User's Guide to the M5 Macro Language: 2^{nd} edition

A. Dain Samples*
Computer Science Division–EECS
University of California at Berkeley
Berkeley, CA 94720

September 8, 1992

Abstract

M5 is a powerful, easy to use, general purpose macro language. M5's syntax allows concise, formatted, and easy to read specifications of macros while still giving the user control over the appearance of the resulting text. M5 macros can have named parameters, can have an unbounded number of parameters, and can manipulate parameters as a single unit. M5 provides separate macro name spaces called 'pools' that simplify the creation and management of context-dependent macros and dynamic macros (macros defined by macros based on the input). Several examples demonstrate m5's power and flexibility, including a random string generator that uses BNF grammars to specify the form of the strings; an implementation of a Turing machine; and a demonstration of how m5 can process LaTeX input and programming language source in the same file to simplify code maintenance and documentation.

*The author is can be reached at the University of Cincinnati, Dept. of ECE, ML#30, Cincinnati, Ohio 45221-0030; or at Dain.Samples@uc.edu.

Contents

1 Introduction

Macro processors are an extremely useful means of transforming text, and are a standard part of many language systems. For example, the C language system has a macro preprocessor as a *de facto* part of the language standard. However, the general UNIX[1] user does not have access to a powerful general purpose macro processor. The macro processors distributed with UNIX (*cpp* and *m4*) do not qualify on either point.

The phrase 'powerful, general purpose macro processor' should mean that the processor implements a robust set of functions[2]. The macro language should implement scopes, much as modern programming languages allow name overloading through scopes. Debugging should be easy. The macro language should also give the programmer great flexibility in dealing with macros with variable numbers of parameters. Recursive macros should be easy to write. Perhaps most importantly, the user should find it easy to read macro definitions the day after they are written.

M5 implements a large set of pre-defined macros that simplify the manipulation of text. It has a powerful scoping mechanism that allows the user to define 'pools' of macros (think of them as a file system subdirectory) and a macro name search stack of such pools. For instance, in a programming language source file, a user defined macro might expand one way if it occurs in a global environment, and another in local environments. M5 can be used to track nesting of function declarations, begin/end blocks, or nesting of C's curly braces. Pools of macros can be defined for the various contexts, and the appropriate pool used to control macro expansion in the desired context. Pools are an associative data structure mapping names to definitions and so are useful for collecting and manipulating many different kinds of data. The example in appendix C.4 demonstrates the usefulness of pools: in the implementation of the m5 processor, the names of parameters to macros can be referenced both in LaTeX documentation and in C code using the same name, but each expands into something useful for that particular context; the LaTeX version of the macros sets the parameter names in a particular font, while the C version expands to the code to access the runtime data structure containing the value of the indicated parameter.

M5 debugging allows the user to track easily each step of a macro's expansion, and control how much debug output is desired.

M5 macros can handle large numbers of named or anonymous parameters, allowing recursive macros to be defined easily and naturally. M5 has many control constructs to simplify the user's task, including `Case`, `Match`, and several flavors of conditional tests.

M5 can be configured to scan various kinds of source files; it understands C, C++, and Ada commenting conventions. The user can disable macro definitions in contexts where their expansion would be a nuisance.

Finally, m5 allows the user precise control over the text that appears as the result of macro expansion without making the definition of the macro difficult to read. Macro definitions can be textually formatted to reflect logical structure without sacrificing the readability of the expanded text.

The latest version of m5 is available via anonymous ftp from the University of Cincinnati at `thor.ece.uc.edu:pub/dain/m5`. A PostScript^{TM} version of this document is available via anonymous ftp from the Sequoia project at the University of California at Berkeley, as well as via

[1] Unix is a trademark of AT&T.

[2] At a minimum, it should be Turing equivalent; see appendix C.6 for a proof that m5 is Turing equivalent. I assert that *cpp* is not Turing equivalent, and have not bothered to check *m4*, though I suspect it is.

anonymous ftp from babbage.ece.uc.edu.

The following sections describe m5 in more detail. There has been a strenuous effort to keep this documentation current with the implementation. The source to m5 is written in C and m5; these m5 macros automatically extract the cross reference manual in section B. Please report any and all errors you may discover in the manual or m5 to the author.

2 Summary of M5 Features

This section summarizes the attractive features of m5. For contrast, it points out the major differences between m5 and m4. While the basic paradigm is the same (macro invocation begins with the recognition of a macro word optionally followed by a parenthesized list of comma separated parameters), there are many differences[3].

Keywords, Predefined Macros: All m5 predefined macros begin with a capital letter. (`Eval`, `Include`, `Ifdef`, etc.). This is necessary if you're using m5 in conjunction with other preprocessors (e.g., cpp) which also use words such as define, include, and ifdef. However, user defined macros may begin with any case of alphabetic character (including underbar and colon) followed by more of the same or digits. A small number of non-alphanumeric characters can be defined as single-character macros. See section 4.

Debugging: M5 allows you to specify several different levels of debug output, from printing the names of recognized macros, to showing the resulting text of each macro expansion. When macros are not behaving as expected, the insertion of a simple call on the Debug macro will allow you to quickly determine where the problem lies. See Section 10.

Comments, Cpp, C, and C++: In addition to m4's '#' comment convention, m5 supports several other ways for commenting m5 code. These additional comments provide control over white space in output, allow the comment character to be tailored to new environments (Section 11), and permit scanning text that has its own special comment conventions. M5 works very well with *cpp*, C, Ada, and C++. See Section 5.

Scoping: Macros are defined in *pools* and macro name resolution is determined by the state of the *pool stack* (see Section 6). This permits macro definitions to be visible only when the user wants them to be visible. It is easy, for instance, to define macros that are visible only when scanning files with a certain suffix, or only after certain keywords have been identified in the text. This is totally under the macro writer's control; Section C.1 details an example.

Environment: M5 reads all variables in the environment and defines them in a special 'Environment' pool. For example, a macro could access the environment variable HOME to define file names.

```
Define(‘myFile’,HOME~Environment‘/bin/src/f1.c’)
```

where `HOME~Environment` specifies that the macro is to be found in the macro pool `Environment`.

Initialization: M5 reads .m5rc in the current directory as the first input file.

Extended parameter specifications: M5 allows an unlimited number of parameters, and has concise notations for manipulating them. A macro's formal parameters can be named, making it much easier to write, read, and debug macro definitions. See Section 4.4.

[3]Most of these comments are with respect to m4 as it existed about 1986-1988.

Arithmetic expression evaluation: M5 adopted m4's method of arithmetic expression evaluation, but increased the number of available numeric functions, and added number formatting capabilities. See Section 7.

Files: Files in m5 can be named, opened, and written to directly. See Section 9.

3 Command line options

–D turns on debugging to Std mode (see Section 10).

–D0 turns debugging mode off.

–Define(M[,V]) Define the macro M to have the value V, or the empty string if there is no V.

– Read the standard input at this point.

Note that filenames and options are activated in the order they appear on the invocation line. So

```
% m5 one -D two -D0 - three "-Define(state,last file)" four
```

reads file `.m5rc` (if it exists), reads file `one`, turns on debugging, reads file `two`, turns off debugging, reads the standard input file to EOF, reads file `three`, defines the macro 'state' to have the value '`last file`', and then reads file `four`. There is no m5 file extension: e.g., m5 does not attempt to find file 'one.m5' if it can't find file 'one'.

4 Defining Macros

The basic processing paradigm of m5 should be familiar to users of *cpp* and m4. For instance, to define a macro:

```
Define(‘macro’,‘100’)
```

Then wherever the string 'macro' is recognized, it will be replaced with the string '100'. Another way of saying this is that the string 'macro' is *expanded* into the string '100'. Macros can have parameters which can be replaced when the body of the macro is expanded:

```
Define(‘helloTo(name)’,‘Hello, $(name)!’)
```

causing '`helloTo(John)`' to expand into '`Hello, John!`'.

4.1 Macro Names

Macro names follow traditional identifier conventions (leading alphabetic character followed by alphanumerics) with underbar '_' and colon ':' considered alphabetics. The user can also define macros whose name is a single non-alphanumeric character *that is not used in any other way.* For instance, the curly braces '{' and '}' are not used in any special way by m5, so they can be defined as macros. However, the right and left parentheses by default delimit macro argument lists. As

long as they are so designated, they cannot be used as macro names. In the following example, notice that nested quoting is used to inhibit recognition and expansion of macros[4].

```
>Define(‘{’,‘open bracket ‘{’’)
>Define(‘}’,‘‘}’ close bracket’)
>{ hello }
open bracket { hello } close bracket
```

4.2 Order of Processing

M5 processes text in four major stages, or modes. (There are some minor modes discussed on page 25, but they are not relevant in the current context.)

Scanning: The input text is scanned for macro name.

Argument collection: Once a macro name has been recognized, if that macro was defined to accept arguments, and if its name is followed immediately by a left parenthesis, then the arguments are collected in scanning mode and pushed onto a stack (M5 uses positional arguments only).

Parameter substitution: Once all of the arguments for a macro are collected, then the body of the macro—also called the replacement text—of the macro is scanned for parameter specifications. Each parameter specification is replaced by the corresponding text from the argument stack.

Macro expansion: Once the parameter specifications in the body of the macro have been replaced with the corresponding arguments from the argument stack, the resulting text is stuffed back into the input stream and reread in scanning mode.

4.3 Defining Macros

Every macro in M5 is processed in the manner outlined above, including the predefined macro that defines macros. This can lead to some confusion for the naive user. For example:

```
>Define(x,y)line 1
line 1
>Define(x,z)line 2
line 2
>line x y z
line z z z
```

In the `Define` on the second line, the ‘x’ is recognized as a previously defined macro and is thereupon expanded to its value, ‘y’. Then ‘y’ and ‘z’ are passed as the two parameters to the `Define` macro, and a new macro ‘y’ is defined with value ‘z’. The macro ‘x’ still has the value ‘y’. To avoid this form of error (if indeed it was not the desired action), always surround things you mean literally with balanced single quotes:

[4] We also introduce the method of displaying examples used throughout the rest of this document. Each line of input to m5 has a prompt-character > in the first column with the resulting expansion text on the immediately following lines. When using m5 interactively, it does not prompt the user, since m5 will be used almost exclusively to process files or act as a filter.

```
>Undefine(‘x’)Undefine(‘y’)Undefine(‘z’)
>Define(‘x’,‘y’)line 1
line 1
>Define(‘x’,‘z’)line 2
line 2
>line x y z
line z y z
```

`Append` appends text to an existing macro, if it exists, otherwise it acts like `Define`. This is useful for accumulating strings but it's hard to imagine a situation where defining macros piecewise could be particularly useful. Again, be careful:

```
>Define(x,‘junk’)
>Append(x,‘ more junk’)
>junk x
 more more more more more more more more ...
```

The string ‘`junk junk more junk` may have been expected, but `Append` did not append to the macro named `x` but to one named `junk`, which was the value of x. Quote those literals!

```
>Define(‘x’,‘junk’)
>Append(‘x’,‘ more junk’)
>junk x
junk junk more junk
```

4.4 Parameter Specification

During macro expansion, parameters are substituted into the body of a macro by use of parameter specifications. The simplest parameter spec is a $ character followed by a single digit n, which refers to the n^{th} parameter (`$0` refers to the macro name itself). If there are more than nine parameters, then the multi-digit number needs to be enclosed in parentheses. Parameters can be named in the definition of a macro, but they are not themselves macros! In the following, note that the (potential) macro name `two` does not interfere with the parameter named `$(two)`, and that parameter names are not subject to macro expansion. In particular, we do not get an error message that the parameter `$(infinity)` is not defined.

```
>Define(‘two’,‘infinity’)
>Define(‘testing(foo,two,baz)’, ‘Here is $(foo), $(two), and $(baz).’)
>testing(two,one,zero)
Here is infinity, one, and zero.
```

Parameter specifications are recognized, parsed, and replaced during the parameter substitution phase (see Section 4 and Section 11). During text scanning mode, every time a macro name is recognized, its parameters are collected on the argument stack. Parameters are read in text scanning mode also, so macros in argument lists are expanded before the invocation of the surrounding macro. When the closing right parenthesis character of an argument list is read, then all parameter specifications in the definition/body of the corresponding macro are replaced with their values. This newly constructed string is then “pushed back” into the input stream, and text scanning begins again.

4.4.1 Delayed Parameter Substitution

Parameter specifications are identified by a special character that you can specify (the default character is '$'; see section 11). All $s are passed through during text scan (no substitutions). A single $ with no valid parameter specification following it is simply passed through during macro expansion. A double $$ is replaced at macro expansion time with a single $ and is not rescanned ('$$1' is an easier way to delay parameter substitution than ''$'1'). In general, all multi-character strings of $$..$$ whose lengths are greater than one are replaced with another string of $s that has one less $. This simplifies the specification of macros with delayed evaluation (e.g., a macro that is defined within the expansion of the body of another macro) by allowing you to simply count dollar signs instead of pairs of dollar signs or nested quotes. For example, if you were to define macros that defined macros that defined macros, etc., then to have the second generation macro's parameters protected would be $$1, third generation's $$$1, etc.

```
>Define(‘DefineMac(m1,p1,p2)’,
>       ‘Define(‘$(m1)($(p1),$(p2))’,‘One: $$($(p1)) and Two: $$($(p2))’)’)
>DefineMac(‘A’,‘xyz’,‘abc’)
>DefineMac(‘B’,‘pqr’,‘mno’)
```

is equivalent to

```
>Define(‘A(xyz,abc)’,‘One: $(xyz) and Two: $(abc)’)
>Define(‘B(pqr,mno)’,‘One: $(pqr) and Two: $(mno)’)
```

The general syntax of a parameter specification is describe by the following extended BNF description. The names of m5-recognized special characters (see section 11) are in upper case, all non-terminals in the grammar are in lower case, and other terminal symbols are capitalized.

```
parameter ::= DOLLAR ( smplparm | parm | parmcnt )
smplparm  ::= Digit
parmcnt   ::= lsep STAR rsep
parm      ::= lsep [phead] parmid [ptail] rsep
lsep      ::= LPAR | LQUOTE
phead     ::= COMMA [STAR]
parmid    ::= Name | Digit+
ptail     ::= (COMMA [STAR] | DASH) [parmid] [sepspec]
rsep      ::= RPAR | RQUOTE
sepspec   ::= COMMA [Char]
```

Here is a summary by example of the effect of various parameter specifications using m5's default character set. Here == means textually equivalent.

```
$(2,4)  == $2,$3,$4       WITH THE COMMAS!
$(2-4)  == $2$3$4         WITHOUT the commas
$(,2,4) == ,$2,$3,$4      insert leading comma,  but
                only if parameter 2 exists (it may be empty,
                but it must exist)
```

```
$(3,)   == $3,$4,...,$n   for all n parameters
$(,3,)  == ,$3,$4,...,$n  for all n parameters
$(,*3,) == ,$3,$4,...,$n  for all n parameters, but the
                leading comma is inserted only if at least one
                of the parameters is non-null
$(,3,*) == ,$3,$4,...,$n for all n parameters, but only the
                non-null parameters are in the list
$(3-)   == $3$4$5...$n for all n parameters
$(*)    == n the number of parameters to this macro invocation.
$‘,2,4’ == ,‘$2’,‘$3’,‘$4’ to prevent another evaluation of
                the parameters
$‘5-’   == ‘$5’‘$6’‘$7’...‘$n’
$(2,4,_)== $2_$3_$4 introduces a new separator character
$(2,4,) == $(2-4) == $2$3$4 the new separator is null
$(2,, ) == $2 $3 $4 ... $n  the new separator is a blank
```

The following sections give more details on the use of these parameter specifications.

4.4.2 Parameter Concatenation

There are parameter specifications which permit manipulation of the parameter list as a whole or in parts. A macro's parameters can easily be passed to another macro without having to know how many parameters were being passed—an essential feature for writing recursive macros.

For instance, to have all of a macro's parameters concatenated and inserted into the replacement text:

```
>Define(‘cat()’,‘$(1-)’)
>cat(a,b,c,d,e)
abcde
```

4.4.3 Parameter List Construction

To write a macro that expands to the list of comma-separated parameters:

```
>Define(‘catwcommas()’,‘$(1,)’)
>catwcommas(a,b,c,d,e)
a,b,c,d,e
```

This is useful for defining recursive macros, or macros that pass some subset of their parameters to another macro:

```
>Define(‘cattwo()’,‘cat($(2,))’)
>cattwo(a,b,c,d,e)
bcde
>cattwo(a,b,,d,e,,,h)
bdeh
```

4.4.4 Null (Empty) Parameters

To make a new list of comma-separated parameters while ignoring any that are empty is accomplished by putting an asterisk after the separating comma in the parameter specification:

```
>Define(‘list()’,‘one=$1 two=$2 three=$3 four=$4 rest=$(5,)’)
>Define(‘catnonulls()’,‘list($(1,*))’)
>catnonulls(a,b,,d,e,,,h,,j,,l,,,o)
one=a two=b three=d four=e rest=h,j,l,o
```

It is important to recognize the difference between an empty parameter and a non-existent parameter. If a macro `M` is invoked with no following parentheses, then it is being passed 0 parameters. If it is invoked with empty parentheses, i.e. `M()`, then it is being passed one empty parameter. The special parameter specification `$(*)` expands into the actual number of parameters passed to the macro.

4.4.5 Parameter List Construction Using Different Separators

To put a different separator other than a comma between the items in the parameter list:

```
>Define(‘hyphenate(args)’,‘$‘args,*,-’’)
>hyphenate(‘a’,one,,two,,three)
a-one-two-three
>Define(‘hyphenate2(args)’,‘$‘,args,*,-’’)
>hyphenate2(a,one,,two,,three)
-x-y-one-two-three
```

4.4.6 Collecting Parameters

When collecting parameters, matching quote characters ‘ and ’ and matching parentheses group the text. In the following example, a macro defines a macro. It also demonstrates that whitespace before a macro argument is ignored unless quoted.

```
>Define(‘MyDefine(name,stuff)’,
        ‘Define(‘$(name)(x,y,z)’,
                ‘Here is: $(stuff) to $$(x), $$(y), and $$(z)’)
Macro $‘name’ defined.
’)
>MyDefine(‘Hello’, ‘Goodbye’)
Macro Hello defined.
>Hello(Tom,Dick,Harry)
Here is: Goodbye to Tom, Dick, and Harry
>Hello(    Jane,    Vic,‘      Sally’)
Here is: Goodbye to Jane, Vic, and       Sally
```

4.4.7 Delaying Parameter Evaluation

Once the arguments to a macro have been collected, it may be necessary to keep them from being recognized as macros in other contexts. This can be done by putting quotes around the parameter specification `‘$(p)’`. However, this won't work if you want to create a list of parameters, and you want each parameter to be quoted individually. For that, use the quote characters instead of parenthesis around the parameter name: `$‘p,’`.

```
>Define(‘MQ()’,‘X$‘,1,’’)
>Define(‘MNQ()’,‘X$(,1,)’)
>Define(‘a’,BIGA)
>Define(‘b’,BIGB)
>Define(‘c’,BIGC)

>MQ(‘a’,‘b’,‘c’)
X,a,b,c                 // a,b,c not expanded because of $‘,1,’
>MNQ(‘a’,‘b’,‘c’)       // a,b,c expanded because of $(,1,)
X,BIGA,BIGB,BIGC
```

4.5 Counting Parameters

We begin a running example to illustrate other useful parameter specification features. We will create a macro that counts the number of *non-empty* parameters passed to a macro. Through the development of this macro, we will also develop familiarity with several m5 idioms.

Here is a first cut at defining a macro to count the number of non-null parameters:

```
>Define(‘cntParms()’,‘$(*)’)
>Define(‘cntNonEmpty()’,
>       ‘args: $(*)   not empty: Incr(cntParms(one,$(1,*)),-1)’)
>cntNonEmpty(a,b,,d,e,,,h,,j,,l,,,o)
args: 15   not empty: 8
>cntNonEmpty()
args: 1   not empty: 1
>cntNonEmpty
args: 0   not empty: 1
```

The results of the last two examples should raise your eyebrows. If you watched how m5 processes the text as it goes (and you can; see Section 10 on debugging), you would see that the macro `cntNonEmpty` expands to:

```
args: 0   not empty: Incr(cntParms(one,),-1)
```

The macro `cntParms` is therefore being passed two parameters, one of them empty. This makes it clear that an empty parameter is different than a non-existent parameter. The solution is to remove the comma just before the parameter specification in the body of `cntNonEmpty`.

4.5.1 Omit leading comma if no parameters

The parameter specification can direct that a leading comma is to be emitted only if the indicated parameters exist; simply put a comma in the specification just before the parameter name/number:

```
>Define(‘cntNonEmpty2()’,
>       ‘args: $(*)   not empty: Incr(cntParms(one$(,1,*)),-1)’)
>cntNonEmpty2
args: 0   not empty: 0
>cntNonEmpty2()
args: 1   not empty: 1
```

Again, the last example isn't quite what we expected. There is one empty argument being passed in, but `cntParms` is expanding to '2', not '1'. The problem is that a leading comma is preceding the arguments passed to `cntParms` even when all the parameters described by `$(1,*)` are empty. In other words, the parameter specification in `cntNonEmpty2` will emit the leading comma whenever there are any parameters, even when all them are empty.

4.5.2 Omit leading comma if all parameters are null

An asterisk after the first comma in the parameter specification directs that the leading comma is to be emitted only if there is at least one non-empty argument:

```
>Define(‘cntNonEmpty3()’,
>       ‘args: $(*)   not empty: Incr(cntParms(one$(,*1,*)),-1)’)
>cntNonEmpty3
args: 0   not empty: 0
>cntNonEmpty3()
args: 1   not empty: 0
```

And now we have a `cntNonEmpty` macro that does what the name implies. Every one of these additions to the parameter specification have been necessary at some time or another at the cost of perhaps stressing the design principle of simple, readable, and understandable notation.

4.5.3 Parameters with Embedded `PARMSEP` Characters

However, we have two final problems to address that do require 'one more parameter specification feature'. Observe what happens when we define a macro 'a' to be the string 'x,y'

```
>Define(‘a’,‘x,y’)
>cntNonEmpty3(a)
args: 2   not empty: 2
```

Obviously, the problem here is that 'a' is being expanded before being passed to the `cntParms` macro. So we quote it to keep it from being expanded:

```
>cntNonEmpty3(‘a’)
args: 1   not empty: 2
```

This still isn't quite right. The macro 'a' is still being expanded before we want it to be. We have to design a `cntNonEmpty` macro to make sure it doesn't expand its parameters. Our first attempt almost works:

```
>Define(‘cntNonEmpty4()’,
>       ‘args: $(*)   not empty: Incr(cntParms(one‘$(,1,*)’),-1)’)
>cntNonEmpty4(‘a’)
args: 1   not empty: 0
>cntNonEmpty4(‘a’,‘a’,‘a’)
args: 3   not empty: 0
```

Obviously, all three parameters are non-empty. The problem is that encapsulating the entire parameter list inside quotes makes `cntParms` think it is getting a single parameter '`one,a,a,a`' (with embedded commas). We want each parameter surrounded by its own single quotes. This is effected by using balanced single quotes in the parameter specification instead of parentheses:

```
>Define(‘cntNonEmpty5()’,
>       ‘args: $(*)   not empty: Incr(cntParms(one$‘,1,*’),-1)’)
>cntNonEmpty5(‘a’)
args: 1   not empty: 1
>cntNonEmpty5(‘a’,‘a’,‘a’)
args: 3   not empty: 3
```

Most of the above examples have used digits for the parameters for simplicity, but named parameters could have been used:

```
>Define(‘cntNonEmpty5(args)’,
>       ‘args: $(*)   not empty: Incr(cntParms(one$‘,args,*’),-1)’)
>cntNonEmpty5(‘a’)
args: 1   not empty: 1
>cntNonEmpty5(‘a’,‘a’,‘a’)
args: 3   not empty: 3
```

5 Comments

There are two kinds of m5 comments. The first corresponds to the m4 # comment. When encountered in input text (that is, in scanning mode, *not* in the expansion of a macro) everything from it to end of line (EOL) is simply ignored *but* passed through as is: no macro expansion occurs in the body of the comment. When encountered during macro expansion (that is, the # is in the body of the macro), everything from the # to EOL is ignored and *not* passed through. This is not particularly pleasant, useful, or understandable. For example, when using m5 as a preprocessor on Makefiles, m4's rules make it next to impossible to write macros that emit Makefile comments. (And if you think you can write macros without providing documentation, please don't use m5.)

Presumably, m4 made '#' a comment character so as not to interfere with *cpp* processing. Even so, that turns out to be a sub-optimal design choice. For instance, it complicates generating *cpp* macro definitions. Another complication arises because m4 doesn't know about *cpp*'s line

continuation convention. Not all of a *cpp* command can be hidden inside a comment: you have to watch for *cpp* lines (those that begin with '#') that are continued across line boundaries.

Hence the addition of the '`@`' (at sign) comment to m5: m5 recognizes this comment character before anything else and deletes it and all characters after it up *through* the EOL. Unlike the m4 macro `dnl` which can be used to provide some control over whitespace, '`@`' comments are not macros, but are recognized by the m5 character scanner. A second character following the '`@`' character specifies what to do with characters that follow the EOL.

'`@@`': *All* whitespace characters after the EOL and up to the first non-whitespace are ignored and discarded. This includes tabs and multiple new lines.

'`@{`': Everything is ignored up to the first occurrence of '`@}`'; this is useful for massive comments. All whitespace after the '`@}`' is also ignored.

'`@N`': *All* whitespace after the EOL is replaced with a single new line character.

'`@;`': Acts exactly like '`@@`', but a warning message is issued if '`@;`' occurs in a macro definition. This provides some error checking that every macro definition ends where you think it ends. E.g. "`Define(‘X’,‘junk’@;`" would result in an error message. This also allows m5 command lines to be separated by mostly blank lines from regular text and yet not contribute a lot of blank lines to the resulting output.

'`@`': If the second character is not one of the above, then the '`@`' and all characters up through the EOL are ignored and discarded.

The following example shows how the `@;` comment affects the output.

```
------------------------beginning of file
real stuff
@;
Define(‘something’,‘to be a macro’)@;

something more that is real
-------------------------end of file
```

upon expansion yields as a result;

```
------------------------beginning of output
real stuff
to be a macro more that is real
-------------------------end of output
```

6 Macro Pools

Since working with a flat name space is difficult, m5 implements name-scoping. Macro name-space is organized into pools, where each pool is its own little macro world with its own symbol table and its own sub-pools. Macro names *and pool names* are looked up according to the pool search path defined by `PushPool` and `PopPool` macro calls. When a macro is defined, it is added to the pool on the top of this stack unless there is a pool pathname appended to the macro name. For example,

```
>PushPool(‘ONE’)
>Define(‘m1’,‘in ONE’)
>Define(‘m1~_M5’,‘in _M5’)
>m1
in ONE
>PopPool(‘ONE’)
>m1
in _M5
```

defines an `m1` in pool `ONE`, and an `m1` in pool `_M5`.

M5 pools are similar to a hierarchical file system. In the UNIX shell *csh*, the file search path specifies a sequence of directories in which to search for executable files. In m5, the current search path is defined by the pool stack. The pool stack specifies the sequence of pools in which to search for macro *and pool* names. Pushing a pool on the pool stack makes its macro definitions and pool names visible, and hides all conflicting macro and pool names that may exist in pools lower in the stack.

There are syntactic and semantic differences between the name of a file in a file hierarchy and the name of a macro in m5. To specify file X in a UNIX file hierarchy, one might write: `a/b/c/d/X`. To specify macro `X` in a pool hierarchy, one writes: `X~a~b~c~d`. In UNIX, `/a/b/c/d/X` specifies an absolute (rooted) path for a file; in m5 it would be `X~_M5~a~b~c~d`.

Pool names are also resolved differently from the way directory names are resolved. A directory in UNIX is found only if it is a sub-directory of the current directory. A pool in m5 is found if it is a sub-pool of the pool on top of the pool stack, *or of any pool lower on the stack.* The pool stack is searched from top to bottom to find the first matching sub-pool. In other words, the pool stack establishes a search path not only for macros but also for pools.

Furthermore, macro names and pool names are in different lexical and semantic name spaces, so it is possible to have macros and pools with the same name with no conflict.

Extra arguments to the `PushPool` macro specify the names of pools that must pushed onto the stack before the first named pool is pushed. `PopPool`, on the other hand, takes at most one argument: the name of the pool that should be popped. A warning message is issued if the name of the pool on top of the stack does not match the non-null argument to `PopPool`. When the pool on top of the stack is popped, all the pools named in the corresponding `PushPool` are also popped. For example,

```
>PushPool(‘MY_POOL’) ...
>PushPool(‘AnotherPool’,‘Cpool’,‘PascalPool’,Pool(0),‘ModulaPool’)
```

says to push `ModulaPool`, the *current* pool (`MY_POOL`[5]), `PascalPool`, `Cpool`, and finally `AnotherPool` onto the pool stack to establish name search order. When `PopPool(AnotherPool)` is expanded, then five pools will be popped off the stack, leaving `MY_POOL` on top of the stack. Note that when we push pools on the stack, we are not pushing copies, but references to the actual pools.

```
>PushPool(MY_POOL)
>PushPool(HIS_POOL)
>PushPool(MY_POOL)
```

[5] `Pool(i)` is a macro that expands to the name of the i^{th} pool on the stack, and top-of-stack is the zero-th pool.

```
>Define(‘Random’,‘musings’)
>PopPool(MY_POOL)
>PopPool(HIS_POOL)
>Random
musings
```

A macro's definition can be explicitly fetched from a specific pool:

```
>PushPool(‘ONE’)
>Define(‘macro’,‘just some stuff’)
>PopPool(‘ONE’)
>PushPool(‘TWO’)
>Define(‘macro’,‘different sorts of stuff’)
>PopPool(‘TWO’)
>macro~TWO
different sorts of stuff
>macro~ONE
just some stuff
```

The macro name ‘`macro~ONE` is roughly equivalent to

```
PushPool(‘ONE’)macro‘’PopPool(‘ONE’)
```

but a lot less cumbersome to type and more efficient in execution. However, the two examples are not equivalent, since the former invocation of `macro` *requires* m5 to find it in a pool named `ONE`, while the latter example leaves m5 free to find a definition for `macro` in any pool on the pool stack.

There are several predefined pools and some special pool aliases. The pool `_M5` is the name of the root of the pool hierarchy, it is also contains all predefined m5 macros. `Environment` is a sub-pool of `_M5` and contains all of the environment values as macros. ‘_’ (a single underscore character) is the null pool: the pool in which nothing is defined, nothing is definable, and when pushed onto the stack it stops all name searching at that point on the pool stack. The pool stack is defined to always have the null pool as the (non-poppable) bottom-most pool on the stack. At startup, the root pool `_M5` is the only other pool on the m5 pool stack.[6]

M5 defines two pool name aliases. `_TOS` is an alias for the name of the pool currently sitting on the top of the pool stack; and `_parent` is an alias for the parent of a pool in the hierarchy. The root pool `_M5` is its own parent, as is the null pool _. Note that `_M5`, _, and `_TOS` are absolute; that is, given a pool stack they can only refer to one and only one pool. However, `_parent` used in a pool path can potentially refer to the parent of any pool on the pool stack during the lookup process.

Finally, there are anonymous (unnamed) pools that exist only as long as they are on the stack, and disappear when popped. An anonymous pool is created by passing `PushPool` no, or an empty, first parameter. Anything defined in an anonymous pool exists only as long as the pool is on the stack[7].

[6] For safety, the root pool should not be poppable, but in this version of m5, it is.

[7] Actually, that's a white lie. Anonymous pools are given unique names, there are ways to get at them (see `AllPoolNames` in appendix B), and for debugging purposes they are never deallocated. It should be an option as to whether the user wants anonymous pools to stay around or whether they should be deleted when popped since it is possible to write macros that can flood heap space by extensive use of anonymous pools. Unfortunately, that is not currently implemented.

The pool stack also defines the pool name lookup order; in a hierarchical file system this would correspond to the file search path also defining a directory search path. The following example demonstrates the effect of the pool stack on pool name lookup:

```
                          @@ initial stack=(_M5, _)
>Define(‘hello’,‘root’)   @@
>PushPool(‘ONE’)          @@ stack=(_M5~ONE, _M5, _)
>Define(‘hello’,‘one’)    @@
>PushPool(‘ONE1’)         @@ stack=(_M5~ONE~ONE1, _M5~ONE, _M5, _)
>Define(‘hello’,‘one1’)   @@
>PopPool(‘ONE1’)          @@
>PushPool(‘TWO’)          @@ stack=(_M5~ONE~TWO, _M5~ONE, _M5, _)
>Define(‘hello’,‘two’)    @@
>PushPool(‘TWO’)          @@ stack=(_M5~ONE~TWO, _M5~ONE~TWO, _M5~ONE, _M5, _)
>Define(‘hello~_TOS’,‘two2’)
>PopPool(‘TWO’)           @@ stack=(_M5~ONE~TWO, _M5~ONE, _M5, _)
>hello
two2
>hello~ONE1
one1
>hello~TWO
two2
>hello~_parent~TWO
two2
>hello~_M5
root
```

For the pedantically inclined, here are m5's name lookup algorithms. Given a macro reference of the form `macro`$\sim$`mp1`$\sim\ldots\sim$`mpm` and a pool stack = $(p_0, p_1, p_2, p_3, ..., p_n)$ with p_0 on top of the stack, and using *nullPool* as a symbolic name for the null pool '_', then the algorithm for resolving the macro reference is:

for i=0 **to** n **do**
  if p_i == *nullPool* **then break**
  if *macro*$\sim$*full_pathname_of*(p_i)$\sim mp_1\sim ...\sim mp_m$ *exists*
    then return *it*
  endfor
error("*macro does not exist*")

Given a pool path (in a `PushPool` invocation, say) of the form `mp1`$\sim$`mp2`$\sim$`mp3`$\sim\ldots\sim$`mpm` and a stack = $(p_0, p_1, p_2, p_3, ..., p_n)$ with p_0 on top of the stack, then

for i=0 **to** n **do**
  if p_i == *nullPool* **then break**
  if *full_pathname_of*(p_i)$\sim mp_1\sim ...\sim mp_m$ *exists*
    then return *it*
  endfor
error("*pool path does not exist*")

These lookup rules make creation of new macros and new sub-pools a bit more complicated than for the analogous name resolution rules in a hierarchical file system. For macros, given a definition command of the form `Define('macro~mp1~...~mpm','stuff')` and a stack defined as in the previous examples, the definition rule is:

for i=0 **to** n **do**
  if p_i == *nullPool* **then break**
  if *macro*~*full_pathname_of*(p_i)~mp_1~...~mp_m *exists* **then**
    redefine that macro
    return
    endif
  endfor
for i=0 **to** n **do**
  if p_i == *nullPool* **then break**
  if *full_pathname_of*(p_i)~mp_1~...~mp_m *exists* **then**
    define macro in full_pathname_of(p_i)~mp_1~...~mp_m
    return
    endif
  endfor
error("*pool path does not exist*")

For pools, given a pool creation command `DefinePool('mp1~mp2~mp3~...~mpm')` and a stack as defined above, the pool creation rule is:

for i=0 **to** n **do**
  if p_i == *nullPool* **then break**
  if *full_pathname_of*(p_i)~mp_1~...~mp_m *exists* **then**
    return *it* (*creation unnecessary*)
    endif
  endfor
for i=0 **to** n **do**
  if p_i == *nullPool* **then break**
  if *full_pathname_of*(p_i)~mp_1~...~mp_{m-1} *exists* **then**
    create subpool mp_m *in full_pathname_of*(p_i)~mp_1~...~mp_{m-1}
    return
    endif
  endfor
error("*pool path does not exist*")

7 Evaluation of Arithmetic Expressions

The m5 predefined macro `Incr` accepts multiple params, each of which is evaluated and added together; the resulting number is the value of the macro. If only one parameter is specified, then the second parameter defaults to '1'.

```
>Define(A,1) Incr(A) Incr(A,3,A)
 2 5
```

Note that after the first `Incr`, the macro `A` still has the value '1'.

If you need to do more complicated expression evaluation, the `Eval` macro accepts a single parameter that it parses as an arithmetic expression replete with operator precedence and type coercion. Computations are performed as 'long ints', unless the presence of a floating point number coerces them to 'double float'. The precedence corresponds to what the intuitions of FORTRAN or C programmers would (probably) tell them it should be. The implementations of the functions come from the C standard mathematics library, so their descriptions can be found in the documentation for that library.

`float(i)`: turn i into a floating point number;

`gcd(i,j)`: greatest common divisor of i and j;

`int(f)`: coerce floating number f to an integer;

`sign(n)`: -1, 0, 1 when n is negative, zero, or positive;

`sqrt(n)`: the (float) square root;

`log(f)`: base e log;

`log10(f)`: base 10 log

`sin(n)`, `cos(n)`, etc.: all of the trigonometric functions

`asin(n)`, `acos(n)`, etc.: all of the arc-trig functions

`hypot(x,y)`: computes the radius of a circle centered at the origin with the point (x, y) on the circumference;

`max(n1,n2,n3,...)` the maximum of the ns.

`min(n1,n2,n3,...)` the minimum of the ns.

The `Eval` macro expands to a textual representation of its result. In order to write macros with many `Eval` commands, and to make sure that exact values are communicated between invocations of `Eval`, m5 has been designed with a special 'exact' format for floating point numbers. What that format will look like will vary from machine to machine, and it isn't human-consumable or portable. You should make it a practice to surround `Eval` invocations with a `Fmtnum` computation in all places where the result of the `Eval` is to be read by humans or other machines, particularly if the result could be a floating point number rather than an integer.

For instance, on a machine with IEEE format for floating point numbers, `Eval(3/5)` expands to `0f3fe3333333333333`. `Fmtnum(Eval(3/5))` expands to `0.600`. Note that once `Eval` drops into double precision mode, it stays there, and it expands into the 'exact' format. Consider:

```
Define(`x',Eval(3/5))

Eval(x*5)
0f4008000000000000
Fmtnum(Eval(x*5))
```

```
3
x
0f3fe3333333333333
Fmtnum(x)
0.600
```

Imposing this extra level of number formatting was prompted by a plotting application. It was necessary to avoid the loss of precision resulting from continuous conversion from decimal representations to binary and back again. This seemed the easiest way to do it at the time, although I'm sure there are better ways (e.g., formatting parameters to `Eval`).

8 String Manipulation

M5 defines the usual panoply of substring and indexing functions. A list of the available functions is in appendix section A.6 and their definitions are listed alphabetically in appendix B.

The workhorse macros are the substring macros `Substr` and `Substrx`. `Substr(s,x,l)` extracts a substring starting after the `x`th character extending for `l` characters. The indexing version, `Substrx(s,i,j)`, extracts the substring starting after the `i`th character and ending just after the `j`th character. Which macro to use depends on whether you're keeping track of the length of substrings or the precise location of substrings. Both macros come in a quoting version (`Substrq` and `Substrxq`) to guard against accidentally expanding to another macro name.

The functionality of these macros has been extended to accept negative arguments. When a position argument is negative, positions are counted from the end of the string. When a length argument is negative, it refers to characters to the left of a position. The following examples should clarify the details: (indices start at 0 in all string functions)

```
>Define(‘t’,‘0123456789abcdef’) >Define(‘t’,‘0123456789abcdef’)
>Substr(t,5,4)                 >Substrx(t,5,4)
5678                           4
>Substr(t,-5,4)                >Substrx(t,-5,4)
bcde                           456789a
>Substr(t,5,-4)                >Substrx(t,5,-4)
1234                           56789ab
>Substr(t,-5,-4)               >Substrx(t,-5,-4)
789a                           b
>Substr(t,5)                   >Substrx(t,5)
56789abcdef                    56789abcdef
>Substr(t,-5)                  >Substrx(t,-5)
bcdef                          0123456789a
>Substr(t,-5,99)               >Substrx(t,-5,99)
bcdef                          bcdef
>Substr(t,-5,-99)              >Substrx(t,-5,-99)
0123456789a                    0123456789a
>Substr(t,5,99)                >Substrx(t,5,99)
56789abcdef                    56789abcdef
>Substr(t,5,-99)               >Substrx(t,5,-99)
01234                          01234
>Substr(t,-1,1)                >Substrx(t,-1,1)
f                              123456789abcde
>Substr(t,-2,1)                >Substrx(t,-2,1)
e                              123456789abcd
>Substr(t,-99,3)               >Substrx(t,-99,3)
012                            012
>Substr(t,3,-2)                >Substrx(t,3,-2)
12                             3456789abcd
>Substr(t,-3,-1)               >Substrx(t,-3,-1)
c                              de
```

9 File I/O

9.1 Input (Include)

Whenever any input file is opened for the first time, the macro `Opening(‘input file name’)` is invoked.

If an `Include` macro diverts scanning to another input file, the macro `Suspending(‘input file name’)` is invoked.

When the `Include` finishes (including its own invocations of these macros), then the user is notified that this file is once again the current input file by the invocation of the macro `Reopening(‘input file name’)`.

At the end of the current input file, `Closing(‘input file name’)` is invoked.

Anytime something happens with standard input, the name passed to the macros will be ‘`<stdin>`’.

When m5 is about to exit a successful run (i.e. no errors) the macro `Exiting` is invoked.

The user can re-define these macros, which are initially defined to be the empty string, to do actions on each file as it is opened, suspended, included, re-opened, and closed.

9.2 Output Diversion

`Divert`, `Divpush`, and `Divpop` accept a single parameter, either a filename or a filenumber (obtainable as the expansion of the `Filename` or `Divnum` macros).

Currently, there is a limit of 20 open output files. They are opened when first diverted to; any diversions that were created using only a filenumber (e.g., `Divert(3)`) and were never given names (which can be done after the fact with the `Filename` macro) are simply appended to standard out in numeric order upon exit. This is another vestigial m4-ism. All the user needs to remember are `Divpush(fn)` which pushes the current output file onto a stack and changes the current output file to `fd`, and `Divpop`, which recovers the previous output file. If you want to append to an already existing file, use the 'A' option of the `Filename` macro (see the definition of `Filename` in appendix B).

There is also a `Print` function which forces its arguments immediately to the current output file, and then returns the empty string. For convenience, there is an `Fprint` version:

```
Define(‘putout(str)’,‘Fprint(‘file’,‘$(str)’)’)
```

which writes the value of `$(str)` directly on to `file`.

10 Debugging

M5 has extensive debugging facilities. `Debug` with no arguments or one null argument is equivalent to `Debug(‘Std’)`. The debug flags listed below are additive, except for '`Off`' and '`Std`'.

`Std`: Equivalent to `Debug(‘Off’)Debug(‘Def’,‘Params,‘Result’, ‘Stderr’,‘Path’)`.

`Off`: Disables all debugging output.

`Dump`: Does a `DumpPools` on exit.

`Def`: Print `\\Define(‘macro’,‘newdef’)//` every time a macro is defined or changed.

`Name`: Print the names of `<<macros>>` when they are recognized. After recognition comes parameter collection, so there may be a lot of debug output before you see the expansion of `<<macro>>`.

`Path`: Print the complete names of `<<macros~with~their~path>>` starting from from the root pool.

`Pools`: Print `::>Pool name::` when a pool is pushed, and `::<Pool name::` when a pool is popped.

`Parms`: Print the `[[macro(‘with’,‘actual parameters’)]]` before expanding. Once all parameters have been collected, this debug mode allows you see exactly what were the values of those parameters before parameter substitution takes place in the body of the macro.

`Result`: Print the `{{expansion of macros}}` just before they are re-scanned.

`Stderr`: Put debug output on stderr (else goes to current output file).

`Intern`: Dump `::internal debug::` information (useful only to implementor of m5).

Any name preceded with `POOLCH` (default: '~') turns that debug option off (except for `Std` and `Off` which ignore it):

```
     INPUT                                  Comments
=============================================================================
>Debug(‘Std’,‘Name’,‘~Stderr’,‘~Pools’)
{{}}                                        -- the result of the Debug cmd
>Define(junk(),This is $1 bunk)
<<Define>>                                  -- first see the macro name Define
[[Define(‘junk()’,‘This is $1 bunk’)]]      -- show the result of expanding all its
                                            -- parameters just before expanding them
\\Define(‘junk()’,‘This is $1 bunk’)//      -- macro defined
>junk("junk(pure)")                         -- invocation of macro just defined
<<junk>>                                    -- sees first junk
<<junk>>                                    -- while expanding params for first
[[junk(‘pure’)]]                            -- how the inner junk looks after
                                            -- expansion of its parms before it is
                                            -- expanded
{{This is pure bunk}}                       -- result of expanding inner junk,
                                            -- i.e. what is pushed back into the
                                            -- input
[[junk(‘"This is pure bunk"’)]]             -- the outer junk with parms expanded
{{This is "This is pure bunk" bunk}}        -- what is pushed back into input
This is "This is pure bunk" bunk            -- the final result.
>Debug(‘Off’)
<<Debug>>
[[Debug(‘Off’)]]
-----------------------------------------------------------------------------
```

There are several ways of dumping the state of the processor.

`DumpExp`: Dump the current state of the macro stack onto the standard error file. All macros currently having parameters collected for them are displayed with their partial parameter lists.

`DumpMacs`: Dump all macros currently defined. They are dumped in the order of the pools stack, bottom-most pool first. This means that if the results of the dump were re-read by m5, the current set of available macros would be recreated. Note that the macro pool hierarchy is not reconstituted: it is flattened into pool _M5. They are written in bottom-to-top order so that multiple definitions would be overridden correctly.

`DumpMacs(M1,M2,...)`: dumps only the specified macros.

`DumpStack`: Same as DumpMacs, except the pools on the stack can be reconstructed from the output of this command, although the new hierarchy might not match the original.

`DumpPools`: Dumps all pools in such a manner that if the output were reread by m5, the macro pool hierarchy and their contents would be recreated exactly.

`DumpPools(P1,P2,...)`: dumps only the specified pools so they can be recreated exactly.

`DumpPool(P1,P2,...)`: exactly the same functionality as DumpPools, except the indicated pools will not be recreated: rather, when re-read by m5, the output of this macro will define all macros in the current pool.

Another form of the above macros (`DebugExp`, `DebugMacs`, `DebugStack`, `DebugPools`, `DebugPool`) send its output to the current debug output file, as defined by the `Debug` command (see the `Stderr` option to `Debug`).

11 Character Configuration

M5 was designed to be useful as a preprocessor of text files and to work in conjunction with other macro processors (e.g. *cpp*) and preprocessors (e.g. eqn, tbl, awk, TEX, LATEX) without getting in their way or stepping on their toes. Therefore, there needed to be a way of scanning a text file and turning m5 processing on and off at will without having to do major surgery on the file. In other words, m5 has the ability to define bracket macros that can turn macro processing on and off as needed. Central to this facility is the ability to change how m5 interprets various characters. To this end, m5 defines a set of configuration control macros.

Certain characters have specific meanings to m5; e.g., '(' and ')' surround parameter lists and parameter specifications; ` ' flags a pool path for macro name lookup, etc. The `Configure` changes the characters that have these specific meanings. This 'feature' has been used extensively, but its `ad hoc` nature precludes any guarantees that it is a consistent implementation. It certainly does not approach the robustness and completeness of a LISP reader table, but it does seem sufficient to simplify certain contextual dependencies.

M5's special characters are listed below in the order of their position in the `Configure` macro's parameter list, along with their names, defaults, and numeric value in hex and decimal.

```
LQ  RQ  DLR  CMT1  CMT2  LPAR  RPAR  COMMA  DASH  COUNT  POOLCH  PARMSEP  HQ
`   '   $    #     @     (     )     ,      -     *      ~       ,        \
96  39  36   35    64    40    41    44     45    42     126     44       92
60  27  24   23    40    28    29    2c     2d    2a     7f      2c       5c
```

`LQ`: The left quote character. It is recognized during text scan mode and disables the recognition and expansion of macros after the left quote and before the matching right quote. Also used in parameter specifications. Matches `RQ`.

`RQ`: The right quote character. Matches `LQ`.

`DLR`: The character signalling the beginning of a parameter specification. Recognized only during macro expansion mode.

`CMT1`: Corresponds to m4's original comment character.

`CMT2`: The new m5 comment character.

`LPAR`: Defines a left parenthesis. Used in macro argument lists, and in parameter specifications. Matches `RPAR`.

`RPAR`: The right parenthesis character. Matches `LPAR`.

`COMMA`: The character separating macro arguments.

`DASH`: The character used in parameter specifications to indicate concatenation of parameter values.

`COUNT`: The character used in parameter specifications to request the number of parameters passed to the macro.

`POOLCH`: The character used to qualify a macro name to a specific macro pool. (See Section 6.)

`PARMSEP`: This is a shorthand for specifying the character to use when parameter specifications are expanded into lists of parameter values: `$(1,)` emits a `PARMSEP` separated list of parameters.

`HQ`: The 'hard' quote character. When a character is hard-quoted, it is never interpreted in any way. The hard-quote stays with the quoted character through all processing, and is designed to remain invisible throughout (although it has no meaning in the middle of parameter specifications). The hard quote is stripped from the text only when m5 emits the text to output after all scanning is completed.

Some of the special characters are special only in certain contexts, and are not recognized as special outside of those contexts. Here is a more detailed description of m5's scanning contexts in no particular order:

1. comments: skipping all characters until end of comment.
2. macro-seeking: looking for the names of macros to expand.
3. macro-expansion: inserting the body of a macro into the input stream, replacing parameter specifications with their values.
4. parm-spec-scan: parsing something beginning with `DLR`.
5. parsing numbers: 0x45, 0faaaa, 0.5, 23.
6. parsing expressions: The text comprising the single parameter to `Eval` is always parsed exactly the same way; the special characters are not special inside a string passed to `Eval`, and changing their definition has no effect on what `Eval` expects as input.
7. scanning quotes: Have seen `LQ`, looking for matching `RQ`.

The following table shows the contexts in which these characters are recognized as special:

```
  modes:   (1) (2) (3) (4) (5) (6) (7)
LQ              *   *   *           *
RQ              *   *   *           *
DLR                 *
CMT1            *   *
CMT2            *   *               *
LPAR                *   *
RPAR                *   *
COMMA               *   *
DASH                    *
COUNT                   *
POOLCH              *
PARMSEP             *
HQ              *   *   *           *
```

If m5 is used to prepare text as input to other text processors that also interpret one or more of these characters, it may be desirable to change the definition of those special characters. An example would be creating makefiles that use *make*'s variable character '`@`': this conflicts with m5's default `CMT2` character '`@`'. In order to write macros that create makefile lines containing this character, the `CMT2` character can be changed within m5.

To do a massive change, use the `Configure` macro. Here is an example that totally changes the set of the characters recognized by m5:

```
>Configure({,},^,|,&,[,],;,!,",%,+,=)
>Define[{x};{y}]
>Define[x;{z}]
>line x y z
line z z z
>Configure[';';$;#;@;(;);,;-;*;~;,;\]
```

The new character definitions take effect *after* the expansion of the `Configure` macro. If you want to change only the comma character, then say `Configure(,,,,,,,;)` and the other characters are left intact. Parameter positions that are empty or not specified are ignored. The last invocation of `Configure` above restores the defaults; a simpler way to restore all defaults would have been to invoke `Configure` with no parameters.

Obviously, you have to be careful which characters you're using and what you may be changing them to. For instance, if the `LQ` and `LPAR` characters were both the same character, results are undefined. Also note that such massive changes to the definitions of the special characters may change the expansion of macros that were defined under another configuration.

These special characters can also be redefined to a different value by invoking the corresponding name macro with a parameter. All of these characters except `CMT2` are accessible through predefined macros. (Once `CMT2` is expanded, then its expansion immediately begins a comment; probably not the desired result.) The names of the macros are the names we have been calling the characters in this section (`LQ`, `RQ`, etc.). For example, `CMT2(CMT1)` makes the two comment characters the same.[8] `LPAR('{')RPAR{'}')` changes the parentheses characters: note *very* carefully

[8] If CMT1 and CMT2 are the same character then that character acts like a CMT1 (default '#').

the characters in those commands—that is not a typo. After `LPAR` has changed the left parenthesis, then that new character must be used with the `RPAR` command when it changes the right parenthesis.

It is useful to change the configuration and then later restore the configuration to whatever its previous state may have been. The macros `PushConfig` and `PopConfig` do just that. `PushConfig` takes an argument list just like `Configure` and defines the special characters as indicated *after* pushing the current configuration settings. `PopConfig` takes no arguments and restores the configuration to what it was prior to the last `PushConfig`. So our previous example for changing all the special characters can more conveniently be done and undone via:

```
PushConfig({,},^,|,&,[,],;,!,",%,+,=)
:
&& your stuff (this is a '@' comment)
:
PopConfig
```

To disable a special character, pass the empty string to the corresponding name macro; that is, `CMT1` expands to the current m4-style comment character, `CMT1('*')` changes the m4-style comment character to an asterisk, and `CMT1()` effectively disables m4-style comments. What actually happens is that the character is set to some byte value that is highly unlikely to appear in text files. In Section C below is a discussion of how this facility can be used to advantage.

There are predefined macros to set up the m5 scanning configurations as necessary for the clean scanning of the input text. Look in section B for the definitions of `PushScanConfig`, `PushConfigC`, `PushConfigCpp`, and `PushConfigAda`. The basic idea is to disable all m5 comments, quoting, poolnames, and, at the same time, turn on recognition of the indicated languages' comments. In these modes, only plain, unadorned macro names (i.e. no pools and no parameters) will be recognized.

Needless to say, if you get serious about playing around with the settings of m5's special characters, be prepared for some serious macro debugging.

12 Bugs, shortcomings, and other non-obvious 'features'

Like most software tools that have evolved under the pressure of getting a job done (as opposed to being designed in, of, and for themselves), m5 carries a lot of baggage from its predecessors. There is much that would be changed if there was an opportunity to 'do it right', but almost all such changes are at the level of syntax, and not semantics. For one thing, early attempts to maintain some compatibility with the old m4 macro processor later became an albatross around the neck of m5.

The configuration character feature is the ugliest one-more-feature wart on m5's body technic. One non-orthoganility is the inability in m5 to change the character definitions for identifiers (the colon ':' as an alphabetic can be annoying when scanning C++ code). It desperately needs generalization and cleanup. A completely general implementation would look something like a LISP read table, or TEX's character classes.

When the `Environment` macros are defined from the environment variables, it is possible to have macros defined that can cause problems, e.g., with uninterpreted special characters. For instance, an environment variable `TROUBLE` with the value `'single quoted string'` will cause all kinds of trouble if it is ever expanded in m5. Normally, you will not be bothered by this unless you try to

`DumpPool(Environment)` (q.v.) and then try read it back in. If you want to dump all pools so the dump file can be used to reconstitute a macro environment, `FlushPool(Environment)DumpPools()` is a useful m5 idiom.

The `-Define(M[,V])`— invocation line definition feature is pretty brain-dead. Its parse is very strict: take everything up to the first comma (this is not affected by any resetting of the special characters `RPAR`, `PARMSEP`, and `LPAR`) and make that the macro name, take everything from just after the comma to just before the last character, which better be a right parenthesis, and make that the value. If there is no comma, then everything from just after the left parenthesis to just before the right parenthesis is made a macroname and defined to expand to the empty string. It would be nice if this feature were completely general, allowing you to define a macro on the invocation line, but m5 and shells do not agree on the meaning of various quote marks, making it extremely difficult to put m5 text on an invocation line. The bottom line is that, if `-Define(foo,bar)` appears on the invocation line, it is equivalent to prepending `Define(‘foo’,‘bar’)` to the next input file. Furthermore, the simplistic parsing algorithm for this invocation option has the side effect that the macro can have at most one formal parameter. A more general implementation of this feature might be to push `Define(‘foo’,‘bar’)` into the input stream and let it be processed normally, but this is too difficult to worry about. Anything more than defining very simple macros requires an input file.

In section 4.4 we spent a lot of time exploring the ramifications of manipulating parameter strings in macros. While parameter specification expansion depends on the values of the parameters, you cannot specify in a parameter specification whether or not the parentheses surrounding a parameter list will appear. In other words, when `cntParms` is invoked in the previous macros, it will always have at least one parameter: it may be empty, but it will be there because the parentheses are there. The problem can be solved by using the `Incr` macro with a conditional test. That is, we make sure `cntParms` always has at least one parameter, and then subtract one from its result. I'm sure that this problem could have been solved with some (more) flourishes to the specification syntax (e.g.: `$[1,]` (note the square brackets) expanding to `($1,$2,...)` iff at least one of the parameters was not empty) but it didn't seem worth the effort. More specifically, it was easier to do the job with macros rather than modify the already overloaded parameter specification mechanism by adding one more special character.

One final annoyance: m5 adopted m4's use of the balanced single quotes ‘ and ’. This is the one feature of m5 that interferes significantly with processing C and C++ files, or any file where single quotes are not balanced. There are several ways to handle this: use pools and configurations to control where macros are recognized, use cpp to `#define cA 'A'` single character literal macros, or remember to hard-quote all C uses of the single quote character. None of these are very pleasant alternatives.

A M5 Predefined Macros by Category

A.1 Macro Definition and Maintenance

```
AllMacroNames(POOL)
Append(NAME,STR1,...,STRn)
Assoc(VAL,POOL)
Assocq(VAL,POOL)
DebugExp
DebugMacs(M1,M2,...)
Define(NAME(P1,P2,...),STR1,STR2,...)
Defn(MACRONAME)
DumpMacs(M1,M2,...)
FlushPool(POOL1, POOL2, ..., POOLn)
Ifdef(A,B,C)
IfdefTops(A,B,C)
Ifempty(MAC,THEN,ELSE)
Ifndef(A,B,C)
IfndefTops(A,B,C)
Ifnempty(MAC,THEN,ELSE)
Newmacname(STR)
Qn(NUM,P1,P2,...,Pn)
QSn(NUM,P1,P2,...,Pn)
Redefine(NAME(P1,P2,...,Pn),STR1,...,STRn)
UndefBlk(MACRONAME)
Undefine(MACRONAME)
Unundefine(MACRONAME)
```

A.2 Pool Definition and Maintenance

```
AllMacroNames(POOL)
AllPoolNames(POOL)
Assoc(VAL,POOL)
Assocq(VAL,POOL)
DebugExp
DebugPool(POOL1,POOL2,...)
DebugPools(POOL1,POOL2,...)
DebugStack
DefinePool(P0,P1,P2,P3,...,Pn)
DumpPool(POOL1,POOL2,...)
DumpPools(POOL1,POOL2,...)
DumpStack
Environment
FlushPool(POOL1, POOL2, ..., POOLn)
IfdefPool(POOL,A,B)
IfdefTops(A,B,C)
```

```
IfndefPool(POOL,A,B)
IfndefTops(A,B,C)
Pool(NUM)
PopPool(NAME)
PushPool(POOL,P0,P1,P2,...,Pn)
\_
\_M5
\_parent
\_TOS
```

A.3 Pool Stack Manipulation

```
DebugExp
DebugMacs(M1,M2,...)
DebugStack
DumpMacs(M1,M2,...)
DumpStack
Pool(NUM)
PopPool(NAME)
PushPool(POOL,P0,P1,P2,...,Pn)
```

A.4 Conditionals, Expansion Control

```
Case(BASE,NUM,C0,C1,C2,...,Cd)
Exit
Ifdef(A,B,C)
IfdefPool(POOL,A,B)
IfdefTops(A,B,C)
Ifempty(MAC,THEN,ELSE)
Ifeq(COND,THEN,ELSE,...)
Ifndef(A,B,C)
IfndefPool(POOL,A,B)
IfndefTops(A,B,C)
Ifnempty(MAC,THEN,ELSE)
Ifneq(COND,THEN,ELSE,...)
Match(ITEM,V0,C0,V1,C1,V2,C2,...,Vn,Cn,Cd)
RandomSelect(C1,C2,...,Cn)
```

A.5 Arithmetic Expression Evaluation

```
Eval(EXPR)
Fmtnum(NUM,FIELD,FRAC,FMT)
Incr(N1,N2,N3,...,Nn)
RandomInt
```

A.6 String Formatting

```
Abort(FMT,STR1,STR2,...,STR5)
AsciiToNum(ARG)
Errprint(FMT,STR1,STR2,STR3,STR4,STR5)
Exit
Fmtnum(NUM,FIELD,FRAC,FMT)
Index(STR1,STR2)
Indexr(STR1,STR2)
Len(STR)
Maketemp(STR)
Newmacname(STR)
Q0(...)
Q1(...)
Q2(...)
Q3(...)
Q4(...)
Qn(NUM,P1,P2,...,Pn)
QS0()
QS1()
QS2()
QS3()
QS4()
QSn(NUM,P1,P2,...,Pn)
Reverse(STR)
Reverseq(STR)
Strchr(S1,S2)
Strcspn(S1, S2)
Strpbrk(S1, S2)
Strrchr(S1,S2)
Strspn(S1, S2)
Substr(STR,POS,LEN)
Substrq(STR,POS,LEN)
Substrx(STR,I,J)
Substrxq(STR,I,J)
Translit(STR, FROM, TO)
```

A.7 Files: Naming, Diverting, I/O

```
Abort(FMT,STR1,STR2,...,STR5)
Closing(FILE)
CurInfileDepth
CurInfileName
Divert(NAMENUM,MODE)
Divnum(FILENAME)
Divpop(NAMENUM)
```

```
Divpush(NAMENUM,MODE)
Errprint(FMT,STR1,STR2,STR3,STR4,STR5)
Filename(FILENAME,NUMBER,MODE)
Fprint(F,S1,S2,...,Sn)
Include(FILENAME)
InputFile
InputLine
Maketemp(STR)
Opening(FILE)
Print(S1,S2,...,Sn)
Reopening(FILE)
Sinclude(FILENAME)
Suspending(FILE)
Undivert(I)
```

A.8 User Hooks for Special Actions

```
Closing(FILE)
Opening(FILE)
Reopening(FILE)
Suspending(FILE)
```

A.9 Debugging

```
Abort(FMT,STR1,STR2,...,STR5)
Debug(WHAT1,WHAT2,...,WHATn)
DebugExp
DebugMacs(M1,M2,...)
DebugPool(POOL1,POOL2,...)
DebugPools(POOL1,POOL2,...)
DebugStack
Errprint(FMT,STR1,STR2,STR3,STR4,STR5)
Exit
```

A.10 Scanning Configurations

```
CMT1()
CMT2()
COMMA()
Configure(ch1,ch2,ch3,...,ch13)
COUNT()
DASH()
DLR()
HQ()
LPAR()
LQ()
```

```
PARMSEP()
POOLCH()
PopConfig
PushConfig(ch1,ch2,ch3,...,ch13)
PushConfigAda(ch1,ch2,ch3,...,ch13)
PushConfigC(ch1,ch2,ch3,...,ch13)
PushConfigCpp(ch1,ch2,ch3,...,ch13)
PushScanConfig(ch1,ch2,ch3,...,ch13)
RPAR()
RQ()
```

A.11 Miscellaneous

```
Dnl
Syscmd(STR)
```

B M5 Predefined Macro Reference

Each macro is listed with its name and named parameter list. (The parenthesized file name is the source file where the corresponding implementation function is found; I hope the reader will excuse the reference manual serving double duty for the implementor as well.) If a macro does not accept parameters, it is not followed by parentheses in its description. Interspersed in the list are special names which m5 recognizes, such as the names of the predefined pools.

`Abort(FMT,STR1,STR2,...,STR5)`: (predefs.m)

Like Errprint, but aborts with exit status 1 after printing the $\texttt{STR}_i$ using the format string `FMT`.

`AllMacroNames(POOL)`: (pools.m)

Expands into a PARMSEP separated list of the names of the macros defined in the indicated pool. Since in my humble opinion this will be primarily for the easy collection of names rather than actually expanding a bunch of macros, the list is returned with each name in the list surrounded by left and right quotes. If the pool is not found, the result is the empty string. If no pool path is specified the current pool is used.

`AllPoolNames(POOL)`: (pools.m)

Expands into a PARMSEP separated list of the names of all pools defined in the indicated pool. Like the expansion of AllMacroNames, the list is returned with each name in the list surrounded by left and right quotes. If the pool is not found the expansion is the empty string. If no pool is specified the current pool is used.

`Append(NAME,STR1,...,STRn)`: (macros.m)

Append `STR1...STRn` to the definition of `NAME`. The first definition found of `NAME` is used. If `NAME` is not found anywhere on the pool stack, `Append` then acts like `Define`.

We have semantic subtleties here: Assume there exists `Define(‘foo(bar)’,‘junk’)` and now we encounter `Append(‘foo’,‘ more’)`. The correct thing to do is to keep the parameter list from the original foo. However, if we encounter `Append(‘foo(baz)’,‘ more’)`, then the parameter list is changed to ‘(baz)’ a warning message is issued that the parameter lists do not match between the original `Define` and the later `Append`. If the previous definition had no formal parameters, and the `Append` specifies parameters, then the new formals take effect.

`AsciiToNum(ARG)`: (ascii.m)

Turn `ARG` into a PARMSEP-separated list of decimal numbers, one ascii code per character; special C-like escape codes apply (`\n`, `\t`, etc.). There are some special escape codes for m5: `\a` returns the m5 comment character (normally the ‘at’ sign); `\h` returns the m4-compatible comment character (default is the ‘hash’ mark `#`); `\p` is the current left-quote character (`\‘`); `\’` is the current right-quote character; an escaped dash introduces a range of characters (`0\-9`); `\0` is the null character, and `\000` is an octal ASCII code; `\x00` is a hex ASCII code, and `\d000` is a decimal code. Any other character after the HQ is just that character.

`Assoc(VAL,POOL)`: (pools.m)

Expands into a PARMSEP separated list of the names of the macros defined in `POOL` whose definition is equal to `VAL`. Since in my humble opinion this will be primarily for the easy collection of names rather than actually expanding a bunch of macros, the list is returned with each name in the list surrounded by left and right quotes. If the pool is not found, the result is the empty string. If no pool path is specified the current pool is used.

`Assocq(VAL,POOL)`: (pools.m)

Same as Assoc except the PARMSEP list of quoted names is also quoted.

`CMT1()`: (predefs.m)

Macro to set a special character, or to return it. See section˜ refCONFIG for a discussion of Special Characters.

`CMT2()`: (predefs.m)

Macro to set a special character, or to return it. See section˜ refCONFIG for a discussion of Special Characters.

`COMMA()`: (predefs.m)

Macro to set a special character, or to return it. See section˜ refCONFIG for a discussion of Special Characters.

`COUNT()`: (predefs.m)

Macro to set a special character, or to return it. See section˜ refCONFIG for a discussion of Special Characters.

`Case(BASE,NUM,C0,C1,C2,...,Cd)`: (predefs.m)

If `NUM` equals `BASE` then `C0` is selected; if `NUM` equals `BASE`+i, then `C`$_i$ is selected; otherwise `Cd`, the default, is selected.

`Closing(FILE)`: (hooks.m)

Invoked whenever the current input file is closed. The single parameter is the name of the file being closed. `FILE` is closed before the macro is invoked.

`Configure(ch1,ch2,ch3,...,ch13)`: (predefs.m)

Allows the user to change any and all of the special characters m5 uses for scanning the input. See section 11 for details.

`CurInfileDepth`: (files.m)

Returns the depth of the current input file. If zero, then input is coming from a file named on the invocation line or stdin. If greater than zero, then input is coming from an `Included` file. If less than zero, there is no input: you are in the midst of pre- or post-processing. When the `Opening` or `Suspending` macros are invoked, CurInfileDepth has already been incremented. When `Closing` or `Reopening` are invoked, the depth count has already been decremented.

`CurInfileName`: (files.m)

Expands to the name of the current source file, '`<stdin>`', or, if in pre- or post-processing mode, the null string .

`DASH()`: (predefs.m)

Macro to set a special character, or to return it. See section˜ refCONFIG for a discussion of Special Characters.

`DLR()`: (predefs.m)

Macro to set a special character, or to return it. See section˜ refCONFIG for a discussion of Special Characters.

`Debug(WHAT1,WHAT2,...,WHATn)`: (predefs.m)

Toggle debugging flags; see section 10 for detailed description of the possible values of the $\texttt{WHAT}_i$.

`DebugExp`: (dumping.m)

Whenever encountered, prints a dump of the current state of the macro expansion stack.

`DebugMacs(M1,M2,...)`: (dumping.m)

Same as DumpMacs, except uses debug output file instead of the current output file.

`DebugPool(POOL1,POOL2,...)`: (dumping.m)

Same as DumpPool, except the output is sent to the current debug output file as defined by the Debug macro

`DebugPools(POOL1,POOL2,...)`: (dumping.m)

Same as DumpPools, except the output is sent to the current debug output file as defined by the Debug macro.

`DebugStack`: (dumping.m)

Same as DumpStack, except the output goes to the debug output file instead of the current output file.

`Define(NAME(P1,P2,...),STR1,STR2,...)`: (macros.m)

`NAME` is defined to be the concatenation of the values of the strings. `NAME`'s parameters can be referenced via $(P1), $(P2), ...in addition to $1, $2, ...(see section 4.4 in the User's Guide). `NAME` is put in the first pool that matches the pool path, if there is one; otherwise, it is put in the current pool (the one on the top of the pool stack). If `NAME` is going to expect parameters, it must have a formal parameter list defined, even if it is empty. For example, Define('MAC()','yo $1') works. Define('NAC','no $1') is an error since $1 would be an empty string no matter whether the macro was invoked with just NAC or NAC(foo). The error will not be caught, however, until an attempt is made to expand the macro.

`DefinePool(P0,P1,P2,P3,...,Pn)`: (pools.m)

If the indicated pools do not exist, they are created and initialized, but not pushed on the stack. This command is equivalent to 'PushPool('P_i')PopPool' for each P_i.

`Defn(MACRONAME)`: (macros.m)

This expands `MACRONAME`, and puts the results in quotes. In this manner, you can get the definition of a macro without having it expanded further.

`Divert(NAMENUM,MODE)`: (files.m)

Send output to the indicated file; see section 9 for details. `Divert(NUM)`: Divert the the NUM'th output file. `Divert(NAME)`: Look for a previously declared file by that name and divert to it. `Divert(NUM,'a')`: Divert to file NUM; if not opened already, then will be opened for append. `Divert(NAME,'a')`: Ditto for file named NAME. If NAME must begin with a digit, then use the HQ character to hide it. If no parameters, the current output file reverts to file 0, which is stdout.

`Divnum(FILENAME)`: (files.m)

Expands to the file number of `FILENAME`; if `FILENAME` not specified, expands to file number of the current divert file. Expands to -1 if `FILENAME` is not an open file.

`Divpop(NAMENUM)`: (files.m)

Part of the output diversion enhancements; see section 9. Effectively closes the current output file, and diverts output to the previous output file.

`Divpush(NAMENUM,MODE)`: (files.m)

Part of the output diversion enhancements; see section 9. Exactly like Divert, except the current output file is pushed onto a stack.

`Dnl`: (predefs.m)

Eats characters from the input up to and including the first new line character.

`DumpMacs(M1,M2,...)`: (dumping.m)

Prints the definitions of the named macros. If no macros are named, then all macros currently on the stack are printed.

`DumpPool(POOL1,POOL2,...)`: (dumping.m)

Put all macros in all named pools onto the current output file such that if that output file were reread by m5, the pool _M5 would have all the macros defined in the named pools redefined in it (barring multiple definitions). The pools would not be recreated, simply glumphed together; see `DumpPools` for an alternative. *Only* the named pools are dumped; subpools are not dumped. If no pools are named, then all macros in all pools are dumped as if in the same pool. Rereading these dumped macro definitions could, therefore, result in lost definitions where the macros have identical names.

`DumpPools(POOL1,POOL2,...)`: (dumping.m)

The macro definitions from the named pools are surrounded with `PushPool`(pool path name) ...`PopPool`(pool path name); if the output of the '`DumpPools`' macro is re-read by m5, pool hierarchy is recreated exactly. No arguments causes all pools to be so dumped. Each named pool is dumped recursively so that the pool hierarchy can be recreated. If no pool is named, dumping begins at the root pool M5.

`DumpStack`: (dumping.m)

Prints the contents of all pools on the pool stack.

`Environment`: (hooks.m)

A predefined pool that contains as macros all variables defined in the environment. The value of the variable is the body of the macro. Be aware that it is possible to define environment variables whose value can confuse m5 during macro reading or expansion.

`Errprint(FMT,STR1,STR2,STR3,STR4,STR5)`: (predefs.m)

The first argument `FMT` is a printf-like specification (except that only the string specification %s can be used in it) with up to five other parameters. Output is to stderr. No other action is taken. (see Abort)

`Eval(EXPR)`: (predefs.m)

`EXPR` is a string that consists of numbers, operators, functions, and parentheses: NOTHING ELSE! All macros must have been expanded at the time Eval is called. All floating point computations are in double precision. Operators are `+ - * / & | && || ! % DIV **`, and correspond to the traditional C and FORTRAN semantics in ways that is left for the user to discover. See the discussion in section 7 for more details

`Exit`: (predefs.m)

A clean way to halt m5 from within the macros.

`Filename(FILENAME,NUMBER,MODE)`: (files.m)

Open the file `FILENAME` (if not already opened) and assign it a file number. If `NUMBER` is not present or empty, then an unused file number is chosen. If the first character of the optional parameter `MODE` is 'a' or 'A', then the file is opened for append, else for write. This macro expands to the file number so it can be used as the argument to `Divert`, `Divpush`, and `Divpop`. Use the 'Divnum' macro to retrieve the filenumber of `FILENAME`, if that is important. (It shouldn't be; all file manipulation macros accept the filename as a parameter. Filenumbers are vestigial m4-isms.)

`FlushPool(POOL1, POOL2, ..., POOLn)`: (pools.m)

Remove all macros defined in the indicated pools. Of course, `FlushPool('_M5')` is discouraged. If $\texttt{POOL}_i$ does not exist, it is created a lá `DefinePool`. The pools are flushed in left to right order. If no pools are specified, the current top-of-stack pool is flushed. Sub-pools in the flushed pools are not removed or affected in any way.

`Fmtnum(NUM,FIELD,FRAC,FMT)`: (predefs.m)

Expects `NUM` to be an integer or floating point number. The number is turned into a string that is `FIELD` characters long, `FRAC` characters of which are for the fractional part of the number (if any). If `FRAC` is zero, no decimal point is printed. The first character of `FMT` is used to select the format. The order of `FIELD`, `FRAC`, and `FMT` is not significant. However, if there are two number fields, they are interpreted sequentially as `FIELD` and `FRAC`, no matter what the position of the `FMT` parameter. `FMT` can be one of 'f', 'e', 'p', 'x', and defaults to 'p'.

		default		examples	
letter	name	`FIELD`	`FRAC`	3	3.1415926
n	normal	6	3	`3.000`	`3.142`
s	scientific	10	3	`3.000e+00`	`3.142e+00`
p	precise	0	3	`3`	`3.142`
x	exact	n/a	n/a	machine dependent	

`Fprint(F,S1,S2,...,Sn)`: (files.m)

Immediately prints the strings into the file `F`. `F` can be a string or a filenumber. This is equivalent to '`Divpush(F)Print(S1,S2,...)Divpop`'.

`HQ()`: (predefs.m)

Macro to set a special character, or to return it. See section˜
refCONFIG for a discussion of Special Characters.

`Ifdef(A,B,C)`: (predefs.m)

If a macro named `A` is defined, expand `B`, else expand `C`.

`IfdefPool(POOL,A,B)`: (pools.m) If `POOL` exists, then expand `A`, else expand `B`.

`IfdefTops(A,B,C)`: (predefs.m)

Same as `Ifdef`, except only the topmost pool on the pool stack is queried. That is, `Ifdef` checks all pools on the stack, but `IfdefTops` checks only the top-most pool.

`Ifempty(MAC,THEN,ELSE)`: (predefs.m)

If there is a macro defined named `MAC` and if it is defined to be the empty string then return `THEN`, else return `ELSE`. This speeds macro expansion up a bit by not requiring the value of the macro to be expanded and stuffed in the input string just to check whether it is the empty string or not. Note: `Ifempty` does *not* check whether the *expansion* of `MAC` is the empty string, only whether the defined macro is empty or not. This is functionally equivalent to

```
Ifdef(‘{\tt MAC}’,‘Ifeq(Defn(‘{\tt MAC}’,,‘{\tt THEN}’,‘{\tt ELSE}’)’)
```

but does not require the expansion of the macro.

`Ifeq(COND,THEN,ELSE,...)`: (predefs.m)

```
Ifeq(A): Return A.
Ifeq(A, B): Return A.
Ifeq(A, B, C): If A == B return C, else return empty string.
Ifeq(A, B, C, D, ...): If A == B return C, else evaluate Ifeq($(D,)).
```

`Ifndef(A,B,C)`: (predefs.m)

If a macro named `A` is *not* defined, expand `B`, else expand `C`.

`IfndefPool(POOL,A,B)`: (pools.m)

If `POOL` does not exist, then expand `A`, else expand `B`.

`IfndefTops(A,B,C)`: (predefs.m)

Same as Ifndef, except only the topmost pool on the pool stack is queried. That is, Ifndef checks all pools on the stack, but IfndefTOPS checks only the current, top-most pool.

`Ifnempty(MAC,THEN,ELSE)`: (predefs.m)

If there is a macro defined named `MAC` and if it is not defined to be the empty string then return `THEN`, else return `ELSE`.

`Ifneq(COND,THEN,ELSE,...)`: (predefs.m)

```
Ifneq(A): Return A.
Ifneq(A, B): Return A.
Ifneq(A, B, C): If A != B return C, else return empty string.
Ifneq(A, B, C, D, ...): If A != B return C, else evaluate Ifneq($(D,)).
```

`Include(FILENAME)`: (files.m)

Include the contents of file `FILENAME` at this point in the input; push the current input file on a stack, and return to it when EOF encountered in `FILENAME`.

`Incr(N1,N2,N3,...,Nn)`: (predefs.m)

Return the sum of the numbers, or if $n = 1$, `N1` $+ 1$.

`Index(STR1,STR2)`: (strings.m)

Return leftmost position in `STR1` where `STR2` occurs, or -1 if no occurrence.

`Indexr(STR1,STR2)`: (strings.m)

Return rightmost position in `STR1` where `STR2` occurs, or -1 if no occurrence.

`InputFile`: (predefs.m)

Returns the name of the current input file being scanned.

`InputLine`: (predefs.m)

Returns the line number of the current input file being scanned.

`LPAR()`: (predefs.m)

Macro to set a special character, or to return it. See section˜ refCONFIG for a discussion of Special Characters.

`LQ()`: (predefs.m)

Macro to set a special character, or to return it. See section˜ refCONFIG for a discussion of Special Characters.

`Len(STR)`: (predefs.m)

Return the length of the argument `STR`.

`Maketemp(STR)`: (files.m)

Return `STR` with any XXXXXX in it replaced with the process id; see the definition of UNIX's mktemp(3).

`Match(ITEM,V0,C0,V1,C1,V2,C2,...,Vn,Cn,Cd)`: (predefs.m)

If `ITEM` equals `V0`, then `C0` is selected; if `ITEM` equals V_i, then C_i is selected; otherwise `Cd` is selected. This avoids having to expand `ITEM` for each one of a series of comparisons.

`Newmacname(STR)`: (macros.m)

Expands to a string of the form `STR`_d* where d* is a string of decimal digits representing the number of times that `Newmacname` has been called. Used primarily for the generation of unique macro names.

`Opening(FILE)`: (hooks.m)

Invoked whenever a file is opened. The parameter is the name of the file. `FILE` is opened before the macro is expanded.

`PARMSEP()`: (predefs.m)

Macro to set a special character, or to return it. See section˜ refCONFIG for a discussion of Special Characters.

`POOLCH()`: (predefs.m)

Macro to set a special character, or to return it. See section˜ refCONFIG for a discussion of Special Characters.

`Pool(NUM)`: (pools.m)

Expands to the name of the `NUM`th macro pool from the top of the macro search stack (the current macro pool is the 0th pool). Invoked with no arguments, it expands to the current depth of the macro pool stack. There is a question of whether this should return the full pathname of the `NUM`th pool. It currently returns only the pool name and not the path.

`PopConfig`: (predefs.m)

Recover the most recently pushed configuration.

`PopPool(NAME)`: (pools.m)

Removes the top pool on the pool stack, and checks that its name is `NAME`, if the parameter is present. The `NAME` is not looked up until after the pop, so it can be evaluated in the same scope (hopefully) as the corresponding push. Be aware of those situations where pools can be created between paired '`PushPool...PopPool`'s causing the name in `PopPool` to evaluate to a different pool than in the corresponding `PushPool`. So, while '`PushPool(`‘`foo`’`) ...PopPool(`‘`foo`’`)`' looks innocent enough, it might lead to warnings that the push and pop don't match, when in fact you know they do. `PopPool` does not check that the simple name of the pool being popped matches the argument; it evaluates the argument to find the named pool, and then compares that the popped pool is the same as the named pool. `NAME` need not be specified at all, in which case no check for balanced Push-Pop pairs is made. The user may find this useful when one macro pushes a pool on the stack, and another must take it off, but the communication of the name of the pool is not worth the bother. Of course, it makes debugging mismatched Push-Pop pairs more difficult.

`Print(S1,S2,...,Sn)`: (files.m)

Prints the arguments immediately on the current output file. See also Fprint.

`PushConfig(ch1,ch2,ch3,...,ch13)`: (predefs.m)

See the Configure macro for a description of the parameters. In addition, PushConfig keeps the current configuration on a stack for later recovery by the PopConfig macro.

`PushConfigAda(ch1,ch2,ch3,...,ch13)`: (predefs.m)

Pushes a special set of character definitions that is very convenient for scanning Ada program text that for the most part is not m5 macros. See the entry for the `PushScanconfig` macro for the settings of the special characters. In addition, the scanner will recognize Ada-style comments, and not macro-expand their contents.

`PushConfigC(ch1,ch2,ch3,...,ch13)`: (predefs.m)

Pushes a special set of character definitions that is very convenient for scanning C program text that for the most part is not m5 macros. See the entry for the `PushScanconfig` macro for the settings of the special characters. In addition, the scanner will recognize C-style comments, and not macro-expand their contents.

`PushConfigCpp(ch1,ch2,ch3,...,ch13)`: (predefs.m)

Pushes a special set of character definitions that is very convenient for scanning C++ program text that for the most part is not m5 macros. See the entry for the `PushScanconfig` macro for the settings of the special characters. In addition, the scanner will recognize C-style and C++-style comments, and not macro-expand their contents.

`PushPool(POOL,P0,P1,P2,...,Pn)`: (pools.m)

Instantiate the pool `POOL` as the first macro pool to search to expand a macro. All P_i pools are pushed on the stack using the current state of the stack to evaluate the pool names. When `POOL` is popped, they are automatically popped, too. They are pushed in right to left order so the order of writing them is the order of evaluation: the top-of-stack is on the left, the

bottom-most pool on the right. (The alternative would have each push change the evaluation environment for the other pool names.) If `POOL` is a pool path that does not exist, then that pool path is created rooted in the current directory. If the null pool is not the last pool in the list of pools, a warning message is emitted.

`PushScanConfig(ch1,ch2,ch3,...,ch13)`: (predefs.m)

Pushes a special set of character definitions that is very convenient for scanning text that for the most part is not m5 macros. Some special characters are set to values that are not expected in text files, while others are not changed since they are not interpreted specially by m5 except in certain contexts.

```
LQ        ^A (control-A, 0x01)
RQ        ^B
DLR       $
CMT1      ^E
CMT2      ^F
LPAR      (
RPAR      )
COMMA     ,
DASH      -
COUNT     *
POOLCH    ^G
PARMSEP   ,
HQ        ^H
```

`Q0(...)`: (predefs.m)

Return the concatenation of the parameters.

`Q1(...)`: (predefs.m)

Return the concatenation of the parameters surrounded by quotes.

`Q2(...)`: (predefs.m)

Return the concatenation of the parameters surrounded by 2 nested quotes.

`Q3(...)`: (predefs.m)

Return the concatenation of the parameters surrounded by 3 nested quotes.

`Q4(...)`: (predefs.m)

Return the concatenation of the parameters surrounded by 4 nested quotes.

`QS0()`: (predefs.m)

Return the `PARMSEP`-separated list of parameters.

`QS1()`: (predefs.m)

Return the `PARMSEP`-separated list of parameters surrounded by quotes.

`QS2()`: (predefs.m)

Return the `PARMSEP`-separated list of parameters surrounded by 2 nested quotes.

`QS3()`: (predefs.m)

Return the `PARMSEP`-separated list of parameters surrounded by 3 nested quotes.

`QS4()`: (predefs.m)

Return the `PARMSEP`-separated list of parameters surrounded by 4 nested quotes.

`QSn(NUM,P1,P2,...,Pn)`: (predefs.m)

The general form of the quoting list macros: it puts `NUM` quotes around the `PARMSEP`-separated list of the P_i.

`Qn(NUM,P1,P2,...,Pn)`: (predefs.m)

The general form of the quoting concatenation macros: it puts `NUM` quotes around the concatenated P_i.

`RPAR()`: (predefs.m)

Macro to set a special character, or to return it. See section˜
refCONFIG for a discussion of Special Characters.

`RQ()`: (predefs.m)

Macro to set a special character, or to return it. See section˜
refCONFIG for a discussion of Special Characters.

`RandomInt`: (random.m)

Expands to a random, signed integer.

`RandomSelect(C1,C2,...,Cn)`: (random.m)

Randomly selects one of the C_i parameters.

`Redefine(NAME(P1,P2,...,Pn),STR1,...,STRn)`: (macros.m)

Exactly like `Define`, except in how it interacts with the pool stack. If `NAME` is resolvable to a macro in any pool on the stack, then that macro in that pool is set to the new value. Otherwise, a macro is created in the pool on top of the pool stack. If a pool path is specified in `NAME`, then it is (re)defined only in the pool matching the path.

`Reopening(FILE)`: (hooks.m)

Invoked to notify the user that an input file previously suspended is once again the current input stream.

`Reverse(STR)`: (strings.m)

Reverse the parameter `STR`; `abcde` becomes `edcba`.

`Reverseq(STR)`: (strings.m)

Reverse the parameter `STR` and return it surrounded by quotes; `abcde` becomes `'edcba'`. Quoting may be necessary to prevent inadvertant macro expansion.

`Sinclude(FILENAME)`: (files.m)

Same as `Include`, but if `FILENAME` doesn't exist, keeps on going; `Include` will abort if `FILENAME` doesn't exist.

`Strchr(S1,S2)`: (strings.m)

Returns index in `S1` of the first occurrence of the first character of `S2`. Returns -1 if there are none. Note that the null (end of string) character counts, so if `S2` is the empty string, Strchr returns the length of `S1`.

`Strcspn(S1, S2)`: (strings.m)

Returns the length of the initial segment of `S1` that consists entirely of characters *not* in `S2`. Returns 0 if none.

`Strpbrk(S1, S2)`: (strings.m)

Returns index in s1 of first occurrence of any character in `S2`. Returns -1 if none.

`Strrchr(S1,S2)`: (strings.m)

Returns index in `S1` of the last occurrence of the first character of `S2`. Returns -1 if there are none.

`Strspn(S1, S2)`: (strings.m)

Returns the length of the initial segment of `S1` that consists entirely of characters in `S2`. Returns 0 if none.

`Substr(STR,POS,LEN)`: (strings.m)

Return substring of `STR` starting at offset `POS` and going `LEN` characters. Let $p = |\texttt{POS}|$, and $n = |\texttt{LEN}|$. Let $S\langle i, j\rangle$ be the substring of S that starts after the i^{th} character and ends after the j^{th} character. Let s be the number of characters in S.

`POS` $>= 0$	`LEN` $>= 0$	$S\langle p, p+n\rangle$
	`LEN` < 0	$S\langle p, p-n\rangle$
	`LEN` absent	$S\langle p, s\rangle$
`POS` < 0	`LEN` $>= 0$	$S\langle s-p, s-p+n\rangle$
	`LEN` < 0	$S\langle s-p, s-p-n\rangle$
	`LEN` absent	$S\langle s-p, s\rangle$

See the examples in section 8.

`Substrq(STR,POS,LEN)`: (strings.m)

Same as Substr, except that the result is returned surrounded by quotes.

`Substrx(STR,I,J)`: (strings.m)

The indexing version of `Substr`. Return substring of `STR` starting after the $\texttt{I}^{th}$ character and ending after the $\texttt{J}^{th}$ character. Indices greater than the length of the string refer to the beginning (end) of the string if the index is positive (negative). The indices may be in either order. If only one index is present, then the other index defaults to the beginning of the string if the first is positive, and the end of the string if the first is negative. See the examples in section 8.

`Substrxq(STR,I,J)`: (strings.m)

Same as Substrx, except that the result is returned surrounded by quotes.

`Suspending(FILE)`: (hooks.m)

Invoked to notify the user that another file is about to be opened. `FILE` is not closed.

`Syscmd(STR)`: (predefs.m)

Submit `STR` as a subprocess.

`Translit(STR, FROM, TO)`: (strings.m)

Transliterate characters in `STR` from the set specified by `FROM` to the set specified by `TO`. If `Len(TO)` $<$ `Len(FROM)` then characters are deleted from `STR`.

`UndefBlk(MACRONAME)`: (macros.m)

"Undefine and force block." Refer to the entry for `Undefine`. This does the same thing and in addition to removing the definition of `MACRONAME` (if it exists), it also forces the definition of an un-macro, no matter whether the `MACRONAME` was defined elsewhere or not.

`Undefine(MACRONAME)`: (macros.m)

Remove `MACRONAME`'s definition. This is tricky. It doesn't seem safe to allow any pool to delete any other pool's macros. And yet the user may want to say: in this pool, this name is not a macro at all. Therefore, if `MACRONAME` is found in the first pool that matches the associated pool path (if any), its defintion is really removed. If `MACRONAME` is found in another pool on the pool stack, then `Undefine` actually creates an un-macro in the first matching pool: the un-macro, when discovered doing lookup, actually forces the macro-lookup mechanism to stop. The definition of `MACRONAME` becomes a stop-search operator and hides all definitions of this macro that are in pools below this one on the stack. Undefining an un-macro has no effect (see Ununderfine).

`Undivert(I)`: (files.m)

Append diversion `I` to the current diversion. The old m4 assumed that every diversion was eventually `Undiverted` and appended to the standard output. M5 does not `Undivert` files that have been opened or diverted to by name, only those that remain unnamed.

`I` have never used this functionality because `I` never understood its usefulness (i.e. `I` don't think it has any), but `I` assume the following would do the trick:

```
>Define(‘filenum’, Filename(‘yourfile’))
>Divpush(filenum)
...
>Divpop(filenum)
>Undivert(filenum)
```

If `I` were a programmer of conviction, `I` would simply remove most of the m4-compatible macros.

`Ununundefine(MACRONAME)`: (macros.m)

This is verging on the ridiculous, but so be it. This macro exists for those times in which a macro has been `Undefine`d resulting in an un-macro definition in a pool, and now the un-macro needs to be removed. If such an un-macro exists, this macro removes it, thereby exposing any definitions that may have been below it on the stack. If the un-macro doesn't exist in the indicated pool, or if the located macro definition is not an un-macro, then this macro has no effect.

`_`: (hooks.m)

The null pool: the pool in which nothing is defined, nothing is definable, and if it is pushed onto the stack, it stops all name searching at that point on the pool stack. The pool stack is defined to always have the null pool as the (non-poppable) bottom-most pool on the stack. At startup, the root pool `_M5` is the only other pool on the m5 pool stack.

`_M5`: (hooks.m)

The root pool of the pool hierarchy. At startup it is the only pool on the stack (other than the null pool).

`_TOS`: (hooks.m)

An alias for the pool that is currently sitting on top of the pool stack.

`_parent`: (hooks.m)

An alias for the name of the pool that contains another pool.

C Notes on Using M5

C.1 Using M5 as a Preprocessor

One application we experimented with was a plotting preprocessor for LaTeX. A LaTeX picture would be generated by a macro whose argument was a string that contained plotted named points in a plane, along with a description of a picture based on those points; e.g., `Line(A,B)` would draw a line from the point named A to the point named B. The output was the necessary *plot*(1) or LaTeX picture commands to draw the picture. Below is an example plot that would be embedded in a TeX or LaTeX file. Let us assume that we want to define a set of macros that will expand to directives in some plotting language (e.g., LaTeX, Pic, PostScript). We want the invocation of these m5 macros to be embeddable in many different kinds of files without worrying about whether

there is text in the file that conflicts with some of our macro names. For instance, if we define a macro `circle(x,y,r)` that accepts the coordinates of a point and a radius, we don't want this macro invoked where the phrase "circle of friends" occurs in an input file. Whenever these macros are used, we will require that they be invoked between occurrences of the macros `BEGINPLOT` and `ENDPLOT`, on the relatively safe assumption that these strings will not occur in the rest of the file; if they do, they are easily changed[9].

```
BEGINPLOT
points(‘
     A                C

        B                D


             E

                F


’)

Box(A,B)
Box(C,D)
Box(E,F)
Vector(midpt(bottom(Box(A,B))),
       midpt(top(Box(E,F))))
Vector(midpt(bottom(Box(C,D))),
       midpt(top(Box(E,F))))
ENDPLOT
```

Therefore, the plotting macros (`circle`, `line`, `square`, etc.) must be invocable only in a certain context, and be invisible everywhere else. We will do this with scanning configuration macros, and by controlling the pool stack. The `BEGINPLOT` macro would enable all of the necessary macros, and the `ENDPLOT` would disable them so the occurrence of the word 'points' in the rest of the file would not trigger an inappropriate macro expansion. Below is a file of macro definitions that do just that:

```
PushPool(PLOT)@;
:
@@ circle, line, square, etc. macros defined here
:
Define(‘Opening’,‘’)@;
Define(‘Closing’,‘’)@;
Define(‘ENDPLOT’,‘PushPool(scanPool,_)PushScanConfig~_M5()’)@;
```

[9]This system may look nice, but it still suffered from LaTeX's brain-damaged support for pictures. We still spent a lot of time moving the boxes around until LaTeX decided it could draw the indicated vectors; this was too much hassle, so we went back to the X-windows based *idraw*.

```
PopPool(PLOT)@;
@;
PushPool(scanPool)@;
Define(‘BEGINPLOT’,‘PopConfig^G_M5()PopPool~_M5(scanPool)’)@;
PopPool(scanPool)@;
@;
Define(‘Opening’,‘PushPool(PLOT)PushPool(scanPool,_)PushScanConfig~_M5()’)@;
Define(‘Closing’,‘PopPool(scanPool)PopConfig()’)@;
```

The plotting macros contained in the file `userfile` are expanded into file `userfile.2` with the m5 invocation:

```
% m5 plotmacros.m5 userfile >userfile.2
```

where `plotmacros.m5` contains the above macro definitions.

Let's look at the contents of `plotmacros.m5` in detail. The overall strategy is to create two macro pools: `PLOT` and `scanPool`. The pool `PLOT` contains all of the plotting macros for lines, squares, and whatever other graphical objects we want. It also contains the macro `ENDPLOT`, which hides all of these macros at the end of a plotting section in the input file.

The pool `scanPool` contains only the macro `BEGINPLOT` which triggers the availability of the plotting macros. Before any occurrence of `BEGINPLOT`, `scanPool` is on the top of the pool stack, and the null pool '_' is immediately below it. This means that m5 will recognize one and only one macro: `BEGINPLOT`. We also make sure that m5 doesn't swallow any quote characters or parentheses by pushing the special scanning configuration via the macro `PushScanConfig`. This pool stack and scanning configuration is set up any time a file is opened (see the definition of the `Opening` and `Closing` macros).

The definition of `BEGINPLOT` requires some explanation. The macro is defined to be a `PopConfig` followed by a `PopPool`. But these macros cannot be invoked if we depend on name lookup through the pool stack since they are defined in the root pool `_M5`, which is below an occurrence of the null pool `_`. Therefore, `BEGINPLOT` must invoke them explicitly with a pool path. Since the `PushScanConfig` macro has redefined the pool character to be a control-G (`^G`), then that character is used in the full pool pathname for the macro `PopConfig`. After `PopConfig` has expanded, we can use the twiddle character to invoke `PopPool`, which exposes the `PLOT` pool to the top of the pool stack.

Also included are rudimentary definitions of the `Opening` and `Closing` macros to show the proper place for checking file names to determine what kind of processing may go on in them. If we opened a LaTeX file, we would want to produce one kind of plotting specifications, but if we opened a C source file, we would want to do another. This was most easily accomplished by defining two pools of circle, line, box, etc. macros. The `Opening` macro could determine which pool to push on the stack based on the file suffix.

Finally, note that `BEGINPLOT` pops pools from the stack and `ENDPLOT` pushes pools. It could have been the other way around, with `BEGINPLOT` pushing the pools for the plot macros along with the null pool, and `ENDPLOT` then popping them from the stack. All that would be required is that the `Opening` macro push the appropriate pools for scanning the file.

C.2 Tips on the construction of macros that construct macros

M5 was designed to enable easy definition of macros that defined macros. For example, assume we have a macro `Set_A(1)` that expands into `Set_1_A`. Such a macro would be defined as `Define(‘Set_A()’,‘Set_$1_A’)`. Assume further that this macro is to be generated from a set of declarations: `DECLARE_VARS(A,B,C,D,E)`. That means that the macro `DECLARE_VARS` defines the macros `Set_A`, `Set_B`, `Set_C`, ... It does this recursively, using another macro to do the actual definition.

```
Define(‘DECLARE_VARS(var,rest)’,
       ‘Ifeq($(var),,,
             ‘DECLARE_VAR(‘$(var)’)@@
              DECLARE_VARS($‘rest,’)’)’)@;
Define(‘DECLARE_VAR(var)’,‘Define(‘Set_$(var)(num)’,‘Set_$$(num)_$(var)’)’)@;
```

So now `DECLARE_VARS(A,B,C)` defines the three macros `Set_A`, `Set_B`, and `Set_C`. They are defined such that `Set_X(Y)` results in the string `Set_Y_X`.

Now assume we need a file of macro definitions which is to be generated by the invocation of `DECLARE_VARS`. The contents of the file should look like:

```
Define(‘Set_A()’,‘Set_$1_A’)
Define(‘Set_B()’,‘Set_$1_B’)
Define(‘Set_C()’,‘Set_$1_C’)
```

This file can then be used as input to modify (macro expand) another file or set of files. We modify `DECLARE_VARS` to define the indicated macros in a special pool:

```
Define(‘DECLARE_VARS(var,rest)’,
       ‘PushPool(SETASIDE)@@
        Ifeq($(var),,,
             ‘DECLARE_VAR(‘$(var)’)@@
              DECLARE_VARS($‘rest,’)’)@@
        PopPool(SETASIDE)@@
        ’)@;
Define(‘DECLARE_VAR(var)’,
       ‘Define(‘Set_$(var)(num)’,‘Set_$$(num)_$(var)’)’)@;
```

Once all of the needed macros are created in the pool `SETASIDE`, then we can create the desired file of macros with:

```
Divpush(‘myfilename.m5’)@@
DumpPool(‘SETASIDE’)@@
Divpop‘’@@
```

which produces the needed file of macro definitions. When file ‘`myfilename.m5`’ is included in other m5 runs, then the desired macros are automatically (re-)defined.

C.3 More Troubles with Premature Evaluation

Consider the situation where we are defining macros by constructing names; e.g. you are reading variables from a file and collecting information about them by defining macros based on the names of the variables. You might like to know if certain information about a variable has been defined before:

```
>Define(‘Defined(macro)’,‘Ifdef(MakeName($‘macro’,‘DEF’),‘YES’,‘NO’)’)
>Define(‘MakeName(var,suffix)’,‘$(var)_$(suffix)’)
>Defined(‘var1’)
NO
>Define(‘var1_DEF’)
>Defined(‘var1’)
<stdin>: 15: m5: lookup: Null name passed to lookup!
<stdin>: 15: m5: ; parsed to end of:
<stdin>: 15: m5: Current state of the pool stack:
<stdin>: 15: m5:   _M5
<stdin>: 15: m5:    (Fatal)
```

The error occurs because `var1_DEF` is defined to be the empty string, which is an illegal first parameter to the `Ifdef` macro. If the macro `MakeName` is going to construct a name which could possibly be a macro name, then the construction must be surrounded by quotes on return. A set of macros `Q0`, `Q1`, `Q2`, `Q3`, and `Q4` return the concatenation of their parameters surrounded by zero, one, two, three or four quotes, respectively.

The use of the `Q1` macro solves our problem:

```
>Define(‘Defined(name)’,‘Ifdef(MakeName($‘name’,‘DEF’),‘YES’,‘NO’)’)
>Define(‘MakeName(var,suffix)’,‘Q1($(var),‘_’,$(suffix))’)
>Defined(‘var1’)
NO
>Define(‘var1_DEF’)
>Defined(‘var1’)
YES
```

This particular problem has two other solutions:

```
>Define(‘MakeName(var,suffix)’,‘‘$(var)_$(suffix)’’)’)
>Defined(‘var1’)
NO
>Define(‘var1_DEF’)
>Defined(‘var1’)
YES
```

and

```
>Define(‘MakeName(parms)’,‘‘$(parms,,_)’’)
>Defined(‘var1’)
NO
```

```
>Define(‘var1_DEF’)
>Defined(‘var1’)
YES
```

C.4 Using M5 in its Own Implementation

The source for m5 resides in a set of source files whose names have the suffix `.m`. M5 is used as a preprocessor that takes `.m` files to `.c` C source files. The code for the functions that implement the predefined macros is much more understandable when we can write

```
MACRO(Predefmacro(PARM1,PARM2),1,2,(cat1,cat2),
^A <documentation in LaTeX format> ^B
)
{
    <C code using PARM1 and PARM2 as variables>
    }
```

(The `^A` and `^B` are control characters used to delimit LaTeX text. These characters are easily typed in most text editors (e.g. emacs) and do not interfere with C or LaTeX.) This creates the .h file containing the `extern` declaration of the function `doPredefmacro`, the `.doc` files which make the appendix B for this *User's Guide*, the category lists used to construct appendix A, and, more importantly, allows the programmer to refer to the parameters to the *macro* by name in the C code. The C functions are only passed a pointer to the runtime argument stack. A reference to a particular argument to the macro is an indexed reference off the parameter list pointer. Using the `MACRO` macro, `doPredefmacro` is the function that implements the predefined macro `Predefmacro`, it can accept as few as one but at most two parameters, and they can be referred to in the C code with the names `PARM1` and `PARM2`. The `MACRO` macro defines the macros `PARM1` and `PARM2` to expand, in the C code body, into the appropriate references to the macro parameter stack (it also defines a few other useful macros such as `PARM1Idx` which is the offset of the `PARM1` parameter on the macro parameter stack). Furthermore, with m5's scoping mechanism, the LaTeX documentation is collected into a file for sorting and inclusion in the source file for this manual.

Another example is the declaration for the FlushPool predefined macro:

```
MACRO(FlushPool(POOL1, POOL2, ..., POOLn),1,,(macros,pools),
^A <documentation> ^B
)
```

which says that `FlushPool` is predefined to accept an unlimited number of parameters (but must have at least one) and we will refer to the last one as `POOLn`. Then the C code can use `POOL1Idx()` and `POOLnIdx` to loop over each of the macro's actual parameters. Furthermore, `POOL1`, etc. are also defined to be macros that will expand in the LaTeX documentation into `{\tt POOL1}`, etc. Since three dots appear in the parameter list to indicate unlimited numbers of parameters, another macro is defined that lets the documentation refer to the arbitrary i^{th} parameter. `POOL_i` in the LaTeX documentation will expand into `${\tt POOL}_i$`.

A final example comes from the `Match` macro, which we will show in more detail. The source code looks like:

```
MACRO(Match(ITEM,V0,C0,V1,C1,V2,C2,...,Vn,Cn,Cd),1,,(condl),
^A
If ITEM equals V0, then C0 is selected;
if ITEM equals V_i,
then C_i is
selected; otherwise Cd is selected.  This avoids having to expand
ITEM for each one of a series of comparisons.
^B)
{
  int i = 0;
  argChk;
  while (C0Idx(i) <= CdIdx) {
      if (strcmp(ITEM,V0(i)) == 0) {
          pushStrIntoInput(C0(i));
          return;
          }
      i += 2;
      }
  pushStrIntoInput(Cd);
  }
```

The MACRO macro expands the documentation into

```
\item{{\tt Match(ITEM,V0,C0,V1,C1,V2,C2,...,Vn,Cn,Cd):} (predefs.m) }

If {\tt ITEM} equals {\tt V0}, then {\tt C0} is selected;
if {\tt ITEM} equals ${\tt V}_i$,
then ${\tt C}_i$ is
selected; otherwise {\tt Cd} is selected.  This avoids having to expand
{\tt ITEM} for each one of a series of comparisons.
```

The resulting source code is:

```
void doMatch(char **argPtr,int c)
{
  int i = 0;
  static char *__fcn_name = "Match";
if (c < 1) {
argChk_min(__fcn_name,1);
return;
}
;
  while ((i+3) <= c) {
      if (strcmp(argPtr[1],argPtr[(i+2)]) == 0) {
          pushStrIntoInput(argPtr[(i+3)]);
          return;
          }
```

```
      i += 2;
      }
  pushStrIntoInput(argPtr[c]);
  }
```

Notice that the symbolic names of the parameters to the *macro* have been converted into the proper data structure references, and that the `argChk` macro expands into code to check that the number of parameters is correct for the `Match` macro.

C.5 Grammar Expansion

The following example was derived from the need to decipher students' context-free grammars. Some of their grammars were sufficiently complex that it was not immediately evident whether they had described the given language or not. These macros were designed to generate random strings from small cfgs.

```
DefinePool(‘LHS’)@;
DefinePool(‘Rules’)@;

@{

The grammar is specified in pairs with the Grammar macro.
{ S->aBc, S->B, S-> } (last rule S goes to empty string) would be entered
in the form:
Grammar(S,aBc,S,B,S,)

We make the (hopefully not too restrictive) assumption that we’re dealing with
smallish grammars, that all of the non-terminals and terminals are single
alphabetic characters.

@}

Define(‘Grammar(rules)’,
       ‘FlushAllLHSs‘’@@
        :Grammar:($‘rules,’)@@
        ’)@;
Define(‘:Grammar:(lhs,rhs,rules)’,
       ‘Ifneq($‘lhs’,‘’,@@ then
           ‘Rule($‘lhs’,$‘rhs’)@@
            :Grammar:($‘rules,’)’)’)@;

Define(‘FlushAllLHSs’,
       ‘FlushPool(‘Rules’)@@
        PushPool(‘LHS’)@@
        :FlushAllLHSs:(AllPoolNames)@@
        PopPool(‘LHS’)@@
```

```
        ')@;

Define(`:FlushAllLHSs:(pool,pools)',
       `Ifneq($`pool',``,@@
              `FlushPool($`pool')@@
               :FlushAllLHSs:($`pools,')')')@;

@{
The Rule macro can take a rule S->aBd in the form (SaBd), (S,aBd),
or (S,a,B,d), whichever is most convenient
@}

Define(`Rule(exps)', @@ we assume single character alphabets
       `:Rule:(QSO(split($`exps,')))')@;

@{
The :Rule: macro expects S->aBd to be (S,a,B,d)
@}

Define(`:Rule:(lhs,rhs)',
       `FlushPool(`LHS')@@
        PushPool(`LHS')@@
        PushPool(`$(lhs)')@@
        Ifeq($`rhs,, ',``,@@ Then
            `Define(`ExpandsToEmpty')',@@ else
            `Define(Newmacname,$`rhs,, ')')@@
        PopPool(`$(lhs)')@@
        PopPool(`LHS')@@
        ')@;

@{
Take (abc,def,ghi,...) and return (a,b,c,d,e,f,g,h,i,...)
@}

Define(`split(exp,exps)',
       `Ifneq($`exp',``,@@ Then
           `Ifeq($`exps,, ',``,`splitc($`exp')',@@ELSE
               `splitc($`exp')`,'split($`exps,')')')')@;

@{
Take (abc) and return (a,b,c)
@}

Define(`splitc(exp)',
       `Ifneq(Len($`exp'),0,@@THEN
```

```
            `Ifeq(Len($`exp'),1,$`exp',@@ELSE
                `Substrq($`exp',0,1)`,'splitc(Substrq($`exp',1))')')')@;

@{
Give Expand a non-terminal, and it will generate a random string from that
non-terminal.
@}

Define(`Expand(S)',
       `ConstructDataBase()@@
        PushPool(`Rules')@@
        $(S) @@
        PopPool(`Rules')')@;

@{
After all the rules have been read in, the macros for the non-terminals
must be created.  For each non-terminal, a macro is defined that randomly
selects from the possible rules with that non-terminal on the LHS.  In order
to ensure that this expansion is likely to halt, this macro for the
non-terminal is defined so that the non-terminal is as likely to expand to
the empty string as to any other non-empty expansion.  For some grammars, this
means that (too) many of the expansions are all empty, but that is better
than having expansions that go forever or fill up the screen.
@}

Define(`ConstructDataBase()',
       @{ first, for each rule with the same LHS, e.g.
          LHS -> rhs1, LHS -> rhs2, LHS -> ,
          construct Define(LHS,RandomSelect(rhs1,rhs2,,)
          Notice that if LHS->e is a rule, then there must be as many
          of them as there are other rules (to better the odds that
          an expansion eventually ends).
       @}
       `PushPool(`LHS')@@
        Ifndef(`RHS',`ForEachLHS(AllPoolNames)')@@
        PopPool(`LHS')@@
        ')@;

@{
This constructs the RHS selection macro.
@}

Define(`ForEachLHS(name,names)',
       `Ifneq($`name',`',@@
           `Redefine(`RHS')@@
```

```
            Undefine(‘EmptyParms’)@@
            PushPool($‘name’)@@
            IfdefTops(‘ExpandsToEmpty’,@@
                ‘Undefine(‘ExpandsToEmpty’)@@
                 Define(‘EmptyParms~_parent’,‘’)’)@@
            ForEachRHS(AllMacroNames)@@
            PopPool($‘name’)@@
            Define(‘$(name)~_M5~Rules’,@@
               ‘RandomSelect(’QS1(Defn(‘RHS’)@@
                                  Ifdef(‘EmptyParms’,‘QS1(Defn(‘EmptyParms’))’)@@
                                  )@@
                             ‘)’)@@
            ForEachLHS($‘names,’)@@
            ’)’)@;

@{
This collects the RHSs.
@}

Define(‘ForEachRHS(name,names)’,
       ‘Ifneq($‘name’,‘’,@@
           ‘Ifempty(‘RHS’,‘Append(‘RHS’,QS2(Defn($‘name’)))’,@@
                          ‘Append(‘RHS’,‘,’QS2(Defn($‘name’)))’)@@
            Ifdef(‘EmptyParms’,‘Append(‘EmptyParms’,‘,’)’)@@
            ForEachRHS($‘names,’)@@
            ’)@@
        ’)@;

Define(‘Do(cnt,symbol)’,
        ‘Ifneq($‘cnt’,‘0’,@@
            ‘Expand($‘symbol’)
@@
             Do(Incr($‘cnt’,-1),$‘symbol’)’)’)@;

Define(‘Do20(symbol)’,‘Do(20,$‘symbol’)’)@;
```

Given the following small grammar, we get as output:

```
Define(TestG,
‘Grammar(
        S,aSd,  S,aTc,  T,aTc,
        T,bXc,  X,bXc,  S,bXc,
        S,bWd,  W,bWd,  W,bXc,
        T,, X,, W,)
    Do(20,S)’
```

```
)@;

TestG

a b  d d
b  d
a b  d d
b b  c c
b  c
a  c
b  c
b b b b  c c c d
b b  c d
a a b b  c c d d
a  c
a a  c d
a a a b b b  c c c c d d
b  c
b b  c c
b b  c c
a a b b b  c c c c d
a b  d d
b  c
b  c
```

C.6 Turing Equivalence

Finally, a proof that m5 is Turing equivalent. This is easily done by implementing a Turing machine in m5. This is a simplified version that does not worry too much about robustness. For example, it is possible to define a Turing machine whose tape can spell out an m5 macro name; that will cause problems for these macro definitions. It is a simple matter to create macros that can prevent this from happening, but it is more macro code than you are probably willing to read.

```
@{
The basic idea is to create a pool corresponding to the name of the
Turing Machine.  There are two subpools, act and nxt, that define the action
and the next state for this machine given a state and a symbol.
For each arc in the state machine there is a macro
state:sym~TM~act which expands into the appropriate action and
state:sym~TM~nxt which expands into the appropriate next state.

The macro CurTM defines the name of the active Turing Machine.
The macro CurState defines the name of the current state.

The macro LeftTape defines the tape to the left of the head.
The macro RightTape defines it to the right.
```

```
The macro Head defines the symbol currently under the head.

@}

Define(‘TuringMachine(name)’,
       ‘DefinePool($‘name’)@@
        DefinePool(‘$(name)~act’)@@
        DefinePool(‘$(name)~nxt’)@@
        Define(‘CurTM’,$‘name’)Dnl‘’)@;

Define(‘Alphabet(str)’,‘Define(‘Alpha’,$‘str’)Dnl’)@;

Define(‘Arc(state,syms,action,new)’,
       ‘Ifneq($‘syms’,‘’,
              ‘Arc:($‘state’,Substr($‘syms’,0,1),$‘action’,$‘new’,CurTM)@@
               Arc($‘state’,Substr($‘syms’,1),$‘action’,$‘new’,CurTM)@@
               ’)Dnl‘’)@;

Define(‘Go(in,st)’,@@ assumes CurTM is the machine to use
       ‘PushPool(Ctl)@@
        Redefine(‘LeftTape’,$‘in’)@@
        Redefine(‘RightTape’,‘’)@@
        Redefine(‘Head’,‘_’)@@
        Redefine(‘CurState’,$‘st’)@@
      Go:‘’@@
        PopPool(Ctl)@N
        LeftTape~Ctl‘’Head~Ctl‘’RightTape~Ctl@N
        ’)@;

Define(‘Arc:(state,sym,action,new,tm)’,
       ‘Define(‘$(state):$(sym)~$(tm)~act’,$‘action’)@@
        Define(‘$(state):$(sym)~$(tm)~nxt’,$‘new’)@@
        ’)@;

PushPool(‘Ctl’)@;

Define(‘Action(sym,tm)’,‘$(sym)~$(tm)~act’)@;
Define(‘NextSt(sym,tm)’,‘Redefine(‘CurState’,$(sym)~$(tm)~nxt)’)@;

Define(‘Go:’,
       ‘Ifneq(CurState,H,‘Next~Ctl(CurState,Head~Ctl)‘’Go:’)’)@;

Define(‘Next(cur,sym)’,
       ‘Action(‘$(cur):$(sym)’,CurTM)‘’@@ side effect: changes CurState
        NextSt(‘$(cur):$(sym)’,CurTM)‘’@@ side effect: modifies tape
```

```
        ')@;

@; Write a new symbol under the head

Define(‘Write(sym)’,‘Redefine(‘Head’,‘$(sym)’)’)@;
Define(‘W(sym)’,‘Write($‘sym’)’)@;

@; Do a move left on the tape

Define(‘L’,‘Left’)@;
Define(‘Left’,
       ‘Redefine(‘RightTape’,Head‘’RightTape)@@
        Redefine(‘Head’,RightMost(LeftTape))@@
        Redefine(‘LeftTape’,Substrx(LeftTape,-1))@@
        ')@;

@; Do a move right on the tape

Define(‘R’,‘Right’)@;
Define(‘Right’,
       ‘Redefine(‘LeftTape’,LeftTape‘’Head)@@
        Redefine(‘Head’,LeftMost(RightTape))@@
        Redefine(‘RightTape’,Substrx(RightTape,1))@@
        ')@;

@; Extract the right-most symbol from the argument; return blank if argument
@; empty

Define(‘RightMost(str)’,
       ‘Ifeq(‘$(str)’,‘’,‘_’,‘Substr(‘$(str)’,-1,1)’)’)@;


@; Extract the left-most symbol from the argument; return blank if argument
@; empty

Define(‘LeftMost(str)’,
       ‘Ifeq(‘$(str)’,‘’,‘_’,‘Substrx(‘$(str)’,0,1)’)’)@;

PopPool(‘Ctl’)@;

@; Example:
@;      convert all b’s to a’s and vice versa

TuringMachine(Complement)
Alphabet(‘ab’)
```

```
Arc(q0,ab_,L,q1)
Arc(q1,a,W(b),q0)
Arc(q1,b,W(a),q0)
Arc(q1,_,R,q2)
Arc(q2,ab,R,q2)
Arc(q2,_,W(_),H)
Go('aaabbbabab',q0)

_bbbaaababa_
```