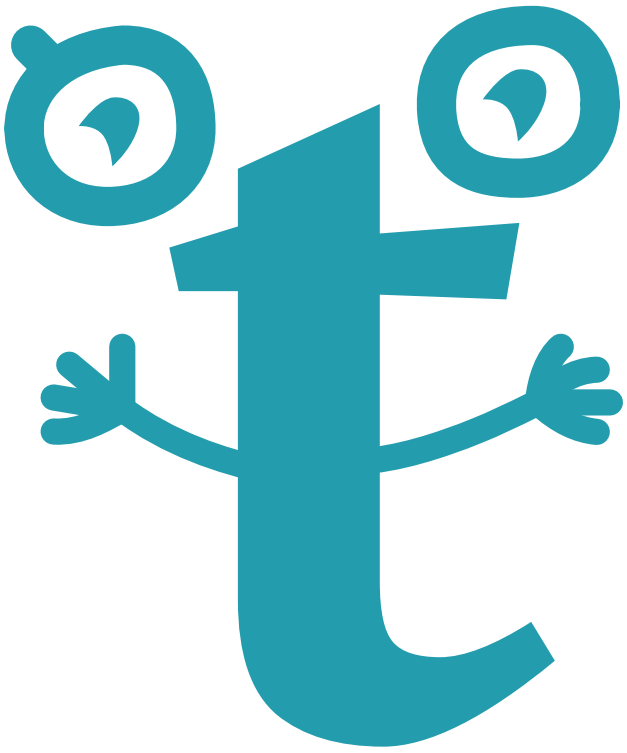
А. Кабанов

Вёрстка документов в Typst



2026

Кабанов А. А.

Вёрстка документов в Typst. — М.: «Литрес: Авторы», 2026. — 105 с.

Этот самоучитель знакомит читателей с основами работы в новой, набирающей популярность, системе подготовки документов Typst. Рассмотрен язык разметки и основы языка сценариев Typst, обеспечивающего широкие возможности настройки внешнего вида и поведения элементов, из которых состоит документ: текста, заголовков, изображений, таблиц, оглавления, библиографии и т. п. Книга снабжена множеством примеров, иллюстрирующих особенности параметров этих элементов.

© Кабанов А. А., 2026 г.

Предисловие

Typst (произносится «тайпст») — это новая, набирающая популярность, система подготовки документов, ориентированная в первую очередь на документы научной тематики, благодаря простоте набора математических выражений. Впрочем, это не значит, что с её помощью нельзя сверстать любые другие документы. Разработчики Typst позиционируют его как альтернативу для LaTeX и более простых инструментов, таких как текстовые процессоры, наподобие Microsoft Word или Google Docs, и их аналогов.

В настоящее время Typst умеет создавать документы в формате PDF, или же в виде набора графических файлов растрового (PNG) или векторного (SVG) формата; каждый файл в последних двух случаях представляет собой одну страницу документа. Существует также возможность вывода в HTML, но как пишут авторы Typst, она пока находится в процессе активной разработки и может не всегда работать правильно.

Авторы Typst предлагают работать с текстами в удобном онлайн-редакторе, доступном по подписке на сайте Typst, в котором, помимо базовых операций редактирования, поддерживается также возможность совместной работы над документом. Есть бесплатный вариант подписки, для которого достаточно просто зарегистрироваться на сайте.

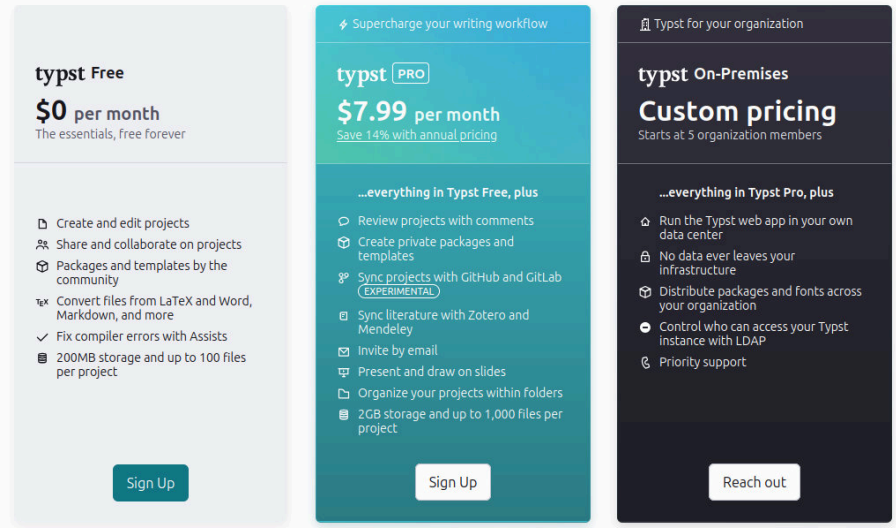


Рис. 1. «Окно онлайн-редактора с открытым шаблоном документа

Если вам по какой-то причине работа над документом онлайн неудобна, или вы просто, как и я, не любите набирать документы таким образом, можно скачать по ссылке на сайте компилятор Typst с интерфейсом командной строки и набирать документ в любом текстовом редакторе.

У компилятора есть очень удобная опция `-watch`, которая позволяет работать над документом практически в режиме реального времени: вы просто вносите правки в текстовом редакторе в файл .typ, и при каждом сохранении исходного файла компилятор автоматически пересоздаёт соответствующий pdf-файл. Если ваша программа просмотра pdf умеет обновлять отображаемый документ при его изменении на диске (а большинство умеют), то вы сразу же увидите результат вашего редактирования.

Некоторых сторонников свободного программного обеспечения отталкивает то, что Typst — это проприетарная программа. Это не совсем так: действительно, онлайн редактор, доступный на сайте, является проприетарным, но вот сам компилятор — это свободная программа, распространяемая под лицензией MIT, и разработчики обещают, что она таковой и останется.

Правда, создавать большие документы с помощью одного только компилятора всё же не очень удобно. Хочется хоть какой-то автоматизации работы, наличия синтаксической подсветки и других уже привычных нам удобств.

К счастью, уже появилась парочка специализированных редакторов Typst, представляющих собой текстовые редакторы с синтаксических подчёркиванием и автодополнением для его языка разметки, к которым уже должным образом подсоединён компилятор и библиотека Typst, так что

пользователю остаётся лишь сохранять документ по мере набора и экспортировать его в pdf-файл. Это **Katvan**, доступный для пользователей Windows и Linux, и **Typesetter**, доступный только для Linux. В общем-то по своим возможностям они не сильно уступают онлайн-редактору.

Также были созданы плагины, позволяющие работать с Typst, для нескольких текстовых редакторов: Visual Studio Code/Codium, Neovim, Emacs, Helix, Zed. Наиболее популярный плагин — это Tinymist, работающий во всех этих редакторах.

В помощь пользователям Typst, в разделе Typst Universe его сайта имеется библиотека из более чем 1000 разработанных сообществом дополнений для Typst, включая **пакеты**, добавляющие в него дополнительные возможности, например рисование диаграмм, удобный набор химических формул и т. п., а также **шаблоны**, представляющие собой готовые стилевые заготовки для различных научных журналов, книг и других документов. Использовать их можно как в онлайн-редакторе, так и в компиляторе, установленном на вашем компьютере.

Typst — ещё очень молодой проект, первый его выпуск стал доступным в 2023 году[1]. На момент написания этой книги доступна версия Typst 0.14.2.

Он быстро эволюционирует, так что не удивляйтесь, если некоторые сведения из этой книги в скором времени окажутся устаревшими.

Немного о создании этой книги

Эта книга о Typst сама полностью написана в Typst, что вполне логично, так как он для этого и предназначен. Считаю нелишним поделиться с читателем некоторыми сложностями, с которыми я столкнулся на своём опыте при её создании.

Я начал писать книгу в редакторе Katvan. Он мне очень понравился быстротой своей работы, видеть изменения в документе по мере набора можно было почти в реальном времени, и поначалу всё шло хорошо. Но после написания около полусотни страниц с редактором возникли проблемы: он периодически, хотя и не очень часто, начал «падать» без каких-либо сообщений об ошибке. Несколько раз после перезапуска исчезала часть только что написанного текста. Благодаря моей маниакальной привычке часто сохранять изменения и создавать резервные копии, больших потерь, к счастью, удалось избежать, но впечатление от программы было изрядно подпорчено.

Всё же жаль. Как я уже сказал, Katvan мне понравился. Но программа ещё молодая и надеюсь, что разработчики всё же доведут редактор до ума.

Пришлось задуматься об использовании редакторов кода, поддерживающих плагин Tinymist. С большинством из перечисленных выше редакторов у меня как то не сложилось, и я остановился на VS Codium.

Не сочтите этот раздел за рекламу, я скажу честно: я не в восторге ни от Visual Studio Code, ни от её «клона» с вырезанной телеметрией VS Codium. Они прожорливые, тормозные и вообще, я не люблю, когда редактор постоянно пытается решать за меня, как мне писать тексты и код. Но после некоторой настройки, заключавшейся в запуске с опцией `--disable-gpu` и отключении всех ненужных мне функций в параметрах программы, работать с Typst в ней оказалось довольно комфортно. Так что, несмотря то, что в нём тоже обнаружились небольшие «глюки» в процессе просмотра результатов редактирования, вся оставшаяся часть книги была дописана в VS Codium. Кстати, плагин Tinymist в нём имеется в наличи в списке доступных плагинов и устанавливается парой щелчков мышкой.

[1] Хотя сами авторы Typst называют датой рождения проекта 2019 год — видимо, тогда у них появилась концепция проекта.

Глава 1. Знакомство с языком разметки Typst

В отличие от LaTeX, в котором большая часть работы по оформлению документа выполняется с помощью тегов разметки, Typst больше похож на язык программирования. В нём есть небольшое количество команд разметки, создающих основу содержимого вашего документа, а всё остальное осуществляется с помощью мощного языка сценариев, который вместе с разметкой и математическими формулами даёт очень большие возможности по созданию и стилизации содержимого документа.

Мы познакомимся с остальными составляющими Typst в следующих главах этой книги, а пока начнём с самой простой его части — языка разметки.

1.1. Три режима Typst

Прежде, чем мы приступим к обзору разметки, надо упомянуть о трёх синтаксических режимах, существующих в Typst, подробнее я расскажу о каждом из них позже:

- **режим разметки** — действует по умолчанию. Если же необходимо вставить какой-то текст в режиме разметки, находясь внутри другого режима, его следует заключить в квадратные скобки.
- **математический режим** — используется для ввода математических формул. Чтобы набрать участок документа в математическом режиме, его нужно заключить между двумя знаками доллара: `$...$`.
- **режим программного кода** — используется для ввода кода языка сценариев Typst, вызовов функций и т. п. Ввод выражения на языке сценариев начинается с символа решётки (#).

1.2. Абзацы текста

Если набрать в редакторе текст без какой-либо разметки, после компиляции он образует абзац обычного текста без какого-либо оформления. Текст, перенесённый на другую строку, не начинает новый абзац, а продолжает текущий. Для того, чтобы начать новый абзац, отделите его от предыдущего **пустой строкой**.

Пример разметки:

```
Это абзац.
Это продолжение того же абзаца, хотя оно
набрано в другой строке.

А вот это уже новый абзац.
```

Результат:

Это абзац. Это продолжение того же абзаца, хотя оно набрано в другой строке.

А вот это уже новый абзац.

Несколько пустых строк подряд воспринимаются как одна, и несколько пробелов подряд тоже воспринимаются как один пробел. Сравните исходный текст и результат:

```
Здесь          несколько пробелов


    И несколько пустых строк
```

Здесь несколько пробелов

И несколько пустых строк

1.3. Принудительный перенос строки

Typst сам переносит длинный текст в абзаце на новую строку. Но иногда бывает необходимо принудительно начать новую строку. Для этого вставьте в то место, где нужен разрыв, обратную косую черту `\` , за которой следует пробел:

```
Это первая строка,\ а это вторая строка, но абзац
продолжается

А это уже другой абзац
```

Это первая строка,
а это вторая строка, но абзац продолжается

А это уже другой абзац

1.4. Символы

Одно из главных преимуществ Typst, по сравнению с LaTeX, является лёгкость работы с символами в кодировке Unicode: в Typst эта кодировка поддерживается изначально. Существует несколько способов ввода символов, которые отсутствуют на клавиатуре.

Самый простой и очевидный способ — найти нужный символ в таблице символов, которые имеются в большинстве операционных систем, и вставить в документ через буфер обмена.

Множество различных символов Unicode можно также ввести благодаря специальным модулям Typst `sym` и `emoji` (подробнее см. с. 16).

Кроме того, любые символы Unicode можно вводить с помощью escape-последовательностей, которые имеют следующий вид: `\u{код}`, где код — это 16-ричный код символа в Unicode. Например, `\u{1f600}` — это символ смайлика 😀.

В режиме разметки есть также несколько комбинаций, которые преобразуются в специальные символы пунктуации:

В исходном файле	В готовом документе	Пояснение
`-`	–	Дефис или знак минуса. Дефис (вводимый клавишей «минус») автоматически преобразуется в минус, если после него идёт число. Сравните длину этих знаков: - и −7
`--`	–	Два дефиса преобразуются в короткое тире (en-dash). Оно применяется, например, для указания диапазона чисел и или дат: 1991–1995
`---`	—	Три дефиса преобразуются в длинное тире (em-dash). Именно его используют там, где по правилам русской грамматики ставится тире: применять дефис вместо тире считается грубой ошибкой
`...`	…	Три точки преобразуются в многоточие. Применять вместо символа многоточия просто три отдельных точки, как многие привыкли, тоже считается неправильным. Разницу между ними можно увидеть в столбцах слева, в некоторых шрифтах она особенно заметна.
`-?`	не отображается	Мягкий перенос слова. Вставьте его в место возможного переноса слова, если автоматический перенос вас почему-либо не устраивает
`~`	не отображается	Неразрывный пробел (nbsp). Используйте его вместо обычного пробелам там, где нужно запретить перенос на другую строку. Например, напишите `и~т.д.`, чтобы сокращение «и т.д.» оставалось целиком на одной строке.

1.5. Курсив

Чтобы в режиме разметки выделить текст курсивом, заключите его между двух символов подчёркивания.

```
Это _текст, набранный курсивом_
```

Это *текст, набранный курсивом*

Этот тег действует только для слов целиком, для нескольких букв внутри слова он не сработает. Более широкими возможностями выделения обладает функция emph, которую мы рассмотрим позже.

1.6. Жирный шрифт

Чтобы в режиме разметки выделить текст жирным шрифтом, заключите его между двумя звёздочками.

```
Это *жирный текст*
```

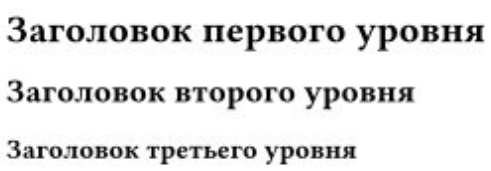
Это **жирный текст**

1.7. Заголовки

Чтобы превратить строку в заголовок, начните её с одного или нескольких знаков равенства:

```
= Заголовок первого уровня
== Заголовок второго уровня
=== Заголовок третьего уровня

и так далее
```



Не забудьте вставить пробел между знаками равенства и текстом заголовка. По умолчанию заголовки не нумеруются и имеют выравнивание влево.

1.8. Списки

В Typst есть разметка для набора нумерованных, ненумерованных списков и списка терминов. Выглядит она так:

Пример разметки:

```
+ первый элемент списка
+ второй элемент списка
+ третий элемент списка

- первый элемент списка
- второй элемент списка
- третий элемент списка

/ Термин 1: описание первого термина
/ Термин 2: описание второго термина
/ Термин 3: описание третьего термина
```

Результат:

1. первый элемент списка
2. второй элемент списка
3. третий элемент списка

- первый элемент списка
- второй элемент списка
- третий элемент списка

Термин 1 описание первого термина
Термин 2 описание второго термина
Термин 3 описание третьего термина

Списки могут быть иерархическими, управляется это отступами: если у строки сделан пробелами отступ больше, чем на ширину маркера элемента, то элемент автоматически становится вложенным. Нумерованные и ненумерованные списки можно вкладывать друг в друга в произвольном порядке.

Пример разметки

```
+ Первый элемент
   + вложенный элемент
   + и ещё один
+ Второй элемент

+ Нумерованный
   - ненумерованный
   - и ещё один
+ Опять нумерованный
```

Результат

1. Первый элемент
   1. вложенный элемент
   2. и ещё один
2. Второй элемент

1. Нумерованный
   - ненумерованный
   - и ещё один
2. Опять нумерованный

Если элементы списка в исходном файле отделять друг от друга пустыми строками, как абзацы, то в готовом документе отступ между элементами будет равен не межстрочному расстоянию, а расстоянию между абзацами, которое обычно немного шире:

1. Межстрочное
2. Межстрочное

1. Межабзацное

2. Межабзацное

1.9. Комментарии

Комментарии служат для пояснений внутри исходного текста, в готовый PDF они не выводятся. Чтобы набрать однострочный комментарий, поставьте перед ним две косые черты — `//`. Если необходимо закомментировать несколько строк, заключите их между символами `/*` и `*/`. Примеры:

```
//Это однострочный комментарий
/*А таким образом можно создать комментарий как из одной строки, так и из нескольких строк, как это показано здесь*/
```

1.10. Гиперссылки

Текст, начинающийся с `http://` или `https://`, автоматически превращается в гиперссылку: `https://typst.app` станет https://typst.app. По умолчанию гиперссылка не отличается от остального текста цветом или подчёркиванием; чтобы изменить её стиль, как в данном примере, надо будет воспользоваться языком сценариев, например, вставить где-нибудь в начале документа:

```
#show link: it => underline(text(fill: blue)[#it])
```

1.11. Неформатированный текст

Эта команда разметки предназначена для того, чтобы вставить текст, который будет отображён как он есть, даже если в нём встречаются выражения разметки или сценариев Typst. Разработчики Typst называют такой формат текста *raw*, то есть «сырой», неформатированный текст. Обычно его используют для вставки в документ примеров программного кода, но можно применять его и в каких-то других целях.

Для ввода такого текста заключите его между символами backtick (`), похожими на обратный апостроф (на большинстве клавиатур он находится на клавише с буквой Ё). Вот так:

```
`Этот текст будет отображён как он есть, хотя в нём есть *символы разметки*, формула $y=x^2$ и даже вызов функции #parbreak()`
```

Этот текст будет отображён как он есть, хотя в нём есть *символы разметки*, формула $y=x^2$ и даже вызов функции #parbreak()

Ещё один вариант — использовать не один, а три символа ` (или больше: главное, чтобы их число было нечётным, и количество символов перед и после текста было одинаковым). Разница вот в чём:

1. Текст с тремя символами backtick выводится не как строчный (inline) элемент, а как отдельный блок, если в нём есть хотя бы один перенос строки.
2. В тексте с тремя символами backtick после открывающих символов можно добавить идентификатор языка программирования и тогда команды языка будут синтаксически подсвечены. Например, для языка C нужно добавить «c» (без кавычек), для Rust — «rust», а для Typst — «typst» или «typ» для общей подсветки, «typc» для кода или «typm» для математического выражения . Если ваш текст начинается с чего-то похожего на идентификатор языка, но им не является, то первая буква может быть обрезана, в этом случае добавьте перед ней пробел.

```
```typst #set page(
binding: left,
margin: (inside: 2cm, outside: 2cm)
)```
```

```
#set page(
binding: left,
margin: (inside: 2cm, outside: 2cm)
)
```

Учтите, что внутри raw-текста невозможно использовать выделение с помощью _..._ или *...*, так как эти символы будут восприняты как есть, а не в качестве разметки.

1.12. Метки и перекрёстные ссылки

Чтобы создать **метку** на какой-либо элемент документа, введите её название, заключённое в угловые скобки: `<имя_метки>`. Она прикрепляется к ближайшему предшествующему объекту, который не является пробелом. У имени метки есть ограничения: в него можно включать только буквы, цифры и символы дефиса, нижнего подчёркивания, двоеточия и точки.

Например, в книге о домашних животных следующая разметка создаст метку с именем «catsdogs» для раздела «Кошки и собаки»:

```
= Кошки и собаки <catsdogs>
```
```

Чтобы впоследствии добавить в документ **перекрёстную ссылку** на раздел с меткой, вставьте в текст символ `@`, за которым следует имя метки. Например, для ссылки на раздел из предыдущего примера наберите команду разметки `@catsdogs`. Она отобразит ссылку в виде «Раздел 1», если активным языком является русский.

Если вас не устраивает стандартный текст перед номером в ссылке, например вы хотите, чтобы в ссылке было написано не «Раздел 1», а «Глава 1», то укажите нужный текст в квадратных скобках:

```
@catsdogs[Глава]
```

Перекрёстные метки работают только в том случае, если для объектов, на которые они ссылаются, включена нумерация. В противном случае компилятор выдаст соответствующую ошибку.

И ещё одна важная вещь: перекрёстные ссылки можно делать не на что угодно, а только на следующие объекты: *заголовки*, *иллюстрации* (figure) (в которые можно включать рисунки и таблицы), *формулы* и *сноски*.

Возникает резонный вопрос: если ссылаться можно не на всё, то зачем делать возможным создание меток для *любых* непробельных элементов документа? Ответ таков: на них тоже можно сослаться, но не с помощью описанной выше командой @, а с помощью «страничной» формы вызова функции перекрёстной ссылки `#ref(<метка>, form: "page")` (см. Раздел 4.17.2). Кроме того, на любые метки можно создавать гиперссылки функцией `link`. Подробнее об обеих этих функциях мы поговорим позже.

* * *

Вот, собственно и весь небогатый набор команд разметки Typst. Всю остальную работу по созданию и оформлению содержимого документа выполняют математический режим, используемый для вставки формул, и язык сценариев, к рассмотрению которого мы и перейдём в следующей главе.

Глава 2. Обзор языка сценариев Typst

В этой главе мы познакомимся с основами языка сценариев Typst. Я пишу «с основами», потому что сценарии — очень интересная и сложная тема, но для набора обычных документов большая часть информации о них вам не понадобится, так что мы рассмотрим их весьма поверхностно. За более подробной информацией обратитесь к официальной документации Typst.

2.1. Выражения

Как я уже говорил, текст, разметка и код в Typst тесно связаны. В процессе набора текста вам то и дело придётся вставлять в него выражения языка сценариев и наоборот. Для вставки в документ выражения на языке сценариев используется простой синтаксис: выражение начинается с символа решётки «#», а после того, как выражение завершено, продолжается обычная разметка. В некоторых случаях компилятор Typst может не понять, что выражение завершилось: тогда можно принудительно завершить его, поставив точку с запятой «;».

Если в каком-то месте требуется, чтобы эти два символа воспринимались как обычные символы, а не как специальные символы языка, можно поставить перед ними обратную косую черту: `\#` или `\;`.

2.2. Литералы

Литералом в программировании называют значение какого то типа, например, число или текст строки. Если добавить в исходный файл выражение в виде литерала, оно появится и в готовом документе.

`#5`	5
`#(-2)`	−2
`#"Это текст строки"`	Это текст строки

Как видите, кавычки, в которые заключён строковый литерал, не выводятся в документ. Отрицательное число взято в скобки, иначе компилятор воспримет его как оператор вычитания и выдаст ошибку, поскольку отсутствует второй операнд.

Если в исходном файле написать какую-нибудь операцию с литералами, то в документ в этом месте будет вставлен её результат.

`#(2+2)`	4

2.3. Переменные и связывание

Кроме литералов, в исходном тексте можно использовать переменные и выполнять с ними различные операции.

Имена переменных, как и другие идентификаторы (например, имена функций), могут содержать латинские буквы, цифры, дефисы и символы подчёркивания. Начинаться идентификатор должен с буквы или подчёркивания.

Для идентификаторов из двух или нескольких слов, с целью единообразия идентификаторов в во всех модулях и шаблонах,ы создатели Typst рекомендуют придерживаться так называемого стиля «Kebab case» — это когда слова записываются в нижнем регистре и разделяются дефисом, например: my-wonderful-identifier. Своё необычное название стиль получил из-за того, что подобные идентификаторы напомнили кому-то кебаб, нанизанный на вертел.

Но если вы не собираетесь делиться исходником своего документа с другими, то можете, разумеется, использовать любой удобный вам стиль именования.

Переменные в Typst объявляются с помощью ключевого слова **let**, которое создаёт переменную и присваивает ей значение, следующее за знаком равенства. Разработчики Typst называют такое выражение *связыванием* (binding). Например:

```
#let x = 10
```

Теперь, вставив в исходник документа выражение `#x`, в готовом документе мы увидим на этом месте число 10.

Явное указание типа переменной при связывании не требуется, компилятор создаёт переменную такого типа, который соответствует указанному после знака равенства значению. Если же возникнет двусмысленность: например, десятичная дробь может быть значением типов float или decimal, то преобразовать значение в нужный тип можно с помощью функции-конструктора данного типа (см. документацию Typst).

Присваивание значения не обязательно: если для переменной не задано никакого значения, то она инициализируется значением `none` (не путать с нулём!).

Если переменная объявлена внутри блока кода, то область её видимости ограничена этим блоком, если нет — то она действует от места объявления до конца файла.

Кстати, ключевое слово `let` позволяет создавать не только переменные, но и пользовательские функции, а также деструктурировать массивы (см. Раздел 3.1.4 и Раздел 3.3.2).

2.4. Операторы

В следующей таблице перечислены операторы языка с указанием их приоритета (чем выше, тем приоритетнее). Унарные операторы работают с одним операндом, а бинарные — с двумя.

Оператор	Действие	Число операндов	Приоритет
`-`	Смена знака	унарный	7
`+`	Никакого действия (введён для симметрии)	унарный	7
`*`	Умножение	бинарный	6
`/`	Деление	бинарный	6
`+`	Сложение	бинарный	5
`-`	Вычитание	бинарный	5
`==`	Проверка равенства	бинарный	4
`!=`	Проверка неравенства	бинарный	4
`<`	Меньше	бинарный	4
`<=`	Меньше или равно	бинарный	4
`>`	Больше	бинарный	4
`>=`	Больше или равно	бинарный	4
`in`	Проверка вхождения в коллекцию	бинарный	4
`not in`	Проверка отсутствия в коллекции	бинарный	4
`not`	Логическое НЕ	унарный	3
`and`	Логическое И	бинарный	3
`or`	Логическое ИЛИ	бинарный	2
`=`	Присваивание	бинарный	1
`+=`	Присваивание со сложением (инкремент)[1]	бинарный	1
`-=`	Присваивание с вычитанием (декремент)	бинарный	1
`*=`	Присваивание с умножением	бинарный	1
`/=`	Присваивание с делением	бинарный	1

[1]Выражение `n+=1` аналогично `n=n+1`, `n-=2` аналогично `n=n-2` и т.п.

2.5. Блоки кода и контента

- **Блок кода**:

```
#{let x = 1; x + 2}
```

При написании кода иногда возникает необходимость объединить несколько выражений в блок. Для этого их заключают в фигурные скобки. Отдельные выражения в нём разделяются либо переводом строки, либо точкой с запятой. Если символ решётки # поставлен перед блоком, то добавлять его каждым выражением в блоке не нужно.

Результаты отдельных выражений в блоке *сливаются* (join) друг с другом, давая одно результирующее значение блока. Но сливаться могут не любые типы данных, а только `array`, `dictionary`, `str` и `content`. Поэтому создать блок, например, из выражений, каждое из которых даёт тип `int`, не получится: компилятор выдаст сообщение об ошибке. Но некоторые выражения, например оператор связывания `let`, вообще не производят результата, точнее, дают результат `none`, поэтому их можно включать в блоки с любыми другими выражениями.

В следующих примерах два массива из одного элемента сливаются в массив из двух элементов, а результат конкатенации (+) двух строк сливается с третьей строкой:

`#{(1,) ; (2,)}`	`(1, 2)`
`#{"кот"+"о"; "пёс"}`	котопёс

- **Блок контента**:

```
[Этот текст с *разметкой* является контентом]
```

Контентом в Typst может быть всё, что входит в документ: текст с командами разметки и код, используемый для стилизации текста. Часто требуется использовать контент в каком-то выражении, либо передать его в некую функцию. В этом случае блок передаваемого контента заключается в квадратные скобки.

Блоки кода и контента могут включаться один в другой произвольным образом. Вот пример, в котором в блоке кода два блока контента [А] и [Б] сначала соединяются (concatenate) со строкой " и ", а потом результат сливается (join) с третьим блоком контента.

`#{`
`  [А]+" и "+[Б]`
`  [ сидели на трубе]`
`}`
	А и Б сидели на трубе

2.6. Пути

В некоторых случаях, например при вставке изображений, обращении к модулям или файлам с данными, требуется указать путь к файлу. Пути в Typst указываются в виде строки. Путь может быть:

- относительным — от местоположения файла, из которого производится обращение к изображению, например:

```
#image("images/flower.jpg")
```

- или абсолютным — от *корня* проекта, такой путь начинается с символа «/», например:

```
#image("/myproject/images/flower.jpg")
```

Корнем проекта (не путайте с корнем файловой системы!) считается каталог, содержащий главный .typ-файл проекта. По соображениям безопасности, у вас нет доступа на чтение файлов за пределами корневого каталога.

2.7. Условное выражение

Условное выражение позволяет вычислять или выводить в документ разные результаты в зависимости от выполнения условия. В Typst оно имеет вид, привычный по другим языкам программирования:

```
#if условие [Блок, выводимый, если true]
  else [Блок, выводимый, если false]
```

Вместо блоков контента, показанных в примере могут быть блоки выражений {..}. Также, при необходимости, секция else может отсутствовать, либо условные выражения могут быть вложенными одно в другое (`else if`). Примеры:

```
if условие {..}
if условие [..]
if условие [..] else {..}
if условие1 [..] else if условие2 {..} else [..]
```

2.8. Циклы

Typst поддерживает два вида циклов — **for** и **while**. Телом цикла, как и в случае условного оператора, могут быть как блок контента, так и блок выражений. В циклах можно использовать привычные по другим языкам программирования команды `break` и `continue`: первая прерывает цикл, а вторая пропускает идущую дальше часть тела цикла и переходит к следующей итерации.

2.8.1. Цикл for

Цикл `for` в Typst предназначен для итерации по объектам составных типов и имеет следующие формы:

- `for i in range(0, 20, step: 2) {..}` — цикл по последовательности значений от 0 до 20 с шагом 2.
- `for value in array {..}` — перебирает все элементы массива.
- `for pair in dict {..}` — перебирает пары `ключ: значение` в словарях. Существует также форма `for (key, value) in dict {..}`, разработчики утверждают, что она более эффективна, поскольку здесь нет необходимости в создании временного массива из всех пар.
- `for letter in "abc" {..}` — выполняет итерации по символам в строке.
- `for byte in bytes("😀") {..}` — выполняет итерации по последовательности байтов.

2.8.2. Цикл while

Этот цикл устроен так же, как и в других языках программирования: он продолжает выполняться, пока выполняется условие. Проиллюстрировать его можно следующим примером, где в каждой итерации выводится значение счётчика цикла:

```
#let i = 1;
#while i <= 10 {
  str(i)+" "
  i = i+1
}
```

1 2 3 4 5 6 7 8 9 10

2.9. Модули

Проекты Typst можно разбивать на отдельные файлы, которые называются **модулями** (module). Нужно помнить, что модуль может «видеть» только те переменные, функции и контент, которые содержатся в этом модуле.

Для обращения к содержимому других модулей предусмотрены следующие конструкции:

- **включение**: `#include "module1.typ"` — обрабатывает `module1.typ` и включает результирующий *контент* в текущий модуль.
- **импорт**: `#import "utils.typ"` — обрабатывает `utils.typ` и вставляет *набор функций и переменных* из него как `utils` (без расширения файла). Затем можно будет обращаться к функциям и переменным из модуля с помощью синтаксиса с точкой: `#utils.add(2, 5)`.
  При желании можно, воспользовавшись ключевым словом `as`, переименовать импортированный набор: `#import "utils.typ" as xyz`, и обращаться к функциям уже как `#xyz.add(2, 5)`.
- **импорт отдельных элементов**: `import "utils.typ": a, b` — в данном случае из модуля `utils.typ` импортируются две переменных `a` и `b` и объявляются в текущем модуле. Указание вместо имён переменных звёздочки `*` импортирует из модуля *все* переменные. Можно также с помощью ключевого слова `as` переименовать импортированные переменные:
  `#import "utils.typ": a as x1, b as x2.`

В Typst есть несколько встроенных модулей:

- calc для выполнения вычислений и различных операций с числовыми значениями,
- std, который содержит различные глобально доступные элементы,
- sym и emoji, которые используются для вставки в документ различных символов и эмодзи,
- sys для взаимодействия с системой, например получения текущей версии компилятора и чтения параметров командной строки.

Подробно рассказывать о них я не буду, при необходимости читайте официальную документацию.

2.10. Шаблоны

Сделав один раз удачные настройки для своего документа, вы можете пожелать в будущем использовать такие же и для других документов. Чтобы не копировать каждый раз все эти настройки из документа в документ, их можно оформить в виде *шаблона*, содержащего переменные и пользовательские функции для выполнения этих настроек. Шаблон, как правило, выполняется в виде отдельного модуля, или даже нескольких, которые затем можно импортировать в текущий документ.

В этой книге я не буду подробно описывать создание шаблонов, поскольку для этого требуется достаточно хорошее владение языком сценариев Typst.

В разделе сайта Typst Universe вы можете найти множество уже готовых шаблонов для документов самой разной тематики. Например, там есть шаблон для оформления курсовых и дипломных работ по ГОСТ 7.32-2017 и даже для написания русской кандидатской диссертации.

2.11. Пакеты

Для повторного использования одинаковых «строительных блоков» Typst (например, модулей и шаблонов) в нескольких проектах можно создавать и затем импортировать **пакеты**. На сайте Typst доступно множество готовых пакетов, созданных сообществом. Импорт пакета в ваш проект выполняется, например, так:

```
#import "@preview/equate:0.3.2"
```

Эта команда импортирует пакет equate версии 0.3.2 из пространства имён `preview`, содержащего пакеты, созданные сообществом. Более точный синтаксис команды и подробное описание нужного пакета ищите в разделе сайта Typst Universe.

Перед тем, как воспользоваться тем или иным пакетом, не забудьте ознакомиться с лицензией на его использование: ведь может оказаться, что она несовместима с целью вашего проекта. Большинство пакетов сообщества доступно под лицензией MIT.

Существует также возможность создавать собственные пакеты и даже собственное хранилище пакетов на вашем компьютере, но эти темы выходят за рамки данной книги: обратитесь к документации Typst.

Глава 3. Типы данных

Эту информацию следовало бы включить в предыдущую главу, но в Typst так много типов данных, что я посчитал необходимым посвятить им отдельную главу. Хотя подробно рассматривать типы данных я не стану, поверхностно ознакомиться с ними всё же стоит, хотя бы для того, чтобы лучше понимать, как именно правильно передавать нужные значения в функции, занимающиеся стилизацией документа.

Все типы данных Typst можно условно разделить на:

- базовые типы: `int`, `float`, `decimal`, `bool` и `symbol`, к ним же можно отнести особые типы `auto` и `none`;
- производные типы, представляющие последовательности из базовых типов: `str`, `bytes`, `array` и `dictionary`;
- функции (`func`), с помощью которых выполняются многие действия с переменными и элементами документа;
- и все остальные типы, которые можно назвать специализированными, например типы, отвечающие за длину, цвет, угол наклона, выравнивание и т.п. Особо среди них выделяется тип `content`, который представляет собой содержимое некоторой части документа.

3.1. Функции

Функции — пожалуй, одна из самых важных составляющих Typst. Они используются практически везде: при инициализации и обработке переменных, для загрузки файлов данных, работы со счётчиками и запросами, для изменения внешнего вида всех элементов документа. Сейчас я открою вам страшную тайну. Готовы? Даже за описанной в Главе 1 разметкой Typst на самом деле скрываются функции! Так что без команд разметки вполне можно было бы обойтись, они добавлены разработчиками просто для удобства.

Функции в Typst устроены так же, как и в других языках программирования. Для их вызова указывается имя функции, за которым в скобках следует ряд передаваемых ей *аргументов*[1], например

```
#myfunction(a, b, c)
```

Для передачи функциям аргумента с типом content, который может быть очень большим, разработчики ввели специальный способ — контент указывается после скобок со списком аргументов в виде блоков контента, то есть в квадратных скобках. Например, если бы у нашей воображаемой функции из примера выше параметры b и c были типа content, то можно было бы написать:

```
#myfunction(a)[b][c]
```

3.1.1. Синтаксис функций. Позиционные и именованные параметры

При описании функции в этой книге, как и в документации Typst, я буду указывать её синтаксис. Вот, например, как выглядит синтаксис функции `strong`, выводящий текст жирным шрифтом:

Синтаксис:

```
strong(
    delta: int,
    content,
) -> content
```

Как мы видим, у этой функции два параметра: один из них (`delta`) — *именованный*, а другой (без имени) — *позиционный*[2]. У позиционных параметров при вызове функции вместо имён параметров сразу подставляются значения в порядке следования этих параметров, а у именованного указывается имя параметра, за которым через двоеточие указывается значение.

Таки образом, вызов этой функции мог бы выглядеть так:

[1] Входные переменные функции обычно называют её *параметрами*, а реально переданные ей значения для этих параметров — *аргументами*

[2] На самом деле имя у него есть, просто авторы документации Typst решили не указывать в синтаксисе функций имена позиционных параметров: указывать только тип, а имя давать в пояснении. Мне это кажется не очень удобочитаемым, но для единообразия с документацией буду придерживаться такого же правила.

```
#strong(delta:500, "Какой-то текст")
```

или вот так (см. на с. 13 об аргументах типа `content`):

```
#strong(delta:500)[Какой-то текст]
```

В конце после стрелки `->` в синтаксисе указан тип возвращаемого функцией значения, в данном случае она возвращает `content`.

В описании некоторых элементов, например, таблиц, вам встретятся функции с *переменным количеством аргументов*, используемые, когда разработчики не знают заранее, сколько именно аргументов вы передадите функции, например, сколько ячеек будет в таблице. Реализовано это за счёт *стока аргументов* (sink), который записывается в синтаксисе функции виде двух точек (. .). В этом случае все «лишние» аргументы попадают в сток, откуда затем функция может их извлечь и обработать.

3.1.2. Конструкторы

У многих типов данных имеется специальная функция, которая называется *конструктором*. В Typst её название совпадает с названием типа и, как правило, она используется для создания экземпляра этого типа путём преобразования из других типов данных.

Например вызов конструктора `#str(-5)` преобразует в строку целочисленное значение −5.

3.1.3. Встроенные функции или методы типов

Большинство типов данных в Typst имеют специальные встроенные функции, аналогичные *функциям-членам* или *методам* классов в объектно-ориентированных языках программирования. Обращение к таким методам выполняется особым образом: после экземпляра данного типа (будь то литерал или переменная) ставится точка и после неё указывается функция-метод, например в выражении `#(-3).signum()` метод signum возвращает знак целочисленного литерала −3.

Поскольку функции в Typst тоже являются типом данных, у них самих имеются два метода, перечисленных ниже:

- **with** — возвращает новую функцию, в которой для части параметров уже подставлены какие-либо аргументы.
- **where** — более полезный для нас метод, который создаёт *селектор*, отфильтровывающий элементы, поля которых совпадают со значениями указанных аргументов. Подробнее о применении селекторов поговорим, когда будем рассматривать правило стилизации `show` (см. Раздел 4.1.2). Вот пример, в котором с помощью функции `where` отбираются заголовки второго уровня и их текст делается синим:

```
#show heading.where(level: 2): set text(blue)
```

3.1.4. Пользовательские функции

При необходимости вы можете создать собственную функцию. Это не так уж сложно, но я не буду подробно описывать этот процесс, вы можете найти детали в документации Typst. Просто опишу в общих чертах.

Пользовательская функция создаётся с помощью ключевого слова **let** за которым следует имя функции и параметры в скобках и, после знака равенства, блок кода с телом функции, в котором выполняются какие-то действия.

Например, создадим функцию, которая выводит передаваемый ей контент в боксе со светло-серым фоном.

```
#let graybox(body) = {
box(fill: silver, outset:5pt)[#body]
}
```

Теперь можно вставить в документ вызов функции

```
#graybox[Пробный текст]
```

и получить результат: Пробный текст

Можно также создавать **анонимные функции** (unnamed function), которые отличаются тем, что у них нет имени функции, и они выполняются непосредственно в месте их объявления. Такие функции иногда используются в правилах show. Выглядят они немного странно, так что если увидите в примере правила где нибудь в документации конструкцию со стрелкой наподобие

```
#show heading: it => {
  set align(center)
  block(smallcaps(it.body))
},
```

не удивляйтесь, это и есть анонимная функция.

3.2. Базовые типы данных

3.2.1. Целые числа (int)

Тип **int** представляет собой 64-битные целочисленные значения. Целочисленные литералы можно задать не только в виде десятичных чисел, но и в виде шестнадцатеричных (начинаются с `0x`), восьмеричных (`0o`) и двоичных (`0b`) значений:

```
#let hexvalue = 0xff
#let octvalue = 0o10
#let binvalue = 0b1001
```

У типа `int` есть функция-конструктор, которая при необходимости преобразует в целочисленный тип значение другого типа, а также несколько методов. Смотрите их описание в документации Typst.

3.2.2. Числа с плавающей точкой (float)

Это действительные 64-битные числа с плавающей десятичной точкой. Они поддерживают значения NaN (не-число) и бесконечность, которые можно задать как `float.nan` и `float.inf`, соответственно.

Вычисления с этим типом имеют ограниченную точность, поэтому нежелательно использовать их там, где требуются вычисления с повышенной точностью, например, связанные с финансовой сферой. Так, результатом выражения `0.1 + 0.2` будет не точно `0.3`, как ожидается, а `0.30000000000000004`.

Там где, функции требуется параметр типа `float`, можно передать ей `int`, он будет воспринят как `float` с нулевой дробной частью.

У типа `float` есть конструктор, преобразующий в него другие типы данных и несколько методов.

3.2.3. Числа с фиксированной точкой (decimal)

Этот редко используемый тип следует применять для вычислений, где требуется повышенная точность. В отличие от `float` он не поддерживает бесконечность и NaN.

Обратите внимание, что базовые арифметические операции поддерживаются между двумя значениями типа `decimal` или между `decimal` и `int`. Операции между `decimal` и `float` не поддерживаются, чтобы избежать потери точности: компилятор в этом случае выдаёт ошибку.

У типа `decimal` есть конструктор для преобразования в него других типов данных. Методов у него нет.

3.2.4. Логический тип (bool)

Логический тип, как и в других языках программирования, может иметь два состояния: `true` (истина) и `false` (ложь). Конструктора и методов не имеет.

3.2.5. Тип auto

Особый тип данных, у которого может быть только одно значение — `auto`. Оно указывает, что нужно использовать то значение по умолчанию, которое задано для функции в данном контексте.

Например если указать для параметра `dir` (направление текста) элемента `text` значение `auto`, то направление будет автоматически выбираться в зависимости от активного языка документа.

Конструктора и методов не имеет

3.2.6. Тип none

Для этого типа данных предусмотрено тоже только одно значение — `none`. Это значение используется для того, чтобы указать отсутствие у параметра функции какого-либо значения, даже нулевого.

Конструктора и методов не имеет.

3.2.7. Символ (symbol)

Представляет собой символ в кодировке Unicode. Для ввода символов, отсутствующих на клавиатуре, в Typst есть два модуля: **sym** для ввода «общих» символов и **emoji** для ввода символов-эмодзи. Обращаться к ним нужно через точку: `#sym.имя_символа` или `#emoji.имя_символа`.

Например, `#sym.alpha` вставит греческую букву альфа (α), а `#emoji.bee` — изображение пчелы (🐝). Полный список доступных символов и их имён слишком большой, чтобы приводить его в этой книге, поэтому чтобы найти имена нужных вам символов, обратитесь к документации на сайте Typst.

Если вы часто пользуетесь каким-то эмодзи, можете *импортировать* его из модуля в документ, например `#import emoji: face`, и в дальнейшем обращаться к нему без имени модуля — просто как `#face`.

У некоторых символов есть по несколько вариантов, добавляемых с помощью указания модификаторов через точку. Например: просто `#emoji.flag` возвращает чёрный флажок 🏴, `#emoji.flag.white` — белый 🏳, а `#emoji.flag.red` — красный 🚩

У типа `symbol` есть конструктор, но не для преобразования типов, а для создания пользовательских символов с модификатором. Методов он не имеет.

3.3. Производные типы данных

Это типы данных, состоящие из наборов базовых типов, описанных выше.

3.3.1. Строка (str)

Строки представляют собой последовательность символов в кодировке Unicode. Строки можно соединять вместе оператором +, а также осуществлять их слияние (см. Раздел 2.5) и умножать на целые числа (результатом будет повтор строки указанное число раз), например:

`#{"Hello, " + "Dolly!"}` даст `"Hello, Dolly!"`,
а написав `#{"Ура! "*3}` получим троекратное `"Ура! Ура! Ура!"`

У цикла **for** (см. с. 11) существует специальный вариант для итерации по всем символам строки:

```
#for letter in "строка" {..}
```

В строках можно использовать несколько escape-последовательностей для вставки символов, которые нельзя указать в строке другим способом:

`\\`	для вставки обратной косой черты (просто вставить \ нельзя, так как она воспринимается, как принудительный разрыв строки)
`\"`	для вставки кавычки
`\n`	для вставки перевода строки
`\r`	для вставки возврата каретки
`\t`	для вставки табуляции
`\u{код}`	для вставки символа Unicode по его шестнадцатеричному коду

У типа `str` имеется конструктор для преобразования в строку других типов данных и довольно большое количество методов.

3.3.2. Массив (array)

Массивы представляют собой последовательность значений, причём значения не обязательно должны относиться к одному типу. Создать массив можно, заключив в круглые скобки последовательность значений через запятую: `#let values = (1, 7, 4, -3, 2)`.

Пустой массив записывается, как `()`. В массиве единичной длины нужно добавить после значения запятую `(1,)`, иначе компилятор может спутать его с выражением в скобках.

К элементу массива можно обратиться по индексу с помощью метода `.at()`. Индекс начинается с нуля. Если указан отрицательный индекс, то он отсчитывается не от начала, а от конца массива.

Массивы, как и строки, можно соединять оператором +, осуществлять их слияние (см. Раздел 2.5) и умножать на целые числа.

У типа `array` есть конструктор, но он не создаёт массив из отдельных элементов, а только преобразует в массив уже существующие наборы значений одного или разных типов. Также имеется довольно большое количество методов для работы с массивом и его отдельными элементами.

В Главе 2 я упоминал, что оператор let, позволяет *деструктурировать* массивы. Выглядит это так:

```
#let (a, .., b)=(1, 2, 3, 4)
```

Здесь `a` получит значение первого элемента массива (1), `b` — последнего (4), а оператор `..` соберёт значения всех оставшихся элементов. Деструктурировать можно не только массивы, но и словари, а также списки аргументов функций, но подробно останавливаться на этом не будем.

3.3.3. Последовательность байтов (bytes)

Концептуально подобна массиву целочисленных значений от 0 до 255. Этот тип используется в основном для работы с большими буферами «сырых» данных, таких как данные изображений. Данные можно загружать из файлов в виде `bytes` с помощью функции **read**.

Существует специальная форма цикла **for** для итерации по последовательности байтов:

```
for byte in bytes("😀") {..}
```

У этого типа есть конструктор, преобразующий в него строку или массив целочисленных значений, а также пара методов для доступа к содержимому.

3.3.4. Словарь (dictionary)

Этот тип данных представляет собой набор данных ключ-значение. В некотором смысле он похож на массив, в котором в роли индекса выступают не целые числа, а строки.

Чтобы создать значение типа dictionary, нужно перечислить в круглых скобках пары `ключ: значение` через запятую, например:

```
#let dict = (
  name: "Иванов И.И.",
  born: 1964,
)
```

Для создания пустого словаря предусмотрена конструкция `(:)`, поскольку просто пустые круглые скобки уже зарезервированы для создания пустых массивов.

Получить значение элемента словаря можно с помощью синтаксиса с точкой: например `#dict.name` возвращает `"Иванов И.И."`, а `#dict.born` — 1964. Существуют специальные формы цикла **for** (см. с. 11) для итерации по элементам массива:

```
for pair in dict {..}
for (key, value) in dict {..}
```

У типа `dictionary` есть конструктор для преобразования в него значения, подобного словарю, и несколько методов.

3.4. Контент (content)

Это очень важный тип в Typst. Он представляет собой часть содержимого документа, включая текст, разметку и вызовы функций, используемых для стилизации. То есть можно считать, что это просто кусок вашего документа. Чтобы создать значение этого типа, достаточно заключить текст с разметкой в квадратные скобки: `[Пример *контента*]`. Именно таким образом контент передаётся в функции.

Значения типа `content` можно, как и строки, соединять друг с другом оператором `+`, осуществлять их слияние (join), а также умножать на целые числа. Там, где функция ожидает параметр типа `content`, можно при необходимости передать ей строку или значение `none`.

3.5. Типы для работы с датой и временем

3.5.1. Дата/время (datetime)

Служит для хранения даты, времени или их комбинации. Может быть создан указанием конкретной даты/времени в функции-конструкторе, либо получением текущей даты с помощью метода `datetime.today`.

У типа `datetime` есть функция-конструктор для создания нового значения типа `datetime`, который выглядит следующим образом:

Синтаксис:

```
datetime(
year: int,
month: int,
day: int,
hour: int,
minute: int,
second: int,
)
```

Смысл параметров понятен из их названий. Можно указывать не все параметры: в зависимости от того, какие указаны, Typst хранит значение следующим образом:
- если указаны год, месяц и число, значение будет просто датой;
- если указаны часы, минуты и секунды, значение будет просто временем;
- если указаны все параметры, то это будет полное значение дата/время.

Например:
```
#let d=datetime(year: 2026, month: 04, day: 26)
```

Также имеется несколько методов, возвращающих значения отдельных полей даты/времени.

3.5.2. Длительность (duration)

Тип данных для представления длительности чего-либо в неделях, днях, часах, минутах и секундах. Можно получить длительность, вычтя из одного значения datetime другое.

Честно говоря, я не особо представляю, зачем может понадобиться длительность при вёрстке документа. Скорее всего, она может пригодиться при создании модулей и шаблонов.

У этого типа тоже есть конструктор, создающий новую длительность и несколько методов, возвращающих длительность, выраженную в секундах, минутах, часах и т.д.

3.6. Метки (label)

Мы уже рассматривали метки в разделе о разметке Typst (с. 6), добавлю лишь небольшие подробности.

Разработчики Typst по каким-то своим соображениям отнесли метки не к элементам, а к типам данных. У них даже есть конструктор, создающий метку из строки:

Синтаксис:

```
label(name: str),
```

где `name` — строка, представляющая собой имя метки.

Но нужно учесть одну тонкость: если, как обычно, добавить после заголовка раздела *в той же строке* метку, созданную с помощью конструктора, то она будет ссылаться не на сам заголовок, как в случае «разметочного» синтаксиса, а на *текст* заголовка, то есть получится то же самое, как если бы вы написали `#heading[Текст заголовка #label("lab")]`. Поэтому при попытке сделать перекрёстную ссылку на эту метку с помощь `#ref` вы получите ошибку «cannot ref to text» (нельзя делать перекрёстную ссылку на текст).

3.7. Размеры и углы

3.7.1. Абсолютный размер (length)

Тип данных для указания размеров какого-либо элемента. Поддерживаются следующие единицы измерения (обратите внимание на отсутствие пробела между числом и единицей измерения):
- пункты: 72pt
- миллиметры: 254mm
- сантиметры: 2.54cm
- дюймы: 1in
- производные от размера шрифта[1]: 2.5em

3.7.2. Доля (ratio)

Это доля какого-либо значения, записываемая как число, за которым следует знак процента, например `25%`. Доли часто используются как часть относительного размера (см. Раздел 3.7.3) для указания размера элемента относительно размера страницы или какого-то контейнера. При отображении в документе значения типа `ratio` для лучшей читаемости округляются до двух знаков после запятой.

[1] Размер `1em` равен ширине буквы M в используемом шрифте.

3.7.3. Относительный размер (relative)

Это тоже размер, но являющийся комбинацией типов `length` и `ratio` (см. выше). Он записывается в виде `length+ratio` или `length-ratio`. Например, в следующем примере ширина отображаемого прямоугольника равна четверти ширины области содержимого страницы плюс ещё 1 см:

```
#rect(width: 25% + 1cm)
```

Не обязательно всегда указывать оба компонента: вместо аргумента типа `relative` можно при необходимости указывать либо только длину (в данном случае 1cm), либо только долю (25%).

Значения типа `relative` можно умножать на целые числа, числа с плавающей точкой и пропорции.

А вот как можно выделить из `relative` его отдельные компоненты:

```
#(25%+10pt).length            10pt            #(25%+10pt).ratio            25%
```

3.7.4. Дробный размер (fraction)

Указывает, как распределяется свободное место между элементами: каждый из элементов получает количества места, пропорциональному отношению дробного размера этого элемента к сумме дробных размеров всех элементов. Например, запись

```
Левый #h(1fr) Средний #h(2fr) Правый
```

распределит три элемента по ширине страницы следующим образом: расстояние между левым и средним элементом будет 1/3 от оставшегося свободного места, а между средним и правым — 2/3.

Левый Средний Правый

Дробный размер может указываться в параметрах функций h и v, а также в таблицах. Например, именно с помощью вертикальных отступов дробного размера сделан титульный лист этой книги.

3.7.5. Угол (angle)

Углы в Typst могут быть заданы в градусах (`180deg`) или в радианах (`3.14rad`). У этого типа данных есть два метода для преобразования между этими единицами:.

- `#60deg.rad()` преобразует градусы в радианы,
- `#1rad.deg()` преобразует радианы в градусы.

3.8. Выравнивание (alignment)

Может принимать значения

- `start`: в начале относительно направления текста (т.е. влево при письме слева направо),
- `end`: в конце относительно направления текста,
- `left`: влево,
- `center`: по центру (горизонтально),
- `right`: вправо,
- `top`: вертикальное выравнивание вверх,
- `horizon`: вертикальное выравнивание по центру,
- `bottom`: вертикальное выравнивание вниз.

Чтобы выровнять сразу по горизонтали и вертикали, значения можно складывать, например `center+bottom`.

3.9. Направление (direction)

Используется, например для указания направления письма в элементе `text` или направления расположения элементов стека (Раздел 4.14).

У этого тип всего четыре значения:

- `ltr` — слева направо,
- `rtl` — справа налево,
- `ttb` — сверху вниз,
- `btt` — снизу вверх.