org-special-block-extras

Musa Al-hassy

April 23, 2020

Warning: Incomplete!

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.

Consequently, we extend the number of block types available to the Emacs Org-mode user **without forcing the user** to learn HTML or LaTeX. Indeed, I am not a web developer and had to learn a number of HTML concepts in the process —the average Org user should not have to do so.

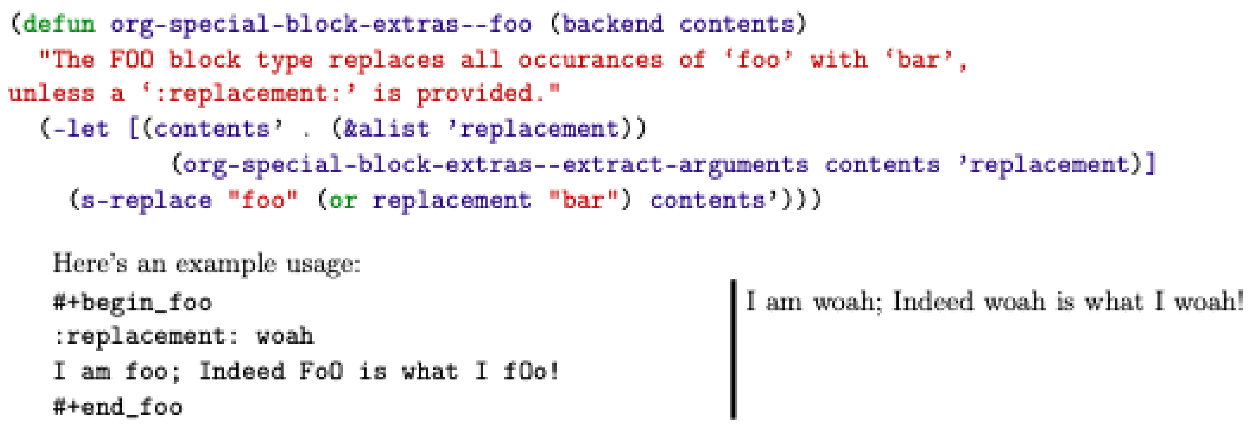
Similarly, we provide a number of 'link types' `[[linktype:label][description]]` for producing in-line coloured text and SVG "badges".

We begin with the first two sections serving as mini-tutorials on special blocks and on link types. The special block setup we use is *extensible* in that a new block named $\mathcal{C}$ will automatically be supported if the user defines a function `org-special-block-extras--`$\mathcal{C}$ that formats the text of a block. *The remaining sections are literate implementation matter, along with examples and screenshots.*

In summary, we provide 20 colour block types and 20 colour link types, an 'editor comment' block type as well as a link type, a 'details' block type, a 'parallel' multiple columns view block type, a 'link here' link type, and 8 badge link types. That is, **we provide 28 block types and 30 link types**.

(Since we're only considering HTML & LaTeX, the Github rendition of this article may not render things as desired.)

This article may be read as a PDF or as HTML or as pure Org!



```
(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!

Figure 1: Extensibility! *Plug and play support for new block types!*

Contents

1 *How do I make a new special block?* —Core Utility

1.1 *What is a special block?*

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.

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`. **]**

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.

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

To use a collection of style settings repeatedly, we may declare them in a `class` —which is just a 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.

I *love* Emacs!

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

1.2 Core Utility

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--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--`$\mathcal{X}$ `'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))
```

1.3 `:argument:` Extraction

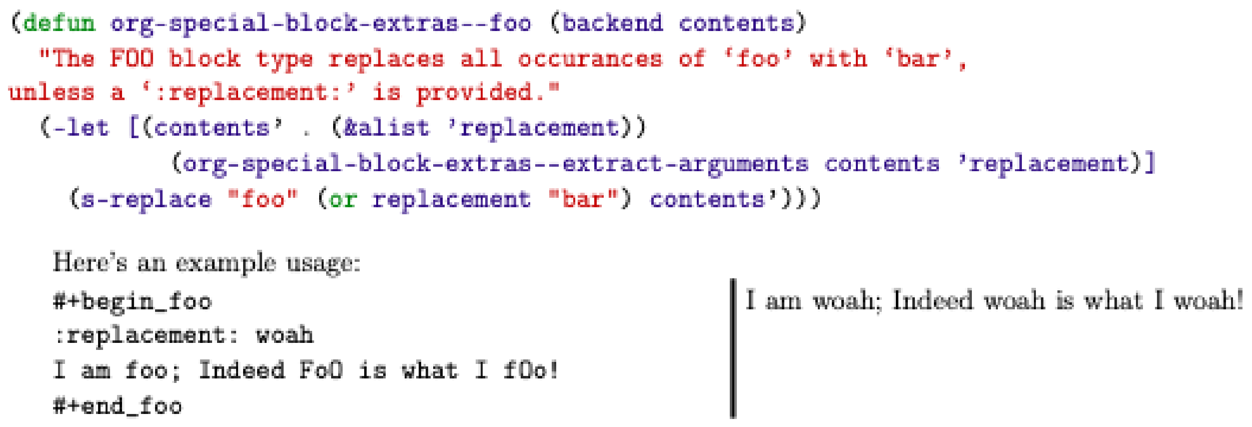
As far as I can tell, there is no way to provide arguments to special blocks. As such, the following utility looks for lines of the form `:argument: value` within the contents of a block and returns an updated contents string that no longer has such lines followed by an association list of such argument-value 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 'k_0 ... 'k_n))
           (...extract-arguments contents 'k_0 ... 'k_n)]
          body)

Within ‘body’, each ‘k_i’ refers to the ‘value’ of argument
‘:k_i:’ in the CONTENTS text and ‘contents’’ is CONTENTS
with all ‘:k_i:’ 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 `edcomm` block.

1.4 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')))
```

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!

See the implementation matter of `edcomm` or `parallel` for a more involved definition that behaves differently depending on the export backend.

1.5 Next Steps

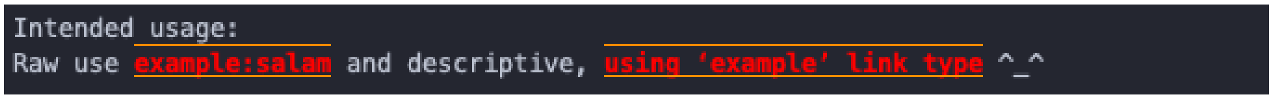
Going forward, it would be nice to have a set of switches that apply to all special blocks. For instance, `:ignore:` to simply bypass the user-defined behaviour of a block type, and `:noexport:` to zero-out a block upon export. These are super easy to do —just need a few minutes to breath. It may also be desirable to provide support for drawers, and to 'fuse' the block-type and link-type approaches used here into one macro.

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

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 ^_^
```

```
(org-link-set-parameters
  ;; The name of the new link type, usage: "example:label"
  "example"

  ;; When you click on such links, "let me google that for you" happens
  :follow (lambda (label) (browse-url (concat "https://lmgtfy.com/?q=" label)))

  ;; Upon export, make it a "let me google that for you" link
  :export (lambda (label description backend)
            (format (pcase backend
                      ('html "<a href=\"%s\">%s</a>")
                      ('latex "\\href{%s}{%s}")
                      (_ "I don't know how to export that!"))
                    (concat "https://lmgtfy.com/?q=" label)
                    (or description label)))

  ;; These links should *never* be folded in descriptive display;
  ;; i.e., "[[example:lable][description]]" will always appear verbatim
  ;; and not hide the first pair [...].
  ;; :display 'full

  ;; The tooltip alongside a link
  :help-echo (lambda (window object position)
               (save-excursion
                 (goto-char position)
                 (-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                             (or description raw-link) (pp window)))))
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.

`help-echo` is a general textual property:
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")
```

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.

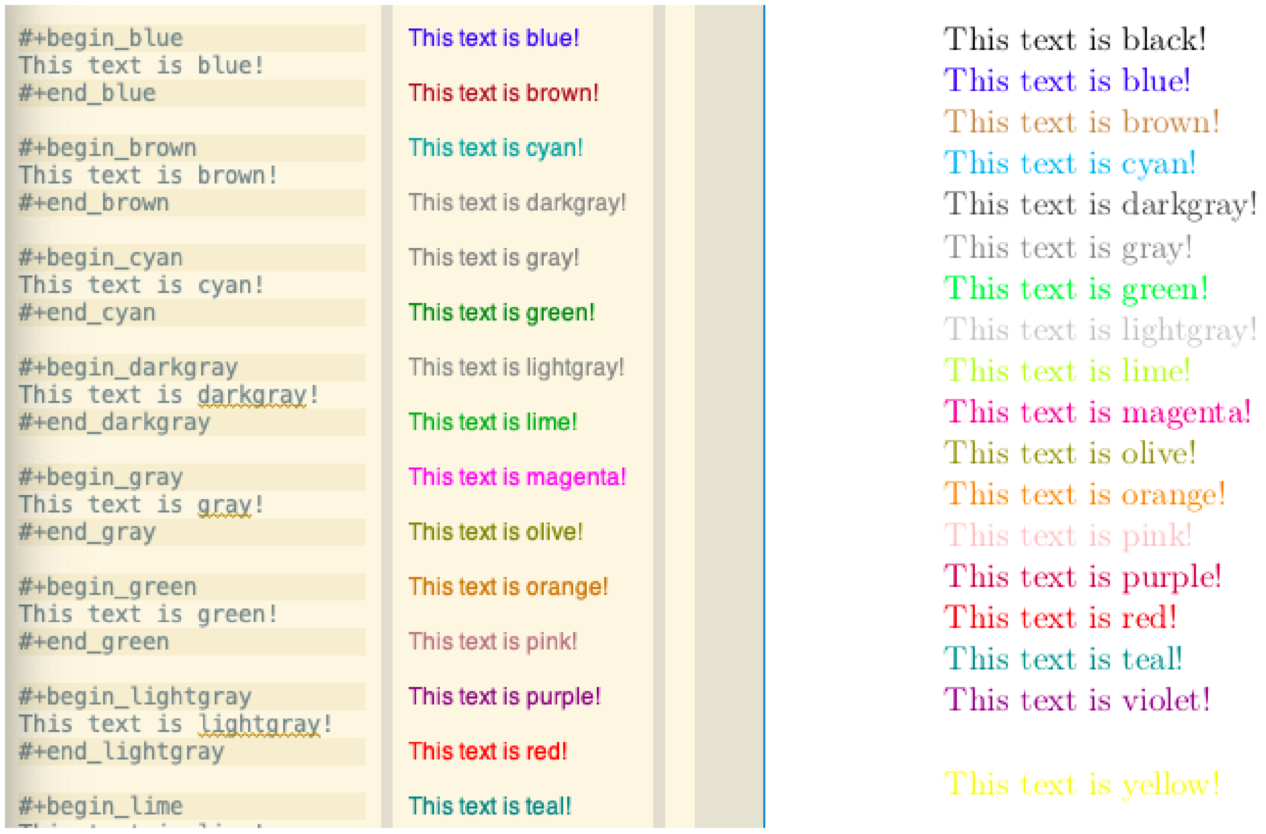
- If `:face` is not provided, the default underlined blue face for Org links is used.
- Learn more about faces!

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

3 Colours

Let's develop blocks for colouring text and link types for inline colouring.

- Use `M-x list-colors-display` to see a list of defined colour names.



```
#+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!

3.1 `org-special-block-extras--`$\mathcal{C}$ where $\mathcal{C} \in$ `org-special-block-extras--colors`

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.

```
(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
                      "(defun org-special-block-extras--%s (backend contents)
                     (format (pcase backend
                     (‘latex \"\\\\begingroup\\\\color{%s}%%s\\\\endgroup\\\\,\")
                     (‘html  \"<span style=\\\"color:%s;\\\">%%s</span>\"))
                     contents))"
                      colour colour colour))))
```

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

```
(defun org-special-block-extras--color (backend contents)
  "Format CONTENTS according to the ‘:color:’ they specify for BACKEND."
  (-let* (((contents’ . (&alist ’color))
           (org-special-block-extras--extract-arguments contents ’color))
          (block-coloring
           (intern (format "org-special-block-extras--%s" (s-trim color)))))
    (if (member (intern (s-trim color)) org-special-block-extras--colors)
        (funcall block-coloring backend contents’)
      (error "Error: “#+begin_color:%s” ⇒ Unsupported colour!" color))))
```

For example: Hello, friends!

3.2 Block Examples

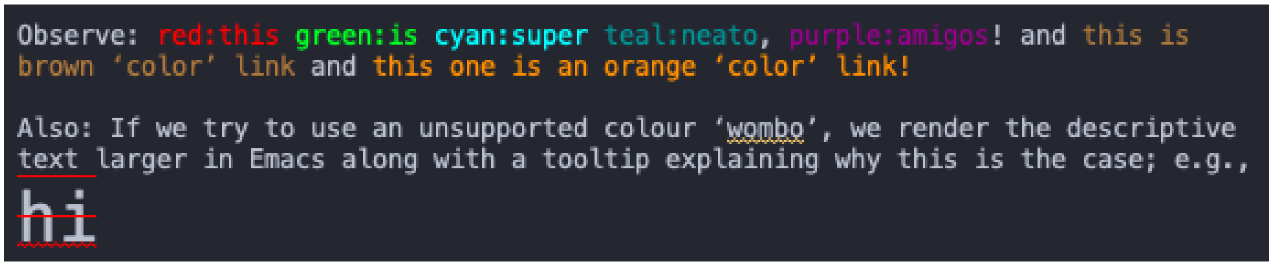
This text is black!
This text is blue!
This text is brown!
This text is cyan!
This text is darkgray!
This text is gray!
This text is green!
This text is lightgray!
This text is lime!
This text is magenta!

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!

3.3 Colour Link Types

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

```
;; [[C:text$_0$][text$_1$]] ⇒ Colour ‘text$_k$’ by C, where k is 1, if present, otherwise 0.
;; If text$_1$ is present, it is suggested to use ‘color:C’, defined below.
(cl-loop for colour in org-special-block-extras--colors
         do (org-link-set-parameters
             (format "%s" colour)
              :follow ‘(lambda (path) (message "Colouring “%s” %s." path (quote ,colour)))
              :export ‘(lambda (label description backend)
                         (-let [block-colouring
                                (intern (format "org-special-block-extras--%s" (quote
↪  ,colour)))]
                           (funcall block-colouring backend (or description label))))
              :face ‘(:foreground ,(format "%s" colour))))
```

```
;; Generic ‘color’ link type [[color:C][text]] ⇒ Colour ‘text’ by C.
;; If C is an unsupported colour, ‘text’ is rendered in large font
;; and surrounded by red lines.
(org-link-set-parameters "color"
   :follow (lambda (_))
   :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")))
 :help-echo (lambda (window object position)
              (save-excursion
                (goto-char position)
                (-let* (((&plist :path) (cadr (org-element-context))))
                  (if (member (intern path) org-special-block-extras--colors)
                      "Colour links just colour the descriptive text"
                    (format "Error: “color:%s” ⇒ Unsupported colour!" path)))))
   :export (lambda (colour description backend)
             (-let [block-colouring
                    (intern (format "org-special-block-extras--%s" colour))]
               (if (member (intern colour) org-special-block-extras--colors)
                   (funcall block-colouring backend description)
                 (error "Error: “color:%s” ⇒ Unsupported colour!" colour)))))
```

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]]`.

3.4 Next Steps

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.

```
(defun org-special-block-extras--latex-definitions (backend contents)
  "Declare but do not display the CONTENTS according to the BACKEND."
  (loop for (this that) in (-partition 2 '("<p>" ""
                                           "</p>" ""
                                           "\\{" "{"
                                           "\\}" "}"))
        do (setq contents (s-replace this that contents)))
  (format (pcase backend
            (`latex "%s")
            (`html "<p style=\"display:none\">\\[%s\\]</p>"))
          contents))
```

- Org escapes `{,}` in LaTeX export, so we need to ‘unescape’ them. This is clearly a hack.

Here is an example usage, where we declare `\LL` to produce a violet left parenthesis. We then use these to produce an example of linear quantification notation —also known as Z-notation.

$$\bigoplus_{x=a}^{b} f\,x \quad = \quad f\,(a) \oplus f\,(a+1) \oplus f\,(a+2) \oplus \cdots \oplus f\,(b)$$

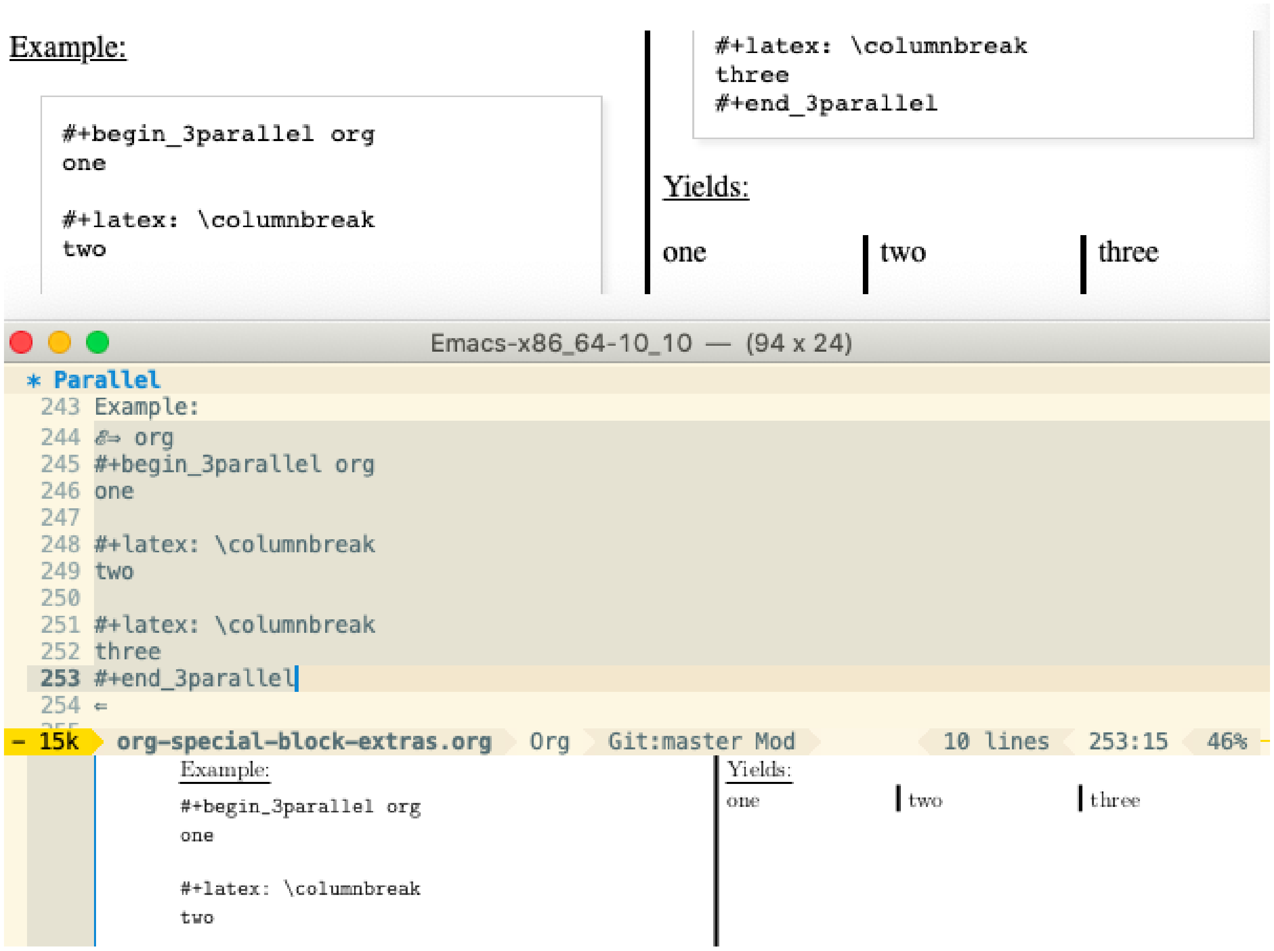
$\oplus$	*Loop sequentially with loop-bodies fused using* $\oplus$
x	*Use* x *as the name of the current element*
a	*Start with* x *being* a
b	*End with* x *being* b
$f\,x$	*At each* x *value, compute* $f\,x$

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 Z-notation 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.

4 Parallel

We want to be able to reduce the amount of whitespace noise in our articles, and so use the `parallel` block to place ideas side-by-side —with up to the chosen limit of 5 columns.



Example:

```
#+begin_3parallel org
one

#+latex: \columnbreak
two
#+latex: \columnbreak
three
#+end_3parallel
```

Yields:

one | two | three

Emacs-x86_64-10_10 — (94 x 24)

```
* Parallel
Example:
&⇒ org
#+begin_3parallel org
one

#+latex: \columnbreak
two

#+latex: \columnbreak
three
#+end_3parallel
⇐
```

```
- 15k  org-special-block-extras.org   Org   Git:master Mod        10 lines   253:15   46%
```

Example:

```
#+begin_3parallel org
one

#+latex: \columnbreak
two
```

Yields:

one | two | three

Figure 2: Displaying thoughts side-by-side ^_^ Top is browser, then Emacs, then PDF

```
#+LATEX_HEADER: \usepackage{multicol}
```

I initially used the names `parallel`n but names ending with a number n did not inherit highlighting, so I shifted the number to being a prefix instead.

- For LaTeX, new lines are used to suggest opportunities for column breaks and are needed even if explicit columnbreaks are declared.
- Use the nullary switch `: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.
- We also provide n`parallelNB` for users who want 'N'o 'B'ar separator between columns.

```
(cl-loop for cols in '("1" "2" "3" "4" "5")
      do (cl-loop for rule in '("solid" "none")
      do (eval (read (concat
"(defun org-special-block-extras--" cols "parallel"
(if (equal rule "solid") "" "NB")
"(backend contents)"
"(format (pcase backend"
"(`html \"<div style=\\\"column-rule-style:" rule ";column-count:" cols ";\\\"%s</div>\")"
"(`latex \"\\\\par \\\\setlength{\\\\columnseprule}{" (if (equal rule "solid") "2" "0") "pt}"
"          \\\\begin{minipage}[t]{\\\\linewidth}"
"          \\\\begin{multicols}{" cols "}"
"          %s"
"          \\\\end{multicols}\\\\end{minipage}\"))"
"(s-replace \":columnbreak:\" (if (equal 'html backend) \"\" \"\\\\columnbreak\")
contents)))")))))
```

We also use `parallel` as an alias for `2parallel`.

```
(defalias #'org-special-block-extras--parallel
          #'org-special-block-extras--2parallel)

(defalias #'org-special-block-extras--parallelNB
          #'org-special-block-extras--2parallelNB)
```

4.1 Example

Example:

```
#+begin_3parallel org
one

#+latex: \columnbreak
two

#+latex: \columnbreak
three
#+end_3parallel
```

Yields:

one | two | three

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

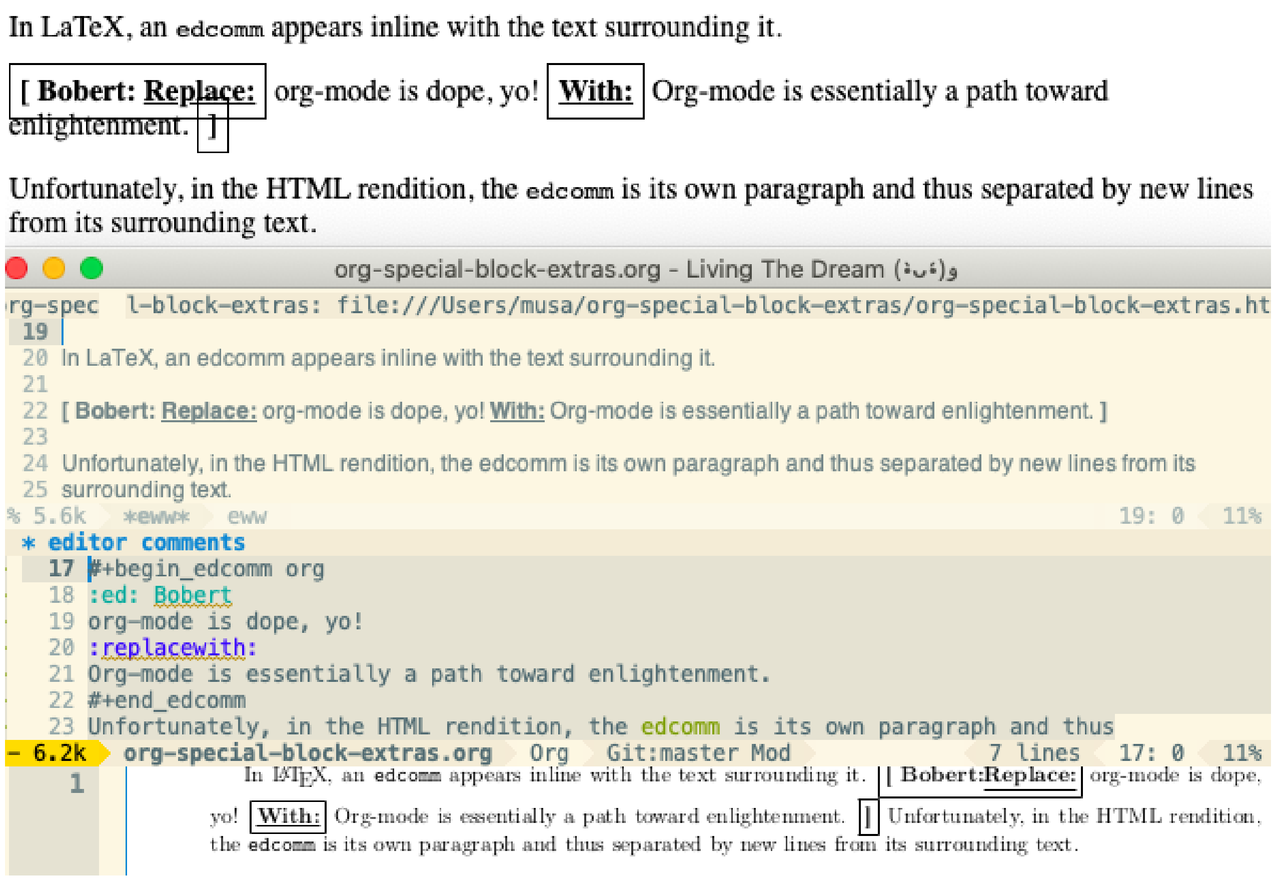
4.2 Next Steps

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.

5 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.

In LaTeX, an `edcomm` appears inline with the text surrounding it. **[Bobert:Replace:** org-mode is dope, yo! **With:** Org-mode is essentially a path toward enlightenment. **]** Unfortunately, in the HTML rendition, the `edcomm` is its own paragraph and thus separated by new lines from its surrounding text.



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

[Bobert: Replace: org-mode is dope, yo! **With:** Org-mode is essentially a path toward enlightenment. **]**

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

```
org-special-block-extras.org - Living The Dream (•̀ᴗ•́)و
rg-spec  l-block-extras: file:///Users/musa/org-special-block-extras/org-special-block-extras.ht

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

[ Bobert: Replace: org-mode is dope, yo! With: Org-mode is essentially a path toward enlightenment. ]

Unfortunately, in the HTML rendition, the edcomm is its own paragraph and thus separated by new lines from its
surrounding text.
% 5.6k  *eww*  eww                                                              19: 0  11%
* editor comments
#+begin_edcomm org
:ed: Bobert
org-mode is dope, yo!
:replacewith:
Org-mode is essentially a path toward enlightenment.
#+end_edcomm
Unfortunately, in the HTML rendition, the edcomm is its own paragraph and thus
- 6.2k  org-special-block-extras.org  Org  Git:master Mod              7 lines  17: 0  11%
```

In LaTeX, an edcomm appears inline with the text surrounding it. [**Bobert:Replace:** org-mode is dope, yo! **With:** Org-mode is essentially a path toward enlightenment.] Unfortunately, in the HTML rendition, the edcomm is its own paragraph and thus separated by new lines from its surrounding text.

Figure 3: In order: Chrome, Emacs Web Wowser, Org source, PDF

Any new —possibly empty— inner lines in the `edcomm` *are desirably preserved*

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

(defun org-special-block-extras--edcomm (backend contents)
"Format CONTENTS as an first-class editor comment according to BACKEND.

The CONTENTS string has two optional argument switches:
1. :ed: ⇒ To declare an editor of the comment.
2. :replacewith: ⇒ [Nullary] The text preceding this clause
   should be replaced by the text after it."
  (-let* (
          ;; Get arguments
          ((contents₁ . (&alist 'ed))
            (org-special-block-extras--extract-arguments contents 'ed))

           ;; Strip out any <p> tags
           (_ (setq contents_1 (s-replace-regexp "<p>" "" contents_1)))
           (_ (setq contents_1 (s-replace-regexp "</p>" "" contents_1)))

           ;; 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_1))
           (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"
                           (if (s-blank? ed) "Editor Comment" ed)
                           (if replacement-clause?
                               replace-keyword
                             "")))
           (contents_2 (if replacement-clause?
                          (format "%s %s %s" this
                                  (funcall boxed with-keyword)
                                  that)
                        contents_1))

           ;; "[Editor Comment:"
           (edcomm-begin (funcall boxed editor))
           ;; "]"
           (edcomm-end (funcall boxed "]")))

    (setq org-export-allow-bind-keywords t) ;; So users can use "#+bind" immediately
    (if org-special-block-extras-hide-editor-comments
        ""
      (format (pcase backend
                ('html "<p> %s %s %s</p>")
                ('latex "%s %s %s"))
              edcomm-begin contents_2 edcomm-end))))
```

In the HTML export, the `edcomm` special block is *not* in-line with the text surrounding it —ideally, it would be inline so that existing paragraphs are not split into multiple paragraphs but instead have an editor's comment indicating suggested alterations.

5.1 Block Examples

All editor comments 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.)

5.1.1 No optional arguments

$\boxed{\textbf{[Editor Comment:}}$ *Please* **change** this section to be more, ya know, professional. $\boxed{\textbf{]}}$

5.1.2 Only declaring an `:ed:` —editor

$\boxed{\textbf{[Bobert:}}$ *Please* **change** this section to be more, ya know, professional. $\boxed{\textbf{]}}$

Possibly with no contents: $\boxed{\textbf{[Bobert:}}$ $\boxed{\textbf{]}}$

5.1.3 Empty contents, no editor, nothing

$\boxed{\textbf{[Editor Comment:}}$ $\boxed{\textbf{]}}$

Possibly with an empty new line: $\boxed{\textbf{[Editor Comment:}}$ $\boxed{\textbf{]}}$

5.1.4 With a `:replacewith:` clause

$\boxed{\textbf{[Editor Comment:}\underline{\textbf{Replace:}}}$ The two-dimensional notation; e.g., $\sum_{i=0}^{n} i^2$ $\boxed{\underline{\textbf{With:}}}$ A linear one-dimensional notation; e.g., $(\Sigma i : 0..n \bullet i^2)$ $\boxed{\textbf{]}}$

Possibly “malformed” replacement clauses.

1. Forget the thing to be replaced. $\boxed{\textbf{[Editor Comment:}\underline{\textbf{Replace:}}}$ $\boxed{\underline{\textbf{With:}}}$ A linear one-dimensional notation; e.g., $(\Sigma i : 0..n \bullet i^2)$ $\boxed{\textbf{]}}$
2. Forget the new replacement thing. $\boxed{\textbf{[Editor Comment:}\underline{\textbf{Replace:}}}$ The two-dimensional notation; e.g., $\sum_{i=0}^{n} i^2$ $\boxed{\underline{\textbf{With:}}}$ $\boxed{\textbf{]}}$
3. Completely lost one’s train of thought. $\boxed{\textbf{[Editor Comment:}\underline{\textbf{Replace:}}}$ $\boxed{\underline{\textbf{With:}}}$ $\boxed{\textbf{]}}$

5.2 Link Type

A block to make an editorial comment could be overkill in some cases; so we provide the `edcomm` link type.

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

```
(org-link-set-parameters
 "edcomm"
  :follow (lambda (_))
  :export (lambda (label description backend)
            (org-special-block-extras--edcomm
             backend
             (format ":ed:%s\n%s" label description)))
  :help-echo (lambda (window object position)
               (save-excursion
                 (goto-char position)
                 (-let* [(&plist :path) (cadr (org-element-context))]
                   (format "%s made this remark" (s-upcase path)))))
  :face '(:foreground "red" :weight bold))
```

For example: **[Jasim:** Hello, where are you? **]**

The `:replacewith:` switch —and usual Org markup— also works with these links: **[Qasim:Replace:** *'j'* **With:** *'q'* **]**

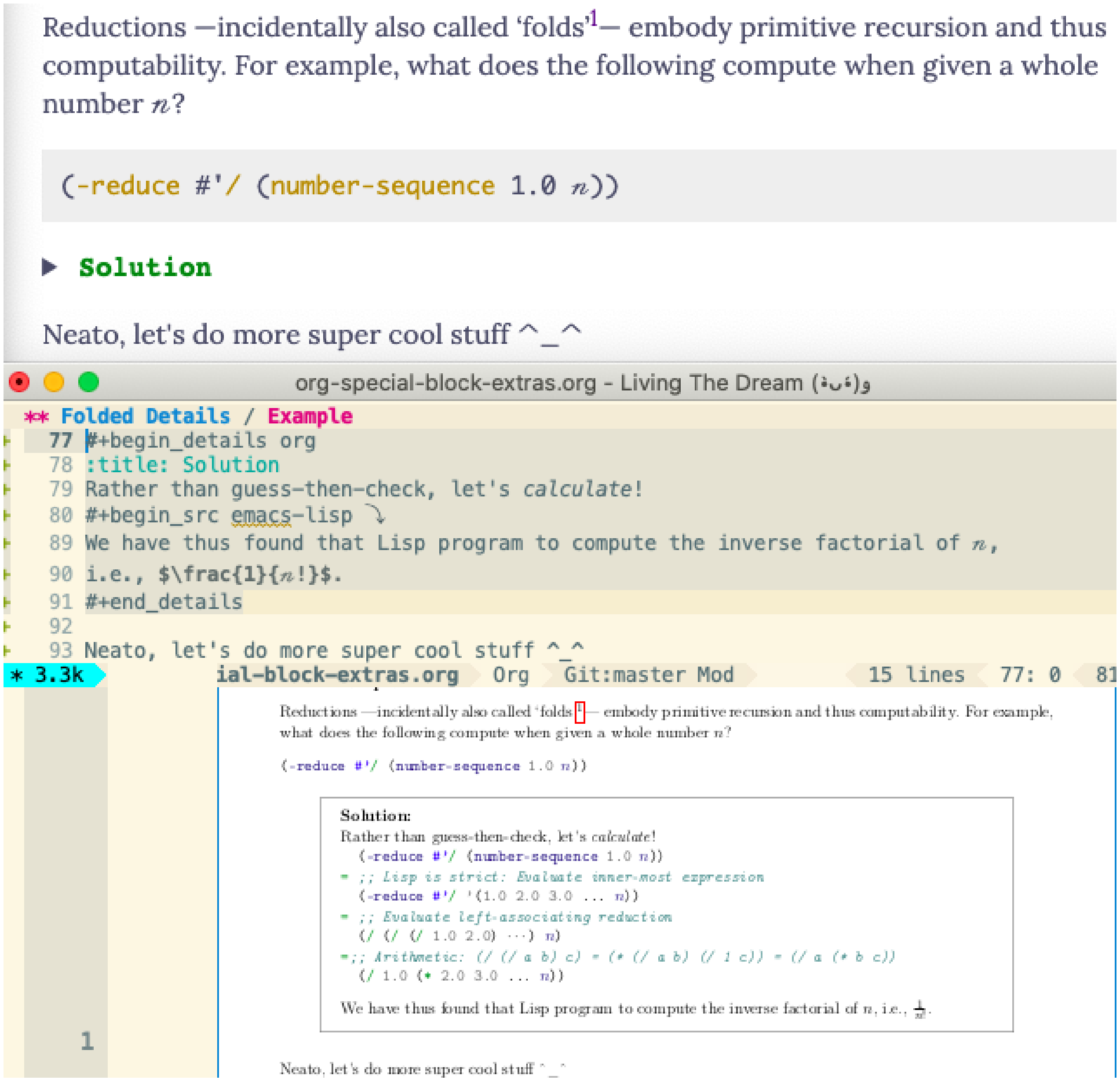
6 Folded Details

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.

`#+LATEX_HEADER: \usepackage{tcolorbox}`

```
(defun org-special-block-extras--details (backend contents)
"Format CONTENTS as a 'folded region' according to BACKEND.

CONTENTS may have a ':title' argument specifying a title for
the folded region."
(-let* (;; Get arguments
        ((contents' . (&alist 'title))
         (org-special-block-extras--extract-arguments contents 'title)))
  (when (s-blank? title) (setq title "Details"))
  (setq title (s-trim title))
  (format
   (s-collapse-whitespace ;; Remove the whitespace only in the nicely presented
                          ;; strings below
    (pcase backend
      (`html "<details class=\"code-details\">
                 <summary>
                   <strong>
                     <font face=\"Courier\" size=\"3\" color=\"green\"> %s
                     </font>
                   </strong>
```



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**

Neato, let's do more super cool stuff ^_^

```
** Folded Details / Example
#+begin_details org
:title: Solution
Rather than guess-then-check, let's calculate!
#+begin_src emacs-lisp ⤵
We have thus found that Lisp program to compute the inverse factorial of n,
i.e., $\frac{1}{n!}$.
#+end_details

Neato, let's do more super cool stuff ^_^
```

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 that Lisp program to compute the inverse factorial of n, i.e., $\frac{1}{n!}$.

Neato, let's do more super cool stuff ^_^

Figure 4: Visually hiding, folding away, details

```
                  </summary>
                  %s
               </details>")
       (‘latex "\\begin{quote}
                  \\begin{tcolorbox}[colback=white,sharp corners,boxrule=0.4pt]
                    \\textbf{%s:}
                    %s
                  \\end{tcolorbox}
                \\end{quote}")))
     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`.

6.1 Example

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 that 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.)

7 *“Link Here!”* OctoIcon

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, this is a possible way to do so.

- Besides the HTML backend, such links are silently omitted.
- SVGs obtained from: https://primer.style/octicons/

```
(org-link-set-parameters
  "link-here"
  :follow (lambda (path) (message "This is a local anchor link named ‘%s’" path))
  :export #’org-special-block-extras--link-here)

(defun org-special-block-extras--link-here (label description backend)
```

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

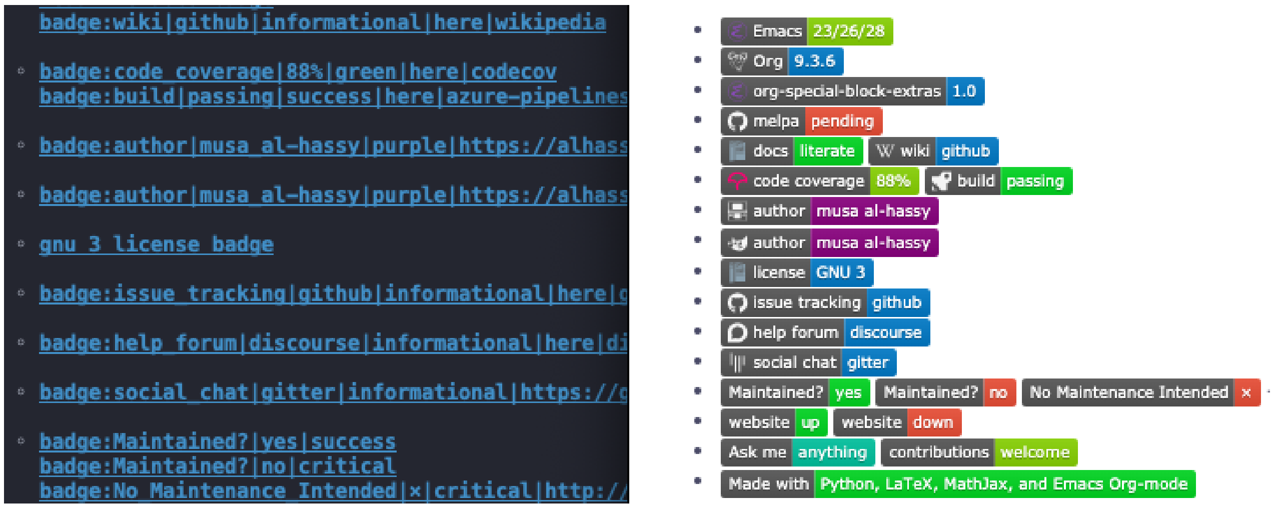
```
  "Export a link to the current location in an Org file.

The LABEL determines the name of the link.

+ Only the syntax ‘link-here:label’ is supported.
+ Such links are displayed using an ‘‘octicon-link’’
  and so do not support the DESCRIPTION syntax
  ‘[[link:label][description]]’.
+ Besides the HTML BACKEND, such links are silently omitted."
    (pcase backend
      (‘html  (format (s-collapse-whitespace
          "<a class=\"anchor\" aria-hidden=\"true\" id=\"%s\"
          href=\"#%s\"><svg class=\"octicon octicon-link\" viewBox=\"0 0 16
          16\" version=\"1.1\" width=\"16\" height=\"16\"
          aria-hidden=\"true\"><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></a>") label label))
      (_ "")))
```

E.g., Neato ^_^

Going forward, it would be desirable to provide a non-whitespace alternative for the LaTeX rendition. More usefully, before the HTML export hook, we could place such ‘link-here’ links before every org-title produce clickable org-headings, similar to Github’s —the necessary ingredients are likely here.

8 Badge Links

Badges provide a quick and colourful summary of key features of a project, such as whether it’s maintained, its license, and if it’s documented.



```
badge:wiki|github|informational|here|wikipedia

◦ badge:code_coverage|88%|green|here|codecov
  badge:build|passing|success|here|azure-pipelines

◦ badge:author|musa_al-hassy|purple|https://alhass

◦ badge:author|musa_al-hassy|purple|https://alhass

◦ gnu_3_license_badge

◦ badge:issue_tracking|github|informational|here|g

◦ badge:help_forum|discourse|informational|here|di

◦ badge:social_chat|gitter|informational|https://g

◦ badge:Maintained?|yes|success
  badge:Maintained?|no|critical
  badge:No_Maintenance_Intended|×|critical|http://
```

- Emacs 23/26/28
- Org 9.3.6
- org-special-block-extras 1.0
- melpa pending
- docs literate · wiki github
- code coverage 88% · build passing
- author musa al-hassy
- author musa al-hassy
- license GNU 3
- issue tracking github
- help forum discourse
- social chat gitter
- Maintained? yes · Maintained? no · No Maintenance Intended ×
- website up · website down
- Ask me anything · contributions welcome
- Made with Python, LaTeX, MathJax, and Emacs Org-mode

Figure 5: An Emacs interface to `https://shields.io/`

> As people who are passionate about writing great code we display "badges" in our code repositories to signal to fellow developers that we set ourselves high standards for the code we write, think of them as the software-equivalent of the brand on your jeans or other reliable product. — repo-badges

8.1 The badge Link and derived Reddit/Github/Twitter socials

The implementation is a bit lengthy since it attempts to capture a useful portion of the shilelds.io badge interface.

```
(org-link-set-parameters "badge"
  :follow (lambda (path) (--> (s-split "|" path)
                         (or (nth 3 it) path)
                         (browse-url it)))
  :export #'org-special-block-extras--link--badge)

(defvar org-special-block-extras--link--twitter-excitement
  "This looks super neat ^_^ :"
  "The string prefixing the URL being shared.")

(defun org-special-block-extras--link--badge
  (label description backend &optional social)
  "Export a link presented as an SVG badge.

The LABEL should be of the shape ‘key|value|color|url|logo’
resulting in a badge “|key|value|” where the ‘key’
is coloured grey and the ‘value’ is coloured ‘color’.

The optional SOCIAL toggle indicates if we want an icon for
Twitter, Reddit, Github, etc, instead of a badge.
When SOCIAL is provided, we interpret LABEL as an atomic string.

+ Only the syntax ‘badge:key|value|color|url’ is supported.
  - ‘key’ and ‘value’ have their underscores interpreted as spaces.
     ⇒ Underscores are interpreted as spaces;
     ⇒ ‘__’ is interpreted as an underscore;
     ⇒ ‘|’ is not a valid substring, but ‘-, %, ?’ are okay.
  - ‘|color|url|logo’ are optional;
     if ‘url’ is ‘|here’ then the resulting badge behaves
     like ‘link-here:key’.
  - ‘color’ may be: ‘brightgreen’ or ‘success’,
                    ‘red’         or ‘important’,
                    ‘orange’      or ‘critical’,
                    ‘lightgrey’   or ‘inactive’,
                    ‘blue’        or ‘informational’,
            or ‘green’, ‘yellowgreen’, ‘yellow’, ‘blueviolet’, ‘ff69b4’, etc.
+ Such links are displayed using a SVG badges
  and so do not support the DESCRIPTION syntax
  ‘[[link:label][description]]’.
+ Besides the HTML BACKEND, such links are silently omitted."
  (-let* (((lbl msg clr url logo) (s-split "|" label))
          (_ (unless (or (and lbl msg) social)
               (error "%s\t⇒\tBadges are at least “badge:key|value”!" label)))
          ;; Support dashes and other symbols
          (_ (unless social
               (setq lbl (s-replace "-" "--" lbl)
                     msg (s-replace "-" "--" msg))
               (setq lbl (url-hexify-string lbl)
                     msg (url-hexify-string msg))))
          (img (format "<img src=\"https://img.shields.io/badge/%s-%s-%s%s\">"
                        lbl msg clr
                        (if logo (concat "?logo=" logo) ""))))
    (when social
      (-->
          `(("reddit"            "https://www.reddit.com/r/%s")
            ("github/followers"  "https://www.github.com/%s?tab=followers")
            ("github/forks"      "https://www.github.com/%s/fork")
            ("github"            "https://www.github.com/%s")
            ("twitter/follow"    "https://twitter.com/intent/follow?screen_name=%s")
            ("twitter/url"
             ,(format
               "https://twitter.com/intent/tweet?text=%s:&url=%%s"
               org-special-block-extras--link--twitter-excitement)
             ,(format
               "<img src=\"https://img.shields.io/twitter/url?url=%s\">"
               label)))
        (--filter (s-starts-with? (first it) social) it)
        (car it)
        (or it (error "Badge: Unsupported social type ‘%s’" social))
        (setq url (format (second it) label)
              img (or (third it)
                      (format "<img src=\"https://img.shields.io/%s/%s?style=social\">"
                      social label)))))
    (pcase backend
        (`html (if url
                 (if (equal url "here")
                     (format "<a id=\"%s\" href=\"#%s\">%s</a>" lbl lbl img)
                   (format "<a href=\"%s\">%s</a>" url img))
               img))
        (_ ""))))
```

We now form the specialised link types for social media.

```
(loop for (social link) in '(("reddit/subreddit-subscribers" "reddit-subscribe-to")
                             ("github/stars")
                             ("github/watchers")
                             ("github/followers")
                             ("github/forks")
                             ("twitter/follow")
                             ("twitter/url?=url=" "tweet"))
      for link' = (or link (s-replace "/" "-" social))
      do (org-link-set-parameters link'
           :export (eval `(-cut org-special-block-extras--link--badge
                                <> <> <> ,social))))
```

8.2 Next Steps

Going forward, it would be desirable to provide non-whitespace alternatives for the LaTeX backend. **[Author:** That is why no examples are shown in the PDF **]** It would also be useful to have badges redirect to their URL, if any, upon a user's click. Finally, it may be useful to colour the |-separated fields of a badge link and provide a tooltip indicating which value corresponds to which field. This would make the interface more welcoming to new users.

9 Summary

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`. Let n be any number from `1..5`.

Idea	Block	Link	Switches
Colours	$\mathcal{C}$	$\mathcal{C}$, `color:`$\mathcal{C}$	`:color:`
Parallel	n`parallel[NB]`	-	`:columnbreak:`
Editorial Comments	`edcomm`	`edcomm`	`:ed:`, `:replacewith:`
Folded Details	`details`	-	`:title:`
Link Here	-	`link-here`	-
Badges	-	`badge`	-

There are also the social badge links: `reddit-subscribe-to`, `github-followers`, `github-forks`, `github-stars`, `github-watchers`, `twitter-follow`, and `tweet`.