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 – 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 – 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>`
Deleted/spoiler text	`~~text~~`	~~text~~	`<del>text</del>`
Emphasis	`*text*`	*text*	`<em>text</em>`
Strong emphasis	`**text**`	**text**	`<strong>text</strong>`
Superscript	`meters/second^^2^^`	meters/second^2	`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. You can print variables, arrays' items, using the (print:) macro.

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.
```

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!]
```

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.

Anonymous (or simple) hooks are hooks which have a macro or a variable attached to their front. This 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 an anonymous 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.

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.

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.
```

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*, *[...VariableToValue]*) → *Command*

Stores data values in variables.

Example usage:

```
(set: $battlecry to "Save a " + $favouritefood + " for me!")
```

Rationale:

Variables are data storage for your game. You can store data values under special names of your choosing, and refer to them later. They persist between passages, and can be used throughout the entire game in other macros, such as (if:).

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.

See also:

(push:), (move:)

The (put:) macro

(put: *VariableToValue*, *[...VariableToValue]*) → *Command*

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*]) → *Command*

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.

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.

Alternatives:

The (if:) macro is 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:)

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:)[
    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.

Note:

You may be familiar with the `if` keyword in other programming languages. Do heed this, then: the (else-if:) and (else:) macros need *not* be paired with (if:)! You can use (else-if:) and (else:) in conjunction with variable attachments, 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.)]
```

See also:

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

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:).

Note:

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:)[In silence, you gently, reverently rub their soles.]
(else:)[Before they can react, you unleash a typhoon of tickles!]
(else:)[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.

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 (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:

(datamap:), (dataset:)

The (datamap:) macro

(datamap: *[...Any]*) → *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:

`(datamap:)` creates an empty datamap. `(datamap: "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:

```
(datamap:
"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 + (datamap: "Name 1", "Value 1", "Name 2", "Value 2"))
```

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

```
(set: $element to $monsterName of (datamap:
"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:), (dataset:)

The (dataset:) macro

(dataset: *[...Any]*) → *Dataset*

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

Example usage:

`(dataset:)` creates an empty dataset, which could be filled with other values later.
`(dataset: "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: ...(dataset: ...$array))`.

See also:

(datamap:), (a:)

The (count:) macro

(count: *Any*, *Any*) → *Number*

Accepts two values, and produces the number of times the second value is inside the first value.

Example usage:

`(count: (a:1,2,3,2,1), 1)` produces 2.

Rationale:

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

Details:

If you use this with a datamap or dataset (which can't have duplicates) then an error will result.

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: (datamap:'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:)

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: (datamap:'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:)

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 (rotated:) macro

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

Identical to the typical array constructor 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.

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:

(subarray:), (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 (sorted:) macro

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

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

Example usage:

```
(set: $a to (a: 'A','C','E','G'))
(print: (sorted: ...$a))
```

Rationale:

Often, you'll be using arrays as 'decks' that will provide string values to other parts of your story in a specific order. If you want, for instance, these 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, this does not strictly use 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 and IE 10), then the strings will be sorted using English language rules (for instance, "é" comes after "e" and before "f", and regardless of the player's computer's language settings. Otherwise, it will sort using ASCII comparison (whereby "é" 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 (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:)

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.
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, replacing its contents with a newer version.

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 different 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 removed and replaced.

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:), (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.

See also:

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

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 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:)

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 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:)

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 produce a structure not unlike a normal passage link: `(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.

See also:

(loadgame:)

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*) → *Number*

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

Example usage:

`(exp: 6)` produces approximately 7.38905609893065.

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, 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 positive whole number randomly selected between the two 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*) → *Undefined*

When this macro is evaluated, a browser pop-up dialog box is shown 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")[, my maid]` adds ", my maid " to the end of every occurrence of "Emily".
- `(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")[Miss ]` adds "Miss " to the start of every occurrence of "Emily".
- `(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 a target destination with the hook's contents. The target is either a text string within the current passage, or a hook reference.

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>` hook to read "smile":

```
A song in your heart, a |face>[song] on your face.
(replace: ?face)[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:).

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:)

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: (saved-games:) 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:

(savegame:) currently has a significant limitation: it will fail if the story's variables are ever (set:) to values which aren't strings, numbers, booleans, arrays, datamaps or datasets. If, for instance, you put a changer command in a variable, like `(set: $fancytext to (font:"Arnold Bocklin"))`, (savegame:) would no longer work.

(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 (saved-games:) 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 (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 (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!]`

Rationale:

Unlike in Twine 1, 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".

See also:

(num:)

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 containining 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.

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]`

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.

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:

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

- "rumble"
- "shudder"

See also:

(css:)

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*) → *Undefined*

When this macro is evaluated, 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*) → *Undefined*

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:) → *Undefined*

When this macro is evaluated, 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.

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 (datamap:). 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.

Array data is referenced much like string characters are. You can refer to data positions using `1st`, `2nd`, `3rd`, and so forth: `$array's 1st` refers to the value in the first position. Additionally, you can use `last` to refer to the last position, `2ndlast` to refer to the second-last, and so forth. Arrays also have a `length` number: `$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)`.

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)
		`(a:4,5) is not (a:5,4)` (is

`is not`	Evaluates to `true` if both sides differ in items or ordering.	true)
`contains`	Evaluates to `true` if the left side contains the right side.	`(a:"Ape") contains "Ape"` `(a:(a:99)) contains (a:99)`
`is in`	Evaluates to `true` if the right side contains the left side.	`"Ape" is in (a:"Ape")`
`+`	Joins arrays.	`(a:1,2) + (a:1,2)` (is `(a:1,2,1,2)`)
`-`	Subtracts arrays.	`(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.	`(a:"Y","Z")'s 1st` (is "Y") `(a:4,5)'s (2)` (is 5)
`of`	Obtains the item at the left numeric position.	`1st of "YO"` (is "Y") `(2) of "PS"` (is "S")

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 $enemies`
`contains`	Evaluates to `true` if the left side contains the right side.	`"Fear" contains "ear"`
`is in`	Evaluates to `true` if the right side contains the left side.	`"ugh" is in "Through"`
`>`	Evaluates to `true` if the left side is greater than the right side.	`$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 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.]
```

Changer commands can be combined using the `+` operator: `(set: $x to (text-colour: red) + (font: "Skia"))` sets $x to a command that can make a hook's text red-coloured and in Skia. This command can be re-used over and over in your story, and is in essence a custom text style.

```
(set: $alertText to (font:"Courier New") + (text-style: "shudder") + (text-colour:"#e74"))
$alertText[This text is red shuddering Courier New.]
$alertText[Fuel warning: the petrol is upside-down.]
$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 built-in data values which can be provided to certain styling macros, such as (background:) or (text-colour:). They consist of the following values:

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.)

Additionally, you can also use HTML hex #xxxxxx and #xxx notation to specify your own 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. It's recommended that you create other colours by combining the built-in keyword values using the `+` operator: `red + orange + white` produces a blend of red and orange, tinted white. `#a4e + black` is a dim purple.

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 *names*, 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: `(datamap: "goose", "honk") + (datamap: "robot", "whirr")` is the same as `(datamap: "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 - `(datamap: "dog", "woof") + (datamap: "dog", "bark")` will act as `(datamap: "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, and (datavalues:) provides a sorted array of its 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`.	`(datamap:"HP",5) is (datamap:"HP",5)` (is true)
`is not`	Evaluates to `true` if both sides differ in items or ordering.	`(datamap:"HP",5) is not (datamap:"HP",4)` (is true) `(datamap:"HP",5) is not (datamap:"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,	`(datamap:"HP",5) contains "HP"` (is true) `(datamap:"HP",5) contains 5` (is false)

	try using `contains` with (datavalues:))	
`is in`	Evaluates to `true` if the right side contains the name on the left.	`"HP" is in (datamap:"HP",5)` (is true)
`+`	Joins datamaps, using the right side's value whenever both sides contain the same name.	`(datamap:"HP",5) + (datamap:"MP",5)`)

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.
- 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`.	`(dataset:1,2) is (dataset 2,1)` (is true)
`is not`	Evaluates to `true` if both sides differ in items.	`(dataset:5,4) is not (dataset:5)` (is true)
`contains`	Evaluates to `true` if the left side contains the right side.	`(dataset:"Ape") contains "Ape"` `(dataset:(dataset:99)) contains (dataset:99)`
`is in`	Evaluates to `true` if the right side contains the left side.	`"Ape" is in (dataset:"Ape")`
`+`	Joins datasets.	`(dataset:1,2,3) + (dataset:1,2,4)` (is `(dataset:1,2,3,4)`)

`-`	Subtracts datasets.	`(dataset:1,2,3) - (dataset:1,3)` (is `(dataset: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, ...(dataset: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.

Other macros can currently interact with named hooks as well: a hook name can be used in the (set:), (put:) and (move:) macros to change the source markup contained in them. `(set: ?red to "//Golly//")` will put the text "*Golly*" in *all* hooks with a `<red|` or `|red>` nametag, in a manner similar to using (replace:). Furthermore, you can (print:) a hook name, which joins the text of all hooks with the same name together into a single string. All of the features in this paragraph, however, are **not** recommended, and **may** be removed in a future version of Harlowe.

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.

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)`.

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"`.

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`.	`$name is "Frederika"`
`is not`	Evaluates to boolean `true` if both sides are not equal, otherwise `false`.	`$friends is not $enemies`
`contains`	Evaluates to boolean `true` if the left side contains the right side, otherwise `false`.	`"Fear" contains "ear"`
`is in`	Checking if the right string contains the left string, otherwise `false`.	`"ugh" is in "Through"`
`'s`	Obtaining the character at the right numeric position.	`"YO"'s 1st` (is "Y") `"PS"'s (2)` (is "S")
`of`	Obtaining the character at the left numeric position.	`1st of "YO"` (is "Y") `(2) of "PS"` (is "S")

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.