M5 Text Processing Language User's Guide

Table of Contents

The M5 macro preprocessor enhances the Gnu M4 macro preprocessor, adding features typical of programming languages.

1. General information

1.1. Overview

M5 is a macro preprocessor on steroids. It is an easy tack-on to any text format to enable arbitrary text processing, extending the capabilities and syntax of the underlying language (or simply text). It is built on the simple principle of text substitution but provides robust features on par with other programming languages. It is optimized for simple use cases and for comprehension by non-experts while being capable of general-purpose programming.

M5 was developed by Redwood EDA, LLC and is used in conjunction with TL-Verilog, but it is appropriate as an advance macro preprocessor or code generator for any target language or even as a stand-alone language. M5 is constructed with a bit of pre-preprocessing, providing syntactic sugar, and then the use of the Gnu M4 macro preprocessor with an extensive library.

This chapter provides background and general information about M5, guidance about this specification, and instructions for using M5.

1.2. About this Specification

This document is intended to stand on its own, independent of the M4 documentation. The M4 documentation can, in fact, be confusing as M5 has philosophical differences. Differences versus M4 are described in M5 Versus M4.

1.3. Getting Started with M5

1.3.1. Configuring M5

M5 adds a minimal amount of syntax, and it is important that this syntax is unlikely to conflict with the output language syntax. Most notably M5 introduces quote characters used to provide literal text that is not subject to macro preprocessing. By default M5 uses `['` and `']`. It can be configured to use different quote characters by modifying two simple scripts that substitute quotes in the input and output files and configure M4 to use the substituted quote characters. Similar scripts must be applied to all `.m4` files including the ones that define M5 to change all `['` / `']` quotes to the desired quotes.

Additionally, M5 defines a comment syntax that can be configured in the pre-preprocessing script.

1.3.2. Running M5

The Linux command:

```
m5 < in-file > out-file
```

(TODO: Provide m5 script that does `--prefix_builtins`.)

runs M5 in its default configuration.

1.3.3. Ensure No Harm

First, be sure M5 processing does nothing on a file with no M5 syntax. As used for TL-Verilog, M5 should output the input text, unaltered, as long as your file contains no:

- quotes, e.g. `[', ']`)
- `m5` or `m4`

In other configurations, the following may also result in processing:

- vanishing comments, e.g. `///`, `/`, `/`
- code blocks, e.g. `[` or `{` followed by a new line or `]` or `}` beginning a line after optional whitespace

1.4. M5's Place in the World

This section describes the history of and motivation for M5 and it's relation to M4 and TL-Verilog.

1.4.1. M5's Association with TL-Verilog

Although M5 was developed for TL-Verilog, it is not specifically tied to TL-Verilog. It does, however, like all M4 libraries, depend upon a specific set of M4 syntax configurations, and these configurations were chosen to best suit TL-Verilog.

The required M4 configurations are described in Getting Started with M5. These configurations establish:

- builtin macro prefix: `m4_`
- quote characters: `[` and `]`

TL-Verilog supports other TL-Verilog-specific macro preprocessing that is documented separately.

TL-Verilog preprocessing supports special code block syntax. To improve readability for TL-Verilog users, this document does assume support for this syntax. Code Blocks describes equivalent syntax that can be used without TL-Verilog preprocessing.

1.4.2. M5 Versus M4

M5 uses M4 to implement a macro-preprocessing language with some subtle philosophical differences. While M4 is adequate for simple substitutions, M5 aims to preserve the conceptual simplicity of macro preprocessing while adding features that improve readability and manageability of more complex use cases.

M4 favors aggressive macro expansion, which frequently leads to the need for multiple levels of nested quoting to prevent unintended substitutions. This leads to obscure bugs. M5 implicitly quotes arguments and returned text, favoring explicit expansion where desired.

1.4.3. M5 Above and Beyond M4

M5 contributes:

- features that feel like a typical, simple programming language
- a categorization of macros as variables, functions, and traditional macros
- named arguments for improved readability
- a moderate level of variable typing
- scope for variable declarations
- an intentionally minimal amount of syntactic sugar
- document generation assistance
- debug aids such as stack traces
- safer parsing and string manipulation
- a richer core library of utilities
- a future plan for modular libraries

1.4.4. Limitations of M5

M4 has certain limitations that M5 is unable to address. M5 uses M4 as is without modifications to the M4 implementation (though these limitations may motivate changes to M4 in the future).

1.4.4.1. Modularity

M4 does not provide any library, namespace, and version management facilities. Though M5 does not currently address these needs, plans have been sketched in code comments.

1.4.4.2. String processing

While macro processing is all about string processing, safely manipulating arbitrary strings is not possible in M4 or it is beyond awkward at best. M4 provides `m4_regexp`, `m4_patsubst`, and `m4_substr`. These return unquoted strings that will necessarily be elaborated, potentially altering the string. While M5 is able to jump through hoops to provide `m5_regexp` and `m5_substr` (for strings of limited length) that return quoted (literal) text, `m4_patsubst` cannot be fixed. The result of `m4_patsubst` can be quoted only by quoting the input string, which can complicate the match expression, or by ensuring that all text is matched, which can be awkward, and quoting substitutions.

In addition to these issues, care must be taken to ensure resulting strings do not contain mismatching quotes or parentheses or combine with surrounding text to result in the same. Such resulting mismatches are difficult to debug. M5 provides a notion of "unquoted strings" that can be safely manipulated using `m5_regex`, and `m5_substr`.

Additionally the regex configuration used by M4 is quite dated. For example, it does not support lookahead, lazy matches, and character codes.

1.4.4.3. Instrospection

Introspection is essentially impossible. The only way to see what is defined is to dump definitions to a file and parse this file.

1.4.4.4. Recursion

Recursion has a fixed (command-line) depth limit, and this limit is not applied reliably.

1.4.4.5. File format

M4 is an old tool and was built for ASCII text. UTF-8 is now the most common text format. It is a superset of ASCII that encodes additional characters as two or more bytes using byte codes (0xFF-0x10) that do not conflict by those defined by ASCII (0x7F-0x00). All such bytes (0xFF-0x10) are treated as characters by M4 with no special meaning, so these characters pass through, unaffected, in macro processing like most others. There are two implications to be aware of. First, `m5_len` provides a length in bytes, not characters. Second, `substr` and regular expressions manipulate bytes, not characters. This can result in text being split in the mid-character, resulting in invalid character encodings.

1.4.4.6. Debugging features

M4's facilities for associating output with input only map output lines to line numbers of top-level calls. (TL-Verilog tools have mechanisms for line tracking.)

M4 does not maintain a call stack. M5 adds one which tracks function names and arguments of calls, but it cannot track line numbers.

M4 and M5 have no debugger to step through code. Printing is the debugging mechanism of choice.

1.5. Status

Certain features documented herein currently work only in conjunction with the TL-Verilog macro preprocessor. The intent is to support them in M5 itself, and they are documented with that in mind. Such features include:

- code blocks
- vanishing comments
- use of control-character quotes

2. Processing Text and Calling Macros

2.1. Macro Preprocessing in General

M5, like other macro preprocessors, processes a text file sequentially with a default behavior of passing the input text through as output text. Parameterized macros may be defined. When a recognized macro name appears in the input text, it (and its optional argument list) will be substituted for new text according to its definition. Quotes (`['` and `']`) may be used around text to prevent substitutions.

2.2. Macro Substitution

The following illustrates a macro call:

```
m5_foo(hello, 5)
```

A well-formed M5 macro name begins with `m5_` and is comprised entirely of word characters (`a-z`, `A-Z`, `0-9`, and `_`).

NOTE	It is possible to define macros with names containing non-word characters, but these will not substitute as described above. They can only be called indirectly. In addition to `m5_` macros, the M4 macros from which M5 is constructed are available, prefixed by `m4_`, though their direct use is discouraged. Though discouraged, be aware that it is possible, using `m4_` macros, to define macros without these prefixes.

When a well-formed macro name appears (in unquoted input text), delimited by non-word characters (or the beginning or end of the file), the name is looked up in the set of defined macro names. If the name is defined, a subsequent `(` would begin an argument list. This list ends with a matching, unquoted `)`. (For details, see Arguments.) Once the argument list has been fully processed, or in the absence of an argument list, the macro is "called". It and its optional argument list are substituted with the evaluation of the text resulting from the macro call. This text is passed through to the output, and processing continues.

Many macros result in literal (quoted) text to avoid subsequent evaluation. In some cases, literal result text is the normal case but alternate macros are provided with unquoted output. By

convention these are named with an `_eval` suffix (or the `eval` macro, itself). Note that the definitions (see [m5_defn]) of `_eval` macros will end with `['']`. This is required by M4 to isolate the resulting text from subsequent text.

2.3. Quotes

Unwanted substitution can be avoided using quotes. In M5, quotes are `[` and `']`. Quoted text begins with `[`. The quoted text is parsed only for `[` and `']` and ends at the corresponding `']`. Intervening characters that would otherwise have special treatment, such as `m5`*, `(`, and `)`, have no special treatment when quoted. The quoted text passes through to the resulting text, including internal matching quotes, without involvement in any substitutions. The outer quotes themselves are discarded. The end quote acts as a word boundary for subsequent text processing.

Quotes can be used to delimit words. For example, the empty quotes below:

```
Index['']m5_Index
```

enable `m5_Index` to substitute, as would:

```
['Index']m5_Index
```

Special syntax is provided for multi-line literal text. (See [blocks].) Outside of those constructs, quoted text should not contain new-lines. Instead, the [m5_nl] macro provides a literal new-line character, for example:

```
['Index']m5_Index['']m5_nl
```

2.4. Comments

2.4.1. Vanishing Comments (/// and /**...**/)

The following illustrates vanishing comments:

```
/// This line comment will disappear.
/** This block comment will also disappear. **/
```

Block comments beginning with `/**` and ending with `**/` and line comments beginning with `///` and ending with a new line are stripped from the source file prior to other processing (except for new lines). As such:

- Vanishing-commented parentheses and quotes are not visible to parenthesis and quote matching checks, etc.
- Vanishing comments may follow the `[` or `{` beginning a code block or after a comma and prior to

an argument that begins on the next line without affecting the code block or argument.

NOTE	Any text immediately following **/ will, after stripping the comment, begin the line. Comments are stripped after indentation checking. It is thus generally recommented that multi-line block comments end with a new line.

2.4.2. Preserved Comments (//)

Line comments in the target language (//) have special treatment to avoid unexpected expansion of commented macros. Unquoted // comments until the next new line, pass through to the output as literal text.

CAUTION	This behavior is both helpful and dangerous. It can hide quotes as a result of dynamic evaluation, leading to mismatched quotes that are inconsistent with static checking which ignores //. It is best to use vanishing quotes to disable macro code.

NOTE	/ and / are not recognized as block comments. In target languages that support this comment style, their use can be convenient for seeing evaluations in output comments. ['//'] (or similar) can also be used to pass macro evaluations in comments.

2.5. Arguments

TODO: Macro categories have not been introduced yet.

Traditional macros and function calls pass arguments within (and) that are comma-separated. For each argument, preceding whitespace is not part of the argument, while postceding whitespace is. Specifically, the argument list begins after the unquoted (. Subsequent text is elaborated sequentially (invoking macros and interpreting quotes). The text value of the first argument begins at the first elaborated non-whitespace charater following the (. Unquoted (are counted as an argument is processed. An argument is terminated by the first unquoted and non-parenthetical , or) in the resulting elaborated text. A subsequent argument, similarly, begins with the first non-whitespace character following the , separator. Whitespace includes spaces, new lines, and tabs. An unquoted) ends the list.

Some examples to illustrate preceding and postceding whitespace:

```
m5_macro(foo, ['Args:$1,$2'])
```

```
m5_foo(  A,  B)        ==> Yields: "Args:A,B"
m5_foo(    ['']  A,B)  ==> Yields: "Args:  A,B"
m5_foo(  A  ,  B  )    ==> Yields: "Args:A  ,B  "
```

Arguments can be empty text, such as () (one empty argument) and (,) (two empty arguments).

`([''])` and `(['', ''])` are identical to the previous cases and are preferred, to express the intended empty arguments more clearly.

There are a few gotchas to watch out for.

When argument lists get long, it is useful to break them up on multiple lines. The new lines should precede, not postcede the arguments. E.g.:

```
m5_foo(long-arg1,
       long-arg2)
```

Notably, the closing parenthesis should **not** be on a the next line by itself. This would include the new line and spaces in the second argument.

3. Multi-line Constructs: Blocks and Bodies

3.1. What are Bodies and Blocks?

A "body" is a parameter or macro value that is to be be evaluated in the context of a caller. Macros, like `m5_if` and `m5_loop` have immediate body parameters. These bodies are to be evaluated by these macros in the context of the caller. The final argument to a function or macro declaration is an indirect body argument. The body is to be evaluated, not by the declaration macro itself, but by the caller of the macro it declares.

NOTE	Declaring macros that evaluate body arguments requires special consideration. See [evaluating_bodies].

"Code blocks" are convenient constructs for multi-line body arguments formatted like code.

A "Text block" construct is also available for specifying multi-line blocks of arbitrary text, indented with the code.

3.2. Macro Bodies

A body argument can be provided as a quoted string of text:

```
m5_if(m5_A > m5_B, ['['Yes, ']m5_A[' > ']m5_B'])   // Might result in "Yes, 4 > 2".
```

Note that the quoting of `['Yes, ']` prevents misinterpretation of the `,` as an argument separator as the body is evaluated.

This syntax is fine for simple text substitutions, but it is essentially restricted to a single line which is unreadable for larger bodies that might define local variables, perform calculations, evaluate code conditionally, iterate in loops, call other functions, recurse, etc.

3.3. Code Blocks

M5 supports a special multi-line syntax convenient for body arguments, called "code blocks". These look more like blocks of code in a traditional programming language. Aside from comments and whitespace, they contain only macro calls and variable elaborations ("statements"). The resulting text of the code block is constructed from the results of these macro calls.

The code below is equivalent to the example above, expressed using a code body, and assuming it is called from within a code body.

```
~if(m5_A > m5_B, [
   ~(['Yes, '])
   ~A
   ~([' > '])
   ~B
])
```

The block begins with `[`, followed immediately by a new line (even if commented by `//`). It ends with a line that begins with `]`, indented consistently with the beginning line. The above code block is "unscoped". A "scoped" code block uses, instead, `{` and `}`. Scopes are detailed in Scope.

The first non-blank line of the block determines the indentation of the block. Indentation uses spaces; tabs are discouraged, but must be used consistently if they are used. All non-blank lines at this level of indentation are either preserved comments or statements (after stripping vanishing comments). (All lines are statements in the above example.) Lines with deeper indentation would continue a statement. A continuation line either begins a macro argument or is part of its own (nested) code block argument.

Statements that produce output (as all statements in the above example do) and variable elaborations must be preceded by `~` (and others may be). This simply helps to identify the source of code block ouput. The `~(⋯)` syntax has the same effect as `~out(m5_⋯)` and is used to directly provide output text. A `m5_` prefix is implicit on statements. In the rare (and discouraged) event that a macro without this prefix is to be called, such as use of an `m4_` macro, using `~out(m4_⋯)` will do the trick.

The above example is interpreted as:

```
m5_if(m5_A > m5_B, m5__block(['
m5_out(['Yes, '])
m5_out_stmt(m5_A)
m5_out([' > '])
m5_out_stmt(m5_B)
'])
```

Top-level M5 content (in TL-Verilog, the content of an \m5 region) is formatted as a non-scoped code block with no output.

3.4. Text Blocks

"Text blocks" provide a syntax for multi-line quoted text that is indented with its surroundings. They are formatted similarly to code blocks, but use standard ([▯/▯]) quotes. The openning quote must be followed by a new line and the closing quote must begin a new line that is indented consistently with the line beginning the block. Their indentation is defined by the first non-blank line in the block. All lines must contain at least this indentation (except the last). This fixed level of indentation and the beginning and ending new line are removed. Aside from the removal of this whitespace, the text block is simply quoted text containing new lines.

Non-evaluating (no "*") text blocks are leaf-level blocks, meaning, there is no parsing for code and text blocks as well as label syntaxes within non-evaluating text blocks. There is parsing of vanishing comments, quotes, and parentheses (counting) and quotes are recognized (and, of course, number parameter substitutions will occur for a text block that is elaborated as part of a macro body).

3.5. Evaluating-Blocks

It can be convenient to form non-body arguments by evaluating code. Syntactic sugar is provided for this in the form of a * preceding the block open quote.

For example, here an evaluating scoped code block is used to form an error message by searching for negative arguments:

```
error(*{
   ~(['Arguments includes negative values: '])
   var(Comma, [''])
   ~for(Value, ['$@'], [
      if(m5_Value < 0, [
         ~Comma
         set(Comma, [', '])
         ~Value
      ])
   ])
   ~(['.'])
})
```

3.6. Block Labels: Escaping Blocks, and Labeled Numbered Parameters

Proper use of quotes can get a bit tedious, especially when it is necessary to escape out of several levels of nested quotes. Though rarely needed, in can improve maintainability, code clarity, and performance to make judicious use of block labels.

Blocks can be labeled using syntax such as:

```
fn(some_function, ..., <sf>{
})
```

Labels can be used in two ways.

- First, to escape out of a block, typically to generate text of the block.
- Second, to specify the block associated with a numbered parameter.

Both use cases are illustrated in the following example that attempts to declare a function for parsing text. This function declares a helper function `ParseError` for reporting parse errors that can be used many times by `my_parser`.

```
fn(my_parser,
   Text: Text to parse,
   What: A description identifying what is begin parsed,
{
```

```
macro(ParseError, {
   error(['Parsing of ']m5_What[' failed with: "$1"'])
})
```

```
   ...
})
```

This code contains, potentially, two mistakes in the error message. First, `m5_What` will be substituted at the time of the call to `ParseError`. As long as `my_parser` does not modify the value of `What`, this is fine, but it might be preferred to expand `m5_What` in the definition itself to avoid this potential masking issue in case `What` is reused.

Secondly, `$1` will be substituted upon calling `my_parser`, not upon calling `ParseError`, and it will be substituted with a null string.

The corrected example is:

```
fn(my_parser,
   Text: Text to parse,
   What: A description identifying what is begin parsed,
{
```

```
macro(ParseError, <err>{
   error(['Parsing of ']<err>m5_What[' failed with: "$<err>1"'])
})
```

```
   ...
})
```

This code corrects both issues:

- `<err>m5_What`

4. Declaring Macros

4.1. Macro Categories

`m5*` macro definitions fall into three general categories:

- variables: These hold literal text values.
- functions: These operate on inputs to produce literal output text and side effects (e.g. macro assignments).
- traditional macros: These are quick-and-dirty M4-style macros whose resulting output text is evaluated. For the most part these are superceded by variables and functions. The primary motivation for supporting these is performance.

Variables, functions, and tranditional macros are defined with a name, such as `foo`, and called (aka instantiated, invoked, expanded, evaluated, or elaborated), with the prefix `m5_`, e.g. `m5_foo`.

Here are some sample uses:

Category	Definition	Call	Resulting Text
Variables	`m5_var(Foo, 5)`	`m5_Foo`	`5`
Traditional macros	`m5_macro(foo, ['['Arg: $1']'])`	`m5_foo(hi)`	`Arg: hi`
Functions	`m5_fn(foo, in, ['m5_out(['Arg: ']m5_in)'])`	`m5_foo(hi)`	`Arg: hi`

4.2. Substituting Dollar Parameter

All types of macros support "dollar" parameters (including "numbered" and "special" parameters) substitution (though their use discouraged for variables). Dollar parameter substitutions are made throughout the entire body string regardless of the use of quotes and adjacent text. The following notations are substituted:

- `$1`, `$2`, etc.: These substitute with corresponding arguments.
- `$#`: The number of arguments (including only those that are numbered). Note that `m5_foo()` has one empty macro argument, while `m5_foo` has zero.
- `$@`: This substitutes with a comma delimited list of the arguments, each quoted so as to be taken literally. So, `m5_macro(foo, ['m5_bar($@)'])` is one way to give `m5_foo(⋯)` the same behavior as

`m5_bar(…)`.

- `$*`: This is rarely useful. It is similar to `$@`, but arguments are unquoted.
- `$0`: `$0__` can be used as a name prefix to localize a macro name to this macro. (See Masking and `$0`.) In traditional macros, `$0` is the name of the macro itself, and it can be used for recursive calls (though see `m5_recurse`). For functions, `$0` is the name of the function body and it should not be used for recursion.

CAUTION	Macros may be declared by other macros in which case the inner macro body appears within the outer macro body. Numbered parameters appearing in the inner body would be substituted as parameters of the outer body. It is generally not recommended to use numbered parameters for arguments of nested macros, though it is possible. For more on the topic, see [Escaping Blocks].

4.3. Variables

Variables are expected to be defined without parameters and to be invoked without a parameter list. They simply map a name to a literal text string.

Variables are defined using: `m5_var`, `m5_set`, `m5_var_str`, `m5_set_str`

Parameters: Though variables are not intended to be used with parameters, numbered/special (`$`) parameters are supported. Since variables result in literal (quoted) text, these parameters can only go so far as to expand arguments literally in the resulting text. Where it may be necessary to avoid inadvertent interpretation of a `$` in a variable value as a parameter reference, access the value of the variable using `m5_value_of`.

4.4. Traditional Macros

A traditional macro call returns the body of the macro definition with numbered parameters substituted with the corresponding arguments. The body is then evaluated (unlike variables), so these macros can perform computations, assign variables, etc. For example:

```
m5_macro(foo,
   ['['Args:$1,$2']'])
```

```
m5_foo(A,B)     ==> Yields: "Args:A,B"
```

Traditional macros are declared using `m5_macro`.

4.5. Functions

Functions are macros that support a richer set of mechanisms for defining and passing parameter. Functions have a body that is generally defined as a [code_block]… Functions are macros that look and act like functions/procedures/subroutines/methods in a traditional programming language,

especially when used with Code Blocks. Function calls pass arguments into parameters. Function bodies contain macro calls that define local variables, perform calculations, evaluate code conditionally, iterate in loops, call other functions, recurse, etc. They may contain comments and whitespace, and these have no impact. They evaluate to literal text that is explicitly returned using `m5_out(…)` and related macros.

There is no mechanism to explicitly print to the standard output stream, as is typical in a programming language (though there are macros for printing to the standard error stream). It is up to the caller what to do with the result. Only a top-level call from the source code will implicitly echo to standard output.

Functions are defined using: `m5_fn`, `m5_eval_fn`, `m5_null_fn`, `m5_lazy_fn`, …

Declarations take the form:

```
m5_fn(<name>, [<param-list>,] ['<body>'])
```

A basic function declaration looks like:

```
m5_fn(mul, val1, val2, ['m5_calc(m5_val1 * m5_val2)'])
```

And is called like:

```
m5_mul(3, 5)  // produces 15
```

4.5.1. Parameters

Several parameter types are provided.

4.5.1.1. Numbered Versus Named Parameters

- **Numbered parameters**: Numbered parameters are the macro parameters supported natively by M4, such as (`$1`, `$2`, etc.). `$@`, `$*`, and `$#` are also supported in the body. Unlike macros, they are substituted before elaborating the body regardless of whether they are contained within quotes or parentheses. For functions, numbered parameters are explicit in the parameter list.
- **Named parameters**: These are available to the body as macros. If from an argument, they return the quoted argument. m5_<name> is pushed prior to evaluation of the body and popped afterward.

4.5.1.2. The Parameter List

The parameter list (`<param-list>`) is a list of `<param-spec>`, where `<param-spec>` is:

- A parameter spec of the form: `[?][[<number>]][[^]<name>][: <comment>]` (in this order), e.g. `?[2]^name: the name of something`:
  - `<name>`: A named parameter.

    - `?`: An optional parameter. Calls are checked to ensure that arguments are provided for all non-optional parameters or are defined for inherited parameters. (Note that m5_foo() has one empty arg.) Non-optional parameters may not follow optional ones.
    - `[<number>]`: A numbered parameter. The first would be `[1]` and would correspond to `$1`, and so on. `<number>` is verified to match the sequential ordering of numbered parameters.
    - `^`: An inherited named parameter. Its definition is inherited from the context of the func definition. If undefined, the empty `['']` value is provided and an error is reported unless the parameter is optional, e.g. `?^<name>`. There is no corresponding argument in a call of this function. It is conventional to list inherited parameters last (before the body) to maintain correspondence between the parameter list of the definition and the argument list of a call.
    - `<comment>`: A description of the parameter. In addition to commenting the code, this can be extracted in documentation. See `m5_enable_doc`.
- `⋯`: Listed after last numbered parameter to allow extra numbered arguments. Without this, extra arguments result in an error. (Note that `m5_foo()` has one empty argument, and this is permitted for a function with no named parameters.)
- `['']`: Empty elements in the parameter list are ignored and do not correspond to any arguments (as a convenience for empty list expansion).

In addition to accessing the list of numbered arguments using `$@`, it can also be accessed as `m5_fn_args`. `m5_func_arg(3)` can be used to access the third argument from `m5_fn_args`, and `m5_fn_arg_cnt` returns the number of numbered arguments.

4.5.2. Function Call Arguments

Function calls will have arguments for all parameters that are not inherited (`^`). Arguments are positional, so misaligning arguments is a common source of errors. There is checking, however, that required arguments are provided and that no extra arguments are given.

4.5.3. When To Use What Type of Parameter

For nested declarations, named parameters are preferred. Nested declarations are declarations within the bodies of other declarations. The use of numbered parameters (`$1`, `$2`, and ...) as well as `$@`, `$*`, and `$#` can be extremely awkward in this case. Unless care is taken, they would substitute based on the outer definition, not the inner ones. Though this can be prevented by generating the body with macros that produce the numbered parameter references, this requires unnatural and bug prone use of quotes. So the use of functions with named parameters is preferred for inner macro declarations. Use of `m5_fn_args` and `m4_func_arg` is also possible with numbered parameters, though for nested functions this is suggested only to access `⋯` arguments or to pass the arguments to other functions.

Additionally, and in summary:

- **Numbered parameters**: These can be convenient to ensure substitution throughout the body without interference from quotes. They can, however, be extremely awkward to use in functions defined within the bodies of other functions/macros as they would substitute with the arguments of the outer function/macro, not the inner one. Being unnamed, readability is an issue, especially for large functions.

- **Named parameters**: These act more like typical function arguments vs. text substitution. Since they are named, they can improve readability. Unlike numbered parameters, they work perfectly well in functions defined within other functions/macros. (Similarly, m5_fn_args and m5_func_arg are useful for nested declarations.) Macros will not evaluate within quoted strings, so typical use requires unquoting, e.g. `['Arg1: ']m5_arg1['.']` vs. `['Arg1: $1.']`.
- **Inherited parameters**: These provide a more natural, readable, and explicit mechanism for customizing a function to the context in which it is defined. For example a function may define another function that is customized to the parameters of the outer function.

4.5.4. Function Arguments Example

In the context of a code block, function `foo` is declared to output its prameters.

```
// Context:
var(Inherit2, two)
// Define foo:
fn(foo, Param1, ?[1]Param2: an optional parameter, ?^Inherit1, [2]^Inherit2, ..., {
  ~nl(Param1: m5_Param1)
  ~nl(Param2: m5_Param2)
  ~nl(Inherit1: m5_Inherit1)
  ~nl(Inherit2: m5_Inherit2)
  ~nl(['numbered args: $@'])
})
```

And it can be called (again, in this example, from a code block):

```
// Call foo:
foo(arg1, arg2, extra1, extra2)
```

And this expands to:

```
Param1: arg1
Param2: arg2
Inherit1:
Inherit2: two
numbered args: ['arg2'],['two'],['extra1'],['extra2']
```

4.5.5. Aftermath

It is possible for a function to make assignments (and, actually do anything) in the calling scope. This can be done using [m5_out_eval], [m5_on_return], or [m5_return_status].

This is important for:

- passing arguments by reference
- returning status

- evaluating body arguments
- tail recursion

Each of these is discussed in its own section, next.

4.5.6. Passing Arguments by Reference

Functions can pass variables by reference and make assignments to the referenced variables. The parameter would be a named parameter, say `FooRef`, passed the name of the referenced variable. A function can modify a variable using a parameter, say `FooRef`, and calling in its code block `on_return(set, m5_FooRef, ['updated value'])`. Similarly, a function can declare a variable using a parameter, again say `FooRef`, and calling in its code block `on_return(var, m5_FooRef, ['init value'])`.

The use of `on_return` avoids a potential masking issue resulting from a local variable of the function having a conflicting name with the referenced variable.

4.5.7. Returning Status

TODO...

4.5.8. Functions (and Tranditional Macros) with Body Arguments

The example below illustrates a function `m5_if_neg` that takes an argument that is a body to evaluate. The body is defined in a calling function, `m5_my_fn` on lines 12-15. Such a body is expected to evaluate in the context of the calling function, `m5_my_fn`. Its side effects from `on_return` in line 13 should be side effects of `m5_my_fn`. If the body is evaluated inside the function body, its side effects would be side effects of `m5_if_neg`, not `m5_my_fn`, as expected. This can be addressed using `m5_on_return`.

Note that `m5_return_status` is called after evaluating `m5_Body`. Both `m5_on_return` and `m5_return_status` add to the "aftermath" of the function, and `m5_status` must be set after evaluating the body (which could affect `m5_status`.

Masking...

TODO... Note that my_fn could contain multiple nested m5_if_neg calls, and each would pass the side effect along, ultimately producing the side effect in m5_my_fn. Also note the distinction between body output and function side effects in that body output is associated with bodies, and function side effects are associated with functions. In order for body output to propogate to its calling function, each nesting level explicitly passes the output along using ~. Propogation is the responsibility of the caller, not the callee.

Example of a body argument.

```
// Evaluate a body if a value is negative.
fn(if_neg, Value, Body, {
   var(Neg, m5_calc(Value < 0))
   if(Neg, [

     ~on_return(Body)
   ])
   ~return_status(if(Neg, [''], else))
})

fn(my_fn, {
   ~if_neg(1, [
      on_return(...)
      ~(...)
   ])
})
```

Traditional macros defined using a scoped code block have a similar issue resolved by using `~out_eval`. TODO: explain and find the right home for this.

4.5.9. Tail Recursion

...

5. Contexts: Scope, Namespaces, and Libraries

5.1. Contexts

The context of a macro comes in three types:

- Universal: Universal macro names are the same for any M5 program. These can be called directly, prefixed with `m5_`. They can be:
- Built-in: These are defined by the M5 library.
- External: These are only defined if explicitly included.
- Namespaced: Namespaces are used to avoid name conflicts between third-party libraries and between different versions of the same library. Namespaces are local to a library or application, and may exist in a hierarchy. The same macro may exist in multiple namespaces of multiple libraries, and its definition is shared. Namespaced macros are called via `m5_my(⋯)`.
- Scoped: Declarations made within a Scope are local to that scope. Naming conventions avoid name conflicts with the other context types.

5.2. Macro Naming Conventions

To avoid masking issues, naming conventions divide the namespace in two styles:

- Lower case with underscores, e.g.: `m5_builtin_macro`
- Pascal case, e.g. `m5_MyVarName`

Names using lowercase with underscores: universal, namespaces, namespaced

Names using Pascal case: scoped macros (variables, functions, and traditional macros)

In both cases, names must be composed of ASCII characters `A-Z`, `a-z`, `0-9`, and `_`, and the first character must be alphabetic.

Libraries may define private macros using double underscore (`__`). A non-private macro in a universal library reserves its own name in the universal namespace and also private names beginning with that name and `__`. To maximize the ability of third-party libraries to share a namespace with other libraries, macros in third-party libraries that are helpers for other macros should use the name of the associated macro before the `__`.

5.3. Scope

5.3.1. Macro Stacks

All macros in M4, and thus in M5, are stacks of definitions that can be pushed and popped. (These stacks are frequently one entry deep.) The top definition of a macro provides the replacement text when the macro is called. The others are only accessible by popping the stack. In M5, pushing and popping are not generally done explicitly, but rather through scoped declarations.

5.3.2. Scoped Code Blocks

Some macros accept body arguments that may be evaluated by calls to the macro. (See Multi-line Constructs: Blocks and Bodies.) Such an argument may be given as a scoped code block. (See Code Blocks.)

Within a code block, declarations made using `m5_var`, ... are scoped. Their definitions are pushed by the declaration, and popped at the end of their scope.

`m5_set`, ... redefine the top entry.

Declarations from outer scopes are visible in inner scopes. Similarly, declarations from calling scopes are visible in callee scopes. Function are generally written without any assumptions about the calling scope and should not use definitions from them. Exceptions should be clearly documented/commented.

NOTE	It is fine to redeclare a variable in the same scope. The redeclaration will override the first, and both definitions will be popped after evaluating the code block. Notably, a variable may be conditionally declared without any negative consequence on stack maintenance.

5.4. Universal Marcros

5.5. Namespaces

5.6. Libraries

6. Status

The variable `m5_status` has a reserved usage. Some macros are defined to set `m5_status`. A non-`['']` value indicates that the macro did not perform its duties to the fullest. Several `m5_if*` macros set non-`['']` status if they do not evaluate a body.

Macros such as `m5_else` and `m5_ifso` take action based on `m5_status`.

Some macros are defined explicitly to preserve the value of `m5_status` (or restore it upon completion). These macros can be used between a status-producing macro and a status-consuming macro.

Macros whose treatment of `m5_status` is not specified may update `m5_status` in unpredictable ways. This can be source of bugs in poorly-constructed code, especially when library versions are updated.

7. Processing Order

WIP...

- Strip vanishing comments.
- Substitute block and label syntax, match quotes and parentheses.
- Produce pre-preprocessed file for M4.
- M4 macro preprocessing (substituting macros).

8. Coding Paradigms, Patterns, Tips, Tricks, and Gotchas

8.1. Arbitrary Strings

It's important to keep in mind that variables are macros, and macro calls substitute `$` parameters,

whether parameters are given or not. (This is legacy from M4, and working around it would impact performance appreciably.) Whenever dealing using variables containing arbitrary strings, use `m5_value_of`, or use `m5_str` and `m5_set_str`. See [[Working with Strings]].

8.2. Masking and `$0`

TODO: Move this to a section about macros that have body arguments.

A common pattern is to declare a variable in an outer-level macro body and assign it in a lower-level macro body. This paradigm fails if a variable by the same name happens to be declared by an intervening macro. This is referred to as "masking".

In macros that only evaluate code provided in the body of the macro itself, any masking is apparent and is unlikely to catch a developer by surprise. Masking becomes an issue when a macro evaluates arbitrary code provided as an input in a body argument.

TODO: Use `_` prefix (w/ Pascal case) instead. To avoid masking, proior to evaluating a body argument, a macro should only declare variables (and other macros) using uniquified names. Uniquified, or "local" macro names can be generated using the prefix `$0__`. In traditional macros, `$0` is the name of the macro. In functions, `$0` is the name given to the function body. In either case, this prefix constructs a name that is implicitly reserved by the macro.

9. Macros Specifications

9.1. Specification Conventions

Macros are listed by category in a logical order. An alphabetical Index of macros can be found at the end of this document (at least in the `.pdf` version). Macros that return integer values, unless otherwise specified, return decimal value strings. Similarly, macro arguments that are integer values accept decimal value strings. Boolean inputs and outputs use `0` and `1`. Behavior for other argument values is undefined if unspecified.

Resulting output text is, by default, literal (quoted). Macros named with a `_eval` suffix generally result in text that gets evaluated.

9.2. Assigning and Accessing Macros Values

9.2.1. Declaring/Setting Variables

`m5_var(Name, Value, ⋯)`

Description: Declare a scoped variable. See [variables].

Side Effect(s): the variable is defined

Parameter(s):
1. `Name`: variable name
2. `Value`(opt) : the value for the variable
3. `...`: additional variables and values to declare (values are required)

Example(s):

```
var(Foo, 5)
```

See also: `m5_macro`, `m5_fn`, `m5_var_str`

`m5_set(Name, Value)`

Description: Set the value of a scoped variable. See [variables].

Side Effect(s): the variable's value is set

Parameter(s):
1. `Name`: variable name
2. `Value`: the value

Example(s):

```
set(Foo, 5)
```

See also: `m5_var`

`m5_push_var(Name, Value)`

Description: Declare a variable that must be explicitly popped.

Side Effect(s): the variable is defined

Parameter(s):
1. `Name`: variable name
2. `Value`: the value

Example(s):

```
push_var(Foo, 5)
...
pop(Foo)
```

See also: `m5_pop`

`m5_pop(Name)`

Description: Pop a variable or traditional macro declared using `push_var` or `push_macro`.

Side Effect(s): the macro is popped

Parameter(s):
1. `Name`: variable name

Example(s):
```
push_var(Foo, 5)
...
pop(Foo)
```

See also: `m5_push_var`, `m5_push_macro`

`m5_var_str(Name, Value)`

Description: Declare a variable and assign it a "string value". A string value evaluates without `$` substitution.

Side Effect(s): the variable is defined

Parameter(s):
1. `Name`: variable name
2. `Value`(opt) : the value for the string variable (empty by default)

Example(s):
```
m5_var_str(OneDollar, ['$1.00'])
m5_OneDollar()
```

Example Output:
```
$1.00
```

See also: `m5_var`, `m5_value_of`

`m5_set_str(Name, Value)`

Description: Set a variable with a string value (so the variable will evaluate without `$` substitution).

Side Effect(s): the variable is defined

Parameter(s):
1. `Name`: variable name
2. `Value`(opt) : the value for the string variable (empty by default)

Example(s):

```
m5_var_str(OneDollar, ['$1.00'])
m5_OneDollar()
```

Example Output:

```
$1.00
```

See also: `m5_var`, `m5_value_of`

`m5_null_vars(…)`

Description: Declare variables with empty values.

Side Effect(s): the variables are declared

Parameter(s): 1. `…`: names of variables to declare

9.2.2. Declaring Functions

`m5_fn(…)`
`m5_lazy_fn(…)`

Description: Declare a function. For details, see Functions. `fn` and `lazy_fn` are functionally equivalent but have different performance profiles, and lazy functions do not support inherited (`^`) parameters. Lazy functions wait until they are used before defining themselves, so they are generally preferred in libraries except for the most commonly-used functions.

Side Effect(s): the function is declared

Parameter(s): 1. `…`: arguments and body

Example(s):

```
fn(add, Addend1, Addend2, {
   ~calc(Addend1 + Addend2)
})
```

See also: `m5_Functions`

9.2.3. Declaring/Setting Traditional Macros

`m5_macro(Name, Body)`
`m5_null_macro(Name, Body)`

Description: Declare a scoped traditional macro. See [macros]. A null macro must produce no output.

Side Effect(s): the macro is declared

Parameter(s):
1. `Name`: the macro name
2. `Body`: the body of the macro

Example(s):

```
m5_macro(ParseError, <p>[
   error(['Failed to parse $<p>1.'])
])
```

See also: `m5_var`, `m5_set_macro`

`m5_set_macro(Name, Body)`

Description: Set the value of a scoped traditional macro. See [macros]. Using this macro is rare.

Side Effect(s): the macro value is set

Parameter(s):
1. `Name`: the macro name
2. `Body`: the body of the macro

See also: `m5_var`, `m5_set_macro`

`m5_push_macro(Name, Body)`

Description: Push a new value of a traditional macro that must be explicitly popped. Using this macro is rare.

Side Effect(s): the macro value is pushed

Parameter(s):
1. `Name`: the macro name
2. `Body`: the body of the macro

See also: `m5_pop`, `m5_macro`, `m5_set_macro`

9.2.4. Accessing Macro Values

`m5_value_of(Name)`

Output: the value of a variable without `$` substitution (even if not assigned as a string)

Parameter(s): 1. `Name`: name of the variable

Example(s):

```
var(OneDollar, ['$1.00'])
value_of(OneDollar)
```

Example Output:

```
$1.00
```

See also: `m5_var_str`, `m5_set_str`

`m5_must_exist(Name)`

Description: Ensure that the `Name`d macro exists.

Parameter(s): 1. `Name`: name of the macro

9.3. Code Constructs

9.3.1. Status

`m5_status` (Universal variable)

Description: This universal variable is set as a side-effect of some macros to indicate an exceptional condition or non-evaluation of a body argument. It may be desirable to check this condition after calling such macros. Macros, like `m5_else` take action based on the value of `m5_status`. An empty value indicates no special condition. Macros either always set it (to an empty or non-empty value) or never set it. Those that set it list this in their "Side Effect(s)".

See also: `m5_fn`, `m5_return_status`, `m5_else`

9.3.2. Conditionals

`m5_if(Cond, TrueBody, …)`
`m5_unless(Cond, TrueBody, FalseBody)`
`m5_else_if(Cond, TrueBody, …)`

Description: An if/else construct. The condition is an expression that evaluates using [m5_calc] (generally boolean (0/1)). The first block is evaluated if the condition is non-0 for `if` and `else_if` or 0 for `unless`, otherwise, subsequent conditions are evaluated, or if only one argument remains, it is the final else block, and it is evaluate. (`unless` cannot have subsequent conditions.) `if_else` does nothing if `m5_status` is initially empty.

> **NOTE** As an alternative to providing else blocks within `m5_if`, [m5_else] and similar macros may be used subsequent to `m5_if` / `m5_unless` and other macros producing Status, and this may be easier to read.

Output: the output of the evaluated body

Side Effect(s): status is set, empty iff a block was evaluated; side-effects of the evaluated body

Parameter(s):

1. `Cond`: ['['['the condition for evaluation']']']']
2. `TrueBody`: ['the body to evaluate if the condition is true (1)']
3. `⋯`: ['either a `FalseBody` or (for `m5_if` only) recursive `Cond`, `TrueBody`, `⋯` arguments to evaluate if the condition is false (not 1)']

Example(s):

```
~if(m5_eq(m4_Ten, 10) && m5_Val > 3, [
   ~do_something(...)
], m5_Val > m5_Ten, [
   ~do_something_else(...)
], [
   ~default_case(...)
])
```

See also: `m5_else`, `m5_case`

`m5_if_eq(String1, String2, TrueBody, ⋯)`
`m5_if_neq(String1, String2, TrueBody, ⋯)`

Description: An if/else construct where each condition is a comparison of an independent pair of strings. The first block is evaluated if the strings match for `if` or mismatch for `if_neq`, otherwise, the remaining arguments are processed in a recursive call, either comparing the next pair of strings or, if only one argument remains, evaluating it as the final else block.

> **NOTE** As an alternative to providing else blocks, [m5_else] and similar macros may be used subsequently, and this may be easier to read.

Output: the output of the evaluated body

Side Effect(s): status is set, empty iff a body was evaluated; side-effects of the evaluated body

Parameter(s):
1. `String1`: the first string to compare
2. `String2`: the second string to compare
3. `TrueBody`: the body to evaluate if the strings match
4. `⋯`: either a `FalseBody` or recursive `String1`, `String2`, `TrueBody`, `⋯` arguments to evaluate if the strings do not match

Example(s):

```
~if_eq(m4_Zero, 0, [
   ~zero_is_zero(...)
], m5_calc(m5_Zero < 0), 1, [
   ~zero_is_negative(...)
], [
   ~zero_is_positive(...)
])
```

See also: `m5_else`, `m5_case`

`m5_if_null(Var, Body, ElseBody)`
`m5_if_def(Var, Body, ElseBody)`
`m5_if_ndef(Var, Body, ElseBody)`
`m5_if_defined_as(Var, Value, Body, ElseBody)`

Description: Evaluate `Body` if the named variable is empty (`if_null`), defined (`if_def`), not defined (`if_ndef`), or not defined and equal to the given value (`if_defined_as`)., or `ElseBody` otherwise.

Output: the output of the evaluated body

Side Effect(s): status is set, empty iff a body was evaluated; side-effects of the evaluated body

Parameter(s):
1. `Var`: the variable's name
2. `Value`: for `if_defined_as` only, the value to compare against
3. `Body`: the body to evaluate based on `m5_Name's existence or definition
4. `ElseBody`(opt) : a body to evaluate if the condition if `Body` is not evaluated

Example(s):

```
if_null(Tag, [
   error(No tag.)
])
```

See also: m5_if

m5_else(Body)
m5_if_so(Body)

Description: Likely following a macro that sets m5_status, this evaluates a body if Status is non-empty (for else) or empty (for if_so).

Output: the output of the evaluated body

Side Effect(s): status is set, empty iff a body was evaluated; side-effects of the evaluated body

Parameter(s):

1. Body: the body to evaluate based on Status

Example(s):

```
~if(m5_Cnt > 0, [
   decrement(Cnt)
])
else([
   ~(Done)
])
```

See also: m5_if, m5_if_eq, m5_if_neq, m5_if_null, m5_if_def, m5_if_ndef, m5_var_regex

m5_else_if_def(Name, Body)

Description: Evaluate Body iff the `Name`d variable is defined.

Output: the output of the evaluated body

Side Effect(s): status is set, empty iff a body was evaluated; side-effects of the evaluated body

Parameter(s):

1. Name: the name of the case variable whose value to compare against all cases
2. Body: the body to evaluate based on Status

Example(s):

```
m5_set(Either, if_def(First, m5_First)m5_else_if_def(Second,
m5_Second))
```

See also: m5_else_if, m5_if_def

`m5_case(Name, Value, TrueBody, ⋯)`

Description: Similar to [m5_if], but each condition is a string comparison against a value in the Name variable.

Output: the output of the evaluated body

Side Effect(s): status is set, empty iff a block was evaluated; side-effects of the evaluated body

Parameter(s):

1. Name: the name of the case variable whose value to compare against all cases
2. Value: the first string value to compare VarName against
3. TrueBody: the body to evaluate if the strings match
4. ⋯: either a FalseBody or recursive Value, TrueBody, ⋯ arguments to evaluate if the strings do not match

Example(s):

```
~case(m5_Response, ok, [
   ~ok_response(...)
], bad, [
   ~bad_response(...)
], [
   error(Unrecognized response: m5_Response)
])
```

See also: m5_else, m5_case

9.3.3. Loops

`m5_loop(InitList, DoBody, WhileCond, WhileBody)`

Description: A generalized loop construct. Implicit variable m5_LoopCnt starts at 0 and increments by 1 with each iteration (after both blocks).

Output: output of the blocks

Side Effect(s): side-effects of the blocks

Parameter(s):

1. `InitList`: a parenthesized list, e.g. `(Foo, 5, Bar, ok)` of at least one variabl, initial-value pair providing variables scoped to the loop, or `['']`
2. `DoBody`: a block to evaluate before evaluating `WhileCond`
3. `WhileCond`: an expression (evaluated with [m5_calc]) that determines whether to continue the loop
4. `WhileBody`(opt) : a block to evaluate if `WhileCond` evaluates to true (1)

Example(s):

```
~loop((MyVar, 0), [
   ~do_stuff(...)
], m5_LoopCnt < 10, [
   ~do_more_stuff(...)
])
```

See also: `m5_repeat`, `m5_for`, `m5_calc`

`m5_repeat(Cnt, Body)`

Description: Evaluate a block a predetermined number of times. Implicit variable `m5_LoopCnt` starts at 0 and increments by 1 with each iteration.

Output: output of the block

Side Effect(s): side-effects of the block

Parameter(s):

1. `Cnt`: the number of times to evaluate the body
2. `Body`: a block to evaluate `Cnt` times

Example(s):

```
~repeat(10, [
   ~do_stuff(...)
])  // Iterates m5_LoopCnt 0..9.
```

See also: `m5_loop`

`m5_for(Var, List, Body)`

Description: Evaluate a block for each item in a listed. Implicit variable `m5_LoopCnt` starts at 0 and increments by 1 with each iteration.

Output: output of the block

Side Effect(s): side-effects of the block

Parameter(s):
1. `Var`: the loop item variable
2. `List`: a list of items to iterate over, the last of which will be skipped if empty; for each item, `Var` is set to the item, and `Body` is evaluated
3. `Body`: a block to evaluate for each item

Example(s):

```
~for(fruit, ['apple, orange, '], [
   ~do_stuff(...)
])  // (also maintains m5_LoopCnt)
```

See also: `m5_loop`

9.3.4. Recursion

`m5_recurse(max_depth, macro, ⋯)`

Description: Call a macro recursively to a given maximum recursion depth. Functions have a built-in recursion limit, so this is only useful for macros.

Output: the output of the recursive call

Side Effect(s): the side effects of the recursive call

Parameter(s):
1. `max_depth`: the limit on the depth of recursive calls made through this macro
2. `macro`: the recursive macro to call
3. `⋯`: arguments for `macro`

Example(s):

```
m5_recurse(20, myself, args)
```

See also: `m5_recursion_limit`, `m5_on_return`

9.4. Working with Strings

9.4.1. Special Characters

`m5_nl()`

Description: Produce a new-line. Programmatically-generated output should always use this macro (directly or indirectly) to produce new-lines, rather than using an actual new-line in the source file. Thus the input file formatting can reflect the code structure, not the output formatting.

Output: a new-line

`m5_open_quote()`
`m5_close_quote()`

Description: Produce an open or close quote. These should rarely (never?) be needed and should be used with extra caution since they can create undetected imbalanced quoting. The resulting quote is literal, but it will be interpreted as a quote if evaluated.

Output: the literal quote

See also: `m5_quote`

`m5_orig_open_quote()`
`m5_orig_close_quote()`

Description: Produce `[' or `']`. These quotes in the original file are translated internally to ASCII control characters, and in output (STDOUT and STDERR) these control characters are translated to single-unicode-character "printable quotes". This original quote syntax is most easily produced using these macros, and once produced, has no special meaning in strings (though `[` and `]` have special meaning in regular expressions).

Output: the literal quote

See also: `m5_printable_open_quote`, `m5_printable_close_quote`

`m5_printable_open_quote()`
`m5_printable_close_quote()`

Description: Produce the single unicode character used to represent `[' or `']` in output (STDOUT and STDERR).

Output: the printable quote

See also: `m5_orig_open_quote`, `m5_orig_close_quote`

`m5_UNDEFINED()`

Description: A unique untypeable value indicating that no assignment has been made. This is not used by any standard macro, but is available for explicit use.

Output: the value indicating "undefined"

Example(s):

```
m5_var(Foo, m5_UNDEFINED)
m5_if_eq(Foo, m5_UNDEFINED, ['['Foo is undefined.']'])
R: Foo is undefined.
```

9.4.2. Slicing and Dicing Strings

`m5_append_var(Name, String)`
`m5_prepend_var(Name, String)`
`m5_append_macro(Name, String)`
`m5_prepend_macro(Name, String)`

Description: Append or prepend to a variable or macro. (A macro evaluates its context; a variable does not.)

Parameter(s):

1. `Name`: the variable name
2. `String`: the string to append/prepend

Example(s):

```
m5_var(Hi, ['Hello'])
m5_append_var([', ']m5_Name['!'])
m5_Hi
```

Example Output:

```
Hello, Joe!
```

`m5_substr(String, From, Length)`
`m5_substr_eval(String, From, Length)`

Description: Extract a substring from `String` starting from `Index` and extending for `Length` characters or to the end of the string if `Length` is omitted or exceeds the string length. The first character of the string has index 0. The result is empty if there is an error parsing `From` or `Length`, if `From` is beyond the end of the string, or if `Length` is negative.

Extracting substrings from strings with quotes is dangerous as it can lead to imbalanced quoting. If the resulting string would contain any quotes, an error is reported suggesting the use of `dequote` and `requote` and the resulting string has its quotes replaced by control characters.

Extracting substrings from UTF-8 strings (supporting unicode characters) is also dangerous. M5 treats characters as bytes and UTF-8 characters can use multiple bytes, so substrings can split UTF-8 characters. Such split UTF-8 characters will result in bytes/M5-characters that have no special treatment in M5. They can be rejoined to reform valid UTF-8 strings.

When evaluating substrings, care must be taken with `,`, `(`, and `)` because of their meaning in argument parsing.

`substr` is a slow operation relative to `substr_eval` (due to limitations of M4).

Output: the substring or its evaluation

Parameter(s):
1. `String`: the string
2. `From`: the starting position of the substring
3. `Length`(opt) : the length of the substring

Example(s):
```
m5_substr(['Hello World!'], 3, 5)
```

Example Output:
```
lo Wo
```

See also: `m5_dequote`, `m5_requote`

`m5_join(Delimiter, ⋯)`

Output: the arguments, delimited by the given delimiter string

Parameter(s):
1. `Delimiter`: text to delimit arguments
2. `⋯`: arguments to concatenate (with delimitation)

Example(s):

```
m5_join([', '], ['new-line'], ['m5_nl'], ['macro'])
```

Example Output:

```
new-line, m5_nl, macro
```

`m5_translit(String, InChars, OutChars)`
`m5_translit_eval(String, InChars, OutChars)`

Description: Transliterate a string, providing a set of character-for-character substitutions (where a character is a unicode byte). `translit_eval` evaluates the resulting string. Note that `[` and `]` are internally single characters. It is possible to substitute these quotes (if balanced in the string and in the result) using `translit_eval` but not using `translit`.

Output: the transliterated string (or its evaluation for `translit_eval`)

Side Effect(s): for `translit_eval` the side-effects of the evaluation

Parameter(s):

1. `String`: the string to tranliterate
2. `InChars`: the input characters to replace
3. `OutChars`: the corresponding character replacements

Example(s):

```
m5_translit(['Testing: 1, 2, 3.'], ['123'], ['ABC'])
```

Example Output:

```
Testing: A, B, C.
```

`m5_uppercase(String)`
`m5_lowercase(String)`

Description: Convert upper-case ASCII characters to lower-case.

Output: the converted string

Parameter(s):

1. `String`: the string

Example(s):

```
m5_uppercase(['Hello!'])
```

Example Output:

```
HELLO!
```

`m5_replicate(Cnt, String)`

Description: Replicate a string the given number of times. (A non-evaluating version of `m5_repeat`.)

Output: the replicated string

Parameter(s):

1. `Cnt`: the number of repetitions
2. `String`: the string to repeat

Example(s):

```
m5_replicate(3, ['.'])
```

Example Output:

```
...
```

See also: `m5_repeat`

`m5_strip_trailing_whitespace_from(Var)`

Description: Strip trailing whitespace from the given variable.

Side Effect(s): the variable is updated

Parameter(s):

1. `Var`: the variable

9.4.3. Formatting Strings

`m5_format_eval(string, ⋯)`

Description: Produce formatted output, much like the C `printf` function. The `string` argument may contain `%` specifications that format values from ⋯ arguments.

From the M4 Manual, `%` specifiers include `c`, `s`, `d`, `o`, `x`, `X`, `u`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, `G`, and `%`. The following are also supported:

- field widths and precisions
- flags `+`, `-`, `` ` ``, `` `0 ``, `#`, and `'`
- for integer specifiers, the width modifiers `hh`, `h`, and `l`
- for floating point specifiers, the width modifier `l`

Items not supported include positional arguments, the `n`, `p`, `S`, and `C` specifiers, the `z`, `t`, `j`, `L` and `ll` modifiers, escape sequences, and any platform extensions available in the native printf (for example, `%a` is supported even on platforms that haven't yet implemented C99 hexadecimal floating point output natively).

For more details on the functioning of `printf`, see the C Library Manual, or the POSIX specification.

Output: the formatted string

Parameter(s):

1. `string`: the string to format
2. ⋯: values to format, one for each `%` sequence in `string`

Example(s):

```
1: m5_var(Foo, Hello)
   m5_format_eval(`String "%s" uses %d chars.', Foo, m5_length(Foo))
2: m5_format_eval(`%*.*d', `-1', `-1', `1')
3: m5_format_eval(`%.0f', `56789.9876')
4: m5_length(m5_format(`%-*X', `5000', `1'))
5: m5_format_eval(`%010F', `infinity')
6: m5_format_eval(`%.1A', `1.999')
7: m5_format_eval(`%g', `0xa.P+1')
```

Example Output:

```
1:
   String "Hello" uses 5 chars.
2: 1
3: 56790
4: 5000
5:        INF
6: 0X2.0P+0
7: 20
```

9.4.4. Inspecting Strings

`m5_length(String)`

Output: the length of a string in ASCII characters (unicode bytes)

Parameter(s):
1. `String`: the string

`m5_index_of(String, Substring)`

Output: the position in a string in ASCII characters (unicode bytes) of the first occurence of a given substring or -1 if not present, where the string starts with character zero

Parameter(s):
1. `String`: the string
2. `Substring`: the substring to find

`m5_num_lines(String)`

Output: the number of new-lines in the given string

Parameter(s):
1. `String`: the string

`m5_for_each_line(Text, Body)`

Description: Evaluate `m5_Body` for every line of `m5_Text`, with `m5_Line` assigned to the line (without any new-lines).

Output: output from `m5_Body`

Side Effect(s): side-effects of `m5_Body`

Parameter(s):
1. `Text`: the block of text
2. `Body`: the body to evaluate for every `m5_if` of `m5_Text`

9.4.5. Safely Working with Strings

`m5_dequote(String)`
`m5_requote(String)`

Description: For strings that may contain quotes, working with substrings can lead to imbalanced quotes and unpredictable behavior. `dequote` replaces quotes for (different) control-character/byte quotes, aka "surrogate-quotes" that have no special meaning. Dequoted strings can be safely sliced and diced, and once reconstructed into strings containing balanced (surrogate) quotes, dequoted strings can be requoted using `requote`.

Output: dequoted or requoted string

Parameter(s): 1. `String`: the string to dequote or requote

`m5_output_with_restored_quotes(String)`

Output: the given string with quotes, surrogate quotes and printable quotes replaced by their original format (['])

Parameter(s): 1. `String`: the string to output

See also: `m5_printable_open_quote`, `m5_printable_close_quote`

`m5_no_quotes(String)`

Description: Assert that the given string contains no quotes.

Parameter(s): 1. `String`: the string to test

9.4.6. Regular Expressions

Regular expressions in M5 use the same regular expression syntax as GNU Emacs. (See [GNU Emacs Regular Expressions](https://www.gnu.org/software/emacs/manual/html_node/emacs/Regexps.html).) This syntax is similar to BRE, Basic Regular Expressions in POSIX and is regrettably rather limited. Extended Regular Expressions are not supported.

`m5_regex(String, Regex, Replacement)`
`m5_regex_eval(String, Regex, Replacement)`

Description: Searches for `Regexp` in `String`, resulting in either the position of the match or the given replacement.

`Replacement` provides the output text. It may contain references to subexpressions of `Regex` to expand in the output. In `Replacement`, `\n` references the nth parenthesized subexpression of `Regexp`, up to nine subexpressions, while `\&` refers to the text of the entire regular expression matched. For all other characters, a preceding `\` treats the character literally.

Output: If `Replacement` is omitted, the index of the first match of `Regexp` in `String` is produced (where the first character in the string has an index of 0), or -1 is produced if there is no match.

If `Replacement` is given and there was a match, this argument provides the output, with `\n` replaced by the corresponding matched subexpressions of `Regex` and `\&` replaced by the entire matched substring. If there was no match result is empty.

The resulting text is literal for `regex` and is evaluated for `regex_eval`.

Side Effect(s): `regex_eval` may result in side-effects resulting from the evaluation of `Replacement`.

Parameter(s):

1. `String`: the string to search
2. `Regex`: the regular expression to match
3. `Replacement`(opt) : the replacement

Example(s):

```
m5_regex_eval(['Hello there'], ['\w+'], ['First word:
m5_translit(['\&']).'])
```

Example Output:

```
First word: Hello.
```

See also: `m5_var_regex`, `m5_if_regex`, `m5_foreach_regex`

`m5_var_regex(String, Regex, VarList)`

Description: Declare variables assigned to subexpressions of a regular expression.

Side Effect(s): `status` is assigned, non-empty iff no match.

Parameter(s):

1. `String`: the string to match
2. `Regex`: the Gnu Emacs regular expression
3. `VarList`: a list in parentheses of variables to declare for subexpressions

Example(s):

```
m5_var_regex(['mul A, B'], ['^\(\w+\)\s+\(w+\),\s*\(w+\)$'],
(Operation, Src1, Src2))
m5_if_so(['m5_DEBUG(Matched: m5_Src1[','] m5_Src2)'])
m5_else(['m5_error(['Match failed.'])'])
```

See also: `m5_regex`, `m5_regex_eval`, `m5_if_regex`, `m5_foreach_regex`

`m5_if_regex(String, Regex, VarList, Body, ⋯)`
`m5_else_if_regex(String, Regex, VarList, Body, ⋯)`

Description: For chaining `var_regex` to parse text that could match a number of formats. Each pattern match is in its own scope. `else_if_regex` does nothing if `m5_status` is non-empty.

Output: output of the matching body

Side Effect(s): `m5_status` is non-null if no expression matched; side-effects of the bodies

Parameter(s):

1. `String`: the string to match
2. `Regex`: the Gnu Emacs regular expression
3. `VarList`: a list in parentheses of variables to declare for subexpressions
4. `Body`: the body to evaluate if the pattern matches
5. `⋯`: additional repeated Regex, VarList, Body, ... to process if pattern doesn't match

Example(s):

```
~if_regex(m5_Instruction, ['^mul\s+\(w+\),\s*\(w+\)$'], (Src1, Src2),
[
   ~m5_calc(m5_Src1 * m5_Src2)
], ['^incr\s+\(w+\)$'], (Src1), [
   ~m5_calc(m5_Src1 + 1)
])
```

See also: `m5_var_regex`

`m5_for_each_regex(String, Regex, VarList, Body)`

Description: Evaluate body for every pattern matching regex in the string. m5_status is unassigned.

Side Effect(s): side-effects of the body

Parameter(s):

1. `String`: the string to match (containing at least one subexpression and no `$`)
2. `Regex`: the Gnu Emacs regular expression
3. `VarList`: a (non-empty) list in parentheses of variables to declare for subexpressions
4. `Body`: the body to evaluate for each matching expression

Example(s):

```
m5_for_each_regex(H1dd3n D1git5, ['\([0-9]\)'], (Digit), ['Found
m5_Digit. '])
```

Example Output:

```
Found 1. Found 3. Found 1. Found 5.
```

See also: m5_regex, m5_regex_eval, m5_if_regex, m5_else_if_regex

9.5. Utilities

9.5.1. Fundamental Macros

m5_defn(Name)

Output: the definition of a macro

Parameter(s):
1. Name: the name of the macro

m5_call(Name, ⋯)

Description: Call a macro. Versus directly calling a ` this indirect mechanism has two primary uses. First it provides a consistent syntax for calls with zero arguments as for calls with a non-zero number of arguments. Second, the macro name can be constructed conveniently.

Output: the output of the called macro

Side Effect(s): the side-effects of the called macro

Parameter(s):
1. Name: the name of the macro to call
2. ⋯: the arguments of the macro to call

Example(s):

```
m5_call(error, ['Fail!'])
```

See also: m5_comma_shift, m5_comma_args, m5_call_varargs

m5_quote(⋯)

Output: a comma-separated list of quoted arguments, i.e. $@

Parameter(s): 1. `⋯`: arguments to be quoted

Example(s):

```
m5_quote(A, ['B'])
```

Example Output:

```
['A'],['B']
```

See also: `m5_nquote`

`m5_nquote(⋯)`

Output: the arguments within the given number of quotes, the innermost applying individually to each argument, separated by commas. A `num` of `0` results in the inlining of `$@`.

Parameter(s): 1. `⋯`:

Example(s):

```
1: m5_nquote(3, A, ['m5_nl'])
2: m5_nquote(3, m5_nquote(0, A, ['m5_nl'])xx)
```

Example Output:

```
1: ['['['A'],['m5_nl']']']
2: ['['['A'],['m5_nlxx']']']
```

See also: `m5_quote`

`m5_eval(Expr)`

Description: Evaluate the argument.

Output: the result of evaluating the argument

Side Effect(s): the side-effects resulting from evaluation

Parameter(s): 1. `Expr`: the expression to evaluate

Example(s):

```
1: m5_eval(['m5_calc(1 + 1)'])
2: m5_eval(['m5_'])calc(1 + 1)
```

Example Output:

```
1: 2
2: m5_calc(1 + 1)
```

`m5_comment(…)`
`m5_nullify(…)`

Output: nothing at all; used to provide a comment (though [comments] are preferred) or to discard the result of an evaluation

Parameter(s):
1. `…`:

9.5.2. Manipulating Macro Stacks

See [stacks].

`m5_defn_ago(Name, Ago)`
`m5_value_ago(Name, Ago)`

Output:

Parameter(s):
1. `Name`: macro name
2. `Ago`: 0 for current definition, 1 for previous, and so on

Example(s):

```
*{
   var(Foo, A)
   var(Foo, B)
   ~defn_ago(Foo, 1)
   ~value_ago(Foo, 0)
}
```

Example Output:

```
['A']
B
```

`m5_depth_of(Name)`

Output: the number of definitions in a macro's stack

Parameter(s):
1. `Name`: macro name

Example(s):

```
m5_depth_of(Foo)
m5_push_var(Foo, A)
m5_depth_of(Foo)
```

Example Output:

```
0

1
```

9.5.3. Argument Processing

`m5_shift(…)`
`m5_comma_shift(…)`

Description: Removes the first argument. `comma_shift` includes a leading `,` if there are more than zero arguments.

Output: a list of remaining arguments, or `['']` if less than two arguments

Side Effect(s): none

Parameter(s): 1. `…`: arguments to shift

Example(s):

```
m5_foo(m5_shift())         ///  has at least 2 arguments
m5_call(foo['']m5_comma_shift()) ///  has at least 1 argument
```

`m5_nargs(…)`

Output: the number of arguments given (useful for variables that contain lists)

Parameter(s): 1. `…`: arguments

Example(s):

```
m5_set(ExampleList, ['hi, there'])
m5_nargs(m5_ExampleList)
```

Example Output:

```
2
```

`m5_argn(ArgNum, …)`

Output: the nth of the given `arguments` or `['']` for non-existent arguments

Parameter(s):
1. `ArgNum`: the argument number (n) (must be positive)
2. `…`: arguments

Example(s):

```
m5_set(ExampleList, ['hi, there'])
m5_argn(2, ExampleList)
```

Example Output:

```
there
```

`m5_comma_args(…)`

Description: Convert a quoted argument list to a list of arguments with a preceding comma. This is necessary to properly work with argument lists that may contain zero arguments.

Parameter(s):
1. `…`: quoted argument list

Example(s):

```
m5_call(first['']m5_comma_args(['$@']), last)
```

See also: `m5_call_varargs`

`m5_echo_args(…)`

Description: For rather pathological use illustrated in the example, …

Output: the argument list (`$@`)

Parameter(s):
1. `…`: the arguments to output

Example(s):

```
m5_macro(append_to_paren_list, ['m5_echo_args$1, $2'])
m5_append_to_paren_list((one, two), three)
```

Example Output:

```
(one,two,three)
```

9.5.4. Arithmetic Macros

`m5_calc(Expr, Radix, Width)`

Description: Calculate an expression. Calculations are done with 32-bit signed integers. Overflow silently results in wraparound. A warning is issued if division by zero is attempted, or if the expression could not be parsed. Expressions can contain the following operators, listed in order of decreasing precedence.

- `()`: For grouping subexpressions
- `+`, `-`, `~`, `!`: Unary plus and minus, and bitwise and logical negation
- `**`: Exponentiation (exponent must be non-negative, and at least one argument must be non-zero)
- `*`, `%`: Multiplication, division, and modulo
- `+ -`: Addition and subtraction
- `<<`, `>>`: Shift left or right (for shift amounts > 32, the amount is implicitly ANDed with `0x1f`)
- `>`, `>=`, `<`, `<=`: Relational operators
- `==`, `!=`: Equality operators
- `&`: Bitwise AND
- `^`: Bitwise XOR (exclusive or)
- `|`: Bitwise OR
- `&&`: Logical AND
- `||`: Logical OR

All binary operators, except exponentiation, are left-associative. Exponentiation is right-associative.

Immediate values in `Expr` may be expressed in any radix (aka base) from 1 to 36 using prefixes as follows:

- (none): Decimal (base 10)
- `0`: Octal (base 8)
- `0x`: hexadecimal (base 16)
- `0b`: binary (base 2)
- `0r:`, where `r` is the radix in decimal: Base `r`.

Digits are `0`, `1`, `2`, ..., `9`, `a`, `b` ... `z`. Lower and upper case letters can be used interchangeably in numbers and prefixes. For radix 1, leading zeros are ignored, and all remaining digits must be `1`.

For the relational operators, a true relation returns 1, and a false relation return 0.

Output: the calculated value of the expression in the given `Radix`; the value is zero-extended as requested by `Width`; values may have a negative sign (`-`) and they have no radix prefix; digits > 9 use lower-case letters; output is empty if the expression is invalid

Parameter(s):

1. `Expr`: the expression to calculate
2. `Radix`(opt) : the radix of the output (default 10)
3. `Width`(opt) : a minimum width to which to zero-pad the result if necessary (excluding a possible negative sign)

Example(s):

```
1: m5_calc(2**3 <= 4)
2: m5_calc(-0xf, 2, 8)
```

Example Output:

```
1: 0
2: -00001111
```

`m5_equate(Name, Expr)`
`m5_operate_on(Name, Expr)`

Description: Set a variable to the result of an arithmetic expression computed by [m5_calc]. For `m5_operate_on`, the variable value implicitly preceeds the expression, similar to `+=`, `*=`, etc. in other languages.

Side Effect(s): the variable is set

Parameter(s):

1. `Name`: name of the variable to set
2. `Expr`: the expression/partial-expression to evaluate

Example(s):

```
m5_equate(Foo, 1+2)
m5_operate_on(Foo, * (3-1))
m5_Foo
```

Example Output:

```
6
```

See also: `m5_set`, `m5_calc`

`m5_increment(Name, Amount)`
`m5_decrement(Name, Amount)`

Description: Increment/decrement a variable holding an integer value by one or by the given amount.

Side Effect(s): the variable is updated

Parameter(s):
1. `Name`: name of the variable to set
2. `Amount`(opt) : the integer amount to increment/decrement, defaulting to zero

Example(s):
```
m5_increment(Cnt)
```

See also: `m5_set`, `m5_calc`, `m5_operate_on`

9.5.5. Boolean Macros

These have boolean (`0` / `1`) results. Note that some [m5_calc] expressions result in boolean values as well.

`m5_is_null(Name)`
`m5_isnt_null(Name)`

Output: [`0` / `1`] indicating whether the value of the given variable (which must exist) is empty

Parameter(s):
1. `Name`: the variable name

`m5_eq(String1, String2, ⋯)`
`m5_neq(String1, String2, ⋯)`

Output: [`0` / `1`] indicating whether the given `String1` is/is-not equivalent to `String2` or any of the remaining string arguments

Parameter(s):
1. `String1`: the first string
2. `String2`: the second string
3. `⋯`: further strings to also compare

Example(s):
```
m5_if(m5_neq(m5_Response, ok, bad), ['m5_error(Unknown response:
m5_Response.)'])
```

9.5.6. Within Functions or Code Blocks

`m5_fn_args()`
`m5_comma_fn_args()`

Description: `fn_args` is the numbered argument list of the current function. This is like `` `, but it can be used in a nested function without escaping (e.g. `$<label>@``). `comma_fn_args` is the same, but has a preceeding comma if the list is non empty.

Output:

Side Effect(s): none

Example(s):

```
m5_foo(1, m5_fn_args)           /// works for 1 or more fn_args
m5_foo(1[''']m5_comma_fn_args)   /// works for 0 or more fn_args
```

See also: `m5_fn_arg`, `m5_fn_arg_cnt`

`m5_fn_arg(Num)`

Description: Access a function argument by position from `m5_fn_args`. This is like, e.g. `$3`, but is can be used in a nested function without escaping (e.g. `$<label>3`), and can be parameterized (e.g. `m5_fn_arg(m5_ArgNum)`).

Output: the argument value.

Parameter(s):
1. `Num`: the argument number

See also: `m5_fn_args`, `m5_fn_arg_cnt`

`m5_fn_arg_cnt()`

Description: The number of arguments in `m5_fn_args` or `$. This is like, e.g. $`, but is can be used in a nested function without escaping (e.g. `$<label>#`).

Output: the argument value.

See also: `m5_fn_args`, `m5_fn_arg`

`m5_comma_fn_args()`

Description: Access a function argument by position from m5_fn_args. This is like, e.g. $3, but is can be used in a nested function without escaping (e.g. $<label>@), and can be parameterized (e.g. m5_fn_arg(m5_ArgNum)).

Output: the argument value.

See also: m5_fn_args, m5_fn_arg_cnt

m5_return_status(Value)

Description: Provide return status. (Shorthand for m5_on_return(set, status, m5_Value).) This negates any prior calls to return_status from the same function.

Side Effect(s): sets m5_status

Parameter(s): 1. Value(opt) : the status value to return, defaulting to the current value of m5_status

See also: m5_on_return, m5_Status, m5_[aftermath]

m5_on_return(…)

Description: Call a macro upon returning from a function. Arguments are those for m5_call. This is most often used to have a function declare or set a variable/macro as a side effect. It is also useful to perform a tail recursive call without growing the call stack.

Side Effect(s): that of the resulting function call

Parameter(s): 1. …:

Example(s):

```
fn(set_to_five, VarName, {
   on_return(set, m5_VarName, 5)
})
```

See also: m5_return_status, m5_[aftermath]

9.6. Checking and Debugging

m5_debug_level(level)

Description: Get or set the debug level.

Output: with zero args, the current debug level

Side Effect(s): sets `debug_level`

Parameter(s): 1. `level`(opt) : [`min`, `default`, `max`] the debug level to set

Example(s):

```
debug_level(max)
use(m5-1.0)
```

9.6.1. Checking and Reporting to STDERR

These macros output text to the standard error output stream (STDERR) (with `[` / `]` quotes represented by single characters). (Note that STDOUT is the destination for the evaluated output.)

`m5_errprint(text)`
`m5_errprint_nl(text)`

Description: Write to STDERR stream (with a trailing new-line for `errprint_nl`).

Parameter(s): 1. `text`: the text to output

Example(s):

```
m5_errprint_nl(['Hello World.'])
```

`m5_warning(message)`
`m5_error(message)`
`m5_fatal_error(message)`
`m5_DEBUG(message)`

Description: Report an error/warning/debug message and stack trace (except for `DEBUG_if`). Exit for fatal_error, with non-zero exit code.

Parameter(s): 1. `message`: the message to report; (`Error:` pre-text (for example) provided by the macro)

Example(s):

```
m5_error(['Parsing failed.'])
```

`m5_warning_if(condition, message)`
`m5_error_if(condition, message)`
`m5_fatal_error_if(condition, message)`
`m5_DEBUG_if(condition, message)`

Description: Report an error/warning/debug message and stack trace (except for `DEBUG_if`) if the given condition is true. Exit for fatal_error, with non-zero exit code.

Parameter(s):

1. `condition`: the condition, as in `m5_if`.
2. `message`: the message to report; (`Error:` pre-text (for example) provided by the macro)

Example(s):

```
m5_error_if(m5_Cnt < 0, ['Negative count.'])
```

`m5_assert(message)`
`m5_fatal_assert(message)`

Description: Assert that a condition is true, reporting an error if it is not, e.g. `Error: Failed assertion: -1 < 0`. Exit for fatal_error, with non-zero exit code.

Parameter(s):

1. `message`: the message to report; (`Error:` pre-text (for example) provided by the macro)

Example(s):

```
m5_assert(m5_Cnt < 0)
```

`m5_verify_min_args(Name, Min, Actual)`
`m5_verify_num_args(Name, Min, Actual)`
`m5_verify_min_max_args(Name, Min, Max, Actual)`

Description: Verify that a traditional macro has a minimum number, a range, or an exact number of arguments.

Parameter(s):

1. `Name`: the name of this macro (for error message)
2. `Min`: the required minimum or exact number of arguments
3. `Max`: the maximum number of arguments
4. `Actual`: the actual number of arguments

Example(s):

```
m5_verify_min_args(my_fn, 2, 0)
```

9.6.2. Uncategorized Debug Macros

`m5_recursion_limit` (Universal variable)

Description: If the function call stack exceeds this value, a fatal error is reported.

`m5_abbreviate_args(max_args, max_arg_length, ⋯)`

Description: For reporting messages containing argument lists, abbreviate long arguments and/or a long argument list by replacing long input args and remaining arguments beyond a limit with ['...'].

Output: a quoted string of quoted args with a comma preceding every arg.

Parameter(s):

1. `max_args`: if more than this number of args are given, additional args are represented as ['...']
2. `max_arg_length`: maximum length in characters to display of each argument
3. `⋯`: arguments to represent in output

Example(s):

```
m5_abbreviate_args(5, 15, $@)
```

10. Syntax Index

M5 processes the following syntaxes:

Table 1. Core Syntax

Use	Reference	Syntax
Vanishing comments	[comments]	`///`, `/**`, `**/`
Preserved comments	[comments]	`//`
Quotes	Quotes	`[']`
Macro calls	[calls]	e.g. `m5_my_fn(arg1, arg2)`
Numbered/special parameters	[numbered_params]	`$` (e.g. `$3`, `$@`, `$#`, `$*`)

Additionally, text and code block syntax is recognized when special quotes are opened at the end of a line or closed at the beginning of a line. See [blocks]. For example:

```
error(*<blk>{
    ~(['Hello World!'])
})
```

Block syntax incudes:

Table 2. Block Syntax

Use	Reference	Syntax
Code block quotes	Code Blocks	`[`, `]`, `{`, `}` (ending/beginning a line)
Text block quotes	Text Blocks	`[']` (ending/beginning a line)
Evaluating Blocks	[evaluating_blocks]	`*[`, `[`, `*{`, `}`, `*[']`
Statement with no output	[statements]	`foo`, `bar(⋯)` (`m5_` prefix implied)
Code block statement with output	Multi-line Constructs: Blocks and Bodies	`~foo`, `~bar(⋯)` (`m5_` prefix implied)
Code block output	Multi-line Constructs: Blocks and Bodies	`~(⋯)`

Though not essential, block labels can be used to improve maintainability and performance in extreme cases.

Table 3. Block Label Syntax

Use	Reference	Syntax
Named blocks	[named_blocks]	`<foo>` (preceding the open quote, after optional `*`) e.g. `*<bar>{` or `<baz>[']`
Quote escape	[escapes]	`']<foo>m5_Bar['`
Labeled number/special parameter reference	[labeled_parameters]	`$<foo>`, e.g. `$<foo>2` or `$<bar>#`

Many macros accept arguments with syntaxes of their own, defined in the macro definition. Functions, for example are fundamental. See [functions].

Index