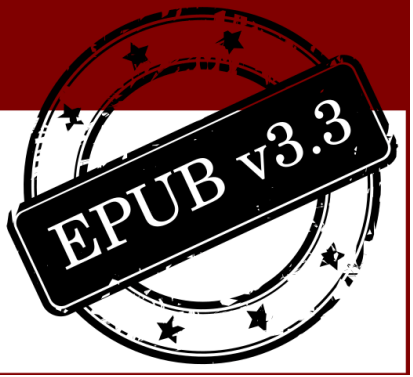
DIY tools and recipes for the common folk

A practical guide to EPUB



Alexander Gromnitsky

A practical guide to EPUB

DIY tools and recipes for the common folk

BY ALEXANDER GROMNITSKY

First edition: March 2023
Latest update: 2024-02-20T12:43:42Z

To download the code examples or leave comments, visit github.com/gromnitsky/pg2e-support.

To download the PDF version of the book, visit https://sigwait.org/~alex/p/pg2e/.

Contents

Preface

To make e-books, you do not have to be a well-off publisher, a melancholy librarian, or a power user of complex page layout software. A text editor and a ZIP compression utility will do just fine, if not better. There are, of course, *details*, and this book is all about them.

EPUB v3 is the standard format for distributing e-books across all e-reader devices and pure software readers. It uses HTML/CSS as the basis for its content, structure, and styling.

This short book explains how to make an EPUB file from scratch and how to automate this somewhat tedious process.

The topics we will discuss include:

- the EPUB container;
- what goes into the EPUB;
- all about metadata;
- pitfalls of XML;
- how to generate *and number* the Table of Contents (TOC);
- equations, code listings, and footnotes.

Like websites, e-books can be of reflowable and fixed layouts. We will only be discussing the former. If you plan to create a comic or a coffee table book, this guide will only help you with the first half of your journey.

Topics we will not cover include DRM, font obfuscation, media overlays, non-European languages, and accessibility.

The book is meant to be read sequentially on the first read and used as a quick reference afterward.

What is expected from a reader

Besides HTML/CSS, you must be comfortable with the command line and familiar with scripting languages. We use Ruby extensively, but if you know Python or JavaScript, you should be fine too.

Making an EPUB unfortunately requires dealing with XML. Using Ruby allows us to use 鋸 (Nokogiri) to ease our job. It is a powerful Ruby gem that enables you to navigate XML trees, modify its elements and attributes, and serialize all back into a string. Nokogiri is sometimes touted as the best tool ever written to handle XML.

Please refer to Appendix 1 for the full list of software that we will be using throughout the book.

1 Container

An *.epub* file is "a website in a box": a collection of *.xhtml* files inside a *.zip* file. Alongside with *.xhtml*, all additional resources (images, style sheets, custom fonts) must be inside the *.zip* file as well.

To qualify as a proper EPUB, a couple of metadata files inside a ZIP container must be present too. Those metadata files help a book reader device to:

- do a quick sanity check of a given *.epub* without parsing the whole file;
- find where concreate metadata for the default book is located (yes, technically a single Epub may contain multiple books);
- find a table of contents (TOC) for a selected book.
- open such a page of the book that the publisher has designated as a starting location (a 1st chapter, a copyright page, a warning of some sort, &c).

In this chapter we'll talk about the first 2 items & why you can run into troble using a regular (ZIP) archiver, whether integrated into the OS or a general-purpose third-party one.

1.1 An importance of a magic + language test

How can you detect, with considerable certainty, that a file named "foo.epub" is indeed an EPUB without parsing the whole file? It is possible to do so, and very quickly. To answer the question, we need to dwell on the structure of the .zip file.

The ZIP file format is of an unorthodox nature: **first**, it allows the usage of different compression algorithms for each file in an archive. The EPUB specification allows only 2:

1. no compression (also called "store");
2. Deflate.

Second, for each file it holds, a ZIP has 2 similar "header" records in different places: before file data & in an index. The index (called *central directory*) is iconoclastically placed in the very end of a *.zip* file.

<table>
<tr><td colspan="3">Magic number</td></tr>
<tr><td rowspan="4">Payload</td><td>local file header 0</td><td>file 0</td></tr>
<tr><td>local file header 1</td><td>file 1</td></tr>
<tr><td colspan="2">...</td></tr>
<tr><td>local file header n</td><td>file n</td></tr>
<tr><td rowspan="4">Index</td><td colspan="2">central directory header 0</td></tr>
<tr><td colspan="2">central directory header 1</td></tr>
<tr><td colspan="2">...</td></tr>
<tr><td colspan="2">central directory header n</td></tr>
</table>

	end of central directory

An array of *central directory headers* + *end of central directory* record comprises the index.

Both *local file* & *central directory* headers contain metadata (name, size, timestamp, &c) for a particular file.

The *central directory header* has expanded metadata, not found in a *local file header* + some annoyingly redundant fields that are equal to the fields in the *local file header*.

The presence of *local file header* records theoretically allows for streaming: you can read zipped data file-by-file without knowing when the files will end.

The *local file header* record consists of a multitude of fields: some of them are of a known fixed size, but other are of a variable length.

Field Name	*Size, B*
version needed to extract	2
general purpose bit flag	2
compressed method	2
last mod file time	2
last mod file date	2
crc-32	4
compressed size	4
uncompressed size	4
file name length	2
extra field length	2
file name	variable
extra field	variable

How would you start detecting an *.epub*? According to the EPUB specification, the 1st file in the ZIP container **must** be a file named "mimetype" that

1. contains a string "application/epub+zip";
2. uncompressed;
3. has no "extra field" present in its *local file header*.

Let's calculate:

- *mn*: ZIP magic number is 2 bytes: `PK` (for Phil Katz).
- *lfh*: *local file header* comes next. If it consists only of fixed fields, we can safely skip n bytes (28, to be presise).
- *fn*: The length of the file name "mimetype" file is known too: 8 bytes.
- *data*: File data immediate follows the *local file header*, & we know the size of it (20 bytes for "application/epub+zip" string).

$$mn + lfh + fn + data = 2 + 28 + 8 + 20 = 58$$

```
$ head -c58 foo.epub | hexdump -C
00000000  50 4b 03 04 14 00 00 00  00 00 ea 7a 4c 56 6f 61  |PK.........zLVoa|
00000010  ab 2c 14 00 00 00 14 00  00 00 08 00 00 00 6d 69  |.,............mi|
00000020  6d 65 74 79 70 65 61 70  70 6c 69 63 61 74 69 6f  |metypeapplicatio|
00000030  6e 2f 65 70 75 62 2b 7a  69 70                    |n/epub+zip|
0000003a
```

Or, this one liner in Ruby exits with 0 if a detection has been successful:

```
$ head -c58 foo.epub | ruby -E ascii-8bit -e 'exit "application/epub+zip" == $<.read.spli
```

```
t(/^PK/)[1]&.[](36..-1)'
$ echo $?
0
```

file(1) also thinks that 58 raw bytes is the whole EPUB book:

```
$ head -c58 foo.epub | file -
/dev/stdin: EPUB document
```

This all breaks down miserably when you pack your *.xhtml* & co. with a general purpose archive program that has no knowledge of EPUB peculiarities whatsoever.

Even if it manages to put 'mimetype' file at the beginning of the archive, it may compress file contents and/or add *extra field* of variable length to the *local file header* record (to fill it with GUI/UID information, &c).

Take libarchive bsdtar(1):

```
$ head -c 65536 /dev/urandom > junk
$ printf application/epub+zip > mimetype
$ bsdtar -a -cf foo.zip mimetype junk

$ zipinfo foo.zip | grep -
-rw-r--r--  2.0 unx       20 bX defN 23-Feb-10 20:53 mimetype
-rw-r--r--  2.0 unx    65536 bX defN 23-Feb-10 20:53 junk
```

- 'bX' in the 5th field means it is a binary with "extra field" present in both in *local file* & *central directory* headers (we do not care about central directory here).
- 'defN' means it is compressed with Deflate algorithm (level: normal).

If we examine foo.zip in any forensic tool that understands ZIP containers, we can see that compressed size is now 22 bytes & we have additional 32 bytes in "extra field".

Without knowing about the ZIP format structure, it's impossible to guess those numbers beforehand => impossible to precalculate the length of a *local file header* record.

bsdtar just tried to be helpful.

1.2 zip(1)

There are couple of things you can do to alleviate the problem. If you are using an archiver integrated in the OS, try to make a ZIP container in 2 steps:

1. make an archive from a single 'mimetype' file, & make sure it was added without compression & "extra field";
2. add the rest of the files to the archive.

Another option is to use a classic zip(1) utility. You'll have to invoke it in 2 steps too:

```
$ zip -qX0 foo.epub mimetype
```

creates `foo.epub` with a single file in it, -X means no "extra filed" (no UID/GID & timestamps), -0 means no compression.

```
$ zip -qX foo.epub chapter1.xhtml style.css
```

appends the rest of the files.

1.3 File names

In the beginning, the ZIP format supported only the original IBM Code Page 437 for encoding file names. Besides English, it had full character support only for Deutsch & Svenska. To address this limitation, 11th bit of "general purpose bit flag" in the *local file header* record was used: when set, a file name was expected to be in UTF-8, if not—in cp437. This bit is also called EFS (language encoding flag).
Unfortunately, various ZIP archivers behave erratically towards it:

- some encode file names in UTF-8, but leave the EFS bit unset; this yields to "hieroglyphs" on the receiving end: the unpacker thinks that a file name is in cp437 & does cp437→user_locale (usually UTF-8) conversion;
- Others, for "compatibility" reasons, encode file names in a pre-Unicode encoding based on the geographical region where the OS was installed. E.g., you may suddenly encounter the ancient cp866 if a .zip file was created on a PC in Eastern Europe. The EFS bit is usually unset as well, and the unpacker, following the ZIP specification, tries to convert the string from cp437 with a depressing result.

To add more excitement, a new solution to the encoding issues was proposed: let old programs do what they want—as they don't check EFS bit, it's unwise to set it & encode files names in UTF-8; let's put file names inside "extra field" in an *local file* or *central directory* headers, & unequivocally encode them (file names) in UTF-8 over there. Then the "new programs" will know what to do after checking the data in the "extra field".

Of course, this behaviour of a user-agent is never advertised. Therefore, unfortunately, whilst using non-ASCII file names, assorted links in your *.xhtml* (or EPUB metadata) expect proper UTF-8 paths everywhere, but the user-agent may compare them with rubbish obtained by vandalising strings with cp437.

Besides our 'stick to ASCII' rule of thumb, EPUB puts additional restriction on file names, viz.:

- NAME_MAX = 255;
- MAX_PATH = 2^{16} - 1;
- no `.` as the last character;
- some characters from the ASCII range are forbidden:
  - C0 and C1 control codes (NULL, `\a`, `\t`, `\n`, and so on);
/	"	*	:	<	>	?	\	\v

1.4 Manifest

After an *.epub* successfully passes the simple check with 'mimetype', a user-agent needs to decide which book inside a ZIP container it should start parsing.

It tries to do that by looking for `META-INF/container.xml` file. E.g., `container.xml` for a collection of *Macworld* magazines for the year 1984 may look like

```
<?xml version="1.0"?>
<container version="1.0"
           xmlns="urn:oasis:names:tc:opendocument:xmlns:container">
  <rootfiles>
    <rootfile full-path="1984-April/package.opf"
              media-type="application/oebps-package+xml" />
    <rootfile full-path="1984-June/package.opf"
              media-type="application/oebps-package+xml" />
    <rootfile full-path="1984-September/package.opf"
              media-type="application/oebps-package+xml" />
    <rootfile full-path="1984-November/package.opf"
              media-type="application/oebps-package+xml" />
    <rootfile full-path="1984-December/package.opf"
              media-type="application/oebps-package+xml" />
  </rootfiles>
</container>
```

The 1st child of `container>rootfiles` node is the default one. Do not get excided too much, though: a support for multiple *renditions* (== books) inside of a single ZIP container is lacking among user-agents, hence in the majority of cases you will only have 1 `container>rootfiles>rootfile` node.

`package.opf` file serves as the entry point for a book's metadata. The name can be anything (`foobar.opf` is fine), but the extension should remain. The grotesque 3-letter acronym (Open Package Format) is a remnant of the now defunct Open eBook Forum working group & their recommendations during the peek of XML mania a couple of years before the dot-com crash.

In contrast with the strict 'mimetype' file position, `META-INF/container.xml` can be anywhere in the *.zip* file, there is no rule that prevents you putting it in the container after all other files.

The META-INF directory is sometimes used to store settings by user-agents (such as saving the last opened page, holding bookmarks, &c) or by post-processing tools. For example, if you start a SaaS business that diacriticalizes texts, a customer could upload their *.epub* to your ~~mainframe~~ cloud & receive a refined version of it with an additional `META-INF/lol.txt` file inside:

> Grätùîtòûslý ëñhäñçëð wíth diacriticalize.el, v1.1. Ïñtêñsífý thé välúè ôf ¥ÔÛR bôök ät diacriticalize.com!

This would be in full conformance with the Epub specification.

1.5 A custom packer in C

(The rest of the chapter can be skipped during the 1st reading.)

We may go even further, & prevent creation of bogus ZIP containers by writing a replacement for zip(1) program. With a custom packer we can reject unsupported files (such as symlinks), automatically add `mimetype` file (thus not even have it in a book repository), & be slightly more intelligent when adding incompressible files (like images) to the container.

The most tricky part here is injecting `mimetype` file correctly. Not every library for creating zip archives has an API that allows simultaneously:

1. override a compression method (we need 'store', i.e. no compression);
2. skip making a not so helpful "extra field" in a *local file header* record.

1.5.1 libzip

When using this library for *writing* archives, it operates in a queueing fashion: each action of adding a file to an archive does not perform IO immediately but defers it to the very last step that runs when a user eventually calls `zip_close()` function.

To compile the example, put all the chunks below in 1 file `epub-zip.c`, install development files for libzip (a package usually called ≈ `libzip-devel`), & run:

```
$ cc epub-zip.c -lzip -o epub-zip
```

1.5.2 main()

Usage:

```
epub-zip output.epub file1 file2 ...
```

A proper Unix program never reimplements steps that can be made by running another program. Hence, it would be unfitting to travel the directory structure recursively (as zip(1) does with -r CLO), when we already have find(1) command, & can pass a list of files like

```
$ cd src
$ epub-zip ../output.epub `find .`
$ zipinfo -1 ../output.epub
mimeinfo
chapter-01.xhtml
chapter-02.xhtml
...
```

Our `main()` is very humble, except for the `argv` slicing & error reporting it only calls `epub` function:

```
#include <sys/stat.h>
#include <err.h>
#include <libgen.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>

#include <zip.h>

typedef struct { char *file; char message[256]; } MyError;
void myerror_set(MyError *e, char *file, const char *msg) {
  e->file = file;
  snprintf(e->message, 256, msg);
}

/* … */
int main(int argc, char **argv) {
  if (argc < 3) errx(1, "Usage: %s out.zip file ...", basename(argv[0]));

  MyError error = {};
  epub(argv[1], argv+2, argc-2, &error);
  if (error.file) errx(1, "%s: %s", error.file, error.message);
}
```

There is a custom little structure for errors, for we can expect 2 kind of faults from

1. system calls, where `errno` is set;
2. libzip library itself.

If we were just to return an integer, there would be no way to distinguish the source of an error, therefore, with `errno` we call `strerror()` to extract a user-friendly description in the guts of the program, & a libzip-specific `zip_strerror()` that expects not an integer, but a pointer to `zip_t` opaque object. `main()`, of course, doesn't need to know that, it only sees `MyError` object.

1.5.3 epub()

This function truncates the output file, allocates a number of buckets (`MyZip#files`) into which libzip puts

(compressed) objects, manually creates the 1st entry in an archive, then iterates through the list of all files, queueing them.

```
typedef struct {
  zip_t *arc;
  zip_source_t **files;
  int idx;
} MyZip;

/* … */
void epub(char *out, char **file_list, int file_list_len, MyError *error) {
  MyZip mz = {
    .arc = zip_open(out, ZIP_CREATE|ZIP_TRUNCATE, NULL),
    .files = malloc((file_list_len + 1)*sizeof(zip_source_t*))
  };
  if (!mz.arc) {
    myerror_set(error, out, "failure to create");
    return;
  }

  mimetype_add(&mz);

  for (char **file = file_list; *file; file++) {
    if (getenv("EPUB_ZIP_DEBUG")) warnx("queueing %s", *file);
    file_add(&mz, *file, error); if (error->file) break;
  }

  if (!error->file) {           /* commit */
    if (zip_close(mz.arc) < 0) myerror_set(error, "zip", zip_strerror(mz.arc));
  }
  if (error->file) {
    zip_discard(mz.arc);
    if (mz.idx > 1) unlink(out);
  }
  free(mz.files);
}
```

Each `MyZip#files` element must survive until `zip_close()` is called. If `file_add()` was to commit changes to the output file immediately, `MyZip#files` would be an unnecessary complication, but `file_add()` only *queues* its operations for a future use by `zip_close()`.

When the function fails to process any file from `argv` slice, it immediately stops the processing, & updates `error` object.

A partially created archive is removed from the filesystem.

1.5.4 file_add()

First, we call stat(2) to get the file type & to catch files with invalid paths right away. We silently ignore directories, for adding `foo/bar` adds nothing useful when a file entry already contains the full path `foo/bar/baz.xhtml`.

If the file is not a regular file (socket, symlink, &c), an error is raised.

```
void file_add(MyZip *mz, char *file, MyError *error) {
  struct stat st; if (-1 == stat(file, &st)) {
    myerror_set(error, file, strerror(errno));
    return;
  }

  if (S_ISREG(st.st_mode)) {
    if (NULL == (mz->files[mz->idx] = zip_source_file(mz->arc, file, 0, -1)) ||
        zip_file_add(mz->arc, file, mz->files[mz->idx], ZIP_FL_ENC_UTF_8) < 0) {
      zip_source_free(mz->files[mz->idx]);
      myerror_set(error, file, zip_strerror(mz->arc));
```

```
      return;
    }
    set_compression(mz, file);
    mz->idx++;
  } else if (!S_ISDIR(st.st_mode)) myerror_set(error, file, strerror(EINVAL));
}
```

`MyZip#idx` is incremented only after a successful add. Knowing an index of a file in the archive it's possible to set additional options for it, like a compression method.

File metadata like a timestamp or permissions is added by libzip automatically via copying them from the underline source file.

1.5.5 Miscellaneous

The least interesting chunks are detecting the compression method (the default one is Deflate, but we set it to 'store' for images, because they are already compressed by virtue of their structure), & a procedure for manually adding 'mimetype' file.

We naïvely discover images by looking at file extensions. The string `heystack` holds a 'database' of image formats we have decided to recognise:

```
.png.jpg.gif.
```

The last dot is not a typo but a sentinel. It helps not to confuse '.g' with '.gif' when doing a substring match. (Not that one should expect any 'foo.g' or 'bar.p' files in an epub, but still.)

```
char *ext(char *file) {
  char *dot = strrchr(file, '.');
  return (!dot || dot == file) ? "" : dot;
}

void set_compression(MyZip *mz, char *file) {
  char *heystack = ".png.jpg.gif.", *e = ext(file), *p = strstr(heystack, e);
  if (0 != strlen(e) && p && '.' == (p+strlen(e))[0])
    zip_set_file_compression(mz->arc, mz->idx, ZIP_CM_STORE, 0);
}
```

`mimetype_add()` is bare-bone, we don't set neither permissions for 'mimetype' file nor timestamp, thus libzip chooses the default value 0666 & the current time correspondingly. We also don't set 'store' method explicitly—the library understands that compressing 20 bytes is rather pointless.

```
void mimetype_add(MyZip *mz) {
  char *buf = strdup("application/epub+zip");
  mz->files[mz->idx] = zip_source_buffer(mz->arc, buf, strlen(buf), 1);
  zip_file_add(mz->arc, "mimetype", mz->files[mz->idx++], ZIP_FL_ENC_UTF_8);
}
```

If we were to set custom permissions, where would they go? libzip uses `zip_file_set_external_attributes()` to update a 4-bytes field in a *central directory header*. The last 2 parameters to the function are

- `zip_uint8_t` *opsys* (set it to 0x03 that in the ZIP specification means Unix);
- `zip_uint32_t` *attributes*, a 4 bit field, of which only the first 2 bits are relevant for "Unix".

To calculate *attributes* parameter, we neet to settle on:

- file type;
- setuid, setgid, sticky bit;
- permissions.

'mimetype' is a regular file, i.e., `S_IFREG` (defined as 0100000 in octal in your libc headers), & permission is easier to write by hand, instead of using `S_Ixxxx` definitions.

E.g., for un file regolare, that has no setuid, setgid or sticky bit set & that has `rw-r--r--` permissions, the resulting number would be

```
0100644u << 16
```

1.6 A container fixer (also in C)

We can solve the problem with 'mimetype' file from a different angle: if you compress book files with an homely generic archiver, it should be possible to move the file to the beginning of a archive & set a suitable *local file header* for it.

The same libzip library allows replacing files inside a ZIP & modifying extra fields in *local file headers* or in *central directory headers* records.

To test this, create a ZIP with 2 files:

```
$ head -c 65536 /dev/urandom > junk
$ printf application/epub+zip > mimetype
$ bsdtar -a -cf foo.zip junk mimetype
```

```
$ zipinfo foo.zip | grep -
-rw-r--r--  2.0 unx    65536 bX defN 23-Feb-10 20:53 junk
-rw-r--r--  2.0 unx       20 bX defN 23-Feb-10 20:53 mimetype
```

Everything is wrong here: 'mimetype' file location, its compression method & the presence of extra fields.

```
 0) file:ch01/foo.zip: ZIP archive (64.4 KB)
    0) header[0]= 0x04034b50: Header (4 bytes)
  + 4) file[0]: File entry: junk (0) (74 bytes)
    78) unparsed[0]= "\xc8\xaf#\x9e\x949\xf5\xbcT\x1e\xe0K`\xe3(...)": Raw data (
    65622) header[1]= 0x08074b50: Header (4 bytes)
  + 65626) spanning[0] (12 bytes)
    65638) header[2]= 0x04034b50: Header (4 bytes)
  - 65642) file[1]: File entry: mimetype (0) (78 bytes)
     + 0) version_needed: Version needed (2 bytes)
     + 2) flags: General purpose flag (2 bytes)
       4) compression= Deflate: Compression method (2 bytes)
     + 6) last_mod= 2023-02-10 20:53:28: Last modification file time (4 bytes)
       10) crc32= 0x00000000: CRC-32 (4 bytes)
       14) compressed_size= 0: Compressed size (4 bytes)
       18) uncompressed_size= 20: Uncompressed size (4 bytes)
       22) filename_length= 8: Filename length (2 bytes)
       24) extra_length= 32: Extra fields length (2 bytes)
       26) filename= "mimetype": Filename (8 bytes)
     - 34) extra: Extra fields (32 bytes)
         - 0) extra[0] (17 bytes)
              0) field_id= extended timestamp: Extra field ID (2 bytes)
              2) field_data_size= 13: Extra field data size (2 bytes)
              4) field_data= "\a\xa8\x92\xe6c\xa8\x92\xe6c\xa8\x92\xe6c": Unknown
         - 17) extra[1] (15 bytes)
              0) field_id= 30837: Extra field ID (2 bytes)
              2) field_data_size= 11: Extra field data size (2 bytes)
              4) field_data= "\1\4\xe8\3\0\0\4d\0\0\0": Unknown field data (11 byt
```

1.6.1 main()

Usage:

```
$ epub-zip-mimetype-fix foo.zip

$ zipinfo foo.zip | grep -
-rw-r--r--  6.3 unx       20 b- stor 23-Feb-10 20:53 mimetype
-rw-r--r--  6.3 unx    65536 bx defN 23-Feb-10 20:53 junk
```

`file.epub` is "fixed" in-place. The program is idempotent—the same file would be patched as many times as you execute the utility (hopefully, with the same outcome).

```
#include <err.h>
#include <libgen.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>

#include <zip.h>

/* … */
int main(int argc, char **argv) {
  if (argc < 2) errx(1, "Usage: %s file.zip", basename(argv[0]));

  char error[BUFSIZ] = "";
  fix(argv[1], error);
  if (0 != strlen(error)) errx(1, "%s: %s", argv[1], error);
}
```

Again, our `main()` is very plain; it ony passes a string buffer to `fix()` function & reports an error.

1.6.2 fix()

After opening the archive, we search for 'mimetype' file inside it. If its index is 0, our job is to remove extra fields & remove compression. If the index != 0, we call `swap()` routine.

```
void fix(char *file, char *error) {
  zip_t *arc = zip_open(file, 0, NULL); if (!arc) {
    snprintf(error, BUFSIZ, "can't open");
    return;
  }

  int idx = zip_name_locate(arc, "mimetype", 0);

  if (-1 == idx) {
    snprintf(error, BUFSIZ, "no mimetype file");
    zip_close(arc);
    return;
  }

  if (0 != idx) swap(arc, 0, idx);

  if (getenv("EPUB_ZIP_DEBUG")) warnx("fixing index 0");
  fix_index_zero(arc);

  if (zip_close(arc) < 0) snprintf(error, BUFSIZ, zip_strerror(arc));
}
```

1.6.3 swap()

If 'mimetype' file isn't located at the beginning of an archive, this means its place is occupied by another file.

```
void swap(zip_t *arc, int a, int b) {
  const char *a_name = zip_get_name(arc, a, 0);
  zip_source_t *a_src = zip_source_zip_create(arc, a, 0, 0, -1, NULL);
  const char *b_name = zip_get_name(arc, b, 0);
  zip_source_t *b_src = zip_source_zip_create(arc, b, 0, 0, -1, NULL);
  if (getenv("EPUB_ZIP_DEBUG"))
    warnx("swapping %d (%s) with %d (%s)", a, a_name, b, b_name);

  zip_file_replace(arc, a, b_src, ZIP_FL_ENC_UTF_8);
  zip_file_replace(arc, b, a_src, ZIP_FL_ENC_UTF_8);
  zip_file_rename(arc, a, "566bfa62-97ff-44df-be9a-ab9d6e4b3f93", 0); // tmp

  zip_file_rename(arc, b, a_name, 0);
  zip_file_rename(arc, a, b_name, 0);
}
```

1.6.4 Last step

We remove extra fields both from a *local file header* & a *central directory header* records. Technically the latter is unnecessary, but if we leave it alone, zipinfo(1) would not draw a helpful hyphen '-' in the 5th field, that indicates the absence of extra fields for an entry.

```
void fix_index_zero(zip_t *arc) {
  zip_file_extra_field_delete(arc, 0, ZIP_EXTRA_FIELD_ALL, ZIP_FL_LOCAL);
  zip_file_extra_field_delete(arc, 0, ZIP_EXTRA_FIELD_ALL, ZIP_FL_CENTRAL);
  zip_set_file_compression(arc, 0, ZIP_CM_STORE, 0);
}
```

```
 0) file:ch01/foo.zip: ZIP archive (64.3 KB)
    0) header[0]= 0x04034b50: Header (4 bytes)
  - 4) file[0]: File entry: mimetype (20) (54 bytes)
     + 0) version_needed: Version needed (2 bytes)
     + 2) flags: General purpose flag (2 bytes)
       4) compression= no compression: Compression method (2 bytes)
     + 6) last_mod= 2023-02-10 20:53:28: Last modification file time (4 bytes)
       10) crc32= 0x2cab616f: CRC-32 (4 bytes)
       14) compressed_size= 20: Compressed size (4 bytes)
       18) uncompressed_size= 20: Uncompressed size (4 bytes)
       22) filename_length= 8: Filename length (2 bytes)
       24) extra_length= 0: Extra fields length (2 bytes)
       26) filename= "mimetype": Filename (8 bytes)
       34) data= "application/ep(...)": File "mimetype" (20 bytes) (20 bytes)
    58) header[1]= 0x04034b50: Header (4 bytes)
  + 62) file[1]: File entry: junk (65556) (64.1 KB)
```

Exercises

epub-zip:

1. Create `META-INF/container.xml` automatically if an .opf file is found in `argv`.
2. Add an option to cut out prefixes from the file names, e.g., `epub-zip -p src/ output.zip `find src`` should allow packing files from `src/` without the need to change the current directory to `src/`.
3. Rewrite the program in your favourite scripting language.

epub-zip-mimetype-fix:

1. Before doing `swap()`, check that 'mimetype' entry is 20 bytes long & contains the exact "application/

epub+zip" string, otherwise we may accidentally mangle wrong containers.

2 Metadata

A valid EPUB file must have at least 5 files inside its ZIP container:

1. mimetype
2. META-INF/container.xml
3. foo.opf
4. chapter1.xhtml
5. mytoc.xhtml

The first 2 file names are static, the rest could be named in whatever way you like. `mytoc.xhtml` here contains a machine-parseble table of contents (TOC) written specifically for a user-agent, which may (or may not) use it to draw a some kind of navigational menu. If you write `mytoc.xhtml` in an accurate manner, you can simultaneously reuse it as a regular book content.

Everything else is optional:

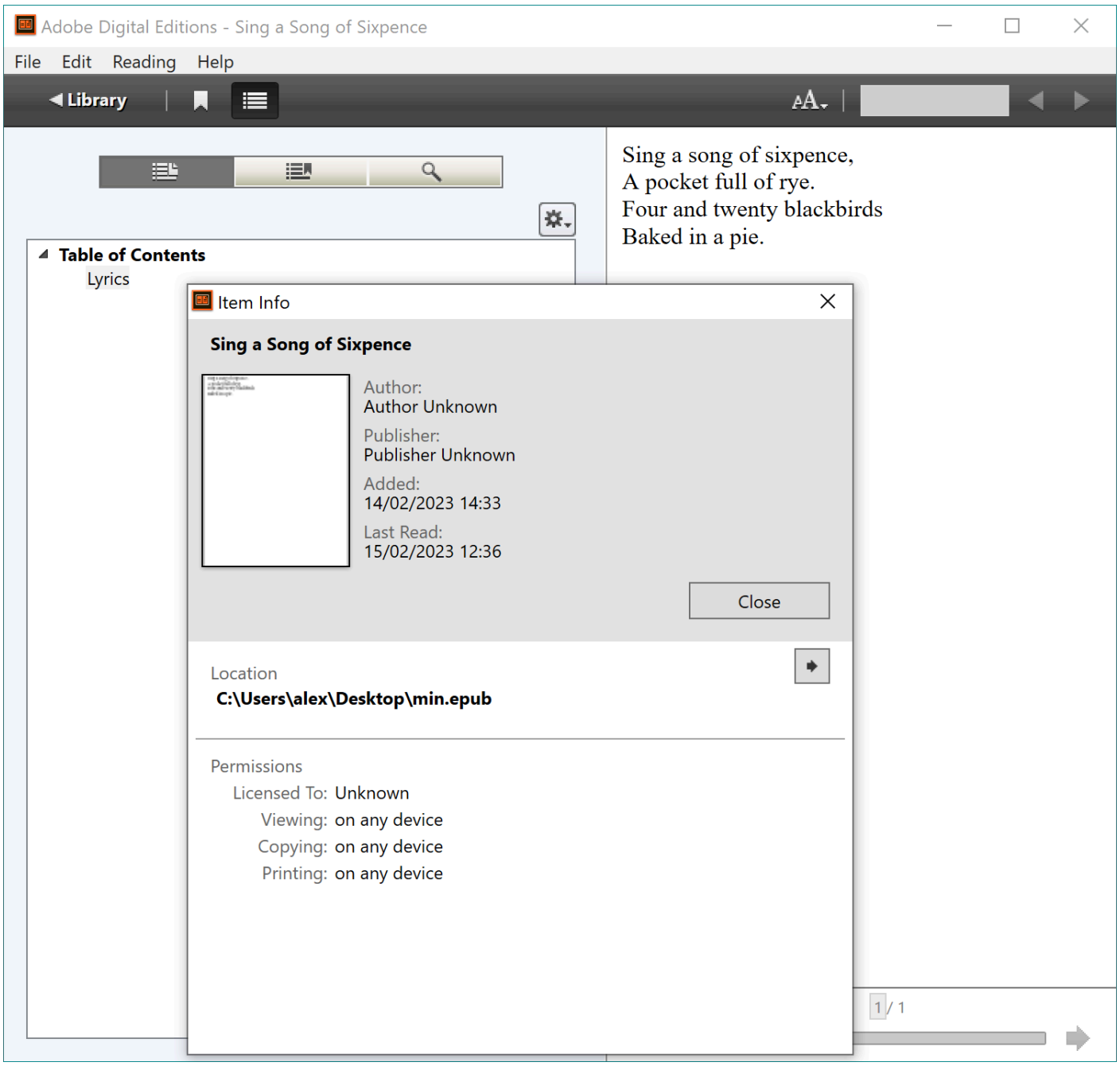
- a cover page;
- CSS stylesheets;
- images;
- fonts.

EPUB doesn't distinguish between a random chapter of the book & a dedication page, an epigraph, or a bibliography. To a user-agent it's just a entry in a TOC.

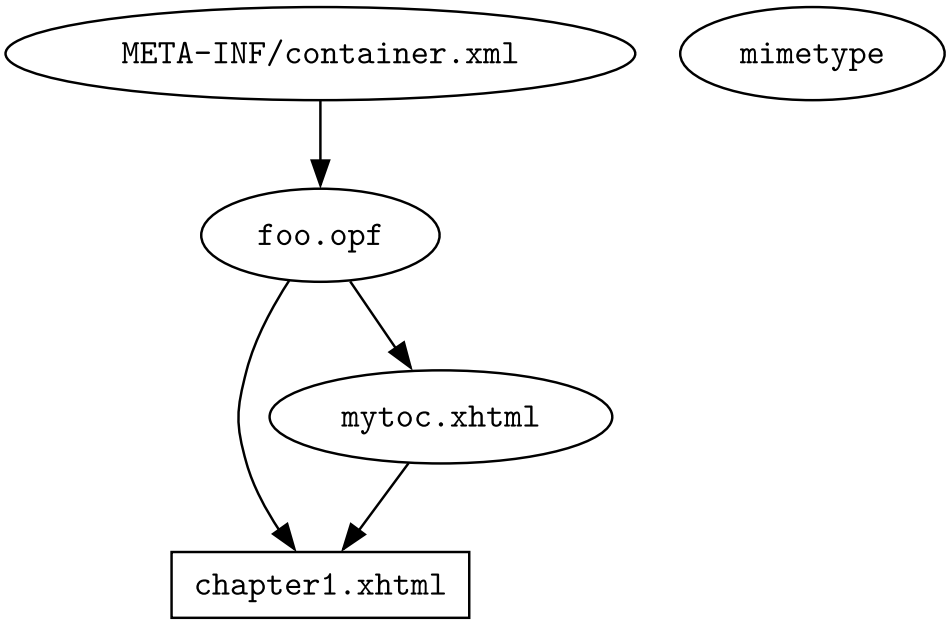
```
$ mkdir min && cd min
$ printf application/epub+zip > mimetype
$ mkdir META-INF

$ cat > META-INF/container.xml <<END
<?xml version="1.0"?>
<container version="1.0" xmlns="urn:oasis:names:tc:opendocument:xmlns:container">
 <rootfiles>
  <rootfile full-path="foo.opf" media-type="application/oebps-package+xml" />
 </rootfiles>
</container>
END

$ cat > foo.opf << END
<?xml version="1.0" encoding="utf-8"?>
<package version="3.0" unique-identifier="bid" xmlns="http://www.idpf.org/2007/opf">
 <metadata xmlns:dc="http://purl.org/dc/elements/1.1/">
  <dc:title>Sing a Song of Sixpence</dc:title>
  <dc:language>en</dc:language>
  <dc:identifier id="bid">urn:uuid:47962a91-8e6c-41a9-ad98-b9efb7210dae</dc:identifier>
  <meta property="dcterms:modified">2023-02-13T00:00:00Z</meta>
```

```
 </metadata>
 <manifest>
  <item id="f1" href="mytoc.xhtml" media-type="application/xhtml+xml" properties="nav"/>
  <item id="f2" href="chapter1.xhtml" media-type="application/xhtml+xml" />
 </manifest>
 <spine><itemref idref="f2" /></spine>
</package>
END

$ cat > chapter1.xhtml << END
<?xml version="1.0" encoding="utf-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>.</title></head>
<body>
 Sing a song of sixpence,<br/>
 A pocket full of rye.<br/>
 Four and twenty blackbirds<br/>
 Baked in a pie.
</body>
</html>
END

$ cat > mytoc.xhtml << END
<?xml version="1.0" encoding="utf-8"?>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:epub="http://www.idpf.org/2007/ops">
<head><title>.</title></head>
<body>
 <nav epub:type="toc">
  <ol><li><a href="chapter1.xhtml">Lyrics</a></li></ol>
 </nav>
</body>
</html>
END
```

This is how it may be rendered by a user-agent on a PC:



The following is a relationship between 5 files in a form of a graph, where an arrow means "links to", and the rendered for a human consumption nodes are box-shaped:



Compile them into `min.epub` & run a linter:

```
$ zip -q -X0 ../min.epub mimetype
$ zip -q -Xr ../min.epub *
$ epubcheck ../min.epub
Validating using EPUB version 3.3 rules.
No errors or warnings detected.
Messages: 0 fatals / 0 errors / 0 warnings / 0 infos
```

2.1 .opf

> OEB is based on XML because of its generality and simplicity.
> — *Open eBook Publication Structure*, 16 Sep 1999

> XML is nasty to parse for humans, and it's a disaster to parse even for computers. There's just no reason for that horrible crap to exist.
> — Linus Torvals, *Google+*, Mar 6 2014

The *.opf* file, to which META-INF/container.xml refers, also known as 'the package file,' is an XML document that

1. specifies basic metadata (title, author, &c);
2. lists all included in the ZIP container *.xhtml* documents, stylesheets images, TOC, &c;
3. sets an order in which *.xhtml* documents should be read.

It's an error to have "an orphan" foo.xhtml, that isn't listed in the *.opf* but is linked from bar.xhtml.

You may use a regular *.xml* extension instead of *.opf*, but the latter is an old convention chosen back in 1999 to visually identify the package file within the group of source files that make up a book.

While the amount of data that may appear in the *.opf* file is infinitely large, for practical reasons and to save space and time, we are going to focus only on the elements that are (a) absolutely required, and (b) expected because they are widely adopted. Don't add something "just in case", remember that (XML) code is not an asset, but a liability.

The root element of the *.opf* file is `package` that requires:

- 2 namespaces:
  - a default one: `xmlns="http://www.idpf.org/2007/opf"` (IDPF is dead & gone, you'll never see them more—in 2017 the trade association was swallowed by W3C);

    - Dublin Core: `xmlns:dc="http://purl.org/dc/elements/1.1/"`;
- 2 attributes:
    - `version`, the current value is "3.0";
    - `unique-identifier`, despite its name, this is not a unique string that globally identifies the book, but a value for `id` attribute of `package>metadata>dc:identifier` node (that actually specifies a global unique book identifier); set it to a short, readable string, like "package_id", "myid", "bookid", &c;
- 3 children nodes (in this exact order):
    - metadata;
    - manifest;
    - spine.

2.2 <metadata>

Children of this node are Dublin Core elements (`dc:` prefix is the most common), & a special `meta` tag, that augments the meaning of a `dc:NAME` node.

The best thing about Dublin Core is its gallant name; in every other aspect, it's a severe disappointment. Originally created to assist early search engines in resource description, it settled on the lowest common denominator for "cross-disciplinary resource discovery" and describability for "a wide range of resources." It tried to please everyone but proved to serve no one, for it constantly requires fine-tuning by other (non-DC) elements.

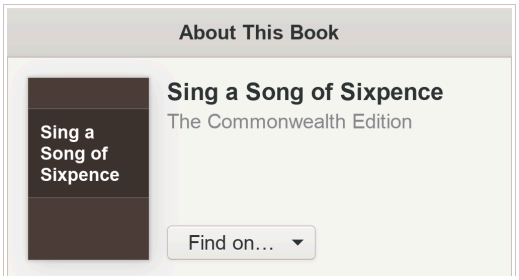
We saw the minimal requirements in the beginning of the chapter:

2.2.1 dc:title

At least 1 is expected, but IRL you oft have a few extra title types as well, "subtitle" & "edition" are quite frequent. DC doesn't know anything about them. You can include multiple `dc:title` elements & augment them with `meta` tag, in hope that a user-agent will recognise your effort:

```
<dc:title id="t0">Sing a Song of Sixpence</dc:title>
<meta refines="#t0" property="title-type">main</meta>
<dc:title id="t1">The Commonwealth Edition</dc:title>
<meta refines="#t1" property="title-type">edition</meta>
```

Supported values for `meta`'s "title-type" are:



- *main*
- *subtitle*
- *short*: an abbreviated form of the *main*; typically used in situations where space is limited, such as on a book cover or in a running header [1];
- *collection*: series of books;
- *edition*: should be used only to distinguish between different editions of the same book;
- *expanded*: a longer version of the *main* that provides more detail or context; may include subtitles or additional descriptive phrases.

2.2.2 dc:language

At least 1 is expected. The value is case insensitive & confusingly called a "language tag" (it has nothing to do with XML tags). The syntax is a list of "tags":

- `language-script-region` (seldom used)
- `language-region`

- `language`

Tag name	*Size, bytes*	*ISO standard*	*Examples*
language	2	639-1	en, uk, pl
script	4	15924	
region	2	3166-1	us, ua, ca

Full valid examples: en-US, uk-UA, en-CA.

2.2.3 dc:identifier

At least 1 is expected & at least 1 of them must refer to the value of package's `unique-identifier` attribute via its own `id` attribute.

To creare a globally unique identifier for the book, a combination of `dc:identifier` & `meta` tag is required:

```
<dc:identifier id="book-id">urn:uuid:47962a91-8e6c-41a9-ad98-b9efb7210dae</dc:identifier>
<meta property="dcterms:modified">2023-02-15T00:00:00Z</meta>
```

Then, if you incorporate errata in the book, you update the contents of the `meta` node, but keep the value of `dc:identifier`. Each new book *edition* should have different `dc:identifiers`, though.

In the above example we used UUID. DC permits any syntactically correct URN strings (see RFC 8141). Popular variants are `urn:isbn` & `urn:doi`.

The value of the `meta` tag is in ISO-8601 format. A quick, portable way of obtaining it is to run:

```
$ ruby -rtime -e 'puts Time::utc(*ARGV).iso8601' 2023 02 15
2023-02-15T00:00:00Z
```

2.2.4 Optional, but recommended

2.2.5 dc:creator & dc:contributor

```
<dc:creator>Mykola Petrenko</dc:creator>
<dc:creator>Olena Petrenko</dc:creator>
<dc:contributor>Jan Kowalski</dc:contributor>
```

Contributor indicates a secondary level of involvement, e.g. an illustrator.

You may leave those nodes as they are, but bookshelf software won't know for sure how to sort the authors' names in the book. Is Mykola a name or a surname?

Again, DC doesn't help at all.

```
<dc:creator id="c1">Mykola Petrenko</dc:creator>
<meta refines="#c1" property="file-as">Petrenko, Mykola</meta>
```

2.2.6 dc:data

Only 1 element is allowed. Represents the date of the Epub publication in ISO-8601 format.

```
<dc:date>2023-02-15T00:00:00Z</dc:date>
```

2.2.7 dc:publisher, dc:rights, dc:subject, dc:source

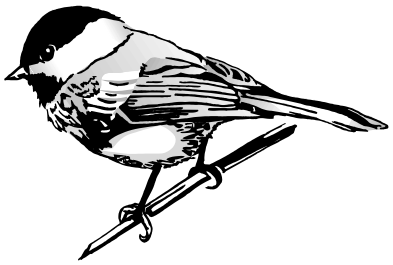
The value of the first 3 is an arbitrary string.

`dc:subject` is intended to be a list of keywords, so more than one element can appear. However, there are examples in the wild from reputable publishers who should have known better, that use a single instance of this node with a bunch of keywords separated by slashes (/) as its value. Don't do that.

`dc:source`'s value *should* (but it's not enforced) contain a URN, & is mainly intended to indicate an ISBN of a print book source, from which the Epub was made. Irrelevant for new books.

2.3 <manifest>

This node lists all files included in a ZIP container (except for mimeinfo, META-INF/*, & *.opf). Each file is represented by an `item` node that accepts at least 3 attributes:



- `href`;
- `media-type`, a MIME type of a file, e.g. "image/svg+xml" for an *.svg* file, or "application/xhtml+xml" for an *.xhtml*;
- `id`, that may **not** be omitted even if the file won't be referenced in a `spine` node;
- `properties`, indicates a additional feature/role this file has.

`properties` value	*Description*	*See chapter*
nav	Contains TOC & landmarks	2
mathml	Contains MathML markup	Ø
svg	Contains *embedded* SVG code	Ø
cover-image	The file is a book cover	3

The order of `item` nodes isn't important.

The quickest way to obtain the MIME type for a particular file is to run:

```
$ alias file.mime='file -b --mime-type'
$ file.mime foo.png
image/png
```

Alas, this returns 'text/plain' for any *.css* file, whereas the correct type should be 'text/css'.

2.4 <spine>

How would a user-agent know which file is the book's default starting position? When you finish reading chapter1.xhtml, how does the user-agent know which .xhtml file to load next?

The answer is it uses a `spine` element, where the default reading order is explicitly enumerated.

E.g., say we have 2 chapters and an appendix:

```
<manifest>
 <item id="file0" href="foo.png" media-type="…" />
 <item id="mytoc" href="mytoc.xhtml" properties="nav" media-type="…" />
 <item id="app1" href="app1.xhtml" media-type="…" />
 <item id="ch1" href="chapter1.xhtml" media-type="…" />
 <item id="ch2" href="chapter2.xhtml" media-type="…" />
</manifest>
```

We want (a) Chapter 1 to be a starting point, (b) appendix to appear after Chapter 2:

```
<spine>
 <itemref idref="ch1" />
 <itemref idref="ch2" />
 <itemref idref="app1" />
</spine>
```

`itemref` may have `linear` attribute whose default value, in its absence, is assumed to be "yes". When it is "no", a user-agent *may* hide the file from the default reading order (some move such files to the very end), but

it will still be accessible if another `itemdef` internally links to it.

Unfortunately, the only items in `spine` that can be referenced are .xhtml & .svg. For all others, even as benign as images, a so-named *fallback* is required.

Say we have a plain text file in `manifest`:

```
<item id="f4" href="humpty-dumpty.txt" media-type="text/plain" />
```

If we include it in `spine`, a linter throws an error akin to "Spine item with non-standard media-type 'text/plain' has no fallback".

Providing a fallback means adding `fallback` attribute with an id of another item, that is presumably an .xhtml/.svg:

```
<item id="f4" href="humpty-dumpty.txt" media-type="text/plain" fallback="f5" />
<item id="f5" href="humpty-dumpty.svg" media-type="image/svg+xml" />
```

Then we can safely add humpty-dumpty.txt to the `spine`:

```
<itemref idref="f4" />
```

Will a user-agent render such an unprecedented text/plain resource is another question (most likely not).

Children of the `spine` are also called *primary content*.

2.5 Table of contents

A TOC cannot be embedded in the *.opf* file—you write it in a separate *.xhtml* & add an entry to `package>manifest` as

```
<item id="mytoc" href="mytoc.xhtml" media-type="application/xhtml+xml" properties="nav"/>
```

& (optionally) to `package>spine`.

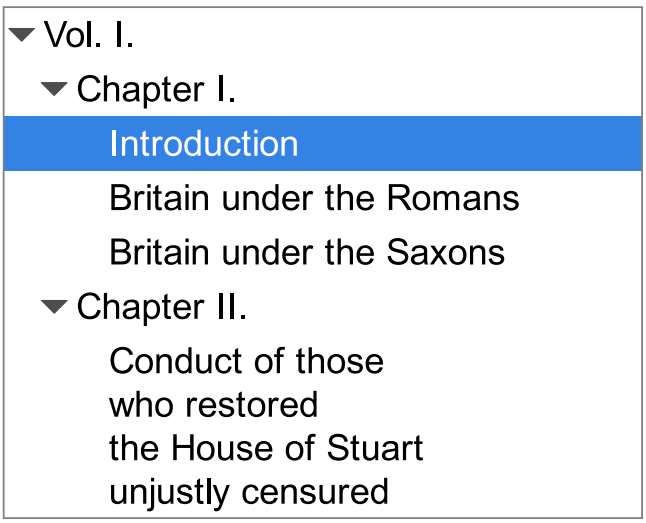
Except for `xmlns:epub="http://www.idpf.org/2007/ops"` namespace declaration, the TOC is a regular *.xhtml* file that uses 2 `nav` elements (direct descendants of `body`) for its domain specific language (DSL).

One of them is TOC itself:

```
<nav epub:type="toc">…</nav>
```

that defines a machine-readable table of contents. The emphasis here is on the "machine-readable" part—by default it's intended to be parsed by a user-agent only, which constructs from it (graphical) UI elements that are accessed independently from the book pages. If the parsing was successful, some user-agents enable a "Table of Contents" button in menus, while others populate a tree view widget & automatically show it to a user.

E.g., an elaborate 3-level structure



Could be written as:

```
<?xml version="1.0" encoding="utf-8"?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:epub="http://www.idpf.org/2007/ops">
<head><title>.</title></head>
<body>
 <nav epub:type="toc">
  <h1>The History of England From The Accession of James II</h1>
  <ol>
   <li>
    <span>Vol. I.</span>
    <ol>
     <li>
      <span>Chapter I.</span>
      <ol>
       <li><a href="ch1.xhtml#s1">Introduction</a></li>
       <li><a href="ch1.xhtml#s2">Britain under the Romans</a></li>
       <li><a href="ch1.xhtml#s3">Britain under the Saxons</a></li>
      </ol>
     </li>
     <li>
      <span>Chapter II.</span>
      <ol>
       <li><a href="ch2.xhtml#s1">Conduct of those
who restored
the <i>House of Stuart</i>
unjustly censured</a></li>
      </ol>
     </li>
    </ol>
   </li>
  </ol>
 </nav>
</body>
</html>
```

In this specific instance, as the screenshot above shows, `<h1>` was striped out completely, `<i>` ignored, & whitespaces in `<li>` (for Chapter II) abided. Also notice the absence of item numbering from `<ol>`—according to the EPUB specification, a user-agent must not show them when applying the TOC data onto its navigational interface.

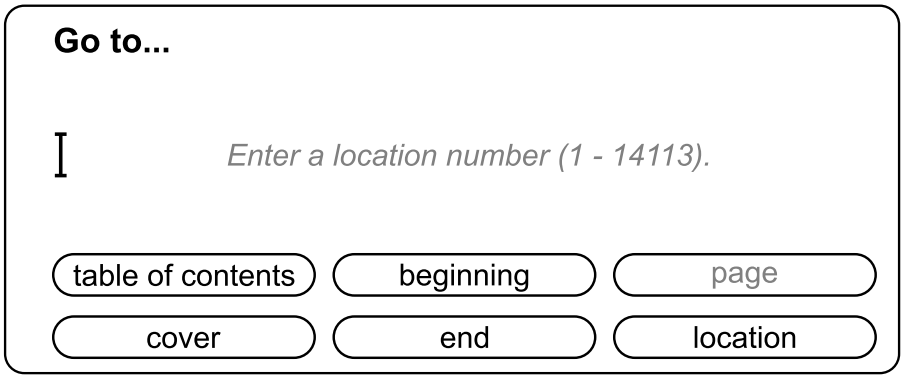
Child nodes of `<nav>` may look like a regular HTML, but this is not a regular HTML, it's a small DSL with strict rules:

Tag name	*Allowed Children (in this order)*
nav	`<h1>-<h6>`, `<ol>`
ol	`<li>`
li	`<span>` *or* `<a>`, `<ol>`
span, a	*phrasing content* (text with markup tags like `<b>`)

I.e., every TOC is a valid HTML, but not every HTML is a valid TOC.

Another `nav` element defines so-called *landmarks*—a simple navigator (usually a couple of buttons) for a set of predefined locations such as an embedded TOC in a book, a starting location, or a list of illustrations. This element is optional, but recommended.

```
<nav epub:type="landmarks">
 <ol>
  <li><a epub:type="toc" href="mytoc.xhtml">Table of Contents</a></li>
  <li><a epub:type="cover" href="cover.xhtml">Cover</a></li>
 </ol>
</nav>
```



Landmark type	*Meaning*
toc	Table of contents
cover	Book cover page
bodymatter	Main content
loi	List of illustrations
lot	List of tables

The trouble with *landmarks* is that user-agents understand only a subset of its types. Usually the first 2 from the table above are recognised, the rest are quietly ignored. Some unadorned user-agents don't offer any interface for *landmarks* at all.

When a user clicks on, say "table of contents" button, the user-agent loads a corresponding .xhtml file. It's your job to make sure it renders in a reasonably pretty fashion. The simplest approach would be to add inline CSS `display: none` rule to *landmarks*:

```
<nav epub:type="landmarks" style="display: none">
```

& add CSS rulesets to change visual appearance for `<ol>`:

```
<style>
  nav li { list-style-type: none; }
  nav ol { padding-left: 1em; }
  nav > ol { padding-left: 0; }
</style>
```

Every anchor in *landmarks* must also has a corresponding entry in `spine`. E.g. if you want to have your TOC

within *landmarks*, but don't want it to appear amidst the regular book content, add it to the *.opf* file with `linear` attribute set to "no":

```
<package … >
 …
 <spine>
  <itemref idref="mytoc" linear="no" />
  …
 </spine>
</package>
```

An *.xhtml* file that contains a TOC with landmarks is called *navigation document*.

Additional reading

- *EPUB Structural Semantics Vocabulary* provides a full list of supported values for `epub:type` attribute.

---

1. Relevant only if the e-book has a printed companion. A "running header" is a short piece of text that appears at the top of each page of the printed book, and includes the title of the book or chapter. ↺

3 Content

The set of possible HTML/CSS features one can use in an EPUB is limited only by the device's rendering engine. Some user-agents use evergreen browser engines, while others ship with web layout components that are several years old. The worst kind use custom HTML/CSS rendering libraries that produce page layouts that only hazily resemble web standards.

Unlike a web browser, a user-agent may disable all CSS rules from an EPUB by default to provide their own, fine-tuned styles that are optimized for novellas, thus discarding your hard presentational work.

The optimum then would be to start with a layout optimised for basic rendering engines & then sprinkle it with CSS, which is easier said than done if the book isn't a simple novel. A variation of a progressive enhancement technique, but for books, is advisable.

Markup that you're expected to put into content files is a subset of evolving HTML features in a strict XML environment. Every *.xhtml* file must be valid XML:

- nodes must be bounded by start- and end- tags;
- nodes must nest properly, with no overlaps;
- attribute values must be quoted;
- empty nodes must be self-closing, e.g., `<br/>`;
- recognised character entities list in XML is a very short one:

Name	*Character*
quot	"
apos	'
lt	<
gt	>
amp	&

  you'll need to convert unacceptable ones to numeric character reference equivalents, e.g. use `&#8531;` for ⅓, because `&frac13;` will raise an error;
- '<', '>' and '&' characters (meant as content) must be escaped as `&lt;`, `&gt;` and `&amp;`.

To do a quick check, run xmllint(1):

```
$ find . -name \*.xhtml | xargs -r xmllint --output /dev/null
```

If find(1) indeed finds *.xhtml* files, but xmllint stays silent, & the exit status is 0, there were no errors.

3.1 Struggle for render under variability of user-agents

EPUBs can be of reflowable or fixed-layout types. This book is about the former. The nature of reflowable

content brings many the same issues a web developer faces every day.

If you want your book to be viewable on devices of any size, both now and in the future, do not rely on known screen dimensions, as the probability of them remaining constant is 0.

Some user-agents prefer a paged layout, while others prefer a "continuous" layout, where the concept of a page does not exist. For the former, the only guaranteed way of obtaining a page break is to put a section of content in a separate *.xhtml* file. CSS *page-break-before* and *page-break-after* attributes may work for some user-agents but mess up navigation for others. In general, it is impossible to guess where the user-agent decides to make a page break, as it depends on the screen and font size.

If you have a list of user-agets in mind that your customers use, try to research what capabilities each user-agent offers. Some manufactures publish official publishing guidelines for their devices. Don't hesitate to ask developers when/if they are planning to support a particular feature (this is especially relevant for developers of boutique applications—the chances of getting a response are higher). Fill bugs for open-source user-agents, otherwise they will never get fixed.

Test each separate section of your book in a regular, evergreen web browser using the same techniques you apply to web applications.

Test on devices emulators & desktop user-agents. The latter may be annoying but still important—if the book is not a novel, but a technical publication, the reader will probably view it on a desktop machine too.

Always check the book with a linter (see Appendix 1).

3.2 Semantics with epub:type

You can use `epub:type` attribute (from `xmlns:epub="http://www.idpf.org/2007/ops"` namespace, the same one that is used in the machine-readable TOC file) to provide structural semantic information to user-agents. Most of the `epub:type` values *in content documents* are ignored by user-agents, & the rest are ex gratia (see Footnotes section below for the example).

This will "semantically enhance" an opening to a document:

```
<section epub:type="preamble">
 <p>
   We the peoples of the United
   Nations determined to save
   succeeding generations from
   the scourge of war
 </p>
</section>
```

Works great.

3.3 Cover page

This can be a standalone image file. Its recommended ratio is

$$r = \frac{H}{W} = 1.6$$

Hence when choosing image dimensions in a vector graphics editor, pick, say, 90 mm for the width & 90*r* = 144 mm for the height.

If you just add some `cover.png` to .opf manifest like

```
<item id="cover" href="cover.png" media-type="image/png" properties="cover-image" />
```

some fraction of user-agents will recognise the intention, but won't allow a reader to *view* the cover page. Maybe they'll render a thumbnail of it somewhere in "about the book" dialog. Adding `itemref` node to the `spine`, that rerefences a bare image without an .xhtml fallback is prohibited, ergo a minimal `cover.xhtml` is still required:

```
<?xml version='1.0'?>
<html xmlns="http://www.w3.org/1999/xhtml" style="height: 100%">
<head><title>.</title></head>
<body style="height: 100%; margin: 0">
 <img src="cover.png" style="display: block; height: 100%; margin: auto"/>
</body>
</html>
```

Both files then must be referenced in multiple sections of the *.opf* file:

```
<package … >
 <metadata>
  …
  <meta name="cover" content="cover" />
 </metadata>
  <manifest>
   <item id="cover" href="cover.png" media-type="image/png" properties="cover-image" />
   <item id="cover_page" href="cover.xhtml" media-type="application/xhtml+xml" />
   …
 <spine>
  <itemref idref="cover_page" linear="no" /> …
 </spine>
</package>
```

& in TOC's *landmarks*:

```
<nav epub:type="landmarks" style="display: none">
 <ol>
  <li><a epub:type="cover" href="cover.xhtml">Cover</a></li>
  …
 </ol>
</nav>
```

3.4 Formulas

If your target device supports MathML, you're golden, otherwise prepare to embed images. For 1-2 formulas you can run a TeX code through a simple converter that outputs an SVG/PNG.

The converter below employs pdflatex & inkscape.

```
$ cat tex2img
#!/usr/bin/make -f

$(if $(f),,$(error "Usage: tex2img f='E=mc^2' output.png"))
dpi := 600
devnull := $(if $(findstring s,$(word 1, $(MAKEFLAGS))),> /dev/null)
inkscape = inkscape --pdf-poppler -Tlp -n 1 $< -o $@

%.pdf:
    pdflatex -jobname "$(basename $@)" -interaction=batchmode '\nofiles\documentclass[borde
r=0.2pt]{standalone}\usepackage{amsmath}\usepackage{varwidth}\begin{document}\begin{varwidt
h}{\linewidth}\[ '$(call se,$(f))' \]\end{varwidth}\end{document}' $(devnull)
    @rm -f "$(basename $@).log"

%.png: %.pdf; $(inkscape) -d $(dpi)
%.svg: %.pdf; $(inkscape)

se = '$(subst ','\'',$1)'
```

It's a makefile disguised for a standalone program. It works by creating a pdf first, than transforming it into an *.svg* or a *.png*, depending on a command line parameter:

```
$ ./tex2img -s f='\gamma = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}' 1.png
```

```
$ xdg-open !$
```

$$\gamma = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}$$

3.5 From HTML to XHTML

If your source files are in the usual HTML, you'll need to convert them to XHTML, preserving whitespaces in `<pre>` & `<code>`:

```
$ cat 5.html
<!doctype html>
<title>.</title>
<link rel="stylesheet" href="foo.css">
Hello,<br>
<code>W  o  r  l  d</code>&excl;

$ ./5-to-xhtml < 5.html
<?xml version="1.0"?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head><title>.</title>
<link rel="stylesheet" href="foo.css" type="text/css"/>
</head>
  <body>Hello,<br/>
<code>W  o  r  l  d</code>!
</body>
</html>
```

With Nokogiri, it's 9 lines of code:

```
$ cat 5-to-xhtml
#!/usr/bin/env ruby

require 'nokogiri'

doc = Nokogiri::HTML5::parse STDIN.read
doc.css(':root>head>link').each do |l|
  l["type"] = "text/css" if l["rel"] == "stylesheet"
end

root = doc.at_css(':root')
root.default_namespace = "http://www.w3.org/1999/xhtml"

puts '<?xml version="1.0"?>'
puts doc.to_xhtml
```

Besides adding a default namespace, we modify `link` nodes (if they refer to a stylesheet) by adding "type" attribute, for some user-agents don't recognise such `link`s without it.

Notice that `&excl;` entity, that is invalid in XML, got auto-converted to '!'.

3.6 Fonts

Think twice before including custom fonts. Makers of dedicated e-book readers have spent a significant amount of time choosing fonts best suited for their devices; when you come up with a bespoke a set of glyphs, it'll most definitely worsen the user experience.

To add a font:

1. append an `item` to the .opf manifest with the font file;
2. write @font-face CSS ruleset.

Format	*File extension*	*media-type*
TrueType	.ttf	font/ttf
OpenType	.oft	font/otf
WOFF2	.woff2	font/woff2

Suppose a target device uses a buck ugly monospace font or you want some code snippets to be slightly "narrower" without resorting to the smaller font size. A variant of Inconsolata is an example of a free font that fits the bill. [1]

Add the font reference to the package file:

```
<package … >
   …
 <manifest>
  <item id="fn1" href="ch03/Inconsolata-CondensedSemiBold.ttf" media-type="font/ttf"/>
   …
 </manifest>
   …
</package>
```

To CSS:

```
@font-face {
  font-family: Inconsolata;
  src: url(ch03/Inconsolata-CondensedSemiBold.ttf);
}

pre.narrow {
  font-family: Inconsolata, monospace;
  line-height: 1;
}
```

& use in .xhtml:

```
<pre class="narrow">
int matrix[4][4] = {
  { 1,  2,  3,  4},
  { 5,  6,  7,  8},
  { 9, 10, 11, 12},
  {13, 14, 15, 16}
};
</pre>
```

3.7 Footnotes

There's no special provisions for footnotes, you may organise them in any form you desire, but ~popular user-agents have managed to come up with a convention that helps EPUB creators display footnotes in a popup window. E.g.

```
<p>… on the banks of the Sha-ho <sup><a href="#ft-1">1</a></sup></p>
```

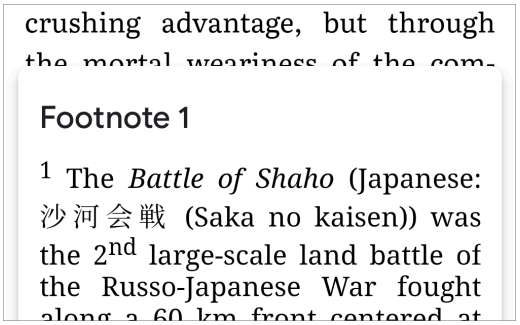
& somewhere in the end of a chapter (technically creating an *endnote*):

```
<aside id="ft-1">1. The <i>Battle of Shaho</i> … </aside>
```

would instruct a user-agent to find a node with "ft-1" id, extract its contents, make a popup window & inject the obtained node's contents into it.

For the reader's convenience, especially on devices that don't draw footnote popups, make footnotes bi-directional (the text is linked to the footnote and the footnote is linked back to the text).

Sometimes you'll see `epub:type="noteref"` attribute added to `<sup>` tags & `epub:type="footnote"` to `<aside>`, but for the most user-agents they are unnecessary.


crushing advantage, but through the mortal weariness of the com-

Footnote 1

[1] The *Battle of Shaho* (Japanese: 沙河会戦 (Saka no kaisen)) was the 2nd large-scale land battle of the Russo-Japanese War fought along a 60 km front centered at

3.8 Hyphenation

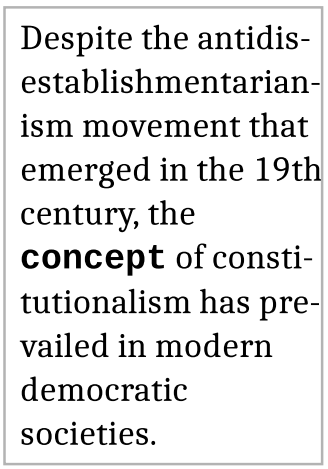
In the past, the only way get a hyphenated paragraph in an EPUB was to insert soft hyphens & hope that the user-agent would act accordingly after seeing them. A soft hyphen is an invisible symbol (`&#173;`) that is used to indicate a point within a word where a hyphenated break is permitted. This required a post-processing tool. [2] Some bookshelf software did such hyphenation when copying EPUBs to an e-book reader.

Another approach was to use *hyphens* CSS property the same way it's used on the Web:

```
<?xml version="1.0"?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
 <style>
  body { hyphens: auto; }
  code { hyphens: none; }
 </style>
</head>
<body lang="en" style="width: 8em">
<p>
 Despite the
 antidisestablishmentarianism
 movement that emerged in
 the 19th century, the
 <b><code>concept</code></b>
 of constitutionalism has
 prevailed in modern democratic
 societies.
</p>
</body>
</html>
```

Then an advanced user-agent would hyphenate `<p>`, using the language hint from `lang` attribute, but leave alone the contents of `<code>` element.

The 3rd & final technique, that has emerged as the most sophisticated one, is to do nothing. Many user-agents try to hyphenate book contents automatically, as if the book already has `hyphens` property set to "auto". It became a user setting. This implies (a) your language metadata in the *.opf* file is correct, (b) if there are chunks of text not in the default book's language, they must be explicitly marked with `lang` attribute.


Despite the antidisestablishmentarianism movement that emerged in the 19th century, the **concept** of constitutionalism has prevailed in modern democratic societies.

3.9 TOC

If you decide not to include a machine-readable TOC into the `spine` due to the rigid structure of `<nav epub:type="toc">` element, but you still want to have a TOC amidst the regular book content, it means you'll have to construct the TOC *twice*. Sometimes books have several TOCs: in addition to a regular one with subheads within chapters (h1-h3), there is an abridged h1-version. All this adds complexity, & the more complexity you introduce, the more errors creep in.

Too detailed a TOC or multiple TOCs in an EPUB is an atavism: printed media couldn't have search, the TOC (& often an index) was the only way to get around the problem of navigation, providing valuable signposts for readers. Any half decent user-agent can do a sequential text search.

3.10 JavaScript

Although the EPUB specification allows the usage of JavaScript in a book, in practice, user-agents either don't support JS at all, or turn it off by default, marking scripts as "unsafe content". This is done for a good reason.

A team that writes a modern web browser consists of 500-1000 programmers. Most people find these numbers astonishing, not realising the level of complexity of the web browser.

Every browser team has a security squad. Those are not guys with flash-lights, tamper-proof keyrings, guns, & tough facial expressions. When a browser reads a chunk of JavaScript, bytecompiles it & starts executing, it does it in a sandbox environment. No matter how evil a peace of JS code may be, it should never break out of the sandbox & gain access to a user's data outside of its origin.

Back to EPUB user-agents. As you can guess, the size of their "security team" is usually 0, unless they're a software provider for a major e-book manufacturer. By not allowing JavaScript in EPUBs, the number of possible attack vectors gets drastically smaller.

It would also be nice not to repeat the mistakes of word processing software, where after introducing scripting capabilities, any incoming document became a potential threat. So far EPUBs are "safe", for a reader doesn't expect them to be able to induce a user-agent to run unknown code.

Another reason not to use JavaScript in a book is existential: what for? There was a time in the 1990s when interactive encyclopædias were all the rage—no less than a revolution in education. Hundreds of millions of dollars were spend on "multimedia" content & its distribution—all to quietly die off.

3.11 Code listings

Because of the über-narrow viewport, many publishers choose to insert images instead of `<pre>` blocks. If you do this, at least have the decency to use SVGs. There is no excuse for raster images in code examples whatsoever.

Using an *.svg* is acceptable when you don't expect listings to be copy-pasted. E.g., all the captured hexdump(1) output from Chapter 1 went through Pango text viewer:

```
$ pango-view -q --font=Liberation\ Mono --margin=0 -o out.svg input.txt
```

A more interesting task is to make `<pre>` blocks useful regardless of a screen size. When a line of code exceeds the viewport width, it needs to be wrapped. Oftentimes, though, there is confusion: is the line wrapped or is it the next line of code?

We can split a multi-line contents of a `<pre>` node & put each line into separate `<code>` tag:

```
<pre>
<code>int strlen(char *s) {</code>
<code> char *p = s;</code>
<code> while (*p) p++;</code>
<code> return p - s;</code>
<code>}</code>
</pre>
```

Then, setting a CSS property *border-left* for the latter would unambiguously mark the beginning of the line. If the line wraps, there would be no border on the left side of it.

Styling for this is not much of a spectacle, except for putting the border into negative space. If a user-agent doesn't understand this, we sacrifice only 4 pixels for an attempt.

```
pre > code {
  padding-left: 4px;
  margin-left: -4px;
```

```
  white-space: pre-wrap;
  hyphens: none;
  border-left: 1px solid gray;

  word-break: break-all;
  overflow-wrap: normal;
}
```

Some would stigmatise this approach as nonstandard.

3.12 Style for novels

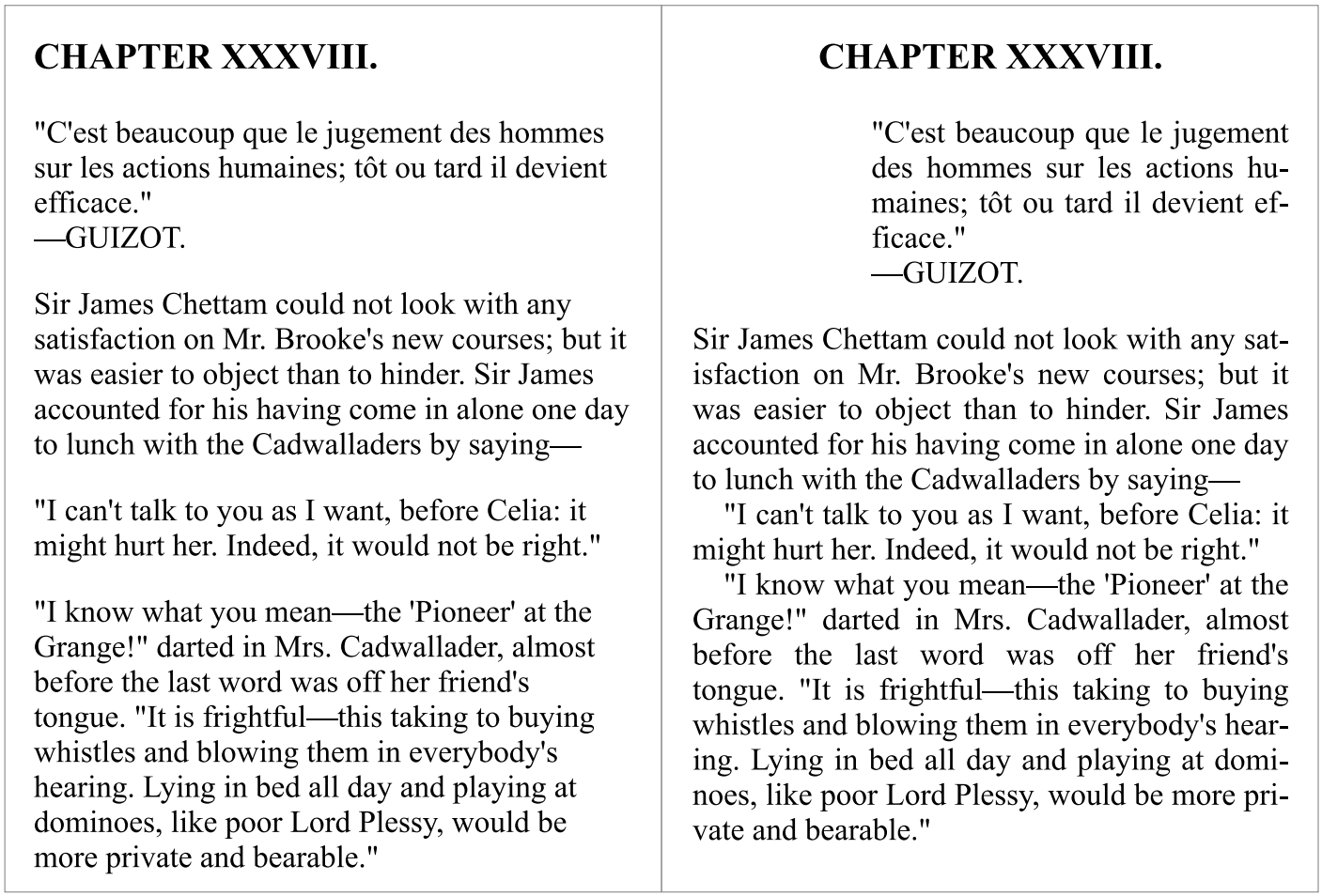
Novel-like rendering means no margin (or a very small one, ≈ 0.2em) between paragraphs. The first line of each paragraph, except for the first paragraph or those after images, tables, &c, should have a text-indent. If a paragraph serves an auxiliary role like an epigraph, it retains the default margin.

This formatting style has been common for several hundred years, a reader expects it when buying a book. The web, however, has established a different behaviour with a blank line before a paragraph (via collapsible margins in <p>), & no first-line text-indent.

Don't invent your own style. If you dislike the classical pre-web one, it may be better to avoid any styling altogether.

In following page example, with the most humble markup:

```
<?xml version="1.0" encoding="utf-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
 <head/>
 <body lang="en">
  <h3>…</h3>
  <p class="epigraph" lang="fr">…</p>
  <p>…</p>
  <p>…</p>
  <p>…</p>
 </body>
</html>
```



CHAPTER XXXVIII.

"C'est beaucoup que le jugement des hommes sur les actions humaines; tôt ou tard il devient efficace."
—GUIZOT.

Sir James Chettam could not look with any satisfaction on Mr. Brooke's new courses; but it was easier to object than to hinder. Sir James accounted for his having come in alone one day to lunch with the Cadwalladers by saying—

"I can't talk to you as I want, before Celia: it might hurt her. Indeed, it would not be right."

"I know what you mean—the 'Pioneer' at the Grange!" darted in Mrs. Cadwallader, almost before the last word was off her friend's tongue. "It is frightful—this taking to buying whistles and blowing them in everybody's hearing. Lying in bed all day and playing at dominoes, like poor Lord Plessy, would be more private and bearable."

CHAPTER XXXVIII.

"C'est beaucoup que le jugement des hommes sur les actions humaines; tôt ou tard il devient efficace."
—GUIZOT.

Sir James Chettam could not look with any satisfaction on Mr. Brooke's new courses; but it was easier to object than to hinder. Sir James accounted for his having come in alone one day to lunch with the Cadwalladers by saying—

"I can't talk to you as I want, before Celia: it might hurt her. Indeed, it would not be right."

"I know what you mean—the 'Pioneer' at the Grange!" darted in Mrs. Cadwallader, almost before the last word was off her friend's tongue. "It is frightful—this taking to buying whistles and blowing them in everybody's hearing. Lying in bed all day and playing at dominoes, like poor Lord Plessy, would be more private and bearable."

the variant with the absence of styling is on the left. The minimal CSS for a novel-like presentation would be:

```
body { hyphens: auto; }
h3 { text-align: center }
p { text-align: justify; }
p:not([class]) { margin: 0; }
p:not([class]) + p:not([class]) {
  text-indent: 1em;
}
p.epigraph { margin-left: 30%; }
```

Additional reading

- *Notes for new Make users* by yours trully may provide insights on how the makefile from Formulas section works.

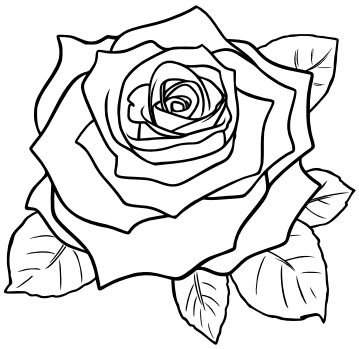
Exercises

1. `5-to-xhtml` program outputs correct XHTML only if you feed it with HTML. Fix it so it could work on either input.

---

1. The font is licensed under the Open Font License by Raph Levien. ↺
2. See github.com/gromnitsky/epub-hyphen for a command line example of such a tool. ↺

4 Automatrons



> SGML and XML are good ideas if you were comatose when you designed your software.
> — Erik Naggum, *comp.lang.lisp*, 22 Jul 2001

Unless you're digitising an XVIII century novel, working on a "manuscript" could require some (or a lot of) automation. Most of it can be foregone if a literary work is of la narrativa di genere, but for (pseudo) technical publications no automation means wasted time & a plethora of unnecessary errors.

4.1 <metadata> in .opf

In dealings with an EPUB package file, there 2 schools of thought:

1. write everything in a *.yaml* or *.json* file, then write a program that reads such configuration & generates an *.opf* file;
2. write the *.opf* file using a templating language, like Ruby's ERB.

The 1st choice is very attractive for it disentangles the end user from XML completely. We'll, of course, pursue the 2nd one for it's a book about the EPUB format, & not a guide on how to write a successful or maintainable software.

The meta tag that describes the last modified date in an obvious candidate for automation. Take your *.opf* file (`foo.opf` from the beginning of Chapter 2 will do), & replace the tag contents with Ruby code enclosed in `<%= … %>`:

```
$ cat metadata.fragment.xml
<% require 'time' %>
<meta property="dcterms:modified">
 <%= Time.now.utc.iso8601 %>
</meta>

$ erb metadata.fragment.xml | grep .
<meta property="dcterms:modified">
 2023-03-01T13:51:47Z
</meta>
```

ERB tags	*Meaning*
`<% … %>`	Evaluate Ruby code & expand to an empty string
`<%= … %>`	Replace with result of evaluation
`<%# … %>`	A comment ('`<% #`' doesn't work)
`<%% or %%>`	Replace with `<%` or `%>`

4.2 <manifest> & <spine> in .opf

If you decide to automate one thing, automate the generation of `items` in `package>manifest` node. Writing/maintaining it by hand is insane, unless, again, the book is a novel with 0 illustrations & fixed beforehand amount of chapters.

Rename your *.opf* file to `package.opf.erb`. Next, define all posible mime types of files we have in our book:

```
<%
mime = {
  ".xhtml" => "application/xhtml+xml",
  ".png" => "image/png",
  ".gif" => "image/gif",
  ".svg" => "image/svg+xml",
  ".css" => "text/css",
  ".ttf" => "font/ttf"
}
%>
```

Then we need a data type to hold our files. Using RBS (Ruby Signature) notation it can be described as:

```
type item = {id: String, mime: String}
files: Hash[String, item]
```

E.g., adding a valid item to `files` hash would look like:

```
files = {}
files["foo.css"] = {id: "f0", mime: "text/css"}
```

Having such a hash, we can later write

```
<itemref idref="<%= files['ch01.xhtml'][:id] %>" />
```

when adding children to `package>spine` element.

The procedure of collecting files for `package>manifest` node is as follows:

1. obtain an array of file names using `Dir.glob`;
2. iterate through the array
    - filtering out files we don't want (like "META-INF/container.xml");
    - adding files to `files` hash;
    - outputing `<item>`s.

```
<%
files = {}
ignore = ["package.opf", "toc.xhtml", "META-INF/container.xml"]
%>

<manifest>
 <item id="toc" href="toc.xhtml" media-type="application/xhtml+xml" properties="nav"/>
<%
Dir.glob("**/*", base: out).each_with_index do |file, idx|
  next if File.directory? File.join(out, file)
  next if ignore.index file
  files[file] = {id: "f#{idx}", mime: mime[File.extname(file)]}
%>
<item id="<%= files[file][:id] %>" href="<%= file %>" media-type="<%= files[file][:mime] %>"
/>
<% end %>
</manifest>
```

We ignore `toc.xhtml` in `Dir.glob` block, for it's a static resource, & we're adding it manually to set `properties` attribute.

Then in `spine`:

```
<spine>
 <itemref idref="toc" linear="no" />
 <itemref idref="<%= files['ch01.xhtml'][:id] %>" />
 …
<spine>
```

To generate `package.opf` from our *.erb* template run:

```
$ erb out=_out package.opf.erb > _out/package.opf
```

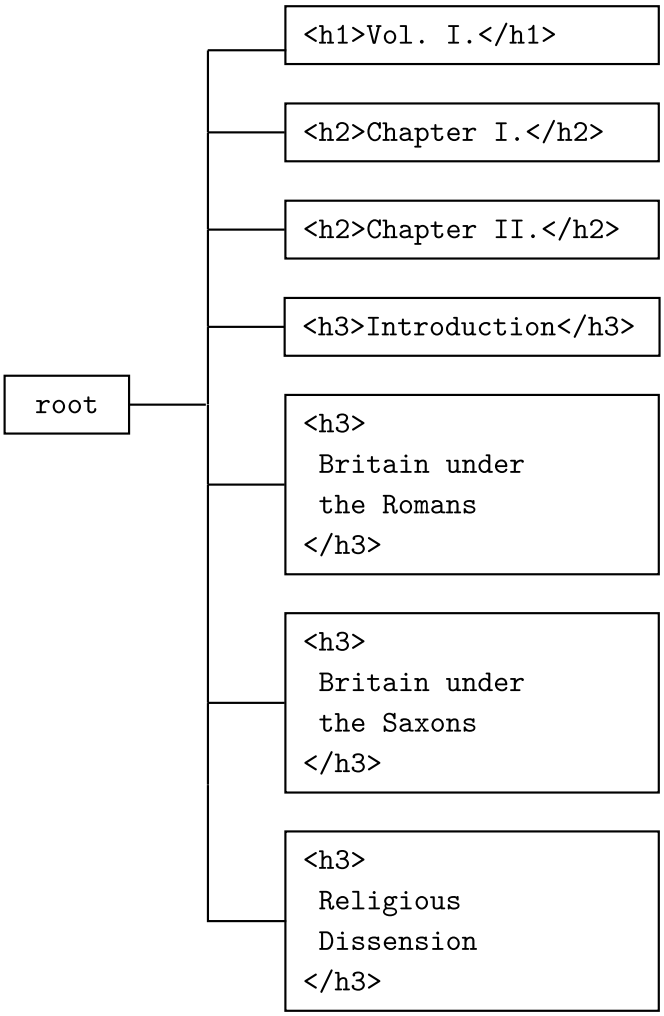
`out=_out` sets a value for the local variable `out` that we use in `Dir.glob` invocation as the base directory for interpreting relative pathnames instead of the current working directory.

4.3 Generating TOC

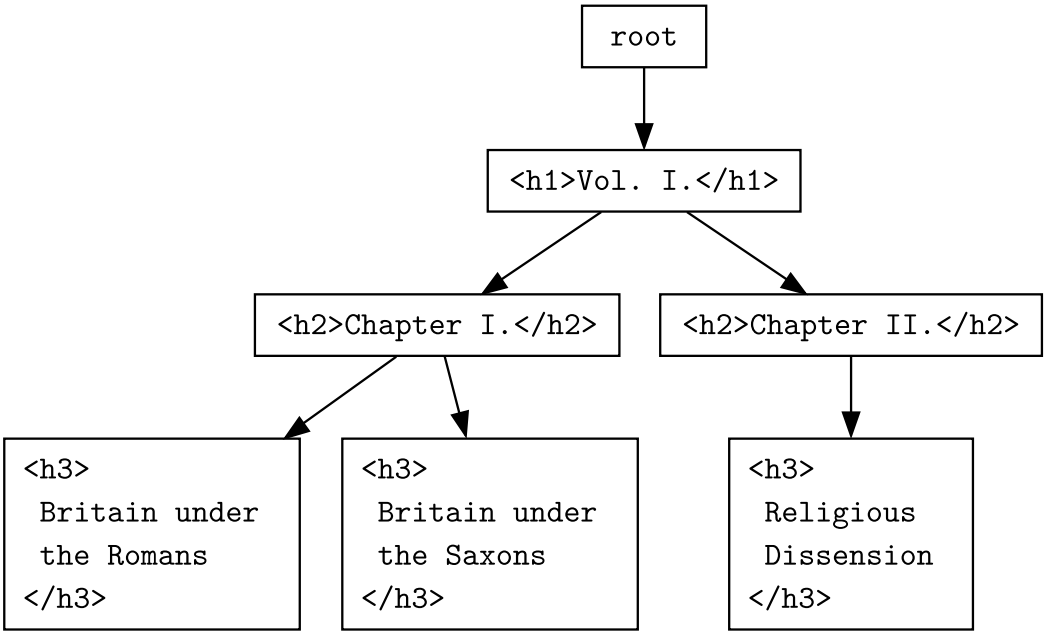
If your TOC is 1 level deep, life is easy, you scan though your *.xhtml* extracting `<h1>`. It becomes trickier when you want a hierarchical composition.

The trouble with reconstructing a graph from *.xhtml* files is that after grabbing `<h1>-<h6>` section headers, the height of a resulting rooted tree would always be 1:

```
<h1 id="v1">Vol. I.</h1>
<h2 id="ch1">Chapter I.</h2>
<h3 id="intro" class="unlisted">Introduction</h3>
<h3 id="romans">Britain under the Romans</h3>
<h3 id="saxons">Britain under the Saxons</h3>
<h2 id="ch2">Chapter II.</h2>
<h3 id="rel">Religious Dissension</h3>
```



Those headers are all sibling nodes. What we want is:



Having such a tree, we can easily output a correct TOC structure. A program we are going to write reads *.xhtml* files & prints EPUB-compatible list items for `<ol>` element:

```
$ cat history.html
<body>
  <h1 id="v1">Vol. I.</h1>
  <h2 id="ch1">Chapter I.</h2>
  <h3 id="intro" class="unlisted">Introduction</h3>
  <h3 id="romans">Britain under the Romans</h3>
  <h3 id="saxons">Britain under the Saxons</h3>
  <h2 id="ch2">Chapter II.</h2>
  <h3 id="rel">Religious Dissension</h3>
</body>

$ ./toc . history.html
<li><a href='history.html#v1'>Vol. I.</a>
<ol>
  <li><a href='history.html#ch1'>Chapter I.</a>
  <ol>
    <li><a href='history.html#romans'>Britain under the Romans</a></li>
    <li><a href='history.html#saxons'>Britain under the Saxons</a></li>
  </ol>
  </li>
  <li><a href='history.html#ch2'>Chapter II.</a>
  <ol>
    <li><a href='history.html#rel'>Religious Dissension</a></li>
  </ol>
  </li>
</ol>
</li>
```

Having such an utility allows us to write

```
<nav epub:type="toc">
 <h1>Contents</h1>
 <ol>
<%= `./toc #{out} #{out}/ch*/index.xhtml` %>
<%= `./toc #{out} #{out}/backmatter/*.xhtml` %>
 </ol>
</nav>
```

in book's `toc.xhtml.erb`, generating the real TOC as:

```
$ erb out=_out toc.xhtml.erb > _out/toc.xhtml
```

We need a tree structure. For our purposes, its only capabilities are (a) adding a child node, (b) checking its level (presumably 1-6), a link to its parent node, & if the node should be ignored later on ("unlisted"):

```
class Header
  def initialize orig
    @text = orig.text
    @hid = orig['id']
    @level = orig.name[1].to_i
    @unlisted = orig.classes.index "unlisted"
    @parent = nil
    @kids = []
  end
  attr_accessor :level, :parent, :unlisted

  def add_kid header
    @kids.push header
    header.parent = self
    header
  end

  def self.fakenode(name = 'h0', text = "", id = "", classes = [])
    n = { 'id' => id }
    n.define_singleton_method(:name) {name}
    n.define_singleton_method(:text) {text}
```

```
    n.define_singleton_method(:classes) {classes}
    n
  end
end
```

`orig` in the constructor is presumably a `Nokogiri::XML::Node`.

Then to create an h1 node:

```
fake_h1 = Header.fakenode "h1", "Vol. I.", "v1"
h1 = Header.new fake_h1
```

Next, when we obtain an array of Header objects, the next step is to make a tree using them. E.g., by adding a class method to `Header`:

```
def self.MkTree root, headers
  last = root

  headers.each do |header|
    if header.level > last.level
      last = last.add_kid header
    else
      p = last
      p = p.parent while header.level != p.level+1
      last = p.add_kid header
    end
  end

  root
end
```

The root node akin to

```
root = Header.fakenode
```

is required, for if we read multiple *.xhtml* files, there can be several h1 nodes in them.

Moving from somewhat abstract `Header`s to concrete ones, each of which doesn't live in a vacuum, but exists in a file, we need a variation of `Header` object that (a) holds additional metadata (file name, base directory) & (b) is able to render itself with its children:

```
require 'cgi'

class TOC_Header < Header
  def initialize orig, file, base
    super(orig)
    @file = file; @base = base
  end

  def link = Pathname.new(@file).relative_path_from(@base).to_s + "\##{@hid}"

  def to_s indent = 0
    return @kids.map {|kid| kid.to_s indent}.join "\n" if @level == 0
    return "" if @unlisted

    prefix = " " * indent*2
    a = prefix + "<li><a href='#{link}'>#{CGI.escapeHTML @text}</a>"
    return a+"</li>" if @kids.size == 0

    r = [a]
    if @kids.size > 0 && !@kids.all? {|kid| kid.unlisted}
      r << "#{prefix}<ol>"
      r << @kids.map {|kid| kid.to_s indent + 1}
      r << "#{prefix}</ol>"
    end
```

```
    r << "#{prefix}</li>"
    r.join "\n"
  end
end
```

`TOC_Header#to_s` is somewhat long & a bit hairy, but it uses recursion, therefore its efficient code is easy to read & understand. (No.)

To finish the utility we grab h1-h3 from provided *.xhtml* files, make a bunch `TOC_Header`s from them, create an h0 fake root, call `Header.MkTree` to construct the tree of `TOC_Header`s:

```
require 'nokogiri'

abort "Usage: #{$0} base_dir file1.xhtml …" if ARGV.size < 2

def nodes files
  files.map do |file|
    doc = Nokogiri::XML File.read file
    doc.css('h1,h2,h3').map do |v|
      TOC_Header.new v, file, ARGV[0]
    end
  end.flatten
end

root = TOC_Header.new Header.fakenode, "", ""
Header.MkTree root, nodes(ARGV[1..-1])
puts root
```

4.4 Postprocessors

The rest of the chapter involves writing small programs that we call *postprocessors*. They are all intended to be used in a pipeline

```
$ cat orig.xhtml | foo1 | foo2 | foo3 > result.xhtml
```

& they all adhere to the common scheme:

1. read the stdin;
2. parse it as XML;
3. find some nodes, transform their contents;
4. serialise everything back to a string & write it to the stdout.

Every program is an Ruby script that employs Nokogiri for XML manoeuvring.

The concept of a *postprocessor* in an homage to troff(1) in the sense that, in the similar way troff combines its different *pre*processors in a pipeline to transform certain parts of a document into troff input, we can combine different XML *post*processors in the pipeline, where each of them modifies only a specific chunk of a document.

4.5 Numbering sections

Sometimes it's done via CSS counters locally in each "chapter", in which case there is nothing to automate, but with such an approach, you'll have to either manually or automatically number sections in a TOC. When parsing the TOC, a user-agent cares only about its hierarchical structure, it doesn't apply CSS to it (at least you should not expect it would do it).

Having a postprocessor that reads an *.xhtml* file and modifies 'h1-h3' headers in-place (embeds numbers in them) allows the program `toc` we wrote earlier to automatically pick up headers with numbers, eliminating any manual work.

Say there is a file

```
$ cat essay.html
<body>
 <h1 id="n">An Essay</h1>
 <h2 id="i" class="unnumbered">Introduction</h2>
 <h2 id="b">Body</h2>
 <h3 id="b1">Evidence</h3>
 <h3 id="b2">Examples</h3>
 <h3 id="b3">Arguments</h3>
 <h2 id="c">Conclusion</h2>
</body>
```

It has is a section with "unnumbered" class that should be ignored during the section numbering. Now, when we produce a new file with slightly different headers:

```
$ cat essay.html | ./headers-numbering 7 > _out/essay.xhtml
$ cat !$
<?xml version="1.0"?>
<body>
 <h1 id="n">7 An Essay</h1>
 <h2 id="i" class="unnumbered">Introduction</h2>
 <h2 id="b">7.1 Body</h2>
 <h3 id="b1">7.1.1 Evidence</h3>
 <h3 id="b2">7.1.2 Examples</h3>
 <h3 id="b3">7.1.3 Arguments</h3>
 <h2 id="c">7.2 Conclusion</h2>
</body>
```

the TOC naturally gets these injected numbers too:

```
$ ./toc _out _out/essay.xhtml
<li><a href='essay.xhtml#n'>7 An Essay</a>
<ol>
  <li><a href='essay.xhtml#i'>Introduction</a></li>
  <li><a href='essay.xhtml#b'>7.1 Body</a>
  <ol>
    <li><a href='essay.xhtml#b1'>7.1.1 Evidence</a></li>
    <li><a href='essay.xhtml#b2'>7.1.2 Examples</a></li>
    <li><a href='essay.xhtml#b3'>7.1.3 Arguments</a></li>
  </ol>
  </li>
  <li><a href='essay.xhtml#c'>7.2 Conclusion</a></li>
</ol>
</li>
```

We reuse `Header` class from the `toc` program (put it in a separate file named `lib.rb`). This time the variation of `Header` object needs a CSS class name of a node & an array to hold section numbers, collected from the root node downwards.

```
require 'nokogiri'
require_relative './lib'

abort "Usage: #{$0} start_number < file.html" if ARGV.size != 1

class NumberedHeader < Header
  def initialize orig
    super(orig)
    @orig = orig
    @unnumbered = orig.classes.index "unnumbered"
    @number = []
  end
  attr_accessor :unnumbered, :number, :orig

  def number! start
    kids = @kids.filter {|kid| !(kid.unnumbered || kid.unlisted)}
    return kids.each_with_index {|kid, idx| kid.number! start+idx} if @level == 0
    @number = [@parent.number, start].flatten
```

```
    kids.each_with_index {|kid, idx| kid.number! idx+1}
  end

  def text_numbered
    @number.size > 0 ? "#{@number.join "."} #{@text}" : @text
  end
end
```

Again, `orig` argument in the constructor is expected to be `Nokogiri::XML::Node` object; from that object we get all required data, & a reference to it we save in `@orig`.

`number!` method fills `@number` array using recursion, ignoring objects that shouldn't be numbered; `text_numbered` method returns a new "pretty" section name.

To tie the program together we need to collect actual headers & create a fake `Nokogiri::XML::Node`-like root node:

```
start_number = ARGV[0].to_i
if start_number < 1
  STDOUT.write STDIN.read
  exit
end

doc = Nokogiri::XML STDIN.read
headers = doc.css('h1,h2,h3').map {|v| NumberedHeader.new v }

root = NumberedHeader.new Header.fakenode
Header.MkTree root, headers
root.number! start_number
```

The final steps are to modify content of the `Nokogiri::XML::Node` elements through the references in `NumberedHeader#orig`, then serialise:

```
headers.each do |hdr|
  hdr.orig.content = hdr.text_numbered
end

puts doc.to_xml
```

4.6 Formulas

If you settled on having images where TeX equations should've been, the simplest approach would be writing a program that scans for

```
<p class="formula">
  \gamma = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}
</p>
```

nodes & feeds the content of every such node to a TeX-compatible engine that outputs an SVG or a PNG image. Having the image, we can either inject it directly into the *.xhtml* file (replacing the `<p>` node) or put an `<img>` with a link to it.

Inlining an *.svg* may seem like the most attractive option, until you test the resulting EPUB with user-agents. In the example below, we'll write an postprocessor, that swaps `p.formula` nodes with `<img>` tags.

```
$ cat ch1.xhtml | ./inline-formulas _out svg > _out/ch1.xhtml
```

Here `inline-formulas` is our script that (a) outputs SVG images (if any) into `_out` directory, & (b) prints to the stdout the modified XHTML.

```
$ cat inline-formulas
#!/usr/bin/env ruby
```

```
require 'nokogiri'
require 'digest'

def img style, tex, path
  doc = Nokogiri::XML::DocumentFragment.parse ''
  node = Nokogiri::XML::Node.new('img', doc)
  node['style'] = style if style
  node['alt'] = tex
  node['class'] = 'formula'
  node['src'] = path
  node
end

abort "Usage: #{$0} out_dir ext < file.xhtml" if ARGV.size != 2

doc = Nokogiri::XML STDIN.read
doc.css('p.formula').each_with_index do |node, idx|
  tex = node.content.gsub(/\s+/, ' ').strip
  job = "formula.#{Digest::SHA1.hexdigest(ARGV[0])[0...7]}.#{idx}"
  image = File.join ARGV[0], "#{job}.#{ARGV[1]}"
  log = File.join ARGV[0], "#{job}.log"
  fail "tex2img failed, see #{log}" unless system './tex2img', '-Bs', image, "f=#{tex}"
  node.replace img(node['style'], tex, File.basename(image))
end

puts doc.to_xml
```

It runs `tex2img` converter (from Chapter 3), replacing `p.formula` nodes with `<img>` tags that point to image files produced by tex2img.

```
job = "formula.#{Digest::SHA1.hexdigest(ARGV[0])[0...7]}.#{idx}"
```

line seems like an unnecessary complex way to construct an image file name, but it serves 2 goals: (a) to make a unique file name for a ZIP container, & (b) while doing so be static, i.e. don't generate a new file name every time you run the converter. Why bother with (a) & (b)? Suppose we had

```
job = "formula.#{idx}"
```

& the generated formulas for the whole book looked like:

```
ch1/formula.0.svg
ch1/formula.1.svg
ch2/formula.0.svg
```

then some notorious user-agents would do a clever "optimisation" & resolve that `ch1/formula.0.svg` == `ch2/formula.0.svg`. Sure, why not, they have the same base name, let's ignore `ch2/formula.0.svg` & render `ch1/formula.0.svg` instead when viewing the second chapter. No.

It's customary for mathematical expressions to be drawn on a separate line clear of text. You can start with a simple CSS that centers formulas & sets their height:

```
img.formula {
  display: block;
  margin: 0.5em auto;
  width: auto;
  height: 3em;
}
```

If symbols in a particular equation render too small, set inline styles on the `p` node & `inline-formulas` will transfer them to the resulting `img`:

```
$ echo '<p class="formula" style="height: 5em">P = NP</p>' | ./inline-formulas . png
```

```
<?xml version="1.0"?>
<img style="height: 5em" alt="P = NP" class="formula" src="formula.3a52ce7.0.png"/>
```

4.7 Code listings

In Chapter 2 we contrived a way of splitting `<pre>` blocks into an array of `<code>` tags. Now we can automate the process:

```
$ printf '<pre><code>1\n2<code></pre>' | ./snippets | xmllint -format -
<?xml version="1.0"?>
<pre>
  <code>1</code>
  <code>2</code>
</pre>
```

This is probably one of the easiest postprocessors we can write.

```
#!/usr/bin/env ruby

require 'nokogiri'
include Nokogiri::XML

doc = Nokogiri::XML STDIN.read
doc.css('pre > code').each do |node|
  lines = node.content.split "\n"
  ns = NodeSet.new Document.new
  lines.each_with_index do |line, idx|
    f = DocumentFragment.parse ''
    nl = Text.new "\n", f
    code = Node.new 'code', f
    code.content = line
    ns << code
    ns << nl unless lines.size-1 == idx
  end

  node.parent.children = ns
end

puts doc.to_xml
```

Notice that it tries not to insert a redundant newline in the end of `<pre>` blocks.

4.8 Footnotes

Due to the way user-agents have agreed on recognising `<sup>` tags with numbers as footnotes, writing even a couple of them is tedious, mainly by the necessity of maintaining (back) referencing to/from `<aside>` tags.

Imagine if we could just write

> **<p>**
> a paragraph with a tag called **<footnote>***the contents of which*</**footnote**> gets auto-converted to `aside` element, moves in the end of the document, & in its place a proper `sup` element appears.
> </**p**>

```
$ cat sixpence.html
<body>
Sing a song of sixpence,
<footnote>£0.025 or ½ s.</footnote><br/>
A pocket full of rye.
<footnote>A cereal grain like wheat.</footnote><br/>
Four and twenty blackbirds<br/>
```

```
Baked in a pie.
<footer />
</body>
```

The result of 'cat sixpence.html | ./footnotes' viewed as an XML file in a web browser:

```
<body>
  Sing a song of sixpence,
  <sup id="ft-ref-0">
    <a href="#ft-0">1</a>
  </sup>
  <br/>
  A pocket full of rye.
  <sup id="ft-ref-1">
    <a href="#ft-1">2</a>
  </sup>
  <br/>
  Four and twenty blackbirds
  <br/>
  Baked in a pie.
  <footer>
    <aside id="ft-0">
      1. £0.025 or ½ s.
      <a href="#ft-ref-0">↵</a>
    </aside>
    <aside id="ft-1">
      2. A cereal grain like wheat.
      <a href="#ft-ref-1">↵</a>
    </aside>
  </footer>
</body>
```

<footer> node is where footnotes are moved to. If you target only devices that support footnote popups you may even hide <footer> with CSS.

```
$ cat footnotes
#!/usr/bin/env ruby

require 'nokogiri'
include Nokogiri::XML

doc = Nokogiri::XML STDIN.read
if !(footer = doc.at_css('footer'))
  puts doc.to_xml
  exit
end

doc.css('footnote').each_with_index do |footnote, idx|
  fra = DocumentFragment.parse ''

  aside = Node.new 'aside', fra
  aside['id'] = "ft-#{idx}"

  sup = Node.new 'sup', fra
  sup['id'] = "ft-ref-#{idx}"
  sup.inner_html = "<a href='##{aside['id']}'>#{idx+1}</a>"

  aside.inner_html = [
    "#{idx+1}.",
    footnote.content,
    "<a href='##{sup["id"]}'>↺</a>"
  ].join ' '

  footnote.replace sup
  footer << aside
end

puts doc.to_xml
```

4.9 Em dash

> For an em dash—one that indicates a break in a sentence like this—either use the em dash character on your word processor or type two hyphens (leave no space on either side).
> — *The Chicago Manual of Style*, 17th ed.

Intense debates about typographic punctuation are out of scope for this book. The problem with leaving `--` characters as is, though, is line wrapping, for it's possible for a layout engine to make the following blunder:

```
three impressionists-
-Monet, Sisley, and
Degas
```

To prevent it, we can replace `--` or `---` with '—'

```
$ sed -E '/----/!s/---?/—/g' < 1.xhtml
```

but if `--` appears inside a code listing, it gets converted too. Not only this enrages the reader, but also shows either grandiose luck of care or catastrophic incompetence.

In a test file

```
$ cat emdash.html
<body>
 Will he---can he---get it?
 <p>--Alright? N--</p>
 <div>
  <p>"We shall never--"</p>
  <p>--.</p>
  <p>---</p>
  <p>--- but ---------</p>
 </div>
 <p>----Not an em dash-----.</p>
 <script>a--;</script>
</body>
```

`--` and `---` should be converted to em dashes, but `----` (and longer) should not.

```
$ cat emdash.html | ./emdash
<?xml version="1.0"?>
<body>
 Will he&#x2014;can he&#x2014;get it?
 <p>&#x2014;Alright? N&#x2014;</p>
 <div>
  <p>"We shall never&#x2014;"</p>
  <p>&#x2014;.</p>
  <p>&#x2014;</p>
  <p>&#x2014; but ---------</p>
 </div>
 <p>----Not an em dash-----.</p>
 <script>a--;</script>
</body>
```

The postprocessor traverses through all nodes, ignoring `<script>`, `<pre>` & the like:

```
#!/usr/bin/env ruby

require 'nokogiri'

$ignored = [
  'script',
  'template',
```

```
  'code',
  'var',
  'pre',
  'kbd',
  'textarea',
  'tt',
  'xmp',
  'samp',
  'math',
  'style',
]

def process n, cb
  return if $ignored.index n.name
  n.content = cb.call(n) if n.text?
  n.children.each {|k| process k, cb}
end

def emdash n
  n.text.gsub(/(?<!-)-{2,3}(?!-)/, '—')
end

doc = Nokogiri::XML STDIN.read
process doc, method(:emdash)
puts doc.to_xml
```

The regular expression inside `emdash` function uses both negative lookbehind & lookahead. Naturally, you can add any other transformations to it, e.g., replace quotes with "curly" equivalents or 3 consecutive dots ("...") with an ellipsis, &c.

4.10 Makefile

The second you introduce any kind of transformation to your *.xhtml* or *.opf* sources, you can't zip the originals up into an EPUB container anymore—a build automation process is required that puts the transformed files into a dedicated directory. That directory is what goes into the final EPUB.

Say we have a book, that consists of `chapter1.xhtml` that is expected to be preprocessed by erb:

```
$ cat src/chapter1.xhtml
<?xml version="1.0" encoding="utf-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>.</title></head>
<body>
 <pre>
<%= File.read 'src/strlen.c' %>
 </pre>
 <img src="graph.svg" />
</body>
</html>
```

Using erb is quite handy—instead of error-prone copy-pasting, we can invoke `File.read` inside to insert the contents of a file.

Our source directory may look like this:

```
$ tree --dirsfirst .
.
├── src
│   ├── META-INF
│   │   └── container.xml
│   ├── chapter1.xhtml
│   ├── graph.dot
│   ├── package.opf
│   ├── strlen.c
│   └── toc.xhtml
└── Makefile
```

`chapter1.xhtml` expects `graph.svg`, but we don't have it, instead we obtain the latter by converting `graph.dot` (a Graphviz DOT script) into an *.svg* image.

For build automation below we use GNU Make. All files (transformed or not), that go into the EPUB container, appear in a single `_out/book` directory.

First, we need a list of targets we're going to produce.

Target	*Source*
_out/book/chapter1.xhtml	src/chapter1.xhtml
_out/book/graph.svg	src/graph.dot

```
out := _out/book
targets = $(patsubst src/%.$1, \
    $(out)/%.$2, \
    $(wildcard src/*.$1))

dest := $(call targets,xhtml,xhtml) \
    $(call targets,dot,svg)

all: $(dest)
```

where `targets` is a macro to help not to write tedious

```
$(patsubst src/%.dot, $(out)/%.svg, $(wildcard src/*.svg))
```

`dest` variable will contain

```
_out/book/chapter1.xhtml
_out/book/toc.xhtml
_out/book/graph.svg
```

Second, we write rules that explain to Make how to create our targets in `dest` variable:

```
mkdir = @mkdir -p $(dir $@)
.DELETE_ON_ERROR:

$(out)/%.xhtml: src/%.xhtml
    $(mkdir)
    erb $< > $@

$(out)/%.svg: src/%.dot
    $(mkdir)
    dot -Tsvg $< -o $@
```

If you put all this into a file name Makefile, run

```
$ make
erb src/chapter1.xhtml > _out/book/chapter1.xhtml
```

```
erb src/toc.xhtml > _out/book/toc.xhtml
dot -Tsvg src/graph.dot -o _out/book/graph.svg
```

There is a slight problem here. If we change `chapter1.xhtml`, it gets rebuild, but if we modify `strlen.c`, that is included in the *.xhtml*, Make doesn't know it has a stale target:

```
$ touch src/strlen.c
$ make
make: Nothing to be done for 'all'.
```

One way of dealing with it is to write a list of dependencies for a particular *.xhtml*, after it gets successfully compiled.

```
$ ./mkdep _out/book/chapter1.xhtml src/chapter1.xhtml
_out/book/chapter1.xhtml: src/strlen.c
```

`mkdep` is a small Ruby script (no, it doesn't use Nokogiri) that tries to find dependencies for its 2nd command line argument. If it finds at least 1, it prints a Make-compatible rule.

```
$ cat mkdep
#!/usr/bin/env ruby

abort "Usage: #{$0} target dep" if ARGV.size != 2
target, dep = ARGV

# finds calls to `File.read`
# inside an erb template
re = /<%=\s+(File.read)\s*\(?\s*["'](.[^,]+?)["']/

fd = File.open dep
while (l = fd.gets)
  puts "#{target}: #{$2}" if l =~ re
end
```

We can add the invocation of `mkdep` to our rule for *.xhtml* targets:

```
$(out)/%.xhtml: src/%.xhtml
    $(mkdir)
    erb $< > $@
    $(make-depend)
```

where `make-depend` is a macro:

```
cache := _out/.cache
define make-depend
@mkdir -p $(cache)/$(dir $<)
./mkdep $@ $< > $(cache)/$(dir $<)/$*.d
endef

# load all .d files
# produced by $(make-depend)
-include $(patsubst %.xhtml, \
    $(cache)/%.d, \
    $(wildcard src/*.xhtml))
```

Now Make should always rebuild `chapter1.xhtml` correctly:

```
$ rm -rf _out
$ make
…
$ make
make: Nothing to be done for 'all'.
```

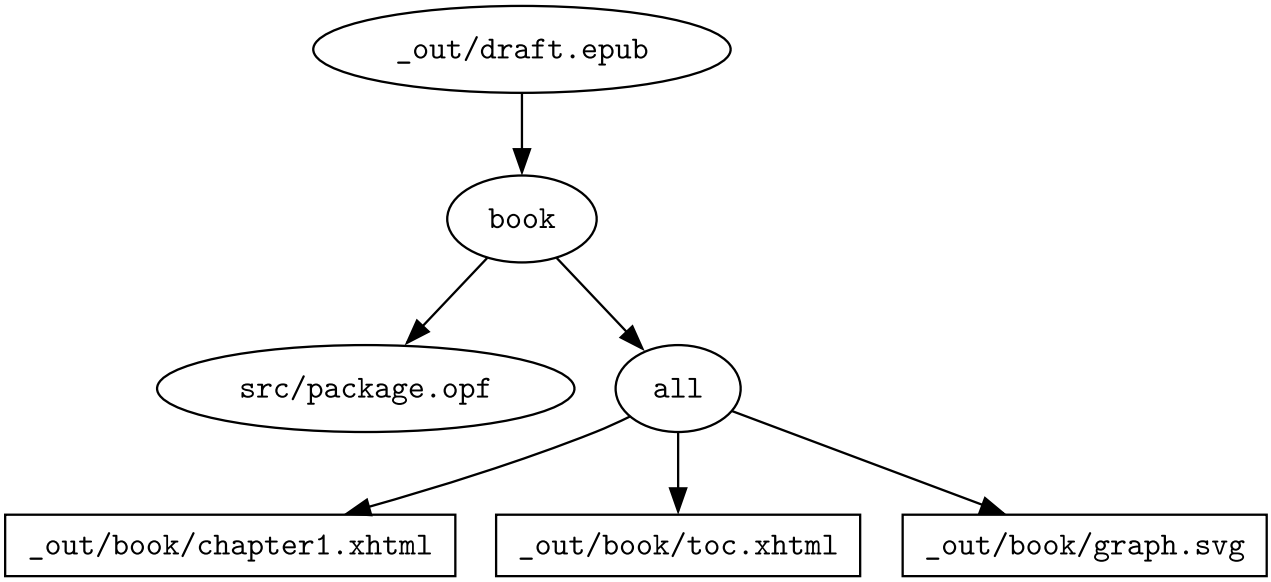
```
$ cat _out/.cache/*/*.d
_out/book/chapter1.xhtml: src/strlen.c

$ touch src/strlen.c
$ make
erb src/chapter1.xhtml > _out/book/chapter1.xhtml
./mkdep _out/book/chapter1.xhtml src/chapter1.xhtml > _out/.cache/src//chapter1.d
```

Third, & final step is to make an EPUB. We create 'mimetype' file, then copy the rest of the required stuff for the EPUB container into `_out/book`, & run zip(1):

```
book: _out/draft.epub
dest += src/package.opf

_out/draft.epub: $(dest)
    rm $@ -f
    printf application/epub+zip > mimetype
    zip -qX0 $@ mimetype
    rm mimetype
    cp -r src/package.opf src/META-INF $(out)
    (cd $(out); zip -qX $(CURDIR)/$@ `find * -type f`)
    -epubcheck $@
```

Then you can run `make book` to get `_out/draft.epub` file.

Below is the full DAG (directed acyclic graph) that our Makefile internally generates (an arrow means *depends on*):



Exercises

1. Rewrite `package.opf.erb` into a regular Ruby script putting the ERB template after a rarely used `__END__` directive to extract the template using global `DATA` file object. Which version is easier to maintain?
2. Add support for inlined formulas to `inline-formulas` script, i.e., $a + a = 2a$ in this sentence.

Glossary

Alphabetically sorted.

Container
: A ZIP file.

Content documents
: Not medatada.

EPUB
: A *container* that holds *.xhtml* files, images &c + a couple of special medatada files.

EPUB creator
: You. (Unsuccessful terms used in the past: Author, Content Provider.)

Manifest
: A list of all included files in a *container*. Lives in a *package file*.

Navigation document
: A special *.xhtml* file that contains a machine-readable TOC (Table of Contents).

.opf file
: See *Package file*.

Package file
: Main metadata file for a *container*; holds entries like a title, an author, a list of all *content documents*.

Reading system
: User-agent.

Rendition
: A single book/magazine/&c. Theoretically, an *EPUB* can contain multiple *renditions*.

Spine
: An ordered list of one or more documents (usually *.xhtml*) for the default reading order. Lives in a *package file*.

Unique identifier
: An id for an EPUB, usually in the form of an UUID or an ISBN. Lives in a *package file*. Each book's new major edition requires a new id.

Appendix 1: Software Requirements

- Unix-like enviroment.
- Ruby v3+.
- zip(1).
- xmllint(1) command line utility (usually packaged with libxml2 library).
- Inkscape.
- pdflatex(1) that is a part of TeX Live distribution.
- GNU Make.

Nokogiri

It is often present in OS package repositories, but the latest version is always available from RubyGems.

```
$ gem install nokogiri
```

installs the library and a handy command line tool `nokogiri`.

EPUB linter

At the time of writing, the only linter worth mentioning is EPUBCheck. It's a slow Java program that is distributed as an executable *.jar* file. Unpack `epubcheck-a.b.c.zip` somewhere (e.g., ~/opt/s/epubcheck) & test:

```
$ alias epubcheck='java -jar ~/opt/s/epubcheck/epubcheck.jar'
$ epubcheck --version
```

You can check with it a whole EPUB:

```
$ epubcheck file.epub
```

or an individual parts like a package file:

```
$ epubcheck --mode opf package.opf
```

a TOC:

```
$ epubcheck --mode nav toc.xhtml
```

or any *.xhtml*:

```
$ epubcheck --mode xhtml ch1.xhtml
```

To use the linter inside makefiles, put the following executable script to a directory in `PATH`:

```
$ cat ~/bin/epubcheck
#!/bin/sh
java -jar ~/opt/s/epubcheck/epubcheck.jar "$@"
```

Appendix 2: Kindle

In the beginning, there was MOBI, which had a competitor called OEB (Open eBook). The former was a proprietary format, and its specification remained closed. The latter was an open format based on XHTML with optional style sheets. OEB eventually died but gave birth to what became EPUB.

Amazon placed its bet on MOBI, while every other manufacturer in the world chose EPUB. Presentation-wise, MOBI was a very limited format that was hardly adequate for anything except poetry collections or romance fiction. Instead of switching to EPUB, the jungle company decided to sit between 2 chairs by inventing a new container format that held 2 different actual e-book formats simultaneously: one was the same old MOBI, while the other was their new proprietary (of course) format "based" on EPUB v3. This new container had *.mobi* file extension to never confuse anyone.

To create a book compatible with user-agents inside Kindle devices and mobile applications, Amazon provided a command-line program called *kindlegen*. The idea was to make a usual EPUB, then convert it with kindlegen to a mysterious *.mobi* file that wasn't an actual MOBI. One day, kindlegen vanished from the jungle website and was replaced with a GUI previewer that amusingly shipped with (no longer publicly available) kindlegen, poorly concealed among the previewer files.

After a while, folks hanging out on mobileread.com reverse-engineered this "new" container format named KF8 (Kindle File Format v8), which kindlegen produced.

File extensions	*Format*	*EPUB support*
.azw, .mobi	MOBI	none
.azw3, .mobi	KF8	v3
.azw8, .kfx	KF10	v3 (with "enhancements" like auto-hyphenation)

To see what's inside of a containerised fake *.mobi*, take `min.epub` example from the beginning of Chapter 2, convert it to *.mobi* & unpack with a free utility KindleUnpack.

```
$ kindlegen min.epub
$ pip install --user mobi
$ mobiunpack min.mobi
```

```
$ tree -F --dirsfirst min
min/
├── HDImages/
├── mobi7/
│   ├── Images/
│   ├── book.html
│   ├── content.opf
│   └── toc.ncx
├── mobi8/
│   ├── META-INF/
│   │   └── container.xml
│   ├── OEBPS/
│   │   ├── Fonts/
│   │   ├── Images/
│   │   ├── Styles/
│   │   ├── Text/
│   │   │   └── part0000.xhtml
│   │   ├── content.opf
│   │   └── toc.ncx
│   ├── mimetype
│   └── min.epub
├── kindlegenbuild.log
└── kindlegensrc.zip
```

`mobi8` directory is where our reformatted EPUB is. When you have troubles understanding why Kindle doesn't render a node properly, use KindleUnpack to see what exactly the user-agent inside Kindle is parsing.

There is no official way to download *kindlegen* anymore, except to fish it out using archive.org.

Appendix 3: opf-grep

If you want to examine the content that publishers put into *the package file*, it is helpful to look at a variety of existing books. Unfortunately, extracting the package file requires parsing of META-INF/container.xml inside a ZIP container. Relying on the *.opf* extension when looking through files in the container is a brittle plan, for the extension name is not mandatory, but only recommended.

`opf-grep` is a Ruby program that allows to search for patterns in `<metadata>` section of package files inside EPUBs.

E.g., this matches `<dc:title>` & prints the element name, its content, & `<meta>` nodes because they refine `<dc:title>`:

```
$ backmatter/opf-grep title _out/draft.epub
_out/draft.epub:package.opf 3.0
<metadata>
 <dc:title id="title1">A practical guide to EPUB</dc:title>
 <meta refines="#title1" property="title-type">main</meta>
 <dc:title id="title2">DIY tools and recipes
                        for the common folk</dc:title>
 <meta refines="#title2" property="title-type">subtitle</meta>
</metadata>
```

To match content only for a specific tag, add `-t` option. This prints all `<dc:subject>` tags, if any present:

```
$ ./opf-grep -t subject . file.epub
file.epub:package.opf 3.0
<metadata>
 <dc:subject>epub</dc:subject>
 <dc:subject>e-book</dc:subject>
 <dc:subject>publishing</dc:subject>
 <dc:subject>ruby</dc:subject>
 <dc:subject>command line</dc:subject>
 <dc:subject>linux</dc:subject>
 <dc:subject>libzip</dc:subject>
</metadata>
```

To dump whole package file use '.' as a pattern:

```
$ ./opf-grep . file.epub
```

A recursive search for `<dc:rights>`:

```
$ find /dir -name \*.epub -print0 | xargs -0 ./opf-grep -M -t rights .
```

The program uses unzip(1) internally.

```
$ cat opf-grep
#!/usr/bin/env ruby

require 'nokogiri'
require 'optionparser'
require 'open3'

opt = ARGV.getopts("Mt:")
abort "Usage: #{$0} [-M] [-t tag] pattern file.epub …" if ARGV.size < 2

def zip_read archive, file
  stdout, stderr, status = Open3.capture3 'unzip', "-p", archive, file
  fail stderr if !status.success?
  stdout
end
```

```
class Epub
  def initialize file, opt
    fail "not an epub" unless "application/epub+zip" == IO.read(file, 58).split(/^P
K/)[1]&.[](36..-1) || opt['M']
    @file = file
  end
  attr_reader :file

  def packages
    doc = Nokogiri::XML zip_read @file, 'META-INF/container.xml'
    doc.css('rootfile').map do |v|
      d = Nokogiri::XML zip_read @file, v['full-path']
      {
        path: v['full-path'],
        ver: d.at_css('package')&.[]("version"),
        node: d.at_css('package>metadata'),
      }
    end
  end
end

def node_new name, nss
  n = Nokogiri::XML::Node.new name, Nokogiri::XML::DocumentFragment.parse('')
  nss.each {|v| n.add_namespace v.prefix, v.href }
  n.add_namespace 'dc', 'http://purl.org/dc/elements/1.1/'
  n
end

def nodeset_add ns, node
  if ns.none? {|v| v == node}
    ns << node
    node.content = node.content.strip
  end
end

def match tag, pattern, opf
  kids = Nokogiri::XML::NodeSet.new Nokogiri::XML::Document.new
  opf.children.each do |node|
    next if node.text?
    next unless node.name.match?(tag)
    next unless node.content.match?(pattern) ||
                node.attributes.any? {|_, a| a.value.match?(pattern) } ||
                (tag == '.' && node.name.match?(pattern))

    # check if node refines something
    if node.name == 'meta' && node['refines']
      n0 = opf.at_css node['refines']
      nodeset_add kids, n0 if n0
    end

    nodeset_add kids, node

    # check if something refines node
    if node.name != 'meta'
      n2 = opf.at_css "meta[refines='##{node['id']}']"
      nodeset_add kids, n2 if n2
    end
  end

  return "" if kids.size == 0
  r = node_new 'metadata', opf.namespace_scopes
  r.children = kids
  r.to_xml(indent: 1).sub(/^<metadata[^>]+?>/m, '<metadata>')
end

def grep opt, pattern, file
```

```
  begin
    epub = Epub.new file, opt
  rescue
    warn "#{file}: #{$!}"
    return false
  end

  found = false
  epub.packages.each do |opf|
    pattern = Regexp.new pattern, 'i'
    r = match(opt['t'] || '.', pattern, opf[:node])
    if r.size > 0
      puts "\e[1m#{epub.file}:#{opf[:path]} #{opf[:ver]}\e[0m"
      puts r
      found = true
    end
  end
  found
end

status = 0
ARGV[1..-1].each do |file|
  status = 1 unless grep(opt, ARGV[0], file) && status == 0
end
exit status
```

`-M` option is sometimes necessary, for it is amazing how many EPUB creators do not bother with making a valid EPUB container.

The program is optimistic & doesn't check `Epub#packages` exceptions in `grep` function. To qualify as a Proper utility program, it should verify if stdout is a terminal before writing ANSI escape sequences. Adding support for reading stdin is also advisable.