M5 Text Processing Language User's Guide

To enrich any text format

M5 version 2.0, document subversion 1, 2024
by Steve Hoover, Redwood EDA, LLC
(steve.hoover@redwoodeda.com)

The M5 text processing language and tool enhances the Gnu M4 macro preprocessor, adding features typical of programming languages.

Table of Contents

1. Background Information

1.1. Overview

M5 is a macro preprocessor on steroids. It is built on the simple principle of text substitution but provides features and syntax on par with other simple programming languages. It is an easy and capable tack-on enhancement to any text format as well as a reasonable general-purpose programming language specializing in text processing. Its broad applicability makes M5 a valuable tool in every programmer/engineer/scientist/AI's toolbelt.

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 covers the M5 language as well as its standard Macro Library. This document's major version reflects the language version, and the minor version reflects the library version. There is also a document subversion distinguishing versions of this document with no corresponding language or library changes.

1.3. M5's Origin Story

I created M5 as a preprocessor for the TL-Verilog hardware language and later decoupled it as a stand-alone tool. The original intent was to use an out-of-the box macro preprocessor to provide a stop-gap solutions to missing TL-Verilog language features for "code construction" as TL-Verilog took shape. While other hardware languages build on existing programming languages to provide code construction, I wanted a simpler approach that would be less intimidating to hardware folks. M4 was the obvious choice as the most broadly adopted macro preprocessor.

M4 proved to be capable, but extremely difficult to work with. After a few years fighting with an approach that was intended to allow me to focus my attention elsewhere, I decided I needed to either find a different approach or clean up the one I had. I felt my struggles had led to some worthwhile insights and that there was a place in the world for a better text processing language/tool, so I carved out some time to polish my mountain of hacks.

Though M5 would benefit from a fresh non-M4/Perl-based implementation, I had to draw the line somewhere. At this point, that legacy is mostly behind the scenes, and while it's not everything I'd like it to be, it's close, and it's way better than any other text preprocessor I'm aware of.

So I hope you enjoy the language I never wanted to write. I'm actually rather proud of it and find new uses for it every day.

1.4. M5 Versus M4

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

This document is intended to stand on its own, independent of the M4 documentation. The M4 documentation can, in fact, be confusing due to M5's philosophical differences with M4.

Beyond M4, M5 contributes:

- features that feel like a typical, simple programming language
- literal string variables
- functions with named arguments
- variable/macro scope
- 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

1.5. 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).

1.5.1. Security

M4 has full access to its host environment (similar to most programming and scripting languages, but unlike many macro preprocessors). Malware can easily do harm. Third- party M5 code should be carefully vetted before use, or M5 should be run within a contained environment. M5 provides a simple mechanism for library inclusion by URL (or it will). This enables easy execution of public third-party code, so use it with extreme caution.

1.5.2. 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.

1.5.3. 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 `regex(…)` and `substr(…)` (for strings of limited length) that return quoted (literal) text, `m4_patsubst` cannot be fixed (though `for_each_regex(…)` is similar). 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 that resulting text does 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 `regex(…)`, and `substr(…)`.

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

1.5.4. Introspection

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

1.5.5. Recursion

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

1.5.6. Unicode

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 (0x10-0xFF) that do not conflict by those defined by ASCII (0x00-0x7F). All such bytes (0x10-0xFF) 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, `length(…)` 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.

1.5.7. Debugging features

M4's facilities for associating output with input only map output lines to line numbers of top-level calls. M4 does not maintain a call stack with line numbers.

M4 and M5 have no debugger to step through code. Printing (see `DEBUG(…)` is the debugging mechanism of choice.

1.5.8. Performance

M5 is intended for text processing, not for compute-intensive algorithms. Use a programming language for that.

1.5.9. Graphics

M5 is for text processing only.

1.5.10. Status

Major next steps include:

- Implementing a better library system.
- Some syntactic sugar (quotes, code blocks) should not be recognized in source context.

See issues file in the M5 repository for more details.

2. Getting Started with the M5 Tool

2.1. Configuring M5

M5 adds a minimal amount of syntax, and it is important that this syntax is unlikely to conflict with the target language syntax. The syntax that could conflict is listed in Ensure No Impact. Currently, there is no easy mechanisms to configure this syntax.

2.2. Running M5

The Linux command:

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

runs M5 in its default configuration.

(Currently, there's a dependency on M4 and perl and no installation script.)

2.3. Ensure No Impact

When enabling the use of M5 on a file, first, be sure M5 processing does nothing to the file. M5 should output the input text, unaltered, as long as your file contains no:

- quotes, e.g. `[', ']`) *(This requirement is being removed with no quote processing in source context.)*
- `m5_` or `m4_`
- M5 comments, e.g. `///`, `/**`, `**/`
- code blocks, e.g. `[` or `{` followed by a newline or `]` or `}` beginning a line after optional whitespace *(This requirement is being removed with no processing in source context.)*

2.4. Tool Flow

Since M5 is simply substituting text, you can do bizarre things, which can be difficult to debug. Understanding the tool flow can help you look one step under the hood to debug issues or understand how syntax is interpreted.

M5 basically processes files in two steps:

- Interpret syntactic sugar.
- Run M4.

(There is a third step as well that is very minor to undo some of the sugaring.)

Object files are generated (run `m5 -h` for options) that expose the interpretation of M5 sugar. Note that quotes and commas are substituted with control characters in these files, so you will need an appropriate tool to view them. Your shell may recognize them as binary files and prompt you about viewing them, which is fine to do.

3. An Overview of M5 Concepts

3.1. Macro Preprocessing in General

Macro preprocessors extend a target programming language, text format, or general text file with the ability to define and call (aka instantiate, invoke, expand, evaluate, or elaborate) parameterized macros that provide text substitutions. Macros are generally used to provide convenient shorthand for commonly-used constructs. A macro preprocessor processes a text file sequentially with a default behavior of passing the input text through as output text. When a macro name is encountered, it and its argument list are substituted for new text according to its definition.

M5 provides convenient syntax for macro preprocessing as well as programatic text processing, sharing the same macros for each. This provides advanced text manipulation to supercharge any programming language or text file.

3.2. Macros Overview

A macro that simply outputs a static text string can be defined within the source file like this:

```
m5_macro(hello, Hello World!)
```

The above text will substitute with an empty string but will define a macro that can be called like this:

```
m5_hello()
```

Resulting in:

```
Hello World!
```

Macros can also be parameterized. Here we define a macro that outputs a string with a single parameter referenced as `$1`:

```
m5_macro(hello, Hello $1!)
```

And call it like this:

```
m5_hello(World)
```

Resulting in:

```
Hello World!
```

For more details on macro syntax, see Declaring Macros, Calling Macros, and Macro Arguments.

3.3. Quotes Overview

Quotes (`[` and `]`) may be used around text to prevent substitutions. For example, to provide a macro whose result includes a comma, quotes are needed:

```
m5_macro(hello, ['Hello, $1!'])
```

Without these quotes, the comma in `Hello, $1!` would be interpreted as a macro argument separator.

Furthermore, a second level of quotes may be needed to prevent the interpretation of the comma after substitution:

```
m5_macro(hello, ['['Hello, $1!']'])
m5_hello(World)
```

The call substitutes with `['Hello, World!']` (actually `['Hello, World!'][']`), which elaborates to the literal text:

```
Hello, World!
```

For more details on quote use, see Quotes.

3.4. Variables Overview

Variables hold string values. They can be thought of as macros without arguments. They are defined as:

```
m5_var(Hello, ['Hello, World!'])
m5_var(Age, 23)
```

And used as:

```
m5_Hello I am m5_Age years old.
```

Resulting in:

```
Hello, World! I am 23 years old.
```

Variables are always returned as literal strings, so a second level of quoting is not required for the definition of `Hello`.

Variables are scoped, and by convention, scoped definitions are named in camel case (strictly speaking, Pascal case).

For more details on variable use, see Variables and Variable Sugar.

3.5. Macro Stacks

All macros and variables, are actually stacks of definitions that can be pushed and popped. (These stacks are frequently one entry deep.) The top definition is active, providing the replacement text when the macro/variable is instantiated. The others are only accessible by popping the stack. Pushing and popping are not generally done explicitly, but rather through scoped declarations. See Scoped Code Blocks.

3.6. Code Syntax Overview

The above syntax is convenient in "source context", embedded into another language. It is clear where substitutions occur because all macro calls and variables are referenced with an `m5_` prefix. This syntax, however, quickly becomes clunky for any substantial text manipulation, requiring excessive `m5_`-prefixing. Additionally, it is difficult to format code readably because carriage returns and other whitespace are generally taken literally. This results in single-line syntax with many levels of nesting that quickly become difficult to follow.

To enable code structure that looks more like a programming language, "code context" can be established within which code syntax is supported.

Take for example this one-line definition in source context of an `assert` macro:

```
m5_macro(assert, ['m5_if(['$1'], ['m5_error(['Failed assertion: $1.'])'])'])
```

This can be written equivalently (though with a slight performance impact) as:

```
m5_macro(assert, {
   if(['$1'], [
      error(['Failed assertion: $1.'])
   ])
})
```

`m5_macro(` enters "argument list context", where parentheses and brackets have special meaning. `{` at the end of its line enters code context, where, most notably, text does not implicitly pass through to the output and `m5_` is implied at the beginning of each code statement (beginning its line). On the final line, `}` and `)` exit these contexts

For more details, see Code Blocks and Contexts.

3.7. Functions and Scope Overview

M5 also provides a syntax for function declarations with named parameters. The assert macro can be defined as a function as:

```
fn(assert, Expr, {
   if(m5_Expr, [
      error(Failed assertion: m5_Expr.)
   ])
})
```

Like any respectable programming language, `Expr`, above, is local to the function. Functions and other macros may produce result text (see Function Output Example and Code Blocks). They may also produce side effects including variable declarations (see Aftermath) and STDERR output (see `error(⋯)`).

For more details on functions, see Functions. For more details on scope, see Scoped Code Blocks.

3.8. Function Output Example

We can add output text to this function indicating assertion failures in the resulting text:

```
fn(assert, Expr, {
   ~if(m5_Expr, [
      error(Failed assertion: m5_Expr.)
      ~(Failed assertion: m5_Expr.)
   ])
})
```

Statements producing output are prefixed with a tilde (`~`).

3.9. Libraries and Namespaces Overview

M5 has a simple and effective import mechanism where a macro library file is simply imported by its URI (URL or local file). Libraries can be imported into their own namespace (though this mechanism is not yet implemented).

3.10. Processing Steps

Several of the above constructs, including code blocks and statements are termed "syntactic sugar" and are processed in a first pass before macro substitution—yes as a pre-preprocessing step.

M5 processing involves the following (ordered) steps:

- Substitute quotes for single control characters.
- Process syntactic sugar (in a single pass):
  - Strip M5 comments.
  - Process other syntactic sugar, including block and label syntax.
  - Process pragmas; check indentation and quote/parenthesis matching.
- Write the resulting file.
- Run M4 on this file (substituting macros).

4. Sugar-Free M5 Details

4.1. Defining "Sugar-Free"

M5 can be used "sugar-free". It's just a bit clunky for humans. Syntactic Sugar is recognized in the source file. Text that is constructed on the fly and evaluated (e.g. by `eval(⋯)`) is evaluated sugar-free.

4.2. Quotes

Unwanted processing, such as macro substitution, can be avoided using quotes. By default, these are `['` and `']` (and a configuration mechanism is not yet available to change this). Like syntactic sugar, they are recognized only when they appear in a source file and cannot be constructed from

their component characters. Quotes, however, are an essential part of M5, not a syntactic convenience.

Quoted text begins with `[']`. The quoted text is parsed only for `[']` and `[']` and ends at the corresponding `[']`. The quoted text passes through to the resulting text, including internal matching quotes, without any substitutions. The outer quotes themselves are discarded. The end quote acts as a word boundary for subsequent text processing.

Within quotes, intervening characters that would otherwise have special treatment, such as commas, parentheses, and `m5_`-prefixed words (after sugar processing), have no special treatment.

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
```

(`Index\m5_Index` is a shorthand for this. See Backslash Word Boundary.)

Quotes can also be used to avoid the interpretation of `m5_foo` as syntactic sugar. (See Macro Call Sugar.)

Special syntax is provided for multi-line literal quoted text. (See Code Blocks.) Outside of those constructs, quoted text should not contain newlines since newlines are used to format code. Instead, the `nl()` variable (or macro) provides a literal newline character, for example:

```
m5_DEBUG(['Line:']m5_nl['  ']m5_Line)
```

4.3. Variables

A variable holds a literal text string. Variables are defined using: `var(…)`, are reassigned using `set(…)`, and are accessed using `get(…)`. For example:

```
m5_var(Foo, 5)
m5_set(Foo, m5_calc(m5_Foo + 1))
m5_get(Foo)
```

Syntactic sugar provides variable access using, e.g., `m5_Foo` rather than `m5_get(Foo)`. (See Variable Sugar.)

4.4. Declaring Macros

Here we declare an `echo` macro.

```
m5_macro(echo, ['['$1']'])
```

where

```
m5_echo(['Hello, World!'])
```

substitutes with `['Hello, World!']`, and this elaborates as `Hello, World!`.

The most direct way to declare a macro is with `macro(…)`. For example:

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

This defines the macro body as `['Args:$1,$2']`.

A macro call returns the body of the macro definition with numbered parameters substituted with the corresponding arguments. Dollar parameter substitutions are made throughout the entire body string regardless of the use of quotes and adjacent text. The result is then evaluated, so these macros can perform computations, assign variables, provide argument lists, etc. In this case, the body is quoted, so its resulting text is literal. For example:

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

A few special dollar parameters are supported in addition to numbered parameters. The following notations are substituted:

- `$1`, `$2`, etc.: These substitute with corresponding arguments.
- `$#`: The number of arguments.
- `$@`: This substitutes with a comma delimited list of the arguments, each quoted so as to be taken literally. So, `m5_macro(foo, ['m5_bar($@)'])` is one way to define `m5_foo(…)` to have the same behavior as `m5_bar(…)`.
- `$*`: This is rarely useful. It is similar to `$@`, but arguments are unquoted.
- `$0`: The name of the macro itself. It can be convenient for making recursive calls (though see `recurse(…)`). `$0__` can also be used as a name prefix to localize a macro name to this macro, though this use model is discouraged. (See Variable Masking.) For Functions, `$0` is the internal name holding the function body. It should not be used for recursion but can be used as a unique prefix.

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 Block Labels.

A richer declaration mechanism is provided by `fn(⋯)`. (See Functions.)

4.5. Calling Macros

The following illustrates a call of the macro named `foo`:

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

NOTE	When this syntax appears in a source file, it is recognized as syntatic sugar and is processed to provide additional checking. Here, we specifically descibe the processing of this syntax when constructed from other processing, noting that syntactic sugar results in similar behavior. (See. Macro Call Sugar.)

A well-formed M5 macro name is comprised of one or more word characters (`a-z`, `A-Z`, `0-9`, and `_`).

When elaboration encounters (in unquoted text and without a preceding word character or immediately following another macro call) `m5_`, followed immediately by the well-formed name of a defined macro, followed immediately by `(` (e.g. `m5_foo(`) an argument list (see Macro Arguments) is processed, then the macro is "called" (or "expanded"). `$` substitutions are performed on the macro body (see Declaring Macros), the resulting text replaces the macro name and argument list followed by an implicit `['']` to create a word boundary, and elaboration is resumed from the start of this substituted text.

Macro names should not be encountered without an argument list. Though this would result in calling the macro with zero arguments, it is discouraged due to the syntactic confusion with variables. Macros can be called with zero arguments using `m5_call(macro_name)` instead. (See `call(⋯)`.)

NOTE	Though discouraged, it is possible to define macros with names containing non-word characters. Such macros can only be called indirectly (e.g. `m5_call(b@d, args)`). (See `call(⋯)`.)

NOTE	In addition to `m5_` macros, the M4 macros from which M5 is constructed are available, prefixed by `m4_`, though their direct use is discouraged and this document does not describe their use. Elaboration of the string `m4_` should by avoided.

4.6. Macro Arguments

Macro 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, newlines, and tabs. An unquoted `)` ends the

list.

Some examples to illustrate preceding and postceding whitespace and nested macros:

If, `m5_foo(A,B)` echoes its arguments to produce literal text `{A;B}`, then:

```
m5_foo(  A,  B)          ==> Yields: "{A;B}"
m5_foo(    ['']  A,B)    ==> Yields: "{  A;B}"
m5_foo(  A  ,  B  )      ==> Yields: "{A  ;B  }"
m5_foo(m5_foo(A, B), C)   ==> Yields: "{{A;B};C}"
m5_foo(m5_foo([')'],B),C)==> Yields: "{{);B};C}"  (with a warning about unbalanced
parentheses)
```

Arguments can be empty text, such as `()` (one empty argument) and `(,)` (two empty arguments). Note that the use of quotes is prefered for clarity. For example, `([''])` and `([''], [''])` are identical to the previous cases.

The above syntax does not permit macro calls with zero arguments, but `m5_call(macro_name)` can be used for this purpose. (See `call(…)`.)

Be aware that when argument lists get long, it is useful to break them up on multiple lines. The newlines should precede, not postcede the arguments, so they are not included in 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 newline and spaces in the second argument.

5. Syntactic Sugar

Syntactic sugar is syntax that is processed directly in the source file prior to macro processing. (See Processing Steps.)

5.1. Comments

5.1.1. M5 Comments (/// and /**...**/)

M5 comments are one form of syntactic sugar. They look like:

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

Block comments begin with `/**` and end with `**/`. Line comments begin with `///` and end with a

newline. Both are stripped prior to any other processing. As such:

- M5-commented parentheses and quotes are not visible to parenthesis and quote matching checks, etc.
- M5 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.

Whitespace preceding a line comment is also stripped. Newlines from block comments are preserved.

NOTE	Text immediately following `**/` may, after stripping the comment, begin the line. Comments are stripped before indentation checking. It is thus generally recommented that multi-line block comments end with a newline.

In case `///` or `/**` are needed in the resulting file, quotes can be used, e.g.: `['//']['/']`, to disrupt the syntax.

5.1.2. Target-Language Comments (E.g. `//`)

Comments in the target language are not recognized as comments by M5. To disable M5 code, it is important to use M5 comments, not target-language comments. (Thus it can be especially problematic when one's editor mode highlights target-language comments in a manner that suggests the code has no impact.)

5.1.3. Statement Comments (E.g. `/`)

These are specific to Code Blocks, introduced later.

5.2. Macro Call Sugar

`m5_\foo(` is syntactic sugar for `m5_\call(foo,`. (See `call(⋯)`.) This transformation (as long as it is evaluated) has no impact other than to verify that the macro exists. `m5_\foo(` should not appear in literal text that is never to be evaluated as it would get undesirably sugared. (See Quotes and Backslash Word Boundary for syntax to avoid undesired sugaring.)

NOTE	M5 may avoid applying this sugar for common macros from the M5 core library that are assumed to be defined.

This `m5_foo(` syntax also enters "argument list context" (see Contexts).

5.3. Variable Sugar

`m5_Foo` (without a postceding `(`) is syntactic sugar for `m5_get(Foo)`. (See `get(⋯)`.) `m5_Foo` should not appear in literal text that is never to be evaluated as it would get undesirably sugared. (For syntax to avoid undesired sugaring, see Quotes and Backslash Word Boundary.)

5.4. Backslash Word Boundary (`m5_\` and `\m5_`)

As more convenient alternatives to quotes:

- `m5_\foo` results in `m5_foo` without sugaring. This should be used in quoted, non-evaluated context when the literal string `m5_foo` is desired.
- `\m5_foo` is shorthand for `['']m5_foo` to provide a word boundary, enabling M5 processing of `m5_foo` when preceded by a word.

5.5. Multi-line Constructs: Blocks and Bodies

5.5.1. 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 `if(…)` and `loop(…)` have "immediate" body parameters. These bodies are to be evaluated by calls to these macros themselves. The final argument to a function or macro declaration is an "indirect" body argument. This 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 Functions with Body Arguments.

Code Blocks are convenient syntactic sugar constructs for multi-line body arguments formatted like code.

[Text blocks] are syntactic sugar for specifying multi-line blocks of arbitrary text, indented with the code.

5.5.2. 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'])
```

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.

5.5.3. Code Blocks

M5 supports special multi-line syntactic sugar 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 itself called from within a code body).

```
/Might result in "Yes, 4 > 2".
~if(m5_A > m5_B, [
   ~(['Yes, '])
   ~A
   ~([' > '])
   ~B
])
```

The block begins with `[`, followed immediately by a newline. 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 Scoped Code Blocks.

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 (after stripping M5 comments) begin a "statement". 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.

Essentially, the body, when evaluated, results in the text produced by its statements, which are macros or variables, listed without their `m5_` prefix, or inline text.

Specifically, statements can be:

- Macro calls, such as `~if(m5_A > m5_B, ⋯)`.
- Variable elaborations, such as `~A`.
- Output statements, such as `~(['Yes, '])`.
- Comments, such as `/A comment`.

Statements that produce output (as all statements in the above example's code block do) must be preceded by `~` (and others may be). This simply helps to identify the source of code block ouput. The `~(⋯)` syntax produces the given 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 earlier example behaves the same as:

```
m5_out(m5_if(m5_A > m5_B, m5__block(['
   m5_out(['Yes, '])
   m5_out(m5_get(A))
   m5_out([' > '])
   m5_out(m5_get(B))
'])))
```

The (internal) `m5__block` macro evaluates its argument and results in any text captured by `m5_out`.

5.5.4. Scoped Code Blocks

Scoped Code Blocks are delimited by { / } quotes. Within a code block, variable declarations (e.g. made by `var(⋯)`) are scoped. Their definitions are pushed by the declaration, and popped at the end of their scope. (See Macro Stacks regarding pushing and popping.)

It is recommended that all indirect body arguments (see Multi-line Constructs: Blocks and Bodies), such as those of `fn(⋯)` be scoped. Immediate body arguments (see Multi-line Constructs: Blocks and Bodies), such as those of `if(⋯)`, are most often unscoped, but scope may be used to isolate the side effects of the block to explicit `out_eval(⋯)` calls. Scoped and unscoped blocks are illustrated in the following example:

```
fn(check, Cond, {
   if(m5_Cond, [
      warning(Check failed.)
   ])
)}
```

Declarations from outer scopes are visible in inner scopes. Similarly, declarations from calling scopes are visible in callee scopes, though functions should generally be written without any assumptions about the calling scope. 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.

By convention, scoped variables and macros use Pascal case, e.g. `MyVar`. (See [Macro Naming Conventions].)

5.5.5. Text Blocks

"Text blocks" provide a syntax for multi-line quoted text that is indented with its surroundings. They are delimited similarly to code blocks, but use standard (`[']` / `[']`) quotes. The openning quote must be followed by a newline 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 newline are removed. For example:

```
macro(copyright, ['['
   Copyright (c) 20xx
   All rights reserved.
']'])
```

This is equivalent to:

```
macro(copyright, ['['Copyright (c) 20xx']m5_nl['All rights reserved.']'])
```

The text of the block is in source context, thus syntactic sugar is interpretted under the assumption that the text is to be evaluated. Text blocks that contain literal (quoted) text that is not evaluated should avoid entering argument list context with `m5_`, using quotes or `$` (if within a macro body), and it should be understood that vanishing comments would be removed.

5.5.6. Evaluate 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 a scoped evaluate code block is used to form an error message by searching for negative arguments:

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

5.5.7. 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. It can improve maintainability, code clarity, and performance to make judicious use of block labels. Note, however, that **the need for block labels is rare** and is mostly replaced by mechanisms provided by Functions.

Blocks can be labeled using syntax such as:

```
macro(my_macro, ..., <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 macro for parsing

text. This macro declares a helper macro `ParseError` for reporting parse errors that can be used many times by `my_parser`.

```
/Parse a block of text.
macro(my_parser, {
   var(Text, ['$1'])  /// Text to parse
   var(What, ['$2'])  /// A description identifying what is begin parsed
   /Report a parse error, e.g. m5_ParseError(['unrecognized character'])
   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 Variable 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 would use:

```
macro(ParseError, <err>{
   error(['Parsing of ']<err>m5_What[' failed with: "$<err>1"'])  /// 2 Fixes!
})
```

This code corrects both issues:

- `']<err>m5_What['`: This syntax acts in this case as `']']m5_nquote(1,m5_get(What))['['`, escaping enough levels of quoting to evaluate `m5_What` in the text of the `err` block and having the effect of using the definition of `m5_What` at the time of the macro definition. (The added level of quotes corresponds to the `{` / `}` block quotes which are sugar for `['` / `']`.)
- `$<err>1`: This syntax associates `$1` with the `err` block and is in this example equivalent to `']']m5_nquote_dollar(1,1)['['`.

5.6. Contexts

The various features of M5 apply in different contexts. This section summarizes the syntaxes that transition among contexts and the syntactic features available in each context. The context in which various features are supported is also summarized in Reference Card. Contexts can be nested, with the innermost context determining which features are available.

The following file illustrates different contexts:

```
Copyright (c) Joe Cool      /// source context
m5_do([                     /// enter argument list context then code context
   var(Ver, 1.0)            ///     code context
   var(Banner, ['           /']['//     code context, enters source context
      Zap™ (v']m5_Ver[')    /']['//       text (escaping to code) context
      Author: Joe Cool      /']['//       text context
   '])                      ///     exits source context
])                          /// exit code context then argument list context
File version: m5_Ver        /// source context
```

5.6.1. Source Context

Source context generally passes text through to the output. It is the default context and is also the context of text blocks.

Features supported in source context are supported in all contexts. For text that is intended to be literal, caution must be taken to avoid inadvertent use of these syntaxes. (See Ensure No Impact.)

The following are recognized in source context:

- Vanishing comments
- Macro calls
- Variable instantiation
- Pragmas

5.6.2. Text Context

Text context is the default context entered by (block or non-block) `['` quotes.

In addition to the features of source context, the following are recognized in text context:

- quotes are parsed and matched (see Quote and Parenthesis Matching)

5.6.3. M5 Context

Argument list context is entered from source and text contexts by, for example, `m5_foo(`. This context is exited by the corresponding `)`. In addition to text context features, the following are recognized in argument list context:

- code and text blocks
- parentheses are matched (see Quote and Parenthesis Matching)

5.6.4. Code Context

Code context is for Code Blocks, supporting syntactic sugar for formatting macro code more like programs.

Code context is entered by [/{ that end a line (after stripping vanishing comments) and is exited by the corresponding]/} beginning a line at matching indentation (also after stripping vanishing comments).

In addition to argument list context features, the following are recognized in code context:

- implicit m5_ beginning lines
- ~ allowing output (including, e.g. ~(hi), ~MyVar, ~nl())
- / comments

5.7. Syntax Checks and Pragmas

5.7.1. Indentation Checks

M5 checks that indentation is consistent for code and text blocks.

5.7.2. Quote and Parenthesis Matching

Parenthesis and quote matching is performed on the code after stripping comments. Quotes (including [/] and { / } quotes for code blocks) must be balanced.

Within each level of quotes, parentheses must be balanced. Parentheses in source and text context are excluded from this check, thus requiring parentheses for macros and parentheses that appear unquoted within macro arguments to be balanced.

Within a line, '] / [' quotes may be used (including nesting) to escape from and return to the same quoted context. This applies to contexts of all quote types, including code blocks, even though they are bound using different quote syntax. The context that is escaped from and returned to is the same context, thus parenthesis matching happens across the escaping. Thus, the parentheses on this code statement line are matching:

```
~hello(']<top>m5_Name[')
```

Here are some other examples:

```
m5_var(Expr, ['m5_calc(6 * (1 +']m5_Val['))'])   /// OK - both match
/// Similar, across two lines:
m5_var(Expr, ['m5_calc(6 * (1 +'])    /// Bad
m5_append_var(Expr, m5_Val['))'])     /// Bad
m5_var(Open, ['('])   /// OK - paren in text context
```

See also, open_quote(), close_quote(), and m5_pragma_[enable/disable]_paren_checks in Pragmas.

5.7.3. Pragmas

In certain cases quote and parenthesis checking gets in the way. It is possible to disable checking

and control debug behavior using pragmas. Pragmas processing happens after M5 comments are stripped. The following strings are recognized as pragmas:

- `where_am_i`: Prints the current quote context to STDERR.
- `[enable/disable]_[paren/quote]_checks`: For disabling parenthesis/quote checking.
- `[enable/disable]_sugar`: For disabling syntactic sugar (`m5_` and code/text blocks).
- `[enable/disable]_debug`: Improves the readability of the file resulting from sugar processing, and continues processing after normally-fatal errors.
- `[enable/disable]_verbose_checks`: Enables or disables verbose checking.

Since the pragmas would pass through to the target file, pragmas are generally expressed using the following macro calls which elaborate to nothing:

- `m5_pragma_where_am_i()`
- `m5_pragma_[enable/disable]_{check}()`, where `{check}` is `paren_checks`, `quote_checks`, `sugar`, `debug`, or `verbose_checks`.

5.8. Literal Commas

A comma (`,`) character appearing in source or text context is a "literal comma". It can never have special meaning as an argument separator even if used to construct a string that is evaluated as a macro call. A comma appearing in argument list or code context is a "non-literal comma". It is expected to be evaluated as a macro argument separator, though if never evaluated, it remains a `,` character and may pass through to the output.

Generally, comma characters will behave as expected, but, caution must be taken in situations where macro calls are constructed, then evaluated. For these rare cases, let's consider a few examples.

Here, the commas are argument separators:

```
m5_foo(A, B, C)
```

while those within quotes (in text context), here, are literal:

```
m5_macro(MyList, ['A, B, C'])
```

and `m5_foo(m5_MyList)` would receive a single parameter.

It is possible to define `MyList` to contain argument-separator commas using the variable, as:

```
m5_macro(MyList, A\m5_arg_comma B\m5_arg_comma C)
```

in which case `m5_foo(m5_MyList)` would receive three parameters.

In this example:

```
m5_macro(MyExpr, ['m5_foo(A, B, C)'])
```

all commas are argument separator commas. This defines `m5_MyExpr()` to invoke `m5_foo` with three parameters.

Below, however, `m5_\foo` is not recognized a macro call (though the `\` disappears), thus the commas separating `A`, `B`, and `C` are in text context and are literal (see Backslash Word Boundary):

```
m5_macro(MyExpr, ['m5_\foo(A, B, C)'])
```

and `m5_MyExpr()` would invoke `m5_foo` with a single parameter. This has the same effect as:

```
m5_macro(MyExpr, ['m5_foo(['A, B, C'])'])
```

(aside from the fact that the latter would be sugared and thus the existence of `m5_foo` would be confirmed).

6. Coding Practices

6.1. Coding Conventions

6.2. Status

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

Macros such as `else(…)` and `if_so(…)` take action based on `$status`.

Well-behaved macros set `$status` always or never (and never is the assumption if no side effect is listed in a macro's documentation). Thus `$status` is more like a return value than a sticky flag. Sticky behavior can be achieved using `sticky_status()`. There is no support for try-catch-like error handling. In bodies of `macro(…)` it may be necessary to explicitly save and restore status to avoid unintended side-effects on `$status` from calls within the bodies. `fn(…)` does this automatically. If `$status` is checked, it is generally checked immediately after a call.

6.3. Functions

All but the simplest of macros are most often declared using `m5_fn` and similar macros. These support a richer set of mechanisms for defining and passing parameter. While `m5_macro` is most often used with a one-line body definition, `m5_fn` is most often used with multi-line bodies as Scoped Code Blocks.

Such `m5_fn` declarations using Scoped Code Blocks look and act like functions/procedures/subroutines/methods in a traditional programming language, and we often refer to them as "functions". Function calls pass arguments into parameters. Functions' code block bodies contain macro calls (statements) that define local variables, perform calculations, evaluate code conditionally, iterate in loops, call other functions, recurse, etc.

Unlike typical programming languages, functions, like all macros, evaluate to text that substitutes for the calls. There is no mechanism to explicitly print to the standard output stream (though there are macros for printing to the standard error stream). Only a top-level call from the source code will implicitly echo to standard output.

Functions are defined using: `fn(…)` and `lazy_fn(…)`.

Declarations take the form:

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

A basic function declaration with a one-line body looks like:

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

Or, equivalently, using a code block body:

```
fn(mul, val1, val2, {
   ~calc(m5_val1 * m5_val2)
})
```

This `mul` function is called (in source context) like:

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

6.3.1. Parameters

6.3.1.1. Parameters Types and Usage

- **Numbered parameters**: Numbered parameters, as in `macro(…)` (see Declaring Macros), can be referenced as `$1`, `$2`, etc. with the same replacement behavior. However, they are explicitly identified in the parameter list (see The Parameter List). Within the function body, similar to `['$3']`, `fn_arg(…)` may also be used to access an argument. For example, `m5_fn_arg(3)` evaluates to the literal third argument value.
- **Special parameters**: As for `macro(…)`, special parameters are supported. Note that: `$@`, `$*`, and `$#` reflect only numbered parameters. Also, `$0` will not have the expected value, however `$0__` can still be used as a name prefix to localize names to this function. (See Variable Masking.) Similar to `$@`, the `fn_args()` macro (or variable) also provides a quoted list of the numbered arguments. Similar to `$#`, the `fn_arg_cnt()` macro also provides the number of numbered

arguments.

- **Named parameters**: These are available locally to the body as variables. They are not available to the Aftermath of the function.

6.3.1.2. The Parameter List

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

- A parameter specification of the form: `[?][[<number>]][[^]<name>][: <comment>]` (in this order), e.g. `?[2]^Name: the name of something`:
  - `<name>`: Name of a named parameter.
  - `?`: Specifies that the parameter is optional. Calls are checked to ensure that arguments are provided for all non-optional parameters or are defined for inherited parameters. Non-optional parameters may not follow optional ones.
  - `[<number>]`: Number of a numbered parameter. The first must be `[1]` and would correspond to `$1` and `m5_fn_arg(1)`, and so on. `<number>` is verified to match the sequential ordering of numbered parameters. Numbered parameters may also be named, in which case they can be accessed either way.
  - `^`: Specifies that the parameter is inherited. It must also be named. Its definition is inherited from the context of the func definition. If undefined, the empty `['']` value is provided and an error is reported unless the parameter is optional, e.g. `?^<name>`. There is no corresponding argument in a call of this function. It is conventional to list inherited parameters last (before the body) to maintain correspondence between the parameter list of the definition and the argument list of a call. Note that `` `$`s `` are problematic in inherited parameter values as undesired substitution may occur.
  - `<comment>`: A description of the parameter. In addition to commenting the code, this can be extracted in documentation.
- `⋯`: Listed after last numbered parameter to allow extra numbered arguments. Without this, extra arguments result in an error (except for the single empty argument of e.g. `m5_foo()`. See Function Call Arguments.)

6.3.2. When To Use What Type of Parameter

For nested declarations, the use of numbered parameters (`$1`, `$2`, ...) and special parameters (`$@`, `$*`, `$#`, and `$0`) can be extremely awkward. Nested declarations are declarations within the bodies of other declarations. Since nested bodies are part of outer bodies, numbered and special parameters within them would actually substitute based on the outer bodies. This can be prevented by generating the body with macros that produce the numbered parameter references, but this requires an unnatural and bug prone use of quotes. Therefore the use of functions with named parameters is preferred for inner macro declarations. Use of `fn_args()` and `fn_arg(⋯)` is also simpler than using special parameters. If parameters are named, these are helpful primarily to access `⋯` arguments or to pass argument lists to other functions.

Additionally, and in summary:

- **Numbered/special parameters**: These can be convenient to ensure substitution throughout the

body without interference from quotes. They can, however, be extremely awkward to use in nested definitions as they would substitute with the arguments of the outer function/macro. 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, `fn_args()` and `fn_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.

6.3.3. Function Call Arguments

Function calls must have arguments for all non-optional, non-inherited (`^`) parameters. 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. `m5_foo()` is permitted for a function `foo` declared with no parameters, though it is passed one emtpy parameter. (`m5_call(foo)` might be preferred.)

6.3.4. Function Arguments Example

In argument list context, function `foo` is declared below to display its parameters.

```
/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 argument list context):

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

And this expands to:

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

6.3.5. Aftermath

It is possible for a function to make assignments (and, actually do anything) in the calling scope. This can be done using `on_return(⋯)` or `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.

6.3.6. Passing Arguments by Reference

Functions can pass variables by reference and make assignments to the referenced variables upon returning from the function. For example:

```
fn(update, FooRef, {
   var(Value, ['updated value'])
   on_return(set, m5_FooRef, m5_Value)
}
set(Foo, ['xxx'])
update(Foo)
~Foo
```

A similar function could be defined to declare a referenced variable by using `var` instead of `set`.

The use of `on_return(⋯)` avoids the potential masking issue that would result from:

```
update(Value)
```

6.3.7. Returning Status

A function's `$status` should be returned via the function's aftermath, using `return_status(⋯)`, e.g.

```
fn(my_fn, Val, {
   if(m5_Val > 10, [''])
   return_status(m5_status)
})
```

Functions automatically restore `$status` after body evaluation to its value prior to body evaluation, so the evaluation of the body has no impact on `$status`. Aftermath is evaluated after this. It is fine to call `return_status(⋯)` multiple times. Only the last call will have a visible effect.

6.3.8. Functions with Body Arguments

The example below illustrates a function `if_neg` that takes an argument that is a body to evaluate. The body is defined in a calling function, e.g. `my_fn` on lines 15-16. Such a body is expected to evaluate in the context of the calling function, `my_fn`. Its assignment of `Neg`, on line 15, should be an assignment of its own local `Neg`, declared on line 12. Its side effects from `return_status(⋯)` on line 15 should be side effects of `my_fn`.

If the body is evaluated inside the function body, its side effects would be side effects of `if_neg`, not `my_fn`. The body should instead be evaluated as aftermath, using `on_return(⋯)`, as on line 6.

Note that `return_status(⋯)` is called after evaluating `m5_Body`. Both `on_return(⋯)` and `return_status(⋯)` add to the Aftermath of the function, and `$status` must be set after evaluating the body (which could affect `$status`).

Example of a body argument.

```
 1: // Evaluate a body if a value is negative.
 2: fn(if_neg, Value, Body, {
 3:    var(Neg, m5_calc(Value < 0))
 4:    ~if(Neg, [
 5:       /~eval(m5_Body)
 6:       on_return(Body)
 7:    ])
 8:    return_status(if(Neg, [''], else))
 9: })
10:
11: fn(my_fn, {
12:    var(Neg, [''])
13:    return_status(['pos'])
14:    ~if_neg(1, [
15:       return_status(['neg'])
16:       set(Neg, ['-'])
17:    ])
18:    ...
19: })
```

Since `macro(⋯)` does not support Aftermath, it is not recommended to use `macro(⋯)` with a body argument.

6.3.9. Tail Recursion

Recursive calls tend to grow the stack significantly, and this can result in an error (see `$recursion_limit`) as well inefficiency. When recursion is the last act of the function ("tail recursion"), the recursion can be performed in aftermath to avoid growing the stack. For example:

```
fn(my_fn, First, ..., {
   ...
   ~unless(m5_Done, [
      ...
      on_return(my_fn\m5_comma_args())
   ])
   ...
})
```

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

6.4.1. Variable Masking

Variable "masking" is an issue that can arise when a macro has side effects determined by its arguments. For example, an argument might specify the name of a variable to assign, or an argument might provide a body to evaluate that could declare or assign arbitrary variables. If the macro declares a local variable, and the side effect updates a variable by the same name, the local variable may inadvertently be the one that is updated by the side effect. This issue is addressed differently depending how the macro is defined. Note that using function Aftermath is the preferred method, but all options are listed here for completeness:

- Functions: Set variables using Aftermath. Using functions for variable-setting macros is preferred.
- Macros declaring their body using a code block: Set variable using `out_eval(⋯)`.
- Macros declaring their body using a string: Push/pop local variables named using `$0__` prefix.

7. Macro Library

This section documents the macros defined by the M5 1.0 library. Some macros documented here are necessary to enable inclusion of this library and are, by necessity, built-into the language. This distinction may not be documented.

7.1. 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.

7.2. Assigning and Accessing Macros/Variables

7.2.1. Declaring/Setting Variables

`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: `macro(⋯)`, `fn(⋯)`

`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: `var(⋯)`

`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: `pop(⋯)`

`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: `push_var(⋯)`, `push_macro(⋯)`

`null_vars(⋯)`

Description: Declare variables with empty values.

Side Effect(s): the variables are declared

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

7.2.2. Declaring Macros

`fn(⋯)`
`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: Functions

`macro(Name, Body)`
`null_macro(Name, Body)`

Description: Declare a scoped macro. See Declaring 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: `var(⋯)`, `set_macro(⋯)`

`set_macro(Name, Body)`

Description: Set the value of a scoped(?) macro. See Declaring 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: `var(…)`, `set_macro(…)`

`push_macro(Name, Body)`

Description: Push a new value of a 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: `pop(…)`, `macro(…)`, `set_macro(…)`

7.2.3. Accessing Macro/Variable Values

`get(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'])
get(OneDollar)
```

Example Output:
```
$1.00
```

See also: `var(…)`, `set(…)`

`must_exist(Name)`
`var_must_exist(Name)`

Description: Ensure that the `Name`d macro (`must_exist`) or variable (`var_must_exist`) exists.

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

7.3. Code Constructs

7.3.1. Status

`$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: `fn(…)`, `return_status(…)`, `else(…)`, `sticky_status()`

`$sticky_status` (Universal variable)

Description: Used by the `sticky_status()` macro to capture the value of `m5_status`.

See also: `$status`, `sticky_status()`

`sticky_status()`

Description: Used to capture the first non-empty status of multiple macro calls.

Side Effect(s): `$sticky_status` is set to `$status` if it is empty and `$status` is not.

Example(s):

```
if(m5_A >= m5_Min, [''])
sticky_status()
if(m5_A <= m5_Max, [''])
sticky_status()
if(m5_reset_sticky_status(), ['m5_error(m5_A is out of range.)'])
```

See also: `$status`, `sticky_status()`, `reset_sticky_status()`

`reset_sticky_status()`

Description: Tests and resets `$sticky_status`.

Output: [`0` / `1`] the original nullness of `$sticky_status`

Side Effect(s): `$sticky_status` is reset (emptied/nullified)

See also: `sticky_status()`

7.3.2. Conditionals

`if(Cond, TrueBody, …)`
`unless(Cond, TrueBody, FalseBody)`
`else_if(Cond, TrueBody, …)`

Description: An if/else construct. The condition is an expression that evaluates using `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`, `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 expression, evaluated as for `m5_calc`
2. `TrueBody`: the body to evaluate if the condition evaluates to true (1)
3. `…`: either a `FalseBody` or (for `m5_if` only) recursive `Cond`, `TrueBody`, `…` arguments to evaluate if the condition evaluates to false (not 1)

Example(s):

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

See also: `else(…)`, `case(…)`, `calc(…)`

`if_eq(String1, String2, TrueBody, …)`
`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, `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(m5_Zero, 0, [
   ~zero_is_zero(...)
], m5_calc(m5_Zero < 0), 1, [
   ~zero_is_negative(...)
], [
   ~zero_is_positive(...)
])
```

See also: `else(⋯)`, `case(⋯)`

`if_null(Var, Body, ElseBody)`
`if_var_def(Var, Body, ElseBody)`
`if_var_ndef(Var, Body, ElseBody)`
`if_defined_as(Var, Value, Body, ElseBody)`

Description: Evaluate `Body` if the named variable is empty (`if_null`), defined (`if_var_def`), not defined (`if_var_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: `if(…)`

`else(Body)`
`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: `if(…)`, `if_eq(…)`, `if_neq(…)`, `if_null(…)`, [m_if_def], [m_if_ndef], `var_regex(…)`

`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_var_def(First, m5_First)m5_else_if_def(Second,
m5_Second))
```

See also: `else_if(…)`, [m_if_def]

`case(Name, Value, TrueBody, …)`

Description: Similar to `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(Response, ok, [
   ~ok_response(...)
], bad, [
   ~bad_response(...)
], [
   error(Unrecognized response: m5_Response)
])
```

See also: `else(…)`, `case(…)`

7.3.3. Loops

`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 variable, 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 `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: `repeat(…)`, `for(…)`, `calc(…)`

`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(...)
])  //{empty}/ Iterates m5_LoopCnt 0..9.
```

See also: `loop(…)`

`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(...)
])  //{empty}/ (also maintains m5_LoopCnt)
```

See also: `loop(⋯)`

7.3.4. Recursion

`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: `$recursion_limit`, `on_return(⋯)`

7.4. Working with Strings

7.4.1. Special Characters

`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

`open_quote()`
`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: `quote(…)`

`$arg_comma` (Universal variable)

Output: A macro argument separator comma.

See also: Literal Commas

`orig_open_quote()`
`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: `printable_open_quote()`, `printable_close_quote()`

`printable_open_quote()`
`printable_close_quote()`

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

Output: the printable quote

See also: `orig_open_quote()`, `orig_close_quote()`

`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.
```

7.4.2. Slicing and Dicing Strings

`append_var(Name, String)`
`prepend_var(Name, String)`
`append_macro(Name, String)`
`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!
```

`substr(String, From, Length)`
`substr_eval(String, From, Length)`

Description: Extract a substring from `String` starting from `Index` and extending for `Length` ASCII characters (unicode bytes) 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: `dequote(…)`, `requote(…)`

`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([', '], ['one'], ['two'], ['three'])
```

Example Output:

```
one, two, three
```

`translit(String, InChars, OutChars)`
`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.
```

`uppercase(String)`
`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!
```

`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: `repeat(…)`

`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

7.4.3. Formatting Strings

`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
```

7.4.4. Inspecting Strings

`length(String)`

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

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

`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

`num_lines(String)`

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

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

`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`

7.4.5. Safely Working with Strings

`dequote(String)`
`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

`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: `printable_open_quote()`, `printable_close_quote()`

`no_quotes(String)`

Description: Assert that the given string contains no quotes.

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

7.4.6. Regular Expressions

Regular expressions in M5 use the same regular expression syntax as GNU Emacs. (See GNU Emacs Regular Expressions.) This syntax is similar to BRE, Basic Regular Expressions in POSIX and is regrettably rather limited. Extended Regular Expressions are not supported.

`regex(String, Regex, Replacement)`
`regex_eval(String, Regex, Replacement)`

Description: Searches for the first occurence of `Regexp` in `String`, resulting in either the position of the match or its 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: `var_regex(…)`, `if_regex(…)`, `for_each_regex(…)`

`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: `regex(…)`, `regex_eval(…)`, `if_regex(…)`, `for_each_regex(…)`

`if_regex(String, Regex, VarList, Body, …)`
`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. `…`: else body or 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),
[
   ~calc(m5_Src1 * m5_Src2)
], ['^incr\s+\(w+\)$'], (Src1), [
   ~calc(m5_Src1 + 1)
])
```

See also: `var_regex(…)`

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

Description: Evaluate body for every pattern matching regex in the first line of `String`. `$status` is unassigned.

Side Effect(s): side-effects of evaluating 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: `regex(…)`, `for_each_line(…)`, `regex_eval(…)`, `if_regex(…)`, `else_if_regex(…)`

7.5. Utilities

7.5.1. Fundamental Macros

`defn(Name)`

Output: the M4 definition of a macro; note that the M4 definition is slightly different from the M5 definition

Parameter(s):

1. `Name`: the name of the macro

`call(Name, …)`

Description: Call a macro. Versus directly calling a macro, 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.

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: `comma_shift(…)`, `comma_args(…)`

`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: `nquote(⋯)`

`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: `quote(⋯)`

`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)
```

`comment(…)`
`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. `…`:

7.5.2. Manipulating Macro Stacks

See Macro Stacks.

`get_ago(Name, Ago)`

Output:

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

Example(s):

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

Example Output:

```
AB
```

`depth_of(Name)`

Output: the number of values on a variable'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
```

7.5.3. Argument Processing

`shift(…)`
`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
```

`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)
m5_nargs(m5_eval(m5_ExampleList))
```

Example Output:

```
1
2
```

`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_argn(2, a, b, c)
```

Example Output:

```
b
```

`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(foo['']m5_comma_args(['$@']), last)
```

See also: `comma_shift(…)`, `comma_fn_args()`

`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, ${empty}2'])
m5_append_to_paren_list((one, two), three)
```

Example Output:

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

7.5.4. Arithmetic Macros

`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 `#` 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
```

`equate(Name, Expr)`
`operate_on(Name, Expr)`

Description: Set a variable to the result of an arithmetic expression computed by `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: `set(…)`, `calc(…)`

`increment(Name, Amount)`
`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: `set(…)`, `calc(…)`, `operate_on(…)`

7.5.5. Boolean Macros

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

`is_null(Name)`
`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

`eq(String1, String2, …)`
`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.)'])
```

7.5.6. Within Functions or Code Blocks

`fn_args()`
`comma_fn_args()`

Description: `m5_fn_args()` results in an unquoted list containing the numbered argument (each individually quoted) of the current function. This is like `$@`, but it can be more convenient in nested functions where the use of Block Labels (e.g. `$<label>@`) would be needed. `m5_comma_fn_args()` is the same, but has a preceeding comma if the list is non-empty. Note that these can be used as variables (`m5_fn_args` and `m5_comman_fn_args`) to provide quoted versions of these.

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: `fn_arg(…)`, `fn_arg_cnt()`

`fn_arg(Num)`

Description: Access a function argument by position from `m5_fn_args`. This is like, e.g. `$3`, but it can be more convenient in nested functions where the use of Block Labels (e.g. `$<label>3`) would be needed, and can be parameterized (e.g. `m5_fn_arg(m5_ArgNum)`).

Output: the argument value.

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

See also: `fn_args()`, `fn_arg_cnt()`

`fn_arg_cnt()`

Description: The number of arguments in `m5_fn_args` or `$#`. This is like, e.g. `$#`, but it can be more convenient in nested functions where the use of Block Labels (e.g. `$<label>#`) would be needed.

Output: the argument value.

See also: `fn_args()`, `fn_arg(…)`

`out(String)`
`out_eval(String)`

Description: These append to code block output that is expanded after the evaluation of the block. `m5_out` captures literal text, while the argument to `m5_out_eval` gets evaluated. Thus `m5_out_eval` is useful for code block side effects. `m5_out` is useful only in pathological cases within statements and by dynamically constructed code since the shorthand syntax `~(…)` is effectively identical to `~out(…)`. Note that these macros are not recommended for use in function blocks as functions have their own mechanism for side effects that applies outside of the function (after popping parameters). (See Aftermath.)

Output: no direct output, though, since these indirectly result in output as a side-effect, it is recommended to use `~` statement syntax with these

Side Effect(s): indirectly, `out_eval` can result in the side effects of its output expression

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

See also: Code Blocks, Aftermath

`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: `on_return(…)`, Status, Aftermath

`on_return(MacroName, …)`

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. `MacroName`: the name of the macro to call
2. `…`: its arguments

Example(s):

```
fn(set_to_five, VarName, {
   on_return(set, m5_VarName, 5)
})
fn(set2, VarName1, Val1, VarName2, Val2, {
   ...
   on_return(eval, {
      set(VarName1, m5_Val1)
      set(VarName2, m5_Val2)
   })
})
```

See also: `return_status(…)`, Aftermath

7.6. Checking and Debugging

`debug_level(level)`

Description: Get or set the debug level.

Output: with zero arguments, 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)
```

7.6.1. 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.)

`errprint(text)`
`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.'])
```

`warning(message)`
`error(message)`
`fatal_error(message)`
`DEBUG(message)`

Description: Report an error/warning/debug message and stack trace (except for `DEBUG`). 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.'])
```

`warning_if(condition, message)`
`error_if(condition, message)`
`fatal_error_if(condition, message)`
`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.'])
```

`assert(condition)`
`fatal_assert(condition)`

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. `condition`: the condition to check (and report)

Example(s):

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

`verify_min_args(Name, Min, Actual)`
`verify_num_args(Name, Min, Actual)`
`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, $#)
```

7.6.2. Uncategorized Debug Macros

`$recursion_limit` (Universal variable)

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

`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, $@)
```

8. Reference Card

M5 processes the following syntaxes:

Table 1. Core Syntax

Feature	Reference	Syntax	Contexts	Must Be Evaluated?
M5 comments	Comments	`///`, `/**`, `**/`	All	N/A
Quotes	Quotes	`[']`, `[']`	All	Yes
Macro calls	Calling Macros	e.g. `m5_my_fn(arg1, arg2)`	All (except as code statement)	Yes
Numbered/special parameters	Declaring Macros	`$` (e.g. `$3`, `$@`, `$#`, `$*`)	Within outermost macro/function (not var) definition body	N/A
Escapes	Backslash Word Boundary	`\m5_foo`, `m5_\foo`	All	`\m5_foo`-Yes, `m5_\foo`-Must not

The contexts listed under the "Contexts" column of Core Syntax are described in Contexts. "Yes" in the "Must Be Evaluated" column indicates that the syntax should not pass through to the output without being evaluated. Doing so may result in an error or unexpected output text.

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 Code Blocks. For example:

```
/Report error.
error(*<blk>{
   ~(['Something went wrong!'])
})
```

Block syntax incudes:

Table 2. Block Syntax

Feature	Reference	Syntax	Contexts
Code block quotes	Code Blocks	`[`, `]`, `{`, `}` (ending/beginning a line)	M5, code
Text block quotes	Text Blocks	`[']` (ending/beginning a line)	M5, code
Evaluate Blocks	Evaluate Blocks	`*[`, `[`, `*{`, `}`, `*[']`	M5, code
Statement comment	Statement Comments (E.g. `/`)	`/Blah blah blah…`	Code
Statement with no output	Code Blocks	`Foo`, `bar(…)` (`m5_` prefix implied)	Code

Feature	Reference	Syntax	Contexts
Code block statement with output	[bCode Blocks]	`~Foo`, `~bar(⋯)` (`m5_` prefix implied)	Code
Code block output	Code Blocks	`~(⋯)`	Code

All syntax in Block Syntax must be evaluated (strictly, must not be unevaluated).

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

Table 3. Block Label Syntax

Feature	Reference	Syntax	Contexts	Must Be Evaluated
Named blocks	Block Labels	`<foo>` (preceding the open block quote, after optional *) e.g. `*<bar>{` or `<baz>['`	M5, Code	Yes
Quote escape	Block Labels	`']<foo>m5_Bar['`	All (within any type of M5 quotes)	`<foo>m5_Bar`- Yes
Labeled number/special parameter reference	Block Labels	`$<foo>`, e.g. `$<foo>2` or `$<bar>#`	All (within corresponding block)	N/A

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

Index

S

T

U

V

W