Joaquín Ataz-López

Не слишком короткое

введение

в ConTEXt Mark IV

Не слишком короткое введение в ConTEXt Mark IV
Version 1.6 [2 января 2021]

Original title: Una introducción (no demasiado breve) a ConTEXt Mark IV
English Translation: A good friend who wishes to remain anonymous.

Version history:

- 18 августа 2020: Version 1.0 (Spanish only): Original document.
- 23 августа 2020: Version 1.1 (Spanish only): Correction of minor errors, typos and misunderstandings by the author.
- 3 сентября 2020: Version 1.15 (Spanish only): More errors, typos and misunderstandings.
- 5 сентября 2020: Version 1.16 (Spanish only): More errors, typos and misunderstandings as well as some very minor changes to make the text clearer (I hope).
- 6 сентября 2020: Version 1.17 (Spanish only): The number of minor errors I am finding is incredible. I would just need to stop re-reading the document to find no more!
- 21 октября 2020: Version 1.5 (Spanish only): Introduction of suggestions and correction of errors reported by NTG-context users.
- 2 января 2021: Version 1.6: Corrections suggested after a new and careful reading of the document, on the occasion of its translation to English. This is the first version in English.

Оглавление

Предисловие*

Внимательный читатель, это документ о ConTEXt, системе набора текста, производной от TEX, который, в свою очередь, является ещё одной системой набора, созданной в период с 1977 по 1982 год Дональдом Э. Кнутом в Стэнфордском университете.

ConTEXt был разработан для создания документов очень высокого типографского качества – бумажных документов или документов, предназначенных для отображения на экране вычислительного устройства. Это не текстовый процессор или текстовый редактор, а, как я уже сказал, *система*, или, другими словами, набор инструментов, предназначенных для набора документов, понимаемых как графический макет и визуализация различных элементов документа на странице или на экране. Таким образом, ConTEXt стремится предоставить все инструменты, необходимые для придания документам наилучшего внешнего вида. Идея состоит в том, чтобы иметь возможность создавать документы, которые не только хорошо написаны, но и выглядят «красивыми». В этом отношении мы можем упомянуть здесь то, что написал Donald E. Knuth, представляя TEX (систему, на которой основана ConTEXt):

> *Если вы просто хотите создать достаточно хороший документ - что-то приемлемое и в основном читаемое, но не очень красивое - обычно будет достаточно более простой системы. Цель TEX - добиться* лучшего *качества; это требует большего внимания к деталям, но вам не составит труда преодолеть дополнительную дистанцию, и вы сможете гордиться готовым продуктом.*

Когда мы готовим рукопись с помощью ConTEXt, мы указываем, как именно она должна быть преобразована в страницы (или экраны), типографское качество которых сравнимо с тем, что можно получить в лучших типографиях мира. Для этого, как только мы изучим систему, нам потребуется немного больше работы, чем требуется для обычного набора текста в любом текстовом процессоре или текстовом редакторе. Фактически, как только мы добились определенной легкости в обращении с ConTEXt, наша общая работа, вероятно, будет меньше, если мы будем иметь в виду, что основные детали форматирования документа описаны глобально в ConTEXt и мы работаем с текстовыми файлами, которые являются – как только мы к ним привыкнем – гораздо более естественным способом создания и редактирования документов; кроме того факта, что такие файлы намного легче и легче в обращении, чем тяжелые двоичные файлы, принадлежащие текстовым процессорам.

По ConTEXt имеется значительный объем документации, почти вся она на английском языке. То, что мы могли бы считать *официальным* дистрибутивом ConTEXt – называлось «ConTEXt Standalone» [2] – например, содержит около 180 файлов PDF с документацией (большая часть на английском языке, но некоторые на голландском и немецком языках), включая руководства, примеры и технические

---

* Это предисловие было задумано как перевод / адаптации к ConTEXt предисловия к „The TEXBook”, документу, который объясняет *всё, что вам нужно знать о TEX*. В конце концов, мне пришлось отклониться от этого; однако я сохранил некоторые фрагменты, которые, надеюсь, для тех, кто знает это, предложат некоторые *отголоски* этого.

[2] В то время, когда была составлена первая версия этого текста, то, что там говорилось, было фактическим; но весной 2020 года вики ConTeXt была обновлена, и с тех пор мы должны предполагать, что «официальный» дистрибутив ConTEXt стал LMTX. Тем не менее, для тех, кто впервые приходит в мир ConTEXt, я все же рекомендую использовать «ConTEXt Standalone», поскольку это более стабильный дистрибутив. Appendix ?? объясняется, как установить любой из дистрибутивов.

статьи; а в сети Pragma ADE (компания, которая родила ConTEXt) есть (в день, когда я проводил подсчет в мае 2020 года) 224 загружаемых документа, большинство из которых распространяется с «ConTEXt Standalone», но также и некоторые другие. Тем не менее, эта огромная документация не особенно полезна для изучения ConTEXt, поскольку, как правило, эти документы не предназначены для читателя, который ничего не знает о системе, но хочет ее изучить. Из 56 файлов PDF, которые «ConTEXt Standalone» вызывают «manuals», только один предполагает, что читатель ничего не знает о ConTEXt. Это документ под названием «ConTEXt Mark IV, Excursion». Однако этот документ, как следует из его названия, ограничивается представлением системы и объяснением того, как делать определенные вещи, которые могут быть выполнены с помощью ConTEXt.

Было бы хорошим введением, если бы за ним последовало несколько более структурированное и систематизированное справочное руководство. Этого руководства не существует, а разрыв между документом, относящимся к ConTEXt «Excursion», и остальной документацией слишком велик.

В 2001 году было написано справочное руководство, которое можно найти на Pragma ADE web site; но, несмотря на это название, с одной стороны, оно не было разработано как *полное руководство*, а с другой стороны, это был (есть) текст, предназначенный для предыдущей версии CONTEXT (называемой Mark II), и поэтому он полностью устарел.

В 2013 году руководство было частично обновлено, но многие его разделы не были переписаны, и оно содержит информацию, относящуюся как к ConTEXt Mark II, так и к ConTEXt Mark IV (текущая версия), но при этом не всегда четко указывается, какая информация относится к каждой из версий. Возможно, по этой причине этого руководства нет среди документов, включенных в «ConTEXt Standalone». Тем не менее, несмотря на эти недостатки, руководство по-прежнему остается лучшим документом для начала изучения ConTEXt после того, как мы прочитали вводную «ConTEXt Mark IV, Excursion». Также очень полезной для начала в ConTEXt является информация, которую можно найти в его wiki, которая на момент написания этого документа была переработана и имеет гораздо более четкую структуру, хотя там также смешиваются объяснения, работающие только в Mark II, с другими для Mark IV или для обеих версий. Это отсутствие различий также можно найти в официальном списке команд ConTEXt [1], В котором не указано, какие команды работают только в одной из двух версий.

По сути, это введение было написано на основе четырех источников информации, перечисленных здесь: ConTEXt «Excursion», руководство 2013 года, содержание вики и официальный список команд, который включает для каждого из них: допустимые варианты конфигурации; Кроме того, конечно же, собственные тесты и выводы. Итак, на самом деле это введение является результатом следственных усилий, и в течение некоторого времени у меня возникало искушение назвать его «Что я знаю о ConTEXt Mark IV» или «Что я узнал о ConTEXt Mark IV». В конце концов, я отказался от этих заголовков, потому что, какими бы правдивыми они ни были, я чувствовал, что они могут отговорить кого-то от входа в ConTEXt; и что можно сказать наверняка, так это то, что хотя документация имеет (на мой взгляд) некоторые недостатки, здесь у нас есть действительно полезный и универсальный инструмент, для которого усилия, необходимые для его изучения, несомненно, того стоят. Используя ConTEXt мы можем управлять текстовыми документами и настраивать их для достижения вещей, которые тот, кто не знаком с системой, просто не может даже вообразить.

> Из-за того, кем я являюсь, я не могу избавиться от того факта, что мои жалобы на нехватку информации время от времени будут появляться в этом документе. Я не хотел бы, чтобы меня неправильно поняли: я безмерно благодарен создателям ConTEXt за разработку такого мощного инструмента и за то, что они сделали его доступным для широкой публики. Просто я не могу не думать, что этот инструмент был бы намного более популярным, если бы его документация была улучшена: нужно потратить много времени на его изучение, не столько из-за его внутренней сложности (которая у него есть, но не больше чем другие подобные инструменты - наоборот), но из-за отсутствия четкой, полной и хорошо организованной информации, которая различает две версии ConTEXt, объясняет функции в каждой из них и, прежде всего, разъясняет что делает каждая команда, аргумент и опция.
>
> Верно, что такая информация требует значительных временных затрат. Но учитывая, что многие команды имеют общие параметры с похожими именами, возможно, можно было бы предоставить своего рода *глоссарий* параметров, который также

[1] Список см. в section ??.

помог бы обнаружить некоторые несоответствия, возникающие в результате того, когда два параметра с одинаковым именем выполняют разные действия или когда делать то же самое, в разных командах используются имена разных опций.

Что касается читателя, который впервые приближается к ConTEXt пусть мои жалобы не разубеждают вас, потому что, хотя может быть правдой, что недостаточная информация увеличивает время, необходимое для её изучения, по крайней мере, или материал, рассматриваемый в этом введении, я уже потратил это время, чтобы читателю не пришлось этого делать. И только благодаря тому, что можно узнать из этого введения, читатели получат в свое распоряжение инструмент, который позволит им создавать документы с легкостью, о которой они и не подозревали.

Поскольку то, что объясняется в этом документе, во многом основано на моих собственных выводах, вполне вероятно, что, хотя я лично проверил большую часть того, что объясняю, некоторые утверждения или мнения могут быть не правильными и не очень ортодоксальными. Я, конечно, буду признателен за любые исправления, нюансы или пояснения, которые читатели могут мне предложить, и их можно отправить по адресу joaquin@ataz.org. Однако, чтобы уменьшить количество случаев, когда я могу ошибаться, я старался не вдаваться в подробности, о которых я не нашел никакой информации и что я не имел возможности (или не хотел) лично опробовать. Иногда это происходит потому, что результаты моих тестов не были окончательными, а иногда потому, что я не всегда тестировал все: количество команд и опций, которые имеет ConTEXt впечатляет, и если бы мне пришлось все попробовать, я никогда бы не закончил это введение. Однако бывают случаи, когда я не могу избежать *предположения*, т.е. сделать заявление, которое я считаю вероятным, но в котором я не совсем уверен. В этих случаях изображение „conjecture” было помещено в левое поле абзаца, в котором я делаю такое предположение. Изображение призвано графически представить предположение. [1] В других случаях у меня нет другого выбора, кроме как признать, что я чего-то не знаю и у меня даже нет разумного предположения об этом: в этом случае изображение, видимое для слева на полях означает нечто большее, чем просто предположение или незнание. [2] Но поскольку я никогда не был очень хорош с графическими изображениями, я не уверен, что выбранные мной изображения действительно подходят передать столько нюансов.

Это введение, с другой стороны, было написано с точки зрения читателя, который ничего не знает ни о TEX, ни о ConTEXt, хотя я надеюсь, что оно также может быть полезно для тех, кто приходит из TEX или LATEX (самая популярная из производных от TEX), которые впервые приближаются к ConTEXt. В то же время я осознаю, что, пытаясь угодить стольким разным читателям, я рискую никого не удовлетворить. Поэтому, в случае сомнения, мне всегда было ясно, что основным адресатом этого документа является новичок в ConTEXt, новичок, который только что пришел в эту увлекательную экосистему.

Быть новичком в ConTEXt не означает также быть новичком в использовании компьютерных инструментов; и хотя в этом введении я не предполагаю какого-либо определенного уровня компьютерной грамотности у читателей, я предполагаю определенную «разумную грамотность», которая подразумевает, например, общее понимание разницы между текстовым процессором и текстовым редактором, знание, как создавать, открывать и манипулировать текстовыми файлами, зная, как устанавливать программы, зная, как открывать терминал и выполнять команду ... и многое другое.

Читая предыдущие части этого введения, когда я пишу эти строки, я понимаю, что иногда я увлекаюсь и вхожу в компьютерные проблемы, которые не являются необходимыми для изучения ConTEXt и могут отпугнуть новичка, в то время как в другое время я занят объяснением довольно очевидных вещей, которые могут утомить опытного читателя. Я прошу снисхождения обоих. Рационально я знаю, что новичку в компьютеризованном управлении текстами очень сложно даже знать, что ConTEXt существует, но с другой точки зрения, в моей профессиональной среде меня окружают люди, которые постоянно борются с текстами, когда они используют текстовые процессоры, и они справляются с этим достаточно хорошо, но никогда не работая с текстовыми файлами как таковыми, они игнорируют такие основные вопросы, как, например, какая кодировка используется в текстовых файлах или в чем разница между текстовым процессором и текстовым редактором.

Тот факт, что это руководство предназначено для людей, которые ничего не знают о ConTEXt или TEX, означает, что я включил информацию, которая явно относится не к ConTEXt а к TEX; но я понял, что нет необходимости обременять читателей информацией, которая для них не имеет отношения,

[1] Я не рисовал изображение, а скачал его из Интернета (https://es.dreamstime.com/), где это говорит, что это изображение без лицензионных отчислений.

[2] Также можно найти в Интернете (https://www.freepik.es/), где разрешено его бесплатное использование.

как это могло бы быть, если определенная команда, которая фактически работает, на самом деле является командой ConTEXt или принадлежит TEX; поэтому только в некоторых случаях, когда это кажется мне полезным, я уточняю, что определенные команды действительно принадлежат TEX.

Что касается организации этого документа, материал сгруппирован в три блока:

- **Первая часть**, состоящая из первых четырех глав, предлагает глобальный обзор ConTEXt, объясняет, что это такое и как мы с ним работаем, показывая первый пример того, как преобразовать документ, чтобы позже можно было объяснить некоторые фундаментальные концепции ConTEXt, а также некоторые вопросы, касающиеся исходных файлов ConTEXt.

  В целом эти главы предназначены для читателей, которые до сих пор знали только, как работать с текстовыми редакторами. Читатель, который уже знает о работе с языками разметки, может пропустить эти первые главы; и если читатель уже знает TeX, или LATEX, он также может пропустить большую часть содержания в главах 3 и 4. Точно так же я бы рекомендовал по крайней мере прочитать:

  – Информацию, относящуюся к командам ConTEXt(Глава 3), и в частности, как они работают, как они сконфигурированы, потому что именно в этом заключается принципиальная разница между концепцией и синтаксисом LATEX и ConTEXt. Поскольку это введение относится только к последнему, эти различия явно не обозначены как таковые, но тот, кто читает эту главу и знает, как работает LATEX сразу поймет разницу в синтаксисе двух языков, а также то, как ConTEXt позволяет нам настроить способ работы почти всех его команд.

  – Информация, относящаяся к многофайловым проектам ConTEXt (Глава 4), которая не так похожа на способ работы с другими системами на основе TEX.

- **Вторая часть**, которая включает главы с 5 по 9, фокусируется на том, что мы считаем основными глобальными аспектами документа ConTEXt.

  – Два аспекта, которые в основном влияют на внешний вид документа, – это размер и макет его страниц и используемый шрифт. Этим вопросам посвящены главы 5 и 6.

    ⋆ Первый фокусируется на страницах: размер, элементы, составляющие страницу, ее макет (то есть, как распределены элементы страницы) и т.д. По систематическим причинам здесь также рассматриваются более конкретные аспекты, например, связанные с разбиением на страницы и механизмы, которые позволяют нам на это влиять.

    ⋆ объясняет команды, связанные со шрифтами, и их работу. Также сюда включено базовое объяснение использования и управления цветами, поскольку, хотя они и не являются строго *характеристикой* шрифтов, они в такой же степени влияют на внешний вид документа.

  – Главы 7 и 8 посвящены структуре документа и инструментам, которые ConTEXt предоставляет автору для написания хорошо структурированных документов. Глава 7 фокусируется на собственно так называемой структуре (структурные подразделения документа), а Глава 8 – на том, как это отражено в Оглавлении; хотя, в соответствии с объяснением этого, мы используем возможность также объяснить, как создавать различные типы индексов с помощью ConTEXt, поскольку для ConTEXt все они относятся к понятию «списки».

  – Наконец, в главе 9 основное внимание уделяется ссылкам, важному глобальному аспекту любого документа, когда нам нужно сослаться на что-то в другой части документа (внутренние ссылки) или на другие документы (внешние ссылки). В последнем случае нас пока интересуют только ссылки (*links*), которые означают переход к внешнему документу. Эти ссылки (которые также могут встречаться во внутренних ссылках) делают наш документ *интерактивным*, и в этой главе мы объясним некоторые особенности ConTEXt для создания таких документов.

  Эти главы не нужно читать в каком-либо определенном порядке, за исключением главы 8, которую может быть легче понять, если сначала прочитать главу 7. В любом случае я постарался

сделать так, чтобы при возникновении вопроса в главе или разделе, рассматриваемом в другом месте этого введения, текст содержал упоминание об этом вместе с гиперссылкой на то место, где рассматривается вопрос. Однако я не могу гарантировать, что так будет всегда.

Наконец, **третья часть** (главы 10 и последующие) посвящена более подробным аспектам. Они независимы не только друг от друга, но даже от своих разделов (кроме, пожалуй, последней главы). Учитывая большое количество утилит, которые включает ConTEXt, эта часть может быть очень обширной; но поскольку я понимаю, что к тому времени, когда они прибудут сюда, читатели уже будут готовы погрузиться в документацию ConTEXt по собственному желанию, я включил только следующие главы:

– В главах 10 и 11 рассматривается то, что мы могли бы назвать *базовыми элементами* любого текстового документа: текст состоит из символов, составляющих слова, сгруппированные в строки, которые, в свою очередь, составляют абзацы, отделенные друг от друга вертикальным пространством... Ясно, что все эти вопросы можно было бы включить в одну главу, но поскольку это было бы слишком долго, я разделил этот вопрос на две главы: одна посвящена символам, словам и горизонтальному пространству, а другая - строкам, абзацам и вертикальным интервалам.

– Глава 12 представляет собой своего рода *мешанину*, касающуюся элементов и конструкций, обычно встречающихся в документах; по большей части академические или научно-технические документы: сноски, структурированные списки, описания, нумерация и т. д.

– Наконец, в главе 13 основное внимание уделяется плавающим объектам, особенно наиболее типичным из них: изображениям, вставленным в документы, и таблицам.

• Введение завершается тремя приложениями. Одно касается установки ConTEXt, второе приложение содержит несколько десятков команд, которые позволяют генерировать различные символы – в основном, но не только для математического использования, а третье приложение содержит алфавитный список команд ConTEXt, объясненных или упомянутых в этом тексте.

Остается объяснить множество вопросов: работа с цитатами и библиографическими ссылками, написание специализированных текстов (математика, химия ...), соединение с XML, интерфейс с кодом Lua, режимы и обработка на основе режимов, работа с MetaPost. для разработки графики и т. д. Вот почему, поскольку я не включаю полное объяснение ConTEXt и не претендую на это, я назвал этот документ «Введение в ConTEXt Mark IV»; и я добавил тот факт, что введение не слишком короткое, потому что, очевидно, это так: текст, который оставил так много вещей еще в разработке, но который уже вышел за пределы 300 страниц, ни в коем случае не является кратким введением. Это потому, что я хочу, чтобы читатель понял логику ConTEXt или, по крайней мере, логику, как я ее понял. Он не претендует на роль справочного руководства, а скорее как руководство для самообучения, которое подготавливает читателя к созданию документов средней сложности (и это включает в себя большинство наиболее вероятных документов) и которое, прежде всего, учит читателя представлять себе, что может быть сделано с помощью этого мощного инструмента, и узнай, как это сделать, в доступной документации. Этот документ не является *учебным пособием*. Учебники предназначены для постепенного повышения уровня сложности, так что то, что нужно выучить, преподается шаг за шагом; в этом отношении я предпочел начать со второй части вместо того, чтобы упорядочивать материал по уровню сложности, чтобы быть более систематизированным. Но хотя это не учебник, я включил очень много примеров.

Возможно, что для некоторых читателей название этого документа напоминает им текст, написанный Oetiker, Partl, Hyna и Schlegl, доступный в Интернете, и один из лучших документов для знакомства с миром LaTeX. Я говорю о «*The Not So Short Introduction to LaTeX* 2_ϵ». Это не совпадение, это дань уважения и признательность: благодаря щедрой работе тех, кто писал подобные тексты, многие люди могут начать работать с такими полезными и мощными инструментами, как LaTeX и ConTEXt. Эти авторы помогли мне начать работу с LaTeX; Я надеюсь сделать то же самое с кем-то, кто хочет начать с ConTEXt, хотя в оригинальной испанской версии этого текста я придерживался исключительно испаноязычного мира, которому не хватало так много документации на их языке. Я надеюсь, что этот

документ оправдывает это ожидание, а тем временем другие великодушно предложили перевести его на другие языки, отсюда и данное английское издание. Спасибо.

Joaquín Ataz-López
Summer 2020

I
Что такое ConTEXt и как мы работаем с ним

Глава 1

ConTEXt: общий обзор

1.1 Что же тогда такое ConTEXt?

ConTEXt – это *система набора текста*, или, другими словами: обширный набор инструментов, направленных на предоставление пользователю абсолютного и полного контроля над внешним видом и представлением конкретного электронного документа, предназначенного для печати на бумаге или для отображения на экране. В этой главе объясняется, что это означает. Но сначала выделим некоторые характеристики ConTEXt.

- Существует две разновидности ConTEXt известные как Mark II и Mark IV соответственно. ConTEXt Mark II заморожен, т.е. считается уже полностью разработанным языком, не предназначенным для дополнительных изменений или добавления новых вещей. Новая версия появится только в том случае, если нужно исправить какую-то ошибку. ConTEXt Mark IV, с другой стороны, продолжает развиваться, поэтому время от времени появляются новые версии, которые вносят некоторые улучшения или дополнительные полезности. Но, хотя он все еще находится в разработке, это очень зрелый язык, в котором изменения, внесенные в новые версии, довольно незначительны и влияют исключительно на низкоуровневую работу системы. Для обычного пользователя эти изменения полностью прозрачны; как будто их не было. Хотя у обоих вкусов много общего, между ними есть и несовместимые черты. Следовательно, это введение посвящено только ConTEXt Mark IV.

- ConTEXt – это *libre* программное обеспечение (или бесплатное программное обеспечение, но не только в смысле бесплатного). Собственно программа (то есть комплекс компьютерных инструментов, составляющих CONTEXT) распространяется под лицензией *GNU General Public Licence*. Документация предлагается по лицензии «*Creative Commons*», которая позволяет ее свободно копировать и распространять.

- ConTEXt не является ни программой текстового процессора, ни программой редактирования текста, а набором инструментов, направленных на *преобразование* текста, который мы ранее написали в нашем любимом текстовом редакторе. Поэтому, когда мы работаем с ConTEXt:

  – Мы начинаем с написания одного или нескольких текстовых файлов с помощью любого текстового редактора.

  – В этих файлах, наряду с текстом, составляющим содержимое документа, есть ряд инструкций, которые сообщают ConTEXt о внешнем виде, который должен иметь окончательный документ, созданный из исходных текстовых файлов. Полный набор инструкций ConTEXt, по сути, является *языком*; и поскольку этот язык позволяет программировать типографское преобразование текста, мы можем сказать, что ConTEXt – *это типографский язык программирования*.

  - После того, как мы напишем исходные файлы, они будут обработаны программой (также называемой «`context`»[1]), которая сгенерирует из них PDF-файл, готовый для отправки в типографию или для отображения на экране.

- Следовательно, в ConTEXt мы должны различать документ, который мы пишем, и документ, который создает ConTEXt. Чтобы избежать каких-либо сомнений, в этом введении я назову текстовый документ, содержащий инструкции по форматированию, исходным файлом, а документ PDF, созданный ConTEXt из исходного файла, я назову *окончательным документом*.

Вышеупомянутые основные положения будут развиваться ниже.

1.2 Верстка текстов

Написание документа (книги, статьи, главы, буклета, распечатки, бумаги ...) и его типографское оформление - это два совершенно разных вида деятельности. Написание документа во многом похоже на его составление; это делает автор, который определяет его содержание и структуру. Документ, созданный непосредственно автором в том виде, в каком он его написал, называется *рукописью*. По самой своей природе доступ к рукописи имеет только автор или лица, которым разрешено читать ее. Его распространение за пределами этой интимной группы требует, чтобы рукопись была *опубликована*. Сегодня опубликовать что-либо - в этимологическом смысле сделать это «общедоступным» - так же просто, как разместить это в Интернете, доступном для всех, кто найдет это и хочет, чтобы это прочитал. Но до относительно недавнего времени публикация была дорогостоящим процессом, зависящим от определенных специалистов, специализирующихся в ней, доступ к которым осуществлялся только теми рукописями, которые из-за своего содержания или из-за их автора считались особенно интересными. И даже сегодня мы стремимся зарезервировать слово *публикация* для такого рода *профессиональных публикаций*, в которых рукопись претерпевает ряд изменений внешнего вида, направленных на улучшение читаемости документа. Эту серию преобразований мы называем *верстkой*.

Целью набора является - вообще говоря, не считая рекламных текстов, которые стремятся привлечь внимание читателя - создать документы с максимальной разборчивостью, что означает качество печатного текста, которое способствует или облегчает его чтение и гарантирует, что читатель чувствует себя комфортно. с этим. Этому способствуют многие вещи; некоторые, конечно, связаны с содержимым документа: (качество, ясность, организация ...), но другие зависят от таких вещей, как тип и размеры используемого шрифта, использование пробелов в документе, визуальное разделение между абзацами и т. д. Кроме того, существуют и другие виды ресурсов, не столько графического или визуального характера, как наличие или иное в документе определенных вспомогательных средств для читателя, таких как верхние и нижние колонтитулы страниц, указатели, глоссарии, использование полужирного шрифта, заголовков на полях и т. д. Знание и правильное обращение со всеми ресурсами, доступными наборщику, можно назвать «искусством набора» или «искусством печати».

Исторически до появления компьютера задачи и роли писателя и наборщика оставались совершенно разными. Автор писал от руки или на машине XIX века, называемой пишущей машинкой, типографские возможности которой были даже более ограниченными, чем у тех, кто писал от руки; а затем автор передал оригиналы издателю или типографии, которые преобразовали их, чтобы получить печатный документ.

Сегодня информатика облегчила автору выбор композиции до мельчайших деталей. Однако это не отменяет того факта, что качества, которые нужны хорошему автору, не такие, как те, которые нуж-

[1] ConTEXt – это одновременно и язык, и программа (помимо прочего). Этот факт в тексте, подобном текущему, создает проблему, из-за которой иногда нам приходится различать эти два аспекта. Вот почему я принял типографское соглашение о написании «ConTeXt» с его логотипом (ConTEXt), когда я хочу сослаться исключительно на язык или и на язык, и на программу. Однако, когда я хочу обратиться исключительно к программе, я буду писать «`context`» строчными буквами и моноширинным шрифтом, типичным для компьютерных терминалов и пишущих машинок. Я также буду использовать этот тип для примеров и упоминаний команд, принадлежащих этому языку.

ны хорошему наборщику. В зависимости от типа документа, с которым имеет дело, автору необходимо понимание предмета, о котором идет речь, ясность изложения, хорошо структурированное мышление, позволяющее создать хорошо организованный текст, творческий подход, чувство ритма и т. Д. и т.д. Но наборщик должен сочетать хорошее знание концептуальных и графических ресурсов, имеющихся в его или ее распоряжении, и достаточно хороший вкус, чтобы иметь возможность использовать их гармонично.

С помощью хорошей программы обработки текста [1] можно получить достаточно хороший типографически подготовленный документ. Но текстовые процессоры, вообще говоря, не предназначены для набора текста, и результаты, хотя они могут быть правильными, несопоставимы с результатами, получаемыми с помощью других инструментов, специально разработанных для управления композицией документа. Фактически, текстовые процессоры - это то, как развивались пишущие машинки, и их использование, поскольку эти инструменты маскируют разницу между авторством текста и его набором, имеет тенденцию создавать неструктурированные и типографически неадекватные тексты. Напротив, такие инструменты, как ConTEXt, произошли от печатного станка; они предлагают гораздо больше возможностей для компоновки, и, прежде всего, невозможно научиться их использовать, не усвоив попутно множество понятий, касающихся набора текста. В этом отличие от текстовых процессоров, которые можно использовать в течение многих лет, не изучая ничего про типографию.

1.3 Языки разметки

Во времена, когда еще не было компьютеров, как я уже говорил, автор готовил рукопись от руки или на пишущей машинке и передавал ее издателю или типографу, который отвечал за преобразование рукописи в окончательный печатный текст. Хотя автор относительно мало участвовал в преобразовании, он или она поддержали некоторое вмешательство, указав, например, что определенные строки рукописи были названиями ее различных частей (глав, разделов ...), или указав что определенные вещи должны быть каким-то образом выделены типографским шрифтом. Эти указания были сделаны автором в самой рукописи, иногда прямо, а иногда через определенные соглашения, которые продолжали развиваться с течением времени. Например, главы всегда начинались на новой странице с вставки нескольких пустых строк перед заголовком, подчеркивания его, написания заглавными буквами или выделения текста в рамку между двумя подчеркиваниями, увеличения отступа абзаца и т.д.

Короче говоря, автор *разметил* текст, чтобы указать, как его следует набирать. Затем, позже, редактор вручную напишет другие обозначения текста для принтера, такие как, например, используемый шрифт и его размер.

Сегодня, в компьютеризованном мире, мы можем продолжать делать это для создания электронных документов с помощью так называемого *языка разметки*. Эти типы языков используют серию *меток* или указаний на то, что программа, обрабатывающая файл, содержащий их, знает, как интерпретировать. Вероятно, самый известный язык разметки сегодня- - это HTML, поскольку сегодня на нем основано большинство веб-страниц. HTML-страница содержит текст веб-страницы вместе с рядом меток, которые сообщают программе браузера, загружающей страницу, как она должна отображаться. Разметка HTML, которую понимают веб-браузеры, вместе с инструкциями о том, как и где их использовать, называется «язык HTML», который является *языком разметки*. Но помимо HTML существует множество других языков разметки; на самом деле они быстро развиваются, и поэтому XML, который является *преимущественно* языком разметки, сегодня можно найти повсюду и используется практически во всем: для проектирования баз данных, для создания конкретных языков, передачи структурированных данных, файлов конфигурации приложений. и т. д. Существуют также

[1] Согласно довольно старому соглашению, мы делаем различие между *текстовыми редакторами* и *текстовыми процессорами*. Первые типы программ редактирования текста работали с неформатированными текстовыми файлами, тогда как другие типы программ работали с двоичными файлами форматированного текста.

языки разметки, предназначенные для графического дизайна (SVG, TikZ или MetaPost), математических формул (MathML), музыки (Lilypond и MusicXML), финансов, геоматики и т. д. И, конечно же, есть также языки разметки, предназначенные для типографское преобразование текста, и среди них выделяются TEX и его производные.

Что касается *типографской разметки*, которая указывает, как должен выглядеть текст, есть два вида, к которым мы можем относиться: *чисто типографская разметка* и *концептуальная разметка* или, если хотите, *логическая разметка*. Чисто типографская разметка ограничивается точным указанием того, какой типографский ресурс следует использовать для отображения определенного текста; например, когда мы указываем, что определенный текст должен быть выделен жирным шрифтом или курсивом. С другой стороны, концептуальная разметка указывает, какая функция соответствует определенному тексту в документе в целом, например, когда мы указываем, что что-то является заголовком, подзаголовком или цитатой. В общем, документы, которые предпочитают использовать этот второй вид разметки, более последовательны и их легче составлять, поскольку они еще раз указывают на разницу между авторством и составом: автор указывает, что такая-то строка является заголовком, или что такая-то и такой фрагмент является предупреждением или цитатой; и наборщик решает, как типографически выделить все заголовки, предупреждения или цитаты; таким образом, с одной стороны, гарантируется согласованность, поскольку все фрагменты, выполняющие одну и ту же функцию, будут выглядеть одинаково, а с другой стороны, это экономит время, потому что формат каждого типа фрагмента нужно указывать только один раз.

1.4 TEX и его производные

TEX был разработан в конце 70-х годов Дональдом Э. Кнутом, профессором (ныне заслуженным профессором) теоретического компьютерного программирования в Стэнфордском университете, который реализовал программу для выпуска своих собственных публикаций и в качестве примера систематически разработанной и аннотированной программы. Наряду с TEX, Knuth разработал дополнительный язык программирования под названием MetaFont, созданный для разработки типографских шрифтов, и он использовал его для разработки шрифта, который он назвал *Computer Modern*, который, наряду с обычными символами любого шрифта, также включал полный набор «Глифов» [1], предназначенные для написания математики. Ко всему этому он добавил несколько дополнительных утилит, и таким образом родилась система набора текста под названием TEX, которая благодаря своей мощности, качеству результатов, гибкости использования и широким возможностям считается одной из лучших компьютеризированных систем для создания текста. Он был разработан для текстов, в которых много математики, но вскоре стало ясно, что возможности системы сделали его пригодным для всех типов текстов.

Внутренне TEX работает так же, как бывшие наборщики в типографии. Для TEX все является *боксом*: буквы содержатся в прямоугольниках, пробелы также являются прямоугольниками, несколько букв (прямоугольники, содержащие несколько букв) образуют новый прямоугольник, в котором заключено слово, а несколько слов вместе с пустым пространством между ними. они образуют рамку, содержащую строку, несколько строк становятся рамкой, содержащей абзац ... и так далее. Все это, кроме того, с необычайной точностью проведения измерений. Учтите, что наименьшая единица, с которой имеет дело TEX, в 65,536 раз меньше типографской точки, с которой измеряются символы и строки, что обычно является наименьшей единицей, обрабатываемой большинством программ обработки текстов. Это означает, что самая маленькая единица измерения, обрабатываемая TEX, составляет примерно 0,000005356 миллиметра.

Название TEX происходит от корня греческого слова τεχνη, написанного заглавными буквами (ΤΕΧΝΗ). Следовательно, последняя буква слова TEX – это не латинское „X”, а греческое „χ”, произносимое - по-видимому - как шотландское «ch» в слове *loch*. Поэтому TEX следует произносить как

[1] В типографике глиф - это графическое представление символа, количества символов или части символа, и сегодня он является эквивалентом буквенного типа (бит, на котором выгравирована буква или подвижный шрифт).

Tech. С другой стороны, это греческое слово означало как «искусство», так и «технологию», и именно по этой причине Кнут выбрал его для названия своей системы. Он писал, что цель этого названия – «напомнить вам, что TEX в первую очередь занимается высококачественными техническими рукописями. Его акцент делается на искусство и технологии, как в основном греческом слове».

Используя соглашение, установленное Knuth, следует записать, TEX:

- В типографически отформатированных текстах, подобных этому, с использованием логотипа, который я использовал до сих пор: три буквы в верхнем регистре, при этом центральная буква „E” слегка смещена ниже, чтобы облегчить более точное выравнивание между буквами „T” и „X”; или другими словами: «TEX».

  Чтобы облегчить написание этого логотипа, Knuth включил в TEX инструкцию по его написанию в окончательный документ: `\TeX`.

- В неформатированном тексте (например, в электронном письме или текстовом файле) буквы „T” и „X” в верхнем регистре, а центральная „e” - - в нижнем регистре; итак: «TeX».

Это соглашение по-прежнему используется во всех производных от TEX, которые включают его собственное имя, как в случае с ConTEXt. При написании в текстовом режиме нам нужно написать «ConTeXt».

1.4.1 Движки TEX

Программа TEX – это бесплатное программное обеспечение *libre*: ее исходный код доступен для всех, и любой может использовать или изменять его по своему усмотрению, с единственным условием, что в случае внесения изменений результат не может быть назван «TEX». Вот почему со временем появились определенные модификации программы, в которые были внесены различные улучшения, которые обычно называются *TEX Engines*. Помимо исходной программы TEX, основными механизмами являются, в хронологическом порядке появления, pdfTEX, ε-TEX, XƎTEX и LuaTEX. Каждый из них должен включать улучшения предыдущего. С другой стороны, эти улучшения, вплоть до появления LuaTEX, не влияли на сам язык, а только на входные файлы, выходные файлы, обработку источников и работу макросов на низком уровне.

Вопрос о том, какой движок TEX использовать, является очень обсуждаемым во вселенной TEX. Я не буду здесь развивать этот вопрос, поскольку ConTEXt Mark IV работает только с LuaTeX. На самом деле в мире ConTEXt обсуждение TEX *Engines* превращается в обсуждение того, использовать ли Mark II (который работает с PdfTeX и XeTeX) или Mark IV (который работает только с LuaTeX).

1.4.2 Форматы

Ядро или сердце TEX понимает только набор из примерно 300 очень простых инструкций, называемых *примитивами*, которые подходят для операций набора и программирования функций. Подавляющее большинство этих инструкций очень *низкого уровня*, что в компьютерной терминологии означает, что они легче понимаются компьютером, чем человеком, поскольку они касаются очень элементарных операций «shift этот символ 0,000725 миллиметров вверх» вид. Таким образом, Knuth увидел, что TEX будет расширяемым, а это означает, что должен существовать механизм, позволяющий определять инструкции на более высоком уровне, более понятном для людей. Эти инструкции, которые во время выполнения разбиваются на другие более простые инструкции, называются *макросами*. Например, инструкция TEX, печатающая логотип (`\TeX`), во время выполнения разбивается следующим образом:

```
T
\kern -.1667em
\lower .5ex
\hbox {E}
\kern -.125em
X
```

Но человеку гораздо легче понять и запомнить, что простая команда «\TeX» выполняет типографские операции, необходимые для печати логотипа.

Разница между *макросом* и *примитивом* имеет значение только с точки зрения разработчика TEX. С точки зрения пользователя это инструкции или, если хотите, *команды*. Кнут назвал их *контрольными последовательностями*.

Возможность расширения языка с помощью *макросов* - одна из характеристик, превративших TEX в такой мощный инструмент. Фактически, сам Knuth создал около 600 макросов, которые вместе с 300 примитивами составляют формат под названием «PlainTEX». Очень часто путают так называемый TEX с PlainTEX и, фактически, почти все, что обычно пишут или говорят о TEX, на самом деле является отсылкой к Plain TeX. Книги, которые утверждают, что посвящены TEX (включая основную «*TheTEXBook*»), действительно относятся к PlainTEX; а те, кто считает, что работают напрямую с TEX на самом деле работают с PlainTEX.

Normally these *formats* are accompanied by a programme that loads the macros that make them up into memory before calling on «tex» (or the actual engine being used for processing) to process the source file. But even though all these formats are actually running TEX, as each of them has different instructions and different syntax rules from the user's point of view, we can think of them as *different languages*. They all draw their inspiration from TEX, but are different from TEX and also different from each other.

1.5 ConTEXt

На самом деле ConTEXt, который начинался как *формат* TEX, сегодня намного больше. ConTEXt включает:

1. Очень широкий набор макросов TEX. Если в PlainTEX около 900 инструкций, то в ConTEXt около 3500; и если мы сложим названия различных опций, поддерживаемых этими командами, мы получим словарь из около 4000 слов. Словарь так велик из-за стратегии ConTEXt, облегчающей его изучение, и эта стратегия означает включение любого количества синонимов для команд и опций.

   Намерение состоит в том, что если должен быть достигнут определенный эффект, то для каждого из способов, которыми носитель английского языка назвал бы этот эффект, есть команда или опция, которая его достигает, что должно облегчить использование языка. Например, чтобы одновременно получить жирное и курсивное письмо, в ConTEXt есть три инструкции, каждая из которых дает один и тот же результат: \bi, \italicbold и \bolditalic.

2. Аналогичным образом широкий набор макросов для MetaPost, графического языка программирования, производного от MetaFont, который, в свою очередь, представляет собой язык для дизайна шрифтов, разработанный Knuth совместно с TEX.

3. Такой же широкий набор макросов для MetaPost, графического языка программирования, полученного из METAFONT, который, в свою очередь, является языком для дизайна шрифтов, который Кнут разработал совместно с TEX.

4. Различные *скрипты*, разработанные на Perl (самый старый), Ruby (некоторые также старые, другие не такие старые) и Lua (самые свежие).

5. Интерфейс, который объединяет TEX, MetaPost, Lua и XML, позволяя писать и обрабатывать документы на любом из этих языков или смешивать элементы из некоторых из них.

   Возможно, вы не поняли многого из предыдущего объяснения? Не беспокойся об этом. Я использовал в нем много компьютерного жаргона и упомянул множество программ и языков. Для использования ConTEXt не обязательно знать все эти различные компоненты. На данном этапе обучения важно придерживаться идеи о том, что ConTEXt объединяет множество инструментов из разных источников, которые вместе составляют систему набора текста.

Именно из-за этой последней особенности интеграции инструментов различного происхождения мы говорим, что ConTEXt – это «гибридная технология», предназначенная для верстки документов. Я понимаю, что это превращает ConTEXt в чрезвычайно продвинутую и мощную систему.

Несмотря на то, что ConTEXt – это гораздо больше, чем набор макросов для TEX, он по-прежнему основан на TEX, и именно поэтому этот документ, который я называю не более, чем *введением*, фокусируется на этом.

ConTEXt, с другой стороны, гораздо более современен, чем TEX. Когда был создан TEX, появление компьютеров было только в начале, и мы были далеки от того, чтобы увидеть, каким будет (станет) Интернет и мультимедийный мир. В этом отношении ConTEXt естественным образом объединяет некоторые вещи, которые всегда были чем-то вроде инородного тела в TEX, такие как включение внешней графики, обработка цвета, гиперссылки в электронных документах, предположение о размере бумаги, подходящем для документа, предназначенного для отображения на экране, и т.д.

1.5.1 Краткая история ConTEXt

ConTEXt родился примерно в 1991 году. Он был создан Hans Hagen и Ton Otten в голландской компании по разработке и обработке документов под названием «*Pragma Advanced Document Engineering*», обычно сокращенно Pragma ADE. Он начинался с того, что представлял собой набор макросов TEX с голландскими именами и неофициально известный как *Pragmatex*, предназначенный для нетехнических сотрудников компании, которым приходилось управлять многими деталями редактирования наборных документов и которые не привыкли к использованию языков разметки или интерфейсов, кроме голландского. Следовательно, первая версия ConTEXt была написана на голландском языке. Идея заключалась в том, чтобы создать достаточное количество макросов с единообразным и согласованным интерфейсом. Примерно в 1994 году *пакет* был достаточно стабильным, чтобы руководство пользователя было написано на голландском языке, а в 1996 году по инициативе Hans Hagen ссылка на него стала называться «ConTEXt». Это имя претендует на то, чтобы означать «Text with TEX» (используя латинский предлог «con», означающий «с»), но в то же время это игра слов на английском (и голландском) слове « CONTEXT». Таким образом, за названием скрывается тройная игра слов, включающая «TEX», «text» и «context».

> Следовательно, поскольку имя основано на игре слов, ConTEXt следует произносить как „context”, а не „context”, поскольку это означало бы потерю игры слов.

Интерфейс начал переводиться на английский язык примерно в 2005 году, в результате чего появилась версия, известная как ConTEXt Mark II, где поясняется „II”, потому что, по мнению разработчиков, предыдущая версия на голландском языке была версией „I”, даже хотя официально это никогда не называлось. После того, как интерфейс был переведен на английский язык, использование системы начало распространяться за пределы Нидерландов, и интерфейс был переведен на другие европейские языки, такие как французский, немецкий, итальянский и румынский. Тем не менее, «official» документация для ConTEXt обычно основана на английской версии, и именно с этой версией работает этот документ.

В своей начальной версии ConTEXt Mark II работал с PdfTeX *TEX engine*. Но позже, с появлением движка XETEX ConTEXt Mark II был изменен, чтобы можно было использовать этот новый движок, дающий ряд преимуществ по сравнению с PdfTeX. Но когда через несколько лет появился LuaTEX, было принято решение внутренне изменить конфигурацию работы ConTEXt, чтобы интегрировать все новые возможности, которые предлагал этот новый движок. Так родился ConTEXt Mark IV, и он был представлен в 2007 году, сразу после презентации LuaTEX. Очень вероятно, что одним из факторов, повлиявших на решение перенастроить ConTEXt для адаптации его к LuaTEX, было то, что два из трех основных разработчиков ConTEXt, Hans Hagen и Taco Hoekwater , также были частью основной команды разработчиков LuaTEX. Вот почему ConTEXt Mark IV и LuaTEX родились одновременно и развивались в унисон. Между ConTEXt и LuaTEX существует синергия, которой нет ни в одной другой производной от TEX; и я сомневаюсь, что кто-то из других может воспользоваться преимуществами LuaTEX, как ConTEXt.

Между Mark II и Mark IV есть много различий, хотя большинство из них являются *внутренними*, то есть они связаны с тем, как макрос фактически работает на более низком уровне, так что с точки зрения пользователя различия не заметны: название и параметры макроса остаются прежними. Однако есть некоторые различия, которые влияют на интерфейс и заставляют действовать по-разному

в зависимости от того, с какой версией он работает. Этих различий относительно немного, но они влияют на очень важные аспекты, такие как, например, кодирование входного файла или обработка шрифтов, установленных в системе.

Однако было бы очень приятно, если бы где-то был документ, объясняющий (или перечисляющий) заметные различия между Mark II и Mark IV. В вики-странице ConTEXt, например, для каждой команды ConTEXt существует два вида синтаксиса (очень часто идентичные). Я предполагаю, что один принадлежит Mark II, а другой - Mark IV; и, исходя из этого предположения, я также предполагаю, что первая версия от Mark II. Но правда в том, что вики ничего об этом нам не сообщает.

Однако было бы очень приятно, если бы где-то был документ, объясняющий (или перечисляющий) заметные различия между Mark II и Mark IV. В ConTEXt wiki, например, для каждой команды ConTEXt есть *два вида синтаксиса* (очень часто идентичные). Я предполагаю, что один принадлежит Mark II, а другой – Mark IV; и, исходя из этого предположения, я также предполагаю, что *первая версия* принадлежит Mark II. Но правда в том, что вики ничего об этом нам не сообщает.

Тот факт, что различий на уровне языков относительно мало, означает, что во многих случаях мы говорим не о двух версиях, а о двух «разновидностях» ConTEXt. Но независимо от того, называете ли вы их тем или другим, факт в том, что документ, подготовленный для Mark II, обычно не может быть скомпилирован с Mark IV и наоборот; и если документ смешивает обе версии, он, скорее всего, не будет хорошо компилироваться ни с одной из них; что подразумевает, что автор исходного файла должен начать с принятия решения о том, писать ли для Mark II или для Mark IV.

Если мы работаем с разными версиями ConTEXt, хорошим приемом для различения с первого взгляда файлов, предназначенных для Mark II, и файлов, предназначенных для Mark IV, является использование разных расширений для имен файлов. Так, например, для любых файлов, которые я написал для Mark II, я использую расширение «`.mkii`», а для файлов, написанных для Mark IV, – «`.mkiv`». Это правда, что ConTEXt ожидает, что все исходные файлы будут иметь расширение «`.tex`», но мы можем изменить расширение файла, если явно укажем расширение файла при применении ConTEXt к файлу.

Дистрибутив ConTEXt, установленный в вики, «ConTEXt Standalone», включает обе версии, и во избежание путаницы – я полагаю – использует разные команды для каждой из них для компиляции файла. Mark II компилируется с помощью команды «`texexec`», а Mark IV – с помощью команды «`context`».

Фактически обе команды, «`context`» и «`texexec`», представляют собой *сценарии* с разными параметрами, которые запускают «`mtxrun`», который, в свою очередь, является сценарием Lua.

Сегодня Mark II заморожен, а Mark IV продолжает развиваться, что означает, что новые версии первого публикуются только при обнаружении ошибок или неисправностей, которые необходимо исправить, в то время как новые версии Mark IV продолжают регулярно публиковаться; иногда два или три раза в месяц, хотя в большинстве этих случаев «новые версии» не вносят заметных изменений в язык, а ограничиваются каким-либо улучшением реализации команды на низком уровне или обновлением некоторых из многих включенных руководств. с раздачей. Даже в этом случае, если мы установили разрабатываемую версию - что я бы порекомендовал и которая установлена по умолчанию вместе с «ConTEXt Standalone» – имеет смысл время от времени обновлять нашу версию (см. Appendix ??, чтобы узнать, как обновить установленную «ConTEXt Standalone» версию).

LMTX и другие альтернативные реализации Mark IV

Разработчики ConTEXt, естественно, неугомонны, и поэтому не прекратили разработку ConTEXt Mark IV; новые версии все еще тестируются и экспериментируются, хотя в целом они очень мало отличаются от Mark IV и не имеют несовместимости при компиляции, которая существует между Mark IV и Mark II.

Таким образом, были разработаны некоторые второстепенные варианты Mark IV, названные соответственно Mark VI, Mark IX и Mark XI. Из них мне удалось найти лишь небольшую ссылку на Mark VI в ConTEXt wiki, где говорится, что единственное отличие от Mark IV заключается в возможности определения команд путем присвоения параметрам, а не числа, как это принято в TEX, но имя, как это обычно делается почти во всех языках программирования.

Я считаю, что более важным, чем эти небольшие вариации, является появление в мире ConTEXt (ConTEXt verse?) новой версии под названием LMTX, имя, которое является аббревиатурой LuaMetaTeX: новый *движок* TEX, который представляет собой упрощенную версию LuaTEX, разработанный с целью экономии ресурсов компьютера; это означает, что LMTX требует меньше памяти и меньше вычислительной мощности, чем ConTEXt Mark IV.

LMTX был представлен весной 2019 года, и предполагается, что он не повлечет за собой каких-либо внешних изменений языка Mark IV. Для автора документа не будет разницы при работе с ним; но при его компиляции нужно будет выбирать между выполнением этого с LuaTEXили с LuaMetaTeX. В Appendix ??, касающемся установки ConTEXt, показана процедура для присвоения разных имен команд каждой из установок (section ??).

1.5.2 ConTEXt против LATEX

Учитывая, что наиболее популярным форматом, производным от TEX, является LATEX, сравнение между этим форматом и ConTEXt неизбежно. Ясно, что мы говорим о разных языках, хотя в некотором роде они связаны друг с другом, поскольку оба они происходят от TEX; эта связь аналогична той, которая существует, например, между испанским и французским: языки, имеющие общее происхождение (латынь), что означает, что их синтаксис похож, и многие слова на каждом из этих языков отражаются на словах в другом. Но помимо этого семейного сходства, LATEXи ConTEXt различаются своей философией и реализацией, поскольку первоначальные цели обоих в некоторой степени противоположны. LATEXутверждает, что облегчает использование TEX, изолируя автора от конкретных типографских деталей, чтобы помочь сосредоточиться на содержании, оставляя детали набора в руках LATEX. Это означает, что упрощение использования TEX происходит за счет ограничения огромной гибкости TEX за счет предопределения основных форматов и ограничения количества типографских проблем, которые должен решить автор. В отличие от этой философии, ConTEXt родился в компании, занимающейся набором документов. Поэтому цель состоит не в том, чтобы изолировать автора от деталей набора, а в том, чтобы дать автору абсолютный и полный контроль над ними. Для этого ConTEXt предоставляет единообразный и согласованный интерфейс, который намного ближе к исходному духу TEX, чем LATEX.

Это различие в философии и основополагающих целях, в свою очередь, приводит к различиям в реализации. LATEX, который стремится максимально упростить вещи, не требует использования всех ресурсов TEX. В чем-то его суть довольно проста. Поэтому, когда есть необходимость расширить его возможности, необходимо специально написать для этого пакет. Эта упаковка, связанная с LATEX, является одновременно достоинством и недостатком: достоинством, потому что огромная популярность LATEXвместе с щедростью его пользователей означает, что почти любая потребность, которая у нас может быть, была кем-то удовлетворена раньше, и что для этого есть пакет; но это также недостаток, потому что эти пакеты часто несовместимы друг с другом, а их синтаксис не всегда единообразен. Это означает, что работа с LATEXтребует постоянного поиска в тысячах уже существующих пакетов, чтобы удовлетворить свои потребности и убедиться, что все они работают вместе.

В отличие от простоты ядра LATEX, которое дополняется его расширяемостью с помощью пакетов, ConTEXt спроектирован так, чтобы иметь в себе все или почти все типографские возможности TEX, поэтому его концепция гораздо более монолитна, но в то же время время он также более модульный. Ядро ConTEXt позволяет нам делать практически всё, и мы гарантируем, что не будет несовместимости между его различными утилитами, не будет необходимости исследовать расширения для того, что нам нужно, и синтаксис языка не изменится только потому, что нам нужен конкретный инструмент.

Это правда, что ConTEXt имеет так называемые модули расширения, некоторые из которых могут рассматривать как выполняющие функцию, аналогичную пакетам LATEX, но на самом деле они работают по-разному: модули ConTEXtпредназначены исключительно для включения дополнительных утилит, которые, поскольку они все еще остаются на экспериментальной стадии, еще не включены в ядро, или чтобы разрешить доступ к расширениям, созданным кем-то, не входящим в команду разработчиков ConTEXt.

Я не считаю, что одна из этих двух *философий* предпочтительнее другой. Вопрос скорее зависит от профиля пользователя и его желаний. Если пользователь не хочет иметь дело с типографскими вопросами, а просто создает стандартизированные документы очень высокого качества, вероятно, было бы предпочтительнее выбрать такую систему, как LATEX; с другой стороны, пользователю, который любит экспериментировать или которому необходимо контролировать каждую деталь документа, или кому-то, кому нужно разработать специальный макет для документа, вероятно, будет лучше использовать такую систему, как ConTEXt, где все управление в руках автора; с риском, конечно, не знать, как правильно использовать этот элемент управления.

1.5.3 Хорошее понимание динамики работы с ConTEXt

Когда мы работаем с ConTEXt, мы всегда начинаем с написания текстового файла (который мы называем исходным файлом), в который, наряду с фактическим содержанием нашего окончательного документа, мы будем включать инструкции (на языке ConTEXt), которые точно указывают как мы хотим отформатировать документ: общий вид, который мы хотим, чтобы его страницы и абзацы имели, поля, которые мы хотим применить к определенным абзацам, шрифт, который мы хотим отображать, фрагменты, которые мы хотим отображать другим шрифтом, и т. д. После того, как мы написали исходный файл, мы применяем «`context`» -программу из терминала, который обработает его и сгенерирует из него другой файл, в котором содержимое нашего документа будет отформатировано в соответствии с инструкциями, включенными в исходный файл для этой цели. Этот новый файл можно отправить на (коммерческий) принтер, отобразить на экране, разместить в Интернете или распространить среди контактов, друзей, клиентов, учителей, учеников ... или, другими словами, всем, для кого мы написали документ.

Это означает, что при работе с ConTEXt автор работает с файлом, внешний вид которого не имеет ничего общего с окончательным документом: файл, с которым автор работает напрямую, представляет собой текстовый файл, не отформатированный типографически. Таким образом, ConTEXt работает иначе, чем программы, известные как *текстовые процессоры*, которые показывают окончательный вид отредактированного документа в то же время, когда мы его пишем. Тем, кто привык к текстовым редакторам, работа с ConTEXt поначалу покажется странной, и может потребоваться некоторое время, чтобы привыкнуть к ней. Однако, как только к этому привыкнешь, понимаешь, что на самом деле этот другой способ работы, различие между рабочим файлом и конечным результатом, на самом деле является преимуществом по многим причинам, среди которых я выделю здесь, без соблюдения какого-либо конкретного порядка, следующий:

1. Поскольку текстовые файлы «легче» обрабатывать, чем двоичные файлы текстового процессора, и их редактирование требует меньше памяти компьютера, они с меньшей вероятностью будут повреждены и не станут неразборчивыми, когда мы изменим версию создающей их программы. Они также совместимы с любой операционной системой и могут редактироваться с помощью многих текстовых редакторов, поэтому для работы с ними компьютерной системе не обязательно иметь программу, с помощью которой был создан файл, установленной на нем: любое другое редактирование программа подойдет; и в каждой компьютерной системе всегда есть программа для редактирования текста.

2. Поскольку различие между рабочим документом и окончательным документом помогает отличить фактическое содержание документа от его внешнего вида, позволяя автору сосредоточиться на содержании на этапе создания и сосредоточиться на внешнем виде на этапе набора.

3. Потому что это позволяет быстро и точно изменить внешний вид документа, так как это определяется командами ConTEXt, которые можно легко идентифицировать.

4. Поскольку эта возможность для изменения внешнего вида, с другой стороны, позволяет нам легко генерировать две (или более) разные версии из одного контента: возможна одна версия, оптимизированная для печати на бумаге, а другая предназначена для отображения на экране, адаптирована к размеру последней и, возможно, включает гиперссылки, которые не имеют смысла в бумажном документе.

5. Потому что типографских ошибок (опечаток), которые часто встречаются в текстовых редакторах, например выделения курсивом за пределы последнего символа слова, также легко избежать.

6. Поскольку, хотя рабочий файл не распространяется и предназначен „только для наших глаз”, можно включать аннотации и наблюдения, комментарии и предупреждения для нас самих для последующих редакций или версий, при этом не сомневаясь, что они не будут появятся в отформатированном файле для распространения.

7. Потому что качество, которое может быть получено при одновременной обработке всего документа, намного выше, чем то, которое может быть достигнуто с помощью программы, которая должна принимать типографские решения во время написания документа.

8. И так далее.

Все вышеперечисленное означает, что, с одной стороны, при работе с ConTEXt, как только мы овладеем им, мы станем более эффективными и продуктивными, а с другой стороны, качество печати, которое мы можем получить, намного превосходит то, что мы можем можно получить с помощью так называемых текстовых редакторов. И хотя последние действительно проще в использовании, на самом деле они не намного проще в использовании. Поскольку, хотя ConTEXt, как мы уже говорили, действительно содержит 3500 инструкций, обычному пользователю ConTEXt не нужно знать их все. Чтобы делать то, что обычно делают с текстовыми редакторами, нам нужно знать только инструкции, которые позволяют нам указать структуру документа, несколько инструкций, касающихся общих типографских ресурсов, таких как жирный шрифт или курсив, и, возможно, как создать список, или сноска. В общей сложности не более 15 или 20 инструкций позволят нам делать почти все то, что делается с помощью текстового процессора. Остальные инструкции позволяют нам делать разные вещи, которые мы обычно не можем делать с текстовым процессором или которые очень трудно выполнить. Мы можем сказать, что хотя научиться использовать ConTEXt сложнее, чем научиться пользоваться текстовым процессором, это потому, что мы можем сделать намного больше с помощью ConTEXt.

1.5.4 Получение помощи для ConTEXt

Хотя мы новички в этом, лучшее место для получения помощи по ConTEXt, несомненно, находится в Wiki, которая изобилует примерами и имеет хорошую поисковую систему, особенно если кто-то хорошо понимает английский. Конечно, мы также можем найти помощь в Интернете, но здесь игра слов в названии ConTEXt сыграет с нами злую шутку, потому что поиск по слову «контекст» вернет миллионы результатов, большинство из которых не будет иметь ничего общего с тем, что мы ищем. Чтобы найти информацию о ConTEXt, вам нужно добавить что-нибудь к слову «контекст»; например, «tex», или «Mark IV», или «Hans Hagen» (один из создателей ConTEXt), или «Pragma ADE», или что-то подобное. Также может быть полезно искать информацию, используя название wiki: «contextgarden».

Когда мы узнаем что-то еще о ConTEXt мы можем проконсультироваться с некоторыми из многих документов, включенных в «ConTEXt Standalone», или даже обратиться за помощью вTeX – LaTeX Stack Exchange или в список рассылки дляConTEXt (NTG-context). К последним относятся люди, которые знают о ConTEXt больше всего, но правила хорошего кибер-этикета требуют, чтобы, прежде чем задавать вопрос, мы должны были заранее постараться найти ответ сами.

Глава 2

Наш первый исходный файл

Содержание: 2.1 Подготовка к эксперименту: основные инструменты:; 2.2 Сам эксперимент; 2.3 Структура нашего тестового файла; 2.4 Некоторые дополнительные сведения о том; 2.5 Managing errors;

Эта глава посвящена нашему первому эксперименту и объяснит основную структуру документа ConTEXt, а также лучшие стратегии для работы с потенциальными ошибками.

2.1 Подготовка к эксперименту: основные инструменты:

Чтобы написать и скомпилировать первый исходный файл, в нашей системе должны быть установлены следующие инструменты.

Текстовый редактор для записи нашего тестового файла. Существует множество текстовых редакторов, и трудно представить себе операционную систему, в которой один из них еще не установлен. Мы можем использовать любой из них: есть простые, более сложные, более мощные, за некоторые вы платите, некоторые бесплатные (как *gratis*), некоторые бесплатные (как *libre*), некоторые специализируются на системах TEX, другие – общего характера и т. д. Если мы привыкли работать с конкретным редактором, лучше продолжать работать с ним; Если мы до сих пор не привыкли работать с одним из них, я советую сначала найти простой редактор, чтобы не усложнять изучение ConTEXtзадачей обучения работе с текстовым редактором. Хотя это правда, что часто самые сложные для изучения программы оказываются самыми мощными.

Я написал этот текст с помощью GNU Emacs, который является одним из самых мощных и универсальных редакторов общего назначения из существующих; правда, у него есть свои особенности и недоброжелатели, но в целом «любителей Emacs» больше, чем «ненавистников Emacs». Существует расширение GNU Emacs под названием AucTeX для работы с файлами TEX или одной из его производных, которое предоставляет редактору несколько очень интересных дополнительных утилит, хотя AucTeX в целом лучше подготовлен к работе с LATEX, чем с файлами ConTEXt. GNU Emacs в сочетании с AucTeX может быть хорошим вариантом, если мы не знаем, какой редактор выбрать; обе программы являются свободно распространяемыми, поэтому существуют их версии для всех операционных систем. Фактически, заявление о том, что GNU Emacs является *свободным программным обеспечением*, является преуменьшением, поскольку эта программа лучше, чем любая другая, воплощает в себе дух того, что такое *свободное программное обеспечение* и что оно означает. В конце концов, его основным разработчиком стал Richard Stallman, основатель и идеолог проекта GNU и *Free Software Foundation*.

Помимо GNU Emacs + AucTeX, другие хорошие варианты, если вы не знаете, что выбрать, - это *Scite* и *TexWorks*. Первый, хотя редактор общего назначения, не предназначенный специально для работы с файлами ConTEXt, легко настраивается, и, поскольку это редактор, который обычно используют разработчики ConTEXt, «ConTEXt Standalone» содержит файлы конфигурации для этого редактора, написанные самим Хансом Хагеном. TexWorks, с другой стороны, является быстрым

текстовым редактором и специализируется на обработке файлов TEX и его производных языков. Настроить его для работы с ConTEXt достаточно просто, и «ConTEXt Standalone» также предусматривает его настройку.

Каким бы ни был редактор, мы не должны использовать в качестве текстового редактора *текстовый процессор*, например, OpenOffice Writer или Microsoft Word. Эти программы, также слишком медленные и тяжелые, на мой взгляд, могут, если это явно указано, сохранить файл как «только текст (txt)», но они не были предназначены для этого, и мы, скорее всего, в конечном итоге сохраним наш файл в двоичном формате, несовместимом с ConTEXt.

1. **Дистрибутив ConTEXt** для обработки нашего тестового файла. Если в нашей системе уже есть установка TEX (или LATEX), возможно, уже установлена версия ConTEXt. Чтобы проверить это, достаточно открыть терминал и набрать

   ```
   $> context --version
   ```

   > **ПРИМЕЧАНИЕ** для тех, кто плохо знаком с работой с терминала – первые два символа, которые я написал («$>»), не нужно писать в терминале. Я просто представил то, что называется подсказкой терминала: маленький мигающий знак, указывающий на то, что терминал ожидает инструкций.

   Если уже установлена версия ConTEXt, появится что-то вроде следующего:

   ```
   mtx-context     | ConTeXt Process Management 1.03
   mtx-context     |
   mtx-context     | main context file: /home/jq/context/LMTX/tex/texmf-context/
                   | tex/context/base/mkiv/context.mkiv
   mtx-context     | current version: 2020.04.30 11:15
   mtx-context     | main context file: /home/jq/context/LMTX/tex/texmf-context/
                   | tex/context/base/mkiv/context.mkxl
   mtx-context     | current version: 2020.04.30 11:15
   ```

   Последняя строка сообщает нам дату выпуска установленной версии. Если он слишком старый, мы должны либо обновить его, либо установить новую версию. Я рекомендую установить дистрибутив под названием «ConTEXt Standalone», инструкции по установке которого можно найти на ConTEXt wiki. Вы можете найти краткое изложение всего этого в Appendix ??.

   Программа для чтения PDF-файлов, чтобы мы могли видеть результат нашего эксперимента на экране. В Windows и Mac OS всегда есть Adobe Acrobat Reader. Он не устанавливается по умолчанию (или не устанавливался, когда я перестал использовать Microsoft Windows более 15 лет назад), но он устанавливается, когда вы впервые пытаетесь открыть файл PDF, поэтому, скорее всего, он уже установлен. В системах Linux / Unix нет текущей версии Acrobat Reader, но она вам и не нужна, поскольку доступны буквально десятки бесплатных и очень хороших программ для чтения PDF. Кроме того, в этих системах почти всегда один из них установлен по умолчанию. Мне больше всего нравится по скорости и простоте использования MuPDF; хотя у него есть некоторые недостатки, если вы используете языки, отличные от английского, с диакритическими знаками, и он не позволяет вам выделять текст или отправлять документ на принтер; это просто читатель; но пользоваться им очень быстро и удобно. Когда мне нужны некоторые средства, которые не работают в MuPDF, я обычно использую Okular или qPdfView. Но опять же, это дело вкуса: каждый может выбрать то, что ему больше нравится.

Мы можем выбрать наш редактор, наш PDF-ридер, наш дистрибутив ConTEXt ... Добро пожаловать в мир *free libre software*!

2.2 Сам эксперимент

Запись исходного файла

Если упомянутые выше инструменты уже доступны, нам нужно открыть наш текстовый редактор и создать с ним файл, который мы назовем «rain.tex». В качестве содержимого этого файла мы напишем следующее:

```
% Первая строка документа

\mainlanguage[en] % Language = English

\setuppapersize[S5] % paper size

\setupbodyfont
  [modern,12pt] % Font = Latin Modern, 12 point

\setuphead      % Format of chapters
  [chapter]
  [style=\bfc]

\starttext  % Begin document contents

\startchapter
  [title=The rain in Spain...]

How kind of you to let me come.
Now once again, where does it rain?
On the plain, on the plain.
And where's that blasted plain?
In Spain, in Spain.
The rain in Spain stays mainly in the plain.
The rain in Spain stays mainly in the plain.

\stopchapter

\stoptext % End of document
```

Во время написания не имеет значения, изменится ли что-либо, особенно при добавлении или удалении пробелов или разрывов строк. Важно то, что слова, следующие за «\», написаны точно так же, как и их содержание в фигурных скобках. В остальном возможны вариации.

Кодировка символов файла

После того, как мы написали то, что указано выше, мы сохраняем файл на диск, убедившись, что кодировка символов – UTF-8. Эта кодировка символов является сегодня стандартной. В любом случае, если мы не уверены, мы можем увидеть кодировку из самого текстового редактора и при необходимости изменить ее. Как это сделать, очевидно, зависит от используемого текстового редактора. В GNU Emacs, например, при одновременном нажатии обеих клавиш CTRL-X, затем Return, за которым следует „f”, в последней строке окна (которое GNU Emacs называет мини-буфером) появится сообщение с запросом для новой кодировки и сообщает нам, какая текущая кодировка. В других редакторах мы обычно можем получить доступ к кодировке в меню «Сохранить как».

После того, как мы проверили правильность кодировки и сохранили файл на диске, мы закрываем редактор и сосредотачиваемся на анализе написанного.

Взглянем на содержимое нашего первого исходного файла

Первая строка начинается с символа «%». Это зарезервированный символ, указывающий ConTEXt не обрабатывать текст между этим символом и концом строки, в которой он находится. Это помогает, когда мы хотим написать комментарий к исходному файлу, который может прочитать только автор, поскольку он не становится частью окончательного документа. В этом примере я использовал этот символ, чтобы привлечь внимание к определенным строкам, объясняя, что они делают.

Следующие строки начинаются с символа «\», еще одного из зарезервированных символов ConTEXt, указывающих на то, что то, что за ним следует, является именем команды. В этом примере показан ряд команд, содержащихся в исходном файле ConTEXt: язык, на котором написан документ, размер бумаги, шрифт, который будет использоваться в документе, и способ форматирования глав. Далее

в других главах мы увидим детали этих команд, но на данный момент меня интересует только то, что читатель видит, как они выглядят: они всегда начинаются с символа «\», затем идет имя команды, а затем, между изогнутыми скобками (также известные как фигурные скобки, но мы будем использовать фигурные скобки в этом документе, чтобы прояснить разницу) или квадратные скобки, в зависимости от ситуации, данные, необходимые команде для выполнения своих действий. Между названием команды и квадратными или фигурными скобками, которые его сопровождают, могут быть пробелы или разрывы строк.

В 9-й строке нашего примера (я считаю только строки с каким-либо текстом в них) стоит важная команда `\starttext`: она сообщает ConTEXt, что содержимое документа начинается с этой точки; и в последней строке нашего примера мы видим команду `\stoptext`, которая говорит, что здесь заканчивается документ. Это две очень важные команды, о которых я скоро скажу больше. Между ними лежит фактическое содержание нашего документа, который, в нашем примере, состоит из знаменитого диалога из «My Fair Lady» «The Rain in Spain...». Я написал это в форме прозы, чтобы мы могли видеть, как ConTEXt форматирует абзац.

Обработка исходного файла

На следующем этапе, убедившись, что ConTEXt правильно установлен в нашей системе, нам нужно открыть терминал в том же каталоге, в котором был сохранен наш исходный файл «rain.tex».

> Многие текстовые редакторы позволяют нам скомпилировать документ, над которым мы работали, без необходимости открывать терминал. Однако *каноническая* процедура обработки документа с помощью ConTEXt подразумевает выполнение её из терминала путём непосредственного выполнения программы. Я собираюсь делать это таким образом (или предполагать, что это делается так) в этом документе по разным причинам; во-первых, я не могу знать, какой текстовый редактор использует читатель. Но наиболее важным является то, что, используя терминал, мы будем иметь доступ к экранному выводу из «context» и можем видеть сообщения, поступающие из программы.

Если дистрибутив ConTEXt, который мы установили, является «ConTEXt Standalone», прежде всего нам нужно выполнить сценарий, который сообщает терминалу путь и расположение файлов, которые ConTEXt должен запускать. В системах Linux / Unix для этого нужно написать следующую команду:

```
$> source ~/context/tex/setuptex
```

предполагая, что мы установили ConTEXt в каталог с именем «`context`».

> Что касается выполнения *script*, о котором мы только что говорили, посмотрите, что говорится в Appendix ?? относительно установки «ConTEXt Standalone».

После того, как переменные, необходимые для запуска «`context`», были загружены в память, мы можем запустить его. Мы делаем это, набирая

```
$> context rain
```

в терминале. Обратите внимание, что хотя исходный файл называется «`rain.tex`», при вызове «`context`» мы опускаем расширение файла. Если бы мы назвали исходный файл, например, «`rain.mkiv`» (что я обычно делаю, чтобы сказать, что этот файл был написан для Mark IV), нам пришлось бы прямо указать расширение файла, написав «`context rain.mkiv`».

После запуска «`context`» в терминале на экране появятся несколько десятков строк, сообщающих нам, что делает ConTEXt. Эта информация появляется со скоростью, за которой человек не может уследить, но нам не следует беспокоиться об этом, поскольку эта информация сохраняется не только на экране, но и во вспомогательном файле с расширением «`.log`». Он создается во время обработки, и при необходимости мы можем спокойно проконсультироваться с ним позже.

Через несколько секунд, если мы записали текст в исходный файл, не допустив серьёзных ошибок, сообщения терминала закончатся. Последнее сообщение сообщит нам, сколько времени потребовалось для компиляции файла. При первой компиляции документа требуется немного больше времени,

поскольку ConTEXt должен загрузить в память любые файлы, сообщающие ему, какие шрифты используются, а для дальнейшей обработки они уже загружены. Когда появится последнее сообщение о затраченном времени, обработка будет завершена. Если все прошло хорошо, каталог, в котором мы запустили «`context`», теперь будет содержать три дополнительных файла:

- rain.pdf
- rain.log
- rain.tuc

Первый из них - это результат нашей обработки, или, другими словами, это получившийся отформатированный PDF-файл. Второй – это файл «`.log`», в котором хранится вся информация, отображаемая на экране во время обработки файла; третий – вспомогательный файл, который ConTEXtгенерирует при компиляции и используется для построения индексов и перекрестных ссылок. А пока, если всё прошло, как ожидалось, мы можем удалить оба файла (`rain.log` и `rain.tuc`). Если возникла какая-либо проблема, информация в этих файлах поможет нам узнать, где она находится, и поможет найти решение.

Если мы не получили этих результатов, вероятно, это связано с:

- либо неправильно установили наш дистрибутив ConTEXt, и в этом случае при написании команды ««`context`»» в терминале мы увидели бы сообщение «command unknown».
- или наш файл не был закодирован как UTF-8, и это привело к ошибке обработки.
- или, возможно, ConTEXt, установленный в нашей системе, был Mark II. В этой версии мы не можем использовать кодировку UTF-8 без явного указания этого в исходном файле. Мы могли бы настроить исходный файл так, чтобы он компилировался должным образом, но, учитывая, что это введение относится к Mark IV, нет смысла продолжать работу с Mark II: для нас было бы лучше установить «ConTEXt Standalone».
- или мы допустили ошибку в исходном файле при записи имени команды или связанных с ней данных.

  Если после запуска «`context`» терминал начал выдавать сообщения, а затем остановился без повторного появления подсказки, перед продолжением нам нужно нажать CTRL-X, чтобы прервать выполнение ConTEXt, которое было прервано из-за ошибки.

Затем нам нужно проверить, что произошло, и разрешить это, пока мы не получим правильную компиляцию.

На figure 2.1 мы видим содержимое «`rain.pdf`». Мы также видим, что ConTEXt пронумеровал страницу и главу и написал текст указанным шрифтом. В этом случае не происходит переноса слов, но по умолчанию ConTEXt будет переносить слова в конце строки в соответствии с правилами расстановки переносов для выбранного языка, и в нашем случае первая строка нашего исходного файла указывает (`\mainlanguage[en]`).

Подводя итог: ConTEXt преобразовал исходный файл и сгенерировал файл, в котором у нас есть документ, отформатированный в соответствии с инструкциями в исходном файле. Все комментарии в нем исчезли, и что касается команд, то теперь у нас есть не их имена, а результаты их выполнения.

2.3 Структура нашего тестового файла

В проекте, разработанном только в одном исходном файле, структура очень проста и помечена командами `\starttext` ... `\stoptext`. Все, что находится между первой строкой файла и командой `\starttext`, называется преамбулой. Содержимое фактического документа вставляется между командами `\starttext` и `\stoptext`. В нашем примере преамбула включает три команды глобальной конфигурации: одна для указания языка нашего документа (`\mainlanguage`), другая для указания размера страниц (`\setuppapersize`), который в нашем случае равен «S5», представляя размеры

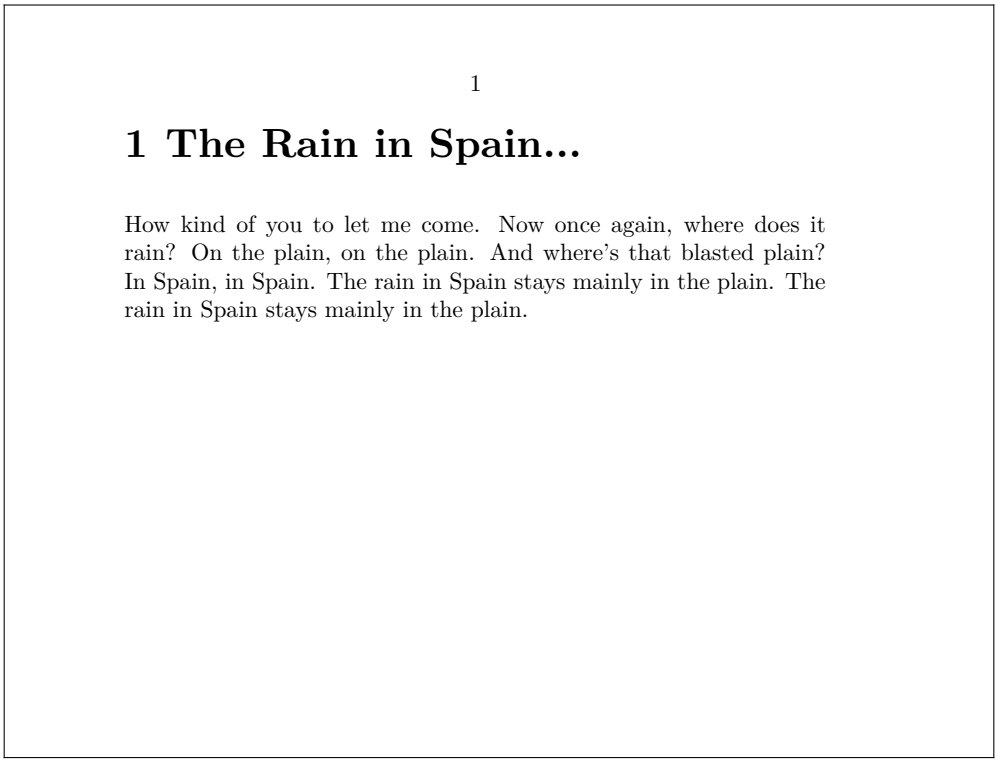


Рисунок 2.1 The rain in Spain...

экрана компьютера и третья команда (`\setuphead`), которая позволяет нам настроить, как будут выглядеть заголовки глав.

Тело документа заключено между командами `\starttext` и `\stoptext`. Эти команды указывают начало и конец обрабатываемого текста соответственно: между ними нам нужно включить весь текст, который мы хотим обработать ConTEXt, вместе с командами, которые не должны влиять на весь документ, а только на его части. А пока отметим, что команды `\starttext` и `\stoptext` обязательны в каждом документе ConTEXt, хотя в дальнейшем, говоря о многофайловых проектах (section 4.6), мы увидим, что есть некоторые исключения из этого.

2.4 Некоторые дополнительные сведения о том

Команда «`context`», с которой мы начали обрабатывать наш первый исходный файл ранее, на самом деле является сценарием Lua Lua *script*, то есть небольшой программы Lua, которая после выполнения некоторых проверок вызывает LuaTEX, поскольку именно она обрабатывает исходный файл.

Мы могли бы вызвать «`context`» различными вариантами. Параметры вводятся сразу после имени команды с двумя дефисами. Если мы хотим ввести более одного варианта, мы разделяем их пробелом. Опция «`help`» дает нам список всех опций с кратким объяснением каждой из них:

```
$>context --help
```

Некоторые наиболее интересные опции следующие:

`interface`: Как я уже сказал во вводной главе, интерфейс ConTEXt переведен на разные языки. По умолчанию интерфейс на английском языке, однако эта опция позволяет нам указать ему использовать голландский (nl), французский (fr), итальянский (it), немецкий (de) или румынский (ro).

purge, purgeall: Удалить вспомогательные файлы, созданные в процессе обработки.

result=Name: Указывает имя, которое должен иметь результирующий файл PDF. По умолчанию он будет таким же, как обрабатываемый исходный файл, с расширением .PDF.

usemodule=list: Загрузить указанные модули перед запуском ConTEXt (модуль - это расширение ConTEXt, которое не является частью его ядра и предоставляет некоторую дополнительную утилиту).

useenvironment=list: Загрузить указанные файлы среды перед запуском ConTEXt (файл среды - это файл с инструкциями по настройке).

version: Показать версию ConTEXt.

help: напечатать информацию помощи опций программы.

noconsole: Запретить отправку сообщений на экран во время компиляции. Однако эти сообщения по-прежнему сохраняются в файле .log.

nonstopmode: Выполнить компиляцию без остановки при возникновении ошибок. Это не означает, что ошибка не возникает, но когда ConTEXt обнаруживает ошибку, даже если он может исправить, он будет продолжать компиляцию до конца или до тех пор, пока не обнаружит ошибку, от которой не может восстановить.

batchmode: Комбинация двух предыдущих вариантов. Он работает без перебоев и не выводит никаких экранных сообщений.

На ранних этапах изучения ConTEXt я не думаю, что стоит использовать последние три параметра, поскольку, когда возникает ошибка, мы не будем иметь ни малейшего представления о том, где она находится и что ее вызвало. И поверьте, дорогие читатели, рано или поздно у вас будет ошибка при обработке.

2.5 Managing errors

При работе с ConTEXt рано или поздно неизбежны ошибки при обработке. Мы можем сгруппировать ошибки по этим четырем категориям:

1. **Ошибки записи**. Они возникают, когда мы ошибаемся в написании команды. В этом случае мы отправим компилятору запрос, который он не понимает. Например, когда вместо записи команды `\TeX` мы пишем `\Tex` с последним нижним регистром „x”, учитывая, что ConTEXt различает верхний и нижний регистр и, следовательно, видит «TeX» и «Tex» как разные слова. ; или если функциональные параметры команды помещены в квадратные скобки вместо фигурных скобок, или если мы пытаемся использовать зарезервированные символы как если бы они были нормальными символами и т. д.

2. **Ошибки упущения**. В ConTEXt есть инструкции, которые запускают задачу, требующую, чтобы мы также явно указали, когда она закончится; как зарезервированный символ $, который включает математический режим, который продолжается до тех пор, пока он не будет отключен, и если мы забудем его отключить, при обнаружении текста или инструкции, которые не имеют смысла в математическом режиме, возникает ошибка. То же самое, если мы начинаем текстовый блок с зарезервированного символа „{” или с команды `\startSomething`, и далее при явном закрытии „}” или команда `\stopsomething` не найдена.

3. **Концептуальные ошибки**. Это то, что я называю ошибками, возникающими, когда вызывается команда, которая требует определенных аргументов, но они не указаны, или когда синтаксис, вызывающий команду, неверен.

4. **Ситуационные ошибки**. Есть некоторые команды, которые предназначены для работы только в определенных контекстах или средах и не распознаются вне их. Это происходит особенно в

математическом режиме: некоторые команды ConTEXt работают только при написании математических формул, и при вызове в другой среде они генерируют ошибку.

Что мы делаем, когда «context» предупреждает нас во время обработки, что произошла ошибка? Первым делом, очевидно, является определение ошибки. Для этого нам необходимо проанализировать файл «.log», сгенерированный в процессе обработки; хотя иногда в этом нет необходимости, поскольку ошибка такого рода вызывает немедленную остановку обработки, и в этом случае сообщение об ошибке будет видно в том же терминале, где мы запустили «context».

```
3     \setuppapersize % Paper size
4       [S5]
5
6     \setupbodyfont
7       [modern,12pt] % Main font
8
9     \setuphead      % Chapter titles in bold
10      [chapter]
11      [style=\bfc]
12
13 >>  \startext  % Begin the document
14
15    \startchapter[title=The rain in Spain]
16
17    How kind of you to let me come.
18    Now once again, where does it rain?
19    On the plain, on the plain.
20    And where's that blasted plain?
21    In Spain, in Spain.
22    The rain in Spain stays mainly in the plain.
23    The rain in Spain stays mainly in the plain.

mtx-context     | fatal error: return code: 256
```

Рисунок 2.2 Screen output in the case of a compilation error

Например, если в нашем тестовом файле «rain.tex» по ошибке вместо \starttext мы написали \starttext (только с одним „t”), очень распространенная ошибка, при запуске «context rain» обработка будет остановлена и в терминале мы увидим информацию, показанную на figure 2.2. Здесь мы видим, что строки нашего исходного файла пронумерованы, и в одной из них, в данном случае под номером 13, между номером и строкой текста, компилятор добавил «>>», чтобы указать, что это строка, где он обнаружил ошибку. Файл «rain.log» даст нам больше подсказок. В нашем примере это не такой уж большой файл, поскольку компилируемый исходный код значительно сокращен; в других случаях он может содержать огромное количество информации. Но мы должны погрузиться в это. Если мы откроем «rain.log» в текстовом редакторе, мы увидим, что он сохранил всё, что делает ConTEXt. Нам нужно найти там строку, которая начинается с предупреждения об ошибке, и для этого мы можем использовать функцию поиска текстового редактора. Мы будем искать «tex error», и это приведет нас к следующим строкам:

```
tex error       > tex error on line 13 in file |
                  /home/jq/context/docs/rain.tex: ! Undefined control sequence

l.13 \startext
              % Begin the document
```

Примечание: первая строка, сообщающая нам об ошибке в файле «rain.log», очень длинная. Чтобы он выглядел хорошо, учитывая ширину страницы, я разделил её на две части. Символ „|” показывает точку, в которой я её разделил.

Если мы обратим внимание на три строки сообщения об ошибке, то увидим, что в первом сообщается, какой номер строки привел к ошибке (строка 13) и что это за ошибка: «Неопределенная управляющая последовательность» «Undefined control sequence»,или, что то же самое: неизвестная последовательность управления, другими словами, неизвестная команда. Две следующие строки файла журнала показывают нам строку 13, разделенную в точке, в которой возникла ошибка. Так что нет никаких сомнений в том, что ошибка кроется в \startext. Мы внимательно его прочитаем и, если повезёт и накопим опыт, то мы поймём, что написали «startext», а не «starttext» (с двойной буквой „t”).

Подумайте о том факте, что компьютеры очень хороши и очень быстро выполняют инструкции, но очень медленно читают наши мысли, и слово «startext» не то же самое, что «starttext». Программа знает, как выполнить последнее, а не первое. Она не знает, что с этим делать.

В других случаях найти ошибку будет не так-то просто. Особенно, когда ошибка состоит в том, что что-то началось, но не указано, где это должно закончиться. Иногда вместо поиска «tex error» в файле «`.log`» нам следует искать звездочку. Этот символ в начале строки файла является не столько фатальной ошибкой, сколько предупреждением. Однако предупреждения могут быть полезны для поиска ошибки.

И если информации в файле «`.log`» будет недостаточно, нам нужно будет просмотреть наш основной файл по крупицам в поисках ошибки. Хорошая стратегия для этого - изменить расположение команды `\stoptext`. Помните, что ConTEXt прекращает обработку текста, когда находит эту команду. Следовательно, если я помещаю `\stoptext` более или менее в середине файла и компилирую его, будет скомпилирована только первая половина; если ошибка повторится, то я знаю, что она находится в первой половине исходного файла, а если нет, то это означает, что она находится во второй половине ... и так далее, по крупицам, меняя расположение `\stoptext` команда, мы сможем найти, где ошибка. Как только мы ее нашли, мы можем попытаться понять и исправить ее или, если мы не можем понять, почему возникла ошибка, по крайней мере, обнаружив, где она находится, мы можем попробовать написать что-то другим способом, чтобы избежать ее воспроизведения. Это последнее решение, конечно, применимо только в том случае, если мы являемся автором. Если мы просто наберем текст для кого-то еще, мы не сможем его изменить, и нам придется продолжать исследование, пока мы не обнаружим причины ошибки и ее возможное решение.

На практике, когда с помощью ConTEXt создается относительно длинный документ, он обычно время от времени компилируется по мере того, как документ готовится, так что, если он выдаст ошибку, мы будем более или менее ясно понимаем новую часть, поскольку в последний раз мы обработал файл, и почему он выдал ошибку.

Глава 3

Команды и другие базовые концепции ConTEXt

Мы уже видели, что в исходном файле, а также в фактическом содержании нашего будущего отформатированного документа мы находим инструкции, необходимые для объяснения ConTEXt того, как мы хотим преобразовать нашу рукопись. Эти инструкции можно назвать «командами», «макросами» или «управляющими последовательностями».

С точки зрения внутреннего функционирования ConTEXt (фактически, функционирования TEX), есть разница между *примитивами* и *макросами*. Примитив – это простая инструкция, которую нельзя разбить на другие более простые инструкции. Макрос – это инструкция, которая может быть разбита на другие более простые инструкции, которые, в свою очередь, также, возможно, могут быть разбиты на другие, и так далее и так далее. Фактически, большинство инструкций ConTEXt являются макросами. С точки зрения программиста, разница между макросами и примитивами важна. Но с точки зрения пользователя проблема не так важна: в обоих случаях у нас есть инструкции, которые выполняются без необходимости беспокоиться о том, как они работают на низком уровне. Поэтому в документации ConTEXt обычно говорится о *команде*, когда она учитывает точку зрения пользователя, и о *макросе*, когда она принимает точку зрения программиста. Поскольку во введении мы рассматриваем только точку зрения пользователя, я буду использовать любой термин, считая их синонимами.

Команды - это приказы, отданные ConTEXt что-либо сделать; через них мы контролируем работу программы. Таким образом, KNUTH, отец TEX, использует термин *управляющие последовательности* для обозначения как примитивов, так и макросов, и я думаю, что это самый точный термин из всех. Я буду использовать его, когда считаю, что важно различать *управляющие символы* и *управляющие слова*.

Инструкции ConTEXt в основном бывают двух видов: зарезервированные символы и собственно так называемые команды.

3.1 Зарезервированные символы ConTEXt

Когда ConTEXt читает исходный файл, состоящий только из текстовых символов, поскольку это текстовый файл, ему необходимо каким-то образом различать, какой текст на самом деле нужно форматировать, и какие инструкции он должен выполнять. Зарезервированные символы ConTEXt позволяют ему делать это различие. В принципе, ConTEXt предполагает, что каждый символ в исходном файле является текстом для обработки, если только это не один из 11 зарезервированных символов, которые следует рассматривать как *инструкцию*.

Всего 11 инструкций? Нет. Всего 11 зарезервированных символов; но поскольку один из них, символ «\», имеет функцию преобразования символа или символов, непосредственно следующих за ним,

в инструкцию, то на самом деле потенциальное количество команд не ограничено. ConTEXt имеет около 3000 команд (включая команды, эксклюзивные для Mark II, Mark IV, и те, которые являются общими для обеих версий).

Зарезервированные символы следующие:

\ % { } # ~ | $ _ ^ &

ConTEXt интерпретирует их следующим образом:

\ Этот символ является для нас самым важным: он указывает на то, что то, что идет сразу после, следует интерпретировать не как текст, а как инструкцию. Он называется «escape-символом» или «escape-последовательностью» (хотя он не имеет ничего общего с клавишей «Esc», которая есть на большинстве клавиатур). [1]

% Сообщает ConTEXt, что то, что следует до конца строки, является комментарием, который не должен обрабатываться или включаться в окончательный форматированный файл. Добавление комментариев в исходный файл чрезвычайно полезно. Комментарий может помочь объяснить, почему что-то было сделано определенным образом, и это очень полезно для завершенных исходных файлов, учитывая более позднюю редакцию, когда иногда мы не можем вспомнить, почему мы сделали то, что мы сделали; или это также может помочь нам в качестве напоминания о том, что нам, возможно, придется пересмотреть. Его даже можно использовать для определения причины определенной ошибки в исходном файле, поскольку, помещая метку комментария в начале строки, мы исключаем эту строку из компиляции и можем увидеть, была ли это та строка, которая вызвала ошибку; его также можно использовать для хранения двух разных версий одного и того же макроса и таким образом получить разные результаты после компиляции; или чтобы предотвратить компиляцию фрагмента, в котором мы не уверены, но не удаляя его из исходного файла на случай, если мы захотим вернуться к нему позже ... и т. д. Как только мы открыли возможность того, что наш исходный файл содержит текст, который никто, кроме нас самих, не должен видеть, наше использование этого символа ограничено только нашим собственным воображением. Признаюсь, это одна из тех утилит, которых мне больше всего не хватает, когда единственным средством для написания текста является текстовый процессор.

{ Этот символ открывает группу. Группы - это блоки текста, на которые влияют определенные функции. О них мы поговорим в section 3.8.1.

} Этот символ закрывает группу, ранее открытую с помощью {.

Этот символ используется для определения макросов. Это относится к аргументам макроса. См. section 3.7.1 в этой главе.

~ Вводит пробел в документ, чтобы предотвратить разрыв строки, что означает, что два слова, разделенные символом ~, всегда будут оставаться в одной строке. Об этой инструкции и о том, где ее использовать, мы поговорим в section 11.3.1.

| Этот символ используется для обозначения того, что два слова, соединенные разделяющим элементом, составляют составное слово, которое можно разделить по слогам на первый компонент, но не на второй компонент. См. section 10.4.

[1] В компьютерной терминологии ключ, который влияет на интерпретацию следующего символа, называется escape-символом. В отличие от этого, клавиша escape на клавиатуре называется так, потому что она генерирует символ 27 в коде ASCII, который используется как escape-символ в этой кодировке. Сегодня использование клавиши Escape больше связано с идеей отмены текущего действия.

`$` Этот символ - переключатель математического режима. Он включает этот режим, если он не был включен, или отключает его, если он был включен. В математическом режиме ConTEXt применяет некоторые шрифты и правила, отличные от обычных, с целью оптимизации написания математических формул. Несмотря на то, что написание математики - очень важное применение ConTEXt я не буду развивать это во введении. Как литератор, мне не до этого!

`_` Этот символ используется в математическом режиме для обозначения нижнего индекса. Так, например, чтобы получить x_1, нам нужно написать `$x_1$`.

`^` Этот символ используется в математическом режиме для обозначения трех верхних индексов. Так, например, чтобы получить $(x+i)^{n^3}$, нам нужно написать `$(x+i)^{n^3}$`.

`&` В документации ConTEXt говорится, что это зарезервированный символ, но не объясняется, почему. В Plain TEX этот символ в основном используется для двух целей: он используется для выравнивания столбцов в базовых средах таблиц и в математическом контексте, так что все последующее следует рассматривать как обычный текст. Во вводном руководстве «ConTEXt Mark IV, an Excursion», хотя не говорится, для чего он предназначен, есть примеры его использования в математических формулах, хотя и не в том виде, в каком он был в Plain TEX, но для выравнивания столбцов в сложных функциях. Поскольку я литературный человек, я не чувствую, что могу проводить дальнейшие тесты, чтобы увидеть, для чего конкретно используется этот зарезервированный символ.

Можно предположить, что при выборе символов, которые будут зарезервированы, они будут символами, доступными на большинстве клавиатур, но обычно не используемыми в письменных сценариях. Однако, хотя это не так часто, всегда есть вероятность, что некоторые из них будут фигурировать в наших документах, например, когда мы хотим написать, что что-то стоит 100 долларов ($100) или что в Испании процент водителей возраст старше 65 лет составлял 16% в 2018 году. В этих случаях мы не должны писать зарезервированный символ напрямую, а использовать *команду*, которая будет правильно выводить зарезервированный символ в окончательный документ. Команда для каждого из зарезервированных символов находится в table 3.1.

Зарезервированные символы	Команды, генерирующие их
\	`\backslash`
%	`\%`
{	`\{`
}	`\}`
#	`\#`
~	`\lettertilde`
|	`|`
$	`\$`
_	`\_`
^	`\letterhat`
&	`\&`

Таблица 3.1 Написание зарезервированных символов

Другой способ получить зарезервированные символы - использовать команду `\type`. Эта команда отправляет то, что принимает в качестве аргумента, в окончательный документ без какой-либо обработки и, следовательно, без его интерпретации. В итоговом документе текст, полученный из `\type`, будет показан моноширинным шрифтом, типичным для компьютерных терминалов и пишущих машинок.

Обычно мы заключаем текст, который должен отображать `\type`, в фигурные скобки. Однако, когда сам этот текст включает открывающие или закрывающие фигурные скобки, вместо них мы должны заключить текст между двумя равными символами, которые не являются частью текста, составляющего аргумент `\type`. Например: `\type*{*` или `\type+}+`.

Если по ошибке мы используем один из зарезервированных символов напрямую, кроме той цели, для которой он предназначен, потому что мы забыли, что это зарезервированный символ и не может использоваться как обычный, тогда могут произойти три вещи:

1. Чаще всего возникает ошибка при компиляции.

2. Получаем неожиданный результат. Это особенно характерно для «~» и «%»; в первом случае вместо «~», который мы ожидали в окончательном документе, будет вставлен пробел; и в последнем случае все в той же строке перестанет обрабатываться, начиная с «%». Неправильное использование «\» также может привести к неожиданному результату, если он или символы сразу после него составляют команду, о которой ConTEXt знает. Однако чаще всего, когда мы неправильно используем «\», мы получаем ошибку компиляции.

3. Проблем не возникает: это происходит с тремя зарезервированными символами, используемыми в основном в математике (_ ^ &): если они используются вне этой среды, они рассматриваются как обычные символы.

Пункт 3 - это мой вывод. По правде говоря, я не нашел нигде в документации ConTEXt где говорилось бы, где эти зарезервированные символы могут использоваться напрямую; однако в своих тестах я не обнаружил никаких ошибок, когда это было сделано; в отличие, например, от LATEX.

3.2 Сами команды

Сами команды всегда начинаются с символа «\». В зависимости от того, что идет сразу после escape-последовательности, различают:

а. **Управляющие символы.** Управляющий символ начинается с escape-последовательности („\”) и состоит исключительно из символа, отличного от буквы, например, „`\,`”, «`\1`», «`\'`» или «`\%`». Любой символ или символ, не являющийся буквой в строгом смысле этого слова, может быть контрольным символом, включая числа, знаки препинания, символы и даже пробел. В этом документе, чтобы представить пустое пространство (пробел), когда его присутствие необходимо выделить, я использую символ ␣. Фактически, «`\␣`» (обратная косая черта, за которой следует пробел) является обычно используемым управляющим символом, как мы скоро увидим.

Пробел или пробел – это символ „невидимый”, что является проблемой в таком документе, где иногда нам нужно четко указать, что нужно записать в исходный файл. Knuth уже знал об этой проблеме, и в своей „The TEXBook” он начал обычай представлять значимые пробелы с помощью символа «␣». Так, например, если мы хотим показать, что два слова в исходном файле должны быть разделены двумя пробелами, мы должны написать „word1␣␣word2”.

б. **Управляющие слова.** Если символ, следующий за обратной косой чертой, является буквой в собственном смысле слова, команда будет *Управляющим словом*. эта группа команд является самой многочисленной и ее особенностью является то, что имя команды может состоять только из букв; цифры, знаки препинания или любые другие символы не допускаются. Только строчные или прописные буквы. С другой стороны, помните, что ConTEXt делает различие между нижним и верхним регистрами, а это означает, что команды `\mycommand` и `\MyCommand` различны. Но `\MyCommand1` и `\MyCommand2` будут считаться одинаковыми, поскольку „1” и „2” не являются буквами, а значит не являются частью имен команд.

Справочное руководство ConTEXt не содержит правил для имен команд, как и остальные «manuals», включенные в «ConTEXt Standalone». В предыдущем абзаце я сделал вывод, основанный на том, что происходит в TEX (где, кроме того, такие символы, как гласные с ударением, которые не встречаются в английском алфавите, не считаются «буквами»). Это правило позволяет предложить хорошее объяснение поглощения пробелов после имени команды.

Когда ConTEXt читает исходный файл и находит escape-символ («\»), он знает, что за ним последует команда. Затем он считывает первый символ, следующий за escape-последовательностью. Если это не буква, это означает, что команда является управляющим символом и состоит только из этого первого символа. Но с другой стороны, если первый символ после escape-последовательности – это

буква, то ConTEXt будет продолжать читать каждый символ, пока не найдет первую не букву, а затем он узнает, что имя команды закончилось. Вот почему имена команд, которые являются управляющими словами, не могут содержать символы, не являющиеся буквами.

Когда «non-letter» в конце имени команды является пробелом, предполагается, что пробел не является частью обрабатываемого текста, а был вставлен исключительно для указания того, где заканчивается имя команды, тогда ConTEXt избавляется от этого пространства. Это производит эффект, который удивляет новичков в ConTEXt, потому что когда эффект рассматриваемой команды подразумевает запись чего-либо в окончательном документе, письменный вывод команды соединяется со следующим словом. Например, следующие два предложения в исходном файле

```
Знание \TeX помогает в изучении \ConTeXt.
Знание \TeX, хотя и не обязательно, помогает в изучении \ConTeXt
```

соответственно дают следующие результаты:

Знание TEXпомогает в изучении ConTEXt.

Знание TEX, хотя и не обязательно, помогает в изучении ConTEXt.[1]

Обратите внимание, как в первом случае слово «TEX» связано со следующим за ним словом, но не во втором случае. Это связано с тем, что в первом случае в исходном файле первая «non-letter» после имени команды `\TeX` была пустым пространством, подавленным, поскольку ConTEXt предполагал, что он был там только для обозначения конца имени команды, в то время как в во втором экземпляре была запятая, и поскольку это не пробел, он не был подавлен.

С другой стороны, эта проблема не решается простым добавлением лишнего пробела и написанием, например,

`Знание \TeX␣␣помогает в изучении \ConTeXt`[2].

не решит проблему, потому что правило ConTEXt (которое мы увидим в section 4.2.1) заключается в том, что пустое пространство поглощает все пробелы и табуляции, которые следуют за ним. Поэтому, когда у нас возникает эта проблема (которая, к счастью, случается не слишком часто), мы должны убедиться, что первое «небуквенное» имя после имени команды не является пробелом. На это есть два кандидата:

- Зарезервированные символы «{}». Зарезервированный символ «{», как я уже сказал, открывает группу, а «}» закрывает группу, поэтому последовательность «{}» вводит пустую группу. Пустая группа не влияет на окончательный документ, но помогает ConTEXt узнать, имя команды до её завершения. Или мы могли бы также создать группу вокруг рассматриваемой команды, например, написав «{`\TeX`}». В любом случае результатом будет то, что первое „non-letter” после `\TeX` не будет пробелом.
- Контрольный символ «`\␣`» (обратная косая черта, за которой следует пробел, см. page 36). Эффект этого управляющего символа заключается во вставке пустого места в окончательный документ. Чтобы правильно понять логику ConTEXt, возможно, стоит потратить некоторое время, чтобы посмотреть, что происходит, когда ConTEXt встречает управляющее слово (например, `\TeX`), за которым следует управляющий символ (например, «`\␣`»):
  - ConTEXt встречает символ \ за которым следует „T”, и зная, что он стоит перед контрольным словом, он продолжает читать символы, пока не дойдет до «небуквенного», что происходит, когда доходит до символа \ , представляющего следующий символ управления.
  - Как только он узнает, что имя команды `\TeX`, он запускает команду и печатает TEX в окончательном документе. Затем он возвращается к точке, в которой он остановил чтение, чтобы проверить символ сразу после второй обратной косой черты.

[1] **Примечание:** два соглашения соблюдаются в тех случаях, когда, чтобы проиллюстрировать что-то в этом введении, написан фрагмент исходного кода, а также результат его компиляции: иногда код и результат его компиляции размещаются рядом друг с другом в абзаце из двух столбцов; в других случаях код написан в темно-пурпурном оттенке, который обычно используется в этом документ для представления команд ConTEXt а результат его компиляции - красным цветом.

[2] Что касается символа «␣», вспомните примечание на page 36.

- Он проверяет, является ли это пустым пространством, что означает «не буква», что означает, что последовательность управления именно такая, поэтому он может ее запустить. Он делает это и вставляет пустое место.
- Наконец, он снова возвращается к точке, в которой он прекратил чтение (пустое пространство, которое было символом управления), и продолжает обрабатывать исходный файл оттуда и далее.

Я объяснил этот механизм довольно подробно, так как устранение пробелов часто удивляет новичков. Однако следует отметить, что проблема относительно незначительна, поскольку контрольные слова обычно не печатаются непосредственно в окончательном документе, но влияют на формат и внешний вид. Напротив, контрольные символы довольно часто печатают что-то в окончательном документе.

Существует третья процедура, позволяющая избежать проблемы с пустым пространством, которая состоит в определении (стиль TEX) аналогичной команды и включении «небуквенного» в конце имени команды. Например, такая последовательность:

```
\def\txt-{\TeX}
```

создаст команду с именем `\txt`, которая будет действовать точно так же, как команда `\TeX`, и будет работать правильно только в том случае, если вызывается с дефисом после нее `\txt-`. Этот дефис технически не является частью имени команды, но он не будет работать, если за именем не будет стоять дефис. Почему это так, связано с механизмом определения макросов TEX, и это слишком сложно объяснять здесь. Но это работает: как только эта команда определена, каждый раз, когда мы используем `\txt-`, ConTEXt заменяет её на `\TeX`, удаляя дефис, но используя его внутренне, чтобы знать, что имя команды уже закончено, поэтому сразу после него появляется пустое пространство, которое не будет удалено.

Этот «трюк» не будет корректно работать с командой `\define`, которая является специальной командой ConTEXt для определения макросов.

3.3 Область действия команд

3.3.1 Команды

Многие из команд ConTEXt, особенно те, которые влияют на функции форматирования шрифтов (полужирный, курсив, маленькие заглавные буквы и т.д.), Включают определенную функцию, которая остается включенной до тех пор, пока не встретится другая команда, которая отключает её или включает другую функцию, несовместимую с ней. Например, команда `\bf` включает полужирный шрифт, и он будет оставаться активным до тех пор, пока не найдет *несовместимую* команду, например, `\tf` или `\it`.

Команды такого типа не должны принимать никаких аргументов, поскольку они не предназначены для применения только к определенному тексту. Как будто они ограничены *включением* любой функции (полужирный, курсив, без засечек, определенный размер шрифта и т.д.).

Когда эти команды выполняются в *группе* (см. section 3.8.1), они также теряют свою действенность, когда группа, в которой они выполняются, закрывается. Поэтому часто для того, чтобы эти команды влияли только на часть текста, нужно сгенерировать группу, содержащую эту команду и текст, на который мы хотим, чтобы она повлияла. Группа создается заключением её в фигурные скобки. Поэтому следующий текст

```
In {\it The \TeX Book}, {\sc Knuth}
explained everything you need to know
about \TeX.
```

In *The TEXBook*, Knuth explained everything you need to know about TEX.

создает две группы: одну для определения области действия команды `\it` (курсив), а другую - для определения области действия команды `\sc` (маленькие заглавные буквы).

В отличие от этого вида команд, есть другие, которые из-за производимого ими эффекта или по другим причинам требуют четкого указания того, к какому тексту они должны применяться. В этих

случаях текст, на который будет воздействовать команда, заключен в фигурные скобки *сразу после команды*. В качестве примера мы могли бы упомянуть `\framed`: эта команда рисует рамку вокруг текста, который принимает в качестве аргумента, так что

```
\framed{Tweedledum and Tweedledee}
```

будет производить

Tweedledum and Tweedledee

Обратите внимание, что хотя в первой группе команд (тех, которые требуют аргумента) фигурные скобки также иногда используются для определения поля действия, это не обязательно для работы команды. Команда предназначена для применения с того места, где она появляется. Таким образом, при определении области применения с помощью скобок команда помещается *внутрь этих скобок*, в отличие от второй группы команд, где скобки, обрамляющие текст, к которому должна применяться команда, идут *после* команды.

В случае команды `\framed` очевидно, что для создаваемого ею эффекта требуется аргумент - текст, к которому она должна применяться. В других случаях от программиста зависит, относится ли команда к тому или иному типу. Так, например, то, что делают команды `\it` и `\color` , очень похоже: они применяют функцию (формат или цвет) к тексту. Но было принято решение запрограммировать первую без аргумента, а вторую как команду с аргументом.

3.3.2 Команды

Существуют определенные команды, которые определяют их область действия, точно указывая точку, в которой они начинают применяться, и точку, в которой они перестают это делать. Таким образом, эти команды бывают парами: одна указывает, когда команда должна быть активирована, а другая - когда это действие должно быть прекращено. «start», за которым следует имя команды, используется для обозначения начала действия, а «stop», за которым также следует имя команды, для обозначения конца. Так, например, команда «`itemize`» становится `\startitemize`, чтобы указать начало *итемизации*, и `\\stopitemize`, чтобы указать, где она заканчивается.

В официальной документации ConTEXt нет специального названия для этих пар команд. Справочное руководство и введение просто называют их «старт ... стоп». Иногда их называют *средами*, это название LaTeX дает аналогичным конструкциям, хотя у этого есть недостаток, заключающийся в том, что в ConTEXt термин «среда» используется для чего-то ещё (особого типа файла, который мы увидим, когда будем говорить про многофайловые проекты в section 4.6). Тем не менее, поскольку термин «среда» ясен, а контекст позволит легко различить, говорим ли мы о *командах среды* или *файлах среды*, я буду использовать этот термин.

Таким образом, среды состоят из команды, которая их открывает или запускает, и другой, которая закрывает или завершает их. Если исходный файл содержит команду на открытие среды, которая не закрывается позже, обычно возникает ошибка. [1] С другой стороны, такие ошибки труднее найти, поскольку ошибка может возникнуть далеко за пределами того места, где происходит команда открытия. Иногда файл «`.log`» покажет нам строку, с которой начинается неправильно закрытая среда; но в других случаях отсутствие закрытия среды означает, что ConTEXt неверно интерпретирует определенный отрывок, а не в этой ошибочной среде, а это означает, что файл «`.log`» не очень помогает нам в поиске проблемы.

Среды могут быть вложенными, то есть другая среда может быть открыта в существующей среде, хотя в случае наличия вложенных сред каждую среду необходимо закрыть внутри той среды, в которой она была открыта. Другими словами, порядок, в котором закрываются среды, должен соответствовать порядку, в котором они были открыты. Я считаю, что это должно быть ясно из следующего примера:

[1] Хотя не всегда; это зависит от рассматриваемой среды и от ситуации в остальной части документа. В этом отношении ConTEXt отличается от LaTeX, который намного строже.

```
\startSomething
  ...
  \startSomethingElse
    ...
    \startAnotherSomethingElse
      ...
    \stopAnotherSomethingElse
  \stopSomethingElse
\stopSomething
```

В этом примере вы можете увидеть, как среда «AnotherSomethingElse» была открыта внутри среды «SomethingElse» и также должна быть закрыта внутри нее. В противном случае при компиляции файла возникнет ошибка.

В общем, команды, разработанные как *среды*, реализуют некоторые изменения, предназначенные для применения к блокам текста размером не меньше абзаца. Например, «более узкая» среда, изменяющая поля, имеет смысл только при применении на уровне абзаца; или среда «framedtext», которая обрамляет один или несколько абзацев. Эта последняя среда может помочь нам понять, почему некоторые команды разработаны как среды, а другие - как отдельные команды: если мы хотим поместить одно или несколько слов в одну строку, мы должны использовать команду \framed, но если то, что мы хотим, обрамить - это целый абзац (или несколько абзацев), тогда мы будем использовать среду «framedtext».

С другой стороны, текст, расположенный в определенной среде, обычно составляет *группу* (см. section 3.8.1), что означает, что, если внутри среды обнаружена команда активации, из тех команд, которые применяются ко всему последующему тексту, эта команда будет применяться только до конца среды, в которой он находится; и на самом деле ConTEXt имеет безымянную *среду*, начинающуюся с команды \start (никакой другой текст не следует; просто *start*. Вот почему я называю ее *безымянной средой*) и заканчивающуюся командой \stop. Я подозреваю, что единственная её функция - создание группы.

Я нигде в документации ConTEXt не читал, что одним из эффектов сред является группировка их содержимого, но это результат моих тестов с рядом предопределенных сред, хотя я должен признать, что мои тесты не были слишком исчерпывающими. Я просто проверил несколько случайно выбранных сред. Мои тесты показывают, однако, что такое утверждение, если оно истинно, будет справедливым только для некоторых предопределенных сред: те, которые созданы с помощью команды \definestartstop (объяснено в section 3.7.2), не создают никаких групп, если только при определении нового В среду мы включаем команды, необходимые для создания группы (см. section 3.8.1).

Также предполагаю, что среда, которую я назвал *безымянной* (\\start) средой, предназначена только для создания группы: она действительно создает группу, но я не знаю, есть ли у нее какое-либо другое применение. Это одна из недокументированных команд в справочном руководстве.

3.4 Опции командных операций

3.4.1 Команды

Многие команды могут работать более чем одним способом. В таких случаях всегда существует предопределенный способ работы, который можно изменить, указав параметры, соответствующие желаемой операции, в скобках после имени команды.

Мы находим хороший пример того, что я только что сказал, с помощью команды \\framed, упомянутой в предыдущем разделе. Эта команда рисует рамку вокруг текста, который принимает в качестве аргумента. По умолчанию рамка имеет высоту и ширину текста, к которому она применяется; но мы можем указать другую высоту и ширину. Таким образом, мы можем видеть разницу между тем, как работает \\framed по умолчанию:

```
\framed{Tweedledum}
```

Tweedledum

и как работает настроенная версия:

```
\framed
  [width=3cm, height=1cm]
  {Tweedledum}
```

Во втором примере между квадратными скобками мы указали конкретную ширину и высоту фрейма, который окружает текст, который он принимает в качестве аргумента. В скобках различные параметры конфигурации разделяются запятой; пробелы и даже разрывы строк (если они не являются двойным разрывом строки) между двумя или более параметрами не принимаются во внимание, поэтому, например, следующие четыре версии одной и той же команды дают точно такой же результат:

```
\framed[width=3cm,height=1cm]{Tweedledum}

\framed[width=3cm,    height=1cm]{Tweedledum}

\framed
  [width=3cm, height=1cm]
  {Tweedledum}

\framed
  [width=3cm,
    height=1cm]
  {Tweedledum}
```

Очевидно, что финальная версия наиболее удобна для чтения: мы с первого взгляда можем увидеть, сколько существует опций и как они используются. В примере, подобном этому, только с двумя вариантами, возможно, это может показаться не таким важным; но в случаях, когда есть длинный список опций, если каждая из них имеет свою собственную строку в исходном файле, это упрощает *понимание* того, что исходный файл просит ConTEXt сделать, а также, при необходимости, обнаружить потенциальную ошибка. Следовательно, этот последний формат (или аналогичный) для написания команд является «предпочтительным» для пользователей.

Что касается синтаксиса параметров конфигурации, см. (section 3.5).

3.4.2 Команды

Мы уже видели, что команды, которые поддерживают различные возможности того, как они работают, всегда имеют способ работы по умолчанию. Если одна из этих команд вызывается несколько раз в нашем исходном файле, и мы хотели бы изменить значение по умолчанию для них всех, вместо того, чтобы изменять эти параметры каждый раз при вызове команды, гораздо удобнее и эффективнее изменить значение по умолчанию. Для этого почти всегда доступна команда, имя которой начинается с `\\setup`, за которым следует имя команды, параметры которой по умолчанию мы хотим изменить.

Команда `\\framed`, которую мы использовали в качестве примера в этом разделе, продолжает оставаться хорошим образцом. Итак, если мы используем много фреймов в нашем документе, но все они нуждаются в точных измерениях, тогда было бы лучше перенастроить, как работает `\\framed`, сделав это с помощью `\\setupframed`. Таким образом

```
\setupframed
  [
    width=3cm,
    height=1cm
  ]
```

будет обеспечено, что с этого момента каждый раз, когда мы вызываем `\\framed`, по умолчанию он будет генерировать рамку шириной 3 см и высотой 1 см, без необходимости указывать это каждый раз явно.

В ConTEXt около 300 команд, которые позволяют нам настраивать работу других команд. Таким образом, мы можем настроить работу по умолчанию фреймов `\(\framed)`, списков («`itemize`»), заголовков глав `\(\chapter)` или заголовков разделов `\(\section)` и т.д.

3.4.3 Настройка индивидуальных версий настраиваемых команд (`\defineSomething`)

Продолжая пример `\framed` , очевидно, что если в нашем документе используются несколько видов фреймов, каждый с разными размерами, идеальным было бы то, что мы могли бы заранее *предопределить* различные конфигурации `\framed` и связать их с определенным именем, чтобы мы могли использовать тот или иной из них по мере необходимости. Мы можем сделать это в ConTEXt с помощью команды `\defineframed`, синтаксис которой:

```
\defineframed[Name][Configuration]
```

где *Name* - это имя, присвоенное конкретному типу настраиваемого фрейма; и *Configuration* - это конкретная *Конфигурация*, связанная с этим именем.

Результатом всего этого будет то, что указанная конфигурация будет связана с установленным нами именем, которое, по сути, будет работать так, как если бы это была новая команда, и мы можем использовать это в любом контексте, где мы могли бы использовать исходную команду `\(\framed)`.

Эта возможность существует не только для конкретного случая команды `\framed`, но и для многих команд, у которых есть возможность `\setup`. Комбинация `\defineSomething` +`\setupSomething` – это механизм, который придает ConTEXt исключительную мощь и гибкость. Если мы подробно рассмотрим, что делает команда `\defineSomething`, мы увидим, что:

- Прежде всего, он клонирует определенную команду, которая поддерживает множество конфигураций.
- Он связывает этот клон с именем новой команды.
- Наконец, он устанавливает предопределенную конфигурацию для клона, отличную от того, как была настроена исходная команда.

В приведенном нами примере мы настраивали специальный фрейм одновременно с его созданием. Но мы также можем сначала создать его, а потом настроить, потому что, как я уже сказал, после создания клона его можно использовать там, где можно было бы использовать оригинал. Так, например, если мы создали фрейм под названием «`MySpecialFrame`», мы можем настроить его с помощью `\setupframed`, указывающего фактический фрейм, который мы хотим настроить. В этом случае команда `\setup` примет новый аргумент с именем настраиваемого фрейма:

```
\defineframed[MySpecialFrame]

\setupframed
  [MySpecialFrame]
  [ ... ]
```

3.5 Краткое описание синтаксиса команд и параметров, а также использования квадратных и фигурных скобок при их вызове

Подводя итог тому, что мы видели до сих пор, мы видим, что в ConTEXt

- Собственно так называемые команды всегда начинаются с символа «\».
- Некоторые команды могут принимать один или несколько аргументов.

- Аргументы, которые сообщают команде, *как* она должна работать или которые каким-то образом влияют на её работу, вводятся в квадратных скобках.
- Аргументы, которые сообщают команде, с какой частью текста она должна действовать, помещаются в фигурные скобки.

  Когда команда будет действовать только с одной буквой, как, например, в случае с командой `\buildtextcedilla` (просто для примера - „ç”, так часто используемая в каталонском языке), фигурные скобки вокруг аргумента можно опустить: команда будет применена к первому символу, не являющемуся пробелом.

- Некоторые аргументы могут быть необязательными, и в этом случае мы можем их опустить. Но мы никогда не сможем изменить порядок аргументов, ожидаемых командой.

Аргументы, заключенные в квадратные скобки, могут быть разных типов. В основном:

- Они могут принимать только одно значение, которое почти всегда будет одним словом или фразой.
- Они могут выбрать различные варианты, и в этом случае они могут:
  - Быть представленным одним словом, которое может быть символическим именем (значение которого ConTEXt знает), мерой или измерением, числом, именем другой команды и т.д.
  - Состоят из имен переменных, которым необходимо присвоить значение. В этом случае официальное определение команды (см. section ??) всегда сообщает нам, какое значение ожидает каждая из опций.
    - Если значение, ожидаемое параметром, является текстом, оно может содержать пробелы, а также команды. В этих случаях иногда удобно заключить значение параметра в фигурные скобки.
    - Когда значение, ожидаемое параметром, является командой, обычно мы можем указать несколько команд в качестве значения параметра, хотя иногда нам нужно заключить все команды, назначенные параметру, в фигурные скобки. Мы также должны заключить содержимое параметра в фигурные скобки, если какая-либо из включенных в него команд принимает параметр в квадратные скобки.

  В обоих случаях разные параметры, которые должны принимать один и тот же аргумент, будут разделены запятыми. Пробелы и разрывы строк (кроме двойных) между различными параметрами игнорируются. Пробелы и разрывы строк между различными аргументами команды также игнорируются.
- Наконец, в случае ConTEXt никогда не бывает, чтобы один и тот же аргумент одновременно принимал параметры, состоящие из слова, и параметры, состоящие из переменной, которой должно быть явно присвоено значение. Другими словами, у нас могут быть такие варианты, как

  ```
  \command[Option1, Option2, ...]
  ```

  и другие как

  ```
  \command[Variable1=value, Variable2=value, ...]
  ```

  Но мы никогда не сможем найти сочетание того и другого:

  ```
  \command[Option1, Variable1=value, ...]
  ```

3.6 Официальный список команд ConTEXt reference

Среди документации ConTEXt есть особенно важный документ со списком всех команд, указывающим для каждой из них, сколько аргументов они ожидают и какого типа, а также различные предусмотренные параметры и их допустимые значения. Этот документ называется «`setup-en.pdf`» и

создается автоматически для каждой новой версии ConTEXt. Его можно найти в каталоге под названием «tex/texmf-context/doc/context/documents/general/qrcs».

Фактически, «qrc» имеет семь версий этого документа, по одной для каждого из языков, имеющих интерфейс ConTEXt: немецкого, чешского, французского, голландского, английского, итальянского и румынского. Для каждого из этих языков в каталоге есть два документа: один с названием «setup-LangCode» (где LangCode - это буквенный код идентификации двух международных языков) и второй документ с именем «setup-mapping-LangCode». Этот второй документ содержит список команд в алфавитном порядке и указывает *прототип* команды, но без дополнительной информации о вероятных значениях для каждого аргумента.

Этот документ является фундаментальным для обучения использованию ConTEXt, потому что именно там мы можем узнать, существует ли определенная команда; это особенно полезно, учитывая command (or environment) + setupcommand + definecommand. Например, если я знаю, что с помощью команды \blank вводится пустая строка, я могу узнать, есть ли команда с именем \setupblank, которая позволяет мне её настроить, и другая, которая позволяет мне настраивать индивидуальную конфигурацию для пустых строк, (\defineblank).

«setup-en.pdf», таким образом, является основополагающим для изучения ConTEXt. Но я бы действительно предпочел, прежде всего, чтобы он сообщал нам, работает ли команда только в Mark II или Mark IV, и особенно, что если бы вместо того, чтобы сообщать нам только количество типов аргументов, которые позволяет каждая команда, она бы расскажите нам, для чего нужны эти аргументы. Это значительно уменьшило бы недостатки документации ConTEXt. Есть некоторые команды, которые допускают необязательные аргументы, которые я даже не упоминаю в этом введении, потому что я не знаю, для чего они нужны, и, поскольку они являются необязательными, нет необходимости их упоминать. Это очень расстраивает.

3.7 Определение новых команд

3.7.1 Общий механизм определения новых команд

Мы только что увидели, как с помощью \defineSomething мы можем клонировать уже существующую команду и разработать на её основе новую версию, которая будет работать во всех смыслах и целях как новая команда.

Наряду с этой возможностью, которая доступна только некоторым конкретным командам (определенно немало, но не всем), ConTEXt имеет общий механизм для определения новых команд, который является чрезвычайно мощным, хотя в некоторых случаях его использование также довольно сложно. В подобном тексте, предназначенном для начинающих, я считаю, что лучше всего представить его, начав с некоторых из его простейших применений. Самый простой из всех - связать фрагменты текста со словом, чтобы каждый раз, когда это слово появляется в исходном файле, оно заменяется связанным с ним текстом. Это позволит нам, с одной стороны, сэкономить много времени на набор текста, а с другой стороны, в качестве дополнительного преимущества, это снижает вероятность ошибок при наборе текста, гарантируя, что рассматриваемый текст всегда записывается так же.

Представим, например, что мы пишем трактат об аллитерации в латинских текстах, где мы часто цитируем латинское предложение «O Tite tute Tati tibi tanta tyranne tulisti» (О Тит Татий, ты тиран, столько ты принес на себя!). Это довольно длинное предложение, в котором два слова являются именами собственными и начинаются с заглавных букв, и где, давайте признаем это, как бы мы ни любили латинскую поэзию, нам легко «споткнуться» при ее написании. В этом случае мы могли бы просто вставить преамбулу нашего исходного файла:

```
\define\Tite{\quotation{O Tite tute Tati tibi tanta tyranne tulisti}}
```

Основываясь на таком определении, каждый раз, когда команда \Tite появляется в нашем исходном файле, она будет заменена указанным предложением, и она также будет заключена в кавычки, как и в исходном определении, что позволяет нам гарантировать, что то, как это предложение появляется, всегда будет одним и тем же. Мы могли бы также написать его курсивом, с большим размером шрифта ... как нам угодно. Важно то, что нам нужно написать его только один раз, и по всему тексту он будет воспроизводиться именно так, как был написан, столько раз, сколько мы захотим. Мы также

могли бы создать две версии команды, называемые \Tite и \tite, в зависимости от того, нужно ли писать предложение заглавными буквами или нет. Текст замены может быть чистым текстом или включать команды, или формировать математические выражения, в которых больше шансов опечатки (по крайней мере, для меня). Например, если выражение $(x_1,...,x_n)$ должно регулярно появляться в нашем тексте, мы могли бы создать команду для его представления. Например

```
\define\xvec{$(x_1,\ldots,x_n)$}
```

так что всякий раз, когда \xvec появляется в тексте, он заменяется связанным с ним выражением.

Общий синтаксис команды \define следующий:

```
\define[NumArguments]\CommandName{TeXtToReplace}
```

где

- **NumArguments** указывает количество аргументов, которые примет новая команда. Если в этом нет необходимости, как в приведенных выше примерах, это будет опущено.
- **CommandName** относится к имени, которое будет иметь новая команда. Здесь применяются общие правила, касающиеся имен команд. Имя может быть отдельным символом, не являющимся буквой, или одной или несколькими буквами без каких-либо «небуквенных» символов.
- **TextToReplace** содержит текст, который будет заменять имя новой команды каждый раз, когда он будет обнаружен в исходном файле.

Например: представим, что мы хотим написать команду, которая производит открытие делового письма. Очень простой вариант:

```
\define\LetterHeading{
  \rightaligned{Peter Smith}\par
  \rightaligned{Consultant}\par
  Maryborough, \date\par
  Dear Sir,\par
  }
```

но было бы предпочтительнее иметь версию команды, которая записывала бы имя получателя в заголовке. Это потребует использования параметра, который сообщает имя получателя новой команде. Для этого потребуется переопределить команду следующим образом:

```
\define[1]\LetterHeading{
  \rightaligned{Peter Smith}\par
  \rightaligned{Consultant}\par
  Maryborough, \date\par
  Dear Mr #1,\par
  }
```

Обратите внимание, что мы внесли два изменения в определение. Прежде всего, между ключевым словом \define и новым именем команды мы заключили 1 в квадратные скобки ([1]). Это сообщает ConTEXt, что определяемая нами команда будет иметь один аргумент. Далее, в последней строке определения команды, мы написали «Dear Mr #1,», используя зарезервированный символ «#». Это указывает на то, что в том месте текста замены, где появляется «#1», будет вставлено содержимое первого аргумента. Если бы у него было два параметра, «#1» относился бы к первому параметру, а «#2» – ко второму. Чтобы вызвать команду (в исходном файле) после имени команды, аргументы должны быть заключены в фигурные скобки, каждый аргумент должен иметь свой собственный набор. Итак, команда, которую мы только что определили, должна вызываться в тексте нашего исходного файла следующим образом:

```
\LetterHeading{Name of addressee}
```

Например: \LetterHeading{Anthony Moore}.

Мы могли бы еще больше улучшить предыдущую функцию, потому что она предполагает, что письмо будет отправлено мужчине (там написано «Уважаемый господин»), поэтому, возможно, мы могли бы включить еще один параметр, чтобы различать мужчин и женщин-адресатов. Например:

```
\define[2]\LetterHeading{
  \rightaligned{Peter Smith}\par
  \rightaligned{Consultant}\par
  Maryborough, \date\par
  #1\ #2,\par
  }
```

чтобы функция вызывалась, например, с

```
\LetterHeading{Dear Ms}{Eloise Merriweather}
```

хотя это не очень элегантно (с точки зрения программирования). Было бы предпочтительно, чтобы для первого аргумента были определены символические значения (мужчина/женщина; 0/1; м/ж), чтобы сам макрос выбирал соответствующий текст в соответствии с этим значением. Но объяснение того, как этого добиться, требует от нас более глубокого понимания, чем, я думаю, начинающий читатель может понять на данном этапе.

3.7.2 Создание новой среды

Для создания новой среды ConTEXt предоставляет команду `\definestartstop`, синтаксис которой следующий:

```
\definestartstop[Name][Options]
```

В *официальном* определении `\definestartstop` (см. section ??) есть дополнительный аргумент, который я не приводил выше, потому что он не является обязательным, и я не смог выяснить, для чего он нужен. Ни вводная «Экскурсия» ConTEXt, ни неполное справочное руководство не объясняют этого. Я предполагал, что этот аргумент (который должен быть введен между именем и конфигурацией) может быть именем некоторой существующей среды, которая будет служить исходной моделью для новой среды, но мои тесты показывают, что это предположение было неверным. Я просмотрел список рассылки ConTEXt и не увидел никакого использования этого возможного аргумента.

where

- **Name** это имя, которое будет иметь новая среда.
- **Configuration** позволяет нам настраивать поведение новой среды. У нас есть следующие значения, с которыми мы можем его настроить:
  - `before` – Команды, которые необходимо выполнить перед входом в среду.
  - `after` – Команды, которые необходимо запустить после выхода из среды.
  - `style` – стиль, который должен иметь текст нового окружения.
  - `setups` – Набор команд, созданных с помощью `\startsetups ... \stopsetups`. Эта команда и её использование не объясняются во введении.

  - `color, inbetween, left, right` – недокументированные варианты, которые мне не удалось заставить работать. Мы можем предположить, что некоторые делают, из-за их имени, например `цвета`, но из большего количества тестов, которые я провел, указывая какое-то значение для этой опции, я не увидел никаких изменений в среде.

Пример определения среды может быть следующим:

```
\definestartstop
  [TextWithBar]
  [before=\startmarginrule\noindentation,
    after=\stopmarginrule,
    style=\ss\sl
  ]

\starttext

The first two fundamental laws of human stupidity state unambiguously
that:

\startTextWithBar
```

```
  \startitemize[n,broad]

    \item Always and inevitably we underestimate the number of stupid
    individuals in the world.

    \item The probability that a given person is stupid is independent
    of any other characteristic of the same person.

  \stopitemize

\stopTextWithBar

\stoptext
```

Результат будет:

The first two fundamental laws of human stupidity state unambiguously that:

1. *Always and inevitably we underestimate the number of stupid individuals in the world.*
2. *The probability that a given person is stupid is independent of any other characteristic of the same person.*

Если мы хотим, чтобы наша новая среда была группой (section 3.8.1), чтобы любое изменение нормального функционирования ConTEXt, происходящее в ней, исчезало при выходе из среды, мы должны включить команду `\bgroup` в параметр «`before`», и команду `\egroup` в опции «`after`».

3.8 Другие базовые концепции

Помимо команд, существуют и другие понятия, которые имеют фундаментальное значение для понимания логики работы ConTEXt. Некоторые из них из-за своей сложности не подходят для введения и поэтому не будут рассматриваться в этом документе; но есть два понятия, которые следует рассмотреть сейчас: группы и размеры.

3.8.1 Группы

Группа - это четко определенный фрагмент исходного файла, который ConTEXt использует как *рабочую единицу* (что это означает, поясняется вкратце). У каждой группы есть начало и конец, которые необходимо четко указать. Группа начинается:

- Зарезервированным символом «{» or или командой `\bgroup`.
- Командой `\begingroup`
- Командой `\start`
- При открытии определенной среды (команда `\startSomething`).
- Началом математического окружения (с зарезервированного символа «$»)

и закрывается

- Зарезервированным символом «}» or или командой `\egroup`.
- Командой `\endgroup`
- Командой `\stop`
- При закрытии среды (команда `\stopSomething`).
- При выходе из математической среды (с зарезервированным символом «$»).

Некоторые команды также автоматически создают группу, например, \hbox, \vbox и, в общем, команды, связанные с созданием боксов [1]. За пределами этих последних случаев (группы, автоматически создаваемые определенными командами), способ закрытия группы должен соответствовать способу её открытия. Это означает, что группа, которая начинается с «{», должна закрываться с помощью «}», а группа, начинающаяся с \begingroup, должна быть закрыта с помощью \endgroup. У этого правила есть только одно исключение: группа, начатая с «{», может быть закрыта с помощью \egroup, а группа, начатая с \bgroup, может быть закрыта с помощью «}»; в действительности это означает, что «{» и \bgroup полностью синонимичны и взаимозаменяемы, как и для «}» и \egroup.

Команды \bgroup и \egroup были разработаны, чтобы иметь возможность определять команды для открытия группы и других для закрытия группы. Следовательно, по причинам, внутренним для синтаксиса TEX, эти группы нельзя было открывать и закрывать с помощью фигурных скобок, поскольку это привело бы к появлению несбалансированных фигурных скобок в исходном файле, и это всегда приводило бы к ошибке при компиляции.

Команды \begingroup и \endgroup,напротив, невзаимозаменяемы с фигурными скобками или командами \bgroup ... \egroup,поскольку группа, начинающаяся с \begingroup должна быть закрыта с помощью \endgroup. Эти последние команды были разработаны для более глубокой проверки ошибок. В общем, обычным пользователям их не нужно использовать.

У нас могут быть вложенные группы (группа внутри другой группы), и в этом случае порядок, в котором группы закрываются, должен соответствовать порядку, в котором они были открыты: любая подгруппа должна быть закрыта внутри группы, в которой она началась. Также могут быть пустые группы, созданные с помощью «{}». Пустая группа, в принципе, не влияет на окончательный документ, но может быть полезна, например, для обозначения конца имени команды.

Основным эффектом групп является инкапсуляция их содержания: как правило, определения, форматы и присвоения значений, которые делаются внутри группы, «забываются», когда мы покидаем группу. Таким образом, если мы хотим, чтобы ConTEXt временно изменил свой нормальный способ функционирования, наиболее эффективным способом является создание группы и изменение её функционирования внутри неё. Таким образом, когда мы выходим из группы, все предыдущие значения и форматы будут восстановлены. Мы уже видели несколько примеров этого при упоминании таких команд, как \it, \bf, \sc и т.д. Но это происходит не только с командами форматирования: группа каким-то образом изолирует своё содержимое, так что любое изменение в любом из многих внутренних переменных, которыми ConTEXt постоянно управляет, будут оставаться эффективными только до тех пор, пока мы находимся в группе, в которой произошло это изменение. Точно так же команда, определенная в группе, не будет известна за её пределами.

Итак, если обработать следующий пример

```
\define\A{B}
\A
{
  \define\A{C}
  \A
}
\A
```

мы увидим, что при первом запуске команды \A результат соответствует результату ее первоначального определения („B”). Затем мы создали группу и переопределили в ней команду \A. Если мы запустим ее сейчас в группе, команда выдаст нам новое определение (в нашем примере „C”), но когда мы выйдем из группы, в которой была переопределена команда \A, если мы запустим ее снова, она напечатает „B” еще раз. Определение, данное в группе, «забывается», как только мы из нее вышли.

Другое возможное использование групп касается тех команд или инструкций, которые предназначены для применения исключительно к символу, который написан после них. В этом случае, если мы хотим, чтобы команда применялась к более чем одному символу, мы должны заключить символы, к которым мы хотим применить команду или инструкцию, в группу. Так, например, зарезервированный символ«^», который, как мы уже знаем, преобразует следующий символ в надстрочный индекс

[1] Понятие *box* также является центральным в ConTEXt, но его объяснение не включено в это введение.

при использовании внутри математической среды; поэтому, если мы напишем, например, «4^2x», мы получим «4^2x». Но если мы напишем «4^{2x}», мы получим «4^{2x}».

Наконец: третье использование группировки - сказать ConTEXt, что то, что заключено в группу, должно рассматриваться как одно целое. Это причина, по которой ранее (section 3.5) было сказано, что в определенных случаях лучше заключить содержимое некоторой опции команды в фигурные скобки.

3.8.2 Размеры

Хотя можно было бы идеально использовать ConTEXt, не беспокоясь о размерах, тем не менее мы не сможем использовать все возможности конфигурации, не уделив им некоторого внимания. Потому что в значительной степени типографское совершенство, достигнутое TEX и его производными, заключается в большом внимании, которое система уделяет внутренним размерам. Символы имеют размеры; пространство между словами, или между строками, или между абзацами имеет размеры; линии имеют размеры; поля, верхние и нижние колонтитулы. Почти для каждого элемента на странице, о котором мы только можем подумать, будет какое-то измерение.

В ConTEXt размеры указываются десятичным числом, за которым следует единица измерения. Единицы, которые можно использовать, указаны в table 3.2.

Наименование	Имя в ConTEXt	Эквивалент
Inch	in	1 in = 2.54 cm
Centimetre	cm	2.54 cm = 1 inch
Millimetre	mm	100 mm = 1 cm
Point	pt	72.27 pt = 1 inch
Big point	bp	72 bp = 1 inch
Scaled point	sp	65536 sp = 1 point
Pica	pc	1 pc = 12 points
Didot	dd	1157 dd = 1238 points
Cicero	cc	1 cc = 12 didots
	em	
	ex	

Таблица 3.2 Единицы измерений в ConTEXt

Первые три единицы в table 3.2 – стандартные меры длины; первый используется в некоторых частях англоязычного мира, а другие - вне его или в некоторых его частях. Остальные единицы взяты из мира типографики. Последние два, для которых я не поставил эквивалента, являются относительными единицами измерения, основанными на текущем шрифте. 1 «em» равен ширине «M», а «ex» равен высоте «x». Использование мер, связанных с размером шрифта, позволяет создавать макросы, которые выглядят одинаково хорошо независимо от источника, используемого в любой момент. Вот почему в целом рекомендуется.

За очень немногими исключениями мы можем использовать любую единицу измерения, которую мы предпочитаем, поскольку ConTEXt преобразует её внутренне. Но всякий раз, когда указывается размер, необходимо указывать единицу измерения, и даже если мы хотим указать размер «0», мы должны сказать «0pt» или «0 см». Между номером и названием единицы мы можем или не можем оставлять пробел. Если единица имеет десятичную часть, мы можем использовать десятичный разделитель, либо (.), Либо запятую (,).

Измерения обычно используются как опция для какой-либо команды. Но мы также можем напрямую присвоить значение некоторой внутренней мере ConTEXt, если мы знаем её имя. Например, отступ первой строки обычного абзаца внутренне контролируется ConTEXt с помощью переменной `\parindent`. Явно присвоив значение этой переменной, мы поменяем измерение, которое ConTEXt использует с этого момента. И поэтому, например, если мы хотим устранить отступ в первой строке, нам нужно только написать в нашем исходном файле:

```
\parindent=0pt
```

Мы также могли написать `\parindent 0pt` (без знака равенства) или `\parindent0pt` без пробела между именем меры и её значением.

Однако присвоение значения непосредственно внутренней мере считается «неэлегантным». Как правило, рекомендуется использовать команды, управляющие этой переменной, и делать это в преамбуле исходного файла. Обратное приводит к исходным файлам, которые очень трудно отлаживать, потому что не все команды конфигурации находятся в одном и том же месте, и действительно трудно добиться определенной согласованности типографских характеристик.

Некоторые измерения, используемые ConTEXt, являются «эластичными», то есть, в зависимости от контекста, они могут принимать ту или иную меру. Этим мерам присваивается следующий синтаксис:

```
\MeasureName plus MaxIncrement minus MaxDecrease
```

Например

```
\parskip 3pt plus 2pt minus 1pt
```

С помощью этой инструкции мы говорим ConTEXt назначить `\parskip` (указывающее вертикальное расстояние между абзацами) обычное измерение в 3 пункта, но если этого требует состав страницы, измерение может составлять до 5 пунктов (3 плюс 2) или всего 2 балла (3 минус 1). В этих случаях ConTEXt останется выбирать расстояние для каждой страницы от минимум 2 до максимум 5 точек.

3.9 Метод самообучения для ConTEXt

Огромное количество команд и опций ConTEXt действительно ошеломляет и может дать нам ощущение, что мы никогда не научимся хорошо с ним работать. Это впечатление было бы ошибкой, потому что одним из преимуществ ConTEXt является единообразный способ обработки всех своих структур: хорошее изучение нескольких структур и более или менее знание того, для чего нужны остальные, когда нам нужна дополнительная утилита, научиться им пользоваться будет относительно легко. Поэтому я считаю это введение своего рода *тренировкой*, которая подготовит нас к проведению собственных исследований.

Чтобы создать документ с помощью ConTEXt, вероятно, необходимо знать только следующие пять вещей (мы могли бы назвать их пятеркой лучших по ConTEXt):

1. Знать, как создать исходный файл или любой проект; это объясняется в Chapter 4 этого введения.
2. Установить основной шрифт для документа и узнать основные команды для изменения шрифта и цвета (Chapter ??).
3. Знать основные команды для структурирования содержимого нашего документа, такие как главы, разделы, подразделы и т.д. Все это объясняется в Chapter 6.
4. Возможно, вы знаете, как работать со средой детализации, подробно описанной в section 12.3.
5. ... и еще немного.

В остальном всё, что нам нужно знать, это то, что это возможно. Конечно, никто не будет использовать утилиту, если не знает, что она существует. Многие из них объясняются в этом введении; но, прежде всего, мы можем неоднократно наблюдать, как ConTEXt всегда действует, когда сталкивается с определенным типом конструкции:

- Сначала будет команда, позволяющая это сделать.
- Во-вторых, почти всегда есть команда, которая позволяет нам настроить и заранее определить, как будет выполняться задача; команда, имя которой начинается с `\setup` и обычно совпадает с основной командой.
- Наконец, часто можно создать новую команду для выполнения аналогичных задач, но с другой конфигурацией.

Чтобы узнать, существуют ли эти команды или нет, посмотрите официальный список команд (см. section ??), который также проинформирует нас о параметрах конфигурации, поддерживаемых этими командами. И хотя на первый взгляд названия этих опций могут показаться загадочными, мы скоро увидим, что есть опции, которые повторяются во многих командах и работают одинаково или очень похоже во всех из них. Если у нас есть сомнения относительно того, что делает опция или как она работает, этого будет достаточно, чтобы сгенерировать документ и протестировать его. Мы также можем ознакомиться с обширной документацией по ConTEXt. Как это принято в мире бесплатного программного обеспечения, «ConTEXt Standalone» включает исходные коды почти всей своей документации в дистрибутив. Утилита, такая как «`grep`» (для систем GNU Linux), может помочь нам найти, используется ли команда или параметр, в отношении которых мы сомневаемся, в каком-либо из этих исходных файлов, чтобы у нас был под рукой пример.

Вот как было задумано изучение ConTeXt во введении подробно объясняются пять (фактически четыре) аспекта, которые я выделил, и многое другое: по мере чтения в нашем сознании сформируется четкая картина последовательности: *команда для выполнения задачи – команда, которая настраивает предыдущую – команда, которая позволяет нам создать аналогичную команду*. Мы также изучим некоторые из основных структур ConTEXt и узнаем, для чего они нужны.

Глава 4

Исходные файлы и проекты

Как мы уже знаем, при работе с ConTEXt мы всегда начинаем с текстового файла, в который, наряду с содержимым окончательного документа, включен ряд инструкций, сообщающих ConTEXt о преобразованиях, которые он должен применить для генерации нашего итоговый правильно отформатированный документ в формате PDF из исходного файла.

Думая о читателях, которые до сих пор знали, как работать только с текстовыми редакторами, я думаю, что стоит потратить некоторое время на сам исходный файл. Или, скорее, исходные файлы, поскольку бывают случаи, когда есть только один исходный файл, а другие, когда мы используем несколько исходных файлов для получения окончательного документа. В этом последнем случае мы можем говорить о «многофайловых проектах».

4.1 Кодировка исходных файлов

Исходные файлы должны быть текстовыми. В компьютерной терминологии это имя, данное файлу, содержащему только читаемый человеком текст, который не включает двоичный код. Эти файлы также называются файлами *простой текст* или *простой текст*.

Поскольку внутренне компьютерные системы обрабатывают только двоичные числа, текстовый файл на самом деле состоит из *чисел*, которые представляют *символов*. *table* используется для соединения числа с символом. Для текстовых файлов существует несколько возможных таблиц. Термин *кодировка текстового файла* относится к определенной таблице соответствия символов -, которую использует конкретный текстовый файл.

> Существование различных таблиц кодирования для текстовых файлов - следствие истории самой информатики. На ранних этапах разработки, когда память и объем памяти компьютерных устройств были недостаточными, было решено использовать таблицу под названием ASCII (расшифровывается как *„Американский стандартный код для обмена информацией”*), которая позволяла только 128 символов и была создана в 1963 году Комитетом по стандартам США. Очевидно, что 128 знаков недостаточно для представления всех знаков и символов, используемых во всех языках мира; но этого было более чем достаточно для представления английского языка, который из всех западных языков имеет меньшее количество символов, поскольку в нем не используются диакритические знаки (диакритические знаки и другие знаки выше или ниже или через другие буквы). Преимущество использования ASCII состояло в том, что текстовые файлы занимали очень мало места, так как 127 (наибольшее число в таблице) может быть представлено 7-значным двоичным числом, а первые компьютеры использовали байт как единицу измерения памяти. , 8-значное двоичное число. Любой символ в таблице поместится в один байт. Поскольку байт состоит из 8 цифр, а ASCII использует только 7 цифр, оставалось даже место для добавления некоторых других символов для представления других языков.
>
> Но когда использование компьютеров расширилось, неадекватность ASCII стала очевидной, и возникла необходимость в разработке em альтернативных таблиц, которые включали символы, не известные в английском алфавите, такие как испанский „ñ”, гласные с ударением, каталонский или французский „ç” и т. д. С другой стороны, не было первоначального соглашения относительно того, какими должны быть эти *альтернативные таблицы* ASCII, поэтому различные специализированные

компьютерные компании постепенно взялись за решение этой проблемы самостоятельно. Таким образом, были созданы не только конкретные таблицы для разных языков или групп языков, но и разные таблицы в зависимости от компании, которая их создала (Microsoft, Apple, IBM и т.д.).

Идея создания единой таблицы, которую можно было бы использовать для всех языков, возникла только с увеличением компьютерной памяти и более низкой стоимостью запоминающих устройств и соответствующим увеличением емкости. Но, опять же, на самом деле это была не одна таблица, содержащая все символы, которые были созданы, а стандартная кодировка (называемая Unicode) вместе с различными способами ее представления (UTF-8, UTF-16, UTF- 32 и т.д.) Из всех этих систем стандартом де-факто стала UTF-8, которая позволяет отображать практически любой живой язык и многие уже исчезнувшие языки, а также многочисленные дополнительные символы, все с использованием чисел переменной длины (от 1 до 4 байтов), что, в свою очередь, помогает оптимизировать размер текстовых файлов. Этот размер не увеличился *слишком* по сравнению с файлами, использующими чистый ASCII.

До появления XeTeX системы, основанные на TeX - который также родился в США и, следовательно, имеет английский как родной язык - предполагали, что кодировка была в чистом ASCII; чтобы использовать другую кодировку, вам нужно было как-то указать это в исходном файле.

ConTEXt Mark IV предполагает, что кодировка будет UTF-8. Однако в менее современных компьютерных системах по-прежнему может использоваться другая кодировка. Я не очень уверен в кодировке по умолчанию, которую использует Windows, учитывая, что стратегия Microsoft по охвату широкой публики заключается в сокрытии сложности (но даже если она скрыта, это не означает, что она исчезла!). Информации о системе кодирования, которую он использует по умолчанию, не так много (или мне не удалось ее найти).

В любом случае, независимо от кодировки по умолчанию, любой текстовый редактор позволяет сохранить файл в нужной кодировке. Исходные файлы, предназначенные для обработки с помощью ConTEXt Mark IV, должны быть сохранены в UTF-8, если, конечно, нет очень веской причины для использования другой кодировки (хотя я не могу понять, в чем может быть эта причина).

Если мы хотим записать файл, написанный в другой кодировке (возможно, старый файл), то мы можем

а. Преобразовать файл в формат UTF-8, рекомендуемый вариант, и для этого существуют различные инструменты; в Linux, например, команды `iconv` или `recode`.

б. Сообщить ConTEXt в исходном файле, что кодировка не UTF-8. Для этого нам нужно использовать команду `\enableregime`, синтаксис которой:

```
\enableregime[Encoding]
```

где *Encoding* относится к имени, по которому ConTEXt знает фактическую кодировку рассматриваемого файла. В table ?? вы найдете различные кодировки и имена, по которым их знает ConTEXt

Encoding	**Name(s) in ConTEXt**	**Notes**
Windows CP 1250	cp1250, windows-1250	Western Europe
Windows CP 1251	cp1251, windows-1251	Cyrillic
Windows CP 1252	cp1252, win, windows-1252	Western Europe
Windows CP 1253	cp1253, windows-1253	Greek
Windows CP 1254	cp1254, windows-1254	Turkish
Windows CP 1257	cp1257, windows-1257	Baltic
ISO-8859-1, ISO Latin 1	iso-8859-1, latin1, il1	Western Europe
ISO-8859-2, ISO Latin 2	iso-8859-2, latin2, il2	Western Europe
ISO-8859-15, ISO Latin 9	iso-8859-15, latin9, il9	Western Europe
ISO-8859-7	iso-8859-7, grk	Greek
Mac Roman	mac	Western Europe
IBM PC DOS	ibm	Western Europe
UTF-8	utf	Unicode
VISCII	vis, viscii	Vietnamese
DOS CP 866	cp866, cp866nav	Cyrillic
KOI8	koi8-r, koi8-u, koi8-ru	Cyrillic
Mac Cyrillic	maccyr, macukr	Cyrillic
Others	cp855, cp866av, cp866mav, cp866tat, ctt, dbk, iso88595, isoir111, mik, mls, mnk, mos, ncc	Various

Таблица 4.1 Основные кодировки в ConTEXt

ConTEXt Mk IV настоятельно рекомендует использовать UTF-8. Я согласен с этой рекомендацией. Начиная с этого введения, мы можем предположить, что всегда используется кодировка UTF-8.

Вместе с `\enableregime` ConTEXt включает команду `\useregime`, которая позволяет нам использовать код для той или иной кодировки в качестве аргумента. Я не нашел информации ни об этой команде, ни о том, чем она отличается от `\enableregime`, только несколько примеров её использования . Я подозреваю, что `\useregime` разработан для сложных проектов, в которых используется много исходных файлов, и предполагается, что не все из них будут иметь одинаковую кодировку. Но это только предположение.

4.2 Символы в исходных файлах

Специальные символы – это имя, которое я дал группе символов, которые отличаются от *зарезервированных символов*. Как видно из section ??, они имеют особое значение для ConTEXt и поэтому не могут использоваться непосредственно как символы в исходном файле. Наряду с ними существует еще одна группа символов, которые, хотя и рассматриваются как таковые ConTEXt, когда он находит их в исходном файле, он обрабатывает их по особым правилам. В эту группу входят пробелы (пробелы), табуляции, разрывы строк и дефисы.

4.2.1 Пробелы (пустые пространства) и табуляции

Табуляторы и пробелы в исходном файле обрабатываются одинаково для всех целей и задач. Символ табуляции (клавиша Tab на клавиатуре) будет преобразован ConTEXt в пробел. И пустые места поглощаются любым другим пустым пространством (или табуляцией) сразу после них. Таким образом, нет абсолютно никакой разницы в исходном файле для записи

```
Tweedledum and Tweedledee.
```

or

```
Tweedledum   and   Tweedledee.
```

ConTEXt считает эти два предложения абсолютно одинаковыми. Следовательно, если мы хотим ввести дополнительный пробел между словами, нам нужно использовать некоторые команды ConTEXt, которые делают это. Обычно он работает с «`\␣`», что означает символ \, за которым следует пробел. Но есть и другие процедуры, которые будут рассмотрены в chapter 10.3 относительно горизонтального пространства.

Поглощение следующих друг за другом пустых пространств позволяет нам писать исходный файл, визуально выделяя части, которые мы хотели бы выделить, просто увеличивая или уменьшая используемый отступ, со спокойствием зная, что это никоим образом не повлияет на окончательный документ. Таким образом, в следующем примере

```
The music group from Madrid at the end of the seventies {\em La Romántica
  Banda Local} wrote songs of an eclectic style that were very difficult to
categorise. In their son “El Egipcio”, for example, they said:
\quotation{{\em Esto es una farsa más que una comedia, página muy seria
  de la histeria musical; sueños de princesa, vicios de gitano pueden en
  su mano acariciar la verdad}}, mixing word, phrases simply because they
have an internal rhythm (comedia-histeria-seria, gitano-mano).
```

вы можете увидеть, как некоторые линии слегка смещены вправо. Это строки, которые являются частью фрагментов, которые будут выделены курсивом. Наличие этих отступов помогает (автору) увидеть, где заканчивается курсив.

Некоторые могут подумать, что за бардак! Придется ли мне возиться с отступами строк? На самом деле этот специальный отступ выполняется автоматически моим текстовым редактором (GNU Emacs), когда он редактирует исходный файл ConTEXt. Это такая небольшая подсказка, которая заставляет вас выбирать работу с одним текстовым редактором, а не с другим.

Правило поглощения пробелов применяется исключительно к последовательным пробелам в исходном файле. Следовательно, если пустая группа («`{}`») помещается в исходный файл между двумя

пробелами, хотя пустая группа ничего не создаст в конечном файле, её присутствие гарантирует, что эти два пробела не будут последовательными. Например, если мы напишем «`Tweedledum {} и Tweedledee`», мы получим «Tweedledum и Tweedledee», где, если вы внимательно присмотритесь, вы увидите два последовательных пробела между первыми двумя словами.

То же самое происходит с зарезервированным символом «~», хотя его эффект заключается в создании пробела, хотя на самом деле это не так: пробел, за которым следует ~ не поглотит последний, а пробел после ~ не будет впитаться тоже.

4.2.2 Разрывы строк

В большинстве текстовых редакторов, когда строка превышает максимальную ширину, автоматически вставляется разрыв строки. Мы также можем явно вставить разрыв строки, нажав клавишу «Enter» или «Return»

ConTEXtприменяет следующие правила к разрывам строк:

а. Одиночный разрыв строки во всех смыслах равен пробелу. Следовательно, если непосредственно перед или после разрыва строки есть какое-либо пустое пространство или табуляция, они будут поглощены разрывом строки или первым пустым пространством, а в окончательный документ будет вставлено простое пустое пространство.

б. Два или более разрыва строки подряд создают разрыв абзаца. Для этого два разрыва строки считаются последовательными, если между первым и вторым разрывом строки нет ничего, кроме пробелов или табуляции (поскольку они поглощаются первым разрывом строки); что, вкратце, означает, что одна или несколько последовательных строк, которые являются абсолютно пустыми в исходном файле (без каких-либо символов в них или только с пробелами или табуляциями), становятся разрывом абзаца.

Обратите внимание, что я сказал «два или более последовательных разрывов строки», а затем «одна или несколько пустых последовательных строк», имея в виду, что если мы хотим увеличить расстояние между абзацами, мы не делаем это просто, вставляя еще один разрыв строки. Для этого нам нужно использовать команду, увеличивающую вертикальное пространство. Если нам нужна только одна дополнительная разделительная строка, мы можем использовать команду `\blank`. Но есть и другие способы увеличения вертикального пространства. Я обращаюсь к section 11.2.

В некоторых случаях, когда разрыв строки становится пробелом, мы можем получить нежелательные и неожиданные пробелы. Особенно, когда мы пишем макросы, где пустое пространство легко «sneak in», даже если мы этого не осознаем. Чтобы избежать этого, мы можем использовать зарезервированный символ «%», который, как мы знаем, приводит к тому, что строка, в которой он окажется, не обрабатывается, что означает, что разрыв в конце строки также не будет обрабатываться. Так, например, команда

```
\define[3]\Test{
  {\em #1}
  {\bf #2}
  {\sc #3}
}
```

который пишет свой первый аргумент курсивом, второй жирным шрифтом, а третий прописными буквами, вставит пробел между каждым из этих аргументов, в то время как

```
\define[3]\Test{%
  {\em #1}%
  {\bf #2}%
  {\sc #3}%
}
```

не будет вставлять пробелы между ними, поскольку зарезервированный символ % предотвращает обработку разрывов строк и просто становится пробелом.

4.2.3 Rules/dashes

Тире – хороший пример разницы между компьютерной клавиатурой и печатным текстом. На обычной клавиатуре обычно есть только один символ для тире (или правила, в типографских терминах),

который мы называем дефисом или («-»); но печатный текст использует до четырех разных длин для правил:

- Короткие черточки (дефисы), подобные тем, которые используются для разделения слогов в переносах в конце строки (-).
- Черточки среднего размера (en тире или en дефисы), немного длиннее предыдущих (–). Они имеют ряд применений, в том числе для некоторых европейских языков (в меньшей степени на английском языке) в начале строки диалога или для отделения меньших от больших цифр в диапазоне дат или страниц; «pp. 12–33».
- Длинные (тире или правила em) (—), используемые в качестве скобок для включения одного предложения в другое.
- Знак минус (−) для обозначения вычитания или отрицательного числа.

Сегодня все вышеперечисленное и другие доступны в кодировке UTF-8. Но поскольку все они не могут быть созданы одной клавишей на клавиатуре, их не так просто создать в исходном файле. К счастью, TEX увидел необходимость включить в наш окончательный документ больше черточек / тире, чем может быть создано с помощью клавиатуры, и разработал простую процедуру для этого. ConTEXt дополнил эту процедуру, добавив также команды, которые генерируют эти различные типы черточек. Мы можем использовать два подхода для создания четырех видов правил: либо обычный метод ConTEXt с помощью команды, либо непосредственно с клавиатуры. Эти процедуры показаны в table 4.2:

Type of rule	Appearance	Written directly	Command
Hyphen	-	-	`\hyphen`
En rule	–	`--`	`\endash`
Em rule	—	`---`	`\emdash`
Minus sign	−	`$-$`	`\minus`

Таблица 4.2 Rules/dashes in ConTEXt

Имена команд texhyphen и `\minus` обычно используются в английском языке. Хотя многие в полиграфической отрасли называют их „rules”, термины TEX, а именно `\endash` и `\emdash`, также распространены в терминологии набора. «*en*» и «*em*» - это названия единиц измерения, используемых в типографике. «en» обозначает ширину „n”, а «em» – ширину „m” в используемом шрифте.

4.3 Простые и многофайловые проекты

В ConTEXt мы можем использовать только один исходный файл, который включает в себя абсолютно все содержимое нашего окончательного документа, а также все детали, относящиеся к нему, и в этом случае мы говорим о «простых проектах» или, напротив, мы могли бы использовать несколько исходных файлов, которые разделяют содержимое нашего окончательного документа, и в данном случае мы говорим о «многофайловых проектах».

Сценарии, при которых типично работать с более чем одним исходным файлом, следующие:

- Если мы пишем документ, в котором участвовало несколько авторов, у каждого из которых есть своя часть, отличная от других; например, если мы пишем праздничную брошюру с участием разных авторов, номера журнала и т.д.
- Если мы пишем объемный документ, в котором каждая часть (глава) имеет относительную автономию, так что их окончательная компоновка допускает несколько возможностей и будет определена в конце. Это происходит с относительной частотой для многих академических текстов (руководств, введений и т.п.), где порядок глав может варьироваться.
- Если мы пишем ряд связанных документов, которые имеют некоторые стилевые характеристики.

- Если, проще говоря, документ, над которым мы работаем, большой, так что компьютер тормозит либо при его редактировании, либо при компиляции; в этом случае разделение материала на несколько исходных файлов значительно ускорит компиляцию каждого из них.
- Кроме того, если мы написали ряд макросов, которые хотим применить в некоторых (или во всех) наших документах, или если мы создали шаблон, который управляет или стилизует наши документы, и мы хотим применить их к ним и т.д.

4.4 Структура исходного файла в простых проектах

В простых проектах, разработанных в одном исходном файле, структура очень проста и вращается вокруг «text» среды, которая, по сути, должна находиться в том же файле. Мы различаем следующие части этого файла:

- **Преамбула документа**: все от первой строки файла до начала «text» среды (\starttext).
- **Тело документа**: это содержимое «text» среды; или, другими словами, все между \starttext и \stoptext.

```
% First line of the document

% Preamble area:
% Containing the global configuration
% commands for the document

\starttext % The body of the document begins here

...
... % Document contents
...

\stoptext % End of the document
```

Рисунок 4.1 file containing a simple project

На figure 4.1 мы видим очень простой исходный файл. Абсолютно все, что находится перед командой \starttext (которая на картинке находится в строке 5, считая только те, на которых есть текст), составляет преамбулу; все между \starttext и \stoptext составляет тело документа. Все, что находится после stoptext, будет проигнорировано.

Преамбула используется для включения команд, влияющих на документ в целом, тех, которые определяют его общую конфигурацию. Писать какую-либо команду в преамбуле не обязательно. Если её нет, ConTEXt примет конфигурацию по умолчанию, которая не очень развита, но подходит для многих документов. В хорошо спланированном документе преамбула будет содержать все команды, влияющие на документ в целом, такие как макросы и настраиваемые команды, которые будут использоваться в исходном файле. В типичной преамбуле это может включать следующее:

- Указание основного языка документа (см. section 10.5).
- Указание размера бумаги (section 5.1) и макета страницы (section 5.3).
- Особенности основного шрифта документов (section ??).
- Настройка используемых команд раздела (section 6.4) и, при необходимости, определение новых команд раздела (section 6.5).
- Макет верхних и нижних колонтитулов (section 5.6).
- Настройки собственных макросов (section 3.7).
- Так далее.

Преамбула предназначена для общей конфигурации документа; поэтому здесь не должно быть ничего, что связано с содержимым *contents* документа или обрабатываемым текстом. Теоретически

любой обрабатываемый текст, включенный в преамбулу, будет проигнорирован, хотя иногда, если он там есть, это вызовет ошибку компиляции.

Тело документа, заключенное между командами `\starttext` и `\stoptext`, включает фактическое содержимое, то есть обрабатываемый текст, а также команды ConTEXt, которые не должны влиять на весь документ.

4.5 Многофайловое управление в стиле TEX

Чтобы работать с более чем одним исходным файлом, TEX включил примитив под названием `\input`, который также работает в ConTEXt, хотя последний включает две специфические команды, которые в некоторой степени улучшают способ функции `\input`.

4.5.1 Команда `\input`

Команда `\input` вставляет содержимое указанного файла. Его формат:

```
\input FileName
```

где *FileName* – имя вставляемого файла. Обратите внимание, что имя файла не обязательно заключать в фигурные скобки, даже если это не приведет к возникновению ошибки. Однако его нельзя заключать в квадратные скобки. Если расширение файла «`.tex`», его можно не указывать.

Когда ConTEXt компилирует документ и находит команду `\input`, он ищет указанный файл и продолжает компиляцию, как если бы этот файл был частью файла, который его вызвал. Когда он завершает его компиляцию, он возвращается к исходному файлу и продолжает с того места, где он остановился; практический результат, таким образом, состоит в том, что содержимое файла, вызываемого с помощью `\input`, вставляется в точку, где он был вызван. Файл, вызываемый с помощью `\input`, должен иметь допустимое имя в нашей операционной системе и без пробелов в имени. ConTEXt будет искать его в рабочем каталоге, и если он не найдет его там, он будет искать его в каталогах, включенных в переменную среды TEXROOT. Если файл в конечном итоге не будет найден, произойдет ошибка компиляции.

Чаще всего команда `\input` используется следующим образом: записывается файл, назовем его «`principal.tex`», и он будет использоваться в качестве контейнера для вызова с помощью команды `\input` различных файлов, составляющих наш проект. Это показано в следующем примере:

```
% General configuration commands:

  \input MyConfiguration

\starttext

  \input PageTitle
  \input Preface
  \input Chap1
  \input Chap2
  \input Chap3

  ...

\stoptext
```

Обратите внимание, как для общей конфигурации документа мы назвали файл «MyConfiguration.tex», который, как мы предполагаем, содержит глобальные команды, которые мы хотим применить. Затем между командами `\starttext` и `\stoptext` мы вызываем несколько файлов, которые содержат содержимое различных частей нашего документа. Если в какой-то момент для ускорения процесса компиляции мы хотим исключить компиляцию некоторых файлов, все, что нам нужно сделать, это поставить отметку комментария в начале строки, вызывающей тот или иной файл. Например, если

мы пишем третью главу и хотим скомпилировать её просто для того, чтобы проверить, нет ли в ней ошибок, нам не нужно компилировать остальную часть, и поэтому мы можем написать:

```
% General configuration commands:

  \input MyConfiguration

\starttext

  % \input PageTitle
  % \input Preface
  % \input Chap1
  % \input Chap2

  \input Chap3

  ...

\stoptext
```

и будет скомпилирована только Глава 3. Обратите внимание, что, с другой стороны, изменить порядок глав так же просто, как изменить порядок вызывающих их строк.

> Когда мы исключаем файл в многофайловом проекте из компиляции, мы увеличиваем скорость обработки, но в результате все ссылки, которые компилируемая часть делает на другие части, которые еще не скомпилированы, больше не будут работать. См. section 8.2.

Важно понимать, что когда мы работаем с `\input`, только главный файл, тот, который вызывает все остальные, должен включать команды`\starttext` и `\stoptext`, потому что, если другие файлы включают их, будет ошибка. Это, с другой стороны, означает, что мы не можем напрямую скомпилировать различные файлы, составляющие проект, но обязательно должны скомпилировать их из основного файла, который содержит основную структуру документа.

4.5.2 `\ReadFile` и `\readfile`

Как мы только что видели, если ConTEXt не найдет файл с именем `\input`, он выдаст ошибку. Для ситуации, когда мы хотим вставить файл только в том случае, если он существует, но с учетом возможности того, что он может и не быть, ConTEXt предлагает вариант команды `\input`. Это

```
\ReadFile{FileName}
```

Эта команда похожа на `\input` во всех отношениях, за исключением того, что если файл, который нужно вставить, не найден, он продолжит компиляцию без каких-либо ошибок. Он также отличается от `\input` своим синтаксисом, поскольку мы знаем, что с `\input` нет необходимости помещать имя файла, который будет вставлен между фигурными скобками. Но с `\ReadFile` это необходимо. Если мы не будем использовать фигурные скобки, ConTEXt будет считать, что имя искомого файла совпадает с первым символом, следующим за командой `\ReadFile`, за которым следует расширение `.tex`. Так, например, если мы напишем

```
\ReadFile MyFile
```

ConTEXt поймет, что файл, который нужно прочитать, называется «`M.tex`», поскольку символ сразу после команды `\ReadFile` (исключая пробелы, которые, как мы знаем, игнорируются в конце имени команды) – это „M”. Поскольку ConTEXt обычно не находит файл с именем «`M.tex`», а `\ReadFile` не генерирует ошибку, если не находит файл, ConTEXt продолжит компиляцию после „M” в «MyFile» и вставит текст «`yFile`».

Более усовершенствованная версия texReadFile – это `\readfile`, формат которой

```
\readfile{FileName}{TextIfExists}{TextIfNotExists}
```

Первый аргумент аналогичен `\ReadFile`: имя файла, заключенное в фигурные скобки. Второй аргумент включает текст, который будет записан, если файл существует, перед вставкой содержимого

файла. Третий аргумент включает текст, который будет записан, если рассматриваемый файл не найден. Это означает, что в зависимости от того, найден ли файл, введенный в качестве первого аргумента, будет выполнен второй аргумент (если файл существует) или третий (если файл не существует).

4.6 ConTEXt проекты как таковые

Третий механизм, который ConTEXt предлагает для многофайловых проектов, является более сложным и полным: он начинается с различения файлов проекта, файлов продукта, файлов компонентов и файлов среды. Чтобы понять взаимосвязь и функционирование каждого из этих типов файлов, я думаю, что лучше всего объяснить их каждый в отдельности:

4.6.1 Файлы *среды*

Файл среды - это файл, в котором хранятся макросы и конфигурации определенного стиля, который предназначен для применения к нескольким документам, независимо от того, являются ли они полностью независимыми документами или частями сложного документа. Таким образом, файл окружения может включать все, что мы обычно пишем перед `\starttext`; то есть: общая конфигурация документа.

> Я сохранил термин «файлы среды» для этих типов файлов, чтобы не отходить от официальной терминологии ConTEXt даже несмотря на то, что я считаю, что лучшим термином, вероятно, будет «файлы форматирования» или «файлы глобальной конфигурации».

Как и все исходные файлы ConTEXt, файлы среды являются текстовыми файлами и предполагают, что расширение будет «`.tex`», хотя, если мы хотим, мы можем изменить его, возможно, на «`.env`». Однако обычно это не делается в ConTEXt. Чаще всего файл среды идентифицируется по имени, начинающемуся или заканчивающемуся на „env”. Например: «`MyMacros_env.tex`» или «`env_MyMacros.tex`». Внутри такой файл окружения будет выглядеть примерно так:

```
\startenvironment MyEnvironment

  \mainlanguage[en]

  \setupbodyfont
    [modern]

  \setupwhitespace
    [big]

  ...

\stopenvironment
```

Другими словами, определения и команды настройки входят в `\startenvironment` и `\stopenvironment`. Сразу после `\startenvironment` мы пишем имя, по которому мы хотим идентифицировать рассматриваемую среду, а затем включаем все команды, из которых мы хотели бы, чтобы наша среда состояла.

> Что касается имени среды, согласно моим тестам, имя, которое мы добавляем сразу после `\startenvironment`, является просто ориентировочным, и если бы мы не дали ему имени, то ничего (плохого) не произошло.

Файлы среды предназначались для работы с компонентами и продуктами (объяснено в следующем разделе). Вот почему одна или несколько сред могут быть вызваны из компонента или продукта с помощью команды `\environment`. Но эта команда также работает, если она используется в области конфигурации (преамбуле) любого исходного файла ConTEXt, даже если это не исходный файл, предназначенный для компиляции по частям.

Команду `\environment` можно вызвать в любом из двух следующих форматов:

`\environment File`

`\environment[File]`

В любом случае результатом этой команды будет загрузка содержимого файла, взятого в качестве аргумента. Если этот файл не найден, он продолжит компилирование обычным способом без каких-либо ошибок. Если расширение файла «`.tex`», его можно не указывать.

4.6.2 Компоненты и продукты

Если мы подумаем о книге, в которой каждая глава находится в отдельном исходном файле, то мы бы сказали, что главы - это *компоненты*, а книга - это *product*. Это означает, что *component* является автономной частью *product*, способной иметь свой собственный стиль и компилироваться независимо. У каждого компонента будет отдельный файл, и, кроме того, будет файл продукта, который объединяет все компоненты.

Типичный компонентный файл выглядит следующим образом

```
\environment MyEnvironment
\environment MyMacros

\startcomponent Chapter1

  \startchapter[title={Chapter 1}]

  ...

\stopcomponent
```

И файл продукта будет выглядеть следующим образом:

```
\environment MyEnvironment
\environment MyMacros

\startproduct MyBook

  \component Chapter1
  \component Chapter2
  \component Chapter3

  ...

\stopproduct
```

Обратите внимание, что фактическое содержимое нашего документа будет распределено между различными файлами «компонентов», а файл продукта ограничен установкой порядка компонентов. С другой стороны, (отдельные) компоненты и продукты могут быть скомпилированы напрямую. При компиляции продукта создается PDF-файл, содержащий все компоненты этого продукта. Если, с другой стороны, один из компонентов компилируется отдельно, он сгенерирует файл pdf, содержащий только скомпилированный компонент.

В файле компонента и перед командой `\startcomponent` мы можем вызвать один или несколько файлов среды с помощью `\environmentEnvironmentName`. То же самое можно сделать в файле продукта перед `\startproduct`. Одновременно могут быть загружены несколько файлов среды. Например, у нас может быть наша любимая коллекция макросов и различных стилей, которые мы применяем к нашим документам, в разных файлах. Однако обратите внимание, что когда мы используем две или более среды, они загружаются в том порядке, в котором они вызываются, так что если одна и та же команда конфигурации была включена в более чем одну среду и имеет разные значения, значения последней загруженной среды. С другой стороны, файлы среды загружаются только один раз, поэтому в предыдущих примерах, в которых среда вызывается из файла продукта и из определенных файлов компонентов, если мы компилируем продукт, это время загрузки среды и в указанном там

порядке; когда среда вызывается из любого из компонентов, ConTEXt проверяет, загружена ли среда уже, и в этом случае ничего не будет делать.

Имя компонента, вызываемого из продукта, должно быть именем файла, содержащего рассматриваемый компонент, хотя, если расширение этого файла - «.tex», его можно не указывать.

4.6.3 Проекты как таковые

В большинстве случаев достаточно проводить различие между продуктами и компонентами. Точно так же ConTEXt имеет еще более высокий уровень, на котором мы можем сгруппировать ряд продуктов: это *проект*.

Типичный файл проекта будет выглядеть примерно следующим образом

```
\startproject MyCollection

  \environment MyEnvironment
  \environment MyMacros

  \product Book01
  \product Book02
  \product Book03

  ...

\stopproject
```

Сценарий, в котором нам понадобится проект, будет, например, когда нам нужно отредактировать коллекцию книг, все с одинаковыми спецификациями формата; или если бы мы редактировали журнал: сборник книг или журнал как таковой были бы проектом; каждая книга или каждый выпуск журнала были бы продуктом; и каждая глава книги или каждая статья в журнальном выпуске будет составляющей.

С другой стороны, проекты не предназначены для непосредственной компиляции. Учтите, что по определению каждый продукт, принадлежащий проекту (каждая книга в коллекции или каждый выпуск журнала), должен быть скомпилирован отдельно и генерировать свой собственный PDF-файл. Поэтому включенная в него команда `\product`, указывающая, какие продукты принадлежат проекту, на самом деле ничего не делает: это просто напоминание для автора.

Ясно, что некоторые могут спросить, зачем у нас проекты, если они не могут быть скомпилированы: ответ заключается в том, что файл проекта связывает определенные среды с проектом. Вот почему, если мы включим команду `\project ProjectName` в файл компонента или продукта, ConTEXt прочитает файл проекта и автоматически загрузит связанные с ним среды. Вот почему команда `\environment` в проектах должна идти после `\startproject`; однако в продуктах и компонентах `\environment` должен *предшествовать* `\startproduct` или `\startcomponent`.

Как и в случае с командами `\environment` и `\component`, команда `\project` позволяет указать имя проекта либо внутри квадратных скобок, либо вообще не использовать квадратные скобки. Это означает, что `\project FileName` и `\Project[FileName]` являются эквивалентными командами.

Резюме различных способов загрузки окружения

Из вышесказанного следует, что окружение может быть загружено любой из следующих процедур:

а. Вставив команду `\environment EnvironmentName` перед `\starttext` или `\startcomponent`. Это загрузит среду только для компиляции рассматриваемого файла.

б. Вставив команду `\environment EnvironmentName` в файл продукта перед `\startproduct`. Это загрузит среду при компиляции продукта, но не в том случае, если его компоненты компилируются индивидуально.

в. Вставив команду `\project` в продукт или среду: это загрузит все среды, связанные с проектом (в файле проекта).

4.6.4 Общие аспекты сред, компонентов, продуктов и проектов

Названия сред, компонентов, продуктов и проектов: мы уже видели, что для всех этих элементов после команды `\start`, запускающей конкретную среду, компонент, продукт или проект, непосредственно вводится его имя. Это имя, как правило, должно совпадать с именем файла, содержащего среду, компонент или продукт, потому что, например, когда ConTEXt компилирует продукт и, согласно файлу продукта, должен загружать среду или компонент, у нас нет способа узнать, к какому файлу относится эта среда или компонент, если только файл не имеет то же имя, что и загружаемый элемент.

В противном случае, согласно моим тестам, имя, написанное после texstartproduct или `\startenvironment` в файлах продукта и среды, является просто ориентировочным. Если он опущен или не соответствует имени файла, ничего плохого не произойдет. Однако в случае компонентов важно, чтобы имя компонента совпадало с именем файла, который его содержит.

Структура каталогов проекта: Мы знаем, что по умолчанию ConTEXt ищет файлы в рабочем каталоге и по пути, указанному переменной TEXROOT. Однако, когда мы используем команды `\project`, `\product`, `\component` или `\environment`, предполагается, что проект имеет структуру каталогов, в которой общие элементы находятся в родительском каталоге, а определенные - в некотором дочернем каталоге. Таким образом, если файл, указанный в рабочем каталоге, не найден, он будет найден в его родительском каталоге, а если он также не найден, в родительском каталоге этого каталога и так далее.

II
Глобальные аспекты документа

Глава 5

Страницы и разбивка документов на страницы

ConTEXt преобразует исходный документ в правильно отформатированные *страницы*. Как выглядят эти страницы, как распределяются текст и пустые места и какие элементы они содержат – все это имеет фундаментальное значение для хорошего набора. Эта глава посвящена всем этим вопросам, а также некоторым другим вопросам, связанным с разбиением на страницы.

5.1 Page size

5.1.1 Установка размеров страницы

По умолчанию ConTEXt предполагает, что документы будут иметь размер A4, европейский стандарт. Мы можем установить другой размер с помощью `\setuppapersize`, который является *нормальной* командой в преамбуле документа. Обычный синтаксис этой команды:

```
\setuppapersize[LogicalPage][PhysicalPage]
```

где оба аргумента принимают символические имена.[1] Первый аргумент, который я назвал *LogicalPage*, представляет размер страницы, который следует учитывать при наборе; а второй аргумент *PhysicalPage* представляет размер страницы, на которой он будет напечатан. Обычно оба размера одинаковы, поэтому второй аргумент можно не указывать; однако в некоторых случаях два размера могут быть разными, как, например, при печати книги на листах по 8 или 16 страниц (обычная техника печати, особенно для академических книг примерно до 1960-х годов). В этих случаях ConTEXt позволяет нам различать оба размера; и если идея состоит в том, что несколько страниц должны быть напечатаны на одном листе бумаги, мы также можем указать схему складывания, которой необходимо следовать, используя команду `\setuparranging` (которая не будет объяснена в этом введении).

[1] Напомним, что в section 3.5 я указал, что параметры, принимаемые командами ConTEXt, в основном бывают двух видов: символьные имена, значение которых уже известно ConTEXt, или переменная, которой мы должны явно присвоить какое-либо значение.

Для набора размеров мы можем указать любой из заранее определенных размеров, используемых в бумажной промышленности (или нами самими). Этот список включает:

- Любая из серий A, B и C, определенных ISO-216 (от A0 до A10, от B0 до B10 и от C0 до C10), обычно используется в Европе.
- Любой из стандартных размеров US: письмо, бухгалтерская книга, таблоид, юридический, фолио, исполнительный.
- Любой из размеров RA и RSA, определенных стандартом ISO-217.
- Размеры G5 и E5 определены швейцарским стандартом SIS-014711 (для докторских диссертаций).
- Для конвертов: любой из размеров, определенных североамериканским стандартом (от конверта 9 до конверта 14) или стандартом ISO-269 (C6/C5, DL, E4).
- CD, для обложек CD.
- S3 – S6, S8, SM, SW для размеров экрана в документах, которые не предназначены для печати, но отображаются на экране.

Вместе с размером бумаги с помощью `\setuppapersize` мы можем указать ориентацию страницы: «книжная» (вертикальная) или «альбомная» (горизонтальная).

Другие параметры, которые разрешает `\setuppapersize`, согласно wiki ConTEXt:«`rotated`», «`90`», «`180`», «`270`», «`mirrored`» и «`negative`». В моих собственных тестах я заметил только некоторые заметные изменения с «`rotated`», который переворачивает страницу, хотя это не совсем инверсия. Числовые значения должны давать эквивалентную степень вращения, сами по себе или в сочетании с «`rotated`», но мне не удалось заставить их работать. Я не совсем понял, для чего нужны «`mirrored`» и «`negative`».

Второй аргумент `\setuppapersize`, о котором я уже говорил, можно опустить, если размер печати не отличается от размера набора, может принимать те же значения, что и первый, с указанием размера и ориентации бумаги. Он также может принимать значение «`oversized`» как значение, которое, согласно ConTEXt wiki, добавляет полтора сантиметра к каждому углу листа.

Согласно wiki есть другие возможные значения для второго аргумента: «`undersized`», «`doublesized`» и «`doubleoversized`». Но в моих собственных тестах я не заметил никаких изменений после использования любого из них; и в официальном определении команды (см. section ??) эти параметры не упоминаются.

5.1.2 Использование нестандартных размеров страниц

Если мы хотим использовать нестандартный размер страницы, мы можем сделать две вещи:

1. Использовать альтернативный синтаксис `\setuppapersize`, который позволяет нам вводить высоту и ширину бумаги как размеры.
2. Определить свой собственный размер страницы, присвоив ему имя и используя его, как если бы это был стандартный размер бумаги.

Альтернативный синтаксис `\setuppapersize`

Помимо синтаксиса, который мы уже видели, `\setuppapersize` позволяет нам использовать еще один:

`\setuppapersize`[Name][Options]

где *Name* - это необязательный аргумент, который представляет имя, присвоенное размеру бумаги с помощью `\definepapersize` (которое мы рассмотрим далее), а *Options* относятся к тому типу, в котором мы назначаем явное значение. Среди допустимых вариантов можно выделить следующие:

- **width, height** которые представляют соответственно ширину и высоту страницы.
- **страница, бумага**. Первый относится к размеру набираемой страницы, а второй - к размеру страницы, на которой будет печататься физически. Это означает, что «page» эквивалентно первому аргументу \setuppapersize в его обычном синтаксисе (объяснено выше) и «page» второму аргументу. Эти параметры могут принимать значения, указанные ранее (A4, S3 и т.д.).
- **шкала**, указывает коэффициент масштабирования для страницы.
- **topspace (верхнее пространство), backspace (нижнее пространство), offset (смещение)**, дополнительные расстояния.

Определение нестандартного размера страницы

Чтобы определить собственный размер страницы, мы используем команду \definepapersize, синтаксис которой:

```
\definepapersize[Name][Options]
```

де *Name* относится к имени, данному новому размеру, а *Options* могут быть:

- Любое из допустимых значений для \setuppapersize в его обычном синтаксисе (A4, A3, B5, CD и т.д.).
- Измерения ширины – width (бумаги), высоты – height (бумаги) и смещения – offset (смещения) или масштабированного значения («scale»).

Что невозможно, так это смешивать значения, разрешенные для \setuppapersize, с измерениями или масштабными коэффициентами. Это связано с тем, что в первом случае параметры являются символическими словами, а во втором - переменными с явным значением; а в ConTEXt, как я уже сказал, невозможно смешивать оба типа параметров.

Когда мы используем \definepapersize, чтобы указать размер бумаги, который совпадает с некоторыми стандартными измерениями, на самом деле, вместо того, чтобы определять новый размер бумаги, мы определяем новое имя для уже существующего размера бумаги. Это может быть полезно, если мы хотим совместить размер бумаги с ориентацией. Так, например, мы можем написать

```
\definepapersize[vertical][A4, portrait]
\definepapersize[horizontal][A4, landscape]
```

5.1.3 Изменение размера страницы в любом месте документа

В большинстве случаев документы имеют только один размер страницы, поэтому \setuppapersize – это типичная команда, которую мы включаем в преамбулу и используем только один раз в каждом документе. Однако в некоторых случаях может потребоваться изменить размер в какой-то момент документа; например, если с определенного момента и далее прилагается приложение, в котором листы альбомные.

В таких случаях мы можем использовать \setuppapersize именно в том месте, где мы хотим, чтобы изменение произошло. Но поскольку размер изменится немедленно, во избежание неожиданных результатов мы обычно вставляем принудительный разрыв страницы перед \setuppapersize.

Если нам нужно изменить размер страницы только для отдельной страницы, вместо того, чтобы использовать \setuppapersize дважды, один раз для изменения на новый размер, а второй - для возврата к исходному размеру, мы можем использовать \adaptpapersize, который изменяет размер страницы, и, спустя страницу, автоматически сбрасывает значение перед его вызовом. И просто так же, как мы сделали с \setuppapersize, перед использованием \adappapersize мы должны вставить принудительный разрыв страницы.

5.1.4 Настройка размера страницы в соответствии с ее содержимым

В ConTEXt есть три среды, которые генерируют страницы точного размера для хранения их содержимого. Это`\startMPpage`, `\startpagefigure` и `\startTEXpage`. Первый генерирует страницу, содержащую графику, сгенерированную с помощью Meta-Post, языка графического дизайна, который интегрируется с ConTEXt, но который я не буду рассматривать в этом введении. Второй делает то же самое с изображением и, возможно, текстом под ним. Он принимает два аргумента: первый определяет файл, содержащий изображение. Я расскажу об этом в главе, посвященной внешним изображениям. Третий (`\startTEXpage`) содержит текст, который является его содержимым на странице. Его синтаксис:

```
\startTEXpage[Options] ... \stopTEXpage
```

где варианты могут быть любыми из следующих:

- **`strut`** распорка. Я не уверен в полезности этой опции. В терминологии ConTEXt *strut* - это символ без ширины, но с максимальной высотой и глубиной, но я не совсем понимаю, какое это имеет отношение к общей полезности этой команды. Согласно вики, эта опция допускает значения «`yes`», «`no`», «`global`» и «`local`», а значение по умолчанию – «`no`».
- **`align`** выровнивание. Указывает на выравнивание текста. Это может быть «`normal`»,«`flushleft`», «`flushright`», «`middle`», «`high`», «`low`» или «`lohi`».
- **`offset`** смещение, чтобы указать количество пустого пространства вокруг текста. Это может быть «`none`», «`overlay`» (наложение), если желателен эффект наложения, или фактическое измерение.
- **`width, height`** где мы можем указать ширину и высоту страницы, или значение «`fit`», чтобы ширина и высота соответствовали тексту, включенному в среду.
- **`frame`** рамка, которая по умолчанию «`off`» «выключена», но может принимать значение «`on`» «включено», если нам нужна рамка вокруг текста на странице.

5.2 Элементы на странице

ConTEXt распознает различные элементы на страницах, размеры которых можно настроить с помощью `\setuplayout`. Мы сразу же рассмотрим это, но заранее было бы лучше описать каждый из элементов страницы, указав имя, которое `\setuplayout` знает каждый из них:

- **Edges:** пустое пространство, окружающее текстовую область. Хотя большинство текстовых процессоров называют их «margins», предпочтительнее использовать терминологию ConTEXt, поскольку она позволяет нам различать края как таковые, где нет текста (хотя в электронных документах могут быть кнопки навигации и тому подобное), и поля, где иногда могут располагаться определенные текстовые элементы, такие как, например, примечания на полях.
  - Высота верхнего края контролируется двумя измерениями: самим верхним краем ((«`top`» «верх») и расстоянием между верхним краем и заголовком («`topdistance`» «расстояние до верха»). Сумма этих двух измерений называется «`topspace`» «верхним пространством».
  - Размер левого и правого краев зависит от измерений «`leftedge`» «`rightedge`» – «левого края» и «правого края» соответственно. Если мы хотим, чтобы оба были одинаковой длины, мы можем настроить их одновременно с помощью опции «`edge`».

    В документах, предназначенных для двусторонней печати, мы говорим не о левом и правом краях, а о внутреннем и внешнем; первый - край, ближайший к точке, где листы будут сшиваться или сшиваться, то есть левый край на страницах с нечетными номерами и правый край на страницах с четными номерами. Внешний край противоположен внутреннему.

  - Высота нижнего края называется «bottom» (дно).

- **Margins** собственно так называемые. ConTEXt вызывает только боковые поля (левое и правое). Поля расположены между краем и основной текстовой областью и предназначены для размещения определенных текстовых элементов, таких как, например, заметки на полях, заголовки разделов или их номера.

  Размеры, определяющие размер поля:

  - **margin**: используется, когда мы хотим одновременно установить одинаковый размер полей.
  - **leftmargin, rightmargin**: установите размер левого и правого полей соответственно.
  - **edgedistance**: расстояние между краем и полем.
  - **leftedgedistance, rightedgedistance**: расстояние между краем и левым и правым полями соответственно.
  - **margindistance**: расстояние между полем и основной текстовой областью.
  - **leftmargindistance, rightmargindistance**: расстояние между основной текстовой областью и правым и левым полями соответственно.
  - **backspace**: это измерение представляет собой пространство между левым углом листа и началом основной текстовой области. Следовательно, он состоит из суммы «leftedge» + «leftedgedistance» + «leftmargin» + «leftmargindistance».

- **Header and footer**: верхний и нижний колонтитулы страницы - это две области, которые расположены, соответственно, в верхней или нижней части письменной области страницы. Обычно они содержат информацию, которая помогает контекстуализировать текст, например номер страницы. , имя автора, название работы, название главы или раздела и т. д. В ConTEXtна эти области на странице влияют следующие параметры:

  - **header**: высота заголовка.
  - **footer**: высота нижнего колонтитула.
  - **headerdistance**: расстояние между заголовком и основной текстовой областью страницы.
  - **footerdistance**: расстояние между нижним колонтитулом и основной текстовой областью страницы.
  - **topdistance, bottomdistance**: оба упомянутых ранее. Это расстояние между верхним краем и заголовком или нижним краем и нижним колонтитулом соответственно.

- **Main text area**: Область основного текста: это самая широкая область на странице, содержащая текст документа. Его размер зависит от переменных «width» (ширина) и «textheight» (высота текста). В свою очередь, переменная «height» измеряется как сумма «header», «headerdistance», «textheight», «footerdistance» и «footer».

Мы можем видеть все эти области на image 5.1 вместе с названиями соответствующих измерений и стрелками, указывающими их протяженность. Толщина стрелок вместе с размером названий стрелок призваны отражать важность каждого из этих расстояний для макета страницы. Если мы присмотримся, то увидим, что это изображение показывает, как страница может быть представлена в виде таблицы с 9 строками и 9 столбцами, или, если мы не принимаем во внимание значения разделения между различными областями, будет пять строк. и пять столбцов, из которых текст может быть только в трех строках и трех столбцах. Пересечение средней строки со средним столбцом составляет основную текстовую область и обычно занимает большую часть страницы.

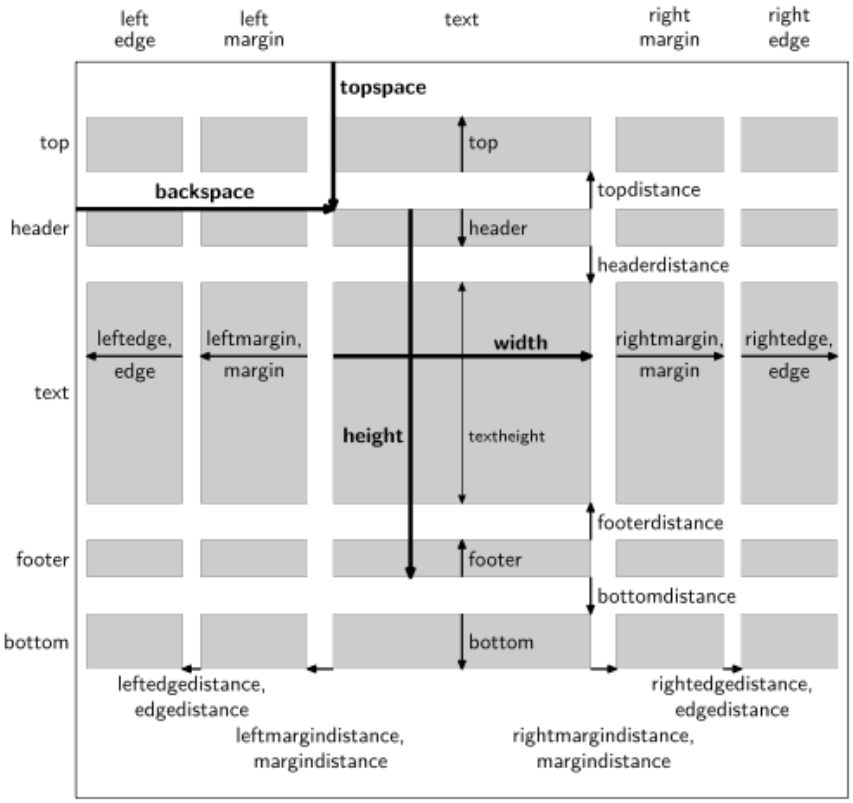


Рисунок 5.1 Области и измерения на странице

На этапе макета нашего документа мы можем увидеть все размеры страницы с помощью `\showsetups`. Чтобы увидеть основные схемы распределения текста, визуально отображаемые на странице, мы можем использовать`\showframe`; а с`\showlayout` мы можем получить комбинацию двух предыдущих команд.

5.3 Page layout (`\setuplayout`)

5.3.1 Assigning a size to the different page components

Дизайн страницы предполагает присвоение определенных размеров соответствующим областям страницы. Это делается с помощью `\setuplayout`. Эта команда позволяет нам изменять любые размеры, упомянутые в предыдущем разделе. Его синтаксис следующий:

```
\setuplayout[Name] [Options]
```

где *Name* – необязательный аргумент, используемый только в том случае, когда мы разработали несколько макетов (см. section 5.3.3), а параметры, помимо других, которые мы увидим позже, включают любые из ранее упомянутых измерений. Однако имейте в виду, что эти измерения взаимосвязаны, поскольку сумма компонентов, влияющих на ширину, из компонентов, влияющих на высоту, должна совпадать с шириной и высотой страницы. В принципе это будет означать, что при изменении длины по горизонтали мы должны отрегулировать оставшиеся длины по горизонтали; и то же самое при регулировке длины по вертикали.

По умолчанию ConTEXt выполняет автоматическую корректировку размеров только в некоторых случаях, которые, с другой стороны, не указаны в полной или систематической документации в доку-

ментации. Проведя несколько тестов, я смог убедиться, например, что ручное увеличение или уменьшение высоты верхнего или нижнего колонтитула влечет за собой корректировку «textheight»; однако ручное изменение некоторых полей не приводит к автоматической настройке (согласно моим тестам) ширины текста («width»). Вот почему наиболее эффективный способ избежать несоответствия между размером страницы (установленным с помощью \setuppapersize) и размером соответствующих компонентов:

- Относительно горизонтальных измерений:
  - Регулируя «backspace» (что включает «leftedge» и «leftmargin»).
  - Регулируя «width» (text width) не с размерами, а со значениями «fit» «middle»:
    - `fit` вычисляет ширину текста на основе ширины остальных горизонтальных компонентов страницы.
    - `middle` делает то же самое, но сначала выравнивает правое и левое поля.
- Относительно вертикальных измерений:
  - Регулируя «верхнее пространство» «topspace».
  - Регулируя значения «fit» или «middle» к «высоте» «height». Они работают так же, как и в случае ширины. Первый вычисляет высоту на основе остальных компонентов, а второй сначала выравнивает верхнее и нижнее поля, а затем вычисляет высоту текста.
  - После настройки «height», при необходимости отрегулировав высоту верхнего или нижнего колонтитула, зная, что в таких случаях «textheight» будет автоматически изменена.
- Другой возможностью косвенного определения высоты основной текстовой области является указание количества строк, которые должны уместиться в ней (с учетом текущего межстрочного пространства и размера шрифта). Вот почему \setuplayout включает параметр «lines».

Размещение логической страницы на физической странице

В случае, когда размер логической страницы не совпадает с размером физической страницы (см. section 5.1.1) \setuplayout позволяет нам настроить некоторые дополнительные параметры, влияющие на размещение логической страницы на физической странице:

- **location**: расположение: этот параметр определяет место, где страница будет размещена на физической странице. Возможные значения: левый, средний, правый, верхний, нижний, односторонний, двусторонний или двусторонний.
- **scale**: указывает коэффициент масштабирования для страницы перед ее размещением на физической странице.
- **marking**: будет печатать визуальные метки на странице, указывающие, где бумага должна быть обрезана.
- **horoffset, veroffset, clipoffset, cropoffset, trimoffset, bleedoffset, artoffset**: серия измерений, указывающих различные смещения на физической странице. Большинство из них объясняется в справочном руководстве 2013 года.

Эти параметры \setuplayout должны сочетаться с указаниями из \setuparranging, которые показывают, как логические страницы должны быть упорядочены на физическом листе бумаги. Я не буду объяснять эти команды в этом введении, так как я не проводил с ними никаких тестов.

Получение ширины и высоты текстовой области

Команды \textwidth и \textheight возвращают ширину и высоту текстовой области соответственно. Значения, предлагаемые этими командами, не могут быть напрямую показаны в окончательном

документе, но их можно использовать для других команд, чтобы установить их ширину или высоту. Так, например, чтобы указать, что нам нужно изображение, ширина которого будет составлять 60% от ширины линии, нам нужно указать в качестве значения параметра «`width`» изображения: «`width=0.6\textwidth`».

5.3.2 Адаптация макета страницы

Может случиться так, что макет нашей страницы на определенной странице дает нежелательный результат; как, например, последняя страница главы, состоящая всего из одной или двух строк, что нежелательно ни с типографской, ни с эстетической точки зрения. Чтобы решить эти проблемы, ConTEXt предоставляет команду `\adaplayout`, которая позволяет нам изменять размер текстовой области на одной или нескольких страницах. Эта команда предназначена для использования только тогда, когда мы уже закончили писать наш документ и вносим некоторые небольшие окончательные корректировки. Поэтому его естественное расположение - в преамбуле документа. Синтаксис команды:

```
\adaptlayout[Pages] [Options]
```

где *Pages* означает номер страницы или страниц, макет которых мы хотим изменить. Это необязательный аргумент, который следует использовать только тогда, когда в преамбуле помещается `\adaplayout`. Мы можем указать как одну страницу, так и несколько страниц, разделяя числа запятыми. Если мы опустим этот первый аргумент, `\adaplayout` будет влиять исключительно на страницу, на которой он находит команду.

Что касается вариантов, то они могут быть:

- **`height`**: позволяет указать в качестве размеров высоту, которую должна иметь рассматриваемая страница. Мы можем указать абсолютную высоту (например, «19 см») или относительную высоту (например, «+1 см», «-0,7 см»).
- **`lines`**: мы можем включить количество строк для добавления или вычитания. Для добавления строк перед значением стоит знак +, а для вычитания строк - знак − (а не просто дефис).

Учтите, что изменение количества строк на странице может повлиять на разбиение на страницы остальной части документа, поэтому рекомендуется использовать `\adaplayout` только в конце, когда в документе не будет дальнейших изменений, и сделать это в преамбуле. Затем мы переходим к первой странице, которую хотим адаптировать, делаем это и проверяем, как она влияет на последующие страницы; если это повлияет на него так, что другая страница нуждается в адаптации, мы добавляем её номер и компилируем еще раз, и так далее.

5.3.3 Использование нескольких макетов страниц

Если нам нужно использовать разные макеты в разных частях документа, лучше всего начать с определения *общего* макета, а затем различных альтернативных, тех, которые изменяют только те размеры, которые должны отличаться. Эти альтернативные макеты унаследуют все функции общего макета, которые не изменятся как часть его определения. Чтобы указать альтернативный макет и дать ему имя, которое мы позже сможем назвать, мы используем команду `\defelayout`, общий синтаксис которой:

```
\definelayout[Name/Number] [Configuration]
```

где *Name/Number* - это имя, связанное с новым дизайном, или номер страницы, на которой новый макет будет автоматически активирован, а *Configuration* будет содержать аспекты макета, которые мы хотим изменить по сравнению с общим макетом.

Когда новый макет связан с именем, чтобы вызвать его в определенном месте в документе, который мы используем:

```
\setuplayout[LayoutName]
```

и вернуться к общему виду:

```
\setuplayout[reset]
```

Если, с другой стороны, новый макет был связан с определенным номером страницы, он будет автоматически активирован при достижении страницы. Однако после активации, чтобы вернуться к общему дизайну, нам нужно будет явно указать это, даже если мы можем *полуавтоматизировать* это. Например, если мы хотим применить макет исключительно к страницам 1 и 2, мы можем написать в преамбуле документа:

```
\definelayout[1][...]
\definelayout[3][reset]
```

Эффект этих команд будет заключаться в том, что макет, определенный в первой строке, активируется на странице 1, а на странице 3 активируется другой макет, функция которого заключается только в возврате к общему макету.

С помощью `\defelayout[even]` мы создаем макет, который активируется на всех четных страницах; а с `\defelayout[odd]` макет будет применен ко всем нечетным страницам.

5.3.4 Прочие вопросы

А. Различение нечетных и четных страниц

В двусторонних печатных документах часто бывает, что верхний колонтитул, нумерация страниц и боковые поля различаются между четными и нечетными страницами. Страницы с четными номерами также называются левыми (оборотными) страницами, а нечетными страницами – правыми (лицевыми) страницами. В этих случаях также обычно меняется терминология, касающаяся полей, и мы говорим о внутренних и внешних полях. Первый расположен в ближайшей точке к тому месту, где страницы будут сшиты или скреплены, а второй-на противоположной стороне. На нечетных страницах внутреннее поле соответствует левому полю, а на четных страницах внешнее поле соответствует правому полю.

`\setuplayout` не имеет опции, прямо позволяющей нам указать, что мы хотим различать макет для четных и нечетных страниц. Это связано с тем, что для ConTEXt разница между обоими типами страниц устанавливается с помощью другого параметра: `\setuppagenumbering`, который мы увидим в section 5.4. Как только это будет установлено, ConTEXt предполагает, что страница, описанная с помощью `\setuplayout`, была нечетной, и строит четную страницу, применяя к ней инвертированные значения для нечетной страницы: спецификации, применимые к нечетной странице, применяются к левой части, на четной странице они применяются справа; и наоборот: те, которые применимы на странице с нечетным номером справа, применяются к странице с четным номером слева.

Б. Страницы с более чем одним столбцом

С помощью `\setuplayout` мы также можем видеть, что текст нашего документа распределен по двум или более столбцам, как это делают, например, газеты и некоторые журналы. Это контролируется опцией «`columns`», значение которой должно быть целым числом. Когда имеется более одного столбца, расстояние между столбцами указывается опцией «`columndistance`».

Этот параметр предназначен для документов, в которых весь текст (или большая его часть) распределен по нескольким столбцам. Если в документе, который в основном представляет собой документ с одним столбцом, мы хотим, чтобы конкретная часть состояла из двух или трех столбцов, нам не нужно изменять макет страницы, а просто использовать среду «`columns`» (см. section 12.2).

5.4 Нумерация страниц

По умолчанию ConTEXt использует арабские числа для нумерации страниц, и номер отображается в центре заголовка. Чтобы изменить эти функции, ConTEXt использует другие процедуры, которые, на мой взгляд, делают его излишне сложным, когда дело касается этого вопроса.

Во-первых, основные характеристики нумерации контролируются двумя разными командами:

`\setuppagenumbering` и `\setupuserpagenumber`.

`\setuppagenumbering` доступны следующие опции:

- **`alternative`**: этот параметр определяет, спроектирован ли документ таким образом, чтобы верхний и нижний колонтитулы были идентичными на всех страницах («`singlesided`» «односторонние»), или же они различают четные и нечетные страницы («`doublesided`» «двусторонние»). Когда эта опция принимает последнее значение, автоматически изменяются значения макета страницы, введенные «`setuplayout`», поэтому предполагается, что то, что указано в «`setuplayout`» относится только к страницам с нечетными номерами, и поэтому то, что расположено для левого поля, фактически относится к внутреннему полю (которое на страницах с четными номерами находится справа), а то, что расположено для правой стороны, фактически относится к внешнему полю, которое на четных страницах находится слева.

- **`state`**: указывает, будет ли отображаться номер страницы. Он допускает два значения: начало (будет отображаться номер страницы) и стоп (номера страниц будут подавлены). Название этих значений (start и stop) может заставить нас думать, что когда у нас есть «`state=stop`», страницы перестают нумероваться, а когда «`state=start`» нумерация начинается снова. Но это не так: эти значения влияют только на то, отображается номер страницы или нет.

- **`location`**: расположение: указывает, где он будет отображаться. Обычно нам нужно указать в этой опции два значения, разделенных запятой. Прежде всего, нам нужно указать, хотим ли мы, чтобы номер страницы был в заголовке («`header`») или в нижнем колонтитуле («`footer`»), а затем, где в верхнем или нижнем колонтитуле: это может быть «`left`», «`middle`», «`right`», «`inleft`», «`inright`», «`margin`», «`inmargin`», «`atmargin`» or «`marginedge`». Например: чтобы выровнять нумерацию по правому краю в нижнем колонтитуле, мы должны указать «`location={footer,right}`». С другой стороны, посмотрите, как мы заключили этот параметр в фигурные скобки, чтобы ConTEXt мог правильно интерпретировать разделительную запятую.

- **`style`**:указывает размер и стиль шрифта, которые будут использоваться для номеров страниц.

- **`color`**: указывает цвет, который будет применен к номеру страницы.

- **`left`**: выбирает команду или текст для выполнения слева от номера страницы.

- **`right`**:выбирает команду или текст для выполнения справа от номера страницы.

- **`command`**: выбирает команду, которой будет передан номер страницы в качестве параметра.

- **`width`**: указывает ширину, занимаемую номером страницы.

- **`strut`**: Я не уверен в этом. В моих тестах, когда «`strut=no`», число печатается точно по верхнему краю заголовка или внизу нижнего колонтитула, тогда как при «`strut=yes`» (значение по умолчанию) – между номером и краем.

`\setupuserpagenumber`, позволяет эти дополнительные параметры:

- **`numberconversion -- преобразование чисел`**: управляет типом нумерации, которая может быть арабской («`n`», «`numbers`»), строчными буквами («`a`», «`characters`»), прописными буквами («`A`»,«`Characters`»), строчными заглавными буквами («`KA`»), римскими строчными буквами («`i`»,

«r», «romannumerals»), латинскими буквами в верхнем регистре («I», «R», «Romannumerals») или прописные римские символы («KR»).

- **number**: указывает номер, который нужно присвоить первой странице, на основе которого будет рассчитываться оставшаяся часть.
- **numberorder**: если мы присвоим этому параметру значение «reverse», нумерация страниц будет в порядке убывания; это означает, что на последней странице будет 1, на предпоследней - 2 и т. д.
- **way**: позволяет нам указать, как будет проходить нумерация. Это может быть: по блокам, по главам, по разделам, по подразделам и т.д.
- **prefix**: позволяет указывать префикс к номерам страниц.
- **numberconversionset**: Объясняется далее.

В дополнение к этим двум командам также необходимо принять во внимание контроль чисел, связанных с макроструктурой документа section 6.6). С этой точки зрения \defineconversionset позволяет нам указывать разные виды нумерации для каждого из блоков макроструктуры. Например:

```
\defineconversionset
  [frontpart:pagenumber][][romannumerals]

\defineconversionset
  [bodypart:pagenumber][][numbers]

\defineconversionset
  [appendixpart:pagenumber][][Characters]
```

увидим, что первый блок в нашем документе (frontmatter) пронумерован строчными римскими числами, центральный блок (bodymatter) - арабскими числами, а приложения - прописными буквами.

Мы можем использовать следующие команды, чтобы получить номер страницы:

- \userpagenumber: возвращает номер страницы в том виде, в котором он был настроен с помощью \setuppagenumbering и \setupuserpagenumber.
- \pagenumber: возвращает то же число, что и предыдущая команда, но все еще в арабских числах.
- \realpagenumber: возвращает действительный номер страницы арабскими цифрами без учета этих характеристик.

Для получения номера последней страницы в документе есть три команды, параллельные предыдущим. Это: \lastuserpagenumber, \lastpagenumber и \lastrealpagenumber.

5.5 Принудительные или предлагаемые разрывы страниц

5.5.1 Команда \page

Алгоритм распределения текста в ConTEXt довольно сложен и основан на множестве вычислений и внутренних переменных, которые сообщают программе, где лучше всего вводить разрыв страницы с точки зрения типографской правильности. Команда \page позволяет нам влиять на этот алгоритм:

a. Предлагая определенные моменты как лучшее или самое неподходящее место для включения разрыва страницы.

- **no**: указывает, что место, где расположена команда, не является подходящим кандидатом для вставки разрыва страницы, поэтому, по возможности, разрыв должен быть сделан в другом месте документа.
- **preference**: сообщает ConTEXt, что точка, в которой он встречает команду, является *хорошим местом* для попытки разрыва страницы, хотя он и не заставит его там.
- **bigpreference**: указывает, что точка, в которой он сталкивается с командой, является *очень хорошим местом* для попытки разрыва страницы, но это тоже не доходит до ее принудительного выполнения.

Обратите внимание, что эти три параметра не вызывают и не предотвращают разрывы страниц, а только сообщают ConTEXt, что при поиске лучшего места для разрыва страницы следует учитывать то, что указано в этой команде. Однако в последнем случае место, где произойдет разрыв страницы, по-прежнему будет определяться ConTEXt.

б. Путем принудительного разрыва страницы в определенном месте; в этом случае мы также можем указать, сколько должно быть разрывов страниц, а также определенные особенности страниц, которые нужно вставить.

- **yes**: принудительный разрыв страницы в этой точке.
- **makeup**: аналогично «`yes`», но принудительный разрыв происходит немедленно, без предварительного размещения каких-либо плавающих объектов, размещение которых ожидается (см. section 13.1).
- **empty**: вставить в документ полностью пустую страницу.
- **even**: вставьте столько страниц, сколько необходимо, чтобы следующая страница стала четной.
- **odd**: вставить столько страниц, сколько необходимо, чтобы следующая страница была нечетной.
- **left, right**: аналогично двум предыдущим параметрам, но применимо только к двусторонним печатным документам, с разными верхними, нижними колонтитулами или полями в зависимости от того, четная или нечетная страница.
- **quadruple**: введите количество страниц, необходимое для того, чтобы следующая страница была кратна 4.

Наряду с этими параметрами, которые конкретно управляют разбиением на страницы, `\page` включает другие параметры, которые влияют на работу этой команды. В частности, параметр «`disable`» «отключить», который заставляет ConTEXt игнорировать команды `\page`, которые он находит оттуда, и параметр «`reset`», который производит противоположный эффект, восстанавливая эффективность будущих команд `\page`.

5.5.2 Соединение определенных строк или абзацев для предотвращения вставки между ними разрыва страницы

Иногда, если мы хотим предотвратить разрыв страницы между несколькими абзацами, использование команды `\page` может быть трудоемким, поскольку ее придется писать в каждой точке, где можно было вставить разрыв страницы. Более простая процедура для этого - поместить материал, который мы хотим сохранить, на той же странице в то, что TEX называет *вертикальным боксом*.

> В начале этого документа (на page 16) я указал, что внутри всё является *боксом* для TEX. Понятие бокса является фундаментальным в TEX для любых *сложных* операций; но управление им слишком сложно, чтобы включать его в это введение. Вот почему я лишь изредка упоминаю боксы.

После создания блоки TEX неделимы, что означает, что мы не можем вставить разрыв страницы, который разделил бы блок пополам. Вот почему, если мы помещаем материал, который хотим хранить вместе, в невидимый бокс, мы избегаем вставки разрыва страницы, которая могла бы разделить этот материал. Для этого используется команда `\vbox`, синтаксис которой

```
\vbox{Material}
```

где *Material* - это текст, который мы хотим сохранить вместе.

Некоторые среды ConTEXt действительно помещают свое содержимое в бокс. Например, «framedtext», поэтому, если мы создадим рамку для материала, который хотим сохранить вместе в этой среде, а также увидим, что рамка невидима (что мы делаем с опцией frame=off), мы добьемся того же.

5.6 Верхние и нижние колонтитулы

5.6.1 Команды для определения содержимого верхних и нижних колонтитулов

Если мы присвоили определенный размер верхнему и нижнему колонтитулам в макете страницы, мы можем включить в них текст с помощью команд \setupheadertexts и \setupfootertexts. Эти две команды похожи, с той лишь разницей, что первая активирует содержимое заголовка, а вторая - содержимое нижнего колонтитула. У обоих от одного до пяти аргументов.

1. Используемый с одним аргументом, он будет содержать текст верхнего или нижнего колонтитула, который будет помещен в центр страницы. Например: \setupfootertexts[pagenumber] напишет номер страницы в центре нижнего колонтитула.

2. При использовании с двумя аргументами содержимое первого аргумента будет размещено в левой части верхнего или нижнего колонтитула, а содержание второго аргумента - в правой части. Например, \setupheadertexts[Preface][pagenumber] наберет заголовок страницы, в котором слово «предисловие» написано с левой стороны, а номер страницы напечатан с правой стороны.

3. Если мы используем три аргумента, первый будет указывать *область*, в которой должны быть напечатаны два других. Под *областью* я имею в виду области страницы, упомянутые в section 5.2, другими словами: край, поле, заголовок ... Два других аргумента содержат текст, который должен быть помещен в левый край, поле и правый край, поле.

Использование его с четырьмя или пятью аргументами эквивалентно использованию его с двумя или тремя аргументами в тех случаях, когда проводится различие между четными и нечетными страницами, что, как мы знаем, происходит, когда установлено «alternate=doublesided» с \setuppagenumbering. В этом случае добавляются два возможных аргумента, чтобы отразить содержимое левой и правой сторон четных страниц.

Важной характеристикой этих двух команд является то, что когда они используются с двумя аргументами, предыдущий центральный верхний или нижний колонтитул (если он существовал) не перезаписывается, что позволяет нам писать другой текст в каждой области, если мы сначала напишем центральный текст (вызов команды с одним аргументом), а затем напишем тексты для любой стороны (вызовите его снова, теперь с двумя аргументами). Так, например, если мы напишем следующие команды

```
\setupheadertexts[and]
\setupheadertexts[Tweedledum][Tweedledee]
```

Первая команда напишет «and» в центре заголовка, а вторая напишет «Tweedledum» слева и «Tweedledee» справа, оставляя центральную область нетронутой, так как ее не было приказано переписывать. Результирующий заголовок теперь будет отображаться как

Tweedledum and Tweedledee

Только что приведенное мной объяснение работы этих команд является моим заключением после многих тестов. Объяснение этих команд, приведенное в обзоре ConTEXt*excursion*, основано на версии с пятью аргументами; а тот, что в справочном руководстве 2013 г., основан на версии с тремя аргументами. Я думаю, что моя яснее. С другой стороны, я не видел объяснения, почему второй вызов команды не перезаписывает предыдущий вызов, но вот как это работает, если мы сначала напишем центральный элемент в верхнем или нижнем колонтитуле, а затем - по обе стороны. Но если мы сначала напишем элементы с любой стороны в верхнем или нижнем колонтитуле, последующий вызов команды для записи центрального элемента

удалит предыдущие верхние или нижние колонтитулы. Почему? Я понятия не имею. Я думаю, что эти мелкие детали вносят ненужное усложнение и должны быть четко объяснены в официальной документации.

Более того, мы можем указать любую комбинацию текста и команд как фактическое содержимое верхнего или нижнего колонтитула. Но также следующие значения:

- **`date, currentdate`**: запишет (любую из них) текущую дату.
- **`pagenumber`**: напишет номер страницы.
- **`part, chapter, section...`**: напишет заголовок, соответствующий части, главе, разделу ... или любому другому структурному подразделению.
- **`partnumber, chapternumber, sectionnumber...`**: запишет номер части, главы, раздела ... или любого другого структурного подразделения.

Внимание: эти символические имена (`дата, текущая дата, номер страницы, глава, номер главы` и т.д.) интерпретируются как таковые только в том случае, если само символическое имя является единственным содержанием аргумента; но если мы добавим другой текст или команду форматирования, эти слова будут интерпретироваться буквально, и поэтому, например, если мы напишем `\setupheadertexts[chapternumber]`, мы получим номер текущей главы; но если мы напишем `\setupheadertexts[chapternumber]`, мы получим: «Chapter chapternumber». В этих случаях, когда содержание команды не является просто символическим словом, мы должны:

- Для `date, currentdate` и `pagenumber` используйте не символическое слово, а команду с тем же именем (`\date`, `\currentdate` или `\pagenumber`).
- Для `part, partnumber, chapter, chapternumber` и т.д. Используйте команду `\getmarking[Mark]`, которая возвращает содержание *Mark*. Так, например, `\getmarking[chapter]` вернет заголовок текущей главы, а `\getmarking[chapternumber]` вернет номер текущей главы.

Чтобы отключить верхние и нижние колонтитулы на определенной странице, используйте команду `\noheaderandfooterlines`, которая действует исключительно на странице, на которой она расположена. Если мы хотим удалить только номер страницы на определенной странице, мы должны использовать команду `\page[blank]`.

5.6.2 Форматирование верхних и нижних колонтитулов

Конкретный формат, в котором отображается текст верхнего или нижнего колонтитула, может быть указан в аргументах для `\setupheadertexts` или `\setupfootertexts` с помощью соответствующих команд форматирования. Однако мы также можем настроить это глобально с помощью `\setupheader` и `\setupfooter`, которые позволяют использовать следующие параметры:

- **`state`**: допускает следующие значения: `start, stop, empty, high, none, normal` или `nomarking`.
- **`style, leftstyle, rightstyle`**: настройка стиля текста верхнего и нижнего колонтитула. `style` влияет на все страницы: `leftstyle` левый стиль – четные страницы и `rightstyle` правый – нечетный.
- **`color, leftcolor, rightcolor`**: цвет верхнего или нижнего колонтитула. Это может повлиять на все страницы (вариант `color`) или только на четные страницы (`leftcolor`) или нечетные страницы (`rightcolor`)
- **`width, leftwidth, rightwidth`**: ширина всех верхних и нижних колонтитулов (`leftwidth`) или верхних / нижних колонтитулов на четных страницах (`leftwidth`) или нечетных (`rightwidth`).
- **`before`**: команда, выполняемая перед написанием верхнего или нижнего колонтитула.

- **after**: команда, выполняемая после написания верхнего или нижнего колонтитула.
- **strut**: если «yes», между заголовком и краем устанавливается вертикальное разделительное пространство. Если установлено «no», верхний или нижний колонтитул упирается в края областей верхнего или нижнего края.

5.6.3 Определение конкретных верхних и нижних колонтитулов и связывание их с командами раздела

Система верхнего и нижнего колонтитула ConTEXtпозволяет нам автоматически изменять текст в верхнем или нижнем колонтитуле, когда мы меняем главы или разделы; или когда мы меняем страницы, если мы установили разные верхние или нижние колонтитулы для нечетных и четных страниц. Но что он не позволяет, так это различать первую страницу (документа, главы или раздела) и остальные страницы. Для достижения последнего мы должны:

1. Определите конкретный верхний или нижний колонтитул.
2. Свяжите его с разделом, к которому он относится.

Определение конкретных верхних или нижних колонтитулов выполняется с помощью команды `\definetext`, синтаксис которой:

```
\definetext
  [Name] [Type]
  [Content1] [Content2] [Content3]
  [Content4] [Content5]
```

где *Name* – это имя, присвоенное верхнему или нижнему колонтитулу, с которым мы имеем дело; *Type* может быть верхний `header` или нижний `footer` колонтитул, в зависимости от того, какой из двух мы определяем, а оставшиеся пять аргументов содержат содержимое, которое мы хотим для нового верхнего или нижнего колонтитула, аналогично тому, как мы видели `\setupheadertexts` и `\setupfootertexts`. функция. Как только мы это сделаем, нам нужно связать новый верхний или нижний колонтитул с некоторым конкретным разделом с помощью `\setuphead`, используя параметры верхнего и нижнего колонтитула (которые не объясняются в Chapter 6).

Таким образом, в следующем примере верхний колонтитул на первой странице каждой главы будет скрыт, а в нижнем колонтитуле будет отображаться номер страницы по центру:

```
\definetext[ChapterFirstPage] [footer] [pagenumber]
\setuphead
  [chapter]
  [header=high, footer=ChapterFirstPage]
```

5.7 Вставка текстовых элементов по краям и полям страницы

Верхний и нижний края, а также правое и левое поля обычно не содержат какого-либо текста. Однако ConTEXt позволяет размещать там некоторые текстовые элементы. В частности, для этого доступны следующие команды:

- `\setuptoptexts`: позволяет нам размещать текст у верхнего края страницы (над областью заголовка).
- `\setupbottomtexts`: позволяет нам размещать текст у нижнего края страницы (под областью нижнего колонтитула).
- `\margintext`, `\atleftmargin`, `\atrightmargin`, `\ininner`, `\ininneredge`, `\ininnermargin`, `\inleft`, `\inleftedge`, `\inleftmargin`, `\inmargin`, `\inother`, `\inouter`, `\inouteredge`, `\inoutermargin`,

`\inright`, `\inrightedge`, `\inrightmargin`: позволяют размещать текст по боковым краям и полям документа.

Первые две команды работают точно так же, как `\setupheadertexts` и `\setupfootertexts`, и формат этих текстов можно даже заранее настроить с помощью `\setuptop` и `\setupbottom`, подобно тому, как `\setupheader` позволяет нам настраивать тексты для `\setupheadertexts`. Для всего этого я ссылаюсь на то, что уже сказал в section 5.6. Единственная небольшая деталь, которую нужно добавить, это то, что текст, настроенный для `\setuptoptexts` или `\setupbottomtexts`, не будет виден, если в макете страницы не было зарезервировано места для верхнего (`top`) или нижнего (`bottom`) края. Для этого см. section 5.3.1.

Что касается команд, предназначенных для размещения текста на полях документа, все они имеют похожий синтаксис:

```
\CommandName[Reference][Configuration]{Text}
```

где *Reference* и *Configuration* – необязательные аргументы; первый используется для возможных перекрестных ссылок, а второй позволяет нам настроить текст на полях. Последний аргумент, заключенный в фигурные скобки, содержит текст, который нужно разместить на поле.

Из этих команд более общей является `\margintext`, поскольку она позволяет размещать текст на любом из полей или боковых краях страницы. Остальные команды, как указывает их название, помещают текст в само поле (правое или левое, внутреннее или внешнее) или край (правое или левое, внутреннее или внешнее). Эти команды тесно связаны с макетом страницы, потому что если, например, мы используем `\inrightedge`, но не зарезервировали место в макете страницы для правого края, ничего не будет видно.

Параметры конфигурации для `\margintext` следующие:

- **`location`**: указывает, на каком поле будет размещен текст. Оно может быть `left`, `right` или, в двусторонних документах, `outer` or `inner`. По умолчанию он `left` в односторонних документах, а в двусторонних – `outer`.
- **`width`**:ширина, доступная для печати текста. По умолчанию будет использоваться вся ширина поля.
- **`margin`**: указывает, будет ли текст размещен на самом поле `margin` или по краю `edge`.
- **`align`**: выравнивание текста. Здесь используются те же значения, что и в `\setupalign` 11.6.1.
- **`line`**: позволяет указать количество строк смещения текста на полях. Итак, `line=1` сместит текст на одну строку ниже, а `line=-1` на одну строку выше.
- **`style`**: команда или команды для указания стиля текста, который будет помещен на поля.
- **`color`**: цвет текста на полях.
- **`command`**: имя команды, которой в качестве аргумента будет передан текст, который будет помещен на поле. Эта команда будет выполнена перед записью текст. Например, если мы хотим нарисовать рамку вокруг текста, мы можем использовать «`[command=\framed]{Text}`».

Остальные команды допускают те же параметры, за исключением `location` местоположения и `margin` поля. В частности, команды `\atrightmargin` и `\atleftmargin` помещают текст полностью прикрепленным к основной части страницы. Мы можем установить разделительное пространство с помощью параметра `distance`, о котором я не упоминал, говоря о `\margintext`, потому что в своих тестах я не заметил никакого эффекта на эту команду.

В дополнение к указанным выше параметрам эти команды также поддерживают другие параметры (`strut, anchor, method, category, scope, option, hoffset, voffset, dy, bottomspace, threshold и stack`), которые я не упомянул, потому что

они не задокументированы и откровенно говоря, я не совсем уверен, для чего они нужны. Мы можем угадать те, у кого есть имена вроде *distance*, а остальные? В вики упоминается только опция `stack` , говоря, что она используется для эмуляции команды marginpars в LaTeX, но мне это не кажется очень понятным.

Команда `\setupmargindata` позволяет нам глобально настраивать тексты на каждом поле. Так, например,

```
\setupmargindata[right][style=slanted]
```

гарантирует, что все тексты в правом поле написаны наклонно.

Мы также можем создать нашу собственную настраиваемую команду с помощью `\definemargindata[Name][Configurati`

Глава 6

Структура документа

6.1 Структурные подразделения в документах

За исключением очень коротких текстов (например, письма), документ обычно разбивается на блоки или текстовые группы, которые обычно следуют иерархическому порядку. Не существует стандартного способа наименования этих блоков: например, в романах структурные подразделения обычно называются «chapters», хотя некоторые - более длинные - имеют блоки большего размера, обычно называемые «parts», которые группируют числа. глав вместе. В театральных произведениях различают «действия» и «сцены». Учебные пособия делятся (иногда) на «части» и «уроки», «темы» или «главы», которые, в свою очередь, также часто имеют внутренние подразделения; такие же сложные иерархические подразделения часто существуют в других академических или технических документах (таких как тексты, подобные настоящему, посвященные объяснению компьютерной программы или системы. Даже законы структурированы в „книги” (самые длинные и самые сложные, такие как в виде Кодексов), «title», «chapters», «section», «subsections». Научно-технические документы также могут достигать шести, семи, а иногда и восьми уровней глубины вложенности для такого рода подразделения.

В этой главе основное внимание уделяется анализу механизма, который ConTEXt предлагает для поддержки этих структурных подразделений. Я буду называть их общим термином «sections».

> Нет четкого термина, который позволил бы нам в общем обозначать все эти виды структурных подразделений. Термин «section», который я выбрал, фокусируется на структурном делении, а не на чем-либо еще, хотя недостатком является то, что одно из заранее определенных структурных подразделений ConTEXtназывается «section». Я надеюсь, что это не вызовет путаницы, полагая, что будет достаточно легко определить из контекста, говорим ли мы о разделе как об общей и общей ссылке на структурные подразделения или о конкретном подразделении, которое ConTEXt называет section.

Каждый «раздел» «section» (в общем) подразумевает:

- Достаточно большое *структурное подразделение документа*, которое, в свою очередь, может включать другие подразделения более низкого уровня. С этой точки зрения «sections» «разделы» подразумевают текстовые блоки с иерархической связью между ними. С точки зрения разделов документ в целом можно рассматривать как дерево. Документ *сам по себе* является стволом, каждая из его глав - ветвью, которая, в свою очередь, может иметь ветви, которые также могут разделяться, и так далее.

  Наличие четкой структуры очень важно для чтения и понимания документа. Но это задача автора, а не наборщика. И хотя ConTEXt не может сделать нас лучшими авторами, чем мы, полный набор команд разделов, которые он включает, с четкой иерархией между ними, может помочь нам в написании более структурированных документов.

- *Имя структуры*, которое мы могли бы назвать ее «заголовком» «title» или «меткой» «label». Это наименование структуры печатается:

- Всегда (или почти всегда) в том месте документа, где начинается структурное деление.
- Иногда также в оглавлении, в верхнем или нижнем колонтитуле страниц, занимаемых данным разделом.

ConTEXt позволяет нам автоматизировать все эти задачи таким образом, что функции форматирования, с помощью которых должно быть напечатано название структурной единицы, должны указываться только один раз, а также следует или не следует включать его в оглавление, или в верхних или нижних колонтитулах. Для этого ConTEXt нужно только знать, где начинается и заканчивается каждая структурная единица, как она называется и на каком иерархическом уровне она находится.

6.2 Типы разделов и их иерархия

ConTEXt различает *пронумерованные* и *ненумерованные разделы*. Первые, как следует из их названия, нумеруются автоматически и отправляются в оглавление, а иногда и в верхние и / или нижние колонтитулы страниц.

ConTEXt имеет иерархически упорядоченные предопределенные команды разделов, представленные table 6.1.

Level	Numbered sections	Unnumbered sections
1	`\part`	–
2	`\chapter`	`\title`
3	`\section`	`\subject`
4	`\subsection`	`\subsubject`
5	`\subsubsection`	`\subsubsubject`
6	`\subsubsubsection`	`\subsubsubsubject`
...	...	...

Таблица 6.1 Команды секций в ConTEXt

В отношении заранее определенных разделов следует сделать следующие пояснения:

- В table 6.1 команды раздела показаны в их традиционном виде. Но мы сразу увидим, что они также могут использоваться как окружения (например, (`\startchapter ... \stopchapter` и что это действительно рекомендуемый подход.
- Таблица содержит только первые 6 уровней раздела. Однако в своих тестах я обнаружил до 12 уровней: после texsubsubsubsection идет `\subsubsubsubsection`, и так далее до`\subsubsubsubsubsubsubsubsubsection`, или `\subsubsubsubsubsubsubsubsubsubject`.

  Но следует иметь в виду, что указанные выше (чрезмерно глубокие) нижние уровни вряд ли улучшат понимание текста! Во-первых, у нас, вероятно, будут большие разделы, которые неизбежно будут посвящены нескольким вопросам, и это затруднит читателю *понимание* их содержания. Чрезмерная глубина уровней также может означать, что читатель теряет общее ощущение текста, и в результате возникает эффект чрезмерной фрагментации задействованного материала. Насколько я понимаю, в целом достаточно четырех уровней; очень иногда может потребоваться перейти на шесть или семь уровней, но большая глубина редко бывает хорошей идеей.

  С точки зрения написания исходного файла тот факт, что создание дополнительных подуровней означает добавление еще одного «подуровня» к предыдущему уровню, может сделать исходный файл почти нечитаемым: пытаться определить уровень команды - не шутка. назвал «subsubsubsubsubsection», так как я должен подсчитать все «subs»! Поэтому я советую, что если нам действительно нужно так много уровней глубины, начиная с пятого уровня (подсекция), нам лучше определить наши собственные команды раздела section 6.5, давая им имена, которые более ясны, чем предопределенные.

- Самый высокий уровень раздела (`\part`) существует только для пронумерованных заголовков и имеет особенность, заключающуюся в том, что заголовок части не печатается. Однако, даже если заголовок не печатается, вводится пустая страница (на которой мы можем предположить, что заголовок печатается после того, как пользователь перенастроил команду), и нумерация *части* принимается во внимание для расчета нумерации главы и других разделов.

> Причина, по которой версия \part по умолчанию ничего не печатает, заключается в том, что, согласно вики ConTEXt, почти всегда заголовок на этом уровне требует определенного макета; и хотя это правда, мне это не кажется достаточно веской причиной, поскольку на практике главы и разделы также часто переопределяются, и тот факт, что части ничего не печатают, вынуждает начинающего пользователя окунуться в документацию чтобы увидеть, что происходит не так.

- Хотя первый уровень разделения является «частью», это только теоретический и абстрактный характер. В конкретном документе первый уровень секционирования будет соответствовать первой команде секционирования в документе. То есть в документе, который включает не части, а главы, глава будет первым уровнем. Но если документ также не включает главы, а только разделы (section), иерархия для этого документа начнется с разделов (section).

6.3 Синтаксис

Все команды раздела, включая любые уровни, созданные пользователем (см. section 6.5), допускают следующие альтернативные формы синтаксиса (если, например, мы используем уровень «`section`»):

```
\section [Label] {Title}
\section [Options]
\startsection [Options] [Variables] ... \stopsection
```

В трех приведенных выше способах аргументы в квадратных скобках необязательны и могут быть опущены. Мы рассмотрим их отдельно, но в первую очередь это помогает прояснить, что в Mark IV рекомендуется третий из этих трех подходов.

- В первой синтаксической форме, которую мы могли бы назвать «*классической*», команда принимает два аргумента: один необязательный в квадратных скобках, а другой - обязательный в фигурных скобках. Необязательный аргумент нужен для того, чтобы связать команду с меткой, которая будет использоваться для внутренних ссылок (см. section 8.2). Обязательным в фигурных скобках является название раздела.

- Две другие формы синтаксиса больше похожи на стиль ConTEXt: все, что нужно знать команде, передается через значения и параметры, заключенные в квадратные скобки.

  > Напомним, что в разделах sections 3.3.1 и 3.4 я сказал, что в ConTEXt область действия команды указывается в фигурных скобках, а ее параметры - в квадратных скобках. Но если подумать, заголовок конкретной команды секционирования не входит в область ее применения, поэтому, чтобы соответствовать общему синтаксису, его следует вводить не в фигурные скобки, а качестве опции. ConTEXt допускает это исключение, потому что это классический способ выполнения действий в TEX, но он предоставляет альтернативные формы синтаксиса, которые больше соответствуют его общему дизайну.

  Параметры относятся к типу присвоения значений (OptionName = Value) и выглядят следующим образом:

  - **`reference`**: метка для перекрестных ссылок.
  - **`title`**: заголовок раздела, который будет напечатан в теле документа.
  - **`list`**: заголовок раздела, который будет напечатан в оглавлении.
  - **`marking`**: заголовок раздела, который будет напечатан в верхних или нижних колонтитулах страницы.
  - **`bookmark`**: заголовок раздела, который будет преобразован в закладку в файле PDF.
  - **`ownnumber`**: эта опция используется в случае, если раздел не нумеруется автоматически; в этом случае эта опция будет включать номер, присвоенный рассматриваемому разделу.

  Конечно, параметры «`list`» «список», «`marking`» «отметка» и «`bookmark`» «закладка» следует использовать только в том случае, если мы хотим использовать другой заголовок для замены основного набора заголовков с опцией «`title`» «заголовок». Это очень полезно, например, когда

заголовок слишком длинный для заголовка; хотя для этого мы также можем использовать команды \nomarking и \nolist (что-то очень похожее). С другой стороны, мы должны иметь в виду, что если текст заголовка (опция «title») включает любые запятые, то его нужно будет заключить в фигурные скобки, как полный текст, так и запятую, чтобы гарантировать, что ConTEXt знает, что запятая является частью заголовка. То же самое и с опциями: «list», «marking» и «bookmark». Поэтому, чтобы не следить за тем, есть ли в названии запятые, я считаю хорошей идеей выработать привычку всегда заключать значение любого из этих параметров в фигурные скобки.

Так, например, следующие строки создадут главу под названием «Тестовая глава», связанную с меткой «test» для перекрестных ссылок, в то время как заголовок будет «Chapter test» вместо «A Test Chapter».

```
\chapter
  [
    title={A Test Chapter},
    reference={test},
    marking={Chapter test}
  ]
```

Синтаксис \startSectionType превращает раздел в *среду*. Это больше согласуется с тем фактом, что, как я сказал в начале, в фоновом режиме каждый раздел представляет собой дифференцированный блок текста, хотя ConTEXt по умолчанию не считает *среды*, созданные командами раздела, *группами*. Точно так же эту процедуру рекомендует Mark IV; вполне возможно, потому что этот способ создания разделов требует от нас четко указывать, где начинается и заканчивается каждый раздел, что упрощает согласованность структуры и, скорее всего, лучше поддерживает вывод XML и EPUB. Фактически, для вывода XML это необходимо.

Когда мы используем \startSectionName, в квадратных скобках разрешается использовать одну или несколько переменных в качестве аргументов. Затем их значение можно будет использовать позже в других местах документа с помощью команды \structureuservariable.

Наличие пользовательских переменных обеспечивает очень расширенное использование в ConTEXt благодаря тому факту, что могут быть приняты решения о том, компилировать фрагмент или нет, или каким способом мы это сделаем, или с помощью какого шаблона в зависимости от значения конкретной переменной. Однако эти утилиты ConTEXt выходят за рамки материала, с которым я хотел бы ознакомиться в этом введении.

6.4 Формат и конфигурация разделов и их заголовков

6.4.1 Команды \setuphead и \setupheads

По умолчанию ConTEXt назначает определенные функции каждому уровню секционирования, которые в основном (но не только) влияют на формат, в котором заголовок отображается в основной части документа, но не на способ отображения заголовка в оглавлении или заголовки и колонтитулы. Мы можем изменить эти функции с помощью команды \setuphead , чей синтаксис:

```
\setuphead[Sections][Options]
```

где

- **Sections** Разделы - это имя одного или нескольких разделов (разделенных запятыми), на которые повлияет команда. Это может быть:
  - Любой из предопределенных разделов (часть, глава, заголовок и т. Д.), И в этом случае мы можем ссылаться на них либо по имени, либо по их уровню. Для обозначения их по уровню мы используем слово «section-*NumLevel*», где *NumLevel* - это номер уровня соответствующего раздела. Таким образом, «section-1» равен «part», «section-2» равен «chapter» и т.д.

  - Любой вид раздела мы сами определили. По этому поводу см. section 6.5.

- **Options** это опции конфигурации. Они относятся к типу явного присвоения значения (OptionName = value). Число подходящих вариантов очень велико (более шестидесяти), поэтому я объясню, сгруппировав их по категориям в соответствии с их функциями. Однако я должен отметить, что мне не удалось определить, для чего предназначены некоторые из этих параметров и как они используются. Об этих вариантах я говорить не буду.

Ранее я сказал, что `\setuphead` влияет на явно указанные разделы. Но это не означает, что изменение одного раздела никоим образом не должно влиять на другие, если они не были прямо упомянуты в команде. На самом деле верно и обратное: изменение раздела влияет на другие разделы, связанные с ним, даже если это не было явно указано в команде. Связь между различными разделами бывает двух видов:

- Ненумерованные команды связаны с соответствующей пронумерованной командой того же уровня, так что изменение внешнего вида пронумерованной команды повлияет на ненумерованную команду того же уровня; но не наоборот: изменение ненумерованной команды не влияет на нумерованную команду. Это означает, например, что если мы изменим какой-то аспект «`chapter`» (level 2) (уровень 2), мы также изменим этот аспект в «заголовке»; но изменение «заголовка» «`title`» не повлияет на «главу» «`chapter`».

- Команды связаны иерархически, так что если мы изменим определенные функции на определенном уровне, это изменение повлияет на все уровни, следующие за ним. Это происходит только с некоторыми функциями. Цвет, например: если мы устанавливаем, что подразделы будут отображаться красным, мы также меняем на красный цвет подподразделы, подподразделы и т.д. Но этого не происходит с другими функциями, например, со стилем шрифта

Наряду с `\setuphead` ConTEXt предоставляет команду `\setupheads`, которая глобально влияет на все команды раздела. Со ссылкой на эту команду в вики ConTEXt говорится, что некоторые люди сказали, что она не работает. Согласно моим тестам, эта команда работает с некоторыми параметрами, но не работает с другими. В частности, он не работает с опцией «`style`» «стиль», что поразительно, поскольку стиль заголовков, скорее всего, единственное, что мы хотели бы изменить глобально, чтобы он повлиял на все заголовки. Но, согласно моим тестам, он работает с другими параметрами, такими как, например,«`number`» или «`color`». Так, например, `\setupheads[color=blue]` гарантирует, что все заголовки в нашем документе будут напечатаны синим цветом.

Поскольку мне немного лень проверять каждую опцию, чтобы увидеть, работает она или нет с `\setupheads` (помните, что их более шестидесяти), в дальнейшем я буду ссылаться только на `\setuphead`.

Наконец: прежде чем исследовать конкретные параметры, мы должны отметить то, что говорится в ConTEXt wiki, хотя, вероятно, это сказано не в нужном месте: некоторые параметры работают только в том случае, если мы используем синтаксис `\start`*`SectionName`*.

> Эта информация относится к `\setupheads`, но не для `\setuphead`, где объясняется большая часть параметров и где, если это должно быть сказано только в одном месте, кажется наиболее разумным местом для этого. С другой стороны, в информации упоминается только вариант опции «изнутри», но не уточняется, происходит ли это также с другими опциями.

6.4.2 Части заголовка раздела

Прежде чем перейти к конкретным параметрам, которые позволяют нам настраивать внешний вид заголовков, рекомендуется начать с указания, что заголовок раздела может иметь до трех разных частей, которые ConTEXt позволяет нам форматировать вместе или по отдельности. Эти элементы заголовка следующие:

- **Само название**, то есть текст, из которого он состоит. В принципе, этот заголовок отображается всегда, за исключением разделов типа «`part`», где заголовок не отображается по умолчанию.

Параметр, определяющий, отображается ли заголовок или нет, - это «placehead», значения которого могут быть «yes», «no», «hidden», «empty» или «section». Смысл первых двух ясен. Но я не так уверен в результатах остальных значений этого параметра. Следовательно, если мы хотим, чтобы заголовок отображался в разделах первого уровня, наша настройка должна быть такой:

```
\setuphead
  [part]
  [placehead=yes]
```

Заголовки определенных разделов, как мы уже знаем, могут автоматически отправляться в колонтитулы и оглавление. Используя list и marking команд раздела, мы можем указать альтернативный заголовок, который будет отправлен вместо него. Также возможно при написании заголовка использовать команды \nolist или \nomarking , чтобы определенные части заголовка были заменены многоточием в оглавлении или заголовке. Например:

```
\chapter{Influences of \nomarking{19th century} impressionism \nomarking{in the 21st century}}
```

Напишет «Influences of ... impressionism ...» в header.

- **Нумерация**. Это справедливо только для пронумерованных разделов (part, chapter, section, subsection...), но не для ненумерованных разделов (title, subject, subsubject). Фактически, пронумерован конкретный раздел или нет, зависит от опций «number» и «incrementnumber», возможные значения которых – yes и no. В пронумерованных разделах оба они установлены как yes, а в ненумерованных разделах - как no.

  Почему есть два варианта управления одним и тем же? Потому что на самом деле две опции управляют двумя вещами; один - пронумерован раздел или нет (incrementnumber инкрементный номер), а другой - отображается ли номер или нет (number номер). Если для раздела установлены incrementnumber=yes и number=no, мы получим раздел, который не пронумерован внешне (визуально), но по-прежнему учитывается внутри. Это было бы полезно для включения такого раздела в оглавление, поскольку обычно оно включает только пронумерованные разделы. По этому поводу см. subsection A в section 7.1.7Подраздел A.

- **The label** Ярлык для заголовка. В принципе этот элемент в заголовках пуст. Но мы можем связать с ним значение, и в этом случае перед номером и фактическим заголовком будет напечатана метка, которую мы присвоили этому уровню. Например, в заголовках глав мы можем захотеть напечатать слово «Глава» или слово «Часть» для частей. Мы не используем для этого \setuphead, а используем команду \setuplabeltext. Эта команда позволяет нам присвоить текстовое значение меткам различных уровней секционирования. Так, например, если мы хотим написать «Глава» в нашем документе перед заголовками глав, мы должны установить:

```
\setuplabeltext
  [chapter=Глава~]
```

В этом примере после присвоенного имени я включил зарезервированный символ «~», который вставляет неразрывное пробел после слова. Если мы не против, чтобы между меткой и числом произошел разрыв строки, мы могли бы просто добавить пробел. Но это пустое пространство (любого рода) важно; без него номер будет привязан к метке, и мы увидим, например, «Chapter1» вместо «Chapter 1».

6.4.3 Контроль нумерации (в пронумерованных разделах)

Мы уже знаем, что предопределенные пронумерованные разделы (часть, глава, раздел ...) и то, пронумерован ли конкретный раздел или нет, зависит от параметров «number» и «incrementnumber», установленных с помощью \setuphead.

По умолчанию нумерация различных уровней выполняется автоматически, если только мы не присвоили значение «yes» параметру «ownnumber». Когда «ownumber=yes» должен быть указан номер, присвоенный каждой команде. Это делается:

- Если команда вызывается с использованием классического синтаксиса, путем добавления аргумента с числом перед текстом заголовка. Например:
  `\chapter{13}{Название главы}` создаст главу, которой вручную был присвоен номер 13.

- Если команда была вызвана с синтаксисом, специфичным для ConTEXt (`\SectionType [Options]` или `\startSectionType[Options]`), с параметром «ownnumber». Например:

Когда ConTEXt автоматически выполняет нумерацию, он использует внутренние счетчики, в которых хранятся номера разных уровней; таким образом, есть счетчик для частей, другой для глав, другой для разделов и т. д. Каждый раз, когда ConTEXt находит команду раздела, он выполняет следующие действия:

- Увеличивает счетчик, связанный с уровнем, соответствующим этой команде, на „1”.
- сбрасывает связанные счетчики на всех уровнях ниже, чем у данной команды, на 0.

Это означает, например, что каждый раз, когда обнаруживается новая глава, счетчик глав увеличивается на 1, а все команды раздела, подраздела, подраздела и т.д. возвращаются на 0; но счетчик не влияет на Части.

Чтобы изменить число, с которого следует начать отсчет, используйте команду `\setupheadnumber` следующим образом:

```
\setupheadnumber[SectionType][Номер, с которого начинается счет]
```

где *Number, от которого следует вести счет*, – это число, от которого будут считаться секции любого типа. Таким образом, если *Number, от которого следует вести счет*, равен нулю, первая секция будет равна 1; если он равен 10, первая секция будет 11.

Эта команда также позволяет нам изменить шаблон для автоматического увеличения; таким образом, мы можем, например, подсчитать главы или разделы попарно или по три. Итак, `\setupheadnumber[section][+5]` даст главы с номерами 5 из пяти; и `\setupheadnumber[chapter][14, +5]` покажет, что первая глава начинается с 15 (14 + 1), вторая будет 20 (15 + 5), третья 25 и т. д.

По умолчанию нумерация разделов отображается на арабском языке. номера, и нумерация всех предыдущих уровней включена. То есть: в документе, в котором есть части, главы, разделы и подразделы, конкретный подраздел будет указывать, какой части, главе и разделу он соответствует. Таким образом, четвертый подраздел второго раздела третьей главы первой части будет «1.3.2.4».

Два основных параметра, управляющих отображением чисел:

- **`conversion`** (преобразование): управляет типом используемой нумерации. Допускает множество значений в зависимости от типа нумерации, которую мы хотим:

  – **Нумерация арабскими цифрами**: Классическая нумерация: 1, 2, 3, ... получается с помощью значений `n, N` or `цифр`.

  – **Нумерация римскими цифрами**. Сделать это можно тремя способами:

    ⋆ Римские числа в верхнем регистре:`I, R, Romannumerals`.
    ⋆ Римские числа в нижнем регистре: `i, r, Romannumerals`.
    ⋆ Римские цифры в заглавных буквах (small caps): `KR, RK`.

  – **Нумерация буквами**. Сделать это можно тремя способами:

  - Заглавные буквы: `A, Символ`
  - Строчные буквы: `a, символ`
  - Буквы в заглавных буквах (small caps): `AK, KA`

- **Нумерация прописью**. То есть мы пишем слово, обозначающее число. Так, например, „3” превращается в „Три”. ТЭто можно сделать двумя способами:

  - Слова, начинающиеся с заглавной буквы: `Слова`.
  - Все слова в нижнем регистре: `слова`.

- **Нумерация с помощью символов**: в нумерации на основе символов используются разные наборы символов, в которых каждому символу присваивается числовое значение. Поскольку наборы символов, используемые ConTEXt имеют очень ограниченное количество, этот тип нумерации целесообразно использовать только тогда, когда максимальное число, которое должно быть достигнуто, не слишком велико. ConTEXt предоставляет четыре различных набора символов: набор 0, набор 1, набор 2 и набор 3 соответственно. Ниже приведены символы, которые каждый из этих наборов использует для нумерации. Обратите внимание, что максимальное число, которое может быть достигнуто, составляет 9 в наборе 0 и наборах 1 и 12 в наборе 2 и наборе 3:

  Set 0: • – ⋆ ⊳ ◦ ● ⬤ □ ✓
  Set 1: ⋆ ⋆⋆ ⋆⋆⋆ ‡ ‡‡ ‡‡‡ * ** ***
  Set 2: ⋆ † ‡ ⋆⋆ †† ‡‡ ⋆⋆⋆ ††† ‡‡‡ ⋆⋆⋆⋆ †††† ‡‡‡‡
  Set 3: ⋆ ⋆⋆ ⋆⋆⋆ ‡ ‡‡ ‡‡‡ ¶ ¶¶ ¶¶¶ § §§ §§§

- **`sectionsegments:`** этот параметр позволяет нам контролировать, отображать ли нумерацию для предыдущих уровней. Мы можем указать, какие предыдущие уровни будут отображаться. Это делается путем определения начального и последнего отображаемых уровней. Идентификация уровня может быть произведена по его номеру (часть = 1, раздел = 2, раздел = 3 и т. Д.) Или имени (часть, глава, раздел и т. Д.). Так, например, «`sectionsegments=2:3`» указывает, что должна отображаться нумерация глав и разделов. Это в точности то же самое, что сказать «`sectionsegments=chapter:section`». Если мы хотим указать, что отображаются все числа выше определенного уровня, мы можем использовать в качестве значения «`optionsegments`» *Initial Level:all* или *InitialLevel:**. Например, «`sectionsegments=3:*`» означает, что нумерация отображается, начиная с уровня 3 (раздел).

So, for example, imagine that we want the parts to be numbered with Roman numerals in capital letters; the chapters with Arabic numerals, but without including the number of the part to which they belong; the sections and subsections with Arabic numerals including the chapter and section numbers, and the subsections with capital letters. We should write the following:

```
\setuphead[part][conversion=I]
\setuphead[chapter]     [conversion=n, sectionsegments=2]
\setuphead[section]     [conversion=n, sectionsegments=2:3]
\setuphead[subsection]  [conversion=n, sectionsgments=2:4]
\setuphead[subsubsection][conversion=A, sectionsegments=5]
```

6.4.4 Цвет и стиль заголовка

У нас есть следующие варианты управления стилем и цветом:

- **The style** Стиль контролируется параметрами «`style`», «`numberstyle`» и «`textstyle`» в зависимости от того, хотим ли мы повлиять на весь заголовок, только на нумерацию или только на текст. С помощью любой из этих опций мы можем включать команды, влияющие на шрифт; а именно: конкретный шрифт, стиль (римский, без засечек или пишущая машинка), альтернативный (курсив, полужирный, наклонный ...) и размер. Если мы хотим указать только одну функцию стиля, мы

можем сделать это, используя имя стиля (например, «жирный» для полужирного), или указав его аббревиатуру («bf»), или команду, которая его генерирует (`\bf`, выделенные жирным шрифтом). Если мы хотим указать несколько функций одновременно, мы должны сделать это с помощью команд, которые их генерируют, записывая их одну за другой. С другой стороны, имейте в виду, что если мы укажем только одну функцию, остальные функции стиля будут установлены автоматически со значениями документа по умолчанию, поэтому редко рекомендуется устанавливать только одну функцию стиля.

- **The colour** Цвет устанавливается с помощью параметров «`color`», «`numbercolor`» и «`textcolor`» в зависимости от того, хотим ли мы установить цвет всего заголовка или только цвет нумерации или текста. Указанный здесь цвет может быть одним из предопределенных цветов ConTEXt или каким-либо другим цветом, который мы определили сами и ранее присвоили имя. Однако здесь мы не можем напрямую использовать команду определения цвета.

В дополнение к этим шести параметрам, есть еще пять доступных вариантов для создания некоторых более сложных функций, с помощью которых мы можем делать практически все, что захотим. Это:«`command`», «`numbercommand`», «`textcommand`»,«`deepnumbercommand`» и «`deeptextcommand`». Начнем с объяснения первых трех:

- **`command`** указывает команду, которая принимает два аргумента: номер и заголовок раздела. Это может быть обычная команда ConTEXt или команда, которую мы определили сами.
- **`numbercommand`** похожа на «`command`», но эта команда принимает только аргумент с номером раздела.
- **`textcommand`** также похожа на «`command`», но принимает только аргумент с текстом заголовка.

Эти три варианта позволяют нам делать практически все, что мы хотим. Например, если я хочу, чтобы разделы были выровнены по правому краю, заключены в рамку и с разрывом строки между номером и текстом, я могу просто создать команду, которая сделает это, а затем указать эту команду как значение параметра «`command`» команда. Это будет достигнуто с помощью следующих строк:

```
\define[2]\AlignSection
  {\framed[frame=on, width=broad,align=flushright]{#1\\#2}}

\setuphead
  [section]
  [command=\AlignSection]
```

Когда мы одновременно устанавливаем параметры «`command`» и «`style`», команда применяется к заголовку с его стилем. Это означает, например, что если мы установили «`textstyle=\em`» и «`textcommand=\WORD`», `\WORD` (которая использует заглавные буквы в тексте, который принимает в качестве аргумента) будет применена к заголовку с его style, то есть: `\WORD{\em Title text}`. Если мы хотим, чтобы это было сделано наоборот, то есть чтобы стиль применялся к содержимому заголовка после применения команды, мы должны использовать вместо параметры «`textcommand`» and «`numbercommand`», параметры «`deeptextcommand`» and «`deepnumbercommand`». Это, в приведенном выше примере, сгенерирует «`{\em\WORD{Title text}}`».

В большинстве случаев не было бы никакой разницы в том, чтобы сделать это так или иначе. Но в некоторых случаях разица может быть.

6.4.5 Расположение номера и текста заголовка

Опция «`alternative`» управляет двумя вещами одновременно: расположением нумерации по отношению к тексту заголовка и расположением самого заголовка (включая номер и текст) по отношению к странице, на которой он отображается, и содержимому раздел. Это две разные вещи, но, поскольку обе они управляются одним и тем же параметром, они управляются одновременно.

Расположение заголовка по отношению к странице и первому абзацу содержания раздела определяется следующими возможными значениями «alternative»:

- **text:** заголовок раздела объединен с первым абзацем его содержания. Эффект аналогичен тому, который создается в LaTeXс помощью \paragraph и \subparagraph.

  The section title is integrated with the first paragraph of its contents. The effect is similar to what is produced in LaTeX with \paragraph and \subparagraph.

- **paragraph:** заголовок раздела будет независимым абзацем.

- **normal:** заголовок раздела будет помещен на место по умолчанию, предоставленное ConTeXt для конкретного типа рассматриваемого раздела. Обычно это «paragraph».

- **middle:** заголовок написан как отдельный абзац по центру. Если это нумерованная команда, номер и текст разделены на разных строках по центру.

  Эффект, аналогичный тому, что достигается с помощью «alternative=middle», достигается с помощью параметра «align», который управляет выравниванием заголовка. Может принимать значения «left», «middle» или «flushright». Но если мы центрируем заголовок с помощью этой опции, номер и текст появятся в одной строке.

- **margintext:** при выборе этого параметра весь заголовок (нумерация и текст) печатается в пространстве, зарезервированном для поля.

Расположение номера по отношению к тексту заголовка обозначается следующими возможными значениями «alternative»:

- **margin/inmargin:** заголовок - это отдельный абзац. Нумерация пишется в отведенном для поля месте. Я не понял разницы между использованием «margin» и «inmargin».

- **reverse:** обратный: заголовок составляет отдельный абзац, но нормальный порядок меняется на обратный, и сначала печатается текст, а затем номер.

- **top/bottom:** в заголовках, текст которых занимает более одной строки, эти два параметра определяют, будет ли нумерация выровнена с первой строкой заголовка или с последней строкой соответственно.

6.4.6 Команды или действия

Можно указать одну или несколько команд, которые выполняются перед печатью заголовка (параметры «before») или после (параметр «after»). Эти параметры широко используются для визуального обозначения заголовка. Например: если мы хотим добавить больше вертикального промежутка между заголовком и текстом, который ему предшествует, «before=\blank» добавит пустую строку. Чтобы добавить еще больше места, мы могли бы написать «before={\blank[3*big]}». В этом случае мы заключили значение параметра в фигурные скобки, чтобы избежать ошибки. Мы также могли бы визуально указать расстояние между предыдущим текстом и следующим текстом с помощью «before=backslash hairline, after=\hairline», который будет рисовать горизонтальную линию до и после заголовка.

Параметры «before» и «after» очень похожи на параметры «commandbefore» и «commandafter». В соответствии с моими тестами я пришел к выводу, что разница в том, что первые два выполняют действия до и после начала набора заголовка как такового, в то время как последние два относятся к командам, которые будут выполняться до и после набора *текста заголовка*.

Если мы хотим вставить разрыв страницы перед заголовком, мы должны использовать параметр «page», который позволяет, среди других значений, «yes» для вставки разрыва страницы, «left», чтобы вставить столько разрывов страницы, сколько необходимо, чтобы гарантировать чтобы заголовок

начинался на четной странице, «right», чтобы гарантировать, что заголовок начинается на нечетной странице, или «no», если мы хотим отключить принудительный разрыв страницы. Эта опция, с другой стороны, для уровней ниже «chapter» будет работать, только если используется «continue=no», в противном случае она не будет работать, если раздел, подраздел или команда находятся на первой странице главы.

> По умолчанию главы начинаются на новой странице в ConTEXtЕсли установлено, что разделы также начинают новую страницу, возникает проблема, что делать с первым разделом главы, который, возможно, находится в начале главы: если этот раздел также начинает разрыв страницы, мы заканчиваем вверх со страницей, которая открывает главу, содержащую только название главы, что не очень эстетично. Вот почему мы можем установить опцию «continue», имя, я должен сказать, что мне не очень понятно: если «continue=yes», разрыв страницы не будет применяться к разделам, которые находятся на первой странице. главы. Если «continue=no», разрыв страницы все равно будет применяться.

Если вместо секционных команд мы используем среды раздела (\start ... \stop), у нас также есть опция «insidesection», с помощью которой мы можем указать одну или несколько команд, которые будут выполнены после того, как заголовок будет набран, и мы уже находимся внутри раздела. Эта опция позволит нам, например, убедиться, что сразу после начала главы оглавление будет автоматически набираться с помощью («insidesection=\placecontent»)

6.4.7 Другие настраиваемые функции

Помимо уже знакомых, с помощью \setuphead мы можем настроить следующие дополнительные функции:

- **Interlined**. Управляется «межстрочным пространством», которое принимает в качестве значения имя межстрочной команды, ранее созданной с помощью \defineinterlinespace и настроенной с помощью \setupinterlinespace.
- **Alignment**. Параметр «align» влияет на выравнивание абзаца, содержащего заголовок. Среди прочего, он может иметь следующие значения: «flushleft» (слева), «flushright» (справа), «middle» (по центру), «inner» (внутреннее поле) и «outer» (внешнее поле).
- **Margin**. С помощью опции «margin» мы можем вручную установить поле заголовка.
- **Indenting the first paragraph**. Отступ первого абзаца. Значение параметра «indentnext» (может быть «да», «нет» или «авто») определяет, будет ли первая строка первого абзаца раздела иметь отступ. Должен ли он быть с отступом (в документе, где первая строка абзацев обычно имеет отступ) - дело вкуса.
- **Width**. По умолчанию заголовки занимают необходимую им ширину, если только она не превышает ширину строки. В этом случае заголовок займет более одной строки. Но с помощью опции «width» мы можем назначить определенную ширину для заголовка. Параметры «numberwidth» и «textwidth» соответственно определяют ширину нумерации или ширину текста заголовка.
- **Отделение цифры и текста**. Параметры «distance» и «textdistance» позволяют нам контролировать расстояние, отделяющее число от его текста.
- Стиль верхних и нижних колонтитулов разделов. Для этого мы используем параметры «header» и «footer».

6.4.8 Другие параметры \setuphead options

С помощью опций, которые мы уже видели, мы можем видеть, что возможности настройки названий разделов практически неограниченны. Однако в \setuphead есть около тридцати опций, о которых я не упоминал. Большинство из них потому, что я не понял, для чего они предназначены или как они используются, некоторые потому, что их объяснение заставило бы меня углубиться в аспекты, которые я не намерен рассматривать в этом введении.

6.5 Определение новых команд раздела

Мы можем определить наши собственные команды раздела с помощью `\definehead`, синтаксис которого:

```
\definehead[CommandName][Model][Configuration]
```

где

- **CommandName** представляет имя, которое будет иметь новая команда раздела.
- **Model** это имя существующей команды раздела, которая будет использоваться в качестве модели, от которой новая команда изначально унаследует все свои характеристики.

  Фактически, новая команда наследует от модели гораздо больше, чем ее начальные характеристики: она становится своего рода настраиваемым экземпляром модели, но совместно с ней, например, внутренний счетчик, контролирующий нумерацию.

- **Configuration** это индивидуальная конфигурация нашей новой команды. Здесь мы можем использовать те же параметры, что и в `\setuphead`.

Нет необходимости настраивать новую команду во время ее создания. Это можно сделать позже с помощью `\setuphead` и, фактически, в примерах, приведенных в руководствах ConTEXt и его вики, это кажется нормальным способом.

6.6 Макроструктура документа

Главы, разделы, подразделы, заголовки ..., структурируют документ; они это организуют. Но вместе со структурой, проистекающей из таких команд, в некоторых печатных книгах, особенно в книгах из академического мира, существует своего рода *макроупорядочение* материала книги, принимая во внимание не его содержание, а функция, которую выполняет каждая из этих больших частей в книге. Вот как мы различаем:

- Начальная часть документа, содержащая титульную страницу, страницу подтверждения, страницу посвящения, оглавление, возможно, предисловие, страницу презентации и т.д.
- Основная часть документа, содержащая основной текст документа, разделенный на части, главы, разделы, подразделы и т. Д. Эта часть обычно является самой обширной и важной.
- Дополнительный материал, состоящий из дополнений или приложений, которые развивают или иллюстрируют некоторые проблемы, рассматриваемые в основной части, или предоставляют дополнительную документацию, написанную не автором основной части, и т.д.
- Заключительная часть документа, где мы можем найти библиографию, указатели, глоссарии и т.д.

В исходном файле мы можем разграничить каждую из этих частей через среды, показанные в table 6.2.

Part of the document	Command
Initial part	\startfrontmatter [Options] ... \stopfrontmatter
Main body	\startbodymatter [Options] ... \stopbodymatter
Appendices	\startappendices [Options] ... \stopappendices
Final part	\startbackmatter [Options] ... \stopbackmatter

Таблица 6.2 Environments that reflect the document's macrostructure

Четыре среды позволяют использовать те же четыре параметра: «`page`», «`before`», «`after`» и «`number`», и их значения и полезность такие же, как и в `\setuphead`. (см. section 6.4), хотя мы должны

отметить, что здесь параметр «`number = no`» устраняет нумерацию всех команд секционирования в среде.

Включение любого из этих больших разделов в наш документ имеет смысл только в том случае, если необходимо провести какое-то различие между ними. Возможно, заголовки или нумерация страниц frontmatter. Конфигурация каждого из этих блоков достигается

`\setupsectionblock` , синтаксис которого:

```
\setupsectionblock[Block name] [Options]
```

где*Block name* может быть `frontpart`, `bodypart`, `appendix` или `backpart`, а параметры могут быть такими же, как только что упомянутые: «`page`»,«`number`», «`before`» и «`after`». Так, например, чтобы гарантировать, что в *frontmatter* страницы пронумерованы римскими числами, в преамбуле нашего документа мы должны написать:

```
\setupsectionblock
  [frontpart]
  [
    before={\setuppagenumbering[conversion=Romannumerals]}
  ]
```

Конфигурация ConTEXt по умолчанию для этих четырех блоков подразумевает, что:

- Четыре блока начинают новую страницу.
- Нумерация разделов изменяется в каждом из этих блоков:
  - По умолчанию в `frontmatter` and `backmatter` все нумерованные разделы не пронумерованы.
  - В `bodymatter` главы имеют арабскую нумерацию.
  - В `appendices` главы нумеруются прописными буквами.

Также можно создавать новые блоки разделов с помощью

`\definesectionblock`.

Глава 7

Оглавление, указатели, списки

Оглавление и указатель - это глобальный аспект документа. Почти все документы будут иметь оглавление, в то время как только некоторые документы будут иметь указатель. Для многих языков (но не для английского) и оглавление, и указатель подпадают под общий термин «указатель». Для англоязычных читателей оглавление обычно находится в начале (документа или, возможно, в некоторых случаях также и в начале глав), а указатель - в конце.

Любой из них подразумевает конкретное применение механизма внутренних ссылок, объяснение которого включено в раздел section 8.2.

7.1 Содержание

7.1.1 Общий вид содержания

В предыдущей главе мы рассмотрели команды, позволяющие установить структуру документа в том виде, в котором он был написан. В этом разделе основное внимание уделяется оглавлению и индексу, которые в некотором роде отражают структуру документа. Оглавление очень полезно для получения представления о документе в целом (оно помогает контекстуализировать информацию) и для поиска точной точки, где может быть расположен конкретный отрывок. Книги с очень сложной структурой, с множеством разделов и подразделов с разной степенью глубины, похоже, требуют иного типа оглавления, поскольку плохо детализированное (возможно, только с первыми двумя или тремя уровнями разбиения на разделы) очень помогает получить общее представление о содержании документа, но не очень полезен для определения местоположения конкретного отрывка; с другой стороны, в отличие от очень подробного оглавления, где легко пропустить лес за деревьями и потерять общий вид документа. Вот почему иногда книги с особенно сложной структурой включают более одного оглавления: одно не слишком подробное в начале, показывающее основные части, и более подробное оглавление в начале каждой главы, а также, возможно, индекс в конце.

Все они могут быть относительно легко сгенерированы ConTEXt автоматически. Мы можем:

- Создавать полное или частичное оглавление в любом месте документа.
- Определиться с содержанием любого из них.

- Настроить их внешний вид до мельчайших деталей.
- Включить в оглавление гиперссылки, которые позволят нам перейти непосредственно к нужному разделу.

Фактически, эта последняя утилита включена по умолчанию во все оглавления при условии, что в документе включена функция интерактивности. См. По этому поводу раздел section 8.3

Объяснение этого в справочном руководстве ConTEXt на мой взгляд, несколько сбивает с толку, что, я думаю, связано с тем, что сразу вводится слишком много информации. Механизм построения оглавлений ConTEXt состоит из множества частей; и тексту, который пытается объяснить их все сразу, трудно быть ясным, особенно для читателя, который плохо знаком с этой темой. Напротив, объяснение в вики wiki практически ограничено примерами: очень полезно для изучения приемов, но неадекватно - я думаю - для понимания механизма и того, как он работает. Вот почему стратегия, которую я решил использовать для объяснения вещей в этом введении, начинается с предположения о том, что не строго верно (или не совсем верно): что в ConTEXt есть что-то, называемое *оглавлением*. Начиная с этого, объясняются обычные команды для создания оглавления, и, когда эти команды и их конфигурация хорошо известны, я думаю, что настал момент представить - хотя и на теоретическом, а не на более практическом уровне - информацию о тех частях механизма, которые до этого были опущены. Знание этих дополнительных частей позволяет нам создавать гораздо более настраиваемые оглавления, чем те, которые мы можем назвать обычными, созданными с помощью команд, объясненных до этого момента; однако в большинстве случаев этого делать не нужно.

7.1.2 Полностью автоматическое оглавление с заголовком

Основные команды для создания автоматически сгенерированного оглавления (TOC) из пронумерованных разделов документа (части, главы, раздела и т. Д.) - это `\completecontent` и `\placecontent`. Основное различие между двумя командами состоит в том, что первая добавляет заголовок к оглавлению; для этого непосредственно перед оглавлением вставляется ненумерованная глава, название которой по умолчанию - «Оглавление».

Следовательно `\completecontent`:

- Вставляет в место нахождения новую ненумерованную главу, озаглавленную «Содержание».

  Напомним, что в ConTEXt для создания ненумерованного раздела на том же уровне, что и главы, используется команда `\title` (see section 6.2). Поэтому на самом деле `\completecontent` вставляет не главу *Chapter* (`\chapter`) а заголовок *Title* (`\title`). Я не сказал этого, потому что думаю, что читателя может сбить с толку использование здесь имен ненумерованных команд раздела, поскольку термин *Title* также имеет более широкий смысл, и читателю легко не отождествить его с конкретным уровнем секционирования, о котором мы говорим.

- Эта *глава* (на самом деле `\title`) отформатирована точно так же, как и остальные ненумерованные главы в документе; который по умолчанию включает разрыв страницы.
- Оглавление печатается сразу после заголовка.

Первоначально сгенерированное оглавление em полное, о чем мы можем сделать вывод из имени команды, которая его генерирует (`\\completecontent`). Но, с одной стороны, мы можем ограничить уровень глубины оглавления, как описано в разделе section 7.1.3, а с другой, поскольку эта команда em чувствительна к тому месту, где она находится в исходном файле (см. То, что сказано далее о `\placecontent`), если `\completecontent` не найдено в начале документа, возможно, что созданное оглавление не является полным; и в некоторых местах исходного файла возможно даже игнорирование команды. Если это произойдет, решение состоит в том, чтобы вызвать команду с параметром «`criterium=all`». Относительно этой опции см. также section 7.1.3.

Чтобы изменить наименование заголовка по умолчанию, присвоенного оглавлению, мы используем команду `\setupheadtext`, синтаксис которой:

```
\setupheadtext[Language][Element=Name]
```

где *Language* является необязательным и относится к идентификатору языка, используемому ConTEXt (section 10.5), а *Element* относится к элементу, имя которого мы хотим изменить ((«content» в случае таблицы содержания), а *Name* - это имя или название, которое мы хотим дать нашему оглавлению. Например

```
\setupheadtext[en][content=Оглавление]
```

гарантирует, что содержание, созданное с помощью \completecontent, будет наименоваться «Оглавление» вместо «Content».

Более того, \completecontent допускает те же параметры конфигурации, что и \placecontent, для объяснения которых я обращаюсь к (следующему разделу).

7.1.3 Автоматическое оглавление без заголовка

Общая команда для вставки оглавления без заголовка, автоматически сгенерированная из команд секционирования документа, - это \placecontent, синтаксис которой:

```
\placecontent[Options]
```

В принципе, таблица содержания будет содержать абсолютно все пронумерованные разделы, хотя мы можем ограничить его глубину с помощью команды \setupcombinedlist (о которой мы поговорим далее). Так, например:

```
\setupcombinedlist[content][list={chapter,section}]
```

ограничит состав оглавления главами и разделами.

Особенностью этой команды является то, что она чувствительна к своему местоположению в исходном файле. Это очень легко объяснить с помощью нескольких примеров, но гораздо сложнее, если мы хотим точно указать, как работает команда и какие заголовки включаются в оглавление в каждом случае. Итак, начнем с примеров:

- \placecontent помещенный в начало документа, перед первой командой раздела (часть, глава или раздел, в зависимости от ситуации) сгенерирует полное оглавление.

  Я не совсем уверен, что оглавление, сгенерированное по умолчанию, является *полным*, я считаю, что оно включает достаточное количество уровней секционирования, чтобы быть полным в большинстве случаев; но я подозреваю, что дальше восьмого уровня секционирования он не пойдет. В любом случае, как упоминалось выше, мы можем отрегулировать уровень секционирования, которого достигает TOC, с помощью

  ```
  \setupcombinedlist[content][list={chapter, section, subsection, ...}]
  ```

- Напротив, та же самая команда, расположенная внутри части, главы или раздела, будет исключительно генерировать оглавление содержимого этого элемента или, другими словами, глав, разделов и других более низких уровней разделения определенной части или разделов (и других уровни) определенной главы или подразделов определенного раздела.

Что касается технического и подробного объяснения, чтобы правильно понять работу \placecontent по умолчанию, важно помнить, что различные разделы на самом деле являются *средами* для ConTEXt Mark IV, которые начинаются с \startSectionType и заканчиваются \stopSectionType и может содержаться в других командах раздела нижнего уровня. Итак, принимая это во внимание, мы можем сказать, что \placecontent по умолчанию генерирует оглавление, которое будет включать только:

- Элементы, принадлежащие к *среде* (уровень раздела), в которой размещена команда. Это означает, что команда, помещенная в главу, не будет включать разделы или подразделы из других глав.

- Элементы, у которых уровень секционирования ниже уровня, соответствующего точке, где расположена команда. Это означает, что если команда находится в главе, будут включены только

разделы, подразделы и другие более низкие уровни; но если команда находится в разделе, она будет разделена, чтобы сделать оглавление уровня подраздела.

Кроме того, для создания оглавления требуется, чтобы `\placecontent` находился *перед* первым разделом главы, в которой оно находится, или перед первым подразделом раздела, в котором оно находится, и т.д.

Я не уверен, что ясно объяснил вышесказанное. Возможно, с несколько более подробным примером, чем предыдущие, мы сможем лучше понять, что я имею в виду: давайте представим следующую структуру документа:

- Chapter 1
  - Section 1.1
  - Section 1.2
    - Subsection 1.2.1
    - Subsection 1.2.2
    - Subsection 1.2.3
  - Section 1.3
  - Section 1.4
- Chapter 2

Итак: `\placecontent`, помещенный перед главой 1, сгенерирует полное оглавление, подобное тому, которое сгенерировано `\completecontent`, но без заголовка. Но если команда размещена в главе 1 и перед разделом 1.1, оглавление будет только главы; и если он находится в начале раздела 1.2, оглавление будет только содержимым этого раздела. Но если команда будет размещена, например, между разделами 1.1 и 1.2, она будет проигнорирована. Она также будет проигнорирована, если будет размещена в конце раздела или в конце документа.

Все это, конечно, относится только к случаю, когда команда не включает параметры. В частности, опция `criterium` изменит это поведение по умолчанию.

Из вариантов, разрешенных `\placecontent` , я объясню только два из них, наиболее важные для настройки оглавления и, более того, единственные, которые (частично) задокументированы в справочном руководстве ConTEXtПараметр `criterium`, который влияет на содержимое оглавления по отношению к месту в исходном файле, где расположена команда; и параметр `alternative`, который влияет на общий макет создаваемого оглавления.

7.1.4 Элементы для включения в TOC: параметр `criterium`

Операция по умолчанию `\placecontent` по отношению к позиции команды в исходном файле была объяснена выше. Опция `criterium` изменяет эту операцию. Среди прочего, она может принимать следующие значения:

- `all`: оглавление будет полным, независимо от места в исходном файле, где находится команда.
- `previous`: оглавление будет включать только команды раздела (уровня, на котором мы находимся) *previous* до `\placecontent`. Эта опция предназначена для оглавлений, которые написаны в конце рассматриваемого документа или раздела.
- `part, chapter, section, subsection...`: подразумевает, что оглавление должно быть ограничено указанным уровнем разделения.
- `component`: в многофайловых проектах (см. section 4.6) будет генерировать только оглавление, соответствующее *компоненту*, в котором найдена команда texplacecontent или `\completecontent`.

7.1.5 Макет оглавления: альтернативный вариант: опция `alternative`

Опция `alternative` управляет общей компоновкой оглавления. Её основные значения можно увидеть в table 7.1.

alternative	Содержание записей TOC	Примечания
a	Number – Title – Page	One line per entry
b	Number – Title – Spaces – Page	One line per entry
c	Number – Title – Leader dots – Page	One line per entry
d	Number – Title – Page	Continuous TOC
e	Title	Framed
f	Title	Выровнено по левому краю, по правому краю или по центру
g	Title	По центру

Таблица 7.1 Способы форматирования оглавления

Первые четыре альтернативных значения предоставляют всю информацию о каждом разделе (его номер, заголовок и номер страницы, с которой он начинается), и поэтому подходят как для бумажных, так и для электронных документов. Последние три альтернативы только информируют нас о заголовке, поэтому они подходят только для электронных документов, где нет необходимости знать номер страницы, на которой начинается раздел, при условии, что оглавление включает гиперссылку на него, что по умолчанию происходит в ConTEXt. .

Кроме того, я считаю, что для того, чтобы по-настоящему оценить различия между различными альтернативами, читателю лучше всего создать тестовый документ, в котором он или она может их подробно проанализировать.

7.1.6 Формат записей оглавления

Мы видели, что опция `alternative` `\placecontent` или `\completecontent` позволяет нам контролировать общий макет оглавления, то есть какая информация будет отображаться для каждого заголовка и будут ли разрывы строк, разделяющие разные заголовки. Окончательная корректировка каждой записи оглавления производится с помощью команды `\setuplist`, синтаксис которой следующий:

```
\setuplist[Element][Configuration]
```

где *Element* относится к определенному типу раздела. Это может быть часть, глава, раздел и т.д. Мы также можем настроить более одного элемента одновременно, разделив их запятыми. *Конфигурация* имеет до 54 возможностей, многие из которых, как обычно, явно не задокументированы; но это не мешает тем, которые задокументированы, или тем, которые недостаточно ясны, разрешить полную настройку TOC.

Теперь я объясню наиболее важные параметры, сгруппировав их в соответствии с их полезностью, но прежде чем переходить к ним, давайте вспомним, что запись оглавления, в зависимости от значения `alternative`, может иметь до трех различных компонентов: номер раздела, заголовок раздела и номер страницы. Параметры конфигурации позволяют нам настраивать различные компоненты глобально или по отдельности:

- *Включение (или нет) различных компонентов*: если мы выбрали альтернативу, которая включает, помимо заголовка, номер раздела и номер страницы (варианты ’a’ ’b’ ’c’ или ’d’), параметры `headnumber=no` или `pagenumber=no`, это означает, что для конкретного уровня, который мы настраиваем, номер раздела (`headnumber`) или номер страницы (`pagenumber`) не отображается.

- *Цвет и стиль*: мы уже знаем, что запись, генерирующая определенный раздел в оглавлении, может иметь (в зависимости от альтернативы) до трех различных компонентов: номер раздела, заголовок и номер страницы. Мы можем совместно указать стиль и цвет для трех компонентов, используя параметры стиля `style` и цвета `color`, или сделать это индивидуально для каждого компонента с помощью `numberstyle, textstyle` или `pagestyle` (для стиля) and `numbercolor, textcolor` или `pagecolor` для цвета.

  Чтобы управлять внешним видом каждой записи, помимо самого стиля, мы можем применить некоторую команду ко всей записи или к одному из ее различных элементов. Для этого есть параметры `command`, `numbercommand`, `pagecommand` and `textcommand`. Указанная здесь команда может

быть стандартной командой ConTEXt или командой, созданной нами самостоятельно. Номер раздела, текст заголовка и номер страницы будут переданы в качестве аргументов опции `command`, в то время как заголовок раздела будет передан в качестве аргумента в текстовую команду, а номер страницы - в `pagecommand`. Так, например, следующее предложение гарантирует, что заголовки разделов будут написаны (ложными) маленькими заглавными буквами:

```
\setuplist[section][textcommand=\Cap]
```

- *Разделение других элементов оглавления*: параметры до `before` и после `after` позволяют нам указать команды, которые будут выполняться перед (`before`) и после (`after`) набора текста оглавления. Обычно эти команды используются для установки либо интервала, либо некоторого разделительного элемента между предыдущей и последующей записями.

- *Отступ элемента*: задается с помощью параметра `margin`, который позволяет нам установить величину отступа слева, который будет иметь записи настраиваемого уровня.

- *Гиперссылки, встроенные в оглавление*: по умолчанию записи указателя включают гиперссылку на страницу документа, где начинается рассматриваемый раздел. Используя опцию взаимодействия, мы можем отключить эту функцию (`interaction=no`) или ограничить часть записи индекса, где будет гиперссылка, которая может быть номером раздела (`interaction=number` или `interaction=sectionnumber`), заголовком раздела (`interaction=text` или `interaction=title`) или номером страницы (`interaction=page` или `interaction=pagenumber`).

- *Прочие аспекты*:

  - `width`: указывает расстояние между номером и заголовком раздела. Это может быть расстояние или ключевое слово, задающее точную ширину номера секции.

  - `symbol`: позволяет заменить номер раздела на *symbol*. Поддерживаются три возможных значения: `один`, `два` и `три`. Значение `none` для этой опции удаляет номер раздела из TOC.

  - `numberalign`: указывает выравнивание элементов нумерации; он может быть left, right, middle, flushright, flushleft.

Среди множества вариантов конфигурации TOC нет ни одного, который позволяет нам напрямую управлять межстрочным интервалом. По умолчанию он будет применяться ко всему документу. Однако часто желательно, чтобы строки в оглавлении были немного более плотными, чем в остальной части документа. Для этого мы должны заключить команду, которая генерирует оглавление(`\placecontent` или `\completecontent`) внутри группы, в которой установлен другой межстрочный интервал. Например:

```
\start
  \setupinterlinespace[small]
  \placecontent
\stop
```

7.1.7 Ручная корректировка таблицы содержания

Мы уже объяснили две основные команды для создания оглавлений (`\placecontent` и `\completecontent`), а также их параметры. С помощью этих двух команд автоматически создаются оглавления, состоящие из существующих пронумерованных разделов в документе или в блоке или сегменте документа, к которому относится оглавление. Теперь я объясню некоторые настройки *settings*, которые мы можем сделать, чтобы содержание оглавления не было таким автоматическим *automatic*. Это подразумевает:

- Возможность также включения некоторых ненумерованных заголовков разделов в оглавление.

- Возможность вручную отправить в оглавление конкретную запись, которая не соответствует наличию пронумерованного раздела.

- Возможность исключения определенного пронумерованного раздела из TOC.
- Возможность того, что заголовок определенного раздела, отраженного в оглавлении, не совпадает в точности с заголовком, включенным в тело документа.

A. Включение ненумерованных разделов в TOC

Механизм, с помощью которого ConTEXt создает оглавление, означает, что все пронумерованные разделы включаются автоматически, что, как я уже сказал (см. section 6.4.2), зависит от двух параметров (`number` and `incrementnumber`) (число и число приращения), которые мы можем изменить с помощью`\setuphead` для каждого вида раздела. Там также объяснялось, что Там также объяснялось, что тип раздела, в котором `incrementnumber=yes` и `number=no`, будет иметь внутреннюю, но не внешнюю нумерацию.

Следовательно, если мы хотим, чтобы конкретный тип ненумерованного раздела - например, заголовок `title` - был включен в оглавление, мы должны изменить значение параметра `incrementnumber` для этого типа раздела, установив его на `yes`, а затем включить этот тип раздела среди тех, которые будут отображаться в оглавлении, что, как объяснено выше, выполняется с помощью `\setupcombinedlist`:

```
  \setuphead
    [title]
    [incrementnumber=yes]

 \setupcombinedlist
   [content]
   [list={chapter, title, section, subsection, subsubsection}]
```

Затем мы можем, если захотим, отформатировать эту запись с помощью `\setuplist` точно так же, как и любые другие; Например:

```
\setuplist[title][style=bold]
```

Примечание: Только что описанная процедура будет включать в себя все экземпляры в нашем документе соответствующего ненумерованного типа раздела (в нашем примере разделы с типом заголовка).

Note: Только что описанная процедура будет включать все экземпляры в нашем документе соответствующего ненумерованного типа раздела (в нашем примере разделы типа `title`). Если мы хотим включить в оглавление только конкретное вхождение этого типа раздела, желательно сделать это с помощью процедуры, описанной ниже.

Б. Ручное добавление записей в оглавление

Мы можем отправить либо запись (имитирующую существование раздела, который на самом деле не существует), либо команду в оглавление из любой точки исходного файла.

Чтобы отправить запись, имитирующую существование раздела, который на самом деле не существует, используйте `\writetolist`, синтаксис которого:

```
\writetolist[SectionType][Options]{Number}{Text}
```

в котором

- Первый аргумент указывает уровень, который эта запись раздела должна иметь в оглавлении: глава `chapter`, раздел `section`, подраздел `subsection` и т.д.
- Второй аргумент, который является необязательным, позволяет настроить эту запись определенным образом. Если отправленный вручную ввод опущен, он будет отформатирован, как и все записи уровня, указанного в первом аргументе; хотя, я должен отметить, что в моих тестах мне не удалось заставить это работать.

Как в официальном списке команд ConTEXt (см. section ??), так и в вики нам сообщают, что этот аргумент допускает те же значения, что и \setuplist, который является командой, которая позволяет нам форматировать различные записи TOC. Но, я настаиваю, в моих тестах мне не удалось каким-либо образом изменить внешний вид записи оглавления, введенной вручную.

- Третий аргумент должен отражать нумерацию элемента, введенного в оглавление, но я также не смог заставить это работать в моих тестах.
- Последний аргумент включает текст, который будет введен в оглавление.

Это полезно, например, если мы хотим отправить конкретный ненумерованный раздел, но только его в оглавление. В section A объясняется, как получить целую категорию ненумерованных разделов для отправки в оглавление; но если мы хотим отправить ему только конкретное вхождение типа раздела, удобнее использовать команду \writetolist. Итак, например, если мы хотим, чтобы раздел нашего документа, содержащий библиографию, не был пронумерованным, но все же включался в оглавление, мы должны написать:

```
\subject{Bibliography}
\writetolist[section]{}{Bibliography}
```

Посмотрите, как мы используем ненумерованную версию раздела `section`, который является `subject`, для раздела, но мы отправляем его в индекс вручную, как если бы это был пронумерованный раздел (`section`).

Другая команда, предназначенная для ручного воздействия на оглавление, - это \writebetweenlist, которая используется для отправки не самой записи, а *команды* оглавлению из определенного места в документе. Например, если мы хотим включить строку между двумя элементами в оглавление, мы могли бы написать следующее в любом месте документа, расположенного между двумя соответствующими разделами:

```
\writebetweenlist[section]{\hrule}
```

B. Исключить конкретный раздел из оглавления

Оглавление построено из *типов разделов*, установленных, как мы уже знаем, с помощью параметра `list` команды \setupcombinedlist, поэтому, если определенный тип раздела должен появиться в оглавлении, нет способа исключить из него конкретный раздел, который для любых причины, по которым мы не хотим в ТОП.

Обычно, если мы не хотим, чтобы раздел отображался там, мы бы использовали его ненумерованное эквивалентное значение, например, `title` вместо `chapter`, `subject` вместо `section` и т.д. Эти разделы не отправляются в оглавление, и они не пронумерованы.

Однако, если по какой-либо причине мы хотим, чтобы определенный раздел был пронумерован, но не отображался в оглавлении, даже если это делают другие типы этого типа, мы можем использовать трюк *trick*, который состоит в создании нового типа раздела, который является клоном рассматриваемого раздела. Например:

```
\definehead[MySubsection][subsection]
\section{First section}
\subsection{First subsection}
\MySubsection{Second subsection}
\subsection{Third subsection}
```

Это гарантирует, что при вставке типа раздела `MySubsection` счетчик подразделов будет увеличиваться, поскольку этот раздел является *клоном* подразделов, но оглавление не будет изменено, поскольку по умолчанию он не включает типы `MySubsection`.

Г. Текст заголовка раздела

Если мы не хотим, чтобы заголовок определенного раздела, включенного в оглавление, был идентичен заголовку, отображаемому в теле документа, у нас есть две доступные нам процедуры:

- Создание раздела не с традиционным синтаксисом (`SectionType{Title}`), а с `\SectionType [Options]` или с `\startSectionType [Options]`, и назначьте текст, который мы хотим записать в оглавлении, параметру списка `list` (см. section 6.3).
- При написании заголовка рассматриваемого раздела в теле документа используйте команду `\nolist`: эта команда заставляет текст, который она принимает в качестве аргумента, заменяется в оглавлении многоточием. Например:

  ```
  \chapter
    [title={An \nolist{approximate and slightly repetitive}
             introduction to the reality of the obvious}]
  ```

  будет набираться как заголовок главы в теле документа «An approximate and slightly repetitive introduction to the reality of the obvious», но в оглавление будет отправлен следующий текст «An ... introduction to the reality of the obvious».

  > **Attention:** то, что я только что указал о команде `\nolist`, изложено как в справочном руководстве ConTEXt так и в вики wiki. Однако у меня это вызывает ошибку компиляции, сообщая мне, что команда `\nolist` не определена.

7.2 Списки, комбинированные списки и оглавление на основе списка

Внутренне для ConTEXt оглавление - это не что иное, как *комбинированный список*, который, в свою очередь, как следует из названия, состоит из комбинации простых списков. Следовательно, основное понятие, на основе которого ConTEXt строит оглавление, - это список. Несколько списков объединены в оглавление. По умолчанию ConTEXt содержит предопределенный комбинированный список, называемый «контент», и это то, с чем до сих пор работали исследованные команды: `\placecontent` и `\completecontent`.

7.2.1 Списки в ConTEXt

В ConTEXt *список* - это диапазон пронумерованных элементов, о которых нам нужно помнить три вещи:

1. Число.
2. Имя или заголовок.
3. Страница, на которой он находится.

Это происходит с пронумерованными разделами; но также и с другими элементами документа, такими как изображения, таблицы и т. д. Как правило, это те элементы, для которых есть команда, имя которой начинается с `\place` , которая помещает их как `\placetable`, `\placefigure` и т. д.

Во всех этих случаях ConTEXt автоматически генерирует список, в котором указано время появления рассматриваемого элемента, его номер, заголовок и страница. Так, например, есть список глав, называемый `chapter`, другой из разделов, называемый `section`; но также и другую из таблиц (называемую `table`) или изображений (называемую `figure`). Списки, автоматически создаваемые ConTEXt, всегда называются так же, как и элементы, которые они хранят.

Список также будет автоматически сгенерирован, если мы создадим, например, новый тип пронумерованного раздела: когда мы создадим его, мы неявно создадим также список, в котором они хранятся. И если для ненумерованного раздела по умолчанию мы установим параметр

incrementnumber=yes, сделав его пронумерованным разделом, мы также неявно создадим список с этим именем.

Вместе с неявными списками (автоматически определяемыми ConTEXt) мы можем создавать наши собственные списки с помощью \definelist, синтаксис которого

```
\definelist[ListName][Configuration]
```

в список добавляются:

- В списках, предопределенных ConTEXtили созданных им в результате создания нового плавающего объекта (см. section 13.5), автоматически каждый раз, когда элемент из списка вставляется в документ либо с помощью команды секционирования, либо с помощью \placeWhatever. Команда для других типов списков, например: \placefigure, вставит любое изображение в документ, но также вставит соответствующую запись в список.
- Вручную в любом виде списка с помощью \Writetolist [ListName], уже объясненного в subsction Б of section 7.1.2 и 7.1.3).

  Так, например:

  ```
  \placelist[section]
  ```

  вставит список разделов, включая гиперссылки на них, если интерактивность документа включена и если в \setuplist мы не установили interaction=no. Список разделов не совсем то же самое, что оглавление, основанное на разделах: идея оглавления обычно включает также и более низкие уровни (подразделы, подподразделы и т.д.). Но список разделов будет включать только сами разделы.

  Синтаксис этих команд:

  ```
  \placelist[ListName][Options]
  ```

  ```
  \setuplist[ListName][Configuration]
  ```

  Параметры \setuplist уже были объяснены в section 7.1.6, а параметры для \placelist такие же, как для \\placecontent (см.section 7.1.3).

7.2.2 Списки или указатели изображений, таблиц и других элементов

Из того, что было сказано до сих пор, можно видеть, что, поскольку ConTEXt автоматически создает список изображений, помещенных в документ с помощью команды \placefigure, создание списка или индекса изображений в определенной точке нашего документа так же просто, как при использовании команды \placelist[figure]. И если мы хотим сгенерировать список с заголовком (аналогично тому, что мы получаем с помощью \completecontent), мы можем сделать это с помощью \\completetelist [figure]. Мы можем сделать то же самое с другими четырьмя предопределенными типами плавающих объектов в ConTEXt: таблицы («table»), графика («graphic»), *intermezzos* ((«intermezzo») и химические формулы («chemical»), хотя для конкретных случаев из них ConTEXtуже включает команду, которая генерирует их без заголовка: (\placelistoffigures, \placelistoftables, \placelistofgraphics, \placelistofintermezzi и \placelistofchemicals), и еще одну, которая генерирует их с заголовком: (\completelistoffigures, \completelistoftables, \completelistofgraphics, \completelistofintermezzi and \completelistofchemicals), аналогично \completecontent

Таким же образом для плавающих объектов, которые мы сами создали (см. section 13.5), будут автоматически созданы \placelistof<FloatName> и \completelistof<FloatName>.

Для списков, которые мы создали с помощью `\definelist`, мы можем создать индекс с помощью `\placelist[ListName]` или с помощью`\completelist[ListName]`.

7.2.3 Комбинированные списки

Комбинированный список, как следует из названия, представляет собой список, который объединяет элементы из различных ранее определенных списков. По умолчанию ConTEXt определяет комбинированный список для таблиц содержимого с именем «`content`», но мы можем создавать другие комбинированные списки с помощью `\definecombinedlist`, синтаксис которого:

```
\definecombinedlist[Name][Lists][Options]
```

где

- *Name*: это имя, которое будет иметь новый объединенный список.
- *Lists*: относится к именам списков, которые нужно объединить, через запятую.
- *Options*: параметры конфигурации для списка. Они могут быть указаны во время определения списка или, что предпочтительно, при вызове списка. Основные варианты (которые уже были объяснены) - это `criterium` (subsection 7.1.4 of section 7.1.3) and `alternative` (в subsection 7.1.5 in the same section).

Побочный эффект создания объединенного списка с помощью `\definecombinedlist` заключается в том, что он также создает команду с именем`\placeListName`, которая служит для вызова списка, то есть для включения его в выходной файл. Так, например,

```
definecombinedlist[TOC]
```

создаст команду `\placeTOC`; а

```
definecombinedlist[content]
```

создаст команду `\placecontent`

Но подождите, `\placecontent`! Разве это не та команда, которая используется для создания *обычного* оглавления? В самом деле: это означает, что стандартное оглавление фактически создается ConTEXt с помощью следующей команды:

```
\definecombinedlist
  [content]
  [part, chapter, section, subsection,
    subsubsection, subsubsubsection,
    subsubsubsubsection]
```

Как только наш объединенный список определен, мы можем настроить его (или перенастроить) с помощью `\setupcombinedlist`, который позволяет использовать уже объясненный параметр `criterium` (см. insubsection[sec:criteriumlist]) и параметр `alternative` (см. subsection 7.1.5 в том же разделе).), а также опция `list` для *изменения* списков, включенных в объединенный список.

> Официальный список команд ConTEXt (см. section ??) не упоминает параметр `list` среди параметров, разрешенных для `\setupcombinedlist`, но он используется в нескольких примерах использования этой команды в вики (которая, к тому же, не упоминает об этом на странице, посвященной этой команде). Я также проверил, работает ли опция.

7.3 Индекс (указатель)

7.3.1 Создание индекса

Предметный указатель состоит из списка важных терминов, обычно расположенных в конце документа, с указанием страниц, на которых можно найти такую тему.

Когда книги набирались вручную, создание предметного указателя было сложной и утомительной задачей. Любое изменение нумерации страниц могло повлиять на все записи в указателе. Поэтому они были не очень распространены. Сегодня компьютерные механизмы набора текста означают, что, хотя задача, вероятно, и дальше будет утомительной, она уже не такая сложная, учитывая, что для компьютерной системы не так сложно поддерживать актуальный список данных, связанных с записями в указателе.

Для создания предметного указателя нам понадобятся:

1. Определите, какие слова, термины или понятия должны быть его частью. Это задача, которую может выполнить только автор.

2. Проверьте, в каких точках документа появляется каждая запись в будущем указателе. Хотя, если быть точным, больше, чем *проверка* мест в исходном файле, где обсуждается концепция или проблема, то, что мы делаем, когда мы работаем с ConTEXt - это разметка этих места, путем вставки команды, которая затем будет служить для автоматического создания индекса. Это утомительная часть.

3. Наконец, мы генерируем и форматируем указатель, помещая его в нужное место в документе. Последнее довольно просто с ConTEXt и требует только одной команды: `\placeindex`.

A. Предварительное определение записей в указателе и маркировка точек в исходном файле, которые ссылаются на них

Фундаментальная работа приходится на второй этап. Верно, что компьютерные системы также способствуют этому в том смысле, что мы можем выполнить глобальный текстовый поиск, чтобы найти места в исходном файле, где рассматривается конкретная тема. Но мы также не должны слепо полагаться на такой текстовый поиск: хороший предметный указатель должен уметь обнаруживать каждое место, где обсуждается конкретный предмет, даже если это делается без использования *standard* термина для его обозначения.

Чтобы отметить *mark* фактическую точку в исходном файле, связав ее со словом, термином или идеей, которые появятся в индексе, мы используем команду `\index`, синтаксис которой следующий:

```
\index[Alphabetical]{Index entry}
```

где *Alphabetical* - необязательный аргумент, который используется для указания текста, альтернативного тексту самой записи указателя, чтобы отсортировать её по алфавиту, а запись указателя *Index entry* – это текст, который появится в указателе, связанный с этой меткой. Мы также можем применить функции форматирования, которые мы хотим использовать, и если в тексте появляются зарезервированные символы, они должны быть записаны обычным способом в ConTEXt.

> Возможность расположить элемент указателя в алфавитном порядке иначе, чем он написан на самом деле, очень полезна. Подумайте, например, об этом документе, если я хочу создать запись в указателе для всех ссылок на команду `\\TeX`. Например, последовательность `\index{\backslash TeX}` перечислит команду не через "t" в "TeX", а среди символов, поскольку термин, отправленный в индекс, начинается с обратной косой черты. Это делается записью `\index[tex]{\backslash TeX}`.

Записи указателя *index entries* будут теми, которые нам нужны. Чтобы предметный указатель был действительно полезным, мы должны немного усерднее поработать, задавая вопрос, какие концепции читатель документа, скорее всего, будет искать; так, например, может быть лучше определить статью как «болезнь Ходжкина», чем определять ее как «болезнь Ходжкина», поскольку более всеобъемлющим термином является «болезнь».

> По соглашению записи в предметном указателе всегда пишутся строчными буквами, если они не являются собственными именами.

Если индекс имеет несколько уровней глубины (допускается до трех), чтобы связать конкретную запись индекса с определенным уровнем, используется символ «+». Следующее:

```
\index{Entry 1+Entry 2}
\index{Entry 1+Entry 2+Entry 3}
```

В первом случае мы определили запись второго уровня под названием *Entry 2* , которая будет подстатьей *Entry 1*. Во втором случае мы определили запись третьего уровня под названием *Entry 3*, которая будет подзаписью *Entry 2*, которая, в свою очередь, подзапись *Entry 1*. Например

```
My \index{dog}dog, is a \index{dog+greyhound}greyhound called Rocket.
He does not like \index{cat+stray}stray cats.
```

Стоит отметить некоторые детали из вышеперечисленного:

- Команда `\index` обычно помещается *перед* словом, с которым она связана, и обычно не отделяется от него пробелом. Это необходимо для того, чтобы команда находилась на той же странице, что и слово, с которым она связана:
  - Если бы между ними был пробел, могла бы быть вероятность, что ConTEXt выберет именно это пространство для разрыва строки, который также может оказаться разрывом страницы, и в этом случае команда будет на одной странице, а слово это связано с находящемся на следующей странице.
  - Если бы команда следовала после *после* слова, это слово могло бы быть разбито по слогам и разрыв строки между двумя его слогами был бы также разрывом страницы, и в этом случае команда указывала бы на следующую страницу, начинающуюся со слова, на которое она указывает.
- Посмотрите, как термины второго уровня вводятся во втором и третьем появлениях команды.
- Также проверьте, как при третьем использовании команды `\index`, хотя слово, которое появляется в тексте, является «кошки», термин, который будет отправлен в указатель, будет «кошка».
- Наконец: посмотрите, как три записи предметного указателя были написаны всего в две строки. Я сказал ранее, что отмечать точные места в исходном файле утомительно. Теперь я добавлю, что слишком большое их количество контрпродуктивно. Слишком обширный указатель ни в коем случае не предпочтительнее более краткого, в котором вся информация актуальна. Вот почему я сказал ранее, что решение о том, какие слова будут входить в указатель, должно быть результатом сознательного решения автора.

Если мы хотим, чтобы наш индекс был действительно полезным, термины, используемые в качестве синонимов, должны быть сгруппированы в индексе под одним заголовком. Но поскольку читатель может искать информацию в индексе по любому из других терминов заголовка, обычно индекс содержит записи, которые ссылаются на другие записи. Например, предметный указатель руководства по гражданскому праву с таким же успехом может быть чем-то вроде

недействительность договора
    видит *недействительность*.

Мы достигаем этого не с помощью команды `\index`, а с помощью `\seeindex`, формат которой:

```
\seeindex[Alphabetical] {Entry1} {Entry2}
```

где *Entry1* - это запись индекса, которая будет ссылаться на другую; и *Entry2* - это реферетная цель. В нашем предыдущем примере нам нужно было бы написать:

```
\seeindex{недействительность договора\}{недействительность}
```

В `\seeindex` мы также можем использовать знак „+” для обозначения подуровней для любого из двух его аргументов в квадратных скобках.

Б. Создание окончательного индекса

Как только мы отметили все записи для индекса в нашем исходном файле, фактическая генерация индекса выполняется с помощью команд
PlaceMacroplaceindex`\placeindex` или `\completindex` commands. Эти две команды сканируют исходный файл на наличие команд `\index` и генерируют список всех записей, которые должны быть в индексе, связывая термин с номером страницы, соответствующим месту, где он нашел команду `\index`. Затем они упорядочивают список терминов в алфавитном порядке, которые появляются в индексе, и объединяют случаи, когда один и тот же термин появляется более одного раза, и, наконец, они вставляют правильно отформатированный результат в итоговый документ.

Разница между `\placeindex` и `\completeindex` аналогична разнице между `\content` и `\completecontent` (см. section 7.1.2): `\placeindex`) ограничивается созданием индекса и его вставкой, в то время как `\completeindex` предварительно вставляет новую главу в итоговый документ, по умолчанию называемый «Index», внутри которого индекс будет набран.

7.3.2 Форматирование предметного указателя

Предметные указатели являются частным применением более общей структуры ConTEXt вызывает «*register*»; поэтому индекс форматируется с помощью команды:

```
\setupregister[index][Configuration]
```

С помощью этой команды мы можем:

- Определить, как будет выглядеть индекс с его различными элементами. А именно:
  - Заголовки указателей, которые обычно представляют собой буквы алфавита. По умолчанию это строчные буквы. С помощью `alternative=A` мы можем установить их в верхнем регистре.
  - Сами записи и номер их страницы. Внешний вид зависит от параметров `textstyle`, `textcolor`, `textcommand` и `deeptextcommand` для фактической записи, а также от `pagestyle`, `pagecolor` и `pagecommand` для номера страницы. С `pagenumber=no` мы также можем создать предметный указатель без номеров страниц (хотя я не знаю, может ли это быть полезно).
  - Опция `distance` измеряет ширину разделения между названием записи и номерами страниц; но он также измеряет размер отступа для подстатьи.

  Я думаю, что имена опций `style`, `textstyle`, `pagestyle`, `color`, `textcolor`, and `pagecolor` достаточно ясны, и говорят сами за себя. Что касается `command`, `pagecommand`, `textcommand` and `deeptextcommand`, я обращаюсь к объяснению одноименных параметров в section 6.4.4, касающегося конфигурации команд раздела.
- Чтобы установить общий вид индекса, который включает, среди прочего, команды для выполнения до (`before`) или после (`after`) индекса, количество столбцов, которое он должен иметь (`n`), должны ли столбцы быть равными или нет (`balance`), выравнивание записей (`align`) и т. д.

7.3.3 Создание других индексов

Я объяснил предметный указатель, как если бы в документе был возможен только один такой указатель; но правда в том, что документы могут иметь сколько угодно индексов. Например, может быть указатель личных имен, в котором собраны имена людей, упомянутых в документе, с указанием места, где они цитируются. Это все еще своего рода указатель. В юридическом тексте мы могли бы также создать специальный указатель для упоминаний Гражданского Кодекса; или, в таком документе, как настоящий, указатель макросов, объясненных в нем, и т. д.

Чтобы создать дополнительный индекс в нашем документе, мы используем команду `\defineregister`, синтаксис которой:

```
\defineregister[IndexName] [Configuration]
```

где *IndexName* – это имя, которое будет иметь новый индекс, а *Configuration* контролирует его работу. Также возможно настроить индекс позже с помощью

```
\setupregister[IndexName] [Configuration]
```

После создания нового названного индекса IndexName в нашем распоряжении будет команда *IndexName*, чтобы пометить записи, которые будет иметь этот индекс, аналогично тому, как записи помечаются с помощью `\index`. Команда seeIndexName также позволяет нам создавать записи, которые ссылаются на другие записи.

Например: мы могли бы создать указатель команд ConTEXt в этом документе с помощью команды:

```
\defineregister[macro]
```

это создаст команду `\macro`. Это позволяет мне пометить все ссылки на команды ConTEXt как элемент указателя, а затем сгенерировать указатель с помощью `\placemacro` или `\completemacro`.

> Создание нового индекса позволяет команде `\IndexName` отмечать его записи, а команды `\placeIndexName` и `\completeIndexName` – для создания индекса. Но эти последние две команды на самом деле являются сокращениями двух более общих команд, применяемых к рассматриваемому индексу. Таким образом, `\placeIndexName` эквивалентен `\placeregister[IndexName]`, а `\completeIndexName` эквивалентен `\completeregister [IndexName]`.

Глава 8

Литературные ссылки и гиперлинки

8.1 Типы ссылок

Научно-техническая документация изобилует ссылкаminces:

– Иногда они ссылаются на другие документы, которые являются основанием для того, что говорится, или которые противоречат тому, что объясняется, или которые развивают или усугубляют нюансы рассматриваемой идеи и т. Д. В этих случаях ссылка считается *внешней* и , если документ должен быть академически строгим, ссылка должна иметь форму *цитат* из литературы.

– Но также обычно в документе в одном из его разделов ссылаются на другой из его разделов, и в этом случае ссылка считается *внутренней*. Также существует внутренняя ссылка, когда точка в документе комментирует какой-либо аспект определенного изображения, таблицы, заметки или элемента аналогичного характера, ссылаясь на него по его номеру или по странице, на которой он находится.

  В целях точности внутренние ссылки должны быть нацелены на точное и легко определяемое место в документе. Следовательно, такого рода ссылки всегда являются ссылкой либо на пронумерованные элементы (как, например, когда мы говорим «см. Таблицу 3.2» или «Главу 7»), либо на номера страниц. Расплывчатые ссылки типа «как мы уже сказали» или «как мы увидим дальше» не являются истинными ссылками, и нет никаких специальных требований для их набора, и для этого нет какого-либо специального инструмента. Кроме того, я лично отговариваю своих аспирантов или магистрантов от привычного использования этой практики.

  > Внутренние ссылки также обычно называют «перекрестными ссылками» (cross references), хотя в этом документе я буду просто использовать термин «ссылки» (references) в целом и «внутренние ссылки» (internal references), когда я хочу быть конкретным.

Чтобы прояснить терминологию, которую я использую для ссылок, я назову точку в документе, где вводится ссылка, источником (*origin*), а место, на которое она указывает, - целью *target*. С этой точки зрения мы бы сказали, что ссылка является внутренней, когда источник и цель находятся в одном документе, и внешней, когда источник и цель находятся в разных документах. С точки зрения верстки документа

– Внешние ссылки не представляют особой проблемы и, следовательно, в принципе не требуют каких-либо инструментов для их введения: все данные, которые мне нужны из целевого

документа, доступны мне, и я могу использовать их в ссылке. Однако, если исходный документ является электронным документом, а целевой документ также доступен в Интернете, то в ссылку можно включить гиперссылку, которая позволяет перейти непосредственно к цели. В этих случаях документ происхождения можно назвать интерактивным *interactive*.

– Напротив, внутренние ссылки создают проблему для верстки документа, поскольку любой, кто имеет опыт подготовки научно-технических документов средней длины, знает, что нумерация страниц, разделов, изображений, таблиц, теорем или аналогичных тому, что указано в ссылке, почти неизбежно изменится во время подготовки документа, что очень затрудняет его обновление.

> В докомпьютерные времена авторы избегали внутренних ссылок; а те, которые были неизбежны, такие как оглавление (которое, если оно сопровождается номером страницы каждого раздела, является примером внутренней ссылки), были написаны в конце.

Даже самые ограниченные системы набора текста, такие как текстовые процессоры, позволяют включать какие-либо внутренние перекрестные ссылки, такие как оглавления. Но это ничто по сравнению с всеобъемлющим механизмом управления ссылками, включенным в ConTEXt, который также может сочетать внутренний механизм управления ссылками, направленный на поддержание актуальности ссылок, с использованием гиперссылок, что, очевидно, не является исключительным для внешних ссылок.

8.2 Внутренние ссылки

Для создания внутренней ссылки необходимы две вещи:

1. Метка или идентификатор в целевой точке. Во время компиляции ConTEXtбудет связывать определенные данные с этой меткой. Какие данные будут связаны, зависит от типа метки; это может быть номер раздела, номер заметки, номер изображения, номер, связанный с конкретным элементом в нумерованном списке, заголовок раздела и т.д.

2. Команда в исходной точке, которая считывает данные, связанные с меткой, связанной с целевой точкой, и вставляет их в исходную точку. Команда различается в зависимости от того, какие данные из метки мы хотим вставить в исходную точку.

Когда мы думаем о ссылке, мы делаем это в терминах «origin ⟶ target» («источник ⟶ цель»), поэтому может показаться, что сначала следует объяснить вопросы, связанные с источником, а затем - с целью. Однако я считаю, что легче понять логику ссылок, если объяснение будет обратным.

8.2.1 Метка в референтной цели

В этой главе под меткой (*label*) я подразумеваю текстовую строку, которая будет связана с целевой точкой ссылки и использоваться внутри для извлечения определенной информации о целевой точке ссылки, такой как, например, номер страницы, номер раздела и т. Д. Фактически, информация, связанная с каждой меткой, зависит от процедуры ее создания. ConTEXt называет эти ярлыки ссылками *references*, но я думаю, что этот последний термин, поскольку он имеет гораздо более широкое значение, менее ясен.

Метка, связанная с целевой ссылкой:

– Необходимо, чтобы каждая потенциальная цель в документе была уникальной, чтобы её можно было без сомнения идентифицировать. Если мы используем одну и ту же метку для разных целей, ConTEXt не вызовет ошибку компиляции, но приведет к тому, что все ссылки будут указывать на первую метку, которую он найдет (в исходном файле), и это будет иметь побочный эффект, заключающийся в том, что некоторые из наших ссылок могут быть неправильными

и, что еще хуже, мы их не заметим. Поэтому при создании метки важно убедиться, что новая метка, которую мы назначаем, еще не была назначена ранее.

– может содержать буквы, цифры, знаки препинания, пробелы и т. Д. Там, где есть пробелы, общие правила ConTEXtв отношении таких символов по-прежнему применяются (см. section 4.2.1), так что, например, quotation `My nice label` и «`My  nice  label`» будут восприниматься как одно и то же, хотя в обоих используется разное количество пробелов.

Поскольку нет ограничений относительно того, какие символы могут быть частью метки и сколько их, я советую использовать понятные имена меток, которые помогут нам понять исходный файл, когда, возможно, мы прочитаем его спустя долгое время после того, как он был первоначально написан. Вот почему пример, который я привел ранее («`My nice label`»), не является хорошим примером, поскольку он ничего не говорит нам о цели, на которую указывает метка. Например, для этого заголовка лучше использовать ярлык „sec:Target labels”.

Чтобы связать конкретную цель с меткой, в основном есть две процедуры:

1. С помощью аргумента или параметра команды, используемого для создания элемента, на который будет указывать метка. С этой точки зрения, все команды, которые создают какую-либо структуру или текстовый элемент, открытый для использования в качестве цели ссылки, включают параметр, называемый «ссылка» «`reference`», который используется для включения метки. Иногда вместо *опции* метка является содержанием всего аргумента.

   Мы находим хороший пример того, что я пытаюсь сказать, в командах раздела, которые, как мы знаем из (section 6.3), допускают несколько видов синтаксиса. В классическом синтаксисе команда записывается как:

   ```
   \section[Label]{Title}
   ```

   а в синтаксисе, специфичном для ConTEXt, команда записывается как

   ```
   \startsection
    [title=Title, reference=Label, ...   ]
   ```

   В обоих случаях команда предусматривает введение метки, которая будет связана с рассматриваемым разделом (или главой, подразделом и т.д.).

   Я сказал, что эта возможность присутствует во *всех командах*, которые позволяют нам создавать текстовый элемент, открытый для использования в качестве цели ссылки. Это все текстовые элементы, которые можно пронумеровать, в том числе, среди прочего, разделы, плавающие объекты всех видов (таблицы, изображения и т.д.), Сноски или конечные примечания, цитаты, нумерованные списки, описания, определения и т.д.

   Когда метка вводится напрямую с аргументом, а не как опция, которой присваивается значение, с помощью ConTEXt можно связать несколько меток с одной целью. Например:

   ```
   \chapter[label1, label2, label3] {My chapter}
   ```

   Мне не ясно, в чем преимущество наличия нескольких разных меток для одной цели и подозреваю , что это можно сделать не потому, что это дает преимущества, а из-за некоторых *внутренних* требований ConTEXt применимых к определенные виды аргументов.

2. С помощью команд PlaceMacropagereference`\pagereference`, `\reference`, или `\textreference` синтаксис которых

   ```
   \pagereference[Label]
   \reference[Label]{Text}
   \textreference[Label]{Text}
   ```

   * Ярлык, созданный с помощью `\pagereference`, позволяет нам получить номер страницы.

   * Ярлыки, созданные с помощью `\reference` и `\textreference`, позволяют нам получить номер страницы, а также связанный с ними текст, который включен в качестве аргумента.

Как в \reference, так и в \textreference текст, связанный с меткой, исчезает как таковой из окончательного документа в точке, где расположена команда (цель ссылки), но может быть извлечен и снова появится в точке происхождения ссылки.

Я сказал ранее, что каждая метка связана с определенной информацией о целевой точке. Что это за информация, зависит от типа этикетки:

- Все метки *запоминают* (в том смысле, что они позволяют извлекать) номер страницы команды, которая их создала. Для ярлыков, прикрепленных к разделам, которые могут состоять из нескольких страниц, этот номер будет номером страницы, с которой начинается рассматриваемый раздел.
- Ярлыки, вставленные с помощью команды, которая создает пронумерованный текстовый элемент (раздел, примечание, таблица, изображение и т.д.), *запоминают* номер, связанный с этим элементом (номер раздела, номер заметки и т.д.)
- Если у этого элемента есть заголовок *title*, как, например, для разделов, но также и для таблиц, если они были вставлены с помощью команды \placetable, они запомнят этот заголовок.
- Ярлыки, созданные с помощью \pagereference, *запоминают* только номер страницы.
- Созданные с помощью \reference или \textreference, также запоминают связанный с ними текст, который эти команды принимают в качестве аргумента.

На самом деле я не уверен в реальной разнице между командами \reference и \textreference. Я думаю, вполне возможно , что дизайн трех команд, которые позволяют создавать метки, пытается работать параллельно с тремя командами, которые позволяют извлекать информацию из меток (что мы увидим чуть позже); но правда в том, что согласно моим тестам, \reference и \textreference кажутся избыточными командами

8.2.2 Команды в исходной точке отсчета для извлечения данных из целевой точки

Команды, которые я объясню далее, извлекают информацию из меток и, кроме того, если наш документ интерактивный, генерируют соединение с эталонной целью. Но важная вещь в этих командах – это информация, полученная с метки. Если мы хотим только сгенерировать соединение, не извлекая никакой информации из метки, мы должны использовать команду \goto, описанную в разделе section 8.4.2.

A. Основные команды для получения информации с метки

Принимая во внимание, что каждая метка, связанная с целевой точкой, может хранить разные элементы информации, логично, что ConTEXt включает три разные команды для получения такой информации: в зависимости от того, какую информацию из опорной целевой точки мы хотим получить, мы используем одну или другую из этих команд:

- Команда \at позволяет нам получить номер страницы ярлыка.
- Для этикеток, которые запоминают номер элемента (номер раздела, номер заметки, номер элемента, номер таблицы и т. д.) В дополнение к номеру страницы, команда \in позволяет нам получить этот номер.
- Наконец, для меток, которые запоминают текст, связанный с меткой (заголовок раздела, заголовок изображения, вставленный с помощью \placefigure, и т.д.), команда \about позволяет нам получить этот текст.

Три команды \at \in \about имеют одинаковый синтаксис:

```
\at{Text}[Label]
\in{Text}[Label]
\about{Text}[Label]
```

- Label - это метка, с которой мы хотим получить информацию
- Текст - это текст, написанный непосредственно перед информацией, которую мы хотим получить с помощью команды. Между текстом и данными метки, которую извлекает команда, будет вставлен неразъемный пробел, и если функция интерактивности включена таким образом, что команда, помимо получения информации, генерирует ссылку, которая позволяет нам переходить до целевой точки текст, включенный в качестве аргумента, будет частью ссылки (это будет кликабельный текст).

Итак, в следующем примере мы видим, как `\in` извлекает номер раздела, а `\at` – номер страницы.

```
In \in{section}[sec:target labels], that begins on \at{page}
[sec:target labels], the
characteristics of labels used for
internal references are explained.
```

In section 8.2.1, that begins on page 111, the characteristics of labels used for internal references are explained.

Обратите внимание, что ConTEXt автоматически создает гиперссылки (см. Раздел section 8.3), и что текст, принимаемый в качестве аргумента `\in` и `\at`, является частью ссылки. Но если бы мы написали иначе, результат был бы таким:

```
In section \in{}[sec:target
labels], that begins on page \at{}
[sec:target labels], the
characteristics of labels used for
internal references are explained.
```

In section 8.2.1, that begins on page 111, the characteristics of labels used for internal references are explained.

Текст остается прежним, но слова*section* и *page*, предшествующие ссылке, не включаются в ссылку, поскольку они больше не являются частью команды.

Если ConTEXt не может найти метку, на которую указывают команды `\at`, `\in` или `\about`, ошибки компиляции не будет, но там, где информация, полученная этими командами, должна появиться в окончательном документе, мы увидим «??».

ConTEXt не может найти метку по двум причинам:

1. Мы ошиблись при написании.
2. Мы компилируем только часть документа, а метка указывает на еще не скомпилированную часть (см. sections 4.5.1 and 4.6).

В первом случае ошибку нужно будет исправить. Поэтому, когда мы закончим компиляцию всего документа (и второй случай больше невозможен), будет хорошей идеей искать все появления «??» в PDF, чтобы убедиться, что в документе нет *неработающих* ссылок.

Б. Получение информации

Каждый из `\at`, `\in` и `\about` извлекает некоторые элементы метки. Доступна другая команда, которая позволяет нам спасти какой-либо элемент указанной метки. Это команда`\ref`, синтаксис которой:

```
\ref[Element to retrieve][Label]
```

где первым аргументом может быть:

- `text`: возвращает текст, связанный с меткой.
- `title`: возвращает текст, связанный с меткой.
- `number`: возвращает число, связанное с меткой. Например, в разделах номер раздела.
- `page`: возвращает номер страницы.

- `realpage`: возвращает фактический номер страницы.
- `default`: возвращает то, что ConTEXt считает *естественным* элементом метки. Обычно это совпадает с тем, что возвращается `number`.

Фактически, `\ref` намного точнее, чем `\at`, `\in` или `\about`, и поэтому, например, он различает номер страницы и фактический номер страницы. Номер страницы может не совпадать с фактическим номером, если, например, нумерация страниц документа начинается с 1500 (потому что этот документ является продолжением предыдущего) или если страницы преамбулы были пронумерованы римскими цифрами и при этом нумерация была перезапущена. Точно так же `\ref` различает *текст* и *заголовок*, связанный со ссылкой, чего, например, не делает `\about`.

Если `\ref` используется для получения информации от метки, в которой такая информация отсутствует (например, заголовок метки, связанный со сноской), команда вернет пустую строку.

B. Определение того

В ConTEXt также есть две команды, которые зависят от адреса ссылки *the link address*. С помощью «link address» я намерен определить, находится ли цель ссылки в исходном файле до или после источника. Например: мы пишем наш документ и хотим сослаться на раздел, который все еще может быть до или после того, который мы пишем в окончательном оглавлении. Просто мы еще не решили. В этой ситуации было бы полезно иметь команду, которая записывает одно или другое в зависимости от того, находится ли цель в конечном итоге до или после источника в окончательном документе. Для подобных нужд ConTEXt предоставляет команду PlaceMacrosomewhere`\somewhere`, синтаксис которой:

```
\somewhere{Text if before}{Text if after}[Label].
```

Например, в следующем тексте:

```
\color[red]{Адрес гиперссылки также можно определить с помощью команды \type{\somewhere}.
Таким образом, мы также можем найти главы или другие текстовые элементы \somewhere {before}{after} [sec:references]
и обсудить их описания в другом месте \somewhere{before}{after} [sec:interactivity].}
```

> Для этого примера я использовал две фактические метки в этой главе в исходном файле.

Другая команда, способная определить, идет ли метка, на которую она указывает, до или после, - это `\atpage`, синтаксис которой:

```
\atpage[label]
```

Эта команда очень похожа на предыдущую, но вместо того, чтобы позволить нам писать текст самостоятельно, в зависимости от того, идет ли метка до или после, `\atpage` вставляет текст по умолчанию для каждого из двух случаев и, если документ интерактивный , также вставляет гиперссылку.

Текст, который вставляет `\atpage`, связан с метками «предыдущая страница» «`precedingpage`» в случае, если метка *label*, которую он принимает в качестве аргумента, находится перед командой, и «здесь после» «`hereafter`» в противоположном случае.

> Когда я подошел к этому моменту, меня предало предыдущее решение: в этой главе я решил назвать то, что ConTEXt называет «ссылкой» «reference», «меткой» «label». Мне это показалось яснее. Но определенные фрагменты текста, генерируемые командами ConTEXt такие как `\atpage`, также называются «labels» «метками» (на этот раз в другом смысле). (См. section 10.5.3). Надеюсь, читатель не запутается. Я думаю, что контекст позволяет нам правильно различать, какое из различных значений ярлыка *label* я имею в виду в каждом конкретном случае.

Следовательно, мы можем изменить текст, вставленный с помощью `\atpage`, так же, как мы меняем текст любой другой метки:

```
\setuplabeltext[en][precedingpage=New text ]
\setuplabeltext[en][hereafter=New text ]
```

По этому поводу я считаю, что есть небольшая ошибка в «ConTEXt Standalone» (дистрибутив, который я использую). Изучая имена предопределенных меток в ConTEXt которые можно изменить с помощью `\setuplabeltext`, мы обнаруживаем две пары меток, которые являются кандидатами на использование `\atpage`:

- «`precedingpage`» and «`followingpage`» («предыдущая страница» и «следующая страница»).
- «`hencefore`» and «`hereafter`» («отсюда» и «далее»).

Можно предположить, что `\atpage` будет использовать либо первую, либо вторую пару. Но на самом деле для предшествующих элементов используется «предыдущая страница» «`precedingpage`», а для следующих - «в дальнейшем» «`hereafter`», что, на мой взгляд, непоследовательно.

8.2.3 Автоматическая генерация префиксов

В большом документе не всегда легко избежать дублирования надписей. Поэтому желательно навести некоторый порядок в том, как мы выбираем, какие этикетки использовать. Одна из практик, которая помогает – это использование префиксов для меток, которые будут варьироваться в зависимости от типа метки. Например, «sec:» для разделов, «fig:» для рисунков, „tbl:” для таблиц и т.д.

Имея это в виду, ConTEXt включает набор инструментов, которые позволяют:

- ConTEXt сам автоматически генерирует метки для всех допустимых элементов.
- Каждая метка, сгенерированная вручную для получения префикса, который мы либо заранее определили, либо автоматически сгенерированы ConTEXt.

Подробное объяснение этого механизма является длинным и, хотя они, несомненно, являются полезными инструментами, я не думаю, что они необходимы. Поэтому, поскольку они не могут быть объяснены в нескольких словах, я предпочитаю не объяснять их и ссылаться на то, что сказано о них в справочном руководстве ConTEXt или в wiki по этому вопросу.

8.3 Интерактивные электронные документы

Интерактивными могут быть только электронные документы, но не все электронные документы являются интерактивными. *Электронный* документ - это документ, который хранится в компьютерном файле и может быть открыт и прочитан непосредственно на экране. С другой стороны, электронный документ, оснащенный утилитами, которые позволяют пользователю взаимодействовать с ним, является интерактивным; то есть мы можем сделать больше, чем просто прочитать его. Существует интерактивность, например, когда в документе есть кнопки, которые выполняют какое-либо действие, или ссылки, с помощью которых мы можем перейти в другую точку документа или во внешний документ; или когда в документе есть области, где пользователь может писать, или есть видео или аудиоклипы, которые можно воспроизводить, и т.д.

Все документы, созданные ConTEXt, являются электронными (поскольку ConTEXt генерирует PDF-файл, который по определению является электронным документом), но они не всегда интерактивны. Чтобы обеспечить им интерактивность, необходимо прямо указать это, как показано в следующем разделе.

Имейте в виду, однако, что, хотя ConTEXt генерирует интерактивный PDF-файл, для того, чтобы оценить эту интерактивность, нам нужна программа чтения PDF, способная на это, поскольку не все программы чтения PDF позволяют нам использовать гиперссылки, кнопки и аналогичные элементы, соответствующие интерактивным документам.

8.3.1 Включение интерактивности в документах

ConTEXt не использует интерактивные функции по умолчанию, если прямо не указано, что обычно делается в преамбуле документа. Команда, которая включает эту утилиту, является:

```
\setupinteraction[state=start]
```

Обычно эта команда используется только один раз и в преамбуле документа, когда мы хотим создать интерактивный документ. Но на самом деле мы можем использовать его так часто, как захотим, изменяя состояние интерактивности документа. Команда «`state = start`» включает интерактивность, а «`state = stop`» отключает ее, поэтому мы можем отключить интерактивность в некоторых главах или *частях* нашего документа, где мы хотим это сделать.

> Я не могу придумать ни одной причины, по которой мы хотели бы иметь в документах неинтерактивные части, которые являются интерактивными. Но что важно в философии ConTEXt, так это то, что что-то технически возможно, даже если мы вряд ли будем это использовать, поэтому он предлагает процедуру для этого. Именно эта философия дает ConTEXt столько возможностей и не позволяет сделать такое простое введение *кратким*.

Как только взаимодействие будет установлено:

- Некоторые команды ConTEXt уже будут включать гиперссылки. Таким образом:
  - Команды для создания оглавлений, которые в принципе и если прямо не указано иное, будут интерактивными, т.е. нажатие на запись в оглавлении приведет к переходу на страницу, с которой начинается соответствующий раздел.
  - Команды для внутренних ссылок, которые мы видели в первой части этой главы, где нажатие на них автоматически переходит к цели ссылки.
  - Сноски и конечные примечания, где щелчок по привязке примечания в основной части текста приведет нас на страницу, где написана сама заметка, а щелчок по метке примечания в тексте заметки приведет нас к точке в основном тексте, где был сделан вызов.
  - и т.д.
- Включена возможность использования других команд, специально предназначенных для интерактивных документов, таких как презентации. В них используются многочисленные инструменты, связанные с интерактивностью, такие как кнопки, меню, наложения изображений, встроенный звук или видео и т.д. Объяснение всего этого было бы слишком длинным, и, кроме того, презентации - это довольно особый вид документа. Поэтому в следующих строках я опишу одну функцию, связанную с интерактивностью: гиперссылки.

8.3.2 Базовая конфигурация для интерактивности

`\setupinteraction`, помимо включения или отключения взаимодействия, позволяет нам настраивать некоторые связанные с ним вопросы; в основном, но не только, цвет и стиль ссылок. Это делается с помощью следующих параметров команды:

- `color`: управляет *обычным* цветом ссылок.
- `contrastcolor`: определяет цвет ссылок, цель которых находится на той же странице, что и исходная. Я рекомендую всегда устанавливать этот параметр на то же содержание, что и предыдущий.
- `style`: управляет стилем ссылки.
- `title, subtitle, author, date, keyword`: значения, присвоенные этим параметрам, будут преобразованы в метаданные PDF-файла, созданного ConTEXt.
- `click`: этот параметр определяет, должна ли ссылка выделяться при нажатии.

8.4 Гиперссылки на внешние документы

Я буду различать команды, которые не создают ссылку, но помогают набрать URL-адрес ссылки, и команды, которые создают гиперссылку. Давайте посмотрим на них отдельно

8.4.1 Команды, которые помогают набирать гиперссылки, но не создают их

URL-адреса, как правило, очень длинные и включают символы всех типов, даже символы, которые являются зарезервированными символами в ConTEXt и не могут использоваться напрямую. Кроме того, когда URL-адрес должен отображаться в документе, очень сложно набрать абзац, поскольку URL-адрес может превышать длину строки и никогда не включать пробелы, которые можно использовать для вставки разрыва строки. Более того, в URL-адресе нецелесообразно переносить слова для вставки разрывов строк, так как читатель вряд ли может знать, действительно ли расстановка переносов является частью URL-адреса.

Поэтому ConTEXt предоставляет две утилиты для *набора* URL-адресов. Первый предназначен в первую очередь для URL-адресов, которые будут использоваться внутри компании, но фактически не будут отображаться в документе. Второй - для URL-адресов, которые должны быть вписаны в текст документа. Давайте посмотрим на них отдельно:

`\useURL`

Эта команда позволяет нам записать URL-адрес в преамбуле документа, связав его с именем, так что, когда мы хотим использовать его в нашем документе, мы можем вызывать его по ассоциированному с ним имени. Это особенно полезно с URL-адресами, которые будут использоваться несколько раз в документе.

Команда допускает два использования:

1. `\useURL[Associated name][URL]`
2. `\useURL[Associated name] [URL] [] [Link text]`

– В первой версии URL просто связан с именем, под которым он будет вызываться в нашем документе. Но затем, чтобы использовать URL-адрес, нам нужно будет каким-то образом указать при его вызове, какой интерактивный текст будет отображаться в документе.

– Во второй версии последний аргумент включает интерактивный текст. Третий аргумент существует на тот случай, если мы хотим разделить URL-адрес на две части, чтобы первая часть содержала адрес доступа, а вторая часть - имя конкретного документа или страницы, которую мы хотим открыть. Например: адрес документа, объясняющего, что такое ConTEXt http://www.pragma-ade.com/general/manuals/what-is-context.pdf. Этот адрес можно записать полностью во второй аргумент, оставив третий пустой:

```
\useURL [WhatIsCTX]
  [http://www.pragma-ade.com/general/manuals/what-is-context.pdf]
  []
  [What is \ConTeXt?]
```

но мы также можем разделить его на два аргумента:

```
\useURL [WhatIsCTX]
  [http://www.pragma-ade.com/general/manuals/]
  [what-is-context.pdf]
  [What is \ConTeXt?]
```

В обоих случаях мы будем связывать этот адрес со словом «`WhatIsCTX`», так что для включения ссылки на этот адрес мы используем команду, которую используем для создания ссылки; вместо самого URL-адреса мы можем просто написать «`WhatIsCTX`».

Если в любом месте текста мы хотим воспроизвести URL-адрес, который мы связали с именем с помощью `\useURL`, мы можем использовать `\url[Associated name]`, который вставляет URL-адрес, связанный с этим именем, в документ. Но эта команда, хотя и записывает URL-адрес, не создает никаких ссылок.

Формат, в котором отображаются URL-адреса, записанные с использованием \url, не является общим, установленным с помощью \setupinteraction, это формат, специально установленный для этой команды с помощью \setupurl, который позволяет нам установить стиль (параметр style) и цвет (параметр цвет).

\hyphenatedurl

Эта команда предназначена для URL-адресов, которые будут записаны в тексте нашего документа, и ConTEXt включает в себя разрывы строк в URL-адресе, если это необходимо, для правильного набора абзаца. Её формат:

```
\hyphenatedurl{URLaddress}
```

Несмотря на название команды \hyphenatedurl, она не переносит имя URL-адреса. Он учитывает, что определенные символы, часто встречающиеся в URL-адресах, являются хорошим поводом для вставки разрыва строки до или после них. Мы можем добавить нужные символы в список символов, для которых разрешен разрыв строки.

Для этого у нас есть три команды:

```
\sethyphenatedurlnormal{Characters}
\sethyphenatedurlbefore{Characters}
\sethyphenatedurlafter{Characters}
```

Эти команды добавляют символы, которые они принимают в качестве аргументов, в список символов, поддерживающих разрывы строк, до и после списка символов, которые поддерживают только разрывы строк, и тех, которые разрешают только обратные разрывы строк, соответственно.

\hyphenatedurl можно использовать всякий раз, когда необходимо написать URL-адрес, который будет отображаться в окончательном документе как есть. Его даже можно использовать в качестве последнего аргумента для \useURL в версии этой команды, где последний аргумент выбирает интерактивный текст, который будет отображаться в окончательном документе. Например:

```
\useURL [WhatIsCTX]
  [http://www.pragma-ade.com/general/manuals/what-is-context.pdf]
  []
  [\hyphenatedurl{http://www.pragma-ade.com/general/manuals/what-is-context.pdf}]
```

В аргументе \hyphenatedurl можно использовать все зарезервированные символы, кроме трех, которые необходимо заменить командами:

- % должен быть заменен на \letterpercent
- # должен быть заменен на \letterhash
- \ должен быть заменен на \letterescape or \letterbackslash.

Каждый раз, когда \hyphenatedurl вставляет разрыв строки, он выполняет команду \hyphenatedurlseparator, которая по умолчанию ничего не делает. Но если мы его переопределим, репрезентативный символ вставится в URL-адрес аналогично тому, как это происходит с обычными словами, где дефис вставляется, чтобы указать, что слово продолжается на следующей строке. Например:

```
\def\hyphenatedurlseparator{\curvearrowright}
```

таким образом отобразит следующий особенно длинный веб-адрес:

https://support.microsoft.com/?scid=http://support.microsoft.com↷
:80/support/kb/articles/Q208/4/27.ASP&NoWebContent=1.

8.4.2 Команды

Чтобы установить ссылки на предопределенные URL-адреса с помощью `\useURL`, мы можем использовать команду `\from`, которая ограничивается установкой ссылки, но не записывает какой-либо интерактивный текст. Текст по умолчанию в `\useURL` будет использоваться в качестве текста ссылки. Его синтаксис:

```
\from[Name]
```

где *Name* – это имя, ранее связанное с URL-адресом с помощью `\useURL`.

Чтобы создавать ссылки и связывать их с интерактивным текстом, который не был ранее определен, у нас есть команда `\goto`, которая используется как для создания внутренних, так и внешних ссылок. Его синтаксис:

```
\goto{Clickable tex}[Target]
```

где *Clickable tex* - это текст, который будет отображаться в окончательном документе, и где щелчок мыши вызовет переход, а *Target* может быть:

- Ярлык из нашего документа. В этом случае `\goto` сгенерирует переход таким же образом, как, например, уже рассмотренные команды `\in` или `\at`. Но, в отличие от этих команд, никакая информация, связанная с меткой, не будет получена.

- Сам URL. В этом случае необходимо прямо указать, что это URL-адрес, написав команду следующим образом:

  ```
  \goto{Clickable text}[url(URL)]
  ```

  где URL, в свою очередь, может быть именем, ранее связанным с URL-адресом с помощью `\useURL`, или самим URL-адресом, и в этом случае при написании URL-адреса мы должны убедиться, что зарезервированные символы ConTEXt написаны правильно в ConTEXt. Запись URL-адреса в соответствии с правилами ConTEXt не повлияет на функциональность ссылки.

8.5 Создание закладок в окончательном PDF-файле

Файлы PDF могут иметь внутренний список закладок, который позволяет читателю видеть содержимое документа в специальном окне программы просмотра PDF и перемещаться по нему, просто щелкая каждый из разделов и подразделов.

По умолчанию ConTEXt не добавляет в выходной PDF-файл список закладок, хотя сделать это так же просто, как включить команду `\placebookmarks`, синтаксис которой:

```
\placebookmarks[List of sections]
```

где *List of sections* - это разделенный запятыми список уровней разделов, которые должны отображаться в списке содержимого.

Имейте в виду следующие наблюдения относительно этой команды:

- По моим тестам `\placebookmarks` не работают, если это в преамбуле документа. Но в теле документа (между `\starttext` и `\stoptext`, или between `\startproduct` и `\stopproduct`) не имеет значения, где вы его разместите: список закладок также будет включать разделы или подразделы, предшествующие команде. Однако я считаю, что наиболее разумным для правильного понимания исходного файла является размещение команды в начале.

- Типы разделов, определенные пользователем (с помощью `\definehead`), не всегда располагаются в нужном месте в списке закладок. Их желательно исключить.

– Если название раздела в каком-либо разделе содержит концевую сноску или сноску, текст сноски считается частью закладки.

– В качестве аргумента вместо списка разделов мы можем просто указать символическое слово «`all`», которое, как указывает его название, будет включать все разделы; однако, согласно моим тестам, это слово, в дополнение к тому, что определенно является разделами, включает тексты, помещенные туда с некоторыми командами без разделения, поэтому результирующий список несколько непредсказуем.

Не все программы просмотра PDF позволяют нам просматривать закладки; и у многих из них эта функция не активирована по умолчанию. Поэтому, чтобы проверить результат этой функции, мы должны убедиться, что наша программа для чтения PDF-файлов поддерживает эту функцию и активирована ли она. Думаю, я помню, что Acrobat, например, по умолчанию не показывает закладки, хотя на его панели инструментов есть кнопка для их отображения.

III
Специфические вопросы

Глава 10

Символы, слова, текст и горизонталь-ное пространство

Основным основным элементом всех текстовых документов является символ: символы сгруппированы в слова, которые, в свою очередь, образуют строки, составляющие абзацы, составляющие страницы.

Текущая глава, начинающаяся с «символа» «*character*», объясняет некоторые утилиты ConTEXt относящиеся к символам, словам и тексту.

10.1 Получение символов, которые обычно не доступны с клавиатуры

В текстовом файле, закодированном как UTF-8 (см. section ??

Почти все западные языки имеют диакритические знаки (за важным исключением английского по большей части), и в целом клавиатура может генерировать диакритические знаки, соответствующие региональным языкам. Таким образом, испанская клавиатура может генерировать все диакритические знаки, необходимые для испанского (в основном акценты и диэрезис), а также некоторые диакритические знаки, используемые в других языках, таких как каталонский (серьезные акценты и cedillas) или французские (cedillas, серьезные и циркумфлексные акценты); но не, например, некоторые, которые используются в португальском языке, такие как тильда на некоторых гласных в таких словах, как «navegação».

TEX был разработан в Соединенных Штатах, где клавиатуры обычно не позволяют нам вводить диакритические знаки; поэтому Дональд Кнут дал ему набор команд, которые позволяют нам получать почти все известные диакритические знаки (по крайней мере, в языках, использующих латинский алфавит). Если мы используем испанскую клавиатуру, не имеет большого смысла использовать эти команды для получения диакритических знаков, которые можно получить непосредственно с клавиатуры. По-прежнему важно знать, что эти команды существуют, и что они из себя представляют, поскольку испанские (или итальянские, или французские ...) клавиатуры не позволяют нам генерировать все возможные диакритические знаки.

Название	Буква	Абревиатура	Команда
Acute accent	ú	\'u	\uacute
Grave accent	ù	\`u	\ugrave
Circumflex accent	û	\^u	\ucircumflex
Dieresis or umlaut	ü	\”u	\udiaeresis, \uumlaut
Tilde	ũ	\~u	\utilde
Macron	ū	\=u	\umacron
Breve	ŭ	\u u	\ubreve

Таблица 10.1 Акценты и другие диакритические знаки

В table 10.1 мы находим команды и сокращения, которые позволяют нам получать эти диакритические знаки. Во всех случаях неважно, используем ли мы команду или сокращение. В таблице я использовал букву „u” в качестве примера, но эти команды работают с любой гласной (большинство из них [1]), а также с некоторыми согласными и некоторыми полугласными звуками.

– Поскольку большинство сокращенных команд представляют собой *управляющие символы* (см. section 3.2), буква, на которой должен стоять диакритический знак, может быть написана сразу после команды или отделена от нее. Так, например: чтобы получить португальский „ã”, мы можем написать символы \=a или \=␣a. [2] Но в случае breve (\u), когда мы имеем дело с *контрольным словом*, пробел является обязательным.

– В случае длинной версии команды буква с диакритическим знаком будет первой буквой имени команды. Так, например, \emacron поместит макрон над строчной буквой „e” (ē), \Emacron сделает то же самое над прописной буквой „E” (Ē), а texAmacron сделает то же самое над прописной буквой „A” (Ā).

Хотя команды в таблице table 10.1 работают с гласными и некоторыми согласными, существуют и другие команды для создания некоторых диакритических знаков и специальных букв, которые работают только с одной или несколькими буквами. Они представлены в таблице table 10.2.

Наименование	Буква	Абревиатура	Команда
Scandinavian O	ø, Ø	\o, \O	
Scandinavian A	å, Å	\aa, \AA, {\r a}, {\r A}	\aring, \Aring
Polish L	ł, Ł	\l, \L	
German Eszett	ß	\ss, \SS	
„i” and „j” wihtout a point	ı, ȷ	\i, \j	
Hungarian Umlaut	ő, Ő	\H u, \H U	
Cedilla	ç, Ç	\c c, \c C	\ccedilla, \Ccedilla

Таблица 10.2 Больше диакритических знаков и специальных букв

Я хотел бы отметить, что некоторые команды в приведенной выше таблице генерируют символы из других символов, в то время как другие команды работают только в том случае, если шрифт, который мы используем, явно предоставлен для рассматриваемого символа. Итак, что касается немецкого Eszett (ß), в таблице показаны две команды, но только один символ, потому что шрифт, который я использую здесь для этого текста, предусматривает только версию немецкого Eszett в верхнем регистре (что-то довольно распространенное).

Вероятно, поэтому я тоже не могу получить скандинавский A в верхнем регистре, хотя «{\r A}» и \Aring работают правильно.

Венгерский умляут также работает с буквой „o”, а седиль - с буквами letters „k”, „l”, „n”, „r”, „s” and „t”, в нижнем или верхнем регистре соответственно. Используются следующие команды: \kcedilla, \lcedilla, \ncedilla ... соответственно.

[1] Из команд, найденных в table 10.1, тильда не работает с буквой „e”, и я не знаю почему.

[2] Помните, что в этом документе мы представляем пробелы , когда важно, чтобы мы их видели, с помощью „␣”.

10.1.1 Традиционные лигатуры

Лигатура образована объединением двух или более графем, которые обычно пишутся отдельно. Это «слияние» «fusion» двух символов часто начиналось как своего рода стенография в рукописных текстах, пока, наконец, они не достигли определенной типографской независимости. Некоторые из них даже были включены в число символов, которые обычно определяются в типографском шрифте, например, амперсанд, „&”, который начался как сокращение латинской связки (соединения) «et» или немецкого Eszett (ß). , который, как следует из названия, начинался как комбинация букв„s” и „z”. В дизайне некоторых шрифтов даже сегодня можно проследить происхождение этих двух символов; или, может быть, я вижу их, потому что знаю, что они там. В частности, со шрифтом Pagella для „&” и с Bookman для „ß”.

В качестве упражнения я предлагаю (после прочтения Chapter ??, где объясняется, как это сделать) попробовать представить эти символы с помощью этих шрифтов с достаточно большим размером (например, 30 pt), чтобы можно было проработать их компоненты.

Другие традиционные лигатуры, которые не стали такими популярными, но все еще иногда используются сегодня, - это латинские окончания «oe» и «ae», которые иногда записывались как „œ” или „ae”, чтобы указать, что они образовали дифтонг на латыни. Эти лигатуры могут быть достигнуты в ConTEXt с помощью команд, приведенных в table 10.3

Лигатура	Абревиатура	Команда
æ, Æ	`\ae, \AE`	`\aeligature, \AEligature`
œ, Œ	`\oe, \OE`	`\oeligature, \OEligature`

Таблица 10.3 Традиционные лигатуры

Лигатура, которая раньше была традиционной для испанского (кастильского) и которая обычно не встречается в шрифтах сегодня, - это „Ð”: сокращение, включающее „D” и „E”. Насколько мне известно, в ConTEXt нет команды, которая позволяла бы нам использовать это, [1], реализованную в пакете fontenc. но мы можем создать ее, как описано в section 10.1.4.

Наряду с предыдущими лигатурами, которые я назвал *традиционными*, потому что они происходят от рукописного ввода, после изобретения печатного станка были разработаны определенные лигатуры печатного текста, которые я назову «типографскими лигатурами», которые ConTEXt считает утилитами шрифтов и которыми автоматически управляет программа, хотя мы можем влиять на то, как эти утилиты шрифтов обрабатываются (включая лигатуры) с помощью `\definefontfeature` (не объясняется в этом введении).

10.1.2 Греческие буквы

Греческие символы обычно используются в математических и физических формулах. Вот почему ConTEXt включил возможность генерации всего греческого алфавита, в верхнем и нижнем регистре. Здесь команда построена на английском имени рассматриваемой греческой буквы. Если первый символ написан в нижнем регистре, у нас будет строчная греческая буква, а если он написан заглавными буквами, мы получим греческую букву в верхнем регистре. Например, команда `\mu` сгенерирует версию этой буквы в нижнем регистре (μ), а команда `\Mu` сгенерирует версию в верхнем регистре (M). В т table 10.4 мы можем увидеть, какая команда генерирует каждую из букв греческого алфавита, нижний и верхний регистры.

Обратите внимание, что для строчных версий некоторых символов (epsilon, kappa, theta, pi, rho, sigma and phi) есть два возможных варианта.

[1] В LATEX, напротив, мы можем использовать команду «`fontenc`» package.

Английское имя	Буква (lc/uc)	Команды (lc/uc)
Alpha	α, A	\alpha, \Alpha
Beta	β, B	\beta, \Beta
Gamma	γ, Γ	\gamma, \Gamma
Delta	δ, Δ	\delta, \Delta
Epsilon	ϵ, ε, E	\epsilon, \varepsilon, \Epsilon
Zeta	ζ, Z	\zeta, \Zeta
Eta	η, H	\eta, \Eta
Theta	θ, ϑ, Θ	\theta, \vartheta, \Theta
Iota	ι, I	\iota, \Iota
Kappa	κ, ϰ, K	\kappa, \varkappa, \Kappa
Lambda	λ, Λ	\lambda, \Lambda
Mu	μ, M	\mu, \Mu
Nu	ν, N	\nu, \Nu
Xi	ξ, Ξ	\xi, \Xi
Omicron	ο, O	\omicron, \Omicron
Pi	π, ϖ, Π	\pi, \varpi, \Pi
Rho	ρ, ϱ, P	\rho, \varrho, \Rho
Sigma	σ, ς, Σ	\sigma, \varsigma, \Sigma
Tau	τ, T	\tau, \Tau
Ypsilon	υ, Υ	\upsilon, \Upsilon
Phi	ϕ, φ, Φ	\phi, \varphi, \Phi
Chi	χ, X	\chi, \Chi
Psi	ψ, Ψ	\psi, \Psi
Omega	ω, Ω	\omega, \Omega

Таблица 10.4 Греческий алфавит

10.1.3 Различные символы

Вместе с символами, которые мы только что видели, TEX (а следовательно, и ConTEXt) предлагает команды для генерации любого количества символов. Таких команд много. Я представил расширенный, хотя и неполный список в Appendix ??.

10.1.4 Определение символов

Если нам нужно использовать какие-либо символы, недоступные с нашей клавиатуры, мы всегда можем найти веб-страницу с этими символами и скопировать их в наш исходный файл. Если мы используем кодировку UTF-8 (как рекомендуется), это почти всегда будет работать. Но также в вики ConTEXt есть страница с кучей символов, которые можно просто скопировать и вставить в наш документ. Чтобы получить их, перейдите по этой ссылке on this link.

Однако, если нам нужно использовать один из рассматриваемых символов более одного раза, тогда этот символ будет связан с командой, которая будет генерировать его каждый раз. Для этого мы используем \definecharacter, синтаксис которого:

\definecharacter *Name Character*

где

- **Name** – это имя, связанное с новым символом. Это не должно быть имя существующей команды, так как это приведет к перезаписи этой команды.
- **Character** – это символ, который создается каждый раз, когда мы запускаем *\Name*. Мы можем указать этот символ тремя способами:
  - ⋆ Просто записав его или вставив в наш исходный файл (если мы скопировали его из другого электронного документа или веб-страницы).
  - ⋆ Указав номер, связанный с этим символом, в шрифте, который мы используем в настоящее время. Чтобы увидеть символы, включенные в шрифт, связанные с ними числа, мы можем использовать команду \showfont[*Font name*].
  - ⋆ Создание нового символа с помощью одной из команд создания составного символа, которую мы увидим сразу после.

В качестве примера первого использования давайте вернемся к разделам, посвященным лигатурам (10.1.1). Там я рассказал о традиционной испанской лигатуре, которую сегодня обычно не встретишь в шрифтах: „Ð”. Мы могли бы связать этот символ, например, с командой `\decontract`, чтобы символ был сгенерирован всякий раз, когда мы пишем `\decontract`. Мы делаем это с помощью:

```
\definecharacter decontract Ð
```

> Чтобы создать новый символ, которого нет в нашем шрифте и который не может быть получен с клавиатуры, как в случае с примером, который я только что привел, сначала мы должны найти текст, в котором находится этот символ, скопировать его и иметь возможность вставить это в наше определение. В фактическом примере, который я только что привел, я изначально скопировал „Ð” из Википедии.

ConTEXt также включает некоторые команды, которые позволяют нам создавать составные символы и которые можно использовать в сочетании с`\definecharacter`. Под составными символами я подразумеваю символы с диакритическими знаками. Команды следующие:

```
\buildmathaccent Accent Character
\buildtextaccent Accent Character
\buildtextbottomcomma Character
\buildtextbottomdot Character
\buildtextcedilla Character
\buildtextgrave Character
\buildtextmacron Character
\buildtextognek Character
```

Например: как мы уже знаем, по умолчанию в ConTEXt есть только команды для написания определенных букв с седилем (c, k, l, n, r, s y t), которые обычно включаются в шрифты. Если бы мы хотели использовать „b”, мы могли бы использовать команду `\buildtextcedilla` следующим образом:

```
\definecharacter bcedilla {\buildtextcedilla b}
```

Эта команда создаст новую команду `\bcedilla`, которая сгенерирует „b” с седилем: „b̧”. Эти команды буквально «создают» новый символ, который будет сгенерирован, даже если в нашем шрифте его нет. Эти команды накладывают один символ на другой, а затем дают имя этому наложению.

> В моих тестах мне не удалось заставить работать `\buildmathaccent` и `\buildtextognek` . Так что я больше не буду их упоминать.

`\buildtextaccent` принимает два символа в качестве аргументов и накладывает один на другой, слегка увеличивая один из них. Хотя это называется «buildtextaccent», не обязательно, чтобы какой-либо из символов, взятых в качестве аргументов, был акцентом; но перекрытие даст лучшие результаты, если это так, потому что в этом случае, накладывая акцент на символ, акцент с меньшей вероятностью перезапишет символ. С другой стороны, перекрытие двух символов, которые имеют одинаковую базовую линию в нормальных условиях, зависит от того факта, что команда слегка приподнимает один из символов над другим. Вот почему мы не можем использовать эту команду, например, для получения упомянутого выше сокращения „Ð” , потому что если мы напишем

```
\definecharacter decontract {\buildtextaccent D E}
```

в нашем исходном файле небольшое возвышение над базовой линией „D”, которое вызывает эта команда («Ɒ»), не очень хорошо. Но если высота символов позволяет, мы могли бы создать комбинацию.. Например,

```
\definecharacter unusual {\buildtextaccent \_ "}
```

будет определять „"̲” символ, который будет связан с `\unusual` необычной командой.

Остальные команды сборки принимают единственный аргумент - символ, к которому будет добавлен диакритический знак, сгенерированный каждой командой. Ниже я покажу пример каждого из них, построенный на букве „z”:

– `\buildtextbottomcomma` добавляет запятую под символом, который он принимает в качестве аргумента („z̦”).

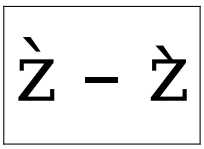
– `\buildtextbottomdot` добавляет точку под символом, который он принимает в качестве аргумента („ẓ”).

– `\buildtextcedilla` добавляет седиль под символом, который принимает в качестве аргумента („ȥ”).

– `\buildtextgrave` добавляет серьезный акцент над символом, который он принимает в качестве аргумента („ż”).

– `\buildtextmacron` добавляет небольшую полосу под символом, который он принимает в качестве аргумента („z̲”).

На первый взгляд, `\buildtextgrave` кажется излишним, учитывая, что у нас есть `\buildtextaccent`; Однако, если вы проверите серьезный акцент, созданный с помощью первой из этих двух команд, он выглядит немного лучше. В следующем примере показан результат обеих команд с достаточным размером шрифта, чтобы оценить разницу:



10.1.5 Использование предопределенных наборов символов

«ConTEXt Standalone» включает, наряду с самим ConTEXt ряд предопределенных наборов символов, которые мы можем использовать в наших документах. Эти наборы называются «`cc`», «`cow`», «`fontawesome`», «`on`», «`move`» и «`nav`». Каждый из этих наборов также включает несколько подмножеств:

– **`cc`** включает «cc».
– **`cow`** включает «cownormal» and «cowcontour».
– **`fontawesome`** включает «fontawesome».
– **`jmn`** включает «navigation 1», «navigation 2», «navigation 3» and «navigation 4».
– **`mvs`** включает «astronomic», «zodiac», «europe», «martinvogel 1», «martinvogel 2» and «martinvogel 3».
– **`nav`** включает «navigation 1», «navigation 2» and «navigation 3».

> В вики также упоминается набор под названием **`was`**, который включает в себя «wasy general», «wasy music», «wasy astronomy», «wasy astrology», «wasy geometry», «wasy physics» и «wasy apl». Но я не смог найти их в своем дистрибутиве, и мои попытки получить их не увенчались успехом.

Чтобы увидеть конкретные символы, содержащиеся в каждом из этих наборов, используется следующий синтаксис:

```
\usesymbols[Set]
\showsymbolset[Subset]
```

Например: если мы хотим видеть символы, включенные в «mvs/zodiac», то в исходном файле нам нужно написать:

```
\usesymbols[mvs]
\showsymbolset[zodiac]
```

и мы получим следующий результат:

Aquarius	♒	♒
Aries	♈	♈
Cancer	♋	♋
Capricorn	♑	♑
Gemini	♊	♊
Leo	♌	♌
Libra	♎	♎
Pisces	♓	♓
Sagittarius	♐	♐
Scorpio	♏	♏
Taurus	♉	♉
Virgo	♍	♍

Обратите внимание, что имя каждого символа указывается вместе с символом. Команда `\symbol` позволяет использовать любой из символов. Его синтаксис:

```
\symbol[Subset][SymbolName]
```

где subset - это одно из подмножеств, связанных с любым из наборов, которые мы ранее загрузили с помощью `\usesymbols`. Например, если бы мы хотели использовать астрологический символ, связанный с Водолеем (найденный в mvs / zodiac), нам нужно было бы написать

```
\usesymbols[mvs]
\symbol[zodiac][Aquarius]
```

который даст нам «♒», и это, для всех намерений и целей, будет рассматриваться как «character» и, следовательно, зависит от размера шрифта, который активен при печати. Мы также можем использовать `\definecharacter`, чтобы связать рассматриваемый символ с командой. Например

```
\definecharacter Aries {\symbol[zodiac][Aries]}
```

создаст новую команду с именем `\Aries`, которая сгенерирует символ «♈».

Мы могли бы также использовать эти символы, например, в окружении элементов. Например:

```
\usesymbols[mvs]
\definesymbol[1][{\symbol[martinvogel 2][PointingHand]}]
\definesymbol[2][{\symbol[martinvogel 2][CheckedBox]}]
\startitemize[packed]
\item item \item item
 \startitemize[packed]
 \item item \item item
 \stopitemize
\item item
\stopitemize
```

даст

- ☑ item
- ☑ item
  - ⋆ item
  - ⋆ item
- ☑ item

10.2 Форматы специальных символов

Строго говоря, именно команды *форматирования* влияют на используемый шрифт, его размер, стиль или вариант. Эти команды объясняются в Chapter ??. Однако, если смотреть в более *широком* смысле, мы также можем рассматривать команды, которые каким-то образом изменяют символы, которые они принимают в качестве аргумента (таким образом изменяя их внешний вид), как команды форматирования. В этом разделе мы рассмотрим некоторые из этих команд. Другие, такие как подчеркнутый или линейный текст со строками над или под текстом (например, там, где мы хотим предоставить место для ответа на вопрос), будут рассмотрены в section 12.5.

10.2.1 Верхний регистр, нижний регистр и псевдо маленькие заглавные буквы

Сами буквы могут быть в верхнем или нижнем регистре. Для ConTEXt прописные и строчные буквы - это разные символы, поэтому в принципе он будет набирать буквы так же, как находит их написанными. Однако есть группа команд, которые позволяют нам гарантировать, что текст, который они принимают в качестве аргумента, всегда записывается в верхнем или нижнем регистре:

- `\word{text}`: преобразует текст, взятый в качестве аргумента, в нижний регистр.
- `\Word{text}`: преобразует первую букву текста, взятого в качестве аргумента, в верхний регистр.
- `\Words{text}`: преобразует первую букву каждого слова, взятого в качестве аргумента, в верхний регистр; остальные в нижнем регистре.
- `\WORD{text}` или `\WORDS{text}`: записывает текст, взятый в качестве аргумента, в верхнем регистре.

Очень похожи на эти команды `\cap` и `\Cap`: они также используют заглавные буквы в тексте, который они принимают в качестве аргумента, но затем применяют к нему коэффициент масштабирования, равный тому, который применяется суффиксом „x” в командах изменения шрифта (см. 2), так что в большинстве шрифтов заглавные буквы будут той же высоты, что и строчные буквы, что дает нам эффект *псевдо* маленьких заглавных букв. По сравнению с настоящими маленькими крышками (см. section ??) они имеют следующие преимущества:

1. `\cap` and `\Cap` будут работать с любым шрифтом, в отличие от настоящих маленьких заглавных букв, которые работают только со шрифтами и стилями, которые явно включают их.

2. С другой стороны, настоящие маленькие заглавные буквы - это вариант шрифта, который как таковой несовместим с любым другим вариантом, таким как полужирный, курсив или наклонный. Тем не мение, `\cap` и `\Cap` полностью совместимы с любым вариантом шрифта.

Разница между `\cap` и `\Cap` заключается в том, что в то время как первый применяет коэффициент масштабирования ко всем буквам слов, составляющих его аргумент, `\Cap` не применяет масштабирование к первой букве каждого слова, таким образом достигая эффекта, аналогичного к тому, что мы получим, если использовать в тексте настоящие заглавные буквы. Если текст, взятый в качестве аргумента в „caps”, состоит из нескольких слов, размер заглавной буквы в первой букве каждого слова будет сохранен.

Таким образом, в следующем примере

```
The UN, whose \Cap{president} has his
office at \cap{uN} headquarters...
```

The UN, whose PRESIDENT has his office at UN headquarters...

Прежде всего, мы должны отметить разницу в размере между первым разом, когда мы пишем «UN» (заглавными буквами), и вторым разом (маленькими заглавными буквами quotationUN). В

этом примере я написал \cap{uN} второй раз, чтобы мы могли видеть, что не имеет значения, пишем ли мы аргумент, который \cap принимает в верхнем или нижнем регистре: команда преобразует все буквы в верхний регистр, а затем применяет коэффициент масштабирования; в отличие от\Cap, который не масштабирует первую букву.

Эти команды также могут быть *вложенными*, и в этом случае коэффициент масштабирования будет применен еще раз, что приведет к дальнейшему уменьшению, как в следующем примере, где слово «capital» в первой строке снова масштабируется:

```
\cap{People who have amassed their
\cap{capital} at the expense of others
are more often than not
{\bf decapitated} in revolutionary
times}.
```

PEOPLE WHO HAVE AMASSED THEIR CAPITAL AT THE EXPENSE OF OTHERS ARE MORE OFTEN THAN NOT **DECAPITATED** IN REVOLUTIONARY TIMES.

Команда \nocap, примененная к тексту, к которому применяется \cap, отменяет эффект \cap в тексте, который является его аргументом. Например:

```
\cap{When I was One I had just begun,
when I was Two I was \nocap{nearly}
new (A.A. Milne)}.
```

WHEN I WAS ONE I HAD JUST BEGUN, WHEN I WAS TWO I WAS nearly NEW (A.A. MILNE).

Мы можем настроить, как \cap работает с \setupcapitals, а также можем определить разные версии команды, каждая со своим именем и определенной конфигурацией. Это можно сделать с помощью \definecapitals.

Обе команды работают одинаково:

```
\definecapitals[Name][Configuration]
\setupcapitals[Name][Configuration]
```

Параметр «Name» в \setupcapitals не является обязательным. Если он не используется, конфигурация повлияет на саму команду \cap. Если он используется, нам нужно дать имя, которое мы ранее присвоили в \definecapitals, какой-то реальной конфигурации.

В любой из двух команд конфигурация допускает три параметра: «title», «sc» and «style», первая и вторая, позволяющие использовать «yes» и «no» в качестве значений. С помощью «title» мы указываем, будет ли использование заглавных букв также влиять на заголовки (что и происходит по умолчанию), а с «sc» мы указываем, должна ли команда быть настоящей маленькой заглавной («yes») или фальшивой заглавной («no»). По умолчанию он использует фальшивые маленькие заглавные буквы, что дает то преимущество, что команда работает, даже если вы используете шрифт, в котором не реализованы маленькие заглавные буквы. Третье значение «style» позволяет нам указать команду стиля, которая будет применяться к тексту, на который воздействует команда \cap.

10.2.2 Надстрочный или подстрочный текст

Мы уже знаем (см. section 3.1), что в математическом режиме зарезервированные символы «_» and «^» преобразуют следующий за ним символ или группу в верхний или нижний индекс. Для достижения этого эффекта вне математического режима ConTEXt включает следующие команды:

- \high{Text}:записывает текст, который принимает в качестве аргумента, в виде надстрочного индекса.
- \low{Text}: записывает текст, который принимает в качестве аргумента, в виде подстрочного индекса.

– `\lohi{Subscript}{Superscript}`: записывает оба аргумента, один над другим: внизу первый аргумент, а сверху второй, что вызывает любопытный эффект:

```
\lohi{below}{above}
```

above
below

10.2.3 Verbatim текст

Латинское выражение *verbatim* (от *verbum* = *word* + суффикс *atim*), которое можно перевести как «буквально» или «дословно», используется в программах обработки текста, таких как ConTEXt для обозначения фрагментов текста, которые не должны обрабатываться вообще, а должны быть выгружены в том виде, в котором они были записаны, в окончательный файл. ConTEXt использует для этого команду `\type`, предназначенную для коротких текстов, которые не занимают более одной строки, и среду `typing`, предназначенную для текстов более чем одной строки. Эти команды широко используются в компьютерных книгах для отображения фрагментов кода, и ConTEXt форматирует эти тексты моноширинными буквами, как пишущая машинка или компьютерный терминал. В обоих случаях текст отправляется в окончательный документ без *обработки*, что означает, что они могут использовать зарезервированные символы или специальные символы, которые будут преобразованы *как есть* в окончательный файл. Аналогично, если аргумент `\type` или содержимое `\starttyping` включает команду, она будет *написана* в окончательном документе, но не будет выполнена.

Кроме того, команда `\type` имеет следующую особенность: её аргумент *может* заключаться в фигурные скобки (как обычно в ConTEXt), но любой другой символ может использоваться для ограничения (окружения) аргумента.

Когда ConTEXt читает команду `\type`, он предполагает, что символ, который не является пробелом сразу после имени команды, будет действовать как разделитель ее аргумента; поэтому он считает, что содержимое аргумента начинается со следующего символа и заканчивается символом до следующего появления *разделителя*.

Несколько примеров помогут нам лучше понять это:

```
\type 1Tweedledum and Tweedledee1
\type |Tweedledum and Tweedledee|
\type zTweedledum and Tweedledeez
\type (Tweedledum and Tweedledee(
```

Обратите внимание, что в первом примере первым символом после имени команды является „1”, во втором - „|”, а в третьем - „z”; Итак: в каждом из этих случаев ConTEXt будет считать, что аргумент `\type` – это все, что находится между этим символом и следующим появлением того же символа. То же самое верно и для последнего примера, который также очень поучителен, потому что в принципе мы могли бы предположить, что если открывающий разделитель аргумента - это ‘(’, то закрывающий должен быть „)”, но это не так, потому что„(” и „)” - это разные символы, а `\type`, как я уже сказал, ищет закрывающий символ-разделитель, который совпадает с символом, используемым в начале аргумента.

Есть только два случая, когда `\type` позволяет использовать открывающий и закрывающий разделители разными символами:

– Если открывающим разделителем является символ „{”, предполагается, что закрывающим разделителем будет „}”.

– Если открывающий разделитель - „<<”, считается, что закрывающий разделитель будет „>>”. Этот случай также уникален тем, что в качестве разделителей используются два последовательных символа.

Однако тот факт, что `\type` допускает любые разделители, не означает, что мы должны использовать «weird (странные)» разделители. С точки зрения читабельности и понятности источника файла, лучше всего ограничивать аргумент `\type` по возможности фигурными скобками, как это обычно бывает с ConTEXt; а когда это невозможно из-за фигурных скобок в аргументе `\type`, используйте символ: предпочтительно тот, который не является зарезервированным символом ConTEXt. Например:

```
\type *This is a closing curly bracket: '}'*.
```

И `\type`, и `\starttyping` можно настроить с помощью `\setuptype` и `\setuptyping`. Мы также можем создать их индивидуальную версию с помощью `\Definetype` и `\Definetyping`. Что касается фактических параметров конфигурации для этих команд, я обращаюсь к «`setup-en.pdf`» (в каталоге `tex/texmf-context/doc/context/documents/general/qrcs`.

Две очень похожие на \type команды:

– \typ: работает аналогично \type, but не отключает расстановку переносов.

– \tex: команда, предназначенная для написания текстов о TEX или ConTEXt: она добавляет пробел перед текстом, который принимает в качестве аргумента. В противном случае эта команда отличается от \type тем, что обрабатывает некоторые из зарезервированных символов, которые она находит в тексте, который принимает в качестве аргумента. В частности, фигурные скобки внутри \tex будут обрабатываться так же, как они обычно обрабатываются в ConTEXt.

10.3 Межсимвольные и межсловные интервалы

10.3.1 Автоматическая установка горизонтального пространства

Пространство между разными символами и словами (называемое *горизонтальным пространством* в TEX) обычно автоматически устанавливается ConTEXt:

– Пространство между символами, составляющими слово, определяется самим шрифтом, который, за исключением шрифтов с фиксированной шириной, обычно использует большее или меньшее количество пробелов в зависимости от разделяемых символов, и поэтому, например, расстояние между буквами «A» и «V» («AV») обычно меньше, чем пространство между „A” и „V” („AV”). Однако, помимо этих возможных вариантов, которые зависят от комбинации рассматриваемых букв и предопределены шрифтом, расстояние между символами, составляющими слово, в целом является фиксированной и неизменной мерой.

– Напротив, интервал между словами в одной строке может быть более эластичным.

  ⋆ В случае слов в строке, ширина которой должна быть такой же, как у остальных строк в абзаце, изменение интервала между словами является одним из механизмов, которые ConTEXt использует для получения строк одинаковой ширины, так как более подробно объяснено в разделе 11.3. В этих случаях ConTEXt установит точно такое же горизонтальное расстояние между всеми словами в строке (за исключением правил, приведенных ниже), при этом гарантируя, что расстояние между словами в разных строках абзаца будет как можно более похожим.

  ⋆ Однако, в дополнение к необходимости растягивать или сокращать интервалы между словами для выравнивания строк, в зависимости от активного языка, ConTEXt принимает во внимание определенные типографские правила, посредством которых в определенных местах типографская традиция, связанная с этим языком, добавляет некоторые дополнительные пустое пространство, как, например, в некоторых частях английской типографской традиции, при котором после точки добавляются дополнительные пробелы.

    Эти дополнительные пробелы работают для английского языка и, возможно, для некоторых других языков (хотя верно и то, что во многих случаях издатели на английском языке в настоящее время предпочитают не оставлять дополнительное пространство после точки), но не для испанского, где типографская традиция отличается. Таким образом, мы можем временно включить эту функцию с помощью \setupspacing[wide] и отключить её с помощью \setupspacing[pack]. Мы также можем изменить конфигурацию по умолчанию для испанского (и в этом отношении для любого другого языка, включая английский), как описано в section 10.5.2.

10.3.2 Изменение расстояния между символами в слове

Изменение пространства по умолчанию для символов, составляющих слово, считается очень плохой практикой с типографской точки зрения, за исключением заголовков и заголовков. Однако

ConTEXt предоставляет команду для изменения этого промежутка между символами в слове: [1] `\растянутый`, синтаксис которого следующий:

```
\stretched[Configuration]{Text}
```

где *Configuration* допускает любой из следующих параметров:

- `factor`: целое или десятичное число, представляющее получаемый интервал. Это число не должно быть слишком большим. Коэффициент 0,05 виден уже невооруженным глазом.
- `width`: указывает общую ширину, которую должен иметь текст, передаваемый команде, таким образом, чтобы команда сама вычисляла необходимый интервал для распределения символов в этом пространстве.

  Согласно моим тестам, когда ширина, установленная с помощью параметра `width`, меньше ширины, необходимой для представления текста с *factor*, равным 0,25, параметр *width* и этот factor игнорируются. Я предполагаю, что это потому, что `\stretch` позволяет нам только *увеличивать* расстояние между символами в слове, но не уменьшать его. Но я не понимаю, почему ширина, необходимая для представления текста с коэффициентом 0,25, используется в качестве минимальной меры для параметра `width`, а не *natural width* текста (с factor от 0).

  According to my tests, when the width established with the `width` option is less than that required to represent the text with a *factor* equal to 0.25, the *width* option and this factor are ignored. I guess that's because `\stretched` allows us only *to increase* the space between the characters in a word, not reduce it. But I don't understand why the width required to represent the text with a factor of 0.25 is used as a minimum measure for the `width` option, and not the *natural width* of the text (with a factor of 0).

- **`style`**: команда стиля или команды, применяемые к тексту, взятому в качестве аргумента.
- **`color`**: цвет, которым будет написан текст, взятый в качестве аргумента.

Итак, в следующем примере мы можем графически увидеть, как команда будет работать при применении к одному и тому же предложению, но с разной шириной:

```
\stretched[width=4cm]{\bf test text}
\stretched[width=6cm]{\bf test text}
\stretched[width=8cm]{\bf test text}
\stretched[width=9cm]{\bf test text}
```

t e s t t e x t
t e s t t e x t
t e s t t e x t
t e s t t e x t

В этом примере можно увидеть, что распределение горизонтального пространства между разными символами неоднородно. „x" и „t" в «text» и „e" и „b" в «test» всегда располагаются гораздо ближе друг к другу, чем другие символы. Мне не удалось выяснить, почему это происходит.

При использовании без аргументов команда будет использовать всю ширину линии. С другой стороны, в тексте, который является аргументом этой команды, команда переопределяется и вместо разрыва строки вставляется горизонтальный пробел. Например:

Команда без аргументов будет использовать всю ширину линии. С другой стороны, в тексте, который является аргументом этой команды, команда `\\` переопределяется и вместо разрыва строки вставляется горизонтальный пробел. Например:

[1] Очень типично для философии ConTEXt включать команду для выполнения чего-то, что сама документация ConTEXt не рекомендует делать. Несмотря на то, что типографское совершенство требуется, цель также состоит в том, чтобы дать автору полный контроль над внешним видом своего документа: короче говоря, лучше или хуже - это его или её ответственность.

```
\stretched{test}\\text}
```

t e s t
t e x t

Мы можем настроить конфигурацию команды по умолчанию с помощью`\setupstretched`.

Однако нет команды `\definestretched`, которая позволила бы нам устанавливать индивидуальные конфигурации, связанные с именем команды , в официальном списке команд (см. section ??) там сказано, что `\setupstretched` происходит из `\setupcharacterkerning`, а есть `\definecharacterkerning` команда. Однако в моих тестах мне не удалось установить какую-либо индивидуальную конфигурацию для `\stretch` с помощью последней, хотя я должен признать, что я тоже не тратил много времени на попытки сделать это.

10.3.3 Команды для добавления горизонтального пробела между словами

Мы уже знаем, что для увеличения промежутка между словами бесполезно добавлять два или более последовательных пробелов, поскольку ConTEXt поглощает все последовательные пробелы, как объяснено в section ??. Если мы хотим увеличить расстояние между словами, нам нужно перейти к одной из команд, которая позволяет нам это сделать:

– cmd , вставляет в документ очень маленькое пустое пространство (называемое тонким пространством). Оно используется, например, для разделения тысяч в наборе чисел (например, 1000000) или для отделения одиночной кавычки от двойных кавычек. Например: quotation`1\,473\,451` создаст «1 473 451».

– `\space` или «`\␣`» (обратная косая черта, за которой следует пробел, который, поскольку это невидимый символ, я представил как «␣») вводит дополнительное пустое пространство.

– `\enskip`, `\quad` и `\qquad` вставит пустое пространство в половину *em*, 1 *em* или 2 *ems* соответственно. Помните, что *em* - это мера, зависящая от размера шрифта и эквивалентная ширине „m”, которая обычно совпадает с размером шрифта в пунктах. Итак, при использовании шрифта размером 12 пунктов `\enskip` дает нам пространство в 6 пунктов, `\quad` дает нам 12 пунктов, а `\qquad` дает нам 24 пункта.

Наряду с этими командами, которые дают нам пустое пространство при точных измерениях, команды `\hskip` и `\hfill` вводят горизонтальное пространство разных размеров:

`\hskip` позволяет нам точно указать, сколько пустого пространства мы хотим добавить. Таким образом:

```
This is \hskip 1cm 1 centimetre\\
This is \hskip 2cm 2 centimetres\\
This is \hskip 2.5cm 2.5 centimetres\\
```

This is 1 centimetre
This is 2 centimetres
This is 2.5 centimetres

Указанный пробел может быть отрицательным, что приведет к наложению одного текста на другой. Таким образом:

```
This is farce rather than
\hskip -1cm comedy
```

Это скорее фарс, чем комедия

`\hfill`, со своей стороны, вводит столько пробелов, сколько необходимо, чтобы занять всю строку, что позволяет нам создавать интересные эффекты, такие как текст с выравниванием по правому краю, центрированный текст или текст с обеих сторон строки, как показано в следующий пример:

```
\hfill On the right\\
On both\hfill sides
```

On both On the right
sides

10.4 Составные слова

Под «составными словами» в этом разделе я подразумеваю слова, которые формально понимаются как одно слово, а не слова, которые просто соединяются. Это различие не всегда легко понять: «радуга» явно состоит из двух слов («rain + bow»), но ни один англоговорящий человек не подумает о комбинированных терминах иначе, как об одном слове. С другой стороны, у нас есть слова, которые иногда объединяются с помощью дефиса или обратной косой черты. Эти два

слова имеют разные значения и употребления, но соединены (и в некоторых случаях могут стать одним словом, но еще нет!). Так, например, мы можем найти такие слова, как «French – Canadian» или «(inter) communication» (хотя мы также можем найти «intercommunication» и обнаружить, что говорящая публика наконец приняла два слова должны быть одним словом. Так развивается язык).

Сложные слова представляют для ConTEXt некоторые проблемы, в основном связанные с их потенциальным переносом в конце строки. Если соединяющим элементом является дефис, то с типографской точки зрения проблем с переносом в конце строки в этой точке не возникает, но нам нужно избегать второго переноса во второй части слова, так как это оставит нас с два подряд идущих дефиса, которые могут вызвать трудности с пониманием.

Доступна команда «||», чтобы сообщить ConTEXt, что два слова составляют составное слово. Эта команда, в исключительных случаях, не начинается с обратной косой черты и допускает два разных использования:

- Мы можем использовать две последовательные вертикальные черты (вертикальные черты) и написать, например, «`Испанский ||Аргентинский`».
- Две вертикальные полосы могут иметь элемент соединения / разделения между двумя словами, заключенными между ними, как, например, «`joining|/|separating`».

В обоих случаях ConTEXt будет знать, что имеет дело со сложным словом, и применит соответствующие правила расстановки переносов для этого типа слова. Разница между использованием двух последовательных вертикальных полос (каналов) или обрамлением ими разделителя слов заключается в том, что в первом случае ConTEXt будет использовать разделитель, который предопределен как `\setuphyphenmark`, или другими словами дефис, который используется по умолчанию («`--`»). Итак, если мы напишем «`picture||frame`», ConTEXt создаст«Picture–frame».

С помощью `\setuphyphenmark` мы можем изменить разделитель по умолчанию (в случае, когда нам нужны две черты). Допустимые значения для этой команды: «`-, ---, -,  , (,), =, /`». Однако имейте в виду, что значение «=» превращается в длинное тире (то же самое, что и «`---`»).

Обычно «||» используется с дефисами, так как это то, что обычно используется между составными словами. Но иногда разделителем может быть скобка, если, например, мы хотим „(inter)space”, или косая черта, как в „input/output”. В этих случаях, если мы хотим, чтобы применялись обычные правила расстановки переносов для составных слов, мы могли бы написать «`(inter|)|space`» or «`input|/|output`». Как я сказал ранее,
MyKey|=| считается сокращением от «`|---|`» и вставляет длинное тире в качестве разделителя (—).

10.5 Язык текста

Символы образуют слова, которые обычно принадлежат какому-либо языку. Для ConTEXt важно знать язык, на котором мы пишем, потому что от этого зависит ряд важных вещей. В основном:

- Расстановка переносов в словах.
- Формат вывода определенных слов.
- Определенные вопросы набора, связанные с традициями набора данного языка.

10.5.1 Установка и изменение языка

ConTEXt предполагает, что языком будет английский. Две процедуры могут изменить это:

- С помощью команды `\mainlanguage`, используемой в преамбуле для изменения основного языка документа.
- С помощью команды `\language`, предназначенной для изменения активного языка в любом месте документа.

Обе команды ожидают аргумент, состоящий из любого идентификатора языка (или кода). Для идентификации языка мы используем либо двухбуквенный международный код языка, установленный в ISO 639-1, который совпадает с используемым, например, в Интернете, либо английское название рассматриваемого языка, либо иногда некоторая аббревиатура названия на английском.

В table 10.5 мы находим полный список языков, поддерживаемых ConTEXt, вместе с кодом ISO для каждого из рассматриваемых языков, а также, при необходимости, кодом для определенных языковых вариантов, специально предусмотренных. [1]

Language	**ISO code**	**Language** (variants)
Afrikaans	af, afrikaans	
Arabic	ar, arabic	ar-ae, ar-bh, ar-dz, ar-eg, ar-in, ar-ir, ar-jo, ar-kw, ar-lb, ar-ly, ar-ma, ar-om, ar-qa, ar-sa, ar-sd, ar-sy, ar-tn, ar-ye
Catalan	ca, catalan	
Czech	cs, cz, czech	
Croatian	hr, croatian	
Danish	da, danish	
Dutch	nl, nld, dutch	
English	en, eng, english	en-gb, uk, ukenglish, en-us, usenglish
Estonian	et, estonian	
Finnish	fi, finnish	
French	fr, fra, french	
German	de, deu, german	de-at, de-ch, de-de
Greek	gr, greek	
Greek (ancient)	agr, ancientgreek	
Hebrew	he, hebrew	
Hungarian	hu, hungarian	
Italian	it, italian	
Japanese	ja, japanese	
Korean	kr, korean	
Latin	la, latin	
Lithuanian	lt, lithuanian	
Malayalam	ml, malayalam	
Norwegian	nb, bokmal, no, norwegian	nn, nynorsk
Persian	pe, fa, persian	
Polish	pl, polish	
Portuguese	pt, portughese	pt-br
Romanian	ro, romanian	
Russian	ru, russian	
Slovak	sk, slovak	
Slovenian	sl, slovene, slovenian	
Spanish	es, sp, spanish	es-es, es-la
Swedish	sv, swedish	
Thai	th, thai	
Turkish	tr, turkish	tk, turkmen
Ukranian	ua, ukrainian	
Vietnamese	vi, vietnamese	

Таблица 10.5 Поддержка языков в ConTEXt

Так, например, чтобы установить испанский (кастильский) в качестве основного языка документа, мы могли бы использовать любой из трех следующих:

```
\mainlanguage[es]
\mainlanguage[spanish]
\mainlanguage[sp]
```

Чтобы включить определенный язык *внутри* документа, мы можем использовать либо команду `\language [Language code]`, либо специальную команду для активации этого языка. Так, напри-

[1] Table 10.5 содержит сводку списка, полученного с помощью следующих команд:
```
\usemodule[languages-system]
\loadinstalledlanguages
\showinstalledlanguages
```
Если вы читаете этот документ спустя много времени после его написания (2020 г.), возможно, в ConTEXt будут включены дополнительные языки, поэтому было бы неплохо использовать эти команды для отображения обновленного списка языков.

мер, \en активирует английский, \fr активирует французский, \es испанский или \ca каталонский. Как только реальный язык был активирован, он остается таковым до тех пор, пока мы не переключимся на другой язык или группа, в которой был активирован язык, не будет закрыта. Таким образом, языки работают так же, как команды смены шрифтов. Однако обратите внимание, что язык, установленный командой \language или одним из ее сокращений (\en, \fr, \de, и т.д.) не влияет на язык, на котором печатаются метки (см. section 10.5.3).

Хотя может быть сложно отметить язык всех слов и выражений, которые мы используем в нашем документе, которые не принадлежат к основному языку документа, это важно сделать, если мы хотим получить окончательный документ с правильным набором, особенно в профессиональной деятельности. Мы должны отмечать не весь текст, а только ту часть, которая не относится к основному языку. Иногда можно автоматизировать маркировку языка с помощью макроса. Например, для этого документа, в котором постоянно цитируются команды ConTEXt, исходным языком которого является английский, я разработал макрос, который, помимо записи команды в соответствующем формате и цвете, помечает ее как английское слово. В моей профессиональной работе, когда мне нужно цитировать много французской и итальянской библиографии, я включил поле в свою библиографическую базу данных, чтобы выбрать язык работы, чтобы я мог автоматизировать указание языка в цитатах и списках библиографические ссылки.

Если мы используем два языка, которые используют разные алфавиты в одном и том же документе (например, английский и греческий или английский и русский), есть уловка, которая избавит нас от необходимости отмечать язык выражений, построенных с помощью альтернативного алфавита: измените настройку основного языка (см. следующий раздел), чтобы она также загружала шаблоны переносов по умолчанию для языка, использующего другой алфавит. Например, если мы хотим использовать английский и древнегреческий языки, следующая команда избавит нас от необходимости отмечать язык текстов на греческом:

```
\setuplanguage[en][patterns={en, agr}]
```

Это работает только потому, что в английском и греческом языках используется другой алфавит, поэтому не может быть конфликта в схемах переноса двух языков, поэтому мы можем загружать их оба одновременно. Но в двух языках, использующих один и тот же алфавит, одновременная загрузка шаблонов расстановки переносов обязательно приведет к неправильной расстановке переносов.

10.5.2 Настройка языка

ConTEXt связывает работу определенных утилит с конкретным языком, активным в любой момент времени. Связи по умолчанию можно изменить с помощью \setuplanguage, синтаксис которого:

```
\setuplanguage[Language][Configuration]
```

где *Language* - это код языка для языка, который мы хотим настроить, а *Configuration* содержит конкретную конфигурацию, которую мы хотим установить (или изменить) для этого языка. В частности, разрешено до 32 различных вариантов конфигурации, но я буду иметь дело только с теми, которые кажутся подходящими для вводного текста, например этого:

- **date**: позволяет нам настроить формат даты по умолчанию. Смотрите дальше на page 139.
- **lefthyphenmin, righthyphenmin**: минимальное количество символов, которое должно быть слева или справа для поддержки переносов слова. Например, \setuplanguage[en][lefthyphenmin=4] не будет переносить любое слово, если слева от возможного дефиса меньше 4 символов.
- **spacing**: возможные значения для этого параметра: «широкий» «broad» или «упакованный» «packed». В первом случае (широком) будут применяться правила расстановки интервалов между словами на английском языке, что означает, что после точки и когда следует другой символ, будет добавлено определенное количество дополнительного пробела. С другой стороны, «spacing=packed» не позволит применять эти правила. Для английского языка по умолчанию используется широкий.
- **leftquote, rightquote**: укажет символы (или команды), соответственно, которые \quote будут использоваться слева и справа от текста, являющегося его аргументом (для этой команды см.page 141).
- **leftquotation, rightquotation**: укажите символы (или команды), соответственно, которые \quotation будет использовать слева и справа от текста, являющегося его аргументом (для этой команды см. page 141)

10.5.3 Ярлыки

Многие команды ConTEXt автоматически генерируют определенные тексты (или *labels*), как, например, команда `\placetable`, которая записывает метку «Table xx» под вставляемой таблицей, или `\placefigure`, который вставляет метку «Figure xx».

Эти *labels* чувствительны к языку, установленному с помощью `\mainlanguage` (но не чувствительны, если установлены с помощью `\language`), и мы можем изменить их с помощью

```
\setuplabeltext[Language][Key=Label]
```

где *Key* – термин, по которму ConTEXt узнает метку, а *Label* – это текст, который мы хотим чтобы был сгенерирован ConTEXtТак например,

`\setuplabeltext[es][figure=Imagen~]` увидит, что когда основным языком является испанский, изображения, вставленные с помощью `\placefigure` будут называться не «Figure x», а «Imagen x».Обратите внимание, что после текста на самой этикетке необходимо оставить пробел, чтобы гарантировать, что этикетка не будет прикреплена к следующему символу. В этом примере я использовал символ «~»; можно было бы написать «`[figure=Imagen{ }]`» заключив пробел между фигурными скобками, чтобы ConTEXt не избавился от него.

Какие ярлыки мы можем переопределить с помощью `\setuplabeltext`? Документация ConTEXt по этому поводу не так полна, как можно было бы надеяться. В справочном руководстве 2013 г. (которое объясняет больше всего об этой команде) упоминаются «`chapter`», «`table`», «`figure`», «`appendix`»... и дополнительные «другие сопоставимые текстовые элементы». Можно предположить, что имена будут английскими названиями рассматриваемого элемента.

> Одним из преимуществ *free libre software* является то, что исходные файлы доступны пользователю; так что мы можем изучить их.Я сделал это и, просматривая *snooping* исходные файлы ConTEXtобнаружил файл «`lang-txt.lua`», доступный в `tex/texmf-context/tex/context/base/mkiv` который, как мне кажется, содержит предопределенные метки и их различные переводы; так что если в любое время ConTEXt сгенерирует переопределенный текст который мы хотим изменить, чтобы увидеть имя метки, с которой связан текст, мы можем открыть соответствующий файл и обнаружить, что мы хотим изменить. Таким образом, мы можем увидеть, какое название ярлыка с ним связано.

Если мы хотим вставить текст, связанный с определенной меткой, где-нибудь в документе, мы можем сделать это с помощью команды `\labeltext` command. Так, например, если я хочу сослаться на таблицу, чтобы гарантировать, что я назову ее так же, как ConTEXt вызывает ее в команде `\placetable`, я могу написать: «`Так же, как показано в \labeltext{table} на следующей странице.` Этот текст в документе, где `\mainlanguage` – английский, будет содержать: «Так же, как показано в Таблица на следующей странице ».

> Некоторые метки, которые можно переопределить с помощью `\setuplabeltext`, по умолчанию пусты; например,«`chapter`» или «`section`». Это связано с тем, что по умолчанию ConTEXt не добавляет метки к командам секционирования. Если мы хотим изменить эту операцию по умолчанию, нам нужно только переопределить эти метки в преамбуле нашего документа, и поэтому, например, `\setuplabeltext[chapter=Chapter~]` увидит, что главам предшествует слово «Chapter».

Наконец, важно отметить, что, хотя, как правило, в ConTEXt команды, которые допускают несколько параметров, разделенных запятыми, в качестве аргумента, последний параметр может заканчиваться запятой, и ничего плохого не происходит. В `\setuplabeltext`, который генерирует ошибку при компиляции.

10.5.4 Некоторые языковые команды

A. Команды

ConTEXt есть три команды, связанные с датой, которые выводят свой результат на активном языке во время выполнения. Это:

– `\currentdate`: запускается без аргументов в документе, в котором основным языком является английский, возвращает системную дату в формате «Day Month Year» «День Месяц Год».

Например: «11 сентября 2020 года». Но мы также можем указать ему использовать другой формат (как это произошло бы в США и некоторых других частях англоязычного мира, которые следуют их системе размещения месяца перед днем, отсюда и печально известная дата, 9/11), или включать название дня недели (дня недели (`weekday`), или включать только некоторые элементы даты (день, месяц, год `day, month, year`).

Чтобы указать другой формат даты, «`dd`» или «`day`» представляют дни, «`mm`» месяцы (в числовом формате), MyKeymonth месяцы в алфавитном формате строчными буквами и «`MONTH`» в верхнем регистре. Что касается года, то «`yy`» будет записывать только последние цифры, а «`year`» или «`y`» - все четыре. Если нам нужен какой-то разделительный элемент между компонентами даты, мы должны написать это прямо. Например

```
\currentdate[weekday, dd, month]
```

when run on 9 September 2020 will write «Wednesday 9 September».

– `\date`: эта команда, запущенная без аргументов, выдает точно такой же результат, что и `\currentdate`, то есть фактическая дата в стандартном формате. Однако в качестве аргумента может быть указана конкретная дата. Для этого даны два аргумента: с помощью первого аргумента мы можем указать день («`d`»), месяц («`m`») и год («`y`»), соответствующие дате, которую мы хотим представить, а вторым аргументом (необязательным) мы можем указать формат представляемой даты. Например, если мы хотим знать, в какой день недели встречались Джон Леннон и Пол Маккартни, событие, которое, согласно Википедии, произошло 6 июля 1957 года, мы могли бы написать

  ```
  \date[d=6, m=7, y=1957][weekday]
  ```

  and so we would find out that such an historical event happened on a Saturday.

– `\month` принимает число в качестве аргумента и возвращает название месяца, соответствующего этому числу.

Б. Команда `\translate`

Команда translate поддерживает серию фраз, связанных с определенным языком, так что тот или иной будет вставлен в окончательный документ в зависимости от языка, активного в любой момент времени. В следующем примере команда translate используется для связывания четырех фраз с испанским и английским, которые сохраняются в буфере памяти (относительно среды `buffer` см. section 12.6):

```
\startbuffer
  \starttabulate[|*{4}{lw(.25\textwidth)|}]
    \NC \translate[es=Su carta de fecha, en=Your letter dated]
    \NC \translate[es=Su referencia, en=Your reference]
    \NC \translate[es=Nuestra referencia, en=Our reference]
    \NC \translate[es=Fecha, en=Date] \NC\NR
  \stoptabulate
\stopbuffer
```

так что если мы вставим *buffer* в ту точку документа, где активирован испанский, будут воспроизводиться испанские фразы, но если в той точке документа, где вставлен буфер, активирован английский, английские фразы будут вставлен. Таким образом:

```
\language[es]
\getbuffer
```

будет генерировать

Su carta de fecha Su referencia Nuestra referencia Fecha

в тоже время

```
\language[en]
\getbuffer
```

будет генерировать

Your letter dated Your reference Our reference Date

C. Команды \quote и \quotation

Одна из наиболее распространенных типографских ошибок в текстовых документах возникает, когда кавычки (одинарные или двойные) открываются, но не закрываются явно. Чтобы этого не произошло, ConTEXt предоставляет команды \quote и \quotation, которые цитируют текст, являющийся их аргументом; \quote будет использовать одинарные кавычки, а \quotation будет использовать двойные кавычки.

Эти команды чувствительны к языку, поскольку они используют символ или набор команд по умолчанию для рассматриваемого языка, чтобы открывать и закрывать кавычки (см. section 10.5.2); и поэтому, например, если мы хотим использовать испанский в качестве стиля по умолчанию для двойных кавычек - гильметов или шевронов (угловых скобок)), типичных для испанского, итальянского, французского языков, мы должны написать:

```
\setuplanguage[es][leftquotation=«, rightquotation=»]
```.

Однако эти команды не управляют вложенными кавычками; хотя мы можем создать утилиту, которая делает это, используя тот факт, что \quote и \quote являются фактическими приложениями того, что ConTEXt вызывает *delimitedtext*, и что можно определить дальнейшие приложения с \definedelimitedtext. Таким образом, следующий пример:

```
\definedelimitedtext
  [CommasLevelA]
  [left=«, right=»]

\definedelimitedtext
  [CommasLevelB]
  [left=“, right=”]

\definedelimitedtext
  [CommasLevelC]
  [left=`, right=']
```

создаст три команды, которые позволят использовать до трех разных уровней цитирования. Первый уровень с боковыми кавычками, второй с двойными кавычками и третий с одинарными кавычками.

Конечно, если мы используем английский в качестве основного языка, то автоматически будут использоваться одинарные и двойные кавычки по умолчанию (фигурные, а не прямые, как вы найдете в этом документе!).

Глава 11

Параграфы, строки и вертикальные пространства

Общий вид документа определяется в основном размером и компоновкой страниц, которые мы видели в Chapter 5, шрифтом, который мы выбрали, с которым мы работали в Chapter ??, а также другими вопросами, такими как межстрочный интервал, выравнивание абзацев, интервал между ними и т. д. В этой главе основное внимание уделяется этим другим вопросам.

11.1 Параграфы и их характеристики

Абзац является основной единицей текста для ConTEXt. Существует две процедуры, чтобы начать абзац:

1. Вставка одной или нескольких последовательных пустых строк в исходный файл.
2. Команды `\par` или `\endgraf`.

Обычно используется первая из этих процедур, поскольку она проще и создает исходные файлы, которые легче читать и понимать. Вставка разрывов абзаца с помощью явной команды обычно выполняется только внутри макроса (см. section 3.7.1) или в ячейке таблицы (см.section 13.3).

В хорошо набранном документе с типографской точки зрения важно, чтобы абзацы визуально выделялись друг от друга. Обычно это достигается двумя процедурами: небольшим отступом в первой строке каждого абзаца или небольшим увеличением пустого пространства между абзацами, а иногда и комбинацией обеих процедур, хотя в некоторых местах это не рекомендуется, поскольку считается типографически избыточным.

> Я не совсем согласен. Простой отступ первой строки не всегда достаточно визуально выделяет разделение между абзацами; но увеличение интервала, не сопровождающееся отступом, создает проблемы в случае абзаца, который начинается в верхней части страницы, и поэтому мы можем не знать, новый ли это абзац или продолжение предыдущей страницы. Комбинация обеих процедур избавляет от сомнений.

Давайте посмотрим, прежде всего, как достигается отступ строк и абзацев с помощью ConTEXt.

11.1.1 Автоматический отступ в первых строках абзацев

По умолчанию автоматическая вставка небольшого отступа в первую строку абзаца отключена. Мы можем включить его, отключить снова и, когда он будет включен, указать степень отступа с

помощью команды \setupindenting, которая позволяет следующим значениям указать, следует ли включать отступы:

- **always**: все абзацы будут иметь отступ независимо от этого.
- **yes**: включить *normal* отступ абзаца. Некоторые абзацы, которым предшествует дополнительный вертикальный интервал, например, первый абзац разделов или абзацы, следующие за определенными средами, не будут иметь отступа.
- **no, not, never, none**:отключить автоматический отступ первой строки в абзацах.

В случаях, когда мы включили автоматический отступ, мы также можем указать с помощью той же команды, какой отступ должен быть. Для этого мы можем явно использовать размер (например, 1,5 см) или символические слова “small”, “medium” и “big”, которые указывают, что нам нужны маленькие, средние или большие отступы.

> В некоторых традициях набора текста (в том числе испанских) отступ по умолчанию составлял два квадрата. В типографии четырехугольник (первоначально *quadrat) представлял собой металлическую прокладку, используемую при наборе текста печатными буквами. Позже этот термин был принят в качестве общего названия для двух общих размеров пространства в типографии, независимо от используемой формы набора текста. Квадрат em - это пространство шириной в один em; шириной в высоту шрифта (Википедия). Таким образом, с буквой из 12 пунктов квадрат будет иметь ширину 12 пунктов и высоту 12 пунктов. ConTEXt содержит две команды quad: quad, которая генерирует одно пространство, указанное выше, и \qquad, которое генерирует вдвое больше, но на основе используемого шрифта. Отступ в два квадрата с буквой из 11 пунктов будет составлять 22 балла, а с буквой из 12 пунктов - 24 балла.*

Когда отступы включены, если мы не хотим, чтобы определенный абзац был с отступом, нам нужно использовать команду \noindentation.

> Обычно я включаю автоматический отступ в своих документах с помощью \setupindenting[yes, big]. Однако в этом документе я этого не делал, потому что, если бы был включен отступ, большое количество коротких предложений и примеров привело бы к визуально неопрятному виду страниц.

11.1.2 Отступ специального абзаца

Одной из графических процедур выделения абзаца является отступ либо справа, либо слева (или с обеих сторон) от абзаца. Это используется, например, для блока двойных кавычек.

ConTEXt имеет среду, которая позволяет нам изменять отступ абзаца, чтобы выделить текст в абзаце. Это среда “narrower”:

```
\startnarrower[Options] ... \stopnarrower
```

где *Options* могут быть:

- **left**: отступ по левому краю.
- **Num*left**:сделает отступ для левого края, умножив *normal* отступ на *Num* (например, 2*left).
- **right**: отступ по правому краю.
- **Num*right**: делает отступ для правого края, умножив *normal* отступ на *Num* (например, tt 2*right).
- **middle**: делает отступ по обоим краям. Это значение по умолчанию.
- **Num*middle**: делает отступ на обеих сторонах, умножая *normal* отступ на *Num*.

При объяснении опций я упомянул *нормальный отступ*; это относится к величине отступа слева и справа, который “narrower” применяет по умолчанию. Этот *amount* можно настроить с помощью \setupnarrower, который позволяет использовать следующие параметры конфигурации:

- **left**: величина отступа для левого края.
- **right**: величина отступа для правогг края.

- **middle**: величина отступа для обоих краев.
- **before**: команда, которую нужно запустить перед входом в среду.
- **after**: команда, которая будет запущена после существующей среды.

Если мы хотим использовать разные конфигурации более узкой среды в нашем документе, мы можем присвоить каждому из них другое имя с помощью `\Definenarrower[Name][Configuration]`

где *Name* - это имя, связанное с этой конфигурацией, а *Configuration* допускает те же значения, что и `\setupnarrower`.

11.2 Вертикальное пространство между параграфами

11.2.1 \setupwhitespace

Как мы уже знаем из (section 4.2.2), для ConTEXt не имеет значения, сколько последовательных пустых строк в исходном файле: одна или несколько пустых строк вставят один разрыв абзаца в итоговый документ. Чтобы увеличить расстояние между абзацами, не нужно добавлять лишнюю пустую строку в исходный файл. Вместо этого эта функция управляется командой `\setupwhitespace`, которая допускает следующие значения:

- **none**: означает, что между абзацами не будет дополнительного вертикального пробела.
- **small, medium, big**: в этих вставках соответственно маленькое, среднее или большое вертикальное пространство. Фактический размер пространства, вставляемого этими значениями, зависит от размера шрифта.
- **line, halfline, quarterline**: измеряет дополнительное пустое пространство по отношению к высоте строк и вставляет дополнительную строку, половину строки или четверть строки соответственно.
- **DIMENSION**: устанавливает фактический размер пространства между абзацами. Например, `\setupwhitespace[5pt]`.

Как правило, не рекомендуется устанавливать точный размер в качестве значения для `\setupwhitespace`. Предпочтительно использовать символьные значения small, medium, big, line, halfline или четверть линии. Это так по двум причинам:

- Символьные значения являются эластичными размерами (см. section ??), что означает, что они имеют *нормальные* размеры, но допускается некоторое уменьшение или увеличение этого значения, чтобы помочь ConTEXt при наборе страниц, для того, чтобы разрывы абзацев были эстетически похожи. Фиксированный размер разделения между абзацами затрудняет достижение хорошей разбивки на страницы для документа.
- Символьные значения small, medium, big и т. Д. Рассчитываются на основе размера шрифта, поэтому, если это изменится в определенных частях, это также изменит величину вертикального интервала между абзацами, и конечный результат всегда будет гармоничным. И наоборот, на фиксированное значение вертикального интервала не повлияют изменения размера шрифта, который обычно переводится в документ с плохо распределенным белым пространством (с эстетической точки зрения) и не в соответствии с правилами типографской настройки.

Если для вертикального интервала между абзацами установлено значение, доступны две дополнительные команды: `\nowhitespace`, которая удаляет лишние пробелы между определенными абзацами, и `\whitespace`, которая делает противоположный. Однако эти команды нужны редко, потому что ConTEXt сам достаточно хорошо управляет вертикальным интервалом между абзацами; особенно, если один из предопределенных размеров был вставлен в качестве значения, рассчитанного на основе текущего активного размера шрифта и интервала.

Значение \nowhitespace очевидно. Но не обязательно \whitespace, потому что какой смысл упорядочивать интервалы по вертикали для определенных абзацев, учитывая, что интервал по вертикали уже обычно установлен для всех абзацев? Однако при написании расширенных макросов \whitespace может быть полезен в контексте цикла, который должен принимать решение на основе значения определенного условия. Это более или менее сложное программирование, и я не буду здесь вдаваться в подробности.

11.2.2 Paragraphs with no extra vertical space between them

Если мы хотим, чтобы в определенных частях нашего документа были абзацы, не разделенные дополнительным вертикальным пространством, мы, конечно, можем изменить общую конфигурацию \setupwhitespace, но это в некотором смысле противоречит философии ConTEXt согласно которой общие команды конфигурации должны быть помещены исключительно в преамбулу исходного файла, чтобы обеспечить согласованный и легко изменяемый общий вид документов. ПОэтому, среда “Packed”, общий синтаксис которой

```
\startpacked[Space] ... \stoppacked
```

где *Space* - необязательный аргумент, указывающий, какое расстояние по вертикали желательно между абзацами в среде. Если этот параметр не указан, дополнительное вертикальное пространство не будет применяться.

11.2.3 Добавление дополнительного вертикального пространства в определенном месте документа

Если в определенной точке документа обычного вертикального интервала между абзацами недостаточно, мы можем использовать команду \blank. При использовании без аргументов \blank вставляет такое же количество вертикального пространства, какое было установлено с помощью \setupwhitespace. Но мы можем указать либо конкретный размер в квадратных скобках, либо одно из символьных значений, вычисленных из размера шрифта: маленький, средний или большой. Мы также можем умножить эти размеры на некоторое целое число и так далее, например, \blank[3*medium] вставит эквивалент трех средних разрывов строки. Мы также можем соединить два размера вместе. Например, \blank [2*big,medium] вставит два больших и средних разрыва.

Поскольку \blank предназначен для увеличения вертикального расстояния между абзацами, он не действует, если разрыв страницы вставляется между двумя абзацами, расстояние между которыми должно быть увеличено; и если мы вставим две или более команд \blank подряд, будет применяться только одна из них (та, в которой будет больше всего места для вставки). Команда \blank, помещенная после разрыва страницы, также не имеет никакого эффекта. Однако в этих случаях мы можем принудительно вставить вертикальный интервал, используя символическое слово “force” в качестве параметра команды. Так, например, если мы хотим, чтобы заголовки глав в нашем документе отображались ниже по странице, чтобы общая длина страницы была меньше, чем остальные страницы (относительно частая типографская практика), мы должны писать в конфигурация команды \chapter, например:

```
\setuphead
  [chapter]
  [
    page=yes,
    before={\blank[4cm, force]},
    after={\blank[3*medium]}
  ]
```

Эта последовательность команд гарантирует, что главы всегда начинаются с новой страницы, а метка главы перемещается на четыре сантиметра вниз. Без использования параметра “force” это не сработает.

11.2.4 \setupblank and \defineblank

Ранее я сказал, что \blank, используемый без аргументов, эквивалентен \blank [big]. Однако мы можем изменить это с помощью \setupblank, установив, например, \setupblank[0.5cm] или \setupblank[medium]. При использовании без аргументов \setupblank подстраивает значение под размер текущего шрифта.

Как и в случае с \setupwhitespace, пробел, вставленный \blank, когда его значение является одним из предопределенных символьных значений, является эластичным размером, допускающим некоторую корректировку. Мы можем изменить это с помощью “fixed”, с возможностью позже восстановить значение по умолчанию с помощью (“flexible”). Так, например, для текста в двойных столбцах рекомендуется установить \setupblank [fixed, line], а при возврате к одиночному столбцу - \setupblank[flexible, default].

С помощью \defineblank мы можем связать определенную конфигурацию с именем. Общий формат этой команды:

```
\defineblank[Name] [Configuration]
```

Как только наша конфигурация пустого пространства определена, мы можем использовать ее с помощью \blank[ConfigurationName].

11.2.5 Другие процедуры для достижения большего вертикального пространства

В TEX команда, которая вставляет лишнее вертикальное пространство, - это \vskip. Эта команда, как и почти все команды TEX также работает в ConTEXt но настоятельно не рекомендуется ее использовать, поскольку она мешает внутреннему функционированию некоторых макросов ConTEXt. Вместо него предлагается использовать \godown, синтаксис которого:

```
\godown[Измерение]
```

где *Dimension* должно быть числом с десятичными знаками или без них, за которым следует единица измерения. Например, \godown [5cm] сместит страницу на 5 сантиметров вниз; хотя, если изменение страницы меньше этой суммы, \godown переместится только на следующую страницу. Точно так же \godown не будет иметь никакого эффекта в начале страницы, хотя мы можем *обмануть его*, написав, например, ‘\␣\godown [3cm]’[1] Который сначала вставит пробел, что будет означать, что мы уже не в начале страницы, а затем опустится на три сантиметра.

> Как мы знаем, \blank также допускает точное измерение в качестве аргумента. Следовательно, с точки зрения пользователя, написание \blank [3cm] или \godown[3cm] практически одно и то же. Однако между ними есть некоторые тонкие различия. Так, например, две последовательные команды \blank не могут быть накоплены, и когда это происходит, применяется только та, которая требует большего расстояния. С другой стороны, две или более команды \godown могут отлично накапливаться.

Другая довольно полезная команда TEX использование которой не вызывает проблем в ConTEXt, - это \vfill. Эта команда вставляет гибкое вертикальное пустое пространство до низа страницы. Как будто команда *pushes* выталкиваетто, что написано после нее. Это позволяет получить интересные эффекты, например, как разместить определенный абзац внизу страницы, просто поставив перед ним \vfill. Теперь эффект \vfill трудно оценить, если его использование не сочетается с принудительными разрывами страниц, потому что нет смысла сдвигать абзац или строку текста вниз, если абзац по мере его роста растет вверх. Так, например, чтобы убедиться, что строка находится внизу страницы, мы должны написать:

```
\vfill
Line at the bottom
```

[1] Напомним, что мы используем в этом документе символ ‘␣’ для обозначения пустого места, когда нам важно его увидеть.

```
\page[yes]
```

Как и все другие команды, которые вставляют вертикальный пробел, `\vfill` не действует в начале страницы. Но мы можем *обмануть его*, поставив перед ним принудительный пробел. Так, например:

```
\page[yes]
\ \vfill
Centre line
\vfill
\page[yes]
```

будет вертикально центрировать фразу центральная линия на странице.

11.3 Как ConTEXt строит строки, образующие абзацы

Одна из основных задач системы набора текста - взять длинную строку слов и разделить ее на отдельные строки подходящего размера. Например, каждый абзац в этом тексте разделен на строки шириной 15 сантиметров, но автору не нужно было беспокоиться о таких деталях, поскольку ConTEXt выбирает точки останова после рассмотрения каждого абзаца целиком, так что последние слова абзаца действительно может повлиять на деление первой линии. В результате пространство между словами во всем абзаце становится максимально равномерным.

> Это один из аспектов, в котором мы можем лучше всего отметить другой способ работы текстовых редакторов и лучшее качество, достигаемое с такими системами, как ConTEXt. Поскольку текстовый процессор, когда он достигает конца строки и переходит к следующей, регулирует пробел в только что завершенной строке, чтобы включить выравнивание по правому краю. Это делается с каждой строкой, и в конце каждая строка в абзаце будет иметь различный интервал между словами. Это может вызвать очень плохой эффект (например, ‘реки’ пробелов, проходящих через текст). ConTEXt, с другой стороны, обрабатывает абзац целиком и для каждой строки вычисляет, сколько точек останова допустимо, а также величину межсловного интервала, которая может возникнуть в результате разрыва строки. Поскольку точка останова строки влияет на потенциальные точки останова следующих строк, общее количество возможностей может быть очень большим; но это не проблема для ConTEXt. Он примет окончательное решение на основе всего абзаца, гарантируя, что расстояние между словами в каждой строке будет *как можно более похожим*, что приводит к гораздо лучшему набранному абзацу; визуально более компактный.

Для этого ConTEXt тестирует разные альтернативы и присваивает каждой из них значение плохой репутации *badness* на основе её параметров. Они были созданы после глубокого изучения искусства книгопечатания. Наконец, изучив все возможности, ConTEXt выбирает наименее неподходящий вариант (с наименьшим значением вредности). В общем, это работает достаточно хорошо, но неизбежно будут случаи, когда точки останова по строкам выбираются не из лучших или которые нам не кажутся лучшими. Поэтому иногда мы хотим сказать программе, что определенные места не являются хорошими точками останова. Тогда в других случаях мы захотим принудительно прервать работу в определенной точке.

11.3.1 Использование зарезервированного символа ‘~’

Основными кандидатами на точки останова на строку, очевидно, являются пробелы между словами. Чтобы указать, что определенный пробел никогда не следует заменять разрывом строки, мы используем, как мы уже знаем, зарезервированный символ ‘~’, который TEX называет *tie*, связывая два слова вместе.

Обычно рекомендуется использовать это неразрывное пространство в следующих случаях:

- Между частями, составляющими сокращение. Например, `U~S`.
- Между сокращениями и термином, к которому они относятся. Например, `Dr~Anne Ruben` or `p.~45`.
- Между числами и связанным с ними термином. Например, `Elizabeth~II`, `45~volumes`.
- Между цифрами и предшествующими или следующими за ними символами, если они не являются надстрочными индексами. Например, `73~km`, `$~53`; however, `35'`.

- В процентах, выраженных словами. Например, `twenty~per~cent`.
- IB группах чисел, разделенных пробелом. Например, `5~357~891`. Хотя в этих случаях предпочтительнее использовать так называемое тонкое пространство *thin space*, достигаемое в ConTEXt с помощью команды `\,` и, следовательно, записью `5\,357\,891`.
- Чтобы аббревиатура не была единственным элементом в этой строке. Например:

```
There are sectors such as entertainment, communications media,
commerce,~etc.
```

К этим случаям Knuth (отец TEX) добавляет следующие рекомендации:

- После сокращения, которое не находится в конце предложения.
- Относительно частей документа, таких как главы, приложения, рисунки и т. Д. Например, `Chapter~12`.
- Между именем и инициалом второго имени лица или между инициалом имени и фамилией. Например, `Donald~E. Knuth`, `A.~Einstein`.
- Между математическими символами в приложении к именам. Например, `dimension~$d$`, `width~$w$`.
- Между последовательными символами. Например, `{1,~2, \dots,~$n$}`.
- Когда число строго связано с предлогом. Например `from 0 to~1`.
- Когда математические символы выражаются словами. Например, `equals~a~$n$`.
- В списках внутри абзаца. Например: `(1)~green, (2)~red, (3)~blue`.

Много случаев? Без сомнения, типографское совершенство требует дополнительных усилий. Понятно, что если мы не хотим этого, нам необязательно применять эти правила, но знать их не помешает. Кроме того, - и здесь я говорю по собственному опыту - как только мы привыкаем применять их (или любой из них), это становится автоматическим. Это похоже на расстановку акцентов на словах, когда мы их пишем (что и нужно делать по-испански): для тех из нас, кто это делает, если мы привыкли писать их автоматически, нам больше не нужно писать слово с помощью акцента, чем для слова без акцента.

11.3.2 Перенос слов

За исключением языков, состоящих в основном из односложных слов, довольно сложно получить оптимальный результат, если точки останова на строке находятся только в пространстве между словами. Следовательно, ConTEXt также анализирует возможность вставки разрыва строки между двумя слогами слова; и для этого важно, чтобы он знал язык, на котором написан текст, поскольку правила расстановки переносов различны для каждого языка. Таким образом, объясняется важность команды`\mainlanguage` в преамбуле документа.

Может случиться так, что ConTEXt не смог правильно расставить слово через дефис. Иногда это может быть связано с тем, что собственные правила разделения слов мешают задаче (например, ConTEXt никогда не разбивает слово на две части, если в этих частях нет минимального количества букв); или потому что слово неоднозначное. В конце концов, что ConTEXt может сделать со словом “объединенный в профсоюзы”? Это слово могло появиться во фразе вроде “объединенная рабочая сила”, но оно также могло появиться в тексте по химии как “объединенная частица” (т.е. неионизированная). А что, если ConTEXt будет иметь дело со словом “непредумышленное убийство” в качестве последнего слова на странице перед разрывом страницы. Он может разделить слово как man-slaughter человеко-убийство (правильно), но он также может разделить его как смех человека mans-laughter (двусмысленно).

Какой бы ни была причина, если мы не удовлетворены тем, как слово было разделено, или оно неверно, мы можем изменить его, явно указав потенциальные точки, в которых слово может быть разделено, с помощью управляющего символа `\-`. Так, например, если “unionized” доставляет нам какие-либо проблемы, мы можем записать это в исходный файл как “`union\-ised`”; или если у нас возникла проблема с “manslaughter”, мы могли бы написать “`man\-slaughter`”.

Если проблемное слово используется в нашем документе несколько раз, то предпочтительнее показать, как оно должно быть расставлено через дефис в нашей преамбуле с помощью команды `\hyphenation`: эта команда, которая предназначена для включения в преамбулу исходного файла, занимает одну или несколько слов (через запятую) в качестве аргумента, указывающих точки, в которых они могут быть разделены дефисом. Например:

```
\hyphenation{union-ised, man-slaughter}
```

Если слово, являющееся предметом этой команды, не содержит дефиса, то в результате слово никогда не будет расставлено через дефис. Того же эффекта можно добиться, используя команду `\hbox`, которая создает неделимое горизонтальное поле вокруг слова, или `\unhyphenated`, предотвращающее перенос слова или слов, которые он принимает в качестве аргументов. Но в то время как `\hyphenation` действует глобально, `\hbox` и `\unhyphenated` действуют локально, что означает, что команда `\hyphenation` влияет на все вхождения в документ слов, включенных в ее аргумент; в отличие от `\hbox` или `\unhyphenated`, которые действуют только в том месте исходного файла, где они встречаются.

Внутри системы расстановка переносов контролируется переменными TEX `\pretolerance` и `\tolerance`. Первый из них контролирует допустимость разделения, сделанного только на пустом пространстве. По умолчанию это 100, но если мы изменим его, например, на 10 000, то ConTEXt всегда будет считать приемлемым наличие разрыва строки, что не означает разделение слов по слогам, что означает, что *de facto*, мы убираем расстановку переносов по слогам. Если, например, мы должны установить значение `\pretolerance` на -1, мы бы заставили ConTEXt каждый раз использовать перенос слов в конце строки.

Мы можем напрямую установить произвольное значение для `\pretolerance`, просто назначив это значение в нашем документе. Например:

```
\pretolerance=10000
```

но мы также можем управлять этим значением с помощью значений “`lesshyphenation`” и “`morehyphenation`” в `\setupalign`. По этому поводу см. section 11.6.1.

11.3.3 Уровень допуска для разрывов строк

При поиске возможных точек разрыва строки ConTEXt обычно довольно строг, что означает, что он предпочитает, чтобы слово выходило за пределы правого поля, потому что оно не могло расставить его через дефис, и предпочитает не вставлять разрыв строки перед слово, если это приводит к слишком большому увеличению межсловного пространства в этой строке. Такое поведение по умолчанию обычно обеспечивает оптимальные результаты, и только в исключительных случаях некоторые линии немного выделяются с правой стороны. Идея состоит в том, что автор (или наборщик) просматривает эти исключительные случаи после того, как документ будет закончен, чтобы принять соответствующее решение, которое может быть командой `\break` перед словом, выходящим за пределы, или может также означать другую формулировку абзаца, так что что это слово меняет позицию в другом месте.

Однако в некоторых случаях низкая толерантность ConTEXt может быть проблемой. В этих случаях мы можем сказать, что он более терпим с пробелами в строках. Для этого у нас есть команда `\setuptolerance`, позволяющая изменять уровень допуска при вычислении разрывов строк, который ConTEXt вызывает “horizontal tolerance” (поскольку он влияет на горизонтальное пространство) и quotationvertical tolerance при расчете разрывов страниц. Мы поговорим об этом в section 11.6.2.

Горизонтальная толерантность (которая влияет на разрывы строк) по умолчанию установлена на значение “`verystrict`”. Мы можем изменить это, установив в качестве альтернативы любое из следующих значений: “`strict`”, “`tolerant`”, “`verytolerant`” или “`stretch`”. Так, например,

```
\setuptolerance[horizontal, verytolerant]
```

сделает практически невозможным выход строки за правое поле, даже если это означает установление очень большого и неприглядного интервала между словами в строке.

11.3.4 Принудительный разрыв строки в определенной точке

Чтобы вызвать разрыв строки в определенной точке, мы используем команды \break, \crlf или \\. Первый из них, \break, вводит разрыв строки в том месте, где он расположен. Это, скорее всего, приведет к эстетической деформации строки, в которой размещена команда, с огромным количеством пробелов между словами в этой строке. Как видно из следующего примера, в котором команда \break в третьей строке (исходного фрагмента слева) приводит ко второй довольно уродливой строке (в отформатированном тексте справа).

```
On the corner of the old quarter I saw him \emph{swagger} along
like the\break tough guys do when they walk, hands always in
their overcoat pockets, so no one can know which of them carries
the dagger.
```

On the corner of the old quarter I saw him *swagger* along like the
tough guys do when they walk, hands always in their overcoat poc-
kets, so no one can know which of them carries the dagger.

Чтобы избежать этого эффекта, мы можем использовать команды \\ или \crlf, которые также вставляют принудительный разрыв строки, но они заполняют исходную строку достаточным количеством пустого места, чтобы выровнять ее по левому краю:

```
On the corner of the old quarter I saw
him \emph{swagger} along like
the\\ tough guys do when they walk,
hands always in their overcoat pockets,
so no one can know which of them carries
the dagger.
```

On the corner of the old quarter I saw him *swagger* along like the
tough guys do when they walk, hands always in their overcoat poc-
kets, so no one can know which of them carries the dagger.

На обычных строках *normal*, насколько мне известно, нет различий между \\ или \crlf; но в названии раздела есть разница:

- \\ создает разрыв строки в теле документа, но не тогда, когда заголовок раздела переносится в оглавление.
- **\crlf** создает разрыв строки, который применяется как в теле документа, так и при переносе заголовка раздела в оглавление.

Разрыв строки не следует путать с разрывом абзаца. Разрыв строки просто завершает текущую строку и начинает следующую строку, но сохраняет нас в том же абзаце, поэтому разделение между исходной строкой и новой строкой будет определяться обычным интервалом внутри абзаца. Следовательно, есть только три сценария, в которых можно рекомендовать принудительный разрыв строки:

- В очень исключительных случаях, когда ConTEXt не смог найти подходящий разрыв строки, так что линия выступает справа. В этих случаях (которые возникают очень редко, в основном, когда строка содержит неделимые *boxes* или *verbatim* текст [см. section 10.2.3]), может быть полезно принудительно разрыв строки с помощью \break непосредственно перед словом, которое выступает за правое поле.
- В абзацах, которые фактически состоят из отдельных строк, каждая из которых содержит информацию, не зависящую от информации в предыдущих строках, например, заголовок письма, в котором первая строка может содержать имя отправителя, вторая - получателя и третье свидание; или в тексте, где говорится об авторстве работы, где в одной строке указано имя автора, в другой - его должность или академическая должность и, возможно, в третьей строке указывается дата и т. д. В этих случаях разрыв строки должен быть принудительно выполнен с помощью символа \\ или \crlf команд. Параграфы такого типа также часто выравниваются по правому краю.
- При написании стихов или подобных текстов для отделения одного стиха от другого. Хотя в последнем случае предпочтительнее использовать среду `lines`, описанную в section 11.5.1.

11.4 Межстрочный интервал

Межстрочный интервал - это расстояние, разделяющее строки, составляющие абзац. ConTEXt вычисляет это автоматически на основе фактического используемого шрифта и, прежде всего, на основе базового размера, установленного с помощью `\setupbodyfont` или `\switchtobodyfont`.

Мы можем влиять на межстрочное пространство с помощью команды `\setupinterlinespace`, которая допускает три различных вида синтаксиса:

- `\setupinterlinespace [..Interline space..]`, где *Interline space* это точное значение или символическое слово, которое назначает предварительно определенный межстрочный интервал:
  - Если это точное значение, это может быть размер (например, 15pt) или простое, целое или десятичное число (например, 1,2). В последнем случае число интерпретируется как "количество строк" на основе межстрочного интервала ConTEXt по умолчанию.
  - Когда это символическое слово, это может быть "`small`", "`medium`" или "`big`", каждый из которых применяет малый, средний или большой межстрочный интервал соответственно, всегда на основе межстрочного интервала по умолчанию ConTEXt будет применяться.
- `\setupinterlinespace [..,..=..,..]`. В этом режиме межстрочный интервал устанавливается путем явного изменения базовых показателей, с помощью которых ConTEXt вычисляет соответствующий межстрочный интервал. В этом режиме интервал устанавливается путем явного изменения мер, на основе которых ConTEXt вычисляет соответствующий интервал. Я ранее говорил, что межстрочный интервал рассчитывается на основе конкретного шрифта и его размера; но это было сделано для того, чтобы все было очень просто: на самом деле шрифт и его размер устанавливают определенные меры, на основе которых рассчитывается межстрочное пространство. Посредством этого подхода `\setupinterlinespace` эти меры изменяются, а следовательно, и межстрочное пространство. Фактические меры и значения, которыми можно манипулировать с помощью этой процедуры (смысл которых я не буду объяснять, потому что он выходит за рамки простого *введения*), следующие: `line`, `height`, `depth`, `minheight`, `mindepth`, `distance`, `top`, `bottom`, `stretch` и `shrink`.
- `\setupinterlinespace [Name]`.В этом режиме мы устанавливаем или настраиваем особый и индивидуальный тип межстрочного интервала, ранее определенный с помощью `\defineinterlinespace`.

С помощью

```
\defineinterlinespace[Name] [Configuration]
```

мы можем связать определенную конфигурацию межстрочного пространства с определенным именем, которое затем мы можем просто активировать в какой-то момент нашего документа с помощью `\setupinterlinespace[Name]`. Чтобы вернуться к нормальному межстрочному пространству, нам нужно будет написать `\setupinterlinespace[reset]`.

11.5 Прочие вопросы, касающиеся строк

11.5.1 Преобразование разрывов строк в исходном файле в разрывы строк в окончательном документе

Как мы уже знаем (см. section 4.2.2), по умолчанию ConTEXt игнорирует разрывы строк в исходном файле, которые он считает простыми пробелами, если нет двух или более последовательных разрывов строки, и в этом случае разрыв абзаца будет вставлен. Тем не менее, могут быть

некоторые ситуации, в которых мы заинтересованы в соблюдении разрывов строк в исходном исходном файле, как они были помещены там, например, при написании стихов. Для этого ConTEXt предлагает нам среду "lines", формат которой:

```
\startlines[Options] ... \stoplines
```

где варианты могут быть, среди прочего, любыми из следующих:

- **space**: пробел: если для этого параметра установлено значение «включено» "on", в дополнение к соблюдению разрывов строк в исходном файле среда также будет учитывать пробелы в исходном файле, временно игнорируя правило поглощения.
- **before**:текст или команда для запуска перед входом в среду.
- **after**: Текст или команда для запуска после выхода из среды.
- **inbetween**: Текст или команда для запуска при входе в среду.
- **indenting**: Значение, указывающее, следует ли делать отступ абзацев в среде (см. section 11.1.1).
- **align**: Выравнивание линий в окружении (см. section 11.6).
- **style**: Команда стиля для применения в среде.
- **color**: Цвет для применения в среде.

Так, например,

```
\startlines
  One-one was a race horse.
  Two-two was one too.
  One-one won one race.
  Two-two won one too.
\stoplines
```

One-one was a race horse.
Two-two was one too.
One-one won one race.
Two-two won one too.

Мы также можем изменить способ работы среды по умолчанию с помощью \setuplines и, как и в случае со многими другими командами ConTEXt, также можно присвоить имя определенной конфигурации этой среды. Мы делаем это с помощью команды \definelines, синтаксис которой:

```
\definelines[Name] [Configuration]
```

где в качестве конфигурации мы можем включить те же параметры, которые были объяснены в целом для среды. После того, как мы определили нашу настроенную линейную среду, чтобы вставить ее, мы должны написать:

```
\startlines[Name] ... \stoplines
```

11.5.2 Нумерация строк

В некоторых типах текстов принято устанавливать некоторую нумерацию строк, например, в текстах по компьютерному программированию, где относительно часто фрагменты кода, предлагаемые в качестве примеров, имеют пронумерованные строки, или в стихах, критических редакциях и т.д. Для всех этих ситуаций ConTEXt предлагает среду нумерации строк linenumbering, формат которой

```
\startlinenumbering[Options] ... \stoplinenumbering
```

Доступные варианты опций:

- **continue**: в случаях, когда несколько частей нашего документа требуют нумерации строк, эта опция видит, что нумерация возобновляется для каждой части (“continue=no”, значение по умолчанию). С другой стороны, если нумерация строк предназначена для продолжения с того места, где была остановлена предыдущая часть, мы выбираем “continue=yes”.
- **start**: указывает номер первой строки в тех случаях, когда мы не хотим, чтобы она была ‘1’ или чтобы она соответствовала предыдущему перечислению.
- **step**: все строки, включенные в среду, будут пронумерованы, но с помощью этой опции мы можем указать, что номер печатается только через определенные промежутки времени. В стихах, например, часто встречается, что число появляется только кратно 5 (стихи 5, 10, 15 ...).

Все эти параметры могут быть указаны, как правило, для всех сред *lnumbering* в нашем документе с помощью \setuplinenumbering. Эта команда также позволяет нам настроить другие аспекты нумерации строк:

- **conversion**: Тип нумерации строк. Это может быть любой из описанных на page 88 относительно нумерации глав и разделов.
- **style**: Команда (или команды), определяющая стиль нумерации строк (шрифт, размер, вариант ...).
- **color**: Цвет, которым будет напечатан номер строки.
- **location**:Где будет размещен номер строки. Это может быть любое из следующих значений: text, begin, end, default, left, right, inner, outer, inleft, inright, margin, inmargin.
- **distance**: Расстояние между номером строки и самой строкой.
- **align**: Выравнивание номеров. Может быть: inner, outer, flushleft, flushright, left, right, middle or auto.
- **command**: Команда, которой перед печатью будет передан номер строки в качестве параметра.
- **width**: Ширина, зарезервированная для печати номера строки.
- **left, right, margin**:

Мы также можем создавать различные настраиваемые конфигурации нумерации строк с помощью \definelinenumbering, чтобы конфигурация была связана с именем:

```
\definelinenumbering[Name] [Configuration]
```

После того, как конкретная конфигурация была определена и связана с именем, мы можем использовать ее с

```
\startlinenumbering[Name] ... \stoplinenumbering
```

11.6 Горизонтальное и вертикальное выравнивание

Командой, которая обычно управляет выравниванием текста, является \setupalign. Эта команда используется для управления как горизонтальным, так и вертикальным выравниванием.

11.6.1 Горизонтальное выравнивание

Когда точная ширина строки текста не занимает всю возможную ширину, возникает проблема: что делать с результирующим пустым пространством. [1] В этом отношении мы можем сделать три вещи:

[1] Под *точной* шириной я подразумеваю ширину строки *до того*, как ConTEXt настроит размер междусловного пространства для обеспечения выравнивания.

1. Накапливайте пространство на одной из двух сторон линии: если мы накапливаем его на левой стороне, линия будет выглядеть немного сдвинутой *a little pushed* вправо, в то время как если мы накапливаем её на правой стороне, линия останется на левой стороне. В первом случае мы говорим о выравнивании по правому краю *right alignment*, а во втором - о выравнивании по левому краю *left alignment*. По умолчанию ConTEXt применяет выравнивание по левому краю к последней строке абзацев.

   Когда несколько последовательных линий выровнены слева, правая сторона нерегулярна; но когда выравнивание происходит справа, неровная сторона оказывается левой. Чтобы назвать параметры, которые выравнивают ту или иную сторону, ConTEXt задает не сторону, где они выровнены, а сторону, где они неровные. Следовательно, опция `flushright` приводит к выравниванию по левому краю, а `flushleft` - к выравниванию по правому краю. В качестве сокращений от `flushright` и `flushleft` `\setupalign` также поддерживает значения `right` и `left`. Но **внимание**: здесь значение слов обманчиво. Хотя *left* означает “left”, а *right* означает “right”, `\setupalign[left]` выравнивается по правому краю, а `\setupalign[right]` выравнивается по левый. Если читатель задается вопросом, почему был сделан этот комментарий, стоит процитировать ConTEXt wiki: ‘ConTEXt использует опции flushleft и flushright. Правое и левое выравнивание происходит в обратном направлении от обычных направлений во всех командах, которые допускают вариант выравнивания, в смысле ‘ragged left’ и ‘ragged right’. К сожалению, когда Ганс впервые писал эту часть ConTEXt, он думал о выравнивании ‘ragged right’ и ‘ragged left’, а не о ‘flush left’ и ‘flush right’. И теперь, когда так было некоторое время, изменить его невозможно, потому что его изменение нарушит обратную совместимость со всеми существующими документами, которые его используют.’

   В документах, подготовленных для двусторонней печати, помимо правого и левого полей есть также внутренние и внешние поля. Значения `flushinner` (или просто `inner`) и `flushouter` (или просто `outer`) устанавливают соответствующее выравнивание в этих случаях.

2. Распределите его по обоим полям. В результате линия будет отцентрирована. Параметр `\setupalign`, который делает это, – `middle`.

3. Распределите его между всеми словами,составляющими строку, при необходимости увеличив межсловное пространство, чтобы строка стала точно такой же ширины, как и доступное для нее пространство. В этих случаях мы говорим об обоснованных линиях. Это также значение ConTEXt по умолчанию, поэтому в `\setupalign` нет специальной опции для его установки. Однако, если мы изменили выравнивание по умолчанию, мы можем восстановить его с помощью `\setupalign[reset]`.

начение для параметра `\setupalign`, которое мы только что видели (`right, flushright, left, flushleft, inner, flushinner, outer, flushouter` и `middle`), может быть объединено с `broad` (широким), что приведёт к несколько более грубому выравниванию.

> Два других возможных значения `\setupalign`, которые влияют на выравнивание по горизонтали, связаны с переносом слов в конце строки, потому что, будет ли это сделано или нет, зависит от того, будет ли точная мера строки больше или меньше; что, в свою очередь, влияет на оставшееся пустое пространство
>
> Для этого `\setupalign` позволяет использовать большее значение расстановки переносов `morehyphenation`, что заставляет ConTEXt труднее находить точки останова на основе расстановки переносов, и меньше расстановки переносов `lesshyphenation`, что дает противоположный эффект. При использовании `\setupalign[horizontal, morehyphenation]` оставшееся свободное пространство в строках будет уменьшено, и поэтому выравнивание будет менее заметным. Напротив, при `\setupalign[horizontal,lesshyphenation]`останется больше пустого пространства, и выравнивание будет более заметным.

`\setupalign`предназначен для включения в преамбулу и влияет на весь документ или для включения в определенный момент и влияет на все от этого момента до конца. Если мы хотим изменить выравнивание только одной или нескольких строк, мы можем использовать:

– Среда “`alignment`”, предназначенная для воздействия на несколько строк. Имеет общий формат:

```
\startalignment[Options] ... \stopalignment
```

где *Options* любые из допустимых для `\setupalign`.

– Команды `\leftaligned`, `\midaligned` или `\rightaligned` которые вызывают выравнивание по левому, центру или правому краю соответственно; и если мы хотим, чтобы последнее слово в абзаце (но только это, а не остальная часть строки) было выровнено по правому краю, мы можем использовать `\wordright`. Все эти команды требуют, чтобы текст был заключен в фигурные скобки.

С другой стороны, обратите внимание, что если слова “right” и “left” в `\setupalign` вызывают выравнивание, противоположное тому, что предлагает название, то же самое не происходит с командами texleftaligned и `\righ-taligned`, которые вызывают именно тот вид выравнивания, который предлагает их название: `left` налево и `right` направа.

11.6.2 Вертикальное выравнивание

Если горизонтальное выравнивание вступает в игру, когда ширина строки не занимает все доступное для нее пространство, вертикальное выравнивание влияет на высоту всей страницы: если точная *exact* высота текста страницы не занимает всю высоту, доступную для это, что нам делать с оставшимся пустым пространством? Мы можем сложить его вверху (“`height`”), что означает, что текст на странице будет сдвинут вниз; мы можем сложить его внизу (“`bottom`”) или распределить по абзацам(“`line`”). Значение по умолчанию для вертикального выравнивания – “`bottom`”.

Вертикальный уровень допуска

Таким же образом мы можем изменить уровень допуска ConTEXt в отношении количества горизонтального пространства, допустимого в строке (горизонтальный допуск) с помощью `\setuptolerance`, мы также можем изменить его вертикальный допуск, то есть допуск на расстояние между абзацами больше, чем у ConTEXt, по умолчанию считается разумным для хорошо набранной страницы. Возможные значения для вертикального допуска такие же, как и для горизонтального допуска: `verystrict, strict, tolerant` and `verytolerant`. Значение по умолчанию – `\setuptolerance [vertical, strict]`.

Контроль над вдовами и сиротами (widows and orphans)

Одним из аспектов, косвенно влияющих на вертикальное выравнивание, является контроль над вдовами и сиротами. Оба явления означают, что разрыв страницы приводит к тому, что одна строка абзаца оказывается изолированной на другой странице от остальной части абзаца. Это не считается типографически подходящим. Если строка, которая отделена от остального абзаца, является первой на странице, мы говорим о вдове *widowed line*; если строка, отделенная от абзаца, является последней на странице, то речь идет о потерянной строке *orphaned line*.

По умолчанию ConTEXt не реализует элемент управления, гарантирующий, что эти строки не появятся. Но мы можем изменить это, изменив некоторые внутренние переменные ConTEXt: `\wi-dowpenalty` контролирует овдовевшие строки, а `\clubpenalty` контролирует потерянные строки. Таким образом, следующие положения в преамбуле к нашему документу обеспечат выполнение этого контроля:

```
\widowpenalty=10000
\clubpenalty=10000
```

Выполнение этого элемента управления означает, что ConTEXt не будет вставлять разрыв страницы, который отделяет первую или последнюю строку абзаца от страницы, на которой находится остальная часть. Это избегание будет более или менее строгим в зависимости от значения, которое мы присваиваем переменным. При значении 10 000, таком как в примере, управление будет абсолютным; со значением, например, 150, элемент управления будет не быть таким строгим, и иногда могут быть некоторые овдовевшие или осиротевшие строки, когда альтернатива хуже в типографских терминах.

Глава 12

Специальные конструкции и абзацы

12.1 Сноски и концевые примечания

Примечания - это “второстепенные текстовые элементы, используемые для различных целей, таких как разъяснение или расширение основного текста, предоставление библиографической ссылки на источники, включая цитаты, ссылки на другие документы или изложение значения текста”[*Libro de Estilo de la Lengua española* (Spanish Language Style Guide), p. 195]. Они особенно важны в текстах академического характера. Их можно разместить в разных местах на странице или в документе. Сегодня наиболее распространены те, которые расположены в конце страницы (поэтому они называются сносками); иногда они также располагаются на одном из полей (примечания на полях), в конце каждой главы или раздела или в конце документа (примечания). В особо сложных документах также могут быть разные серии примечаний: примечания автора, примечания переводчика, обновления и т. Д. В частности, в критических редакциях прибор для примечаний может стать довольно сложным, и лишь несколько систем набора могут его поддерживать. ConTEXt – одно из них. Для создания и настройки различных типов заметок доступно множество команд.

Чтобы объяснить это, полезно начать с указания различных элементов, которые могут быть включены в заметку:

– *Mark* или *anchor* заметки: знак, помещаемый в тексте, чтобы указать, что с ним связана заметка. Не все типы примечаний имеют привязку *anchor*, связанную с ними, но когда она есть, эта привязка *anchor* появляется в двух местах: в той точке основного текста, к которой относится примечание, и в начале самого текста примечания. Наличие одинаковой отметки в обоих местах позволяет связать заметку с основным текстом.

– *ID или идентификатор* заметки: буква, цифра или символ, который идентифицирует заметку и отличает ее от других заметок. У некоторых заметок, например заметок на полях, может отсутствовать идентификатор. Когда это не так, идентификатор обычно совпадает с привязкой *anchor*.

> Если мы будем думать исключительно о сносках, мы не увидим разницы между тем, что я только что назвал ссылкой *reference mark*, и идентификатором *id*. Мы ясно видим разницу в других типах заметок: например, у линейных заметок есть id, но нет ссылки (reference mark).

- *Текст* or *содержимое* заметки, всегда расположенные в другом месте на странице или в документе, чем команда, которая создает заметку и указывает ее содержимое.

- *Label* Ярлык, связанный с примечанием, которое не отображается в окончательном документе, но позволяет нам обратиться к нему и получить его идентификатор в другом месте документа.

12.1.1 ТТипы заметок в ConTEXt и связанные с ними команды

В ConTEXt есть различные типы заметок. На данный момент я лишь перечислю их, описав их в общих чертах и предоставив информацию о командах, которые их генерируют. Позже я разработаю первые два:

- **Footnotes:** Сноски: несомненно, самые популярные, поскольку для всех типов заметок принято называть сноски *footnotes*. Сноски представляют собой метку *mark* с идентификатором *id* заметки в том месте документа, где находится команда, и вставляют текст самой заметки в нижней части страницы, где появляется метка. Они создаются с помощью команды `\footnote`

- **Endnotes:** Эти примечания, которые создаются с помощью команды `\endnote`, вставляются в то место в документе, где обнаружена отметка с идентификатором примечания; но содержимое заметки вставляется в другую точку документа, и вставка производится другой командой (`\placenotes`).

- **Margin notes:** Как следует из их названия, они написаны на полях текста, и в теле документа нет идентификатора или автоматически сгенерированной метки или привязки. Две основные команды (но не единственные), которые их создают, - это `\inmargin` и `\margintext`.

- **Line notes:** Тип примечания, типичный для сред, в которых строки нумеруются, например, в случае `\startlinenumbering ... \stoplinenumbering` (см. section ??). Примечание, которое обычно пишется внизу, относится к определенному номеру строки. Они создаются с помощью команды `\lnenote`, которая настраивается с помощью `\setuplinenote`. Эта команда не печатает *mark* в теле текста, но в самом примечании печатает номер строки, на которую ссылается примечание (используется как *ID*).

Сейчас я буду развивать только первые два типа заметок:

- Примечания на полях рассматриваются в другом месте (section 5.7).

- Строчные заметки имеют узкоспециализированное использование (особенно в критических редакциях), и я считаю, что во вводном документе, подобном этому, читателю достаточно знать, что они существуют.

  Тем не менее, для заинтересованного читателя я рекомендую видео (на испанском) с текстом (также на испанском) о критических редакциях в ConTEXt, автором которых является Пабло Родригес. Он доступен по адресу эта ссылка. Это также очень полезно для понимания некоторых общих настроек заметок в целом.

12.1.2 Внимательный взгляд на сноски и концевые сноски

Синтаксис команд сносок и концевых сносок, а также имеющиеся у них механизмы конфигурации и настройки весьма схожи, поскольку на самом деле оба типа примечаний являются частными экземплярами более общей конструкции (заметок), другие экземпляры которой могут быть установлены с помощью Команда `\Definenote`(см. section ??).

Синтаксис команды, создающей каждый из этих видов заметок, следующий:

```
\footnote[Label]{Text}
\endnote[Label]{Text}
```

где

– *Label* необязательный аргумент, который присваивает заметке метку, которая позволит нам ссылаться на нее в другом месте документа.

– *Text* это содержание заметки. Он может быть сколь угодно длинным и включать в себя специальные абзацы и настройки, хотя следует отметить, что когда дело доходит до сносок, правильный макет страницы довольно затруднен в документах с обильными и чрезмерно длинными примечаниями.

В принципе, в тексте заметки можно использовать любую команду, которую можно использовать в основном тексте. Однако мне удалось убедиться, что определенные конструкции и символы, которые не создают никаких проблем в основном тексте, действительно вызывают ошибку компиляции, когда они встречаются в тексте заметки. Эти случаи я обнаружил во время тестирования, но никак не организовал их.

Когда аргумент *Label* использовался для установки метки для заметки, команда `\note` позволяет нам получить идентификатор рассматриваемой заметки. Эта команда печатает ID заметки, связанной с этикеткой, которую она принимает в качестве аргумента в документе. Так, например:

```
Humpty Dumpty\footnote[humpty]{Probably the
best-known English nursery rhyme character}
sat on a wall, Humpty Dumpty\note[humpty]
had a great fall.\\
All the king's horses and
all the king's men Couldn't put
Humpty\note[humpty] together again
```

Humpty Dumpty[1] sat on a wall,
Humpty Dumpty[1] had a great fall.
All the king's horses and all the king's men
Couldn't put Humpty[1] together again

[1] Probably the best-known English nursery rhyme character

Основное различие между `\footnote` и `\endnote` - это место, где появляется примечание:

`\footnote` As a rule, it prints the note text at the bottom of the page on which the command is located, so that the note mark and its text (or the beginning of the text, if it is to be spread over two pages) will appear on the same page. To do this, ConTEXt will make the necessary adjustments in typesetting the page by calculating the space required by the location of the note at the bottom of the page.

Но в некоторых средах `\footnote` вставляет текст примечания не внизу самой страницы, а под средой. Это имеет место, например, в таблицах или в среде `columns`. В этих случаях, если мы хотим, чтобы примечания внутри среды располагались внизу страницы, вместо `\footnote` мы должны использовать команду `\footnotetext` в сочетании с командой `\note`. упомянутый выше. Первый, который также поддерживает метку в качестве необязательного аргумента, печатает только текст примечания, но не метку. Но поскольку `\note` печатает только метку без текста, их комбинация позволяет нам разместить заметку в том месте, где мы хотим. Так, например, мы могли бы написать `\note [MyLabel]` в таблице или среде с несколькими столбцами, а затем, выйдя из этой среды, `\footnotetext[MyLabel]{Note text}`.

Другой пример использования `\footnotext` в сочетании с `\note` - это примечания внутри других примечаний. Например:

```
This%
\footnote{or this\note[noteB], if you prefer.}%
\footnotetext[noteB]
{or possibly even this one\note[noteC].}
\footnotetext[noteC]{could be something
entirely different.}
is a sentence with nested notes.
```

This[2] is a sentence with nested notes.

[2] or this[3], if you prefer.
[3] or possibly even this[4].
[4] could be something entirely different.

`\endnote` печатает привязку заметки только в той точке исходного файла, где она расположена. Фактическое содержание заметки вставляется в другую точку документа с помощью другой команды (`\placenotes[endnote]`), которая в том месте, где она расположена, вставляет содержимое *all* концевые сноски документа (или рассматриваемой главы или раздела).

12.1.3 Местные заметки

Среда `\startlocalfootnotes` означает, что включенные в нее сноски считаются *local* примечаниями, а это означает, что их нумерация будет сброшена и содержимое примечаний

не будет автоматически вставлено вместе с остальной частью заметки, но только в том месте документа, где найдена команда \placelocalfootnotes, которая может находиться или не находиться в среде.

12.1.4 Creating and using customised types of notes

Мы можем создавать заметки особых типов с помощью команды \Definenote. Это может быть полезно в сложных документах, где есть примечания от разных авторов, или для разных целей, чтобы графически различать каждый из типов примечаний в нашем документе с помощью разного формата и другой нумерации.

Синтаксис \Definenote следующий:

```
\definenote[Name][Model][Configuration]
```

где

- *Name* это имя, которое мы присваиваем нашему новому типу заметок.
- *Model* это модель примечания, которая будет использоваться изначально. Это может быть `footnote` или `endnote`; в первом случае наша модель заметок будет работать как сноски, а во втором случае как концевые сноски, хотя для вставки их в документ мы бы не использовали \placenotes[endnote], но \placenotes[Name] (имя, которое мы присвоили этим видам заметок).

  Теоретически этот аргумент является необязательным, хотя в моих тестах некоторые заметки, созданные без него, не были видны, и у меня не хватило терпения выяснить, в чем причина.

- *Configuration* - необязательный второй аргумент, который позволяет нам отличать наш новый тип заметок от его модели: либо путем установки другого формата, либо другого типа нумерации, либо и того, и другого.

  Согласно официальному списку команд ConTEXt (см. section ??) настройки, которые могут быть предоставлены при создании заметки нового типа, основаны на тех, которые могут быть предоставлены позже с помощью \setupnote. Однако, как мы вскоре увидим, на самом деле есть две возможные команды для настройки заметок: \setupnote и \setupnotation. Поэтому я думаю, что предпочтительнее опустить этот аргумент при создании типа заметки, а затем настроить наши новые заметки с помощью соответствующих команд. По крайней мере, это легче объяснить.

Например, следующий элемент создаст новый тип заметки под названием ‘BlueNote’, который будет похож на сноски, но его содержимое будет напечатано жирным шрифтом и синим цветом:

```
\definenote  [BlueNote] [footnote]
\setupnotation
  [BlueNote]
  [color=blue, style=bf]
```

После того, как мы создали новый тип заметки, например *BlueNote* команда, позволяющая нам использовать его, будет доступна. В нашем примере это будет \BlueNote, синтаксис которого будет похож на \footnote:

```
\BlueNote[Label]{Text}
```

12.1.5 Настройка примечаний

Конфигурация примечаний (сноски или концевые сноски, примечания, созданные с помощью \Definenote, а также строчные примечания, созданные с помощью \lnenote), достигается двумя командами: \setupnote и \setupnotation[1]. Синтаксис для обоих похож:

[1] \setupnote имеет 35 параметров конфигурации *direct* и 45 дополнительных параметров, унаследованных от \setupframed; \setupnotation имеет 45 параметров прямой конфигурации и еще 23, унаследованных от \setupcounter. Поскольку эти параметры не задокументированы, и, хотя для многих из них мы можем определить их полезность по их названию, нам необходимо

```
\setupnote[NoteType][Configuration]
\setupnotation[NoteType][Configuration]
```

где *NoteType* относится к типу заметки, которую мы настраиваем (`footnote`, `endnote` или имя некоторого типа заметки, который мы сами создали), а *configuration* содержит конкретную конфигурацию параметры для команды.

Проблема в том, что названия этих двух команд не очень ясно показывают, в чем разница между ними или что каждая настраивает; и тот факт, что многие параметры этих команд не задокументированы, также не сильно помогает. После долгого тестирования я не смог прийти к какому-либо выводу, который позволил бы мне понять, почему одни вещи настроены с одним, а другие с другим, [1] За исключением того, что, возможно, из-за того, что я сделал выбор, чтобы заставить ее работать, `\setupnotation` всегда влияет на текст заметки или идентификатор, который печатается с текстом примечания, а в `\setupnote` есть некоторые параметры, которые влияют на отметку для примечания, вставленного в основной текст.

Теперь я попытаюсь упорядочить то, что я узнал после проведения некоторых тестов с различными вариантами обеих команд. Я оставляю большинство вариантов для обоих в стороне, поскольку они не задокументированы, и я не смог сделать никаких выводов относительно того, для чего они предназначены или при каких условиях их следует использовать:

– **ID используемый для отметки:**

  Примечания всегда обозначаются номером. Здесь мы можем настроить:

  ⋆ *Первая цифра*: контролируется `start` в `\setupnotation`. Значение должно быть целым числом, и оно используется для начала подсчета заметок.

  ⋆ *Система нумерации* Система нумерации, которая зависит от `numberconversion` опции для `\setupnotation`. Значениями могут быть:

    ⊳ *A Арабские цифры*: `n, N` или `numbers`.
    ⊳ *Римские цифры*: `I, R, Romannumerals, i, r, romannumerals`. Первые три - это римские цифры в верхнем регистре, а последние три - в нижнем регистре.
    ⊳ *Нумерация буквами*: `A, Character, Characters, a, character, characters`в зависимости от того, хотим ли мы, чтобы буквы были в верхнем регистре (первые три опции) или в нижнем регистре (остальные).
    ⊳ *Нумерация словами*. Другими словами, мы пишем слово, обозначающее число, и, например, ‘3’ становится ‘three’. Возможны два метода. Параметр `Words` записывает слова в верхнем регистре, а `words` в нижнем регистре.

проверить, верна наша интуиция или нет; а также принимая во внимание, что многие из этих опций допускают ряд значений, и все они должны быть протестированы ... Вы увидите, что для того, чтобы написать это объяснение, мне пришлось провести довольно много тестов; и хотя сделать тест быстро, делать множество тестов медленно и скучно. Поэтому я надеюсь, что читатель простит меня, если я скажу вам, что помимо двух общих команд настройки для заметок, которые я упоминаю в основном тексте и на которых я сосредоточусь в следующем объяснении, я оставлю еще четыре потенциальных возможности настройки в объяснение:

- `\setupnotes` и `\setupnotations`: Другими словами, то же имя, но во множественном числе. В вики говорится, что версия команды в единственном и множественном числе синонимичны, и я этому верю.

- `\setupfootnotes` и `\setupendnotes`: Мы предполагаем, что это конкретные приложения для сносок и концевых сносок соответственно. Возможно, объяснить конфигурацию заметок на основе этих команд было бы проще, поскольку я не смог заставить работать первый вариант (`numberconversion`), который я пробовал с `\setupfootnotes`, хотя я знаю, что другие параметры из этих команд действительно работают ... Мне было лень добавлять тесты, необходимые для включения этих двух команд, в объяснение многих тестов, которые мне уже пришлось сделать, чтобы написать то, что следует.

  Но я придерживаюсь мнения (исходя из нескольких выполненных мной случайных тестов), что все, что работает в этих двух командах, но чье объяснение я опускаю, также работает в командах, для которых я даю пояснения.

[1] Есть страница в ConTEXt wiki, которую я обнаружил случайно (поскольку он не предназначен специально для заметок), что говорит о том, что разница в том, что `\setupnotation` управляет текстом вставляемой заметки и `\setupnote` средой заметки, в которой она будет помещена (?). Но это несовместимо с тем фактом, что, например, ширина текста заметки (которая имеет отношение к его *вставке*) управляется параметром `width` в `\setupnote`, а не параметром `\setupnotation` с тем же именем. Здесь контролируется ширина промежутка между меткой и текстом заметки.

- ⊳ *Нумерация символами.*: мы можем использовать четыре различных набора символов в зависимости от выбранной опции: `set 0, set 1, set 2` о `set 3`. На page 89 есть пример символов, используемых в каждой из этих опций.

* *Событие, определяющее перезапуск нумерации заметок*: Это зависит от параметра `way` в `\setupnotation`. Если значение равно `bytext`, все примечания в документе будут пронумерованы последовательно без сброса нумерации. Когда это `bychapter, bysection, bysubsection и т.д.`, Счетчик заметок будет сбрасываться каждый раз при изменении главы, раздела или подраздела, а когда это `byblock`, он сбрасывает нумерацию каждый раз, когда мы меняем блоки в макроструктуре документа (см. section 6.6). Значение `bypage` заставляет счетчик заметок перезапускаться при каждом изменении страницы.

– **Настройка метки примечания:**

* Показывать это или нет: `number` опция для `\setupnotation`.

* Размещение метки по отношению к тексту примечания: опция `alternative` в `\setupnotation`:может принимать любое из следующих значений: `left, inleft, leftmargin, right, inright, rightmargin, inmargin, margin, innermargin, outermargin, serried, hanging, top, command`.

* Формат маркировки в самой заметке: опция `numbercommand` в `\setupnotation`.

* Формат маркировки в теле текста: опция `textcommand` в `\setupnote`.

  Опции `numbercommand` and `textcommand` должны состоять из команды, которая принимает содержимое метки в качестве аргумента. Это может быть самоопределенная команда. Однако я обнаружил, что простые команды форматирования (`\bf`, `\it` и т.д.) работают, хотя они не являются командами, которые должны принимать аргумент.

* Расстояние между меткой и текстом (в самой заметке): параметры `distance` и `width` в `\setupnotation`. Мне не удалось обнаружить разницу (если она действительно есть) между использованием того или иного варианта.

* Наличие или отсутствие гиперссылки, позволяющей переходить между отметкой в основном тексте и отметкой в самой заметке: Опция `interaction` в `\setupnote`. При `yes` в качестве значения будет ссылка, а при `no` не будет.

– **Настройка самого текста заметки.**

Мы можем влиять на следующие аспекты:

* Размещение: зависит от опции `location` в `\setupnote`.

  В принципе, мы уже знаем, что сноски помещаются внизу страницы (`location = page`), а концевые сноски в том месте, где команда `\placenotes [endnote]` (`location=text`) найден, однако мы можем настроить эту функцию и установить сноски, например, как `location=text`, что приведет к тому, что сноски будут работать так же, как концевые сноски, поэтому они появятся в том месте в документе, где `\placenotes[footnote]` найдена команда или специальная команда для сносок `\placefootnotes`. С помощью этой процедуры мы могли бы, например, распечатать заметки под абзацем, в котором они находятся.

* Разделение абзацев между примечаниями: по умолчанию каждая примечание печатается в отдельном абзаце, но мы можем распечатать все примечания на одной странице в одном абзаце, установив для параметра `paragraph` в `\setupnote` значение “`yes`”.

* Стиль, в котором будет написан сам текст заметки: опция `style` в `\setupnotation`.

* Размер букв: опция `bodyfont` в `\setupnote`.

  Эта опция предназначена только для случая, когда мы хотим вручную установить размер шрифта для сносок. Это почти никогда не бывает хорошей идеей, поскольку по умолчанию ConTEXt регулирует размер шрифта сносок так, чтобы он был меньше основного текста, но с размером, *пропорциональным* размеру шрифта в основном тексте.

  - Левое поле для текстовой заметки: опция `margin` в `\setupnotation`.
  - Максимальная ширина: опция `width` в `\setupnote`.
  - Количество столбцов: опция `n` в `\setupnote` определяет, что текст заметки будет в двух или более столбцах. Значение ‘n’ должно быть целым числом.

- **Пространство между заметками или между заметками и текстом:** тут возможны следующие опции:
  - опция `rule` в `\setupnote` устанавливает будет ли линия между областью заметки и областью страницы с основным текстом. Тут возможны значения `yes, on, no` и `off`. Первые два включают линию, а последний отключает её.
  - `before` в `\setupnotation`: команда или комнады, которые должны быть выполнены перед вставкой текста примечания. Служит для вставки дополнительных интервалов, разделительных линий между заметками и т.д.

12.1.6 Временное исключение заметок при компиляции

Команды `\notesenabledfalse` и `\notesenabledtrue` сообщают ConTEXt, что нужно включить или отключить компиляцию заметок соответственно. Эта функция может быть полезна, если мы хотим получить версию без примечаний, когда в документе есть многочисленные и обширные примечания. По моему личному опыту, например, когда я правлю докторскую диссертацию, то предпочитаю прочитать её сначала за один раз, без заметок, а затем прочесть повторно с включенными заметками.

12.2 Абзацы с несколькими столбцами

Набор текста более чем в одном столбце - это возможность, которая может быть установлена:

a. Как общая особенность макета страницы.

b. Как особенность определенных конструкций, таких как, например, структурированные списки, сноски или концевые сноски.

c. Как функция, применяемая к определенным абзацам в документе.

В любом из этих случаев большинство команд и сред будут работать идеально, даже если мы работаем с более чем одним столбцом. Однако есть некоторые ограничения; в основном по отношению к плавающим объектам в целом (см. section 13.1) и таблицам в частности (section 13.3), даже если они не являются плавающими.

Что касается количества разрешенных столбцов, то ConTEXt не имеет теоретических ограничений. Однако необходимо учитывать физические ограничения:

- Ширина бумаги: неограниченное количество столбцов требует неограниченной ширины бумаги (если документ должен быть распечатан) или экрана (если это документ, предназначенный для отображения на экране). На практике, принимая во внимание *обычную* ширину форматов бумаги, которые продаются и используются для изготовления книг, а также экранов компьютерных устройств, текст, состоящий из более чем четырех или пяти столбцов, трудно уместить.
- Размер памяти компьютера: в справочном руководстве ConTEXt указано, что в *обычных* системах (не особо мощных и не особо ограниченных в ресурсах) можно обрабатывать от 20 до 40 столбцов.

В этом разделе я остановлюсь на использовании многоколоночного механизма в специальных абзацах или фрагментах, поскольку

- Несколько столбцов как вариант макета страницы уже обсуждались (в subsection B и section 5.3.4).
- Возможность, предлагаемая некоторыми конструкциями, такими как структурированные списки или сноски, набор текста более чем в одном столбце, обсуждается применительно к рассматриваемой конструкции или среде.

12.2.1 Среда `\startcolumns`

Обычная процедура для вставки в документ фрагментов, состоящих из нескольких столбцов, заключается в использовании среды столбцов, формат которой:

```
\startcolumns[Configuration] ... \stopcolumns
```

где *Configuration* позволяет нам контролировать многие аспекты среды. Мы можем указывать желаемую конфигурацию каждый раз, когда вызываем среду, или адаптировать работу среды по умолчанию для всех вызовов среды, последнее должно быть достигнуто с помощью

```
\setupcolumns[Configuration]
```

В обоих случаях параметры конфигурации одинаковы. Наиболее важные из них, упорядоченные в соответствии с их функциями, следующие:

- **Параметры, управляющие количеством столбцов и расстоянием между ними:**
  - `n`: контролирует количество столбцов. Если это не указано, будут созданы два столбца.
  - `nleft, nright`: эти параметры используются в двустороннем макете документа (см. subsection A of section 5.3.4), чтобы установить количество столбцов на левой (четной) и правой (нечетной) страницах соответственно.
  - `distance`: пространство между колонками.
  - `separator`: определяет, что отмечает разделение между столбцами. Это может быть `space` - пробел (значение по умолчанию) или `rule` - линия, в этом случае между столбцами будет создана линия. В случае, если между столбцами установлена линия, это, в свою очередь, можно настроить с помощью следующих двух опций:
    - `rulecolor`: цвет линий.
    - `rulethickness`: толщина линий.
  - `maxwidth`: максимальная ширина столбцов + расстояние между ними.
- **Параметры, управляющие распределением текста по столбцам:**
  - `balance`: по умолчанию ConTEXt *балансирует* столбцы, то есть распределяет текст между ними так, чтобы в них было примерно одинаковое количество текста. Однако мы можем установить эту опцию с “`no`”, текст не будет начинаться в столбце, пока предыдущий не будет заполнен.
  - `direction`: определяет, в каком направлении текст распределяется между столбцами. По умолчанию соблюдается естественный порядок чтения (слева направо), но при выборе этой опции `reverse` приводит к порядку справа налево.
- **Параметры, влияющие на набор текста в среде:**
  - `tolerance`: текст, записанный более чем в одном столбце, означает, что ширина строки в столбце меньше, и, как объяснялось при описании механизма, который ConTEXt использует для построения строк (section 11.3), это затрудняет поиск оптимальных точек для вставки

разрывов строк. Эта опция позволяет нам временно изменять горизонтальный допуск в среде (см. section 11.3.3), чтобы облегчить набор текста.

* `align`: контролирует горизонтальное выравнивание линий в среде. Может принимать любое из следующих значений: `right, flushright, left, flushleft, inner, flushinner, outer, flushouter, middle` or `broad`. Относительно значения этих параметров см.section 11.6.1.

* `color`: указывает название цвета, которым будет написан текст в окружении.

12.2.2 Параллельные абзацы

Конкретный вариант многоколоночной композиции - параллельные абзацы. В этом типе конструкции текст распределяется по двум столбцам (обычно, хотя иногда и более чем по двум), но ему не разрешается свободно перемещаться между ними, и вместо этого поддерживается строгий контроль над тем, что будет отображаться в каждом столбце. Это очень полезно, например, для создания документов, которые противопоставляют две версии текста, такие как новая и старая версия недавно измененного закона, или в двуязычных изданиях; или также написать глоссарии для определенных текстовых определений, где текст, который должен быть определен, появляется слева, а определение справа, и т. д.

Обычно мы используем механизм таблиц для обработки таких параграфов; но это связано с тем, что большинство текстовых процессоров не так мощны, как ConTEXt, в котором есть команды `\defineparagraphs` и `\setupparagraphs`, которые создают этот тип абзаца с использованием механизма столбцов, который, хотя и имеет ограничения, более гибкий, чем механизм таблиц.

> Насколько мне известно, у этих абзацев нет особого названия. Я назвал их «параллельными абзацами», потому что мне кажется, что это более описательный термин, чем тот, который ConTEXt использует для их обозначения: “*paragraphs*”.

Основная команда здесь `\defineparagraphs`, синтаксис которой:

```
\defineparagraphs[Name][Configuration]
```

где *Name* – это имя, которое мы даем этой конструкции, а *Configuration* – это функции, которые она будет иметь, которые также можно установить позже с помощью

```
\setupparagraphs[Name][Column][Configuration]
```

where *Name* is the name given when creating it, *Column* is an optional argument allowing us to configure a particular column, and *Configuration* allows us to determine how it works in practice.

это *Name*, указанное при его создании, *Column* - это необязательный аргумент, позволяющий нам настроить конкретный столбец, а *Configuration* позволяет нам определить, как он работает на практике.

Например:

```
\defineparagraphs
  [MurciaFacts]
  [n=3, before={\blank},after={\blank}]

\setupparagraphs
  [MurciaFacts][1]
  [width=.1\textwidth, style=bold]

\setupparagraphs
  [MurciaFacts][2]
  [width=.4\textwidth]
```

Приведенный выше фрагмент создаст среду с тремя столбцами под названием MurciaFacts, а затем установит, что первый столбец будет занимать 10 процентов ширины строки и будет

выделен жирным шрифтом, а второй столбец будет занимать 40 процентов ширины строки. Поскольку третий столбец не настроен, он будет иметь оставшуюся ширину, то есть 50%.

Как только среда будет создана, мы можем использовать ее, чтобы написать краткую историю Murcia:

```
\startMurciaFacts
  825
\MurciaFacts
  City of Murcia founded.
\MurciaFacts
  The origins of the city of Murcia are uncertain, but there is evidence
  that it was ordered to be founded under the name of Madina (or Medina)
  Mursiya in the year 825 by the Emir of al-Àndalus Abderramán II,
  probably built over a much earlier settlement.
\stopMurciaFacts
```

825 City of Murcia founded.

The origins of the city of Murcia are uncertain, but there is evidence that it was ordered to be founded under the name of Madina (or Medina) Mursiya in the year 825 by the Emir of al-Àndalus Abderramán II, probably built over a much earlier settlement.

Если бы мы хотели продолжить рассказ о Мурсии, для следующего события потребовался бы новый экземпляр среды (`\startMurciaFacts`), потому что с помощью этого механизма невозможно включить несколько строк *rows*.

Из только что приведенного примера я хотел бы выделить следующие детали:

- После создания среды, скажем, с помощью `\defineparagraphs[MaryPoppins]`, она становится нормальной средой, которая начинается с `\startMaryPoppins` и заканчивается `\stopMaryPoppins`.
- Также создается команда `\MaryPoppins`, которая используется в среде, чтобы указать, когда следует изменить столбец.

Что касается параметров конфигурации для параллельных абзацев (`\setupparagraphs`), я понимаю, что на этом этапе введения и с учетом только что приведенного примера читатель уже подготовлен к определению назначения каждого из вариантов, поэтому ниже я буду указывать только название и тип опций и, где это уместно, возможные значения. Однако помните, что `\setupparagraphs [Name] [Configuration]` устанавливает конфигурации, которые влияют на всю среду, в то время как `\setupparagraphs [Name] [NumColumn] [Configuration]` применяет конфигурацию исключительно к указанному столбцу.

- `n`: Number
- `before`: Command
- `after`: Command
- `width`: Dimension
- `distance`: Dimension
- `align`: Derived from `\setupalign`
- `inner`: Command
- `rule`: on off
- `rulethickness`: Dimension
- `rulecolor`: Rule colour
- `style`: Style Command
- `color`: Colour

Приведенный выше список опций не является полным; Я исключил из списка вариантов те, которые я обычно здесь не объясняю. Я также воспользовался тем фактом, что мы находимся в разделе, посвященном столбцам, чтобы показать список параметров в тройных столбцах, хотя я не делал этого ни с одной из команд, описанных в этом разделе, а с опцией `columns` в среде `itemize`, которой посвящен следующий раздел.

12.3 Структурированные списки

Когда информация представлена упорядоченно, читателю легче ее понять. Но если расположение также заметно визуально, то это подчеркивает для читателя тот факт, что здесь у нас есть структурированный текст. Вот почему существуют определенные *конструкции* или *механизмы*, которые пытаются выделить визуальное расположение текста, тем самым способствуя его структурированию. Из инструментов, которые ConTEXt предоставляет автору

для этой цели, наиболее важным из них, о котором идет речь в этом разделе, является среда `itemize`, которая используется для разработки того, что мы могли бы назвать *структурированными списками*.

Списки состоят из последовательности *текстовых элементов* (которые я буду называть *элементами*), каждому из которых предшествует символ, который помогает выделить его, отличая его от остальных, и который я буду называть «разделителем». Разделителем может быть цифра, буква или символ. Обычно (но не всегда) *элементы* являются абзацами, и список форматируется так, чтобы обеспечить *видимость* разделителя для каждого элемента; обычно с помощью выступа, который выделяет его [1]. В случае *вложенных списков* отступ для каждого постепенно увеличивается. Язык HTML обычно вызывает списки, в которых разделителем является число или символ, которые последовательно увеличиваются, упорядоченные списки, что означает, что каждый элемент списка будет иметь другой разделитель, который позволит нам ссылаться на каждый элемент по его номеру или идентификатору; и дает имена неупорядоченным спискам, в которых один и тот же символ или символ используется для каждого элемента в списке.

ConTEXt автоматически управляет нумерацией или алфавитным порядком разделителя в нумерованных списках, а также отступами, которые должны иметь вложенные списки; и, в случае вложения неупорядоченных списков, он также следит за выбором другого символа или символа, который позволяет сразу различать уровень элемента *item* в списке в соответствии с символом, который ему предшествует.

> В справочнике написано, что максимальный уровень вложенности в списки – 4, но я предполагаю, что так было в 2013 году, когда было написано руководство. Согласно моим тестам, кажется, что нет предела вложенности упорядоченных списков (в моих тестах я достиг до 15 уровней вложенности). В то время как для неупорядоченных списков, похоже, также нет предела в том смысле, что независимо от того, сколько вложений мы включаем, никаких ошибок генерироваться не будет; но для неупорядоченных списков ConTEXt применяет символы по умолчанию только для первых девяти уровней вложенности.
>
> В любом случае следует указать, что чрезмерное использование вложенности в списки может иметь эффект, противоположный тому, что мы намеревались сделать, а именно: читатель чувствует себя потерянным, неспособным найти каждый элемент в общей структуре списка. По этой причине я лично считаю, что, хотя списки являются мощным инструментом для структурирования текста, почти никогда не стоит выходить за пределы третьего уровня вложенности; и даже третий уровень следует использовать только в определенных случаях, когда мы можем это оправдать.

Общий инструмент для написания списков в ConTEXt - это среда `\itemize`, синтаксис которой следующий:

```
\startitemize[Options][Configuration] ... \stopitemize
```

где два аргумента являются необязательными. Первый позволяет использовать в качестве содержимого символические имена, которым ConTEXt присвоил точное значение; второй аргумент, который используется редко, позволяет присвоить определенные значения определенным переменным, которые влияют на функционирование среды.

12.3.1 Выбор типа списка и разделителя между *элементами*

A. Неупорядоченные списки

По умолчанию список, созданный с помощью `itemize`, представляет собой неупорядоченный список, в котором разделитель будет автоматически выбран в зависимости от уровня вложенности:

- Для первого уровня вложенности.
  - Для второго уровня вложенности.
    - Для третьего уровня вложенности.
      - Для четвертого уровня вложенности.
        - Для пятого уровня вложенности.
          - Для шестого уровня вложенности.

[1] В типографике отступ, который применяется ко всем строкам абзаца, кроме первой, называется *висячим отступом*, который позволяет легко найти первое слово или символ абзаца.

● Для седьмого уровня вложенности.
□ Для восьмого уровня вложенности.
✓ Для девятого уровня вложенности.

Однако мы можем прямо указать, что мы хотим использовать символ, связанный с определенным уровнем, просто передав номер уровня в качестве аргумента. Таким образом, `\startitemize[4]` сгенерирует неупорядоченный список, в котором символ ▷ будет использоваться в качестве разделителя, независимо от уровня вложенности списка.

Мы также можем изменить предопределенный символ для каждого уровня с помощью `\definesymbol`:

```
\definesymbol[Level]{Symbol associated with the level}
```

Например:

```
\definesymbol[1][\diamond]
```

заставит первый уровень неупорядоченных списков использовать символ ⋄C помощью этой же команды мы можем назначить некоторые символы уровням вложенности выше девяти. Так, например,

```
\definesymbol[10][\copyright]
```

присвоит международный символ авторских прав: *copyright* уровню вложенности 10.

B. Упорядоченные списки

Чтобы сгенерировать упорядоченный список, нам нужно указать `itemize` тип упорядочивания, который мы хотим

n	1, 2, 3, 4, ...	**a**	a, b, c, d, ...	**r**	i, ii, iii, iv, ...
m	1, 2, 3, 4, ...	**A**	A, B, C, D, ...	**R**	I, II, III, IV, ...
g	α, β, γ, δ, ...	**KA**	A, B, C, D, ...	**KR**	I, II, III, IV, ...
G	A, B, Γ, Δ, ...				

Разница между `n` и `m` заключается в шрифте, используемом для представления числа: `n` использует шрифт, включенный в этот момент, тогда как `m` использует другой, более элегантный, почти каллиграфический шрифт.

> Я не знаю название шрифта, который использует `m`, и поэтому в приведенном выше списке я не смог точно представить тип чисел, генерируемых этой опцией. Предлагаю читателям проверить это на себе.

12.3.2 Введение элементов в список

Как правило, элементы в списке, созданном с помощью `\startitemize`, вводятся с помощью команды `\item`, которая также имеет версию в форме среды, которая больше подходит для стиля Mark IV: `\startitem ... \stopitem`. Таким образом, следующий пример:

```
\startitemize[a, packed]
\startitem Первый элемент \stopitem
\startitem Второй элемент \stopitem
\startitem Третий элемент \stopitem
\stopitemize
```

a. Первый элемент
b. Второй элемент
c. Третий элемент

даст такой же результат как:

```
\startitemize[a, packed]
\item Первый элемент
\item  Второй элемент
\item Третий элемент
\stopitemize
```

a. Первый элемент
b. Второй элемент
c. Третий элемент

`\item` или `\startitem` это *общая* команда для добавления элемента в список. Наряду с этим есть следующие дополнительные команды, когда мы хотим добиться особого результата:

`\head` Эту команду следует использовать вместо `\item`, когда мы не хотим вставлять разрыв страницы после рассматриваемого элемента.

> Распространенной конструкцией является включение вложенного списка или текстового блока непосредственно под элементом списка, так что элемент списка, в некотором смысле, функционирует как *title* заголовок подсписка или текстового блока. В этих случаях разрыв страницы между этим элементом и последующими абзацами не рекомендуется. Если мы используем `\head` вместо `\item` для ввода этих элементов, ConTEXt (*постарается* насколько это возможно) не отделять такой элемент от следующего блока.

`\sym` The `\sym{Text}` команда вводит элемент, в котором текст, используемый в качестве аргумента `\sym`, используется как *разделитель*, а не число или символ. Этот список, например, состоит из элементов, вводимых с помощью `\sym` вместо `\item`.

`\sub` Эта команда, которую следует использовать только в упорядоченных списках (где каждому элементу предшествует число или буква в алфавитной последовательности), заставляет вводимый с ним элемент сохранять номер предыдущего элемента и для указания того, что номер повторяется, и это не ошибка, слева напечатан знак ‘+’. Это может быть полезно, если мы ссылаемся на предыдущий список, для которого мы предлагаем изменения, но где для ясности следует сохранить нумерацию исходного списка.

`\mar` Эта команда сохраняет нумерацию элементов, но добавляет букву или символ в поле (которое передается ей в качестве аргумента в фигурных скобках). Я не совсем понимаю, насколько это полезно.

Есть две дополнительные команды для ввода элементов, комбинация которых дает очень *интересные* эффекты, и, если можно так выразиться, я думаю, что лучше объяснить их на примере. `\ran` (сокращение от *range*) и `\its`, сокращение от *items*. Первый принимает аргумент (в фигурных скобках), а второй повторяет символ, используемый в качестве разделителя в списке, x количество раз (по умолчанию 4 раза, но мы можем изменить это, используя параметр `items`). В следующем примере показано, как эти две команды могут работать вместе для создания списка, имитирующего анкету:

```
After reading the following introduction, answer the following questions:

\startitemize[5, packed][width=8em, distance=2em, items=5]
\ran{No \hss Yes}
\its I will never use \ConTeXt, it is too difficult.
\its I will only use it for writing big books.
\its I will always use it.
\its I like it so much I will call my next child \quotation{Hans}, as a tribute to Hans Hagen.
\stopitemize
```

After reading this introduction, answer the following questions:

No Yes

- ◦ ◦ ◦ ◦ ◦ I will never use ConTEXt, it is too difficult.
- ◦ ◦ ◦ ◦ ◦ I will only use it for writing big books.
- ◦ ◦ ◦ ◦ ◦ I will always use it.
- ◦ ◦ ◦ ◦ ◦ I like it so much I will call my next child “Hans”, as a tribute to Hans Hagen.

12.3.3 Базовая конфигурация списка

Напомним, что “`itemize`” допускает два аргумента. Мы уже видели, как первый аргумент позволяет нам выбрать нужный тип списка. Но мы также можем использовать его для указать другие характеристики списка; это делается с помощью следующих опций для “`itemize`” в его первом аргументе:

- `columns`: этот параметр определяет, что список состоит из двух или более столбцов. После опции столбцов желаемое количество столбцов должно быть записано в виде слов, разделенных запятой: два, три, четыре, пять, шесть, семь, восемь или девять. `Columns`, за которыми нет числа, дают два столбца.

- `intro`: эта опция пытается не отделять список разрывом строки от предшествующего абзаца.

- `continue`: в упорядоченных списках (числовых или алфавитных) эта опция заставляет список продолжать нумерацию с последнего пронумерованного списка. Если опция `continue` используется, необязательно указывать, какой тип списка мы хотим, так как предполагается, что он будет таким же, как последний пронумерованный список.

- `packed`: это один из наиболее часто используемых вариантов. Его использование приводит к уменьшению вертикального расстояния между различными *items* элементами в списке, насколько это возможно.

- `nowhite:` производит эффект, похожий на `packed`, но более радикальный: он не только уменьшает вертикальное пространство между элементами, но также и вертикальное пространство. между списком и окружающим текстом.

- `broad`: увеличивает горизонтальное пространство между разделителем элементов и текстом элемента. Пространство можно увеличить, умножив число на `broad`, например, для пример `\startitemize[2*broad]`.

- `serried`: убирает горизонтальное пространство между разделителем элементов и текстом.

- `intext`: убирает отступ.

- `text`: убирает отступ и уменьшает расстояние между элементами по вертикали.

- `repeat`: во вложенных списках нумерация дочернего уровня *повторяет* тот же уровень, что и предыдущий уровень. Таким образом, на первом уровне у нас будет: 1, 2, 3, 4; на втором уровне: 1.1, 1.2, 1.3 и т. д. Опция должна быть указана для внутреннего списка, а не для внешнего списка.

- `margin, inmargin`: по умолчанию разделитель списка печатается слева, но внутри самой текстовой области (`atmargin`). Параметры `margin` и `inmargin` перемещают разделитель на поле.

12.3.4 Дополнительная настройка списка

Второй аргумент, также необязательный, в `\startitemize` позволяет более детально и тщательно настраивать списки.

- `before, after`: команды, выполняемые перед запуском или после закрытия среды itemize соответственно.

- `inbetween`: команда, выполняемая между двумя элементами `items`.

- `beforehead, afterhead`: команда, запускаемая до или после ввода элемента с помощью команды `\head` command.

- `left, right`: символ, печатаемый слева или справа от разделителя. Например, чтобы получить алфавитный список, в котором буквы заключены в круглые скобки, мы должны написать:

  ```
  \startitemize[a][left=(, right=)]
  ```

- `stopper`: указывает символ, который нужно написать после разделителя. Работает только в упорядоченных списках.

- `width, maxwidth`: ширина пространства, зарезервированного для разделителя и, следовательно, для отступа.
- `factor`: репрезентативное число коэффициента разделения между разделителем и текстом.
- `distance`: мера расстояния между разделителем и текстом.
- `leftmargin, rightmargin, margin`: margin to be added to the left (leftmargin) or right (rightmargin) of the items.
- `start`: number from which the numbering of items will start.
- `symalign, itemalign, align`: выравнивание элементов. Допускает те же значения, что и `\setupalign`. `symalign` контролирует выравнивание разделителя; `itemalign` текст элемента и `align` выравнивание обоих.
- `identing`: отступ первой строки в абзацах в среде. См. section 11.1.1.
- `indentnext`: указывает, должен ли абзац после окружения иметь отступ или нет. Значения *yes*, *no* и `auto`.
- `items`: в элементах, введенных как input с `\its`, указывает, сколько раз должен быть воспроизведен разделитель.
- `style, color; headstyle, headcolor; marstyle, marcolor; symstyle, symcolor`: эти параметры управляют стилем и цветом элементов при их вводе в среду с помощью команд `\item`, `\head`, `\mar` or `\sym` commands.

12.3.5 Простые списки с помощью команды `\items`

Альтернативой среде `itemize` для очень простых ненумерованных списков, где элементы не слишком большие, является команда `\items`, синтаксис которой:

```
\items[Configuration]{Item 1, Item 2, ..., item n}
```

Различные элементы, которые будет в списке, отделяются друг от друга запятыми. Например:

```
Graphics files can
have, among other things, the
following extensions:

\items{png, jpg, tiff, bmp}
```

Graphics files can have, among other things, the following extensions:

- png
- jpg
- tiff
- bmp

Параметры конфигурации, поддерживаемые этой командой, являются подмножеством параметров `itemize`, за исключением двух конкретных параметров для этой команды:

- `symbol`: этот параметр определяет тип создаваемого списка. Он поддерживает те же значения, которые используются для `itemize`, чтобы выбрать какой-либо тип списка.
- `n`: эта опция указывает, от какого номера позиции будет разделитель.

12.3.6 Предварительное определение поведения списка и создание наших собственных типов списков

В предыдущих разделах мы видели, как указать, какой тип списка нам нужен и какие характеристики он должен иметь. Но делать это каждый раз, когда вызывается список,

неэффективно и обычно приводит к бессвязному документу, в котором каждый список имеет свой собственный внешний вид, но без другого внешнего вида, отвечающего каким-либо критериям.

Предпочтительный результат для этого:

– Заранее определите *нормальное* поведение элементов `itemize` и `\items` в преамбуле документа.

– Создавайте собственные индивидуальные списки. Например: список с алфавитной нумерацией, который мы хотим назвать *ListAlpha*, список, пронумерованный римскими цифрами (*ListRoman*) и т.д.

Первого мы достигаем с помощью команд `\setupitemize` и `\setupitems`. Второй требует использования команды`\defineitemgroup` или `\defineitems`. Первый создаст среду списка, аналогичную `itemize`, а второй - команду, аналогичную `items`.

12.4 Описания и перечисления

Описания и перечисления - это две конструкции, которые позволяют согласованно набирать абзацы или группы абзацев, которые развивают, описывают или определяют фразу или слово.

12.4.1 Descriptions

Для описаний мы различаем заголовок *title* и его объяснение или развитие. Мы можем создать новое описание с помощью:

```
\definedescription[Name] [Configuration]
```

где *Name* – это имя, под которым будет известна эта новая конструкция, а Configuration управляет тем, как будет выглядеть наша новая структура. После предыдущего оператора у нас будет новая команда и среда с выбранным нами именем. Таким образом:

```
\definedescription[Concept]
```

создаст команды:

```
\Concept{Title}
\startConcept {Title} ... \stopConcept
```

Мы будем использовать команду для случая, когда пояснительный текст заголовка состоит только из одного абзаца, и среду для заголовков, описание которых занимает более одного абзаца. При использовании команды следующий за ней абзац будет считаться пояснительным текстом заголовка. Но когда используется среда, весь контент будет отформатирован с соответствующим отступом, чтобы было понятно, как оно соотносится с заголовком.

Например:

```
\definedescription
  [Concept]
  [alternative=left, width=1cm, headstyle=bold]

\Concept{Contextualise}

Place something in a certain context, or typeset a text with the typesetting system called \ConTeXt. The ability
to correctly contextualise in any situation is considered a sign of intelligence and good sense.
```

Это даст следующий результат:

Contextualise Place something in a certain context, or typeset a text with the typesetting system called ConTEXt. The ability to correctly contextualise in any situation is considered a sign of intelligence and good sense.

Как обычно бывает с ConTEXt, внешний вид нашей новой конструкции может быть указан во время ее создания с помощью аргумента *Configuration* или позже с помощью `\setupdescription`:

```
\setupdescription[Name] [Configuration]
```

где *Name* это имя нашего нового описания, а *Configuration* определяет, как оно выглядит. Среди различных возможных вариантов конфигурации выделю:

- `alternative`: эта опция принципиально контролирует внешний вид конструкции. Он определяет размещение заголовка по отношению к его описанию. Возможные значения: left, right, inmargin, inleft, inright, margin, leftmargin, rightmargin, innermargin, outermargin, serried, зависает, их названия достаточно ясны, чтобы получить представление о результате, хотя, в случае сомнения, это лучше сделать тест, чтобы посмотреть, как это выглядит.
- `width`: управляет шириной поля, в котором будет написан заголовок. В зависимости от значения `alternative` это расстояние также будет частью отступа, с которым написан пояснительный текст.
- `distance`: контролирует расстояние между заголовком и пояснением.
- `headstyle, headcolor, headcommand`: влияют на то, как будет выглядеть сам заголовок: Style (`headstyle`) и color (`headcolor`). С помощью headcommand мы можем указать команду, которой текст заголовка будет передан в качестве аргумента. Например: Например: `headcommand=\WORD` будет следить за тем, чтобы текст заголовка был в верхнем регистре.
- `style, color`: управляет внешним видом описательного текста заголовка.

12.4.2 Перечисления

Перечисления представляют собой пронумерованные текстовые элементы, структурированные на нескольких уровнях. Каждый элемент начинается с заголовка, который по умолчанию состоит из имени структуры и ее номера, хотя мы можем изменить заголовок с помощью опции `text`. Они очень похожи на описания, но отличаются тем, что:

- Все элементы в перечислении имеют одинаковый заголовок.
- Поэтому они отличаются друг от друга своей нумерацией.

Эта структура может быть очень полезной, например, для написания формул, задач или упражнений в учебнике, обеспечивая их правильную нумерацию и единообразный формат. Создаем перечисление с помощью

```
\defineenumeration[Name] [Configuration]
```

где *Name* это имя новой конструкции, а *Configuration* определяет, как она будет выглядеть.

Итак, в следующем примере:

```
\defineenumeration
  [Exercise]
  [alternative=top, before=\blank, after=\blank, between=\blank]
```

Мы создали новую структуру под названием *Exercise*, и, как и в случае с перечислениями, у нас будут доступны следующие новые команды:

```
\Exercise
\startExercise
```

Команда используется только для *упражнений* с одним абзацем и среда для*упражнений* с несколькими абзацами. Но поскольку перечисления могут иметь до четырех уровней глубины, также будут созданы следующие команды и среды:

```
\subExercise
\startsubExercise
\stopsubExercise
\subsubExercise
\startsubsubExercise
\stopsubsubExercise
\subsubsubExercise
\startsubsubsubExercise
\stopsubsubsubExercise
```

А для управления нумерацией используются следующие дополнительные команды:

– `\setEnumerationName`: устанавливает текущее значение нумерации.

– `\resetEnumerationame`: устанавливает счетчик перечисления на ноль.

– `\nextEnumerationName`: увеличивает счетчик перечисления на единицу.

Внешний вид перечислений можно определить во время их создания или позже с помощью `\setupenumeration`, формат которого:

`\setupenumeration[Name] [Configuration]`.

Для каждого перечисления мы можем настроить каждый его уровень отдельно. Так, например, `\setupenumeration [subExercise] [Configuration]` повлияет на второй уровень перечисления под названием “Exercise”.

Параметры и значения, настраиваемые с помощью `\setupenumeration`, аналогичны тем, что указаны в `\setupdescription`.

12.5 Линии и рамки

В справочном руководстве ConTEXt говорится, что TEX обладает огромными возможностями управления текстом, но очень слаб в управлении графической информацией. Хочу не согласиться: это правда, что для обработки строк и фреймов возможности ConTEXt (на самом деле TEX) не так велики, как когда дело доходит до набора текста. Но говорить о слабости системы в этом отношении, я думаю, будет натяжкой. Я не знаю ни одной функции с линиями и рамками, которые другие системы набора могут выполнять для документов, которые ConTEXt не может создать. А если объединить ConTEXt с MetaPost, или с TiKZ (в ConTEXt для этого есть модуль расширения), то возможности ограничиваются только нашим воображением.

Однако в следующих разделах я ограничусь объяснением того, как создавать простые горизонтальные и вертикальные линии и рамки вокруг слов, предложений или абзацев.

12.5.1 Простые линии

Самый простой способ нарисовать горизонтальную линию - использовать команду `\hairline`, которая генерирует горизонтальную линию, занимающую всю ширину обычной текстовой строки.

В строке, где находится строка, созданная с помощью `\hairline`, не может быть никакого текста. Чтобы сгенерировать строку, способную сосуществовать с текстом в одной строке, нам нужна команда `\thinrule`. Эта вторая команда будет использовать всю ширину линии. Следовательно, в изолированном абзаце он будет иметь тот же эффект, что и `\hairline`, в то время как в противоположном случае `\thinrule` будет производить такое же горизонтальное расширение, что и `\hfill` (см.section 10.3.3), но вместо заполнения горизонтального пространства с помощью пробел (как и `\hfill`), он заполняет его линией.

```
On the left\thinrule\\
\thinrule On the right\\
On both\thinrule sides\\
\thinrule centred\thinrule
```

On the left
On the right
On both sides
centred

С помощью команды\thinrules мы можем сгенерировать несколько строк. Например, \thinrules[n=2] будет генерировать две последовательные линии, каждая шириной нормальной линии. Строки, созданные с помощью \thinrules, также можно настроить либо при фактическом вызове команды, указав конфигурацию в качестве одного из ее аргументов, либо, как правило, с помощью \setupthinrules. Конфигурация включает толщину линии (rulethickness), ее цвет (color), цвет фона (background), межстрочный интервал (interlinespace) и т.д.

> Я оставлю ряд вариантов без объяснения причин. Читатель может ознакомиться с ними в setup-en.pdf (см. section ??). Некоторые варианты отличаются от других только нюансами (то есть между ними практически нет разницы), и я думаю, что читателю быстрее попытаться *увидеть* разницу, чем мне попытаться передать ее словами. Например: толщина линии, которую я только что сказал, зависит от параметра rulethickness. Но на это также влияют параметры height и depth.

Строки меньшего размера можно создать с помощью команд\hl и \vl. Первый создает горизонтальную линию, а второй – вертикальную. Оба принимают число в качестве параметра, который позволяет нам вычислить длину линии. В \hl число измеряет длину в *ems* (нет необходимости указывать единицу измерения в команде), а в \vl аргумент относится к текущей высоте строки.

Таким образом, \hl[3] генерирует горизонтальную линию в 3 ems, а \vl[3] генерирует вертикальную линию высотой, соответствующую трем линиям. Помните, что индикатор линейных размеров должен быть заключен между квадратными скобками, а не между фигурными скобками. В обеих командах аргумент не обязателен. Если он не введен, предполагается значение 1.

\fillinline – еще одна команда для создания горизонтальных линий точной длины. Он поддерживает больше настроек, в которых мы можем указать (или заранее определить с помощью \setupfillinlines опцию width) в дополнение к некоторым другим функциям.

Особенностью этой команды является то, что текст, который написан справа, будет помещен слева от строки, отделяя этот текст от строки необходимым пробелом, чтобы занять всю строку. Например:

```
\fillinline[width=6cm] Name
```

будет генерировать

Name

> Я подозреваю, что эта странная операция связана с тем, что этот макрос был разработан для записи форм, в которых за текстом есть горизонтальная линия, на которой должно быть написано что-то . Фактически само название команды fillinline означает заполнить строку.

Помимо ширины линии, мы можем настроить поле (margin), расстояние (distance), толщину (rulethickness) и цвет (color). \fillinline почти идентична \fillinrules, хотя эта команда позволяет нам вставить более одной строки (опция “n”).

```
\fillinrules[Configuration] {Text} {Text}
```

где три аргумента необязательны.

12.5.2 Строки, связанные с текстом

Хотя некоторые из команд, которые мы только что видели, могут генерировать строки, которые сосуществуют с текстом в одной строке, на самом деле эти команды сосредоточены на макете строки. Для написания строк, связанных с определенным текстом, в ConTEXt есть команды:

– которые генерируют текст между строками.

– генерирующие строки под текстом (подчеркивание - underlining), над текстом (перекрытие - overlining) или сквозь него (зачеркивание - strikethrough).

Чтобы сгенерировать текст между строками, обычно используется команда \textrule. Эта команда рисует линию, пересекающую всю ширину страницы, и записывает текст, который она принимает в качестве параметра, слева (но не на полях). Например:

```
\textrule{Example text}
```


Предполагается, что \textrule допускает необязательный первый аргумент с тремя возможными значениями: top, middle и bottom. Но после некоторых тестов мне так и не удалось выяснить, какой эффект производят такие варианты.

Подобно \texrule, есть среда \starttextrule, которая, помимо вставки строки с текстом в начале среды, вставляет горизонтальную строку в конце. Формат этой команды:

```
\starttextrule[Configuration]{Text on the line} ... \stoptextrule
```

И \textrule, и starttextrule можно сконфигурировать с помощью \setuptextrule.

Чтобы рисовать линии под текстом, над текстом или сквозь него, используются следующие команды:

```
\underbar{Text}
\underbars{Text}
\overbar{Text}
\overbars{Text}
\overstrike{Text}
\overstrikes{Text}
```

Как мы видим, для каждого типа строки (под, над или через текст) есть две команды. Версия команды в единственном числе рисует одну линию под, поверх или через весь текст, взятый в качестве аргумента, в то время как версия команды во множественном числе рисует линию только над словами, но не через пробел.

Эти команды несовместимы друг с другом, то есть две из них не могут применяться к одному и тому же тексту. Если мы попробуем, всегда будет преобладать последний. С другой стороны, \underbar может быть вложенным, подчеркивая то, что уже было подчеркнуто.

The reference manual points out that \underbar disables hyphenation of words in the text that constitute its argument. It is not clear to me whether that statement refers only to \underbar or to the six commands we are examining.

12.5.3 Слова или тексты в рамке

Чтобы окружить текст рамкой или сеткой, мы используем:

– Команды \framed or \inframed если текст относительно короткий и не занимает более одной строки.

– Среда \startframedtext предназначается для более длинных текстов.

Разница между \framed и \inframed заключается в точке, из которой отрисовывается рамка. В \frame рамка нарисована вверх от идеальной линии, называемой базовой линией, на которую опираются буквы, но некоторые буквы проходят вниз. При \inframed рамка отрисовывается также вверх от самой низкой точки на линии. Например:

```
Here there are \framed{two} good
\inframed{frames}.
```

Here there are two good frames.

И текст в рамке, и текст в фреймах можно настроить с помощью `\setupframed`, а `\startframedtext` - с помощью `\setupframedtext`. Варианты настройки для обеих команд очень похожи. Они позволяют нам указать размеры рамки (`height, width, depth`), форму углов (`framecorner`), которые могут быть `rectangular` или круглыми (`round`), цвет рамки (`framecolor`), толщина линии (`framethickness`), выравнивание содержимого (`align`), цвет текста (`foregroundcolor`), цвет фона (`background` и `backgroundcolor`) и т. д.

Для `\startframedtext` есть также явно странное свойство: `frame=off`, из-за которого рамка не отрисовывается (хотя она все еще там, но невидима). Это свойство существует потому, что, поскольку рамка вокруг абзаца неделима, обычно весь абзац помещается в среду `framedtext` с отключенным параметром рисования рамки, чтобы гарантировать, что в абзац не вставлены разрывы страниц.

Мы также можем создать индивидуальную версию этих команд с помощью `\defineframed` и `\defineframedtext`.

12.6 Другие среды и конструкции, представляющие интерес

В ConTEXt еще существует много сред, о которых я даже не упоминал или только очень мимоходом. В качестве примера:

- **`buffer`** *Buffers* - это фрагменты текста, хранящиеся в памяти для последующего повторного использования. *buffer* определяется где-то в документе с помощью `\startbuffer[BufferName] ... \stopbuffer` и может извлекаться сколько угодно часто в другом месте документа с помощью `\getbuffer[BufferName]`.

- **`chemical`**

  Эта среда позволяет нам помещать в нее химические формулы. Если TEX выделяется, среди прочего, своей способностью правильно набирать текст с математическими формулами, с самого начала ConTEXt стремился распространить эту способность на химические формулы, и у него есть среда, в которой команды и структуры разрешены для написание химических формул.

- **`combination`**

  Эта среда позволяет нам объединять несколько плавающих элементов на одной странице. Это особенно полезно для выравнивания различных подключенных внешних изображений в нашем документе.

- **`formula`** Это среда, предназначенная для набора математических формул.

- **`hiding`** скрытие текста, хранящегося в этой среде, не будет скомпилировано и, следовательно, не появится в окончательном документе. Это полезно для временного отключения компиляции определенных фрагментов в исходном файле. То же самое достигается путем выделения одной или нескольких строк в качестве комментария. Но когда фрагмент, который мы хотим отключить, относительно длинный, более эффективным, чем пометка десятков или сотен строк исходного файла в качестве комментария, является вставка команды `\starthiding` в начале фрагмента и `\stophiding` в конце.

- **`legend`** В математическом контексте TEX применяет другие правила, чтобы обычный текст не мог быть написан. Однако иногда формула сопровождается описанием используемых в ней

элементов. Для этой цели существует среда `\startlegend`, которая позволяет нам помещать обычный текст в математический контекст.

- **linecorrection** Обычно ConTEXt правильно управляет вертикальным пространством между строками, но иногда строка может содержать что-то, из-за чего она выглядит неправильно. Это происходит в основном со строками, фрагменты которых заключены в рамку с помощью `\framed`. В таких случаях эта среда корректирует межстрочный интервал таким образом, чтобы абзац отображался правильно.

- **mode** Эта среда предназначена для включения фрагментов в исходный файл, которые будут скомпилированы только в том случае, если активен соответствующий режим. Использование *modes* не является предметом этого введения, но это очень интересный инструмент, если мы хотим иметь возможность создавать несколько версий в разных форматах из одного исходного файла. Дополнительной средой к этой является `\startnotmode`.

- **opposite** эта среда используется для набора текста, когда содержимое левой и правой страниц связано.

- **quotation** Очень похоже на среду `narrower`, предназначенную для вставки умеренно длинных цитат. Среда гарантирует, что текст внутри будет заключен в кавычки, а поля увеличены, чтобы абзац с цитатой визуально выделялся на странице. Но следует отметить, что в соответствии с обычным стилем блочных кавычек на английском языке не должно быть открывающих и закрывающих кавычек, что делает эту команду или среду менее полезной.

- **standardmakeup** Эта среда предназначена для создания страниц с заголовком документа, что относительно часто встречается в академических документах определенной длины, таких как докторские диссертации, магистерские диссертации и т.д.

Чтобы узнать о любой из этих сред (или других, которые я не упомянул), я предлагаю следующие шаги:

1. Информацию о среде ищите в справочном руководстве ConTEXt. В этом руководстве не упоминаются все среды; но в нем что-то говорится о каждом элементе в списке выше.

2. Напишите тестовый документ, в котором используется среда.

3. Найдите в официальном списке команд ConTEXt (см. section ??) параметры конфигурации для рассматриваемой среды, затем протестируйте их, чтобы точно увидеть что они делают.

Глава 13

Изображения, таблицы и другие плавающие объекты

Эта глава в основном посвящена плавающим объектам (поплавкам). Но, продолжая эту концепцию, он использует ее для объяснения двух типов объектов, которые не обязательно являются плавающими, хотя они часто настраиваются так, как если бы они были: внешние изображения и таблицы. Глядя на оглавление этой главы, можно подумать, что все это очень неаккуратно: сначала речь идет о плавающих объектах, затем идет речь об изображениях и таблицах, а в конце снова говорится о плавающих объектах. Причины такой неопрятности носят *педагогический характер*: изображения и таблицы можно объяснить, не слишком настаивая на том, что они обычно являются плавающими; и все же, когда мы начинаем их изучать, нам очень помогает обнаружить, что, к удивлению, мы уже знаем о двух плавающих объектах.

13.1 Что такое плавающие объекты и что они делают?

Если бы документ содержал только *обычный* текст, разбить его на страницы было бы относительно просто: знание максимальной высоты текстовой области страницы достаточно, чтобы измерить высоту различных абзацев, чтобы знать, куда вставлять разрывы страниц. Проблема в том, что во многих документах есть объекты, фрагменты или неделимые блоки текста, такие как изображение, таблица, формула, абзац в рамке и т. Д.

Иногда эти объекты могут занимать большую часть страницы, что, в свою очередь, создает проблему, заключающуюся в том, что если вам нужно вставить их в определенную точку документа, они могут не поместиться на текущей странице и должны быть внезапно прерваны, оставляя большое пустое пространство внизу, так что рассматриваемый объект и текст, следующий за ним, перемещаются на следующую страницу. Однако правила хорошего набора указывают, что, за исключением последней страницы главы, на каждой странице должно быть одинаковое количество текста.

Поэтому рекомендуется избегать появления больших пустых вертикальных пространств; а плавающие *floating* объекты - главный механизм для достижения этого. «Плавающий объект» “floating object” – это объект, который не обязательно должен располагаться в определенной

точке документа, но может *перемещаться* или *плавать* вокруг него. Идея состоит в том, чтобы позволить ConTEXt выбрать лучшее место с точки зрения нумерации страниц для размещения таких объектов, даже разрешив им перейти на другую страницу; но всегда стараюсь не отходить слишком далеко от точки включения в исходный файл.

Следовательно, нет объектов, которые должны быть *как таковые*. Но есть объекты, которые иногда нужно будет поплавать. В любом случае решение остается за автором или лицом, отвечающим за набор, если это два разных человека.

Несомненно, возможность изменения точного размещения неделимого объекта очень облегчает задачу набора хорошо сбалансированных страниц; но проблема, связанная с этим, заключается в том, что, поскольку мы не знаем точно, где окажется такой объект во время написания оригинала, на него трудно ссылаться. Так, например, если я только что поместил в свой документ команду, которая вставляет изображение, и в следующем абзаце я хочу описать это и написать что-нибудь об этом, например: «Как вы можете видеть из предыдущего рисунка», когда рисунок *floats* вполне может быть размещен *after* после того, что я только что написал, и в результате возникает несогласованность: читатель ищет изображение *before* перед текстом, который на него ссылается, и не может его найти, потому что после перемещения изображение закончилось после этого ссылка.

Это фиксируется с помощью *нумерации* плавающих объектов (после распределения их по категориям), так что вместо ссылки на изображение как “предыдущее изображение” или “следующее изображение”, мы будем ссылаться на него как “изображение 1.3”, поскольку мы можем использовать внутренний механизм ссылок ConTEXt, чтобы гарантировать, что номер изображения всегда обновляется (см. section 8.2). С другой стороны, нумерация таких объектов упрощает создание их индекса (указатель таблиц, графиков, изображений, уравнений и т.д.). О том, как это сделать, см. (section 7.2).

Механизм работы с плавающими объектами в ConTEXt довольно сложен и временами настолько абстрактен, что может не сделать его подходящим для новичков. Поэтому в этой главе я начну с объяснения этого, используя два частных случая: изображения и таблицы. Затем я попытаюсь несколько обобщить, чтобы мы могли понять, как распространить этот механизм на другие типы объектов.

13.2 Внешние изображения

Как читатель на данном этапе знает (поскольку это было объяснено в section 1.5), ConTEXt отлично интегрирован с MetaPost и может генерировать изображения и графику, которые *запрограммированы* в большей части так же, как программируются преобразования текста. Существует также модуль расширения для ConTEXt footnoteConTEXt extension modules, предоставляет ему дополнительные утилиты, но не включены в это введение., Который позволяет ему работать с TiKZ.[1] Но такие изображения не рассматриваются в этом введении (поскольку это, вероятно, заставит его длину удвоиться) . Здесь я имею в виду использование внешних изображений, которые находятся в файле на нашем жестком диске или загружаются непосредственно из Интернета с помощью ConTEXt.

13.2.1 Прямая вставка изображений

Чтобы напрямую вставить изображение (не как плавающий объект), мы используем команду `\externalfigure`, синтаксис которой

[1] Это язык программирования графики, предназначенный для работы с системами на основе TEX. Это “рекурсивная аббревиатура” из немецкого предложения “TiKZ ist keinen Zeichenprogramm”, что в переводе означает: “TiKZ не является программой для рисования”. Рекурсивные сокращения - это своего рода шутка программистов. Оставляя в стороне MetaPost (который я не знаю, как использовать), я считаю, что TiKZ - отличная система для программирования графики.

```
\externalfigure[Name] [Configuration]
```

где

- *Name* может быть либо именем файла, содержащего изображение, либо веб-адресом изображения, найденного в Интернете, либо символическим именем, которое мы ранее связали с изображением с помощью команды `\useexternalfigure`, формат которой похож на `\externalfigure`, хотя он принимает первый аргумент с символическим именем, которое будет связано с рассматриваемым изображением.
- *Configuration* – необязательный аргумент, который позволяет нам применять определенные преобразования к изображению до того, как оно будет вставлено в наш документ. Мы рассмотрим этот аргумент более подробно в section 13.2.4.

Допустимые форматы изображений: pdf, mps, jpg, png, jp2, jbig, jbig2, jb2, svg, eps, gif или tif. ConTEXt может напрямую управлять восемью из них, в то время как остальные (svg, eps, gif или tif) необходимо преобразовать с помощью внешнего инструмента перед их открытием, который изменяется в соответствии с форматом и, следовательно, должен быть установлен в системе, чтобы ConTEXt может работать с такими файлами.

> Среди форматов, поддерживаемых `\externalfigure`, есть также несколько видеоформатов. В частности: QuickTime (расширение .mov), Flash Video (расширение .flv) и MPeg 4 (расширение .mp4). Но большинство проигрывателей PDF не знают, как обрабатывать файлы PDF со встроенным в них видео. Я не могу сказать много об этом, так как я не делал никаких тестов.

Нет необходимости указывать расширение файла: ConTEXt будет искать файл с указанным именем и одним из расширений для известных форматов изображений. Если кандидатов несколько, сначала используется формат PDF, если он есть, а при его отсутствии - формат MPS (графика, генерируемая MetaPost). В отсутствие этих двух используется следующий порядок: jpeg, png, jpeg 2000, jbig и jbig2.

> Если фактический формат изображения не соответствует расширению файла, в котором оно хранится, ConTEXt не сможет открыть его, если мы не укажем фактический формат изображения с помощью параметра `method`.

Если изображение не размещается само по себе за пределами абзаца, а интегрировано в текстовый абзац, и его высота больше, чем межстрочный интервал, строка будет скорректирована так, чтобы изображение не перекрывало предыдущие строки, как в пример, который

сопровождает эту строку .

По умолчанию ConTEXt ищет изображения в рабочем каталоге, в его родительском каталоге и в родительском каталоге этого каталога. Мы можем указать местоположение каталога, содержащего изображения, с которыми мы будем работать, используя параметр `directory`

команды \setupexternalfigures, который добавит этот каталог к пути поиска. Если мы хотим, чтобы поиск выполнялся только в каталоге изображений, мы также должны установить опцию location. Так, например, чтобы наш документ искал все нужные нам изображения в каталоге "img", мы должны написать:

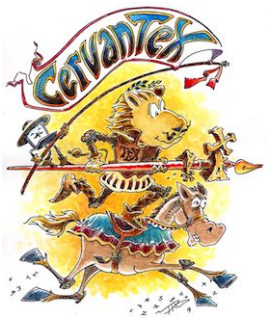
```
\setupexternalfigures
  [directory=img, location=global]
```

В параметре directory в \setupexternalfigures мы можем включить более одного каталога, разделяя их запятыми; но в этом случае нам нужно заключить каталоги в фигурные скобки. Например, "directory={img, ~/imágenes}".

В directory мы всегда используем символ '/' как разделитель между каталогами; в том числе в Microsoft Windows, операционная система которой использует '\' в качестве разделителя каталогов.

\externalfigure также может использовать изображения, размещенные в Интернете. Так, например, следующий фрагмент вставит логотип CervanTeX прямо из Интернета в документ. Это группа пользователей TEX, говорящая по-испански: [1]

```
\externalfigure
[http://www.cervantex.es/files/
cervantex/cervanTeXcolor-small.jpg]
```



Когда документ, содержащий удаленный файл, сначала компилируется, он загружается с сервера и сохраняется в каталоге кэша LuaTeX. Этот кешированный файл используется при последующих компиляциях. Обычно удаленный образ загружается снова, если образ в кеше старше 1 дня. Чтобы изменить этот порог, см. ConTEXt wiki.

Если ConTEXt не находит изображение, которое следует вставить, ошибка не генерируется, но вместо изображения будет вставлен текстовый блок с информацией об изображении, которое должно туда попасть. Размер этого блока будет размером изображения (если известен ConTEXt) или, в противном случае, стандартным размером. Пример этого есть в section ??.

13.2.2 Вставка изображений с помощью \placefigure

Изображения могут быть вставлены напрямую. Но предпочтительно делать это с помощью \placefigure. Эта команда вызывает ConTEXt:

- чтобы знать, что изображение вставляется, которое должно быть включено в список изображений в документе, который затем может быть использован, если мы хотим, для создания индекса изображений.
- присвоить изображению номер, облегчая тем самым внутренние ссылки на него.
- для добавления заголовка к изображению, создавая текстовый блок между изображением и его заголовком, что означает, что они не могут быть разделены.
- для автоматической установки пустого пространства (по горизонтали и вертикали), необходимого для правильного просмотра изображения.

[1] Интернет-адреса очень длинные, и не так много места для отображения примера с двумя столбцами. Поэтому, чтобы правильно разместить порядок в левом столбце, я вставил разрыв строки в веб-адрес. Если кто-то захочет скопировать и вставить пример, он не сработает, если этот разрыв строки не будет удален.

- для размещения изображения в указанном месте, при необходимости обтекая его текстом.
- преобразовать изображение в плавающий объект, если это возможно, с учетом его размера и местоположения. [1]

Синтаксис этой команды следующий:

```
\placefigure[Options][Label] {Title} {Image}
```

Различные аргументы имеют следующие значения:

- *Options* представляют собой набор указателей, которые обычно указывают, где разместить изображение. Поскольку эти параметры одинаковы в этой и других командах, я объясню их вместе позже (в section 13.4.1). Сейчас я буду использовать опцию `here` в качестве примеров. Он сообщает ConTEXt, что, насколько это возможно, он должен разместить изображение точно в том месте документа, где находится команда, которая его вставляет.
- *Label* - это текстовая строка для внутренней ссылки на этот объект, чтобы мы могли ссылаться на него (см. section 8.2).
- *Title* текст заголовка, добавляемый к изображению.
- *Image* это команда, которая вставляет изображение.

Например:

```
\placefigure
  [here]
  [fig:texknuth]
  {\TeX\ logo and photo of {\sc Knuth}}
  {\externalfigure[https://i.ytimg.com/vi/8c5Rrfabr9w/maxresdefault.jpg]}
```

Рисунок 13.1 TEX logo and photo of Knuth

Как мы видим в примере, при вставке изображения (что, кстати, было сделано непосредственно из изображения, размещенного в Интернете), есть некоторые изменения, касающиеся того, что происходит при прямом использовании команды `\externalfigure`. Добавляется вертикальное пространство, изображение центрируется и добавляется заголовок. Это *external* изменения, очевидные на первый взгляд. С внутренней точки зрения команда произвела и другие, не менее важные эффекты:

- Прежде всего, изображение было вставлено в “список изображений”, который ConTEXt поддерживает внутри для объектов, вставленных в документ. Это, в свою очередь, означает, что изображение появится в индексе изображений, который можно сгенерировать с помощью `\placelist [figure]` (см. section 7.2), хотя есть две специальные команды для генерации индекс изображения, которые

[1] Это последний мой вывод, учитывая, что среди вариантов размещения есть такие, как `force` или `split`, что противоречит истинному представлению о плавающем объекте.

`\placelistoffigures` or `\completelistoffigures`.

– Во-вторых, изображение было связано с меткой, которая была добавлена в качестве второго аргумента команды `\placefigure`, что означает, что с этого момента мы можем делать внутренние ссылки на него, используя эту метку (см. section 8.2).

– Наконец, изображение стало плавающим, что означает, что если для набора текста (разбивки на страницы) его нужно было переместить, ConTEXt изменит его размещение.

На самом деле `\placefigure`, несмотря на свое название, используется не только для вставки изображений. С его помощью мы можем вставить что угодно, включая текст. Однако текст или другие элементы, вставленные в документ с помощью `\placefigure`, будут обрабатываться *, как если бы они были изображением*, даже если это не так; они будут добавлены в список изображений, находящихся под внутренним управлением ConTEXt, и мы сможем применять преобразования, аналогичные тем, которые мы используем для изображений, такие как масштабирование или поворот, кадрирование и т. д. Таким образом, следующий пример:

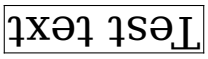


Рисунок 13.2 Using \placefigure for text transformations

which is achieved as follows:

```
\placefigure
  [here, force]
  [fig:testtext]
  {Using \backslash placefigure for text transformations}
  {\rotate[rotation=180]{\framed{\tfd Test text}}}
```

13.2.3 Вставка изображений, интегрированных в текстовый блок

За исключением очень маленьких изображений, которые можно интегрировать в строку без слишком большого нарушения интервала между абзацами, изображения обычно вставляются в абзац, который содержит только они (или, другими словами, изображение можно рассматривать как абзац в своем абзаце). владеть правом). Если изображение вставлено с помощью `\placefigure` и его размер позволяет, в зависимости от того, что мы указали относительно его размещения (см. section 13.4.1), ConTEXt разрешит текст из предыдущего и последующие абзацы обтекают изображение. Однако, если мы хотим гарантировать, что определенное изображение не будет отделено от определенного текста, мы можем использовать среду `figuretext`, синтаксис которой следующий:

```
\startfiguretext
  [Options]
  [Label]
  {Title}
  {Image}

  ... Text

\stopfiguretext
```

Аргументы среды точно такие же, как для `\placefigure`, и имеют то же значение. Но здесь варианты больше не варианты размещения плавающего объекта, а указания относительно интеграции изображения в абзац; так, например, “`left`” здесь означает, что изображение будет размещено слева, а текст - вправо, а “`left, bottom`” будет означать, что изображение должно быть размещено в нижнем левом углу. текста, связанного с ним.

The text written within the environment is what will flow around the image.

13.2.4 Конфигурация и преобразование вставленных изображений

A. Параметры команды вставки, вызывающие некоторую трансформацию изображения

Последний аргумент в команде `\externalfigure` позволяет нам выполнять определенные настройки вставленного изображения. Мы можем внести эти корректировки:

- Как правило, для вставки всех изображений в документ; или для вставки всех изображений из определенной точки. В этом случае мы выполняем настройку с помощью команды `\setupexternalfigures`.
- Для конкретного изображения, которое мы хотим вставить в документ несколько раз. В этом случае настройка выполняется в последнем аргументе команды `\useexternalfigure`, которая связывает внешнюю фигуру с символическим именем.
- В тот момент, когда мы вставляем конкретное изображение. В этом случае настройка выполняется в самой команде `\externalfigure`.

Изменения изображения, которые могут быть достигнуты этим путем, следующие:

Изменение размера изображения. Мы можем сделать это:

- *Назначив точную ширину или высоту*, что-то сделано с параметрами `width` и `height` соответственно; если регулируется только одно из двух значений, другое автоматически адаптируется для сохранения пропорции.

  Мы можем назначить точную высоту или ширину или указать ее в процентах от высоты страницы или ширины строки. Например:

  ```
  width=.4\textwidth
  ```

  обеспечит ширину изображения, равную 40 % от ширины линии.

- *Масштабирование изображения*: опция `xscale` масштабирует изображение по горизонтали; `yscale` будет делать это по вертикали, а `scale` будет делать это по горизонтали и вертикали. Эти три параметра предполагают, что число, представляющее коэффициент масштабирования, умноженное на 1000. То есть: `scale=1000` оставит изображение в исходном размере, а `scale=500` уменьшит его вдвое, а `scale=2000` удвоит его размер.

  Условное масштабирование, которое применяется только в том случае, если изображение превышает определенные размеры, достигается с помощью параметров `maxwidth` и `maxheight`, которые принимают размер. Например, `maxwidth=.2\textwidth` масштабирует изображение только в том случае, если его ширина превышает 20 % от ширины линии.

Вращение изображения. Чтобы повернуть изображение, мы используем параметр `orientation`, который принимает число, представляющее количество градусов поворота, которые будут применены. Вращение осуществляется против часовой стрелки.

> Вики подразумевает, что вращение, которое может быть достигнуто с помощью этой опции, должно быть кратным 90 (90, 180 или 270), но для достижения другого вращения нам нужно будет использовать команду `\rotate`. Однако у меня не было проблем с применением поворота на 45 градусов к изображению только с `orientation=45`, без необходимости использовать команду `\rotate`.

Обрамление изображения. Мы также можем окружить изображение рамкой, используя параметр `frame=on`, и настроить его цвет (`framecolor`), расстояние между рамкой и изображением (`frameoffset`), толщину линии, которая рисует рамку (`rulethickness`) или форму ее углов (`framecorner`), которая может быть закругленной (`round`) или прямоугольной.

Другие настраиваемые аспекты изображений. В дополнение к уже рассмотренным аспектам, которые подразумевают преобразование вставляемого изображения, с помощью `\setupexternalfigures` мы можем настроить другие аспекты, например, где искать изображение (параметр `directory`), независимо от того, изображение следует искать только в указанном каталоге (`location=global`) или должно ли оно также включать рабочий каталог и его родительские каталоги (`location=local`), и будет ли изображение быть интерактивным (`interaction`) и т.д.

B. Специальные команды для преобразования изображения

В ConTEXt есть три команды, которые производят некоторое преобразование изображения и могут использоваться в сочетании с `\externalfigure`. Эти:

- *Mirror image*: достигается с помощью команды `\mirror`.
- *Clipping*: это достигается с помощью команды `\clip`, где ширина (`width`), высота (`height`), размеры горизонтального (`hoffset`) и вертикального смещения заданы (`voffset`). Например:

```
\clip
  [width=2cm, height=1cm, hoffset=3mm, voffset=5mm]
  {\externalfigure[logo.pdf]}
```

- *Rotation*.

  Третья команда, способная применять преобразования к изображению, - это команда `\rotate`. Его можно использовать вместе с `\externalfigure`, но обычно в этом нет необходимости, учитывая, что последний имеет, как мы видели, параметр tt orientation, который дает тот же результат.

Обычно эти команды используются с изображениями, но на самом деле они действуют на *box*. Вот почему мы можем применить их к любому тексту, просто заключив его в рамку (что команда делает автоматически), что даст любопытные эффекты, подобные следующему:

```
\mirror{Test text}\\
\rotate[rotation=20] {Test text}
```

Tɘsʇ ʇɘxʇ

Test text

13.3 Tables

13.3.1 Общие представления о таблицах и их размещении в документе

Таблицы - это структурированные объекты, которые содержат текст, формулы или даже изображения, расположенные в серии *cells*, которые позволяют нам графически увидеть взаимосвязь между содержимым каждой ячейки. Для этого информация организована в строки и столбцы: все данные (или записи) в одной строке имеют определенное отношение друг к другу, как и все данные (или записи) в одном столбце. Ячейка - это пересечение строки со столбцом, как показано в figure 13.3.

Таблицы идеально подходят для отображения текста или данных, которые связаны друг с другом, потому что, поскольку каждая из них заблокирована в своей собственной ячейке, даже если ее содержимое или содержимое остальных ячеек изменяется, относительное положение одной по отношению к другим не изменится.

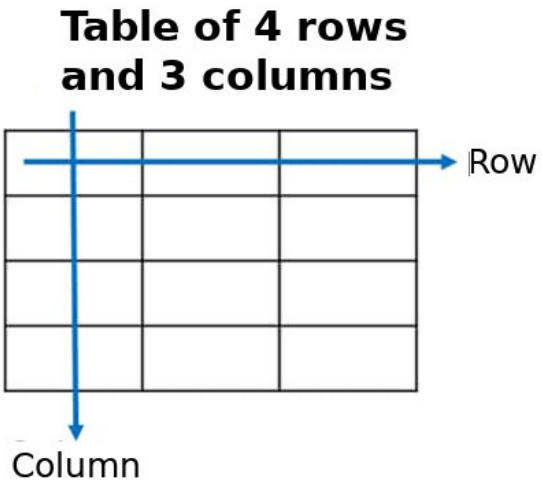


Рисунок 13.3 Image of a simple table

Из всех задач, связанных с набором текста, создание таблиц - единственная, которую, на мой взгляд, проще выполнить в графической программе (тип текстового процессора), чем в ConTEXt. Потому что проще *нарисовать* таблицу (это то, что вы делаете в текстовом редакторе), чем *описать её*, что мы и делаем, когда работаем с ConTEXt. Каждое изменение строки и столбца требует присутствия команды, а это означает, что реализация таблицы занимает гораздо больше времени, вместо того, чтобы просто указывать, сколько строк и столбцов мы хотим.

ConTEXt имеет три различных механизма для создания таблиц; среда `tabulate`, которая рекомендуется для простых таблиц и которая больше всего вдохновлена TEX tables; так называемые *естественные таблицы*, вдохновленные таблицами HTML, подходящие для таблиц с особыми требованиями дизайна, где, например, не все строки имеют одинаковое количество столбцов; и так называемые *крайние таблицы*, явно основанные на XML и рекомендуемые для средних или длинных таблиц, занимающих более одной страницы. Из трех я объясню только первое. Естественные таблицы достаточно хорошо объяснены в “ConTEXt Mark IV an excursion”, а для *крайних таблиц* есть документ о них в “ConTEXt Standalone” documentation.

Что-то похожее на то, что происходит с изображениями, происходит в таблицах: мы можем просто написать необходимые команды в некоторой точке документа, чтобы сгенерировать таблицу, и она будет вставлена именно в этом месте, или мы можем использовать `\placetable` команда для вставки таблицы. У этого есть некоторые преимущества:

– ConTEXt нумерует таблицу и добавляет ее в список таблиц, разрешая внутренние ссылки на таблицу (через ее нумерацию), включая ее в конечный индекс таблиц.

– Мы получим гибкость в размещении таблиц в документе, что облегчит задачу разбиения на страницы.

Формат `\placetable` аналогичен тому, который мы видели для `\placefigure` (см. section 13.2.2):

```
\placetable[Options][Label] {Title} {table}
```

Я обращаюсь к разделам13.4.1 и 13.4.2 относительно опций, относящихся к размещению таблицы и настройке заголовка. Однако среди вариантов есть один, который, похоже, предназначен исключительно для столов. Это параметр “`split`”, который, если установлен, разрешает ConTEXt расширять таблицу на две или более страниц, и в этом случае таблица больше не может быть плавающим объектом.

В общем, мы можем установить конфигурацию для таблиц с помощью команды `\setuptables`. Также, как и с изображениями, можно создать индекс таблиц с `\placelistoftables` или `\completelistoftables`. В связи с этим см. section 7.2.2.

13.3.2 Простые таблицы со средой `tabulate`

Самые простые таблицы - это те, которые создаются в среде *tabulate*, формат которой:

```
\starttabulate[Table column layout]
  ... % Table contents
  ...
  ...
\stoptabulate
```

Если аргумент, взятый в квадратных скобках, описывает (в коде) количество столбцов в таблице и (иногда косвенно) указывает их ширину. Я говорю, что аргумент описывает дизайн *в коде*, потому что на первый взгляд он кажется очень загадочным: он состоит из последовательности символов, каждый из которых имеет особое значение. Я буду объяснять это понемногу и поэтапно, потому что я думаю, что так легче понять.

> Это типичный случай, когда огромное количество аспектов, которые мы можем настроить, означает, что нам нужно много текста для его описания. Кажется, это чертовски сложно. На самом деле для большинства таблиц, которые строятся на практике, достаточно пунктов 1 и 2. Остальные - это дополнительные возможности, о существовании которых полезно знать, но не обязательно знать для набора таблицы.

1. **Columns delimiter**: символ “`|`” используется для разделения столбцов таблицы. Так, например,“`[|lT|rB|]`” будет описывать таблицу с двумя столбцами, один из которых будет иметь характеристики, связанные с индикаторами “`l`” и “`T`” (которые мы увидим сразу после), а второй столбец будет иметь характеристики, связанные с “`r`” и “`B`”. Например, простая таблица из трех столбцов, выровненная по левому краю, будет описана как “`[|l|l|l|]`”.

2. **Определение основной природы ячеек в столбце:** Первое, что нужно определить, когда мы создаем нашу таблицу, - это то, хотим ли мы, чтобы содержимое каждой ячейки было записано в одной строке, или, наоборот, если текст любого столбца слишком длинный, мы хотим, чтобы наша таблица распределяла его по двум или более строкам. В среде `tabulate` этот вопрос не решается ячейка за ячейкой, а считается характеристикой столбцов.

   a. *One line cells:* Если содержимое ячеек в столбце, независимо от их длины, должно быть записано в одной строке, мы должны указать выравнивание текста в столбце, который можно оставить (“`l`”, из *left*), right (“`r`”, from *right*) или по центру (“`c`”, из *center*).

      > В принципе, эти столбцы будут настолько широкими, насколько это необходимо, чтобы поместиться в самую широкую ячейку. Но мы можем ограничить ширину столбца с помощью спецификатора “`w(Width)`”. Например, “`[|rw(2cm)|c|c|]`” будет описывать таблицу с двумя столбцами, первый из которых выровнен по правому краю и имеет точную ширину 2 сантиметра, а два других по центру и без ограничения ширины.
      >
      > Следует отметить, что ограничение ширины однострочных столбцов может привести к тому, что текст в одном столбце будет перекрывать текст в следующем столбце. Поэтому я советую, когда нам нужны столбцы фиксированной ширины, всегда используйте многострочные столбцы ячеек.

   b. *Ячейки, которые при необходимости могут занимать более одной строки*: спецификатор “`p`” генерирует столбцы, в которых текст в каждой ячейке будет занимать столько строк, сколько необходимо. Если мы просто укажем “`p`”, ширина столбца будет равна всей доступной ширине. Но также можно указать “`p(Ширина)`”, и в этом случае ширина будет точно указанной. Таким образом, следующие примеры:

      ```
      \starttabulate[|l|r|p|]
      \starttabulate[|l|p(4cm)|]
      \starttabulate[|r|p(.6\textwidth)|]
      \starttabulate[|p|p|p|]
      ```

      В первом примере будет создана таблица с тремя столбцами, первая и вторая из одной строки, выровненные по левому и правому краю соответственно, и третий, который займет оставшуюся ширину и высоту, необходимые для размещения всего ее содержимого. Во втором примере ширина второго столбца будет ровно четыре сантиметра, независимо от его

содержания; но если он не умещается в этом пространстве, он займет более одной строки. В третьем примере ширина второго столбца вычисляется пропорционально максимальной ширине линии, а в последнем примере будет три столбца, ширина которых будет доступна в равных частях.

Обратите внимание, что в действительности, если ячейка является четырехугольником, спецификатор “p” разрешает переменную высоту для ячеек в столбце в зависимости от длины текста.

3. **Добавление указаний к описанию столбца, касающемуся стиля и варианта шрифта, который будет использоваться**: после того, как основной характер столбца (ширина и высота, автоматические или фиксированные, ячеек) определен, мы все еще можем добавьте в описание содержимого столбца символ, представляющий *формат*, в котором он должен быть записан. Этими символами могут быть “B” для полужирного шрифта, “I” для курсива, “S” для наклонного шрифта, “R” для букв в римском стиле или “T” для надписей в стиле *пишущей машинки*.

4. **Другие дополнительные аспекты, которые могут быть указаны в описании столбцов таблицы**

   :

   - *Столбцы с математическими формулами*: спецификаторы “m” и “M” включают математический режим в столбце без необходимости указывать его в каждой из его ячеек. Ячейки в этом столбце не могут содержать обычный текст.

     Хотя TEX, предшественник ConTEXt, появился для набора любых математических задач, до сих пор я почти ничего не говорил о написании математики. В математическом режиме (который я не буду объяснять) ConTEXt изменяет наши обычные правила и даже использует другие шрифты. Математический режим имеет две разновидности: один мы можем назвать *linear*, где формула помещается в строку, содержащую обычный текст (индикатор “m”), и режим *complete maths*, который отображает формулы в среда, в которой нет обычного текста. Основное различие между двумя режимами в таблице заключается в основном в размере, в котором будет записана формула, а также в горизонтальном и вертикальном пространстве вокруг нее.

   - *Добавление дополнительных горизонтальных пробелов вокруг содержимого ячеек в столбце*: с помощью индикаторов “in”, “jn” и “kn” мы можем добавить дополнительные пробелы слева от содержимое столбца (“in”) справа (“jn”) или по обе стороны (“kn”). Во всех трех случаях “n” представляет собой число, на которое нужно умножить пробел, который обычно остается без одного из этих спецификаторов (по умолчанию среднее значение равно *em*). Так, например, “|j2r|” будет указывать на то, что мы столкнулись со столбцом, который будет выровнен по правому краю и в котором нам нужно пустое пространство шириной 1 *em*.

   - *Добавление текста до или после содержимого каждой ячейки в столбце*. Спецификаторы b{Text} и a{Text} приводят к тому, что текст между фигурными скобками записывается до (“b”, от *before*) или после (“a”, от *after*) содержимого ячейки.

   - *Применение команды форматирования ко всему столбцу*. Указанные ранее индикаторы “B”, “I”, “S”, “R” “T” не охватывают все возможности форматирования: например, нет индикатора для маленьких заглавных букв, или для *sans serif*, или который влияет на размер шрифта. С помощью индикатора “f \Command” мы можем указать команду форматирования, которая автоматически применяется ко всем ячейкам в столбце. Например,“|lf\sc|” наберет содержимое столбца маленькими заглавными буквами.

   - *Применение любой команды ко всем ячейкам в столбце*. Наконец, индикатор “h\Command” применит указанную команду ко всем ячейкам в столбце.

В table 13.1 показаны некоторые примеры строк спецификации формата таблицы.

Спецификатор формата	Значение
`|l|`	Создает столбец, ширина которого автоматически выравнивается по левому краю.
`|rB|`	Создает столбец, ширина которого автоматически выравнивается по правому краю и выделяется жирным шрифтом.
`|cIm|`	Создает столбец для математического содержания. По центру и курсивом.
`|j4cb{---}|`	ТЭтот столбец будет иметь содержимое по центру, он будет начинаться с длинного тире (-) и добавит 2 пробела справа.
`|l|p(.7\textwidth)|`	генерирует два столбца: первый выравнивается по левому краю и имеет автоматическую ширину. Второй занимает 70

Таблица 13.1 Некоторые примеры того, как указать формат столбцов в `tabulate`

После того, как таблица была спроектирована, необходимо ввести ее содержимое. Чтобы объяснить, как это сделать, я начну с описания того, как должна быть заполнена таблица, где у нас есть линии, разделяющие строки и столбцы:

– Мы всегда начинаем с рисования горизонтальной линии. В таблице это делается с помощью команды `\HL` (от *Horizontal Line*).

– Затем мы пишем первую строку: в начале каждой ячейки мы должны указать, что начинается новая ячейка и что необходимо провести вертикальную линию. Это делается с помощью команды `\VL` (от *Vertical Line*). Итак, мы начинаем с этой команды и записываем содержимое каждой ячейки. Каждый раз, когда мы меняем ячейки, мы повторяем команду `\VL`.

– В конце строки мы явно указываем, что новая строка будет запущена с помощью команды `\NR` (от *Next Row*). После этого мы повторяем `\HL`, чтобы нарисовать новую горизонтальную линию.

– И так, одну за другой, записываем все строки таблицы. Когда мы закончим, мы добавим дополнительно команду `\NR` и еще одну команду `\HL`, чтобы закрыть сетку нижней горизонтальной линией.

Если мы не хотим рисовать сетку таблицы, мы удаляем команды `\HL` и заменяем команды `\VL` на `\NC` (из *New Column*).

Это не особенно сложно, когда мы разбираемся в этом, хотя, когда мы смотрим на исходный код таблицы, трудно понять, как она будет выглядеть. В table 13.2 мы видим команды, которые можно (и нужно) использовать в таблице. Некоторые из них я не объяснил, но думаю, что приведенного мной описания достаточно.

Команда	Значение
`\HL`	Вставляет горизонтальную линию
`\NC`	Начинает новый столбец
`\NR`	Начинает новую строку
`\VL`	Вставляет вертикальную линию, ограничивающую столбец (используется вместо `\NC`)
`\NN`	Начинает столбец в математическом режиме (используется вместо `\NC`)
`\TB`	Добавляет дополнительное вертикальное пространство между двумя рядами
`\NB`	Указывает, что следующая строка начинает неделимый блок, внутри которого не может быть разрыва страницы

Таблица 13.2 Команды, используемые в таблице

А теперь в качестве примера расскажу код, которым была написана table 13.2.

```
\placetable
  [here]
  [tbl:tablecommands]
  {Команды, используемые в таблице}
{\starttabulate[|l|p(.6\textwidth)|]
\HL
\NC {\bf Команда}
\NC {\bf Значение}
\NR
\HL
\NC \tex{HL}
\NC Вставляет горизонтальную линию
\NR
\NC \tex{NC}
\NC Начинает новый столбец
\NR
\NC \tex{NR}
\NCНачинает новую строку
\NR
\NC \tex{VL}
\NC Вставляет вертикальную линию, ограничивающую столбец (используется вместо \tex{NC})
\NR
\NC \tex{NN}
\NC Начинает столбец в математическом режиме (используется вместо \tex{NC})
\NR
\NC \tex{TB}
\NC Добавляет дополнительное вертикальное пространство между двумя рядами
\NR
\NC \tex{NB}
\NC Указывает, что следующая строка начинает неделимый блок, внутри которого не может быть разрыва страницы
\NR
\HL
\stoptabulate}
```

Читатель заметит, что в целом я использовал одну (или две) строки текста для каждой ячейки. В реальном исходном файле я бы использовал только строку текста для каждой ячейки; в этом примере я разделил слишком длинные строки. Использование одной строки в ячейке облегчает мне запись таблицы, потому что я пишу содержимое каждой ячейки без команд разделения строк или столбцов. Когда все написано, я выбираю текст из таблицы и прошу свой текстовый редактор вставлять “`\NC `” в начало каждой строки. После этого каждые две строки (поскольку в таблице два столбца) я вставляю строку, которая добавляет команду `\NR`, потому что каждые два столбца начинают новую строку. Наконец, вручную я вставляю команды `\HL` в те точки, где я хочу, чтобы появилась горизонтальная линия. На то, чтобы описать это, у меня уходит больше времени, чем на то, чтобы сделать это!

Но также посмотрите, как в таблице мы можем использовать обычные команды ConTEXt. В частности, в этой таблице мы постоянно используем `\tex`, что объясняется в section 10.2.3.

13.4 Общие аспекты изображений, таблиц и других плавающих объектов

Мы уже знаем, что изображения и таблицы не обязательно должны быть плавающими объектами, но они являются хорошими кандидатами на это, хотя они должны быть вставлены в документ с помощью команд `\placefigure` или `\placetable`. В дополнение к этим двум командам и с той же структурой в ConTEXt у нас есть команда `\placechemical` (для вставки химикатов в формулы), команда `\placegraphic` (для вставки графики) и команду `\placeintermezzo` для вставки структуры, которая ConTEXt вызывает *Intermezzo* и которая, как я подозреваю, относится к фрагментам текста во фреймах. Все эти команды, в свою очередь, являются конкретными приложениями более общей команды `\placefloat`, синтаксис которой следующий:

```
\placefloat[Name] [Options] [Label] {Title} {Contents}
```

Обратите внимание, что `\placefloat` идентичен `\placefigure` и `\placetable`, за исключением первого аргумента, который в `\placefloat` принимает имя плавающего объекта. Это потому, что *каждого типа плавающего объекта можно вставить в документ двумя разными командами*: `\placefloat[TypeName]` или `\placeTypeName`. Другими словами: `\placefloat[figure]` и `\placefigure` - это одна и та же команда, точно так же, как `\placefloat[table]` - это та же команда, что и `\placetable`.

Поэтому с этого момента я буду говорить о `\placefloat`, но учтите, что все, что я говорю, также применимо к `\placefigure` или `\placetable`, которые являются конкретными приложениями указанной команды.

Аргументы `\placefloat`:

– *Name*. относится к рассматриваемому плавающему объекту. Это может быть какой-то заранее определенный плавающий объект (`figure, table, chemical, intermezzo`) или плавающий объект, созданный нами с помощью `\definefloat` (см. section 13.5).

– *Options*. Набор символических слов, которые говорят ConTEXt как следует вставить объект. Подавляющее большинство из них ссылаются на *where*, чтобы вставить его. Мы увидим это в следующем разделе.

– *Label*. Метка для будущих внутренних ссылок на этот объект.

– *Title*. Текст заголовка, который будет добавлен к объекту. Относительно его конфигурации см. section 13.4.2.

– *Contents*. Это, конечно, зависит от типа объекта. Для изображений обычно используется команда `\externalimage`; для таблиц - команды, которые будут создавать таблицу; для *intermezzi* - фрагмент текста в рамке; и т.п.

Первые три аргумента, заключенные в квадратные скобки, необязательны. Последние два (заключенные в фигурные скобки) являются обязательными, хотя могут быть пустыми. Так, например:

`\placefloat{}{}` вставит

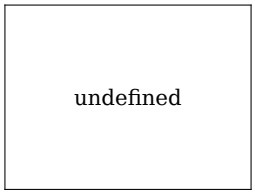


Рисунок 13.4

в документ.

Примечание: Мы видим, что ConTEXt считал, что вставляемый объект был изображением, поскольку он был пронумерован как изображение и включен в список images. Это заставляет меня предположить, что изображения являются плавающими объектами по умолчанию.

13.4.1 Параметры вставки плавающих объектов

Аргумент *Options* в `\placefigure`, `\placetable` и `\placefloat` управляет различными аспектами вставки этих типов объектов. В основном это место на странице, куда будет вставлен объект. Здесь поддерживаются несколько значений, каждое из которых имеет различную природу:

- Некоторые места вставки устанавливаются по отношению к элементам страницы (`top, bottom inleft, inright, inmargin, margin, leftmargin, rightmargin, leftedge, rightedge, innermargin, inneredge, outeredge, inner, outer`). Конечно, это должен быть объект, который может уместиться в той области, где он предназначен для размещения, и для этого элемента должно быть зарезервировано место в макете страницы. Об этом см. раздел 5.2 и 5.3.

- Другие возможные места вставки больше связаны с текстом, окружающим объект, и указывают, где следует разместить объект, чтобы текст обтекал его. По сути, значения `left` и `right`.

- Параметр `here` интерпретируется как рекомендация сохранить объект в той точке исходного файла, где он расположен. Эта *рекомендация* не будет соблюдаться, если это не разрешено требованиями к разбиению на страницы. Это указание усиливается, если мы добавляем параметр `force`, который означает именно это: принудительно вставить объект в этой точке. Обратите внимание, что при принудительной вставке в определенной точке объект больше не будет плавающим.

- Другие возможные варианты относятся к странице, на которую должен быть вставлен объект: “`page`” вставляет его на новую страницу; “`opposite`” вставляет его на страницу напротив текущей; “`leftpage`” на четной странице; “`rightpage`” на нечетной странице.

Есть некоторые варианты, не связанные с расположением объекта. Из их:

- `none`: Эта опция подавляет заголовок.

- `split`: Эта опция позволяет объекту занимать более одной страницы. Конечно, это должен быть объект, который делится по своей природе, например, таблица. Когда этот параметр используется и объект разбит, его нельзя больше назвать плавающим.

13.4.2 Настройка заголовков плавающих объектов

Если мы не используем параметр “`none`” в `\placefloat`, по умолчанию плавающие объекты связаны с заголовком, состоящим из трех элементов:

- Имя типа рассматриваемого объекта. Это имя в точности соответствует типу объекта; поэтому, если, например, мы определяем новый плавающий объект с именем “последовательность” и вставляем “последовательность” как плавающий объект, заголовок будет “Последовательность 1”. Просто напишите название объекта с заглавной буквы.

  > Несмотря на то, что только что было сказано, если основным языком документа не является английский, английское имя для предопределенных объектов, таких как, например, объекты “`figure`” или “`table`”, будет переведено; Так, например, объект “`figure`” в документах на испанском языке называется “`Figura`”, а объект “`table`” называется “`Tabla`”. Эти испанские имена для предопределенных объектов можно изменить с помощью `\setuplabeltext`, как описано в section 10.5.3.

- Его номер. По умолчанию объекты пронумерованы по главам, поэтому первой таблицей в главе 3 будет таблица ‘3.1’.

- Его содержимое. Введен как аргумент `\placefloat`.

С помощью`\setupcaptions` или `\setupcaption[Object]` мы можем изменить систему нумерации и внешний вид самого заголовка. Первая команда повлияет на все заголовки всех объектов, а вторая повлияет только на заголовок определенного типа объекта:

- Что касается системы нумерации, она контролируется параметрами `number`, `way`, `prefixsegments` и `numberconversion`:

  * `number` может принимать значения `yes`, `no` или `none` и определяет, будет ли число или нет.
  * `way` указывает, будет ли нумерация последовательной по всему документу (`way=bytext`), или она будет возобновляться в начале каждой главы (`way=bychapter`) или раздела (`way=bysection`). В случае перезапуска целесообразно согласовать значение этой опции с опцией `prefixsegments`.
  * `prefixsegments` указывает, будет ли у номера префикс и какой он будет. Таким образом, `prefixsegments=chapter` заставляет количество объектов всегда начинаться с номера главы, в то время как `prefixsegments=section` будет предшествовать номеру объекта с номером раздела.
  * `numberconversion` управляет типом нумерации. Значения для этого параметра могут быть: арабскими цифрами. (“`numbers`”), lower case (“`a`”, “`characters`”), upper case (“`A`”, “`Characters`”), small caps “`KA`”), upper case Roman numerals (“`I`”, “`R`”, “`Romannumerals`”), lower case (“`i`”, “`r`”, “`romannumerals`” or small caps (“`KR`”)).

– Внешний вид самого заголовка регулируется множеством опций. Я перечислю их, но для подробного объяснения значения каждого из них я обращаюсь к section 6.4.4, где объясняется управление внешним видом команд секционирования, поскольку параметры в основном такой же. Рассматриваемые варианты:
  * Чтобы контролировать формат всех элементов заголовка, `style, color, command`.
  * Чтобы управлять форматом только имени типа объекта: `headstyle, headcolor, headcommand, headseparator`.
  * Для управления только форматом нумерации: `numbercommand`.
  * Чтобы управлять только форматом самого заголовка: `textcommand`.

– Мы также можем контролировать другие аспекты, такие как расстояние между различными элементами, составляющими заголовок, ширину заголовка, его размещение по отношению к объекту и т. Д. Здесь я ссылаюсь на информацию в ConTEXt wiki относительно параметров, которые можно настроить с помощью этой команды.

13.4.3 Комбинированная вставка двух и более объектов

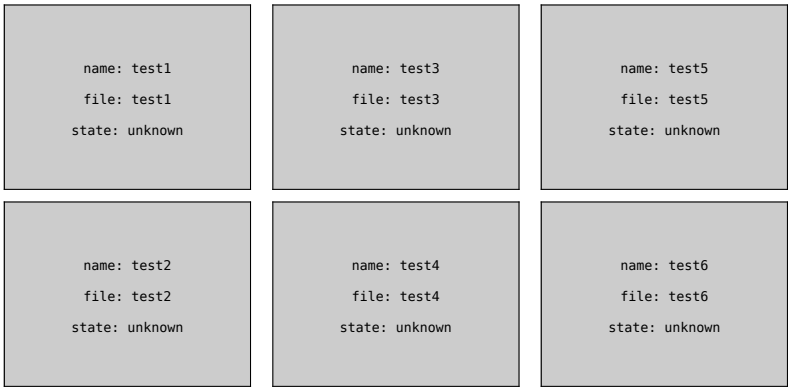
Чтобы вставить в документ два или более разных объекта, чтобы ConTEXt удерживал их вместе и работал с ними как с одним объектом, у нас есть среда `\startcombination`, синтаксис которой:

```
\startcombination[Ordering] ... \stopcombination
```

где *Ordering* указывает, как объекты должны быть упорядочены: если все они должны быть упорядочены по горизонтали, *Ordering* указывает только количество объектов, которые необходимо объединить. Но если мы хотим объединить объекты в две или более строк, нам нужно будет указать номер объекта в строке, затем количество строк и разделить оба числа символом *. Например:

```
\startcombination[3*2]
  {\externalfigure[test1]}
  {\externalfigure[test2]}
  {\externalfigure[test3]}
  {\externalfigure[test4]}
  {\externalfigure[test5]}
  {\externalfigure[test6]}
\stopcombination
```

что приведет к следующему выравниванию изображений.



В предыдущем примере изображения, которые я объединил, на самом деле не существуют, поэтому вместо изображений ConTEXt сгенерировал текстовые поля с информацией о них.

С другой стороны, посмотрите, как каждый элемент, объединяемый в `\startcombination`, заключен в фигурные скобки.

Фактически, `\startcombination` позволяет нам не только соединять и выравнивать изображения, но и любые блоки, такие как текст внутри среды`\startframedtext`, таблицы и т.д. Чтобы настроить комбинацию, мы можем использовать команду `\setupcombination`, а также мы можем создавать предварительно сконфигурированные комбинации, используя `\definecombination`.

13.4.4 Общая конфигурация плавающих объектов

Мы уже видели, что с помощью `\placefloat` мы можем контролировать местоположение вставляемого плавающего объекта и некоторые другие детали. Также возможно настроить:

- Глобальные характеристики конкретного типа плавающего объекта. Это делается с помощью `\setupfloat[Название типа плавающего объекта]`.
- Глобальные характеристики всех плавающих объектов в нашем документе. Это делается с помощью `\setupfloats`.

Помните, что так же, как `\placefloat[figure]` эквивалентен `\placefigure`, эквивалентен `\setupfloat[figure]`, а `\setupfloat[table]` эквивалентен `\setuptables`.

Что касается настраиваемых параметров для них, я отсылаю к официальному списку команд ConTEXt (section ??).

13.5 Определение дополнительных плавающих объектов

Команда `\definefloat` позволяет нам определять наши собственные плавающие объекты. Её синтаксис:

```
\definefloat[Singular name] [Plural name] [Configuration]
```

Где аргумент *Configuration* является необязательным аргументом, который позволяет нам уже указывать конфигурацию этого нового объекта во время его создания. Мы также можем сделать это позже с помощью `\setupfloat[Name in the singular]`.

Поскольку мы заканчиваем наше введение этим разделом, я собираюсь воспользоваться им, чтобы немного глубже погрузиться в кажущиеся джунгли *jungle* команд ConTEXt, которые, однажды понятые, не так уж похожи на джунгли *jungle*, но, на самом деле, вполне рациональны.

Давайте начнем с того, что спросим себя, что на самом деле представляет собой плавающий объект для ConTEXt, и ответим, что это объект со следующими характеристиками:

– Что у него есть определенная свобода в том, что касается его расположения на странице.

– С ним связан список *list*, который позволяет ему пронумеровать такие типы объектов и, в конечном итоге, создать их индекс.

– Что у него есть заголовок

– Что, когда объект действительно может плавать, он должен рассматриваться как неотделимая единица, то есть (в терминологии TEX) *заключенный в рамку*.

Другими словами, плавающий объект фактически состоит из трех элементов: самого объекта, связанного с ним списка и заголовка. Для управления самим объектом нам нужна только одна команда для установки его местоположения и другая для вставки объекта в документ; для настройки аспектов списка достаточно общих команд управления списком, а для настройки аспектов заголовка - общих команд управления заголовком.

И здесь проявляется гениальность ConTEXt: простая команда для управления плавающими объектами (`\setupfloats`) и простая команда для вставки плавающих объектов: `\placefloat` можно было бы спроектировать: но ConTEXt делает следующее:

1. Создайте команду, чтобы связать имя с определенной конфигурацией плавающего объекта. Это команда define float, которая на самом деле связывает не одно имя, а два имени, одно в единственном числе и одно во множественном числе.

2. Создайте вместе с командой глобальной конфигурации плавающих объектов команду, которая позволяет нам настраивать только определенный тип объекта: `\setupfloat[Object]`.

3. Добавьте к команде определения местоположения плавающего объекта (`\placefloat`) аргумент, который позволяет нам различать тот или иной тип: (`\placefloat[Object]`).

4. Создавайте команды, включая имя объекта, для всех действий плавающего объекта. Некоторые из этих команд (которые на самом деле являются клонами других более общих команд) будут использовать имя объекта в единственном числе, а другие будут использовать его во множественном числе.

Поэтому, когда мы создаем новый плавающий объект и сообщаем ConTEXt его имя в единственном и множественном числе, ConTEXt:

– Резервирует место в памяти для хранения конкретной конфигурации этого типа объекта.

– Создает новый список с единственным именем этого типа объекта, поскольку со списком связаны плавающие объекты.

– Создает новый вид «заголовка», связанный с этим новым типом объекта, чтобы поддерживать настраиваемую конфигурацию этих заголовков.

– И, наконец, он создает группу новых команд, специфичных для этого нового типа объекта, имя которого фактически является синонимом более общей команды.

В table 13.3 мы можем видеть команды, которые автоматически создаются при определении нового плавающего объекта, а также более общие команды, синонимами которых они являются:

Команда	Синоним	Пример
\completelistof<PluralName>	\completelist[PluralName]	\completelistoffigures
\place<SingularName>	\placefloat[SingularName]	\placefigure
\placelistof<PluralName>	\placelist[PluralName]	\placelistoffigures
\setup<SingularName>	\setupfloat[SingularName]	\setupfigure

Таблица 13.3 Команды, которые автоматически создаются при создании нового плавающего объекта

Фактически, создаются некоторые дополнительные команды, которые являются синонимами предыдущих, и, поскольку я не включил их в объяснение главы, я исключил их из таблицы table 13.3: \start<NameSingular>, \start<NameSingular>text и \startplace<NameSingular>.

Я использовал команду, используемую для изображений, в качестве примера команд, созданных при определении нового плавающего объекта; и я сделал это, потому что изображения, такие как таблицы и остальные числа с плавающей запятой, предопределенные ConTEXt, являются фактическими случаями \definefloat:

```
\definefloat[chemical][chemicals]
\definefloat[figure][figures]
\definefloat[table][tables]
\definefloat[intermezzo][intermezzi]
\definefloat[graphic][graphics]
```

Наконец, мы видим, что в действительности ConTEXt никоим образом не управляет каким-либо материалом, включенным в каждый конкретный плавающий объект; предполагается, что это работа автора. Вот почему мы также можем вставлять текст с помощью команд \placefigure или \placetable. Однако текст, вводимый с помощью placefigure, включается в список изображений, а если вводится с помощью \placetable, в список таблиц.