

## 1 Standard Library

<b>Dom</b>		[observable]
canvas	context2d	download element
input	range	select svg
text	uid	
<b>Files</b>		[observable]
buffer	text	url
<b>FileAttachments</b>		[attachment]
achmt = FileAttachment("file")	# construct	
url	text	json csv
tsv	image	arrayBuffer stream
blob		
<b>Promises</b>		[observable]
delay	tick	when
<b>Generators</b>		[observable]
disposable	filter	input
map	observe	queue
range	valueAt	worker

**Require** [require]  
req = require("d3-array") # from JSDelivr  
resolve alias

**Literals** [observable]  
The following are top-level objects. See sections on markdown and literals.

md html tex dot

**Reactive Variables** [observable]  
These are top level objects that observe and react to changes on the notebook and server.  
invalidation now width

## 2 Data

### Inline

For small datasets:  
primes = [2,3,5,7,11] # manual entry  
csv = d3.csvParse(`rawPaste`) # copy-paste

### Attachments

Attach a file in the UI and then:  
a = FileAttachment("file") # attachment object  
d = await a.json(); # promised data  
t = d3.hierarchy(d) # in-mem tree

In lieu of d3, FileAttachment has its own csv, json, and text parsers that implicitly await:  
c = FileAttachment("x.csv").csv()

Alternatively, upload directly from local f.s.:  
viewof file = html`<input type=file>`  
html`<img src=\${URL.createObjectURL(file)}>`

### Databases

Observable is meant to share notebooks and data. It is also great for prototyping. Using the runtime, you may even use it to construct full web-apps.

However, notebooks themselves are not meant to serve full applications. Hence, UI-created db connections are only allowed for private notebooks.  
npm install -g @observablehq/database-proxy  
... # run locally

### Secrets

Use the UI to create a secret (a la Github). Then:  
Secret(`MY\_KEY`) # expose it, or embed:  
dat = d3Fetch.json(url + `{MY\_KEY}`)

### Files

Use this API to retrieve from local filesystem.  
viewof myText = html`<input type=file>`  
Files.text(myText) # invoke the API

## 3 Inputs

### General Use

import {Checkbox} from ... # from here  
data = [...]; options = {...}; # setup  
viewof cb = Checkbox(data, options) # create  
html`checked: # reference  
\${cb.map(c => html`\${c}`)}`

Most canonical inputs have the 2 arguments above, where data is an array, and options is an object-literal, as described below.

### Options

Most inputs allow arbitrarily-formatted data but then require using non-trivial keyof, valuesof closures for data extraction. Other options are mostly cosmetic.

	Button	Chckbx	Toggle	Radio	Range	Search	Text	Textar.	Select	Table
Label	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Value	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Width	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Disabl'd	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Sort		✓							✓	✓
Unique		✓							✓	✓
Locale		✓				✓			✓	
Format		✓		✓	✓	✓			✓	✓
Keyof		✓							✓	
Valueof		✓	✓						✓	
Plachldr						✓	✓	✓		
Columns						✓	✓	✓		✓
Sp'ch'k						✓	✓	✓		
Requ'd						✓	✓	✓		✓
Rows						✓	✓	✓		✓
Datalst						✓	✓			
Multip.									✓	✓
Reado'ly							✓	✓		
Minlen.							✓	✓		
Maxlen.							✓	✓		
Validate							✓	✓		
Submit							✓	✓		

The following are exclusive to one input type:

- button required, reduce
- toggle values
- range step, transform, invert
- select size
- text type, pattern
- search query, filter
- table reverse, align, maxwidth, height, maxHeight, layout

### Idiosyncratic / User-defined Inputs

- 2D Slider
- Binary Input
- Ternary Slider
- Fine-range Slider
- Form Input
- Range Slider
- Scrubber
- Player
- Inputs in a Grid
- Copier
- US Counties
- Periodic Table

## 4 Imports & Exports

### Cell Imports

Successively: static import; multiple cells; with dependency injection; aliasing; non-public.  
import {chart} from '@mbostock/phases-moon'  
import {chart, viewof year} from ...  
import {chart} with {mydata as data} from ...  
import {chart as barchar} ...

Note: Imported cells evaluate dependencies without binding them to the local notebook. Bindings are live (Δ with underlying dependencies), but lazily evaluated. Since you can only import named cells, consider “forking” a notebook instead.

### Require

Used for plain JavaScript library imports.  
mmnt = require("moment") # moment lib  
mmnt = require("moment@2") # vers. 2  
ab = require("ä", "b") # multiple → 1

Libs fetch from npm by default. Override like:

d3 = require("my.server/d3.js") # www loc  
d3 = require("localhost/d3.js") # local

As an alternative, consider ES6-native imports:

d3\_esm = import("https://some.cdn/d3@6")

### Sharing

Notebooks default to “private” access but can also be “shared” (exposing an obfuscated url) or “published,” all as available in the UI. Alternatively, click to left of a cell to download its contents to png or json, etc.

### Embedding

To the left of a cell, select “embed” to generate copy-pasteable html IFrame code (which will, in turn, include the observable runtime). Here is a tool that generates an IFrame for multiple cells. Alternatively, download a zip archive of project contents by clicking “download code.” The generated archive is described here. Even better, just install it via npm

using the url in the download code menu option:  
unix-shell\$ @observablehq/runtime  
unix-shell\$ npm install "<url>"

### Observable Runtime

A Runtime instance can spawn a module, which, when passed an observer, spawns a variable, which is a piece of state in a reactive program. The standard observer is Inspector, which renders the current value of the given variable to its associated DOM element.

```
<html><body><script type="module">  
import {Runtime, Inspector} from ...  
import define from "https://...";  
const runtime = new Runtime();  
const main = runtime.module(define,  
  name => { if (name === "hello") {  
    return new Inspector(document.body);  
  }  
});  
</script>
```

## 5 Markdown

### Headers

# Title # H1 header  
## Title # H2 header  
### Title # H3 header

### Emphasis

**text** # bold text  
*text* # italicized text  
~~text~~ # striken-thru text  
> text # quoted text

### Hyperlinks, Images, & Videos

[http://google.com](click here!)  
![http://clip.art](pic's subtitle)

Embed youtube video:

```
[[description]  
(http://img.youtube.com/vi/myvidid/0.jpg)]  
(http://www.youtube.com/watch?v=myvidid)
```

### Lists

Unordered      Ordered  
\* item1          1. item1  
\* item2          1. item2  
\* item3          1. item3

Mixed, Nested lists:

```
1. First major item  
  * 1st sub-item  
  * 2nd sub-item # whitespace matters!  
1. Second major item
```

**Code**

normal `code()`text	# inline
```	# start ...
myFunction({})	# code block
```	# ...end

In observable, backticks must be escaped (see below), so really, things look more like:

<code>template literal</code>	<code>tagged TL</code>
<code>```</code>	<code>```html</code>
<code>myFunction({})</code>	<code>&lt;i&gt;ital&lt;/i&gt;</code>
<code>```</code>	<code>```</code>

### Escaping

The following characters must be escaped: `*`, and `>`. Within observable, additionally escape ```.

### Tables

For all but the simplest tables, embed html, but for simple tables, use raw markdown syntax:

head1   head2   head3	# headers
——— ——— ———	# horizontal rule
val   val   val	# rows

### Checklists

- [ ] Mercury	# unchecked
- [x] Venus	# checked
- [x] Earth	# note: '-' for list
- [ ] Mars	# space ↔ empty

### HTML

Most html can be embedded successfully. Some idioms are below, the first of which is used to add dynamic html text highlighting:

<code>&lt;span \${spanStyle('link')}&gt;*Link*&lt;/span&gt;</code>	
<code>&lt;span&gt;&amp;# 10004&lt;/span&gt;</code>	# unicode char
<code>&lt;dl&gt;</code>	# begin definition list
<code>&lt;dt&gt;&lt;/dt&gt;</code>	# definition header
<code>&lt;dd&gt;&lt;/dd&gt;</code>	# definition
<code>&lt;/dl&gt;</code>	# end
<code>&lt;figcaption&gt;text&lt;/figcaption&gt;</code>	# nice captn
<code>&lt;figure style='max-width:50px'&gt;</code>	# nice image
<code>&lt;img src='https://pic.location'&gt;</code>	#
<code>&lt;/figure&gt;</code>	# end

## 6 Literals

No matter the language below, observable always requires “tagging” the “template literal,” by prefixing the target language prior to the template-demarcating graves ( ``` ).

### Markdown

(See separate section on md.)

`md`# Page Title``

### HTML

<code>html`&lt;p&gt;I am a &lt;i&gt;par&lt;/i&gt; element!&lt;/p&gt;`</code>	
<code>html`&lt;style&gt;</code>	# this css
<code>.highlight {background: yellow;}</code>	# applies
<code>&lt;/style&gt;`</code>	# globally

### SVG

`svg`<svg width=60 height=60>`  
`${svg.fragment``  
`<circle cx=30 cy=30 r=30></circle>``  
`</svg>``

### GraphViz / DOT

`dot`digraph { a -> b; }``

### T<sub>E</sub>X

<code>tex`E = mc^2`</code>	# inline
<code>tex.block`...`</code>	# block-mode

### Lorem Ipsum

Really just a cell import from here, then:

`md` ${loremipsum({using: "Bro Ipsum"})}``

### Template Expansion

All literals are actually templates that are parsed into intermediate languages (like T<sub>E</sub>X and markdown) and ultimately resolved to html. Prior to parsing, observable expands “interpolated expressions” (those inside `${...}`) into their “substituted” string value.

``<h1>Hello ${value}</h1>``

### Nesting Literals

`md`Some tex: ${tex`\tau`}.`` # T<sub>E</sub>X in md

Deep nesting: given observable / js variable:

`array = [tex`A=1`, tex`A=2`]`  
*...nest T<sub>E</sub>X inside js inside html inside js inside html inside md!*  
`md`A list of elements:`  
`<ul>`  
`${clone(array).map(a => html`<li>${a}`)}`  
`</ul>``

## 7 Keyboard Shortcuts

### Cell Editing

<code>↵↩</code>	<code>^S</code>	# run current cell
<code>↵↵↓</code>		# focus next cell
<code>↵↵↑</code>		# focus prev. cell
<code>↵↩</code>		# split cell @ cursor
<code>↵↩↩</code>		# ibid, focusing 1st
<code>↵↩</code>		# insert cell, after
<code>↵↵↩</code>		# insert cell, before
<code>↵↵↑</code>		# move cell up
<code>↵↵↓</code>		# move cell down

### Cursor

<code>↵↵</code>	# merge cell w/ prev.
<code>↵↵-D</code>	# merge cell w/ next
<code>↵↵-P</code>	# [un]pin cell
<code>↵J</code>	# jump to ref'd cell
<code>🔍</code>	# blur current cell
<b>Cursor</b>	
<code>↵←</code>	# previous word
<code>↵→</code>	# next word
<code>↵←</code>	# start line
<code>↵→</code>	# end line
<code>↵-HOME</code>	# to start of cell
<code>↵-END</code>	# to end of cell
<code>↵-CLICK</code>	# multiple carets

### Selection

<code>↵←←</code>	# extend sel. left
<code>↵→→</code>	# extend sel. right
<code>↵↑</code>	# extend sel. up
<code>↵↓</code>	# extend sel. down
<code>↵↵←</code>	# extend sel. word
<code>↵↵→</code>	# extend sel. word
<code>↵↵←→</code>	# to start of line
<code>↵↵←→</code>	# to end of line

### Text

<code>↵-BACKSPACE</code>	# delete word before
<code>↵↵↵</code>	# delete word after
<code>↵→ </code>	# auto-indent
<code>↵- </code>	# indent once
<code>↵-[</code>	# de-indent once
<code>↵-/</code>	# toggle comment (JS-only)

### Cell Navigation

<code>k</code>	# select cell above
<code>j</code>	# select cell below
<code>↵↩↩</code>	# run selected cells
<code>↵-k</code>	# expand selection up
<code>↵-j</code>	# expand selection down
<code>↵↑</code>	# move cells up
<code>↵↓</code>	# move cells down
<code>o</code>	# insert cell after
<code>↵-o</code>	# insert cell before
<code>x</code>	# toggle select cell
<code>↵-a</code>	# select all cells
<code>🔍</code>	# deselect all cells
<code>↩</code>	# open cell editor
<code>d</code>	# delete selected cells
<code>p</code>	# [un]pin selected cells

### Other

<code>↵-c</code>	# copy selection / line
<code>↵-x</code>	# cut selection / line
<code>↵-v</code>	# paste selection / line
<code>↵-z</code>	# undo
<code>↵-y</code>	# redo
<code>↵-f</code>	# search text

<code>↵-g</code>	# find next occurrence
<code>↵↵-g</code>	# previous occurrence

## 8 Compiler

### Grammar

Cell:

- ImportCell
- NamedCell
- Block
- Expression

ImportCell:

- import NamedImports
- from moduleSpecifier
- import NamedImports
- with namedImports
- from moduleSpecifier

NamedCell:

- Identifier = Block
- Identifier = Expression
- FunctionExpression
- ClassExpression

Identifier:

- IdentifierName
- viewof IdentifierName
- mutable IdentifierName

### Parser

See examples here, or do the following to see how parser creates a json tree or your choosing:

```
import {parseCell} from "@observablehq/parser";
const cell = parseCell(`hello = "world"`);
```

## 9 Misc. Techniques

The below are cool techniques that blend much of the above in a unique, non-categorizable way:

- Gallery of gifs
- CLI-based notebook rendering
- GIF animations from Canvas using gif.js