org-special-block-extras

A unified interface for special blocks and links: defblock

Musa Al-hassy

November 28, 2020

Abstract

The aim is to write something once using Org-mode markup then generate the markup for multiple backends. That is, ***write once, generate many!***

In particular, we are concerned with *'custom', or 'special', blocks* which delimit how a particular region of text is supposed to be formatted according to the possible export backends. In some sense, special blocks are meta-blocks. Rather than writing text in, say, LaTeX environments using LaTeX commands or in HTML `div`'s using HTML tags, we promote using Org-mode markup in special blocks —Org markup cannot be used explicitly within HTML or LaTeX environments.

Special blocks, like `centre` and `quote`, allow us to use Org-mode as the primary interface regardless of whether the final result is an HTML or PDF article; sometime we need to make our own special blocks to avoid a duplication of effort. However, this can be difficult and may require familiarity with relatively advanced ELisp concepts, such as macros and hooks; as such, users may not be willing to put in the time and instead use ad-hoc solutions.

We present a new macro, defblock, which is similar in-spirit to Lisp's standard defun except that where the latter defines functions, ours defines new special blocks for Emacs' Org-mode —as well as, simultaneously, defining new Org link types. Besides the macro, the primary contribution of this effort is an interface for special blocks that *admits* arguments and is familar to Org users —namely, we 'try to reuse' the familiar `src`-block interface, including header-args, but for special blocks.

It is hoped that the ease of creating custom special blocks will be a gateway for many Emacs users to start using Lisp.

Almost little to no attention has been afforded to making the PDF "look nice"; consider reading the HTML.

The full article may be read as a PDF or as HTML —or visit the repo. Installation instructions are below.

Contents

1 A unified interface for special blocks and links: `defblock` **3**
1.1 *What is a special block?* . . . 4
1.2 *How do I make a new link type?* . . . 6
1.3 The Core Utility: `defblock` and friends . . . 10
1.4 Modularity with `thread-blockcall` . . . 16
1.5 What's the rest of this article about? . . . 18
1.6 The Older `org-special-block-extras--`$\mathcal{X}$ Utility . . . 18

2 Editor Comments **20**

3 Folded Details —As well as boxed text and subtle colours **24**
3.1 Example: *Here's a nifty puzzle, can you figure it out?* . . . 26
3.2 Boxed Text . . . 26

4 Parallel **28**

5 Colours **30**
5.1 `latex-definitions` for hiding LaTeX declarations in HTML . . . 33

6 Nice Keystroke Renditions: `C-h h` **34**

7 *“Link Here!”* & OctoIcons **36**

8 Tooltips for Glossaries, Dictionaries, and Documentation **38**
8.1 Implementation Details: `doc` link, `documentation` block, and tooltipster . . . 39
8.2 Wait, what about the LaTeX? . . . 45
8.3 Next Steps . . . 47

9 Summary **47**
9.1 Installation Instructions . . . 47
9.2 Minimal working example . . . 48
9.3 Bye! . . . 48

The full article may be read as a PDF or as HTML —or visit the repo. Installation instructions are below.

1 A unified interface for special blocks and links: `defblock`

An Org-mode block of type $\mathcal{X}$ is a bunch of text enclosed in `#+begin_`$\mathcal{X}$ and `#+end_`$\mathcal{X}$, upon export it modifies the text —e.g., to center it, or to display it as code. We show how to make *new* blocks using a simple interface —defblock— that lets users treat $\mathcal{X}$ as a string-valued function in the style of defun.

The notable features of the system are as follows.

- Familiar `defun` syntax for making block —`defblock`
- Familiar `src` syntax for passing arguments —e.g., `:key value`
- Modular: New blocks can be made out of existing blocks really quickly using `blockcall` —similar to Lisp's `funcall`.

EmacsConf 2020 Abstract

Users will generally only make use of a few predefined *special blocks*, such as `example, centre, quote`, and will not bother with the effort required to make new ones. When new encapsulating notions are required, users will either fallback on HTML or LaTeX specific solutions, usually littered with `#+ATTR` clauses to pass around configurations or parameters.
Efforts have been exerted to mitigate the trouble of producing new special blocks. However, the issue of passing parameters is still handled in a clumsy fashion; e.g., by having parameters be expressed in a special block's content using specific keywords.
We present a novel approach to making special blocks in a familiar fashion and their use also in a familiar fashion. We achieve the former by presenting `defblock`, an anaphoric macro exceedingly similar to `defun`, and for the latter we mimic the usual `src`-block syntax for argument passing to support special blocks.
For instance, here is a sample declaration.

```
(defblock stutter () (reps 2)
  "Output the CONTENTS of the block REPS many times"
  (org-parse (s-repeat reps contents)))
```

Here is an invocation that passes an optional argument; which defaults to 2 when not given.

```
#+begin_stutter 5
Emacs for the win :-)
#+end_stutter 5
```

Upon export, to HTML or LaTeX for instance, the contents of this block are repeated (*stuttered*) 5 times. The use of `src`-like invocation may lead to a decrease in `#+ATTR` clauses.
In the presentation, we aim to show a few **practical** special blocks that users may want: A block that . . .

- translates *some selected* text —useful for multilingual blogs
- hides *some selected* text —useful for learning, quizzes
- folds/boxes text —useful in blogs for folding away details

In particular, all of these examples will be around ~5 lines long!
We also have a larger collection of more useful block types, already implemented.
The notable features of the system are as follows.

- Familiar `defun` syntax for making block —`defblock`
- Familiar `src` syntax for passing arguments —e.g., `:key value`
- Fine-grained control over export translation phases —c.f., `org-parse` above
- Modular: New blocks can be made out of existing blocks really quickly using `blockcall` —similar to Lisp's `funcall`. We will show how to fuse two blocks to make a new one, also within ~5 lines.

1.1 *What is a special block?*

An Org mode block is a region of text surrounded by `#+BEGIN_`$\mathcal{X}$ ... `#+END_`$\mathcal{X}$; they serve various purposes as summarised in the table below. However, we shall **use such blocks to execute arbitrary code on their contents**.

$\mathcal{X}$	Description
`example`	Format text verbatim, leaving markup as is
`src`	Format source code
`center`	Centre text
`quote`	Format text as a quotation —ignore line breaks
`verse`	Every line is appended with a line break
`tiny`	Render text in a small font; likewise `footnotesize`
`comment`	Completely omit the text from export

- They can be folded and unfolded in Emacs by pressing TAB in the `#+BEGIN` line.
- The contents of blocks can be highlighted as if they were of language $\mathcal{L}$ such as `org`, `html`, `latex`, `haskell`, `lisp`, `python`, ... by writing `#+BEGIN_`$\mathcal{X}$ $\mathcal{L}$ on the starting line, where $\mathcal{X}$ is the name of the block type.
- Verbatim environments `src` and `example` may be followed by switch `-n` to display line numbers for their contents.

I use snippets in my init to quickly insert special blocks (●◡●);

You can 'zoom in temporarily', *narrowing* your focus to only on a particular block, with org-narrow-to-element, `C-x n e`, to make your window only show the block. Then use `C-x n w` to *widen* your vision of the buffer's contents.

Warning! Special blocks of the same kind do not nest!

By their very nature, special blocks of *the same name* cannot be nested —e.g., try to put one `quote` block within another and see what (does not) happen.
Moreover, special blocks cannot contain unicode in their names and no underscore, '_', in their names; e.g., a special block named `quote`$_0$ will actually refer to `quote`.

Our goal is to turn Org blocks into LaTeX environments and HTML divs.
Why not use LaTeX or HTML environments directly?

- Can no longer use Org markup in such settings.
- Committed to one specific export type.

[Aside: The export syntax `@@backend:` $\mathcal{X}$`@@` inserts text $\mathcal{X}$ literally as-is precisely when the current backend being exported to is `backend`. This is useful for inserting `html` snippets or `latex` commands. We can use `@@comment:` $\mathcal{X}$`@@` to mimic inline comments ;-) —Since there is [hopefully] no backend named `comment`.
]

In general, a "special block" such as

```
#+begin_𝒳
I /love/ Emacs!
#+end_𝒳
```

Exports to LaTeX as:

```
\begin{𝒳}
I \emph{love} Emacs!
\end{𝒳}
```

Exports to HTML as:

```
<div class="𝒳">
I <em>love</em> Emacs!
</div>
```

Notice that the standard org markup is also translated according to the export type.

If the $\mathcal{X}$ environment exists in a backend —e.g., by some `\usepackage{···}` or manually with `\newenvironment{X}{···}{···}` in LaTeX— then the file will compile without error. Otherwise, you need to ensure it exists —e.g., by defining the backend formatting manually yourself.

[Aside: LaTeX packages that a user needs consistently are declared in the list `org-latex-packages-alist`. See its documentation, with `C-h o`, to learn more. To export to your own LaTeX classes, `C-h o org-latex-classes`.
]

What is an HTML 'div'?

A `div` tag defines a division or a section in an HTML document that is styled in a particular fashion or has JavaScript code applied to it. For example —placing the following in an `#+begin_export html` ··· `#+end_export`— results in a section of text that is editable by the user —i.e., one can just alter text in-place— and its foreground colour is red, while its background colour is light blue, and it has an uninformative tooltip.

Source

```
<div contenteditable="true"
     title="woah, a tool tip!"
     style="color:red; background-color:lightblue">
This is some editable text! Click me & type!
</div>
```

Result

What is a CSS 'class'?

To use a collection of style settings repeatedly, we may declare them in a `class` —which is just an alias for the ;-separated list of `attribute:value` pairs. Then our `div`'s refer to that particular `class` name.

For example, in an HTML export block, we may declare the following style class named `red`.

```
#+begin_export html
<style>
.red { color:red; }
</style>
#+end_export
```

Now, the above syntax with $\mathcal{X}$ replaced by `red` works as desired in HTML export: HTML now knows of a class named `red`.

For instance, now this

```
#+begin_red
I /love/ Emacs!
#+end_red
```

Results in:
I *love* Emacs!

This approach, however, will not work if we want to produce LaTeX and so requires a duplication of efforts. We would need to declare such formatting once for each backend.

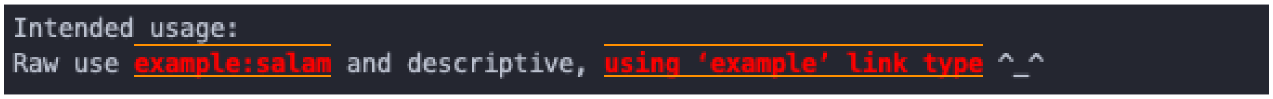
1.2 *How do I make a new link type?*

Sometimes using a block is too verbose and it'd be better to 'inline' it; for this, we use Org's link mechanism.

Use `(org-link-set-parameters params)` to add a new link type —an older obsolete method is `org-add-link-type`. The list of all supported link types is `org-link-parameters`; its documentation identifies the possibilities for `params`.

Let's produce an example link type, then discuss its code.

Intended usage: Raw use salam and descriptive, using ‘example’ link type ^_^



```
Intended usage:
Raw use example:salam and descriptive, using ‘example’ link type ^_^
```

```
1  (org-link-set-parameters
2    ;; The name of the new link type, usage: “example:label”
3    "example"
4
5    ;; When you click on such links, “let me google that for you” happens
6    :follow (lambda (label) (browse-url (concat "https://lmgtfy.com/?q=" label)))
7
8    ;; Upon export, make it a “let me google that for you” link
9    :export (lambda (label description backend)
10             (format (pcase backend
11                       ('html "<a href=\"%s\">%s</a>")
12                       ('latex "\\href{%s}{%s}")
13                       (_ "I don’t know how to export that!"))
14                     (concat "https://lmgtfy.com/?q=" label)
15                     (or description label)))
16
17   ;; These links should *never* be folded in descriptive display;
18   ;; i.e., “[[example:lable][description]]” will always appear verbatim
19   ;; and not hide the first pair [...].
20   ;; :display 'full
21
22   ;; The tooltip alongside a link
23   :help-echo (lambda (window object position)
24                (save-excursion
25                  (goto-char position)
26                  (-let* (((&plist :path :format :raw-link :contents-begin :contents-end)
27                           (cadr (org-element-context)))
28                          ;; (org-element-property :path (org-element-context))
29                          (description
30                           (when (equal format 'bracket)
31                             (copy-region-as-kill contents-begin contents-end)
32                             (substring-no-properties (car kill-ring)))))
33                    (format "“%s’’ :: Let me google ‘‘%s’’ for you -__-"
34                            raw-link (or description raw-link)))))
35
36   ;; How should these links be displayed
37   :face '(:foreground "red" :weight bold
38           :underline "orange" :overline "orange"))
```

Line 3 `"example"` Add a new `example` link type.

- If the type already exists, update it with the given arguments.

The syntax for a raw link is `example:path` and for the bracketed descriptive form `[[example:path][description]]`.

- Some of my intended uses for links including colouring text and doing nothing else, as such the terminology ‘path’ is not sufficiently generic and so I use the designation ‘label’ instead.

Line 6 `:follow` What should happen when a user clicks on such links?

This is a function taking the link path as the single argument and does whatever is necessary to "follow the link", for example find a file or display a message. In our case, we open the user's browser and go to a particular URL.

Line 9 `:export` How should this link type be exported to HTML, LaTeX, etc?

This is a three-argument function that formats the link according to the given backend, the resulting string value os placed literally into the exported file. Its arguments are:

1. `label` ⇒ the path of the link, the text after the link type prefix
2. `description` ⇒ the description of the link, if any
3. `backend` ⇒ the export format, a symbol like `html` or `latex` or `ascii`.

In our example above, we return different values depending on the `backend` value.

- If `:export` is not provided, default Org-link exportation happens.

Line 20 `:display` Should links be prettily folded away when a description is provided?

Line 23 `:help-echo` What should happen when the user's mouse is over the link?

This is **either a string or a string-valued function** that takes the current window, the current buffer object, and its position in the current window.

In our example link, we go to the position of the object, destructure the Org link's properties using `-let`, find the description of the link, if any, then provide a string based on the link's path and description.

The general textual property 'help-echo'

We may use `help-echo` to attach tooltips to arbitrary text in a file, as follows. I have found this to be useful in **metaprogramming** to have elaborated, generated, code shown as a tooltip attached to its named specification.

```
;; Nearly instantaneous display of tooltips.
(setq tooltip-delay 0)

;; Give user 30 seconds before tooltip automatically disappears.
(setq tooltip-hide-delay 300)

(defun tooltipify (phrase notification &optional underline)
  "Add a tooltip to every instance of PHRASE to show NOTIFICATION.

We only add tooltips to PHRASE as a standalone word, not as a subword.

If UNDERLINE is provided, we underline the given PHRASE so as to
provide a visual clue that it has a tooltip attched to it.

The PHRASE is taken literally; no regexp operators are recognised."
  (assert (stringp phrase))
  (assert (stringp notification))
  (save-excursion  ;; Return cursour to current-point afterwards.
    (goto-char 1)
    ;; The \b are for empty-string at the start or end of a word.
    (while (search-forward-regexp (format "\\b%s\\b" (regexp-quote phrase))
                                  (point-max) t)
      ;; (add-text-properties x y ps)
      ;; ⇒ Override properties ps for all text between x and y.
      (add-text-properties (match-beginning 0)
                           (match-end 0)
                           (list 'help-echo (s-trim notification))))))
 ;; Example use
(tooltipify
  "Line"
  "A sequential formatation of entities or the trace of a particle in linear
  ↪  motion")
```

We will use the tooltip documentation later on ^_^
Useful info on tooltips:

- Changing text properties —GNU
- Tooltips on text in Emacs —Kitchin
- Getting graphical feedback as tooltips in Emacs —Kitchin
- Defining new tooltips in Emacs —Stackoverflow

Line 37 `:face` What textual properties do these links possess?

This is **either a face or a face-valued function** that takes the current link's path label as the only argument. That is, we could change the face according to the link's label —which is what we will do for the `color` link type as in `[[color:brown][hello]]` will be rendered in brown text.

- If `:face` is not provided, the default underlined blue face for Org links is used.
- Learn more about faces!

More See `org-link-parameters` for documentation on more parameters.

1.3 The Core Utility: `defblock` and friends

To have a unified, and pleasant, interface for declaring new blocks and links, we take the following approach:

0. (*Fuse* the process of link generation and special block support into one macro, defblock which is like defun.)
1. The user writes as string-valued function named $\mathcal{X}$, possibly with arguments, that has access to a `contents` and `backend` variables.

‘defblock’ Implementation

```
(defun org-special-block-extras--org-export (x)
  "Wrap the given X in an export block for the current backend."
  (format "\n#+begin_export %s \n%s\n#+end_export\n"
  ↪  org-special-block-extras--current-backend x))

(defun org-special-block-extras--org-parse (x)
  "This should ONLY be called within an ORG-EXPORT call."
  (format "\n#+end_export\n%s\n#+begin_export %s\n" x
  ↪  org-special-block-extras--current-backend))

(cl-defmacro org-special-block-extras--defblock
  (name main-arg kwds &optional (docstring "") &rest body)
  "Declare a new special block, and link, in the style of DEFUN.

A full featured example is at the end of this documentation string.

This is an anaphoric macro that provides export support for
special blocks *and* links named NAME. Just as an Org-mode src-block
consumes as main argument the language for the src block,
our special blocks too consume a MAIN-ARG; it may be a symbol
or a cons-list consisting of a symbolic name (with which
to refer to the main argument in the definition of the block)
followed by a default value, then, optionally, any information
for a one-time setup of the associated link type.

The main arg may be a sequence of symbols separated by spaces,
and a few punctuation with the exception of comma ‘,’ since
it is a special Lisp operator. In doubt, enclose the main arg
in quotes.

Then, just as Org-mode src blocks consume key-value pairs, our
special blocks consume a number of KWDS, which is a list of the
form (key_0 value_0 ... key_n value_n).

After that is a DOCSTRING, a familar feature of DEFUN.
The docstring is displayed as part of the tooltip
for the produced link type.

Finally, the BODY is a (sequence of) Lisp forms ---no progn needed---
that may refer to the names BACKEND and CONTENTS
which refer to the current export backend and the contents
of the special block ---or the description clause of a link.

CONTENTS refers to an Org-mode parsed string; i.e.,
Org-markup is acknowledged.

In, hopefully, rare circumstances, one may refer
to RAW-CONTENTS to look at the fully unparsed contents.

----------------------------------------------------------------------

The relationship between links and special blocks:
```

```
  [ [type:label][description]]
```

2. We tell Org to please look at all special blocks

   ```
   #+begin_𝒳 main-arg :key_0 value_0 ... :key_n value_n
   contents
   #+end_𝒳
   ```

   Then, before export happens, to replace all such blocks with the *result* of calling the user's $\mathcal{X}$ function; i.e., replace them by, essentially,

   ```
   (𝒳 main-arg :key_0 value_0 ... :key_n value_n)
   ```

Implementing the hooking mechanism

The mechanism that rewrites your source…

```
(defun org-special-block-extras--pp-list (xs)
  "Given XS as (x_1 x_2 ... x_n), yield the string “x_1 x_2 ... x_n”, no parens.
  When n = 0, yield the empty string “”."
  (s-chop-suffix ")" (s-chop-prefix "(" (format "%s" (or xs "")))))

(defvar org-special-block-extras--supported-blocks nil
  "Which special blocks, defined with DEFBLOCK, are supported.")

(defvar org-special-block-extras--current-backend nil
  "A message-passing channel updated by
org-special-block-extras--support-special-blocks-with-args
and used by DEFBLOCK.")

(defun org-special-block-extras--support-special-blocks-with-args (backend)
  "Remove all headlines in the current buffer.
BACKEND is the export back-end being used, as a symbol."
  (setq org-special-block-extras--current-backend backend)
  (loop for blk in org-special-block-extras--supported-blocks
        for kwdargs = nil
        for blk-start = nil
        do (goto-char (point-min))
        (while (ignore-errors (re-search-forward (format "^\s*\\#\\+begin_%s"
        ↪  blk)))
          ;; MA: HACK: Instead of a space, it should be any non-whitespace,
          ↪  optionally;
          ;; otherwise it may accidentlly rewrite blocks with one being a
          ↪  prefix of the other!
          ; (kill-line)
          ; (error (format "(%s)" (substring-no-properties (car kill-ring))))
          (setq blk-start (line-beginning-position))
          (setq header-start (point))
          (setq body-start (1+ (line-end-position)))
          (setq kwdargs (read (format "(%s)" (buffer-substring-no-properties
          ↪  header-start (line-end-position)))))
          (setq kwdargs (--split-with (not (keywordp it)) kwdargs))
          (setq main-arg (org-special-block-extras--pp-list (car kwdargs)))
          (setq kwdargs (cadr kwdargs))
          ; (beginning-of-line) (kill-line)
          (forward-line -1)
          (re-search-forward (format "^\s*\\#\\+end_%s" blk))
          (setq contents (buffer-substring-no-properties body-start
          ↪  (line-beginning-position)))
          ; (beginning-of-line)(kill-line) ;; Hack!
          (kill-region blk-start (point))
          (insert
             (eval `(,(intern (format "org-special-block-extras--%s" blk))
                     backend
                     contents
                     main-arg
                     ,@(--map (list 'quote it) kwdargs)))
             )
          ;; the --map is so that arguments may be passed
          ;; as "this" or just ‘this’ (raw symbols)
```

‘header-args’ Implementation

Then:

```
(defvar org-special-block-extras--header-args nil
  "Alist (name plist) where ‘‘:main-arg’’ is a special plist key.

  It serves a similar role to that of Org's src ‘header-args’.

  See doc of SET-BLOCK-HEADER-ARGS for more information.")

(defmacro org-special-block-extras--set-block-header-args (blk &rest kvs)
  "Set default valuts for special block arguments.

This is similar to, and inspired by, Org-src block header-args.

Example src use:
    #+PROPERTY: header-args:Language :key value

Example block use:
    (defblock-header-args Block :main-arg mainvalue :key value)

A full, working, example can be seen by ‘‘C-h o RET defblock’’.
"
  `(add-to-list 'org-special-block-extras--header-args (list (quote ,blk)
  ↪  ,@kvs)))

(defun org-special-block-extras-short-names ()
  "Expose shorter names to the user.

Namely,

  org-special-block-extras--set-block-header-args   ↦  set-block-header-args
  org-special-block-extras--set-block-header-args   ↦  defblock
  org-special-block-extras--subtle-colors           ↦  subtle-colors
"
  (defalias 'defblock              'org-special-block-extras--defblock)
  (defalias 'set-block-header-args
  ↪  'org-special-block-extras--set-block-header-args)
  (defalias 'thread-block-call
  ↪  'org-special-block-extras--thread-blockcall)
  (defalias 'subtle-colors         'org-special-block-extras--subtle-colors))
```

This interface is essentially that of Org’s `src` blocks, with the `main-arg` being the first argument to $\mathcal{X}$ and the only argument not needing to be preceded by a key name —it is done this way to remain somewhat consistent with the Org `src` interface. The user definition of $\mathcal{X}$ decides on *how optional* the arguments actually are.

Perhaps an example will clarify things . . .

1.3.1 Example: Jasim providing in-place feedback to Bobbert

Suppose we want to devise a simple special block for editors to provide constructive feedback to authors so that the feedback appears as top-level elements of the resulting exported file —instead of comments that may accidentally not be handled by the author.

In order to showcase the multiple bells and whistles of the system, the snippet below is twice as long than it needs to be, but it is still reasonably small and accessible. (The documentation string to `defblock` is mandatory.)

```
;; We can use variable values when defining new blocks
(setq angry-red '(:foreground "red" :weight bold))

;; This is our 𝒳, “remark”.
;; As a link, it should be shown angry-red;
;; it takes two arguments: “color” and “signoff”
;; with default values being "red" and "".
;; (Assuming we already called org-special-block-extras-short-names. )
(defblock rremark
  (editor "Editor Remark" :face angry-red) (color "red" signoff "")
  "Top level (HTML & LaTeX) editorial remarks; in Emacs they're angry red."
  (format (if (equal backend 'html)
             "<strong style=\"color: %s;\">⟦%s:  %s%s⟧</strong>"
             "{\\color{%s}\\bfseries %s:  %s%s}")
          color editor contents signoff))

;; I don't want to change the definition, but I'd like to have
;; the following as personalised defaults for the “remark” block.
;; OR, I'd like to set this for links, which do not have argument options.
(set-block-header-args rremark :main-arg "Jasim Jameson" :signoff "( Aim for success! )")
```

Example use

```
The sum of the first $n$ natural numbers is $\sum_{i = 0}^n i = {n × (n + 1)
\over 2}$. Note that $n × (n + 1)$ is even.
[[rremark:Jasim Jameson][Why are you taking about “$\mathsf{even}$” here?]]
#+begin_rremark Bobbert Barakallah :signoff "Thank-you for pointing this out!" :color green
I was trying, uh ...

Yeah, to explain that ${\large n × (n + 1) \over 2}$ is always an integer.
#+end_rremark

Hence, we only need to speak about whole numbers.
[[rremark:][Then please improve your transition sentences.]]
```

Resulting rendition

The sum of the first n natural numbers is $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$. Note that $n(n+1)$ is even.

Jasim Jameson:

Why are you taking about “even” here?

(Aim for success!)

Bobbert Barakallah: I was trying, uh . . .

Yeah, to explain that $\frac{n(n+1)}{2}$ is always an integer.

Thank-you for pointing this out!

Hence, we only need to speak about whole numbers.

Jasim Jameson:

Then please improve your transition sentences.

(Aim for success!)

Notice that the result contains text —the signoff message— that the user Jasim did not write explicitly.

… Why the *stuttered* `rremark`? Because this package comes with a `remark` block that has more bells and whistles … keep reading ;-)

1.4 Modularity with `thread-blockcall`

Since defblock let's us pretend block —and link— types are string-valued functions, then one would expect that we can compose blocks *modularly* as functions compose. Somewhat analogously to funcall and thread-last, we provide a macro thread-blockcall.

Example

```
(thread-blockcall raw-contents
                  (box name)
                  (details (upcase name) :title-color "green")

=

#+begin_details NAME :title-color "green"
#+begin_box name
contents
#+end_box
#+end_details
```

Implementation

First, we need to handle the case of one block. . .

```
(cl-defmacro org-special-block-extras--blockcall (blk &optional main-arg &rest
↪  keyword-args-then-contents)
  "An anaologue to `funcall` but for blocks.

Usage: (blockcall blk-name main-arg even-many:key-values raw-contents)

One should rarely use this directly; instead use
org-special-block-extras--thread-blockcall.
"
  `(concat "#+end_export\n" (,(intern (format "org-special-block-extras--%s"
   ↪  blk))
    backend ;; defblock internal
    ; (format "\n#+begin_export html\n\n%s\n#+end_export\n" ,(car (last
    ↪  keyword-args-then-contents))) ;; contents
    ,@(last keyword-args-then-contents) ;; contents
    ,main-arg
    ,@(-drop-last 1 keyword-args-then-contents)) "\n#+begin_export"))
```

Using the above sequentially does not work due to the plumbing of `defblock`, so we handle that plumbing below . . .

```
(defmacro org-special-block-extras--thread-blockcall (body &rest forms)
  "Thread text through a number of blocks.

BODY is likely to be ‘raw-contents’, possibly with user manipulations.

Each FORMS is of the shape “(block-name main-argument
:key-value-pairs)”

(thread-blockcall x)       = x
(thread-blockcall x (f a)) = (blockcall f a x)
(thread-blockcall x f₁ f₂) ≈ (f₂ (f₁ x))

The third is a ‘≈’, and not ‘=’, because the RHS contains
‘blockcall’s as well as massages the export matter
between conseqeuctive blockcalls.

A full example:

    (org-special-block-extras--defblock nesting (name) nil
      \"Show text in a box, within details, which contains a box.\"

      (org-special-block-extras--thread-blockcall raw-contents
                        (box name)
                        (details (upcase name) :title-color \"green\")
                        (box (format \"⇒ %s ⇐\" name) :background-color
↪  \"blue\")
                        ))
"
  (if (not forms) body
     `(-let [result (org-special-block-extras--blockcall ,@(car forms) ,body)]
    ,@(loop for b in (cdr forms)
          collect `(setq result (org-special-block-extras--blockcall ,@b
```

1.5 What's the rest of this article about?

The rest of the article showcases the special blocks declared with `defblock` to allow the above presentation —with folded regions, coloured boxes, tooltips, parallel columns of text, etc.

Enjoy ;-)

1.6 The Older `org-special-block-extras--`$\mathcal{X}$ Utility

For posterity, below is the original route taken to solve the same problem. In particular, the route outlined below *may* be faster.

Why is `defblock` better?

- The approach below requires an awkward way to handle arguments, key-values.
- It requires the user to learn a new interface.
- Even if it's slower, `defblock` uses a very familiar interface and requires less Lisp mastery on the user's part.

The simplest route is to 'advise' —i.e., function patch or overload— the Org export utility for special blocks to consider calling a method `org-special-block-extras--`$\mathcal{X}$ whenever it encounters a special block named $\mathcal{X}$.

```
(advice-add #'org-html-special-block
   :before-until (apply-partially #'org-special-block-extras--advice 'html))

(advice-add #'org-latex-special-block
   :before-until (apply-partially #'org-special-block-extras--advice 'latex))
```

Here is the actual advice:

```
(defun org-special-block-extras--advice (backend blk contents _)
  "Invoke the appropriate custom block handler, if any.

A given custom block BLK has a TYPE extracted from it, then we
send the block CONTENTS along with the current export BACKEND to
the formatting function ORG-SPECIAL-BLOCK-EXTRAS--TYPE if it is
defined, otherwise, we leave the CONTENTS of the block as is.

We also support the seemingly useless blocks that have no
contents at all, not even an empty new line."
  (let* ((type    (nth 1 (nth 1 blk)))
         (handler (intern (format "org-special-block-extras--%s" type))))
    (ignore-errors (apply handler backend (or contents "") nil))))
```

To support a new block named $\mathcal{X}$:

1. Define a function `org-special-block-extras--`$\mathcal{X}$.
2. It must take two arguments:
   - `backend` ⇒ A symbol such as `'html` or `'latex`,
   - `content` ⇒ The string contents of the special block.
3. The function must return a string, possibly depending on the backend being exported to. The resulting string is inserted literally in the exported file.

4. Test out your function as in `(org-special-block-extras--𝒳 'html "some input")` —this is a quick way to find errors.

5. Enjoy ^_^

If no such function is defined, we export $\mathcal{X}$ blocks using the default mechanism, as discussed earlier, as a LaTeX environment or an HTML `div`.

An example is provided at the end of this section.
Of-course, when the user disables our mode, then we remove such advice.

```
(advice-remove #'org-html-special-block
               (apply-partially #'org-special-block-extras--advice 'html))

(advice-remove #'org-latex-special-block
               (apply-partially #'org-special-block-extras--advice 'latex))
```

1.6.1 `:argument:` Extraction

As far as I can tell, there is no way to provide arguments to special blocks. As such, the following utility looks for lines of the form `:argument: value` within the contents of a block and returns an updated contents string that no longer has such lines followed by an association list of such argument-value pairs.

```
(defun org-special-block-extras--extract-arguments (contents &rest args)
"Get list of CONTENTS string with ARGS lines stripped out and values of ARGS.

Example usage:

    (-let [(contents' . (&alist 'k₀ ... 'kₙ))
           (...extract-arguments contents 'k₀ ... 'kₙ)]
          body)

Within ‘body’, each ‘kᵢ’ refers to the ‘value’ of argument
‘:kᵢ:’ in the CONTENTS text and ‘contents’’ is CONTENTS
with all ‘:kᵢ:’ lines stripped out.

+ If ‘:k:’ is not an argument in CONTENTS, then it is assigned value NIL.
+ If ‘:k:’ is an argument in CONTENTS but is not given a value in CONTENTS,
  then it has value the empty string."
  (let ((ctnts contents)
        (values (cl-loop for a in args
                         for regex = (format ":%s:\\(.*\\)" a)
                         for v = (cadr (s-match regex contents))
                         collect (cons a v))))
    (cl-loop for a in args
             for regex = (format ":%s:\\(.*\\)" a)
             do (setq ctnts (s-replace-regexp regex "" ctnts)))
    (cons ctnts values)))
```

For example, we use this feature to indicate when a column break should happen in a `parallel` block and which person is making editorial remarks in an `remark` block.

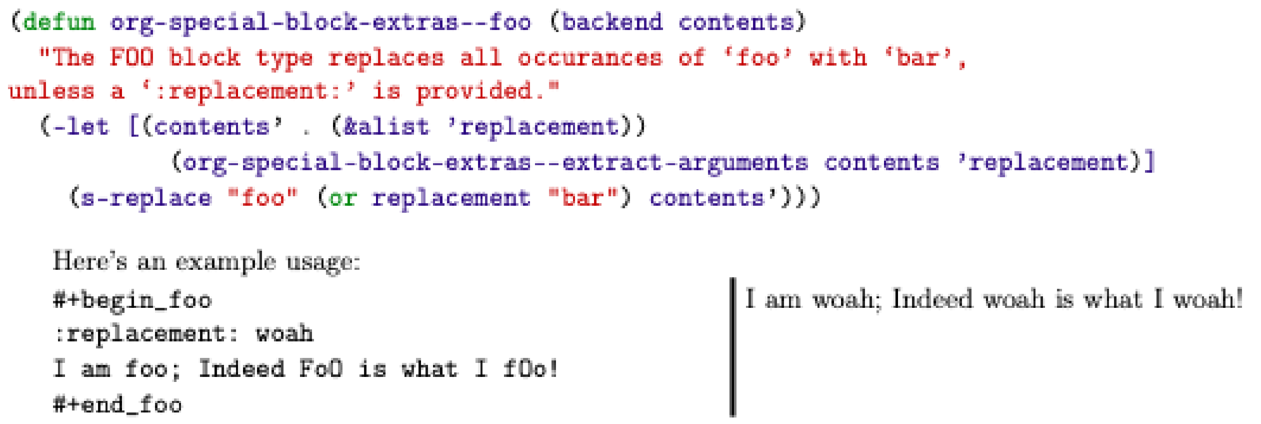
Why the `:`$\mathcal{X}$`:` notation? At the start of a line, a string of this form is coloured —I don't recall why that is— and that's a good enough reason to make use of such an existing support.

[Aside: In org-mode, ‘drawers’ are pieces of text that begin with `:my_drawer_name:` on a line by itself and end with `:end:` on a line by itself, and these delimiters allow us to fold away such regions and possibly exclude them from export. That is, drawers act as a light-weight form of blocks. Anyhow, Org colours drawer delimiters,

]

1.6.2 An Example Special Block —`foo`

Herein we show an example function `org-special-block-extras--`$\mathcal{X}$ that makes use of arguments. In a so-called `foo` block, all occurrences of the word `foo` are replaced by `bar` unless the argument `:replacement:` is given a value.



```
(defun org-special-block-extras--foo (backend contents)
  "The FOO block type replaces all occurances of ‘foo’ with ‘bar’,
unless a ‘:replacement:’ is provided."
  (-let [(contents’ . (&alist ’replacement))
          (org-special-block-extras--extract-arguments contents ’replacement)]
    (s-replace "foo" (or replacement "bar") contents’)))
```

Here’s an example usage:

```
#+begin_foo
:replacement: woah
I am foo; Indeed FoO is what I fOo!
#+end_foo
```

I am woah; Indeed woah is what I woah!

```
(defun org-special-block-extras--foo (backend contents)
  "The FOO block type replaces all occurances of ‘foo’ with ‘bar’,
unless a ‘:replacement:’ is provided."
  (-let [(contents’ . (&alist 'replacement))
          (org-special-block-extras--extract-arguments contents 'replacement)]
    (s-replace "foo" (or replacement "bar") contents’)))
```

2 Editor Comments

“Editor Comments” are intended to be top-level first-class comments in an article that are inline with the surrounding text and are delimited in such a way that they are visible but drawing attention. I first learned about this idea from Wolfram Kahl —who introduced me to Emacs many years ago. We implement editor comments as special blocks named remark.

Example

<u>This</u>

```
In LaTeX, an =remark= appears inline with the text surrounding it.
#+begin_remark Bobert
org-mode is dope, yo!
#+replacewith:
Org-mode is essentially a path toward enlightenment.
#+end_remark
Unfortunately, in the HTML rendition, the =remark= is its own paragraph and thus
separated by new lines from its surrounding text.
```

<u>Yields</u>

In LaTeX, an `remark` appears inline with the text surrounding it.

[Bobert:<u>Replace:</u> org-mode is dope, yo!

With:

Org-mode is essentially a path toward enlightenment.

]

Unfortunately, in the HTML rendition, the `remark` is its own paragraph and thus separated by new lines from its surrounding text.

All editor comments, remarks, are disabled by declaring, in your Org file:

```
#+bind: org-special-block-extras-hide-editor-comments t
```

The `#+bind:` keyword makes Emacs variables buffer-local during export —it is evaluated *after* any `src` blocks. To use it, one must declare in their Emacs init file the following line, which our mode ensures is true.

```
(setq org-export-allow-bind-keywords t)
```

(Remember to `C-c C-c` the `#+bind` to activate it, the first time it is written.)

Example: No optional arguments

[Editor Remark: *Please* **change** this section to be more, ya know, professional.
]

Example: Only providing a main argument —i.e., the remark author, the editor

[Bobert: *Please* **change** this section to be more, ya know, professional.
]

Example: Possibly with no contents:

[Bobert:
]

Example: Empty contents, no authour, nothing

[Editor Remark:
]

Example: Possibly with an empty new line

[Editor Remark:
]

Example: Possibly “malformed” replacement clauses

Forgot the thing to be replaced…

[Editor Remark:<u>Replace:</u>

<u>With:</u>

A linear one-dimensional notation; e.g., $(\Sigma i : 0..n \bullet i^2)$

]

Forgot the new replacement thing…

[Editor Remark:<u>Replace:</u> The two-dimensional notation; e.g., $\sum_{i=0}^{n} i^2$

<u>With:</u>

]

Completely lost one’s train of thought…

[Editor Remark:<u>Replace:</u>

<u>With:</u>

]

Implementing ‘remark’ with more bells and whistles

```
(defvar org-special-block-extras-hide-editor-comments nil
  "Should editor comments be shown in the output or not.")

(org-special-block-extras--defblock remark
      (editor "Editor Remark" :face '(:foreground "red" :weight bold)) (color
      ↪  "black" signoff "" strong nil)
"Format CONTENTS as an first-class editor comment according to BACKEND.

The CONTENTS string has an optional switch: If it contains a line
with having only ‘#+replacewith:’, then the text preceding this
clause should be replaced by the text after it; i.e., this is
what the EDITOR (the person editing) intends and so we fromat the
replacement instruction (to the authour) as such.

In Emacs, as links, editor remarks are shown with a bold red; but
the exported COLOR of a remark is black by default and it is not
STRONG ---i.e., bold---. There is an optional SIGNOFF message
that is appended to the remark.
"
  (-let* (;; Are we in the html backend?
          (html? (equal backend 'html))

          ;; fancy display style
          (boxed (lambda (x)
                   (if html?
                       (concat "<span style=\"border-width:1px"
                               ";border-style:solid;padding:5px\">"
                               "<strong>" x "</strong></span>")
                     (concat "\\fbox{\\bf " x "}"))))

          ;; Is this a replacement clause?
          ((this that) (s-split "\\#\\+replacewith:" contents))
          (replacement-clause? that) ;; There is a ‘that’
          (replace-keyword (if html? "&nbsp;<u>Replace:</u>"
                             "\\underline{Replace:}"))
          (with-keyword    (if html? "<u>With:</u>"
                             "\\underline{With:}"))
          (editor (format "[%s:%s" editor
                          (if replacement-clause?
                              replace-keyword
                            "")))
          (contents’ (if replacement-clause?
                         (format "%s %s %s" this
                                 (org-special-block-extras--org-export
                                 ↪  (funcall boxed with-keyword))
                                 that)
                       contents))

          ;; “[Editor Comment:”
          (edcomm-begin (funcall boxed editor))
          ;; “]”
          (edcomm-end (funcall boxed "]")))
    (setq org-export-allow-bind-keywords t) ;; So users can use “#+bind”
```

A block to make an editorial comment could be overkill in some cases; luckily defblock automatically provides an associated link type for the declared special blocks.

- Syntax: `[[remark:person_name][editorial remark]]`.
- This link type exports the same as the `remark` block type; however, in Emacs it is shown with an 'angry' —bold— red face.

Example: Terse remarks via links

`[[edcomm:Jasim][Hello, where are you?]]` **[Jasim:**	Hello, where are you? **]**
The `#+replacewith:` switch —and usual Org markup— also works with these links: `[[remark:Qasim][/'j'/ #+replacewith: /'q'/]]` **[Qasim:Replace:**	*'j'* **With:** *'q'* **]**

3 Folded Details —As well as boxed text and subtle colours

How did we fold away those implementations?

Sometimes there is a remark or a code snippet that is useful to have, but not relevant to the discussion at hand and so we want to *fold away such details.*

- 'Conversation-style' articles, where the author asks the reader questions whose answers are "folded away" so the reader can think about the exercise before seeing the answer.
- Hiding boring but important code snippets, such as a list of import declarations or a tedious implementation.

Requires: `,,#+latex_header: \usepackage{tcolorbox}`

Implementation

```
(org-special-block-extras--defblock details (title "Details") (title-color
↪  "green")
  "Enclose contents in a folded up box, for HTML.

For LaTeX, this is just a boring, but centered, box.

By default, the TITLE of such blocks is ‘‘Details’’
and its TITLE-COLOR is green.

In HTML, we show folded, details, regions with a nice greenish colour.

In the future ---i.e., when I have time---
it may be prudent to expose more aspects as arguments,
such as ‘background-color’.
"
  (format
   (pcase backend
     (`html "<details class=\"code-details\"
                 style =\"padding: 1em;
                          background-color: #e5f5e5;
                          /* background-color: pink; */
                          border-radius: 15px;
                          color: hsl(157 75% 20%);
                          font-size: 0.9em;
                          box-shadow: 0.05em 0.1em 5px 0.01em  #00000057;\">
                  <summary>
                    <strong>
                      <font face=\"Courier\" size=\"3\" color=\"%s\">
                         %s
                      </font>
                    </strong>
                  </summary>
                  %s
               </details>")
       (`latex "\\begin{quote}
                  \\begin{tcolorbox}[colback=%s,title={%s},sharp
↪  corners,boxrule=0.4pt]
                    %s
                  \\end{tcolorbox}
                \\end{quote}"))
   title-color title contents))
```

We could use `\begin{quote}\fbox{\parbox{\linewidth}{\textbf{Details:} ...}}\end{quote}`; however, this does not work well with minted for coloured source blocks. Instead, we use `tcolorbox`.

3.1 Example: *Here's a nifty puzzle, can you figure it out?*

Reductions —incidentally also called 'folds'[1]— embody primitive recursion and thus computability. For example, what does the following compute when given a whole number n?

```
(-reduce #'/ (number-sequence 1.0 n))
```

Solution

Rather than guess-then-check, let's *calculate*!

```
  (-reduce #'/ (number-sequence 1.0 n))
= ;; Lisp is strict: Evaluate inner-most expression
  (-reduce #'/ '(1.0 2.0 3.0 ... n))
= ;; Evaluate left-associating reduction
  (/ (/ (/ 1.0 2.0) ···) n)
=;; Arithmetic: (/ (/ a b) c) = (* (/ a b) (/ 1 c)) = (/ a (* b c))
  (/ 1.0 (* 2.0 3.0 ... n))
```

We have thus found the above Lisp program to compute the inverse factorial of n; i.e., $\frac{1}{n!}$.

Neato, let's do more super cool stuff ^_^
(In the Emacs Web Wowser, folded regions are displayed unfolded similar to LaTeX.)

3.2 Boxed Text

Folded regions, as implemented above, are displayed in a super neat text box which may be useful to enclose text to make it standout —without having it folded away. As such, we provide the special block box to enclosing text in boxes.

Requires: `,#+latex_header: \usepackage{tcolorbox}`

[1]See *A tutorial on the universality and expressiveness of fold* and *Unifying Structured Recursion Schemes*

Implementation

```
(org-special-block-extras--defblock box (title "") (background-color nil)
  "Enclose text in a box, possibly with a title.

By default, the box's COLOR is green for HTML and red for LaTeX,
and it has no TITLE.

The HTML export uses a padded div, whereas the LaTeX export
requires the tcolorbox pacakge.

In the future, I will likely expose more arguments.
"
  (apply #'concat
  (pcase backend
    (`html `("<div style=\"padding: 1em; background-color: "
             ,(org-special-block-extras--subtle-colors (format "%s" (or
             ↪  background-color "green")))
             ";border-radius: 15px; font-size: 0.9em"
             "; box-shadow: 0.05em 0.1em 5px 0.01em #00000057;\">"
             "<h3>" ,title "</h3>"
            ,contents "</div>"))
   (`latex `("\\begin{tcolorbox}[title={" ,title "}"
             ",colback=" ,(pp-to-string (or background-color 'red!5!white))
             ",colframe=red!75!black, colbacktitle=yellow!50!red"
             ",coltitle=red!25!black, fonttitle=\\bfseries,"
             "subtitle style={boxrule=0.4pt, colback=yellow!50!red!25!white}]"
             ,contents
             "\\end{tcolorbox}")))))
```

Example: A super boring observation presented obscurely

If you start walking —say, counterclockwise— along the unit circle from its right-most point and walk 180^o, then you will be at its left-most point. That is,

$$e^{i\pi} \;=\; -1$$

How did we get that nice light blue? What is its HTML code? That's not something I care to remember, so let's make a handy dandy utility … Now when users request a colour to `box` their text, it will be a 'subtle colour' ;-)

Implementation for Subtle Colours

```
(defun org-special-block-extras--subtle-colors (c)
  "HTML codes for common colours.

Names are very rough approximates.

   Translations from: https://www.december.com/html/spec/softhues.html"
  (pcase c
    ("teal"    "#99FFCC") ;; close to aqua
    ("brown"   "#CCCC99") ;; close to moss
    ("gray"    "#CCCCCC")
    ("purple"  "#CCCCFF")
    ("lime"    "#CCFF99") ;; brighter than 'green'
    ("green"   "#CCFFCC")
    ("blue"    "#CCFFFF")
    ("orange"  "#FFCC99")
    ("peach"   "#FFCCCC")
    ("pink"    "#FFCCFF")
    ("yellow"  "#FFFF99")
    ("custard" "#FFFFCC") ;; paler than 'yellow'
    (c c)
  ))
```

It may be useful to *fuse* the `box` and `details` concepts into one. Something for future me —or another contributor— to think about ;-)

4 Parallel

Articles can get lengthy when vertical whitespace is wasted on thin lines; instead, one could save space by using *parallel columns of text*.

Requires: `,#+latex_header: \usepackage{multicol}`

Implementation

```
(org-special-block-extras--defblock parallel (cols 2) (bar nil)
  "Place ideas side-by-side, possibly with a seperator.

There are COLS many columns, and they may be seperated by black
solid vertical rules if BAR is a non-nil value.

Writing ‘‘#+begin_parallel n :bar (any text except ‘nil’)’’
will produce a parallel of n many columns, possibly
seperated by solid rules, or a ‘‘bar’’.

The contents of the block may contain ‘#+columnbreak:’ to request
a columnbreak. This has no effect on HTML export since HTML
describes how text should be formatted on a browser, which can
dynamically shrink and grow and thus it makes no sense to have
hard columnbreaks.
"
  (let ((rule (pcase backend
               (`html  (if bar "solid" "none"))
               (`latex (if bar 2 0))))
        (contents’  (s-replace "#+columnbreak:"
                               (if (equal 'html backend) "" "\\columnbreak")
                               contents)))
  (format (pcase backend
   (`html "<div style=\"column-rule-style: %s;column-count: %s;\">%s</div>")
   (`latex "\\par \\setlength{\\columnseprule}{%s pt}
          \\begin{minipage}[t]{\\linewidth}
          \\begin{multicols}{%s}
          %s
          \\end{multicols}\\end{minipage}"))
   rule cols contents’)))
```

Going forward, it would be desirable to have the columns take a specified percentage of the available width —whereas currently it splits it uniformly. Such a feature would be useful in cases where one column is wide and the others are not.

Example

This

```
#+begin_parallel 2 :bar yes-or-any-other-text
X

Y

#+columnbreak:

Z
#+end_parallel
```

Yields

X
Y

Z

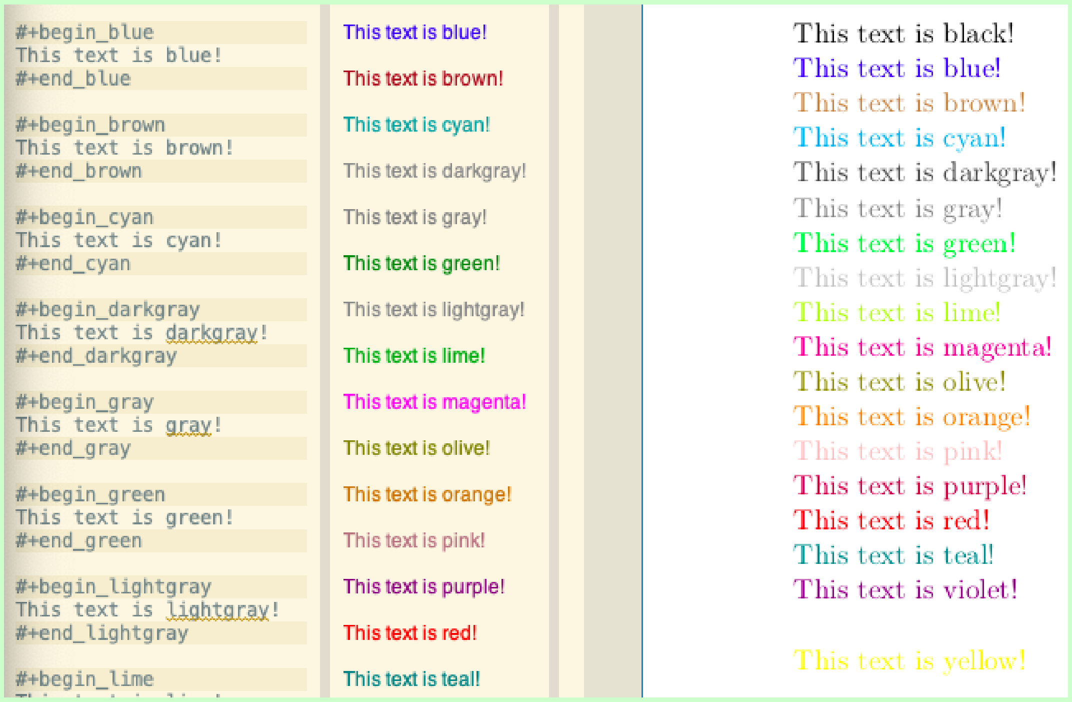
(The Emacs Web Wowser, `M-x eww`, does not display `parallel` environments as desired.)

5 Colours

Let's develop blocks for colouring text and link types for inline colouring; e.g., color and teal.

Use `M-x list-colors-display` to see a list of defined colour names in Emacs —see xcolor for the LaTeX side and htmlcolorcodes.com for the HTML side, or just visit `http://latexcolor.com/` for both.

A Picture and Block Examples



```
#+begin_blue
This text is blue!
#+end_blue

#+begin_brown
This text is brown!
#+end_brown

#+begin_cyan
This text is cyan!
#+end_cyan

#+begin_darkgray
This text is darkgray!
#+end_darkgray

#+begin_gray
This text is gray!
#+end_gray

#+begin_green
This text is green!
#+end_green

#+begin_lightgray
This text is lightgray!
#+end_lightgray

#+begin_lime
```

This text is blue!

This text is brown!

This text is cyan!

This text is darkgray!

This text is gray!

This text is green!

This text is lightgray!

This text is lime!

This text is magenta!

This text is olive!

This text is orange!

This text is pink!

This text is purple!

This text is red!

This text is teal!

This text is black!
This text is blue!
This text is brown!
This text is cyan!
This text is darkgray!
This text is gray!
This text is green!
This text is lightgray!
This text is lime!
This text is magenta!
This text is olive!
This text is orange!
This text is pink!
This text is purple!
This text is red!
This text is teal!
This text is violet!

This text is yellow!

This text is black!

This text is blue!

This text is brown!

This text is darkgray!

This text is gray!

This text is lightgray!

This text is lime!

This text is magenta!

This text is olive!

This text is orange!

This text is pink!

This text is purple!

This text is red!

This text is teal!

This text is violet!

This text is white!

This text is yellow!

Implementation of numerous colour blocks/links

We declare a list of colors that should be available on most systems. Then using this list, we evaluate the code necessary to produce the necessary functions that format special blocks.
By default, Org uses the `graphicx` LaTeX package which let's us colour text —see its documentation here. For example, in an `#+begin_export latex` block, the following produces blue coloured text.
`{  \color{blue}  This is a sample text in blue.  }`

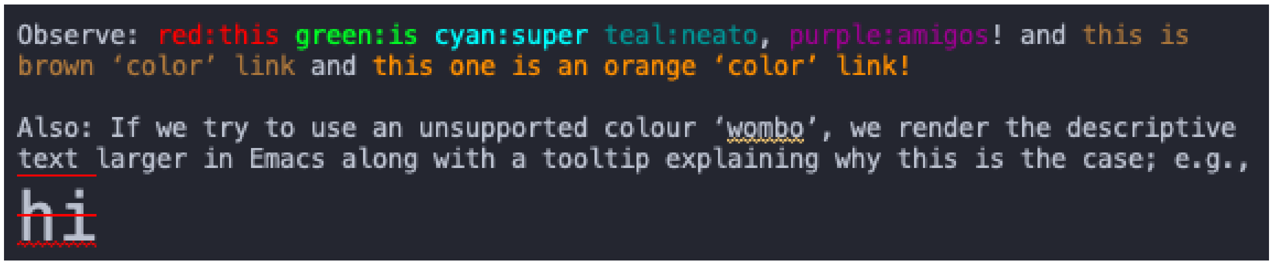
```
(defvar org-special-block-extras--colors
  '(black blue brown cyan darkgray gray green lightgray lime
          magenta olive orange pink purple red teal violet white
          yellow)
  "Colours that should be available on all systems.")

(cl-loop for colour in org-special-block-extras--colors
      do (eval (read (format
                        "(org-special-block-extras--defblock %s (_ \"\" :face
                         ↪  '(:foreground \"%s\")) nil
                          \"Show text in %s color.\"
                       (format (pcase backend
                       (`latex
↪  \"\\\\begingroup\\\\color{%s}%%s\\\\endgroup\\\\,\")
                       (_  \"<span style=\\\"color:%s;\\\">%%s</span>\"))
                       contents))" colour colour colour colour colour))))
```

Implementation of 'color'

For faster experimentation between colours, we provide a generic `color` block that consumes a color argument.

```
(org-special-block-extras--defblock color (color black    :face (lambda
↪  (colour)
             (if (member (intern colour) org-special-block-extras--colors)
                 `(:foreground ,(format "%s" colour))
               `(:height 300
                 :underline (:color "red" :style wave)
                 :overline  "red" :strike-through "red")))) nil
  "Format text according to a given COLOR, which is black by default."
  (format (pcase backend
            (`latex "\\begingroup\\color{%s}%s\\endgroup\\,")
            (`html  "<span style=\"color:%s;\">%s</span>"))
          color contents))
```

We want the syntax `red:text` to *render* 'text' with the colour red in **both** the Emacs interface and in exported backends.



Observe: red:this green:is cyan:super teal:neato, purple:amigos! and this is brown ‘color’ link and this one is an orange ‘color’ link!

Also: If we try to use an unsupported colour ‘wombo’, we render the descriptive text larger in Emacs along with a tooltip explaining why this is the case; e.g., hi

Observe:
this

is

super

neato

,
amigos

! and
this is brown ‘color’ link

and
this one is an orange ‘color’ link!

Also: If we try to use an unsupported colour ‘wombo’, we render the descriptive text larger in Emacs along with a tooltip explaining why this is the case; e.g., `[[color:wombo][hi]]`.

(Markdown does not support colour; go look at the HTML or PDF!)

5.1 `latex-definitions` for hiding LaTeX declarations in HTML

Before indicating desirable next steps, let us produce an incidentally useful special block type.

We may use LaTeX-style commands such as `{\color{red} x}` by enclosing them in $-symbols to obtain x and other commands to present mathematical formulae in HTML. This is known as the MathJax tool —Emacs' default HTML export includes it.

It is common to declare LaTeX definitions for convenience, but such declarations occur within $-delimiters and thereby produce undesirable extra whitespace. We declare the `latex_definitions` block type which avoids displaying such extra whitespace in the resulting HTML.

‘latex-definitions’ Implementation

```
(org-special-block-extras--defblock latex-definitions nil nil
  "Declare but do not display the CONTENTS according to the BACKEND."
  (format (pcase backend
            ('html "<p style=\"display:none\">\\[%s\\]</p>")
            (_ "%s"))
          raw-contents))
```

Unfortunately, MathJax does not easily support arbitrary HTML elements to occur within the `$`-delimiters —see this and this for 'workarounds'. As such, the MathJax producing the above example is rather ugly whereas its subsequent explanatory table is prettier on the writer's side.

Going forward,

it would be nice to easily have our colour links work within a mathematical special block.

Moreover,

it would be nice to extend the `color` block type to take multiple arguments, say, $\texttt{c}_1$ $\texttt{c}_2$ ... $\texttt{c}_n$ such that:

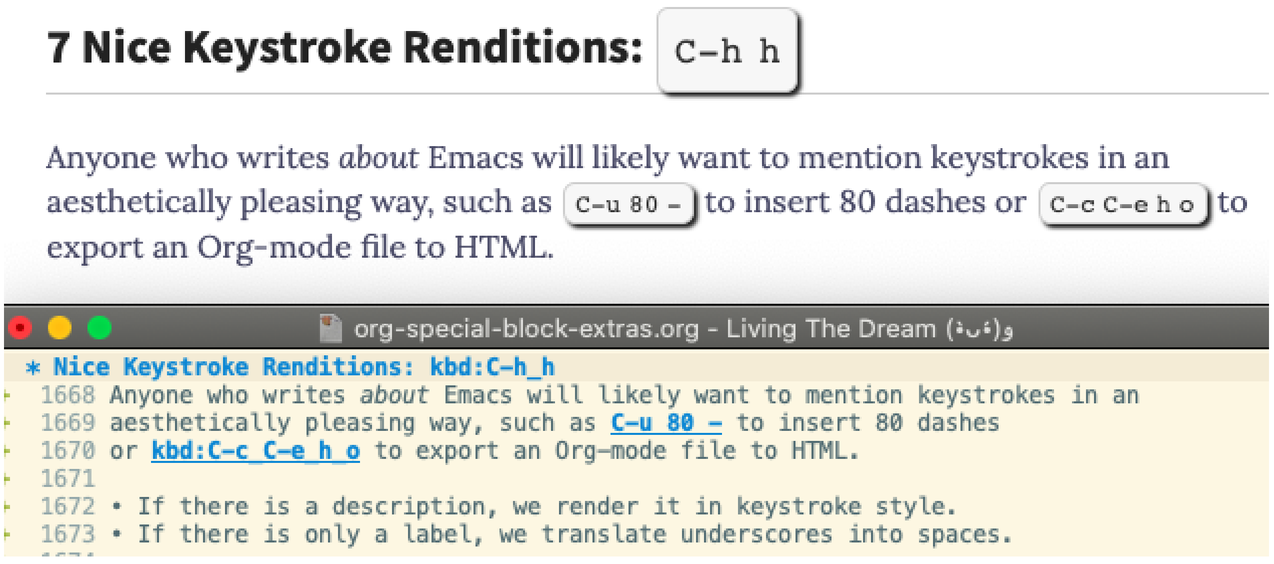
n	Behaviour
0	No colouring; likewise if no arguments altogether
1	Colour all entries using the given colour c_1
n	Paragraph –region separated by a new line– `i` is coloured by $\texttt{c}_k$ where `k = i mod n`

Besides having a colourful article, another usage I envision for this generalisation would be when rendering text in multiple languages; e.g., use red and blue to interleave Arabic poetry with its English translation.

6 Nice Keystroke Renditions: `C-h h`

Anyone who writes *about* Emacs will likely want to mention keystrokes in an aesthetically pleasing way, such as `C-u 80 -` to insert 80 dashes or `C-c C-e h o` to export an Org-mode file to HTML.

- If there is a description, we render it in keystroke style.
- If there is only a label, we translate underscores into spaces.



7 Nice Keystroke Renditions: `C-h h`

Anyone who writes *about* Emacs will likely want to mention keystrokes in an aesthetically pleasing way, such as `C-u 80 -` to insert 80 dashes or `C-c C-e h o` to export an Org-mode file to HTML.

```
* Nice Keystroke Renditions: kbd:C-h_h
Anyone who writes about Emacs will likely want to mention keystrokes in an
aesthetically pleasing way, such as C-u 80 - to insert 80 dashes
or kbd:C-c_C-e_h_o to export an Org-mode file to HTML.

• If there is a description, we render it in keystroke style.
• If there is only a label, we translate underscores into spaces.
```

Implementation

```
(org-link-set-parameters
 "kbd"
  :follow (lambda (_))
  :export (lambda (label description backend)
            (format (pcase backend
                      ('html "<kbd> %s </kbd>")
                      ('latex "\\texttt{%s}")
                      (_ "%s"))
                    (or description (s-replace "_" " " label)))))
```

The following styling rule is used to make the keystrokes displayed nicely.

```
(defvar org-special-block-extras--kbd-html-setup nil
  "Has the necessary keyboard styling HTML beeen added?")

(unless org-special-block-extras--kbd-html-setup
  (setq org-special-block-extras--kbd-html-setup t)
(setq org-html-head-extra
 (concat org-html-head-extra
"
<style>
/* From: https://endlessparentheses.com/public/css/endless.css */
/* See also:
↪ https://meta.superuser.com/questions/4788/css-for-the-new-kbd-style */
kbd
{
  -moz-border-radius: 6px;
  -moz-box-shadow: 0 1px 0 rgba(0,0,0,0.2),0 0 0 2px #fff inset;
  -webkit-border-radius: 6px;
  -webkit-box-shadow: 0 1px 0 rgba(0,0,0,0.2),0 0 0 2px #fff inset;
  background-color: #f7f7f7;
  border: 1px solid #ccc;
  border-radius: 6px;
  box-shadow: 0 1px 0 rgba(0,0,0,0.2),0 0 0 2px #fff inset;
  color: #333;
  display: inline-block;
  font-family: 'Droid Sans Mono', monospace;
  font-size: 80%;
  font-weight: normal;
  line-height: inherit;
  margin: 0 .1em;
  padding: .08em .4em;
  text-shadow: 0 1px 0 #fff;
  word-spacing: -4px;

  box-shadow: 2px 2px 2px #222; /* MA: An extra I've added. */
}
</style>")))
```

Going forward

, it would be nice to render `kbd:`$\mathcal{X}$ links in a pretty way within Emacs itself.

7 *"Link Here!"* & OctoIcons

Use the syntax `link-here:name` to create an anchor link that alters the URL with `#name` as in "" —it looks and behaves like the Github generated links for a heading. Use case: Sometimes you want to explicitly point to a particular location in an article, such as within a `#+begin_details` block, this is a possible way to do so.

Likewise, get OctoIcons with the syntax `octoicon:`$\mathcal{X}$ where $\mathcal{X}$ is one of `home, link, mail, report, tag, clock`: , , , , , .

- Within `octoicon:`$\mathcal{X}$ and `link-here:`$\mathcal{X}$ the label $\mathcal{X}$ determines the OctoIcon shown and the name of the local link to be created, respectively.
  - Descriptions, as in `[[link:label][description]]`, are ignored.
- Besides the HTML backend, such links are silently omitted.

Six OctoIcons and Implementation

The following SVGs are obtained from: https://primer.style/octicons/

```
(defvar
 org-special-block-extras--supported-octoicons
 (-partition 2
 '(
   home
   "<svg xmlns=\"http://www.w3.org/2000/svg\" viewBox=\"0 0 16
   16\" width=\"16\" height=\"16\"><path fill-rule=\"evenodd\"
   d=\"M16 9l-3-3V2h-2v2L8 1 0 9h2l1 5c0 .55.45 1 1 1h8c.55 0
   1-.45 1-1l1-5h2zm-4 5H9v-4H7v4H4L2.81 7.69 8 2.5l5.19 5.19L12
   14z\"></path></svg>"

   link
   "<svg xmlns=\"http://www.w3.org/2000/svg\" viewBox=\"0 0 16
   16\" width=\"16\" height=\"16\"><path fill-rule=\"evenodd\"
   d=\"M4 9h1v1H4c-1.5 0-3-1.69-3-3.5S2.55 3 4 3h4c1.45 0 3 1.69
   3 3.5 0 1.41-.91 2.72-2 3.25V8.59c.58-.45 1-1.27 1-2.09C10
   5.22 8.98 4 8 4H4c-.98 0-2 1.22-2 2.5S3 9 4 9zm9-3h-1v1h1c1 0
   2 1.22 2 2.5S13.98 12 13 12H9c-.98 0-2-1.22-2-2.5
   0-.83.42-1.64 1-2.09V6.25c-1.09.53-2 1.84-2 3.25C6 11.31 7.55
   13 9 13h4c1.45 0 3-1.69 3-3.5S14.5 6 13 6z\"></path></svg>"

   mail
   "<svg xmlns=\"http://www.w3.org/2000/svg\" viewBox=\"0 0 14
   16\" width=\"14\" height=\"16\"><path fill-rule=\"evenodd\"
   d=\"M0 4v8c0 .55.45 1 1 1h12c.55 0 1-.45
   1-1V4c0-.55-.45-1-1-1H1c-.55 0-1 .45-1 1zm13 0L7 9 1 4h12zM1
   5.5l4 3-4 3v-6zM2 12l3.5-3L7 10.5 8.5 9l3.5 3H2zm11-.5l-4-3
   4-3v6z\"></path></svg>"

   report
   "<svg xmlns=\"http://www.w3.org/2000/svg\" viewBox=\"0 0 16
   16\" width=\"16\" height=\"16\"><path fill-rule=\"evenodd\"
   d=\"M0 2a1 1 0 011-1h14a1 1 0 011 1v9a1 1 0 01-1 1H7l-4
   4v-4H1a1 1 0 01-1-1V2zm1 0h14v9H6.5L4 13.5V11H1V2zm6
   6h2v2H7V8zm0-5h2v4H7V3z\"></path></svg>"

   tag
   "<svg xmlns=\"http://www.w3.org/2000/svg\" viewBox=\"0 0 15
   16\" width=\"15\" height=\"16\"><path fill-rule=\"evenodd\"
   d=\"M7.73 1.73C7.26 1.26 6.62 1 5.96 1H3.5C2.13 1 1 2.13 1
   3.5v2.47c0 .66.27 1.3.73 1.77l6.06 6.06c.39.39 1.02.39 1.41
   0l4.59-4.59a.996.996 0 000-1.41L7.73 1.73zM2.38
   7.09c-.31-.3-.47-.7-.47-1.13V3.5c0-.88.72-1.59
   1.59-1.59h2.47c.42 0 .83.16 1.13.47l6.14 6.13-4.73
   4.73-6.13-6.15zM3.01 3h2v2H3V3h.01z\"></path></svg>"

   clock
   "<svg xmlns=\"http://www.w3.org/2000/svg\" viewBox=\"0 0 14
   16\" width=\"14\" height=\"16\"><path fill-rule=\"evenodd\"
   d=\"M8 8h3v2H7c-.55 0-1-.45-1-1V4h2v4zM7 2.3c3.14 0 5.7 2.56
   5.7 5.7s-2.56 5.7-5.7 5.7A5.71 5.71 0 011.3 8c0-3.14 2.56-5.7
   5.7-5.7zM7 1C3.14 1 0 4.14 0 8s3.14 7 7 7 7-3.14 7-7-3.14
   7-7-3.14-7-7-7z\"></path></svg>"))
```

Source

link-here:example-location (Click the icon and see the URL has changed!)

Result

(Click the icon and see the URL has changed!)

8 Tooltips for Glossaries, Dictionaries, and Documentation

Let's make a link type `doc` that shows a tooltip documentation —e.g., glossary or abbreviation— for a given label. E.g., user-declared Category Theory and Emacs-retrieved loop and thread-last ^_^

Full Example: ...

User enters . . .

```
#+begin_documentation Existential Angst :label "ex-angst"
A negative feeling arising from freedom and responsibility.

Also known as
1. /Existential Dread/, and
2. *Existential Anxiety*.

Perhaps a distraction, such as [[https://www.w3schools.com][visiting W3Schools]], may help ;-)

Then again, ~coding~ can be frustrating at times, maybe have
a slice of pie with maths by reading ‘‘$e^{i×π} + 1 = 0$’’ as a poem ;-)
#+end_documentation
```

Then. . . `doc:ex-angst` gives us Existential Angst, or using a description: "existence is pain"? (`[[doc:ex-angst][‘‘ex` `is pain’’?]]`)

As it stands, Emacs tooltips **only** appear after an export has happened: The export updates the dictionary variable which is used for the tooltips utility.

The `:label` is optional; when not provided it defaults to the name of the entry with spaces replaced by '_'. For more, see org-special-block-extras–documentation.

Entry names do not need to be unique, but labels do! (The labels are like the addresses used to look up an entry.)

More Examples

Supported	Example	Source
No description	Category Theory	`doc:cat`
With description and code font	`polymorphism`	`[[doc:nat-trans][=polymorphism=]]`
Fallback; e.g., arbitrary ELisp Docs	thread-first	`doc:thread-first`

Notice how hovering over items makes them appear, but to make them disappear you should click on them or scroll away. This is ideal when one wants to have multiple 'definitions' visible ;-)

Below are the entries in my personal glossary ;-)

("Category Theory" "Category Theory" "Category Theory" "Natural Transformation" "Natural Transformat

In the export, it looks like some names are repeated, but in the source one would notice that we have *labels* in both upper and lower cases and with underscores ;-) They just happen to be referring to the same *named* entry.

8.1 Implementation Details: `doc` link, `documentation` block, and tooltipster

We begin by making use of a list of documentation-glossary entries —a lightweight database of information, if you will.

```
(defvar org-special-block-extras--docs nil
  "An alist of (label name description) entries; our glossary.

Example use: (-let [(name description) (cdr (assoc 'label docs))] ···)")
```

For example, we may use `add-to-list` to add an entry only if it is not already in the list.

```
(add-to-list 'org-special-block-extras--docs
  '("cat" "Category Theory" "A theory of typed  composition; e.g., typed monoids."))
```

We may wish to use Emacs' `documentation` command to retrieve entries —this is useful for an online article that refers to unfamiliar Emacs terms ;-) To avoid copy-pasting documentation entries from one location to another, users may declare a fallback method. Besides Emacs' `documentation`, the fallback can be refer to a user's personal 'global glossary' variable —which may live in their Emacs' init file—, for more see: org-special-block-extras-docs-load-libraries

```
(defvar org-special-block-extras--docs-fallback
  (lambda (label) (list label label (documentation (intern label))))
  "The fallback method to retriving documentation or glossary entries.")
```

Implementation matter for user libraries

```
(defvar org-special-block-extras--docs-libraries
  '("documentation.org")
  "List of Org files that have ‘#+begin_documentation’ blocks that should be
  ↪  loaded
   for use with the ‘doc:𝒳’ link type.")

(cl-defun org-special-block-extras-docs-load-libraries
    (&optional (libs org-special-block-extras--docs-libraries))
"Load user's personal documentation libraries.

If no LIBS are provided, simply use those declared
org-special-block-extras--docs-libraries.

See org-special-block-extras--docs-from-libraries.
"
(interactive)
(loop for lib in libs
      do (with-temp-buffer
           (insert-file-contents lib)
           ;; doc only activates after an export
           (-let [org-export-with-broken-links t] (org-html-export-as-html))
           (kill-buffer)
           (delete-window)
           (setq org-special-block-extras--docs-from-libraries (-concat
           ↪  org-special-block-extras--docs
           ↪  org-special-block-extras--docs-from-libraries))
           (setq org-special-block-extras--docs nil))))

(defvar org-special-block-extras--docs-from-libraries nil

  "The alist of (label name description) entries loaded from the libraries.

The initial value ‘-1’ is used to indicate that no libraries have been loaded.
The ‘doc:𝒳’ link will load the libraries, possibly setting this variable to
↪  ‘nil’,
then make use of this variable when looking for documentation strings.

Interactively call org-special-block-extras-docs-load-libraries
to force your documentation libraries to be reloaded.

See also org-special-block-extras--docs-libraries.
")
```

When the mode is actually enabled, then we load the user’s libraries.

```
;; Ensure user's documentation libraries have loaded
(org-special-block-extras-docs-load-libraries)
```

Let’s keep track of where documentation comes from —either the current article or from the fallback— so that we may process it later on.

```
(defvar org-special-block-extras--docs-GLOSSARY nil
  "Which words are actually cited in the current article.

We use this listing to actually print a glossary using
‘show:GLOSSARY’.")
```

Now HTML exporting such links as tooltips and displaying them in Emacs as tooltips happens in two stages: First we check the documentation, if there is no entry, we try the fallback —if that falls, an error is reported at export time. E.g., upon export `doc:wombo` will produce a no-entry error.

‘doc’ link implementation

```
(-let [name&doc
       (lambda (lbl)
         ;; Look for ‘lbl’ from within the current buffer first, otherwise
         ↪  look among the loaded libraries.
         (-let [(_ name doc) (or (assoc lbl org-special-block-extras--docs)
         ↪  (assoc lbl org-special-block-extras--docs-from-libraries))]
           ;; If there is no documentation, try the fallback.
           (unless doc
             (setq doc
                   (condition-case nil
                       (funcall org-special-block-extras--docs-fallback lbl)
                     (error
                      (error "Error: No documentation-glossary entry for
                      ↪  ‘“%s”!"
                             lbl))))
             (setq name (nth 1 doc))
             (setq doc (nth 2 doc)))
           (list name doc)))]

(org-link-set-parameters
 "doc"
 :follow (lambda (_) ())
 :export
   `(lambda (label description backend)
     (-let [(name docs) (funcall ,name&doc label)]
       (add-to-list 'org-special-block-extras--docs-GLOSSARY
                    (list label name docs))
       (setq name (or description name))
       (pcase backend
         (`html  (format "<abbr class=\"tooltip\" title=\"%s\">%s</abbr>"
                         ;; Make it look pretty!
                         (thread-last docs
                           (s-replace "  " "&emsp;") ; Preserve newlines
                           (s-replace "\n" "<br>")   ; Preserve whitespace
                           ;; Translate Org markup
                           (s-replace-regexp "/\\(.+?\\)/" "<em>\\1</em>")
                           (s-replace-regexp "\\*\\(.+?\\)\\*"
                           ↪  "<strong>\\1</strong>")
                           (s-replace-regexp "\\~\\([^ ].*?\\)\\~"
                           ↪  "<code>\\1</code>")
                           (s-replace-regexp "=\\([^ ].*?\\)="
                           ↪  "<code>\\1</code>")
                           (s-replace-regexp "\\$\\(.+?\\)\\$" "<em>\\1</em>")
                           (s-replace-regexp
                           ↪  "\\[\\[\\(.*\\)\\]\\[\\(.*\\)\\]\\]" "\\2
                           ↪  (\\1)")
                           ;; Spacing in math mode
                           (s-replace-regexp "\\\\quad" "&#x2000;")
                           (s-replace-regexp "\\\\," "&#8194;") ;; en space
                           (s-replace-regexp "\\\\;" "&#8195;") ;; em space
                           ;; The presence of ‘\"’ in tooltips breaks things,
                           ↪  so omit them.
                           (s-replace-regexp "\\\"" "''"))
```

Things look great at the HTML side and on the Emacs side for **consuming** documented text. Besides being inconvenient, we cannot with good conscious force the average user to write Lisp as we did for the `doc:cat` entry. We turn to the problem of **producing** documentation entries with a block type interface…

‘documentation’ Implementation: An Example of Mandatory Arguments and Using ‘raw-contents’

```
(org-special-block-extras--defblock documentation
  (name (error "Documentation block: Name must be provided"))
  (label nil show nil color "green")
  "Register the dictionary entries in CONTENTS to the dictionary variable.

The dictionary variable is ‘org-special-block-extras--docs’.

A documentation entry may have its LABEL, its primary identifier,
be:
1. Omitted
2. Given as a single symbol
3. Given as a many aliases '(lbl₀ lbl₁ ... lblₙ)

The third case is for when there is no canonical label to refer to
an entry, or it is convenient to use multiple labels for the same
entry.

In all of the above cases, two additional labels are included:
The entry name with spaces replaced by underscores, and again but
all lower case.

Documentation blocks are not shown upon export;
unless SHOW is non-nil, in which case they are shown
using the ‘box’ block, with the provided COLOR passed to it.

In the futture, it may be nice to have an option to render tooltips.
That'd require the ‘doc:𝒳’ link construction be refactored via a ‘defun’."
  (unless (consp label) (setq label (list label)))
  (push (s-replace " " "_" name) label)
  (push (downcase (s-replace " " "_" name)) label)
  (loop for l in label
        do  (add-to-list 'org-special-block-extras--docs
                         (mapcar #'s-trim (list (format "%s" l) name
                         ↪  (substring-no-properties raw-contents)))))
  ;; Should the special block show something upon export?
  ""); (if show (org-special-block-extras--blockcall box name
  ↪  :background-color color raw-contents) "")
```

Thus far, Org entities are converted into HTML tags such as `<i>` for italicised text. However, HTML’s default tooltip utility —using `title="···"` in a `div`— does not render arbitrary HTML elements. Moreover, the default tooltip utility is rather slow. As such, we move to using *tooltipster*. The incantation below sets up the required magic to make this happen.

Fast, Sleek, and Beautiful Tooltips: Touching ‘org-html-head-extra’

```
(defvar org-special-block-extras--tooltip-html-setup nil
  "Has the necessary HTML beeen added?")

(unless org-special-block-extras--tooltip-html-setup
  (setq org-special-block-extras--tooltip-html-setup t))
(setq org-html-head-extra
 (concat org-html-head-extra
"
<link rel=\"stylesheet\" type=\"text/css\"
↪  href=\"https://alhassy.github.io/org-special-block-extras/tooltipster/dist/css/tooltipster

<link rel=\"stylesheet\" type=\"text/css\"
↪  href=\"https://alhassy.github.io/org-special-block-extras/tooltipster/dist/css/plugins/too
↪  />

<script type=\"text/javascript\">
    if (typeof jQuery == 'undefined') {
        document.write(unescape('%3Cscript
↪  src=\"https://code.jquery.com/jquery-1.10.0.min.js\"%3E%3C/script%3E'));
    }
</script>

 <script type=\"text/javascript\"
↪  src=\"https://alhassy.github.io/org-special-block-extras/tooltipster/dist/js/tooltipster.b

  <script>
         $(document).ready(function() {
             $('.tooltip').tooltipster({
                 theme: 'tooltipster-punk',
                 contentAsHTML: true,
                 animation: 'grow',
                 delay: [100,500],
                 // trigger: 'click'
                 trigger: 'custom',
                 triggerOpen: {
                     mouseenter: true
                 },
                 triggerClose: {
                     originClick: true,
                     scroll: true
                 }
 });
         });
     </script>

<style>
   abbr {color: red;}

   .tooltip { border-bottom: 1px dotted #000;
              color:red;
              text-decoration: none;}
</style>
")))
```

8.2 Wait, what about the LaTeX?

A PDF is essentially a fancy piece of paper, so tooltips will take on the form of glossary entries: Using `doc:`$\mathcal{X}$ will result in the word $\mathcal{X}$ being printed as a hyperlink to a glossary entry, which you the user will eventually declare using `show:GLOSSARY`; moreover, the glossary entry will also have a link back to where the `doc:`$\mathcal{X}$ was declared. E.g., defmacro and lambda.

We make a `show:`$\mathcal{X}$ link type to print the value of the variable $\mathcal{X}$ as follows, with `GLOSSARY` being a reserved name.

Implementation of ‘show’

```
(let ((whatdo (lambda (x)
                (message
                          (concat "The value of variable  %s  will be placed "
                                  "here literally upon export, "
                                  "which is: \n\n %s")
                          (s-upcase x)
                          (if (equal x "GLOSSARY")
                              (format "A cleaned up presentation of ...\n%s"
                                      org-special-block-extras--docs-GLOSSARY)
                          (pp (eval (intern x))))))))
  (org-link-set-parameters
    "show"
    :face '(:underline "green")
    :follow whatdo
    :help-echo `(lambda (_ __ position)
                  (save-excursion
                    (goto-char position)
                    (-let [(&plist :path) (cadr (org-element-context))]
                      (funcall ,whatdo path))))
    :export
     (lambda (label _description backend)
      (cond ((not (equal label "GLOSSARY")) (prin1 (eval (intern label))))
            ((equal 'html backend) "") ;; Do not print glossary in HTML
            (t
             (-let ((fstr (concat "\\vspace{1em}\\phantomsection"
                                  "\\textbf{%s}\\quad"

                                  ↪  "\\label{org-special-block-extras-glossary-%s}"
                                  "%s See page "
                                  "\\pageref{org-special-block-extras"
                                  "-glossary-declaration-site-%s}"))
                    (preserve ;; preserve whitespace
                     (lambda (x)
                       (s-replace "\n" " \\newline{\\color{white}.}"
                                  (s-replace "  " " \\quad "
                                             ;; Hack!
                                             (s-replace "&" "\\&" x))))))
               (s-join "\n\n"
                       (cl-loop for (label name doc)
                             in org-special-block-extras--docs-GLOSSARY
                             collect (format fstr name label
                                             (when doc (funcall preserve doc))
                                             label)))))))))
```

As an example, we know have generic sentences:

```
My name is show:user-full-name and I am using Emacs show:emacs-version ^_^
```

My name is Musa Al-hassy and I am using Emacs 27.1 ^_^

For example, here is a word whose documentation is obtained from Emacs rather than me being written:

thread-last. If you click on it, in the LaTeX output, you will be directed to the glossary at the end of this article —glossaries are not printed in HTML rendering.

Neato! The whitespace in the documentation is preserved in the LaTeX output as is the case for HTML.

If we decide to erase a `doc:`$\mathcal{X}$, then it should not longer appear in the printed glossary listing. Likewise, a `documentation` block has its Org markup exported according to the backend being exported to, hence if we decide to switch between different backends then only the first rendition will be used *unless* we erase the database each time after export.

```
;; Actual used glossary entries depends on the buffer; so clean up after each export
(advice-add #'org-export-dispatch
  :after (lambda (&rest _)
  (setq org-special-block-extras--docs-GLOSSARY nil
        org-special-block-extras--docs nil)))
```

8.3 Next Steps

Going forward, it'd be nice to have URLs work well upon export for `documentation` block types; whereas they currently break the HTML export.

- If an entry is referenced multiple times, such as Category Theory, then it would be nice if the glossary referred to the pages of all such locations rather than just the final one.
- The glossary current prints in order of appearance; we may want to have the option to print it in a sorted fashion.
- Perhaps use the line activation feature to provide link tooltips immediately rather than rely on exportation.
- The `show` link type could accept an arbitrary Lisp expression as a bracketed link.
- When one clicks on a `doc` documentation link, it would be nice to 'jump' to its associated `#+begin_documentation` declaration block in the current buffer, if possible.

9 Summary

The full article may be read as a PDF or as HTML —or visit the repo.

Let $\mathcal{C}$ be any of the following: `black`, `blue`, `brown`, `cyan`, `darkgray`, `gray`, `green`, `lightgray`, `lime`, `magenta`, `olive` `orange`, `pink`, `purple`, `red`, `teal`, `violet`, `white`, `yellow`.

Idea	Documentation	Link only?
Colours	$\mathcal{C}$, latex-definitions, color	
Parallel	parallel	
Editorial Comments	remark	
Folded Details	details , box	
Keystrokes		`kbd`
OctoIcons & Link Here		`octoicon`, `link-here`
Documentation-Glossary	documentation	`doc`

9.1 Installation Instructions

Manually or using quelpa:

```
;; ⟨0⟩ Download the org-special-block-extras.el file manually or using quelpa
(quelpa '(org-special-block-extras :fetcher github :repo
"alhassy/org-special-block-extras"))
```

```
;; ⟨1⟩ Have this always active in Org buffers
(add-hook #'org-mode-hook #'org-special-block-extras-mode)

;; ⟨1’⟩ Or use: “M-x org-special-block-extras-mode” to turn it on/off
```

Or with use-package:

```
(use-package org-special-block-extras
  :ensure t
  :hook (org-mode . org-special-block-extras-mode))
```

Then, provide support for a new type of special block named `foo` that, say replaces all words *foo* in a block, by declaring the following.

```
(defun org-special-block-extras--foo (backend contents)
  "The FOO block type replaces all occurances of ‘foo’ with ‘bar’,
unless a ‘:replacement:’ is provided."
  (-let [(contents’ . (&alist 'replacement))
          (org-special-block-extras--extract-arguments contents 'replacement)]
    (s-replace "foo" (or replacement "bar") contents’)))
```

9.2 Minimal working example

The following example showcases the prominent features of this library.

```
#+begin_parallel
[[color:orange][Are you excited to learn some Lisp?]] blue:yes!

Pop-quiz: How does doc:apply work?
#+end_parallel

#+begin_details
link-here:solution
Syntactically, ~(apply f '(x0 ... xN)) = (f x0 ... xN)~.

[[remark:Musa][Ain't that cool?]]

[[color:purple][We can apply a function to a list of arguments!]]
#+end_details

#+html: <br>
#+begin_box
octoicon:report Note that kbd:C-x_C-e evaluates a Lisp form!
#+end_box

#+LATEX_HEADER: \usepackage{multicol}
#+LATEX_HEADER: \usepackage{tcolorbox}
#+latex: In the LaTeX output, we have a glossary.

show:GLOSSARY
```

9.3 Bye!