved input is only reread when the end of normal input is seen, and not if `m4exit` is used to exit `m4`.

It is safe to call `m4wrap` from saved text, but then the order in which the saved text is reread is undefined. If `m4wrap` is not used recursively, the saved pieces of text are reread in the opposite order in which they were saved (LIFO—last in, first out).

11 File inclusion

`m4` allows you to include named files at any point in the input.

11.1 Including named files

There are two built-in macros in `m4` for including files:

```
include(filename)
sinclude(filename)
```

both of which cause the file named *filename* to be read by `m4`. When the end of the file is reached, input is resumed from the previous input file.

The expansion of `include` and `sinclude` is therefore the contents of *filename*.

It is an error for an `included` file not to exist. If you don't want error messages about non-existent files, `sinclude` can be used to include a file, if it exists, expanding to nothing if it does not.

```
include(‘no-such-file’)
⇒
error m4:30.include:2: can’t open no-such-file: No such file or directory
sinclude(‘no-such-file’)
⇒
```

Assume in the following that the file `incl.m4` contains the lines:

```
Include file start
foo
Include file end
```

Normally file inclusion is used to insert the contents of a file into the input stream. The contents of the file will be read by `m4` and macro calls in the file will be expanded:

```
define(‘foo’, ‘FOO’)
⇒
include(‘incl.m4’)
⇒Include file start
⇒FOO
⇒Include file end
⇒
```

The fact that `include` and `sinclude` expand to the contents of the file can be used to define macros that operate on entire files. Here is an example, which defines ‘`bar`’ to expand to the contents of `incl.m4`:

```
define(‘bar’, include(‘incl.m4’))
⇒
This is ‘bar’:  >>>bar<<<
⇒This is bar:  >>>Include file start
⇒foo
⇒Include file end
⇒<<<
```

This use of `include` is not trivial, though, as files can contain quotes, commas and parentheses, which can interfere with the way the `m4` parser works.

11.2 Searching for include files

GNU `m4` allows included files to be found in other directories than the current working directory.

If a file is not found in the current working directory, and the file name is not absolute, the file will be looked for in a specified search path. First, the directories specified with the '`-I`' option will be searched, in the order found on the command line. Second, if the '`M4PATH`' environment variable is set, it is expected to contain a colon-separated list of directories, which will be searched in order.

If the automatic search for include-files causes trouble, the '`p`' debug flag (see Section 9.3 [Debug Levels], page 19) can help isolate the problem.

12 Diverting and undiverting output

Diversions are a way of temporarily saving output. The output of `m4` can at any time be diverted to a temporary file, and be reinserted into the output stream, *undiverted*, again at a later time.

Up to ten numbered diversions (numbered from 0 to 9) are supported in `m4`, of which diversion number 0 is the normal output stream. The number of available diversions can be increased with the '`-N`' option.

12.1 Diverting output

Output is diverted using `divert`:

```
divert(opt number)
```

where *number* is the diversion to be used. If *number* is left out, it is assumed to be zero.

The expansion of `divert` is void.

Diverted output, that hasn't been explicitly undiverted, will be undiverted when all the input has been processed.

```
divert(1)
This text is diverted.
divert
⇒
This text is not diverted.
⇒This text is not diverted.
^D
⇒
⇒This text is diverted.
```

Several calls of `divert` with the same argument do not overwrite the previous diverted text, but append to it.

If output is diverted to an non-existent diversion, it is simply discarded. This can be used to suppress unwanted output. A common example of unwanted output is the trailing newlines after macro definitions. Here is how to avoid them.

```
divert(-1)
define(‘foo’, ‘Macro ‘foo’.’)
define(‘bar’, ‘Macro ‘bar’.’)
divert
⇒
```

This is a common programming idiom in `m4`.

12.2 Undiverting output

Diverted text can be undiverted explicitly using the built-in `undivert`:

```
undivert(opt number, ...)
```

which undiverts the diversions given by the arguments, in the order given. If no arguments are supplied, all diversions are undiverted, in numerical order.

The expansion of `undivert` is void.

```
divert(1)
This text is diverted.
divert
⇒
This text is not diverted.
⇒This text is not diverted.
undivert(1)
⇒
⇒This text is diverted.
⇒
```

Notice the last two blank lines. One of them comes from the newline following `undivert`, the other from the newline that followed the `divert`! A diversion often starts with a blank line like this.

When diverted text is undiverted, it is *not* reread by `m4`, but rather copied directly to the current output, and it is therefore not an error to undivert into a diversion.

When a diversion has been undiverted, the diverted text is discarded, and it is not possible to bring back diverted text more than once.

```
divert(1)
This text is diverted first.
divert(0)undivert(1)dnl
⇒
⇒This text is diverted first.
undivert(1)
⇒
divert(1)
This text is also diverted but not appended.
divert(0)undivert(1)dnl
⇒
⇒This text is also diverted but not appended.
```

Attempts to undivert the current diversion are silently ignored.

GNU `m4` allows named files to be undiverted. Given a non-numeric argument, the contents of the file named will be copied, uninterpreted, to the current output. This complements the built-in `include` (see Section 11.1 [Include], page 24). To illustrate the difference, assume the file `foo` contains the word ‘`bar`’:

```
define(‘bar’, ‘BAR’)
⇒
undivert(‘foo’)
⇒bar
⇒
include(‘foo’)
⇒BAR
⇒
```

12.3 Diversion numbers

The built-in `divnum`:

```
divnum
```

expands to the number of the current diversion.

```
Initial divnum
⇒Initial 0
divert(1)
Diversion one: divnum
divert(2)
Diversion two: divnum
divert
⇒
^D
⇒
⇒Diversion one: 1
⇒
⇒Diversion two: 2
```

The last call of `divert` without argument is necessary, since the undiverted text would otherwise be diverted itself.

12.4 Discarding diverted text

Often it is not known, when output is diverted, whether the diverted text is actually needed. Since all non-empty diversion are brought back when the end of input is seen, a method of discarding a diversion is needed. If all diversions should be discarded, the easiest is to end the input to `m4` with '`divert(-1)`':

```
divert(1)
Diversion one: divnum
divert(2)
Diversion two: divnum
divert(-1)
^D
```

No output is produced at all.

Clearing selected diversions can be done with the following macro:

```
define(‘cleardivert’,
‘pushdef(‘_num’, divnum)divert(-1)undivert($@)divert(_num)popdef(‘_num’)’)
⇒
```

It is called just like `undivert`, but the effect is to clear the diversions, given by the arguments. (This macro has a nasty bug! You should try to see if you can find it and correct it.)

13 Macros for text handling

There are a number of built-ins in `m4` for manipulating text in various ways, extracting substrings, searching, substituting, and so on.

13.1 Calculating length of strings

The length of a string can be calculated by `len`:

`len(`*`string`*`)`

which expands to the length of *string*, as a decimal number.

```
len()
⇒0
len(‘abcdef’)
⇒6
```

13.2 Searching for substrings

Searching for substrings is done with `index`:

`index(`*`string`*`, `*`substring`*`)`

which expands to the index of the first occurrence of *substring* in *string*. The first character in *string* has index 0. If *substring* does not occur in *string*, `index` expands to ‘`-1`’.

```
index(‘gnus, gnats, and armadillos’, ‘nat’)
⇒7
index(‘gnus, gnats, and armadillos’, ‘dag’)
⇒-1
```

13.3 Searching for regular expressions

Searching for regular expressions is done with the built-in `regexp`:

`regexp(`*`string`*`, `*`regexp`*`, opt `*`replacement`*`)`

which searches for *regexp* in *string*. The syntax for regular expressions is the same as in GNU Emacs. See Section “Syntax of Regular Expressions” in *The GNU Emacs Manual*.

If *replacement* is omitted, `regexp` expands to the index of the first match of *regexp* in *string*. If *regexp* does not match anywhere in *string*, it expands to -1.

```
regexp(‘GNUs not Unix’, ‘\<[a-z]\w+’)
⇒5
regexp(‘GNUs not Unix’, ‘\<Q\w*’)
⇒-1
```

If *replacement* is supplied, `regexp` changes the expansion to this argument, with ‘`\&`’ substituted by *string*, and ‘`\`*`n`*’ substituted by the text matched by the *n*th parenthesized sub-expression of *regexp*, ‘`\0`’ being the text the entire regular expression matched.

```
regexp(‘GNUs not Unix’, ‘\w\(\w+\)$’, ‘*** \0 *** \1 ***’)
⇒*** Unix *** nix ***
```

13.4 Extracting substrings

Substrings are extracted with `substr`:

```
substr(string, from, opt length)
```

which expands to the substring of *string*, which starts at index *from*, and extends for *length* characters, or to the end of *string*, if *length* is omitted. The starting index of a string is always 0.

```
substr(‘gnus, gnats, and armadillos’, 6)
⇒gnats, and armadillos
substr(‘gnus, gnats, and armadillos’, 6, 5)
⇒gnats
```

13.5 Translating characters

Character translation is done with `translit`:

```
translit(string, chars, replacement)
```

which expands to *string*, with each character that occurs in *chars* translated into the character from *replacement* with the same index.

If *replacement* is shorter than *chars*, the excess characters are deleted from the expansion. If *replacement* is omitted, all characters in *string*, that are present in *chars* are deleted from the expansion.

Both *chars* and *replacement* can contain character-ranges, e.g., ‘`a-z`’ (meaning all lowercase letters) or ‘`0-9`’ (meaning all digits). To include a dash ‘`-`’ in *chars* or *replacement*, place it first or last.

It is not an error for the last character in the range to be ‘larger’ than the first. In that case, the range runs backwards, i.e., ‘`9-0`’ means the string ‘`9876543210`’.

```
translit(‘GNUs not Unix’, ‘A-Z’)
⇒s not nix
translit(‘GNUs not Unix’, ‘a-z’, ‘A-Z’)
⇒GNUS NOT UNIX
translit(‘GNUs not Unix’, ‘A-Z’, ‘z-a’)
⇒tmfs not fnix
```

The first example deletes all uppercase letters, the second converts lowercase to uppercase, and the third ‘mirrors’ all uppercase letters, while converting them to lowercase. The two first cases are by far the most common.

13.6 Substituting text by regular expression

Global substitution in a string is done by `patsubst`:

```
patsubst(string, regexp, opt replacement)
```

which searches *string* for matches of *regexp*, and substitutes *replacement* for each match. The syntax for regular expressions is the same as in GNU Emacs.

The parts of *string* that are not covered by any match of *regexp* are copied to the expansion. Whenever a match is found, the search proceeds from the end of the match, so

a character from *string* will never be substituted twice. If *regexp* matches a string of zero length, the start position for the search is incremented, to avoid infinite loops.

When a replacement is to be made, *replacement* is inserted into the expansion, with '`\&`' substituted by *string*, and '`\n`' substituted by the text matched by the *n*th parenthesized sub-expression of *regexp*, '`\0`' being the text the entire regular expression matched.

The *replacement* argument can be omitted, in which case the text matched by *regexp* is deleted.

```
patsubst(‘GNUs not Unix’, ‘^’, ‘OBS: ’)
⇒OBS: GNUs not Unix
patsubst(‘GNUs not Unix’, ‘\<’, ‘OBS: ’)
⇒OBS: GNUs OBS: not OBS: Unix
patsubst(‘GNUs not Unix’, ‘\w*’, ‘(\0)’)
⇒(GNUs)() (not)() (Unix)
patsubst(‘GNUs not Unix’, ‘\w+’, ‘(\0)’)
⇒(GNUs) (not) (Unix)
patsubst(‘GNUs not Unix’, ‘[A-Z][a-z]+’)
⇒GN not
```

Here is a slightly more realistic example, which capitalizes individual word or whole sentences, by substituting calls of the macros `upcase` and `downcase` into the strings.

```
define(‘upcase’, ‘translit(‘$*’, ‘a-z’, ‘A-Z’)’)dnl
define(‘downcase’, ‘translit(‘$*’, ‘A-Z’, ‘a-z’)’)dnl
define(‘capitalize1’,
     ‘regexp(‘$1’, ‘^\(\w\)\(\w*\)’, ‘upcase(‘\1’)‘’downcase(‘\2’)’)’)dnl
define(‘capitalize’,
     ‘patsubst(‘$1’, ‘\w+’, ‘capitalize1(‘\0’)’)’)dnl
capitalize(‘GNUs not Unix’)
⇒Gnus Not Unix
```

13.7 Formatted output

Formatted output can be made with `format`:

```
format(format-string, ...)
```

which works much like the C function `printf`. The first argument is a format string, which can contain '`%`' specifications, and the expansion of `format` is the formatted string.

Its use is best described by a few examples:

```
define(‘foo’, ‘The brown fox jumped over the lazy dog’)
⇒
format(‘The string "%s" is %d characters long’, foo, len(foo))
⇒The string "The brown fox jumped over the lazy dog" is 38 characters long
```

Using the `forloop` macro defined in See Section 8.3 [Loops], page 16, this example shows how `format` can be used to produce tabular output.

```
forloop(‘i’, 1, 10, ‘format(‘%6d squared is %10d
’, i, eval(i^2))’)
⇒     1 squared is     1
⇒     2 squared is     4
```

```
⇒      3 squared is     9
⇒      4 squared is    16
⇒      5 squared is    25
⇒      6 squared is    36
⇒      7 squared is    49
⇒      8 squared is    64
⇒      9 squared is    81
⇒     10 squared is   100
```

The built-in `format` is modeled after the ANSI C '`printf`' function, and supports the normal '`%`' specifiers: '`c`', '`s`', '`d`', '`o`', '`x`', '`X`', '`u`', '`e`', '`E`' and '`f`'; it supports field widths and precisions, and the modifiers '`+`', '`-`', ' ', '`0`', '`#`', '`h`' and '`l`'. For more details on the functioning of `printf`, see the C Library Manual.

14 Macros for doing arithmetic

Integer arithmetic is included in `m4`, with a C-like syntax. As convenient shorthands, there are built-ins for simple increment and decrement operations.

14.1 Decrement and increment operators

Increment and decrement of integers are supported using the built-ins `incr` and `decr`:

```
incr(number)
decr(number)
```

which expand to the numerical value of *number*, incremented, or decremented, respectively, by one.

```
incr(4)
⇒5
decr(7)
⇒6
```

14.2 Evaluating integer expressions

Integer expressions are evaluated with `eval`:

```
eval(expression, opt radix, opt width)
```

which expands to the value of *expression*.

Expressions can contain the following operators, listed in order of decreasing precedence.

`-` Unany minus

`** ^` Exponentiation

`* / %` Multiplication, division and modulo

`+ -` Addition and subtraction

`== != > >= < <=`
Relational operators

`!` Logical negation

`&` Bitwise and

`|` Bitwise or

`&&` Logical and

`||` Logical or

All operators, except exponentiation, are left associative.

Numbers can be given in decimal, octal (starting with `0`), or hexadecimal (starting with `0x`).

Parentheses may be used to group subexpressions whenever needed. For the relational operators, a true relation returns `1`, and a false relation return `0`.

Here are a few examples of use of `eval`.

```
eval(-3 * 5)
```

```
⇒-15
eval(index(‘Hello world’, ‘llo’) >= 0)
⇒1
define(‘square’, ‘eval(($1)^2)’)
⇒
square(9)
⇒81
square(square(5)+1)
⇒676
define(‘foo’, ‘666’)
⇒
eval(‘foo’/6)
error m4:51.eval:14: bad expression in eval: foo/6
⇒
eval(foo/6)
⇒111
```

As the second to last example shows, `eval` does not handle macro names, even if they expand to a valid expression (or part of a valid expression). Therefore all macros must be expanded before they are passed to `eval`.

If *radix* is specified, it specifies the radix to be used in the expansion. The default radix is 10. The result of `eval` is always taken to be signed. The *width* argument specifies a minimum output width. The result is zero-padded to extend the expansion to the requested width.

```
eval(666, 10)
⇒666
eval(666, 11)
⇒556
eval(666, 6)
⇒3030
eval(666, 6, 10)
⇒0000003030
eval(-666, 6, 10)
⇒-000003030
```

Please take note that *radix* cannot be larger than 36 in the current implemention. Which characters can be used as digits, if the radix is larger than 36? Currently any radix larger than 36 are rejected.

15 Running Unix commands

There are a few built-in macros in `m4` that allow you to run Unix commands from within `m4`.

15.1 Executing simple commands

Any shell command can be executed, using `syscmd`:

```
syscmd(shell-command)
```

which executes *shell-command* as a shell command.

The expansion of `syscmd` is void.

The expansion is *not* the output from the command! Instead the standard input, output and error of the command are the same as those of `m4`. This means that output or error messages from the commands are not read by `m4`, and might get mixed up with the normal output from `m4`. This can produce unexpected results. It is therefore a good habit to always redirect the input and output of shell commands used with `syscmd`.

15.2 Reading the output of commands

If you want `m4` to read the output of a Unix command, use `esyscmd`:

```
esyscmd(shell-command)
```

which expands to the standard output of the shell command *shell-command*.

The error output of *shell-command* is not a part of the expansion. It will appear along with the error output of `m4`. Assume you are positioned into the `checks` directory of GNU `m4` distribution, then:

```
define(‘vice’, ‘esyscmd(grep Vice COPYING)’)
⇒
vice
⇒  Ty Coon, President of Vice
⇒
```

Note how the expansion of `esyscmd` has a trailing newline.

15.3 Exit codes

To see whether a shell command succeeded, use `sysval`:

```
sysval
```

which expands to the exit status of the last shell command run with `syscmd` or `esyscmd`.

```
syscmd(‘false’)
⇒
ifelse(sysval, 0, zero, non-zero)
⇒non-zero
syscmd(‘true’)
⇒
sysval
⇒0
```

15.4 Making names for temporary files

Commands specified to `syscmd` or `esyscmd` might need a temporary file, for output or for some other purpose. There is a built-in macro, `maketemp`, for making temporary file names:

```
maketemp(template)
```

which expands to a name of a non-existent file, made from the string *template*, which should end with the string '`XXXXXX`'. The six `X`'s are then replaced, usually with something that includes the process id of the `m4` process, in order to make the filename unique.

```
maketemp(‘/tmp/fooXXXXXX’)
⇒/tmp/fooa07346
maketemp(‘/tmp/fooXXXXXX’)
⇒/tmp/fooa07346
```

As seen in the example, several calls of `maketemp` might expand to the same string, since the selection criteria is whether the file exists or not. If a file has not been created before the next call, the two macro calls might expand to the same name.

16 Miscellaneous built-in macros

This chapter describes various built-ins, that don't really belong in any of the previous chapters.

16.1 Printing error messages

You can print error messages using `errprint`:

```
errprint(message, ...)
```

which simply prints *message* and the rest of the arguments on the standard error output.

The expansion of `errprint` is void.

```
errprint(‘Illegal arguments to forloop
’)
error Illegal arguments to forloop
⇒
```

A trailing newline is *not* printed automatically, so it must be supplied as part of the argument, as in the example.

To make it possible to specify the location of the error, two utility built-ins exist:

```
__file__
__line__
```

which expands to the quoted name of the current input file, and the current input line number in that file.

```
errprint(‘m4:’__file__:__line__: ‘Input error
’)
error m4:56.errprint:2: Input error
⇒
```

16.2 Exiting from `m4`

If you need to exit from `m4` before the entire input has been read, you can use `m4exit`:

```
m4exit(opt code)
```

which causes `m4` to exit, with exit code *code*. If *code* is left out, the exit code is zero.

```
define(‘fatal_error’, ‘errprint(‘m4: ’__file__: __line__‘: fatal error: $*
’)m4exit(1)’)
⇒
fatal_error(‘This is a BAD one, buster’)
error m4: 57.m4exit: 5: fatal error: This is a BAD one, buster
```

After this macro call, `m4` will exit with exit code 1. This macro is only intended for error exits, since the normal exit procedures are not followed, e.g., diverted text is not undiverted, and saved text (see Section 10.4 [M4wrap], page 23) is not reread.

17 Compatibility with other versions of m4

This chapter describes the differences between this implementation of m4, and the implementation found under Unix, notably System V, Release 3.

17.1 Extensions in GNU m4

This version of m4 contains a few facilities, that do not exist in System V m4. These extra facilities are all suppressed by using the '-G' command line option, unless overridden by other command line options.

- In the $n notation for macro arguments, n can contain several digits, while the System V m4 only accepts one digit. This allows macros in GNU m4 to take any number of arguments, and not only nine (see Section 7.2 [Arguments], page 10).
- Files included with include and sinclude are sought in a user specified search path, if they are not found in the working directory. The search path is specified by the '-I' option and the 'M4PATH' environment variable (see Section 11.2 [Search Path], page 25).
- Arguments to undivert can be non-numeric, in which case the named file will be included uninterpreted in the output (see Section 12.2 [Undivert], page 26).
- Formatted output is supported through the format built-in, which is modeled after the C library function printf (see Section 13.7 [Format], page 31).
- Searches and text substitution through regular expressions are supported by the regexp and patsubst built-ins (see Section 13.3 [Regexp], page 29, and See Section 13.6 [Patsubst], page 30).
- The output of shell commands can be read into m4 with esyscmd (see Section 15.2 [Esyscmd], page 35).
- There is indirect access to any built-in macro with builtin (see Section 7.8 [Builtin], page 14).
- Macros can be called indirectly through indir (see Section 7.7 [Indir], page 14).
- The name of the current input file and the current input line number are accessible through the built-ins __file__ and __line__ (see Section 16.1 [Errprint], page 37).
- The format of the output from dumpdef and macro tracing can be controlled with debugmode (see Section 9.3 [Debug Levels], page 19).
- The destination of trace and debug output can be controlled with debugfile (see Section 9.4 [Debug Output], page 20).

In addition to the above extensions, GNU m4 implements the following command line options: '-V', '-d', '-l', '-o', '-N', '-I' and '-t'. For a description of these options, see Chapter 4 [Invoking m4], page 4,

Also, the debugging and tracing facilities in GNU m4 are much more extensive than in most other versions of m4.

17.2 Facilities in System V m4 not in GNU m4

The version of m4 from System V contains a few facilities that have not been implemented in GNU m4 yet.

- System V m4 supports multiple arguments to defn. This is not implemented in GNU m4. Its usefulness is unclear to me.

17.3 Other incompatibilities

There are a few other incompatibilities between this implementation of m4, and the System V version.

- GNU m4 implements sync lines differently from System V m4, when text is being diverted. GNU m4 outputs the sync lines when the text is being diverted, and System V m4 when the diverted text is being brought back.

  The problem is which lines and filenames should be attached to text that is being, or has been, diverted. System V m4 regards all the diverted text as being generated by the source line containing the undivert call, whereas GNU m4 regards the diverted text as being generated at the time it is diverted.

  Which is right? I expect the sync line option to be used mostly when using m4 as a front end to a compiler. If a diverted line causes a compiler error, I believe that the error messages should refer to the place where the diversion were made, and not where it was inserted again. Comments anyone?

  Anyway, GNU m4's approach causes a serious bug, if calls to undivert aren't alone on the input line. See the file examples/divert.m4 for a demonstration of the bug. I don't feel it is acceptable to insert newlines in the output the user hasn't asked for.

- GNU m4 without '-G' option will define the macro __gnu__ to expand to the empty string.

  On Unix systems, GNU m4 without the '-G' option will define the macro __unix__, otherwise the macro unix. Both will expand to the empty string.

Concept index

M

N

O

P

Q

R

S

T

U

Macro index

References are exclusively to the places where a built-in is introduced the first time. Names starting and ending with '__' have these characters removed in the index.

Short Contents

Table of Contents