Tilde Text Processing Language User's Guide

To enrich any text format

Tilde version 1.0, document subversion 1, 2024
by Steve Hoover, Redwood EDA, LLC
(steve.hoover@redwoodeda.com)

Tilde is a simple but capable language for text processing. It is built to extend the capabilities of any text file format without getting in the way. It extends concepts of macro preprocessing into the realm of programming, restricted to processing text.

Table of Contents

1. Background Information . . . 3
1.1. Overview . . . 3
1.2. About this Specification . . . 3
1.3. An Overview of Tilde Concepts . . . 4
1.4. Use Models . . . 5
1.5. Tilde's Origin Story . . . 5
1.6. Tilde Versus M4 . . . 5
1.7. Limitations of Tilde . . . 6
1.7.1. Security . . . 6
1.7.2. Modularity . . . 6
1.7.3. String processing . . . 6
1.7.4. Introspection . . . 7
1.7.5. Recursion . . . 7
1.7.6. Unicode . . . 7
1.7.7. Debugging features . . . 7
1.7.8. Performance . . . 7
1.7.9. Graphics . . . 7
1.7.10. Status . . . 7
2. Getting Started with the Tilde Tool . . . 8
2.1. Configuring Tilde . . . 8
2.2. Running Tilde . . . 8
2.3. Ensure No Impact . . . 8
2.4. Tool Flow . . . 8
3. Reading Tilde Code . . . 9

3.1. A Quick Overview by Example . . . 9
3.2. Literal Strings . . . 10
3.2.1. Quoted Literal Strings . . . 10
3.2.2. Unquoted Literal Characters . . . 11
3.2.3. Multiline Strings . . . 11
3.3. Arithmetic Expressions . . . 12
3.4. Variables . . . 12
3.5. Macro Calls . . . 13
3.6. Code Blocks . . . 13
3.7. Declaring Macros . . . 14
3.8. Special Code Syntax . . . 15
3.9. Functions . . . 16
3.10. Functions . . . 16
3.10.1. Parameters . . . 17
3.10.2. When To Use What Type of Parameter . . . 18
3.10.3. Function Call Arguments . . . 18
3.10.4. Function Arguments Example . . . 18
3.10.5. Aftermath . . . 19
3.10.6. Passing Arguments by Reference . . . 19
3.10.7. Returning Status . . . 20
3.10.8. Functions with Body Arguments . . . 20
3.10.9. Tail Recursion . . . 21
3.11. Block Labels . . . 21
3.12. String Labels . . . 23
3.13. Syntax Configuration . . . 23
4. Writing Tilde Code . . . 23
4.1. Code Formatting . . . 23
4.2. Unquoted Character Restrictions . . . 23
4.2.1. Unquoted Argument Characters . . . 23
4.2.2. Unquoted Arithmetic Expressions Characters . . . 24
4.3. Constructing Code . . . 24
5. Standard Library Reference . . . 25
5.1. Coding Paradigms, Patterns, Tips, Tricks, and Gotchas . . . 25
5.1.1. Variable Masking . . . 25
6. Macro Library . . . 25
6.1. Specification Conventions . . . 25
6.2. Assigning and Accessing Macros/Variables . . . 26
6.2.1. Declaring/Setting Variables . . . 26
6.2.2. Declaring Macros . . . 27
6.2.3. Accessing Macro/Variable Values . . . 29
6.3. Code Constructs . . . 29

6.3.1. Status . . . 29
6.3.2. Conditionals . . . 30
6.3.3. Loops . . . 34
6.3.4. Recursion . . . 36
6.4. Working with Strings . . . 36
6.4.1. Special Characters . . . 37
6.4.2. Slicing and Dicing Strings . . . 38
6.4.3. Formatting Strings . . . 41
6.4.4. Inspecting Strings . . . 43
6.4.5. Safely Working with Strings . . . 43
6.4.6. Regular Expressions . . . 44
6.5. Utilities . . . 47
6.5.1. Fundamental Macros . . . 47
6.5.2. Manipulating Macro Stacks . . . 49
6.5.3. Argument Processing . . . 50
6.5.4. Arithmetic Macros . . . 52
6.5.5. Boolean Macros . . . 55
6.5.6. Within Functions or Code Blocks . . . 56
6.6. Checking and Debugging . . . 58
6.6.1. Checking and Reporting to STDERR . . . 58
6.6.2. Uncategorized Debug Macros . . . 60
7. Reference Card . . . 60
Index . . . 62

1. Background Information

1.1. Overview

Tilde 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 Tilde a valuable tool in every programmer/engineer/scientist/AI's toolbelt.

This chapter provides background and general information about the Tilde language and tool and guidance about this specification.

1.2. About this Specification

This document covers the Tilde 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.

To target different users, this document contains the following sections:

- Background Information
- Getting Started with the Tilde Tool
- Reading Tilde Code
- Writing Tilde Code
- Standard Library Reference

To use Tilde on an existing Tilde file, you only need to read Getting Started with the Tilde Tool.

To understand a file that uses Tilde, you only need to read the Reading Tilde Code and consult the Standard Library Reference as needed.

To write Tilde code, you need to read the Writing Tilde Code section as well and become generally familiar with the Standard Library Reference.

This Background Information section is optional for everyone.

1.3. An Overview of Tilde Concepts

Tilde is built to enhance, not replace, any programming language, text format, or even plain-text file. Tilde aims to be:

- non-meddlesome, avoiding any syntax that could conflict with a target language
- easy to learn to write, and even easier to read
- capable (not optimal) as a general-purpose language (within the realm of text processing)
- enabling of custom embedded syntaxes
- easy to debug (potentially—debugging tools are not yet available)

Tilde may execute shell commands and can thus be used as a scripting language.

Libraries are included by URI (URL or local file) and may be cached locally. Thus no library installation or package manager is required.

There are only two data types: text and code. Code is to be evaluated. Text may be formatted to represent other data types. For example, a function may produce an "integer" value, meaning it produces a string containing the decimal representation of an integer value.

Multiline strings are easily expressed using indentation to avoid the need for escape characters. Multiline strings are a great way to make unrestricted embedded formats.

"Variables" typically hold text string values, and "macros" (most of which are "functions") hold code. Variables and functions are global name/value pairs, though each name can hold a stack of values, with the top value being active. These stacks are used to provide scoped variables/macros.

Macros have ordered and (generally) named parameters. They evaluate to strings. The macro body is code that contains a series of: literal text strings, variable instantiations, arithmetic expressions, and macro calls. Each may append to the output of the function or assign/push/pop variables/macros.

1.4. Use Models

Tilde can be enabled on any source file in your project. For example, in a Make-based build flow, add this to your Makefile:

```
%: %~
  tilde $< > $@
```

To then enable Tilde on any specific source file, simply rename the file with a trailing `~`.

Tilde can also be used to define parameterized web files. A webserver can be configured to serve parameterized files using Tilde (similar to PHP). For example, `http://example.com/index.html?foo=bar` can invoke Tilde on `index.html~` with a Tilde variable `foo` set to `bar` that is used to configure `index.html~`.

1.5. Tilde's Origin Story

I created Tilde 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 Tilde 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.

1.6. Tilde Versus M4

Tilde uses M4 to implement a text-preprocessing language with some subtle philosophical differences. Tilde 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 Tilde's philosophical differences with M4.

Beyond M4, Tilde contributes:

- features that feel like a typical, simple programming language
- literal string variables
- functions with named parameters
- 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

1.7. Limitations of Tilde

M4 has certain limitations that Tilde is unable to address without a re-implementation that is not based on M4 or modifications to M4 itself.

1.7.1. 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 Tilde code should be carefully vetted before use, or Tilde should be run within a contained environment. Tilde 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.

1.7.2. Modularity

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

1.7.3. 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 Tilde 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. Tilde 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.

1.7.4. Introspection

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

1.7.5. Recursion

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

1.7.6. 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.

1.7.7. 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 Tilde have no debugger to step through code. Printing (see `DEBUG(⋯)` is the debugging mechanism of choice.

1.7.8. Performance

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

1.7.9. Graphics

Tilde is for text processing only.

1.7.10. Status

This specification documents intended language syntax prior to a clean re-implementation. Stay tuned.

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 Tilde Tool

2.1. Configuring Tilde

Tilde 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 Tilde

The Linux command:

```
tilde in-file > out-file
```

runs Tilde 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 Tilde on a file, first, be sure Tilde processing does nothing to the file. Tilde should output the input text, unaltered, as long as your file contains no:

- `<">`

2.4. Tool Flow

Since Tilde 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.

Tilde 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 `tilde -h` for options) that expose the interpretation of Tilde 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. Reading Tilde Code

3.1. A Quick Overview by Example

The following example gives a general view of typical Tilde code and provides context for describing its language features. This example defines a "macro" called `withdraw` that updates a `Balance` variable (referenced as `$Balance`) and returns text describing the transaction. It should be fairly readable even without knowledge of Tilde syntax.

```
# Withdraw credits from $Balance.
# Args:
#   Amount: The number of credits to withdraw.
# Side Effects:
#   Updates $Balance.
# Output:
#   Text describing the transaction.
#   E.g.:
#     Withdrawing 40.
#     Starting balance: 100.
#     New balance: 60.
macro(withdraw, Amount, {
   # Report the transaction and balance.
   "Withdrawing " $Amount "." |     # Note: "|" produces a newline.
   "Starting balance: " $Balance "." |
   if([$Balance >= $Amount], {
      # Make/report the withdrawal.
      decrement(Balance, $Amount)
      "New balance: " $Balance "." |
   }, {
      # No withdrawal.
      "Insufficient balance." |
   })
})
```

This macro is called below:

```
var(Balance, 100)   # Declare variable $Balance with value 100.
withdraw(40)        # Withdraw 40 from $Balance.
```

and produces the text:

```
Withdrawing 40.
Starting balance: 100.
New balance: 60.
```

Macro arguments are supplied as "code"--a sequence of "statements", with optional whitespace separation, where statements may be:

- Literal strings, e.g. `"Withdrawing "` and `Balance`
- Variable instantiations, e.g. `$Amount`
- Arithmetic expressions: e.g. `[Balance >= $Amount]`
- Macros calls: e.g. `var(Balance, 100)`
- Special syntax: e.g. `|` for newline
- Code blocks: i.e. `{…}`

Aside from code blocks, each of these produces (possibly empty) text, and the result of evaluating the code is the concatenation of this text.

A macro argument can be as simple as a literal string or as complex as a function body. For example, the code above calls `macro` with three arguments as `macro(withdraw, Amount, {…})`. The first two arguments are unquoted strings (aka "tokens"), and the third is the entire body of the `withdraw` macro.

When calling a macro, the code of each argument is evaluated, and the resulting strings are passed into the parameters. For an argument that is enclosed in `{`/`}`, the parameter becomes the code itself, and this code may be evaluated later. The arguments to the `if` macro call above, for example, include two code arguments, and the `if` macro evaluates one of them.

The remainder of this section describes these language features in greater detail using their default syntax. Note that some syntax is configurable. See Syntax Configuration.

3.2. Literal Strings

3.2.1. Quoted Literal Strings

There are two syntaxes for quoted literal strings:

- `"`/`"`: Light quotes. These are used for literal text strings.
- `<">`/`<">`: Heavy quotes. These are also used for literal text strings and are generally used for larger, often multiline text.

Heavily quoted strings differ from lightly quoted strings only in the quote characters used. Heavy quotes reduce the likelihood of conflicts with end quote characters in the strings.

The only characters with special (non-literal) meaning within the string are the corresponding end quote characters. If the end quote characters are needed literally in the string, they can be followed by a `~` character. The `~` will be dropped and the string will continue until a non-literal end quote is encountered. Thus:

```
"Say, "~Cheese"~!"
```

represents the string: `Say, "Cheese"!`. Alternatively, heavy quotes can be used to avoid the need for the literal (`~`) character:

```
<">Say, "~Cheese"!<">
```

NOTE Most languages use an escape character that precedes, not follows, the character to be escaped. Such a prefix escape character would itself need to have an escape sequence. Using a postfix character avoids this need. Other escape scenarios are handled in Tilde by ending the string and starting a new one later, which is concatenated, e.g.: `"Hi, "$Name"!"`.

3.2.2. Unquoted Literal Characters

Argument characters that have no special meaning may pass through to a macro parameter literally.

For example:

```
var(Player.1.Score, 0)
```

is equivalent to:

```
var("Player.1.Score", "0")
```

For details on which characters are permitted without quotes, see Unquoted Character Restrictions.

3.2.3. Multiline Strings

Multiline strings begin with light or heavy quotes (`"` or `<">`) that are followed immediately by a newline. The string is indented beneath the line containing the begin quote. It is terminated by the corresponding close quote beginning a line at the same level of indentation as the line on which the string began. The string's indentation, as well as the initial and final newline are excluded from the string. For example:

```
set(MyPythonCode, <">
  def foo():
    print("Hello, world!")
<">)
```

The indentation is determined by the first non-whitespace character of the string. To enable multiline strings that begin with whitespace, a line containing only `v` can be used to point to the first character of the string. For example:

```
set(MyPythonCode, <">
  v
    def foo():
      print("Hello, world!")
<">)
```

Defines a similar string with two spaces of indentation.

3.3. Arithmetic Expressions

Arithmetic expressions can be provided within `[/]` and cannot be immediately preceded by word characters.

For example:

```
var(Val, [1 + 2 * 3])
$Val    # Yields: "7"
```

`[1 + 2 * 3]` is essentially shorthand syntax for `calc(1+2*3)`. (See `calc(…)`.)

Like other macro argument text, the text of the expression is code that evaluates to a string. The string, in this case, must be a valid expression as defined by `calc(…)`. The example below illustrates that expressions may contain subexpressions, variable instantiations, macro calls, and unquoted characters.

```
var(Val, [[$Val + 1] * if($Negate, -1, 1)])
```

The set of characters that may be unquoted is different for expressions than for macro arguments. (See [Argument Text].)

3.4. Variables

Variables are name/value pairs where each may be an arbitrary string. Values may also be code. (See [Code].)

Variables are global, though each can hold a stack of values, with the top value being active. These stacks are used to provide scoped variables.

Tilde does not specifically support data structures, but these can be implemented by libraries. For example, the variable `Player.1.Score` might represent a field `Score` of entry `1` of the `Player` array.

Variables are defined (or pushed to their stack) using: `var(…)`, are reassigned using `set(…)`, and are accessed using `get(…)`. For example:

```
var(Foo, 5)
increment(Foo)
get(Foo)
```

The syntax `$Foo` is shorthand for `get(Foo)`. Thus:

```
$Foo
```

3.5. Macro Calls

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

```
foo(hello, 5)
```

A well-formed macro name is comprised of one or more word characters (`a-z`, `A-Z`, `0-9`, and `_`). Though discouraged, it is possible to define macros with names containing non-word characters. Such macros can only be called indirectly, e.g. `call("b@d", args)`. (See `call(⋯)`.)

The text string for an argument, between `(`, `,`, and/or `)`, is code or a code block. (See [Code].)

Empty argument text, as in `foo(, )`, is not permitted. Empty argument values are typically provided as `""`, thus `foo("", "")` is legal syntax. `foo()` represents a zero-argument call, not a call with a single empty argument.

3.6. Code Blocks

Code blocks are code enclosed in `{`/`}`. They capture code for future evaluation.

A code block can be evaluated (to a string) using the `eval(⋯)` macro. For example:

```
eval({set(Foo, 5)})
```

Has the same effect as:

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

The `if(⋯)` macro, below, has a first argument that is a condition and second and third arguments that may be evaluated (using `eval(⋯)`) depending on the value of the condition. For example:

```
if($Okay, {doit()}, {error("Bad.")})
```

If `eval(⋯)` is given a string rather than a code block, it simply returns the string. Thus a macro argument that is to be evaluated as code may generally also be given without `{/}` to pre-evaluate the code. For example:

```
DEBUG("BoolVar is " if($BoolVar, {"true"}, {"false"}) ".")
```

can be simplified to:

```
DEBUG("BoolVar is " if($BoolVar, "true", "false") ".")
```

Generally, as above, a macro argument will either be a single code block or statements producing strings. There are, however, no restrictions on mixing code blocks and other statements. An argument can contain a mix of pre-evaluated strings and code blocks. Consider the following example, which iterates over items, listing each one with leading text:

```
repeat($NumItems, {$Indentation ": " GetItem($LoopCnt)})
```

The first argument of the `repeat` macro is the number of iterations. The second is code to evaluate for each iteration. `$LoopCnt` is implicitly defined by `repeat` as the loop index. In this example, `$Indentation ": "` is the same for every iteration and can be pre-evaluated. We can achieve this as:

```
repeat($NumItems, {} $Indentation ": " {GetItem($LoopCnt)})    # TODO: Is {} needed to
coerce here or will coercion necessarily happen later anyway.
repeat($NumItems, {:$Indentation ": " GetItem($LoopCnt)})      # Or, code block have
to be a complete argument?
```

TODO: Macro has everything a function has except scope for the body. That includes aftermath and push/pop of parameters (popped before aftermath). Function in not built-in and adds scope to the body. Code block is a macro with zero args. Nope: Calls push/pop $1… Functions add scope and push/pop params and are built-in so $1… can be assigned to … args only. Code blocks are functions with no parameters (so there can be no push/pop of $1… and no scope). Use foo() or eval(foo) to evaluate code blocks?? Nope: Macros are not built-in. Functions/code blocks are special built-in stacked variables and have text representations of their bodies accessible via body_of(…). (Functions are scoped; code blocks are not.) {} are code quotes. (They can be passed as code parameters as {foo()} (deferred) or get_code(foo).) Macros are a library function that modifies the code to replace $1, etc. TODO: Returning unquoted (inline) text to be evaluated, such as arg lists? Looks like I got rid of that in M5? Maybe comma should be allowed in a code block as an argument separator—no, too dangerous.

3.7. Declaring Macros

Macros are in some respects, like variables. While variables can only be assigned to strings, macros are assigned to code. Unlike variables, macros take parameters. Though macros are rarely defined to be scoped, they can be and they have stacks, like variables.

Macros declarations take the form:

```
macro(<name>, [<param-list>,] {<body>})
```

For example:

```
macro(hello, Name, {"Hello, " $Name "!"})
```

defines a macro named `Hello` that takes a parameter `Name`. This macro can be instantiated as:

```
hello("Joe")
```

resulting in the string: `Hello, Joe!`.

3.8. Special Code Syntax

`|` is shorthand for `nl()` and produces a newline.

`~` is used to force preceding end quote characters to be taken literally and to continue a string.

code(CallBarABC, {call(bar(a, b, c))}) CallBarABC() # bar(a, b, c) code_var(MyArgs, {, 1, 2, 3}) # Code may start with "," and, if it does, it may contain other top-level commas. args_code(MyArgs, 1, 2, 3) # Equivalent to the above. append_code(MyArgs, {, 4}) with_args({args_var(NewArgs, arg(1) args(3..))}, MyArgs) # var(NewArgs, {, 1, 3, 4}) args_var(MyArgs, ", 1, 2, 3") # Must be empty or begin with ","; legal, balanced code delimited by ",". # Only code_snippet can take incomplete code arg. call(MyFn MyArgs()) # MyFn(1, 2, 3) code(CallFoo, to_code("call(foo" $MyArgs ")")) CallFoo() # foo(1, 2, 3) call(arg(0), shift(2)) # args(3..) == shift(2)

Somewhere… a code block that represents one or more arguments must begin with `,`. An empty code block may represent zero arguments.

`@` begins a reference to zero or more numbered parameters. A reference to a negative number of parameters results in an error. Some syntaxes refer specifically to one parameter. These may appear anywhere in the code, thus they may contribute to part of an argument. Syntaxes which may refer to multiple parameters must produce a discrete number of arguments must be placed where an argument would be placed, preceded and followed by argument delimitation: `,`/`(` and `,`/`)` or in a code block which will begin with `,`. Though it appears as as an argument, it may result in zero arguments, where a preceding or following comma is ignored, or, if delimited by `(`/`)`, it may result in zero arguments.

`@1` refers to the first unnamed parameter (which must exist), `@15` the 15th.

`@2?` refers to the second unnamed parameter or "" if it doesn't exist.

`@3..` refers to all arguments from the third, or none, in which case the preceding `,` is ignored.

`@1..3` is equivalent to `@1, @2, @3`.

`@1..3?` is equivalent to `@1?, @2?, @3?`.

`@1..-1` is arguments @1 up to the one before the last.

`@(…)` evaluates the parenthetical code and interprets the results as the text after `@` in cases above.

`@foo(…)` evaluates the resulting code, which must begin with a comma argument separator or be empty. This comma will be ignored or an empty result represents zero arguments.

`@#` is the number of arguments.

3.9. Functions

Functions are macros that evaluate in their own scope.

```
fn()
```

3.10. 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
```

3.10.1. Parameters

3.10.1.1. 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.

3.10.1.2. 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.

    - `<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.)

3.10.2. 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.

3.10.3. 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.)

3.10.4. 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']
```

3.10.5. 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.

3.10.6. 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)
```

3.10.7. 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.

3.10.8. 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.

```
// Evaluate a body if a value is negative.
fn(if_neg, Value, Body, {
   var(Neg, m5_calc(Value < 0))
   ~if(Neg, [
      /~eval(m5_Body)
      on_return(Body)
   ])
   return_status(if(Neg, [''], else))
})

fn(my_fn, {
   var(Neg, [''])
   return_status(['pos'])
   ~if_neg(1, [
      return_status(['neg'])
      set(Neg, ['-'])
   ])
   ...
})
```

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

3.10.9. 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())
   ])
   ...
})
```

3.11. Block Labels

Code can be nested within several layers of code blocks. Generally, code evaluates sequentially within its block, but it can be useful to pre-evaluate code. Block labels provide convenient syntax for this.

In the simple example below, `h` is a label for the code block.

```
code(hello, h:{
   "Hello, " h:$Name "." |
})
```

The label is used within the block as `h:$Name`, providing scope to the variable `$Name`. `h:$Name` takes on the value of `Name` outside of the block labeled `h`. It evaluates when the `h` block is parsed. The above is equivalent to:

```
code(hello, {
   "Hello, "} $Name {"."
})
```

but the use of labels allows us to align `{/}` pairs more naturally (and legally, see [xxx]). Also, consider the modification without the use of a label:

```
code(hello, {
   if($Cond, {
      "Hello, "} $Name {"."
   }
})
```

This modification would no longer evaluate `$Name` at the time of the code definition, whereas the labeled reference is more explicit and avoids this easy mistake. Note that the following would, in concept, fix this bug:

```
code(hello, {
   if($Cond, {
      "Hello, "}} $Name {{"."
   }
})
```

though this is not permitted due to the apparent mismatching parentheses and for its lack of readability. (See [xxx].) The correct code, using a label, is:

```
code(hello, h:{
   "Hello, " h:$Name "."
})
```

Labels may be associated with calls (`label:foo()`) and variables (`label:$Foo`).

```
label:foo(a1)          # Evaluates using the definition of `foo` at the time the
labeled block is parsed.
label:(foo(a1)...)     # Evaluates `call(foo, a1)...` at the time the labeled block is
parsed. (Continue applying the label
                       # through the argument list.) Maybe, no. Just require `var(tmp,
foo(a1)...)` outside, then `label:$tmp`??
```

3.12. String Labels

String labels use a similar syntax to Block Labels, but they serve a different purpose. They have no impact functionally. Instead they are used to convey to an editor or IDE the syntax used in a multiline string. For example, if Tilde is being used to process Python code, the following might be used:

```
set(MyPythonCode, python:<">
  def foo():
    print("Hello, world!")
<">)
```

An editor that recognizes the `python:` label might provide appropriate syntax highlighting for the Python code.

3.13. Syntax Configuration

4. Writing Tilde Code

4.1. Code Formatting

4.2. Unquoted Character Restrictions

There are restrictions on the use of unquoted characters in arguments. This affords greater flexibility for syntax configuration. It also encourages quoting to avoid missing quotes around special characters.

The restrictions are different for argument text and arithmetic expressions.

4.2.1. Unquoted Argument Characters

Unquoted text is only permitted in argument text as a string of non-whitespace characters without other statements. This string of characters may be referred to as a "token".

Tokens may include:

- word characters: `a-z`, `A-Z`, `0-9`, `_`

- some symbols: `~^&*-+.?`

Tokens may not include:

- whitespace
- comma (`,`)
- paired characters: `()[]{}<>`
- quote characters: `"'\``
- certain symbols that have other significance by default or that may be configured to have other significance: `!#@%=:;$/\\|`

4.2.2. Unquoted Arithmetic Expressions Characters

In arithmetic expressions, unquoted characters are free to mingle with other statements.

Permitted characters include:

- whitespace (which is ignored and irrelevant to `calc(…)`)
- digits: `0-9`
- symbols: `+-*/!=<>%&|`

And these cannot be used as unquoted literal characters:

- word characters: `a-z`, `A-Z`, `_`
- paired characters: `()[]{}`
- quote characters: `"'\``
- symbols: `~$#@^:;\\.?`

`|` is not treated as a newline character in arithmetic expressions.

4.3. Constructing Code

Though rarely needed, code may be constructed from strings and converted to code using the `compile` macro. For example this modifies the above example to call either `error`, `warning`, or `info` based on the value of `ErrorLevel` which holds one of those three strings. (Note, there are other ways to accomplish this.):

```
var(ErrorLevel, "error")  # or "warning" or "info"
...
var(BadCodeString, $ErrorLevel ("Ugh!"))
    # E.g. error("Ugh!")
var(BadCode, compile($BadCodeString))
    # E.g. {error("Ugh!")}
if($Okay, {doit()}, $BadCode)
```

Code blocks can most directly be evaluated using the `eval` macro. For example:

```
eval($BadCode)   # Reports "Ugh!".
```

5. Standard Library Reference

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

5.1.1. 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.

6. Macro Library

This section documents the macros defined by the Tilde 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.

6.1. 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.

6.2. Assigning and Accessing Macros/Variables

6.2.1. 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

6.2.2. 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(…)`

6.2.3. 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

6.3. Code Constructs

6.3.1. 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()`

6.3.2. 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(⋯)

6.3.3. 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(…)`

6.3.4. 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(…)`

6.4. Working with Strings

6.4.1. 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.
```

6.4.2. 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. Tilde 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/Tilde-characters that have no special treatment in Tilde. 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

6.4.3. 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
```

6.4.4. 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`

6.4.5. 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

6.4.6. Regular Expressions

Regular expressions in Tilde 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 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(…)`, `regex_eval(…)`, `if_regex(…)`, `else_if_regex(…)`

6.5. Utilities

6.5.1. Fundamental Macros

`defn(Name)`

Output: the M4 definition of a macro; note that the M4 definition is slightly different from the Tilde 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. `$@`, thus turning multiple arguments into a single literal string.

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. `…`:

6.5.2. 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
```

6.5.3. 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)
```

6.5.4. 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(…)`

6.5.5. 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.)'])
```

6.5.6. 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`, Returning Status, Aftermath

`on_return(…, D, MacroName, …)`

Parameter(s):
1. `…`
2. `D`: Call a macro upon returning from a function. Arguments are those for m5_call.
3. `MacroName`: the name of the macro to call
4. `…`: its arguments

6.6. 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)
```

6.6.1. 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, $#)
```

6.6.2. 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, $@)
```

7. Reference Card

Tilde processes the following syntaxes:

Table 1. Core Syntax

Feature	**Reference**	**Syntax**	**Contexts**	**Must Be Evaluated?**
Tilde 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

Feature	Reference	Syntax	Contexts	Must Be Evaluated?
Escapes	[prefix_escapes]	`\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)	Tilde, code
Text block quotes	[Text Blocks]	`['`, `']` (ending/beginning a line)	Tilde, code
Evaluate Blocks	[Evaluate Blocks]	`*[`, `[`, `*{`, `}`, `*['`, `']`	Tilde, code
Statement comment	[statement_comments]	`/Blah blah blah…`	Code
Statement with no output	Code Blocks	`Foo`, `bar(…)` (`m5_` prefix implied)	Code
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>['`	Tilde, Code	Yes
Quote escape	[block_labels]	`']<foo>m5_Bar['`	All (within any type of Tilde 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

A

abbreviate_args, 60
append_macro, 38
append_var, 38
arg_comma, 37
argn, 51
assert, 59

C

calc, 52
call, 47
case, 34
close_quote, 37
comma_args, 51
comma_fn_args, 56
comma_shift, 50
comment, 49

D

DEBUG, 58
DEBUG_if, 59
debug_level, 58
decrement, 54
defn, 47
depth_of, 49
dequote, 43

E

echo_args, 51
else, 33
else_if, 31
else_if_def, 33
else_if_regex, 46
eq, 55
equate, 54
error, 58
error_if, 59
errprint, 58
errprint_nl, 58
eval, 48

F

fatal_assert, 59
fatal_error, 58
fatal_error_if, 59
fn, 27
fn_arg, 56
fn_arg_cnt, 56
fn_args, 56
for, 35
for_each_line, 43
for_each_regex, 46
format_eval, 41

G

get, 29
get_ago, 49

I

if, 30
if_defined_as, 32
if_eq, 31
if_neq, 31
if_null, 32
if_regex, 46
if_so, 33
if_var_def, 32
if_var_ndef, 32
increment, 54
index_of, 43
is_null, 55
isnt_null, 55

J

join, 39

L

lazy_fn, 27
length, 43
loop, 34
lowercase, 40

M

macro, 28
must_exist, 29

N

nargs, 50
neq, 55
nl, 37
no_quotes, 44
nquote, 48
null_macro, 28
null_vars, 27
nullify, 49
num_lines, 43

O

on_return, 57
open_quote, 37
operate_on, 54
orig_close_quote, 37
orig_open_quote, 37
out, 57
out_eval, 57
output_with_restored_quotes, 44

P

pop, 27
prepend_macro, 38
prepend_var, 38
printable_close_quote, 37
printable_open_quote, 37
push_macro, 28
push_var, 26

Q

quote, 47

R

recurse, 36
recursion_limit, 60
regex, 44
regex_eval, 44
repeat, 35
replicate, 41
requote, 43
reset_sticky_status, 30
return_status, 57

S

set, 26
set_macro, 28
shift, 50
status, 29
sticky_status, 30, 30
strip_trailing_whitespace_from, 41
substr, 38
substr_eval, 38

T

translit, 40
translit_eval, 40

U

UNDEFINED, 38
unless, 30
uppercase, 40

V

var, 26
var_must_exist, 29
var_regex, 45
verify_min_args, 59
verify_min_max_args, 59
verify_num_args, 59

W

warning, 58
warning_if, 59