LiaScript

To see this document as an interactive LiaScript rendered version, click on the following link/badge:

If you need help, feel free to ask us any questions in the chat:

With LiaScript, we tried to implement an extended Markdown format that should enable everyone to create, share, adapt, translate or correct online courses without the need of being a web-developer. We believe that a language-based approach, instead of a tooling-centered one, provides more flexibility, freedom of creativity, and sustainability. Therefor we tried to develop a simplistic syntax that extends the static Markdown with quizzes, animations, spoken text, automated visualization ASCII-art and much more. Everything has been woven around Markdown, so that the content can still be read and interpreted with any kind of editor or Markdown-interpreter.

What is LiaScript?

- a Markdown dialect for interactive courses and data-driven publishing,
- everything is implemented in Elm/JavaScript and runs directly within the browser (online),
- the interpreter itself is also a reader, which allows for storing documents as well as the progress,
- courses can also be taken offline, since the interpreter is also a progressive web app (PWA), that allows to store documents and progress directly within the browser (locally),
- everything is private, we do not store any data about the courses nor the users and their progress

There are a couple of problems that we currently see in the creation of Open educational resources (OER). One of them is isolation, that means people, who want to create content, are seldomly connected via the applied technologies, instead, they are separated by platforms, authoring tools and used core technologies (programming languages). Furthermore, it is not possible to simply grab an educational website/project and to adapt its content for another audience. Additionally, it seems to be nearly impossible for people without a technical background to simply set up a small course. Thus, they stuck with Word, PowerPoint, and PDF, since they provide a simple continuation of the static formats people have used before the computer-era. "*If I want to publish content for the computer, I want my audience to dive in, experiment, simulate, play with the content... but not only read.*"

Goals

- Simplicity: with a human-centered markup-language, anyone should be enabled to create and modify content.
- Interactive: the browser is the next operating systems and although content with LiaScript is developed within a "static" markup-language it should not be presented that way.
- Extendability: everything that is not part of LiaScript shall be embeddable and importable.
- Durability: platforms go down, the development of proprietary software/formats is discontinued, but LiaScript is not hosted on one platform (it can be hosted everywhere) and even without the LiaScript interpreter the content is still readable and interpretable with every editor; you could even print or engrave it on stone or clay. Furthermore, if you use some kind of versioning system (e.g. git) you can refer to any previous version of your course.

Imagine a world where everyone would have the same access to high quality educational content for free. Imagine all kind of schoolbooks, technical or scientific literature could become open-courSe projects and more interactive, with collaborating teachers and students. Everything that is required is a simple text-editor and a web-browser.

Tools

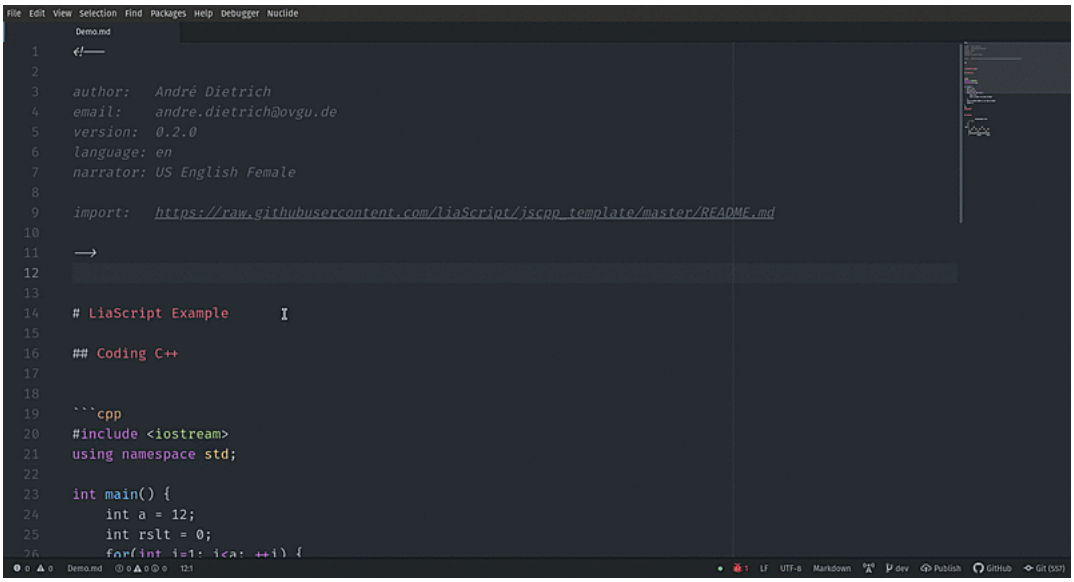
As already mentioned all you need to work with LiaScript is an text-editor, but it can be usefull to apply one of the following tools. At least we apply them to see the result of a change within the course document immediately. You will see, that the development of online-courses will speed up, especially if there is no need for memorizing complex point and click sequences.

> "*Let the editor be your canvas and the keyboard your brush.*"

Editing

Atom:

There are currently 2 plugins for the Atom Editor and Visual-Studio-Code available, which are intended to ease and simplify the development of online courses with LiaScript.



Screencast of the Atom-editor with the liascript-preview and liascript-snippts installed.

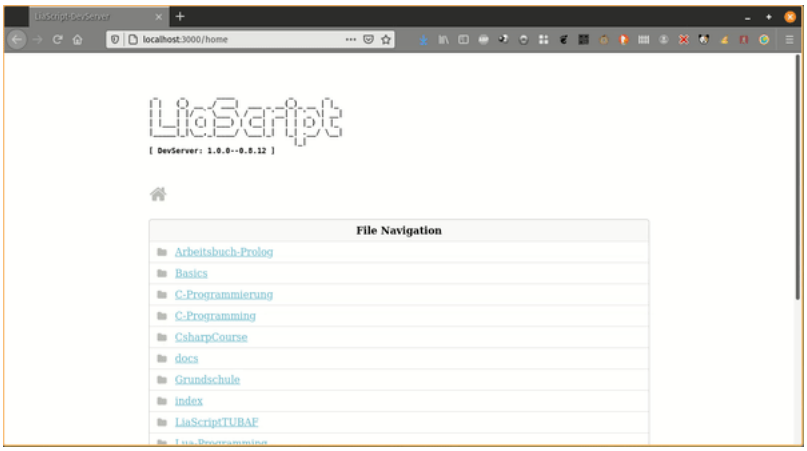
1. Atom: This is the free and open and official GitHub editor, with lots of plugins for various use cases.
    1. liascript-preview: Is a tiny previewer that, if it was toggled (`Alt+L`), updates the view on your course each time you save your document.
    2. liascript-snippets: If you start typing `lia` in your Markdown document you switch on a fuzzy search, that contains a lot of LiaScript help, examples, and snippets.

    Detailed installation instructions can be found here.
2. Visual-Studio-Code: This is Microsofts new flagship with various plugins and extensions.
    1. liascript-preview: Is a tiny previewer that, if it was toggled (`Alt+L`), updates the view on your course each time you save your document.
    2. liascript-snippets: If you start typing `lia` in your Markdown document you switch on a fuzzy search, that contains a lot of LiaScript help, examples, and snippets.

    Detailed installation instructions can be found here.

LiaScript-DevServer:

If you are used to another editor, you can also start a LiaScript development server locally. It works like the plugin for Atom and updates your course within your browser on every save, but this can also be used to monitor multiple projects. And you can also use it for testing purposes, as if you would deploy your projects.



Screencast of the liascript-devserver while navigating through a folder-structure.

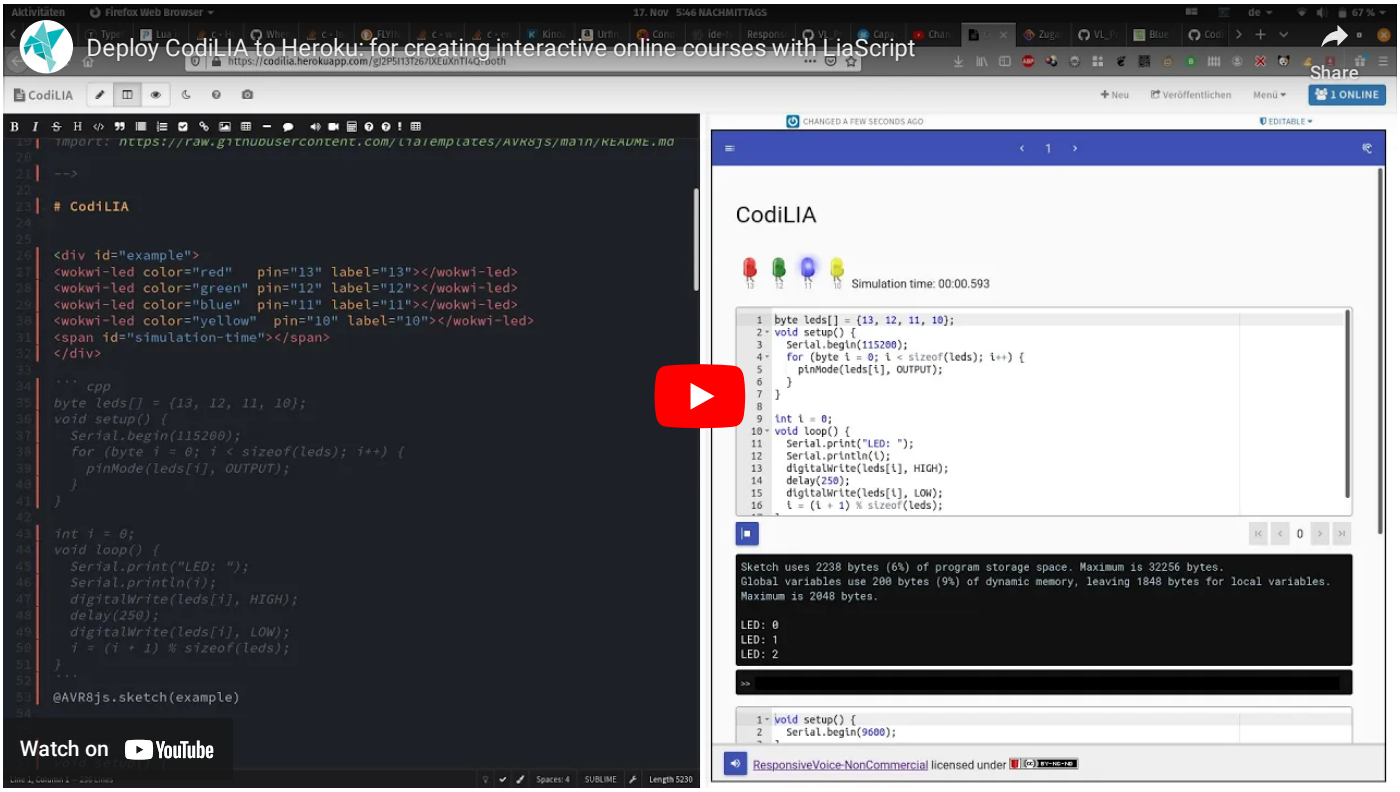
Get the project from:

- npmjs: https://www.npmjs.com/package/@liascript/devserver
- GitHub: https://github.com/LiaScript/LiaScript-DevServer

CodiLIA:

If you prefer to work with multiple authors simultaneously, then you should give CodiLIA a try. It is a fork of the collaborative editor CodiMD/Hedgehoc, but instead of a Markdown preview, you will have a LiaScript preview, and you can immediately publish your courses.

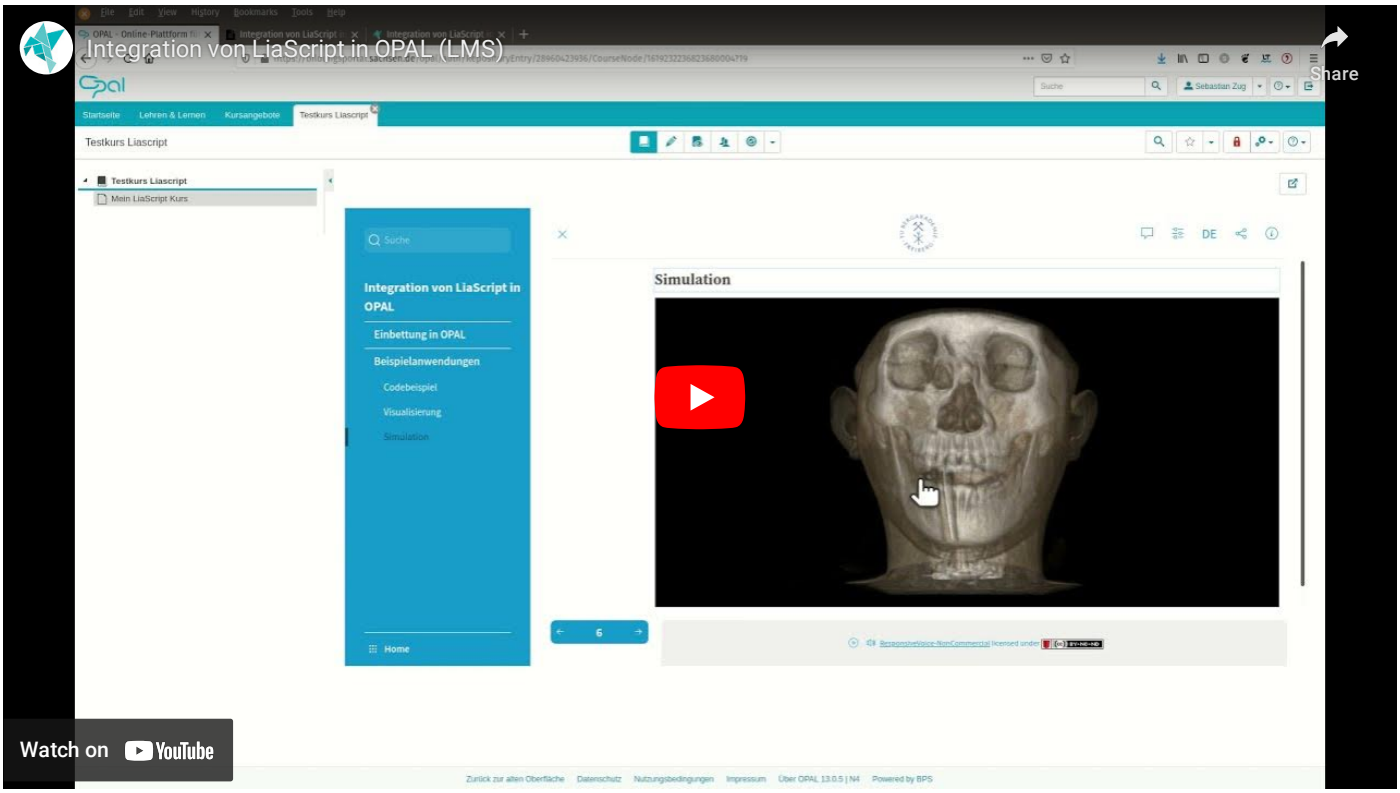
Project: https://github.com/liascript/codilia



Movie: A HowTo deploy CodiLIA server to Heroku for free.

Projects

The easiest way of importing a LiaScript into another website or Learning Management System (LMS) such as Moodle, ILIAS, or OPAL, is via importing an external website or if possible via an `iframe`.



Movie: Screencast of a LiaScript course that is hosted via CodiLIA and imported as an external resource into OPAL.

LiaScript-Exporter:

However, there might be cases where you want to store the progress within the LMS. We therefor have developed an experimental exporter, which allows to bundle your entire project as an SCORM compliant zip-file that can be imported into most common LMS. Other formats than SCORM can be added too, simply write us a mail or create an issue, if you require another one.

- npmjs: https://www.npmjs.com/package/@liascript/exporter
- GitHub: https://github.com/liascript/liascript-exporter

Preview-Lia:

If you want to refer to your own courses or to foreign ones on your personal website or blog, you can make use of our "preview web component". This creates preview cards, which are updated at client-side, so that there is no need to manually update all information whenever there is a change in the course. Simply add the `script` tag as depicted in the code-snippet and link to your courses via the tag `preview-lia`.

```
<html>
  <head>
    <script type="text/javascript" src="https://liascript.github.io/course
      /preview-lia.js"></script>
  </head>
  <body>
    ...
    <preview-lia src="https://raw.githubusercontent.com/liaScript/docs/master
      /README.md">
    </preview-lia>

    <preview-lia src="https://liascript.github.io/course/?https://raw
      .githubusercontent.com/liaScript/docs/master/README.md">
    </preview-lia>

    ...
  </body>
</html>
```

For more information visit the blog entry: Markdown just got a new preview tag

However, you can also use this to refer to your personal GitHub projects.

LiaScript was originally developed for supporting programming courses for embedded systems. You can see an example of the previous eLab remote laboratory installation.

Movie: A review on the historical eLab system, the predecessor to LiaScript.

Now it is the opposite, our main focus lays in the development of the Markup language, but parts of the old systems can still be used. Especially if you want to teach programming in (*Java*, *C*, *C++*, *C#*, *Mono*, *Go* and *Python*). The CodeRunner is an open-source project that enables remote compiling and execution of code. We apply it to teach procedural and object-oriented programming. You can either host your own server or use our free herokuapp:

CodeRunner:

- GitHub: https://github.com/liascript/CodeRunner
- Try the interactive LiaScript version at:

  https://liascript.github.io/course/?https://github.com/liascript/CodeRunner
- Or if you want to import this functionality into your course, then add the following statement into the main header of your LiaScript course:

  `import: https://raw.githubusercontent.com/LiaScript/CodeRunner/master/README.md`

Publishing

By now you should have a basic idea of what you can do with LiaScript, but probably not how you can publish your courses. The best way is actually to use GitHub, this way no prior versions of your course get lost, and you give others (you can even invite them) the opportunity to contribute to your project.

No further hosting is required, no further compilation step, the JavaScript interpreter of LiaScript does everything else directly within the browser at client-side.

As you can see from the *example*, it is also possible to refer to a specific slide. You only have to add a `#` with the number of the slide, or you can add the name of the specific slide as well.

You can also add additional tags to your project to make it discoverable. We currently use three distict categories: `liascript` to mark it to be related to the projects, while the others `liascript-course` and `liascript-template` are used to distinguish the projects into courses or extension, which can be added to courses.

The same way you can also refer to courses that you have put into your DropBox, ownCloud, NextCloud, or if you have access to some old-fashioned webspace then you can also store all of your files there. You only have to make them publically available and to refer to the raw or in other words, the text document. All other sources are loaded relative to this URL.

0. Create a free account at https://github.com
1. Refer to your projects as via a URL parameter:

   `https://LiaScript.github.io/course/?YOUR_RAW_COURSE_URL.md`
2. Example with reference to a specific slide:

   https://liascript.github.io/course/?https://raw.githubusercontent.com/liaScript/docs/master/README.md#5
3. Make your document discoverable by adding the tags `liascript` and or `liascript-course`, `liascript-template` to make it appear in one of the following GitHub topics:

   - general: https://github.com/topics/liascript
   - free courses: https://github.com/topics/liascript-course
   - extensions: https://github.com/topics/liascript-template

   More information on tagging projects can be found here.
4. Use other ways of hosting repositories as well (e.g. DropBox, ownCloud, NextCloud).

LiaBooks

However, we have no idea who is using LiaScript elsewhere, so it might be hard to find some resources online. From time to time we translate open-books into LiaScript and make them more interactive. You can see some of our the experiments at the following URL and use them as a source of inspiration:

https://github.com/LiaBooks

Full overview on courses via the topic `liascript-course`:

https://github.com/topics/liascript-course

LiaTemplates

If you tried out CodeRunner, you will have probably noticed that you can reuse functionality from different courses, simply by using the keyword `import:` within the main definition of your LiaScript document. Such a functionality is defined with the help of macros. We will dive deeper into this feature at the end of this document, but if you are interested you can inspect some of our templates, which shall provide self-explaining courses of how to embed and use the implemented macros.

https://github.com/LiaTemplates

Full overview on extensions via the topic `liascript-template`:

https://github.com/topics/liascript-template

Markdown-Syntax

This section is intended to give a brief overview on the basic Markdown syntax elements. The only difference to common Markdown at this point is, that you can define meta-information such as author, language, voice, etc. within a HTML-comment at the beginning of every document. We will describe all of these elements in more detail in section: Macros. All of these `macros` start with a single word, which is followed by a colon. If you require more space, like for `comment:` or `link:` you can use multiple lines, but every following line has to start with an indentation.

Initial LIA-comment-tag for basic definitions:

```
<!--

author:   Andre Dietrich

email:    LiaScript@web.de

version:  0.0.1

language: en

narrator: US English Female

comment:  Write a short abstract of your course, that
          might contain multiple lines and sentences.

script:   https://javascript_resourse_url

script:   https://another_javascript_resourse_url

link:     https://some_css_stuff
          https://and_some_more_css
-->
```

The meta-information from your document is later shown within the information-section as well as within the home-section.

If you already know Markdown, then you can skip most of the content in this section. However, there are some slight differences that will be marked with a trailing star at the section title.

> Something might be different 💫

Structuring

A course is structured as any other Markdown document with starting hash-tags, whereby the number of hash-tags is used to define the hierarchy of your document.

```
# Main Title

 ...

## Section Title 1

 ...

### Subsection Title

 ...
## Section Title 2
```

Every section is presented separately. In contrast to most Markdown-parsers, LiaScript applies a two step-approach. Sections are parsed at first, which means that the parsers searches for patterns as depicted below. Parsing the content of a section is quite time-consuming, that is why the section-content gets only analyzed, if this specific section should be displayed to the user. However, this happens only at the first appearance, afterwards the resulting view is restored from a local cache.

Preprocessing pattern: `## foo bar`

Semantic Correct HTML

As mentioned earlier, the preprocessor searches for patterns `## header` at the beginning of a line to identify sections. However, there might be cases where you want to have multiple different sections on one slide, with different headers. Semantic HTML can help us to deal with this, especially the two HTML5 tags `section` and `article`.

```
# Slide-Title

<section>
## Section-Title

...
</section>

<article>
## Article-Title

...
</article>
```

LiaScript will identify these HTML-tags and the content, such that the content within cannot be used as a separator. If you use these two semantic tags, your content is grouped in semanic correct way, which improves the readability if screen-readers are used or keyboard navigation is used.

When to use which tag might be philosophical question. We can say, if you just want to structure your content with different sub-headers, then use `<section>`. If your content represents a self-contained document, then use `<article>`. However, the visually presented result will be the same, such that you could also use a `<div>` to structure your content.

Section-Title

> The `<section>` HTML element represents a generic standalone section of a document, which doesn't have a more specific semantic element to represent it. Sections should always have a heading, with very few exceptions.
>
> Source: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/section

Article-Title

> The `<article>` HTML element represents a self-contained composition in a document, page, application, or site, which is intended to be independently distributable or reusable (e.g., in syndication). Examples include: a forum post, a magazine or newspaper article, or a blog entry, a product card, a user-submitted comment, an interactive widget or gadget, or any other independent item of content.
>
> Source: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/article

Sub-Titles 💫

There might be some cases, where you want to add further headings quickly. We therefor apply the following syntax with underlining "equal signs" or "dashes". In common Markdown, this alternative syntax is applied to define level-1 and level-2 headings. We use it to create headings that are one level `=` or two levels `-` below the main heading. However, these subsections will not be part of the table of contents, and since their interpretation is slightly different to common Markdown, you should use the method presented in the previous section.

```
...

## Section Title

Local Subsection
================

Local Sub-SubSection
--------------------
```

Content blocks

How would you separate paragraphs or other content elements from each other, if you only have a type-writer? The easiest way is a spatial separation, and in Markdown this is done via an empty line. Thus, whenever you have blocks such as paragraphs, enumerations, or tables, it is common practice to separate them via a newline. This makes it easier for you to edit and structure your course, and it prevents the interpreter from too much work.

```
This is a paragraph that consist only of one line.

Here comes another paragraph
with multiple lines.
And multiple sentences.
```

Text-Formatting 💫

How does text-highlighting work in a text file and thus within a paragraph? Well, Markdown defines some basic characters that can be used to surround a word or a collection of words. We tried to use the GitHub flavored Markdown style for simple formatting, thus simply use multiple asterisks or underscores to mark certain parts of a text.

- `*italic*` → *italic*
- `**bold**` → **bold**
- `***bold and italic ***` → ***bold and italic***
- `_also italic_` → *also italic*
- `__also bold__` → **also bold**
- `___also bold and italic___` → ***also bold and italic***
- `~strike~` → ~~strike~~

We define some additions to common Markdown, such as underline and superscript, which can be defined with the following syntax:

- `~~underline~~` → underline
- `~~~strike and underline~~~` → ~~strike and underline~~
- `^superscript^` → superscript

Combinations

As you can see from the examples, you can combine and nest all elements freely.

- `**bold _bold italic_**` → **bold *bold italic***
- `**~bold strike~ ~~bold underline~~**` → **~~bold strike~~ bold underline**
- `*~italic strike~ ~~italic underline~~*` → *~~italic strike~~ italic underline*

Escape Characters

If you want to use asterisks, hash-tags, or other syntax elements within your document without applying their functionality, then you can escape or in other words indicate them with a starting backslash. If you want to escape a backslash, you will have to write two subsequent backslashes. But you do not have to use it, if there is only one asterisk within a line, this will be interpreted as a single character. So you will have to apply this kind of escaping only to prevent misunderstandings between you and the interpreter.

```
\*, \~, \_, \#, \{, \}, \[, \], \|, \`, \$, \@, \\, \<, \>
```

Result: *, ~, _, #, {, }, [,], |, `, $, @, \, <, >

Symbols & Unicode 💫

One thing that we missed in standard Markdown, was an implementation for arrows. The verbatim text shows, how arrows are defined in our Markdown implementation with their result on the right.

`->` →, `->>` ↠, `>->` ↣, `<-` ←, `<-<` ↢, `<<-` ↞, `<->` ↔, `=>` ⇒, `<=` ⇐, `<=>` ⇔

`-->` ⟶, `<--` ⟵, `<-->` ⟷, `==>` ⟹, `<==` ⟸, `<==>` ⟺

`~>` ⇝, `<~` ⇜

But you can also use some basic smileys. We will try to extend this partial support in the future.

`:-)` 🙂, `;-)` 😉, `:-D` 😀, `:-O` 😮, `:-(` 😕, `:-|` 😐, `:-/` 😕, `:-P` 😛, `:-*` 😗, `:')` 😂, `:'(` 😢

However, since LiaScript accepts Unicode, you can also copy and paste any kind of character including emojis.

References

The next section shows how external resources can be referenced and integrated.

Simple Links

There are two ways of adding links to a Markdown document, either by inlining the URL directly or you can name it (as shown in listing 2), by applying the typical brackets and parenthesis notation, the optional information is put in double quotes at the end of the URL. This optional information is used as a title attribute, and it is shown, when the user hovers the link with the mouse.

1. Example of an URL-link → http://goo.gl/fGXNvu

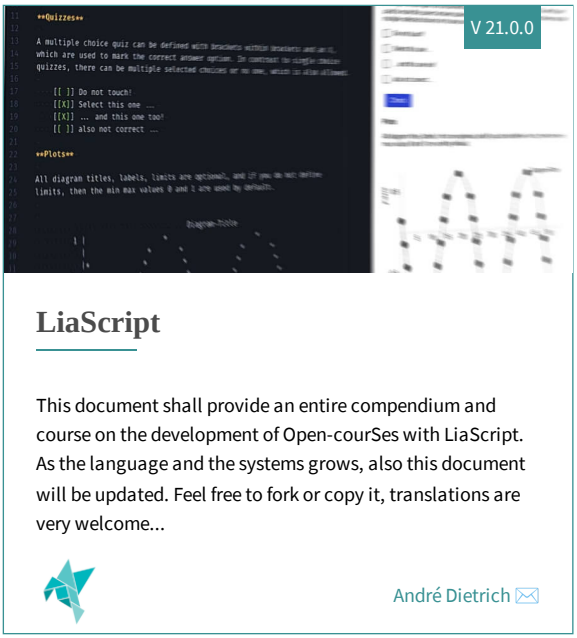
   text-formatting can be applied also `*** http://goo.gl/fGXNvu ***` → ***http://goo.gl/fGXNvu***
2. Naming the link (`[title](http://goo.gl/fGXNvu "optional info")`) → title
3. For internal navigation you can refer to the slide number or to title with a starting `#`
   - `[next slide](#18)` → next slide
   - `[next slide](#preview-lia-💫)` → next slide
   - If your internal link includes parentheses or other characters, you can also use percent-encoding. An opening parentheses would then be written as `%28` and a closing one as `%29`.

     For more information visit: https://developer.mozilla.org/en-US/docs/Glossary/percent-encoding

Preview-Lia 💫

LiaScript has an advanced preview link, that will load the remote course and parse the meta-information such as title, version, comment, logo, email, tags, form your main HTML-comment at the top of your document. To do this, you will have to use `[preview-lia]` as the title of your link, which is followed by the raw URL of your course document.

`[preview-lia](https://raw.githubusercontent.com/LiaScript/docs/master/README.md)`



You can use this technique also to create previews for other courses on your personal website or for other GitHub projects, as it was described in section Projects. For more information follow the link:

https://aizac.herokuapp.com/markdown-just-got-a-new-preview-tag/

QR-Codes 💫

Sometimes it might be required to have both, a link and a visual representation of it as an **QR-Codes**. Similar to previews, you simply name your link `qr-code`:

- Syntax: `[qr-code](https://LiaScript.github.io)`
- Example:

You can add further information to your link by adding a title. Markdown is also allowed within the link title. In case of an image or media link, the title will be used as a subtitle and displayed accordingly.

- Syntax: `[qr-code](https://LiaScript.github.io "Checkout the LiaScript website or the __[blog at heroku](https://aizac.herokuapp.com)__")`
- Example:

Checkout the LiaScript website or the **blog at heroku**

Images 💫

Images are defined similar to links, but they are indicated with a starting exclamation mark.

The name of the link or the alt-text is not wasted, since it is not displayed anymore. Instead, it is displayed, if the image cannot be loaded for some reasons, and it is used by screen readers to give visually impaired people a hint, of what will be visible on the image. So please, don't leave it empty.

The URL can be either relative to your Markdown document or it can be absolute, which means it is pointing to an external resource.

The optional title in LiaScript is not only used as a title attribute, instead it is used as a real sub-title for all media links.

Image-notation: ![`alt-text`](url "optional sub-title")

- relative URL: `![Beautiful Lenna](img/lenna.jpg)`

- absolute URL: `![Annunciation of ...](https://upload.wiki...jpg)`

Annunciation c. 1472–1476: is thought to be Leonardo's earliest complete work

Note, that LiaScript is smart enough to scale your images to the optimal size. If your image is smaller than the current maximal applicable width, it is shown in full size. If it is larger, than it is scaled to fit in width and also in height. You can further click on all images to open them as modal and if the image is quite large, such as Leonardo's painting, then you can also zoom and inspect it, by hovering with the mouse or thumb.

Additionally, if you start a paragraph with an image, LiaScript expects it to be a floating object, which is depicted with a maximal size of 50% of your view-port, if it is not smaller than that.

```
![Beautiful Lenna](img/lenna.jpg "subtitles are allowed too")
```

subtitles are allowed too

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Galleries 💫

How would you interpret a paragraph full of images? We thought that the only reasonable depiction of this could be a gallery.

```
![img1](url) ![img2](url) ![img3](url)
![img4](url)
![img5](url)
```

And we like this idea... You can click on every image and inspect it also with the zooming feature.

La Belle Ferronnière, c. 1490–1498

Lady with an Ermine, c. 1489–1491, Czartoryski Museum, Kraków, Poland

Mona Lisa or La Gioconda c. 1503–1516, Louvre, Paris

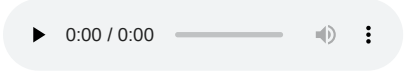
The Virgin and Child with Saint Anne, c. 1501–1519, Louvre, Paris

The Death of Leonardo da Vinci, by Ingres, 1818

Audio 💫

If an exclamation mark indicates visual content, why not using a question mark to indicate auditive content. (From our perspective it resembles an ear.) Everything is similar to images and the URLs can be either relative or absolute.

- Syntax: `?[a horse](https://www.w3schools.com/html/horse.mp3 "hear a horse")`
- Example:



hear a horse

Additionally, you can also directly reference music from the SoundCloud website or from Music.YouTube. The associated song will be automatically embedded for you.

- Syntax: `?[soundcloud](https://soundcloud.com/glennmorrison/beethoven-moonlight-sonata)`
- Example:

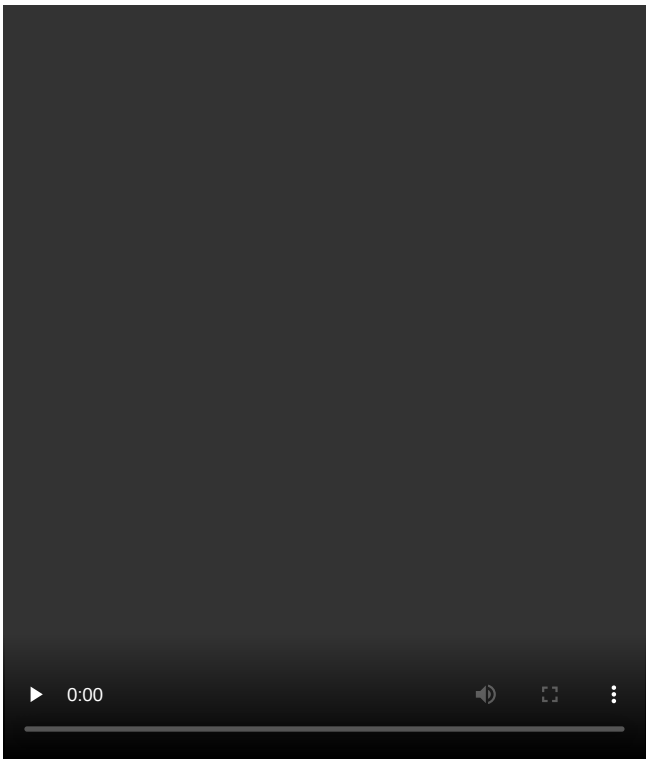
Movies 💫

Images are marked with a starting exclamation mark before the link, audio by a starting question mark and movies are made of images and sound, that is why you combine both marks `!?`. Defining resources this way shows at least the links correctly in other Markdown parsers or on GitHub. There is baked-in support for YouTube, Vimeo, PeerTube, DailyMotion and TeacherTube, which means that you only have to include the link and the resource will be embedded appropriately.

Movie-notation: `!?[alt-text](movie-url)`

- YouTube: `!?[The Future of Programming](https://www.youtube.com/watch?v=8pTEmbeENF4)`

- relative path: `!?[Something about math](vid/math.mp4)`



- **List of supported Video Platforms**
  - DailyMotion
  - PeerTube
  - TeacherTube
  - Video TU-Freiberg
  - Vimeo
  - YouTube

If you required more control over your video, such as autoplay, muted, start-time, and probably also size and colors, the you can also apply custom styling rules, then you should take a look at the following section:

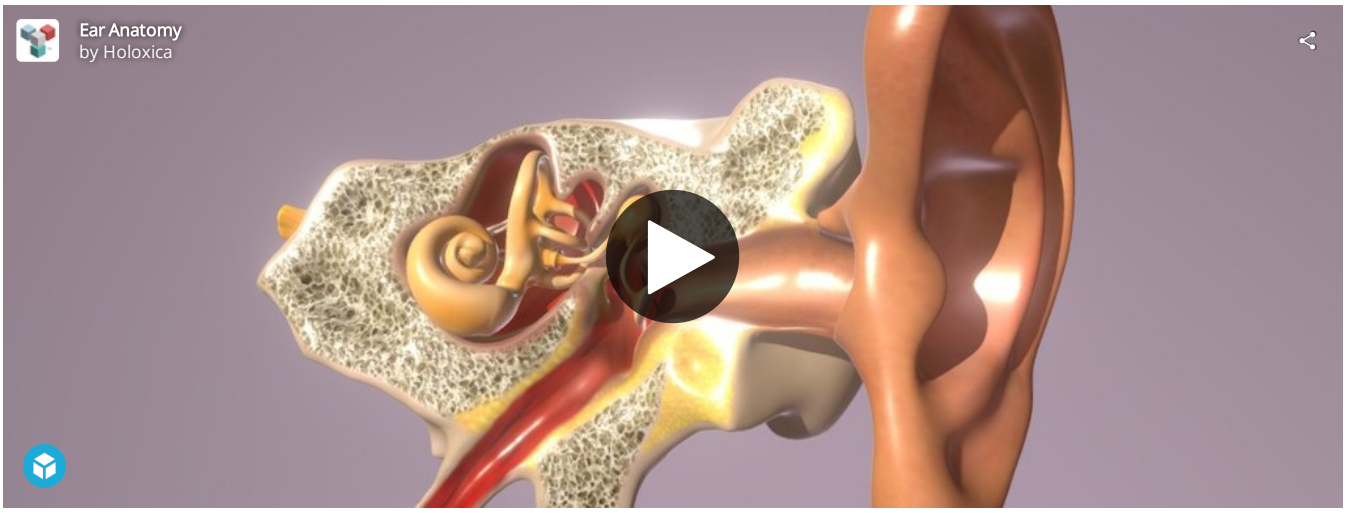
Customizing-Multimedia

So what is left?? 💫

If it is something else that you want to embed something else from another website, then you should try out the link syntax with two starting question marks. This means, LiaScript will try to use the oEmbed service, which is offered by a couple of websites. If this succeeds, this will embed only a specific part. If it fails, then LiaScript will at least try to embed the website via an `iframe`.

Examples:

- SketchFab: `??[ear model](https://sketchfab.com/3d-models/ear-anatomy-468e2039bde34a3fabb9e90bff9cd56b)`



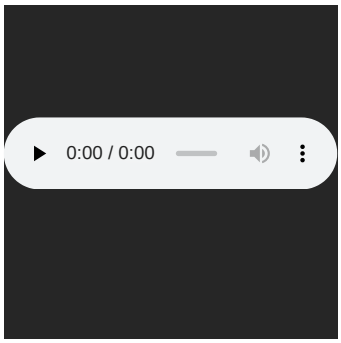
- StoryMaps: `??[presentation](https://storymaps.arcgis.com/stories/583f8b48a857442cb8d27411c93a9664)`

- CirquitJS: `??[Simulation: Noninverting Amplifier](https://www.falstad.com/circuit/circuitjs.html?startCircuit=amp-noninvert.txt)`

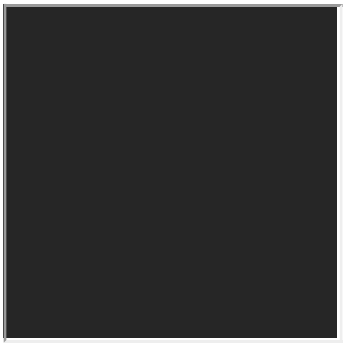
Galleries #2 💫

What you have seen previously with images is also possible with any kind of multimedia link. Simply put everything into one paragraph and LiaScript will automatically generate a gallery for you:

```
![img](url) ?[audio](url) !?[movie](url)
??[something else](url)
??[something else](url)
```



hear a horse



Markdown-Blocks

So far we had introduced Markdown only on a tiny level, which means, by now you should know how to emphasize words and phrases within a paragraph, how to add images and how you can use references to point to internal or external resources. But, all we did so far is to work with **paragraphs**. But as pointed out in the quote below, Markdown can do even more.

> *... Markdown's syntax is comprised entirely of punctuation characters, which punctuation characters have been carefully chosen so as to look like what they mean. E.g., asterisks around a word actually look like *emphasis*. Markdown lists look like, well, lists. Even blockquotes look like quoted passages of text, assuming you've ever used email.*
>
> — Markdown philosophy by John Gruber

Within the following part we will learn how to deal with Markdown blocks and how to format your content to define the following elements:

1. Lists
2. Blockquotes
3. Tables
4. HTML
5. Code-Snippets
6. Horizontal rules

Lists

The GitHub-flavored Markdown supports two types of lists, ordered and unordered ones, and so does LiaScript. If you every used a typewriter then the following syntax for lists would look natural to you. The only thing that matters here is the correct indentation.

> **Use spaces for correct indentation!** Tabs are allowed too, but might be confusing, since editors tend to use varying lengths from 2 to 4 spaces to display them...

Unordered Lists

To define an unordered list, starting asterisks `*`, pluses `+`, and dashes `-` can be used and mixed. If one point has more than one line, you can also use multiple lines to define paragraphs. All other Markdown elements, you will get to know, can be included in the same way.

Markdown-Syntax:

```
* alpha
+ **beta**
- gamma
  and delta

  new Paragraph

  - and another
  - important list

- epsilon
```

As you can see from the result, you can apply all Markdown styling elements freely. The starting characters will be interpreted equally, thus it makes no difference, if you use asterisks, pluses and dashes. To improve the readability of your document, we would recommend to stick with one format for every level. Starting with asterisks on the first level and dashes within the second level, etc.

Result:

- alpha
- **beta**
- gamma and delta

  new Paragraph

  - and another
  - important list
- epsilon

Note: At the moment it is required to separate blocks by at least one empty line. The following example will be interpreted as a single paragraph:

```
* this is one single
  - paragraph with a dash.
```

Whereby the following will result in a bullet point with another one nested

```
* separate paragraph

  - and this is a separate sub listing
```

Ordered Lists 💫

Ordered lists start with a number and a dot. As you can see from the example, the numbering is important. In contrast to the GitHub flavored Markdown or the original Markdown, where the list below would result in **two** separate lists, and the numbering for every list would start at 1, ignoring your numbering order. With the LiaScript interpretation you can separate your lists, add more explanations in between, or use animations to make certain parts appear or disappear.

Markdown-Syntax:

```
0. alpha
1. **beta**

Something else ...

3. * gamma
   * delta
   * and epsilon
2. probably zeta
```

Result:

0. alpha
1. **beta**

Something else ...

3. - gamma
   - delta
   - and epsilon
2. probably zeta

Blockquotes

If you, from time to time, reply to emails, than the following notation will look quite familiar to you. To make use of quoted text, simply start a line with a `>` greater than character.

Markdown-Syntax:

```
> This was said some time ago ...
>
>> This was said even longer ago,
> > I guess ...
>
> * aleph
> * beth
```

As you can see from the example, all Markdown elements can be used within a blockquote and vice versa. Everything you have learned so far can be easily combined, it could also be a gallery or an embedded object...

Result:

> This was said some time ago ...
>
> > This was said even longer ago, I guess ...
>
> - aleph
> - beth

Citations 💫

Blockquotes are often used for citations, and so do we. You can use the following pattern to mark a blockquote as a citation. Simply use two paragraphs within a blockquote and start the second one with two dashes `--`.

LiaScript-Syntax:

```
> “Live as if you were to die tomorrow.
> Learn as if you were to live forever.”
>
> -- Mahatma Gandhi
```

The resulting blockquote looks slightly different. Furthermore, the paragraph followed by dashes is put into and HTML `cite`-tag.

> *“Live as if you were to die tomorrow. Learn as if you were to live forever.”*
>
> — Mahatma Gandhi

You can use this syntax, with starting dashes, everywhere within a LiaScript document and your corresponding paragraph it will be rendered within a `cite`-tag. But, at this time it will only affect the representation of blockquotes. We are not sure yet, how this can also be applied to images, tables, lists, etc.

```
lorem ipsum ....

-- Some more citations
```

Tables

Tables, *as we hope*, are easy to interpret and to create. Simply use horizontal rules to separate cells. The header is always defined by the first line, while the second line is used to separate the table header from the body visually and to define the column alignment.

Markdown-format:

```
| Tables               |      Are      |  Cool |
| -------------------- |:-------------:| -----:|
| *** columns 3 is *** | right-aligned | $1600 |
| ** column 2 is **    |   centered    |   $12 |
| * zebra stripes *    |   are neat    |    $1 |
```

As you can see in the result, you can sort tables, by clicking onto the icon that appears on the right of every header cell. A table will then be either sorted ascending, descending, or not sorted, which means your initial row order will be restored.

Result:

Tables	Are	Cool
columns 3 is	right-aligned	$1600
column 2 is	centered	$12
zebra stripes	are neat	$1

The position of the colon defines whether a column should be centered, aligned to the left or to the right. By default, if you do not use colons, all columns are aligned to the left.

- centered ⟶ `:---:`
- right ⟶ `---:`
- left ⟶ `:---` or `---` (default)

Tables ⟷ Data (Demo) 💫

But why stopping here? A table, in many cases, is just a representation of a dataset. If so, why not simply visualizing it accordingly and plot a graph, display a chart or a map, or whatever fits the most for your data. At the moment we apply simple rules to identify the nature of your dataset and thus choose a visual representation.

> For more details have a look at section Fun With Tables, which provides and extensive overview onto all supported representation schemes.

The easiest and probably most obvious representation of a simple plot, would be the following. A header with some names and columns that contain numbers. The first column is interpreted as the main column and thus defines the *x* values, the rest is up to you. A cell is then only associated with a number, if the first "word", *sequence of characters separated by a space*, can be parsed as a number. The `0km` within this example gets ignored. So if you want certain values to be ignored, simply attach something directly to the number, or add a character in front of it.

```
| x's |  some y's  |                  dist |
| --- |:----------:| ---------------------:|
| 1   |    1 \$    |                 16 km |
| 2.2 |    2 \$    |                 12 km |
| 3.3 |    5 \$    |                  1 km |
| 4   | -12.333 \$ | 0km _will be ignored_ |
```

LiaScript identifies this pattern and automatically adds a button above the table, which allows to switch between the table and the "line chart" representation. You can modify the chart interactively and even download the resulting image.

x's	some y's	dist
1	1 $	16 km
2.2	2 $	12 km
3.3	5 $	1 km
4	-12.333 $	0km *will be ignored*

A function cannot possess different *y*-values for one *x*-value, thus, if you have two or more equal *x*-values, the resulting plot will be a scatter plot.

```
| x's |  some y's  |  dist |
| --- |:----------:| -----:|
| 1   |    1 \$    | 16 km |
| 2.2 |    2 \$    | 12 km |
| 3.3 |    5 \$    |  1 km |
| 4   | -12.333 \$ | -5 km |
| 4   |            |     1 |
```

x's	some y's	dist
1	1 $	16 km
2.2	2 $	12 km
3.3	5 $	1 km
4	-12.333 $	-5 km
4		1

Last but not least *bar-charts*. If the first column contains at least one cell, that cannot be parsed as a number while the others do have, then this table gets interpreted as a bar-chart. The first column thus defines your set of groups. It is now also possible to sort your table according to different columns, and to see this ordering also within the bar-chart representation.

```
| Animal          | weight in kg | Lifespan years | Mitogen |
```

Animal	weight in kg	Lifespan years	Mitogen
Mouse	0.028	02	95
Flying squirrel	0.085	15	50
Brown bat	0.020	30	10
Sheep	90	12	95
Human	68	70	10

As mentioned earlier, this is only a brief introduction into this topic. So check out section Fun With Tables for a complete overview.

Editing

Editing tables might seem tedious, but actually it is not. There is a huge number of plugins for different editors that you can use, which do the formatting for you. You can use them to quickly navigate through your cells, and some even allow sorting.

Editors: Plugins

- Atom: markdown-table-editor
- VS-Code: Markdown Table
- Obsidion: Advanced Tables

HTML

You can also use plain HTML in your Markdown, if you miss something. It will mostly work pretty good, but it should be used with caution, since some interpreters apply different rules. Some interpret everything within an HTML tag as HTML, while others allow mixing. Thus, HTML can contain Markdown, which contains HTML, which contains... By the way, LiaScript allows mixing. Thus, keep in mind that newlines and indentation are still relevant.

Markdown-Syntax:

```
<div style="color: green">

Test <q>**bold**</q> and <b>HTML bold</b> works also inline

![Beautiful Lenna](img/lenna.jpg "Image of Lenna with a hat")

</div>
```

The result shows how the inline-CSS is applied to all nested Markdown elements. However, if you want to apply some styling to your document, LiaScript supports another minimal invasive way of doing that. We will describe this in detail in section Styling.

Result:

Test “**bold**” and **HTML bold** works also inline

Image of Lenna with a hat

> See the following list for an complete overview onto all HTML elements:
>
> HTML Element Reference

If you use custom HTML instead of Markdown, then no styling will be applied. You can of course create more complex content or tables, this way you can apply your own styling to all elements.

CSS-Reference

If you want to, you can also copy the generated LiaScript structure and use our classes. Most Browsers include an inspector, which can be used to interactively inspect the entire DOM-tree.

Open Inspector: Ctrl+Shift+i or Ctrl+Shift+k

But, you can also import your own styles within the main document comment by using the `link` definition. We will explain this in more details within the macro section link.

```
<!--
...
link: file.css
...
-->

# Main Title
```

Details & Summary

The `details` and `summary` tags are standard HTML tags and GitHub also supports their usage with Markdown. These tags offer a neat way to define something what is nowadays called accordion. Thus, your user can click on the summary text to make the body of the `details`-tag appear.

Syntax:

```
<details>

<summary>**Honest Textbook ads (click to enlarge)**</summary>

!?[If High School and College Textbooks Were Honest - Honest
Ads](https://www.youtube.com/watch?v=lhSjYT7pWkw)

</details>
```

Result:

<details>
<summary>**Honest Textbook ads (click to enlarge)**</summary>

!?[If High School and College Textbooks Were Honest - Honest Ads](https://www.youtube.com/watch?v=lhSjYT7pWkw)

</details>

`<lia-keep>` 💫

If you want to embed more complex HTML, and only HTML, without taking care about indentation and formatting, then should use the `lia-keep` tag to surround your code.

```
<lia-keep>
  <style>
    table, th, td {
```

As it is demonstrated in the result, everything within this tag will be treated as HTML only, no Markdown parsing will be applied and indentation will be checked.

Result:

Header 1	**Header 2**
Cell 1	Cell 2
Cell 3	

This way, you could for example also import even more complex HTML-tables, pictures with multiple sources for different screen-sizes, and more. *With great power comes great responsibility.* Thus, you will also be responsibile for your styling.

Code

In Markdown, you can use a sequence of at least three subsequent backticks `` ` `` to indicate a code-block that should not be treated as Markdown, but instead contains some kind of code for which syntax-highlighting should be used, if possible. The first word after the backticks is used as an indicator, for which kind of syntax-highlighting should be applied.

```
``` python
import time
# Quick, count to ten!
for i in range(10):
    # (but not *too* quick)
    time.sleep(0.5)
    print(i)
```
```

In case you are wondering, how to embed a code-block into a code-block with backticks? Three backticks are the minimum, thus you can surround your Markdown code-block example with a sequence of 4 or more backticks. If you start with four backticks, LiaScript will parse everything as code until it reaches a matching number of backticks.

```
import time
# Quick, count to ten!
for i in range(10):
  # (but not *too* quick)
  time.sleep(0.5)
  print(i)
```

However, we are still in the Markdown world with static code visualization. LiaScript has also support for interactive programming, thus all of your code-snippet can be made executable and editable. This will be described in more detail in section Interactive Coding.

Differences to Markdown 💫

Markdown also supports adding code by using tilde `~` characters instead of backticks. This is at the moment not supported by LiaScript, but might be added in the future.

```
~~~ javascript
var a = "b"
~~~
```

Additionally, it is also possible in standard Markdown to use indentation with at least 4 spaces to mark a block or a line as code. In LiaScript this is treated differently. You can use indentation to keep your document readable. The two indicators for text-to-speech in the example are treated equally by LiaScript, but another Markdown interpreter will interpret the second example as a single paragraph, while the indicator in the first example will be treated as code, and thus be easier to read with any other Markdown interpreter (including the representation on GitHub).

```
This is not code ...

    Any kind of text with a 4 space
    indentation will be treated as code
    in Markdown ...

                --{{1}}--
This text will be spoken out loud in LiaScript.

--{{2}}--
This text will be spoken out loud too.
```

Projects 💫

If you want to bundle a couple of code-blocks into something that mirrors a project, you can achieve this with the following syntax. All code-blocks are simply attached to each other, in order to indicate a grouping. If you separate them at least by one newline, they will be treated separately. This will be pretty neat, if we introduce the concept of interactive code-blocks.

````
``` js     -EvalScript.js
let who = data.first_name + " " + data.last_name;

if(data.online) {
  who + " is online"; }
else {
  who + " is NOT online"; }
```
``` json    +Data.json
{
  "first_name" :  "Sammy",
  "last_name"  :  "Shark",
  "online"     :  true
}
```
````

You can define optional names within the head of your code-block. The starting plus `+` and minus `-` symbols are used to indicate, if the resulting code-blocks should be visible or hidden. However, you can change this, by clicking onto the resulting title-bar to either maximize or minimize the code-block.

EvalScript.js

```
let who = data.first_name + " " + data.last_name;

if(data.online) {
  who + " is online"; }
else {
  who + " is NOT online"; }
```

Data.json

```
{
  "first_name" :  "Sammy",
  "last_name"  :  "Shark",
  "online"     :  true
}
```

If you do not add a plus or a minus as a prefix to your file, the plus is used as default.

Supported Languages 💫

In most cases you can simply add the name of the language or the common filename ending into the head of a code snippet. Most Markdown interpreters will use highlight.js for language rendering, since we require also an editor with syntax highlighting capabilities, we use ace. Thus, the language support might differ to other systems. We therefor apply a mapping, so that you can still use all highlight.js short-codes but also those of ace.

Overview: https://github.com/LiaScript/docs/blob/master/Code.md

Horizontal rules 💫

At the moment it is possible to insert horizontal rules by adding lines with at least 3 dashes, longer sequences of dashes are allowed too. Common Markdown also allows to define such rules with asterisks `*`, but this is used in LiaScript to group blocks, as we will described later...

Markdown-Syntax:

```
some paragraph

---

something else

----------------
```

Result:

some paragraph

---

something else

Custom Styling

In order to support a nearly equal experience for all Markdown interpreters you should stick with the basic Markdown notation or use simple HTML-tags as much as possible, for example if you want to change the color of a sentence or a word.

However, LiaScript allows you also to inject attributes into all Markdown blocks and inline elements by attaching an HTML-comment to that specific object. If the content of this HTML-comment can be parsed as HTML-attributes, then these settings will be applied to the element attached.

```
<!--
style="color: red;" id = "elementID" class="foo bar"
-->
```

Further resources:

The following resources will give you a full overview onto the most common HTML attributes and onto styling elements. It might be pretty overwhelming at first glance, what is possible, but you will see, that with some basic elements you can already achieve a lot. And when it comes to HTML and styling, you can find examples for pretty much everything on your prefered search-engine, e.g. ecosia.

- **HTML attributes:**

  Checkout the following link to get a full overview onto all HTML attributes, which you can also apply in LiaScript: w3schools: HTML Attribute Reference

- **Styling with CSS:**

  In most cases inline-styling will be applied, the following website covers all CSS-Syntax elements, which then can be used simply by attaching comments in the following way:

  ```
  <!-- style="color: red; font-size: 20px;" -->
  ```

  w3schools CSS-Tutorial

Block-Styling

So what is actually a block in LiaScript or Markdown? Basically it is everything that is separated by a newline, such as a paragraph, a table, a code-block or a list. But it can also be a block of multiple blocks, such as a list, which may consist of different bullet points, where every bullet point can be a list of multiple blocks by itself.

- **Blocks:**
  - tables
  - paragraphs
  - lists (ordered/unordered)
  - blockquotes
  - horizontal rules
  - HTML-blocks (if standing alone)
- **LiaScript-Extensions:**
  - comments
  - citations
  - effects
  - quizzes
  - surveys
  - tables
  - ASCII-Art & Charts
  - executable code-blocks and projects

Settings for entire blocks can be set with a **starting** comment that includes all required HTML-attributes and can even contain animation settings. This can also be used to highlight specific elements of your slides.

LiaScript-Syntax:

```
<!-- class = "animated rollIn" style = "animation-delay: 3s; color: purple" -->
The whole text-block should appear in purple color and with a wobbling effect.
Which is a **bad** example, please use it with caution ...
```

Result: (be patient)

The whole text-block should appear in purple color and with a wobbling effect. Which is a **bad** example, please use it with caution ...

Additionally, this method can be used to overwrite some aspects of all Markdown element. The example shows how you can change the background color for a certain element. This comes quite handy, if you want to emphasize further some parts of your document.

```
<!-- style="background-color: tomato;"-->
> **Warning**
>
> You have to be aware, that this does not affect the
> font-color (dark/light mode). Try to use  pastel
> colors, or overwrite the color manually with:
>
>  `color: white;`
```

> **Warning**
>
> You have to be aware, that this does not affect the font-color (dark/light mode). Try to use pastel colors, or overwrite the color manually with:
>
> `color: white;`

The following example depicts the interconnection of nested block-elements. For the entire table and also for all other blocks, it is possible to set the properties for width, font-color and font size, which will be applied onto every cell. And every cell can overwrite these values simply by adding a style-comment as the first element. These settings are even preserved, if you reorder the table.

```
        <!-- style="width: 50%; min-width: 400px; color: red;
        font-size:20px" -->
| <!-- style="background: azure"--> Header 1 | <!-- style="background: brown"-->
Header 2        |
|:----------------------------------------
|:---------------------------------------------- |
| <!-- style="background: coral"--> Item 1   | <!-- style="background:
rgb(12,12,111)"--> Item 2 |
| <!-- style="background: cyan" --> Item 3   | <!-- style="background:
#b88608"--> Item 4        |
```

Result

Header 1	Header 2
Item 1	Item 2
Item 3	Item 4

There are some special (internal) formats for changing the appearance of code-blocks and how to deal with tables. These topics will be handled separately.

Inline-Styling

So what are inline elements? These are basically all the tiny parts, such as single words, bold-text, links, inline-code, but also images and videos. In contrast to blocks where you attach the comment to front, inline elements can be modified by attaching a comment to the end. That's it ...

LiaScript-Syntax:

```
This **is an important**<!-- style="color: red" --> part
of the text<!-- style="color: green; font-size: 10rem;" -->.


![image](...Creative-Tail-Animal-lion.svg)<!--
style = "width: 300px;
         border: 10px solid;"
class = "animated infinite bounce"
-->

Some blurry and black-and-white video:

!?[movie](https://www.youtube.com/watch?v=8pTEmbeENF4)<!--
style = "filter: blur(2px); grayscale(80%);"
-->
```

As you can see from the results, the entire bold text is treated as one block, whereby in the second case only the single word "text" gets modified.

Result:

This **is an important** part of the text.

Some blurry and black-and-white video:

CSS is a pretty powerful tool and by using HTML-comments to tweak your Markdown, you can still read the document with any ordinary Markdown interpreter that simply ignores these comments.

Images and Styling

Styling images might happen quite often. However, you have to be aware of the fact, that the modal view functionality is only possible if LiaScript is in total control of the image. Thus, it will handle the optimal scaling for you and adds a click-event to switch to the modal view.

```
![The Wave](...ave_off_Kanagawa_%28Kanagawa_oki_nami.jpg "without attribute
injection")

![Workplace](https://www.w3schools.com/htmL/workplace.jpg "with attribute are
added")<!-- usemap="#workmap" -->

<map name="workmap">
  <area shape="rect" coords="34,44,270,350" title="Computer" onclick="alert('You
  clicked the Computer')" >
  <area shape="rect" coords="290,172,333,250" title="Phone" href="#12">
  <area shape="circle" coords="337,300,44" title="Cup of coffee"
  href="#Projects">
</map>
```

Thus, if you click onto the first image, you will be able to inspect it in more detail. If you click onto the second image, then a map associated with this image is in charge of it, which handles click-events differently.

Result

without attribute injection

with attribute are added

> *— The example for the map was taken from w3schools.*

What else can you do

The following examples present some useful application of combining attribute injection into LiaScript components.

Hiding Content

There might be use cases where you either want to show some parts only on GitHub and provide and alternative view at LiaScript. As it was shortly introduced in the sections before, you can add comments to the start of every block to add additional attributes. These attributes can also be used as a trigger to hide or show content.

```
<!-- style="display:block" -->
<div style="display:none">

Visible only in LiaScript, but not on GitHub.

</div>

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

<!-- style="display:none" -->
<div style="display:block" id="fooBar">

Not visible in LiaScript, but on GitHub!

</div>
```

The attributes within the comment will overwrite the attributes within the block. Thus, if there would be more stuff within style, this will be overwritten too, but other attributes like `id` that are not contained within the comment won't be affected...

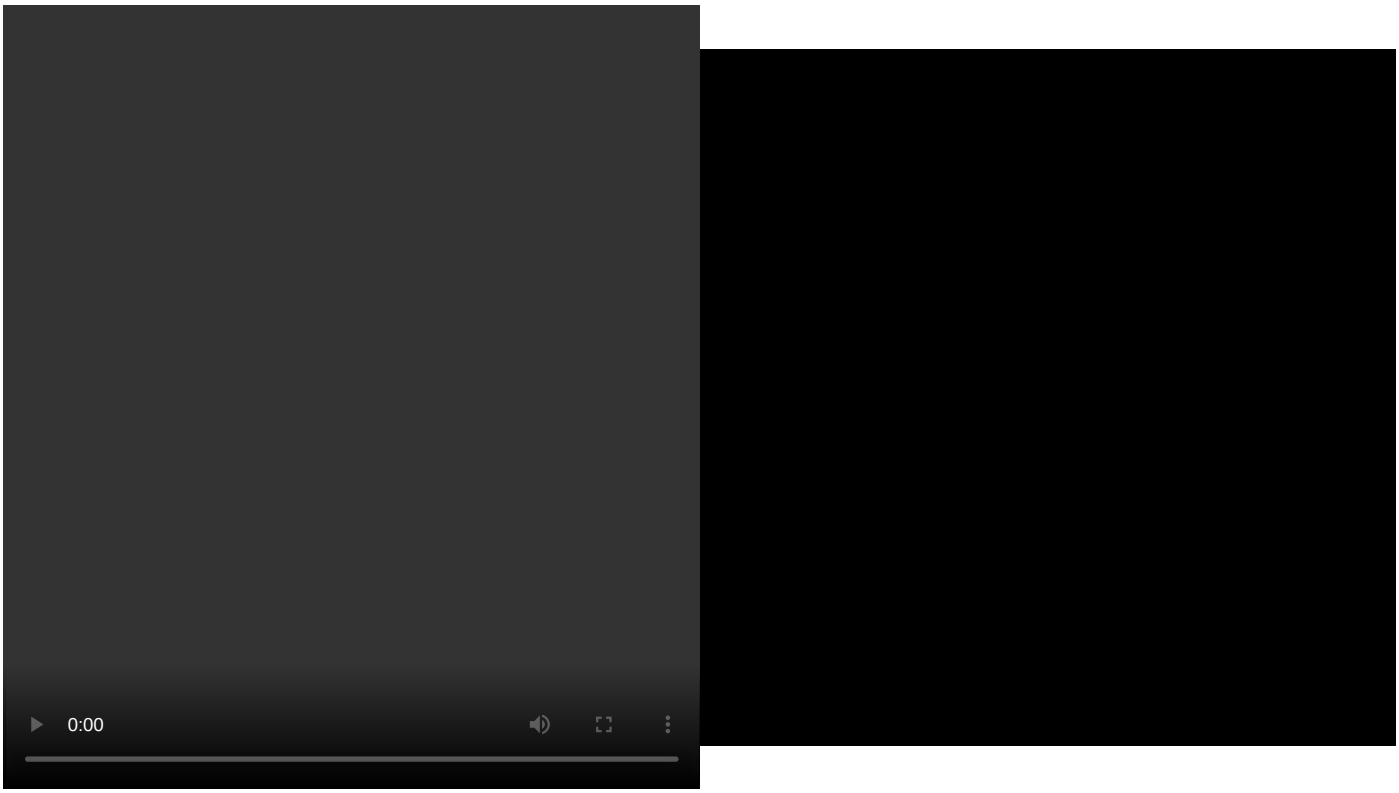
Customizing Multimedia

As you have seen previously sizing videos and applying CSS filters is easy. However, there might also be the case that you want to start a video from some special point or to play it automatically, when it appears.

If the video is yours, then you can use the following attributes `àutoplay` and `muted` to control the behavior and the additional fragment `#t=4,12` attached to the URL of the video, will tell the browser where to start and where to stop the video. The stop-parameter should be optional.

```
!?[Something about math](vid/math.mp4#t=4,12)<!--
autoplay="true"
muted="true"
-->
```

The resulting video starts immediately at second 4 and stops at second 10, and of course it will be muted.



Platform Diversity

However, if you are referencing other resource on platforms such as YouTube, Vimeo, DailyMotion, PeerTube or TeacherTube you can achieve something similar, but in a slightly different way. These settings have to be added to the URL of your resource and different platforms might have different capabilities.

In most cases, you can use something like `&autoplay=1`, `&muted=true` or `&mute=false` as it is depicted below:

```
!?[Multimedia](url/...?autoplay=1&mute=1&start=1895&end=1905)
```

But, different platforms support different functionalities. Here is a link list of the different possible settings. For PeerTube and TeacherTube we could not find any settings so far.

URL Parameters:

- DailyMotion
- Vimeo
- YouTube
- PeerTube
- TeacherTube

Math & Formulas

The following will not work on GitHub, but most Markdown-interpreters do support formulas with a LaTeX-like syntax. As it is common in many Markdown dialects, such as in Pandoc-Markdown, you can apply dollar signs to surround a "math"-environment. Everything within the dollar signs belongs to the formula only, there is currently no nesting of other HTML or LiaScript/Markdown allowed.

Use single dollar signs to define an inline formula, which will be treated as an ordinary text element.

Inline math-mode `$ \frac{a}{\sum{b+i}} $` → $\frac{a}{\sum b+i}$

Use double dollar signs to indicate a formula-block. This way you can also use multiple lines to define a formula or a set of formulas that will furthermore be displayed larger.

Multi-line math-mode can be applied by double dollars `$$ formula $$`

$$\frac{a}{\sum b+i}$$

Currently, we apply the KaTeX library for typesetting. If you are already familiar with LaTeX or MathJAX, as another alternative library, then you can start immediately to define formulas. If not, then check out some of the following resources.

- via KaTeX: http://katex.org
- Supported functions:

  https://katex.org/docs/supported.html
- Alphabetically sorted features:

  https://katex.org/docs/support_table.html
- Detexify: get the appropriate command by drawing symbols

  http://detexify.kirelabs.org/classify.html

Sometimes it might be tedious to find the right, command for the intended symbol, if you can draw it, then you should give detexify a try.

Formula Playground

Alignment is a typical use case for formulas. The code below, shows how this can be achieved by using ampersands, which are used as an anchor for the center of a formula.

```
\begin{split}
  a &=b+c \\
    &=e+f \\
    &=g+h+i+j\\
a+b+&c+d=12\\
\end{split}
```

$$
\begin{aligned}
a &= b + c \\
&= e + f \\
&= g + h + i + j \\
a + b + c + d &= 12
\end{aligned}
$$

If you want to number your formulas, we recommend using the `\tag` command to add or overwrite the reference number. Automatic numbering does not work well at the moment, since the formulas are rendered within a web component, and it does conflict with the LiaScript idea of animation, which we describe in a later part of this series.

```
\tag{33}
\begin{equation}
 a =b+c
\end{equation}
```

$$
a = b + c \tag{33}
$$

And finally it is possible to add some styling, but with KaTeX-functionalities, this includes some basic styling, with the same inline CSS, as we had described it in section Custom Styling. (Can you spot the strange looking german character *Eszett*.) And you can mark elements as links with `\href` and add images with the command `\includegraphics`.

```
\begin{Bmatrix}
    a & b & c & d & e & f \\
    g & h & i & j & k & l \\
    m & n & o & p & q & r \\
    s & t & u & v & w & x \\
    y & z & ä & ö & ü &
    \htmlStyle{color: red; font-size: 26px}{ß}
\end{Bmatrix}
\\
\href{https://katex.org/docs/supported.html#html}{\KaTeX HTML support}
\\
\includegraphics[height=0.8em, totalheight=0.9em, width=0.9em, alt=KA
logo]{https://katex.org/img/khan-academy.png}
```

$$
\left\{
\begin{matrix}
a & b & c & d & e & f \\
g & h & i & j & k & l \\
m & n & o & p & q & r \\
s & t & u & v & w & x \\
y & z & ä & ö & ü & \htmlStyle{color: red; font-size: 26px}{ß}
\end{matrix}
\right\}
$$

KaTeX HTML support

Footnotes

Before moving on to the LiaScript specific features, such as quizzes, surveys, animations, ASCII-art, etc., we would like present a last feature that is common to many Markdown dialects and these are footnotes. So, what are footnotes in general and when to use them.

> Footnotes are notes at the foot of the page while endnotes are collected under a separate heading at the end of a chapter, volume, or entire work. Unlike footnotes, endnotes have the advantage of not affecting the layout of the main text, but may cause inconvenience to readers who have to move back and forth between the main text and the endnotes.
>
> — *Wikipedia*

In LiaScript a "section", which is defined by a header and a body, resembles a page. Thus, the body might contain a couple of footnote-marks, while the actual footnotes have to be defined at the end of the body. Other Markdown interpreters might define a more "wide-spread" usage of footnotes, but in LiaScript this is not possible at the moment. We parse/interpret only one section at a time and not the entire document. So keep this in mind when using footnotes.

```
### header 3

body

body

body

footnotes

## ...
```

Standard-Footnotes

The standard way of creating footnotes is to attach a marker to important elements within your sections. A marker is defined by brackets with a starting `^`. Then you can insert numbers, symbols and words.

```
Something[^1] important[^2] about[^🦶] notes.
```

At the end of your body, you simply add a list with all indented remarks you want to add. This list starts with your markers, that are followed by a colon. Your notes or footnote-bodies can consist of multiple paragraphs and all other block-elements that you have mentioned so far, but they have to be indented with at least 2 spaces.

```
[^1]: **Something** and **anything** are concepts of existence in ontology,
  contrasting with the concept of nothing. Both are used to describe the
  understanding that what exists is not nothing without needing to address the
  existence of everything. The philosopher, David Lewis, has pointed out that
  these are necessarily vague terms, asserting that "ontological ...

  -- [Wikipedia](https://en.wikipedia.org/wiki/Something_%28concept%29)

[^2]:> It is **important** to mention that anything can be added to a footnote
  > also some piece of code, images, videos, etc.
  >
  > ??[How to Use Footnotes](https://www.youtube.com/watch?v=Gg6vXoHO95I)

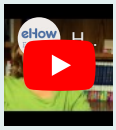
[^🦶]: Actually you are not forced to use numbers, you can use any kind of
      symbol or even words too.

      _But, please be consise._
```

By clicking onto the footnote, all of your comments will be shown in a modal view in all view-modes. If you are in "textbook" mode, then these footnotes are additionally displayed at the very end of the current "page".

Result:

Something [1] important [2] about [🦶] notes.

[1]	**Something** and **anything** are concepts of existence in ontology, contrasting with the concept of nothing. Both are used to describe the understanding that what exists is not nothing without needing to address the existence of everything. The philosopher, David Lewis, has pointed out that these are necessarily vague terms, asserting that "ontological ... — *Wikipedia*
[2]	It is **important** to mention that anything can be added to a footnote also some piece of code, images, videos, etc. 

Inline-Footnotes

The following idea of defining inline-footnotes was borrowed from LaTeX. If you want to add only some text without further Markdown-syntax then you can add this in parentheses to your footnote-mark. In this case it is not necessary to add a more sophisticated explanation to the end of the sections.

`Inline Footnote [x](__explanation__ in one line)` ⇒ Inline Footnote [x]

[1]	1. Actually you are not forced to use numbers, you can use any kind of * symbol or even words too.
[x]	**explanation** in one line

But please stick to one explicit format.

State

Some words about state. The LiaScript runtime-environment preserves the internal state of the following elements, depending on where you execute your courses and depending on the version.

- Tasks
- Quizzes
- Surveys
- Code
- `(<script>)`

LiaScript currently runs in three different environments. You can either share and read courses via the project website, generate SCORM-packages, which can be uploaded to an LMS, or create self-containing websites.

1. **LiaScript PWA**

   If you share your course via the website, then all the courses and the current course states are preserved within the Browser database IndexedDB locally. **All of the states?** Well ... the internal state of the previous components is only preserved, if your course has a version greater than or equal to `1.0.0`. The default version is `0.0.1`, which means that the course is in *development mode*, and thus the structure of your course might change every time. If a user reloads the course, then all states are cleared.

   - Where: https://LiaScript.github.io/course/
   - Source: liascript/src/typescript/connectors/Browser

2. **SCORM 1.2**

   SCORM stands for Sharable Content Object Reference Model and allows you to store the course state directly within every LMS that has support for SCORM 1.2. For this case you have to use our exporter, which translates your course into a "SCORM-compliant" zip-file. You can check if your LMS has support for SCORM, by visiting our LMS-overview.

   - Where: https://github.com/LiaScript/LiaScript-Exporter#scorm12
   - Source: liascript/src/typescript/connectors/SCORM1.2
   - Does your LMS support SCORM 1.2: LMS-Overview

3. **Base**

   You can also use our exporter-tool to create single websites of your course. But, this "base" does only support to store user-settings, such as style, mode, etc. This "base-connector" is also used for the live-server and editors. If you are a developer and want to extend the LiaScript support to store state within your backend, then this is the right place to look at. This module provides an abstract class, to which all statefull data is sent. You can simply inherit from this class and implement the access to your system.

   - Where: https://github.com/LiaScript/LiaScript-Exporter#web
   - Source: liascript/src/typescript/connectors/Base

For more information on versioning and how to use it, check out section `version` within the macros section. But for now, it is okay to know, that you can adapt, restructure and share your courses freely, if you are in dev-mode. If your course is ready to be launched, use versioning to preserve the state of your users.

Tasks

Before we show you how to create quizzes, we would like to introduce **tasks**, which were also a source of inspiration for us, for developing a similar syntax for quizzes and surveys. The GitHub flavored Markdown offers a very intuitive way of creating check-lists or task-lists.

```
**Which topics did you master so far?**

- [ ] Biology
- [ ] Chemistry
- [ ] Computer Science
- [X] Something about LiaScript
```

It is basically a list where the brackets are used to symbolize check-boxes and by using an upper- or lowercase x, it is possible to mark a checkbox as checked. In LiaScript however, it is possible for the user to manipulate these states, and they will be preserved if your document is in version 1 or greater.

Which topics did you master so far?

- [ ] Biology
- [ ] Chemistry
- [ ] Computer Science
- [x] Something about LiaScript

Tasks and Scripting

In case you are wondering where the **"Script"** in LiaScript comes from? We wanted to make documents more interactive, by embedding native support for scripting. Coding will become an essential skill in the future so why not using it to directly extend the native capabilities of static documents and its elements. Scripts can either be executed separately or they can be attached to tasks, quizzes, surveys, and code-snippets.

The example might look a bit weird for the moment, but it is only meant as a demonstration. You can use scripting in LiaScript, if you want to, but it is not required. Thus, whenever you change the state of the task-list, then the script below gets executed and the `@input`-macro will be substituted by current input. `output="tasks"` says, that the result is published under the topic "tasks", thus, every script that contains an `@input(`tasks`)` will be executed afterwards as well with the changed input. Task-lists simply produces and array with boolean values.

```
**What do you want to learn today?**

- [X] Biology
- [ ] Chemistry
<script output="tasks">"@input"</script>

<script style="width: 100%">
try {
  let task = @input(`tasks`) // interpret the output="tasks"

  if(task[0]) {
    send.liascript(`## Biology

Hey, great, you want to learn something about Biology.

* resource 1
* resource 2

The input from the tasks above was: \`[${task}]\``)
  } else send.clear()
} catch(e) { }
</script>
```

Depending on the input scripts can generate different results, it is even possible to return LiaScript code, which will be analyzed and rendered dynamically. If an empty string or undefined gets returned, then the script will not be visible to the user. We will describe all Lia-Scripting capabilities and features in more detail in a later chapter.

What do you want to learn today?

- [x] Biology
- [ ] Chemistry

Biology

Hey, great, you want to learn something about Biology.

- resource 1
- resource 2

The input from the tasks above was: `[true,false]`

If we are talking about embedding scripts, that perform some kind of calculation, data analysis, etc. Why shouldn't this be visible to the user as well? We all know what can happen when you cannot get access to primary data and the code that was used to analyze it. In LiaScript you can inspect these highlighted elements with rounded corners, which represent the result of a script, simply by double-clicking or double-tabbing. The user can manipulate them and observe the results, simply by changing the code. If the editor is closed, then the code gets reevaluated.

> Europe's entire austerity policy after the debt crisis was based on wrong conclusions drawn from incorrect data and false calculations from a **single Excel spreadsheet**.
>
> https://en.wikipedia.org/wiki/Growth_in_a_Time_of_Debt

But, this is enough for the moment, let us continue with quizzes...

Quizzes

Quizzes are an essential element of every online course, which allow students to reflect and test their knowledge. As mentioned before, we used the GitHub-flavored Markdown notation for tasks as an inspiration for quizzes.

Long story short, everything that is related to quizzes is defined as a combination of brackets with brackets or parentheses.

- `[[ ]]`
- `[( )]`

LiaScript currently supports 5 different types of quizzes and one, so-called generic type, which can be used to created custom quizzes of any kind.

1. Multiple-Choice: `[[X|x| ]] ...`
2. Single-Choice: `[(X|x| )] ...`
3. Matrix: as a combination of multiple- and single-choice quizzes
3. Text-Quiz: `[[solution]]`
4. Selection-Quiz: `[[ opt. 1 | opt. 2 | (solution opt.) ]]`
0. "Generic": `[[!]]`

Additionally, it is possible to tweak every quiz with additional features, such as hints, a resolution that is presented if the users solves the quiz by their own or clicks onto the "show solution" button. And finally, as you have seen it with tasks, you can attach scripts to each quiz in order to implement more sophisticated quizzes or simply to log the output.

1. Hints: `[[?]]`
2. Solutions: some arbitrary Markdown content
3. Associated scripts: to be executed, when the user clicks onto the "check"-button

Quiz Types

Within the following section we will introduce the 5 types of quizzes, which are currently supported by LiaScript. Additionally, you have to know that users cannot fail, by default it is possible to retry a quiz until it is solved, or the user clicks onto the resolve button. The only thing that is counted is the number of trials.

via GIPHY

If testing knowledge becomes something deeply personal, without any limitation on the number of trials, why should anyone cheat at all. In this sense, LiaScript is meant to be a language for learning and education instead of assessment. But, if you want to, you can also tweak this by using scripts 😉.

1. Multiple-Choice

A multiple-choice quiz can also be interpreted as a task list with a predefined solution. Thus, the only thing that has to be done, is to surround your "ASCII" checkboxes with additional brackets, which are followed by a line of Markdown/LiaScript.

```
- [[ ]] Empty means not checked
- [[X]] Uppercase `X` means checked ...
- [[x]] ... and lowercase `x` too ...
- [[ ]] **as defined in the first line** ...
```

As you can see from the example it does not matter, whether you use lowercase or uppercase `X` to mark a checked element.

Result:

- [ ] Empty means not checked
- [ ] Uppercase `X` means checked ...
- [ ] ... and lowercase `x` too ...
- [ ] **as defined in the first line** ...

The starting dashes for quizzes are also optional, you can omit them and GitHub and other Markdown renderer will present a single paragraph. Or, you can also use an indentation of at least 4 spaces, which will be presented by other interpreters as code. However, all the following representations will be interpreted by LiaScript as the same quiz.

```
Without starting dashes (paragraph):

[[ ]] Empty means not checked
[[X]] Uppercase `X` means checked ...
[[x]] ... and lowercase `x` too ...
[[ ]] **as defined in the first line** ...

With indentation of at least 4 spaces (code):

    [[ ]] Empty means not checked
    [[X]] Uppercase `X` means checked ...
    [[x]] ... and lowercase `x` too ...
    [[ ]] **as defined in the first line** ...
```

2. Single-Choice

If brackets are used to define checkboxes, why not using parentheses to indicate radio-buttons. In contrast to multiple-choice quizzes only one option can be selected.

```
- [( )] Not selected
- [(X)] **This one has to be selected**
- [( )] ~~Do not select this one ...~~
```

You can also omit the starting dashes and/or use indentation as with multiple-choice quizzes and use upper- or lowercase `X` to indicate the solution.

Result:

- ( ) Not selected
- ( ) **This one has to be selected**
- ( ) ~~Do not select this one ...~~

However, you can also define multiple possible solutions a user might select only one option, but multiple might be correct.

```
What is the correct spelling of H(D)D?

    [(X)] hard (**disk**) drive
    [( )] hard (**desk**) drive
    [(x)] hard (**disc**) drive
```

Result:

What is the correct spelling of H(D)D?

◯ hard (**disk**) drive

◯ hard (**desk**) drive

◯ hard (**disc**) drive

Thus, if the user selects the first or the last option, the quiz will be marked as solved, or if the user clicks onto the "show solution" button, both options will be presented.

How to create a true/false quiz?

Using single-choice quizzes, it is also possible to define something simple as a True or False quiz.

```
Do you know an easier way of creating quizzes?

    [( )] Yes
    [(X)] No
```

Result:

Do you know an easier way of creating quizzes?

◯ Yes

◯ No

3. Matrix-Quiz

You can also combine multiple single- and multiple-choice quizzes into one larger matrix-quiz. The "vector" quizzes are now simply defined as rows, whereby you have to define a custom header, which represent the solution options. It makes no real difference if you use parentheses or brackets within the header line, use brackets if you want to insert parenthesis and vice versa.

```
- [[male (der)] (female [die]) [neuter (das)]]
- [    [X]           [ ]             [ ]     ]  Mann - German for man
- [    ( )           (X)             ( )     ]  Frau - German for woman
```

However, within the quiz-body, which is contains the quiz-vectors as rows, the brackets and parenthesis make a difference. The first row gets presented as multiple-choice with checkboxes, while the second line defines a single-choice with radio buttons.

Man or woman is obvious, but you guess the remaining German grammatical genders?

male (der)	female [die]	neuter (das)	
☐	☐	☐	Mann - German for man
◯	◯	◯	Frau - German for woman
☐	☐	☐	Junge - German for boy
◯	◯	◯	Mädchen - German for girl
☐	☐	☐	Paprika - German for bell pepper
◯	◯	◯	Joghurt - German for yogurt

Fun fact:

The number of elements within a row does not necessarily have to match the number of elements within the head. Although it might be rendered strangely, this does not cause an error. You as a course designer can see, that there might have a mistake occurred, but maybe you use it with a purpose.

```
[[option 1] [option 2]]
[   [X]                    ] only with option 1
[   [ ]        [X]         ] with two options
[   [X]        [ ]       [X]] with three options
```

option 1	option 2		
☐	only with option 1		
☐	☐	with two options	
☐	☐	☐	with three options

4. Text-Quiz

A simple text-input quiz is defined only by two brackets that contain the solution string. For these simple kind of quizzes, it is currently not allowed to add a starting dash, but you can use optional indentation. The reason for this is, that in the future we would like to extend the usage of text- and selection-quizzes to be used everywhere, within paragraphs or tables to from clozes.

```
What did the fish say when he swam into the wall?

    [[dam]]
```

Result:

What did the fish say when he swam into the wall?

The user input will be compared with your solution-string, and if and only if they are equal the quiz will be labelled as solved. For different reasons it might be necessary to clean up the input, in order to deal with different types of spelling, uppercase and lowercase, etc. Fort this purpose scripts can be associated to a quiz, see therefor section Associated Scripts.

5. Selection-Quiz

Finally, a selection-quiz is a collection of LiaScript/Markdown options that are separated by vertical bars `|`. The solution is defined by the option or options that are surrounded by parenthesis. An option can be any kind of LiaScript inline element and since vertical bars `|` are used as separators, you can simplify the readability by using newlines.

```
What is the derivative function of $f(x) = x^6$?

[[ $f'(x) = 6$ | ( $f'(x) = 6x^5$ ) | $f'(x) = 5x^6$ ]]

Can be also written as:

    [[  $f'(x) = 6$
    | ( $f'(x) = 6x^5$ )
    |   $f'(x) = 5x^6$
    | ( **This will be counted as correct too...** )
    ]]
```

This type of quiz allows you to have multiple correct options and starting dashes are not supported yet, due to their future usage as native input elements for clozes. And you can use indentation, such as it is possible for all quizzes.

Result:

What is the derivative function of $f(x) = x^6$?

selection

Can be also written as:

selection

0. Generic Quizzes

Generic quizzes can be used to define any kind of quiz. The course-developer can define own types of quizzes, whereby the following combination of brackets with an additional exclamation mark has to be associated with a script.

```
[[!]]
<script>
  const random = Math.random()

  alert("random value: "+ random)

  random < 0.2
</script>
```

This script does only generate a random number, outputs the result in an `alert` and checks if the random value is less than 0.2. And if so, the quiz will be marked as solved otherwise not. Thus, the last statement has to be `true` in order to mark a quiz as solved, any other value will be accounted as `false`.

The usage of associated scripts for quizzes will be described in detail in section Associated Scripts.

Notes about Questions

Not all of the previous examples we have shown were perfect in the sense of accessibility. Although these quizzes look good "for us", they might not be perfectly for people that require a screen-reader to go through a course. However, you can improve your quizzes by asking your question as a paragraph, which is followed by your quiz. This way LiaScript will associate or label the quiz with your question.

```
Ask a question as an ordinary Markdown paragraph,
which is followed by what?

    [[quiz]]
```

If your question requires more elements, then it is also possible to group it with supplemental elements by putting it into a single HTML-element such as a `div`. The `div` will get a unique ID and the question will be labeled with this ID by using the aria-labelledby attribute.

```
<div>

Ask a question as an ordinary Markdown paragraph,
which is followed by what?

1. important
2. also relevant ...
3. [an image](http://...)

</div>

    [[quiz]]
```

Tweaks

All of the following elements can be added to any type of quiz that you have seen before. This includes and arbitrary number of hints, a more detailed solution or a custom script that handles the user input.

via GIPHY

Hints

To any type of quiz you can add as many hints as you want, the pattern is simply two brackets with an additional question mark. If you want to, you can also add starting dashes to the hints.

```
What is $37 + 15$?

   [[52]]
   [[?]] the solution is larger than 50
   [[?]] it is less than 55
   [[?]] it should be an even number

This is also a valid quiz ...

[[52]]
- [[?]] the solution is larger than 50
- [[?]] it is less than 55
- [[?]] it should be an even number
```

The user can reveal these hints, step by step by clicking onto the light bulb or "show hint" button.

What is $37 + 15$?

Solution

And finally, some quizzes might require some more explanations, if they are solved or not. Therefor, simply use two "lines" that are defined by at least three asterisks to group your solution. The solution explanation can contain an arbitrary number of LiaScript elements.

```
What is $37 + 15$?

[[52]]
[[?]] the solution is larger than 50
[[?]] it is less than 55
[[?]] it should be an even number
***************************************************************************

52 is the correct solution, you get this by adding:

``` ascii
                                .------.
                                |      |
                                |      v
                                |
                                |     (1)
  37            3(7)          |     (3)x          37
+ 15          + 1(5)          |   + (1)x        + 15
---- -->      ------ --> |    ------ -->     ----
  ??            (12)          |     (5)2          52
                 |            |                  ====
                 '--------'
                    carry
```

??[MS-DOS Math Game](https://archive.org/embed/msdos_Super_Solvers_Teasure_MathStorm_1992)

****
```

In this case the solution contains an ASCII-art drawing, but it can be anything, such as a an image, a video, or even something like a small game

What is $37 + 15$?

Associated Scripts

The following section contains a list of use-cases and ideas how to use quizzes and scripts in association to generate a more complex user experience.

via GIPHY

Cleaning up

If you are using for example the text-quiz, and you want to react to different ways of spelling or to clean starting and trailing white-spaces. Then you can attach a script to your quiz. The `@input` is marker that will be substituted by the current user input. `trim` and `toLowerCase` are JavaScript functions/methods that are used to clean the input string. The last statement of your quiz-script defines if the quiz is marked as solved or not. Only if the last statement evaluates to `true` it is marked as solved, for any other value it is simply a failed trial.

```
What did the fish say when he swam into the wall?

[[dam]]
<script>
let input = "@input".trim().toLowerCase()

input == "dam" || input == "damn"
</script>
```

Try out the following quiz. You can now enter "damn" or "dam" in different ways, surrounded by spaces.

What did the fish say when he swam into the wall?

[ ]

Quiz `@input`s

`@input` will be replaced by the current state of the quiz that should be checked. If you are not sure, how the input might look like, you can always experiment with a simple `alert` that shows the current input.

Text-quiz:

The output or in LiaScript terms the `@input` gets substituted by a string, without apostrophes. This might look strange since you have to add apostrophes within the code, but all inputs are substituted without any additional type formatting, this way they can be used in different ways and even contain peaces of code.

```
[[text]]
<script> alert("@input") </script>
```

Result:

[ ]

Selection-quiz:

A selection quiz is defined by a number that represents the current input, starting from `0`. The initial state is marked with `-1` to indicate that nothing has been selected so far. In this example `A` would be represented by `0`, `B` by `1`, `C` by `2` and so on.

```
[[A|B|C|(D)]]
<script> alert(@input) </script>
```

Result:

selection

Multiple-choice:

A multiple-choice quiz is represented by an array of ones and zeros, whereby a `1` means checked and `0` the opposite.

```
[[X]] A
[[ ]] B
[[X]] C
<script> alert("@input") </script>
```

Result:

- [ ] A
- [ ] B
- [ ] C

Single-choice:

A single-choice quizzes behave similar as selections, the state is represented by a number, starting from `0`, which represents the first option. A quiz that has been not touched by the user will return `-1` as input value.

```
[(X)] 0
[( )] 1
[(X)] 2
<script> alert("@input") </script>
```

Result:

- ◯ 0
- ◯ 1
- ◯ 2

Matrix:

As matrices are arrays of "vector" quizzes, their input state is represented as an array of multiple-choice and single-choice states. Thus, a list of lists with zeros and ones as well as numbers.

```
Weird, isn't it ;-)

[[male (der)]   (female [die])   [neuter (das)]]
[    [X]           [ ]             [ ]     ]  Mann - German for man
[    ( )           (X)             ( )     ]  Frau - German for woman
[    [X]           [ ]             [ ]     ]  Junge - German for boy
[    ( )           ( )             (X)     ]  Mädchen - German for girl
[    [X]           [X]             [ ]     ]  Paprika - German for bell pepper
[    (X)           (X)             (X)     ]  Joghurt - German for yogurt
<script>alert("@input")</script>
```

Result:

male (der)	female [die]	neuter (das)	
▢	▢	▢	Mann - German for man
◯	◯	◯	Frau - German for woman
▢	▢	▢	Junge - German for boy
◯	◯	◯	Mädchen - German for girl
▢	▢	▢	Paprika - German for bell pepper
◯	◯	◯	Joghurt - German for yogurt

Generic:

In case of a generic quiz, the developer is responsible for defining and storing state. Thus, there will be no input. The only case where `@input` gets substituted by a value, is when the user clicks onto the resolve button. This is indicated by the input `true`.

```
[[!]]
<script>alert("@input")</script>
```

Async & Errors

There might be the rare case, where you have to send the user input to an external service to check them. Scripts in LiaScript can also define asynchronous code, as it is displayed within the example.

```
What is $37 + 15$?

[[52]]
<script>
setTimeout(function(){
  alert("your input was: '@input'")
  send.lia("true")
}, 1000)

"LIA: wait"
```

In LiaScript, we can use some command strings that trigger a certain behavior, these are global to all scripts and will be described in more detail in section Communication.

- `"LIA: wait"`: mark the script as still running
- `"LIA: stop"`: halt the execution of the script

Additionally, you can see a function-call within the example. `setTimeout` triggers the execution of the internally defined function, which performs an `alert` with the user input and then sends "true" back to the internal quiz-event handler. Every script is executed with an additional `send` object and `send.lia` means send this back to LiaScript. Thus, the result could also be evaluated by an external service, and this result will then be used to control the quiz.

- `send`: general object to for communicating with the LiaScript internal event-handler for a specific entity, for more information see section Communication

Let us extend this example a little bit further. The `send` to `lia` can have two meanings, one if everything was ok while the second might be the result of an error. `send.lia` can have three values, the first is the message, the second one contains more detailed information about the result, which is not required at the moment, while the last value tells LiaScript if everything was ok or by using `false` that an error has occurred. The default value is always ok.

```
What is $37 + 15$?

[[52]]
<script>
setTimeout(function(){
  alert("your input was: '@input'")

  if ("@input" === "") {
    // generic error message
    send.lia("You have to fill in something into the input field", [], false)
  } else {
    // this is equal to send.lia("some message", [], true)
    send.lia("true")
  }
}, 1000)

"LIA: wait"
</script>
```

Thus, if you try out the following example, you will have to wait one second until the result is evaluated, which in this case always results in a solved quiz. But, if you do not add any input, then an error message will be displayed.

Result:

What is $37 + 15$?

Obfuscate Quizzes

We would like to note, that this is only one method to obfuscate a quiz, by using JavaScript, in this case by using the function `btoa`.

```
[[...]]
<script>
// btoa("solution") == "c29sdXRpb24="
"c29sdXRpb24=" == btoa( "@input".trim().toLowerCase() )
</script>
```

`btoa` creates a Base64-encoded ASCII string, while `atob` does the opposite. You can try out to the following two scripts to encode and decode different strings.

- `btoa`: solution » c29sdXRpb24=

  (Beautiful TO Awful)

- `atob`: c29sdXRpb24= » solution

  (Awful TO Beautiful)

The result is a quiz, where it is not possible to see the solution immediately, simply by getting a glimpse onto the raw course document.

Result:

Quizzes & Macros

It might look tedious to create these trailing scripts and to add them again and again to your document, which makes a course harder to maintain and to develop. However, in LiaScript you can define macros, that can be used to solve repetitive tasks.

As depicted in the example, you can define custom macros within an HTML-comment of a section or within the main comment at the head of every document. Wherever you write the macro `@customQuiz` within your document, this macro will be substituted by the content defined in the HTML-comment, which comes between `@customQuiz` and `@end`. For more information see section Macros.

```
##### Quizzes & Macros
<!--

@customQuiz
[[...]]
<script>
"@0" == btoa( "@input".trim().toLowerCase() )
</script>
@end

-->

...

@customQuiz(c29sdXRpb24=)


@customQuiz(NTI=)
```

You can further parameterize your macros, as you can see in the code, there is an `@0`, which can be interpreted as a placeholder that shall be substituted by the first parameter of your macro call.

That's it, you can add as much as `@customQuiz`-macros and if you do some changes, you do not have to go through all scripts, but instead you only have to change one single macro.

The solution should be "solution":

[ ]

What is $37 + 15$?

[ ]

By defining macros it is furthermore possible to create LiaScript libraries that can imported into other courses, see therefor also section Macros.

Final thoughts on wrong solution

The quiz for adding two numbers was actually taken from the presentation of Greg Wilson, and it gives a good example for what scripts can be used for. Instead of logging and checking if an answer is correct or not, we should use it to assist students while they are learning and try to identify patterns in wrong answers.

As you have seen it for tasks, the result of a script can also be published, in this case the topic is defined by the `output` attribute.

```
What is $37 + 15$?

[[52]]
<script output="quiz:37+15">
  if ("@input" == "52") {
    true
  } else {
    "@input"
  }
</script>
```

Other scripts can be defined, that are subscribed to the output of the quiz, see the different `@input` macro. These scripts are executed when the output of the previous script changes. Every script can be used to identify another pattern, since students can be wrong for different reasons. We need to identify such patterns and react properly instead of measuring only fail and success.

```
<script style="display: block">
//   3(7)       (3)7
// + 1(5)     + (1)5       The first and second steps were correct,
// ------ ->  ------  ->   but the student forgot to carry the 1
//   (12)       (4)2       in the 10 digit column.

if ("@input(`quiz:37+15`)" == "42") {
  send.liascript(`## Maybe you forgot to carry...

Checkout the follow video to recap carrying.

!?[Carrying for larger digits](https://www.youtube.com/watch?v=VPsYRPdlIpU)
`)
} else ""
</script>

<script style="display: block">
//   3(7)       (3)7       The first and second steps were also
// + 1(5)     + (1)5       correct, but carrying was not provided
// ------ ->  ------  ->   properly, maybe there is a problem with
//   (12)      (4)12       understanding digit columns...


if ("@input(`quiz:37+15`)" == "412") {
  send.lia(`You are nearly there, there might be a problem with the 10 digit
  columns...`)
```

With LiaScript you do not have to come up immediately with a perfect online course, you can adapt it in little steps and let it increasingly grow.

What is $37 + 15$? This time you should make the mistakes 42 and 412.

Generic++

Ok, German articles behave strangely. This is a more complex example of a generic quiz that is connected to multiple scripts with an input, which are implemented with a single LiaScript macro.

Do you know which German articles belong to which gender in which case?

Case	Masculine	Feminine	Neuter	Plural
Nominativ	select	select	select	select
Genitiv	select	select	select	select
Dativ	select	select	select	select
Accusativ	select	select	select	select

The following code-block shows how this quiz was actually implemented.

```
##### Generic++
<!--
@article: <script output="@0" value="select" input="select"
options="der|die|das|dem|den|des">
            if("@input(`articles`)" == "true") {
              "@1"
            } else {
              "@input"
            }
          </script>
-->

                                --{{0}}--
Ok, German articles behave strangely. This is a more complex example of a
generic quiz that is connected to multiple scripts with an input, which are
implemented with a single LiaScript macro.

Do you know which German articles belong to which gender in which case?

[[!]]
<script output="articles">
if ("@input" !== "true") {
  "@input(`11`)"=="der"  &&  "@input(`21`)"=="die"  &&  "@input(`31`)"=="das"  &&
   "@input(`41`)"=="die" &&
  "@input(`12`)"=="des"  &&  "@input(`22`)"=="der"  &&  "@input(`32`)"=="des"  &&
   "@input(`42`)"=="der" &&
  "@input(`13`)"=="dem"  &&  "@input(`23`)"=="der"  &&  "@input(`33`)"=="dem"  &&
   "@input(`43`)"=="den" &&
  "@input(`14`)"=="den"  &&  "@input(`24`)"=="die"  &&  "@input(`34`)"=="das"  &&
   "@input(`44`)"=="die"
}
</script>

| Case      | Masculine        | Feminine         | Neuter           | Plural
      |
| --------- | ---------------- | ---------------- | ---------------- |
---------------- |
| Nominativ | @article(11,der) | @article(21,die) | @article(31,das) |
@article(41,die) |
| Genitiv   | @article(12,des) | @article(22,der) | @article(32,des) |
@article(42,der) |
| Dativ     | @article(13,dem) | @article(23,der) | @article(33,dem) |
```

Surveys & Classrooms

A survey or questionnaire from our perspective is a quiz without a predefined solution. Thus, the syntax is the same to quizzes, but instead of a solution you have to provide options. If you use the LiaScript-Exporter to generate SCORM-packages of a course, then the state of the quizzes, surveys, and tasks will be stored within the LMS-backend.

via GIPHY

But, if you are also using LiaScript for your live presentations, you can also open a classroom directly from your Browser and create a shared experience, where all the connected peers will get the same and anonymous view. We will explain the classroom-idea in more detail at the end of this section.

Text-Inputs

As already mentioned, the difference to quizzes is, that you have to provide options or placeholders. In case of a text-input, you have to provide a placeholder, which consists of at least 3 underscores that are separated spaces.

You can have a single text-input by using the following pattern:

```
**This is a one liner, you can use commas `,` to separate your inputs:**

    [[___]]
```

This is a one-liner, you can use `commas` to separate your inputs:

Enter some text...

In this type of survey, you can use commas to separate different phrases or keyword/topics. Apply this for short comments or reactions only.

Similar to text-quizzes, use the following syntax to define a text-survey, where the number of underlines defines the presented line numbers:

```
Please describe your opinion in a few sentences:

    [[___ ___ ___ ___]]

    [[______   ___   ___   ______]]
```

Thereby, it does not matter how many spaces you add or how long the underscore lines are, both definitions will result in the same `textarea` input.

Please describe your opinion in a few sentences:

Enter some text...

Single-Choice Vector

And also this kind of survey is similar to a single choice quiz, but in this case, numbers within parenthesis are used to define some kind of variable identifier. That is why they do not have to be in order.

```
Select one option:

    [(1)] option 1
    [(2)] option 2
    [(3)] option 3
    [(0)] option 0
```

The result will look like the one presented below:

Select one option:

- ◯ option 1
- ◯ option 2
- ◯ option 3
- ◯ option 0

But, instead of numbers, you can also define more complex option names. **There is only one difference, the options that start with numbers, will be plotted in the classroom presentation as distributions, whereby not numbers will be presented as categorical values.**

```
Select one option:

- [(very good)]       I like it very much
- [(good)]            It is ok
- [(bad)]             I don't like it
- [(something else)]  I am not sure
```

As you can see from the example, you can apply different styles for encoding surveys too. Either you use indentation, so that other Markdown interpreter will show it as code, or you use lists.

Select one option:

- ◯ I like it very much
- ◯ It is ok
- ◯ I don't like it
- ◯ I am not sure

Multi-Choice Vector

Similar to multi-choice quizzes, you can define multi-choice survey vectors with a number in double square brackets. But, and this is also possible for all other kinds of surveys, you can define some kind of variable name with a starting colon.

```
What are your favorite colors?

[[red]]         is it red
[[green]]       green
[[blue]]        or blue
[[dark purple]] last chance ;-)
```

Within a LiaScript classroom, this represents categorical values.

What are your favorite colors?

- ☐ is it red
- ☐ green
- ☐ or blue
- ☐ no one likes purple (last chance 😉)

If you want to turn this into a continuous representation, you have to work with starting numbers.

```
What are your favorite colors?
```

What are your favorite colors?

- [ ] is it red
- [ ] green
- [ ] or blue
- [ ] last chance 😉

Single-Choice Matrix

For defining survey blocks, you only have to have a header row, whose definition is also used by the trailing rows.

Markdown-format:

```
What is your opinion about LiaScript?

    [(totally)(agree)(unsure)(maybe not)(disagree)]
    [                                             ] LiaScript is great?
    [                                             ] I would use it to make online
    **courses**?
    [                                             ] I would use it for online
    **surveys**?
```

Result:

What is your opinion about LiaScript?

totally	agree	unsure	maybe not	disagree	
◯	◯	◯	◯	◯	LiaScript is great?
◯	◯	◯	◯	◯	I would use it to make online **courses**?
◯	◯	◯	◯	◯	I would use it for online **surveys**?

And also in this case, again, you can use numbers to display the summary as categorical.

As depicted below, the brackets `[ ]` do not have to match with the first line.

Markdown-format:

```
What is your opinion about LiaScript?

- [(1 totally)(2 agree)(3 unsure)(4 maybe not)(5 disagree)]
- [                ] LiaScript is great?
- [                ] I would use it to make online **courses**?
- [                ] I would use it for online **surveys**?
```

Result:

What is your opinion about LiaScript?

1 totally	2 agree	3 unsure	4 maybe not	5 disagree	
◯	◯	◯	◯	◯	LiaScript is great?
◯	◯	◯	◯	◯	I would use it to make online **courses**?
◯	◯	◯	◯	◯	I would use it for online **surveys**?

Multi-Choice Matrix

The multi-choice blocks are self-explanatory and behave exactly as the single-choice blocks from the previous section.

Markdown-format:

```
[[1][2][3][4][5][6][7]]
[                      ] question 1 ?
[                      ] question 2 ?
[                      ] question 3 ?
```

Result:

1	2	3	4	5	6	7	
▢	▢	▢	▢	▢	▢		question 1 ?
▢	▢	▢	▢	▢	▢		question 2 ?
▢	▢	▢	▢	▢	▢		question 3 ?

Associated Scripts

As it was done with tasks and quizzes, you can also associate a script to every survey to change its behavior or to send the result to an external service. By default, you have to provide some input in order to submit the data. Within the following example, both cases are treated as errors, whether the user just hits the submit button or if only spaces and tabs will be entered. The script returns `true`, if the input matches your experience. Otherwise, an error message will be presented to the user, depending on the input length. If you just return `false`, then no information will be presented to the user.

```
Please enter some spaces at first:

[[___]]
<script>
  let input = `@input`.trim()

  if (input.length > 4) {
    true
  } else if (input.length == 0) {
    send.lia("Please enter some text", [], false)
  } else {
    send.lia("Please provide some meaningfull input", [], false)
  }
</script>
```

Result:

Enter some text...

The input-data for different types of summaries is the same as for quizzes, but you can check the input always manually via `alert` function. **Note that `console.log` will not work in this case at the moment.**

```
What are your favorite colors?

[[1 red]]         is it red
[[2 green]]       green
[[3 blue]]        or blue
[[4 dark purple]] last chance ;-)
<script>
  alert(`@input`)
</script>
```

What are your favorite colors?

- [ ] is it red
- [ ] green
- [ ] or blue
- [ ] last chance 😉

Classroom Experience

We try to develop a simple classroom experience, light without any centralized authority or server. Therefor, we currently apply distributed Web3.0 technologies, which synchronize the state of a classroom across multiple connected users/browsers. At the moment, we can synchronize and visualize quizzes and surveys and display an anonymous overview onto the results. The basic idea is, if you join a room, you bring your data with you, if you leave the room then all of your data will be removed from the global state. Thus, nothing is stored nothing is logged, and you have the control over your data. All associated servers run only as relays.

I don't want Classrooms

You can also disable this feature for your course, simply by adding the command `classroom: disable` or `classroom: false` to your main definition.

```
<!--
author: ...

classroom: disable
-->

# Title
```

Working with Classrooms

If you are on the LiaScript website and if you have a course started, you can directly switch to the classroom settings by clicking onto the share button.

When you click onto the classroom button, you should be presented with the classroom settings, where you have to choose one backend service. We would prefer to use GunDB. Some services like Beaker require you to run your course from another browser, or you will have different settings.

We provide different information for the different services that can be applied. However, what is similar to all is that you have to define a room name that must be unique. To help you, you can click onto the circle arrow symbol and a name will be generated randomly for you. The passwords are optional.

If you then click onto connect and a connection could be established, the classroom settings will be closed automatically. Otherwise, an error message should be provided. If everything worked fine, you will see, at least one user within the classroom and the URL of your course will have changed. You can now either share the new URL, which contains all required configurations, or you can send the room name and the password and the course-URL separately to your peers. In this case, they will have to repeat these steps.

In order to disconnect, you will have to go to the classroom settings again and click onto the disconnect button. Again, the URL of the course will change back to the original representation.

Note: You can try this out, if you open LiaScript on different browsers and go back to the quizzes and surveys sections. You should experiment a bit with the resulting presentations. Additionally, try to disconnect and observe the effect on the connected instances.

Classroom Experience

The following video shows how to open, share, and close a LiaScript classroom.

Enabling the Classrooms in any LMS

If you have exported your course to a SCORM or IMS package with the LiaScript-Exporter you can also establish classrooms between different users of the same course within a single LMS or between different ones, such as Moodle or ILIAS.

By default, the classroom-feature is disabled within any exports, but as you can disable it for your course, it is also possible to enable it. Thus, at first, you have to enable this feature within your course, before you export it.

```
<!--
...

classroom: enable

-->

# Your Course
```

Secondly, the classroom name has to be truly unique and surrounded by quotations. By default, the URL of your Markdown file and the room-name are used to generate a unique ID for the classroom, which prevents collisions between different courses, which accidentally use the same room name. However, by surrounding the room-name with single or double quotation marks, you instruct LiaScript to use this specific room name and to ignore the course URL. **All the roommates have to enter exactly the same name for the classroom.** That's it!

Room name: `"This has to be a truely unique name 129281715#123"`

Future Classrooms

As already mentioned, we currently only synchronize quizzes and surveys. Other elements will be added in the future, such as distributed pair-programming, user roles, asking questions, etc.

via GIPHY

If you are interested in the implementation stuff, we build classrooms with the help of CRDTs and try to make LiaScript as backend-agnostic as possible. You can add support for your own systems here.

Implementation: https://github.com/LiaScript/LiaScript/tree/development/src/typescript/sync

Effects

There are currently three types of what we call effects, these are:

1. Animations
2. Comments via Text-to-Speech
3. and Playback functions

All of these elements can be used inline and on block-level. Every effect is defined by two braces:

```
  --{{1}}--
Spoken comments.

  {{1-2}}
Blocks that appear at animation
step 1 and disappear on step 2.

  {{|>}}
To be read aloud when user clicks
the on the play button...
```

Animations are only visible and comments are read out aloud in "Presentation" or in "Slides" mode. If you set the mode to "Textbook", then all animations and comments will be displayed on one slide, where you have placed them within the document. Within the other two modes, they are revealed step by step on every click.

You need to balance these features properly, so that your course can be read in Textbook mode, used for presentations, and more.

Animations

Animations are defined by two curly braces and one starting and one optional ending number. Animations can be associated to single blocks, multiple blocks and also inlined.

```
{{start-stop?}} | {start-stop?}{inline}
```

Block Animations

Animations can be associated to blocks, simply by adding two curly braces on top of a block. We recommend applying indentation of at least spaces to an animations definitions. Other Markdown renderer will highlight this as code, such that it is easier to read.

```
  {{1}}
This is an example for a *single* block animations.

  {{2-3}}
This one will appear on animation step 2 and disappear on 3.

{{4}} This is also ok, but it will look be harder to spot on GitHub.
```

Use a starting and an ending number, if you want the element to disappear at a certain point.

Result:

This is an example for a *single* block animations.

This one will appear on animation step 2 and disappear on 3.

This is also ok, but it will look be harder to spot on GitHub.

Multi-Block Animations

Similarly, as it was done for quizzes, as described in section Quiz-Solution, you can group multiple markdown blocks by lines of asterisks. Simply add the curly braces with the animation definition above the upper line.

```
      {{1-2}}
**********************************
```

Blocks can also have a starting and disappearing number.

Depending on your preferred style, you can also use HTML-tags to group blocks. These will then be displayed slightly different on GitHub.

Result:

This is an example...

that	contains
multiple	blocks.

As an alternative, you can also use HTML-tags ...

... to surround multiple blocks.

Inline Animations

Inline effects can be used in nearly all LiaScript elements. In this you will have to unpack the curly braces, the first pair surrounds the animation definition, while the second pair contains all inline elements that should appear and or disappear.

```
* no effect here
* but in this line {2}{show ***second***}
* as well as this one {1-2}{show ***first*** remove on __second__}
```

- no effect here
- but in this line show ***second***
- as well as this one show ***first*** remove on **second**

Combinations & Styling

Animations can also be grouped freely, such that one multi-block animation can contain multiple block animations and one block can also contain further inline animations:

```
<!--
class="animate__animated animate__backInUp"
style="background:#CCC; padding:3rem; min-height: 40vh; border-radius: 3rem"
-->
                {{1}}
*****************************************

This block contains {2}{multiple} inline animations.
With some
{2-3}{styled}<!-- class="animate__animated animate__flash" -->
elements as well.

<!-- class="animate__animated animate__backInDown" -->
                  {{3}}
!?[Animated paintings](https://www.youtube.com/watch?v=-DDphfCnFQc&autoplay=true)

*****************************************
```

This block contains multiple inline animations. With some styled elements as well.

Any kind of CSS can also be added to an animation, as it was already described in section Custom Styling. Additionally, we had added an additional CSS stylesheet to the main HTML-comment of this document. This will load Animate CSS, but you can use any other CSS library or custom styling as well.

Animate.css documentation: https://animate.style/

```
<!--
author: ...

link:   https://cdnjs.cloudflare.com/ajax/libs/animate.css/4.1.1/animate.min.css
-->

# LiaScript
```

Comments: Text 2 Speech

The idea of a comment is that they should be associated to witch animations. When animation x is revealed, then the comment x is read aloud. Like in a PowerPoint presentations, when one element appears and the presenter says something, clicks the next element appears and is also commented. Thus, a comment is a paragraph that is marked by two curly braces, which contain a number, and two dashes around the braces. If multiple comments have the same number, then they will be replayed in the order of appearance.

```
        --{{1}}--
This will be spoken out loud.

        --{{2}}--
This will be spoken out loud too,
but at animation step 2.

        --{{2}}--
Don't forget me.

    {{1}}
__I am animation 1 {2}{and 2 too}.__
```

This will be spoken out loud.

This will be spoken out loud too, but at animation step 2.

Don' forget me.

I am animation 1 and 2 too.

If you change the presentation mode, then you will see that these comments will be displayed in place in "Textbook" mode. In "Slides" mode they will also be presented to the user, while they will be hidden in "Presentation" mode. If you switch off the sound, then this is the mode that can be used for presenting content, while the others can be used for self studying.

Voices & Language

But where does the voice come from? In LiaScript within the initial comment, you can use the `lang` macro to define the document translation and `narrator` to define the default voice. Currently, we are using responsivevoice to offer the same language capabilities in all browsers and devices. It is possible to change the `narrator` on different layers, globally within the main comment tag, per slide, and also per comment, by inserting the voice into the comment definition.

1. Speech-Engine: https://responsivevoice.org
2. Default `narrator` must be defined within the initial comment, otherwise `US English Male` is used
3. You can overwrite the default `narrator` per slide, by attaching a comment to the title tag
4. Use `--{{number}}--` to indicate what is spoken and when
5. Use `--{{number voice}}--` to change the voice for this particular comment
6. It is also possible to define custom macros for language definitions
7. You can have multiple comments with the same number, those will be combined, but only the voice of the first one is used
8. See a complete list of all supported voices in section `narrator`

```
<!--
author:   ...
lang:     en
narrator: US English Male
-->

# Title

...

#### Voices & Language
<!--
narrator: UK English Female
@Tanja:   Russian Female
-->

                --{{1}}--
The entire ***Markdown*** paragraph right below the effect definition in double
minus notation is sent to responsivevoice to speak the text out loud in
`Presentation` or `Slides` mode.

                --{{3 Deutsch Female}}--
Markdown ist eine vereinfachte Auszeichnungssprache, die von John Gruber und
Aaron Swartz entworfen und im Dezember 2004 mit Version 1.0.1 spezifiziert ...

                --{{4 @Tanja}}--
«Для торжества зла достаточно бездействия хороших людей».
```

The entire ***Markdown*** paragraph right below the effect definition in double minus notation is sent to responsivevoice to speak the text out loud in `Presentation` or `Slides` mode.

Markdown ist eine vereinfachte Auszeichnungssprache, die von John Gruber und Aaron Swartz entworfen und im Dezember 2004 mit Version 1.0.1 spezifiziert ...

«Для торжества зла достаточно бездействия хороших людей».

Hidden Comments

Sometimes, it might be necessary to add a comment or to read a part aloud to underline a certain point, which might be necessary in the narrated mode, but not in the Textbook. Therefor, it is possible to put your TTS output into simple HTML comments. This won't be shown to anyone and also not visible on most other Markdown parsers and renderer.

```
<!-- --{{1}}--
Speak this out, but do not show it to anyone.
-->
```

The associated hidden comment to this point will not be visble in Textbook mode!

Translations

If you click on the language settings, you can either click on the element "Translate with Google (experimental)" and select another language. In this case, a JavaScript library will be injected that implements the translation feature. As you can see from the example, not all parts will be translated. Code blocks will not be translated by default, as well as comments that have been marked with another voice than the default. These comments remain as they are, while LiaScript tries to find an appropriate voice for the new language and gender.

Screenshot of a translation to Hebrew with Google-translate.

You can attach specific parameters to the comment that prohibit or enforce translation. By default, Google will search for the `class` definition `translate` or `notranslate`, but other external browser plugins might also take into account the HTML5 attribute `translate`. So it is always good to use both definitions.

```
<!-- class="notranslate" translate="no" style="color: red" -->
                --{{2}}--
I will not be translated!

                --{{3 Russian Female}}--
«Для торжества зла достаточно бездействия хороших людей».
```

I will not be translated!

«Для торжества зла достаточно бездействия хороших людей».

You can attach these language attributes to any kind of Markdown block or inline element to control the translation. Note that code environments and foreign language comments will be tagged automatically with no-translate.

Playback

Since Text2Speech output is baked into the LiaScript notation, why not using it on purpose for language learners. Simply add a stylized play-button to the effect definition to indicate, what should be spoken out loud. You can of also use different voices.

```
  {{|>}}
This entire paragraph will be spoken out __LOUD__.

  {{!> Australian Female}}
* But in this case, this can also be combined
* with a couple of
* - different
  - Markdown elements
  - whether it makes sense or not.
```

This entire paragraph will be spoken out **LOUD**.

- But in this case, this can also be combined
- with a couple of
- - different
  - Markdown elements
  - whether it makes sense or not.

Playback-Blocks

And like it was introduced for animations, you can also group multiple block. Simply add as many Markdown-blocks between two lines of asterisks, and they will be interpreted as one larger block.

```
<!--style="background: #EEE; padding:2rem"-->
                {{|>}}
**************************************************
This entire paragraph will be spoken out __LOUD__.

* But in this case, this can also be combined
* with a couple of
* - different
  - Markdown elements
  - whether it makes sense or not.

**************************************************
```

This entire paragraph will be spoken out **LOUD**.

- But in this case, this can also be combined
- with a couple of
- - different
  - Markdown elements
  - whether it makes sense or not.

As an **alternative**, you can also use an HTML-tag like `<section>` or `<div>` to group blocks, the LiaScript result will stay the same, but it will be rendered differently on other Markdown interpreters.

Playback-Inlines

And as presented before, you can also use inlining for Playback elements, as it was used for animations by simply using two pairs of braces. Depending on your preferences and the current context, it is also possible to define the stylized play-button with a vertical line or an exclamation mark. Like in the presented example, the exclamation will not interfere with the table definition.

```
| English              | German                          | Russian
                    | Arabic male               | Arabic female               |
| -------------------- |:--------------------------------
|:------------------------------------- |:------------------------- |
--------------------------- |
| {!>}{I go}           | {!> Deutsch Male}{ich gehe}       | {!> Russian Male}{я
хожу}                | {!> Arabic Male}{أَذْهَبُ}  |                             |
| {!>}{you go}         | {!> Deutsch Male}{du gehst}       | {!> Russian Male}{ты
ходишь}              | {!> Arabic Male}{تَذْهَبُ}  | {!> Arabic Female}{تَذْهبين} |
| {!>}{he/she/it goes} | {!> Deutsch Male}{er/sie/es geht} | {!> Russian Male}{он
/ она / оно ходит} | {!> Arabic Male}{يَذْهَبُ}  | {!> Arabic Female}{تَذْهَبُ}  |
| {!>}{we go}          | {!> Deutsch Male}{wir gehen}      | {!> Russian Male}{мы
ходим}               | {!> Arabic Male}{نَذْهَبُ}  |                             |
| {!>}{you go}         | {!> Deutsch Male}{ihr geht}       | {!> Russian Male}{вы
ходите}              | {!> Arabic Male}{تَذْهبون} | {!> Arabic Female}{تَذْهَبْنَ} |
| {!>}{they go}        | {!> Deutsch Male}{sie gehen}      | {!> Russian
Male}{они ходят}           | {!> Arabic Male}{يَذْهبون} | {!> Arabic
Female}{يَذْهَبْنَ} |
```

The result is displayed within the table, all elements can be played on demand.

English	German	Russian	Arabic male	Arabic female
I go	ich gehe	я хожу	أَذْهَبُ	
you go	du gehst	ты ходишь	تَذْهَبُ	تَذْهبين
he/she/it goes	er/sie/es geht	он / она / оно ходит	يَذْهَبُ	تَذْهَبُ
we go	wir gehen	мы ходим	نَذْهَبُ	
you go	ihr geht	вы ходите	تَذْهبون	تَذْهَبْنَ
they go	sie gehen	они ходят	يَذْهبون	يَذْهَبْنَ

Hiding Text

If you only want to show the play buttons but not the text, it is possible to use some HTML tricks. The easiest way is to put your text into an HTML element and to remove it from the screen by using a `span` whose content is shifted from the screen. Simply styling the element with `display: none` will not work, since the TTS function requires the text to be rendered within the DOM and the translation via Google will not work, if the element is not visible.

But, since it is possible to define custom Macros, we can also apply a more elegant way. We define a set of local macros directly within a comment attached to the current heading. The `@play` macro has two parameters, one for the voice and the other for the text, the other macros are simply shortcuts for the voice that pass the text as the second parameter to the `@play` macro. Within the Arabic macro, it is also possible to define the gender of the narrator.

```
#### Hiding Text
<!--
@play: {!> @0}{<span style="display: inline-block; text-indent:
-10000px">@1</span>}
@en: @play(UK English Male,@0)
@de: @play(Deutsch Male,@0)
@ru: @play(Russian Female,@0)
@ar: @play(Arabic @0,@1)
-->

| go        |         EN          |         DE          |            RU
   |      AR male      |      AR female      |
| --------- |:-------------------:|:-------------------:|:-----------------------
--:|:-----------------:|:-------------------:|
| I         |      @en(I go)      |    @de(ich gehe)    |        @ru(я хожу)
   | @ar(Male,أَذْهَبُ)   |                     |
| you       |     @en(you go)     |    @de(du gehst)    |      @ru(ты ходишь)
   | @ar(Male,تَذْهَبُ)   | @ar(Female,تَذْهَبِين) |
| he/she/it | @en(he/she/it goes) | @de(er/sie/es geht) | @ru(он / она / оно
ходит) | @ar(Male,يَذْهَبُ)   | @ar(Female,تَذْهَبُ)   |
| we        |     @en(we go)      |   @de(wir gehen)    |       @ru(мы ходим)
   | @ar(Male,نَذْهَبُ)   |                     |
| you       |     @en(you go)     |    @de(ihr geht)    |      @ru(вы ходите)
   | @ar(Male,تَذْهَبُون) | @ar(Female,تَذْهَبْنَ)  |
| they      |    @en(they go)     |   @de(sie gehen)    |      @ru(они ходят)
   | @ar(Male,يَذْهَبُون) | @ar(Female,يَذْهَبْنَ)  |
```

The result is a table with playback buttons only, where the text is hidden and where the main language will be translated while the other languages remain.

go	EN	DE	RU	AR male	AR female
I	▷	▷	▷	▷	
you	▷	▷	▷	▷	▷
he/she/it	▷	▷	▷	▷	▷
we	▷	▷	▷	▷	
you	▷	▷	▷	▷	▷
they	▷	▷	▷	▷	▷

Animations to Playback

Since we are using the double braces notation for playback elements, this can also be used in combination with animations. Simply by adding an appearance number, or an appearance and disappearance number. Depending on the current state of the animation, this will result in different sentences.

```
    {{1 |>}}
This is an example where {|> 1-2}{I go} _{|> 2}{I am going}_ to work.
```

You have to keep in mind, that this will work as intended if the user is not in Textbook mode. Otherwise, all elements will be read out loud and nothing is hidden.

▷

This is an example where ▷ I go ▷ *I am going* to work.

Interactive Code

In section Code we had already introduced how code-snippets can be defined in Markdown and LiaScript. In this part we will introduce how such code can be made executable and editable.

via GIPHY

Starting simple

Any code snippet can be made interactive by attaching a script-tag to the end. The idea is the same as for tasks, quizzes, and surveys. The `@input` is simply a placeholder that gets replaced by the current user-input.

````
``` javascript
var i=0;
var j=0;
var result = 0;

for(i = 0; i<10; i++) {
    for(j = 0; j<i; j++) {
        result += j;
        console.log("(", i, ",", j, ") result", result)
    }
}
// the last statement defines the return statement
result;
```
<script>@input</script>
````

If the code is in JavaScript and provides a full and executable example, not only pseudo-code, then it can be directly executed. Just click on the run-button directly below the editor. The console output will be piped into the local shell that appears below the code-snippet.

```
var i=0;
var j=0;
var result = 0;

for(i = 0; i<10; i++) {
    for(j = 0; j<i; j++) {
        result += j;
        console.log("(", i, ",", j, ") result", result)
    }
}
// the last statement defines the return statement
result;
```

```
( 1 , 0 ) result 0
( 2 , 0 ) result 0
( 2 , 1 ) result 1
( 3 , 0 ) result 1
( 3 , 1 ) result 2
( 3 , 2 ) result 4
( 4 , 0 ) result 4
( 4 , 1 ) result 5
( 4 , 2 ) result 7
( 4 , 3 ) result 10
( 5 , 0 ) result 10
( 5 , 1 ) result 11
( 5 , 2 ) result 13
( 5 , 3 ) result 16
( 5 , 4 ) result 20
( 6 , 0 ) result 20
( 6 , 1 ) result 21
( 6 , 2 ) result 23
( 6 , 3 ) result 26
( 6 , 4 ) result 30
( 6 , 5 ) result 35
( 7 , 0 ) result 35
( 7 , 1 ) result 36
( 7 , 2 ) result 38
( 7 , 3 ) result 41
( 7 , 4 ) result 45
( 7 , 5 ) result 50
( 7 , 6 ) result 56
( 8 , 0 ) result 56
( 8 , 1 ) result 57
( 8 , 2 ) result 59
( 8 , 3 ) result 62
( 8 , 4 ) result 66
( 8 , 5 ) result 71
( 8 , 6 ) result 77
( 8 , 7 ) result 84
( 9 , 0 ) result 84
( 9 , 1 ) result 85
( 9 , 2 ) result 87
( 9 , 3 ) result 90
( 9 , 4 ) result 94
( 9 , 5 ) result 99
( 9 , 6 ) result 105
( 9 , 7 ) result 112
( 9 , 8 ) result 120
120
```

It is possible to re-run the example, but you can also make changes, such as the number of loops. When you execute your modified example, a new version will be added to the end of the list. It is possible to go back and forth between versions and if a previous version gets modified, this will create append also new version.

Projects

Multiple different code snippets can be combined to form a larger projects too. It requires to write them in a row. You can give them names, if you add a second parameter after the highlighting definition. Add a `+` or `-` to the front of your filename, in order to indicate, if it should be visible by default or not.

As previously mentioned the `@input` macro gets substituted by the input of the editor, but you can pass also a number to indicate which macro should be substituted by which code block (`@input(0)` is equivalent to `@input`).

```
``` js     -EvalScript.js
let who = data.first_name + " " + data.last_name;

if(data.online) {
  who + " is online"; }
else {
  who + " is NOT online"; }
```
``` json    +Data.json
{
```
```

The result is a project which consists of two files. Each file can be edited separately, while the script tag provides only some basic glue-code that tells LiaScript what to do with the input.

EvalScript.js

```
let who = data.first_name + " " + data.last_name;

if(data.online) {
  who + " is online"; }
else {
  who + " is NOT online"; }
```

Data.json

```
{
  "first_name" :  "Sammy",
  "last_name"  :  "Shark",
  "online"     :  true
}
```

```
Sammy Shark is online
```

Loading external Resources

If you want to make use of some external functionality, you can also load additional JavaScript modules into LiaScript. Simply add the `script` command to the main definition of your LiaScript header. For more information on this, take a look at section `script` in the Macros chapter.

```
<!--
author: ...

script: https://cdn.rawgit.com/davidedc/Algebrite/master/dist/algebrite.bundle-fo
r-browser.js
-->

# Title

...
```

If this library has been loaded, it can be used also directly within the script-tag. In this case we made use of the Computer-Algebra-System (Algebrit), which is used to solve some algebraic equations.

```
```javascript
f=sin(t)^4-2*cos(t/2)^3*sin(t)

f=circexp(f)

defint(f,t,0,2*pi)
```
<script> Algebrite.run(`@input`) </script>
```

The result is a fully functional computer algebra editor where you can experiment and modify equations.

```
f=sin(t)^4-2*cos(t/2)^3*sin(t)

f=circexp(f)
```

```
-16/5+3/4*pi
```

Code API

The easiest way to execute some code, is simply to add a script-tag to the end of your code-block. But, sometimes an execution takes longer or requires to execute some asynchronous code. For this purpose, LiaScript offers a simple API and event system that will be explained in more detail. For simplicity all of the following examples will only contain JavaScript code that gets interpreted just by executing `@input`.

`send` - Object

To every executed piece of code a `send` module is associated, which handles all required communication with that specific code-block or project and the outer world. Thus, every `send` module does only exists in this particular scope and it offers different methods for different problems.

- `send.log` : output logging information
- `send.lia` : send messages and control commands
- `send.handle` : handler terminal inputs
- `send.register` :
- `send.dispatch` :

To start with, there is a `log` method, which can be used to send different types of outputs directly to the console.

```
/* send.log(type, sep, content)
*
* params:
*  - type: one of the follow strings "debug", "info", "warn", "error",
  "html", "stream"
*  - sep: a string like separator, mostly for newlines "\n"...
*  - content : represents a list of list
*/

send.log("debug", "", ["This is a debug information"])
send.log("warn", "", ["This is a warning", 12, [1,2,3]])
"fin"
```

```
This is a debug information
This is a warning 12 [1,2,3]
fin
```

But we provide shortcuts for this, via the internal `console`, which does actually the same. Nevertheless, `send.log` gives you a little more power when you start to create your own LiaScript libraries and you have to handle multiple outputs. And as you can see, you can also pass HTML content directly.

```
console.debug("these are short hands for send.log('debug' ... ")
console.warn("warn")
console.log("info")
console.error("and red for errors")

console.html("<b>Some more fancy stuff:</b> <img width='40px' src='https
  ://www.math.uni-bielefeld.de/~jsauter/Octahedron.gif'/>")

"fin"
```

```
these are short hands for send.log('debug' ...
warn
info
and red for errors
Some more fancy stuff:
fin
```

Return types

As you may have noticed, the last statement of an executed code-block does also define the `return` statement. The output is alway interpreted as a string and handled this way.

```
1089
```

However, there are some results that are treated differently. These are strings that start with `"LIA: "`. The string `"LIA: stop"` for example, is used to tell the system to simply stop the execution, there will be no further output. In this case, you can used `console.log` to print the result to the terminal.

```
33*33

"LIA: stop" // if you remove this string the result of the
            // calculation will be visible in the shell
```

If you directly want to send control messages at various stages of the script, you can use `send.lia`, which will also directly output the result at the console

If the execution of your code may take longer, include some asynchronous calls or you need to call an external service, you can tell this by finishing with the statement `"LIA: wait"`, which will show arrows that loop forever or until they receive a `"LIA: stop"` signal.

```
setTimeout(function(){
  console.warn("end of execution")

  send.lia("some other ouput")

  send.lia("LIA: stop") // this stops the execution
}, 3000)


"LIA: wait" // wait until you you send a stop
```

```
end of execution
some other ouput
```

You might probably wonder, why there is a need for `send.lia`, if you could also pipe output to the console by using `send.log` or the shorthand `console.log`. As it will be described in the section about error handling, `send.lia` gives you more control, not only over the shell but also over the editor, see section Error Handling.

The code example below, depicts how the terminal can be exposed, following the previous logic, we only have to define `"LIA: terminal"` as the last statement in order to switch on the terminal mode, that will be active until the user clicks on the stop button, formerly known as the execute button, or a `"LIA: stop"` is send from the script, or the browser gets refreshed.

In order to handle the `input` and the `stop` events from the terminal in the JavaScript world, we have to define handles, as it is done in the code below. Simply add a function, that evaluates or sends your terminal inputs to a foreign server or a websocket, or ... whatever you want ... And use the stop handle to close a connection or do some other stuff, if the user clicks onto the stop button.

```
send.handle("input", input => {
  try{
    console.log(eval(input))
  } catch (e) {
    console.error(e.message);
  }
})

send.handle("stop", e => { alert("execution stopped") })

function close(){
  send.lia("LIA: stop")
}

"LIA: terminal" // execute the code and
```

A little side note, there is also a function called `console.clear`, that does what it says it clears the console, but it is not a realized by `send.log`, instead this is a shortcut for `send.lia("LIA: clear")`. These were all

`"LIA: ..."` commands, the next section you will learn how to use the custom built in event system to get even more stuff done.

Ping Pong

All previous functions were executed only within the scope of a single code-block, but sometimes it is necessary to connect with other external events. For this purpose, the `send` object also contains two further generic functions, these are `register` and `dispatch`. Simply use a unique identifier to name an event, which is followed by a callback-function that does whatever you want. Both callbacks are defined in local scope, but by using `register` and `dispatch` you can send

messages to any code-block. The example below also delivers the messages successfully, if both code-blocks would be defined on different slides.

```
send.register("ping", function(e){
  console.warn("ping", e)
})

send.handle("input", input => {
  send.dispatch("pong", input)
})

"LIA: terminal" // execute the code and
```

```
send.register("pong", function(e){
  console.warn("pong", e)
})

send.handle("input", input => {
  send.dispatch("ping", input)
})

"LIA: terminal" // execute the code and
```

Adding such script-tags to the end of every code-block can be very tedious and cumbersome. Your Markdown will contain a lot of not necessary copy & paste code. To reduce this, LiaScript offers the possibility to define custom macros, and even more, you can directly import macros from other courses. Section Macros is used to describe this issue in more.

Error Handling

As mentioned earlier, `send.lia` can do more then just passing messages to the terminal output. The editor that is currently used by LiaScript is ace, which allows to mark lines with warnings and errors. Since there is no name associated to a file (like with the `@input(0)` macro). You have to use a list of lists, which contain all necessary information that you want to pass to the editor. The list element is associated with the code-block, starting from top to bottom.

```
send.lia( "ups, something went wrong",
  [[{ row : 1,
      column : 1,
      text : "insert your error info here",
      type : "error"
    },
    {
      row : 2,
      column : 1,
      text : "some general warnings about this line",
      type : "warning"
    }
  ]],
  false); // use true if the output should appear in "info" mode

"LIA: stop";
```

```
ups, something went wrong
```

If this seems to complicated, you can also throw and error by using `LiaError`. The second parameter is used to define the number of code-blocks you use and then the only thing you require, is to add as much detailed information, which is similar to the previous example.

```
// create a new error with an error message that is
// connected to only one code-block, if you have more,
// increase the number ...
var error = new LiaError("ups something went wrong", 1);

error.add_detail(0, // <- associated code-block
  "insert your error info here",
  "error", 1, 1);

error.add_detail(0, // <- associated code-block
  "some general warnings about this line",
  "warning", 2, 1);


throw(error);
```

```
ups something went wrong
```

Examples

Running JSCPP

Teaching other language-basics is also possible, for this example we applied JSCPP to run simple C++ programs:

```cpp
#include <iostream>
using namespace std;

int main() {
    int a = 120;
    int rslt = 0;
    for(int i=1; i<a; ++i) {
        rslt += i;
        cout << "rslt: " << rslt << endl;
    }
    cout << "final result = " << rslt << endl;
    return 0;
}
```

```
rslt: 1
rslt: 3
rslt: 6
rslt: 10
rslt: 15
rslt: 21
rslt: 28
rslt: 36
rslt: 45
rslt: 55
rslt: 66
rslt: 78
rslt: 91
rslt: 105
rslt: 120
rslt: 136
rslt: 153
rslt: 171
rslt: 190
rslt: 210
rslt: 231
rslt: 253
rslt: 276
rslt: 300
rslt: 325
rslt: 351
rslt: 378
rslt: 406
rslt: 435
rslt: 465
rslt: 496
rslt: 528
rslt: 561
rslt: 595
rslt: 630
rslt: 666
rslt: 703
rslt: 741
rslt: 780
rslt: 820
rslt: 861
rslt: 903
rslt: 946
rslt: 990
rslt: 1035
rslt: 1081
rslt: 1128
rslt: 1176
rslt: 1225
rslt: 1275
rslt: 1326
rslt: 1378
rslt: 1431
rslt: 1485
rslt: 1540
rslt: 1596
rslt: 1653
rslt: 1711
rslt: 1770
rslt: 1830
rslt: 1891
rslt: 1953
rslt: 2016
rslt: 2080
rslt: 2145
```

```
rslt: 2211
rslt: 2278
rslt: 2346
rslt: 2415
rslt: 2485
rslt: 2556
rslt: 2628
rslt: 2701
rslt: 2775
rslt: 2850
rslt: 2926
rslt: 3003
rslt: 3081
rslt: 3160
rslt: 3240
rslt: 3321
rslt: 3403
rslt: 3486
rslt: 3570
rslt: 3655
rslt: 3741
rslt: 3828
rslt: 3916
rslt: 4005
rslt: 4095
rslt: 4186
rslt: 4278
rslt: 4371
rslt: 4465
rslt: 4560
rslt: 4656
rslt: 4753
rslt: 4851
rslt: 4950
rslt: 5050
rslt: 5151
rslt: 5253
rslt: 5356
rslt: 5460
rslt: 5565
rslt: 5671
rslt: 5778
rslt: 5886
rslt: 5995
rslt: 6105
rslt: 6216
rslt: 6328
rslt: 6441
rslt: 6555
rslt: 6670
rslt: 6786
rslt: 6903
rslt: 7021
rslt: 7140
final result = 7140
```

Interactive Coding

Why should code examples not be interactive and editable, especially if it is JavaScript or any other language that has been ported to it? Simply add the required resources to the initial comment with keyword `script`.

1. Add resource to main-comment: `script: url.js`
2. Add a trailing script-tag to your code: `<script>@input</script>`
3. A project with multiple files can be realized with `@input(0)`, `@input(1)`, ..., `@input(n)`.

And add an additional script tag to the end of your language definition with an `@input` macro. This element is afterwards substituted with your code and executed. We provide some basic examples within the following section.

Use the `@input` macro as a parameterized function in projects. The number defines the the file, starting from 0.

Styling

As for Tables, it is also possible to apply some basic styling attributes to the editor. At default code-snippets that ar not executable will not show line numbers in order to be used also for pseudo-code, while executeable blocks will show line numbers, while the later ones can be edited and not the others. However, you can apply the follow attributes to make some definitions explicit. We tried to apply the common ACE notation. Attributes have to be applied per code-block, as it is shown in the example:

````
<!-- data-showGutter="false" -->
```cpp
// some C++ code without line numbers
:::
```
<!-- data-readOnly="true" -->
```hpp
// some header-file with lineNumbers,
// that cannot be edited
```
<script>
...// your execution code
</script>
````

Attributes:

- `data-firstLineNumber`: change the initial line number to any number you prefer (default: `data-firstLineNumber="0"`).
- `data-fontSize`: change the default font-size, which has to be defined with `pt` (default `data-fontSize="12pt"`).
- `data-readOnly`: whether it is an executable snippet or not, there are different default values, you can either set only `data-readOnly` to make it read-only or pass it a boolean value (`data-readOnly="false"`)
- `data-showGutter`: same as with read-only
- `data-tabSize`: this takes an integer to represent the default tab-size replacement (default `data-tabSize="2"`)
- `data-theme`: your default theme as in your settings is applied, but you can change this to any of the ace-themes, eg: `Chaos`, `Eclipse`, `Soliarized Light`, ...
- `data-marker`: use this to highlight aspects of your code, you have to apply the following pattern `data-marker="y1 x1 y2 x2 color type;"`. You start with a row and column and end with a row and a column. Then you can apply one of the predefined colors, for `error`, `log`, `warn`, `debug` or `info`, or you can set your own color with the css rgba function, **do not use spaces in this function**!

  The type is optional, but you can choose between one of the following ace-marker types: `text`, (default `fullLine`), `screenLine`

  If you want more than one marker, then simply separate different marker definitions with a colon ...

```
this will be red

this is blue ...
until the next
line

and this is rgba(55,255,100,0.5)
```

Examples

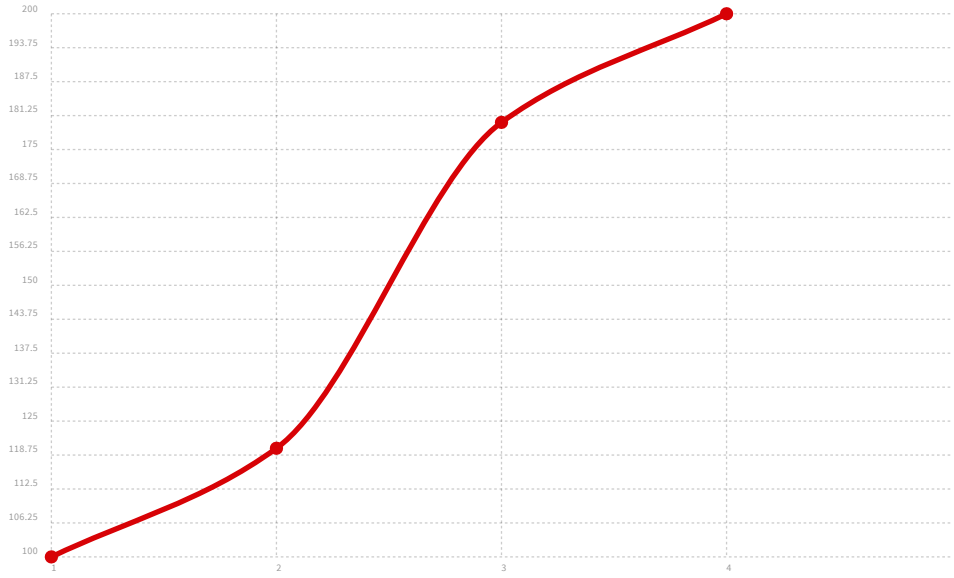
JavaScript Chartist

A drawing example, for demonstrating that any javascript library can be used, also for drawing.

```
// Initialize a Line chart in the container with the ID chart1
new Chartist.Line('#chart1', {
  labels: [1, 2, 3, 4],
  series: [[100, 120, 180, 200]]
});

// Initialize a Line chart in the container with the ID chart2
new Chartist.Bar('#chart2', {
  labels: [1, 2, 3, 4],
  series: [[5, 2, 8, 3]]
});
```

```
[object Object]
```



Computer-Algebra

An example of a Computer-Algebra-System (Algebrit), see http://algebrite.org for more examples:

```
2*x
```

```
f=sin(t)^4-2*cos(t/2)^3*sin(t)

f=circexp(f)

defint(f,t,0,2*pi)
```

```
-16/5+3/4*pi
```

Elm

```
-- Read more about this program in the official Elm guide:
-- https://guide.elm-lang.org/architecture/user_input/buttons.html

import Html exposing (beginnerProgram, div, button, text)
import Html.Events exposing (onClick)


main =
  beginnerProgram { model = 0, view = view, update = update }


view model =
  div []
    [ button [ onClick Decrement ] [ text "-" ]
    , div [] [ text (toString model) ]
    , button [ onClick Increment ] [ text "+" ]
    ]


type Msg = Increment | Decrement


update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1
```

C++

Teaching other language-basics is also possible, for this example we applied JSCPP to run simple C++ programs:

```
#include <iostream>
using namespace std;

int main() {
    int a = 120;
    int rslt = 0;
    for(int i=1; i<a; ++i) {
        rslt += i;
        cout << "rslt: " << rslt << endl;
    }
    cout << "final result = " << rslt << endl;
    return 0;
}
```

```
rslt: 1
rslt: 3
rslt: 6
rslt: 10
rslt: 15
rslt: 21
rslt: 28
rslt: 36
rslt: 45
rslt: 55
rslt: 66
rslt: 78
rslt: 91
rslt: 105
rslt: 120
rslt: 136
rslt: 153
rslt: 171
rslt: 190
rslt: 210
rslt: 231
rslt: 253
rslt: 276
rslt: 300
rslt: 325
rslt: 351
rslt: 378
rslt: 406
rslt: 435
rslt: 465
rslt: 496
rslt: 528
rslt: 561
rslt: 595
rslt: 630
rslt: 666
rslt: 703
rslt: 741
rslt: 780
rslt: 820
rslt: 861
rslt: 903
rslt: 946
rslt: 990
rslt: 1035
rslt: 1081
rslt: 1128
rslt: 1176
rslt: 1225
rslt: 1275
rslt: 1326
rslt: 1378
rslt: 1431
rslt: 1485
rslt: 1540
rslt: 1596
rslt: 1653
rslt: 1711
rslt: 1770
rslt: 1830
rslt: 1891
rslt: 1953
rslt: 2016
rslt: 2080
rslt: 2145
```

```
rslt: 2211
rslt: 2278
rslt: 2346
rslt: 2415
rslt: 2485
rslt: 2556
rslt: 2628
rslt: 2701
rslt: 2775
rslt: 2850
rslt: 2926
rslt: 3003
rslt: 3081
rslt: 3160
rslt: 3240
rslt: 3321
rslt: 3403
rslt: 3486
rslt: 3570
rslt: 3655
rslt: 3741
rslt: 3828
rslt: 3916
rslt: 4005
rslt: 4095
rslt: 4186
rslt: 4278
rslt: 4371
rslt: 4465
rslt: 4560
rslt: 4656
rslt: 4753
rslt: 4851
rslt: 4950
rslt: 5050
rslt: 5151
rslt: 5253
rslt: 5356
rslt: 5460
rslt: 5565
rslt: 5671
rslt: 5778
rslt: 5886
rslt: 5995
rslt: 6105
rslt: 6216
rslt: 6328
rslt: 6441
rslt: 6555
rslt: 6670
rslt: 6786
rslt: 6903
rslt: 7021
rslt: 7140
final result = 7140
```

Prolog

See the implementation details at: https://curiosity-driven.org/prolog-interpreter

Load Database and Rules:

```
exists(A, list(A, _, _, _, _)).
exists(A, list(_, A, _, _, _)).
exists(A, list(_, _, A, _, _)).
exists(A, list(_, _, _, A, _)).
exists(A, list(_, _, _, _, A)).
```

```
database loaded
```

Query: (it may take some time 😉)

```
1  solution(WaterDrinker, ZebraOwner)
```

```
Cannot read properties of undefined (reading 'query')
```

More

We provide a list of templates with more examples that can be used to start developing your own courses. See:

Charts

In many cases, a diagram is only used to present some kind of signal paths, some primitive functions, some clusters or point clouds.

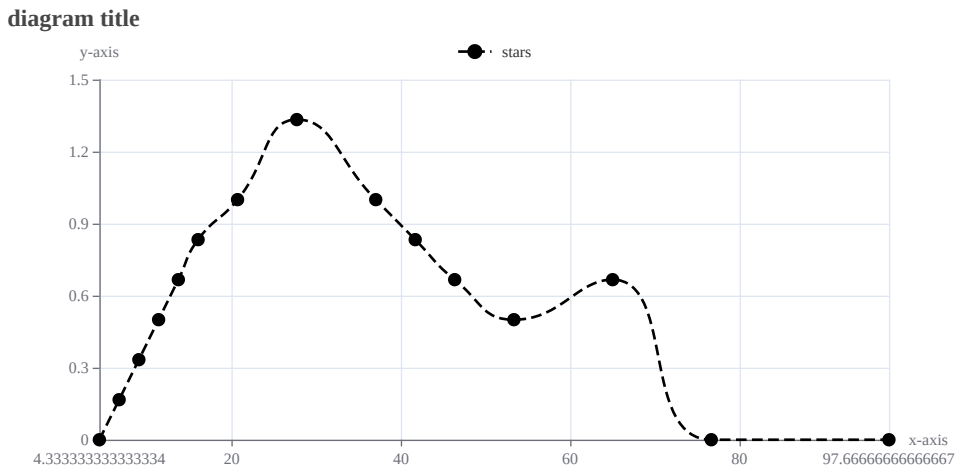
You can still generate images, but why not applying some basic kind of ASCII-art to solve the most common tasks.

Line-Plots 1

Markdown-format:

```
                                diagram title
1.5 |           *
    |                                 (* stars)
  y |         *       *
  - |       *           *
  a |      *              *       *
  x |     *                 *
  i |    *
  s |   *
    |  *                              *       *
  0 +-----------------------------------------
    2.0              x-axis                100
```

Result:

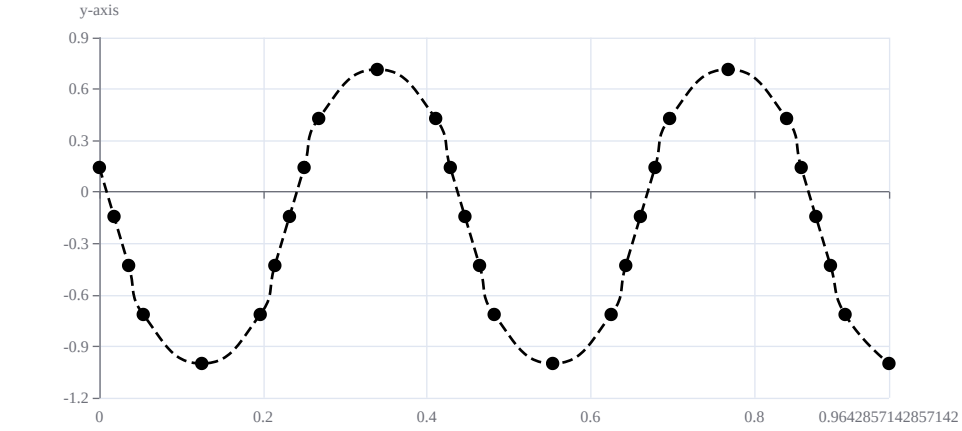


Line-Plots 2

All diagram titles, labels, limits are optional, and if you do not define limits, then the min max values 0 and 1 are used by default.

Markdown-format:

```
    1 |                  *                         *
      |               *       *                 *       *
      |*            *         *               *         *
y-axis | *         *           *             *           *
      |   *       *             *         *             *
      |    *    *                *       *               *
      |      *                      *                     *
   -1 +-------------------------------------------------------
```

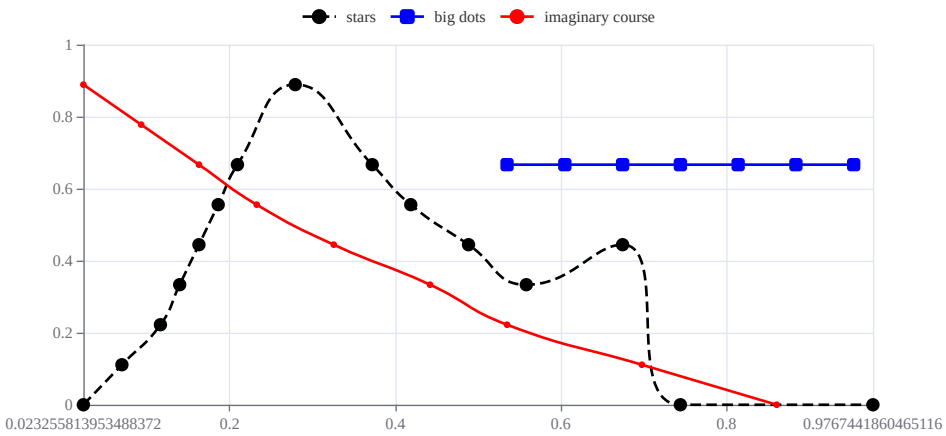


Multi-Line-Plots

Next to stars, you can also use any kind of character to define another line, where the character defines the color. For example an r marks the color red and a b the color blue.

Markdown-format:

```
| r          *                              (* stars)
|    r                                      (r imaginary course)
|       r *      *       B  B  B  B  B  B  B (B big dots)
|        * r       *
|       *      r      *       *
|      *            r    *
|     *                 r
|   *                        r
| *                            *    r    *
+-----------------------------------------
```

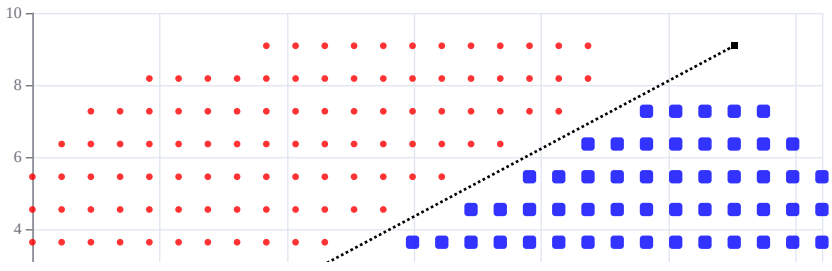


Dot-Plots

If the there are more point with the same character for one x-value, then only dots are plotted. And by using upper and lower case characters you can also define the size of the dots.

Markdown-format:

```
  10 |        rrrrrrrrrrr    x
     |    rrrrrrrrrrrrrrr
     |  rrrrrrrrrrrrrrrr  BBBBB
     | rrrrrrrrrrrrrrr  BBBBBBBB
     |rrrrrrrrrrrrrr  BBBBBBBBBB
     |rrrrrrrrrrr  BBBBBBBBBBBB
     |rrrrrrrrrr  BBBBBBBBBBBBB
     | rrrrrrr  BBBBBBBBBBBBBBB
     |  rrrrr  BBBBBBBBBBBBBBBB
     |        BBBBBBBBBBBBBBBB
     |   x    BBBBBBBBBBBB
     +----------------------------
      0                         2
```



Colors

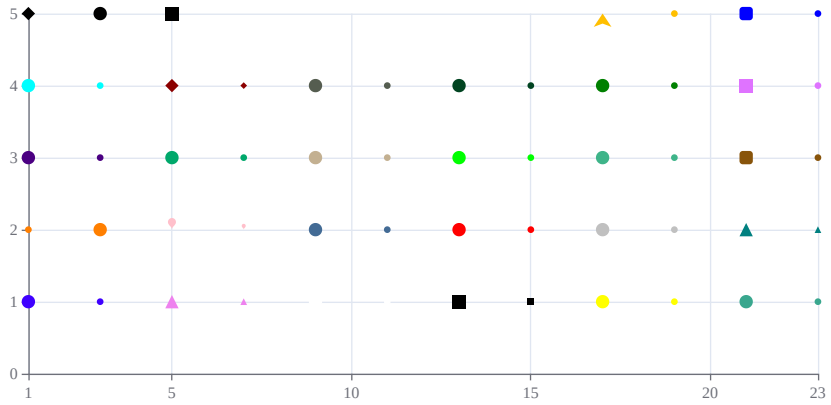
The color codes are somehow defined by the character itself, see the list.

char	color	hex
a	Amber	#FFBF00
b	Blue	#0000FF
c	Cyan	#00FFFF
d	Dark red	#8B0000
e	Ebony	#555D50
f	Forest green	#014421
g	Green	#008000
h	Heliotrope	#DF73FF
i	Indigo	#4B0082
j	Jade	#00A86B
k	Kaki	#C3B091
l	Lime	#00FF00
m	Mint	#3EB489
n	browN	#88540B
o	Orange	#FF7F00
p	Pink	#FFC0CB
q	Queen blue	#436B95
r	Red	#FF0000
s	Silver	#C0C0C0
t	Teal	#008080
u	Ultramarine	#3F00FF
v		#EE82EE
w		#FFFFFF
y		#FFFF00
z	Zomp	#39A78E

Shapes

The shape of the dot is also defined by the character, see the example below.

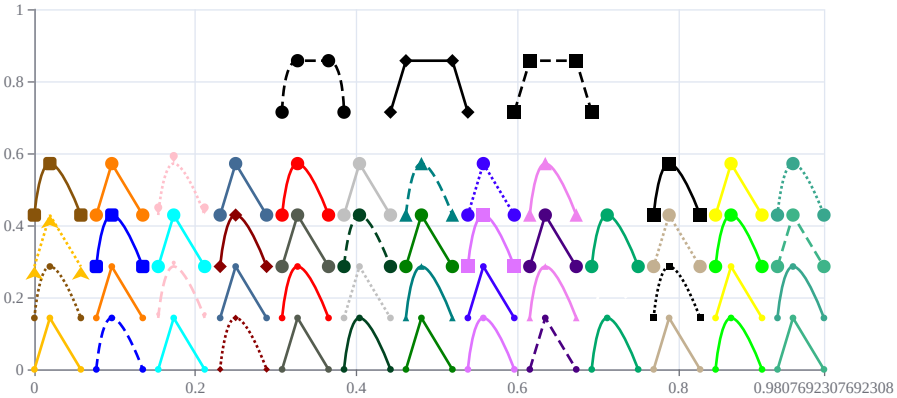
```
6 | + * #           A a B b
  | C c D d E e F f G g H h
  | I i J j K k L l M m N n
  | o O P p Q q R r S s T t
  | U u V v W w X x Y y Z z
1 +------------------------
  0                      24
```



Line types

As depicted in the line diagrams below, next to different colors, lines and dots can have different shapes, whether they are dashed, dotted, smoothed or not.

```
|                 * *    +  +    #  #
|                *   *  +    +  #    #
| N   O   P   Q   R   S   T   U   V   W   X   Y   Z
|NA NOB OPC PQD QRE RSF STG TUH UVI VWJ WXK XYL YZM Z
|An ABo BCp CDq DEr EFs FGt GHu HIv IJw JKx KLy LMz M
|na nob opc pqd qre rsf stg tuh uvi vwj wxk xyl yzm z
|a  ab  bc  cd  de  ef  fg  gh  hi  ij  jk  kl  lm  m
+-----------------------------------------------------
```



Fun with Tables

As already mentioned, tables cannot only be interpreted as structural elements within a Markdown document, but also as datasets. In fact, there is little difference between a diagram and collection of values.

The same values collected within a list. If you click onto the little icon above this list, you will get the same plot as depicted above.

x	dots
0	0
10	2
20	4
30	6

One of the biggest problems in science is the lack of primary data. Projects such as the Open-Science-Framework (http://osf.io) try to leverage this, by offering a plattform, where scientists can store and document vast amounts of data. But why not directly add and visulaize the data, so that they could be used and inspected by others, instead of using external tools to create visualizations.

Everyone who is creating some kind of data or putting it into a Markdown already performs some kind of configuration. Based on that table's/data's structural settings, we can **visualize data automatically**. Actually it is quite surprising, why there has never been an attempt to treat Markdown tables as datasets. The following sections are intended to give a brief overview on different visualization options and how the systems determines, which one gets applied. And of course, you can also enforce your visualization style.

`LinePlot`

The following dataset was taken from https://ourworldindata.org and it shows the government expenditure on education in percentage to the GDP. Thus the first column defines the x-values while the later ones define the categories.

If you click on the image icon again, you will see a more elaborate representation with title and labels.

Year	Finland	USA	Germany	China
1995	6.80942		4.42079	1.84192
1996	6.86052		4.48319	1.85338
1997				
1998			4.45345	1.84432
1999	5.86960			1.88803
2000	5.71687			
2001	5.84797			
2002	6.02477			
2003	6.17476			
2004	6.16849			
2005	6.03605			
2006	5.93809		4.27930	
2007	5.68608		4.34302	
2008	5.84676		4.40954	
2009	6.48517		4.88047	
2010	6.54070	5.42001	4.91368	
2011	6.48200	5.22389	4.80779	
2012	7.19254	5.19485	4.93331	
2013	7.15848	4.94378	4.93496	
2014	7.15155	4.98948	4.93112	

https://ourworldindata.org/financing-education#all-charts-preview

The reason for this is, you can actually add additional settings as it was done to style different Markdown elements, simply by attaching an HTML-comment to the front of this table. The type of representation is still automatically determined based on the table structure, but it is still possible to add attributes like `data-title`, `data-xlabel`, `data-ylabel` to tweak the graphical representation. See section Attributes for more information.

```
<!--
data-title="Government expenditure on education"
data-xlabel="year"
data-ylabel="% of GDP"
-->
| Year | Finland | USA | Germany |   China |
| ---- | -------:| ---:| -------:| -------:|
| 1995 | 6.80942 |     | 4.42079 | 1.84192 |
| 1996 | 6.86052 |     | 4.48319 | 1.85338 |
| ...  |     ... | ... |     ... |     ... |
```

You can of course also visualize any kind of table, that does not fullfill this type of classification to line or scatterplot. Define another kind of presentation and if not all values within the first column can be parsed as numbers, then they are interpreted as categories. If you change the order of the table, then also the order of categories in the visualization is changed.

```
<!-- data-type="line" -->
| Animal          | weight in kg | Lifespan years | Mitogen |
| --------------- | ------------:| --------------:| -------:|
| Mouse           |        0.028 |              2 |      95 |
```

Animal	weight in kg	Lifespan years	Mitogen
Mouse	0.028	2	95
Flying squirrel	0.085	15	50
Brown bat	0.020	30	10
Sheep	90	12	95
Human	68	70	10

ScatterPlot

If your table is similar to the one in a LinePlot, but the first column contains numbers which appear twice or more times, than this data cannot be interpreted as a "function" in a mathematical sense. This data is then simply visualized as a scatter plot only showing the dots.

Random	I	II
5.0	1.0	5
6.0	1.0	4
7.0	1.0	5
8.0	1.0	5
9.0	1.0	4
10.0	1.0	5
5.0	10.0	7
6.0	10.0	8
7.0	10.0	7
8.0	10.0	7
9.0	10.0	8
10.0	10.0	7

BoxPlot

If you have a ScatterPlot like representation, but actually want to use this data as primary data for your BoxPlot, you can manually change the type of visualization to BoxPlot, simply by adding the following attribute to the head of your table, as it is shown in the snippet below. Columns are then treated as datasets and get visualized accordingly.

```
<!-- data-type="boxplot" -->
| Random |    I |  II |
| ------:| ----:| ---:|
|    5.0 |  1.0 |   5 |
|    ... |  ... |  .. |
```

Random	I	II
5.0	1.0	5
6.0	1.0	4
7.0	1.0	5
8.0	1.0	5
9.0	1.0	4
10.0	1.0	5
5.0	10.0	7
6.0	10.0	8
7.0	10.0	7
8.0	10.0	7
9.0	10.0	8
10.0	10.0	7
		1

BarChart

In contrast to a line or a scatter plot, if the first colum contains at least one entry thant cannot be parsed as a number, this might be represented also as BarChart. Which works perfectly with the following example. If the maximum values of the columns do not differ to much, then this dataset it represented as a BarChart, otherwise you might end up seeing only one huge bar, while the other bars are indistiguishable from each other. In this case other visualization are chosen.

Animal	weight in kg	Lifespan years	Mitogen
Mouse	0.028	2	95
Flying squirrel	0.085	15	50
Brown bat	0.020	30	10
Sheep	90	12	95
Human	68	70	10

Radar

If for example humans and sheeps are removed from the dataset, then wheight in kg would not be visible in a BarChart at all. In this case a Radar is selected, that allows to analyze data visually with different "y"-axis.

Animal	weight in kg	Lifespan years	Mitogen
Mouse	0.028	02	95
Flying squirrel	0.085	15	50
Brown bat	0.020	30	10

`PieChart`

If you have a table with only one row full of numbers, this will be automatically presented as an pie chart. The head represents the categories and the body the quatities.

Classic	Country	Reggae	Hip-Hop	Hard-Rock	Samba
50	50	100	200	350	250

You can use the first column to give some more information about your data. If the first element of the list body contains a text, that cannot be directly interpreted as a number, then these two text snippets are used to the define the main title and the subtitle of your chart.

Music-Style **1994**	**Classic**	**Country**	**Reggae**	**Hip-Hop**	**Hard-Rock**	**Samba**
Student rating	50	50	100	200	350	250

`PieChart`(s)

The default behavior for the Table below, would be to represent it as a bar-chart. But, you can enforce the usage of pie charts, simply by adding the attribute `piechart` into the HTML comment, directly above the table:

```
<!-- data-type="PieChart" -->
| Music-Style | Classic | Country | Reggae | Hip-Hop | Hard-Rock | Samba |
|:----------- | -------:| -------:| ------:| -------:| ---------:| -----:|
| 1994        |      50 |      50 |    100 |     200 |       350 |   250 |
| ...         |     ... |     ... |    ... |     ... |       ... |   ... |
```

The result looks as follows:

Music-Style	**Classic**	**Country**	**Reggae**	**Hip-Hop**	**Hard-Rock**	**Samba**
1994	50	50	100	200	350	250
2014	20	30	100	220	400	230
demo 2034	5	12	98	293	345	32

Since data is parsed at runtime, you can also use animations to change the values of chart, while go on in your slide or move back. But keep in mind, that this might lead to negative effects, if your audience preferes the textbook mode:

Music-Style **1994 2014**	**Classic**	**Country**	**Reggae**	**Hip-Hop**	**Hard-Rock**	**Samba**
Student rating	50 20	50 30	100	200 220	350 400	250 230

If the upper table might be too long and you prefer to use only two columns and grow your data vertically, then you can use the attribute `data-transpose`, which flips mirrows your data along an imaginary vertical axis.

```
<!-- data-transpose -->
| Music-Style {0-1}{1994} {1}{2014} |    Student rating |
|:--------------------------------- | -----------------:|
| Classic                           | {0-1}{50} {1}{20} |
| Country                           | {0-1}{50} {1}{30} |
| ...                               |               ... |
```

The result is the same as above, but it might be easier to handle your data.

Music-Style 1994 2014	Student rating
Classic	50 20
Country	50 30
Reggae	100
Hip-Hop	200 220
Hard-Rock	350 400
Samba	250 230

Funnel

Funnel is a similar representation as PieChart, but it is not set automatically. If you want to use funnel, you will have to set the `data-type` parameter to funnel.

```
<!-- data-type="funnel" -->
| Classic | Country | Reggae | Hip-Hop | Hard-Rock | Samba |
| -------:| -------:| ------:| -------:| ---------:| -----:|
|      50 |      50 |    100 |     200 |       350 |   250 |
```

Classic	Country	Reggae	Hip-Hop	Hard-Rock	Samba
50	50	100	200	350	250

The rest is the same as for piecharts, you can also use effects to generate animated diagrams.

Music-Style 1994 2014	Student rating
Classic	50 20
Country	50 30
Reggae	100
Hip-Hop	200 220
Hard-Rock	350 400
Samba	250 230

Map

A map is similar to a BarChart from the table structure, but if you want to depict your data on a real map, you will have to add a geojson-file, that contains all relevant data about the form of your countries, states, cities, etc. The first column has to match the names of your objects in your geojson data, that is attached to your table in the following way:

```
<!-- data-type="map" data-src="https://code.highcharts.com/mapdata/custom/europe.
geo.json" -->
| Country               | percent |
| --------------------- | ------- |
| Albania               | 73.5    |
| Andorra               | 98.9    |
```

Country	percent
Albania	73.5
Andorra	98.9
Armenia	72.4
Austria	87.9
Azerbaijan	79.8
Belarus	79.7
Belgium	93.9
Bosnia and Herzegovina	80.8
Bulgaria	66.7
Croatia	91.5
Cyprus	84.4
Czech Republic	87.7
Denmark	97.8
Estonia	97.9
Finland	94.0
France	92.3
Georgia	68.1
Germany	96.0
Greece	72.9
Hungary	89.0
Iceland	99.0
Ireland	91.9
Italy	92.5
Latvia	87.1
Liechtenstein	98.1
Lithuania	90.9
Luxembourg	97.8
Macedonia	79.2
Malta	83.1
Moldova	76.1
Monaco	97.5
Montenegro	71.5

Netherlands	95.6
Norway	98.4
Poland	78.2
Portugal	78.2
Republic of Serbia	73.4
Romania	73.8
Russia	80.9
San Marino	60.2
Slovakia	84.9
Slovenia	79.9
Spain	92.5
Sweden	96.4
Switzerland	93.7
Turkey	83.3
Ukraine	93.4
United Kingdom	94.9
Vatican City	60.1

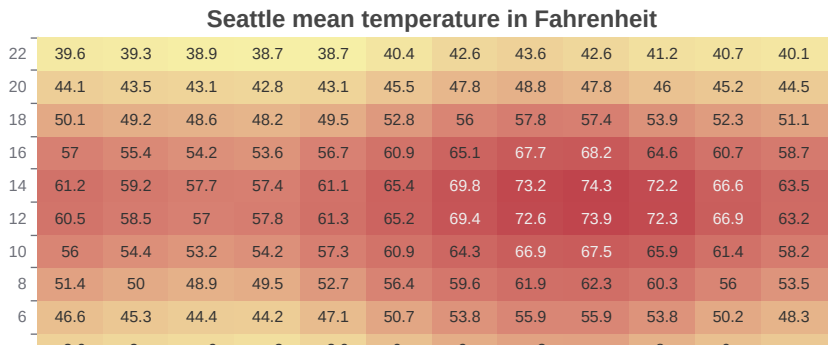
Currently there is only support to visualize one column, but this will be fixed in the future ...

HeatMap

Another type of visualization is a HeatMap, which is used, if the table head and the first column do only contain numbers, in other words coordinates. If you want to use categories instead of coordinate numbers, you can enforce the usage of a heatmap, with the comment shown below:

```
<!--
data-type="heatmap"
data-title="Seattle mean temperature in Fahrenheit"
data-show
-->
| Seattle |  Jan |  Feb |  Mar |  Apr |  May |  ... |
| -------:| ----:| ----:| ----:| ----:| ----:| ----:|
|       0 | 40.7 | 41.5 | 43.6 | 46.6 | 51.4 |  ... |
|       2 |  ... |  ... |  ... |  ... |  ... |  ... |
```

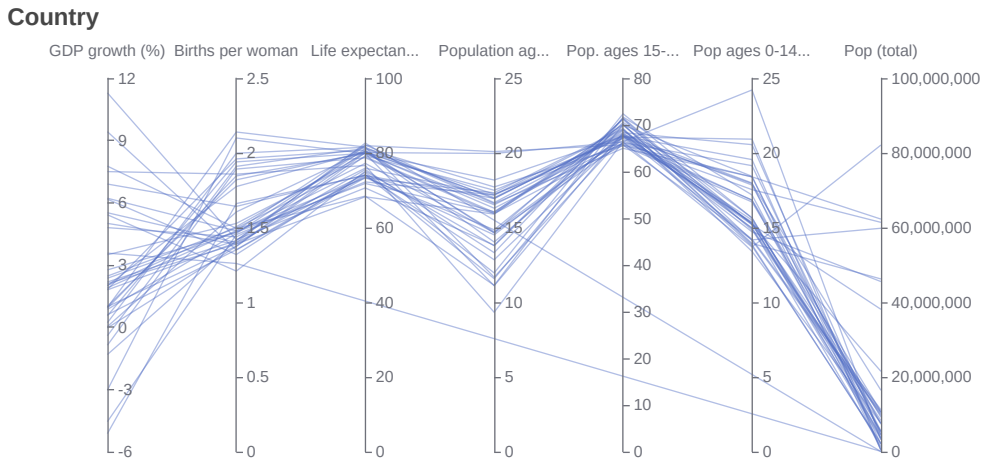
The attribute `data-show` simply shows the diagram at default, instead of using the table.



https://datavizpyr.com/heatmaps-with-seaborn-in-python/

Parallel

A Parallel representation jumps in, if there are simply too many categories, so that your BarChart would contain only thin lines.



Graph

If the first column and the head of the table are equal, then the interpreter tries to interpret the content of the table as an adjacency matrix, which defines a graph. If those values are symetrical according to the diagonal, then the matrix defines an **undirected graph**.

Graph	A	B	C	D	E
A	0	1	0	1	0
B	1	0	0	1	0
C	0	0	0	0	0
D	1	1	0	0	1
E	0	0	0	1	0

In contrast to this, if those values differ, then the result is simply an an **directed graph**, whereby the values define the strength of the line.

Graph	A	B	C	D	E
A	0	12	0	1	0
B	-22	0	0	0.4	0
C	0	0	0	0	0
D	2	12	0	0	1
E	0	0	0	2	0

Unfortunatelly, self referenceing or multigraphs are currently not supported.

`Sankey`

A Sankey diagram is a special type of directed graph that can be used to streams or the flow of something, such as engergy, money, etc.

https://en.wikipedia.org/wiki/Sankey_diagram

Sankey	A	B	C	D	E
A		2			
B	3				
C	1	1			
D		1	1		
E	2	1	1	1	

`None`

Simply `data-type="none"` to prevent any kind of visualization.

Sankey	A	B	C	D	E
A		2			
B	3				
C	1	1			
D		1	1		
E	2	1	1	1	

Attributes

- `data-type`: You can use `data-type="map|boxplot|barchart|..."` to overwrite the automatically identfied representation with your desired one. The names can be taken from the previous titles, it is not relevant if you use lower or upper-case. This way it is also possible to use types that cannot be automatically infered at the moment, such as Sankey or BoxPlot.

  If you do not want to show tables as diagrams, you can also use `data-type="None"` and only the table will be visible.
- `data-show`: Simply add this attribute or set it to true (`data-show="true"`), if you want to visualize your data immediately, without the need to click in the switch-button. It is still possible for your users to switch to the table representation.
- `data-transpose`: Like in the mathematical sense, set this attribute or set it to true (`data-transpose="true"`), if you want to switch rows and columns. One benefit is, that you can for example use PieChart and let your table grow vertically instead of using a horizontal monster.
- `data-title`: Normaly, the first cell defines the title of your diagram, but if you want larger titles and not have to write gigantic table headers, apply this attribute `data-title="Use whatever title you want to ..."`
- `data-xlabel`: As above, you can also define the strings for the labels, in this case for the x label
- `data-ylabel`: or the y label.
- `data-src`: Currently this attribute is used to refere to your geojson data, if you use the `data-type="Map"` representation, but this might change in the future to load and visualize data directly, such as csv.

  If you are using geojson files from external websites such as:

  https://code.highcharts.com/mapdata/

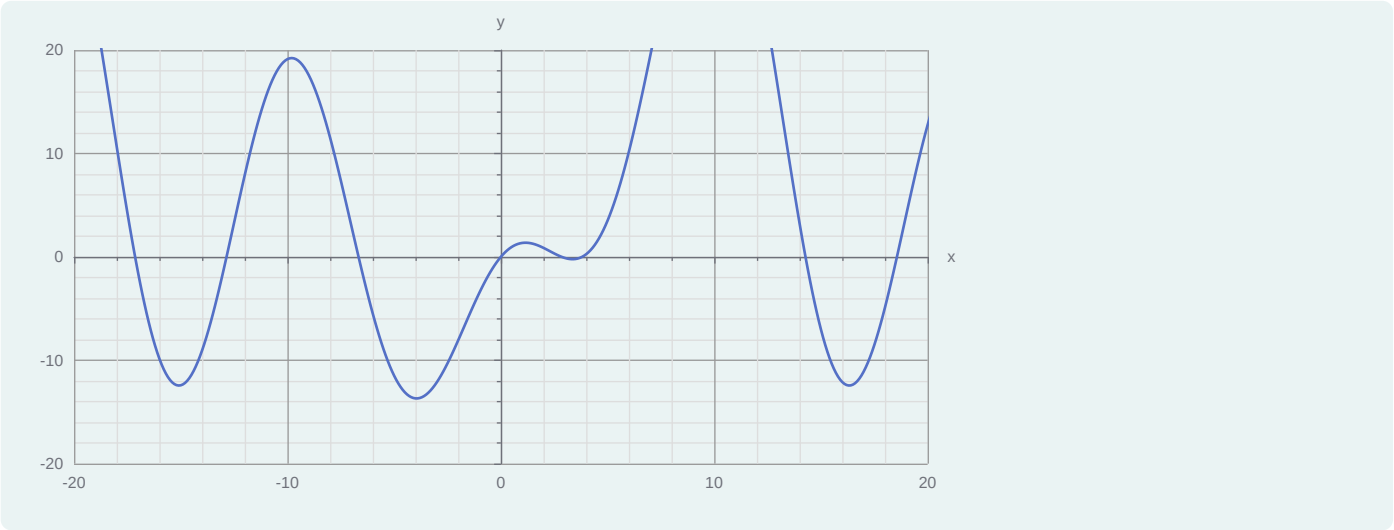
  It can be usefull to use anycors, if the data cannot be visualized due to CORS restrictions:

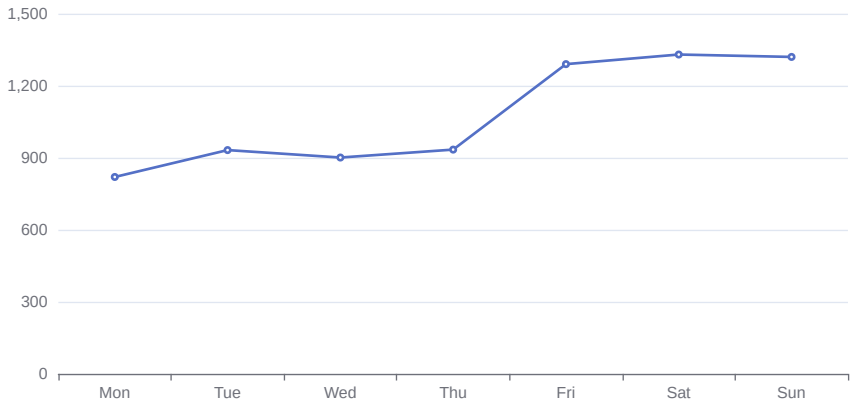
  `data-src="https://cors-anywhere.herokuapp.com/https://code.highcharts.com/mapdata/custom/europe.geo.json"`

custom

2

50





ASCII-Art #2

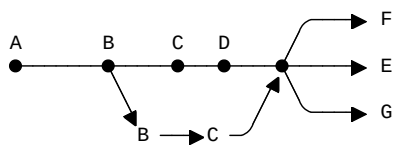
Well, thanks to the great project SvgBob the newest version of LiaScript also has support for some basic ASCII art drawings (not everything is supported yet). Simply use 4 or more backtics to enclose your artwork and draw whatever you want. And as with any other element, you can add some styling within HTML comments at the head of this element.

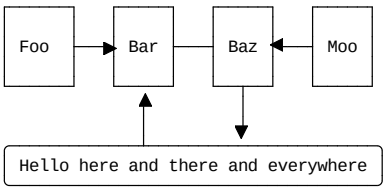
The following examples are taken from the examples on the SvgBob project site.

If you want to use a drawing tool for this, visit the online editor at:

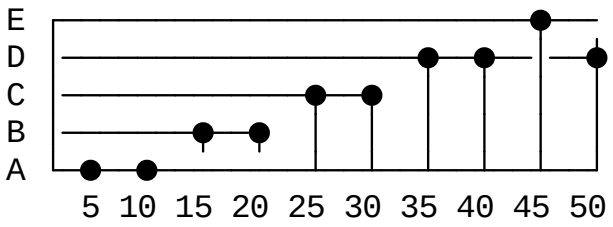
https://ivanceras.github.io/svgbob-editor/

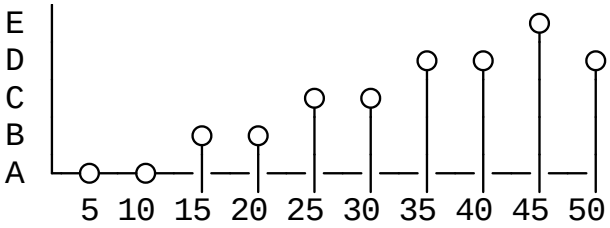
Graphs

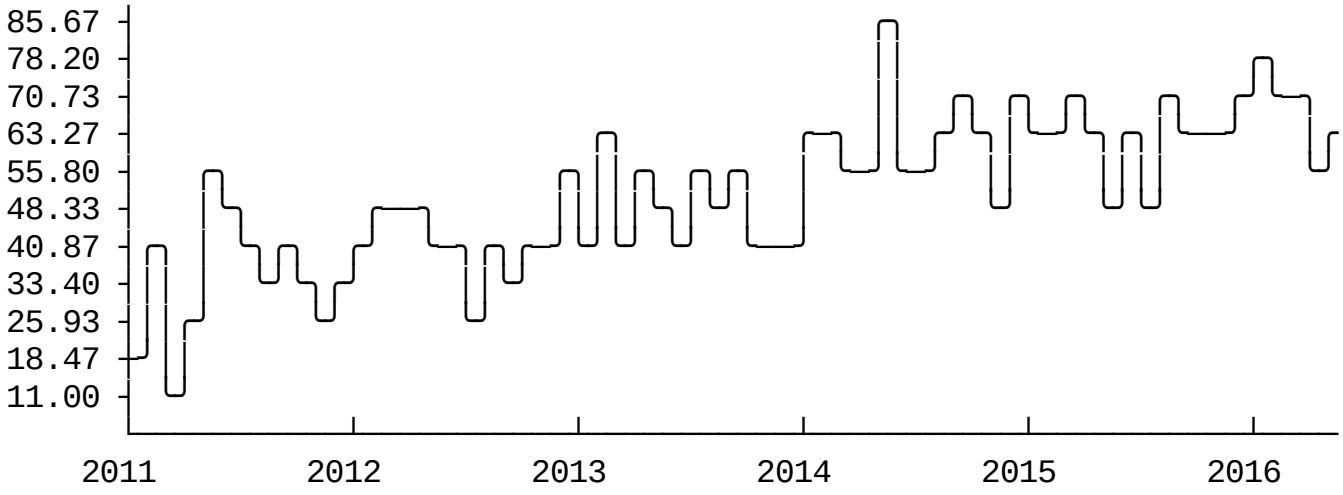




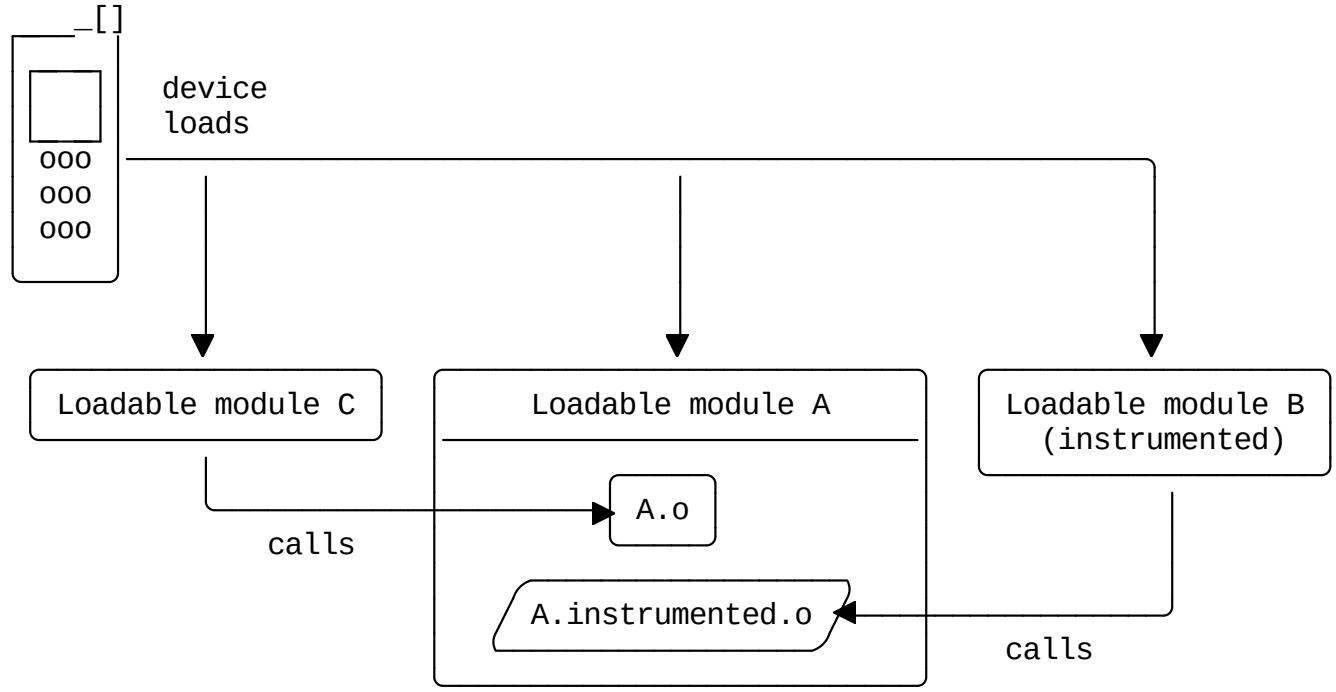
Diagrams

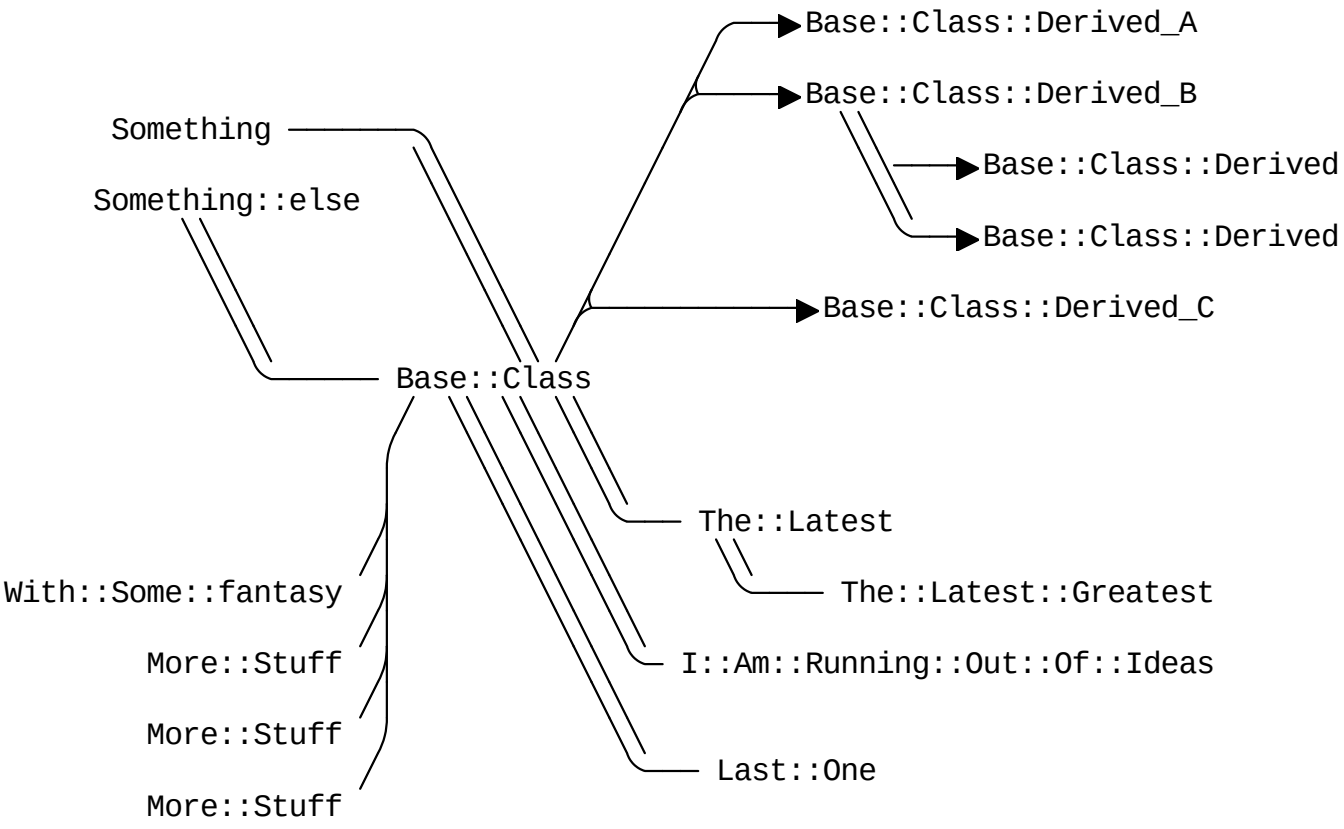




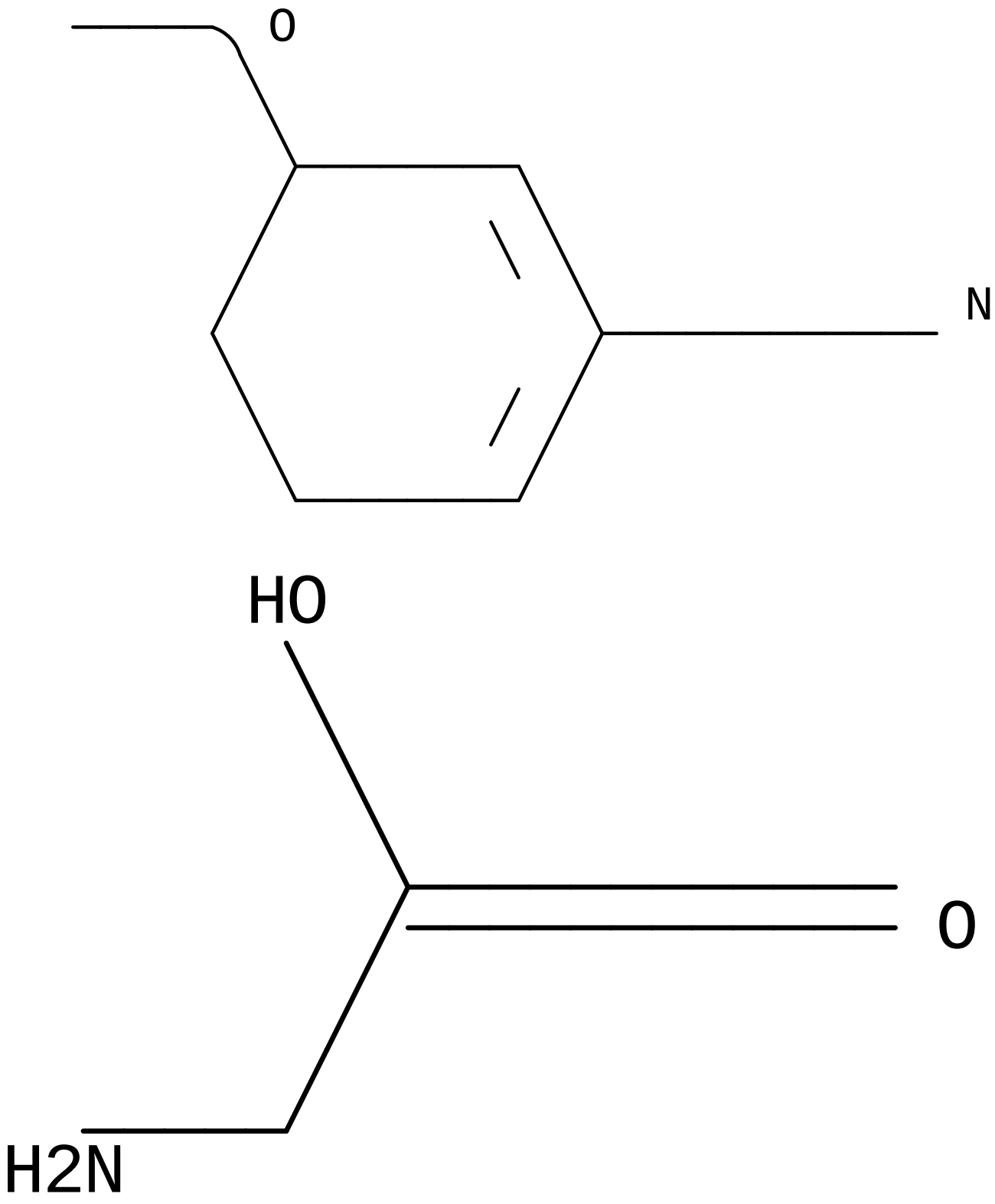


UML

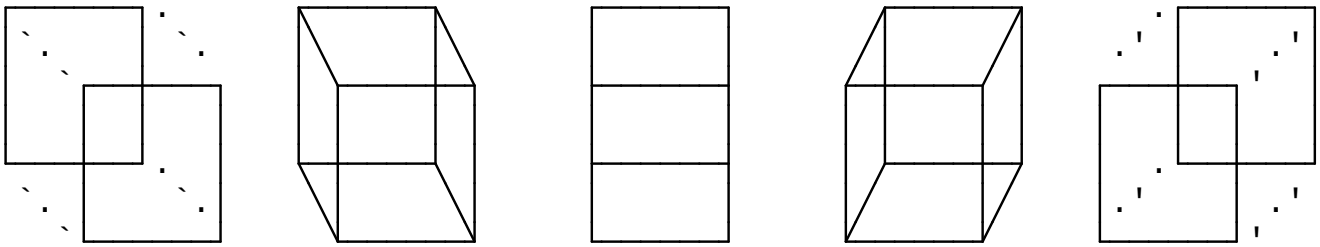


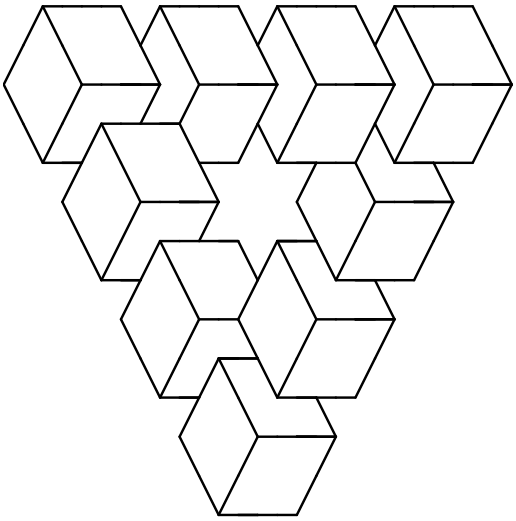


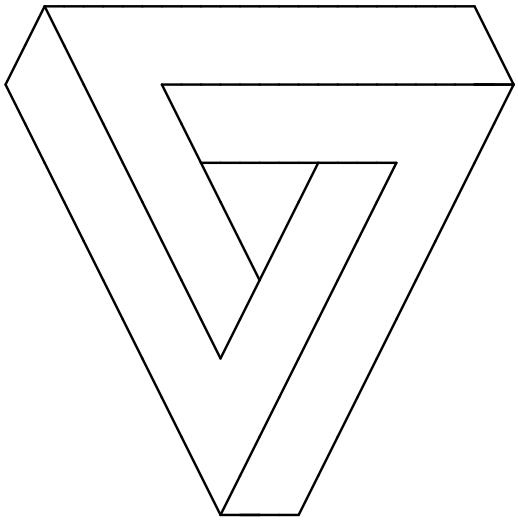
Chemical Structures

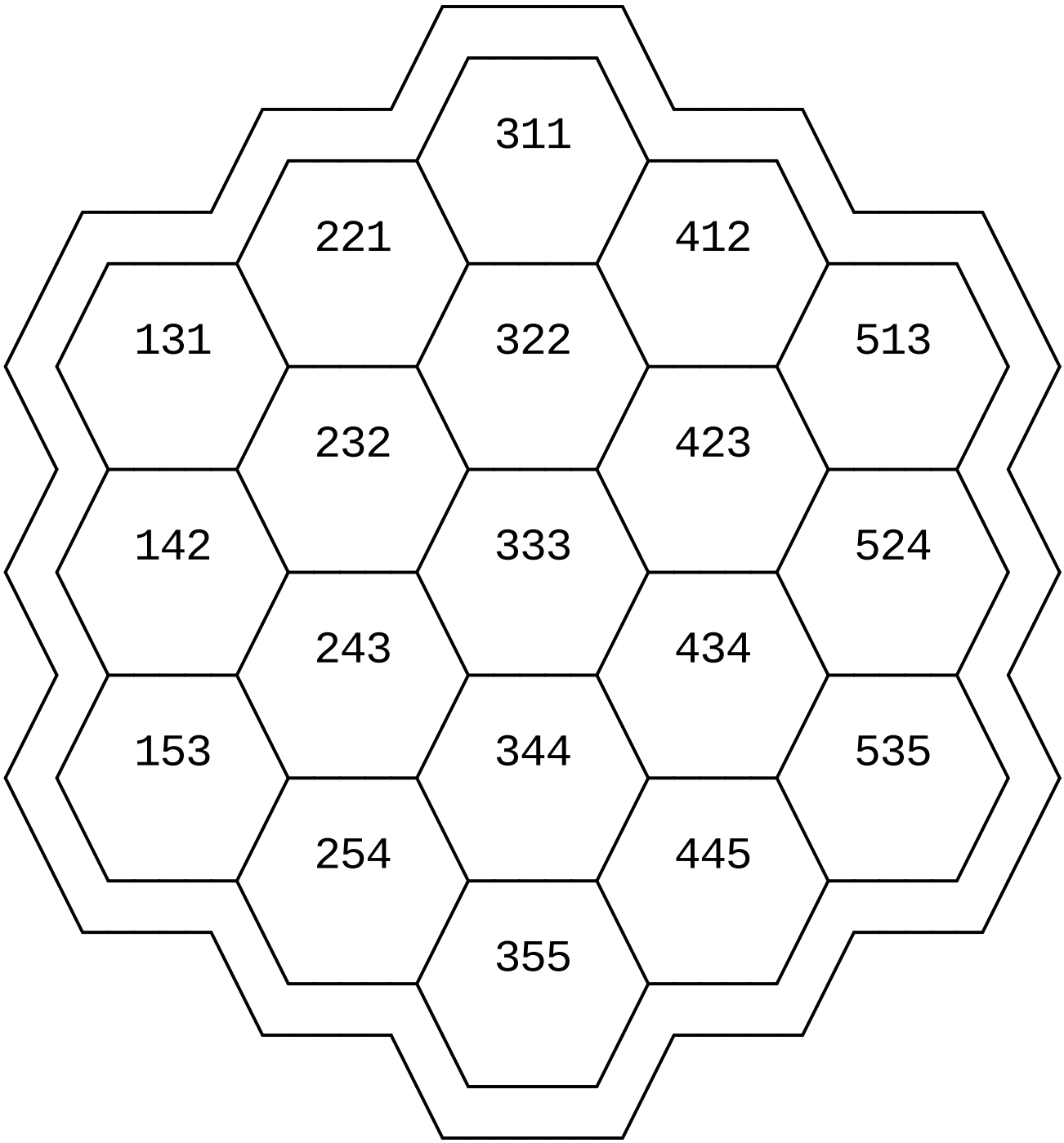


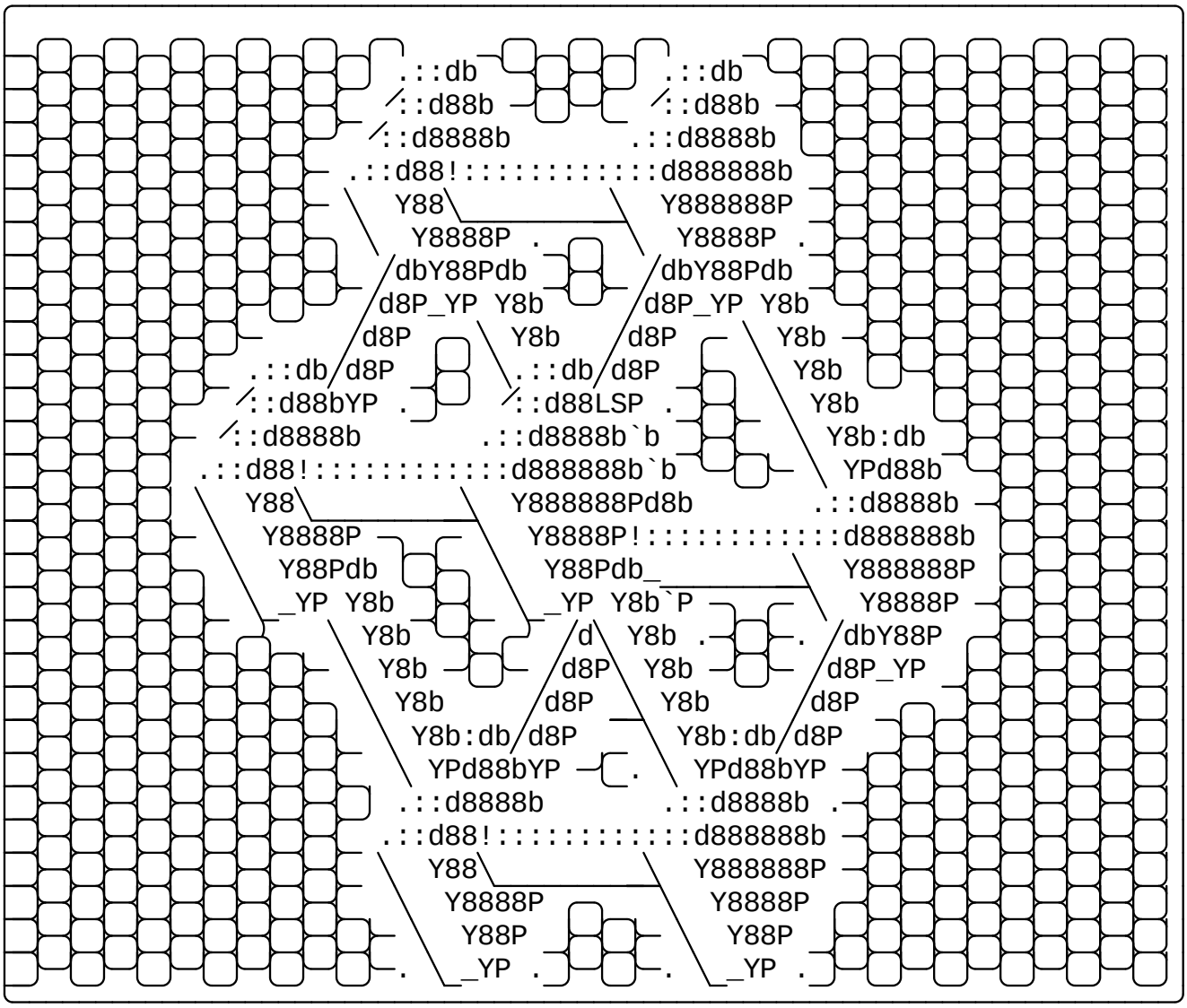
Geometrical Shapes



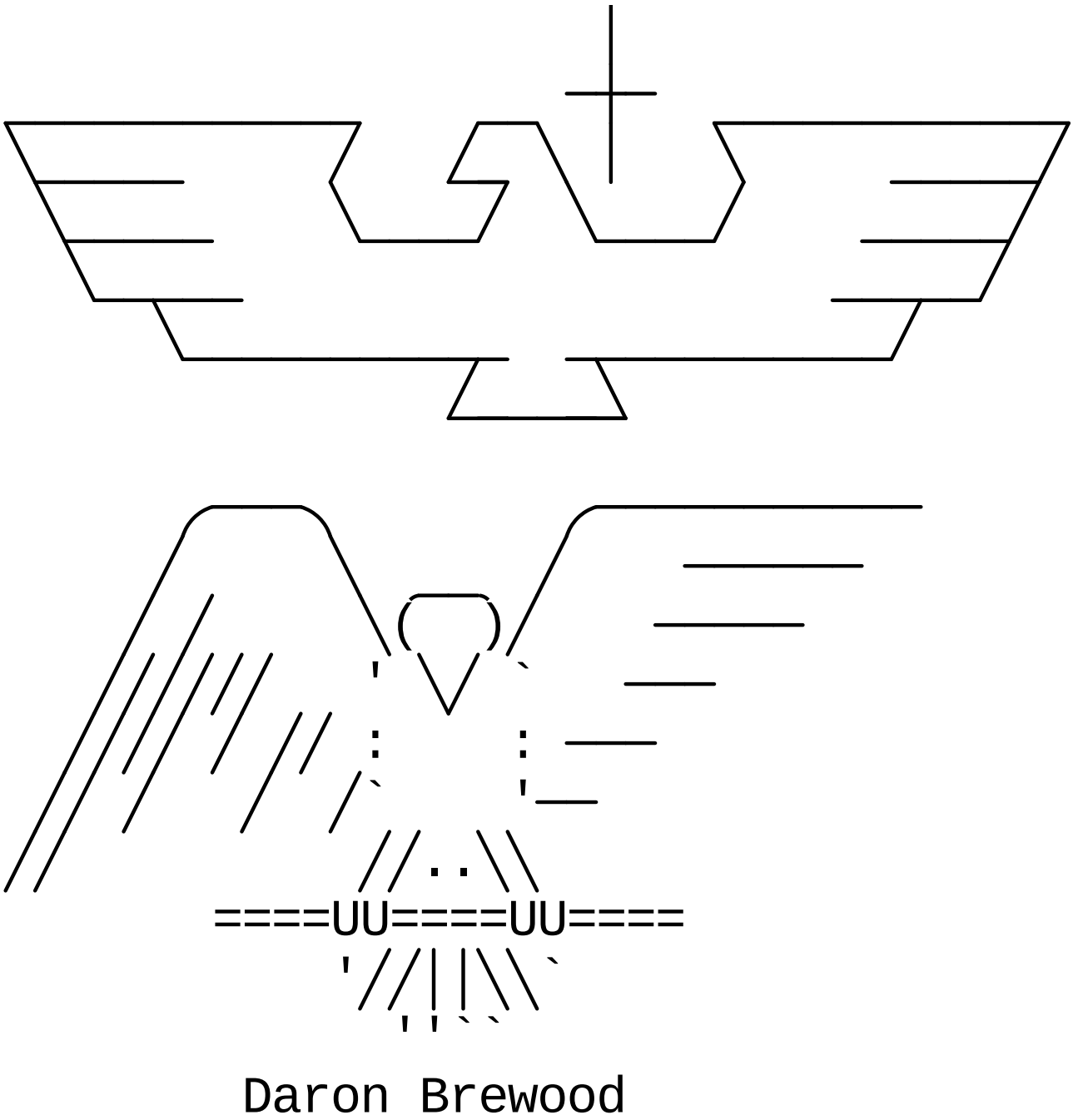





311
221
412
131
322
513
232
423
142
333
524
243
434
153
344
535
254
445
355

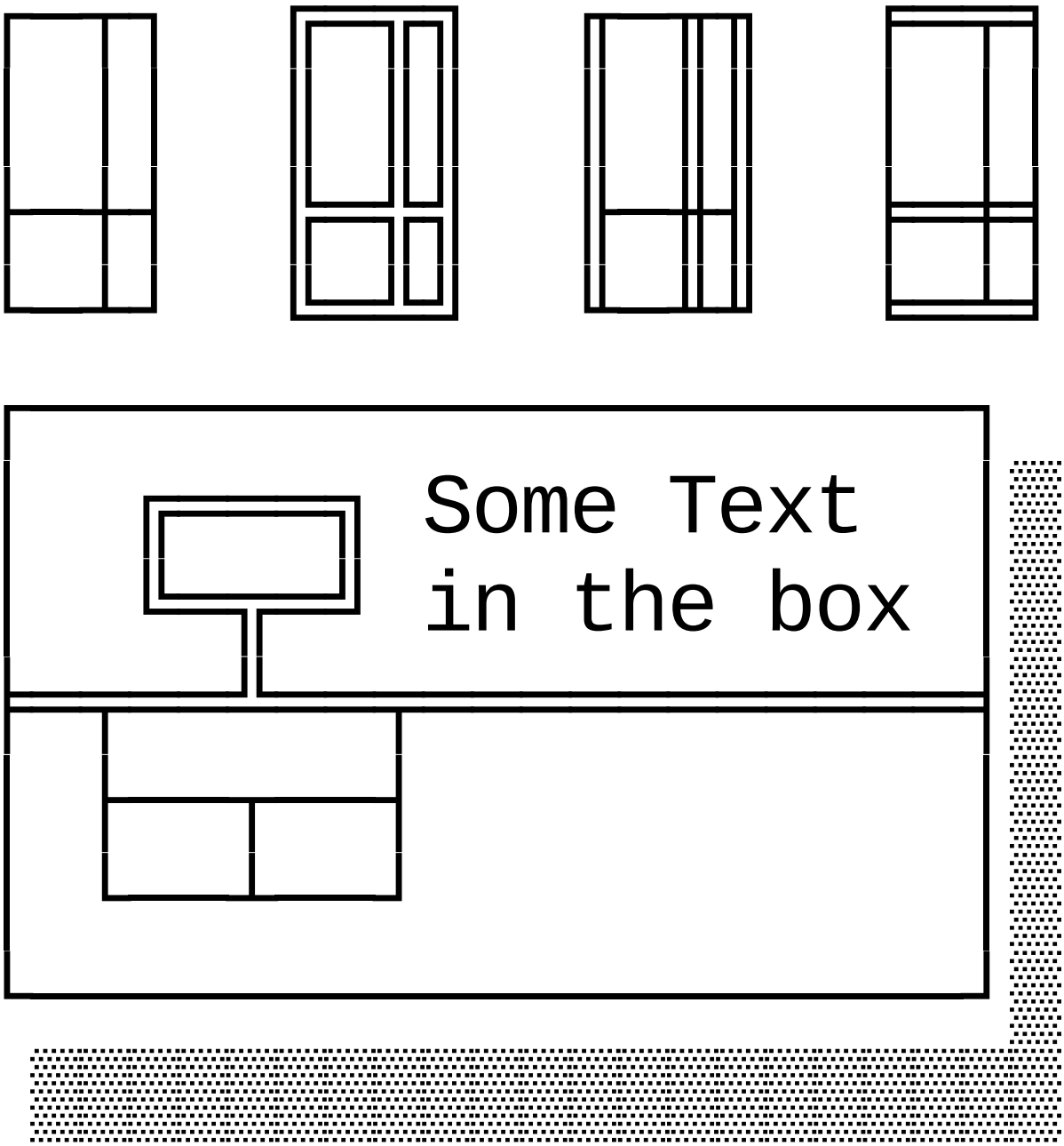


Fun



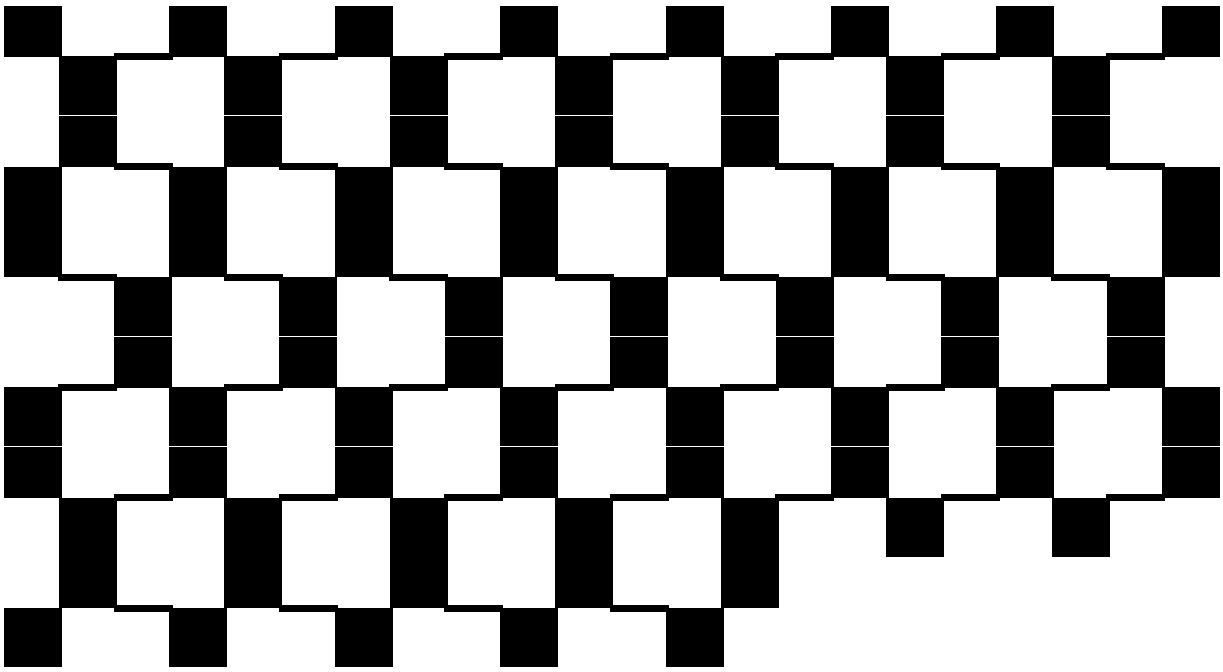
Unicode

And of course, if your ASCII table does not give you enough pleasure, you can also use any kind of Unicode symbol (also within the text).

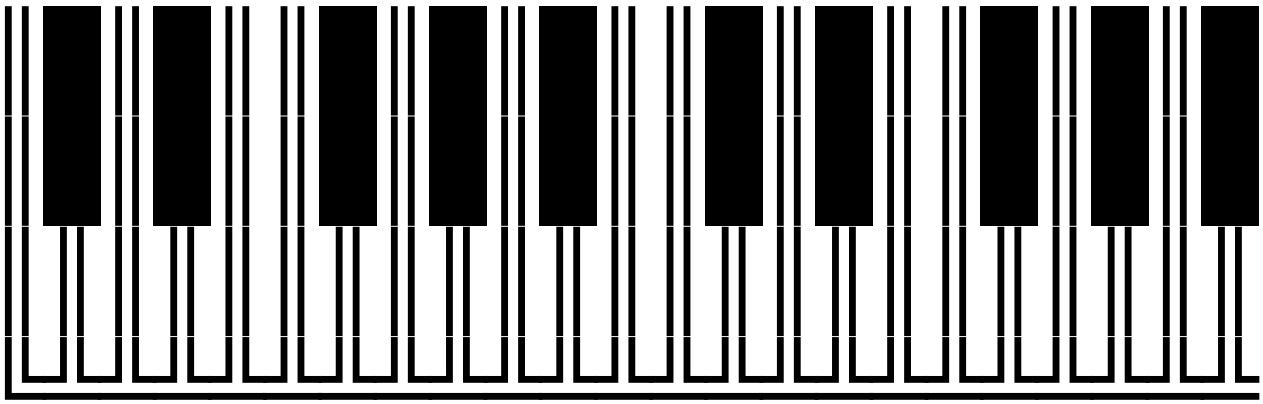


Unicode Art

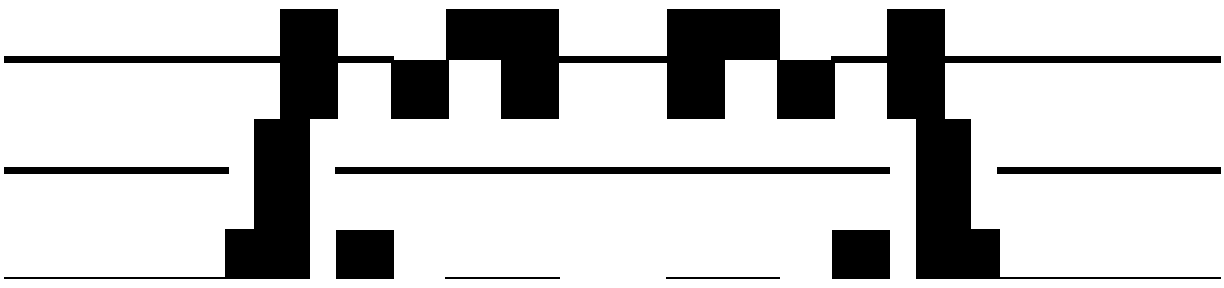
4 ◇ Sep @tw1tt3rart



23 ◇ Apr @tw1tt3rart



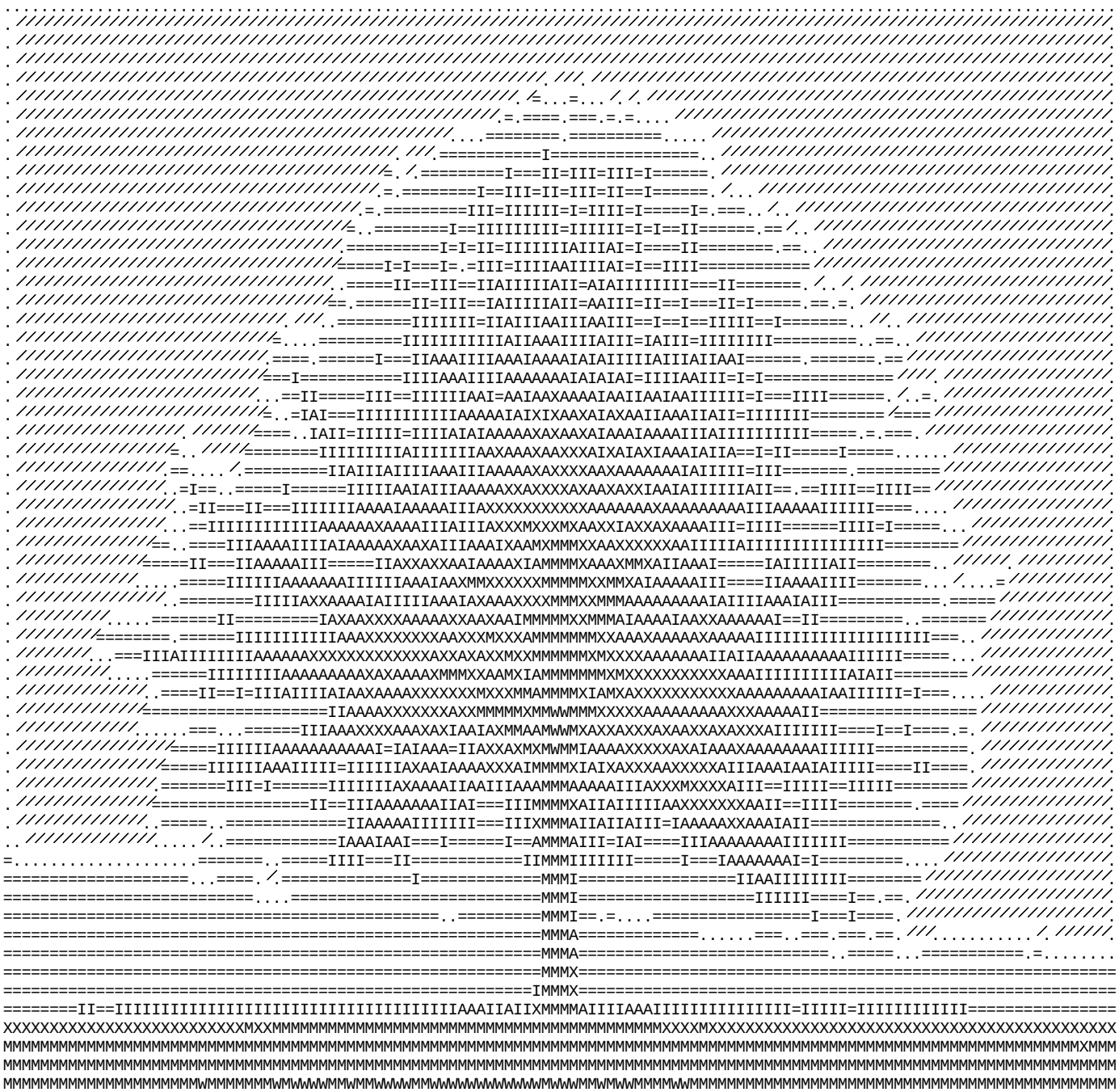
19 ◇ Jan @tw1tt3rart



Unicode Converter

Image generated with:

http://www.underware.nl/fonts/zeitung_mono/features/ASCII+/#feature_info



Macros

Macros are an easy way in LiaScript to hide reoccurring or tedious tasks. In essence, they define a simple textsubstitution-system that can be parameterized and also imported from other courses. Actually, everything that starts with an `@`-symbol can be replaced by something that has been defined in the main-header or sub-header of your document.

By now, you will have seen or changed macros like `author`, `logo`, or `comment`, but you can do even more.

```
<!--
author: someone who wants to create something new

email: your contact-information
-->
```

Single Line

There are actually two types of macros, single line macros and block macros. While the first one only returns a single line, the latter one preserves your custom indentation. This single line macros start with a macro-name, which is followed by a colon. The name has to start with a word-character. The starting `@`-symbol is optional, but I would suggest to use it for custom macros and to increase readability.

```
<!--
author: someone who wants to create something new

@Single.line: you can add as much content as you
  want to your single-line macro!

  The only thing that is important, is to use
    indentation.

    Not __matter__ how [much](#12) it is.

-->

# Main Title

@Single.line  <-- this will be replaced at compile-time by: v

you can add as much content as you want to your single-line macro! The only
thing that is important, is to use indentation. Not __matter__ how [much](#12)
it is.

@author <-- by: someone who wants to create something new
```

By the way, macros are case-sensitive, thus there is a difference between `@author` and `@Author`.

Blocks

Block-macros are not followed by an colon and are parsed until they reach the `@end` symbol. Every macro can also be used to define Markdown content, HTML, CSS, or even JavaScript.

```
<!--
@smile: ;-)

@block
this type of macro preserves the structure.

<h1>
@smile and you can also use macros, that
define other macros
</h2>

| Header 1   | Header 2   | Header 3   |
| :--------- | :--------- | :--------- |
| Item 1     | Item 2     | Item 3     |

<script>alert("hello world")</script>
@end
-->

# Main Title

@block
```

Comments

Commenting macros out or simply adding additional "personal" information to to a section happens with two additional `@@` for single-line, and three `@@@`s to comment blocks...

```
<!--
@@comment: this is a single line macro
  that has been commented out
```

Overwriting Macros

Actually every section can define its own set of macros, that only exist in that in that single section. Only the initial main-header at the start of a document defines a set of global macros. This might be useful, if you want to switch the voice of the narrator, authorship, etc.

```
<!--
...
narrator: US English Female

-->

# Main Title

....

### Some other section
<!--
author: another author for this section
...
narrator: Australian Male

-->
```

Passing Parameters

It is also possible to pass parameters to macros, which looks like calling a function in most common programming languages. Simply call a macro with parentheses and put into everything, that you want to place within the macro, where something is placed is simply defined by an `@ + a number`, thus `@0`, `@1`, .., `@n`. Like most programming languages start counting at `0`, also in LiaScript `@0` defines the first parameter.

```
<!--
@highlight: <b style="color: red">@0</b>
-->

@highlight(I want this text to be read and bold)
```

The following example shows, how multiple parameters can be passed to a macro, they are simply separated by commas. The main problem which appears is that, if you want to pass a string that contains commas, then this string has to be placed in backticks. As also depicted, a macro can also call/substitute another macro, which might be extremely useful if you have defined a very complex macro. You can define more "simple" macros, that call the "complex" one by setting some some default values, see therefor section uid.

```
<!--
@highlight: <b style="color: red">@0</b>

@red_and_green:
  @highlight(@0) <i style="color: green">@1</i>
-->

@red_and_green(red,`simply, simply, green`)
```

Last but not least, in order to go on with the basic Markdown syntax, it is also possible to pass multi-line parameters. How would you do this? Of course simply by using a common Markdown code-block with three backtics.

```
@red_and_green(red,```please to not use
this in production...
```

As mentioned in the code, try to avoid this, since it generates ugly Markdown code on GitHub or on any other Markdown-interpreter/editor. But you can also use a code-block, which contains information about syntax highlighting and a macro, that defines the title. The part within the code-block is then simply passed to the macro as the last and multi-line parameter.

```
<!--
link:   https://pannellum.org/css/style.css
        https://cdn.pannellum.org/2.4/pannellum.css

script: https://cdn.pannellum.org/2.4/pannellum.js

@panorama
<div id="panorama_@0" style="width: 100%; height: 400px;"></div>
<script>
  pannellum.viewer('panorama_@0', {
    "type": "equirectangular",
    "panorama": "@1",
    "autoLoad": false,
    "hotSpots": [@2]
});
</script>
@end
-->

```json @panorama("0",https://pannellum.org/images/cerro-toco-0.jpg)
{
  "pitch": 14.1,
  "yaw": 1.5,
  "type": "info",
  "text": "Baltimore Museum of Art",
  "URL": "https://artbma.org/"
},
{
  "pitch": -0.9,
  "yaw": 144.4,
  "type": "info",
  "text": "North Charles Street"
}
```
```

Escaping

In some cases, for example if you want to pass content to a javascript string and you need to escape the content of the LiaScript content, wich could be a multiline string for example. Then you can simply add a `'` to your macro, for example:

- `@'input` will result in an escaped version of the input string
- `@'input(1)` is the same as above
- `@'1` as a parameter will also get escaped
- similarly to any other `@'macro(with, some, params)`

> But be carefull to apply it only once in your macro chain as multiple escape sequences might result in multiple backslashes (`\\\"`) for special characters.

Debugging

Creating macros, can be quite difficult, especially if it is about creating and calling macros nested macros with various parameters. And defining macros in Atom is somehow similar to navigating in the dark, since it is not possible to inspect the DOM. But you can escape a macro by and additional `@`, which outputs a gray and escaped HTML `pre` `code` block.

```
<!--
@highlight: <b style="color: red">@0</b>

@red_and_green:
  @highlight(@0) <i style="color: green">@1</i>
-->
```

```
```

```
red simply, simply, green
```

Special Macros

The following macros are special ones that are used by the LiaScript to deal with a couple of convenience functions.

attribute

Attribution is an important issue. With the attribute command, you can define the attribution that is showed within the info field within the navigation panel. These elements get also imported if you import the functionality from another course.

A good attribution might look like the following ones...

```
<!--
attribute: [AlaSQL](https://alasql.org)
  by [Andrey Gershun](agershun@gmail.com)
  & [Mathias Rangel Wulff](m@rawu.dk)
  is licensed under [MIT](https://opensource.org/licenses/MIT)

attribute: [PapaParse](https://www.papaparse.com)
  by [Matthew Holt](https://twitter.com/mholt6)
  is licensed under [MIT](https://opensource.org/licenses/MIT)
-->
```

author

The author information is visible within the information panel as well as on the course-card on the home-screen.

```
<!--
author: Your name
-->
```

comment

This information is shown at the course-card at the home-screen. It should contain a short and precise description of your course. This macro will only show one paragraph, even if you define more content.

```
<!--
comment: Learn something about ...
  even if you insert multiple paragraphs

  there will only be one paragraph
-->
```

date

A convenience function that can be used to show the latest update time to the user. This is also shown at the information panel.

```
<!--
date: 08/03/2020
-->
```

dark

You can change the default appearance of your document, either if you prefer dark mode or light mode. This will not change the user preferences. The default mode is defined by the user settings.

```
<!--
dark: true

@@ or ...

dark: false
-->
```

email

Simply add some contact information that is shown in the information panel. This can also be overwritten for every section.

```
<!--
email: contact@web.de
-->
```

`import`

You can import the main macros of other courses, simply by using the `import` command, which is followed by the raw URL of the foreign course. You can also import various different courses, every URL will be loaded before the course is loaded. Only the definitions in the main header are loaded, this includes, scripts, css, macros, attribution, and onload. Our basic templates are currently hosted at https://github.com/LiaTemplates . Every course basically describes its macros and how to apply them. Only the content of this course is loaded for speed-reasons, it is currently not possible to load a course, that requires macros of another course and so on.

```
<!--
import: https://raw.githubusercontent.com/liaTemplates/rextester_template/master/
README.md
    https://other_url

import: this will import a third url
-->

``` python
print("Hello World")
```
@Rextester.eval(@Python)
```

`input`

Use this only in conjunction with executable code and projects or with quizzes. This macro can only be used in a script tag and gets replaced by the current user input.

To refer to the inputs in a project, use the parameterized macro:

```
@input(0)   <== equal to @input
@input(1)
```

`language`

Set the internationalization of the course. This will set basic button information The currently available languages are defined in:

https://github.com/liaScript/lia-localization

```
<!--
@@bulgarian
language: bg

@@english
language: en

@@german
language: de

@@persion
language: fa

@@armenian
language: hy

@@dutch
language: nl

@@ukrainian
language: ua
-->
```

`link`

If you need to load additional CSS files, use `link` followed by the URL of your stylesheet. Similar to `import` or `script`, you can load multiple files. See https://github.com/liaScript/custom-style for more information on how to define custom styling.

```
<!--
link: https://some.css
  https://another.css
```

```
```

`logo`

The logo definition requires a URL of an image, weather absolute or relative. It is used to define a background image for the course-card at the home-screen.

```
<!--
logo: ./pics/logo.png
-->
```

`mode`

You can change the default style of your document, either if you do not have any effects you can set mode to `Textbook` or start with and interactive `Presentation`. The three modes a the same as defined within the document at the upper right button. The default mode is defined by the user settings.

```
<!--
mode: Presentation

mode: Slides

mode: Textbook
-->
```

`narrator`

Set the narrator-voice of your course-speaker. The voice is a service of https://responsivevoice.org that is free for non-commercial educational content. The list below shows all currently available voices and languages.

```
<!--
narrator: Afrikaans Male
-->
```

Female	Male
UK English Female	UK English Male
US English Female	US English Male
	Afrikaans Male
	Albanian Male
Arabic Female	Arabic Male
	Armenian Male
Australian Female	Australian Male
Bangla Bangladesh Female	Bangla Bangladesh Male
Bangla India Female	Bangla India Male
	Bosnian Male
Brazilian Portuguese Female	Brazilian Portuguese Male
	Catalan Male
Chinese Female	Chinese Male
Chinese (Hong Kong) Female	Chinese (Hong Kong) Male
Chinese Taiwan Female	Chinese Taiwan Male
	Croatian Male
Czech Female	Czech Male
Danish Female	Danish Male
Deutsch Female	Deutsch Male
Dutch Female	Dutch Male
	Esperanto Male
	Estonian Male
Filipino Female	
Finnish Female	Finnish Male
French Canadian Female	French Canadian Male
French Female	French Male
Greek Female	Greek Male
Hindi Female	Hindi Male
Hungarian Female	Hungarian Male
	Icelandic Male
Indonesian Female	Indonesian Male
Italian Female	Italian Male

Japanese Female	Japanese Male
Korean Female	Korean Male
Latin Female	Latin Male
	Latvian Male
	Macedonian Male
Moldavian Female	Moldavian Male
	Montenegrin Male
Nepali	Nepali
Norwegian Female	Norwegian Male
Polish Female	Polish Male
Portuguese Female	Portuguese Male
Romanian Female	Romanian Male
Russian Female	Russian Male
	Serbian Male
	Serbo-Croatian Male
Sinhala	Sinhala
Slovak Female	Slovak Male
Spanish Female	Spanish Male
Spanish Latin American Female	Spanish Latin American Male
	Swahili Male
Swedish Female	Swedish Male
Tamil Female	Tamil Male
Thai Female	Thai Male
Turkish Female	Turkish Male
Ukrainian Female	
Vietnamese Female	Vietnamese Male
	Welsh Male

`onload`

Sometimes it might be necessary to preload some JavaScript code that is gets executed before the course is loaded. Or you want to define some global functions that can be used afterwards everywhere. `onload` actually does the same as its HTML counterpart and is used to accomplish this task.

```
<!--
@onload
// this macro contains some important functions that are loaded
// only once ...

alert("Hello World")
@end
-->
```

`script`

Load all required JavaScript files/libraries with this command.

```
<!--
script: https://some.js
  https://another.js
  ...
-->
```

`translation`

If you already have some translated versions of your course or know where they can be found, then use this macro. Simply add the name and the URL. This links will be also visible within the information panel of your course.

```
<!--
translation: Deutsch  translations/German.md
translation: Français translations/French.md
translation: Русский  translations/Russian.md
-->
```

`uid`

This macro can only be used within the document, not within the header. Sometimes it is necessary to define unique identifiers if you want to access some HTML nodes and do not want to name every manually. In this case it might be useful to define hidden macros, that make use of `@uid` which generates a unique identifier, whenever it is called. See for example https://github.com/liaTemplates/Rextester to see the application of this pattern for creating macros.

```
<!--
@eval: @hidden_eval(@uid,@input)

@hidden_eval
<script>
// eval @input
...
// and do somethint with the content of "name_.."
document.getElementByName("name_@0")
</script>


<div id="id_@0"></div>

@end
-->
```

`version`

A very important information is the `version`, it follows the major.minor.patch notion. If you start to develop a course, you can leave this out, and you will be in some kind of development-version or as long as you are in major-version 0 the course should be parsed, every time it is loaded. This is fine for small courses and no persistent content. But, if you include executable code or quizzes, this content is stored persistently at IndexedDB directly within your browser.

If your course is ready or very large, it might make sense to define a major version, such as `1.0.12`. The benefit of this is, that every time, the user loads the course, the app tries to download the raw Markdown file and then it checks the version information, if the version is the same as the one that is currently stored within IndexedDB, then it does not require to preprocess the entire document again, instead it directly loads the preprocessed version directly. This is especially useful for smart phones.

```
<!--
version: 0.0.1
...
-->

# Main Title
```

So how to change the version information?

If you only fix typos or add or change static content, it is sufficient enough to increase the patch element and thus for example `1.0.14` to `1.0.15`.

If you add slides to the end of your document, it is okay to increase the minor version number, eg. going from `1.0.15` to `1.1.0`.

Major versions changes are required, if you change the structure of your document, especially quizzes and code. As already mentioned these persistent state information are stored within IndexedDB, but moving code or quizzes from one slide to another will might destroy previous states. For this purpose major version changes are required (`1.1.0` → `2.0.0`), which will result in a new version also in IndexedDB. The old code/state is still available and can be restored.

JavaScript

In contrast to common Markdown-Parsers it is also possible to include and execute javascript code. If you combine it with your HTML elements, you are free to integrate whatever you want.

The last statement of your script defines also the result, that will be shown, if and only if it is not `undefined`, or simply use `console.log` to log the script activities. As the examples below show, you can combine your scripts with LiaScript animations. Thus, they will only be execute in the right fragment/context. But. you can do much much more with scripts.

> Checkout Section JavaScript for more information!

```
Do some internal calculation <script> 99 * 88  </script>, the next script
contains an error <script> 99 * a </script>.

                                {{1}}
<div class="ct-chart ct-golden-section" id="chart"></div>
<script>
    // Initialize a Line chart in the container with the ID chart
    new Chartist.Line('#chart', {
        labels: [1, 2, 3, 4],
        series: [[100, 120, 180, 200]]
    });

    console.debug("loaded #chart") // or undefined
</script>
```

Do some internal calculation `8712`, the next script contains an error: `a is not defined`.

Note, you have to include all required JavaScript-resourses in the initial comment after the script definition. And by combining this feature with LiaScript effects, you can build even more sophisticated courses.

JavaScript or JS-Components

> By the time of writing this, I do strongly believe that the script-tag, that was introduced by Netscape in 1995 (cf. Wikipedia), is used in the wrong way. It is still somehow outside of HTML, although, if it could embedded as part of the DOM, we would not have to search the DOM for IDs and try to manimpulate the content of a certain node. Much of the work, that we try to put into the development of Web-Components could be easily achieved by using only the script-tag slightly different.

In LiaScript we now have the power to insert script everywhere and to connect them with a simple publish-subscribe mechanism, in order to create even more interactive books and courses. Thus you can add additional calculations everywhere within your document and the internal LiaScript- event-system handles their execution. This can be seen as an inverse attempt to Jupyther- or R-Notebooks, where content is placed around code for documentation. We try to integrate code as a native element into content.

> All scripts are executed only after all external javascript-files have been imported.

Basics

Simply use the well-know `<script> ... </script>` (nearly) everywhere within your document to perform any kind of calculation 10 percent from 99 $ is equal to `9.9` . Or, you can also query any kind of external service such as https://www.wikidata.org, weather forecasting from https://openweathermap.org or random gifs of cats from https://giphy.com, as it is depicted below.

> If you are "reading" this course in LiaScript, you will have noticed, that the background of those script-results is actually gray. You can double-click on them to see what is actually inside and edit the code. If you click somewhere else, so that the object loses its focus, the script gets reevaluted.

Usage

The snippet below depicts how such scripts can be defined. The result of a script is directly used as the content of the cell and placed exactly there where the script-tag was originaly defined. The result of the first cell is obvious, it is a number. The larger second script actually has two different results, the first result is the string `loading`, the second one is generated by the promise of the JavaScript fetch-API, it triggers a download process for the given url and if this is successful, the send-object is used to send back a string that is internally interpreted as a new HTML-image.

```
Some inline calculation: <script>99 * 0.1</script>

<script run-once>
fetch('https://api.giphy.com/v1/gifs/random?api_key=...tag=cat')
  .then(response => response.json())
  .then(data => send.html(`<img height="250px"
  src="${data.data["image_url"]}">`))

"loading"
</script>
```

The following code snippet shows a more obvious example, the script generates a timeout that triggers the execution of the function after 5 seconds. Which means, that the first result is the string `waiting for 5 seconds` and after this time a new result is generated: `I am ready!`.

```
<script run-once>
setTimeout(function(){
  send.lia("I am ready!")
}, 5000)

"waiting for 5 seconds"
</script>
```

However, there is a better way of doing this. Within the example above, the script is marked by LiaScript that is not running anymore, since a first valid result was already generated, which means, that this script is triggerd also for the next time, if you go to the next slide and return. One way of dealing with this issue, is to used the attribute `run-once` or `run-once="true"` to indicate, that the code should only be executed once. The result of the script is stored and the script is never again executed. But, you can also use the same methids as they were introduced in section Communication.

That means, you can use the two return strings, but you can use also the two functions:

- `"LIA: wait"` ⟶ `send.wait()`
- `"LIA: stop"` ⟶ `send.stop()`

```
<script>
setTimeout(function(){
  send.lia("I am ready!")
}, 5000)

"LIA: wait" // or send.wait()
</script>
```

The function `send.lia` marks the execution of the script to be finished, but in the case that your produce continous results, you can use the following:

```
<script>
function counter(i) {
  if (i > 0) {
    send.output("HTML: <h"+i+" style='display: inline-block'>hallo " + i
    +"</h"+i+">")
    setTimeout(() => counter(i-1), 1000)
  } else {
    send.stop()
  }
}

counter(6)

send.wait() // or "LIA: wait"
</script>
```

Actually only two functions, `send.lia` and `send.output`. The functions `send.wait` and `send.stop` are actually only shortcuts for `send.lia("LIA: wait")` and `send.lia("LIA: stop")`. In the same way `send.html` is only a shortcut for `send.lia("HTML: " + ...)`, as it is depicted below.

```
<script>
let html_string = "<h2>Markdown</h2>"
send.html(html_string)
send.lia("HTML: " + html_string)
"HTML: " + html_string
</script>
```

Errors

Every script can contain errors, and so does the follow script `a is not defined`. The error message is printed in red, even if you only wanted to execute a script, that does not produce any outputs. You can double-click onto the error message to edit the code and define the variable `a`.

Execution & Animations

You can combine scripts with LiaScript animations and thus trigger their execution. If you are in textbook mode all scripts will be executed. If a script does not belong to an animation, it will be executed immediately when you open a slide on every time you visit the slide, if you do not set the parameter `run-once` or if the the slide is not marked as running.

```
<script>alert("0")</script>

{{1}} <script>alert("1")</script>

{{2}} <script>alert("2")</script>

{{3}} <script>alert("3")</script>
```

A script is only executed within the local scope of a slide. It is not possible to trigger the execution of scripts from slides that have not been visited. But scripts can remain active, even if you go to another slide.

`input`

Using only static scripts is actually boring, we want to interact with the scripts and pass input values. In LiaScript this is achieved through a combination of script-tags with input-tags. Simply add the parameter `input` to with an additional type information to the script tag.

> All possible types that are defined by the HTML5 standard can be used, as well as their specific parameters. These are automatically passed to the generated input:
>
> https://www.w3schools.com/tags/tag_input.asp

Scripts that are combined with an input are marked by a thin frame, you can click on them and change the input. And, different inputs pruduce different result and might trigger the execution slightly differently 😏

`button`

The code as depicted below, will implement a simple clickable button. The script is loaded once, when the slide is rendered and then every time you click on it.

```
<script input="button">
alert("click")

"click me"
</script>
```

click me

`text`

This is also the default input. It shows also the usage of the `@input` macro. It defines the place, where the current the input value should be placed. Since with the same input, text, numbers, code, etc. could be defined, you have to decide if it is interpreted as a string in `""` or used as part of your code. The attribute `value` is used as the default initial input to your script, for the first execution. If no `value` is defined, then the default input is an empty string.

```
<script input="text" value="reverse">
let str = "@input"
// the input string gets reversed
str.split("").reverse().join("")
</script>
```

Text inputs are evaluted emediately on every change.

esrever

For more information about text inputs see:

https://www.w3schools.com/tags/att_input_type_text.asp

`number`

The number input is similar to text, but only numbers are allowed as input and you can set additional parameters such aus `min`, `max`, and `step`.

```
<script input="number" value="1" min="0" max="1000000">
let i = @input // direct usage as a number

"Square of " + i + " = " + i * i
</script>
```

As for texts, the input is evaluted on every change.

Square of 1 = 1

For more information about number inputs see:

https://www.w3schools.com/tags/att_input_type_number.asp

`range`

The range is actually a slider, that generates numbers as input and you can set additional parameters such aus `min`, `max`, and `step`.

```
<script input="number" value="1" min="0" max="1000" step="0.1">
let i = @input // direct usage as a number

"Square of " + i + " = " + i * i
</script>
```

The input is evaluted on every change.

Sinus of 0 = 0

For more information about range inputs see:

https://www.w3schools.com/tags/att_input_type_range.asp

`search`

The search input is actually text input, but in contrast to text, the script is only executed after the input field looses its context.

```
<script input="search" value="abcdefg">
let str = "@input"

"reverse of \"" + str + "\" -> " + str.split("").reverse().join("")
</script>
```

reverse of "abcdefg" -> gfedcba

For more information about search inputs see:

https://www.w3schools.com/tags/att_input_type_search.asp

`password`

The password input is actually text input, but in contrast to text the input is not directly visible and the script is only executed after the input field looses its context.

```
<script input="password">
let password = "@input"

if (password == "LiaScript") {
  "You got it!"
} else {
```

please enter the correct password

> For more information about password inputs see:
>
> https://www.w3schools.com/tags/att_input_type_password.asp

`radio`

In contrast to the commonly usage of radio buttons, which requires the definition of multiple radio inputs, LiaScript achieves the same with input `radio` and the definition of the `options` parameter. All possible options are separated by `|`. Thus, the user can only select one of the defined options.

```
<script input="radio" value="1" options="1|2|3|Hello World|true">
"Selected option: @input"
</script>
```

Changes will trigger an immediate execution of the script.

Selected option: 1

> `options` is not a standard HTML parameter, for more information on radio inputs see:
>
> https://www.w3schools.com/tags/att_input_type_radio.asp

`select`

`select` is actually not an input type, but we added it, since it allows to perform the same task as radio-buttons, but the representation is a drop-down list. The usage of `options` and their separation by `|` is equal to radio buttons.

```
<script input="select" value="1" options="1|2|3|Hello World|true">
"Selected option: @input"
</script>
```

Changes will trigger an immediate execution of the script.

Selected option: 1

> `options` is not a standard HTML parameter, for more information on radio inputs see:
>
> https://www.w3schools.com/tags/tag_select.asp

`checkbox`

Checkboxes in contrast to radio-buttons or select allow you to select multiple elements at once or none.

If no options are defined, a checkbox is treated as a single input that switches between true and false, wheather, the checkbox is checked or not:

```
<script input="checkbox" value="true">
"@input"
</script>
```

true

If you define `options` the current value as well as the result is treated as a list of strings in JSON format:

```
<script input="checkbox" value='["Ben"]' options="Ben|Jerry|Tom|Hardy" >
@input
</script>
```

["Ben"]

This might not be a sufficient and readable, therefor it is also possible to define formats for outputs (see section Formatting with Intl). The `list` format for example allows to add language specific textual formatting for list. If you do not add locale information the document language is used as a default.

```
<script
  input="checkbox"
  value='["Ben", "Jerry", "Tom"]'
  options=" Ben | Jerry | Tom | Hardy "

  format="list" locale="en"
>
  let list = @input

  if (list.length == 0)
    ["no one"]
  else
    list
</script>
```

Ben, Jerry, and Tom

> For more information about checkbox inputs see:
>
> https://www.w3schools.com/tags/att_input_type_checkbox.asp

`date`

Date allows offers a datepicker. The normal return value is a string of the format `YYYY-MM-DD` (Year-Month-Day), but it is also possible to tweak this output according to some language specific formats.

```
<script input="date" value="2020-10-10" format="datetime" locale="en">
"@input"
</script>
```

The execution is triggered on every change.

10/10/2020

> For more information about date inputs see:
>
> https://www.w3schools.com/tags/att_input_type_date.asp

`datetime-local`

Datetime-local includes the date and time. The standard return format is `YYYY-MM-DDTHH:MM` (Year-Month-DayTHour:Minutes). The execution is triggered on every change.

```
<script input="datetime-local" value="2020-11-20T12:30">
"@input"
</script>
```

2020-11-20T12:30

But of course, it is also possible to format time and date:

```
<script input="datetime-local"
        value="2020-11-20T12:30"
        format="datetime"
        locale="en"
        dateStyle='full'
        timeStyle='short'>
"@input"
</script>
```

Friday, November 20, 2020 at 12:30 PM

For more information about datetime-local inputs see:

https://www.w3schools.com/tags/att_input_type_datetime-local.asp

`time`

```
<script input="time" value="11:30">
"@input"
</script>
```

11:30

For more information about time inputs see:

https://www.w3schools.com/tags/att_input_type_time.asp

`email`

```
<script input="email" placeholder="please your Email">
let email = "@input"

if (email) {
  email
} else {
  "email"
}
</script>
```

email

For more information about time inputs see:

https://www.w3schools.com/tags/att_input_type_email.asp

`url`

```
<script input="url" value="url">
"@input"
</script>
```

url

For more information about url inputs see:

https://www.w3schools.com/tags/att_input_type_url.asp

`tel`

```
<script input="tel" pattern="[0-9]{3}-[0-9]{2}-[0-9]{3}" placeholder="123-45-678"
  >
let tel = "@input"

if (tel) {
  tel
} else {
  "tel"
}
</script>
```

tel

For more information about tel inputs see:

https://www.w3schools.com/tags/att_input_type_tel.asp

`submit`

I qually used as the `button` input, see therefor section button

`month`

```
<script input="month" value="1999-12">
"@input"
</script>
```

1999-12

For more information about month inputs see:

https://www.w3schools.com/tags/att_input_type_month.asp

`week`

```
<script input="week" value="2020-W40">
"@input"
</script>
```

2020-W40

For more information about week inputs see:

https://www.w3schools.com/tags/att_input_type_week.asp

`file`

not working yet ...

For more information about file inputs see:

https://www.w3schools.com/tags/att_input_type_file.asp

`hidden`

This type is special, as it does not produce any visible output, but as shown in section output, this can be used to hide complex calculation, that might be used by various different elements as input.

```
<script input="hidden">
{value: 12, height: "12px", points: 123.12}
</script>
```

`color`

Defines a color picker. The normal return value has the format `#RRGGBB` (red green blue). The number values are defined in hex ranging from 00 to FF.

```
<script input="color" value="#FF0000">
"HTML: <b style='color:@input'>@input</b>"
</script>
```

#FF0000

For more information about color inputs see:

https://www.w3schools.com/tags/att_input_type_color.asp

`image`

Not used atm.

`textarea`

This is also not a input type, but instead the application of the textarea HTML element. This can be used to edit more complex multiline strings. The script is only executed after the element has lost its context.

```
<script input="textarea" value="<span style='font-size: 32px'>hallo</span>" style
  ="width:100%">
`HTML: @input`
</script>
```

hallo

> For more information about textareas see:
>
> https://www.w3schools.com/tags/tag_textarea.asp

`output`

You can use the `output="channel-name"` parameter to define a dedicated channel, on which the script publishes its changed outputs and other scripts can subscribe to these changes via `input(channel-name)`, but the channel name has to be in markdownish style backtics, as depicted below:

```
publishing node P
<script input="checkbox" value="true" output="P">
@input
</script>
AND Q
<script input="checkbox" value="true" output="Q">
@input
</script>
result in:
<script output="AND">
@input(`P`) && @input(`Q`)
</script>
```

publishing node P `true` AND publishing node Q `true` result in: Invalid or unexpected token

Within the example above you can also see the difference between scripts with and without associated inputs. Scripts with inputs do have a frame, while scripts without inputs do only have grey backgound.

Combining scripts with macros

Such scripts are of course complicated to read and require a lot of boilerplate code, which could and should be hidden behind macros, to make the text event more readable. The example below, shows how two macros can be defined that could be used everywhere within your code, also with changing parameters.

```
### `output`
<!--
boolean: <script input="checkbox" value="true" output="@0" modify="false">
         @input
         </script>

and:     <script modify="false">@input(`@0`) && @input(`@1`) </script>
-->

@boolean(p), @boolean(q) --> @and(p,q)
```

`true`, `true` ⟶ true

> The parameter `modify="false"` removes the the gray backgound, so that scripts without any input become indistinguishable from the rest of the content. This gray backgound also indicates, that the code behind the script is **editable**. You can switch to the edit mode via a double-click on the script element. Thus you can inspect every script and also edit it, the code will only be executed after the textinput has lost the focus.

`default` & Execution-Graphs

If you connect scripts, this connection should actually form a directed graph, without cycles. But of course, if you want to, can produce also cycles, that stop only if the results do not change anymore.

```
<script input="range" value="1" default="1" output="x">
@input * @input(`y`)
</script>
<script input="number" value="1" default="1" output="y">
@input + @input(`x`)
</script>
```

The `default` parameter is used to define a the default output of this script. Otherwise the initial calculation will result in an error, since the scripts do require the outputs of the other script that can never be calculated, due to the circular dependency.

1 2

Further settings

`value`

Set the default input value, if it is a number or a string or something else, depends on the usage of the `@input` macro within the script.

`update-on-change`

As mentioned earlier, every input-field has a specific update handling, `range`, `number`, `text`, `radio`, `checkbox`, trigger the execution of the script on every change, while the others are triggered only after the user hits enter or if the input field loses the focus. However, by using the parameter `update-on-change` you can change this behavior. It can be switched off or on by passing true or false, if you only pass `update-on-change` true is used as the default:

- `update-on-change`
- `update-on-change="true"`
- `update-on-change="false"`

`input-active`

Inputs are only visible, if the user clicks on the script representation. If you pass the parameter `input-active`, then the input will be visible on the first appearance, but it will be closed if the element loses the focus again.

`input-always-active`

Using this parameter, it is possible to switch on the input-field for ever, it will not be closed if the focus is lost.

`run-once`

A script will be executed multiple times, if the site is rendered, or if it is associated to a certain effect number. If the calculation should be executed only once use this parameter. The result is preserved and displayed on every appearance.

`modify`

By double-clicking onto a script, you get into edit-mode. The code is displayed to the user and can be edited and it is executed again if the code-input field loses the context. By setting `modify="false"` to false, this editing function is switched of furthermore there is **no gray background** displayed.

This is a script: 12

Formatting with Intl

By using the `format` parameter you can set a specific kind of visual formatting. This will be visible to the user, but if you connect different script with input/output then the original result of an execution will be passed to the subsequent scripts. The links below show contain all information to the associated formatting all params are directly passed to the formatting function.

Use `locale` to change the type of language/localization. If you do not pass such a value, then the default document language setting is used as default.

> There is one difference by using the `style` parameter. `style` is used format the output with inline CSS. But some formattings also contain a `style` parameter. In order to don't mess up with styles you have to use `localeStyle` to set the locale language formatting style ;)

- `format="datetime"`:
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl/DateTimeFormat/DateTimeFormat
- `format="number"`:
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl/NumberFormat/NumberFormat
- `format="list"`
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl/ListFormat/ListFormat
- `format="relativetime"`:
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl/RelativeTimeFormat/RelativeTimeFormat
- `format="pluralrules"`:
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl/PluralRules/PluralRules

Further Examples

Tables

```
#### Tables
<!--
sin: <script format="number"
                localeStyle="currency"
                currency="EUR"
                locale="de-DE"
                modify="false"
    > Math.sin(@input(`result`) + 0.5 * @0) </script>
-->

<script run-once
        default="0"
        output="result"
        input="range" value="2" min="0" max="25" step="0.1"
        >
@input
</script>


<script run-once
        default="0"
        output="amp"
        input="range" value="2" min="0" max="25" step="0.1"
        >
@input
</script>


<!-- data-type="barchart" id="tabelle" -->
| Header 1 | <script>@input(`result`)</script> |
|:-------- |--------: |
| 1        | @sin(1)  |
| 2        | @sin(2)  |
| ...      | ....     |
```

Pos: 2 and amplitude: 1

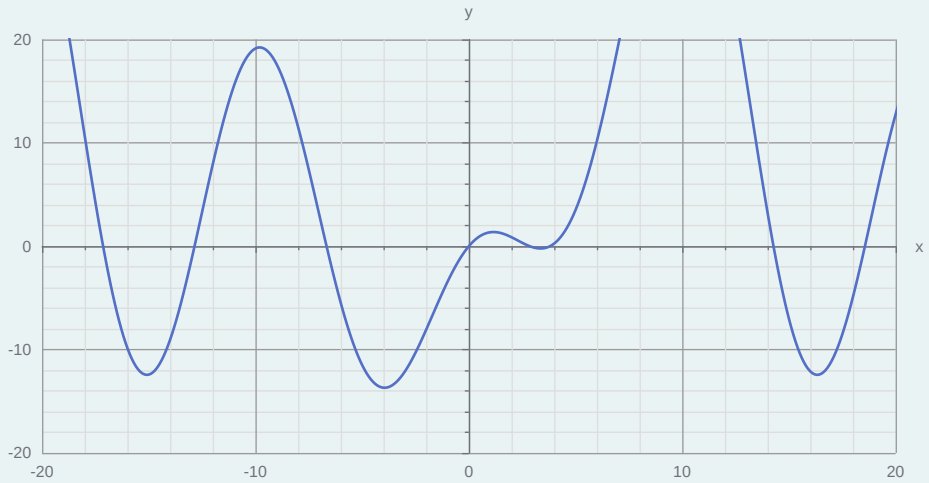
Header 1	2
1	0,14 €
2	-0,76 €
3	-0,96 €
4	-0,28 €
5	0,66 €
6	0,99 €
7	0,41 €
8	-0,54 €
9	-1,00 €

Search

To change the current search subject `cats` to something else, simply click on the search and start to type... If you click on the image below, a new random search will be triggerd.

Diagrams

The first value defines some kind of range: `2` , while the second can be interpreted as range `50`. You can double-click on any gray element to inspect and edit its javascript code.



Calculator

```
#### Calculator
<!--
calc: <script run-once format="number" locale="de" notation="compact">@0</script>
-->

Interim calculation @calc(22*12345.98726) or @calc(`Math.pow(3.141592, 12)`)...
```

Interim calculation `272K` or `924K` ...

Future Work

- Better integration with github/gitlab and the versioning of courses
- Automagically analyzed surveys
- Integration of WebGl and a 3D navigation
- Better Inline function-plotter

Contributors and Credit

André Dietrich

Programming paradigm experimenter and creator of LiaScript and SelectScript...

Sebastian Zug

The mind in the dark and the man behind the eLab-project ...

Karl Fessel

Embedded systems developer, creator or arduinoview, and Markdown evangelist ...

Steve Rehm

CSS & SASS crack and friendly face behind new face of LiaScript ...

Appendix