GNU M4, version 1.4.8

A powerful macro processor
Edition 1.4.8, 20 November 2006

by René Seindal

This manual is for GNU M4 (version 1.4.8, 20 November 2006), a package containing an implementation of the m4 macro language.

Table of Contents

GNU `m4` is an implementation of the traditional UNIX macro processor. It is mostly SVR4 compatible, although it has some extensions (for example, handling more than 9 positional parameters to macros). `m4` also has builtin functions for including files, running shell commands, doing arithmetic, etc. Autoconf needs GNU `m4` for generating '`configure`' scripts, but not for running them.

GNU `m4` was originally written by René Seindal, with subsequent changes by François Pinard and other volunteers on the Internet. All names and email addresses can be found in the files '`m4-1.4.8/AUTHORS`' and '`m4-1.4.8/THANKS`' from the GNU M4 distribution.

This is release 1.4.8. It is now considered stable: future releases in the 1.4.x series are only meant to fix bugs, increase speed, or improve documentation. However...

An experimental feature, which would improve `m4` usefulness, allows for changing the syntax for what is a *word* in `m4`. You should use:

```
./configure --enable-changeword
```

if you want this feature compiled in. The current implementation slows down `m4` considerably and is hardly acceptable. In the future, `m4` 2.0 will come with a different set of new features that provide similar capabilities, but without the inefficiencies, so changeword will go away and *you should not count on it*.

1 Introduction and preliminaries

This first chapter explains what GNU `m4` is, where `m4` comes from, how to read and use this documentation, how to call the `m4` program, and how to report bugs about it. It concludes by giving tips for reading the remainder of the manual.

The following chapters then detail all the features of the `m4` language.

1.1 Introduction to `m4`

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

The `m4` macro processor is widely available on all UNIXes, and has been standardized by POSIX. Usually, only a small percentage of users are aware of its existence. However, those who find it often become committed users. The popularity of GNU Autoconf, which requires GNU `m4` for *generating* '`configure`' scripts, is an incentive for many to install it, while these people will not themselves program in `m4`. GNU `m4` is mostly compatible with the System V, Release 3 version, except for some minor differences. See Chapter 16 [Compatibility], page 87, for more details.

Some people find `m4` to be fairly addictive. They first use `m4` for simple problems, then take bigger and bigger challenges, learning how to write complex sets of `m4` macros along the way. Once really addicted, users pursue writing of sophisticated `m4` applications even to solve simple problems, devoting more time debugging their `m4` scripts than doing real work. Beware that `m4` may be dangerous for the health of compulsive programmers.

1.2 Historical references

`GPM` was an important ancestor of `m4`. See C. Stratchey: "A General Purpose Macro generator", Computer Journal 8,3 (1965), pp. 225 ff. `GPM` is also succinctly described into David Gries classic "Compiler Construction for Digital Computers".

The classic B. Kernighan and P.J. Plauger: "Software Tools", Addison-Wesley, Inc. (1976) describes and implements a Unix macro-processor language, which inspired Dennis Ritchie to write `m3`, a macro processor for the AP-3 minicomputer.

Kernighan and Ritchie then joined forces to develop the original `m4`, as described in "The M4 Macro Processor", Bell Laboratories (1977). It had only 21 builtin macros.

While `GPM` was more *pure*, `m4` is meant to deal with the true intricacies of real life: macros can be recognized without being pre-announced, skipping whitespace or end-of-lines is easier, more constructs are builtin instead of derived, etc.

Originally, the Kernighan and Plauger macro-processor, and then `m3`, formed the engine for the Rational FORTRAN preprocessor, that is, the `Ratfor` equivalent of `cpp`. Later, `m4` was used as a frontend for `Ratfor`, `C` and `Cobol`.

René Seindal released his implementation of `m4`, GNU `m4`, in 1990, with the aim of removing the artificial limitations in many of the traditional `m4` implementations, such as maximum line length, macro size, or number of macros.

The late Professor A. Dain Samples described and implemented a further evolution in the form of M5: "User's Guide to the M5 Macro Language: 2nd edition", Electronic Announcement on comp.compilers newsgroup (1992).

François Pinard took over maintenance of GNU m4 in 1992, until 1994 when he released GNU m4 1.4, which was the stable release for 10 years. It was at this time that GNU Autoconf decided to require GNU m4 as its underlying engine, since all other implementations of m4 had too many limitations.

More recently, in 2004, Paul Eggert released 1.4.1 and 1.4.2 which addressed some long standing bugs in the venerable 1.4 release. Then in 2005 Gary V. Vaughan collected together the many patches to GNU m4 1.4 that were floating around the net and released 1.4.3 and 1.4.4. And in 2006, Eric Blake joined the team and prepared patches for the release of 1.4.5, 1.4.6, 1.4.7, and 1.4.8.

Meanwhile, development has continued on new features for m4, such as dynamic module loading and additional builtins. When complete, GNU m4 2.0 will start a new series of releases.

1.3 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-m4@gnu.org. Please include the version number of m4 you are using. You can get this information with the command *m4 --version*. Also provide details about the platform you are executing on.

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.

1.4 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
```

The sequence '^D' in an example indicates the end of the input file. The majority of these examples are self-contained, and you can run them with similar results by invoking *m4*

`-d`. In fact, the testsuite that is bundled in the GNU M4 package consists of the examples in this document!

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.,

`example` (*`string`*, [*`count`* = ‘`1`’], [*`argument`*]...) [Composite]

> This is a sample prototype. There is not really a macro named `example`, but this documents that if there were, it would be a Composite macro, rather than a Builtin. It requires at least one argument, *string*. Remember that in `m4`, there must not be a space between the macro name and the opening parenthesis, unless it was intended to call the macro without any arguments. The brackets around *count* and *argument* show that these arguments are optional. If *count* is omitted, the macro behaves as if count were ‘`1`’, whereas if *argument* is omitted, the macro behaves as if it were the empty string. A blank argument is not the same as an omitted argument. For example, ‘`example(‘a’)`’, ‘`example(‘a’,‘1’)`’, and ‘`example(‘a’,‘1’,)`’ would behave identically with *count* set to ‘`1`’; while ‘`example(‘a’,)`’ and ‘`example(‘a’,‘’)`’ would explicitly pass the empty string for *count*. The ellipses (‘`...`’) show that the macro processes additional arguments after *argument*, rather than ignoring them.

All macro arguments in `m4` are strings, but some are given special interpretation, e.g., as numbers, file names, regular expressions, etc. The documentation for each macro will state how the parameters are interpreted, and what happens if the argument cannot be parsed according to the desired interpretation. Unless specified otherwise, a parameter specified to be a number is parsed as a decimal, even if the argument has leading zeros; and parsing the empty string as a number results in 0 rather than an error, although a warning will be issued.

This document consistently writes and uses *builtin*, without a hyphen, as if it were an English word. This is how the `builtin` primitive is spelled within `m4`.

2 Invoking m4

The format of the m4 command is:

m4 [*option*...] [*file*...]

All options begin with '-', or if long option names are used, with '--'. A long option name need not be written completely, any unambiguous prefix is sufficient. POSIX requires m4 to recognize arguments intermixed with files, even when POSIXLY_CORRECT is set in the environment. Most options take effect at startup regardless of their position, but some are documented below as taking effect after any files that occurred earlier in the command line. The argument '--' is a marker to denote the end of options.

With short options, options that do not take arguments may be combined into a single command line argument with subsequent options, options with mandatory arguments may be provided either as a single command line argument or as two arguments, and options with optional arguments must be provided as a single argument. In other words, *m4 -QPDfoo -d* a *-d+f* is equivalent to *m4 -Q -P -D foo -d -d+f -- ./*a, although the latter form is considered canonical.

With long options, options with mandatory arguments may be provided with an equal sign ('=') in a single argument, or as two arguments, and options with optional arguments must be provided as a single argument. In other words, *m4 --def foo --debug* a is equivalent to *m4 --define=foo --debug= -- ./*a, although the latter form is considered canonical (not to mention more robust, in case a future version of m4 introduces an option named '--default').

m4 understands the following options, grouped by functionality.

2.1 Command line options for operation modes

Several options control the overall operation of m4:

--help
: Print a help summary on standard output, then immediately exit m4 without reading any input files or performing any other actions.

--version
: Print the version number of the program on standard output, then immediately exit m4 without reading any input files or performing any other actions.

-E
--fatal-warnings
: Stop execution and exit m4 once the first warning has been issued, considering all of them to be fatal.

-i
--interactive
-e
: Makes this invocation of m4 interactive. This means that all output will be unbuffered, and interrupts will be ignored. The spelling '-e' exists for compatibility with other m4 implementations, and issues a warning because it may be withdrawn in a future version of GNU M4.

`-P`
`--prefix-builtins`
: Internally modify *all* builtin macro names so they all start with the prefix '`m4_`'. For example, using this option, one should write '`m4_define`' instead of '`define`', and '`m4___file__`' instead of '`__file__`'. This option has no effect if '`-R`' is also specified.

`-Q`
`--quiet`
`--silent`
: Suppress warnings, such as missing or superfluous arguments in macro calls, or treating the empty string as zero.

`-W` *REGEXP*
`--word-regexp=`*REGEXP*
: Use *REGEXP* as an alternative syntax for macro names. This experimental option will not be present in all GNU `m4` implementations (see Section 8.4 [Changeword], page 50).

2.2 Command line options for preprocessor features

Several options allow `m4` to behave more like a preprocessor. Macro definitions and deletions can be made on the command line, the search path can be altered, and the output file can track where the input came from. These features occur with the following options:

`-D` *NAME*[`=`*VALUE*]
`--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. This option may be given more than once; order with respect to file names is significant, and redefining the same *NAME* loses the previous value.

`-I` *DIRECTORY*
`--include=`*DIRECTORY*
: Make `m4` search *DIRECTORY* for included files that are not found in the current working directory. See Section 9.2 [Search Path], page 56, for more details. This option may be given more than once.

`-s`
`--synclines`
: Generate synchronization lines, for use by the C preprocessor or other similar tools. Order is significant with respect to file names. This option 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 directives of the form '`#line` *linenum* `"`*file*`"`', which are inserted as needed into the middle of the output. Such directives mean that the following line originated or was expanded from the contents of input file *file* at line *linenum*. The '`"`*file*`"`' part is often omitted when the file name did not change from the previous directive.

Synchronization directives are always given on complete lines by themselves. When a synchronization discrepancy occurs in the middle of an output line, the associated synchronization directive is delayed until the beginning of the next generated line.

`-U` *NAME*
`--undefine=`*NAME*
: This deletes any predefined meaning *NAME* might have. Obviously, only predefined macros can be deleted in this way. This option may be given more than once; undefining a *NAME* that does not have a definition is silently ignored. Order is significant with respect to file names.

2.3 Command line options for limits control

There are some limits within `m4` that can be tuned. For compatibility, `m4` also accepts some options that control limits in other implementations, but which are automatically unbounded (limited only by your hardware and operating system constraints) in GNU `m4`.

`-G`
`--traditional`
: Suppress all the extensions made in this implementation, compared to the System V version. See Chapter 16 [Compatibility], page 87, for a list of these.

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

`-L` *NUM*
`--nesting-limit=`*NUM*
: Artificially limit the nesting of macro calls to *NUM* levels, stopping program execution if this limit is ever exceeded. When not specified, nesting is limited to 1024 levels. A value of zero means unlimited; but then heavily nested code could potentially cause a stack overflow.

  The precise effect of this option might be more correctly associated with textual nesting than dynamic recursion. It has been useful when some complex `m4` input was generated by mechanical means. Most users would never need this option. If shown to be obtrusive, this option (which is still experimental) might well disappear.

  This option does *not* have the ability to break endless rescanning loops, since these do not necessarily consume much memory or stack space. Through clever usage of rescanning loops, one can request complex, time-consuming computations from `m4` with useful results. Putting limitations in this area would break `m4` power. There are many pathological cases: ‘`define(‘a’, ‘a’)a`’ is only the simplest example (but see Chapter 16 [Compatibility], page 87). Expecting GNU `m4` to detect these would be a little like expecting a compiler system to detect and diagnose endless loops: it is a quite *hard* problem in general, if not undecidable!

`-B` *NUM*
`-S` *NUM*
`-T` *NUM*
: These options are present for compatibility with System V `m4`, but do nothing in this implementation. They may disappear in future releases, and issue a warning to that effect.

`-N` *NUM*
`--diversions=`*NUM*
: These options are present only for compatibility with previous versions of GNU `m4`, and were controlling the number of possible diversions which could be used at the same time. They do nothing, because there is no fixed limit anymore. They may disappear in future releases, and issue a warning to that effect.

2.4 Command line options for frozen state

GNU `m4` comes with a feature of freezing internal state (see Chapter 15 [Frozen files], page 83). This can be used to speed up `m4` execution when reusing a common initialization script.

`-F` *FILE*
`--freeze-state=`*FILE*
: Once execution is finished, write out the frozen state on the specified *FILE*. It is conventional, but not required, for *FILE* to end in '`.m4f`'.

`-R` *FILE*
`--reload-state=`*FILE*
: Before execution starts, recover the internal state from the specified frozen *FILE*. The options '`-D`', '`-U`', and '`-t`' take effect after state is reloaded, but before the input files are read.

2.5 Command line options for debugging

Finally, there are several options for aiding in debugging `m4` scripts.

`-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 7.3 [Debug Levels], page 41, for more details on the format and meaning of *FLAGS*. If omitted, *FLAGS* defaults to '`aeq`'.

`--debugfile=`*FILE*
`-o` *FILE*
`--error-output=`*FILE*
: Redirect `dumpdef` output, debug messages, and trace output to the named *FILE*. Warnings, error messages, and `errprint` output are still printed to standard error. If unspecified, debug output goes to standard error; if empty, debug output is discarded. See Section 7.4 [Debug Output], page 42, for more details. The spellings '`-o`' and '`--error-output`' are misleading and inconsistent with other GNU tools; for now they are silently accepted as synonyms of

'--debugfile', but in a future version of M4, using them will cause a warning to be issued.

-l *NUM*
--arglength=*NUM*
: Restrict the size of the output generated by macro tracing to *NUM* characters per trace line. If unspecified or zero, output is unlimited. See Section 7.3 [Debug Levels], page 41, for more details.

-t *NAME*
--trace=*NAME*
: This enables tracing for the macro *NAME*, at any point where it is defined. *NAME* need not be defined when this option is given. This option may be given more than once, and order is significant with respect to file names. See Section 7.2 [Trace], page 39, for more details.

2.6 Specifying input files on the command line

The remaining arguments on the command line are taken to be input file names. If no names are present, standard input is read. A file name of '-' is taken to mean standard input. It is conventional, but not required, for input files to end in '.m4'.

The input files are read in the sequence given. Standard input can be read more than once, so the file name '-' may appear multiple times on the command line; this makes a difference when input is from a terminal or other special file type. It is an error if an input file ends in the middle of argument collection, a comment, or a quoted string.

The options '--define' ('-D'), '--undefine' ('-U'), '--synclines' ('-s'), and '--trace' ('-t') only take effect after processing input from any file names that occur earlier on the command line.

If none of the input files invoked m4exit (see Section 14.3 [M4exit], page 81), the exit status of m4 will be 0 for success, 1 for general failure (such as problems with reading an input file), and 63 for version mismatch (see Section 15.1 [Using frozen files], page 83).

If you need to read a file whose name starts with a '-', you can specify it as './-file', or use '--' to mark the end of options.

3 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. GNU `m4` does not yet understand locales; all operations are byte-oriented rather than character-oriented. However, `m4` is eight-bit clean, so you can use non-ASCII characters in quoted strings (see Section 8.2 [Changequote], page 46), comments (see Section 8.3 [Changecom], page 48), and macro names (see Section 5.7 [Indir], page 27), with the exception of the NUL character (the zero byte ‘`''\0''`’).

3.1 Macro names

A name is any sequence of letters, digits, and the character ‘`_`’ (underscore), where the first character is not a digit. `m4` will use the longest such sequence found in the input. If a name has a macro definition, it will be subject to macro expansion (see Chapter 4 [Macros], page 15). Names are case-sensitive.

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

3.2 Quoting input to `m4`

A quoted string is a sequence of characters surrounded by quote strings, defaulting to ‘`‘`’ and ‘`’`’, where the nested begin and end quotes within the string are balanced. The value of a string token is the text, with one level of quotes stripped off. Thus

```
‘’
⇒
```

is the empty string, and double-quoting turns into single-quoting.

```
‘‘quoted’’
⇒‘quoted’
```

The quote characters can be changed at any time, using the builtin macro `changequote`. See Section 8.2 [Changequote], page 46, for more information.

3.3 Comments in `m4` input

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 commenting effect of the begin-comment string can be inhibited by quoting it.

```
‘quoted text’ # ‘commented text’
⇒quoted text # ‘commented text’
‘quoting inhibits’ ‘#’ ‘comments’
⇒quoting inhibits # comments
```

The comment delimiters can be changed to any string at any time, using the builtin macro `changecom`. See Section 8.3 [Changecom], page 48, for more information.

3.4 Other kinds of input tokens

Any character, that is neither a part of a name, nor of a quoted string, nor a comment, is a token by itself. When not in the context of macro expansion, all of these tokens are just copied to output. However, during macro expansion, whitespace characters (space, tab, newline, formfeed, carriage return, vertical tab), parentheses (‘`(`’ and ‘`)`’), comma (‘`,`’), and dollar (‘`$`’) have additional roles, explained later.

3.5 How `m4` copies input to output

As `m4` reads the input token by token, it will copy each token directly to the output immediately.

The exception is when it finds a word with a macro definition. In that case `m4` will calculate the macro's expansion, possibly reading more input to get the arguments. It then inserts the expansion in front of the remaining input. In other words, the resulting text from a macro call will be read and parsed into tokens again.

`m4` expands a macro as soon as possible. If it finds a macro call when collecting the arguments to another, it will expand the second call first. For a running example, examine how `m4` handles this input:

```
format(‘Result is %d’, eval(‘2**15’))
```

First, `m4` sees that the token ‘`format`’ is a macro name, so it collects the tokens ‘`(`’, ‘`‘Result is %d’`’, ‘`,`’, and ‘ ’, before encountering another potential macro. Sure enough, ‘`eval`’ is a macro name, so the nested argument collection picks up ‘`(`’, ‘`‘2**15’`’, and ‘`)`’, invoking the eval macro with the lone argument of ‘`2**15`’. The expansion of ‘`eval(2**15)`’ is ‘`32768`’, which is then rescanned as the five tokens ‘`3`’, ‘`2`’, ‘`7`’, ‘`6`’, and ‘`8`’; and combined with the next ‘`)`’, the format macro now has all its arguments, as if the user had typed:

```
format(‘Result is %d’, 32768)
```

The format macro expands to ‘`Result is 32768`’, and we have another round of scanning for the tokens ‘`Result`’, ‘ ’, ‘`is`’, ‘ ’, ‘`3`’, ‘`2`’, ‘`7`’, ‘`6`’, and ‘`8`’. None of these are macros, so the final output is

```
⇒Result is 32768
```

The order in which `m4` expands the macros can be explored using the trace facilities of GNU `m4` (see Section 7.2 [Trace], page 39).

This process continues until there are no more macro calls to expand and all the input has been consumed.

4 How to invoke macros

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

4.1 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.

4.2 Preventing macro invocation

An innovation of the `m4` language, compared to some of its predecessors (like Stratchey's `GPM`, for example), is the ability to recognize macro calls without resorting to any special, prefixed invocation character. While generally useful, this feature might sometimes be the source of spurious, unwanted macro calls. So, GNU `m4` offers several mechanisms or techniques for inhibiting the recognition of names as macro calls.

First of all, many builtin macros cannot meaningfully be called without arguments. For any of these macros, whenever an opening parenthesis does not immediately follow their name, the builtin macro call is not triggered. This solves the most usual cases, like for '`include`' or '`eval`'. Later in this document, the sentence "This macro is recognized only with parameters" refers to this specific provision.

There is also a command line option ('`--prefix-builtins`', or '`-P`', see Section 2.1 [Invoking m4], page 7) that renames all builtin macros with a prefix of '`m4_`' at startup. The option has no effect whatsoever on user defined macros. For example, with this option, one has to write `m4_dnl` and even `m4_m4exit`. It also has no effect on whether a macro requires parameters.

Another alternative is to redefine problematic macros to a name less likely to cause conflicts, See Chapter 5 [Definitions], page 19.

If your version of GNU `m4` has the `changeword` feature compiled in, it offers far more flexibility in specifying the syntax of macro names, both builtin or user-defined. See Section 8.4 [Changeword], page 50, for more information on this experimental feature.

Of course, the simplest way to prevent a name from being interpreted as a call to an existing macro is to quote it. The remainder of this section studies a little more deeply how quoting affects macro invocation, and how quoting can be used to inhibit macro invocation.

Even if quoting is usually done over the whole macro name, it can also be done over only a few characters of this name (provided, of course, that the unquoted portions are not also a macro). It is also possible to quote the empty string, but this works only *inside* the name. For example:

```
‘divert’
⇒divert
‘d’ivert
⇒divert
di‘ver’t
⇒divert
div‘’ert
⇒divert
```

all yield the string ‘`divert`’. While in both:

```
‘’divert
⇒
divert‘’
⇒
```

the `divert` builtin macro will be called, which expands to the empty string.

The output of macro evaluations is always rescanned. The following example would yield the string ‘`de`’, exactly as if `m4` has been given ‘`substr(‘abcde’, ‘3’, ‘2’)`’ as input:

```
define(‘x’, ‘substr(ab’)
⇒
define(‘y’, ‘cde, ‘3’, ‘2’)’)
⇒
x‘’y
⇒de
```

Unquoted strings on either side of a quoted string are subject to being recognized as macro names. In the following example, quoting the empty string allows for the second `macro` to be recognized as such:

```
define(‘macro’, ‘m’)
⇒
macro(‘m’)macro
⇒mmacro
macro(‘m’)‘’macro
⇒mm
```

Quoting may prevent recognizing as a macro name the concatenation of a macro expansion with the surrounding characters. In this example:

```
define(‘macro’, ‘di$1’)
⇒
macro(‘v’)‘ert’
⇒divert
macro(‘v’)ert
⇒
```

the input will produce the string ‘`divert`’. When the quotes were removed, the `divert` builtin was called instead.

4.3 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. However, some builtins are documented to behave differently for a missing optional argument than for an explicit empty string. If there are too many arguments, the excess arguments are ignored. Unquoted leading whitespace is stripped off all arguments, but whitespace generated by a macro expansion or occuring after a macro that expanded to an empty string remains intact. Whitespace includes space, tab, newline, carriage return, vertical tab, and formfeed.

```
define(‘macro’, ‘$1’)
⇒
macro( unquoted leading space lost)
⇒unquoted leading space lost
macro(‘ quoted leading space kept’)
⇒ quoted leading space kept
macro(
 divert ‘unquoted space kept after expansion’)
⇒ unquoted space kept after expansion
macro(macro(‘
’)‘whitespace from expansion kept’)
⇒
⇒whitespace from expansion kept
macro(‘unquoted trailing whitespace kept’
)
⇒unquoted trailing whitespace kept
⇒
```

Normally `m4` will issue warnings if a builtin macro is called with an inappropriate number of arguments, but it can be suppressed with the ‘`--quiet`’ command line option (or ‘`--silent`’, or ‘`-Q`’, see Section 2.1 [Invoking m4], page 7). 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`’. To understand why the first argument contains whitespace, remember that unquoted leading whitespace is never part of an argument, but trailing whitespace always is.

It is possible for a macro’s definition to change during argument collection, in which case the expansion uses the definition that was in effect at the time the opening ‘`(`’ was seen.

```
define(‘f’, ‘1’)
⇒
f(define(‘f’, ‘2’))
⇒1
f
```

```
⇒2
```

It is an error if the end of file occurs while collecting arguments.

```
hello world
⇒hello world
define(
^D
error m4:stdin:2: ERROR: end of file in argument list
```

4.4 On Quoting Arguments to macros

Each argument has unquoted leading 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 ‘`() (() (`’. Commas separate arguments, except when they occur inside quotes, comments, or unquoted parentheses. See Section 5.3 [Pseudo Arguments], page 21, for examples.

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 or convenient 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. For consistency, this manual follows the rule of thumb that each layer of parentheses introduces another layer of single quoting, except when showing the consequences of quoting rules. This is done even when the quoted string cannot be a macro, such as with integers when you have not changed the syntax via `changeword` (see Section 8.4 [Changeword], page 50).

4.5 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`’.

5 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, and bring back the original value at a later time.

5.1 Defining a macro

The normal way to define or redefine macros is to use the builtin `define`:

`define` (*name*, [*expansion*]) [Builtin]

Defines *name* to expand to *expansion*. If *expansion* is not given, it is taken to be empty.

The expansion of `define` is void. The macro `define` is recognized only with parameters.

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 8.1 [Dnl], page 45, for details.

The first argument to `define` should be quoted; otherwise, if the macro is already defined, you will be defining a different macro. This example shows the problems with underquoting, since we did not want to redefine `one`:

```
define(foo, one)
⇒
define(foo, two)
⇒
one
⇒two
```

GNU `m4` normally replaces only the *topmost* definition of a macro if it has several definitions from `pushdef` (see Section 5.6 [Pushdef], page 25). Some other implementations of `m4` replace all definitions of a macro with `define`. See Section 16.2 [Incompatibilities], page 88, for more details.

As a GNU extension, the first argument to `define` does not have to be a simple word. It can be any text string, even the empty string. A macro with a non-standard name cannot be invoked in the normal way, as the name is not recognized. It can only be referenced by the builtins `Indir` (see Section 5.7 [Indir], page 27) and `Defn` (see Section 5.5 [Defn], page 24).

Arrays and associative arrays can be simulated by using this trick.

```
define(‘array’, ‘defn(format(‘‘array[%d]’’, ‘$1’))’)
⇒
define(‘array_set’, ‘define(format(‘‘array[%d]’’, ‘$1’), ‘$2’)’)
```

```
⇒
array_set(‘4’, ‘array element no. 4’)
⇒
array_set(‘17’, ‘array element no. 17’)
⇒
array(‘4’)
⇒array element no. 4
array(eval(‘10 + 7’))
⇒array element no. 17
```

Change the `%d` to `%s` and it is an associative array.

5.2 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. Replacement of arguments happens before rescanning, regardless of how many nesting levels of quoting appear in the expansion. Here is an 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
```

See Section 4.4 [Quoting Arguments], page 18, for an explanation of the double quotes. (You should try and improve this example so that clients of `exch` do not have to double quote; or see Section 17.1 [Answers], page 93).

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 zeroth 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.

5.3 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
nargs(‘commas can be quoted, like this’)
⇒1
nargs(arg1#inside comments, commas do not separate arguments
still arg1)
⇒1
nargs((unquoted parentheses, like this, group arguments))
⇒1
```

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..
echo1(‘foo’)
⇒This is macro foo.
echo2(foo)
⇒This is macro foo.
echo2(‘foo’)
⇒foo
```

See Section 7.2 [Trace], page 39, if you do not understand this. As another example of the difference, remember that comments encountered in arguments are passed untouched to the macro, and that quoting disables comments.

```
define(‘echo1’, ‘$*’)
⇒
define(‘echo2’, ‘$@’)
⇒
define(‘foo’, ‘bar’)
⇒
echo1(#foo’foo
foo)
⇒#foo’foo
⇒bar
echo2(#foo’foo
foo)
⇒#foobar
⇒bar’
```

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 ‘`$12`’, the judicious use of nested quoting can put a safe character between the `$` and the next character, relying on the rescanning to remove the nested quote. This will prevent `m4` from interpreting the `$` sign as a reference to an argument.

```
define(‘foo’, ‘no nested quote: $1’)
⇒
foo(‘arg’)
⇒no nested quote: arg
define(‘foo’, ‘nested quote around $: ‘$’1’)
⇒
foo(‘arg’)
⇒nested quote around $: $1
define(‘foo’, ‘nested empty quote after $: $‘’1’)
⇒
```

```
foo(‘arg’)
⇒nested empty quote after $: $1
define(‘foo’, ‘nested quote around next character: $‘1’’)
⇒
foo(‘arg’)
⇒nested quote around next character: $1
define(‘foo’, ‘nested quote around both: ‘$1’’)
⇒
foo(‘arg’)
⇒nested quote around both: arg
```

5.4 Deleting a macro

A macro definition can be removed with `undefine`:

`undefine` (*name*...) [Builtin]

For each argument, remove the macro *name*. The macro names must necessarily be quoted, since they will be expanded otherwise.

The expansion of `undefine` is void. The macro `undefine` is recognized only with parameters.

```
foo bar blah
⇒foo bar blah
define(‘foo’, ‘some’)define(‘bar’, ‘other’)define(‘blah’, ‘text’)
⇒
foo bar blah
⇒some other text
undefine(‘foo’)
⇒
foo bar blah
⇒foo other text
undefine(‘bar’, ‘blah’)
⇒
foo bar blah
⇒foo bar blah
```

Undefining a macro inside that macro's expansion is safe; the macro still expands to the definition that was in effect at the ‘(’.

```
define(‘f’, ‘‘$0’:$1’)
⇒
f(f(f(undefine(‘f’)‘hello world’)))
⇒f:f:f:hello world
f(‘bye’)
⇒f(bye)
```

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

5.5 Renaming macros

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

`defn (`*name*`)` [Builtin]

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 builtin, the expansion is a special token, which points to the builtin's internal definition. This token is only meaningful as the second argument to `define` (and `pushdef`), and is silently converted to an empty string in most other contexts.

The macro `defn` is recognized only with parameters.

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 builtin macros. Even if the original macro is removed, the other name can still be used to access the definition.

The fact that macro definitions can be transferred also explains why you should use `$0`, rather than retyping a macro's name in its definition:

```
define(‘foo’, ‘This is ‘$0’’)
⇒
define(‘bar’, defn(‘foo’))
⇒
bar
⇒This is bar
```

Macros used as string variables should be referred through `defn`, to avoid unwanted expansion of the text:

```
define(‘string’, ‘The macro dnl is very useful
’)
⇒
string
⇒The macro
defn(‘string’)
⇒The macro dnl is very useful
⇒
```

However, it is important to remember that `m4` rescanning is purely textual. If an unbalanced end-quote string occurs in a macro definition, the rescan will see that embedded quote as the termination of the quoted string, and the remainder of the macro's definition

will be rescanned unquoted. Thus it is a good idea to avoid unbalanced end-quotes in macro definitions or arguments to macros.

```
define(‘foo’, a’a)
⇒
define(‘a’, ‘A’)
⇒
define(‘echo’, ‘$@’)
⇒
foo
⇒A’A
defn(‘foo’)
⇒aA’
echo(foo)
⇒AA’
```

Using `defn` to generate special tokens for builtin macros outside of expected contexts can sometimes trigger warnings. But most of the time, such tokens are silently converted to the empty string.

```
defn(‘defn’)
⇒
define(defn(‘divnum’), ‘cannot redefine a builtin token’)
error m4:stdin:2: Warning: define: invalid macro name ignored
⇒
divnum
⇒0
```

5.6 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 builtins `pushdef` and `popdef`:

`pushdef` (*name*, [*expansion*]) [Builtin]
`popdef` (*name*...) [Builtin]

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` behaves like `undefine`.

The expansion of both `pushdef` and `popdef` is void. The macros `pushdef` and `popdef` are recognized only with parameters.

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

```
pushdef(‘foo’, ‘Expansion two.’)
⇒
foo
⇒Expansion two.
pushdef(‘foo’, ‘Expansion three.’)
⇒
pushdef(‘foo’, ‘Expansion four.’)
⇒
popdef(‘foo’)
⇒
foo
⇒Expansion three.
popdef(‘foo’, ‘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
```

Local variables within macros are made with `pushdef` and `popdef`. At the start of the macro a new definition is pushed, within the macro it is manipulated and at the end it is popped, revealing the former definition.

It is possible to temporarily redefine a builtin with `pushdef` and `defn`.

5.7 Indirect call of macros

Any macro can be called indirectly with `indir`:

`indir` (*name*, [*args...*]) [Builtin]

Results in a call to the macro *name*, which is passed the rest of the arguments *args*. If *name* is not defined, an error message is printed, and the expansion is void.

The macro `indir` is recognized only with parameters.

This can be used to call macros with computed or "invalid" 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 builtin `indir`.

One other point to observe is that argument collection occurs before `indir` invokes *name*, so if argument collection changes the value of *name*, that will be reflected in the final expansion. This is different than the behavior when invoking macros directly, where the definition that was in effect before argument collection is used.

```
define(‘f’, ‘1’)
⇒
f(define(‘f’, ‘2’))
⇒1
indir(‘f’, define(‘f’, ‘3’))
⇒3
indir(‘f’, undefine(‘f’))
error m4:stdin:4: undefined macro ‘f’
⇒
```

When handed the result of `defn` (see Section 5.5 [Defn], page 24) as one of its arguments, `indir` defers to the invoked *name* for whether a token representing a builtin is recognized or flattened to the empty string.

```
indir(defn(‘defn’), ‘divnum’)
error m4:stdin:1: Warning: indir: invalid macro name ignored
⇒
indir(‘define’, defn(‘defn’), ‘divnum’)
error m4:stdin:2: Warning: define: invalid macro name ignored
⇒
indir(‘define’, ‘foo’, defn(‘divnum’))
⇒
foo
⇒0
indir(‘divert’, defn(‘foo’))
```

```
error m4:stdin:5: empty string treated as 0 in builtin ‘divert’
⇒
```

5.8 Indirect call of builtins

Builtin macros can be called indirectly with `builtin`:

`builtin` (*name*, [*args...*]) [Builtin]

Results in a call to the builtin *name*, which is passed the rest of the arguments *args*. If *name* does not name a builtin, an error message is printed, and the expansion is void.

The macro `builtin` is recognized only with parameters.

This can be used even if *name* has been given another definition that has covered the original, or been undefined so that no macro maps to the builtin.

```
pushdef(‘define’, ‘hidden’)
⇒
undefine(‘undefine’)
⇒
define(‘foo’, ‘bar’)
⇒hidden
foo
⇒foo
builtin(‘define’, ‘foo’, defn(‘divnum’))
⇒
foo
⇒0
builtin(‘define’, ‘foo’, ‘BAR’)
⇒
foo
⇒BAR
undefine(‘foo’)
⇒undefine(foo)
foo
⇒BAR
builtin(‘undefine’, ‘foo’)
⇒
foo
⇒foo
```

The *name* argument only matches the original name of the builtin, even when the ‘`--prefix-builtins`’ option (or ‘`-P`’, see Section 2.1 [Invoking m4], page 7) is in effect. This is different from `indir`, which only tracks current macro names.

Note that `indir` and `builtin` can be used to invoke builtins without arguments, even when they normally require parameters to be recognized; but it will provoke a warning, and result in a void expansion.

```
builtin
⇒builtin
```

```
builtin()
error m4:stdin:2: undefined builtin ‘’
⇒
builtin(‘builtin’)
error m4:stdin:3: Warning: too few arguments to builtin ‘builtin’
⇒
builtin(‘builtin’,)
error m4:stdin:4: undefined builtin ‘’
⇒
```

6 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. For that, 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.

6.1 Testing if a macro is defined

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

`ifdef` (*name*, *string-1*, [*string-2*]) [Builtin]

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).

The macro `ifdef` is recognized only with parameters.

```
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
ifdef(‘no_such_macro’, ‘yes’, ‘no’, ‘extra argument’)
error m4:stdin:4: Warning: excess arguments to builtin ‘ifdef’ ignored
⇒no
```

6.2 If-else construct, or multibranch

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*) [Builtin]
`ifelse` (*string-1*, *string-2*, *equal*, [*not-equal*]) [Builtin]
`ifelse` (*string-1*, *string-2*, *equal-1*, *string-3*, *string-4*, *equal-2*, ...) [Builtin]

Used with only one argument, the `ifelse` simply discards it and produces no output.

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*. A final fifth argument is ignored, after triggering a warning.

If called with six or more arguments, and *string-1* and *string-2* are equal, `ifelse` expands into *equal-1*, otherwise the first three arguments are discarded and the processing starts again.

The macro `ifelse` is recognized only with parameters.

Using only one argument 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.

```
ifelse(‘some comments’)
⇒
ifelse(‘foo’, ‘bar’)
error m4:stdin:2: Warning: too few arguments to builtin ‘ifelse’
⇒
```

Using three or four arguments provides decision points.

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

Notice how the first argument was used unquoted; it is common to compare the expansion of a macro with a string. With this macro, you can now reproduce the behavior of many of the builtins, where the macro is recognized only with arguments.

```
define(‘foo’, ‘ifelse(‘$#’, ‘0’, ‘‘$0’’, ‘arguments:$#’)’)
⇒
foo
⇒foo
foo()
⇒arguments:1
foo(‘a’, ‘b’, ‘c’)
⇒arguments:3
```

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-1*, otherwise the procedure is repeated with the first three arguments discarded. This calls for an example:

```
ifelse(‘foo’, ‘bar’, ‘third’, ‘gnu’, ‘gnats’)
error m4:stdin:1: Warning: excess arguments to builtin ‘ifelse’ ignored
⇒gnu
ifelse(‘foo’, ‘bar’, ‘third’, ‘gnu’, ‘gnats’, ‘sixth’)
⇒
ifelse(‘foo’, ‘bar’, ‘third’, ‘gnu’, ‘gnats’, ‘sixth’, ‘seventh’)
⇒seventh
ifelse(‘foo’, ‘bar’, ‘3’, ‘gnu’, ‘gnats’, ‘6’, ‘7’, ‘8’)
error m4:stdin:4: Warning: excess arguments to builtin ‘ifelse’ ignored
⇒7
```

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.

6.3 Recursion in m4

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 builtin macro, shift, which can, among other things, be used for iterating through the actual arguments to a macro:

shift (*arg1*, ...) [Builtin]

Takes any number of arguments, and expands to all its arguments except *arg1*, separated by commas, with each argument quoted.

The macro shift is recognized only with parameters.

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

An example of the use of shift is this macro:

reverse (...) [Composite]

Takes any number of arguments, and reverses their order.

It is implemented as:

```
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. It also shows that shift is usually used with ‘$@’. Sometimes, a recursive algorithm requires adding quotes to each element:

quote (...) [Composite]
dquote (...) [Composite]
dquote_elt (...) [Composite]

Takes any number of arguments, and adds quoting. With quote, only one level of quoting is added, effectively removing whitespace after commas and turning multiple arguments into a single string. With dquote, two levels of quoting are added, one around each element, and one around the list. And with dquote_elt, two levels of quoting are added around each element.

An actual implementation of these three macros is distributed as '`m4-1.4.8/examples/quote.m4`' in this package. First, let's examine their usage:

```
include(‘quote.m4’)
⇒
-quote-dquote-dquote_elt-
⇒----
-quote()-dquote()-dquote_elt()-
⇒--‘’-‘’-
-quote(‘1’)-dquote(‘1’)-dquote_elt(‘1’)-
⇒-1-‘1’-‘1’-
-quote(‘1’, ‘2’)-dquote(‘1’, ‘2’)-dquote_elt(‘1’, ‘2’)-
⇒-1,2-‘1’,‘2’-‘1’,‘2’-
define(‘n’, ‘$#’)dnl
-n(quote(‘1’, ‘2’))-n(dquote(‘1’, ‘2’))-n(dquote_elt(‘1’, ‘2’))-
⇒-1-1-2-
dquote(dquote_elt(‘1’, ‘2’))
⇒‘‘1’’,‘‘2’’
dquote_elt(dquote(‘1’, ‘2’))
⇒‘‘1’,‘2’’
```

The last two lines show that when given two arguments, `dquote` results in one string, while `dquote_elt` results in two. Now, examine the implementation. Note that `quote` and `dquote_elt` make decisions based on their number of arguments, so that when called without arguments, they result in nothing instead of a quoted empty string; this is so that it is possible to distinquish between no arguments and an empty first argument. `dquote`, on the other hand, results in a string no matter what, since it is still possible to tell whether it was invoked without arguments based on the resulting string.

```
undivert(‘quote.m4’)dnl
⇒divert(‘-1’)
⇒# quote(args) - convert args to single-quoted string
⇒define(‘quote’, ‘ifelse(‘$#’, ‘0’, ‘’, ‘‘$*’’)’)
⇒# dquote(args) - convert args to quoted list of quoted strings
⇒define(‘dquote’, ‘‘$@’’)
⇒# dquote_elt(args) - convert args to list of double-quoted strings
⇒define(‘dquote_elt’, ‘ifelse(‘$#’, ‘0’, ‘’, ‘$#’, ‘1’, ‘‘‘$1’’’,
⇒                             ‘‘‘$1’’,$0(shift($@))’)’)
⇒divert‘’dnl
```

6.4 Iteration by counting

Here is an example of a loop macro that implements a simple for loop.

`forloop` (*iterator*, *start*, *end*, *text*) [Composite]

Takes the name in *iterator*, which must be a valid macro name, and successively assign it each integer value from *start* to *end*, inclusive. For each assignment to *iterator*, append *text* to the expansion of the `forloop`. *text* may refer to *iterator*. Any definition of *iterator* prior to this invocation is restored.

It can, for example, be used for simple counting:

```
include(‘forloop.m4’)
⇒
forloop(‘i’, ‘1’, ‘8’, ‘i ’)
⇒1 2 3 4 5 6 7 8
```

For-loops can be nested, like:

```
include(‘forloop.m4’)
⇒
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 the iterator has reached the final value. If it has not finished, it increments the iterator (using the predefined macro `incr`, see Section 12.1 [Incr], page 69), and recurses.

Here is an actual implementation of `forloop`, distributed as ‘`m4-1.4.8/examples/forloop.m4`’ in this package:

```
undivert(‘forloop.m4’)dnl
⇒divert(‘-1’)
⇒# forloop(var, from, to, stmt) - simple version
⇒define(‘forloop’, ‘pushdef(‘$1’, ‘$2’)_forloop($@)popdef(‘$1’)’)
⇒define(‘_forloop’,
⇒       ‘$4‘’ifelse($1, ‘$3’, ‘’, ‘define(‘$1’, incr($1))$0($@)’)’)
⇒divert‘’dnl
```

Notice the careful use of quotes. Certain macro arguments are left unquoted, each for its own reason. Try to find out *why* these arguments are left unquoted, and see what happens if they are quoted. (As presented, these two macros are useful but not very robust for general use. They lack even basic error handling for cases like *start* less than *end*, *end* not numeric, or *iterator* not being a macro name. See if you can improve these macros; or see Section 17.2 [Answers], page 93).

6.5 Iteration by list contents

Here is an example of a loop macro that implements list iteration.

`foreach` (*iterator*, *paren-list*, *text*) [Composite]
`foreachq` (*iterator*, *quote-list*, *text*) [Composite]

Takes the name in *iterator*, which must be a valid macro name, and successively assign it each value from *paren-list* or *quote-list*. In `foreach`, *paren-list* is a comma-separated list of elements contained in parentheses. In `foreachq`, *quote-list* is a comma-separated list of elements contained in a quoted string. For each assignment

to *iterator*, append *text* to the overall expansion. *text* may refer to *iterator*. Any definition of *iterator* prior to this invocation is restored.

As an example, this displays each word in a list inside of a sentence, using an implementation of `foreach` distributed as '`m4-1.4.8/examples/foreach.m4`', and `foreachq` in '`m4-1.4.8/examples/foreachq.m4`'.

```
include(‘foreach.m4’)
⇒
foreach(‘x’, (foo, bar, foobar), ‘Word was: x
’)dnl
⇒Word was: foo
⇒Word was: bar
⇒Word was: foobar
include(‘foreachq.m4’)
⇒
foreachq(‘x’, ‘foo, bar, foobar’, ‘Word was: x
’)dnl
⇒Word was: foo
⇒Word was: bar
⇒Word was: foobar
```

It is possible to be more complex; each element of the *paren-list* or *quote-list* can itself be a list, to pass as further arguments to a helper macro. This example generates a shell case statement:

```
include(‘foreach.m4’)
⇒
define(‘_case’, ‘  $1)
    $2=" $1";;
’)dnl
define(‘_cat’, ‘$1$2’)dnl
case $‘’1 in
⇒case $1 in
foreach(‘x’, ‘(‘(‘a’, ‘vara’)’, ‘(‘b’, ‘varb’)’, ‘(‘c’, ‘varc’)’)’,
        ‘_cat(‘_case’, x)’)dnl
⇒  a)
⇒    vara=" a";;
⇒  b)
⇒    varb=" b";;
⇒  c)
⇒    varc=" c";;
esac
⇒esac
```

The implementation of the `foreach` macro is a bit more involved; it is a wrapper around two helper macros. First, `_arg1` is needed to grab the first element of a list. Second, `_foreach` implements the recursion, successively walking through the original list. Here is a simple implementation of `foreach`:

```
undivert(‘foreach.m4’)dnl
```

```
⇒divert(‘-1’)
⇒# foreach(x, (item_1, item_2, ..., item_n), stmt)
⇒#   parenthesized list, simple version
⇒define(‘foreach’, ‘pushdef(‘$1’)_foreach($@)popdef(‘$1’)’)
⇒define(‘_arg1’, ‘$1’)
⇒define(‘_foreach’, ‘ifelse(‘$2’, ‘()’, ‘’,
⇒  ‘define(‘$1’, _arg1$2)$3‘’$0(‘$1’, (shift$2), ‘$3’)’)’)
⇒divert‘’dnl
```

Unfortunately, that implementation is not robust to macro names as list elements. Each iteration of `_foreach` is stripping another layer of quotes, leading to erratic results if list elements are not already fully expanded. The first cut at implementing `foreachq` takes this into account. Also, when using quoted elements in a *paren-list*, the overall list must be quoted. A *quote-list* has the nice property of requiring fewer characters to create a list containing the same quoted elements. To see the difference between the two macros, we attempt to pass double-quoted macro names in a list, expecting the macro name on output after one layer of quotes is removed during list iteration and the final layer removed during the final rescan:

```
define(‘a’, ‘1’)define(‘b’, ‘2’)define(‘c’, ‘3’)
⇒
include(‘foreach.m4’)
⇒
include(‘foreachq.m4’)
⇒
foreach(‘x’, ‘(‘‘a’’, ‘‘(b’’, ‘‘c)’’)’, ‘x
’)
⇒1
⇒(2)1
⇒
⇒, x
⇒)
foreachq(‘x’, ‘‘‘a’’, ‘‘(b’’, ‘‘c)’’’, ‘x
’)dnl
⇒a
⇒(b
⇒c)
```

Obviously, `foreachq` did a better job; here is its implementation:

```
undivert(‘foreachq.m4’)dnl
⇒include(‘quote.m4’)dnl
⇒divert(‘-1’)
⇒# foreachq(x, ‘item_1, item_2, ..., item_n’, stmt)
⇒#   quoted list, simple version
⇒define(‘foreachq’, ‘pushdef(‘$1’)_foreachq($@)popdef(‘$1’)’)
⇒define(‘_arg1’, ‘$1’)
⇒define(‘_foreachq’, ‘ifelse(quote($2), ‘’, ‘’,
⇒  ‘define(‘$1’, ‘_arg1($2)’)$3‘’$0(‘$1’, ‘shift($2)’, ‘$3’)’)’)
⇒divert‘’dnl
```

Notice that `_foreachq` had to use the helper macro `quote` defined earlier (see Section 6.3 [Shift], page 33), to ensure that the embedded `ifelse` call does not go haywire if a list element contains a comma. Unfortunately, this implementation of `foreachq` has its own severe flaw. Whereas the `foreach` implementation was linear, this macro is quadratic in the number of list elements, and is much more likely to trip up the limit set by the command line option '`--nesting-limit`' (or '`-L`', see Section 2.3 [Invoking m4], page 9). (It is possible to have robust iteration with linear behavior for either list style. See if you can learn from the best elements of both of these implementations to create robust macros; or see Section 17.3 [Answers], page 94).

7 How to debug macros and input

When writing macros for `m4`, they often do not work as intended on the first try (as is the case with most programming languages). Fortunately, there is support for macro debugging in `m4`.

7.1 Displaying macro definitions

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

`dumpdef` ([*names...*]) [Builtin]

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 to the current debug file (usually standard error), and is sorted by name. If an unknown name is encountered, a warning is printed.

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 builtin macros definitions are displayed. The definition that is dumped corresponds to what would occur if the macro were to be called at that point, even if other definitions are still live due to redefining a macro during argument collection.

```
pushdef(‘f’, ‘‘$0’1’)pushdef(‘f’, ‘‘$0’2’)
⇒
f(popdef(‘f’)dumpdef(‘f’))
error f: ‘‘$0’1’
⇒f2
f(popdef(‘f’)dumpdef(‘f’))
error m4:stdin:3: undefined macro ‘f’
⇒f1
```

See Section 7.3 [Debug Levels], page 41, for information on controlling the details of the display.

7.2 Tracing macro calls

It is possible to trace macro calls and expansions through the builtins `traceon` and `traceoff`:

`traceon` ([*names...*]) [Builtin]
`traceoff` ([*names...*]) [Builtin]

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 macros listed in *names* are affected, whether or not they are currently defined.

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 to the current debug file (defaulting to standard error, see Section 7.4 [Debug Output], page 42).

```
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. The maximum number that will appear between dashes is controlled by the option ‘`--nesting-limit`’ (see Section 2.3 [Invoking m4], page 9).

Tracing by name is an attribute that is preserved whether the macro is defined or not. This allows the ‘`-t`’ option to select macros to trace before those macros are defined.

```
traceoff(‘foo’)
⇒
traceon(‘foo’)
⇒
foo
⇒foo
define(‘foo’, ‘bar’)
⇒
foo
error m4trace: -1- foo -> ‘bar’
⇒bar
undefine(‘foo’)
⇒
ifdef(‘foo’, ‘yes’, ‘no’)
⇒no
indir(‘foo’)
error m4:stdin:8: undefined macro ‘foo’
⇒
define(‘foo’, ‘blah’)
⇒
```

```
foo
error m4trace: -1- foo -> ‘blah’
⇒blah
traceoff
⇒
foo
⇒blah
```

Tracing even works on builtins. However, `defn` (see Section 5.5 [Defn], page 24) does not transfer tracing status.

```
traceon(‘eval’, ‘m4_divnum’)
⇒
define(‘m4_eval’, defn(‘eval’))
⇒
define(‘m4_divnum’, defn(‘divnum’))
⇒
eval(divnum)
error m4trace: -1- eval(‘0’) -> ‘0’
⇒0
m4_eval(m4_divnum)
error m4trace: -2- m4_divnum -> ‘0’
⇒0
```

See Section 7.3 [Debug Levels], page 41, for information on controlling the details of the display.

7.3 Controlling debugging output

The ‘`-d`’ option to `m4` (see Section 2.5 [Invoking m4], page 10) 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:

`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`.

`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.

`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`.

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

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

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

`p` Print a message when a named file is found through the path search mechanism (see Section 9.2 [Search Path], page 56), giving the actual file name used.

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

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

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

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

If no flags are specified with the '`-d`' option, the default is '`aeq`'. The examples throughout this manual assume the default flags.

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

`debugmode` ([*flags*]) [Builtin]

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, all debugging flags are cleared (as if no '`-d`' was given), and with an empty argument the flags are reset to the default of '`aeq`'.

The expansion of `debugmode` is void.

```
define(‘foo’, ‘FOO’)
⇒
traceon(‘foo’)
⇒
debugmode()
⇒
foo
error m4trace: -1- foo -> ‘FOO’
⇒FOO
debugmode
⇒
foo
error m4trace: -1- foo
⇒FOO
debugmode(‘+l’)
⇒
foo
error m4trace:8: -1- foo
⇒FOO
```

7.4 Saving debugging output

Debug and tracing output can be redirected to files using either the '`--debugfile`' option to `m4` (see Section 2.5 [Invoking m4], page 10), or with the builtin macro `debugfile`:

`debugfile` ([*file*]) [Builtin]

Sends all further debug and trace output to *file*, opened in append mode. If *file* is the empty string, debug and trace output are discarded. If `debugfile` is called

without any arguments, debug and trace output are sent to standard error. This does not affect warnings, error messages, or `errprint` output, which are always sent to standard error. If *file* cannot be opened, the current debug file is unchanged, and an error is issued.

The expansion of `debugfile` is void.

```
traceon(‘divnum’)
⇒
divnum(‘extra’)
error m4:stdin:2: Warning: excess arguments to builtin ‘divnum’ ignored
error m4trace: -1- divnum(‘extra’) -> ‘0’
⇒0
debugfile()
⇒
divnum(‘extra’)
error m4:stdin:4: Warning: excess arguments to builtin ‘divnum’ ignored
⇒0
debugfile
⇒
divnum
error m4trace: -1- divnum -> ‘0’
⇒0
```

8 Input control

This chapter describes various builtin macros for controlling the input to `m4`.

8.1 Deleting whitespace in input

The builtin `dnl` stands for "Discard to Next Line":

`dnl` [Builtin]

All characters, up to and including the next newline, are discarded without performing any macro expansion. A warning is issued if the end of the file is encountered without a newline.

The expansion of `dnl` is void.

It is often used in connection with `define`, to remove the newline that follows 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 3.3 [Comments], page 13).

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.

```
dnl(‘args are ignored, but side effects occur’,
define(‘foo’, ‘like this’)) while this text is ignored: undefine(‘foo’)
error m4:stdin:1: Warning: excess arguments to builtin ‘dnl’ ignored
See how ‘foo’ was defined, foo?
⇒See how foo was defined, like this?
```

If the end of file is encountered without a newline character, a warning is issued and dnl stops consuming input.

```
m4wrap(‘m4wrap(‘2 hi
’)0 hi dnl 1 hi’)
⇒
define(‘hi’, ‘HI’)
⇒
^D
error m4:stdin:1: Warning: end of file treated as newline
⇒0 HI 2 HI
```

8.2 Changing the quote characters

The default quote delimiters can be changed with the builtin `changequote`:

`changequote` ([*start* = ‘’], [*end* = ‘’]) [Builtin]

This sets *start* as the new begin-quote delimiter and *end* as the new end-quote delimiter. If both arguments are missing, the default quotes (‘ and ’) are used. If *start* is void, then quoting is disabled. Otherwise, if *end* is missing or void, the default end-quote delimiter (’) is used. The quote delimiters can be of any length.

The expansion of `changequote` is void.

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

The quotation strings can safely contain eight-bit characters. If no single character is appropriate, *start* and *end* can be of any length. Other implementations cap the delimiter length to five characters, but GNU has no inherent limit.

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

Calling `changequote` with *start* as the empty string will effectively disable the quoting mechanism, leaving no way to quote text. However, using an empty string is not portable, as some other implementations of `m4` revert to the default quoting, while others preserve the prior non-empty delimiter. If *start* is not empty, then an empty *end* will use the default end-quote delimiter of ‘’, as otherwise, it would be impossible to end a quoted string. Again, this is not portable, as some other `m4` implementations reuse *start* as the end-quote delimiter, while others preserve the previous non-empty value. Omitting both arguments restores the default begin-quote and end-quote delimiters; fortunately this behavior is portable to all implementations of `m4`.

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

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

If the quotes should be changed from, say, '`[`' to '`[[`', temporary quote characters have to be defined. To achieve this, two calls of `changequote` must be made, one for the temporary quotes and one for the new quotes.

Macros are recognized in preference to the begin-quote string, so if a prefix of *start* can be recognized as part of a potential macro name, the quoting mechanism is effectively disabled. Unless you use `changeword` (see Section 8.4 [Changeword], page 50), this means that *start* should not begin with a letter, digit, or '`_`' (underscore). However, even though quoted strings are not recognized, the quote characters can still be discerned in macro expansion and in trace output.

```
define(‘echo’, ‘$@’)
⇒
define(‘hi’, ‘HI’)
⇒
changequote(‘q’, ‘Q’)
⇒
q hi Q hi
⇒q HI Q HI
echo(hi)
⇒qHIQ
changequote
⇒
changequote(‘-’, ‘EOF’)
⇒
- hi EOF hi
⇒ hi  HI
changequote
⇒
changequote(‘1’, ‘2’)
⇒
hi1hi2
⇒hi1hi2
hi 1hi2
⇒HI hi
```

Quotes are recognized in preference to argument collection. In particular, if *start* is a single '`(`', then argument collection is effectively disabled. For portability with other implementations, it is a good idea to avoid '`(`', '`,`', and '`)`' as the first character in *start*.

```
define(‘echo’, ‘$#:$@:’)
⇒
define(‘hi’, ‘HI’)
⇒
changequote(‘(’,‘)’)
⇒
echo(hi)
⇒0::hi
```

```
changequote
⇒
changequote(‘((’, ‘))’)
⇒
echo(hi)
⇒1:HI:
echo((hi))
⇒0::hi
changequote
⇒
changequote(‘,’, ‘)’)
⇒
echo(hi,hi)bye)
⇒1:HIhibye:
```

If *end* is a prefix of *start*, the end-quote will be recognized in preference to a nested begin-quote. In particular, changing the quotes to have the same string for *start* and *end* disables nesting of quotes. When quote nesting is disabled, it is impossible to double-quote strings across macro expansions, so using the same string is not done very often.

```
define(‘hi’, ‘HI’)
⇒
changequote(‘""’, ‘"’)
⇒
""hi"""hi"
⇒hihi
""hi" ""hi"
⇒hi hi
""hi"" "hi"
⇒hi" "HI"
changequote
⇒
‘hi‘hi’hi’
⇒hi‘hi’hi
changequote(‘"’, ‘"’)
⇒
"hi"hi"hi"
⇒hiHIhi
```

It is an error if the end of file occurs within a quoted string.

```
‘hello world’
⇒hello world
‘dangling quote
^D
error m4:stdin:2: ERROR: end of file in string
```

8.3 Changing the comment delimiters

The default comment delimiters can be changed with the builtin macro `changecom`:

`changecom` ([*start*], [*end* = `NL`]) [Builtin]

This sets *start* as the new begin-comment delimiter and *end* as the new end-comment delimiter. If both arguments are missing, or *start* is void, then comments are disabled. Otherwise, if *end* is missing or void, the default end-comment delimiter of newline is used. 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 if they were quoted strings. If you want the text inside a comment expanded, quote the begin-comment delimiter.

Calling `changecom` without any arguments, or with *start* as the empty string, will effectively disable the commenting mechanism. To restore the original comment start of ‘`#`’, you must explicitly ask for it. If *start* is not empty, then an empty *end* will use the default end-comment delimiter of newline, as otherwise, it would be impossible to end a comment. However, this is not portable, as some other `m4` implementations preserve the previous non-empty delimiters instead.

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

The comment strings can safely contain eight-bit characters. If no single character is appropriate, *start* and *end* can be of any length. Other implementations cap the delimiter length to five characters, but GNU has no inherent limit.

Comments are recognized in preference to macros. However, this is not compatible with other implementations, where macros and even quoting takes precedence over comments, so it may change in a future release. For portability, this means that *start* should not begin with a letter, digit, or ‘`_`’ (underscore), and that neither the start-quote nor the start-comment string should be a prefix of the other.

```
define(‘hi’, ‘HI’)
⇒
define(‘hi1hi2’, ‘hello’)
```

```
⇒
changecom(‘q’, ‘Q’)
⇒
q hi Q hi
⇒q hi Q HI
changecom(‘1’, ‘2’)
⇒
hi1hi2
⇒hello
hi 1hi2
⇒HI 1hi2
```

Comments are recognized in preference to argument collection. In particular, if *start* is a single ‘(’, then argument collection is effectively disabled. For portability with other implementations, it is a good idea to avoid ‘(’, ‘,’, and ‘)’ as the first character in *start*.

```
define(‘echo’, ‘$#:$@:’)
⇒
define(‘hi’, ‘HI’)
⇒
changecom(‘(’,‘)’)
⇒
echo(hi)
⇒0::(hi)
changecom
⇒
changecom(‘((’, ‘))’)
⇒
echo(hi)
⇒1:HI:
echo((hi))
⇒0::((hi))
changecom(‘,’, ‘)’)
⇒
echo(hi,hi)bye)
⇒1:HI,hi)bye:
```

It is an error if the end of file occurs within a comment.

```
changecom(‘/*’, ‘*/’)
⇒
/*dangling comment
^D
error m4:stdin:2: ERROR: end of file in comment
```

8.4 Changing the lexical structure of words

The macro `changeword` and all associated functionality is experimental. It is only available if the ‘`--enable-changeword`’ option was given to `configure`, at GNU `m4` installation time. The functionality will go away in the future, to

be replaced by other new features that are more efficient at providing the same capabilities. *Do not rely on it.* Please direct your comments about it the same way you would do for bugs.

A file being processed by `m4` is split into quoted strings, words (potential macro names) and simple tokens (any other single character). Initially a word is defined by the following regular expression:

```
[_a-zA-Z][_a-zA-Z0-9]*
```

Using `changeword`, you can change this regular expression:

`changeword (`*regex*`)` [Optional builtin]

Changes the regular expression for recognizing macro names to be *regex*. If *regex* is empty, use '`[_a-zA-Z][_a-zA-Z0-9]*`'. *regex* must obey the constraint that every prefix of the desired final pattern is also accepted by the regular expression. If *regex* contains grouping parentheses, the macro invoked is the portion that matched the first group, rather than the entire matching string.

The expansion of `changeword` is void. The macro `changeword` is recognized only with parameters.

Relaxing the lexical rules of `m4` might be useful (for example) if you wanted to apply translations to a file of numbers:

```
ifdef(‘changeword’, ‘’, ‘errprint(‘ skipping: no changeword support
’)m4exit(‘77’)’)dnl
changeword(‘[_a-zA-Z0-9]+’)
⇒
define(‘1’, ‘0’)1
⇒0
```

Tightening the lexical rules is less useful, because it will generally make some of the builtins unavailable. You could use it to prevent accidental call of builtins, for example:

```
ifdef(‘changeword’, ‘’, ‘errprint(‘ skipping: no changeword support
’)m4exit(‘77’)’)dnl
define(‘_indir’, defn(‘indir’))
⇒
changeword(‘_[_a-zA-Z0-9]*’)
⇒
esyscmd(‘foo’)
⇒esyscmd(foo)
_indir(‘esyscmd’, ‘echo hi’)
⇒hi
⇒
```

Because `m4` constructs its words a character at a time, there is a restriction on the regular expressions that may be passed to `changeword`. This is that if your regular expression accepts '`foo`', it must also accept '`f`' and '`fo`'.

```
ifdef(‘changeword’, ‘’, ‘errprint(‘ skipping: no changeword support
’)m4exit(‘77’)’)dnl
define(‘foo
```

```
', `bar
')
⇒
dnl This example wants to recognize changeword, dnl, and `foo\n'.
dnl First, we check that our regexp will match.
regexp(`changeword', `[cd][a-z]*\|foo[
]')
⇒0
regexp(`foo
', `[cd][a-z]*\|foo[
]')
⇒0
regexp(`f', `[cd][a-z]*\|foo[
]')
⇒-1
foo
⇒foo
changeword(`[cd][a-z]*\|foo[
]')
⇒
dnl Even though `foo\n' matches, we forgot to allow `f'.
foo
⇒foo
changeword(`[cd][a-z]*\|fo*[
]?')
⇒
dnl Now we can call `foo\n'.
foo
⇒bar
```

`changeword` has another function. If the regular expression supplied contains any grouped subexpressions, then text outside the first of these is discarded before symbol lookup. So:

```
ifdef(`changeword', `', `errprint(` skipping: no changeword support
')m4exit(`77')')dnl
ifdef(`__unix__', ,
      `errprint(` skipping: syscmd does not have unix semantics
')m4exit(`77')')dnl
changecom(`/*', `*/')dnl
define(`foo', `bar')dnl
changeword(`#\([_a-zA-Z0-9]*\)')
⇒
#esyscmd(`echo foo \#foo')
⇒foo bar
⇒
```

`m4` now requires a '`#`' mark at the beginning of every macro invocation, so one can use `m4` to preprocess plain text without losing various words like '`divert`'.

In m4, macro substitution is based on text, while in TEX, it is based on tokens. changeword can throw this difference into relief. For example, here is the same idea represented in TEX and m4. First, the TEX version:

```
\def\a{\message{Hello}}
\catcode`\@=0
\catcode`\\=12
@a
@bye
⇒Hello
```

Then, the m4 version:

```
ifdef(`changeword', `', `errprint(` skipping: no changeword support
')m4exit(`77')')dnl
define(`a', `errprint(`Hello')')dnl
changeword(`@\([_a-zA-Z0-9]*\)')
⇒
@a
⇒errprint(Hello)
```

In the TEX example, the first line defines a macro a to print the message 'Hello'. The second line defines @ to be usable instead of \ as an escape character. The third line defines \ to be a normal printing character, not an escape. The fourth line invokes the macro a. So, when TEX is run on this file, it displays the message 'Hello'.

When the m4 example is passed through m4, it outputs 'errprint(Hello)'. The reason for this is that TEX does lexical analysis of macro definition when the macro is *defined*. m4 just stores the text, postponing the lexical analysis until the macro is *used*.

You should note that using changeword will slow m4 down by a factor of about seven, once it is changed to something other than the default regular expression. You can invoke changeword with the empty string to restore the default word definition, and regain the parsing speed.

8.5 Saving text until end of 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 builtin m4wrap:

m4wrap (*string*, ...) [Builtin]

Stores *string* in a safe place, to be reread when end of input is reached. As a GNU extension, additional arguments are concatenated with a space to the *string*.

The expansion of m4wrap is void. The macro m4wrap is recognized only with parameters.

```
define(`cleanup', `This is the `cleanup' action.
')
⇒
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 action.
```

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). However, this behavior is likely to change in a future release, to match POSIX, so you should not depend on this order.

Here is an example of implementing a factorial function using `m4wrap`:

```
define(‘f’, ‘ifelse(‘$1’, ‘0’, ‘Answer: 0!=1
’, eval(‘$1>1’), ‘0’, ‘Answer: $2$1=eval(‘$2$1’)
’, ‘m4wrap(‘f(decr(‘$1’), ‘$2$1*’)’)’)’)
⇒
f(‘10’)
⇒
^D
⇒Answer: 10*9*8*7*6*5*4*3*2*1=3628800
```

Invocations of `m4wrap` at the same recursion level are concatenated and rescanned as usual:

```
define(‘aa’, ‘AA
’)
⇒
m4wrap(‘a’)m4wrap(‘a’)
⇒
^D
⇒AA
```

however, the transition between recursion levels behaves like an end of file condition between two input files.

```
m4wrap(‘m4wrap(‘)’)len(abc’)
⇒
^D
error m4:stdin:1: ERROR: end of file in argument list
```

9 File inclusion

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

9.1 Including named files

There are two builtin macros in m4 for including files:

include (*file*) [Builtin]
sinclude (*file*) [Builtin]

Both macros cause the file named *file* 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 *file*.

If *file* does not exist (or cannot be read), the expansion is void, and include will fail with an error while sinclude is silent. The empty string counts as a file that does not exist.

The macros include and sinclude are recognized only with parameters.

```
include(‘none’)
error m4:stdin:1: cannot open ‘none’: No such file or directory
⇒
include()
error m4:stdin:2: cannot open ‘’: No such file or directory
⇒
sinclude(‘none’)
⇒
sinclude()
⇒
```

The rest of this section assumes that m4 is invoked with the ‘-I’ option (see Section 2.2 [Invoking m4], page 8) pointing to the ‘m4-1.4.8/examples’ directory shipped as part of the GNU m4 package. The file ‘m4-1.4.8/examples/incl.m4’ in the distribution 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. GNU `m4` seamlessly concatenates the file contents with the next character, even if the included file ended in the middle of a comment, string, or macro call. These conditions are only treated as end of file errors if specified as input files on the command line.

In GNU `m4`, an alternative method of reading files is using `undivert` (see Section 10.2 [Undivert], page 58) on a named file.

9.2 Searching for include files

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

If the ‘`--prepend-include`’ or ‘`-B`’ command-line option was provided (see Section 2.2 [Invoking m4], page 8), those directories are searched first, in reverse order that those options were listed on the command line. Then `m4` looks in the current working directory. Next comes the directories specified with the ‘`--include`’ or ‘`-I`’ option, in the order found on the command line. Finally, 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 7.3 [Debug Levels], page 41) can help isolate the problem.

10 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.

Numbered diversions are counted from 0 upwards, diversion number 0 being the normal output stream. The number of simultaneous diversions is limited mainly by the memory used to describe them, because GNU `m4` tries to keep diversions in memory. However, there is a limit to the overall memory usable by all diversions taken altogether (512K, currently). When this maximum is about to be exceeded, a temporary file is opened to receive the contents of the biggest diversion still in memory, freeing this memory for other diversions. When creating the temporary file, `m4` honors the value of the environment variable `TMPDIR`, and falls back to '`/tmp`'. So, it is theoretically possible that the number and aggregate size of diversions is limited only by available disk space.

Diversions make it possible to generate output in a different order than the input was read. It is possible to implement topological sorting dependencies. For example, GNU Autoconf makes use of diversions under the hood to ensure that the expansion of a prerequisite macro appears in the output prior to the expansion of a dependent macro, regardless of which order the two macros were invoked in the user's input file.

10.1 Diverting output

Output is diverted using `divert`:

`divert` ([*number* = '0']) [Builtin]

The current diversion is changed to *number*. If *number* is left out or empty, it is assumed to be zero. If *number* cannot be parsed, the diversion is unchanged.

The expansion of `divert` is void.

When all the `m4` input will have been processed, all existing diversions are automatically undiverted, in numerical order.

```
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. Diversions are printed after any wrapped text is expanded.

```
define(‘text’, ‘TEXT’)
⇒
divert(‘1’)‘diverted text.’
divert
```

```
⇒
m4wrap(‘Wrapped text preceeds ’)
⇒
^D
⇒Wrapped TEXT preceeds diverted text.
```

If output is diverted to a negative 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 a common programming idiom in `m4` for avoiding them.

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

Traditional implementations only supported ten diversions. But as a GNU extension, diversion numbers can be as large as positive integers will allow, rather than treating a multi-digit diversion number as a request to discard text.

```
divert(eval(‘1<<28’))world
divert(‘2’)hello
^D
⇒hello
⇒world
```

Note that `divert` is an English word, but also an active macro without arguments. When processing plain text, the word might appear in normal text and be unintentionally swallowed as a macro invocation. One way to avoid this is to use the ‘`-P`’ option to rename all builtins (see Section 2.1 [Invoking m4], page 7). Another is to write a wrapper that requires a parameter to be recognized.

```
We decided to divert the stream for irrigation.
⇒We decided to  the stream for irrigation.
define(‘divert’, ‘ifelse(‘$#’, ‘0’, ‘‘$0’’, ‘builtin(‘$0’, $@)’)’)
⇒
divert(‘-1’)
Ignored text.
divert(‘0’)
⇒
We decided to divert the stream for irrigation.
⇒We decided to divert the stream for irrigation.
```

10.2 Undiverting output

Diverted text can be undiverted explicitly using the builtin `undivert`:

`undivert` ([*diversions*...]) [Builtin]

Undiverts the numeric *diversions* given by the arguments, in the order given. If no arguments are supplied, all diversions are undiverted, in numerical order.

As a GNU extension, *diversions* may contain non-numeric strings, which are treated as the names of files to copy into the output without expansion. A warning is issued if a file could not be opened.

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. Undiverting the empty string is the same as specifying diversion 0; in either case nothing happens since the output has already been flushed.

```
divert(‘1’)diverted text
divert
⇒
undivert()
⇒
undivert(‘0’)
⇒
undivert
⇒diverted text
⇒
```

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. Thus, when the current diversion is not 0, the current diversion does not get rearranged among the other diversions.

```
divert(‘1’)one
```

```
divert(‘2’)two
divert(‘3’)three
divert(‘2’)undivert‘’dnl
divert‘’undivert‘’dnl
⇒two
⇒one
⇒three
```

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 builtin `include` (see Section 9.1 [Include], page 55). To illustrate the difference, the file ‘`m4-1.4.8/examples/foo`’ contains the word ‘`bar`’:

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

If the file is not found (or cannot be read), an error message is issued, and the expansion is void.

10.3 Diversion numbers

The current diversion is tracked by the builtin `divnum`:

`divnum` [Builtin]

Expands to the number of the current diversion.

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

10.4 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 on the main output stream 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’)`’ followed by an explicit ‘`undivert`’:

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

No output is produced at all.

Clearing selected diversions can be done with the following macro:

`cleardivert` ([*diversions...*]) [Composite]

Discard the contents of each of the listed numeric *diversions*.

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

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; or see Section 17.4 [Answers], page 97).

11 Macros for text handling

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

11.1 Calculating length of strings

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

`len` (*string*) [Builtin]

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

The macro `len` is recognized only with parameters.

```
len()
⇒0
len(‘abcdef’)
⇒6
```

11.2 Searching for substrings

Searching for substrings is done with `index`:

`index` (*string*, *substring*) [Builtin]

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`’.

The macro `index` is recognized only with parameters.

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

Omitting *substring* evokes a warning, but still produces output.

```
index(‘abc’)
error m4:stdin:1: Warning: too few arguments to builtin ‘index’
⇒0
```

11.3 Searching for regular expressions

Searching for regular expressions is done with the builtin `regexp`:

`regexp` (*string*, *regexp*, [*replacement*]) [Builtin]

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.

If *replacement* is supplied, and there was a match, `regexp` changes the expansion to this argument, with ‘`\n`’ substituted by the text matched by the *n*th parenthesized sub-expression of *regexp*, up to nine sub-expressions. The escape ‘`\&`’ is replaced by the text of the entire regular expression matched. For all other characters, ‘`\`’ treats

the next character literally. A warning is issued if there were fewer sub-expressions than the '\n' requested, or if there is a trailing '\'. If there was no match, `regexp` expands to the empty string.

The macro `regexp` is recognized only with parameters.

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

Here are some more examples on the handling of backslash:

```
regexp(‘abc’, ‘\(b\)’, ‘\\\10\a’)
⇒\b0a
regexp(‘abc’, ‘b’, ‘\1\’)
error m4:stdin:2: Warning: sub-expression 1 not present
error m4:stdin:2: Warning: trailing \ ignored in replacement
⇒
regexp(‘abc’, ‘\(\(d\)?\)\(c\)’, ‘\1\2\3\4\5\6’)
error m4:stdin:3: Warning: sub-expression 4 not present
error m4:stdin:3: Warning: sub-expression 5 not present
error m4:stdin:3: Warning: sub-expression 6 not present
⇒c
```

Omitting *regexp* evokes a warning, but still produces output.

```
regexp(‘abc’)
error m4:stdin:1: Warning: too few arguments to builtin ‘regexp’
⇒0
```

11.4 Extracting substrings

Substrings are extracted with `substr`:

`substr` (*string*, *from*, [*length*]) [Builtin]

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. The expansion is empty if there is an error parsing *from* or *length*, if *from* is beyond the end of *string*, or if *length* is negative.

The macro `substr` is recognized only with parameters.

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

Omitting *from* evokes a warning, but still produces output.

```
substr(‘abc’)
error m4:stdin:1: Warning: too few arguments to builtin ‘substr’
⇒abc
substr(‘abc’,)
error m4:stdin:2: empty string treated as 0 in builtin ‘substr’
⇒abc
```

11.5 Translating characters

Character translation is done with `translit`:

`translit` (*string*, *chars*, [*replacement*]) [Builtin]

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 of *chars* are deleted from the expansion; if *chars* is shorter, the excess characters in *replacement* are silently ignored. If *replacement* is omitted, all characters in *string* that are present in *chars* are deleted from the expansion. If a character appears more than once in *chars*, only the first instance is used in making the translation. Only a single translation pass is made, even if characters in *replacement* also appear in *chars*.

As a GNU extension, 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 in the entire string, or as the last character of a range. Back-to-back ranges can share a common endpoint. 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`’. The expansion of a range is dependent on the underlying encoding of characters, so using ranges is not always portable between machines.

The macro `translit` is recognized only with parameters.

```
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
translit(‘+,-12345’, ‘+--1-5’, ‘<;>a-c-a’)
⇒<;>abcba
translit(‘abcdef’, ‘aabdef’, ‘bcged’)
⇒bgced
```

In the ASCII encoding, 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, even though they are not portable to EBCDIC or other encodings. The fourth example shows a range ending in ‘`-`’, as well as back-to-back ranges. The final example shows that ‘`a`’ is mapped to ‘`b`’, not ‘`c`’; the resulting ‘`b`’ is not further remapped to ‘`g`’; the ‘`d`’ and ‘`e`’ are swapped, and the ‘`f`’ is discarded.

Omitting *chars* evokes a warning, but still produces output.

```
translit(‘abc’)
error m4:stdin:1: Warning: too few arguments to builtin ‘translit’
⇒abc
```

11.6 Substituting text by regular expression

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

`patsubst` (*`string`*, *`regexp`*, [*`replacement`*]) [Builtin]

Searches *string* for matches of *regexp*, and substitutes *replacement* for each match. The syntax for regular expressions is the same as in GNU Emacs (see Section 11.3 [Regexp], page 63).

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 ‘`\`*`n`*’ substituted by the text matched by the *n*th parenthesized sub-expression of *patsubst*, for up to nine sub-expressions. The escape ‘`\&`’ is replaced by the text of the entire regular expression matched. For all other characters, ‘`\`’ treats the next character literally. A warning is issued if there were fewer sub-expressions than the ‘`\`*`n`*’ requested, or if there is a trailing ‘`\`’.

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

The macro `patsubst` is recognized only with parameters.

```
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*’, ‘(\&)’)
⇒(GNUs)() (not)() (Unix)()
patsubst(‘GNUs not Unix’, ‘\w+’, ‘(\&)’)
⇒(GNUs) (not) (Unix)
patsubst(‘GNUs not Unix’, ‘[A-Z][a-z]+’)
⇒GN not
patsubst(‘GNUs not Unix’, ‘not’, ‘NOT\’)
error m4:stdin:6: Warning: trailing \ ignored in replacement
⇒GNUs NOT Unix
```

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.

`upcase` (*`text`*) [Composite]
`downcase` (*`text`*) [Composite]

`capitalize` (*text*) [Composite]

Expand to *text*, but with capitalization changed: `upcase` changes all letters to upper case, `downcase` changes all letters to lower case, and `capitalize` changes the first character of each word to upper case and the remaining characters to lower case.

```
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(‘\&’)’)’)dnl
capitalize(‘GNUs not Unix’)
⇒Gnus Not Unix
```

While `regexp` replaces the whole input with the replacement as soon as there is a match, `patsubst` replaces each *occurrence* of a match and preserves non-matching pieces:

```
define(‘patreg’,
‘patsubst($@)
regexp($@)’)dnl
patreg(‘bar foo baz Foo’, ‘foo\|Foo’, ‘FOO’)
⇒bar FOO baz FOO
⇒FOO
patreg(‘aba abb 121’, ‘\(.\)\(.\)\1’, ‘\2\1\2’)
⇒bab abb 212
⇒bab
```

Omitting *regexp* evokes a warning, but still produces output.

```
patsubst(‘abc’)
error m4:stdin:1: Warning: too few arguments to builtin ‘patsubst’
⇒abc
```

11.7 Formatted output

Formatted output can be made with `format`:

`format` (*format-string*, ...) [Builtin]

Works much like the C function `printf`. The first argument *format-string* can contain ‘`%`’ specifications which are satisfied by additional arguments, and the expansion of `format` is the formatted string.

The macro `format` is recognized only with parameters.

Its use is best described by a few examples:

```
define(‘foo’, ‘The brown fox jumped over the lazy dog’)
⇒
format(‘The string "%s" uses %d characters’, foo, len(foo))
⇒The string "The brown fox jumped over the lazy dog" uses 38 characters
format(‘%.0f’, ‘56789.9876’)
⇒56790
```

```
len(format(‘%-*X’, ‘300’, ‘1’))
⇒300
```

Using the `forloop` macro defined earlier (see Section 6.4 [Forloop], page 34), this example shows how `format` can be used to produce tabular output.

```
include(‘forloop.m4’)
⇒
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 builtin `format` is modeled after the ANSI C ‘`printf`’ function, and supports these ‘`%`’ specifiers: ‘`c`’, ‘`s`’, ‘`d`’, ‘`o`’, ‘`x`’, ‘`X`’, ‘`u`’, ‘`e`’, ‘`E`’, ‘`f`’, ‘`F`’, ‘`g`’, ‘`G`’, and ‘`%`’; 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.

For now, unrecognized specifiers are silently ignored, but it is anticipated that a future release of GNU `m4` will support more specifiers, and give warnings when problems are encountered. Likewise, escape sequences are not yet recognized.

12 Macros for doing arithmetic

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

12.1 Decrement and increment operators

Increment and decrement of integers are supported using the builtins `incr` and `decr`:

`incr` (*number*) [Builtin]
`decr` (*number*) [Builtin]

Expand to the numerical value of *number*, incremented or decremented, respectively, by one. Except for the empty string, the expansion is empty if *number* could not be parsed.

The macros `incr` and `decr` are recognized only with parameters.

```
incr(‘4’)
⇒5
decr(‘7’)
⇒6
incr()
error m4:stdin:3: empty string treated as 0 in builtin ‘incr’
⇒1
decr()
error m4:stdin:4: empty string treated as 0 in builtin ‘decr’
⇒-1
```

12.2 Evaluating integer expressions

Integer expressions are evaluated with `eval`:

`eval` (*expression*, [*radix* = ‘10’], [*width*]) [Builtin]

Expands to the value of *expression*. The expansion is empty if an error is encountered while parsing the arguments. If specified, *radix* and *width* control the format of the output.

The macro `eval` is recognized only with parameters.

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

`+ -` Unary plus and minus

`**` Exponentiation

`* / %` Multiplication, division and modulo

`+ -` Addition and subtraction

`<< >>` Shift left or right

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

`!` Logical negation

`~`	Bitwise negation
`&`	Bitwise and
`^`	Bitwise exclusive-or
`|`	Bitwise or
`&&`	Logical and
`||`	Logical or

All operators, except exponentiation, are left associative.

Note that some older `m4` implementations use '`^`' as an alternate operator for exponentiation, although POSIX requires the C behavior of bitwise exclusive-or. On the other hand, the precedence of '`~`' and '`!`' are different in GNU `m4` than they are in C, matching the precedence in traditional `m4` implementations. This behavior is likely to change in a future version to match POSIX, so use parentheses to force the desired precedence.

Within *expression*, (but not *radix* or *width*), numbers without a special prefix are decimal. A simple '`0`' prefix introduces an octal number. '`0x`' introduces a hexadecimal number. '`0b`' introduces a binary number. '`0r`' introduces a number expressed in any radix between 1 and 36: the prefix should be immediately followed by the decimal expression of the radix, a colon, then the digits making the number. For radix 1, leading zeros are ignored and all remaining digits must be '`1`'; for all other radices, the digits are '`0`', '`1`', '`2`', ... Beyond '`9`', the digits are '`a`', '`b`' ... up to '`z`'. Lower and upper case letters can be used interchangeably in numbers prefixes and as number digits.

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
eval(‘0r1:0111 + 0b100 + 0r3:12’)
⇒12
define(‘square’, ‘eval(‘(’$1‘)**2’)’)
⇒
square(‘9’)
⇒81
square(square(‘5’)‘+1’)
⇒676
define(‘foo’, ‘666’)
⇒
eval(‘foo/6’)
error m4:stdin:8: bad expression in eval: foo/6
⇒
eval(foo/6)
⇒111
```

As the last two lines show, `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`.

All evaluation is done with 32-bit signed integers, assuming 2's-complement with wrap-around. The shift operators are defined in GNU `m4` by doing an implicit bit-wise and of the right-hand operand with 0x1f, and sign-extension with right shift.

```
eval(0x80000000 / -1)
⇒-2147483648
eval(0x80000000 % -1)
⇒0
eval(0x7fffffff)
⇒2147483647
incr(eval(0x7fffffff))
⇒-2147483648
eval(-4 >> 33)
⇒-2
```

If *radix* is specified, it specifies the radix to be used in the expansion. The default radix is 10; this is also the case if *radix* is the empty string. It is an error if the radix is outside the range of 1 through 36, inclusive. The result of `eval` is always taken to be signed. No radix prefix is output, and for radices greater than 10, the digits are lower case. The *width* argument specifies the minimum output width, excluding any negative sign. The result is zero-padded to extend the expansion to the requested width. It is an error if the width is negative. On error, the expansion of `eval` is empty.

```
eval(‘666’, ‘10’)
⇒666
eval(‘666’, ‘11’)
⇒556
eval(‘666’, ‘6’)
⇒3030
eval(‘666’, ‘6’, ‘10’)
⇒0000003030
eval(‘-666’, ‘6’, ‘10’)
⇒-0000003030
eval(‘10’, ‘’, ‘0’)
⇒10
‘0r1:’eval(‘10’, ‘1’, ‘11’)
⇒0r1:01111111111
eval(‘10’, ‘16’)
⇒a
```

13 Macros for running shell commands

There are a few builtin macros in `m4` that allow you to run shell commands from within `m4`.

Note that the definition of a valid shell command is system dependent. On UNIX systems, this is the typical `/bin/sh`. But on other systems, such as native Windows, the shell has a different syntax of commands that it understands. Some examples in this chapter assume `/bin/sh`, and also demonstrate how to quit early with a known exit value if this is not the case.

13.1 Determining the platform

Sometimes it is desirable for an input file to know which platform `m4` is running on. GNU `m4` provides several macros that are predefined to expand to the empty string; checking for their existence will confirm platform details.

`__gnu__` [Optional builtin]
`__os2__` [Optional builtin]
`os2` [Optional builtin]
`__unix__` [Optional builtin]
`unix` [Optional builtin]
`__windows__` [Optional builtin]
`windows` [Optional builtin]

Each of these macros is conditionally defined as needed to describe the environment of `m4`. If defined, each macro expands to the empty string.

When GNU extensions are in effect (that is, when you did not use the '`-G`' option, see Section 2.3 [Invoking m4], page 9), GNU `m4` will define the macro `__gnu__` to expand to the empty string.

```
__gnu__
⇒
ifdef(‘__gnu__’, ‘Extensions are active’)
⇒Extensions are active
```

On UNIX systems, GNU `m4` will define `__unix__` by default, or `unix` when the '`-G`' option is specified.

On native Windows systems, GNU `m4` will define `__windows__` by default, or `windows` when the '`-G`' option is specified.

On OS/2 systems, GNU `m4` will define `__os2__` by default, or `os2` when the '`-G`' option is specified.

If GNU `m4` does not provide a platform macro for your system, please report that as a bug.

```
define(‘provided’, ‘0’)
⇒
ifdef(‘__unix__’, ‘define(‘provided’, incr(provided))’)
⇒
ifdef(‘__windows__’, ‘define(‘provided’, incr(provided))’)
⇒
```

```
ifdef(‘__os2__’, ‘define(‘provided’, incr(provided))’)
⇒
provided
⇒1
```

13.2 Executing simple commands

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

`syscmd` (*shell-command*) [Builtin]

Executes *shell-command* as a shell command.

The expansion of `syscmd` is void, *not* the output from *shell-command*! Output or error messages from *shell-command* are not read by `m4`. See Section 13.3 [Esyscmd], page 74, if you need to process the command output.

Prior to executing the command, `m4` flushes its buffers. The default standard input, output and error of *shell-command* are the same as those of `m4`.

The macro `syscmd` is recognized only with parameters.

```
define(‘foo’, ‘FOO’)
⇒
syscmd(‘echo foo’)
⇒foo
⇒
```

Note how the expansion of `syscmd` keeps the trailing newline of the command, as well as using the newline that appeared after the macro.

As an example of *shell-command* using the same standard input as `m4`, the command line *echo* `"m4wrap(\‘syscmd(\‘cat’)’)"` | *m4* will tell `m4` to read all of its input before executing the wrapped text, then hand a valid (albeit emptied) pipe as standard input for the `cat` subcommand. Therefore, you should be careful when using standard input (either by specifying no files, or by passing ‘`-`’ as a file name on the command line, see Section 2.6 [Invoking m4], page 11), and also invoking subcommands via `syscmd` or `esyscmd` that consume data from standard input. When standard input is a seekable file, the subprocess will pick up with the next character not yet processed by `m4`; when it is a pipe or other non-seekable file, there is no guarantee how much data will already be buffered by `m4` and thus unavailable to the child.

13.3 Reading the output of commands

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

`esyscmd` (*shell-command*) [Builtin]

Expands to the standard output of the shell command *shell-command*.

Prior to executing the command, `m4` flushes its buffers. The default standard input and standard error of *shell-command* are the same as those of `m4`. The error output of *shell-command* is not a part of the expansion: it will appear along with the error output of `m4`.

The macro `esyscmd` is recognized only with parameters.

```
define(‘foo’, ‘FOO’)
⇒
esyscmd(‘echo foo’)
⇒FOO
⇒
```

Note how the expansion of `esyscmd` keeps the trailing newline of the command, as well as using the newline that appeared after the macro.

Just as with `syscmd`, care must be exercised when sharing standard input between `m4` and the child process of `esyscmd`.

13.4 Exit status

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

`sysval` [Builtin]

Expands to the exit status of the last shell command run with `syscmd` or `esyscmd`. Expands to 0 if no command has been run yet.

```
syscmd(‘false’)
⇒
ifelse(sysval, ‘0’, ‘zero’, ‘non-zero’)
⇒non-zero
syscmd(‘exit 2’)
⇒
sysval
⇒2
syscmd(‘true’)
⇒
sysval
⇒0
esyscmd(‘false’)
⇒
ifelse(sysval, ‘0’, ‘zero’, ‘non-zero’)
⇒non-zero
esyscmd(‘exit 2’)
⇒
sysval
⇒2
esyscmd(‘true’)
⇒
sysval
⇒0
```

`sysval` results in 127 if there was a problem executing the command, for example, if the system-imposed argument length is exceeded, or if there were not enough resources to fork. It is not possible to distinguish between failed execution and successful execution that had an exit status of 127.

On UNIX platforms, where it is possible to detect when command execution is terminated by a signal, rather than a normal exit, the result is the signal number shifted left by eight bits.

```
dnl This test assumes kill is a shell builtin, and that signals are
dnl recognizable.
ifdef(‘__unix__’, ,
      ‘errprint(‘ skipping: syscmd does not have unix semantics
’)m4exit(‘77’)’)dnl
syscmd(‘kill -13 $$’)
⇒
sysval
⇒3328
esyscmd(‘kill -9 $$’)
⇒
sysval
⇒2304
```

13.5 Making temporary files

Commands specified to `syscmd` or `esyscmd` might need a temporary file, for output or for some other purpose. There is a builtin macro, `mkstemp`, for making a temporary file:

`mkstemp` (*`template`*) [Builtin]
`maketemp` (*`template`*) [Builtin]

Expands to a name of a new, empty file, made from the string *template*, which should end with the string ‘`XXXXXX`’. The six ‘`X`’ characters are then replaced with random characters matching the regular expression ‘`[a-zA-Z0-9._-]`’, in order to make the file name unique. If fewer than six ‘`X`’ characters are found at the end of `template`, the result will be longer than the template. The created file will have access permissions as if by *`chmod =rw,go=`*, meaning that the current umask of the `m4` process is taken into account, and at most only the current user can read and write the file.

The traditional behavior, standardized by POSIX, is that `maketemp` merely replaces the trailing ‘`X`’ with the process id, without creating a file, and without ensuring that the resulting string is a unique file name. In part, this means that using the same *template* twice in the same input file will result in the same expansion. This behavior is a security hole, as it is very easy for another process to guess the name that will be generated, and thus interfere with a subsequent use of `syscmd` trying to manipulate that file name. Hence, POSIX has recommended that all new implementations of `m4` provide the secure `mkstemp` builtin, and that users of `m4` check for its existence.

The macros `mkstemp` and `maketemp` are recognized only with parameters.

If you try this next example, you will most likely get different output for the two file names, since the replacement characters are randomly chosen:

```
maketemp(‘/tmp/fooXXXXXX’)
⇒/tmp/fooa07346
ifdef(‘mkstemp’, ‘define(‘maketemp’, defn(‘mkstemp’))’,
      ‘define(‘mkstemp’, defn(‘maketemp’))dnl
```

```
errprint(‘warning: potentially insecure maketemp implementation
’)’)
⇒
mkstemp(‘doc’)
⇒docQv83Uw
```

Unless you use the ‘`--traditional`’ command line option (or ‘`-G`’, see Section 2.3 [Invoking m4], page 9), the GNU version of `maketemp` is secure. This means that using the same template to multiple calls will generate multiple files. However, we recommend that you use the new `mkstemp` macro, introduced in GNU M4 1.4.8, which is secure even in traditional mode.

```
syscmd(‘echo foo??????’)dnl
⇒foo??????
define(‘file1’, maketemp(‘fooXXXXXX’))dnl
ifelse(esyscmd(‘echo foo??????’), ‘foo??????’, ‘no file’, ‘created’)
⇒created
define(‘file2’, maketemp(‘fooXX’))dnl
define(‘file3’, mkstemp(‘fooXXXXXX’))dnl
ifelse(len(file1), len(file2), ‘same length’, ‘different’)
⇒same length
ifelse(file1, file2, ‘same’, ‘different file’)
⇒different file
ifelse(file2, file3, ‘same’, ‘different file’)
⇒different file
ifelse(file1, file3, ‘same’, ‘different file’)
⇒different file
syscmd(‘rm ’file1 file2 file3)
⇒
sysval
⇒0
```

14 Miscellaneous builtin macros

This chapter describes various builtins, that do not really belong in any of the previous chapters.

14.1 Printing error messages

You can print error messages using `errprint`:

`errprint` (*message*, ...) [Builtin]

Prints *message* and the rest of the arguments to standard error, separated by spaces. Standard error is used, regardless of the '`--debugfile`' option (see Section 2.5 [Invoking m4], page 10).

The expansion of `errprint` is void. The macro `errprint` is recognized only with parameters.

```
errprint(‘Invalid arguments to forloop
’)
error Invalid arguments to forloop
⇒
errprint(‘1’)errprint(‘2’,‘3
’)
error 12 3
⇒
```

A trailing newline is *not* printed automatically, so it should be supplied as part of the argument, as in the example. Unfortunately, the exact output of `errprint` is not very portable to other `m4` implementations: POSIX requires that all arguments be printed, but some implementations of `m4` only print the first. Furthermore, some BSD implementations always append a newline for each `errprint` call, regardless of whether the last argument already had one, and POSIX is silent on whether this is acceptable.

14.2 Printing current location

To make it possible to specify the location of an error, three utility builtins exist:

`__file__` [Builtin]
`__line__` [Builtin]
`__program__` [Builtin]

Expand to the quoted name of the current input file, the current input line number in that file, and the quoted name of the current invocation of `m4`.

```
errprint(__program__:__file__:__line__: ‘input error
’)
error m4:stdin:1: input error
⇒
```

Line numbers start at 1 for each file. If the file was found due to the '`-I`' option or `M4PATH` environment variable, that is reflected in the file name. The syncline option ('`-s`', see Section 2.2 [Invoking m4], page 8), and the '`f`' and '`l`' flags of `debugmode` (see Section 7.3 [Debug Levels], page 41), also use this notion of current file and line. Redefining

the three location macros has no effect on syncline, debug, or warning message output. Assume this example is run in the '`m4-1.4.8/checks`' directory of the GNU M4 package, using '`--include=../examples`' in the command line to find the file '`incl.m4`' mentioned earlier:

```
define(‘foo’, ‘‘$0’ called at __file__:__line__’)
⇒
foo
⇒foo called at stdin:2
include(‘incl.m4’)
⇒Include file start
⇒foo called at ../examples/incl.m4:2
⇒Include file end
⇒
```

The location of macros invoked during the rescanning of macro expansion text corresponds to the location in the file where the expansion was triggered, regardless of how many newline characters the expansion text contains. As of GNU M4 1.4.8, the location of text wrapped with `m4wrap` (see Section 8.5 [M4wrap], page 53) is the point at which the `m4wrap` was invoked. Previous versions, however, behaved as though wrapped text came from line 0 of the file "".

```
define(‘echo’, ‘$@’)
⇒
define(‘foo’, ‘echo(__line__
__line__)’)
⇒
echo(__line__
__line__)
⇒4
⇒5
m4wrap(‘foo
’)
⇒
foo(errprint(__line__
__line__
))
error 8
error 9
⇒8
⇒8
__line__
⇒11
^D
⇒6
⇒6
```

The `__program__` macro behaves like '`$0`' in shell terminology. If you invoke `m4` through an absolute path or a link with a different spelling, rather than by relying on a `PATH` search for plain '`m4`', it will affect how `__program__` expands. The intent is that you can use it to

produce error messages with the same formatting that `m4` produces internally. It can also be used within `syscmd` (see Section 13.2 [Syscmd], page 74) to pick the same version of `m4` that is currently running, rather than whatever version of `m4` happens to be first in `PATH`. It was first introduced in GNU M4 1.4.6.

14.3 Exiting from `m4`

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

`m4exit` ([*code* = ‘0’]) [Builtin]

Causes `m4` to exit, with exit status *code*. If *code* is left out, the exit status is zero. If *code* cannot be parsed, or is outside the range of 0 to 255, the exit status is one. No further input is read, and all wrapped and diverted text is discarded.

```
m4wrap(‘This text is lost due to ‘m4exit’.’)
⇒
divert(‘1’) So is this.
divert
⇒
m4exit And this is never read.
```

A common use of this is to abort processing:

`fatal_error` (*message*) [Composite]

Abort processing with an error message and non-zero status. Prefix *message* with details about where the error occurred, and print the resulting string to standard error.

```
define(‘fatal_error’,
       ‘errprint(__program__:__file__:__line__‘: fatal error: $*
’)m4exit(‘1’)’)
⇒
fatal_error(‘this is a BAD one, buster’)
error m4:stdin:4: fatal error: this is a BAD one, buster
```

After this macro call, `m4` will exit with exit status 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 8.5 [M4wrap], page 53) is not reread. (This macro could be made more robust to earlier versions of `m4`. You should try to see if you can find weaknesses and correct them; or see Section 17.5 [Answers], page 98).

Note that it is still possible for the exit status to be different than what was requested by `m4exit`. If `m4` detects some other error, such as a write error on standard output, the exit status will be non-zero even if `m4exit` requested zero.

If standard input is seekable, then the file will be positioned at the next unread character. If it is a pipe or other non-seekable file, then there are no guarantees how much data `m4` might have read into buffers, and thus discarded.

15 Fast loading of frozen state

Some bigger m4 applications may be built over a common base containing hundreds of definitions and other costly initializations. Usually, the common base is kept in one or more declarative files, which files are listed on each m4 invocation prior to the user's input file, or else each input file uses include.

Reading the common base of a big application, over and over again, may be time consuming. GNU m4 offers some machinery to speed up the start of an application using lengthy common bases.

15.1 Using frozen files

Suppose a user has a library of m4 initializations in 'base.m4', which is then used with multiple input files:

```
m4 base.m4 input1.m4
m4 base.m4 input2.m4
m4 base.m4 input3.m4
```

Rather than spending time parsing the fixed contents of 'base.m4' every time, the user might rather execute:

```
m4 -F base.m4f base.m4
```

once, and further execute, as often as needed:

```
m4 -R base.m4f input1.m4
m4 -R base.m4f input2.m4
m4 -R base.m4f input3.m4
```

with the varying input. The first call, containing the '-F' option, only reads and executes file 'base.m4', defining various application macros and computing other initializations. Once the input file 'base.m4' has been completely processed, GNU m4 produces on 'base.m4f' a *frozen* file, that is, a file which contains a kind of snapshot of the m4 internal state.

Later calls, containing the '-R' option, are able to reload the internal state of m4, from 'base.m4f', *prior* to reading any other input files. This means instead of starting with a virgin copy of m4, input will be read after having effectively recovered the effect of a prior run. In our example, the effect is the same as if file 'base.m4' has been read anew. However, this effect is achieved a lot faster.

Only one frozen file may be created or read in any one m4 invocation. It is not possible to recover two frozen files at once. However, frozen files may be updated incrementally, through using '-R' and '-F' options simultaneously. For example, if some care is taken, the command:

```
m4 file1.m4 file2.m4 file3.m4 file4.m4
```

could be broken down in the following sequence, accumulating the same output:

```
m4 -F file1.m4f file1.m4
m4 -R file1.m4f -F file2.m4f file2.m4
m4 -R file2.m4f -F file3.m4f file3.m4
m4 -R file3.m4f file4.m4
```

Some care is necessary because not every effort has been made for this to work in all cases. In particular, the trace attribute of macros is not handled, nor the current setting

of `changeword`. Currently, `m4wrap` and `sysval` also have problems. Also, interactions for some options of `m4`, being used in one call and not in the next, have not been fully analyzed yet. On the other end, you may be confident that stacks of `pushdef` definitions are handled correctly, as well as undefined or renamed builtins, and changed strings for quotes or comments. And future releases of GNU M4 will improve on the utility of frozen files.

When an `m4` run is to be frozen, the automatic undiversion which takes place at end of execution is inhibited. Instead, all positively numbered diversions are saved into the frozen file. The active diversion number is also transmitted.

A frozen file to be reloaded need not reside in the current directory. It is looked up the same way as an `include` file (see Section 9.2 [Search Path], page 56).

If the frozen file was generated with a newer version of `m4`, and contains directives that an older `m4` cannot parse, attempting to load the frozen file with option '`-R`' will cause `m4` to exit with status 63 to indicate version mismatch.

15.2 Frozen file format

Frozen files are sharable across architectures. It is safe to write a frozen file on one machine and read it on another, given that the second machine uses the same or newer version of GNU `m4`. It is conventional, but not required, to give a frozen file the suffix of `.m4f`.

These are simple (editable) text files, made up of directives, each starting with a capital letter and ending with a newline (NL). Wherever a directive is expected, the character '`#`' introduces a comment line; empty lines are also ignored if they are not part of an embedded string. In the following descriptions, each *len* refers to the length of the corresponding strings *str* in the next line of input. Numbers are always expressed in decimal. There are no escape characters. The directives are:

`C` *`len1`* `,` *`len2`* `NL` *`str1 str2`* `NL`
: Uses *str1* and *str2* as the begin-comment and end-comment strings. If omitted, then '`#`' and NL are the comment delimiters.

`D` *`number`*`,` *`len`* `NL` *`str`* `NL`
: Selects diversion *number*, making it current, then copy *str* in the current diversion. *number* may be a negative number for a non-existing diversion. To merely specify an active selection, use this command with an empty *str*. With 0 as the diversion *number*, *str* will be issued on standard output at reload time. GNU `m4` will not produce the '`D`' directive with non-zero length for diversion 0, but this can be done with manual edits. This directive may appear more than once for the same diversion, in which case the diversion is the concatenation of the various uses. If omitted, then diversion 0 is current.

`F` *`len1`* `,` *`len2`* `NL` *`str1 str2`* `NL`
: Defines, through `pushdef`, a definition for *str1* expanding to the function whose builtin name is *str2*. If the builtin does not exist (for example, if the frozen file was produced by a copy of `m4` compiled with changeword support, but the version of `m4` reloading was compiled without it), the reload is silent, but any subsequent use of the definition of *str1* will result in a warning. This directive may appear more than once for the same name, and its order, along with '`T`', is important. If omitted, you will have no access to any builtins.

`Q` *`len1`* `,` *`len2`* `NL` *`str1 str2`* `NL`
: Uses *str1* and *str2* as the begin-quote and end-quote strings. If omitted, then ‘`‘`’ and ‘`’`’ are the quote delimiters.

`T` *`len1`* `,` *`len2`* `NL` *`str1 str2`* `NL`
: Defines, though `pushdef`, a definition for *str1* expanding to the text given by *str2*. This directive may appear more than once for the same name, and its order, along with ‘`F`’, is important.

`V` *`number`* `NL`
: Confirms the format of the file. `m4` 1.4.8 only creates and understands frozen files where *number* is 1. This directive must be the first non-comment in the file, and may not appear more than once.

16 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.

There are also differences in BSD flavors of m4. No attempt is made to summarize these here.

16.1 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 (see Section 2.3 [Invoking m4], page 9), 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 5.2 [Arguments], page 20).

  This means that `define(‘foo’, ‘$11’)` is ambiguous between implementations. To portably choose between grabbing the first parameter and appending 1 to the expansion, or grabbing the eleventh parameter, you can do the following:

```
define(‘a1’, ‘A1’)
⇒
dnl First argument, concatenated with 1
define(‘_1’, ‘$1’)define(‘first1’, ‘_1($@)1’)
⇒
dnl Eleventh argument, portable
define(‘_9’, ‘$9’)define(‘eleventh’, ‘_9(shift(shift($@)))’)
⇒
dnl Eleventh argument, GNU style
define(‘Eleventh’, ‘$11’)
⇒
first1(‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’, ‘h’, ‘i’, ‘j’, ‘k’)
⇒A1
eleventh(‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’, ‘h’, ‘i’, ‘j’, ‘k’)
⇒k
Eleventh(‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’, ‘h’, ‘i’, ‘j’, ‘k’)
⇒k
```

- The `divert` (see Section 10.1 [Divert], page 57) macro can manage more than 9 diversions. GNU m4 treats all positive numbers as valid diversions, rather than discarding diversions greater than 9.
- 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 9.2 [Search Path], page 56).
- Arguments to `undivert` can be non-numeric, in which case the named file will be included uninterpreted in the output (see Section 10.2 [Undivert], page 58).
- Formatted output is supported through the `format` builtin, which is modeled after the C library function `printf` (see Section 11.7 [Format], page 67).

- Searches and text substitution through regular expressions are supported by the `regexp` (see Section 11.3 [Regexp], page 63) and `patsubst` (see Section 11.6 [Patsubst], page 66) builtins.
- The output of shell commands can be read into `m4` with `esyscmd` (see Section 13.3 [Esyscmd], page 74).
- There is indirect access to any builtin macro with `builtin` (see Section 5.8 [Builtin], page 28).
- Macros can be called indirectly through `indir` (see Section 5.7 [Indir], page 27).
- The name of the program, the current input file, and the current input line number are accessible through the builtins `__program__`, `__file__`, and `__line__` (see Section 14.2 [Location], page 79).
- The format of the output from `dumpdef` and macro tracing can be controlled with `debugmode` (see Section 7.3 [Debug Levels], page 41).
- The destination of trace and debug output can be controlled with `debugfile` (see Section 7.4 [Debug Output], page 42).
- The `maketemp` (see Section 13.5 [Mkstemp], page 76) macro behaves like `mkstemp`, creating a new file with a unique name on every invocation, rather than following the insecure behavior of replacing the trailing '`X`' characters with the `m4` process id.

In addition to the above extensions, GNU `m4` implements the following command line options: '`-F`', '`-G`', '`-I`', '`-L`', '`-R`', '`-V`', '`-W`', '`-d`', '`-i`', '`-l`', '`--debugfile`' and '`-t`'. See Chapter 2 [Invoking m4], page 7, for a description of these options.

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

16.2 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. Additionally, POSIX requires some behaviors that GNU `m4` has not implemented yet. Relying on these behaviors is non-portable, as a future release of GNU `m4` may change.

- System V `m4` supports multiple arguments to `defn`, and POSIX requires it. This is not yet implemented in GNU `m4`. Unfortunately, this means it is not possible to mix builtins and other text into a single macro; a helper macro is required.
- POSIX requires an application to exit with non-zero status if it wrote an error message to stderr. This has not yet been consistently implemented for the various builtins that are required to issue an error (such as `include` (see Section 9.1 [Include], page 55) when a file is unreadable, `eval` (see Section 12.2 [Eval], page 69) when an argument cannot be parsed, or using `m4exit` (see Section 14.3 [M4exit], page 81) with a non-numeric argument).
- Some traditional implementations only allow reading standard input once, but GNU `m4` correctly handles multiple instances of '`-`' on the command line.
- POSIX requires `m4wrap` (see Section 8.5 [M4wrap], page 53) to act in FIFO (first-in, first-out) order, but GNU `m4` currently uses LIFO order. Furthermore, POSIX states that only the first argument to `m4wrap` is saved for later evaluation, bug GNU `m4` saves and processes all arguments, with output separated by spaces.

However, it is possible to emulate POSIX behavior by including the file 'm4-1.4.8/examples/wrapfifo.m4' from the distribution:

```
undivert(‘wrapfifo.m4’)dnl
⇒dnl Redefine m4wrap to have FIFO semantics.
⇒define(‘_m4wrap_level’, ‘0’)dnl
⇒define(‘m4wrap’,
⇒‘ifdef(‘m4wrap’_m4wrap_level,
⇒       ‘define(‘m4wrap’_m4wrap_level,
⇒               defn(‘m4wrap’_m4wrap_level)‘$1’)’,
⇒       ‘builtin(‘m4wrap’, ‘define(‘_m4wrap_level’,
⇒                                  incr(_m4wrap_level))dnl
⇒m4wrap’_m4wrap_level)dnl
⇒define(‘m4wrap’_m4wrap_level, ‘$1’)’)’)dnl
include(‘wrapfifo.m4’)
⇒
m4wrap(‘a‘’m4wrap(‘c
’, ‘d’)’)m4wrap(‘b’)
⇒
^D
⇒abc
```

- POSIX requires that all builtins that require arguments, but are called without arguments, behave as though empty strings had been passed. For example, `a‘’define‘’b` would expand to `ab`. But GNU `m4` ignores certain builtins if they have missing arguments, giving `adefineb` for the above example.
- Traditional implementations handle `define(‘f’,‘1’)` (see Section 5.1 [Define], page 19) by undefining the entire stack of previous definitions, and if doing `undefine(‘f’)` first. GNU `m4` replaces just the top definition on the stack, as if doing `popdef(‘f’)` followed by `pushdef(‘f’,‘1’)`.
- POSIX requires `syscmd` (see Section 13.2 [Syscmd], page 74) to evaluate command output for macro expansion, but this appears to be a mistake in POSIX since traditional implementations did not do this. GNU `m4` follows traditional behavior in `syscmd`, and provides the extension `esyscmd` that provides the POSIX semantics.
- At one point, POSIX required `changequote(`*arg*`)` (see Section 8.2 [Changequote], page 46) to use newline as the close quote, but this was a bug, and the next version of POSIX is anticipated to state that using empty strings or just one argument is unspecified. Meanwhile, the GNU `m4` behavior of treating an empty end-quote delimiter as ‘’ is not portable, as Solaris treats it as repeating the start-quote delimiter, and BSD treats it as leaving the previous end-quote delimiter unchanged. For predictable results, never call changequote with just one argument, or with empty strings for arguments.
- At one point, POSIX required `changecom(`*arg*`,)` (see Section 8.3 [Changecom], page 48) to make it impossible to end a comment, but this is a bug, and the next version of POSIX is anticipated to state that using empty strings is unspecified. Meanwhile, the GNU `m4` behavior of treating an empty end-comment delimiter as newline is not portable, as BSD treats it as leaving the previous end-comment delimiter unchanged. It is also impossible in BSD implementations to disable comments, even though that

is required by POSIX. For predictable results, never call changecom with empty strings for arguments.

- Most implementations of `m4` give macros a higher precedence than comments when parsing, meaning that if the start delimiter given to `changecom` (see Section 8.3 [Changecom], page 48) starts with a macro name, comments are effectively disabled. POSIX does not specify what the precedence is, so the GNU `m4` parser recognizes comments, then macros, then quoted strings.
- Traditional implementations allow argument collection, but not string and comment processing, to span file boundaries. Thus, if '`a.m4`' contains '`len(`', and '`b.m4`' contains '`abc)`', *`m4 a.m4 b.m4`* outputs '`3`' with traditional `m4`, but gives an error message that the end of file was encountered inside a macro with GNU `m4`. On the other hand, traditional implementations do end of file processing for files included with `include` or `sinclude` (see Section 9.1 [Include], page 55), while GNU `m4` seamlessly integrates the content of those files. Thus `include(‘a.m4’)include(‘b.m4’)` will output '`3`' instead of giving an error.
- Traditional `m4` treats `traceon` (see Section 7.2 [Trace], page 39) without arguments as a global variable, independent of named macro tracing. Also, once a macro is undefined, named tracing of that macro is lost. On the other hand, when GNU `m4` encounters `traceon` without arguments, it turns tracing on for all existing definitions at the time, but does not trace future definitions; `traceoff` without arguments turns tracing off for all definitions regardless of whether they were also traced by name; and tracing by name, such as with '`-tfoo`' at the command line or `traceon(‘foo’)` in the input, is an attribute that is preserved even if the macro is currently undefined.
- POSIX requires `eval` (see Section 12.2 [Eval], page 69) to treat all operators with the same precedence as C. However, GNU `m4` currently follows the traditional precedence of other `m4` implementations, where bitwise and logical negation ('`~`' and '`!`') have lower precedence than equality operators, rather than equal precedence with other unary operators. Use explicit parentheses to ensure proper precedence. As extensions to POSIX, GNU `m4` treats the shift operators '`<<`' and '`>>`' as well-defined on signed integers (even though they are not in C), and adds the exponentiation operator '`**`'.
- POSIX requires `translit` (see Section 11.5 [Translit], page 65) to treat each character of the second and third arguments literally, but GNU `m4` treats '`-`' as a range operator.
- POSIX requires `m4` to honor the locale environment variables of `LANG`, `LC_ALL`, `LC_CTYPE`, `LC_MESSAGES`, and `NLSPATH`, but this has not yet been implemented in GNU `m4`.
- POSIX states that only unquoted leading newlines and blanks (that is, space and tab) are ignored when collecting macro arguments. However, this appears to be a bug in POSIX, since most traditional implementations also ignore all whitespace (formfeed, carriage return, and vertical tab). GNU `m4` follows tradition and ignores all leading unquoted whitespace.

16.3 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 file names 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.

  The sync line option is used mostly when using m4 as a front end to a compiler. If a diverted line causes a compiler error, the error messages should most probably refer to the place where the diversion were made, and not where it was inserted again.

- GNU m4 makes no attempt at prohibiting self-referential definitions like:

  ```
  define(‘x’, ‘x’)
  ⇒
  define(‘x’, ‘x ’)
  ⇒
  ```

  There is nothing inherently wrong with defining ‘x’ to return ‘x’. The wrong thing is to expand ‘x’ unquoted. In m4, one might use macros to hold strings, as we do for variables in other programming languages, further checking them with:

  ```
  ifelse(defn(‘holder’), ‘value’, ...)
  ```

  In cases like this one, an interdiction for a macro to hold its own name would be a useless limitation. Of course, this leaves more rope for the GNU m4 user to hang himself! Rescanning hangs may be avoided through careful programming, a little like for endless loops in traditional programming languages.

17 Correct version of some examples

Some of the examples in this manuals are buggy or not very robust, for demonstration purposes. Improved versions of these composite macros are presented here.

17.1 Solution for `exch`

The `exch` macro (see Section 5.2 [Arguments], page 20) as presented requires clients to double quote their arguments. A nicer definition, which lets clients follow the rule of thumb of one level of quoting per level of parentheses, involves adding quotes in the definition of `exch`, as follows:

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

17.2 Solution for `forloop`

The `forloop` macro (see Section 6.4 [Forloop], page 34) as presented earlier can go into an infinite loop if given an iterator that is not parsed as a macro name. It does not do any sanity checking on its numeric bounds, and only permits decimal numbers for bounds. Here is an improved version, shipped as ‘`m4-1.4.8/examples/forloop2.m4`’; this version also optimizes based on the fact that the starting bound does not need to be passed to the helper `_forloop`.

```
undivert(‘forloop2.m4’)dnl
⇒divert(‘-1’)
⇒# forloop(var, from, to, stmt) - improved version:
⇒#   works even if VAR is not a strict macro name
⇒#   performs sanity check that FROM is larger than TO
⇒#   allows complex numerical expressions in TO and FROM
⇒define(‘forloop’, ‘ifelse(eval(‘($3) >= ($2)’), ‘1’,
⇒  ‘pushdef(‘$1’, eval(‘$2’))_forloop(‘$1’,
⇒    eval(‘$3’), ‘$4’)popdef(‘$1’)’)’)
⇒define(‘_forloop’,
⇒  ‘$3‘’ifelse(indir(‘$1’), ‘$2’, ‘’,
⇒    ‘define(‘$1’, incr(indir(‘$1’)))$0($@)’)’)
⇒divert‘’dnl
include(‘forloop2.m4’)
⇒
forloop(‘i’, ‘2’, ‘1’, ‘no iteration occurs’)
⇒
forloop(‘’, ‘1’, ‘2’, ‘ odd iterator name’)
⇒ odd iterator name odd iterator name
forloop(‘i’, ‘5 + 5’, ‘0xc’, ‘ 0x‘’eval(i, ‘16’)’)
⇒ 0xa 0xb 0xc
```

```
forloop(‘i’, ‘a’, ‘b’, ‘non-numeric bounds’)
error m4:stdin:6: bad expression in eval (bad input): (b) >= (a)
⇒
```

Of course, it is possible to make even more improvements, such as adding an optional step argument, or allowing iteration through descending sequences. GNU Autoconf provides some of these additional bells and whistles in its `m4_for` macro.

17.3 Solution for `foreach`

The `foreach` and `foreachq` macros (see Section 6.5 [Foreach], page 35) as presented earlier each have flaws. First, we will examine and fix the quadratic behavior of `foreachq`:

```
include(‘foreachq.m4’)
⇒
traceon(‘shift’)debugmode(‘aq’)
⇒
foreachq(‘x’, ‘‘1’, ‘2’, ‘3’, ‘4’’, ‘x
’)dnl
⇒1
error m4trace: -3- shift(‘1’, ‘2’, ‘3’, ‘4’)
error m4trace: -2- shift(‘1’, ‘2’, ‘3’, ‘4’)
⇒2
error m4trace: -4- shift(‘1’, ‘2’, ‘3’, ‘4’)
error m4trace: -3- shift(‘2’, ‘3’, ‘4’)
error m4trace: -3- shift(‘1’, ‘2’, ‘3’, ‘4’)
error m4trace: -2- shift(‘2’, ‘3’, ‘4’)
⇒3
error m4trace: -5- shift(‘1’, ‘2’, ‘3’, ‘4’)
error m4trace: -4- shift(‘2’, ‘3’, ‘4’)
error m4trace: -3- shift(‘3’, ‘4’)
error m4trace: -4- shift(‘1’, ‘2’, ‘3’, ‘4’)
error m4trace: -3- shift(‘2’, ‘3’, ‘4’)
error m4trace: -2- shift(‘3’, ‘4’)
⇒4
error m4trace: -6- shift(‘1’, ‘2’, ‘3’, ‘4’)
error m4trace: -5- shift(‘2’, ‘3’, ‘4’)
error m4trace: -4- shift(‘3’, ‘4’)
error m4trace: -3- shift(‘4’)
```

Each successive iteration was adding more quoted `shift` invocations, and the entire list contents were passing through every iteration. In general, when recursing, it is a good idea to make the recursion use fewer arguments, rather than adding additional quoted uses of `shift`. By doing so, `m4` uses less memory, invokes fewer macros, is less likely to run into machine limits, and most importantly, performs faster. The fixed version of `foreachq` can be found in ‘`m4-1.4.8/examples/foreachq2.m4`’:

```
include(‘foreachq2.m4’)
⇒
undivert(‘foreachq2.m4’)dnl
```

```
⇒include(‘quote.m4’)dnl
⇒divert(‘-1’)
⇒# foreachq(x, ‘item_1, item_2, ..., item_n’, stmt)
⇒#   quoted list, improved version
⇒define(‘foreachq’, ‘pushdef(‘$1’)_foreachq($@)popdef(‘$1’)’)
⇒define(‘_arg1q’, ‘‘$1’’)
⇒define(‘_rest’, ‘ifelse(‘$#’, ‘1’, ‘’, ‘dquote(shift($@))’)’)
⇒define(‘_foreachq’, ‘ifelse(‘$2’, ‘’, ‘’,
⇒  ‘define(‘$1’, _arg1q($2))$3‘’$0(‘$1’, _rest($2), ‘$3’)’)’)
⇒divert‘’dnl
traceon(‘shift’)debugmode(‘aq’)
⇒
foreachq(‘x’, ‘‘1’, ‘2’, ‘3’, ‘4’’, ‘x
’)dnl
⇒1
error m4trace: -3- shift(‘1’, ‘2’, ‘3’, ‘4’)
⇒2
error m4trace: -3- shift(‘2’, ‘3’, ‘4’)
⇒3
error m4trace: -3- shift(‘3’, ‘4’)
⇒4
```

Note that the fixed version calls unquoted helper macros in `_foreachq` to trim elements immediately; those helper macros in turn must re-supply the layer of quotes lost in the macro invocation. Contrast the use of `_arg1q`, which quotes the first list element, with `_arg1` of the earlier implementation that returned the first list element directly.

For a different approach, the improved version of `foreach`, available in ‘`m4-1.4.8/examples/foreach2.m4`’, simply overquotes the arguments to `_foreach` to begin with, using `dquote_elt`. Then `_foreach` can just use `_arg1` to remove the extra layer of quoting that was added up front:

```
include(‘foreach2.m4’)
⇒
undivert(‘foreach2.m4’)dnl
⇒include(‘quote.m4’)dnl
⇒divert(‘-1’)
⇒# foreach(x, (item_1, item_2, ..., item_n), stmt)
⇒#   parenthesized list, improved version
⇒define(‘foreach’, ‘pushdef(‘$1’)_foreach(‘$1’,
⇒  (dquote(dquote_elt$2)), ‘$3’)popdef(‘$1’)’)
⇒define(‘_arg1’, ‘$1’)
⇒define(‘_foreach’, ‘ifelse(‘$2’, ‘(‘’)’, ‘’,
⇒  ‘define(‘$1’, _arg1$2)$3‘’$0(‘$1’, (dquote(shift$2)), ‘$3’)’)’)
⇒divert‘’dnl
traceon(‘shift’)debugmode(‘aq’)
⇒
foreach(‘x’, ‘(‘1’, ‘2’, ‘3’, ‘4’)’, ‘x
’)dnl
```

```
error m4trace: -4- shift(‘1’, ‘2’, ‘3’, ‘4’)
error m4trace: -4- shift(‘2’, ‘3’, ‘4’)
error m4trace: -4- shift(‘3’, ‘4’)
⇒1
error m4trace: -3- shift(‘‘1’’, ‘‘2’’, ‘‘3’’, ‘‘4’’)
⇒2
error m4trace: -3- shift(‘‘2’’, ‘‘3’’, ‘‘4’’)
⇒3
error m4trace: -3- shift(‘‘3’’, ‘‘4’’)
⇒4
error m4trace: -3- shift(‘‘4’’)
```

In summary, recursion over list elements is trickier than it appeared at first glance, but provides a powerful idiom within `m4` processing. As a final demonstration, both list styles are now able to handle several scenarios that would wreak havoc on the original implementations. This points out one other difference between the two list styles. `foreach` evaluates unquoted list elements only once, in preparation for calling `_foreach`. But `foreachq` evaluates unquoted list elements twice while visiting the first list element, once in `_arg1q` and once in `_rest`. When deciding which list style to use, one must take into account whether repeating the side effects of unquoted list elements will have any detrimental effects.

```
include(‘foreach2.m4’)
⇒
include(‘foreachq2.m4’)
⇒
dnl 0-element list:
foreach(‘x’, ‘’, ‘<x>’) / foreachq(‘x’, ‘’, ‘<x>’)
⇒ /
dnl 1-element list of empty element
foreach(‘x’, ‘()’, ‘<x>’) / foreachq(‘x’, ‘‘’’, ‘<x>’)
⇒<> / <>
dnl 2-element list of empty elements
foreach(‘x’, ‘(‘,’)’, ‘<x>’) / foreachq(‘x’, ‘‘,’’, ‘<x>’)
⇒<><> / <><>
dnl 1-element list of a comma
foreach(‘x’, ‘(‘,’)’, ‘<x>’) / foreachq(‘x’, ‘‘,’’, ‘<x>’)
⇒<,> / <,>
dnl 2-element list of unbalanced parentheses
foreach(‘x’, ‘(‘(’, ‘)’)’, ‘<x>’) / foreachq(‘x’, ‘‘(’, ‘)’’, ‘<x>’)
⇒<(><)> / <(><)>
define(‘active’, ‘ACT, IVE’)
⇒
traceon(‘active’)
⇒
dnl list of unquoted macros; expansion occurs before recursion
foreach(‘x’, ‘(active, active)’, ‘<x>
’)dnl
error m4trace: -4- active -> ‘ACT, IVE’
```

```
error m4trace: -4- active -> ‘ACT, IVE’
⇒<ACT>
⇒<IVE>
⇒<ACT>
⇒<IVE>
foreachq(‘x’, ‘active, active’, ‘<x>
’)dnl
error m4trace: -3- active -> ‘ACT, IVE’
error m4trace: -3- active -> ‘ACT, IVE’
⇒<ACT>
error m4trace: -3- active -> ‘ACT, IVE’
error m4trace: -3- active -> ‘ACT, IVE’
⇒<IVE>
⇒<ACT>
⇒<IVE>
dnl list of quoted macros; expansion occurs during recursion
foreach(‘x’, ‘(‘active’, ‘active’)’, ‘<x>
’)dnl
error m4trace: -1- active -> ‘ACT, IVE’
⇒<ACT, IVE>
error m4trace: -1- active -> ‘ACT, IVE’
⇒<ACT, IVE>
foreachq(‘x’, ‘‘active’, ‘active’’, ‘<x>
’)dnl
error m4trace: -1- active -> ‘ACT, IVE’
⇒<ACT, IVE>
error m4trace: -1- active -> ‘ACT, IVE’
⇒<ACT, IVE>
dnl list of double-quoted macro names; no expansion
foreach(‘x’, ‘(‘‘active’’, ‘‘active’’)’, ‘<x>
’)dnl
⇒<active>
⇒<active>
foreachq(‘x’, ‘‘‘active’’, ‘‘active’’’, ‘<x>
’)dnl
⇒<active>
⇒<active>
```

17.4 Solution for `cleardivert`

The `cleardivert` macro (see Section 10.4 [Cleardivert], page 60) cannot, as it stands, be called without arguments to clear all pending diversions. That is because using undivert with an empty string for an argument is different than using it with no arguments at all. Compare the earlier definition with one that takes the number of arguments into account:

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

```
divert(‘1’)one
divert
⇒
cleardivert
⇒
undivert
⇒one
⇒
define(‘cleardivert’,
  ‘pushdef(‘_num’, divnum)divert(‘-1’)ifelse(‘$#’, ‘0’,
    ‘undivert‘’’, ‘undivert($@)’)divert(_num)popdef(‘_num’)’)
⇒
divert(‘2’)two
divert
⇒
cleardivert
⇒
undivert
⇒
```

17.5 Solution for `fatal_error`

The `fatal_error` macro (see Section 14.3 [M4exit], page 81) is not robust to versions of GNU M4 earlier than 1.4.8, where invoking `__file__` (see Section 14.2 [Location], page 79) inside `m4wrap` would result in an empty string, and `__line__` resulted in ‘0’ even though all files start at line 1. Furthermore, versions earlier than 1.4.6 did not support the `__program__` macro. If you want `fatal_error` to work across the entire 1.4.x release series, a better implementation would be:

```
define(‘fatal_error’,
  ‘errprint(ifdef(‘__program__’, ‘__program__’, ‘‘m4’’)’dnl
‘:ifelse(__line__, ‘0’, ‘’,
    ‘__file__:__line__:’)‘ fatal error: $*
’)m4exit(‘1’)’)
⇒
m4wrap(‘divnum(‘demo of internal message’)
fatal_error(‘inside wrapped text’)’)
⇒
^D
error m4:stdin:6: Warning: excess arguments to builtin ‘divnum’ ignored
⇒0
error m4:stdin:6: fatal error: inside wrapped text
```

Appendix A How to make copies of this manual

A.1 GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix B Indices of concepts and macros

B.1 Index for many concepts

M

N

O

P

Q

R

S

T

U

V

W

B.2 Index for all `m4` macros

References are exclusively to the places where a builtin is introduced the first time.