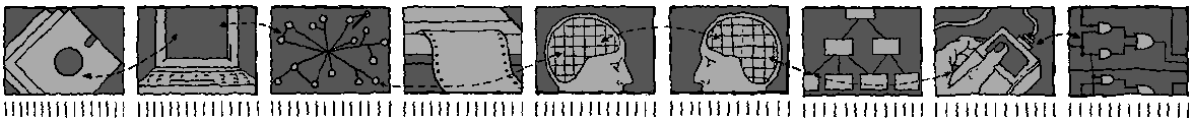
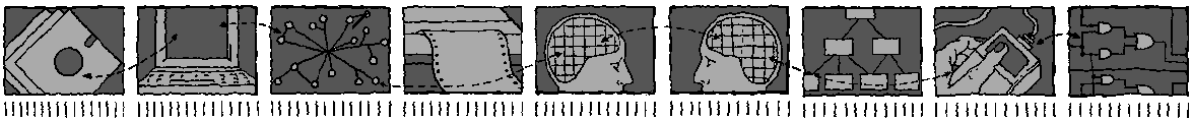


Department of Computing Science and Mathematics
University of Stirling



Exploiting the *m4* Macro Language

Kenneth J. Turner

Technical Report CSM-126

September 1994

Department of Computing Science and Mathematics
University of Stirling

Exploiting the *m4* Macro Language

Kenneth J. Turner

Department of Computing Science and Mathematics, University of Stirling
Stirling FK9 4LA, Scotland

Telephone +44-1786-467421, Facsimile +44-1786-464551
Email kjt@compsci.stirling.ac.uk

Technical Report CSM-126

September 1994

Abstract

m4 is a macro language and macro processor that is available with a number of operating systems such as UNIX and MS-DOS. The role of *m4* is discussed, and it is concluded that *m4* has a useful part to play even though it has not achieved widespread popularity. A summary of *m4* is given along with some basic examples. A cookbook of techniques is then described, allowing surprisingly sophisticated macros to be developed. Some applications of the approach are described for specifying Open Systems Interconnection standards, telecommunications service creation and digital logic circuits. To show the range of possibilities for *m4*, a few uses by others for specification and text-processing are briefly discussed.

1 Introduction

1.1 The Role of *m4*

m4 is a macro language and an associated macro processor designed by Dennis M. Ritchie [7]. *m4* has been supplied with UNIX systems for some years, and is also available for other environments such as MS-DOS. *m4* is quite simple but can be used in surprisingly sophisticated ways.

A macro processor like *m4* is a useful software engineering tool. Apart from simple macro features such as named constants and conditional expansion, *m4* can be used to provide syntactic sugar for a language and to support translation between languages. UNIX systems are supplied with a number of other tools that could be used to achieve roughly similar results, but none of them is as convenient for these purposes. *sed*, the stream editor, can be used for simple text substitution but little more. More could be done with *awk*, but parameterisation would have to be programmed and other features such as re-scanning output for macros would be hard to emulate. For a full-blown translator, *lex* and *yacc* could certainly be used. However, they would impose a comparatively complex solution involving definition of a full language grammar. The only other commonly available macro processor is *cpp*, the C preprocessor. This is perfectly adequate for conditional compilation, but lacks the more advanced features found in *m4*.

Despite is availability and utility, *m4* has somehow never achieved widespread popularity. It certainly deserves to be better known. The purpose of this paper is to show how *m4* can be effectively exploited using a variety of techniques.

1.2 Structure of the Paper

Section 2 summarises the facilities of *m4* and gives some basic examples. Section 3 lists a variety of techniques in cookbook fashion for using *m4* in more sophisticated ways. Section 4 briefly overviews some applications of the techniques described. The principal applications concern specification of Open Systems Interconnection standards, telecommunications service creation and digital logic circuits. Other applications briefly discussed are related to text-processing. The macro libraries described in this paper are available from the author on request.

2 Summary of Facilities

m4 seems to have escaped much mention in the literature, and appears to be documented mainly in manual pages; one of the few discussions in the literature is [6: Chapter 8]. The only tutorials known to the author are the original paper on *m4* [7] and one published by SUN Microsystems with their system documentation. A brief overview of *m4* is therefore appropriate.

The *m4* built-in commands (macros) are summarised in Table 1. A pleasant feature of *m4* is that its commands are themselves macros, so that the core language and the macros defined with it have a uniform style. In practice, *m4* is a smaller language than even Table 1 would suggest. The most frequently used commands are **define**, **eval**, **ifelse**, **index**, **len** and **substr**; a close second are **ifdef**, **include**, **incr** and **translit**. Despite the apparent paucity of the language, a remarkable range of effects can be achieved – as Section 3 will show.

m4 allows macros to be defined with some name and some text with which instances of the name are to be replaced. Names begin with a letter and may be followed by letters, digits and underscores; case in names is significant. As an example:

define(Operator,Plus)

defines a macro called *Operator* with value *Plus*. White space is not allowed in a macro call between the macro name (here, **define**) and the opening parenthesis. The value of a macro may be any piece of text including white space; however, literal commas and parentheses need to be quoted[1]. The value is

Built-in Command	Description
#…	Ignore comment text up to and including the newline
changequote(*left*,*right*)	Define quote characters as *left* and *right* (instead of '…')
define(*name*,*text*)	Define macro *name* to be *text*
divert(*stream_number*)	Divert future output to stream *stream_number* (1 to 9); the default of 0 is standard output
divnum	Expand to currently active diversion number
dnl	Delete up to newline, generally to prevent extra white space in macro definitions from appearing in output
dumpdef('*name*',…)	Print the definition of one or *name*s, generally for debugging; names are quoted to prevent expansion
errprint(*string*,…)	Print one or more *string*s to standard error
eval(*expression*)	Expand to result of numeric expression; the arithmetic and logical operators are essentially those of C
ifdef('*name*',*def_text*,*undef_text*)	If macro *name* (quoted to prevent expansion) is defined, expand to *def_text* otherwise *undef_text*
ifelse(*text1*,*text2*,*eq_text*,*ne_text*)	If *text1* and *text2* are equal strings, expand to *eq_text* otherwise *ne_text*; conditions may be repeated
include(*file_name*)	Expand to contents of *file_name*
incr(*number*)	Expand to *number*+1
index(*string1*,*string2*)	Expand to position in *string1* where *string2* occurs (0 is start, -1 means not found)
len(*string*)	Expand to length of *string*
maketemp(…*XXXXXX*…)	Expand to temporary filename, replacing *X*s with a process id; like C function *mktemp*
sinclude(*file_name*)	Expand to contents of *file_name*, without complaint if it does not exist ('silent include')
substr(*string*,*position*,*number*)	Expand to *string* from *position* (0 is start) for *number* characters (default to end of string)
syscmd(*string*)	Execute the command *string*; like C function *system*
translit(*string*,*from*,*to*)	Expand to *string* with *from* characters replaced by corresponding *to* characters (or deleted if there is no corresponding *to* character); like UNIX command *tr*
undefine('*name*')	Delete definition of *name* (quoted to prevent expansion)
undivert(*stream_number*,…)	Retrieve text from diversion *stream_number* (all diversions as default) and append to current diversion (usually standard output)

Table 1 Summary of *m4* Built-In Commands (Macros)

treated as text even if it is numeric. A macro name is replaced by its value whenever the name is used subsequently (or until it is undefined). The macro name must appear in a call as a separate identifier (or, more exactly, token); thus *Operator*s would not call *Operator*, though *Operator's* would.

Quotes are used to prevent macro expansion in the macro value. The macro processor effectively strips off one layer of quotes (there may be several) when reading the value. Quotes are generally used to prevent macros being expanded prematurely when they appear in the value of a macro; quotes usually do no harm if they are not needed, but see Section 3.2.

[1]Matching parentheses and any commas they enclose are also taken literally, e.g. *(a,b)*.

Suppose in the above definition of *Operator* that *Plus* had already been defined as a macro:

define(Plus,+)

A macro must be defined before it is called, so the definition of *Plus* must come before that of *Operator.* Since parameters are evaluated before a macro is expanded, the definition of *Operator* is equivalent to:

define(Operator,+)

It may be that the literal value *Plus* is required for *Operator.* In this case, the value of the macro must be quoted:

define(Operator,‘Plus’)

Although *Operator* will now expand to *Plus,* the output of a macro is re-scanned for new macros. As a result, *Plus* will be expanded to +.

A macro without parameters may be used simply by giving its name, e.g. *Operator.* A parameterless macro can also be called with an empty parameter list, e.g. *Operator().* A macro may be defined to have text parameters numbered $1 to $9, $0 being the name of the macro itself. When a parameterised macro is called, the parameters are given in parentheses.

Parameters are assigned to $1, $2, etc. in order. Actual parameters need not all be used in the definition. Conversely, $*n* will be assigned the empty string if the *n*th parameter is not actually supplied in the call. Unquoted white space before the start of a parameter is discarded; white space within a parameter is preserved. If it is necessary to include a comma or parenthesis in a parameter, it should be quoted.

As an example of parameterisation, here is a decrement macro used later that complements the built-in **incr** macro:

define(decr,‘**eval**($1-1)’)

The quotes are necessary here because the macro parameter is not known at the time the macro is defined. The following two-parameter macro expands to the average of the given numeric values:

define(Average,
‘**eval**(($1 Operator $2)/2)’)

When the macro is called, e.g. *Average(6,3),* it will expand to ***eval****((6+3)/2)*. This in turn will expand to *4*, since ‘/’ truncates to an integer.

3 Tricks of the Trade

The level of facilities described in Section 2 is perfectly adequate for simple text substitution. However, a wide range of effects can be achieved with some additional thought. The following subsections describe a few techniques, and also some of the pitfalls with *m4*. This section can be used as a cookbook of ideas to be adapted as required.

With judicious selection of techniques, *m4* can be used almost like an imperative or a functional programming language! In some senses it is even declarative. The following techniques tend to an imperative style: variables (Section 3.3), conditionals (Section 3.5), side effects (Section 3.12) and iteration (Section 3.13). The following techniques tend to a functional style: conditionals (Section 3.5), higher-order macros (Section 3.8), curried macros (Section 3.9), recursion (Sections 3.10 and 3.11) and list processing (Sections 3.14 and 3.15).

A functional style of macro definition offers freedom from side-effects and referential transparency as in functional programming. It is not intended to promote this style particularly, but it has proven useful in a number of applications described in Section 4.1.

3.1 Use of Files

Although *m4* will read macro definitions from standard input, it is best to store macros in a file and use them by giving the filename on the command line: *m4 filename*. A filename extension of '*.m4*' is suggested for such files. It is also convenient to separate files containing only macro definitions from files that use them. In this way, generic macro packages can be built up and then used for a variety of applications. A macro package can be read in using **include**. If a file to be read may not exist (e.g. it is conditionally generated) then use **sinclude**.

3.2 Quoting

The value of a macro should almost invariably be quoted. When a macro is defined its name should always be quoted if there is a risk that it is already defined as a macro. This would apply to redefining a macro, and also to other uses of macro names such as with **dumpdef**, **ifdef** and **undefine**. Quoting requires more thought when a macro definition includes calls to other macros. Again, the safest rule is to quote subsidiary calls so they are not expanded until necessary:

define(Macro1,
'Macro2(Parameter1,'Macro3(Parameter2)')')

It is very easy to get quotes and parentheses mismatched with complex macros. Unless the text editor used to produce the *m4* file has facilities to help, there may be no solution but to copy the file and cut out pieces of text until the problem is found.

Insufficient quotes will cause a macro to be expanded prematurely. This will cause macros to be expanded in the wrong order, cause parameters to be used before they have been defined properly, or cause argument stack overflow. Note that all parameters are evaluated before calling a macro. Excess quotes will prevent macro expansion at the correct time, usually causing macro names or parameters to be taken literally.

Sometimes it is necessary to introduce empty quotes in order to have the input parsed correctly. For example, a macro that expands to the value of *Operator* twice in succession cannot be defined as:

define(Double,'OperatorOperator')

because *OperatorOperator* is parsed as a single token. The following is required:

define(Double,'Operator''Operator')

3.3 Variables

m4 does not provide conventional variables. However, macros may be treated as (text) variables. Suppose that a *Count* variable has to be periodically incremented. This might be achieved with:

define('Count','**incr**(Count)')

This replaces the current value of *Count* with its incremented value. Note that the first reference to *Count* must be quoted since it is already a macro. Failing to quote it will define a macro whose name is the current value of *Count*!

Macro names in *m4* are unfortunately global; there is no notion of nested scope. As a result, local variables within macros must be named carefully to avoid clashes. It is suggested that local variables start with the name of the macro in which they are defined, e.g. if *Average* required a local sum it might be called *AverageSum*. This could be achieved automatically using $0, the current macro name. The local variable of *Average* might therefore be called *$0Sum*. Some examples of local variables appear in Appendix 3.14.

3.4 Controlling White Space

It is unfortunately easy to introduce extraneous white space in the output of a macro. White space after a parameter is usually the problem. For example, the following inadvertently produces a newline after the result:

```
define(Average,
  `eval(($1 Operator $2)/2)
')
```

To include white space intentionally, include it inside quotes:

```
define(Operator,`
  Plus
')
```

This expands to a newline, some spaces, *Plus* and a newline.

In a macro package, it is desirable not to generate white space in the output while the macro definitions are being read. One way to do this is to join the end of one definition onto the start of another:

```
define(Macro1,
  Text1)define(Macro2,
  Text2)
```

However, this is rather untidy. An alternative is to use the **dnl** command to discard text up to and including the next line:

```
define(Macro1,
  Text1)dnl
define(Macro2,
  Text2)
```

However, this is rather ugly too. A better solution is to include macros inside a diversion. Diversions are discussed further in Section 3.12, but for now it is sufficient to note the following approach:

```
divert(-1)
define(Macro1,
  Text1)
define(Macro2,
  Text2)
divert
```

The first **divert** sends all subsequent output to stream -1 (‘the void’), causing it to be ignored. The second **divert** restores normal output[2]. This pair of **divert** commands would normally appear around a macro package, ensuring that reading in the package causes no output.

3.5 Conditionals

The **ifelse**(*text1*,*text2*,*eq_text*,*nel_text*) command corresponds to the Pascal form:

```
if text1 = text2
  then eq_text
  else ne_text
```

Either *eq_text* or *ne_text* may be the empty string, meaning that there is no result. In particular, if there is no ‘else’ alternative then the fourth parameter may be omitted. There may only be a check for equality of the *text1* and *text2* strings. For logical conditions, use the **eval** command and check the result against 1 (true) and 0 (false). For example:

[2] **dnl** might be used after the final **divert** to prevent it generating a newline.

ifelse(**eval**(4<7),1,is less,not less)

If conditionals contain macro calls as parameters it is likely that these should be quoted. Conditionals may be nested, as in the following macro that expands to text giving the sign of its operand:

```
define(Sign,
  ‘ifelse(eval($1<0),1,
    negative,
    ifelse(eval($1==0),1,
      zero,
      positive))’)
```

Nested conditionals like this are so common that *m4* supports a convenient short-hand: the ‘else’ clause can be replaced by further ‘if then’ clauses. *Sign* can therefore be rewritten more simply as:

```
define(Sign,
  ‘ifelse(eval($1<0),1,
    negative,
    eval($1==0),1,
      zero,
      positive)’)
```

It is also possible to build up nested conditionals using **ifdef** to check if macros are defined or not.

3.6 Recording Definitions

Due to conditional expansion, a particular part of a macro package may not actually be used. In such a case it may be useful to record that a certain expansion took place. It is possible to define a macro with no value (or more exactly an empty string as value) just to record that its definition was met. The existence of the macro can be checked with **ifdef**:

```
define(Used)

ifdef(‘Used’,was defined,not defined)
```

Note that there is no equivalent of **ifndef** in *cpp*; the ‘else’ part of **ifdef** should be used instead.

3.7 Nested Definitions

More sophisticated macros may themselves define macros. Suppose, for example, that is useful to know whether macros to perform arithmetic have been called:

```
define(Minus,‘define(‘Maths_Used’)eval($1-$2)’)
define(Times,‘define(‘Maths_Used’)eval($1*$2)’)
… Minus(5,3) …
ifdef(‘Maths_Used’,Maths_Functions)
```

When *Minus* or *Times* is invoked, the *Maths_Used* macro is defined. (A common macro called by *Minus* and *Times* could also be defined to do this.) *Maths_Used* can later be tested to take appropriate action (say, to include mathematics functions in the output).

As discussed in Section 3.10, a macro may even redefine itself.

3.8 Higher-Order Macros

It can be convenient to define higher-order macros: macros that take macros as parameters. Fortunately this is achieved quite straightforwardly. The macro as parameter must be quoted on the call and used unquoted in the definition. The reason this works is that *m4* scans the expansion of a macro for

the presence of other macros. An example might be a macro that applies the ‘operation’ it is given to the rest of the parameters:

```
define(Apply,$1($2,$3))
```

This allows for operations with one or two parameters, such as:

```
Apply(‘incr’,42)          incr(42) or 43
Apply(‘Average’,15,8)   Average(15,8) or 11
```

Because macros usually ignore extra parameters[3], the same definition of *Apply* can be used for unary or binary operators. Obviously this can be extended to *n*-ary operators (for *n* up to 8 due to the parameter limit in *m4*).

3.9 Curried Macros

A curried[4] function is one that has been partially applied to its initial parameters. Similarly it may be useful to define a curried macro that has some of its parameters fixed.

Suppose a *Times3* macro is required that acts like *Times* with its first parameter fixed at 3:

```
define(Curry1,‘define($1,‘$2’($3,$‘’1,$‘’2))’)
Curry1(Times3,‘Times’,3)

Times3(5)
```

Curry1 defines its first parameter to be a macro that always calls the second parameter with the third parameter plus others. The second parameter must be quoted as it will be a macro name.

There is a complication with parameters for curried macros. The first parameter is supplied as $3 on the call of *Curry1*. Other parameters will be supplied only when the curried macro is called, so they cannot be included in the definition as $1, $2, etc. By including empty quotes, as in $‘’1, the definition of the curried macro is made to refer to literal $1 and not to the value of the first parameter when *Curry1* is called. When the curried macro is called, its first parameter (actually the second parameter of *Times*) will be used. The definition above allows the curried macro to have two parameters, but this could be extended to more parameters.

Unfortunately a single *Curry* macro cannot be defined since the number of parameters to be fixed must be known. Currying *n* parameters is, however, possible. For example, for two parameters:

```
define(Curry2,‘define($1,‘$2’($3,$4,$‘’1,$‘’2))’)
Curry2(Times35,‘Times’,3,5)

Times35
```

This defines a *Times35* macro that always multiplies 3 and 5; *Times35* always returns the constant result 15. Although *Curry2* allows for the curried macro to have additional parameters (here two), they are not actually used in this case.

3.10 Recursion

The expansion of a macro may include a call to the same macro. (Recall that the expansion is scanned for further macro calls.) As with any language, there must be a condition to stop the recursion.

As an example, here is the classical factorial function:

```
define(Fact,
  ‘ifelse(eval($1<=1),1,
    1,
```

[3]The approach here would not work if the applied macro specifically checked that spare parameters were unused.

[4]The name derives from Curry the mathematician.

```
    ‘eval($1*Fact(decr($1)))’)’)

fact(15)
```

If the argument is 1 or less then the factorial is 1, otherwise the result is the argument times factorial of the argument decremented.

There are two problems with the definition above. One is that large arguments (perhaps as little as 16) may cause integer overflow. The other is that a complex expression has to be finally evaluated: **eval***(15**__**eval***(14*...*1))*, possibly causing overflow of the *m4* argument stack. This is a more serious problem which is solved in Section 3.11.

The classical Fibonacci function can also be defined recursively:

```
define(Fib,
  ‘ifelse(eval($1<=1),1,
    $2,
    ‘ifelse(eval($1==2),1,
      $3,
      ‘eval(Fib(decr(decr($1)),$2,$3)+Fib(decr($1),$2,$3))’)’)’)

fib(17,1,1)
```

The first parameter is the number of the term required (starting at 1), and the other two parameters are the ‘seeds’ for the series. As usual, the first term is the first seed and the second term is the second seed. For later terms, the *n*th term is calculated as *Fib(n-2)+Fib(n-1)*. As with *Fact*, trying to calculate too high a term may cause integer overflow or argument stack overflow.

Mutual recursion is also possible. Macros *Flip* and *Flop* call each other the number of times given by their parameter, outputting *On* or *Off* each time:

```
define(Flip,‘ifelse(eval($1>0),1,‘On Flop(decr($1))’)’)
define(Flop,‘ifelse(eval($1>0),1,‘Off Flip(decr($1))’)’)

Flip(5)
```

3.11 Dealing with Arbitrary Recursion

Because recursion is often used in the definition of a macro, some solution should be found to the problem of argument stack overflow if there are many levels of recursion. A general approach is to accumulate the results of recursion in auxiliary parameters; these can be initialised with the base values for recursion. To preserve the same calling interface, an auxiliary macro should be invoked by one that supplies these parameters invisibly. As an example of this, here is a reworked factorial macro:

```
define(Fact,‘FactAux($1,1)’)

define(FactAux,
  ‘ifelse(eval($1<=0),1,
    $2,
    ‘FactAux(decr($1),eval($1*$2))’)’)
```

The auxiliary parameter to *FactAux* is initially given as 1 by *Fact*. On each recursive call, the auxiliary parameter is multiplied by the main parameter. *Fact(15)*, for example, will be evaluated progressively as 15*1, 14*15, …, 2*1002155008, 2004310016. This avoids the arbitrarily nested list of calculations that can cause argument stack overflow with the first definition of *Fact*. The same approach can be applied to other recursive macros, such as those for list-processing is given in Appendix 3.14.

3.12 Side-Effects

Sometimes it is necessary for a macro to have side-effects, say to generate definitions or text that are used in later processing. One way of achieving this is a via a diversion using the **divert** command. This causes subsequent output to be accumulated in a temporary file identified by a stream number A diversion is ended by a new **divert** command. By default, all diversions are output at the end of processing in ascending order of stream number. More explicit control is possible with the **undivert** command which appends the diverted output to the current diversion (usually the standard output). Note that text recovered from a diversion is not scanned again for macros.

Suppose that it is required to record which arithmetic macros are called and to output a list of their names at the end of processing:

define(Minus,‘Called(-)**eval**($1-$2)’)
define(Times,‘Called(*)**eval**($1*$2)’)
define(Called,‘**divert**(1)‘ Called $1**’divert**(0)’)

The *Called* macro records the call of each macro in diversion 1, then reverts to diversion 0 for normal output; a plain **divert** would have the same effect as **divert**(0). During processing, the result of each macro call would be recorded in the diversion. At the end of processing, the text accumulated in diversion 1 would be output automatically. The diversion could be output at an earlier point in processing by calling **undivert**, or **undivert**(1) if there were other diversions.

Another way of using an intermediate file of results is to write to a temporary file explicitly and then read this back in again using **include**. To write anything to such a file requires **syscmd** since *m4* output can be redirected only with **divert**. Temporary files should therefore be used for the output of system commands, and diversions should be used for the output of *m4* commands. Note, however, that use of **syscmd** may reduce portability by use of system-specific commands.

The fact that diverted text is not scanned again for macros may be desirable or not. In fact the above example takes advantage of this since the word *Called* is stored in the diversion; it certainly should not invoke macro *Called* when the diversion is recovered. If the stored text is to be re-scanned for macros, it must be appended to a text variable that is later output. Here is a more complex revision of *Called* using this idea:

define(Called_List)
define(Call_text, ‘Call of $1’)
define(Called,‘**define**(‘Called_List’,Called_List ‘Call_Text($1)’)’)
…
Called_List

Now *Called* maintains a list of calls in variable *Called_List* which is initialised to the empty string. On a new call, the details are appended to the current value of *Called_List*. Instead of storing *Call of …* directly, the name of the *Call_Text* macro is stored. When the list is generated by using *Called_List*, *Call_Text* is replaced by its expansion[5].

3.13 Iteration

Iteration can be achieved only through recursion. For example, the following macro and its call will result in the output *Counter is 10…Counter is 1*:

define(loop,
  ‘**ifelse**($2,0,,
    ‘**define**(‘$1’,$2)$3‘’loop(‘$1’,decr($2),‘$3’)’)’)

loop(‘i’,10,‘Counter is i’)

[5]In truth, *Call_Text* is expanded each time *Called_List* is redefined. It is only the last *Call_Text* that is expanded when *Called_List* is used.

The name of the loop counter, the number of iterations and the loop body are given as parameters. The name of the loop counter will have to be quoted if it is used previously, since it is redefined as a macro inside *loop*; the loop counter is progressively decremented. The loop body will almost certainly need to be quoted since it should not be expanded until *loop* is being executed. The reason for the empty quotes after $3 is to prevent the loop body from being combined with the recursive call of *loop*; here, the loop expansion would be *Counter is iloop...* and thus not be parsed properly.

A Pascal-like **for** loop to do the same thing could be achieved with:

```
define(for,
  `ifelse(eval(($4==0) || (($4>0) && ($2>$3)) || (($4<0) && ($2<$3))),1,,
    `define(`$1',$2)$5``for(`$1',eval($2 + $4),$3,$4,`$5')')')

for(`i',10,1,-1,`Counter is i')
```

The name of the loop counter, the start value, the finish value, the step value and the loop body are given as parameters. The condition for stopping the loop is that the step is zero, or the step is positive and start is greater than finish, or the step is negative and start is less than finish.

To achieve a Pascal-like **while** loop requires:

```
define(while,
  `ifelse(eval($1),1,$2`while(`$1',`$2')')')

define(`i',10)

while(`i!=0',`Counter is i``define(`i',`decr(i)')')
```

The parameters here are the loop condition and the loop body, both quoted to prevent premature expansion. The definition of the *while* macro is straightforward, but its use is more complex. The loop condition need not involve a counter, though it does so here. The condition is assumed to be a logical expression to be checked using **eval**; other possibilities not catered for would include the use of **ifdef**. The initial value of the counter must therefore be defined, the loop terminating condition must be framed in terms of this, and the loop body must redefine the counter (here decrementing it).

A Pascal-like **repeat** loop is very similar:

```
define(repeat,
  `$1``ifelse(eval($2),0,`repeat(`$1',`$2')')')

define(`i',10)

repeat(`Counter is i``define(`i',`decr(i)')',`i==1')
```

Following Pascal syntax, the loop body and loop condition are given in the opposite order to **while**. As in Pascal, the loop body is always executed at least once for **repeat**.

3.14 List Notation

List-valued macros may be defined. The simplest solution is to separate list elements by, say, white space though this will not work if list elements may contain white space. More conventional list notation may be used, but care has to be taken over values containing commas and parentheses. These can cause considerable problems because they are part of the *m4* meta-syntax. If it is really necessary to use a list notation such as *(1,2,3)*, the best solution is to use this notation only externally (i.e. as input or output of a macro package). A less troublesome notation such as *[1;2;3]* should preferably be used internally (i.e. between macros of the package). The following macros convert between these representations:

```
define(IntList,`translit($1,(,),[;])')
define(ExtList,`translit($1,[;],(,))')
```

The text string of three characters *(,)* is considered to be one parameter because parentheses are matched. These characters correspond to the internal list characters *[;]*. For generality, it should be possible to change the internal list characters; they might be defined as macros *ListOpen*, *ListSep* and *ListClose*. However, for simplicity the characters have been fixed in the examples discussed here.

Because the internal notation avoids commas and parentheses, there is no risk of an internal list being parsed as other than one parameter string. When macros require to input a list, they should apply *IntList* to convert them to internal form. When macros require to output a list, they should call *ExtList* to reverse the conversion.

3.15 List Processing

m4 lends itself to list processing, for which a number of functions are fairly standard. As in Section 3.14, lists are used in the form *[…;…]*. The implementation of list processing functions is rather detailed and so is given in Appendix 3.14. The functions may be used as follows:

Cons(abc,[de;fghi])	construct new list element	[abc;de;fghi]
Head([alpha;beta;gamma])	first element of list	alpha
Tail([aleph;beth])	all but first element	[beth]
Concat([1;2],[3;4])	concatenate lists	[1;2;3;4]
Length([x;y;z])	length of list	3
Reverse([aa;bb;cc])	reverse order of elements	[cc;bb;aa]
AppList(‘decr’,[2;-8;7])	apply macro to list	1 -9 6
MapList(‘**incr**’,[2;-8;7])	map elements using macro	[3;-7;8]

3.16 Debugging

All except very simple macro packages will need debugging. When it is not clear what is happening during macro expansion, the best approach is to put ‘debugging statements’ inside macro values. These are not statements at all, of course, just text to be expanded for debugging purposes.

Suppose that inside *Average* it seems that *Operator* is being expanded incorrectly. The definition of *Average* can be temporarily altered to produce diagnostic output:

define(Average
  ‘‘Operator’ is Operator in **eval**(($1 Operator $2)/2)’)

Note that the first occurrence of *Operator* is quoted so that it is taken as a literal string. The second occurrence will be replaced by the current value of *Operator.* The result of expanding *Average(6,3)* would thus be *Operator is + in 4*. To make it easier to distinguish diagnostic output, it may be delimited in some way (say, using {…}).

An alternative is to use **errprint** to send diagnostic output to standard error. This has the advantage of separating diagnostics and normal output. However, the approach above has the advantage of showing diagnostics at the relevant place in the output.

dumpdef may occasionally be useful during debugging to confirm the definition of a macro, particularly one defined dynamically inside another or where the effect of quoting is unclear.

4 Applications of *m4*

The techniques described in Section 3 were evolved by the author to support formal specification in a number of domains. In each case a macro library was developed to allow translation from some surface syntax to an underlying specification language. The purpose of this section is to give a feel for the range of applications that *m4* can support; the work of some colleagues is therefore included. It is not practicable to describe each application in detail here.

4.1 Specification Applications

The formal specification language LOTOS (Language Of Temporal Ordering Specification [5]) uses algebraically specified data types. The data typing facilities are powerful but very basic, and the standard library of data types is rather impoverished. Work is in progress to enhance the data types in LOTOS, but it will take time before these are standardised and supported by tools. In the interim, the author and Junhai Lee (University of Stirling) have developed support for a provisional syntax of the new data types. This includes characters, integers, enumerated types, records, arrays, sets and various kinds of lists. The data type syntax is translated into the current version of LOTOS using about 40 macros in 560 non-comment lines of *m4*. The macro library uses some of the techniques described in Section 3.

When applying a formal language to the specification of an architecture, it is desirable to develop an architectural semantics to relate architectural concepts to language concepts [8, 10, 12]. The architectural semantics gives a denotation for each architectural concept and can be embedded in specification templates. These can be realised as a library of macros that expand to the required specification fragments.

OSI (Open Systems Interconnection [4]) is a complex set of standards for layered communications systems. Using *m4,* the author has developed a library of templates to support the specification of OSI standards in LOTOS. As an example, a connection-oriented communications service is one in which a connection must established before data is transferred, and the connection must be broken on completion of data transfer. Interactions with a service take place using service primitives, which in the following look like function calls. A typical service of this type might be described as follows:

```
include(coserv.m4)                  # import connection-oriented service library

co_serv_spec(                       # generate connection-oriented service specification
  co,                               # name of channel for communicating with service
   Conn_Request(Addr1,Addr2)        # connection attempt from address 1 to address 2
   Conn_Indication(Addr1,Addr2)     # notification of connection attempt
   Conn_Response(Addr)              # acceptance of connection at address
   Conn_Confirm(Addr)               # confirmation of connection set-up
   Data_Request(Data)               # request to transfer normal data
   Data_Indication(Data)            # notification of normal data arriving
   Exped_Request(Data)              # request to transfer expedited (priority) data
   Exped_Indication(Data)           # notification of expedited data arriving
   Disconn_Request(Reason)          # disconnection for given reason
   Disconn_Indication(Reason)       # notification of disconnection
)
```

The top-level macro *co_serv_spec* relies on about 70 other macros in 530 non-comment lines of *m4*. The macro library is quite intricate and exploits nearly all of the techniques described in Section 3. The result of the call to *co_serv_spec* is about 490 lines of LOTOS – an expansion of roughly 1:35. Some details of the translation can be found in [10].

Another template library was developed for flexible creation of telecommunications services [9]. Services are defined as combinations of service facilities – the basic functions of a service provider. Service facilities are characterised by their interaction pattern and the way in which multiple instances of facilities are ordered. Combinations of service facilities reflect the kinds of service behaviour found in OSI, from simple interleaving of data transfer in each direction to complex interdependencies between connection establishment and data transfer. For service creation, the macro library defines a special-purpose language called SAGE (Service Attribute Generator) to describe telecommunications services. About 35 macros in 650 non-comment lines of *m4* carry out the translation from SAGE to LOTOS. The macro library is relatively complex, but more in terms of the LOTOS to be generated than the *m4*. As an example of service declaration using SAGE, the following describes a simple datagram service for connection-less message transfer:

```
global(                                 # define global service behaviour
  cl,                                   # name of channel for communicating with service
    forall(                             # applies to all instances of service facility
      facility(                         # declare service facility
        12,                             # direction is from user 1 to user 2
        provider_confirmed,             # service provider confirms delivery
        reliable,                       # data transfer is reliable
        Datagram(Addr,Addr,Data)        # datagrams carry from address, to address and data
      )
    )
)
```

This description results in about 30 lines of LOTOS. The description of more complex services results in roughly a 1:25 expansion from *m4* to LOTOS. See the paper cited above for a fuller explanation of SAGE and more examples of its use.

The final application of *m4* to be discussed is specifying digital logic [9, 11]. Although the macro library for this again defines a surface syntax, the layer of syntactic sugar is rather thin. The library mainly acts as a repository of LOTOS specifications for digital logic components, earning the name DILL (Digital Logic in LOTOS). In fact, many of the macros are unparameterised and generate fixed text. However, parameterisation is taken advantage of when it comes to defining *n*-input logic gates; these can be described generically and then instantiated with the number of inputs and the kind of logic function to be performed (e.g. a two-input 'nand'). The library contains about 60 macros in 650 non-comment lines of *m4*. The following example describes a 'not and' circuit that realises the logical function $\neg IP_1 \wedge IP_2$:

```
circuit(                                # define circuit
  'NotAnd2 [Ip1, Ip2, Op]',             # functionality of LOTOS specification
  'hide NotIp1 in                       # behaviour of LOTOS specification …
    Inverter [Ip1, NotIp1] |[NotIp1]| And2 [NotIp1, Ip2, Op]
  where                                 # give subsidiary declarations
    Inverter_Decl                       # inclusion of inverter gate declaration
    And2_Decl                           # inclusion of two-input and gate declaration
')
```

The LOTOS expressions here are unimportant. The *circuit* macro wraps up the specification of a whole circuit. *Inverter_Decl* and *And2_Decl* are references to component declarations in the library whose specifications are to be automatically included in the generated LOTOS. The use of *m4* in the macro library is quite straightforward, the only slightly tricky aspect being to make sure that each required component is specified only once in the generated LOTOS. The library is described fully in [11].

4.2 Other Applications

Some systems programs such as *sendmail* and application programs such as FrameMaker™ make use of *m4* as a preprocessor. However, the possible uses of *m4* are very wide. The following applications were developed by Richard Bland (University of Stirling [3]).

Creating pictures using LaTeX is tedious and error-prone. Some of the problems stem from using literal coordinates and the difficulty of piecing together a complex picture from self-contained parts. Although some of these problems can be solved directly in LaTeX, picture definitions still suffer from an ungainly syntax. *m4* can be used to provide a more pleasant picture-drawing syntax that translates into LaTeX. [2] describes a macro package that provides facilities for drawing rectangles, ovals, circles, arrows, etc. A useful bonus is that standard *m4* features allow symbolic coordinates and easy arithmetic calculations using coordinates.

A collection of files or documents will typically have some kind of interdependency, often hierarchic. It is useful to able to process the files according to their relationships, say to list them in a spe-

cific order. The relationships could be captured in a separate file (much as the UNIX utility *make* does). However, this removes the close link between the definition of a file and its relationships. A preferable solution is to include the relationships directly in the files as part of their 'documentation' – an approach that is reminiscent of Donald E. Knuth's literate programming. The adopted approach declares these links using *m4* macros as comments in the files. (Of course, the user need only know that these are stylised comments and need not know they are macros.) The macros allow *m4* to recursively traverse the tree of files and extract the necessary information in the right order. The documentation in [1] was produced using this technique.

A similar problem arises where it is necessary for a file to refer to a named section of another file. This file cannot simply be included as only part of it is required, and the section should not be literally copied in case there are later changes to the file. A set of *m4* macros can be defined for extraction and inclusion of the sections as required. In a particular application, the sections are tables produced by a statistical package. The tables are referenced by name, type and file using a few *m4* macros of the general form *TableType(TableName,FileName)*. The advantages of the approach are easy inclusion (or later exclusion) of particular tables, and guaranteed consistency of the extracts.

5 Conclusion

A summary of *m4* has been given along with some basic examples. A cookbook of techniques has been described, allowing surprisingly sophisticated macros to be developed. Many of the 'recipes' support a style that is reminiscent of functional programming. Some applications of the approach have been briefly described for the specification of OSI standards, telecommunications service creation and digital logic circuits. It is hoped that this paper will stimulate interest in *m4* and its wider application in software engineering.

Acknowledgements

The author is grateful to Richard Bland (University of Stirling) for careful and helpful comments on a draft of this paper.

A Implementation of List Processing Functions

As in Section 3.14, lists are used in the form *[…;…]*. There is no check on the syntactic integrity of lists (e.g. that opening and closing brackets are balanced).

The list constructor *Cons* prefixes an element to a list:

```
define(Cons,
  `ifelse($2,[],
    [$1],
    [$1;substr($2,1,decr(len($2))))')
```

If the existing list is empty then a one-element list is made, otherwise the element is prefixed to the rest of the list[6]. The opening bracket of the list is supplied literally and is omitted when extracting the rest of the string.

Head and tail are common operations on lists. It is convenient to define the supporting macro *HeadEnd* first to determine the end position of the head element in the list. Because lists may be nested, *HeadEnd* is defined in terms of an auxiliary macro *HeadEndAux* that takes the list, the position to begin searching for the head element, and the current level inside the list.

[6] Here and elsewhere, an expression like **substr**($2,1) should extract the rest of the list. Unfortunately the version of *m4* available to the author fails to do this for strings longer than 128 characters.

```
define(HeadEnd,‘HeadEndAux($1,1,0)’)

define(HeadEndAux,
  ‘define(‘$0Char’,
    substr($1,$2,1))ifelse($0Char,[,
      ‘HeadEndAux($1,incr($2),incr($3))’,
      $0Char,],
        ‘ifelse($3,0,
          decr($2),
          ‘HeadEndAux($1,incr($2),decr($3))’)’,
        $0Char,;,
          ‘ifelse($3,0,
            decr($2),
            ‘HeadEndAux($1,incr($2),$3)’)’,
          ‘HeadEndAux($1,incr($2),$3)’)’)
```

Initially the head element is scanned from just after the opening bracket. If a further opening bracket is found, the list level is incremented; the matching closing bracket must be found before the head element can be considered as terminated. For all elements, a semicolon or the end of the list terminates the head element. The introduction of *HeadEndAux* avoids argument stack overflow for long lists using the technique described in Section 3.11. The auxiliary parameters are the list position and list level, to be updated as required on a recursive step. The current character *$0Char* is named as a local variable using the technique described in Section 3.3.

Now the head and tail of a list can be defined fairly directly:

```
define(Head,‘substr($1,1,HeadEnd($1))’)

define(Tail,
  ‘define(‘$0Pos’,
    HeadEnd($1))define(‘$0Len’,
      len($1))ifelse($0Pos,eval($0Len-2),
      [],
      ‘[substr($1,eval($0Pos+2),eval($0Len-$0Pos-2))’)’)
```

The head element begins just after the opening bracket and finishes as calculated by *HeadEnd*. The head of an empty list is returned as the empty string. The tail begins after the head element and its the semicolon unless there is just one element (i.e. the head element finishes just before the closing bracket).

The concatenation of two lists requires a check to see if one is empty; if so the result is the other list. Otherwise the lists are concatenated by stripping off the first closing bracket and the second opening bracket, inserting a semicolon in between:

```
define(Concat,
  ‘ifelse($1,[],
    $2,
    $2,[],
      $1,
      ‘substr($1,0,decr(len($1)));substr($2,1,decr(len($2)))’)’)
```

To avoid excessive recursion causing argument stack overflow, the length of a list is calculated using an auxiliary function that updates the current list length given as parameter prior to recursing:

```
define(Length,‘LengthAux($1,0)’)

define(LengthAux,
  ‘ifelse($1,[],
```

```
    $2,
    'LengthAux(Tail($1),incr($2))')')
```

The next macro reverses the order of elements in a list. An auxiliary macro takes the partially reversed list as parameter:

```
define(Reverse,'ReverseAux($1,[])')

define(ReverseAux,
  'ifelse($1,[],
    $2,
    'ReverseAux(Tail($1),Cons(Head($1),$2))')')
```

It is often convenient to apply a macro to all elements of a list. *AppList* does so without regard to the expansion of the macro, *MapList* rebuilds the list by replacing each element with the result of the mapping:

```
define(AppList,
  'ifelse($2,[],
    ,
    '$1(Head($2))''AppList('$1',Tail($2))')')

define(MapList,'MapListAux('$1',$2,[])')

define(MapListAux,
  'ifelse($2,[],
    $3],
    'ifelse($3,[,
      'MapListAux('$1',Tail($2),$3$1(Head($2)))',
      'MapListAux('$1',Tail($2),$3;$1(Head($2)))')')')
```

AppList and *MapList* take as parameters the (quoted) name of a macro to be applied to each element and a list. *AppList* repeatedly calls the macro with each element of the list as parameter. The macro might simply have a side-effect or may expand to text. In case of the latter, the output of the macro is separated by empty quotes from the recursive call of *AppList*. *MapList*, however, uses the result of the macro call to recreate the list element. *MapListAux* takes as auxiliary parameter the current list value (initially just an opening bracket). The first element is just appended to the list, later elements are prefixed by a semicolon. At the end of the list the closing bracket is added.

References

[1] Richard Bland. Progress with logging subsystem O: A prototype knowledge source. Technical Report CSM-071, Department of Computing Science, University of Stirling, UK, April 1991.

[2] Richard Bland. Relative moves in LaTeX pictures. *TUGBoat*, 14(4):433–437, December 1993.

[3] Richard Bland. Applications of *m4*. Private communication, September 1994.

[4] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Basic Reference Model*. ISO/IEC 7498. International Organization for Standardization, Geneva, Switzerland, 1984.

[5] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. ISO/IEC 8807. International Organization for Standardization, Geneva, Switzerland, 1989.

[6] Brian W. Kernighan and P. J. Plauger. *Software Tools*. Addison-Wesley, Reading,

Massachusetts, USA, 1976.

[7] Brian W. Kernighan and Dennis M. Ritchie. The *m4* macro processor. Technical report, Bell Laboratories, Murray Hill, New Jersey, USA, 1977.

[8] Kenneth J. Turner. An architectural semantics for LOTOS. In Harry Rudin and Colin H. West, editors, *Proc. Protocol Specification, Testing and Verification VII*, pages 15–28. North-Holland, Amsterdam, Netherlands, 1988.

[9] Kenneth J. Turner. An engineering approach to formal methods. In André A. S. Danthine, Guy Leduc, and Pierre Wolper, editors, *Proc. Protocol Specification, Testing and Verification XIII*, pages 357–380. North-Holland, Amsterdam, Netherlands, June 1993.

[10] Kenneth J. Turner. Relating architecture and specification. September 1994. Forthcoming.

[11] Kenneth J. Turner and Richard O. Sinnott. DILL: Specifying digital logic in LOTOS. In Richard L. Tenney, Paul D. Amer, and M. Ümit Uyar, editors, *Proc. Formal Description Techniques VI*, pages 71–86. North-Holland, Amsterdam, Netherlands, 1994.

[12] Kenneth J. Turner and Marten van Sinderen. LOTOS specification style for OSI. In Jeroen van de Lagemaat and Tommaso Bolognesi, editors, *The LOTOSPHERE Project*. Kluwer, 1994. Forthcoming.