Harlowe 2.1.0 manual

1.2.4 | **2.1.0**

Introduction

Passage markup

basics

coding

extra

list

section

- Horizontal rule

whitespace

- Whitespace
- Collapsing whitespace
- Escaped line break

List of macros

basics

- (set:) Instant
- (put:) Instant
- (move:) Instant
- (print:) Command
- (display:) Command
- (if:) Changer
- (unless:) Changer
- (else-if:) Changer
- (else:) Changer
- (for:) Changer

  (loop:)

- (either:) Any
- (enchant:) Command

colour

- (hsl:) Colour
- (hsla:) Colour
- (rgb:) Colour
- (rgba:) Colour

data structure

- (a:) Array

  (array:)

- (dm:) Datamap

  (datamap:)

- (ds:) Dataset

(dataset:)

- (all-pass:) Boolean
- (altered:) Array
- (count:) Number
- (dataentries:) Array
- (datanames:) Array
- (datavalues:) Array
- (find:) Array
- (folded:) Any
- (interlaced:) Array
- (none-pass:) Boolean
- (range:) Array
- (repeated:) Array
- (rotated:) Array
- (shuffled:) Array
- (some-pass:) Boolean
- (sorted:) Array

date and time

- (current-date:) String
- (current-time:) String
- (monthday:) Number
- (weekday:) String

game state

- (history:) Array
- (passage:) Datamap

links

- (link:) Changer

(link-replace:)

- (link-reveal:) Changer
- (link-repeat:) Changer
- (link-goto:) Command
- (click:) Changer
- (link-undo:) Command
- (click-replace:) Changer
- (click-append:) Changer
- (click-prepend:) Changer

- (mouseover:) Changer
- (mouseover-replace:) Changer
- (mouseover-append:) Changer
- (mouseover-prepend:) Changer
- (mouseout:) Changer
- (mouseout-replace:) Changer
- (mouseout-append:) Changer
- (mouseout-prepend:) Changer
- (go-to:) Command
- (undo:) Command

live

- (live:) Changer
- (stop:) Command

maths

- (abs:) Number
- (cos:) Number
- (exp:) Number
- (log:) Number
- (log10:) Number
- (log2:) Number
- (max:) Number
- (min:) Number
- (pow:) Number
- (sign:) Number
- (sin:) Number
- (sqrt:) Number
- (tan:) Number

number

- (ceil:) Number
- (floor:) Number
- (num:) Number

  (number:)

- (random:) Number
- (round:) Number

popup

- (alert:) Command
- (confirm:) Boolean
- (prompt:) String

revision

- (append:) Changer
- (prepend:) Changer
- (replace:) Changer

saving

- (load-game:) Command
- (save-game:) Boolean
- (saved-games:) Datamap

showing and hiding

- (hidden:) Changer
- (show:) Command

string

- (lowercase:) String
- (lowerfirst:) String
- (text:) String

  (string:)

- (uppercase:) String
- (upperfirst:) String
- (words:) Array

styling

- (align:) Changer
- (background:) Changer
- (css:) Changer
- (font:) Changer
- (hook:) Changer
- (hover-style:) Changer
- (text-colour:) Changer

  (colour:), (text-color:), (color:)

- (text-rotate:) Changer
- (text-style:) Changer
- (transition-time:) Changer

  (t8n-time:)

- (transition:) Changer

  (t8n:)

url

- (goto-url:) Command
- (open-url:) Command
- (page-url:) String
- (reload:) Command

deprecated

- (substring:) String
- (subarray:) Array

Types of data

- Any
- Array
- Boolean
- Changer
- Colour
- Command
- Datamap
- Dataset
- HookName
- Instant
- Lambda
- Number
- String
- VariableToValue

Special keywords

- it
- time

Introduction

Report bugs and suggest features

If you think you've found a bug to report in Harlowe, want to make a feature suggestion, or wish to see what future features are already planned, simply visit the project's issues page.

Some of what's new in 2.0

A lot of features have been added to Harlowe 2.0, many of which are designed to shorten existing code idioms or make certain workarounds unnecessary. The changes to existing features you should first familiarise yourself with are:

- The default Harlowe colour scheme is now white text on black, in adherence to SugarCube and Sugarcane. You can change it back to white using the instructions below.
- Expressions like `$a < 4 and 5` will now be interpreted as `$a < 4 and it < 5` instead of always producing an error.
- Using `is` with comparison operators, like `$a is < 3`, is now valid.

- Changers can be attached to hooks with whitespace between them - `(if: $coverBlown) [Run!]` is now valid.
- Changers can be attached to named hooks - `(if: true) |moths>[Several moths!]` is now valid.
- Changers can be added together using + while attaching them to a hook - `(font:'Shatter')+(text-style:'outline')[CRASH!]` is now valid.
- The default CSS has been changed such that the story's `font` must be overridden on `tw-story` rather than `html` (for consistency with other CSS properties).

The following new features also deserve your attention.

- The built-in `?page`, `?passage`, `?sidebar` and `?link` hooks
- Hidden hooks, and the (show:) and (hidden:) command macros
- Temp variables (see the (set:) article)
- The special `any` and `all` data names for arrays, strings and datasets (see each type's articles)
- The (for:) changer macro
- The (enchant:) command macro
- The (find:), (altered:) and (folded:) data macros
- The (dm:) and (ds:) aliases for (datamap:) and (dataset:)
- Column markup
- `tw-passage` elements now have a `tags` attribute.

For a complete list of changes, consult the change log section.

Changing back from dark to light:

You may want to use the black-on-white colour scheme of Harlowe 1 instead of the new white-on-black colour scheme. A few of the new features described above can help you do this without using CSS! Simply create a `header` tagged passage (a passage with the tag 'header'), and include this in it:

```
(enchant: ?page, (text-colour: black) + (background: white))
```

This uses the new ?page built-in hook to target the entire page, and the new (enchant:) macro to apply changer commands to it directly. In the future, more features are planned that will allow styling the page in this way without CSS, staying within Harlowe code, and letting you use variables and other macros inside it.

Passage markup
Link markup

Hyperlinks are the player's means of moving between passages and affecting the story. They consist of *link text*, which the player clicks on, and a *passage name* to send the player to.

Inside matching non-nesting pairs of `[[` and `]]`, place the link text and the passage name, separated by either `->` or `<-`, with the arrow pointing to the passage name.

You can also write a shorthand form, where there is no `<-` or `->` separator. The entire content is treated as a passage name, and its evaluation is treated as the link text.

Example usage:

```
[[Go to the cellar->Cellar]] is a link that goes to a passage named
"Cellar".
[[Parachuting<-Jump]] is a link that goes to a passage named
"Parachuting".
[[Down the hatch]] is a link that goes to a passage named "Down the
hatch".
```

Details:

The interior of a link (the text between `[[` and `]]`) may contain any character except `]`. If additional `->`s or `<-`s appear, the rightmost right arrow or leftmost left arrow is regarded as the canonical separator.

```
[[A->B->C->D->E]] has a link text of
A->B->C->D
and a passage name of
E
```

```
[[A<-B<-C<-D<-E]] has a link text of
B<-C<-D<-E
and a passage name of
A
```

This syntax is not the only way to create links â€“ there are many link macros, such as (link:), which can be used to make more versatile hyperlinks in your story.

Style markup

Often, you'd like to apply styles to your text â€“ to italicize a book title, for example. You can do this with simple formatting codes that are similar to the double brackets of a link. Here is what's available to you:

Styling	Markup code	Result	HTML produced
Italics	`//text//`	*text*	`<i>text</i>`
Boldface	`''text''`	**text**	`<b>text</b>`

Styling	Markup code	Result	HTML produced
Strikethrough text	`~~text~~`	~~text~~	`<s>text</s>`
Emphasis	`*text*`	*text*	`<em>text</em>`
Strong emphasis	`**text**`	**text**	`<strong>text</strong>`
Superscript	`meters/second^^2^^`	meters/second²	`meters/second<sup>2</sup>`

Example usage:

```
You //can't// be serious! I have to go through the ''whole game''
again? ^^Jeez, louise!^^
```

Details:

You can nest these codes - `''//text//''` will produce ***bold italics*** - but they must nest symmetrically. `''//text''//` will not work.

A larger variety of text styles can be produced by using the (text-style:) macro, attaching it to a text hook you'd like to style. And, furthermore, you can use HTML tags like `<mark>` as an additional styling option.

Macro markup

A macro is a piece of code that is inserted into passage text. Macros are used to accomplish many effects, such as altering the game's state, displaying different text depending on the game's state, and altering the manner in which text is displayed.

There are many built-in macros in Harlowe. To use one, you must *call* upon it in your passage by writing the name, a colon, and some data values to provide it, all in parentheses. For instance, you call the (print:) macro like so: `(print: 54)`. In this example, `print` is the macro's name, and `54` is the value.

The name of the macro is case-insensitive, dash-insensitive and underscore-insensitive. This means that any combination of case, dashes and underscores in the name will be ignored. You can, for instance, write `(go-to:)` as `(goto:)`, `(Goto:)`, `(GOTO:)`, `(GoTo:)`, `(Go_To:)`, `(Got--o:)`, `(-_-_g-o-t-o:)`, or any other combination or variation.

You can provide any type of data values to a macro call - numbers, strings, booleans, and so forth. These can be in any form, as well - `"Red" + "belly"` is an expression that produces a single string, "Redbelly", and can be used anywhere that the joined string can be used. Variables, too, can be used with macros, if their contents matches what the macro expects. So, if `$var` contains the string "Redbelly", then `(print: $var)`, `(print: "Redbelly")` and `(print: "Red" + "belly")` are exactly the same.

Furthermore, each macro call produces a value itself - (num:), for instance, produces a number, (a:) an array - so they too can be nested inside other macro calls. `(if: (num:"5") > 2)` nests the (num:) macro inside the (if:) macro.

If a macro can or should be given multiple values, separate them with commas. You can give the `(a:)` macro three numbers like so: `(a: 2, 3, 4)`. The final value may have a comma after it, or it may not - `(a: 2, 3, 4,)` is equally valid. Also, if you have a data value that's an array, string or dataset, you can "spread out" all of its values into the macro call by using the `...` operator: `(either: ...$array)` will act as if every value in $array was placed in the (either:) macro call separately

Variable markup

As described in the documentation for the (set:) macro, variables are used to remember data values in your game, keep track of the player's status, and so forth. They start with `$` (for normal variables) or `_` (for temp variables, which only exist inside a single passage, hook or lambda).

Due to this syntax potentially conflicting with dollar values (such as $1.50) in your story text, variables cannot begin with a numeral.

You can print the contents of variables, or any further items within them, using the (print:) and (for:) macros. Or, if you only want to print a single variable, you can just enter the variable's name directly in your passage's prose.

```
Your beloved plushie, $plushieName, awaits you after a long work day.
```

```
You put your _heldItem down and lift it for a snuggle.
```

Furthermore, if the variable contains a changer command, such as that created by (text-style:) and such, then the variable can be attached to a hook to apply the changer to the hook:

```
$robotText[Good golly! Your flesh... it's so soft!]
```

```
_assistantText[Don't touch me, please! I'm ticklish.]
```

Hook markup

A hook is a means of indicating that a specific span of passage prose is special in some way. It essentially consists of text between single `[` and `]` marks. Prose inside a hook can be modified, styled, controlled and analysed in a variety of ways using macros.

A hook by itself, such as `[some text]`, is not very interesting. However, if you attach a macro or a variable to the front, the attached value is used to change the hook in some way, such as hiding it based on the game state, altering the styling of its text, moving its text to elsewhere in the passage.

```
(font: "Courier New")[This is a hook.
```

```
As you can see, this has a macro instance in front of it.]
```

```
This text is outside the hook.
```

The (font:) macro is one of several macros which produces a special styling command, instead of a basic data type like a number or a string. In this case, the command changes the attached hook's font to Courier New, without modifying the other text.

You can save this command to a variable, and then use it repeatedly, like so:

```
(set: $x to (font: "Skia"))
```

```
$x[This text is in Skia.]
```

```
$x[As is this text.]
```

The basic (if:) macro is used by attaching it to a hook, too:

```
(if: $x is 2)[This text is only displayed if $x is 2.]
```

For more information about command macros, consult the descriptions for each of them in turn.

Named hook markup

For a general introduction to hooks, see their respective markup description. Named hooks are a less common type of hook that offer unique benefits. To produce one, instead of attaching a macro, attach a "nametag" to the front or back:

```
[This hook is named 'opener']<opener|
```

```
|s2>[This hook is named 's2']
```

(Hook nametags are supposed to resemble triangular gift box nametags.)

A macro can refer to and alter the text content of a named hook by referring to the hook as if it were a variable. To do this, write the hook's name as if it were a variable, but use the `?` symbol in place of the `$` symbol:

```
[Fie and fuggaboo!]<shout|
```

```
(click: ?shout)[ (replace: ?shout)["Blast and damnation!"] ]
```

The above (click:) and (replace:) macros can remotely refer to and alter the hook using its name. This lets you, for instance, write a section of text full of tiny hooks, and then attach behaviour to them further in the passage:

```
Your [ballroom gown]<c1| is [bright red]<c2| with [silver streaks]<c3|,
and covered in [moonstones]<c4|.

(click: ?c1)[A hand-me-down from your great aunt.]
(click: ?c2)[A garish shade, to your reckoning.]
(click: ?c3)[Only their faint shine keeps them from being seen as grey.]
(click: ?c4)[Dreadfully heavy, they weigh you down and make dancing
arduous.]
```

As you can see, the top sentence remains mostly readable despite the fact that several words have (click:) behaviours assigned to them.

Built in names:

There are four special built-in hook names, ?Page, ?Passage, ?Sidebar and ?Link, which, in addition to selecting named hooks, also affect parts of the page that you can't normally style with macros. They can be styled using the (enchant:) macro.

- `?Page` selects the page element (to be precise, the `<tw-story>` element) and using it with the (background:) macro lets you change the background of the entire page.
- `?Passage` affects just the element that contains the current passage's text (to be precise, the `<tw-passage>` element) and lets you, for instance, change the (text-colour:) or (font:) of all the text, or apply complex (css:) to it.

- `?Sidebar` selects the passage's sidebar containing undo/redo icons (`<tw-sidebar>`). You can style it with styling macros, or use (replace:) or (append:) to insert your own text into it.
- `?Link` selects all of the links (passage links, and those created by (link:) and other macros) in the passage.

(Note that, as mentioned above, if you use these names for your own hooks, such as by creating a named hook like `|passage>[]` , then they will, of course, be included in the selections of these names.)

Hidden hook markup

Hidden hooks are an advanced kind of named hook that can be shown using macros like (show:). For a general introduction to named hooks, see their respective markup description.

There may be hooks whose contained prose you don't want to be visible as soon as the passage appears - a time delay, or the click of a link should be used to show them. You can set a hook to be *hidden* by altering the hook tag syntax - replace the `>` or `<` mark with a parenthesis.

```
|visible>[This hook is visible when the passage loads.]
```

```
|cloaked)[This hook is hidden when the passage loads, and needs a macro
like `(show:?cloaked)` to reveal it.]
```

```
[My commanding officer - a war hero, and a charismatic face for the
military.]<sight|
```

```
[Privately, I despise the man. His vacuous boosterism makes a mockery of
my sacrifices.](thoughts|
```

(You can think of this as being visually similar to the pointed tails of comic speech balloons vs. round, enclosed thought balloons.)

In order to be useful, hidden hooks must have a name, which macros like (show:) can use to show them. Hence, there's no way to make a hidden unnamed hook - at least, without using a conditional macro like (if:).

HTML markup

If you are familiar with them, HTML tags (like `<img>`) and HTML elements (like `&sect;`) can be inserted straight into your passage text. They are treated very naively - they essentially pass through Harlowe's markup-to-HTML conversion process untouched.

Example usage:

```
<mark>This is marked text.
```

```
&para; So is this.
```

```
And this.</mark>
```

Details:

HTML elements included in this manner are given a `data-raw` attribute by Harlowe, to distinguish them from elements created via markup.

You can include a `<script>` tag in your passage to run Javascript code. The code will run as soon as the containing passage code is rendered.

You can also include a `<style>` tag containing CSS code. The CSS should affect the entire page until the element is removed from the DOM.

Finally, you can also include HTML comments `<!-- Comment -->` in your code, if you wish to leave reminder messages or explanations about the passage's code to yourself.

Verbatim markup

As plenty of symbols have special uses in Harlowe, you may wonder how you can use them normally, as mere symbols, without invoking their special functionality. You can do this by placing them between a pair of `` ` `` marks.

If you want to escape a section of text which already contains single `` ` `` marks, simply increase the number of `` ` `` marks used to enclose them.

Example usage:

- ``` I want to include `[[double square brackets]]` in my story, so I use grave ` marks. ```
- ``` I want to include ``single graves ` in my story``, so I place them between two grave marks. ```

There's no hard limit to the amount of graves you can use to enclose the text.

Bulleted list markup

You can create bullet-point lists in your text by beginning lines with an asterisk `*`, followed by whitespace, followed by the list item text. The asterisk will be replaced with an indented bullet-point. Consecutive lines of bullet-point items will be joined into a single list, with appropriate vertical spacing.

Remember that there must be whitespace between the asterisk and the list item text! Otherwise, this markup will conflict with the emphasis markup.

If you use multiple asterisks (`**`, `***` etc.) for the bullet, you will make a nested list, which is indented deeper than a normal list. Use nested lists for "children" of normal list items.

Example usage:

```
 * Bulleted item
```

```
    *    Bulleted item 2
```

```
  ** Indented bulleted item
```

Numbered list markup

You can create numbered lists in your text, which are similar to bulleted lists, but feature numbers in place of bullets. Simply begin single lines with `0.`, followed by whitespace, followed by the list item text. Consecutive items will be joined into a single list, with appropriate vertical spacing. Each of the `0.`s will be replaced with a number corresponding to the item's position in the list.

Remember that there must be whitespace between the `0.` and the list item text! Otherwise, it will be regarded as a plain number.

If you use multiple `0.` tokens (`0.0.`, `0.0.0.` etc.) for the bullet, you will make a nested list, which uses different numbering from outer lists, and are indented deeper. Use nested lists for "children" of normal list items.

Example usage:

```
0. Numbered item
```

```
   0. Numbered item 2
```

```
 0.0. Indented numbered item
```

Aligner markup

An aligner is a special single-line token which specifies the alignment of the subsequent text. It is essentially 'modal' - all text from the token onward (until another aligner is encountered) is

wrapped in a `<tw-align>` element (or unwrapped in the case of left-alignment, as that is the default).

- Right-alignment, resembling `==>` is produced with 2 or more `=` s followed by a `>` .
- Left-alignment, resembling `<==` is restored with a `<` followed by 2 or more `=` .
- Justified alignment, resembling `<==>` is produced with `<` , 2 or more `=` , and a closing `>` .
- Mixed alignment is 1 or more `=` , then `><` , then 1 or more `=` . The ratio of quantity of left `=` s and right `=` s determines the alignment: for instance, one `=` to the left and three `=` s to the right produces 25% left alignment.

Any amount of whitespace is permitted before or after each token, as long as it is on a single line.

Example usage:

```
==>
This is right-aligned
   =><=
This is centered
  <==>
This is justified
<==
This is left-aligned (undoes the above)
```

```
===><=
This has margins 3/4 left, 1/4 right
   =><=====
This has margins 1/6 left, 5/6 right.
```

Column markup

Column markup is, like aligner markup, a special single-line token which indicates that the subsequent text should be laid out in columns. They consist of a number of `|` marks, indicating the size of the column relative to the other columns - the total width of all columns equals the page width, and this is divided among the columns by their `|` marks. They also have a number of `=` marks surrounding it, indicating the size of the column's margins in CSS "em" units (which are about the width of a capital M).

All text from the token onward, until the next token is encountered, is contained in the specified column. A `|==|` token ends the set of columns and returns the page to normal.

Columns are currently laid out from left to right, in order of appearance.

Any amount of whitespace is permitted before or after each token, as long as it is on a single line.

Example usage:

```
|==
This is in the leftmost column, which has a right margin of about 2
letters wide.
    =|||=
This is in the next column, which has margins of 1 letter wide. It is
three times as wide as the left column.
  =====||
```

```
This is in the right column, which has a right margin of about 5 letters
wide. It is twice as wide as the left column.
    |==|
This text is not in columns, but takes up the entire width, as usual.
```

You can create nested columns by enclosing the inner set of columns in an unnamed hook, like so:

```
|==
This is the outer left column.
==|
This is outer right column.
[\
    |==
This is the inner left column, inside the outer right column.
    ==|
This is the inner right column, inside the outer right column.
\]
```

Heading markup

Heading markup is used to create large headings, such as in structured prose or title splash passages. It is almost the same as the Markdown heading syntax: it starts on a fresh line, has one to six consecutive `f#` s, and ends at the line break.

Example usage:

```
#Level 1 heading renders as an enclosing <h1>
```

```
   ###Level 3 heading renders as an enclosing <h3>
```

```
 ######Level 6 heading renders as an enclosing <h6>
```

As you can see, unlike in Markdown, opening whitespace is permitted before the first #.

Horizontal rule markup

A hr (horizontal rule) is a thin horizontal line across the entire passage. In HTML, it is a `<hr>` element. In Harlowe, it is an entire line consisting of 3 or more consecutive hyphens `-`.

Example usage:

```
        ---
```

```
   ----
```

```
      -----
```

Again, opening whitespace is permitted prior to the first `-` and after the final `-`.

Whitespace markup

"Whitespace" is a term that refers to "space" characters that you use to separate programming code tokens, such as the spacebar space, and the tab character. They are considered interchangeable in type and quantity - using two spaces usually has the same effect as using one space, one tab, and so forth.

Harlowe tries to also recognise most forms of Unicode-defined whitespace, including the quads, the per-em and per-en spaces, but not the zero-width space characters (as they may cause confusion and syntax errors if unnoticed in your code).

Collapsing whitespace markup

When working with macros, HTML tags and such, it's convenient for readability purposes to space and indent the text. However, this whitespace will also appear in the compiled passage text. You can get around this by placing the text between `{` and `}` marks. Inside, all runs of consecutive whitespace (line breaks, spaces) will be reduced to just one space.

Example usage:

```
{
    This sentence
    will be
    (set: $event to true)
    written on one line
    with only single spaces.
}
```

Details:

You can nest this markup within itself - `{Good { gumballs!}}` - but the inner pair won't behave any differently as a result of being nested.

Text inside macro calls (in particular, text inside strings provided to macro) will not be collapsed. Neither will text *outputted* by macro calls, either - `{(print:" ")}` will still print all 3 spaces, and `{(display:"Attic")}` will still display all of the whitespace in the "Attic" passage.

Also, text inside the verbatim syntax, such as `Thunder` `hound`, will not be collapsed either.

If the markup contains a (replace:) command attached to a hook, the hook will still have its whitespace collapsed, even if it is commanded to replace text outside of the markup.

If you only want to remove specific line breaks, consider the escaped line break markup.

Escaped line break markup

Sometimes, you may want to write an especially long line, potentially containing many macros. This may not be particularly readable in the passage editor, though. One piece of markup that may help you is the `\` mark - placing it just before a line break, or just after it, will cause the line break to be removed from the passage, thus "joining together" the lines.

Example usage:

```
This line\
```

```
and this line
```

```
\and this line, are actually just one line.
```

Details:

There must not be any whitespace between the `\` and the line break. Otherwise, it won't work.

Like most passage text markup, this cannot be used inside a macro call (for instance, `(print: \` `3)`) - but since line breaks between values in macro calls are ignored, this doesn't matter.

List of macros

The (set:) macro

(set: ...*VariableToValue*) → *Instant*

Stores data values in variables.

Example usage:

- `(set: $battlecry to "Save a " + $favouritefood + " for me!")` sets a variable called $battlecry.

- `(set: _dist to $altitude - $enemyAltitude)` sets a temp variable called _dist.

Rationale:

Variables are data storage for your game. You can store data values under special names of your choosing, and refer to them later.

There are two kinds of variables. Normal variables, whose names begin with `$`, persist between passages, and should be used to store data that will be needed throughout the entire game. Temp variables, whose names begin with `_`, only exist inside the hook or passage that they're first (set:), and are forgotten after the hook or passage ends. You should use temp variables if you're writing passage code that mustn't accidentally affect any other passages' variables (by using (set:) on a variable name that someone else was using for something different). This can be essential in collaborative work with other authors working on the same story independently, or when writing code to be used in multiple stories.

Variables have many purposes: keeping track of what the player has accomplished, managing some other state of the story, storing hook styles and changers, and other such things. You can display variables by putting them in passage text, attach them to hooks, and create and change them using the (set:) and (put:) macros.

Details:

In its basic form, a variable is created or changed using `(set:` variable `to` value `)`. You can also set multiple variables in a single (set:) by separating each VariableToValue with commas: `(set: $weapon to 'hands', $armour to 'naked')`, etc.

You can also use `it` in expressions on the right-side of `to`. Much as in other expressions, it's a shorthand for what's on the left side: `(set: $vases to it + 1)` is a shorthand for `(set: $vases to $vases + 1)`.

If the destination isn't something that can be changed - for instance, if you're trying to set a bare value to another value, like `(set: true to 2)` - then an error will be printed. This includes modifying arrays - `(set: (a:2,3)'s 1st to 1)` is also an error.

Due to the variable syntax potentially conflicting with dollar values (such as $1.50) in your story text, variables cannot begin with a numeral.

See also:

(push:), (move:)

The (put:) macro

(put: ...*VariableToValue*) → *Instant*

A left-to-right version of (set:) that requires the word `into` rather than `to`.

Rationale:

This macro has an identical purpose to (set:) - it creates and changes variables. For a basic explanation, see the rationale for (set:).

Almost every programming language has a (set:) construct, and most of these place the variable on the left-hand-side. However, a minority, such as HyperTalk, place the variable on the right. Harlowe allows both to be used, depending on personal preference. (set:) reads as `(set:` variable `to` value `)`, and (put:) reads as `(put:` value `into` variable `)`.

Details:

Just as with (set:), a variable is changed using `(put:` value `into` variable `)`. You can also set multiple variables in a single (put:) by separating each VariableToValue with commas: `(put: 2 into $batteries, 4 into $bottles)`, etc.

`it` can also be used with (put:), but, interestingly, it's used on the right-hand side of the expression: `(put: $eggs + 2 into it)`.

See also:

(set:), (move:)

The (move:) macro

(move: ...*VariableToValue*) → *Instant*

A variant of (put:) that deletes the source value after copying it - in effect moving the value from the source to the destination.

Example usage:

```
(move: $arr's 1st into $var)
```

Rationale:

You'll often use data structures such as arrays or datamaps as storage for values that you'll only use once, such as a list of names to print out. When it comes time to use them, you can remove it from the structure and retrieve it in one go.

Details:

You must use the `into` keyword, like (put:), with this macro. This is because, like (put:), the destination of the value is on the right, whereas the source is on the left.

You can also set multiple variables in a single (move:) by separating each VariableToValue with commas: `(move: $a's 1st into $b, $a's 2nd into $c)`, etc.

If the value you're accessing cannot be removed - for instance, if it's an array's `length` - then an error will be produced.

See also:

(push:), (set:)

The (print:) macro

(print: *Any*) → *Command*

This command prints out any single argument provided to it, as text.

Example usage:

```
(print: $var + "s")
```

Details:

It is capable of printing things which (text:) cannot convert to a string, such as changer commands - but these will usually become bare descriptive text like `[A (font: )` `command]`. You may find this useful for debugging purposes.

This command can be stored in a variable instead of being performed immediately. Notably, the expression to print is stored inside the command, instead of being re-evaluated when it is finally performed. So, a passage that contains:

```
(set: $name to "Dracula")
(set: $p to (print: "Count " + $name))
(set: $name to "Alucard")
$p
```

will still result in the text `Count Dracula`. This is not particularly useful compared to just setting `$p` to a string, but is available nonetheless.

See also:

(text:), (display:)

The (display:) macro

(display: *String*) → *Command*

This command writes out the contents of the passage with the given string name. If a passage of that name does not exist, this produces an error.

Example usage:

`(display: "Cellar")` prints the contents of the passage named "Cellar".

Rationale:

Suppose you have a section of code or source that you need to include in several different passages. It could be a status display, or a few lines of descriptive text. Instead of manually copy-pasting it into each passage, consider placing it all by itself in another passage, and using (display:) to place it in every passage. This gives you a lot of flexibility: you can, for instance, change the code throughout the story by just editing the displayed passage.

Details:

Text-targeting macros (such as (replace:)) inside the displayed passage will affect the text and hooks in the outer passage that occur earlier than the (display:) command. For instance, if passage A contains `(replace:"Prince")[Frog]`, then another passage containing `Princes(display:'A')` will result in the text `Frogs`.

Like all commands, this can be set into a variable. It's not particularly useful in that state, but you can use that variable in place of that command, such as writing `$var` in place of `(display: "Yggdrasil")`.

The (if:) macro

(if: *Boolean*) → *Changer*

This macro accepts only booleans, and produces a command that can be attached to hooks to hide them "if" the value was false.

Example usage:

`(if: $legs is 8)[You're a spider!]` will show the `You're a spider!` hook if `$legs` is `8`. Otherwise, it is not run.

Rationale:

In a story with multiple paths or threads, where certain events could occur or not occur, it's common to want to run a slightly modified version of a passage reflecting the current state of the world. The (if:), (unless:), (else-if:) and (else:) macros let these modifications be switched on or off depending on variables, comparisons or calculations of your choosing.

Details:

Note that the (if:) macro only runs once, when the passage or hook containing it is rendered. Any future change to the condition (such as a (link:) containing a (set:) that changes a variable) won't cause it to "re-run", and show/hide the hook anew.

However, if you attach (if:) to a named hook, and the (if:) hides the hook, you can manually reveal the hook later in the passage (such as, after a (link:) has been clicked) by using the (show:) macro to target the hook. Named hooks hidden with (if:) are thus equivalent to hidden named hooks like `|this)[]`.

Alternatives:

The (if:) and (hidden:) macros are not the only attachment that can hide or show hooks! In fact, a variable that contains a boolean can be used in its place. For example:

```
(set: $isAWizard to $foundWand and $foundHat and $foundBeard)
```

```
$isAWizard[You wring out your beard with a quick twisting spell.]
```

```
You step into the ruined library.
```

```
$isAWizard[The familiar scent of stale parchment comforts you.]
```

By storing a boolean inside `$isAWizard`, it can be used repeatedly throughout the story to hide or show hooks as you please.

See also:

(unless:), (else-if:), (else:), (hidden:)

The (unless:) macro

(unless: *Boolean*) → *Changer*

This macro is the negated form of (if:): it accepts only booleans, and returns a command that can be attached hooks to hide them "if" the value was true.

For more information, see the documentation of (if:).

The (else-if:) macro

(else-if: *Boolean*) → *Changer*

This macro's result changes depending on whether the previous hook in the passage was shown or hidden. If the previous hook was shown, then this command hides the attached hook. Otherwise, it acts like (if:), showing the attached hook if it's true, and hiding it if it's false. If there was no preceding hook before this, then an error message will be printed.

Example usage:

```
Your stomach makes {
```

```
(if: $size is 'giant')[
```

```
    an intimidating rumble!
```

```
](else-if: $size is 'big')[
```

```
    a loud growl
```

```
](else:â€‹)[
      a faint gurgle
]}.
```

Rationale:

If you use the (if:) macro, you may find you commonly use it in forked branches of source: places where only one of a set of hooks should be displayed. In order to make this so, you would have to phrase your (if:) expressions as "if A happened", "if A didn't happen and B happened", "if A and B didn't happen and C happened", and so forth, in that order.

The (else-if:) and (else:) macros are convenient variants of (if:) designed to make this easier: you can merely say "if A happened", "else, if B happened", "else, if C happened" in your code.

Details:

Just like the (if:) macro, (else-if:) only checks its condition once, when the passage or hook contaning it is rendered.

The (else-if:) and (else:) macros do not need to only be paired with (if:)! You can use (else-if:) and (else:) in conjunction with boolean variables, like so:

```
$married[You hope this warrior will someday find the sort of love you
know.]
(else-if: not $date)[You hope this warrior isn't doing anything this
Sunday (because
you've got overtime on Saturday.)]
```

If you attach (else-if:) to a named hook, and the (else-if:) hides the hook, you can reveal the hook later in the passage by using the (show:) macro to target the hook.

See also:

(if:), (unless:), (else:), (hidden:)

The (else:) macro

(else:) → *Changer*

This is a convenient limited variant of the (else-if:) macro. It will simply show the attached hook if the preceding hook was hidden, and hide it otherwise. If there was no preceding hook before this, then an error message will be printed.

Rationale:

After you've written a series of hooks guarded by (if:) and (else-if:), you'll often have one final branch to show, when none of the above have been shown. (else:) is the "none of the above" variant of (else-if:), which needs no boolean expression to be provided. It's essentially the same as `(else-if: true)`, but shorter and more readable.

For more information, see the documentation of (else-if:).

Notes:

Just like the (if:) macro, (else:) only checks its condition once, when the passage or hook contaning it is rendered.

Due to a mysterious quirk, it's possible to use multiple (else:) macro calls in succession:

```
$isUtterlyEvil[You suddenly grip their ankles and spread your warm smile
into a searing smirk.]
(else:â€‹)[In silence, you gently, reverently rub their soles.]
(else:â€‹)[Before they can react, you unleash a typhoon of tickles!]
(else:â€‹)[They sigh contentedly, filling your pious heart with joy.]
```

This usage can result in a somewhat puzzling passage source structure, where each (else:) hook alternates between visible and hidden depending on the first such hook. So, it is best avoided.

If you attach (else:) to a named hook, and the (else:) hides the hook, you can reveal the hook later in the passage by using the (show:) macro to target the hook.

See also:

(if:), (unless:), (else-if:), (hidden:)

The (for:) macro

(for: *Lambda*, ...*Any*) → *Changer*

Also known as: (loop:)

A command that repeats the attached hook, setting a temporary variable to a different value on each repeat.

Example usage:

- `(for: each _item, ...$arr) [You have the _item.]` prints "You have the " and the item, for each item in $arr.
- `(for: _ingredient where it contains "petal", ...$reagents) [Cook the _ingredient?]` prints "Cook the " and the string, for each string in $reagents which contains "petal".

Rationale:

Suppose you're using arrays to store strings representing inventory items, or character datamaps, or other kinds of sequential game information - or even just built-in arrays like (history:) - and you want to print out a sentence or paragraph for each item. The (for:) macro can be used to print something "for each" item in an array easily - simply write a hook using a temp variable where each item should be printed or used, then give (for:) an "each" lambda that uses the same temp variable.

Details:

Don't make the mistake of believing you can alter an array by trying to (set:) the temp variable in each loop - such as `(for: each _a, ...$arr)[(set: _a to it + 1)]`. This will NOT change $arr - only the temp variable will change (and only until the next loop, where another $arr value will be put into it). If you want to alter an array item-by-item, use the (altered:) macro.

The temp variable inside the hook will shadow any other identically-named temp variables outside of it: if you `(set: _a to 1)`, then `(for: each _a, 2,3)[ (print: _a) ]`, the inner hook will print "2" and "3", and you won't be able to print or set the "outer" _a.

You may want to simply print several copies of a hook a certain number of times, without any particular array data being looped over. You can use the (range:) macro with it instead: `(for: each _i, ...(range:1,10))`, and not use the temp variable inside the hook at all.

As it is a changer macro, (for:)'s value is a changer command which can be stored in a variable - this command stores all of the values originally given to it, and won't reflect any changes to the values, or their container arrays, since then.

Alternatives:

You may be tempted to use (for:) not to print anything at all, but to find values inside arrays using (if:), or form a "total" using (set:). The lambda macros (find:) and (folded:), while slightly less straightforward, are recommended to be used instead.

See also:

(find:), (folded:), (if:)

The (either:) macro

(either: ...*Any*) → *Any*

Give this macro several values, separated by commas, and it will pick and return one of them randomly.

Example usage:

`A (either: "slimy", "goopy", "slippery") puddle` will randomly be "A slimy puddle", "A goopy puddle" or "A slippery puddle".

Rationale:

There are plenty of occasions where you might want random elements in your story: a few random adjectives or flavour text lines to give repeated play-throughs variety, for instance, or a

few random links for a "maze" area. For these cases, you'll probably want to simply select from a few possibilities. The (either:) macro provides this functionality.

Details:

As with many macros, you can use the spread `...` operator to place all of the values in an array or dataset into (either:), and pick them randomly. `(either: ...$array)`, for instance, will choose one possibility from all of the array contents.

If you want to pick two or more values randomly, you may want to use the (shuffled:) macro, and extract a subarray from its result.

See also:

(random:), (shuffled:)

The (enchant:) macro

(enchant: *HookName* or *String*, *Changer*) → *Command*

Applies a changer to every occurrence of a hook or string in a passage, and continues applying that changer to any further occurrences that are made to appear in the same passage later.

Example usage:

- `(enchant: "gold", (text-colour: yellow) + (text-style:'bold'))` makes all occurrences of "gold" in the text be bold and yellow.
- `(enchant: ?dossier, (link: "Click to read"))` makes all the hooks named "dossier" be hidden behind links reading "Click to read".

Rationale:

While changers allow you to style or transform certain hooks in a passage, it can be tedious and error-prone to attach them to every occurrence as you're writing your story, especially if the attached changers are complicated. You can simplify this by storing changers in short variables, and attaching just the variables, like so:

```
(set: _ghost to (text-style:'outline'))
```

```
_ghost[Awoo]
```

```
_ghost[Ooooh]
```

Nevertheless, this can prove undesirable: you may want to remove the _ghost styling later in development, which would force you to remove the attached variables to avoid producing an error; you may want to only style a single word or phrase, and find it inconvenient to place it in a hook; you may simply not like having code, like that (set:) macro, be at the start of your passage; you may not want to keep track of which variables hold which changers, given the possibility (if you're using normal variables) that they could be changed previously in the story.

Instead, you can give the hooks the name "ghost", and then (enchant:) them afterward like so:

```
|ghost>[Awoo]
```

```
|ghost>[Ooooh]
```

```
(enchant: ?ghost, (text-style:'outline'))
```

The final (enchant:) macro can target words instead of hooks, much like (click:) - simply provide a string instead of a hook name.

This macro works well in "header" tagged passages - using a lot of (enchant:) commands to style certain words or parts of every passage, you can essentially write a "styling language" for your story, where certain hook names "mean" certain colours or behaviour. (This is loosely comparable to using CSS to style class names, but exclusively uses macros.)

Details:

As with (click:), the "enchantment" affects the text produced by (display:) macros, and any hooks changed by (replace:) etc. in the future, until the player makes their next turn.

The built-in hook names, ?Page, ?Passage, ?Sidebar and ?Link, can be targeted by this macro, and can be styled on a per-passage basis this way.

See also:

(click:)

The (hsl:) macro

(hsl: *Number*, *Number*, *Number*) → *Colour*

This macro creates a colour using the given hue (h) angle in degrees, as well as the given saturation (s) and lightness (l) percentages.

Example usage:

- `(hsl: 120, 0.8, 0.5)` produces a colour with 120 degree hue, 80% saturation and 50% lightness.
- `(hsl: 28, 1, 0.4)'s h` produces the number 28.

Rationale:

The HSL colour model is regarded as easier to work with than the RGB model used for HTML hexadecimal notation and the (rgb:) macro. Being able to set the hue with one number instead of three, for instance, lets you control the hue using a single variable, and alter it at will.

Details:

This macro takes the same range of numbers as the CSS `hsl()` function.

Giving saturation or lightness values higher than 1 or lower than 0 will cause an error. However, you can give any kind of hue number to (hsl:), and it will automatically round it to fit the 0-359 degree range. This allows you to cycle through hues easily by providing a steadily increasing variable or a counter, such as `(hsl: time / 100, 1, 0.5)`.

See also:

(rgb:), (rgba:), (hsla:)

The (hsla:) macro

(hsla: *Number*, *Number*, *Number*, *Number*) → *Colour*

A special version of (hsl:), this macro allows you to supply not just the hue (h) angle in degrees, saturation (s) and lightness (l) percentages, but also the transparency (alpha, or a) percentage, which is a fractional value between 0 (fully transparent) and 1 (fully visible).

Anything drawn with a partially transparent colour will itself be partially transparent. You can then layer such elements to produce a few interesting visual effects.

Example usage:

```
(hsla: 120, 0.5, 0.8, 0.6)
```
produces a 40% transparent faint green.

Details:

This macro takes the same range of numbers as the CSS `rgba()` function.

Giving alpha percentages higher than 1 or lower than 0 will cause an error.

See also:

(rgb:), (rgba:), (hsl:)

The (rgb:) macro

(rgb: *Number*, *Number*, *Number*) → *Colour*

This macro creates a colour using the three red (r), green (g) and blue (b) values provided, whose values are whole numbers between 0 and 255.

Example usage:

- `(rgb: 255, 0, 47)` produces a colour with 255 red, 0 blue and 47 green.
- `(rgb: 90, 0, 0)'s r` produces the number 90.

Rationale:

The RGB additive colour model is commonly used for defining colours: the HTML hexadecimal notation for colours (such as #9263AA) simply consists of three hexadecimal

values placed together. This macro allows you to create such colours computationally, by providing variables for certain components.

Details:

This macro takes the same range of numbers as the CSS `rgb()` function.

Giving values higher than 255 or lower than 0, or with a fractional part, will cause an error.

See also:

(rgba:), (hsl:), (hsla:)

The (rgba:) macro

(rgba: *Number*, *Number*, *Number*, *Number*) → *Colour*

A special version of (rgb:), this macro allows you to supply not just the red (r), green (g) and blue (b) values, but also the transparency (alpha, or a) percentage, which is a fractional value between 0 (fully transparent) and 1 (fully visible).

Anything drawn with a partially transparent colour will itself be partially transparent. You can then layer such elements to produce a few interesting visual effects.

Example usage:

```
(rgba: 178, 229, 178, 0.6)
```
produces a 40% transparent faint green.

Details:

This macro takes the same range of numbers as the CSS `rgba()` function.

Giving alpha percentages higher than 1 or lower than 0 will cause an error.

See also:

(rgb:), (hsl:), (hsla:)

The (a:) macro

(a: *[...Any]*) → *Array*

Also known as: (array:)

Creates an array, which is an ordered collection of values.

Example usage:

`(a:)` creates an empty array, which could be filled with other values later. `(a: "gold", "frankincense", "myrrh")` creates an array with three strings. This is also a valid array, but with its elements spaced in a way that makes them more readable:

```
(a:
"You didn't sleep in the tiniest bed",
"You never ate the just-right porridge",
"You never sat in the smallest chair",
)
```

Rationale:

For an explanation of what arrays are, see the Array article. This macro is the primary means of creating arrays - simply supply the values to it, in order.

Details:

Note that due to the way the spread `...` operator works, spreading an array into the (a:) macro will accomplish nothing: `(a: ...$array)` is the same as just the `$array`.

See also:

(dm:), (ds:)

The (dm:) macro

(dm: *[...Any]*) → *Datamap*

Also known as: (datamap:)

Creates a datamap, which is a data structure that pairs string names with data values. You should provide a string name, followed by the value paired with it, and then another string name, another value, and so on, for as many as you'd like.

Example usage:

`(dm:)` creates an empty datamap. `(dm: "Cute", 4, "Wit", 7)` creates a datamap with two names and values. The following code also creates a datamap, with the names and values laid out in a readable fashion:

```
(dm:
"Susan", "A petite human in a yellow dress",
"Tina", "A ten-foot lizardoid in a three-piece suit",
"Gertie", "A griffin draped in a flowing cape",
)
```

Rationale:

For an explanation of what datamaps are, see the Datamap article. This macro is the primary means of creating datamaps - simply supply a name, followed by a value, and so on.

In addition to creating datamaps for long-term use, this is also used to create "momentary" datamaps which are used only in some operation. For instance, to add several values to a datamap at once, you can do something like this:

```
(set: $map to it + (dm: "Name 1", "Value 1", "Name 2", "Value 2"))
```

You can also use (dm:) as a kind of "multiple choice" structure, if you combine it with the `'s` or `of` syntax. For instance...

```
(set: $element to $monsterName of (dm:
"Chilltoad", "Ice",
"Rimeswan", "Ice",
"Brisketoid", "Fire",
"Slime", "Water"
))
```

...will set $element to one of those elements if $monsterName matches the correct name. But, be warned: if none of those names matches $monsterName, an error will result.

See also:

(a:), (ds:)

The (ds:) macro

(ds: *[...Any]*) → *Dataset*

Also known as: (dataset:)

Creates a dataset, which is an unordered collection of unique values.

Example usage:

`(ds:)` creates an empty dataset, which could be filled with other values later. `(ds: "gold", "frankincense", "myrrh")` creates a dataset with three strings.

Rationale:

For an explanation of what datasets are, see the Dataset article. This macro is the primary means of creating datasets - simply supply the values to it, in any order you like.

Details:

You can also use this macro to remove duplicate values from an array (though also eliminating the array's order) by using the spread `...` operator like so: `(a: ...(ds: ...$array))`.

See also:

(dm:), (a:)

The (all-pass:) macro

(all-pass: *Lambda*, ...*Any*) → *Boolean*

This takes a "where" lambda and a series of values, and evaluates to true if the lambda, when run using each value, always evaluated to true.

Example usage:

- `(all-pass: _num where _num > 1 and _num < 14, 6, 8, 12, 10, 9)` is true.
- `(all-pass: _room where "Egg" is not in _room's objs, ...$rooms)` is true if each datamap in $rooms doesn't have the string `"Egg"` in its "objs".

Rationale:

While the `contains` and `is in` operators can be used to quickly check if a sequence of values contains an exact value or values, you'll often find yourself wanting to check that the values in a sequence merely resemble a kind of value - for instance, that they're positive numbers, or strings beginning with "E".

The (all-pass:) macro lets you perform these checks easily using a lambda, identical to that used with (find:) - simply write a "temp variable `where` a condition" expression, and every value will be put into the temp variable one by one, and the condition checked for each.

Details:

Of course, if any condition should cause an error, such as checking if a number contains a number, then the error will appear.

The temp variable, which you can name anything you want, is controlled entirely by the lambda - it doesn't exist outside of it, it won't alter identically-named temp variables outside of it, and you can't manually (set:) it within the lambda.

You can refer to other variables, including other temp variables, in the `where` condition. For instance, you can write `(set: _name to "Eva")(all-pass: _item where _item is _name, "Evan", "Eve", "Eva")`. However, for obvious reasons, if the outer temp variable is named the same as the lambda's temp variable, it can't be referred to in the condition.

See also:

(sorted:), (count:), (find:), (some-pass:), (none-pass:)

The (altered:) macro

(altered: *Lambda*, ...*Any*) → *Array*

This takes a "via" lambda and a sequence of values, and creates a new array with the same values in the same order, but altered via the operation in the lambda's "via" clause.

Example usage:

- `(altered: _monster via "Dark " + _monster, "Wolf", "Ape", "Triffid")` produces `(a: "Dark Wolf", "Dark Ape", "Dark Triffid")`
- `(altered: _player via _player + (dm: "HP", _player's HP - 1), ...$players)` produces an array of $players datamaps whose "HP" datavalue is decreased by 1.

Rationale:

Transforming entire arrays or datasets, performing an operation on every item at once, allows arrays to be modified with the same ease that single values can - just as you can add some extra text to a string with a single +, so too can you add extra text to an entire array of strings using a single call to (altered:).

This macro uses a lambda (which is just the "temp variable `via` an expression" expression) to take each item in the sequence and produce a new value to populate the resulting array. For `(altered: _a via _a + 1, 10,20,30)` it will produce 10 + 1, 20 + 1 and 30 + 1, and put those into a new array.

Details:

Of course, if any operation applied to any of the values should cause an error, such as trying to add a string to a number, an error will result.

The temp variable, which you can name anything you want, is controlled entirely by the lambda - it doesn't exist outside of it, it won't alter identically-named temp variables outside of it, and you can't manually (set:) it within the lambda.

You can refer to other variables, including other temp variables, in the `via` expression. For instance, you can write `(altered: _object via _playerName + "'s " + _object, "Glove", "Hat", "Purse")`. However, for obvious reasons, if the outer temp variable is named the same as the lambda's temp variable, it can't be referred to in the expression.

If no values are given to (altered:) except for the lambda, an empty array will be produced.

See also:

(for:), (folded:)

The (count:) macro

(count: *Array* or *String*, ...*Any*) → *Number*

Accepts a string or array, followed by a value, and produces the number of times any of the values are inside the string or array.

Example usage:

`(count: (a:1,2,3,2,1), 1, 2)` produces 4. `(count: "Though", "ugh","u","h")` produces 4.

Rationale:

You can think of this macro as being like the `contains` operator, but more powerful. While `contains` produces `true` or `false` if occurrences of the right side appear in the left side, (count:) produces the actual number of occurrences.

Note that if you only want to check if an array or string contains any or all of the values, it's easier to use `contains` with the `all` property like so: `$arr contains all of (a:1,2)` and `$arr contains any of (a:1,2)`. But, if you need an exact figure for the number of occurrences, this macro will be of use.

Details:

If you use this with a number, boolean, datamap, dataset (which can't have duplicates), or anything else which can't have a value, then an error will result.

If you use this with a string, and the values aren't also strings, then an error will result.

Substrings are counted separately from each other - that is, the string "Though" contains "ugh" once and "h" once, and `(count: "Though","ugh","h")` results in 3. To check for "h" occurrences that are not contained in "ugh", you can try subtracting two (count:)s - `(count: "Though","ugh") - (count: "Though","h")` produces 1.

See also:

(datanames:), (datavalues:)

The (dataentries:) macro

(dataentries: *Datamap*) → *Array*

This takes a datamap, and returns an array of its name/value pairs. Each pair is a datamap that only has "name" and "value" data. The pairs are ordered by their name.

Example usage:

- `(datapairs: (dm:'B',24, 'A',25))` produces the following array: `(a: (dm: "name", "A", "value", 25), (dm: "name", "B", "value", 24))`
- `(altered: _entry via _entry's name + ":" + _entry's value, ...(datapairs: $m))` creates an array of strings from the $m datamap's names and values.

Rationale:

There are occasions where operating on just the names, or the values, of a datamap isn't good enough - you'll want both. Rather than the verbose process of taking the (datanames:) and (datavalues:) arrays and using them (interlaced:) with each other, you can use this macro instead, which allows the name and value of each entry to be referenced using "name" and "value" properties.

See also:

(datanames:), (datavalues:)

The (datanames:) macro

(datanames: *Datamap*) → *Array*

This takes a datamap, and returns a sorted array of its data names, sorted alphabetically.

Example usage:

`(datanames: (dm:'B','Y', 'A','X'))` produces the array `(a: 'A','B')`

Rationale:

Sometimes, you may wish to obtain some information about a datamap. You may want to list all of its data names, or determine how many entries it has. You can use the (datanames:) macro to do these things: if you give it a datamap, it produces a sorted array of all of its names. You can then (print:) them, check the length of the array, obtain a subarray, and other things you can do to arrays.

See also:

(datavalues:), (dataentries:)

The (datavalues:) macro

(datavalues: *Datamap*) → *Array*

This takes a datamap, and returns an array of its values, sorted alphabetically by their name.

Example usage:

```
(datavalues: (dm:'B',24, 'A',25))
```
produces the array
```
(a: 25,24)
```

Rationale:

Sometimes, you may wish to examine the values stored in a datamap without referencing every name - for instance, determining if 0 is one of the values. (This can't be determined using the `contains` keyword, because that only checks the map's data names.) You can extract all of the datamap's values into an array to compare and analyse them using (datavalues:). The values will be sorted by their associated names.

See also:

(datanames:), (dataentries:)

The (find:) macro

(find: *Lambda*, ...*Any*) → *Array*

This searches through the given values, and produces an array of those which match the given search test (which is expressed using a temp variable, the `where` keyword, and a boolean condition). If none match, an empty array is produced.

Example usage:

- `(find: _person where _person is not "Alice", ...$people)` produces a subset of $people not containing the string `"Alice"`.
- `(find: _item where _item's 1st is "A", "Thorn", "Apple", "Cryptid", "Anchor")` produces `(a: "Apple", "Anchor")`.
- `(find: _num where (_num >= 12) and (it % 2 is 0), 9, 10, 11, 12, 13, 14, 15, 16)` produces `(a: 12, 14, 16)`.
- `(find: _val where _val + 2, 9, 10, 11)` produces an error, because `_item + 2` isn't a boolean.
- `1st of (find: _room where _room's objs contains "Egg", ...$rooms)` finds the first datamap in $rooms whose "objs" contains the string `"Egg"`.

Rationale:

Selecting specific data from arrays or sequences based on a user-provided boolean condition is one of the more common and powerful operations in programming. This macro allows you to immediately work with a subset of the array's data, without caring what kind of subset it is. The subset can be based on each string's characters, each datamap's values, each number's evenness or oddness, whether a variable matches it... anything you can write.

This macro uses a lambda (which is just the "temp variable `where` a condition" expression) to check every one of the values given after it. For `(find: _item where _item > 40, 30, 60, 90)`, it will first check if `30 > 40` (which is `false`), if `60 > 40` (which is `true`), and if `90 > 40` (which is `true`), and include in the returned array those values which resulted in `true`.

Details:

Of course, if any condition should cause an error, such as checking if a number contains a number, then the error will appear.

The temp variable, which you can name anything you want, is controlled entirely by the lambda - it doesn't exist outside of it, it won't alter identically-named temp variables outside of it, and you can't manually (set:) it within the lambda.

You can refer to other variables, including other temp variables, in the `where` condition. For instance, you can write `(set: _name to "Eva")(find: _item where _item is _name, "Evan", "Eve", "Eva")`. However, for obvious reasons, if the outer temp variable is named the same as the lambda's temp variable, it can't be referred to in the condition.

There isn't a way to examine the position of a value in the condition - you can't write, say, `(find: _item where _pos % 2 is 0, "A", "B", "C", "D")` to select just "B" and "D".

You shouldn't use this macro to try and alter the given values! Consider the (altered:) or (folded:) macro instead.

See also:

(sorted:), (all-pass:), (some-pass:), (none-pass:)

The (folded:) macro

(folded: *Lambda*, ...*Any*) → *Any*

This takes a "making" lambda and a sequence of values, and creates a new value (the "total") by feeding every value in the sequence to the lambda, akin to folding a long strip of paper into a single square.

Example usage:

- `(folded: _enemy making _allHP via _allHP + _enemy's hp, ...$enemies)` will first set _sum to $enemies's 1st's hp, then add the remaining hp values in $enemies to it.

- `(folded: _name making _allNames via _allNames + "/" + _name, ...(history: ))` will create a string of every passage name in the (history:) array, separated by a forward slash.

Rationale:

The (for:) macro, while intended to display multiple copies of a hook, can also be used to run a single macro call multiple times. You may wish to use this to repeatedly (set:) a variable to itself plus one of the looped values (or some other operation). (folded:) is meant to let you perform this in a shorter, more fluid fashion.

Consider, first of all, a typical (for:) and (set:) loop such as the following:

```
{(set:$allNames to "")
(for: each _name, ...(history: ))[
    (set:$allNames to it + "/" _name)
]}
You've visited: $allNames
```

This can be rewritten using (folded:) as follows. While this version may seem a little harder to read if you're not used to it, it allows you to accomplish the same thing in a single line, by immediately using the macro's provided value without a variable:

```
You've visited: (folded: _name making _allNames via _allNames + "/" +
_name, ...(history: ))
```

This macro uses a lambda (which is the "temp variable `making` another temp variable `via` expression" expression) to run the expression using every provided value, much like those repeated (set:) calls.

If you need to perform this operation at various different times in your story, you may wish to (set:) the lambda into a variable, so that you, for instance, might need only write:

```
You've visited: (folded: $namesWithForwardSlashes, ...(history: ))
```

Details:

Of course, if at any time the expression should cause an error, such as adding a number to a string, then an error will result.

Both of the temp variables, the value and the total, can be named anything you want. As with other lambda macros, they don't exist outside of it, won't alter identically-named temp variables outside of it, and can't be manually (set:) within the lambda.

You can refer to other variables, including other temp variables, in the `via` expression. For instance, you can write `(folded: _score making _totalScore via _totalScore + _score * _bonusMultiplier)`. However, for obvious reasons, if the outer temp variable is named the same as the lambda's temp variables, it can't be referred to in the expression.

You can also use a "where" clause inside the "making" lambda to prevent an operation from occurring if a value isn't suitable - `(folded: _item making _total via _total + _item where _item > 0, ...$arr)` will only sum up the values in $arr which are greater than 0.

See also:

(for:), (altered:)

The (interlaced:) macro

(interlaced: *Array*, ...*Array*) → *Array*

Takes multiple arrays, and pairs up each value in those arrays: it creates an array containing each array's first value followed by each array's second value, and so forth. If some values have no matching pair (i.e. one array is longer than the other) then those values are ignored.

Example usage:

`(interlaced: (a: 'A', 'B', 'C', 'D'), (a: 1, 2, 3))` is the same as `(a: 'A',1,'B',2,'C',3)`

Rationale:

There are a couple of other macros which accept data in pairs - the most notable being (dm:), which takes data names and data values paired. This macro can help with using such macros. For instance, you can supply an array of (datanames:) and (datavalues:) to (interlaced:), and supply that to (dm:), to produce the original datamap again. Or, you can supply just the names, and use a macro like (repeated:) to fill the other values.

However, (interlaced:) can also be of use alongside macros which accept a sequence: you can use it to cleanly insert values between each item. For instance, one can pair an array with another array of spaces, and then convert them to a string with (text:). `(text: ...(interlaced: $arr, (repeated: $arr's length, ' '))` will create a string containing each element of $arr, followed by a space.

Details:

If one of the arrays provided is empty, the resulting array will be empty, as well.

See also:

(a:), (rotated:), (repeated:)

The (none-pass:) macro

(none-pass: *Lambda*, ...*Any*) → *Boolean*

This can be thought of as the opposite of (all-pass:): it produces true if every value, when given to the lambda, evaluated to false. For more information, consult the description of (all-pass:).

The (range:) macro

(range: *Number*, *Number*) → *Array*

Produces an array containing an inclusive range of whole numbers from a to b, in ascending order.

Example usage:

`(range:1,14)` is equivalent to `(a:1,2,3,4,5,6,7,8,9,10,11,12,13,14)`

`(range:2,-2)` is equivalent to `(a:-2,-1,0,1,2)`

Rationale:

This macro is a shorthand for defining an array that contains a sequence of integer values. Rather than writing out all of the numbers, you can simply provide the first and last numbers.

Details:

Certain kinds of macros, like (either:) or (dataset:), accept sequences of values. You can use (range:) with these in conjunction with the `...` spreading operator: `(dataset:` `...(range:2,6))` is equivalent to `(dataset: 2,4,5,6,7)`, and `(either:` `...(range:1,5))` is equivalent to `(random: 1,5)`.

See also:

(a:)

The (repeated:) macro

(repeated: *Number*, ...*Any*) → *Array*

When given a number and a sequence of values, this macro produces an array containing those values repeated, in order, by the given number of times.

Example usage:

- `(repeated: 5, false)` produces `(a: false, false, false, false, false)`
- `(repeated: 3, 1,2,3)` produces `(a: 1,2,3,1,2,3,1,2,3)`

Rationale:

This macro, as well as (range:), are the means by which you can create a large array of similar or regular data, quickly. Just as an example: you want, say, an array of several identical, complex datamaps, each of which are likely to be modified in the game, you can use (repeated:) to make those copies easily. Or, if you want, for instance, a lot of identical strings accompanied by a lone different string, you can use (repeated:) and add a `(a: "string")` to the end.

When you already have an array variable, this is similar to simply adding that variable to itself several times. However, if the number of times is over 5, this can be much simpler to write.

Details:

An error will, of course, be produced if the number given is 0 or less, or contains a fraction.

See also:

(a:), (range:)

The (rotated:) macro

(rotated: *Number*, *[...Any]*) → *Array*

Similar to the (a:) macro, but it also takes a number at the start, and moves each item forward by that number, wrapping back to the start if they pass the end of the array.

Example usage:

- `(rotated: 1, 'A','B','C','D')` is equal to `(a: 'D','A','B','C')`.
- `(rotated: -2, 'A','B','C','D')` is equal to `(a: 'C','D','A','B')`.

Rationale:

Sometimes, you may want to cycle through a number of values, without repeating any until you reach the end. For instance, you may have a rotating set of flavour-text descriptions for a thing in your story, which you'd like displayed in their entirety without the whim of a random picker. The (rotated:) macro allows you to apply this "rotation" to a sequence of data, changing their positions by a certain number without discarding any values.

Remember that, as with all macros, you can insert all the values in an existing array using the `...` syntax: `(set: $a to (rotated: 1, ...$a))` is a common means of replacing an array with a rotation of itself.

Think of the number as being an addition to each position in the original sequence - if it's 1, then the value in position 1 moves to 2, the value in position 2 moves to 3, and so forth.

Incidentally... you can also use this macro to rotate a string's characters, by doing something like this: `(string: ...(rotated: 1, ...$str))`

Details:

To ensure that it's being used correctly, this macro requires three or more items - providing just two, one or none will cause an error to be presented.

See also:

(sorted:)

The (shuffled:) macro

(shuffled: *Any*, *Any*, *[...Any]*) → *Array*

Identical to (a:), except that it randomly rearranges the elements instead of placing them in the given order.

Example usage:

```
(set: $a to (a: 1,2,3,4,5,6))
```

```
(print: (shuffled: ...$a))
```

Rationale:

If you're making a particularly random story, you'll often want to create a 'deck' of random descriptions, elements, etc. that are only used once. That is to say, you'll want to put them in an array, then randomise the array's order, preserving that random order for the duration of a game.

The (either:) macro is useful for selecting an element from an array randomly (if you use the spread `...` syntax), but isn't very helpful for this particular problem. The (shuffled:) macro

is the solution: it takes elements and returns a randomly-ordered array that can be used as you please.

Details:

To ensure that it's being used correctly, this macro requires two or more items - providing just one (or none) will cause an error to be presented.

See also:

(a:), (either:), (rotated:)

The (some-pass:) macro

(some-pass: *Lambda*, ...*Any*) → *Boolean*

This is similar to (all-pass:), but produces true if one or more value, when given to the lambda, evaluated to true. It can be thought of as shorthand for putting `not` in front of (none-pass:).

For more information, consult the description of (all-pass:).

The (sorted:) macro

(sorted: *Number* or *String*, ...*Number or String*) → *Array*

Similar to (a:), except that it requires only numbers or strings, and orders them in English alphanumeric sort order, rather than the order in which they were provided.

Example usage:

```
(set: $a to (a: 'A','C','E','G', 2, 1))
```

```
(print: (sorted: ...$a))
```

Rationale:

Often, you'll be using arrays as 'decks' that will provide values to other parts of your story in a specific order. If you want, for instance, several strings to appear in alphabetical order, this macro can be used to create a sorted array, or (by using the spread `...` syntax) convert an existing array into a sorted one.

Details:

Unlike other programming languages, strings aren't sorted using ASCII sort order, but alphanumeric sorting: the string "A2" will be sorted after "A1" and before "A11". Moreover, if the player's web browser supports internationalisation (that is, every current browser except Safari 6-8 and IE 10), then the strings will be sorted using English language rules (for instance, "Ã©" comes after "e" and before "f", and regardless of the player's computer's language settings. Otherwise, it will sort using ASCII comparison (whereby "Ã©" comes after "z").

Currently there is no way to specify an alternative language locale to sort by, but this is likely to be made available in a future version of Harlowe.

To ensure that it's being used correctly, this macro requires two or more items - providing just one (or none) will cause an error to be presented.

See also:

(a:), (shuffled:), (rotated:)

The (current-date:) macro

(current-date:) → *String*

This date/time macro produces a string of the current date the current player's system clock, in the format "Thu Jan 01 1970".

Example usage:

```
Right now, it's (current-date:).
```

The (current-time:) macro

(current-time:) → *String*

This date/time macro produces a string of the current 12-hour time on the current player's system clock, in the format "12:00 AM".

Example usage:

```
The time is (current-time:).
```

The (monthday:) macro

(monthday:) → *Number*

This date/time macro produces a number corresponding to the day of the month on the current player's system clock. This should be between 1 (on the 1st of the month) and 31, inclusive.

Example usage:

```
Today is day (monthday:).
```

The (weekday:) macro

(weekday:) → *String*

This date/time macro produces one of the strings "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" or "Saturday", based on the weekday on the current player's system clock.

Example usage:

```
Today is a (weekday:).
```

The (history:) macro

(history:) → *Array*

This returns an array containing the string names of all of the passages the player has visited up to now, in the order that the player visited them.

Example usage:

`(history:) contains "Cellar"` is true if the player has visited a passage called "Cellar" at some point.

Rationale:

Often, you may find yourself using "flag" variables to keep track of whether the player has visited a certain passage in the past. You can use (history:), along with data structure operators such as the `contains` operator, to obviate this necessity.

Details:

This includes duplicate names if the player has visited a passage more than once, or visited the same passage two or more turns in a row.

This does *not* include the name of the current passage the player is visiting.

See also:

(passage:), (savedgames:)

The (passage:) macro

(passage: *[String]*) → *Datamap*

When given a passage string name, this provides a datamap containing information about that passage. If no name was provided, then it provides information about the current passage.

Example usage:

```
(passage:"Cellar")
```

Rationale:

There are times when you wish to examine the data of the story as it is running - for instance, checking what tag a certain passage has, and performing some special behaviour as a result. This macro provides that functionality.

Details:

The datamap contains the following names and values.

Name	Value
source	The source markup of the passage, exactly as you entered it in the Twine editor
name	The string name of this passage.

Name	Value
tags	An array of strings, which are the tags you gave to this passage.

The "source" value, like all strings, can be printed using (print:). Be warned that printing the source of the current passage, while inside of it, may lead to an infinite regress.

Interestingly, the construction `(print: (passage: "Cellar")'s source)` is essentially identical in function (albeit longer to write) than `(display: "Cellar")`.

See also:

(history:), (savedgames:)

The (link:) macro

(link: *String*) → *Changer*

Also known as: (link-replace:)

Makes a command to create a special link that can be used to show a hook.

Example usage:

`(link: "Stake")[The dracula crumbles to dust.]` will create a link reading "Stake" which, when clicked, disappears and shows "The dracula crumbles to dust."

Rationale:

As you're aware, links are what the player uses to traverse your story. However, links can also be used to simply display text or run macros inside hooks. Just attach the (link:) macro to a hook, and the entire hook will not run or appear at all until the player clicks the link.

Note that this particular macro's links disappear when they are clicked - if you want their words to remain in the text, consider using (link-reveal:).

Details:

This creates a link which is visually indistinguishable from normal passage links.

See also:

(link-reveal:), (link-repeat:), (link-goto:), (click:)

The (link-reveal:) macro

(link-reveal: *String*) → *Changer*

Makes a command to create a special link that shows a hook, keeping the link's text visible after clicking.

Example usage:

```
(link-reveal: "Heart")[broken]
```

will create a link reading "Heart" which, when clicked, changes to plain text, and shows "broken" after it.

Rationale:

This is similar to (link:), but allows the text of the link to remain in the passage after it is clicked. It allows key words and phrases in the passage to expand and reveal more text after themselves. Simply attach it to a hook, and the hook will only be revealed when the link is clicked.

Details:

This creates a link which is visually indistinguishable from normal passage links.

If the link text contains formatting syntax, such as "**bold**", then it will be retained when the link is demoted to text.

See also:

(link:), (link-repeat:), (link-goto:), (click:)

The (link-repeat:) macro

(link-repeat: *String*) → *Changer*

Makes a command to create a special link that shows a hook, and, when clicked again, re-runs the hook, appending its contents again.

Example usage:

`(link-repeat: "Add cheese")[(set:$cheese to it + 1)]` will create a link reading "Add cheese" which, when clicked, adds 1 to the $cheese variable using (set:), and can be clicked repeatedly.

Rationale:

This is similar to (link:), but allows the created link to remain in the passage after it is clicked. It can be used to make a link that displays more text after each click, or which must be clicked multiple times before something can happen (using (set:) and (if:) to keep count of the number of clicks).

Details:

This creates a link which is visually indistinguishable from normal passage links. Each time the link is clicked, the text and macros printed in the previous run are appended.

See also:

(link-reveal:), (link:), (link-goto:), (click:)

The (link-goto:) macro

(link-goto: *String*, *[String]*) → *Command*

Takes a string of link text, and an optional destination passage name, and makes a command to create a link that takes the player to another passage. The link functions identically to a standard link. This command should not be attached to a hook.

Example usage:

- `(link-goto: "Enter the cellar", "Cellar")` is approximately the same as `[[Enter the cellar->Cellar]]`.
- `(link-goto: "Cellar")` is the same as `[[Cellar]]`.

Rationale:

This macro serves as an alternative to the standard link syntax (`[[Link text->Destination]]`), but has a couple of slight differences.

- The link syntax lets you supply a fixed text string for the link, and an expression for the destination passage's name. However, it does not provide any other means of computing the link. (link-goto:) also allows the link text to be any expression - so, something like `(link-goto: "Move " + $name + "to the cellar", "Cellar")` can be written.
- The resulting command from this macro, like all commands, can be saved and used elsewhere. If you have a complicated link you need to use in several passages, you could (set:) it to a variable and use that variable in its place.

Details:

As a bit of trivia... the Harlowe engine actually converts all standard links into (link-goto:) macro calls internally - the link syntax is, essentially, a syntactic shorthand for (link-goto:).

See also:

(link:), (link-reveal:), (link-repeat:), (link-undo:), (goto:)

The (click:) macro

(click: *HookName* or *String*) → *Changer*

Produces a command which, when attached to a hook, hides it and enchants the specified target, such that it visually resembles a link, and that clicking it causes the attached hook to be revealed.

Example usage:

- `There is a small dish of water. (click: "dish")[Your finger gets wet.]` causes "dish" to become a link that, when clicked, reveals "Your finger gets wet." at the specified location.

- `[Fie and fuggaboo!]<shout| (click: ?shout)[Blast and damnation!]`

  does something similar to every hook named `<shout|`.

Rationale:

The (link:) macro and its variations lets you make passages more interactive, by adding links that display text when clicked. However, it can often greatly improve your passage code's readability to write a macro call that's separate from the text that it affects. You could want to write an entire paragraph, then write code that makes certain words into links, without interrupting the flow of the prose in the editor.

The (click:) macro lets you separate text and code in this way. Place (click:) hooks at the end of your passages, and have them affect named hooks, or text strings, earlier in the passage.

Details:

Text or hooks targeted by a (click:) macro will be styled in a way that makes them indistinguishable from passage links, and links created by (link:). When any one of the targets is clicked, this styling will be removed and the hook attached to the (click:) will be displayed.

Additionally, if a (click:) macro is removed from the passage, then its targets will lose the link styling and no longer be affected by the macro.

Targeting ?Page or ?Passage:

When a (click:) command is targeting the ?Page or ?Passage, instead of transforming the entire passage text into a link, something else will occur: a blue link-coloured border will surround the page, and the mouse cursor (on desktop browsers) will resemble a hand no matter what it's hovering over.

Clicking a link when a (click:) is targeting the ?Page or ?Passage will cause both the link and the (click:) to activate at once.

Using multiple (click:) commands to target the ?Page or ?Passage will require multiple clicks from the player to activate all of them. They activate in the order they appear on the page - top to bottom.

See also:

(link:), (link-reveal:), (link-repeat:), (mouseover:), (mouseout:), (replace:), (click-replace:)

The (link-undo:) macro

(link-undo: *String*) → *Command*

Takes a string of link text, and produces a link that, when clicked, undoes the current turn and sends the player back to the previously visited passage. The link appears identical to a typical passage link. This command should not be attached to a hook.

Example usage:

`(link-undo:"Retreat")` behaves the same as `(link:"Retreat")[[(undo:)](#macro_undo)]`.

Rationale:

The ability to undo the player's last turn, as an alternative to (go-to:), is explained in the documentation of the (undo:) macro. This macro provides a shorthand for placing (undo:) inside a (link:) attached hook.

You may, as part of customising your story, be using (replace:) to change the ?sidebar, and remove its default "undo" link. If so, you can selectively provide undo links at certain parts of your story instead, by using this macro.

Details:

As with (undo:), if this command is used on the play session's first turn, an error will be produced (as there is yet nothing to undo at that time.) You can check which turn it is by examining the `length` of the (history:) array.

See also:

(undo:), (link-goto:)

The (click-replace:) macro

(click-replace: *HookName* or *String*) → *Changer*

A special shorthand combination of the (click:) and (replace:) macros, this allows you to make a hook replace its own text with that of the attached hook whenever it's clicked. `(click: ?1)[(replace:?1)[...]]` can be rewritten as `(click-replace: ?1)[...]`.

Example usage:

```
My deepest secret.
```

```
(click-replace: "secret")[longing for you].
```

See also:

(click-prepend:), (click-append:)

The (click-append:) macro

(click-append: *HookName* or *String*) → *Changer*

A special shorthand combination of the (click:) and (append:) macros, this allows you to append text to a hook or string when it's clicked. `(click: ?1)[(append:?1)[...]]` can be rewritten as `(click-append: ?1)[...]`.

Example usage:

```
I have nothing to fear.
```

```
(click-append: "fear")[ but my own hand].
```

See also:

(click-replace:), (click-prepend:)

The (click-prepend:) macro

(click-prepend: *HookName* or *String*) → *Changer*

A special shorthand combination of the (click:) and (prepend:) macros, this allows you to prepend text to a hook or string when it's clicked. `(click: ?1)[(prepend:?1)[...]]` can be rewritten as `(click-prepend: ?1)[...]`.

Example usage:

```
Who stands with me?
```

```
(click-prepend: "?")[ but my shadow].
```

See also:

(click-replace:), (click-append:)

The (mouseover:) macro

(mouseover: *HookName* or *String*) → *Changer*

A variation of (click:) that, instead of showing the hook when the target is clicked, shows it when the mouse pointer merely hovers over it. The target is also styled differently, to denote this hovering functionality.

Rationale:

(click:) and (link:) can be used to create links in your passage that reveal text or, in conjunction with other macros, transform the text in myriad ways. This macro is exactly like (click:), except that instead of making the target a link, it makes the target reveal the hook when the mouse hovers over it. This can convey a mood of fragility and spontaneity in your stories, of text reacting to the merest of interactions.

Details:

This macro is subject to the same rules regarding the styling of its targets that (click:) has, so consult (click:)'s details to review them.

This macro is not recommended for use in games or stories intended for use on touch devices, as the concept of "hovering" over an element doesn't really make sense with that input method.

See also:

(link:), (link-reveal:), (link-repeat:), (click:), (mouseout:), (replace:), (mouseover-replace:), (hover-style:)

The (mouseover-replace:) macro

(mouseover-replace: *HookName* or *String*) → *Changer*

This is similar to (click-replace:), but uses the (mouseover:) macro's behaviour instead of (click:)'s. For more information, consult the description of (click-replace:).

The (mouseover-append:) macro

(mouseover-append: *HookName* or *String*) → *Changer*

This is similar to (click-append:), but uses the (mouseover:) macro's behaviour instead of (click:)'s. For more information, consult the description of (click-append:).

The (mouseover-prepend:) macro

(mouseover-prepend: *HookName* or *String*) → *Changer*

This is similar to (click-prepend:), but uses the (mouseover:) macro's behaviour instead of (click:)'s. For more information, consult the description of (click-prepend:).

The (mouseout:) macro

(mouseout: *HookName* or *String*) → *Changer*

A variation of (click:) that, instead of showing the hook when the target is clicked, shows it when the mouse pointer moves over it, and then leaves. The target is also styled differently, to denote this hovering functionality.

Rationale:

(click:) and (link:) can be used to create links in your passage that reveal text or, in conjunction with other macros, transform the text in myriad ways. This macro is exactly like (click:), but rather than making the target a link, it makes the target reveal the hook when the mouse stops hovering over it. This is very similar to clicking, but is subtly different, and conveys a sense of "pointing" at the element to interact with it rather than "touching" it. You can use this in your stories to give a dream-like or unearthly air to scenes or places, if you wish.

Details:

This macro is subject to the same rules regarding the styling of its targets that (click:) has, so consult (click:)'s details to review them.

This macro is not recommended for use in games or stories intended for use on touch devices, as the concept of "hovering" over an element doesn't really make sense with that input method.

See also:

(link:), (link-reveal:), (link-repeat:), (click:), (mouseover:), (replace:), (mouseout-replace:), (hover-style:)

The (mouseout-replace:) macro

(mouseout-replace: *HookName* or *String*) → *Changer*

This is similar to (click-replace:), but uses the (mouseout:) macro's behaviour instead of (click:)'s. For more information, consult the description of (click-replace:).

The (mouseout-append:) macro

(mouseout-append: *HookName* or *String*) → *Changer*

This is similar to (click-append:), but uses the (mouseout:) macro's behaviour instead of (click:)'s. For more information, consult the description of (click-append:).

The (mouseout-prepend:) macro

(mouseout-prepend: *HookName* or *String*) → *Changer*

This is similar to (click-prepend:), but uses the (mouseout:) macro's behaviour instead of (click:)'s. For more information, consult the description of (click-prepend:).

The (go-to:) macro

(go-to: *String*) → *Command*

This command stops passage code and sends the player to a new passage. If the passage named by the string does not exist, this produces an error.

Example usage:

```
(go-to: "The Distant Future")
```

Rationale:

There are plenty of occasions where you may want to instantly advance to a new passage without the player's volition. (go-to:) provides access to this ability.

(go-to:) can accept any expression which evaluates to a string. You can, for instance, go to a randomly selected passage by combining it with (either:) - `(go-to: (either: "Win", "Lose", "Draw"))`.

(go-to:) can be combined with (link:) to accomplish the same thing as (link-goto:): `(link:"Enter the hole")[(go-to:"Falling")]` However, you can include other macros inside the hook to run before the (go-to:), such as (set:), (put:) or (save-game:).

Details:

If it is performed, (go-to:) will "halt" the passage and prevent any macros and text after it from running. So, a passage that contains:

```
(set: $listen to "I love")
(go-to: "Train")
(set: $listen to it + " you")
```

will *not* cause `$listen` to become `"I love you"` when it runs.

Going to a passage using this macro will count as a new "turn" in the game's passage history, much as if a passage link was clicked. If you want to go back to the previous passage, forgetting the current turn, then you may use (undo:).

See also:

(link-goto:), (undo:), (loadgame:)

The (undo:) macro

(undo:) → *Command*

This command stops passage code and "undoes" the current turn, sending the player to the previous visited passage and forgetting any variable changes that occurred in this passage.

Example usage:

```
You scurry back whence you came... (live:2s)[(undo:)]
```
will undo the current turn after 2 seconds.

Rationale:

The (go-to:) macro sends players to different passages instantly. But, it's common to want to send players back to the passage they previously visited, acting as if this turn never happened. (undo:) provides this functionality.

By default, Harlowe offers a button in its sidebar that lets players undo at any time, going back to the beginning of the game session. However, if you wish to use this macro, and only permit undos in certain passages and occasions, you may remove the button by using (replace:) on the ?sidebar in a header tagged passage.

Details:

If this is the first turn of the game session, (undo:) will produce an error. You can check which turn it is by examining the `length` of the (history:) array.

Just like (go-to:), (undo:) will "halt" the passage and prevent any macros and text after it from running.

See also:

(go-to:), (link-undo:)

The (live:) macro

(live: [*Number*]) → *Changer*

When you attach this macro to a hook, the hook becomes "live", which means that it's repeatedly re-run every certain number of milliseconds, replacing the source inside of the hook with a newly computed version.

Example usage:

```
{(live: 0.5s)[
```

```
    (either: "Bang!", "Kaboom!", "Whammo!", "Pow!")
```

```
]}
```

Rationale:

Twine passage text generally behaves like a HTML document: it starts as code, is changed into a rendered page when you "open" it, and remains so until you leave. But, you may want a part of the page to change itself before the player's eyes, for its code to be re-renders "live" in front of the player, while the remainder of the passage remains the same.

Certain macros, such as the (link:) macro, allow a hook to be withheld until after an element is interacted with. The (live:) macro is more versatile: it re-renders a hook every specified number of milliseconds. If (if:) or (unless:) macros are inside the hook, they of course will be re-evaluated each time. By using these two kinds of macros, you can make a (live:) macro repeatedly check if an event has occurred, and only change its text at that point.

Details:

Live hooks will continue to re-render themselves until they encounter and print a (stop:) macro.

The (stop:) macro

(stop:) → *Command*

This macro, which accepts no arguments, creates a (stop:) command, which is not configurable.

Example usage:

```
{(live: 1s)[
```

```
    (if: $packedBags)[OK, let's go!(stop:)]
```

```
    (else: )[(either:"Are you ready yet?","We mustn't be late!")]
```

```
]}
```

Rationale:

Clunky though it looks, this macro serves a single important purpose: inside a (live:) macro's hook, its appearance signals that the macro must stop running. In every other occasion, this macro does nothing.

See also:

(live:)

The (abs:) macro

(abs: *Number*) → *Number*

This maths macro finds the absolute value of a number (without the sign).

Example usage:

`(abs: -4)` produces 4.

The (cos:) macro

(cos: *Number*) → *Number*

This maths macro computes the cosine of the given number of radians.

Example usage:

`(cos: 3.14159265)` produces -1.

The (exp:) macro

(exp: *Number*) → *Number*

This maths macro raises Euler's number to the power of the given number, and provides the result.

Example usage:

```
(exp: 6)
```
produces approximately 403.

The (log:) macro

(log: *Number*) → *Number*

This maths macro produces the natural logarithm (the base-e logarithm) of the given number.

Example usage:

```
(log: (exp:5))
```
produces 5.

The (log10:) macro

(log10: *Number*) → *Number*

This maths macro produces the base-10 logarithm of the given number.

Example usage:

```
(log10: 100)
```
produces 2.

The (log2:) macro

(log2: *Number*) → *Number*

This maths macro produces the base-2 logarithm of the given number.

Example usage:

```
(log2: 256)
```
produces 8.

The (max:) macro

(max: ...*Number*) → *Number*

This maths macro accepts numbers, and evaluates to the highest valued number.

Example usage:

```
(max: 2, -5, 2, 7, 0.1)
```
produces 7.

The (min:) macro

(min: ...*Number*) → *Number*

This maths macro accepts numbers, and evaluates to the lowest valued number.

Example usage:

```
(min: 2, -5, 2, 7, 0.1)
```
produces -5.

The (pow:) macro

(pow: *Number*, *Number*) → *Number*

This maths macro raises the first number to the power of the second number, and provides the result.

Example usage:

```
(pow: 2, 8)
```
produces 256.

The (sign:) macro

(sign: *Number*) → *Number*

This maths macro produces -1 when given a negative number, 0 when given 0, and 1 when given a positive number.

Example usage:

```
(sign: -4)
```
produces -1.

The (sin:) macro

(sin: *Number*) → *Number*

This maths macro computes the sine of the given number of radians.

Example usage:

```
(sin: 3.14159265 / 2)
```
produces 1.

The (sqrt:) macro

(sqrt: *Number*) → *Number*

This maths macro produces the square root of the given number.

Example usage:

```
(sqrt: 25)
```
produces 5.

The (tan:) macro

(tan: *Number*) → *Number*

This maths macro computes the tangent of the given number of radians.

Example usage:

```
(tan: 3.14159265 / 4)
```
produces approximately 1.

The (ceil:) macro

(ceil: *Number*) → *Number*

This macro rounds the given number upward to a whole number. If a whole number is provided, it returns the number as-is.

Example usage:

`(ceil: 1.1)` produces 2.

The (floor:) macro

(floor: *Number*) → *Number*

This macro rounds the given number downward to a whole number. If a whole number is provided, it returns the number as-is.

Example usage:

`(floor: 1.99)` produces 1.

The (num:) macro

(num: *String*) → *Number*

Also known as: (number:)

This macro converts strings to numbers by reading the digits in the entire string. It can handle decimal fractions and negative numbers. If any letters or other unusual characters appear in the number, it will result in an error.

Example usage:

`(num: "25")` results in the number `25`.

Rationale:

Unlike in Twine 1 and SugarCube, Twine 2 will only convert numbers into strings, or strings into numbers, if you explictly ask it to using macros such as this. This extra carefulness

decreases the likelihood of unusual bugs creeping into stories (such as performing `"Eggs: " + 2 + 1` and getting `"Eggs: 21"`).

Usually, you will only work with numbers and strings of your own creation, but if you're receiving user input and need to perform arithmetic on it, this macro will be necessary.

See also:

(text:)

The (random:) macro

(random: *Number*, *[Number]*) → *Number*

This macro produces a whole number randomly selected between the two whole numbers, inclusive (or, if the second number is absent, then between 0 and the first number, inclusive).

Example usage:

`(random: 1,6)` simulates a six-sided die roll.

See also:

(either:), (shuffled:)

The (round:) macro

(round: *Number*) → *Number*

This macro rounds the given number to the nearest whole number - downward if its decimals are smaller than 0.5, and upward otherwise. If a whole number is provided, it returns the number as-is.

Example usage:

`(round: 1.5)` produces 2.

The (alert:) macro

(alert: *String*) → *Command*

This macro produces a command that, when evaluated, shows a browser pop-up dialog box with the given string displayed, and an "OK" button to dismiss it.

Example usage:

```
(alert:"Beyond this point, things get serious. Grab a snack and buckle up.")
```

Details:

This is essentially identical to the Javascript `alert()` function in purpose and ability. You can use it to display a special message above the game itself. But, be aware that as the box uses the player's operating system and browser's styling, it may clash visually with the design of your story.

When the dialog is on-screen, the entire game is essentially "paused" - no further computations are performed until it is dismissed.

See also:

(prompt:), (confirm:)

The (confirm:) macro

(confirm: *String*) → *Boolean*

When this macro is evaluated, a browser pop-up dialog box is shown with the given string displayed, as well as "OK" and "Cancel" button to confirm or cancel whatever action or fact the string tells the player. When it is submitted, it evaluates to the boolean true if "OK" had been pressed, and false if "Cancel" had.

Example usage:

```
(set: $makeCake to (confirm: "Transform your best friend into a cake?"))
```

Details:

This is essentially identical to the Javascript `confirm()` function in purpose and ability.

You can use it to ask the player a question directly, and act on the result immediately. But, be aware that as the box uses the player's operating system and browser's styling, it may clash visually with the design of your story.

When the dialog is on-screen, the entire game is essentially "paused" - no further computations are performed until it is dismissed.

See also:

(alert:), (prompt:)

The (prompt:) macro

(prompt: *String*, *String*) → *String*

When this macro is evaluated, a browser pop-up dialog box is shown with the first string displayed, a text entry box containing the second string (as a default value), and an "OK" button to submit. When it is submitted, it evaluates to the string in the text entry box.

Example usage:

```
(set: $name to (prompt: "Your name, please:", "Frances Spayne"))
```

Details:

This is essentially identical to the Javascript `prompt()` function in purpose and ability. You can use it to obtain a string value from the player directly, such as a name for the main character. But, be aware that as the box uses the player's operating system and browser's styling, it may clash visually with the design of your story.

When the dialog is on-screen, the entire game is essentially "paused" - no further computations are performed until it is dismissed.

See also:

(alert:), (confirm:)

The (append:) macro

(append: ...*HookName* or *String*) → *Changer*

A variation of (replace:) which adds the attached hook's contents to the end of each target, rather than replacing it entirely.

Example usage:

- `(append: "Emily", "Em")[, my maid]` adds ", my maid " to the end of every occurrence of "Emily" or "Em".
- `(append: ?dress)[ from happier days]` adds " from happier days" to the end of the `|dress>` hook.

Rationale:

As this is a variation of (replace:), the rationale for this macro can be found in that macro's description. This provides the ability to append content to a target, building up text or amending it with an extra sentence or word, changing or revealing a deeper meaning.

See also:

(replace:), (prepend:)

The (prepend:) macro

(prepend: ...*HookName* or *String*) → *Changer*

A variation of (replace:) which adds the attached hook's contents to the beginning of each target, rather than replacing it entirely.

Example usage:

- `(prepend: "Emily", "Em")[Miss ]` adds "Miss " to the start of every occurrence of "Emily" or "Em".
- `(prepend: ?dress)[my wedding ]` adds "my wedding " to the start of the `|dress>` hook.

Rationale:

As this is a variation of (replace:), the rationale for this macro can be found in that macro's description. This provides the ability to prepend content to a target, adding preceding sentences or words to a text to change or reveal a deeper meaning.

See also:

(replace:), (append:)

The (replace:) macro

(replace: ...*HookName* *or* *String*) → *Changer*

Creates a command which you can attach to a hook, and replace target destinations with the hook's contents. The targets are either text strings within the current passage, or hook references.

Example usage:

This example changes the words "categorical catastrophe" to "**dog**egorical **dog**astrophe"

```
A categorical catastrophe!
```

```
(replace: "cat")[**dog**]
```

This example changes the `|face>` and `|heart>` hooks to read "smile":

```
A |heart>[song] in your heart, a |face>[song] on your face.
```

```
(replace: ?face, ?heart)[smile]
```

Rationale:

A common way to make your stories feel dynamic is to cause their text to modify itself before the player's eyes, in response to actions they perform. You can check for these actions using macros such as (link:), (click:) or (live:), and you can make these changes using macros such as (replace:).

Using (replace:) is only one way of providing this dynamism, however - the (show:) macro also offers similar functionality. See that macro's article for an explanation of when you might prefer to use it over (replace:), and vice-versa.

Details:

(replace:) lets you specify a target, and a block of text to replace the target with. The attached hook will not be rendered normally - thus, you can essentially place (replace:) commands anywhere in the passage text without interfering much with the passage's visible text.

If the given target is a string, then every instance of the string in the current passage is replaced with a copy of the hook's contents. If the given target is a hook reference, then only named hooks with the same name as the reference will be replaced with the hook's contents. Use named hooks when you want only specific places in the passage text to change.

If the target doesn't match any part of the passage, nothing will happen. This is to allow you to place (replace:) commands in `header` tagged passages, if you want them to conditionally affect certain named hooks throughout the entire game, without them interfering with other passages.

See also:

(append:), (prepend:), (show:)

The (load-game:) macro

(load-game: *String*) → *Command*

This command attempts to load a saved game from the given slot, ending the current game and replacing it with the loaded one. This causes the passage to change.

Example usage:

```
{(if: $Saves contains "Slot A")[
```

```
   (link: "Load game")[(load-game:"Slot A")]
```

```
]}
```

Details:

Just as (save-game:) exists to store the current game session, (load-game:) exists to retrieve a past game session, whenever you want. This command, when given the string name of a slot, will attempt to load the save, completely and instantly replacing the variables and move history with that of the save, and going to the passage where that save was made.

This macro assumes that the save slot exists and contains a game, which you can check by seeing if `[(saved-games:)](#macro_saved-games) contains` the slot name before running (load-game:).

See also:

(save-game:), (saved-games:)

The (save-game:) macro

(save-game: *String*, *[String]*) → *Boolean*

This macro saves the current game's state in browser storage, in the given save slot, and including a special filename. It can then be restored using (load-game:).

Rationale:

Many web games use browser cookies to save the player's place in the game. Twine allows you to save the game, including all of the variables that were (set:) or (put:), and the passages the player visited, to the player's browser storage.

(save-game:) is a single operation that can be used as often or as little as you want to. You can include it on every page; You can put it at the start of each "chapter"; You can put it inside a (link:) hook, such as

```
{(link:"Save game")[
```

```
   (if:(save-game:"Slot A"))[
```

```
      Game saved!
```

```
   ](else: )[
      Sorry, I couldn't save your game.
   ]
]}
```

and let the player choose when to save.

Details:

(save-game:)'s first string is a slot name in which to store the game. You can have as many slots as you like. If you only need one slot, you can just call it, say, `"A"`, and use `(save-game:"A")`. You can tie them to a name the player gives, such as `(save-game: $playerName)`, if multiple players are likely to play this game - at an exhibition, for instance.

Giving the saved game a file name is optional, but allows that name to be displayed by finding it in the $Saves datamap. This can be combined with a (load-game:)(link:) to clue the players into the save's contents:

```
(link: "Load game: " + ("Slot 1") of Saves)[
   (load-game: "Slot 1")
]
```

(save-game:) evaluates to a boolean - true if the game was indeed saved, and false if the browser prevented it (because they're using private browsing, their browser's storage is full, or some other reason). Since there's always a possibility of a save failing, you should use (if:) and (else:) with (save-game:) to display an apology message in the event that it returns false (as seen above).

See also:

(load-game:), (saved-games:)

The (saved-games:) macro

(saved-games:) → *Datamap*

This returns a datamap containing the names of currently occupied save game slots.

Example usage:

`(print (saved-games:)'s "File A")` prints the name of the save file in the slot "File A". `(if: (saved-games:) contains "File A")` checks if the slot "File A" is occupied.

Rationale:

For a more thorough description of the save file system, see the (save-game:) article. This macro provides a means to examine the current save files in the user's browser storage, so you can decide to print "Load game" links if a slot is occupied, or display a list of all of the occupied slots.

Details:

Each name in the datamap corresponds to an occupied slot name. The values are the file names of the files occupying the slot.

Changing the datamap does not affect the save files - it is simply information.

See also:

(save-game:), (load-game:)

The (hidden:) macro

(hidden:) → *Changer*

Produces a command that can be attached to hooks to hide them.

Example usage:

```
Don't you recognise me? (hidden:)|truth>[I'm your OC, brought to life!]
```

The above example is the same as

```
Don't you recognise me? |truth)[I'm your OC, brought to life!]
```

Rationale:

While there is a way to succinctly mark certain named hooks as hidden, by using parentheses instead of `<` or `>` marks, this macro provides a clear way for complex changers to hide their attached hooks. This works well when added to the (hook:) macro, for instance, to specify a hook's name and visibility in a single changer.

This macro is essentially identical in behaviour to `(if:false)`, but reads better.

See also:

(if:), (hook:), (show:)

The (show:) macro

(show: ...*HookName*) → *Command*

Reveals hidden hooks, running the code within.

Example usage:

```
|fan)[The overhead fan spins lazily.]
```

```
(link:"Turn on fan")[(show:?fan)]
```

Rationale:

The purpose of hidden hooks is, of course, to eventually show them - and this macro is how you show them. You can use this command inside a (link:), trigger it in real-time with a (live:) macro, or anywhere else.

Using (show:) vs (replace:): There are different reasons for using hidden hooks and (show:) instead of (replace:). For your stories, think about whether the prose being revealed is part of the "main" text of the passage, or is just an aside. In neatly-coded stories, the main text should appear early in a passage's code, as the focus of the writer's attention.

When using (replace:), the replacement prose is written far from its insertion point. This can improve readability when the insertion point is part of a long paragraph or sentence, and the

prose is a minor aside or amendment, similar to a footnote or post-script, that would clutter the paragraph were it included inside. Additionally, (replace:) can be used in a "header" or "footer" tagged passage to affect certain named hooks throughout the story.

```
You turn away from her, facing the grandfather clock, its [stern
ticking]<1| filling the tense silence.
```

```
(click-replace: ?1)[echoing, hollow ticking]
```

When using (show:), the hidden hook's position is fixed in the passage prose. This can improve readability when the hidden hook contains a lot of the "main" text of a passage, which provides vital context and meaning for the rest of the text.

```
I don't know where to begin... |1)[The weird state of my birth, the
prophecy made centuries ago,
my first day of school, the day of the meteors, the day I awoke my
friends' powers... so many strands in
the tapestry of my tale, and no time to unravel them.] ...so for now I'll
start with when we fell down the hole.
```

```
(link:"Where, indeed?")[(show:?1)]
```

But, there aren't any hard rules for when you should use one or the other. As a passage changes in the writing, you should feel free to change between one or the other, or leave your choice as-is.

Details:

(show:) will reveal every hook with the given name. To only reveal a specific hook, you can use the possessive syntax, as usual: `(show: ?shrub's 1st)`.

If you provide to (show:) a hook which is already visible, an error will be produced.

See also:

(hidden:), (replace:)

The (lowercase:) macro

(lowercase: *String*) → *String*

This macro produces a lowercase version of the given string.

Example usage:

```
(lowercase: "GrImAcE")
```
is the same as
```
"grimace"
```

Details:

The results of this macro for non-ASCII characters currently depends on the player's browser's Unicode support. For instance, 'Ä°' in lowercase should be 'iÌ‡', but some browsers don't support this.

See also:

(uppercase:), (lowerfirst:), (upperfirst:)

The (lowerfirst:) macro

(lowerfirst: *String*) → *String*

This macro produces a version of the given string, where the first alphanumeric character is lowercase, and other characters are left as-is.

Example usage:

```
(lowerfirst: " College B")
```
is the same as
```
" college B"
```

Details:

If the first alphanumeric character cannot change case (for instance, if it's a number) then nothing will change in the string. So, "8DX" won't become "8dX".

The results of this macro for non-ASCII characters currently depends on the player's browser's Unicode support. For instance, 'Ä°' in lowercase should be 'iÌ‡', but some browsers don't support this.

See also:

(uppercase:), (lowercase:), (upperfirst:)

The (text:) macro

(text: ...*[Number or String or Boolean or Array]*) → *String*

Also known as: (string:)

(text:) accepts any amount of expressions and tries to convert them all to a single String.

Example usages:

- `(text: $cash + 200)`
- `(if: (text: $cash)'s length > 3)[Phew! Over four digits!]`
- `(text: ...$arr)`

Rationale:

Unlike in Twine 1 and SugarCube, Twine 2 will only convert numbers into strings, or strings into numbers, if you explictly ask it to. This extra carefulness decreases the likelihood of unusual bugs creeping into stories (such as adding 1 and "22" and getting "122"). The (text:) macro (along with (num:)) is how you can convert non-string values to a string.

Details:

This macro can also be used much like the (print:) macro - as it evaluates to a string, and strings can be placed in the story source freely,

If you give an array to (text:), it will attempt to convert every element contained in the array to a String, and then join them up with commas. So, `(text: (a: 2, "Hot", 4, "U"))` will result in the string "2,Hot,4,U". If you'd rather this not occur, you can also pass the array's individual elements using the `...` operator - this will join them with nothing in between. So, `(text: ...(a: 2, "Hot", 4, "U"))` will result in the string "2Hot4U".

See also:

(num:)

The (uppercase:) macro

(uppercase: *String*) → *String*

This macro produces an uppercase version of the given string.

Example usage:

`(uppercase: "GrImAcE")` is the same as `"GRIMACE"`

Details:

The results of this macro for non-ASCII characters currently depends on the player's browser's Unicode support. For instance, 'ÃŸ' in uppercase should be 'SS', but some browsers don't support this.

See also:

(lowercase:), (upperfirst:), (lowerfirst:)

The (upperfirst:) macro

(upperfirst: *String*) → *String*

This macro produces a version of the given string, where the first alphanumeric character is uppercase, and other characters are left as-is.

Example usage:

`(upperfirst: " college B")` is the same as `" College B"`

Details:

If the first alphanumeric character cannot change case (for instance, if it's a number) then nothing will change in the string. So, "4ever" won't become "4Ever".

The results of this macro for non-ASCII characters currently depends on the player's browser's Unicode support. For instance, 'ÃŸ' in uppercase should be 'SS', but some browsers don't support this.

See also:

(uppercase:), (lowercase:), (lowerfirst:)

The (words:) macro

(words: *String*) → *Array*

This macro takes a string and creates an array of each word ("word" meaning a sequence of non-whitespace characters) in the string.

Example usage:

`(words: "god-king Torment's peril")` is the same as `(a: "god-king", "Torment's", "peril")`

Rationale:

It can be useful to explicitly distinguish individual words within a string, in a manner not possible with just the `contains` operator - for instance, seeing if a string contains the bare word "to" - not "torn" or any other larger word. This macro allows a string's words to be split up and examined individually - you can safely check if `(words: $a) contains "to"`, or

check on a particular word in the sequence by asking if, say, `(words: $a)'s 2nd is 'goose'`.

Details:

If the string was empty or contained only whitespace, then this will create an empty array. Moreover, if the string contained no whitespace, then the array will contain just the entire original string.

The whitespace characters recognised by this macro include line breaks, non-breaking spaces, and other uncommon space characters.

See also:

(startcase:)

The (align:) macro

(align: *String*) → *Changer*

This styling command changes the alignment of text in the attached hook, as if the `===>` arrow syntax was used. In fact, these same arrows (`==>`, `=><=`, `<==>`, `====><=` etc.) should be supplied as a string to specify the degree of alignment.

Example usage:

```
(align: "=><==")[Hmm? Anything the matter?]
```

Details:

Hooks affected by this command will take up their own lines in the passage, regardless of their placement in the story prose. This allows them to be aligned in the specified manner.

The (background:) macro

(background: *Colour* or *String*) → *Changer*

This styling command alters the background colour or background image of the attached hook. Supplying a colour, or a string contanining a CSS hexadecimal colour (such as `#A6A612`) will set the background to a flat colour. Other strings will be interpreted as an image URL, and the background will be set to it.

Example usage:

- `(background: red + white)[Pink background]`
- `(background: "#663399")[Purple background]`
- `(background: "marble.png")[Marble texture background]`

Details:

Combining two (background:) commands will do nothing if they both influence the colour or the image. For instance `(background:red) + (background:white)` will simply produce the equivalent `(background:white)`. However, `(background:red) + (background:"mottled.png")` will work as intended if the background image contains transparency, allowing the background colour to appear through it.

Currently, supplying other CSS colour names (such as `burlywood`) is not permitted - they will be interpreted as image URLs regardless.

No error will be reported if the image at the given URL cannot be accessed.

See also:

(colour:)

The (css:) macro

(css: *String*) → *Changer*

This takes a string of inline CSS, and applies it to the hook, as if it were a HTML "style" property.

Usage example:

```
(css: "background-color:indigo")
```

Rationale:

The built-in macros for layout and styling hooks, such as (text-style:), are powerful and geared toward ease-of-use, but do not entirely provide comprehensive access to the browser's styling. This changer macro allows extended styling, using inline CSS, to be applied to hooks.

This is, however, intended solely as a "macro of last resort" - as it requires basic knowledge of CSS - a separate language distinct from Harlowe - to use, and requires it be provided a single inert string, it's not as accommodating as the other such macros.

See also:

(text-style:)

The (font:) macro

(font: *String*) → *Changer*

This styling command changes the font used to display the text of the attached hook. Provide the font's family name (such as "Helvetica Neue" or "Courier") as a string.

Example usage:

```
(font:"Skia")[And what have we here?]
```

Details:

Currently, this command will only work if the font is available to the player's browser. If font files are embedded in your story stylesheet using base64 (an explanation for which is beyond the scope of this macro's description) then it can be uses instead.

No error will be reported if the provided font name is not available, invalid or misspelled.

See also:

(text-style:)

The (hook:) macro

(hook: *String*) → *Changer*

A command that allows the author to give a hook a computed tag name.

Example usage:

```
(hook: $name)[]
```

Rationale:

You may notice that it isn't possible to attach a nametag to hooks with commands already attached - in the case of `(font:"Museo Slab")[The Vault]<title|`, the nametag results in an error. This command can be added with other commands to allow the hook to be named: `(font:"Museo Slab")+(hook: "title")`.

Furthermore, unlike the nametag syntax, (hook:) can be given any string expression: `(hook: "eyes" + (string:$eyeCount))` is valid, and will, as you'd expect, give the hook the name of `eyes1` if `$eyeCount` is 1.

See also:

(hidden:)

The (hover-style:) macro

(hover-style: *Changer*) → *Changer*

Given a style-altering changer, it makes a changer which only applies when the hook or expression is hovered over with the mouse pointer, and is removed when hovering off.

Example usage:

The following makes a (link:) that turns cyan and italic when the mouse hovers over it.

```
(hover-style:(text-color:cyan) + (text-style:'italic'))+(link:"The lake")
```

```
[The still, cold lake.]
```

Rationale:

Making text react in small visual ways when the pointer hovers over it is an old hypertext tradition. It lends a degree of "life" to the text, making it seem aware of the player. This feeling of life is best used to signify interactivity - it seems to invite the player to answer in turn, by clicking. So, adding them to (link:) changers, instead of just bare words or paragraphs, is highly recommended.

Details:

True to its name, this macro can only be used for subtle style changes. Only the following changers (and combinations thereof) may be given to (hover-style:) - any others will produce an error:

- (align:)
- (background:)
- (css:)
- (font:)
- (text-colour:)
- (text-rotate:)
- (text-style:)

More extensive mouse-based interactivity should use the (mouseover:) and (mouseout:) macros.

This macro is not recommended for use in games or stories intended for use on touch devices, as the concept of "hovering" over an element doesn't really make sense with that input method.

See also:

(mouseover:), (mouseout:)

The (text-colour:) macro

(text-colour: *String* or *Colour*) → *Changer*

Also known as: (colour:), (text-color:), (color:)

This styling command changes the colour used by the text in the attached hook. You can supply either a string with a CSS-style colour (a colour name or RGB number supported by CSS), or a built-in colour object.

Example usage:

```
(colour: red + white)[Pink]
```
combines the built-in red and white colours to make pink.

```
(colour: "#696969")[Gray]
```
uses a CSS-style colour to style the text gray.

Details:

This macro only affects the text colour. To change the text background, call upon the (background:) macro.

See also:

(background:)

The (text-rotate:) macro

(text-rotate: *Number*) → *Changer*

This styling command visually rotates the attached hook clockwise by a given number of degrees. The rotational axis is in the centre of the hook.

Example usage:

```
(text-rotate:45)[Tilted]
```
will produce Tilted

Details:

The surrounding non-rotated text will behave as if the rotated text is still in its original position - the horizontal space of its original length will be preserved, and text it overlaps with vertically will ignore it.

A rotation of 180 degrees will, due to the rotational axis, flip the hook upside-down and back-to-front, as if the (text-style:) styles "mirror" and "upside-down" were both applied.

Due to browser limitations, hooks using this macro will have its CSS `display` attribute set to `inline-block`.

See also:

(text-style:)

The (text-style:) macro

(text-style: *String*) → *Changer*

This applies a selected built-in text style to the hook's text.

Example usage:

```
The shadow (text-style: "shadow")[flares] at you!
```
will style the word "flares" with a shadow.

```
(set: $s to (text-style: "shadow")) The shadow $s[flares] at you!
```
will also style it with a shadow.

Rationale:

While Twine offers markup for common formatting styles like bold and italic, having these styles available from a command macro provides some extra benefits: it's possible, as with all such style macros, to (set:) them into a variable, combine them with other commands, and re-use them succinctly throughout the story (by using the variable in place of the macro).

Furthermore, this macro also offers many less common but equally desirable styles to the author, which are otherwise unavailable or difficult to produce.

Details:

At present, the following text strings will produce a particular style.

String	Example
"none"	

String	Example
"bold"	
"italic"	
"underline"	
"strike"	
"superscript"	
"subscript"	
"mark"	
"outline"	
"shadow"	
"emboss"	
"condense"	
"expand"	
"blur"	
"blurrier"	
"smear"	
"mirror"	
"upside-down"	
"blink"	
"fade-in-out"	
"rumble"	
"shudder"	

You can use the "none" style to remove an existing style from a combined changer.

Due to browser limitations, hooks using "mirror", "upside-down", "rumble" or "shudder" will have its CSS `display` attribute set to `inline-block`.

See also:

(css:)

The (transition-time:) macro

(transition-time: *Number*) → *Changer*

Also known as: (t8n-time:)

A command that, when added to a (transition:) command, adjusts the time of the transition.

Example usage:

```
(set: $slowTransition to (transition:"shudder") + (transition-time: 2s))
```

creates a transition style which uses "shudder" and takes 2 seconds.

Details:

Much like (live:), this macro should be given a number of milliseconds (such as `50ms`) or seconds (such as `10s`). Providing 0 or fewer seconds/milliseconds is not permitted and will result in an error.

See also:

(transition:)

The (transition:) macro

(transition: *String*) → *Changer*

Also known as: (t8n:)

A command that applies a built-in CSS transition to a hook as it appears.

Example usage:

`(transition: "pulse")[Gleep!]` makes the hook `[Gleep!]` use the "pulse" transition when it appears.

Details:

At present, the following text strings will produce a particular transition:

- "dissolve" (causes the hook to gently fade in)
- "shudder" (causes the hook to instantly appear while shaking back and forth)
- "pulse" (causes the hook to instantly appear while pulsating rapidly)

All transitions are 0.8 seconds long, unless a (transition-time:) command is added to the command.

See also:

(text-style:), (transition-time:)

The (goto-url:) macro

(goto-url: *String*) → *Command*

When this command is used, the player's browser will immediately attempt to leave the story's page, and navigate to the given URL in the same tab. If this succeeds, then the story session will "end".

Example usage:

```
(goto-url: "http://www.example.org/")
```

Details:

If the given URL is invalid, no error will be reported - the browser will simply attempt to open it anyway.

Much like the `<a>` HTML element, the URL is treated as a relative URL if it doesn't start with "http://", "https://", or another such protocol. This means that if your story file is hosted at "http://www.example.org/story.html", then `(open-url: "page2.html")` will actually open the URL "http://www.example.org/page2.html".

See also:

(open-url:)

The (open-url:) macro

(open-url: *String*) → *Command*

When this macro is evaluated, the player's browser attempts to open a new tab with the given URL. This will usually require confirmation from the player, as most browsers block Javascript programs such as Harlowe from opening tabs by default.

Example usage:

```
(open-url: "http://www.example.org/")
```

Details:

If the given URL is invalid, no error will be reported - the browser will simply attempt to open it anyway.

Much like the `<a>` HTML element, the URL is treated as a relative URL if it doesn't start with "http://", "https://", or another such protocol. This means that if your story file is hosted at "http://www.example.org/story.html", then `(open-url: "page2.html")` will actually open the URL "http://www.example.org/page2.html".

See also:

(goto-url:)

The (page-url:) macro

(page-url:) → *String*

This macro produces the full URL of the story's HTML page, as it is in the player's browser.

Example usage:

`(if: (page-url:) contains "#cellar")` will be true if the URL contains the `#cellar` hash.

Details:

This **may** be changed in a future version of Harlowe to return a datamap containing more descriptive values about the URL, instead of a single string.

The (reload:) macro

(reload:) → *Command*

When this command is used, the player's browser will immediately attempt to reload the page, in effect restarting the entire story.

Example usage:

```
(click:"Restart")[(reload:)]
```

Details:

If the first passage in the story contains this macro, the story will be caught in a "reload loop", and won't be able to proceed. No error will be reported in this case.

The (substring:) macro

(substring: *String*, *Number*, *Number*) → *String*

This macro produces a substring of the given string, cut from two inclusive number positions.

Example usage:

`(substring: "growl", 3, 5)` is the same as `"growl"'s (a:3,4,5)`

Rationale:

You can obtain substrings of strings without this macro, by using the `'s` or `of` syntax along with an array of positions. For instance, `$str's (range:4,12)` obtains a substring of $str containing its 4th through 12th characters. But, for compatibility with previous Harlowe versions which did not feature this syntax, this macro also exists.

Details:

If you provide negative numbers, they will be treated as being offset from the end of the string - `-2` will specify the `2ndlast` character, just as 2 will specify the `2nd` character.

If the last number given is smaller than the first (for instance, in `(substring: "hewed", 4, 2)`) then the macro will still work - in that case returning "ewe" as if the numbers were in the correct order.

See also:

(subarray:)

The (subarray:) macro

(subarray: *Array*, *Number*, *Number*) → *Array*

When given an array, this returns a new array containing only the elements whose positions are between the two numbers, inclusively.

Example usage:

`(subarray: $a, 3, 4)` is the same as `$a's (a:3,4)`

Rationale:

You can obtain subarrays of arrays without this macro, by using the `'s` or `of` syntax along with an array of positions. For instance, `$a's (range:4,12)` obtains a subarray of $a containing its 4th through 12th values. But, for compatibility with previous Harlowe versions which did not feature this syntax, this macro also exists.

Details:

If you provide negative numbers, they will be treated as being offset from the end of the array - `-2` will specify the `2ndlast` item, just as 2 will specify the `2nd` item.

If the last number given is larger than the first (for instance, in `(subarray: (a:1,2,3,4), 4, 2)`) then the macro will still work - in that case returning (a:2,3,4) as if the numbers were in the correct order.

See also:

(substring:), (rotated:)

Types of data

Any data

A macro that is said to accept "Any" will accept any kind of data without complaint, as long as the data does not contain any errors.

Array data

There are occasions when you may need to work with a whole sequence of values at once. For example, a sequence of adjectives (describing the player) that should be printed depending on what a numeric variable (such as a health point variable) currently is. You could create many, many variables to hold each value, but it is preferable to use an array containing these values.

Arrays are one of the two major "data structures" you can use in Harlowe. The other, datamaps, are created with (dm:). Generally, you want to use arrays when you're dealing with values that directly correspond to *numbers*, and whose *order* and *position* relative to each other matter. If you instead need to refer to values by a name, and don't care about their order, a datamap is best used.

You can refer to and extract data at certain positions inside arrays using `1st`, `2nd`, `3rd`, and so forth: `$array's 1st`, also written as `1st of $array`, refers to the value in the first position. Additionally, you can use `last` to refer to the last position, `2ndlast` to refer to the second-last, and so forth. Arrays also have a `length` number: `$array's length` tells you how many values are in it. If you don't know the exact position

to remove an item from, you can use an expression, in brackers, after it: `$array's ($pos - 3)`.

To see if arrays contain certain values, you can use the `contains` and `is in` operators like so: `$array contains 1` is true if it contains the number 1 anywhere, and false if it does not. `1 is in $array` is another way to write that. If you want to check if an array contains some, or all of the values, in another array, you can compare with a special `any` or `all` name on the other array: `$array contains any of (a:2,4,6)`, and `$array contains all of (a:2,4,6)` will check if `$array` contains some, or all, of the numbers 2, 4 and 6.

(Incidentally, `any` and `all` can also be used with other operators, like `is`, `is not`, `>`, `<`, `>=`, and `<=`, to compare every value in the array with a number or other value. For instance, `all of (a:2,4) >= 2` is true, as is `any of (a:2,4) >= 4`.)

Arrays may be joined by adding them together: `(a: 1, 2) + (a: 3, 4)` is the same as `(a: 1, 2, 3, 4)`. You can only join arrays to other arrays. To add a bare value to the front or back of an array, you must put it into an otherwise empty array using the (a:) macro: `$myArray + (a:5)` will make an array that's just $myArray with 5 added on the end, and `(a:0) + $myArray` is $myArray with 0 at the start.

You can make a subarray by providing a range (an array of numbers, such as those created with (range:)) as a reference - `$arr's (a:1,2)` produces an array with only the first 2 values of $arr. Additionally, you can subtract items from arrays (that is, create a copy of an array with

certain values removed) using the `-` operator: `(a:"B","C") - (a:"B")` produces `(a:"C")`. Note that multiple copies of a value in an array will all be removed by doing this: `(a:"B","B","B","C") - (a:"B")` also produces `(a:"C")`.

You may note that certain macros, like (either:), accept sequences of values. A special operator, `...`, exists which can "spread out" the values inside an array, as if they were individually placed inside the macro call. `(either: ...$array)` is a shorthand for `(either: $array's 1st, $array's 2nd, $array's 3rd)`, and so forth for as many values as there are inside the $array. Note that you can still include values after the spread: `(either: 1, ...$array, 5)` is valid and works as expected.

To summarise, the following operators work on arrays.

Operator	Purpose	Example
`is`	Evaluates to boolean `true` if both sides contain equal items in an equal order, otherwise `false`.	`(a:1,2) is (a:1,2)` (is true)
`is not`	Evaluates to `true` if both sides differ in items or ordering.	`(a:4,5) is not (a:5,4)` (is true)

`contains`	Evaluates to `true` if the left side contains the right side.	`(a:"Ape") contains "Ape"`
		`(a:(a:99)) contains (a:99)`
		`(a:1,2) contains any of (a:2,3)`

Operator	Purpose	Example

`is in`	Evaluates to `true` if the right side contains the left side.	`(a:1,2) contains all of (a:2,1)`
		`"Ape" is in (a:"Ape")`
		`(a:99) is in (a:(a:99))`
		`any of (a:2,3) is in (a:1,2)`
		`all of (a:2,1) is in (a:1,2)`

`+`	Joins arrays.	`(a:1,2) + (a:1,2)` (is `(a:1,2,1,2)`)
`-`	Subtracts arrays, producing an array containing every value in the left side but not the right.	`(a:1,1,2,3,4,5) - (a:1,2)` (is `(a:3,4,5)`)
`...`	When used in a macro call, it separates each value in the right side.	`(a: 0, ...(a:1,2,3,4), 5)` (is `(a:0,1,2,3,4,5)`)
`'s`	Obtains the item at the right numeric position, or the `length`, `any` or `all` values.	`(a:"Y","Z")'s 1st` (is "Y") `(a:4,5)'s (2)` (is 5) `(a:5,5,5)'s length` (is 3)

Operator	Purpose	Example
`of`	Obtains the item at the left numeric position, or the `length`, `any` or `all` values.	`1st of (a:"Y","O")` (is "Y") `(2) of (a:"P","S")` (is "S") `length of (a:5,5,5)` (is 3)

Boolean data

Computers can perform more than just mathematical tasks - they are also virtuosos in classical logic. Much as how arithmetic involves manipulating numbers with addition, multiplication and such, logic involves manipulating the values `true` and `false` using its own operators.

Those are not text strings - they are values as fundamental as the natural numbers. In computer science, they are both called *Booleans*, after the 19th century mathematician George Boole.

`is` is a logical operator. Just as + adds the two numbers on each side of it, `is` compares two values on each side and evaluates to `true` or `false` depending on whether they're identical. It works equally well with strings, numbers, arrays, and anything else, but beware - the string `"2"` is not equal to the number 2.

There are several other logical operators available.

Operator	Purpose	Example
`is`	Evaluates to `true` if both sides are equal, otherwise `false`.	`$bullets is 5`
`is not`	Evaluates to `true` if both sides are not equal.	`$friends is not $enemies`

Operator	Purpose	Example
`contains`	Evaluates to `true` if the left side contains the right side.	`"Fear" contains "ear"`
`is in`	Evaluates to `true` if the right side contains the left side.	`"ugh" is in "Through"`
`>`	Evaluates to `true` if the left side is greater than the right side.	`$money > 3.75`
`>=`	Evaluates to `true` if the left side is greater than or equal to the right side.	`$apples >= $carrots + 5`
`<`	Evaluates to `true` if the left side is less than the right side.	`$shoes < $people * 2`
`<=`	Evaluates to `true` if the left side is less than or equal to the right side.	`65 <= $age`
`and`	Evaluates to `true` if both sides evaluates to `true`.	`$hasFriends and $hasFamily`
`or`	Evaluates to `true` if either side is `true`.	`$fruit or $vegetable`
`not`	Flips a `true` value to a `false` value, and vice versa.	`not $stabbed`

Conditions can quickly become complicated. The best way to keep things straight is to use parentheses to group things.

Changer data

Changer commands (changers) are similar to ordinary commands, but they only have an effect when they're attached to hooks, and modify the hook in a certain manner. Macros that work like this include (text-style:), (font:), (transition:), (text-rotate:), (hook:), (click:), (link:), and more.

You can save changer commands into variables, and re-use them many times in your story:

```
(set: $robotic to (font:'Courier New'))
$robotic[Hi, it's me. Your clanky, cold friend.]
```

Alternatively, you may prefer to use the (enchant:) macro to accomplish the same thing using only hook names:

```
|robotic>[Hi, it's me. Your clanky, cold friend.]
(enchant: ?robotic, (font:'Courier New'))
```

Changers can be combined using the `+` operator: `(text-colour: red) + (font: "Courier New")[This text is red Courier New.]` styles the text using both changers at once. These combined changers, too, can be saved in variables or used with (enchant:).

```
(set: _alertText to (font:"Courier New") + (text-style: "shudder") +
(text-colour:"#e74"))
_alertText[Social alert: no one read the emails you sent yesterday.]
_alertText[Arithmetic error: I forgot my seven-times-tables.]
```

Colour data

Colours are special data values which can be provided to certain styling macros, such as (background:) or (text-colour:). You can use built=in named colour values, or create other colours using the (rgb:) or (hsl:) macros.

The built-in values consist of the following:

Value	HTML colour equivalent
`red`	#e61919
`orange`	#e68019
`yellow`	#e5e619
`lime`	#80e619
`green`	#19e619
`aqua` or `cyan`	#19e5e6
`blue`	#197fe6
`navy`	#1919e6
`purple`	#7f19e6
`magenta` or `fuchsia`	#e619e5
`white`	#fff
`black`	#000
`grey` or `gray`	#888

(These colours were chosen to be visually pleasing when used as both background colours and text colours, without the glaring intensity that certain HTML colours, like pure #f00 red, are known to exhibit.)

In addition to these values, and the (rgb:) macro, you can also use HTML hex #xxxxxx and #xxx notation to specify colours, such as `#691212` or `#a4e`. (Note that these are *not*

strings, but bare values - `(background: #a4e)` is valid, as is `(background:navy)`.) Of course, HTML hex notation is notoriously hard to read and write, so this isn't recommended.

If you want to quickly obtain a colour which is the blending of two others, you can blend them using the `+` operator: `red + orange + white` produces a blend of red and orange, tinted white. `#a4e + black` is a dim purple.

Like datamaps, colour values have a few read-only data names, which let you examine the **r**ed, **g**reen and **b**lue components that make up the colour, as well as its **h**ue, **s**aturation and **l**ightness.

Data name	Example	Meaning
`r`	`$colour's r`	The red component, a whole number from 0 to 255.
`g`	`$colour's g`	The green component, a whole number from 0 to 255.
`b`	`$colour's b`	The blue component, a whole number from 0 to 255.
`h`	`$colour's h`	The hue angle in degrees, a whole number from 0 to 359.
`s`	`$colour's s`	The saturation percentage, a fractional number from 0 to 1.
`l`	`$colour's l`	The lightness percentage, a fractional number from 0 to 1.

These values can be used in the (hsl:) and (rgb:) macros to produce further colours. Note that some of these values do not transfer one-to-one between representations! For instance, the hue of a gray is essentially irrelevant, so grays will usually have a `h` value equal to 0, even if you provided a different hue to (hsl:). Furthermore, colours with a lightness of 1 are always white, so their saturation and hue are irrelevant.

Command data

Commands are special kinds of data which perform an effect when they're placed in the passage. Most commands are created from macros placed directly in the passage, but, like all forms of data, they can be saved into variables using (set:) and (put:), and stored for later use.

Macros that produce commands include (display:), (print:), (go-to:), (save-game:), (load-game:), (link-goto:), and more.

Datamap data

There are occasions when you may need to work with collections of values that "belong" to a specific object or entity in your story - for example, a table of numeric "statistics" for a monster - or that associate a certain kind of value with another kind, such as a combination of adjectives ("slash", "thump") that change depending on the player's weapon name ("claw", "mallet") etc. You can create datamaps to keep these values together, move them around en masse, and organise them.

Datamaps are one of the two major "data structures" you can use in Harlowe. The other, arrays, are created with (a:). You'll want to use datamaps if you want to store values that directly correspond to *strings*, and whose *order* and *position* do not matter. If you need to preserve the order of the values, then an array may be better suited.

Datamaps consist of several string *name*s, each of which maps to a specific *value*. `$animals's frog` and `frog of $animals` refers to the value associated with the name 'frog'. You can add new names or change existing values by using (set:) - `(set: $animals's wolf to "howl")`.

You can express the name as a bare word if it doesn't have a space or other punctuation in it - `$animals's frog` is OK, but `$animals's komodo dragon` is not. In that case, you'll need to always supply it as a string - `$animals's "komodo dragon"`.

Datamaps may be joined by adding them together: `(dm: "goose", "honk") + (dm: "robot", "whirr")` is the same as `(dm: "goose", "honk", "robot", "whirr")`. In the event that the second datamap has the same name as the first one, it will override the first

one's value - `(dm: "dog", "woof") + (dm: "dog", "bark")` will act as `(dm: "dog", "bark")`.

You may notice that you usually need to know the names a datamap contains in order to access its values. There are certain macros which provide other ways of examining a datamap's contents: (datanames:) provides a sorted array of its names, (datavalues:) provides a sorted array of its values, and (dataentries:) provides an array of names and values.

To summarise, the following operators work on datamaps.

Operator	Purpose	Example
`is`	Evaluates to boolean `true` if both sides contain equal names and values, otherwise `false`.	`(dm:"HP",5) is (dm:"HP",5)` (is true)
`is not`	Evaluates to `true` if both sides differ in items or ordering.	`(dm:"HP",5) is not (dm:"HP",4)` (is true) `(dm:"HP",5) is not (dm:"MP",5)` (is true)
`contains`	Evaluates to `true` if the left side contains the name on the right. (To check that a datamap contains a value, try using `contains` with (datavalues:))	`(dm:"HP",5) contains "HP"` (is true) `(dm:"HP",5) contains 5` (is false)
`is in`	Evaluates to `true` if the right side contains the name on the left.	`"HP" is in (dm:"HP",5)` (is true)
`+`	Joins datamaps, using the right side's value whenever both sides contain the same name.	`(dm:"HP",5) + (dm:"MP",5)`

Operator	Purpose	Example
`'s`	Obtaining the value using the name on the right.	`(dm:"love",155)'s love` (is 155).
`of`	Obtaining the value using the name on the left.	`love of (dm:"love",155)` (is 155).

Dataset data

Arrays are useful for dealing with a sequence of related data values, especially if they have a particular order. There are occasions, however, where you don't really care about the order, and instead would simply use the array as a storage place for values - using `contains` and `is in` to check which values are inside.

Think of datasets as being like arrays, but with specific restrictions:

- You can't access any positions within the dataset (so, for instance, the `1st`, `2ndlast` and `last` aren't available, although the `length` still is) and can only use `contains` and `is in` to see whether a value is inside (or, by using `any` and `all`, many values).
- Datasets only contain unique values: adding the string "Go" to a dataset already containing "Go" will do nothing.
- Datasets are considered equal (by the `is` operator) if they have the same items, regardless of order (as they have no order).

These restrictions can be helpful in that they can stop programming mistakes from occurring - you might accidentally try to modify a position in an array, but type the name of a different array that should not be modified as such. Using a dataset for the second array, if that is what best suits it, will cause an error to occur instead of allowing this unintended operation to continue.

Operator	Purpose	Example
`is`	Evaluates to boolean `true` if both sides contain equal items, otherwise `false`.	`(ds:1,2) is (ds 2,1)` (is true)
`is not`	Evaluates to `true` if both sides differ in items.	`(ds:5,4) is not (ds:5)` (is true)

`contains`	Evaluates to `true` if the left side contains the right side.	`(ds:"Ape") contains "Ape"`
		`(ds:(ds:99)) contains (ds:99)`
		`(ds: 1,2,3) contains all of (a:2,3)`
		`(ds: 1,2,3) contains any of (a:3,4)`

`is in`	Evaluates to `true` if the right side contains the left side.	`"Ape" is in (ds:"Ape")` `(a:3,4) is in (ds:1,2,3)`
`+`	Joins datasets.	`(ds:1,2,3) + (ds:1,2,4)` (is `(ds:1,2,3,4)`)
`-`	Subtracts datasets.	`(ds:1,2,3) - (ds:1,3)` (is `(ds:2)`)
`...`	When used in a macro call, it separates each value in the right side. The dataset's values are sorted before they are spread out.	`(a: 0, ...(ds:1,2,3,4), 5)` (is `(a:0,1,2,3,4,5)`)

HookName data

A hook name is like a variable name, but with `?` replacing the `$` sigil. When given to a macro that accepts it, it signifies that *all* hooks with the given name should be affected by the macro. For instance, `(click: ?red)` will cause *all* hooks with a `<red|` or `|red>` nametag to be subject to the (click:) macro's behaviour.

In earlier Harlowe versions, it was possible to also use hook names with (set:), (put:) and (move:) to modify the text of the hooks, but macros such as (replace:) should be used to accomplish this instead.

If you only want some of the hooks with the given name to be affected, you can treat the hook name as a sort of read-only array: access its `1st` element (such as by `?red's 1st`) to only affect the first such named hook in the passage, access the `last` to affect the last, and so forth. (Even specifying an array of positions, like `?red's (a:1,3,5)`, will work.) Unlike arrays, though, you can't access their `length`, nor can you spread them with `...`.

If you need to, you cal also add hook names together to affect both at the same time: `(click: ?red + ?blue's 1st)` will affect all hooks tagged `<red|`, as well as the first hook tagged `<blue|`.

Note: if a hook name does not apply to a single hook in the given passage (for instance, if you type `?rde` instead of `?red`) then no error will be produced. This is to allow macros such as (click:) to be placed in the `header` or `footer` passages, and thus easily affect hooks in every passage, even if individual passages lack the given hook name. Of course, it means that you'll have to be extra careful while typing the hook name, as misspellings will not be easily identified by Harlowe itself.

Instant data

A few special macros in Harlowe perform actions immediately, as soon as they're evaluated. These can be used in passages, but cannot have their values saved using (set:) or (put:), or stored in data structures.

Lambda data

Suppose you want to do a complicated task with an array, like, say, convert all of its strings to lowercase, or check if its datamaps have "health" data equal to 0, or join all of its strings together into a single string. You want to be able to tell Harlowe to search for "each string where the string's 1st letter is A". You want to write a "function" for how the search is to be conducted.

Lambdas are user-created functions that let you tell certain macros, like (find:), (altered:) and (folded:), precisely how to search, alter, or combine the data provided to them.

There are several types of lambdas.

- "where" lambdas, used by the (find:) macro, are used to search for and filter data. The lambda `_item where _item's 1st is "A"` tells the macro to searches for items whose `1st` is the string "A".
- "via" lambdas, used by the (altered:) macro, are used to transform and change data. The lambda `_item via _item + "s"` tells the macro to add the string "s" to the end of each item.
- "making" lambdas, used by the (folded:) are used to build or "make" a single data value by adding something from each item to it. The lambda `_item making _total via _total + (max: _item, 0)` tells the macro to add each item to the total, but only if the item is greater than 0. (Incidentally, you can also use "where" inside a "making" lambda - you could rewrite that lambda as `_item making _total via _total + _item where _item > 0`.)
- For certain macros, like (for:), you may want to use a "where" lambda that doesn't filter out any of the values - `_item where true`, for instance, will include every item.

There is a special, more readable shorthand for this type of "where" lambda: writing just `each _item` is equivalent.

Lambdas use temp variables as "placeholders" for the actual values. For instance, in `(find: _num where _num > 2, 5,6,0)`, the temp variable `_num` is used to mean each individual value given to the macro, in turn. It will be 5, 6 and 0, respectively. Importantly, this will *not* alter any existing temp variable called `_num` - the inside of a lambda can be thought of as a hook, so just as the inner `_x` in `(set: _x to 1) |a>[ (set:_x to 2) ]` is different from the outer `_x`, the `_num` in the lambda will not affect any other `_num`.

An important feature is that you can save lambdas into variables, and reuse them in your story easily. You could, for instance, `(set: $statsReadout to (_stat making _readout via _readout + "|" + _stat's name + ":" + _stat's value))`, and then use $printStats with the (folded:) macro in different places, such as `(folded: $statsReadout, ...(dataentries: $playerStats))` for displaying the player's stats, `(folded: $statsReadout, ...(dataentries: $monsterStats))` for a monster's stats, etc.

Lambdas are named after the lambda calculus, and the "lambda" keyword used in many popular programming languages. They may seem complicated, but as long as you think of them as just a special way of writing a repeating instruction, and understand how their macros work, you may find that they are very convenient.

Number data

Number data is just numbers, which you can perform basic mathematical calculations with. You'll generally use numbers to keep track of statistics for characters, count how many times an event has occurred, and numerous other uses.

You can do all the basic mathematical operations you'd expect to numbers: `(1 + 2) / 0.25 + (3 + 2) * 0.2` evaluates to the number 13. The computer follows the normal order of operations in mathematics: first multiplying and dividing, then adding and subtracting. You can group subexpressions together and force them to be evaluated first with parentheses.

If you're not familiar with some of those symbols, here's a review, along with various other operations you can perform.

Operator	Function	Example
`+`	Addition.	`5 + 5` (is 10)
`-`	Subtraction. Can also be used to negate a number.	`5 - -5` (is 10)
`*`	Multiplication.	`5 * 5` (is 25)
`/`	Division.	`5 / 5` (is 1)
`%`	Modulo (remainder of a division).	`5 % 26` (is 1)
`>`	Evaluates to boolean `true` if the left side is greater than the right side, otherwise `false`.	`$money > 3.75`
`>=`	Evaluates to boolean `true` if the left side is greater than or equal to the right side, otherwise `false`.	`$apples >= $carrots + 5`
`<`	Evaluates to boolean `true` if the left side is less than the right side, otherwise `false`.	`$shoes < $people * 2`
`<=`	Evaluates to boolean `true` if the left side is less than or equal to the right side, otherwise `false`.	`65 <= $age`

You can only perform these operations (apart from `is`) on two pieces of data if they're both numbers. Adding the string "5" to the number 2 would produce an error, and not the number 7 nor the string "52". You must convert one side or the other using the (num:) or (text:) macros.

String data

A string is just a block of text - a bunch of text characters strung together.

When making a story, you'll mostly work with strings that you intend to insert into the passage source. If a string contains markup, then the markup will be processed when it's inserted. For instance, `"The ''biiiiig'' bellyblob"` will print as "The **biiiiig** bellyblob". Even macro calls inside strings will be processed: printing `"The (print:2*3) bears"` will print "The 6 bears". If you wish to avoid this, simply include the verbatim markup inside the string: `"`It's (exactly: as planned)`"` will print "It's (exactly: as planned)".

You can add strings together to join them: `"The" + ' former ' + "Prime Minister's"` pushes the strings together, and evaluates to "The former Prime Minister's". Notice that spaces had to be added between the words in order to produce a properly spaced final string. Also, notice that you can only add strings together. You can't subtract them, much less multiply or divide them.

Strings are similar to arrays, in that their individual characters can be accessed: `"ABC"'s 1st` evaluates to "A", `"Gosh"'s 2ndlast` evaluates to "s", and `"Exeunt"'s last` evaluates to "t". They, too, have a "length": `"Marathon"'s length` is 8. If you don't know the exact position of a character, you can use an expression, in brackers, after it: `$string's ($pos - 3)`. And, you can access a substring by providing an array of positions in place of a single position: `"Dog"'s (a: 2,3)` is "og".

Also, you can use the `contains` and `is in` operators to see if a certain string is contained within another: `"mother" contains "moth"` is true, as is `"a" is in "a"`. Again, like arrays, strings have special `any` and `all` data names which can be used with `contains` and `is in` to check all their characters - `all of $string is not "w"` is true if the string doesn't contain "w", and `$string contains any of "aeiou"` is true if the string contains those five letters.

To summarise, here are the operations you can perform on strings.

Operator	Function	Example
`+`	Joining.	`"A" + "Z"` (is "AZ")
`is`	Evaluates to boolean `true` if both sides are equal, otherwise `false`.	`$name is "Frederika"` `any of "Buxom" is "x"`
`is not`	Evaluates to boolean `true` if both sides are not equal, otherwise `false`.	`$friends is not $enemies` `all of "Gadsby" is not "e"`
`contains`	Evaluates to boolean `true` if the left side contains the right side, otherwise `false`.	`"Fear" contains "ear"`

Operator	Function	Example
`is in`	Checking if the right string contains the left string, otherwise `false`.	`"ugh" is in "Through"`
`'s`	Obtaining the character or substring at the right numeric position.	`"YO"'s 1st` (is "Y") `"PS"'s (2)` (is "S") `"ear"'s (a: 2,3)` (is "ar")
`of`	Obtaining the character at the left numeric position.	`1st of "YO"` (is "Y") `(2) of "PS"` (is "S") `(a: 2,3) of "ear"` (is "ar")

VariableToValue data

This is a special value that only (set:) and (put:) make use of. It's created by joining a variable and a value with the `to` or `into` keywords: `$emotion to 'flustered'` is an example of a VariableToValue. It exists primarily to make (set:) and (put:) more readable.

Special keywords

it keyword

This keyword is a shorthand for the closest leftmost value in an expression. It lets you write `(if: $candles < 2 and it > 5)` instead of `(if: $candles < 2 and $candles > 5)`, or `(set: $candles to it + 3)` instead of `(set: $candles to $candles +`

`3)`. (You can't, however, use it in a (put:) or (move:) macro: `(put:$red + $blue into it)` is invalid.)

Since `it` uses the closest leftmost value, `(print: $red > 2 and it < 4 and $blue > 2 and it < 4)` is the same as `(print: $red > 2 and $red < 4 and $blue > 2 and $blue < 4)`.

`it` is case-insensitive: `IT`, `iT` and `It` are all acceptable as well.

In some situations, the `it` keyword will be *inserted automatically* by Harlowe when the story runs. If you write an incomplete comparison expression where the left-hand side is missing, like `(print: $red > 2 and < 4)`, then, when running, the `it` keyword will automatically be inserted into the absent spot - producing, in this case, `(print: $red > 2 and it < 4)`. Note that in situations where the `it` keyword would not have an obvious value, such as `(print: < 4)`, an error will result nonetheless.

If the `it` keyword equals a datamap, string, array, or other "collection" data type, then you can access data values using the `its` variant - `(print: $red is 'egg' and its length is 3)` or `(set:$red to its 1st)`. Much like the `'s` operator, you can use computed values with `its` - `(if: $red's length is 3 and its $position is $value)` will work as expected.

time keyword

This keyword evaluates to the number of milliseconds passed since the passage was displayed. Its main purpose is to be used alongside changers such as (live:) or (link:).

```
(link:"Click")[(if: time > 5s)[...]]
```

, for instance, can be used to determine if 5 seconds have passed since this passage was displayed, and thus whether the player waited 5 seconds before clicking the link.

When the passage is initially being rendered, `time` will be 0.

`time` used in (display:) macros will still produce the time of the host passage, not the contained passage. So, you can't use it to determine how long the (display:)ed passage has been present in the host passage.

Special passage tags

header tag

It is often very useful to want to reuse a certain set of macro calls in every passage, or to reuse an opening block of text. You can do this by giving the passage the special tag `header`, or `footer`. All passages with these tags will have their source text included at the top (or, for `footer`, the bottom) of every passage in the story, as if by an invisible (display:) macro call.

If many passages have the `header` tag, they will all be displayed, ordered by their passage name, sorted alphabetically, and by case (capitalised names appearing before lowercase names).

footer tag

This special tag is identical to the `header` tag, except that it places the passage at the bottom of all visited passages, instead of the top.

startup tag

This special tag is similar to `header`, but it will only cause the passage to be included in the very first passage in the game.

This is intended to simplify the story testing process: if you have setup code which creates variables used throughout the entire story, you should put it in a passage with this tag, instead of the starting passage. This allows you to test your story from any passage, and, furthermore, easily change the starting passage if you wish.

All passages tagged with `startup` will run, in alphabetical order by their passage name, before the passages tagged `header` will run.

debug-header tag

This special tag is similar to the `header` tag, but only causes the passage to be included if you're running the story in debug mode.

This has a variety of uses: you can put special debug display code in this passage, which can show the status of certain variables or provide links to change the game state as you see fit, and have that code be present in every passage in the story, but only during testing.

All passages tagged with `debug-header` will run before the passages tagged `header` will run, ordered by their passage name, sorted alphabetically, and by case (capitalised names appearing before lowercase names).

debug-footer tag

This special tag is identical to the `debug-header` tag, except that it places the passage at the bottom of all visited passages, instead of the top.

All passages tagged with `debug-footer` will run, in alphabetical order by their passage name, after the passages tagged `footer` have been run.

debug-startup tag

This special tag is similar to the `startup` tag, but only causes the passage to be included if you're running the story in debug mode.

This has a variety of uses: you can put special debugging code into this passage, or set up a late game state to test, and have that code run whenever you use debug mode, no matter which passage you choose to test.

All passages tagged with `debug-startup` will run, in alphabetical order by their passage name, after the passages tagged `startup` will run.

Change log

2.1.0 changes

Bugfixes

- Now, using `(enchant:)` to change the `(text-colour:)` of `?Link` (normal links) will correctly override the default CSS link colour.
- Fixed a bug where the alternative macro spellings `(text-color:)` and `(color:)` were displayed as erroneous in the editor.
- Now, `(enchant:)` correctly displays an error when the changer provided to it includes a `(replace:)`, `(append:)` or `(prepend:)` command.
- Re-fixed the bug where `(pow:)` only accepted 1 value instead of 2, and also fixed `(sqrt:)` and the `(log:)` variants, which weren't working at all.
- Fixed a parsing bug where `5*3-2`, without whitespace around the minus sign, would break the order of operations.

Alterations

- Changed the `~~` markup to produce a strikethrough style using an `<s>` element, instead of a censor-bar style using a `<del>` element. The censor-bar style, which was used in all previous versions but not ever properly documented, was bugged to always be black even if the text colour was not black. It can be replicated in stories by simply using a `(background-colour:)` macro (preferably set to a variable) in its place.
- Removed the default `line-height` CSS for `<h1>` and other header elements, because it was causing problems with line-wrapped headers.

Additions

Debug Mode

- Added a button to hide/show the variables pane at will.
- Reduced the maximum CSS height of the variables pane from 90vh (90% of the window's height) to 40vh.
- Gave variable rows a flex-shrink of 0, which I'm told prevents rows from contracting to unreadability when the pane requires scrolling.
- The variables pane should now also list temporary variables, and their locations. This currently only lists those that have been explicitly (set:) or (put:), and ignores those that are created inside (for:) loops.

2.0.1 changes

Bugfixes

- Fixed a bug where `(enchant:)` applied to ?Page couldn't override CSS properties for `<tw-story>` (including the default background colour and colour).
- Fixed a Passage Editor display bug where the left margin obscured the first letter of lines.

Appendix

Summarised history of Harlowe's design

You may have a number of questions about various aspects of Harlowe's design. For posterity, I'd like to use this section to chronicle my motivations for most of Harlowe's major divergences from Twine 1, point out several of its outside inspirations, and describe how several of its own idiosyncracies came to be.

Macro syntax

Users of the default story format in Twine 1 know that they are "powered by TiddlyWiki", as they proudly proclaim. This reflects their codebase's origin as a fork of TiddlyWiki, a single-page-app wiki. (Twee, the command-line predecessor to Twine, is named after TiddlyWiki, and "Twine", the windowed successor, is just a clever portmanteau of "windowed" + "Twee".) Each passage is actually, from the runtime's point of view, a wiki article (a "tiddler"), though by now all of TiddlyWiki's code has been heavily developed beyond recognition.

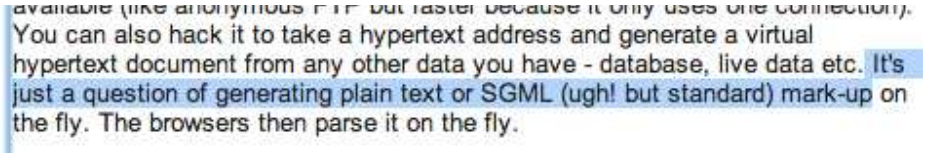
A lot of TiddlyWiki's syntax was used verbatim in the story formats, and remains to this day, such as the delineators for text styling and comments, or the `<html>` blocks to hold "raw HTML". More importantly, however, the macro syntax (and the term "macro" itself) and its delineators `<<` and `>>` remained. These were clearly based on HTML tags - the name of the macro and its arguments are separated solely by whitespace, and macros designed to enclose a block, such as `<<if>>`, required a matching `<<endif>>`. To distinguish them from actual HTML, of course, the brackets were doubled.

I'd always had misgivings with the Twine 1 macro syntax, all of which were rooted in its use of angle brackets. This choice of delimiter had already impacted other aspects of the syntax: due to a limitation of the parser in an early Twine version, the `>` and `<` symbols couldn't be used as operators inside macros, resulting in awkward `gt` and `lt` operators being created instead (though this was eventually fixed, these operators had become entrenched in what few Twine tutorials existed at the time, including Anna Anthropy's). Furthermore, due to the use of doubled brackets, nesting macros inside each other was out of the question: designing a custom macro for nesting, like `<<set $a to <<visited "Penthouse">>>>`, would be visually unreadable. Instead, Javascript functions were used for nested calls, such as `<<set $a to visited("Penthouse")>>` - a somewhat awkward and clashing second tier of programming, distinct from the macro layer.

Despite these specific issues, however, I've always found "angle brackets" delimiters ugly and visually displeasing. For one, they're not true angle brackets, like `âŸ¨ âŸ©`, but repurposed ASCII greater-than and less-than signs, which are wider than they are tall, and so barely appear to enclose their contents.

So, why are angle-brackets used by HTML, currently the most pervasive markup format on the planet? HTML was derived from SGML, a prevalent 1980s markup format and predecessor of XML, which used them in its default concrete syntax (which, unlike HTML and XML, could be configured to use different symbols).

In mid-2013, while working on the first release of Harlowe, I learned a mind-opening fact: Tim Berners-Lee, in his original announcement of World Wide Web felt it necessary to leverage the existing SGML standard and toolchains, but, *from the moment of HTML's inception*, regretted angle brackets:


available (like anonymous FTP but faster because it only uses one connection).
You can also hack it to take a hypertext address and generate a virtual
hypertext document from any other data you have - database, live data etc. It's
just a question of generating plain text or SGML (ugh! but standard) mark-up on
the fly. The browsers then parse it on the fly.

Some weeks after I learned this - a truly incisive example of how bad design decisions perpetuate themselves, even against their human designers' will, unless someone riskily strives to rebuke them - I felt motivated to take the initiative and rethink the macro syntax. Twine is a sufficiently young, niche language that making a sharp break in its most important syntax would not be too costly, and to ignore this opportunity for a new codebase would be foolish.

As mentioned above, the double-layer of function vs macro was additionally weighing on my mind, and my first move was to replace macro syntax entirely with function syntax as-is - one could place `visited("Penthouse")` in both the passage text and the interior of a macro call, and it would behave the same. (One could amusingly see parallels between XML, a HTML-derived data file format, being replaced by JSON, a Javascript-derived format, but such parallels were not on my mind at the time.) Of course, the obvious issue with not being able to embed macros directly next to readable text, such as `THUNDERuppercase($weapontype)`, as well as the equally obvious collisions with prose parens like `person(s)`, would cause me to soon tweak the function syntax to something more LISP-like: `THUNDER(uppercase:$weapontype)`. Unlike LISP, though, the presence of a `:` between name and arguments, which is mandatory even for zero-arity calls like `(stop:)`, is

the single detail that separates prose parens from code parens. (While there are still plenty of possible prose collisions, those were, in my mind at the time, vanishingly smaller in number than those caused by the prior syntax.)

The `:`, as well as the residual use of `,` for argument separation, had another characteristic that I liked: clearly indicating to the writer the structure of the syntax, where the name stops and the arguments start, and hopefully precluding people thinking of the macro call as this atomic, monolithic invocation where, say, `if` is always followed by one space and a `$`.

It was at the same time as I was reinventing the macro syntax that I was also devising the "hook" structure, which would step in at this time and obviate the need for `(endif:)` and other HTML-style end-tag macros, in a way completing the sundering of SGML's legacy. So, let's turn to discuss where hooks came from:

Hooks

It was at this point in the story that Leon was called away from the fire by impending business, leaving the rest untold.

Syntax comparison with SugarCube 1

SugarCube 1, one of the other story formats included in Twine 2, uses different markup and syntax to Harlowe. Additionally, its offered features and design philosophy also differ. This table offers a *very rough guide* to some of the major differences. (Note that a feature which is "not offered" may still be possible to implement by the addition of story CSS or JavaScript, or a combination of other extant features.)

Markup or syntax feature	**Harlowe example**
Special passages	
Startup passages	"startup" tagged passages
Pre-render passages	"header" tagged passages
Post-render passages	"footer" tagged passages

Markup or syntax feature	Harlowe example
Story sidebar modification	`(replace:?sidebar)[Sidebar code]`
Debug-only passages	"debug-header", "debug-footer" and "debug-startup" tagged passages
Styling markup	
Aligner markup	`==><=`
Strikethrough markup	`I'm <s>not</s> a bomb`
Underline markup	`Do <u>what</u>`
Subscript markup	`H<sub>2</sub>O`
Highlight markup	`<mark>battalions</mark>`
Verbatim markup	`` `Text` ``
Other styles	`(text-style:"outline")[Outlined text]`
Coloured text	`(text-colour:fuchsia)[Fuchsia text]`
Custom text styles	`(css:"border: 1px solid black")[Some text]`
Adding and saving styles	`(set: $x to (text-color:fuchsia) + (font: "Skia"))` `$x[Some text]`
White-space control	
Escaped line break	`\` at start or end of line
Collapsing whitespace markup	`{Some text}`

Markup or syntax feature	Harlowe example
"No " macro	Not offered
"Silently" macro	Not offered
Image and link markup	
Images	`<img src="http://example.org/image.png">`
Image links	`(link: '<img src="http://example.org/image.png">')[(goto:"Tower")]`
Setter links	`(link: "Text")[(set: $x to 1)(goto: "Passage")]`
Image setter links	`(link: '<img src="http://example.org/image.png">')[(set: $x to 1)(goto:"Tower")]`
Multimedia macros	
Audio macros	Not offered
Operators	
"Loosely equals" operators	Not offered
"Strictly does not equal" operator	`3 is not "3"`

Markup or syntax feature	**Harlowe example**
Inequality operators	`3 > 2`, `3 >= 3`, `2 < 3`, `2 <= 3`
"Is variable defined" operator	Not offered
"Contains" operation for arrays	`$arr contains "Pink"`, `$arr's (a:3, -1) contains "Pink"`
"Contains all" operation for arrays	`$arr contains all of (a: "pink", "green")`
"Contains any" operation for arrays	`$arr contains any of (a: "pink", "green")`
"Count" operation	`(count: $arr, "Frog")`
Increment/decrement	`$a to it + 1`, `$a to it - 1`
Spread operator	`...$arr`
Data model	
Passing	All data passed by value: arrays, datamaps and such are deep-cloned when (set:).
Type coercion	Operators and macros do not coerce types.

Markup or syntax feature	Harlowe example
Element access	
Array/string element access	`$arr's 1st`, `$arr's ($index)` (where `$index` is 1-based)
Last element access	`$arr's last`, `$arr's (-$index)` (where `$index` is 1-based)
Array/string slicing	`$arr's (a:1,4)`
Randomness	
"Either" macro/function	`(either: 1, 1, 3, 4)`
Random whole numbers	`(random: 1, 4)`
Random floating-point numbers	Not offered
Random array elements	`(either: ...$arr's (a:1,4))`
PRNG seeding	Not offered
"Shuffled" macro/function	`(shuffled: ...$arr)`
Game state	
"Time" identifier/function	`time`
Number of turns elapsed	`(history: )'s length`
Current passage's name	`(passage: )'s name`

Markup or syntax feature	Harlowe example
Previous passage's name	`(history: )'s last`
Current passage's tags	`(passage: )'s tags`
Times a passage is visited	`(count: (history: ), "Passage")`
Times a tag is visited	Not offered
Basic macros	
"Print" macro	`(print: $var)`
"Set" macro	`(set: $x to 2)`
"Unset" macro	Not offered
"Remember" macro	Not offered
"Run" macro	Not offered
Inline Javascript	`<script>document.title = "Huh?"</script>`
"Display" macro	`(display: "Duel")`, `<div>(display: "Duel")</div>`
"If" macro	`(if: $armed)[well-armed]`
"For" macro	`(for: each _dog, ...$dogs)[_dog]`
Data value macros	

Markup or syntax feature	Harlowe example
Converting to string	`(text: $num)`
Converting to number	`(num: $str)`
Creating arrays	`(a: 1, 2, 5)`
Maths macros	`(sin: 90)` etc.
Navigation macros	
"Choice" macro	Not offered
"Actions" macro	Not offered
"Go to" macro	`(goto: "Cloister")`
"Return" macro	`(link:"Go back")[(goto: (history: )'s last)]`, `(link:"Go back")[(goto: (history: )'s 3rdlast)]`
"Undo" macro	`(link-undo: "Go back")`
UI element macros	
"Click/Link" macro	`(link: "Grab")[You grabbed it]`
"Mouseover" macro	`â^£p>[A bubble] (mouseover-replace: ?p)[Pop!]`
"Mouseout" macro	`â^£p>[A bubble] (mouseout-replace: ?p)[Pop!]`
"Checkbox" macro	Not offered

Markup or syntax feature	Harlowe example
"Radio Button" macro	Not offered
"Text Area" macro	Not offered
"Textbox" macro	Not offered
DOM class macros	Not offered
Revision macros	
"Append" macro	`â^£p>[grand piano] (append: ?p)[ (lid open)]`
"Prepend" macro	`â^£c>[casket] (prepend: ?c)[open ]`
"Replace" macro	`â^£g>[green gem] (replace: ?g)[worthless glass]`
Append/prepend/replace arbitrary text	`grand piano (append: "grand piano")[ (lid open)]`
Structured programming	
Custom macros	Not offered

Markup or syntax feature	Harlowe example
Game saving	
Saving and loading macros/functions	`(savegame: "Slot 1")`, `(saved-games: ) contains "Slot 1"` and `(loadgame: "Slot 1")`
Built-in save menu	Not offered

This manual was generated at: Wed Apr 11 2018 20:01:01 GMT+1000 (EST)