GNU m4

A simple macro processor

by Rene' Seindal

Edition 0.05.

last updated 22 Jan 1990,

for `m4`, Version 0.50.

This is Edition 0.05 Beta of the *GNU m4 Manual*,
last updated 22 Jan 1990,
for `m4` Version 0.50.

1. Introduction to m4

`M4` is a macro processor, in the sense that in copies its input to the output, expanding macros as it goes. Macros can be either builtin or user-defined, and can take any number of arguments. Besides just doing macro expansion, `m4` has builtin functions for such things as including named files, running Unix commands, it supports integer arithmetic, various forms of text manipulation, recursive macros, etc...

`M4` can be used either as a front-end to a compiler (this is the traditional use), or as a macro processor in its own right.

GNU `m4` is mostly compatible with the System V, release 3 version, except for some minor differences. See chapter 16 [Compatibility], page 33 for more details.

2. Using this manual

This is a draft of the GNU `m4` manual. Neither the program, nor this manual should be regarded as finished, so comments and bug reports are always welcome. You will find the addresses to write to in the file '`README`' in the source distribution.

This manual contains a number of examples of `m4` input and output, and a simple notation is used to destinguish 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 destinguish input from output, all output from `m4` are prefixed by the string '`=>`', and all error messages by the string '`*>`'. Thus

```
Example of input line
=>Output line from m4
*>and an error message
```

If somebody has a better idea for this, I would very much like to hear about it.

3. Problems and Bugs

If you have problems with GNU `m4` or think you've found a bug, please report it to Rene' Seindal; he doesn't promise to do anything but he might well want to fix 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 email to (Internet) '`bug-gnu-utils@prep.ai.mit.edu`' or (UUCP) '`mit-eddie!prep.ai.mit.edu!bug-gnu-utils`'. Please include the version number of `m4` you are using. You can get this information with the command '`m4 -V /dev/null`'.

Non-bug suggestions are always welcome as well. If you have questions about things that are unclear in the documentation or are just obscure features, ask Rene' Seindal; he'll be happy to help you out (but no promises). You can send him electronic mail at Internet address '`seindal@diku.dk`'.

4. Invoking m4

The format of the `m4` command is:

`m4 [`*options*`] [`*macro-definition*`] [`*input-files*`]`

All options begin with '`-`'. GNU `m4` currently understands the following options:

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

`-s` Generate sync lines, for use by the C preprocessor. This is useful, if `m4` is being used as a front end to a compiler. The lines have the format '`#line` *lineno* `"`*file*`"`', as described in the manual for the C preprocessor.

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

`-d`*n* Set the debug-level to *n*. The debug-level controls the format and amount of information presented by the macros for debugging. See chapter 9 [Debugging], page 18 for more details.

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

`-G` Suppress all the extensions made in this implementation, compared to the System V version. For a list of these, see chapter 16 [Compatibility], page 33.

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

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

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

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

All macro definitions and deletions must appear *after* the other options.

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

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

5. Lexical and syntatic conventions

When `m4` scans 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 one of a name or a string. Input to `m4` can also contain comments.

5.1 Names

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

5.2 Quoted strings

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

```
‘’
```

is the empty string, and

```
‘‘quoted’’
```

is the string

```
‘quoted’
```

The quote characters can be changed at any time, using the builtin macro `changequote`. See chapter 15 [Miscellaneous], page 30 for more information.

5.3 Other tokens

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

5.4 Comments

Comments in `m4` are normally delimited by the characters `#` and newline. All characters between the comment delimiters are ignored. The begin comment character can be included in the input by quoting it.

The comment delimiter can be changed at any time, using the builtin macro `changecom`. See chapter 15 [Miscellaneous], page 30 for more information.

6. How to invoke macros

Macro invokations has one of the forms

```
name
```

which is a macro invokation without any arguments, and

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

which is a macro invokation with *n* arguments. The opening parenthesis *must* follow the *name* directly, with no spaces in between. Macros can have any number of arguments.

6.1 Macro arguments

When a name is seen, and it has a macro definition, it will be expanded as a macro. If the name is not followed immediately by an opening parenthesis, the macro will be called with no arguments.

If the name is followed by an opening parenthesis, the arguments will be collected, and the macro called. If too few arguments are supplied, the missing arguments are taken to be the empty string. If there are too many arguments, the excess arguments are ignored. Normally `m4` will issue warnings if a macro is called with an inappropriate number of arguments.

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

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

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

6.2 Quoting macro arguments

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

```
foo(() (‘(’) ‘(’)
```

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

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

```
foo(‘() (() (’)
```

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

6.3 Macro expansion

When the arguments, if any, to a macro call have been collected, the expansion text are 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 macros expansion text.

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 being reread and expanded, into ‘`Hello world`’.

7. How to define new macros

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

7.1 Defining a macro

The normal way to define or redefine macros is to use the builtin `define`, which takes two arguments, the first being the macro name, and the second the expansion text.

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 builtin `dnl` (see chapter 15 [Miscellaneous], page 30 for details).

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

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

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

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

For an explanation of the double quotes, see chapter 5 [Syntax], page 6.

GNU `m4` allows the number following the '`$`' to consist of one or more digits, allowing macros to have any number of arguments.

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.

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 simple macro to display the number of arguments given, can be

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

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

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

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

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

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

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

If you don’t understand this, see chapter 9 [Debugging], page 18.

A ‘`$`’ sign in the expansion text, that isn’t followed by anything `m4` understands, is simply copied to the macro expansion.

```
define(‘foo’, ‘$$$ hello $$$’)
=>
foo
=>$$$ hello $$$
```

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

7.2 Deleting a macro

A macro definition can be removed with the builtin `undefine`. It takes one argument, which is the name of the macro to be removed. It is best illustrated by an example

```
foo
=>foo
define(‘foo’, ‘expansion text’)
=>
```

```
foo
=>expansion text
undefine(‘foo’)
=>
foo
=>foo
```

It is not an error to try to undefine an already undefined name.

7.3 Renaming macros

It is possible to rename an already defined macro, with the builtin `defn`. It returns the *quoted definition* of its first argument. If the argument is not a defined macro, the expansion is empty.

If the argument is the name of a user defined macro, the quoted definition is simply the quoted expansion. If, instead, it is a builtin, the expansion can be seen as a special token, which points to the builtins definition. This last case is best understood through an example.

```
define(‘zap’, defn(‘undefine’))
=>
zap(‘undefine’)
=>
undefine(‘zap’)
=>undefine(zap)
```

In this way, `defn` can be used to copy macro definitions, even builtins definitions. Even if the original macro is removed, the other name can still be used to access the definition.

7.4 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`. These macros work in a stack-like fashion.

Macros are temporarily redefined with `pushdef`, which is called exactly like `define`, but instead of replacing an existing definition, the previous definition is saved, 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`, which is called just like `undefine`. If there is no previous definition, `popdef` does nothing.

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

If a macro has several definitions, and its name is given to `define`, the topmost definition is *replaced* with the new definition.

If a macro has several definitions, and its name is given to `undefine`, *all* the definitions are removed, not only the topmost one.

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

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

8. Conditionals, loops and recursion

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

8.1 Conditionals

There are two different builtin conditionals in `m4`. The first is `ifdef`, which makes it possible to test for whether a macro is defined or not. The first argument to `ifdef` is the name of a macro, and if the macro is defined, `ifdef` expands to the second arguments, else to the third argument. If the third argument is omitted, it is taken to be the empty string (according to the normal rules).

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

The other conditional, `ifelse`, is much more powerful. It takes three or more arguments. If called with three or four arguments, `ifelse` expands into the third argument, if the first and second arguments are the same, otherwise into the fourth argument (which is taken to be the empty string, if omitted). Thus

```
ifelse(foo, bar, ‘true’)
=>
ifelse(foo, foo, ‘true’)
=>true
ifelse(foo, bar, ‘true’, ‘false’)
=>false
ifelse(foo, foo, ‘true’, ‘false’)
=>true
```

Now, `ifelse` can take more than four arguments. If given more than four arguments, `ifelse` works like a multibranch. If the first and the second arguments are equal, it expands into the third argument, else the procedure are repeated with the first three arguments discarded, for example

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

Naturally, the normal case will be slightly more advanced that these examples. A common use of `ifelse` is in macros, used for loops.

8.2 Loops and recursion

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

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

```
shift(bar)
=>
shift(foo, bar, baz)
=>bar, baz
```

A simple example of use of `shift`, is this macro, which concatenates all its arguments, separated by commas.

```
define(‘concat’, ‘ifelse($2, , ‘$1’, ‘$1,concat(shift($@))’)’)
=>
concat(foo)
=>foo
concat(foo, bar, gnats,and gnus)
=>foo,bar,gnats,and gnus
```

Another example is a looping macro, that implements a simple forloop. It can for example be used for simple counting.

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

The arguments are a name for the iteration variable, the starting value, the final value, and the

text to be expanded for each iteration. With this macro, the macro `i` is defined only within the loop. After the loop, it retains whatever value it might have had before.

For-loops can be nested, like

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

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

The macro `_forloop` expands the fourth argument once, and tests to see if it is finished. If it has not finished, it increments the iteration variable (using the predefined macro `incr` (see chapter 13 [Arithmetic], page 26 for details)), and recurses.

Here is the actual implementation of `forloop`:

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

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

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

9. How to debug macros and input

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

The '`-d`' option is supposed to control the amount of details presented, when using the macros described in the following sections, but it is not implemented in the current version of GNU `m4`.

9.1 Displaying macro definitions

If you want to see what a name expands into, you can use the builtin `dumpdef`. It can be called with any number of arguments. If called with zero arguments, it displays the definitions of all known names, otherwise it displays the definitions of the names given. The output is printed directly on the standard error output. For example

```
define(‘foo’, ‘Hello world.’)
=>
dumpdef(‘foo’)
*>foo:    ‘Hello world.’
=>
dumpdef(‘define’)
*>define: <define>
=>
```

The last example shows how builtin macros definitions are displayed.

9.2 Tracing macro calls

It is possible to trace macro calls and expansions through the builtins `traceon` and `traceoff`. Both builtins can be called with any number of arguments. If called without any arguments, they will turn tracing on resp. off for all known macros, otherwise only the named macros are affected.

Whenever a traced macro is called and the arguments collected, the call is displayed. If the expansion is non-null, the expansion is displayed after the call. The output is printed directly on the standard error output. For example

```
define(‘foo’, ‘Hello World.’)
```

```
=>
define(‘echo’, ‘$@’)
=>
traceon(‘foo’, ‘echo’)
=>
foo
*>m4 trace (1): foo -> ‘Hello World.’
=>Hello World.
echo(gnus, and gnats)
*>m4 trace (1): echo( ‘gnus’, ‘and gnats’ ) -> ‘‘gnus’, ‘and gnats’’
=>gnus, and gnats
```

Tracing normally quotes all arguments and the expansion, in case they contain whitespace.

The number in parentheses 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 contains unquoted macro calls.

10. File inclusion

The builtin macro `include` takes one argument, which is a file name of a file to be read by `m4`. Subsequent input will be read from the named file. When the end of file is reached, input is resumed from the previous input file.

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

Some examples are needed here!

11. Diverting and undiverting output

Diversions are a way of temporarily saving output, which can be brought back at a later time.

11.1 Diverting output

Output can be saved temporarily by diverting it. Up to ten numbered diversions (numbered from 0 to 9) are supported in `m4`. Diversion number 0 is the normal output stream. Output are diverted using the builtin `divert`. It takes zero or one argument, which is the diversion number, or 0 if not supplied.

Diverted data, that haven't been explicitly undiverted, will be brought back when all other input have been processed.

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

Several calls of `divert` with the same argument does not overwrite the previous diveretd text, but rather appends to it.

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

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

This is a common programming ideom in `m4`.

11.2 Undiverting output

Diverted text can be brought back explicitly using the builtin `undivert`, which brings back the diversions given as arguments, in the order given. If no arguments are supplied, all diversions are brought back, in numerical order.

```
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 brought back, it is *not* reread by `m4`, but rather copied directly to the current output. It is not an error to undivert into a diversion. When a diversion has been brought back, the diverted text is discarded. It is therefore not possible to bring back diverted text more than once.

11.3 Diversion numbers

The current diversion number is returned by the builtin `divnum`. Thus

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

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

11.4 Discarding diverted text

Often it is not known, when output is diverted, whether the diverted text is actually needed. Since all non-empty diversion are brought back when the end of input is seen, a method of discarding a diversion is needed.

This is done very easy, using a macro like this.

```
define(‘cleardivert’,
‘pushdef(‘_divnum’, divnum)divert(-1)undivert($@)divert(_divnum)popdef(‘_divnum’)’)
=>
```

It is called just like `undivert`, but the effect is to clear the diversions, given by the arguments.

12. Macros for text handling

There is a number of builtins in `m4` for manipulating strings of text.

12.1 Length of strings

The builtin `len` takes one argument, and returns the length of the string. For example

```
len(‘’)
=>0
len(‘abcdef’)
=>6
```

12.2 The index function

The builtin `index` takes two arguments, and returns the index of the first occurence of the second argument in the first. The first character has index 0. If the second argument does not occur in the first argument, it returns -1. Thus

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

12.3 Extracting substrings

Substrings can be extracted using the builtin `substr`. It takes two or three arguments. The first argument is the string, from which the substring is taken, the second argument is the index of the first charater of the substring, and the third argument is length of the substring. If the last argument is omitted, the substring extends to the end of the input string.

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

12.4 Translating characters

There is a builtin, `translit`, for character translation. The first argument is the string to be worked upon and the second is the characters to be translated.

Each character in the first argument, that also occurs in the second argument, is translated into the character with the same index from the third argument. If the third argument is shorter than the second, the excess characters are deleted. If the third argument is omitted, all characters in the first, that are also present in the second, are deleted.

```
translit(‘abcdefghijklmnopqrstuvwxyz’, ‘aeiou’, ‘01234’)
=>0bcd1fgh2jklmn3pqrst4vwxyz
translit(‘abcdefghijklmnopqrstuvwxyz’, ‘aeiou’, ‘AEI’)
=>AbcdEfghIjklmnpqrstvwxyz
translit(‘abcdefghijklmnopqrstuvwxyz’, ‘aeiou’)
=>bcdfghjklmnpqrstvwxyz
```

If you think this resembles the Unix program `tr`, you are quite right. The `translit` macro does *not*, however, allow shorthands like ‘`a-z`’ to mean all lowercase letters.

13. 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.

13.1 Decrement and increment operators

Increment and decrement of integers are supported using the builtins `incr` and `decr`. Each macro takes one argument, and returns the numerical value, incremented, resp. decremented by one.

```
incr(4)
=>5
decr(7)
=>6
```

13.2 Evaluating integer expressions

The builtin `eval` evaluates general integer expressions. Expressions can contain the following operators, listed in decreasing precedence.

`-`	Unany minus
`** ^`	Exponentation
`* / %`	Multiplication, division and modulo
`+ -`	Addition and subtraction
`== != > >= < <=`	Relational operators
`!`	Logical negation
`&`	Bitwise and
`|`	Bitwise or
`&&`	Logical and
`||`	Logical or

All operators, but exponentation, are left associative.

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

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

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

```
eval(-3 * 5)
=>-15
eval(index(‘Hello world’, ‘llo’) >= 0)
=>1
define(‘square’, ‘eval(($1)^2)’)
=>
square(9)
=>81
square(square(5)+1)
=>676
define(‘foo’, ‘666’)
=>
eval(‘foo’/6)
*>examples/arithmetic.2: 14: bad expression in eval: foo/6
=>
eval(foo/6)
=>111
```

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

A second argument to `eval` can be used to specify the radix to be used in the output. The default radix is 10. If the specified radix is different from 10, the result of `eval` is taken to be unsigned, otherwise it is signed. A third argument specifies a minimum output width. The result is then zero-padded to match the requested width.

```
eval(666, 10)
=>666
eval(666, 11)
=>556
eval(666, 6)
=>3030
eval(666, 6, 10)
=>0000003030
eval(-16, 16)
=>fffffff0
```

14. Running Unix commands

There is a few builtin macros to interface to the operating system. It allows you to run Unix commands from within `m4`.

Any shell command can be run, using the builtin `syscmd`. It takes one argument, which is the command to run.

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

To test whether the shell command succeeded, the builtin `sysval` can be used. It returns the exit status of the last shell command run with `syscmd`.

```
syscmd(‘false’)
=>
sysval
=>1
syscmd(‘true’)
=>
sysval
=>0
```

There is a builtin macro, `maketemp`, for making temporary file names, for use with `syscmd`. It takes one argument, which is a file name template, ending with the string ‘`XXXXXX`’. The six `X`’s are then replaced, usually with something that includes the process id of the `m4` process, as to make the file name unique.

```
maketemp(‘/tmp/fooXXXXXX’)
=>/tmp/fooa07346
maketemp(‘/tmp/fooXXXXXX’)
=>/tmp/fooa07346
```

As seen in the example, several calls of `maketemp` might return the same string, since the selection criteria is whether the file exists or not. If the file returned by the first call has not been created before the second call, the same name might be returned.

If you want to capture the output of a shell command, run with `syscmd`, you must make a name of an unused temporary file with the builtin `maketemp`, and redirect the shell commands output to it. If the shell command succeeds, the file can be included.

```
define(‘tmpfile’, maketemp(‘/tmp/fooXXXXXX’))
=>
syscmd(‘who >’ tmpfile)
=>
ifelse(sysval, 0, ‘include(tmpfile)’)
=>herskind ttyp0   Dec 18 19:49   (tau)
=>seindal  ttyp5   Dec 18 11:54   (vale:0.0)
=>carllp   ttyp8   Dec 18 11:58   (vigrid:0.0)
=>seindal  ttype   Dec 18 12:36   (vale:0.0)
=>kaiip    ttypf   Dec 18 15:41   (gamma)
=>sigfried ttyq0   Dec 18 13:11   (rimfaxe)
=>
```

Of course, you should remove the temporary file afterwards.

15. Miscellaneous builtin macros

This chaper describes various builtins, that doesn't really belong in any of the previous chapters.

15.1 Deleting whitespace in input

The builtin `dnl` reads and discards all characters, upto and including the first newline. It is often used in connection with `define`, to remove the newline that follow the call to `define`. Thus

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

15.2 Changing the quote characters

The default quote characters can be changed with the builtin `changequote`. It takes up to two arguments, specifying the left resp. the right quote. Only the first character from each argument is used.

```
changequote([,])
=>
define([foo], [Macro [foo].])
=>
foo
=>Macro foo.
```

There is no way in `m4` to make a string containing a single left quote, less of using `changequote` to change the current quotes.

If `changequote` is called with too few arguments, the default quote characters (‘’) are used instead of the missing arguments.

15.3 Changing comment delimeters

The default comment delimeters can be changed with the builtin `changecom`. It takes up to two

arguments, specifying the start resp. the end comment delimeter. Only the first character from each argument is used.

```
define(‘comment’, ‘COMMENT’)
=>
# A normal comment
=># A normal comment
changecom(‘@’, ‘*’)
=>
# Not a comment anymore
=># Not a COMMENT anymore
But: @ this is now a comment *
=>But: @ this is now a comment *
```

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

15.4 Printing error messages

You can print error messages using the builtin `errprint`, which simply prints all its arguments on the standard error output.

```
errprint(‘Illegal arguments to forloop
’)
*>Illegal arguments to forloop
=>
```

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

15.5 Exiting from `m4`

If you need to exit from `m4` before the entire input has been read, you can use the builtin `m4exit`. It causes `m4` to exit, with exit code as supplied by the first argument. If no arguments are given, the exit code is zero.

```
define(‘fatal_error’, ‘errprint(‘m4: fatal error: $*
’)m4exit(1)’)
=>
```

```
fatal_error(‘This is a BAD one, buster’)
*>m4: fatal error: This is a BAD one, buster
```

After this input, `m4` will exit with exit code 1. This macro is only intended for error exits, since the normal exit procedures are not followed, e.g., diverted text are not brought back, and saved text (see next section) are not reread.

15.6 Saving input

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

To save input text, use the builtin `m4wrap`, which takes any number of arguments. Thus

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

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

It is safe to call `m4wrap` from saved text, but then the order the saved text is reread is undefined. If `m4wrap` isn't used recursively, the saved pieces of text are reread in the opposite order in which they were saved.

15.7 Indirect call of builtins

The builtin `builtin` can be used to call builtins indirectly. The builtin given by the first argument is called with the rest of the arguments.

16. Compatibility with other wersions of m4

This chapter describes the differences between this implementation of `m4`, and the implementation found in System V, release 3.

16.1 Facilites 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.

- Multicharacter comment delimiters are not supported in GNU `m4`. System V `m4` supports comment delimiters of upto 5 characters.
- Multicharacter quote delimiters are not supported in GNU `m4`. System V `m4` supports quote delimiters of upto 5 characters.
- System V `m4` supports multiple arguments to `defn`. This is not implemented in GNU `m4`. Its usefulness is unclear to me.

16.2 Facilites in GNU `m4` not in System V `m4`

This version of `m4` contains a few facilities, that does not exist in System V `m4`. These extra facilities are suppressed by using the '`-G`' command line switch.

- `m4wrap` takes several arguments, whereas the System V `m4` only takes one.
- `errprint` takes several arguments, whereas the System V `m4` only takes one.
- 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.

16.3 Other incompatibilities

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

- GNU `m4` implements the builtins `incr` and `decr` as ordinary macros, expanding to '`eval(($1)+1)`' resp. '`eval(($1)-1)`'. This causes error messages, if the arguments to `incr` and `decr` are ill-formed expressions. System V `m4` only looks at a numerical prefix to the argument. Consequently, the input '`incr(3foo)`' fails in GNU `m4`, and not in System V `m4`.
- GNU `m4` implements sync lines differently that 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 that should be attached to text that are being, or have 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.

Table of Contents

1 Introduction to m4 ..... **1**

2 Using this manual **2**

3 Problems and Bugs **3**

4 Invoking m4 ..... **4**

5 Lexical and syntatic conventions **6**

5.1 Names 6
5.2 Quoted strings 6
5.3 Other tokens 6
5.4 Comments 7

6 How to invoke macros **8**

6.1 Macro arguments 8
6.2 Quoting macro arguments 8
6.3 Macro expansion 9

7 How to define new macros **10**

7.1 Defining a macro 10
7.2 Deleting a macro 12
7.3 Renaming macros 13
7.4 Temporarily redefining macros 13

8 Conditionals, loops and recursion **15**

8.1 Conditionals 15
8.2 Loops and recursion 16

9 How to debug macros and input **18**

9.1 Displaying macro definitions 18
9.2 Tracing macro calls 18

10 File inclusion **20**

11 Diverting and undiverting output 21

11.1 Diverting output 21
11.2 Undiverting output 22
11.3 Diversion numbers 22
11.4 Discarding diverted text 23

12 Macros for text handling 24

12.1 Length of strings 24
12.2 The index function 24
12.3 Extracting substrings 24
12.4 Translating characters 25

13 Macros for doing arithmetic 26

13.1 Decrement and increment operators 26
13.2 Evaluating integer expressions 26

14 Running Unix commands 28

15 Miscellaneous builtin macros 30

15.1 Deleting whitespace in input 30
15.2 Changing the quote characters 30
15.3 Changing comment delimeters 30
15.4 Printing error messages 31
15.5 Exiting from `m4` 31
15.6 Saving input 32
15.7 Indirect call of builtins 32

16 Compatibility with other wersions of m4 33

16.1 Facilites in System V `m4` not in GNU `m4` 33
16.2 Facilites in GNU `m4` not in System V `m4` 33
16.3 Other incompatibilities 33