GAPDoc

(Version 1.6.7)

February 2024

Frank Lübeck
Max Neunhöffer

Frank Lübeck Email: Frank.Luebeck@Math.RWTH-Aachen.De
Homepage: https://www.math.rwth-aachen.de/~Frank.Luebeck

Copyright

Contents

Chapter 1

Introduction and Example

The main purpose of the GAPDoc package is to define a file format for documentation of GAP-programs and -packages (see [GAP06]). The problem is that such documentation should be readable in several output formats. For example it should be possible to read the documentation inside the terminal in which GAP is running (a text mode) and there should be a printable version in high typesetting quality (produced by some version of TeX). It is also popular to view GAP's online help with a Web-browser via an HTML-version of the documentation. Nowadays one can use LaTeX and standard viewer programs to produce and view on the screen `dvi-` or `pdf`-files with full support of internal and external hyperlinks. Certainly there will be other interesting document formats and tools in this direction in the future.

Our aim is to find a *format for writing* the documentation which allows a relatively easy translation into the output formats just mentioned and which hopefully makes it easy to translate to future output formats as well.

To make documentation written in the GAPDoc format directly usable, we also provide a set of programs, called converters, which produce text-, hyperlinked LaTeX- and HTML-output versions of a GAPDoc document. These programs are developed by the first named author. They run completely inside GAP, i.e., no external programs are needed. You only need `latex` and `pdflatex` to process the LaTeX output. These programs are described in Chapter 5.

1.1 XML

The definition of the GAPDoc format uses XML, the "eXtendible Markup Language". This is a standard (defined by the W3C consortium, see `https://www.w3c.org`) which lays down a syntax for adding markup to a document or to some data. It allows to define document structures via introducing markup *elements* and certain relations between them. This is done in a *document type definition*. The file `gapdoc.dtd` contains such a document type definition and is the central part of the GAPDoc package.

The easiest way for getting a good idea about this is probably to look at an example. The Appendix A contains a short but complete GAPDoc document for a fictitious share package. In the next section we will go through this document, explain basic facts about XML and the GAPDoc document type, and give pointers to more details in later parts of this documentation.

In the last Section 1.3 of this introductory chapter we try to answer some general questions about the decisions which lead to the GAPDoc package.

1.2 A complete example

In this section we recall the lines from the example document in Appendix A and give some explanations.

from 3k+1.xml
```
<?xml version="1.0" encoding="UTF-8"?>
```

This line just tells a human reader and computer programs that the file is a document with XML markup and that the text is encoded in the UTF-8 character set (other common encodings are ASCII or ISO-8895-X encodings).

from 3k+1.xml
```
<!--   A complete "fake package" documentation
-->
```

Everything in a XML file between "`<!--`" and "`-->`" is a comment and not part of the document content.

from 3k+1.xml
```
<!DOCTYPE Book SYSTEM "gapdoc.dtd">
```

This line says that the document contains markup which is defined in the system file `gapdoc.dtd` and that the markup obeys certain rules defined in that file (the ending `dtd` means "document type definition"). It further says that the actual content of the document consists of an element with name "Book". And we can really see that the remaining part of the file is enclosed as follows:

from 3k+1.xml
```
<Book Name="3k+1">
  [...] (content omitted)
</Book>
```

This demonstrates the basics of the markup in XML. This part of the document is an "element". It consists of the "start tag" `<Book Name="3k+1">`, the "element content" and the "end tag" `</Book>` (end tags always start with `</`). This element also has an "attribute" `Name` whose "value" is `3k+1`.

If you know HTML, this will look familiar to you. But there are some important differences: The element name `Book` and attribute name `Name` are *case sensitive*. The value of an attribute must *always* be enclosed in quotes. In XML *every* element has a start and end tag (which can be combined for elements defined as "empty", see for example `<TableOfContents/>` below).

If you know LaTeX, you are familiar with quite different types of markup, for example: The equivalent of the `Book` element in LaTeX is `\begin{document} ...  \end{document}`. The sectioning in LaTeX is not done by explicit start and end markup, but implicitly via heading commands like `\section`. Other markup is done by using braces `{}` and putting some commands inside. And for mathematical formulae one can use the `$` for the start *and* the end of the markup. In XML *all* markup looks similar to that of the `Book` element.

The content of the book starts with a title page.

from 3k+1.xml
```
<TitlePage>
  <Title>The <Package>ThreeKPlusOne</Package> Package</Title>
  <Version>Version 42</Version>
  <Author>Dummy Authör
    <Email>3kplusone@dev.null</Email>
  </Author>
```

```
  <Copyright>&copyright; 2000 The Author. <P/>
    You can do with this package what you want.<P/> Really.
  </Copyright>
</TitlePage>
```

The content of the TitlePage element consists again of elements. In Chapter 3 we describe which elements are allowed within a TitlePage and that their ordering is prescribed in this case. In the (stupid) name of the author you see that a German umlaut is used directly (in ISO-latin1 encoding).

Contrary to LaTeX- or HTML-files this markup does not say anything about the actual layout of the title page in any output version of the document. It just adds information about the *meaning* of pieces of text.

Within the Copyright element there are two more things to learn about XML markup. The <P/> is a complete element. It is a combined start and end tag. This shortcut is allowed for elements which are defined to be always "empty", i.e., to have no content. You may have already guessed that <P/> is used as a paragraph separator. Note that empty lines do not separate paragraphs (contrary to LaTeX).

The other construct we see here is &copyright;. This is an example of an "entity" in XML and is a macro for some substitution text. Here we use an entity as a shortcut for a complicated expression which makes it possible that the term *copyright* is printed as some text like (C) in text terminal output and as a copyright character in other output formats. In GAPDoc we predefine some entities. Certain "special characters" must be typed via entities, for example "<", ">" and "&" to avoid a misinterpretation as XML markup. It is possible to define additional entities for your document inside the <!DOCTYPE ...> declaration, see 2.2.3.

Note that elements in XML must always be properly nested, as in this example. A construct like <a>... is *not* allowed.

from 3k+1.xml

```
<TableOfContents/>
```

This is another example of an "empty element". It just means that a table of contents for the whole document should be included into any output version of the document.

After this the main text of the document follows inside certain sectioning elements:

from 3k+1.xml

```
<Body>
  <Chapter> <Heading>The <M>3k+1</M> Problem</Heading>
    <Section Label="sec:theory"> <Heading>Theory</Heading>
      [...] (content omitted)
    </Section>
    <Section> <Heading>Program</Heading>
      [...] (content omitted)
    </Section>
  </Chapter>
</Body>
```

These elements are used similarly to "\chapter" and "\section" in LaTeX. But note that the explicit end tags are necessary here.

The sectioning commands allow to assign an optional attribute "Label". This can be used for referring to a section inside the document.

The text of the first section starts as follows. The whitespace in the text is unimportant and the indenting is not necessary.

```
                              from 3k+1.xml
      Let  <M>k \in  &NN;</M> be  a  natural number.  We consider  the
      sequence <M>n(i, k), i \in &NN;,</M> with <M>n(1, k) = k</M> and
      else
```

Here we come to the interesting question how to type mathematical formulae in a GAPDoc document. We did not find any alternative for writing formulae in TEX syntax. (There is MATHML, but even simple formulae contain a lot of markup, become quite unreadable and they are cumbersome to type. Furthermore there seem to be no tools available which translate such formulae in a nice way into TEX and text.) So, formulae are essentially typed as in LATEX. (Actually, it is also possible to type unicode characters of some mathematical symbols directly, or via an entity like the `&NN;` above.) There are three types of elements containing formulae: “M”, “Math” and “Display”. The first two are for in-text formulae and the third is for displayed formulae. Here “M” and “Math” are equivalent, when translating a GAPDoc document into LATEX. But they are handled differently for terminal text (and HTML) output. For the content of an “M”-element there are defined rules for a translation into well readable terminal text. More complicated formulae are in “Math” or “Display” elements and they are just printed as they are typed in text output. So, to make a section well readable inside a terminal window you should try to put as many formulae as possible into “M”-elements. In our example text we used the notation `n(i, k)` instead of `n_i(k)` because it is easier to read in text mode. See Sections 2.2.2 and 3.9 for more details.

A few lines further on we find two non-internal references.

```
                              from 3k+1.xml
      problem, see <Cite Key="Wi98"/> or
      <URL>http://mathsrv.ku-eichstaett.de/MGF/homes/wirsching/</URL>
```

The first within the “Cite”-element is the citation of a book. In GAPDoc we use the widely used BibTEX database format for reference lists. This does not use XML but has a well documented structure which is easy to parse. And many people have collections of references readily available in this format. The reference list in an output version of the document is produced with the empty element

```
                              from 3k+1.xml
  <Bibliography Databases="3k+1" />
```

close to the end of our example file. The attribute “Databases” give the name(s) of the database (`.bib`) files which contain the references.

Putting a Web-address into an “URL”-element allows one to create a hyperlink in output formats which allow this.

The second section of our example contains a special kind of subsection defined in GAPDoc.

```
                              from 3k+1.xml
      <ManSection>
        <Func Name="ThreeKPlusOneSequence" Arg="k[, max]"/>
        <Description>
          This  function computes  for a  natural number  <A>k</A> the
          beginning of the sequence  <M>n(i, k)</M> defined in section
          <Ref Sect="sec:theory"/>.  The sequence  stops at  the first
          <M>1</M>  or at  <M>n(<A>max</A>, k)</M>,  if <A>max</A>  is
          given.
  <Example>
  gap> ThreeKPlusOneSequence(101);
```

```
"Sorry, not yet implemented. Wait for Version 84 of the package"
</Example>
        </Description>
      </ManSection>
```

A “ManSection” contains the description of some function, operation, method, filter and so on. The “Func”-element describes the name of a *function* (there are also similar elements “Oper”, “Meth”, “Filt” and so on) and names for its arguments, optional arguments enclosed in square brackets. See Section 3.4 for more details.

In the “Description” we write the argument names as “A”-elements. A good description of a function should usually contain an example of its use. For this there are some verbatim-like elements in GAPDoc, like “Example” above (here, clearly, whitespace matters which causes a slightly strange indenting).

The text contains an internal reference to the first section via the explicitly defined label `sec:theory`.

The first section also contains a “Ref”-element which refers to the function described here. Note that there is no explicit label for such a reference. The pair `<Func Name="ThreeKPlusOneSequence" Arg="k[, max]"/>` and `<Ref Func="ThreeKPlusOneSequence"/>` does the cross referencing (and hyperlinking if possible) implicitly via the name of the function.

Here is one further element from our example document which we want to explain.

from 3k+1.xml

```
<TheIndex/>
```

This is again an empty element which just says that an output version of the document should contain an index. Many entries for the index are generated automatically because the “Func” and similar elements implicitly produce such entries. It is also possible to include explicit additional entries in the index.

1.3 Some questions

Are those XML files too ugly to read and edit?

Just have a look and decide yourself. The markup needs more characters than most TeX or LaTeX markup. But the structure of the document is easier to see. If you configure your favorite editor well, you do not need more key strokes for typing the markup than in LaTeX.

Why do we not use LaTeX alone?

LaTeX is good for writing books. But LaTeX files are generally difficult to parse and to process to other output formats like text for browsing in a terminal window or HTML (or new formats which may become popular in the future). GAPDoc markup is one step more abstract than LaTeX insofar as it describes meaning instead of appearance of text. The inner workings of LaTeX are too complicated to learn without pain, which makes it difficult to overcome problems that occur occasionally.

Why XML and not a newly defined markup language?

XML is a well defined standard that is more and more widely used. Lots of people have thought about it. Years of experience with SGML went into the design. It is easy to explain, easy to parse and lots of tools are available, there will be more in the future.

Chapter 2

How To Type a GAPDoc Document

In this chapter we give a more formal description of what you need to start to type documentation in GAPDoc XML format. Many details were already explained by example in Section 1.2 of the introduction.

We do *not* answer the question “How to *write* a GAPDoc document?” in this chapter. You can (hopefully) find an answer to this question by studying the example in the introduction, see 1.2, and learning about more details in the reference Chapter 3.

The definite source for all details of the official XML standard with useful annotations is:

`https://www.xml.com/axml/axml.html`

Although this document must be quite technical, it is surprisingly well readable.

2.1 General XML Syntax

We will now discuss the pieces of text which can occur in a general XML document. We start with those pieces which do not contribute to the actual content of the document.

2.1.1 Head of XML Document

Each XML document should have a head which states that it is an XML document in some encoding and which XML-defined language is used. In case of a GAPDoc document this should always look as in the following example.

Example
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Book SYSTEM "gapdoc.dtd">
```

See 2.1.13 for a remark on the “encoding” statement.

(There may be local entity definitions inside the `DOCTYPE` statement, see Subsection 2.2.3 below.)

2.1.2 Comments

A “comment” in XML starts with the character sequence “`<!--`” and ends with the sequence “`-->`”. Between these sequences there must not be two adjacent dashes “`--`”.

2.1.3 Processing Instructions

A "processing instruction" in XML starts with the character sequence "`<?`" followed by a name ("`xml`" is only allowed at the very beginning of the document to declare it being an XML document, see 2.1.1). After that any characters may follow, except that the ending sequence "`?>`" must not occur within the processing instruction.

And now we turn to those parts of the document which contribute to its actual content.

2.1.4 Names in XML and Whitespace

A "name" in XML (used for element and attribute identifiers, see below) must start with a letter (in the encoding of the document) or with a colon "`:`" or underscore "`_`" character. The following characters may also be digits, dots "`.`" or dashes "`-`".

This is a simplified description of the rules in the standard, which are concerned with lots of unicode ranges to specify what a "letter" is.

Sequences only consisting of the following characters are considered as *whitespace*: blanks, tabs, carriage return characters and new line characters.

2.1.5 Elements

The actual content of an XML document consists of "elements". An element has some "content" with a leading "start tag" (2.1.6) and a trailing "end tag" (2.1.7). The content can contain further elements but they must be properly nested. One can define elements whose content is always empty, those elements can also be entered with a single combined tag (2.1.8).

2.1.6 Start Tags

A "start-tag" consists of a less-than-character "`<`" directly followed (without whitespace) by an element name (see 2.1.4), optional attributes, optional whitespace, and a greater-than-character "`>`".

An "attribute" consists of some whitespace and then its name followed by an equal sign "`=`" which is optionally enclosed by whitespace, and the attribute value, which is enclosed either in single or double quotes. The attribute value may not contain the type of quote used as a delimiter or the character "`<`", the character "`&`" may only appear to start an entity, see 2.1.9. We describe in 2.1.11 how to enter special characters in attribute values.

Note especially that no whitespace is allowed between the starting "`<`" character and the element name. The quotes around an attribute value cannot be omitted. The names of elements and attributes are *case sensitive*.

2.1.7 End Tags

An "end tag" consists of the two characters "`</`" directly followed by the element name, optional whitespace and a greater-than-character "`>`".

2.1.8 Combined Tags for Empty Elements

Elements which always have empty content can be written with a single tag. This looks like a start tag (see 2.1.6) *except* that the trailing greater-than-character "`>`" is substituted by the two character sequence “/>”.

2.1.9 Entities

An “entity” in XML is a macro for some substitution text. There are two types of entities.

A “character entity” can be used to specify characters in the encoding of the document (can be useful for entering non-ASCII characters which you cannot manage to type in directly). They are entered with a sequence “&#”, directly followed by either some decimal digits or an “x” and some hexadecimal digits, directly followed by a semicolon “;”. Using such a character entity is just equivalent to typing the corresponding character directly.

Then there are references to “named entities”. They are entered with an ampersand character “&” directly followed by a name which is directly followed by a semicolon “;”. Such entities must be declared somewhere by giving a substitution text. This text is included in the document and the document is parsed again afterwards. The exact rules are a bit subtle but you probably want to use this only in simple cases. Predefined entities for GAPDoc are described in 2.1.10 and 2.2.3.

2.1.10 Special Characters in XML

We have seen that the less-than-character “<” and the ampersand character “&” start a tag or entity reference in XML. To get these characters into the document text one has to use entity references, namely “<” to get “<” and “&” to get “&”. Furthermore “>” must be used to get “>” when the string “]]>” appears in element content (and not as delimiter of a CDATA section explained below).

Another possibility is to use a CDATA statement explained in 2.1.12.

2.1.11 Rules for Attribute Values

Attribute values can contain entities which are substituted recursively. But except for the entities < or a character entity it is not allowed that a < character is introduced by the substitution (there is no XML parsing for evaluating the attribute value, just entity substitutions).

2.1.12 CDATA

Pieces of text which contain many characters which can be misinterpreted as markup can be enclosed by the character sequences “<![CDATA[” and “]]>”. Everything between these sequences is considered as content of the document and is not further interpreted as XML text. All the rules explained so far in this section do *not apply* to such a part of the document. The only document content which cannot be entered directly inside a CDATA statement is the sequence “]]>”. This can be entered as “]]>” outside the CDATA statement.

Example
```
A nesting of tags like <a> <b> </a> </b> is not allowed.
```

2.1.13 Encoding of an XML Document

We suggest to use the UTF-8 encoding for writing GAPDoc XML documents. But the tools described in Chapter 5 also work with ASCII or the various ISO-8859-X encodings (ISO-8859-1 is also called latin1 and covers most special characters for western European languages).

2.1.14 Well Formed and Valid XML Documents

We want to mention two further important words which are often used in the context of XML documents. A piece of text becomes a “well formed” XML document if all the formal rules described in this section are fulfilled.

But this says nothing about the content of the document. To give this content a meaning one needs a declaration of the element and corresponding attribute names as well as of named entities which are allowed. Furthermore there may be restrictions how such elements can be nested. This *definition of an XML based markup language* is done in a “document type definition”. An XML document which contains only elements and entities declared in such a document type definition and obeys the rules given there is called “valid (with respect to this document type definition)”.

The main file of the GAPDoc package is gapdoc.dtd. This contains such a definition of a markup language. We are not going to explain the formal syntax rules for document type definitions in this section. But in Chapter 3 we will explain enough about it to understand the file gapdoc.dtd and so the markup language defined there.

2.2 Entering GAPDoc Documents

Here are some additional rules for writing GAPDoc XML documents.

2.2.1 Other special characters

As GAPDoc documents are used to produce LaTeX and HTML documents, the question arises how to deal with characters with a special meaning for other applications (for example “&”, “#”, “$”, “%”, “~”, “\”, “{”, “}”, “_”, “^”, “ ” (this is a non-breakable space, “~” in LaTeX) have a special meaning for LaTeX and “&”, “<”, “>” have a special meaning for HTML (and XML). In GAPDoc you can usually just type these characters directly, it is the task of the converter programs which translate to some output format to take care of such special characters. The exceptions to this simple rule are:

- & and < must be entered as & and < as explained in 2.1.10.
- The content of the GAPDoc elements <M>, <Math> and <Display> is LaTeX code, see 3.8.
- The content of an <Alt> element with Only attribute contains code for the specified output type, see 3.9.1.

Remark: In former versions of GAPDoc one had to use particular entities for all the special characters mentioned above (&tamp;, &hash;, $, &percent;, ˜, &bslash;, &obrace;, &cbrace;, &uscore;, &circum;, &tlt;, &tgt;). These are no longer needed, but they are still defined for backwards compatibility with older GAPDoc documents.

2.2.2 Mathematical Formulae

Mathematical formulae in GAPDoc are typed as in LaTeX. They must be the content of one of three types of GAPDoc elements concerned with mathematical formulae: “Math”, “Display”, and “M” (see Sections 3.8.1 and 3.8.2 for more details). The first two correspond to LaTeX's math mode and display math mode. The last one is a special form of the “Math” element type, that imposes certain restrictions on the content. On the other hand the content of an “M” element is processed in a well defined way for text terminal or HTML output. The “`Display`” element also has an attribute such that its content is processed as in “`M`” elements.

Note that the content of these element is LaTeX code, but the special characters “<” and “&” for XML must be entered via the entities described in 2.1.10 or by using a `CDATA` statement, see 2.1.12.

2.2.3 More Entities

In GAPDoc there are some more predefined entities:

`&GAP;`	GAP
`&GAPDoc;`	GAPDoc
`&TeX;`	TeX
`&LaTeX;`	LaTeX
`&BibTeX;`	BibTeX
`&MeatAxe;`	MeatAxe
`&XGAP;`	XGAP
`&copyright;`	©
` `	“ ”
`–`	–

Table: Predefined Entities in the GAPDoc system

Here ` ` is a non-breakable space character.

Additional entities are defined for some mathematical symbols, see 3.8 for more details.

One can define further local entities right inside the head (see 2.1.1) of a GAPDoc XML document as in the following example.

Example
```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE Book SYSTEM "gapdoc.dtd"
  [ <!ENTITY MyEntity "some longish <E>text</E> possibly with markup">
  ]>
```

These additional definitions go into the `<!DOCTYPE` tag in square brackets. Such new entities are used like this: `&MyEntity;`

Chapter 3

The Document Type Definition

In this chapter we first explain what a "document type definition" is and then describe `gapdoc.dtd` in detail. That file together with the current chapter define how a GAPDoc document has to look like. It can be found in the main directory of the GAPDoc package and it is reproduced in Appendix B.

We do not give many examples in this chapter which is more intended as a formal reference for all GAPDoc elements. Instead, we provide a separate help book, see **GAPDoc Example: Various types of text**. This uses all the constructs introduced in this chapter and you can easily compare the source code and how it looks like in the different output formats. Furthermore recall that many basic things about XML markup were already explained by example in the introductory chapter 1.

3.1 What is a DTD?

A document type definition (DTD) is a formal declaration of how an XML document has to be structured. It is itself structured such that programs that handle documents can read it and treat the documents accordingly. There are for example parsers and validity checkers that use the DTD to validate an XML document, see 2.1.14.

The main thing a DTD does is to specify which elements may occur in documents of a certain document type, how they can be nested, and what attributes they can or must have. So, for each element there is a rule.

Note that a DTD can *not* ensure that a document which is "valid" also makes sense to the converters! It only says something about the formal structure of the document.

For the remaining part of this chapter we have divided the elements of GAPDoc documents into several subsets, each of which will be discussed in one of the next sections.

See the following three subsections to learn by example, how a DTD works. We do not want to be too formal here, but just enable the reader to understand the declarations in `gapdoc.dtd`. For precise descriptions of the syntax of DTD's see again the official standard in:

`https://www.xml.com/axml/axml.html`

3.2 Overall Document Structure

A GAPDoc document contains on its top level exactly one element with name `Book`. This element is declared in the DTD as follows:

3.2.1 <Book>

From gapdoc.dtd
```
<!ELEMENT Book (TitlePage,
                TableOfContents?,
                Body,
                Appendix*,
                Bibliography?,
                TheIndex?)>
<!ATTLIST Book Name CDATA #REQUIRED>
```

After the keyword `ELEMENT` and the name `Book` there is a list in parentheses. This is a comma separated list of names of elements which can occur (in the given order) in the content of a `Book` element. Each name in such a list can be followed by one of the characters "?", "*" or "+", meaning that the corresponding element can occur zero or one time, an arbitrary number of times, or at least once, respectively. Without such an extra character the corresponding element must occur exactly once. Instead of one name in this list there can also be a list of elements names separated by "|" characters, this denotes any element with one of the names (i.e., "|" means "or").

So, the `Book` element must contain first a `TitlePage` element, then an optional `TableOfContents` element, then a `Body` element, then zero or more elements of type `Appendix`, then an optional `Bibliography` element, and finally an optional element of type `TheIndex`.

Note that *only* these elements are allowed in the content of the `Book` element. No other elements or text is allowed in between. An exception of this is that there may be whitespace between the end tag of one and the start tag of the next element - this should be ignored when the document is processed to some output format. An element like this is called an element with "element content".

The second declaration starts with the keyword `ATTLIST` and the element name `Book`. After that there is a triple of whitespace separated parameters (in general an arbitrary number of such triples, one for each allowed attribute name). The first (`Name`) is the name of an attribute for a `Book` element. The second (`CDATA`) is always the same for all of our declarations, it means that the value of the attribute consists of "character data". The third parameter `#REQUIRED` means that this attribute must be specified with any `Book` element. Later we will also see optional attributes which are declared as `#IMPLIED`.

3.2.2 <TitlePage>

From gapdoc.dtd
```
<!ELEMENT TitlePage (Title, Subtitle?, Version?, TitleComment?,
                     Author+, Date?, Address?, Abstract?, Copyright?,
                     Acknowledgements? , Colophon? )>
```

Within this element information for the title page is collected. Note that more than one author can be specified. The elements must appear in this order because there is no sensible way to specify in a DTD something like "the following elements may occur in any order but each exactly once".

Before going on with the other elements inside the `Book` element we explain the elements for the title page.

3.2.3 <Title>

From gapdoc.dtd
```
<!ELEMENT Title (%Text;)*>
```

Here is the last construct you need to understand for reading gapdoc.dtd. The expression "%Text;" is a so-called "parameter entity". It is something like a macro within the DTD. It is defined as follows:

From gapdoc.dtd
```
<!ENTITY % Text "%InnerText; | List | Enum | Table">
```

This means, that every occurrence of "%Text;" in the DTD is replaced by the expression

From gapdoc.dtd
```
%InnerText; | List | Enum | Table
```

which is then expanded further because of the following definition:

From gapdoc.dtd
```
<!ENTITY % InnerText "#PCDATA |
                      Alt |
                      Emph | E |
                      Par | P | Br |
                      Keyword | K | Arg | A | Quoted | Q | Code | C |
                      File | F | Button | B | Package |
                      M | Math | Display |
                      Example | Listing | Log | Verb |
                      URL | Email | Homepage | Address | Cite | Label |
                      Ref | Index" >
```

These are the only two parameter entities we are using. They expand to lists of element names which are explained in the sequel *and* the keyword #PCDATA (concatenated with the "or" character "|").

So, the element (Title) is of so-called "mixed content": It can contain *parsed character data* which does not contain further markup (#PCDATA) or any of the other above mentioned elements. Mixed content must always have the asterisk qualifier (like in Title) such that any sequence of elements (of the above list) and character data can be contained in a Title element.

The %Text; parameter entity is used in all places in the DTD, where "normal text" should be allowed, including lists, enumerations, and tables, but *no* sectioning elements.

The %InnerText; parameter entity is used in all places in the DTD, where "inner text" should be allowed. This means, that no structures like lists, enumerations, and tables are allowed. This is used for example in headings.

3.2.4 <Subtitle>

From gapdoc.dtd
```
<!ELEMENT Subtitle (%Text;)*>
```

Contains the subtitle of the document.

3.2.5 <Version>

From gapdoc.dtd
```
<!ELEMENT Version (#PCDATA|Alt)*>
```

Note that the version can only contain character data and no further markup elements (except for Alt, which is necessary to resolve the entities described in 2.2.3). The converters will *not* put the word "Version" in front of the text in this element.

3.2.6 <TitleComment>

From gapdoc.dtd
```
<!ELEMENT TitleComment (%Text;)*>
```

Sometimes a title and subtitle are not sufficient to give a rough idea about the content of a package. In this case use this optional element to specify an additional text for the front page of the book. This text should be short, use the `Abstract` element (see 3.2.10) for longer explanations.

3.2.7 <Author>

From gapdoc.dtd
```
<!ELEMENT Author (%Text;)*>    <!-- There may be more than one Author! -->
```

As noted in the comment there may be more than one element of this type. This element should contain the name of an author and probably an `Email`-address and/or `WWW-Homepage` element for this author, see 3.5.6 and 3.5.7. You can also specify an individual postal address here, instead of using the `Address` element described below, see 3.2.9.

3.2.8 <Date>

From gapdoc.dtd
```
<!ELEMENT Date (#PCDATA)>
```

Only character data is allowed in this element which gives a date for the document. No automatic formatting is done.

3.2.9 <Address>

From gapdoc.dtd
```
<!ELEMENT Address (#PCDATA|Alt|Br)*>
```

This optional element can be used to specify a postal address of the author or the authors. If there are several authors with different addresses then put the `Address` elements inside the `Author` elements.

Use the `Br` element (see 3.9.3) to mark the line breaks in the usual formatting of the address on a letter.

Note that often it is not necessary to use this element because a postal address is easy to find via a link to a personal web page.

3.2.10 <Abstract>

From gapdoc.dtd
```
<!ELEMENT Abstract (%Text;)*>
```

This element contains an abstract of the whole book.

3.2.11 <Copyright>

From gapdoc.dtd
```
<!ELEMENT Copyright (%Text;)*>
```

This element is used for the copyright notice. Note the `&copyright;` entity as described in section 2.2.3.

3.2.12 <Acknowledgements>

From gapdoc.dtd
```
<!ELEMENT Acknowledgements (%Text;)*>
```

This element contains the acknowledgements.

3.2.13 <Colophon>

From gapdoc.dtd
```
<!ELEMENT Colophon (%Text;)*>
```

The “colophon” page is used to say something about the history of a document.

3.2.14 <TableOfContents>

From gapdoc.dtd
```
<!ELEMENT TableOfContents EMPTY>
```

This element may occur in the `Book` element after the `TitlePage` element. If it is present, a table of contents is generated and inserted into the document. Note that because this element is declared to be `EMPTY` one can use the abbreviation

Example
```
<TableOfContents/>
```

to denote this empty element.

3.2.15 <Bibliography>

From gapdoc.dtd
```
<!ELEMENT Bibliography EMPTY>
<!ATTLIST Bibliography Databases CDATA #REQUIRED
                       Style CDATA #IMPLIED>
```

This element may occur in the `Book` element after the last `Appendix` element. If it is present, a bibliography section is generated and inserted into the document. The attribute `Databases` must be specified, the names of several data files can be specified, separated by commas.

Two kinds of files can be specified in `Databases`: The first are BibTEX files as defined in [Lam85, Appendix B]. Such files must have a name with extension `.bib`, and in `Databases` the name must be given *without* this extension. Note that such `.bib`-files should be in latin1-encoding (or ASCII-encoding). The second are files in BibXMLext format as defined in Section 7.2. These files must have an extension `.xml` and in `Databases` the *full* name must be specified.

We suggest to use the BibXMLext format because it allows to produce potentially nicer bibliography entries in text and HTML documents.

A bibliography style may be specified with the `Style` attribute. The optional `Style` attribute (for LATEX output of the document) must also be specified without the `.bst` extension (the default is `alpha`). See also section 3.5.3 for a description of the `Cite` element which is used to include bibliography references into the text.

3.2.16 <TheIndex>

```
From gapdoc.dtd
<!ELEMENT TheIndex EMPTY>
```

This element may occur in the Book element after the Bibliography element. If it is present, an index is generated and inserted into the document. There are elements in GAPDoc which implicitly generate index entries (e.g., Func (3.4.2)) and there is an element Index (3.5.4) for explicitly adding index entries.

3.3 Sectioning Elements

A GAPDoc book is divided into *chapters*, *sections*, and *subsections*. The idea is of course, that a chapter consists of sections, which in turn consist of subsections. However for the sake of flexibility, the rules are not too restrictive. Firstly, text is allowed everywhere in the body of the document (and not only within sections). Secondly, the chapter level may be omitted. The exact rules are described below.

Appendices are a flavor of chapters, occurring after all regular chapters. There is a special type of subsection called “ManSection”. This is a subsection devoted to the description of a function, operation or variable. It is analogous to a manpage in the UNIX environment. Usually each function, operation, method, and so on should have its own ManSection.

Cross referencing is done on the level of Subsections, respectively ManSections. The topics in GAP’s online help are also pointing to subsections. So, they should not be too long.

We start our description of the sectioning elements “top-down”:

3.3.1 <Body>

The Body element marks the main part of the document. It must occur after the TableOfContents element. There is a big difference between *inside* and *outside* of this element: Whereas regular text is allowed nearly everywhere in the Body element and its subelements, this is not true for the *outside*. This has also implications on the handling of whitespace. *Outside* superfluous whitespace is usually ignored when it occurs between elements. *Inside* of the Body element whitespace matters because character data is allowed nearly everywhere. Here is the definition in the DTD:

```
From gapdoc.dtd
<!ELEMENT Body  ( %Text;| Chapter | Section )*>
```

The fact that Chapter and Section elements are allowed here leads to the possibility to omit the chapter level entirely in the document. For a description of %Text; see 3.2.3.

(Remark: The purpose of this element is to make sure that a *valid* GAPDoc document has a correct overall structure, which is only possible when the top element Book has element content.)

3.3.2 <Chapter>

```
From gapdoc.dtd
<!ELEMENT Chapter (%Text;| Heading | Section)*>
<!ATTLIST Chapter Label CDATA #IMPLIED>    <!-- For reference purposes -->
```

A `Chapter` element can have a `Label` attribute, such that this chapter can be referenced later on with a `Ref` element (see section 3.5.1). Note that you have to specify a label to reference the chapter as there is no automatic labelling!

`Chapter` elements can contain text (for a description of `%Text;` see 3.2.3), `Section` elements, and `Heading` elements.

The following *additional* rule cannot be stated in the DTD because we want a `Chapter` element to have mixed content. There must be *exactly one* `Heading` element in the `Chapter` element, containing the heading of the chapter. Here is its definition:

3.3.3 `<Heading>`

From gapdoc.dtd
```
<!ELEMENT Heading (%InnerText;)*>
```

This element is used for headings in `Chapter`, `Section`, `Subsection`, and `Appendix` elements. It may only contain `%InnerText;` (for a description see 3.2.3).

Each of the mentioned sectioning elements must contain exactly one direct `Heading` element (i.e., one which is not contained in another sectioning element).

3.3.4 `<Appendix>`

From gapdoc.dtd
```
<!ELEMENT Appendix (%Text;| Heading | Section)*>
<!ATTLIST Appendix Label CDATA #IMPLIED>   <!-- For reference purposes -->
```

The `Appendix` element behaves exactly like a `Chapter` element (see 3.3.2) except for the position within the document and the numbering. While chapters are counted with numbers (1., 2., 3., ...) the appendices are counted with capital letters (A., B., ...).

Again there is an optional `Label` attribute used for references.

3.3.5 `<Section>`

From gapdoc.dtd
```
<!ELEMENT Section (%Text;| Heading | Subsection | ManSection)*>
<!ATTLIST Section Label CDATA #IMPLIED>    <!-- For reference purposes -->
```

A `Section` element can have a `Label` attribute, such that this section can be referenced later on with a `Ref` element (see section 3.5.1). Note that you have to specify a label to reference the section as there is no automatic labelling!

`Section` elements can contain text (for a description of `%Text;` see 3.2.3), `Heading` elements, and subsections.

There must be exactly one direct `Heading` element in a `Section` element, containing the heading of the section.

Note that a subsection is either a `Subsection` element or a `ManSection` element.

3.3.6 `<Subsection>`

From gapdoc.dtd
```
<!ELEMENT Subsection (%Text;| Heading)*>
<!ATTLIST Subsection Label CDATA #IMPLIED> <!-- For reference purposes -->
```

The Subsection element can have a Label attribute, such that this subsection can be referenced later on with a Ref element (see section 3.5.1). Note that you have to specify a label to reference the subsection as there is no automatic labelling!

Subsection elements can contain text (for a description of %Text; see 3.2.3), and Heading elements.

There must be exactly one Heading element in a Subsection element, containing the heading of the subsection.

Another type of subsection is a ManSection, explained now:

3.4 ManSection–a special kind of subsection

ManSections are intended to describe a function, operation, method, variable, or some other technical instance. It is analogous to a manpage in the UNIX environment.

3.4.1 <ManSection>

From gapdoc.dtd

```
<!ELEMENT ManSection ( Heading?,
                      ((Func, Returns?) | (Oper, Returns?) |
                       (Meth, Returns?) | (Filt, Returns?) |
                       (Prop, Returns?) | (Attr, Returns?) |
                       (Constr, Returns?) |
                       Var | Fam | InfoClass)+, Description )>
<!ATTLIST ManSection Label CDATA #IMPLIED> <!-- For reference purposes -->

<!ELEMENT Returns (%Text;)*>
<!ELEMENT Description (%Text;)*>
```

The ManSection element can have a Label attribute, such that this subsection can be referenced later on with a Ref element (see section 3.5.1). But this is probably rarely necessary because the elements Func and so on (explained below) generate automatically labels for cross referencing.

The content of a ManSection element is one or more elements describing certain items in GAP, each of them optionally followed by a Returns element, followed by a Description element, which contains %Text; (see 3.2.3) describing it. (Remember to include examples in the description as often as possible, see 3.7.10). The classes of items GAPDoc knows of are: functions (Func), operations (Oper), constructors (Constr), methods (Meth), filters (Filt), properties (Prop), attributes (Attr), variables (Var), families (Fam), and info classes (InfoClass). One ManSection should only describe several of such items when these are very closely related.

Each element for an item corresponding to a GAP function can be followed by a Returns element. In output versions of the document the string "Returns: " will be put in front of the content text. The text in the Returns element should usually be a short hint about the type of object returned by the function. This is intended to give a good mnemonic for the use of a function (together with a good choice of names for the formal arguments).

ManSections are also sectioning elements which count as subsections. Usually there should be no Heading-element in a ManSection, in that case a heading is generated automatically from the first Func-like element. Sometimes this default behaviour does not look appropriate, for example when there are several Func-like elements. For such cases an optional Heading is allowed.

3.4.2 <Func>

From gapdoc.dtd
```
<!ELEMENT Func EMPTY>
<!ATTLIST Func Name  CDATA #REQUIRED
               Label CDATA #IMPLIED
               Arg   CDATA #REQUIRED
               Comm  CDATA #IMPLIED>
```

This element is used within a `ManSection` element to specify the usage of a function. The `Name` attribute is required and its value is the name of the function. The value of the `Arg` attribute (also required) contains the full list of arguments including optional parts, which are denoted by square brackets. The argument names can be separated by whitespace, commas or the square brackets for the optional arguments, like `"grp[, elm]"` or `"xx[y[z] ]"`. If GAP options are used, this can be followed by a colon `:` and one or more assignments, like `"n[, r]: tries := 100"`.

The name of the function is also used as label for cross referencing. When the name of the function appears in the text of the document it should *always* be written with the `Ref` element, see 3.5.1. This allows to use a unique typesetting style for function names and automatic cross referencing.

If the optional `Label` attribute is given, it is appended (with a colon `:` in between) to the name of the function for cross referencing purposes. The text of the label can also appear in the document text. So, it should be a kind of short explanation.

Example
```
<Func Arg="x[, y]" Name="LibFunc" Label="for my objects"/>
```

The optional `Comm` attribute should be a short description of the function, usually at most one line long (this is currently nowhere used).

This element automatically produces an index entry with the name of the function and, if present, the text of the `Label` attribute as subentry (see also 3.2.16 and 3.5.4).

3.4.3 <Oper>

From gapdoc.dtd
```
<!ELEMENT Oper EMPTY>
<!ATTLIST Oper Name  CDATA #REQUIRED
               Label CDATA #IMPLIED
               Arg   CDATA #REQUIRED
               Comm  CDATA #IMPLIED>
```

This element is used within a `ManSection` element to specify the usage of an operation. The attributes are used exactly in the same way as in the `Func` element (see 3.4.2).

Note that multiple descriptions of the same operation may occur in a document because there may be several declarations in GAP. Furthermore there may be several `ManSections` for methods of this operation (see 3.4.5) which also use the same name. For reference purposes these must be distinguished by different `Label` attributes.

3.4.4 <Constr>

From gapdoc.dtd
```
<!ELEMENT Constr EMPTY>
<!ATTLIST Constr Name  CDATA #REQUIRED
                 Label CDATA #IMPLIED
```

```
                  Arg   CDATA #REQUIRED
                  Comm  CDATA #IMPLIED>
```

This element is used within a ManSection element to specify the usage of a constructor. The attributes are used exactly in the same way as in the Func element (see 3.4.2).

Note that multiple descriptions of the same constructor may occur in a document because there may be several declarations in GAP. Furthermore there may be several ManSections for methods of this constructor (see 3.4.5) which also use the same name. For reference purposes these must be distinguished by different Label attributes.

3.4.5 <Meth>

From gapdoc.dtd

```
<!ELEMENT Meth EMPTY>
<!ATTLIST Meth Name  CDATA #REQUIRED
               Label CDATA #IMPLIED
               Arg   CDATA #REQUIRED
               Comm  CDATA #IMPLIED>
```

This element is used within a ManSection element to specify the usage of a method. The attributes are used exactly in the same way as in the Func element (see 3.4.2).

Frequently, an operation is implemented by several different methods. Therefore it seems to be interesting to document them independently. This is possible by using the same method name in different ManSections. It is however required that these subsections and those describing the corresponding operation are distinguished by different Label attributes.

3.4.6 <Filt>

From gapdoc.dtd

```
<!ELEMENT Filt EMPTY>
<!ATTLIST Filt Name  CDATA #REQUIRED
               Label CDATA #IMPLIED
               Arg   CDATA #IMPLIED
               Comm  CDATA #IMPLIED
               Type  CDATA #IMPLIED>
```

This element is used within a ManSection element to specify the usage of a filter. The first four attributes are used in the same way as in the Func element (see 3.4.2), except that the Arg attribute is optional.

The Type attribute can be any string, but it is thought to be something like “Category” or “Representation”.

3.4.7 <Prop>

From gapdoc.dtd

```
<!ELEMENT Prop EMPTY>
<!ATTLIST Prop Name  CDATA #REQUIRED
               Label CDATA #IMPLIED
               Arg   CDATA #REQUIRED
               Comm  CDATA #IMPLIED>
```

This element is used within a ManSection element to specify the usage of a property. The attributes are used exactly in the same way as in the Func element (see 3.4.2).

3.4.8 <Attr>

From gapdoc.dtd

```
<!ELEMENT Attr EMPTY>
<!ATTLIST Attr Name  CDATA #REQUIRED
               Label CDATA #IMPLIED
               Arg   CDATA #REQUIRED
               Comm  CDATA #IMPLIED>
```

This element is used within a ManSection element to specify the usage of an attribute (in GAP). The attributes are used exactly in the same way as in the Func element (see 3.4.2).

3.4.9 <Var>

From gapdoc.dtd

```
<!ELEMENT Var  EMPTY>
<!ATTLIST Var  Name  CDATA #REQUIRED
               Label CDATA #IMPLIED
               Comm  CDATA #IMPLIED>
```

This element is used within a ManSection element to document a global variable. The attributes are used exactly in the same way as in the Func element (see 3.4.2) except that there is no Arg attribute.

3.4.10 <Fam>

From gapdoc.dtd

```
<!ELEMENT Fam  EMPTY>
<!ATTLIST Fam  Name  CDATA #REQUIRED
               Label CDATA #IMPLIED
               Comm  CDATA #IMPLIED>
```

This element is used within a ManSection element to document a family. The attributes are used exactly in the same way as in the Func element (see 3.4.2) except that there is no Arg attribute.

3.4.11 <InfoClass>

From gapdoc.dtd

```
<!ELEMENT InfoClass EMPTY>
<!ATTLIST InfoClass Name  CDATA #REQUIRED
                    Label CDATA #IMPLIED
                    Comm  CDATA #IMPLIED>
```

This element is used within a ManSection element to document an info class. The attributes are used exactly in the same way as in the Func element (see 3.4.2) except that there is no Arg attribute.

3.5 Cross Referencing and Citations

Cross referencing in the GAPDoc system is somewhat different to the usual LaTeX cross referencing in so far, that a reference knows "which type of object" it is referencing. For example a "reference to a function" is distinguished from a "reference to a chapter". The idea of this is, that the markup must contain this information such that the converters can produce better output. The HTML converter can for example typeset a function reference just as the name of the function with a link to the description of the function, or a chapter reference as a number with a link in the other case.

Referencing is done with the `Ref` element:

3.5.1 <Ref>

From gapdoc.dtd

```
<!ELEMENT Ref EMPTY>
<!ATTLIST Ref Func      CDATA #IMPLIED
              Oper      CDATA #IMPLIED
              Constr    CDATA #IMPLIED
              Meth      CDATA #IMPLIED
              Filt      CDATA #IMPLIED
              Prop      CDATA #IMPLIED
              Attr      CDATA #IMPLIED
              Var       CDATA #IMPLIED
              Fam       CDATA #IMPLIED
              InfoClass CDATA #IMPLIED
              Chap      CDATA #IMPLIED
              Sect      CDATA #IMPLIED
              Subsect   CDATA #IMPLIED
              Appendix  CDATA #IMPLIED
              Text      CDATA #IMPLIED

              Label     CDATA #IMPLIED
              BookName  CDATA #IMPLIED
              Style (Text | Number) #IMPLIED>  <!-- normally automatic -->
```

The `Ref` element is defined to be `EMPTY`. If one of the attributes `Func`, `Oper`, `Constr`, `Meth`, `Prop`, `Attr`, `Var`, `Fam`, `InfoClass`, `Chap`, `Sect`, `Subsect`, `Appendix` is given then there must be exactly one of these, making the reference one to the corresponding object. The `Label` attribute can be specified in addition to make the reference unique, for example if more than one method with a given name is present. (Note that there is no way to specify in the DTD that exactly one of the first listed attributes must be given, this is an additional rule.)

A reference to a `Label` element defined below (see 3.5.2) is done by giving the `Label` attribute and optionally the `Text` attribute. If the `Text` attribute is present its value is typeset in place of the `Ref` element, if linking is possible (for example in HTML). If this is not possible, the section number is typeset. This type of reference is also used for references to tables (see 3.6.5).

An external reference into another book can be specified by using the `BookName` attribute. In this case the `Label` attribute or, if this is not given, the function or section like attribute, is used to resolve the reference. The generated reference points to the first hit when asking "?book name: label" inside GAP.

The optional attribute `Style` can take only the values `Text` and `Number`. It can be used with references to sectioning units and it gives a hint to the converter programs, whether an explicit section number is generated or text. Normally all references to sections generate numbers and references to a GAP object generate the name of the corresponding object with some additional link or sectioning information, which is the behavior of `Style="Text"`. In case `Style="Number"` in all cases an explicit section number is generated. So

Example
```
<Ref Subsect="Func" Style="Text"/> described in section
<Ref Subsect="Func" Style="Number"/>
```

produces: '`<Func>`' described in section 3.4.2.

3.5.2 <Label>

From gapdoc.dtd
```
<!ELEMENT Label EMPTY>
<!ATTLIST Label Name CDATA #REQUIRED>
```

This element is used to define a label for referencing a certain position in the document, if this is possible. If an exact reference is not possible (like in a printed version of the document) a reference to the corresponding subsection is generated. The value of the `Name` attribute must be unique under all `Label` elements.

3.5.3 <Cite>

From gapdoc.dtd
```
<!ELEMENT Cite EMPTY>
<!ATTLIST Cite Key CDATA #REQUIRED
               Where CDATA #IMPLIED>
```

This element is for bibliography citations. It is `EMPTY` by definition. The attribute `Key` is the key for a lookup in a BibTeX database that has to be specified in the `Bibliography` element (see 3.2.15). The value of the `Where` attribute specifies the position in the document as in the corresponding LaTeX syntax `\cite[Where value]{Key value}`.

3.5.4 <Index>

From gapdoc.dtd
```
<!ELEMENT Index (%InnerText;|Subkey)*>
<!ATTLIST Index Key    CDATA #IMPLIED
                Subkey CDATA #IMPLIED>
<!ELEMENT Subkey (%InnerText;)*>
```

This element generates an index entry. The content of the element is typeset in the index. It can optionally contain a `Subkey` element. If one or both of the attributes `Key` and `Subkey` are given, then the attribute values are used for sorting the index entries. Otherwise the content itself is used for sorting. The attributes should be used when the content contains markup. Note that all `Func` and similar elements automatically generate index entries. If the `TheIndex` element (3.2.16) is not present in the document all `Index` elements are ignored.

3.5.5 <URL>

From gapdoc.dtd
```
<!ELEMENT URL (#PCDATA|Alt|Link|LinkText)*>  <!-- Link, LinkText
     variant for case where text needs further markup -->
<!ATTLIST URL Text CDATA #IMPLIED>   <!-- This is for output formats
                                          that have links like HTML -->
<!ELEMENT Link     (%InnerText;)*> <!-- the URL -->
<!ELEMENT LinkText (%InnerText;)*> <!-- text for links, can contain markup -->
```

This element is for references into the internet. It specifies an URL and optionally a text which can be used for a link (like in HTML or PDF versions of the document). This can be specified in two ways: Either the URL is given as element content and the text is given in the optional `Text` attribute (in this case the text cannot contain further markup), or the element contains the two elements `Link` and `LinkText` which in turn contain the URL and the text, respectively. The default value for the text is the URL itself.

3.5.6 <Email>

From gapdoc.dtd
```
<!ELEMENT Email (#PCDATA|Alt|Link|LinkText)*>
```

This element type is the special case of an URL specifying an email address. The content of the element should be the email address without any prefix like “`mailto:`”. This address is typeset by all converters, also without any prefix. In the case of an output document format like HTML the converter can produce a link with a “`mailto:`” prefix.

3.5.7 <Homepage>

From gapdoc.dtd
```
<!ELEMENT Homepage (#PCDATA|Alt|Link|LinkText)*>
```

This element type is the special case of an URL specifying a WWW-homepage.

3.6 Structural Elements like Lists

The GAPDoc system offers some limited access to structural elements like lists, enumerations, and tables. Although it is possible to use all LaTeX constructs one always has to think about other output formats. The elements in this section are guaranteed to produce something reasonable in all output formats.

3.6.1 <List>

From gapdoc.dtd
```
<!ELEMENT List ( ((Mark,Item)|Item)+ )>
<!ATTLIST List Only CDATA #IMPLIED
               Not  CDATA #IMPLIED>
```

This element produces a list. Each item in the list corresponds to an `Item` element. Every `Item` element is optionally preceded by a `Mark` element. The content of this is used as a marker for the item.

Note that this marker can be a whole word or even a sentence. It will be typeset in some emphasized fashion and most converters will provide some indentation for the rest of the item.

The `Only` and `Not` attributes can be used to specify, that the list is included into the output by only one type of converter (`Only`) or all but one type of converter (`Not`). Of course at most one of the two attributes may occur in one element. The following values are allowed as of now: “`LaTeX`”, “`HTML`”, and “`Text`”. See also the `Alt` element in 3.9.1 for more about text alternatives for certain converters.

3.6.2 <Mark>

From gapdoc.dtd

```
<!ELEMENT Mark ( %InnerText;)*>
```

This element is used in the `List` element to mark items. See 3.6.1 for an explanation.

3.6.3 <Item>

From gapdoc.dtd

```
<!ELEMENT Item ( %Text;)*>
```

This element is used in the `List`, `Enum`, and `Table` elements to specify the items. See sections 3.6.1, 3.6.4, and 3.6.5 for further information.

3.6.4 <Enum>

From gapdoc.dtd

```
<!ELEMENT Enum ( Item+ )>
<!ATTLIST Enum Only CDATA #IMPLIED
               Not  CDATA #IMPLIED>
```

This element is used like the `List` element (see 3.6.1) except that the items must not have marks attached to them. Instead, the items are numbered automatically. The same comments about the `Only` and `Not` attributes as above apply.

3.6.5 <Table>

From gapdoc.dtd

```
<!ELEMENT Table ( Caption?, (Row | HorLine)+ )>
<!ATTLIST Table Label   CDATA #IMPLIED
                Only    CDATA #IMPLIED
                Not     CDATA #IMPLIED
                Align   CDATA #REQUIRED>
                <!-- We allow | and l,c,r, nothing else -->
<!ELEMENT Row   ( Item+ )>
<!ELEMENT HorLine EMPTY>
<!ELEMENT Caption ( %InnerText;)*>
```

A table in GAPDoc consists of an optional `Caption` element followed by a sequence of `Row` and `HorLine` elements. A `HorLine` element produces a horizontal line in the table. A `Row` element consists of a sequence of `Item` elements as they also occur in `List` and `Enum` elements. The `Only` and `Not` attributes have the same functionality as described in the `List` element in 3.6.1.

The `Align` attribute is written like a LaTeX tabular alignment specifier but only the letters "`l`", "`r`", "`c`", and "`|`" are allowed meaning left alignment, right alignment, centered alignment, and a vertical line as delimiter between columns respectively.

If the `Label` attribute is there, one can reference the table with the `Ref` element (see 3.5.1) using its `Label` attribute.

Usually only simple tables should be used. If you want a complicated table in the LaTeX output you should provide alternatives for text and HTML output. Note that in HTML-4.0 there is no possibility to interpret the "`|`" column separators and `HorLine` elements as intended. There are lines between all columns and rows or no lines at all.

3.7 Types of Text

This section covers the markup of text. Various types of "text" exist. The following elements are used in the GAPDoc system to mark them. They mostly come in pairs, one long name which is easier to remember and a shortcut to make the markup "lighter".

Most of the following elements are thought to contain only character data and no further markup elements. It is however necessary to allow `Alt` elements to resolve the entities described in section 2.2.3.

3.7.1 `<Emph>` and `<E>`

From gapdoc.dtd

```
<!ELEMENT Emph (%InnerText;)*>    <!-- Emphasize something -->
<!ELEMENT E    (%InnerText;)*>    <!-- the same as shortcut -->
```

This element is used to emphasize some piece of text. It may contain `%InnerText;` (see 3.2.3).

3.7.2 `<Quoted>` and `<Q>`

From gapdoc.dtd

```
<!ELEMENT Quoted (%InnerText;)*>   <!-- Quoted (in quotes) text -->
<!ELEMENT Q (%InnerText;)*>        <!-- Quoted text (shortcut) -->
```

This element is used to put some piece of text into " "-quotes. It may contain `%InnerText;` (see 3.2.3).

3.7.3 `<Keyword>` and `<K>`

From gapdoc.dtd

```
<!ELEMENT Keyword (#PCDATA|Alt)*>  <!-- Keyword -->
<!ELEMENT K (#PCDATA|Alt)*>        <!-- Keyword (shortcut) -->
```

This element is used to mark something as a *keyword*. Usually this will be a GAP keyword such as "`if`" or "`for`". No further markup elements are allowed within this element except for the `Alt` element, which is necessary.

3.7.4 <Arg> and <A>

From gapdoc.dtd
```
<!ELEMENT Arg (#PCDATA|Alt)*>     <!-- Argument -->
<!ELEMENT A (#PCDATA|Alt)*>       <!-- Argument (shortcut) -->
```

This element is used inside Descriptions in ManSections to mark something as an *argument* (of a function, operation, or such). It is guaranteed that the converters typeset those exactly as in the definition of functions. No further markup elements are allowed within this element.

3.7.5 <Code> and <C>

From gapdoc.dtd
```
<!ELEMENT Code (#PCDATA|Arg|Alt)*>     <!-- GAP code -->
<!ELEMENT C (#PCDATA|Arg|Alt)*>        <!-- GAP code (shortcut) -->
```

This element is used to mark something as a piece of *code* like for example a GAP expression. It is guaranteed that the converters typeset this exactly as in the Listing element (compare section 3.7.9). The only further markup elements allowed within this element are <Arg> elements (see 3.7.4).

3.7.6 <File> and <F>

From gapdoc.dtd
```
<!ELEMENT File (#PCDATA|Alt)*>    <!-- Filename -->
<!ELEMENT F (#PCDATA|Alt)*>       <!-- Filename (shortcut) -->
```

This element is used to mark something as a *filename* or a *pathname* in the file system. No further markup elements are allowed within this element.

3.7.7 <Button> and <B>

From gapdoc.dtd
```
<!ELEMENT Button (#PCDATA|Alt)*>  <!-- "Button" (also Menu, Key, ...) -->
<!ELEMENT B (#PCDATA|Alt)*>       <!-- "Button" (shortcut) -->
```

This element is used to mark something as a *button*. It can also be used for other items in a graphical user interface like *menus*, *menu entries*, or *keys*. No further markup elements are allowed within this element.

3.7.8 <Package>

From gapdoc.dtd
```
<!ELEMENT Package (#PCDATA|Alt)*>   <!-- A package name -->
```

This element is used to mark something as a name of a *package*. This is for example used to define the entities GAP, XGAP or GAPDoc (see section 2.2.3). No further markup elements are allowed within this element.

3.7.9 <Listing>

From gapdoc.dtd
```
<!ELEMENT Listing (#PCDATA)>  <!-- This is just for GAP code listings -->
<!ATTLIST Listing Type CDATA #IMPLIED> <!-- a comment about the type of
                                            listed code, may appear in
                                            output -->
```

This element is used to embed listings of programs into the document. Only character data and no other elements are allowed in the content. You should *not* use the character entities described in section 2.2.3 but instead type the characters directly. Only the general XML rules from section 2.1 apply. Note especially the usage of `<![CDATA[` sections described there. It is guaranteed that all converters use a fixed width font for typesetting `Listing` elements. Compare also the usage of the `Code` and `C` elements in 3.7.5.

The `Type` attribute contains a comment about the type of listed code. It may appear in the output.

3.7.10 <Log> and <Example>

From gapdoc.dtd
```
<!ELEMENT Example (#PCDATA)>  <!-- This is subject to the automatic
                                   example checking mechanism -->
<!ELEMENT Log (#PCDATA)>      <!-- This not -->
```

These two elements behave exactly like the `Listing` element (see 3.7.9). They are thought for protocols of GAP sessions. The only difference between the two is that `Example` sections are intended to be subject to an automatic manual checking mechanism used to ensure the correctness of the GAP manual whereas `Log` is not touched by this (see section 5.4 for checking tools).

To get a good layout of the examples for display in a standard terminal we suggest to use `SizeScreen([72]);` (see `SizeScreen` (**Reference: SizeScreen**)) in your GAP session before producing the content of `Example` elements.

3.7.11 <Verb>

There is one further type of verbatim-like element.

From gapdoc.dtd
```
<!ELEMENT Verb  (#PCDATA)>
```

The content of such an element is guaranteed to be put into an output version exactly as it is using some fixed width font. Before the content a new line is started. If the line after the end of the start tag consists of whitespace only then this part of the content is skipped.

This element is intended to be used together with the `Alt` element to specify pre-formatted ASCII alternatives for complicated `Display` formulae or `Tables`.

3.8 Elements for Mathematical Formulae

3.8.1 <Math> and <Display>

From gapdoc.dtd
```
<!-- Normal TeX math mode formula -->
<!ELEMENT Math (#PCDATA|A|Arg|Alt)*>
<!-- TeX displayed math mode formula -->
```

```
<!ELEMENT Display (#PCDATA|A|Arg|Alt)*>
<!-- Mode="M" causes <M>-style formatting -->
<!ATTLIST Display Mode CDATA #IMPLIED>
```

These elements are used for mathematical formulae. As described in section 2.2.2 they correspond to LaTeX's math and display math mode respectively.

The formulae are typed in as in LaTeX, *except* that the standard XML entities, see 2.1.9 (in particular the characters < and &), must be escaped - either by using the corresponding entities or by enclosing the formula between "`<![CDATA[`" and "`]]>`". (The main reference for LaTeX is [Lam85].)

It is also possible to use some unicode characters for mathematical symbols directly, provided that it can be translated by `Encode` (6.2.2) into `"LaTeX"` encoding and that `SimplifiedUnicodeString` (6.2.2) with arguments `"latin1"` and `"single"` returns something sensible. Currently, we support entities `&CC;`, `&ZZ;`, `&NN;`, `&PP;`, `&QQ;`, `&HH;`, `&RR;` for the corresponding black board bold letters $\mathbb{C}$, $\mathbb{Z}$, $\mathbb{N}$, $\mathbb{P}$, $\mathbb{Q}$, $\mathbb{H}$ and $\mathbb{R}$, respectively.

The only element type that is allowed within the formula elements is the `Arg` or `A` element (see 3.7.4), which is used to typeset identifiers that are arguments to GAP functions or operations.

If a `Display` element has an attribute `Mode` with value `"M"`, then the formula is formatted as in `M` elements (see 3.8.2). Otherwise in text and HTML output the formula is shown as LaTeX source code.

For simple formulae (and you should try to make all your formulae simple!) attempt to use the `M` element or the `Mode="M"` attribute in `Display` for which there is a well defined translation into text, which can be used for text and HTML output versions of the document. So, if possible try to avoid the `Math` elements and `Display` elements without attribute or provide useful text substitutes for complicated formulae via `Alt` elements (see 3.9.1 and 3.7.11).

3.8.2 <M>

From gapdoc.dtd
```
<!-- Math with well defined translation to text output -->
<!ELEMENT M (#PCDATA|A|Arg|Alt)*>
```

The "`M`" element type is intended for formulae in the running text for which there is a sensible text version. For the LaTeX version of a GAPDoc document the `M` and `Math` elements are equivalent. The remarks in 3.8.1 about special characters and the `Arg` element apply here as well. A document which has all formulae enclosed in `M` elements can be well readable in text terminal output and printed output versions.

Compared to former versions of GAPDoc many more formulae can be put into `M` elements. Most modern terminal emulations support unicode characters and many mathematical symbols can now be represented by such characters. But even if a terminal can only display ASCII characters, the user will see some not too bad representation of a formula.

As examples, here are some LaTeX macros which have a sensible ASCII translation and are guaranteed to be translated accordingly by text (and HTML) converters (for a full list of handled Macros see `RecNames(TEXTMTRANSLATIONS)`):

\ast	`*`
\bf	
\bmod	`mod`
\cdot	`*`
\colon	`:`
\equiv	`=`
\geq	`>=`
\germ	
\hookrightarrow	`->`
\iff	`<=>`
\langle	`<`
\ldots	`...`
\left	
\leq	`<=`
\leftarrow	`<-`
\Leftarrow	`<=`
\limits	
\longrightarrow	`-->`
\Longrightarrow	`==>`
\mapsto	`->`
\mathbb	
\mathop	
\mid	`|`
\pmod	`mod`
\prime	`'`
\rangle	`>`
\right	
\rightarrow	`->`
\Rightarrow	`=>`
\rm, \sf, \textrm, \text	
\setminus	`\`
\thinspace	
\times	`x`
\to	`->`
\vert	`|`
\!	
\,	
\;	
\{	`{`
\}	`}`

Table: LaTeX macros with special text translation

In all other macros only the backslash is removed (except for some macros describing more exotic symbols). Whitespace is normalized (to one blank) but not removed. Note that whitespace is not added, so you may want to add a few more spaces than you usually do in your LaTeX documents.

Braces `{}` are removed in general, however pairs of double braces are converted to one pair of braces. This can be used to write `<M>x^{12}</M>` for `x^12` and `<M>x_{{i+1}}</M>` for `x_{i+1}`.

3.9 Everything else

3.9.1 <Alt>

This element is used to specify alternatives for different output formats within normal text. See also sections 3.6.1, 3.6.4, and 3.6.5 for alternatives in lists and tables.

From gapdoc.dtd
```
<!ELEMENT Alt (%InnerText;)*>  <!-- This is only to allow "Only" and
                                    "Not" attributes for normal text -->
<!ATTLIST Alt Only CDATA #IMPLIED
              Not  CDATA #IMPLIED>
```

Of course exactly one of the two attributes must occur in one element. The attribute values must be one word or a list of words, separated by spaces or commas. The words which are currently recognized by the converter programs contained in GAPDoc are: "LaTeX", "HTML", and "Text". If the Only attribute is specified then only the corresponding converter will include the content of the element into the output document. If the Not attribute is specified the corresponding converter will ignore the content of the element. You can use other words to specify special alternatives for other converters of GAPDoc documents.

In the case of "HTML" there is a second word which is recognized and this can either be "MathJax" or "noMathJax". For example a pair of Alt elements with `<Alt Only="HTML noMathJax">...` and `<Alt Not="HTML noMathJax">...` could provide special content for the case of HTML output without use of MathJax and every other output.

We fix a rule for handling the content of an Alt element with Only attribute. In their content code for the corresponding output format is included directly. So, in case of HTML the content is HTML code, in case of LaTeX the content is LaTeX code. The converters don't apply any handling of special characters to this content. In the case of LaTeX the formatting of the code is not changed.

Within the element only %InnerText; (see 3.2.3) is allowed. This is to ensure that the same set of chapters, sections, and subsections show up in all output formats.

3.9.2 <Par> and <P>

From gapdoc.dtd
```
<!ELEMENT Par EMPTY>    <!-- this is intentionally empty! -->
<!ELEMENT P EMPTY>      <!-- the same as shortcut  -->
```

This EMPTY element marks the boundary of paragraphs. Note that an empty line in the input does not mark a new paragraph as opposed to the LaTeX convention.

(Remark: it would be much easier to parse a document and to understand its sectioning and paragraph structure when there was an element whose *content* is the text of a paragraph. But in practice many paragraph boundaries are implicitly clear which would make it somewhat painful to enclose each paragraph in extra tags. The introduction of the P or Par elements as above delegates this pain to the writer of a conversion program for GAPDoc documents.)

3.9.3

From gapdoc.dtd
```
<!ELEMENT Br EMPTY>     <!-- a forced line break  -->
```

This element can be used to force a line break in the output versions of a GAPDoc element, it does not start a new paragraph. Please, do not use this instead of a `Par` element, this would often lead to ugly output versions of your document.

3.9.4 <Ignore>

From gapdoc.dtd
```
<!ELEMENT Ignore (%Text;| Chapter | Section | Subsection | ManSection |
                  Heading)*>
<!ATTLIST Ignore Remark CDATA #IMPLIED>
```

This element can appear anywhere. Its content is ignored by the standard converters. It can be used, for example, to include data which are not part of the actual GAPDoc document, like source code, or to make not finished parts of the document invisible.

Of course, one can use special converter programs which extract the contents of `Ignore` elements. Information on the type of the content can be stored in the optional attribute `Remark`.

Chapter 4

Distributing a Document into Several Files

In GAPDoc there are facilities to distribute a single document over several files. This is for example interesting, if one wants to store the documentation of some code in the same file as the code itself. Or, if one just wants to store chapters of a document in separate files. There is a set of conventions how this is done and some tools to collect the text for further processing.

The technique can also be used to distribute and collect other types of documents into respectively from several files (e.g., source code, examples).

4.1 The Conventions

In this description we use the string `GAPDoc` for marking pieces of a document to collect.

Pieces of documentation that shall be incorporated into another document are marked as follows:

Example
```
##  <#GAPDoc Label="MyPiece">
##  <E>This</E> is the piece.
##  The hash characters are removed.
##  <#/GAPDoc>
```

This piece is then included into another file by a statement like: `<#Include Label="MyPiece">` Here are the exact rules, how pieces are gathered:

- All lines up to a line containing the character sequence "`<#GAPDoc Label="`" (exactly one space character) are ignored. The characters on the same line before this sequence are stored as "prefix". The characters after the sequence up to the next double quotes character (which should not contain whitespace) are stored as "label". All other characters in the line are ignored.

- The following lines up to a line containing the character sequence "`<#/GAPDoc>`" are stored under the label. These lines are processed as follows: The longest possible substring from the beginning of the line that equals the corresponding substring of the prefix is removed.

Having stored a list of labels and pieces of text gathered as above this can be used as follows.

- In GAPDoc documentation files all statements of the form "`<#Include Label="Key">`" are replaced by the sequence of lines stored under the label `Key`.

- Additionally, every occurrence of a statement of the form "`<#Include SYSTEM "Filename">`" is replaced by the whole file stored under the name `Filename` in the file system.
- These substitutions are done recursively (although one should probably avoid to use this extensively).

Here is another example:

```
# # <#GAPDoc Label="AnotherPiece">  some characters
# # This text is not indented.
#  This text is indented by one blank.
#Not indented.
#<#/GAPDoc>
```

replaces `<#Include Label="AnotherPiece">` by

```
This text is not indented.
 This text is indented by one blank.
Not indented.
```

Since these rules are very simple it is quite easy to write a program in almost any programming language which does this gathering of text pieces and the substitutions. In GAPDoc there is the GAP function `ComposedDocument` (4.2.1) which does this.

Note that the XML-tag-like markup we have used here is not a legal XML markup, since the hash character is not allowed in element names. The mechanism described here is a preprocessing step which composes a document.

4.2 A Tool for Collecting a Document

4.2.1 ComposedDocument

▷ `ComposedDocument(`*`tagname, path, main, source[, info]`*`)` (function)
▷ `ComposedXMLString(`*`path, main, source[, info]`*`)` (function)

Returns: a document as string, or a list with this string and information about the source positions

The argument *`tagname`* is the string used for the pseudo elements which mark the pieces of a document to collect. (In 4.1 we used `GAPDoc` as *`tagname`*. The second function `ComposedXMLString( ... )` is an abbreviation for `ComposedDocument("GAPDoc", ... )`.

The argument *`path`* must be a path to some directory (as string or directory object), *`main`* the name of a file and *`source`* a list of file names. These file names are relative to *`path`*, except they start with `"/"` to specify an absolute path or they start with `"gap://"` to specify a file relative to the GAP roots (see `FilenameGAP` (4.2.3)). The document is constructed via the mechanism described in Section 4.1.

First the files given in *`source`* are scanned for chunks of the document marked by `<#`*`tagname`*` Label="...">` and `</#`*`tagname`*`>` pairs. Then the file *`main`* is read and all `<#Include ... >`-tags are substituted recursively by other files or chunks of documentation found in the first step, respectively.

If the optional argument *`info`* is given and set to `true` this function returns a list `[str, origin]`, where `str` is a string containing the composed document and `origin` is a sorted list of entries of the form [pos, filename, line]. Here pos runs through all character positions of starting lines or text pieces from different files in str. The filename and line describe the origin of this part of the collected document.

Without the fourth argument only the string str is returned.

By default ComposedDocument runs into an error if an <#Include ...>-tag cannot be substituted (because a file or chunk is missing). This behaviour can be changed by setting DOCCOMPOSEERROR := false;. Then the missing parts are substituted by a short note about what is missing. Of course, this feature is only useful if the resulting document is a valid XML document (e.g., when the missing pieces are complete paragraphs or sections).

Example
```
gap> doc := ComposedDocument("GAPDoc", "/my/dir", "manual.xml",
> ["../lib/func.gd", "../lib/func.gi"], true);;
```

4.2.2 OriginalPositionDocument

▷ OriginalPositionDocument(*srcinfo*, *pos*) (function)

Returns: A pair [filename, linenumber].

Here *srcinfo* must be a data structure as returned as second entry by ComposedDocument (4.2.1) called with *info*=true. It returns for a given position *pos* in the composed document the file name and line number from which that text was collected.

4.2.3 FilenameGAP

▷ FilenameGAP(*fname*) (function)

Returns: file name as string or fail

This functions returns the full path of a file with name *fname* relative to a GAP root path, or fail if such a file does not exist. The argument *fname* can optionally start with the prefix "gap://" which will be removed.

Example
```
gap> FilenameGAP("hsdkfhs.g");
fail
gap> FilenameGAP("lib/system.g");
"/usr/local/gap4/lib/system.g"
gap> FilenameGAP("gap://lib/system.g");
"/usr/local/gap4/lib/system.g"
```

Chapter 5

The Converters and an XML Parser

The GAPDoc package contains a set of programs which allow us to convert a GAPDoc book into several output versions and to make them available to GAP's online help.

Currently the following output formats are provided: text for browsing inside a terminal running GAP, LaTeX with hyperref-package for cross references via hyperlinks and HTML for reading with a Web-browser.

5.1 Producing Documentation from Source Files

Here we explain how to use the functions which are described in more detail in the following sections. We assume that we have the main file MyBook.xml of a book "MyBook" in the directory /my/book/path. This contains <#Include ...>-statements as explained in Chapter 4. These refer to some other files as well as pieces of text which are found in the comments of some GAP source files ../lib/a.gd and ../lib/b.gi (relative to the path above). A BibTeX database MyBook.bib for the citations is also in the directory given above. We want to produce a text-, pdf- and HTML-version of the document. (A LaTeX version of the manual is produced, so it is also easy to compile dvi-, and postscript-versions.)

All the commands shown in this Section are collected in the single function MakeGAPDocDoc (5.1.1).

First we construct the complete XML-document as a string with ComposedDocument (4.2.1). This interprets recursively the <#Include ...>-statements.

Example

```
gap> path := Directory("/my/book/path");;
gap> main := "MyBook.xml";;
gap> files := ["../lib/a.gd", "../lib/b.gi"];;
gap> bookname := "MyBook";;
gap> doc := ComposedDocument("GAPDoc", path, main, files, true);;
```

Now doc is a list with two entries, the first is a string containing the XML-document, the second gives information from which files and locations which part of the document was collected. This is useful in the next step, if there are any errors in the document.

Next we parse the document and store its structure in a tree-like data structure. The commands for this are ParseTreeXMLString (5.2.1) and CheckAndCleanGapDocTree (5.2.8).

```
gap> r := ParseTreeXMLString(doc[1], doc[2]);;
gap> CheckAndCleanGapDocTree(r);
true
```

We start to produce a text version of the manual, which can be read in a terminal (window). The command is `GAPDoc2Text` (5.3.2). This produces a record with the actual text and some additional information. The text can be written chapter-wise into files with `GAPDoc2TextPrintTextFiles` (5.3.3). The names of these files are `chap0.txt`, `chap1.txt` and so on. The text contains some markup using ANSI escape sequences. This markup is substituted by the GAP help system (user configurable) to show the text with colors and other attributes. For the bibliography we have to tell `GAPDoc2Text` (5.3.2) the location of the BibTEX database by specifying a `path` as second argument.

```
gap> t := GAPDoc2Text(r, path);;
gap> GAPDoc2TextPrintTextFiles(t, path);
```

This command constructs all parts of the document including table of contents, bibliography and index. The functions `FormatParagraph` (6.1.4) for formatting text paragraphs and `ParseBibFiles` (7.1.1) for reading BibTEX files with GAP may be of independent interest.

With the text version we have also produced the information which is used for searching with GAP's online help. Also, labels are produced which can be used by links in the HTML- and `pdf`-versions of the manual.

Next we produce a LaTEX version of the document. `GAPDoc2LaTeX` (5.3.1) returns a string containing the LaTEX source. The utility function `FileString` (6.3.5) writes the content of a string to a file, we choose `MyBook.tex`.

```
gap> l := GAPDoc2LaTeX(r);;
gap> FileString(Filename(path, Concatenation(bookname, ".tex")), l);
```

Assuming that you have a sufficiently good installation of TEX available (see `GAPDoc2LaTeX` (5.3.1) for details) this can be processed with a series of commands like in the following example.

```
cd /my/book/path
pdflatex MyBook
bibtex MyBook
pdflatex MyBook
makeindex MyBook
pdflatex MyBook
mv MyBook.pdf manual.pdf
```

After this we have a `pdf`-version of the document in the file `manual.pdf`. It contains hyperlink information which can be used with appropriate browsers for convenient reading of the document on screen (e.g., `xpdf` is nice because it allows remote calls to display named locations of the document). Of course, we could also use other commands like `latex` or `dvips` to process the LaTEX source file. Furthermore we have produced a file `MyBook.pnr` which is GAP-readable and contains the page number information for each (sub-)section of the document.

We can add this page number information to the indexing information collected by the text converter and then print a `manual.six` file which is read by GAP when the manual is loaded. This is done with `AddPageNumbersToSix` (5.3.4) and `PrintSixFile` (5.3.5).

```
gap> AddPageNumbersToSix(r, Filename(path, "MyBook.pnr"));
gap> PrintSixFile(Filename(path, "manual.six"), r, bookname);
```

Finally we produce an HTML-version of the document and write it (chapter-wise) into files `chap0.html`, `chap1.html` and so on. They can be read with any Web-browser. The commands are `GAPDoc2HTML` (5.3.7) and `GAPDoc2HTMLPrintHTMLFiles` (5.3.8). We also add a link from `manual.html` to `chap0.html`. You probably want to copy stylesheet files into the same directory, see 5.3.9 for more details. The argument `path` of `GAPDoc2HTML` (5.3.7) specifies the directory containing the BibTEX database files.

```
gap> h := GAPDoc2HTML(r, path);;
gap> GAPDoc2HTMLPrintHTMLFiles(h, path);
```

5.1.1 MakeGAPDocDoc

▷ `MakeGAPDocDoc(`*`path, main, files, bookname[, gaproot][, ...]`*`)` (function)

This function collects all the commands for producing a text-, `pdf`- and HTML-version of a GAPDoc document as described in Section 5.1. It checks the `.log` file from the call of `pdflatex` and reports if there are errors, warnings or overfull boxes.

Note: If this function works for you depends on your operating system and installed software. It will probably work on most `UNIX` systems with a standard LaTeX installation. If the function doesn't work for you look at the source code and adjust it to your system.

Here *path* must be the directory (as string or directory object) containing the main file *main* of the document (given with or without the `.xml` extension. The argument *files* is a list of (probably source code) files relative to *path* which contain pieces of documentation which must be included in the document, see Chapter 4. And *bookname* is the name of the book used by GAP's online help. The optional argument *gaproot* must be a string which gives the relative path from *path* to the main GAP root directory. If this is given, the HTML files are produced with relative paths to external books.

If the string `"nopdf"` is given as optional argument then `MakeGAPDocDoc` will not produce a `pdf`-version of the help book (the source `.tex`-file is generated). Consequently, the index for the help system will not contain page numbers for the `pdf`-version. This variant of `MakeGAPDocDoc` should work independently of the operating system because no external programs are called. It is recommended that distributed manuals contain the `pdf`-version.

`MakeGAPDocDoc` can be called with additional arguments `"MathJax"`, `"Tth"` and/or `"MathML"`. If these are given additional variants of the HTML conversion are called, see `GAPDoc2HTML` (5.3.7) for details.

It is possible to use GAPDoc with other languages than English, see `SetGapDocLanguage` (5.3.13) for more details.

5.2 Parsing XML Documents

Arbitrary well-formed XML documents can be parsed and browsed by the following functions. A proper validation can be done with an external program, see `XMLValidate` (5.2.11) below.

5.2.1 ParseTreeXMLString

▷ `ParseTreeXMLString(str[, srcinfo][, entitydict])` (function)
▷ `ParseTreeXMLFile(fname[, entitydict])` (function)

Returns: a record which is root of a tree structure

The first function parses an XML-document stored in string *str* and returns the document in form of a tree.

The optional argument *srcinfo* must have the same format as in `OriginalPositionDocument` (4.2.2). If it is given then error messages refer to the original source of the text with the problem.

With the optional argument *entitydict* named entities can be given to the parser, for example entities which are defined in the `.dtd`-file (which is not read by this parser). The standard XML-entities do not need to be provided, and for GAPDoc documents the entity definitions from `gapdoc.dtd` are automatically provided. Entities in the document's `<!DOCTYPE` declaration are parsed and also need not to be provided here. The argument *entitydict* must be a record where each component name is an entity name (without the surrounding & and ;) to which is assigned its substitution string.

The second function is just a shortcut for `ParseTreeXMLString( StringFile(fname), ... )`, see `StringFile` (6.3.5).

After these functions return the list of named entities which were known during the parsing can be found in the record `ENTITYDICT`.

A node in the result tree corresponds to an XML element, or to some parsed character data. In the first case it looks as follows:

Example Node
```
rec( name := "Book",
     attributes := rec( Name := "EDIM" ),
     content := [ ... list of nodes for content ...],
     start := 312,
     stop := 15610,
     next := 15611     )
```

This means that `str{[312..15610]}` looks like `<Book Name="EDIM"> ... content ... </Book>`.

The leaves of the tree encode parsed character data as in the following example:

Example Node
```
rec( name := "PCDATA",
     content := "text without markup "     )
```

This function checks whether the XML document is *well formed*, see 2.1.14 for an explanation. If an error in the XML structure is found, a break loop is entered and the text around the position where the problem starts is shown. With `Show();` one can browse the original input in the `Pager` (**Reference: Pager**), starting with the line where the error occurred. All entities are resolved when they are either entities defined in the GAPDoc package (in particular the standard XML entities) or if their definition is included in the `<!DOCTYPE ..>` tag of the document.

Note that `ParseTreeXMLString` does not parse and interpret the corresponding document type definition (the `.dtd`-file given in the `<!DOCTYPE ..>` tag). Hence it also does not check the *validity* of the document (i.e., it is no *validating XML parser*).

If you are using this function to parse a GAPDoc document you can use `CheckAndCleanGapDocTree` (5.2.8) for some validation and additional checking of the document structure.

5.2.2 StringXMLElement

▷ `StringXMLElement(tree)` (function)

Returns: a list `[string, positions]`

The argument *tree* must have a format of a node in the parse tree of an XML document as returned by `ParseTreeXMLString` (5.2.1) (including the root node representing the full document). This function computes a pair `[string, positions]` where `string` contains XML code which is equivalent to the code which was parsed to get *tree*. And `positions` is a list of lists of four numbers `[eltb, elte, contb, conte]`. There is one such list for each XML element occuring in `string`, where `eltb` and `elte` are the begin and end position of this element in `string` and where `contb` and `conte` are begin and end position of the content of this element, or both are `0` if there is no content.

Note that parsing XML code is an irreversible task, we can only expect to get equivalent XML code from this function. But parsing the resulting `string` again and applying `StringXMLElement` again gives the same result. See the function `EntitySubstitution` (5.2.3) for back-substitutions of entities in the result.

5.2.3 EntitySubstitution

▷ `EntitySubstitution(xmlstring, entities)` (function)

Returns: a string

The argument *xmlstring* must be a string containing XML code or a pair `[string, positions]` as returned by `StringXMLElement` (5.2.2). The argument *entities* specifies entity names (without the surrounding *&* and `;`) and their substitution strings, either a list of pairs of strings or as a record with the names as components and the substitutions as values.

This function tries to substitute non-intersecting parts of `string` by the given entities. If the `positions` information is given then only parts of the document which allow a valid substitution by an entity are considered. Otherwise a simple text substitution without further check is done.

Note that in general the entity resolution in XML documents is a complicated and non-reversible task. But nevertheless this utility may be useful in not too complicated situations.

5.2.4 DisplayXMLStructure

▷ `DisplayXMLStructure(tree)` (function)

This utility displays the tree structure of an XML document as it is returned by `ParseTreeXMLString` (5.2.1) (without the `PCDATA` leaves).

Since this is usually quite long the result is shown using the `Pager` (**Reference: Pager**).

5.2.5 ApplyToNodesParseTree

▷ `ApplyToNodesParseTree(tree, fun)` (function)
▷ `AddRootParseTree(tree)` (function)
▷ `RemoveRootParseTree(tree)` (function)

The function `ApplyToNodesParseTree` applies a function *fun* to all nodes of the parse tree *tree* of an XML document returned by `ParseTreeXMLString` (5.2.1).

The function `AddRootParseTree` is an application of this. It adds to all nodes a component `.root` to which the top node tree *tree* is assigned. These components can be removed afterwards with `RemoveRootParseTree`.

Here are two more utilities which use `ApplyToNodesParseTree` (5.2.5).

5.2.6 GetTextXMLTree

▷ `GetTextXMLTree(tree)` (function)

Returns: a string

The argument *tree* must be a node of a parse tree of some XML document, see `ParseTreeXMLFile` (5.2.1). This function collects the content of this and all included elements recursively into a string.

5.2.7 XMLElements

▷ `XMLElements(tree, eltnames)` (function)

Returns: a list of nodes

The argument *tree* must be a node of a parse tree of some XML document, see `ParseTreeXMLFile` (5.2.1). This function returns a list of all subnodes of *tree* (possibly including *tree*) of elements with name given in the list of strings *eltnames*. Use `"PCDATA"` as name for leave nodes which contain the actual text of the document. As an abbreviation *eltnames* can also be a string which is then put in a one element list.

And here are utilities for processing GAPDoc XML documents.

5.2.8 CheckAndCleanGapDocTree

▷ `CheckAndCleanGapDocTree(tree)` (function)

Returns: nothing

The argument *tree* of this function is a parse tree from `ParseTreeXMLString` (5.2.1) of some GAPDoc document. This function does an (incomplete) validity check of the document according to the document type declaration in `gapdoc.dtd`. It also does some additional checks which cannot be described in the DTD (like checking whether chapters and sections have a heading). For elements with element content the whitespace between these elements is removed.

In case of an error the break loop is entered and the position of the error in the original XML document is printed. With `Show();` one can browse the original input in the `Pager` (**Reference: Pager**).

5.2.9 AddParagraphNumbersGapDocTree

▷ `AddParagraphNumbersGapDocTree(tree)` (function)

Returns: nothing

The argument *tree* must be an XML tree returned by `ParseTreeXMLString` (5.2.1) applied to a GAPDoc document. This function adds to each node of the tree a component `.count` which is of form `[Chapter[, Section[, Subsection, Paragraph] ] ]`. Here the first three numbers should be the same as produced by the LaTeX version of the document. Text before the first chapter is counted as chapter `0` and similarly for sections and subsections. Some elements are always considered to start a new paragraph.

5.2.10 InfoXMLParser

▷ `InfoXMLParser` (info class)

The default level of this info class is 1. Functions like `ParseTreeXMLString` (5.2.1) are then printing some information, in particular in case of errors. You can suppress it by setting the level of `InfoXMLParser` to 0. With level 2 there may be some more information for debugging purposes.

5.2.11 XMLValidate

▷ `XMLValidate(`*doc, dtdpath*`)` (function)

Returns: `fail` or a record

The argument *doc* must be a string which is an XML document, and *dtdpath* is a path containing the corresponding DTD file.

The function returns `fail` if the program `xmllint` cannot be called.

Otherwise the document is validated via the external program `xmllint` via the function `IO_PipeThroughWithError` (**IO: IO_PipeThroughWithError**), and its resulting record is returned.

5.2.12 ValidateGAPDoc

▷ `ValidateGAPDoc(`*doc*`)` (function)

Returns: `fail`, `true` or a record

The argument *doc* must be a string which is a GAPDoc XML document or a pair of a string and list as returned by `ComposedDocument` (4.2.1) with argument *info* set to `true`.

The function returns `fail` in case of a problem.

Otherwise the document is validated using `XMLValidate` (5.2.11). If the validation was successful this function returns `true`. In the case of validation errors some information is printed and the result of `XMLValidate` (5.2.11) is returned.

Example

```
gap> fn := Filename(DirectoriesPackageLibrary("gapdoc", ""),
>                                            "3k+1/3k+1.xml");;
gap> doc := ComposedDocument("GAPDoc", "", fn, [], true);;
gap> doc[1][220] := 't';;
gap> check := ValidateGAPDoc(doc);;
ValidateGAPDoc found problems:
Line 11:  parser error : Opening and ending tag mismatch
   source position: /opt/gap/pkg/GAPDoc-1.6.4/3k+1/3k+1.xml, line 11
```

5.3 The Converters

Here are more details about the conversion programs for GAPDoc XML documents.

5.3.1 GAPDoc2LaTeX

▷ `GAPDoc2LaTeX(`*tree*`)` (function)

Returns: LaTeX document as string

▷ `SetGapDocLaTeXOptions([...])` (function)

Returns: Nothing

The argument *tree* for this function is a tree describing a GAPDoc XML document as returned by `ParseTreeXMLString` (5.2.1) (probably also checked with `CheckAndCleanGapDocTree` (5.2.8)). The output is a string containing a version of the document which can be written to a file and processed with LaTeX or pdfLaTeX (and probably BibTeX and `makeindex`).

The output uses the `report` document class and needs the following LaTeX packages: `amssymb`, `inputenc`, `makeidx`, `color`, `fancyvrb`, `psnfss`, `pslatex`, `enumitem` and `hyperref`. These are for example provided by the texlive distribution of TeX (which in turn is used for most TeX packages of current Linux distributions); see `https://www.tug.org/texlive/`.

In particular, the resulting `pdf`-output (and `dvi`-output) contains (internal and external) hyperlinks which can be very useful for onscreen browsing of the document.

The LaTeX processing also produces a file with extension `.pnr` which is GAP readable and contains the page numbers for all (sub)sections of the document. This can be used by GAP's online help; see `AddPageNumbersToSix` (5.3.4). Non-ASCII characters in the GAPDoc document are translated to LaTeX input in ASCII-encoding with the help of `Encode` (6.2.2) and the option `"LaTeX"`. See the documentation of `Encode` (6.2.2) for how to proceed if you have a character which is not handled (yet).

This function works by running recursively through the document tree and calling a handler function for each GAPDoc XML element. Many of these handler functions (usually in `GAPDoc2LaTeXProcs.<ElementName>`) are not difficult to understand (the greatest complications are some commands for index entries, labels or the output of page number information). So it should be easy to adjust layout details to your own taste by slight modifications of the program.

Former versions of GAPDoc supported some XML processing instructions to add some extra lines to the preamble of the LaTeX document. Its use is now deprecated, use the much more flexible `SetGapDocLaTeXOptions` instead: The default layout of the resulting documents can be changed with `SetGapDocLaTeXOptions`. This changes parts of the header of the LaTeX file produced by GAPDoc. You can see the header with some placeholders by `Page(GAPDoc2LaTeXProcs.Head);`. The placeholders are filled with components from the record `GAPDoc2LaTeXProcs.DefaultOptions`. The arguments of `SetGapDocLaTeXOptions` can be records with the same structure (or parts of it) with different values. As abbreviations there are also three strings supported as arguments. These are `"nocolor"` for switching all colors to black; then `"nopslatex"` to use standard LaTeX fonts instead of postscript fonts; and finally `"utf8"` to choose UTF-8 as input encoding for the LaTeX document.

5.3.2 GAPDoc2Text

▷ `GAPDoc2Text(tree[, bibpath][, width])` (function)

Returns: record containing text files as strings and other information

The argument *tree* for this function is a tree describing a GAPDoc XML document as returned by `ParseTreeXMLString` (5.2.1) (probably also checked with `CheckAndCleanGapDocTree` (5.2.8)). This function produces a text version of the document which can be used with GAP's online help (with the `"screen"` viewer, see `SetHelpViewer` (**Reference: SetHelpViewer**)). It includes title page, bibliography and index. The bibliography is made from BibXMLext or BibTeX databases, see 7. Their location must be given with the argument *bibpath* (as string or directory object).

The output is a record with one component for each chapter (with names `"0"`, `"1"`, ..., `"Bib"` and `"Ind"`). Each such component is again a record with the following components:

`text`
: the text of the whole chapter as a string

`ssnr`
: list of subsection numbers in this chapter (like `[3, 2, 1]` for chapter 3, section 2, subsection 1)

`linenr`
: corresponding list of line numbers where the subsections start

`len`
: number of lines of this chapter

The result can be written into files with the command `GAPDoc2TextPrintTextFiles` (5.3.3).

As a side effect this function also produces the `manual.six` information which is used for searching in GAP's online help. This is stored in *`tree`*`.six` and can be printed into a `manual.six` file with `PrintSixFile` (5.3.5) (preferably after producing a LaTeX version of the document as well and adding the page number information to *`tree`*`.six`, see `GAPDoc2LaTeX` (5.3.1) and `AddPageNumbersToSix` (5.3.4)).

The text produced by this function contains some markup via ANSI escape sequences. The sequences used here are usually ignored by terminals. But the GAP help system will substitute them by interpreted color and attribute sequences (see `TextAttr` (6.1.2)) before displaying them. There is a default markup used for this but it can also be configured by the user, see `SetGAPDocTextTheme` (5.3.6). Furthermore, the text produced is in UTF-8 encoding. The encoding is also translated on the fly, if `GAPInfo.TermEncoding` is set to some encoding supported by `Encode` (6.2.2), e.g., `"ISO-8859-1"` or `"latin1"`.

With the optional argument *`width`* a different length of the output text lines can be chosen. The default is 76 and all lines in the resulting text start with two spaces. This looks good on a terminal with a standard width of 80 characters and you probably don't want to use this argument.

5.3.3 GAPDoc2TextPrintTextFiles

▷ `GAPDoc2TextPrintTextFiles(`*`t[, path]`*`)` (function)

Returns: nothing

The first argument must be a result returned by `GAPDoc2Text` (5.3.2). The second argument is a path for the files to write, it can be given as string or directory object. The text of each chapter is written into a separate file with name `chap0.txt`, `chap1.txt`, ..., `chapBib.txt`, and `chapInd.txt`.

If you want to make your document accessible via the GAP online help you must put at least these files for the text version into a directory, together with the file `manual.six`, see `PrintSixFile` (5.3.5). Then specify the path to the `manual.six` file in the packages `PackageInfo.g` file, see (**Reference: The PackageInfo.g File**).

Optionally you can add the `dvi`- and `pdf`-versions of the document which are produced with `GAPDoc2LaTeX` (5.3.1) to this directory. The files must have the names `manual.dvi` and `manual.pdf`, respectively. Also you can add the files of the HTML version produced with `GAPDoc2HTML` (5.3.7) to this directory, see `GAPDoc2HTMLPrintHTMLFiles` (5.3.8). The handler functions in GAP for this help format detect automatically which of the optional formats of a book are actually available.

5.3.4 AddPageNumbersToSix

▷ `AddPageNumbersToSix(`*`tree, pnrfile`*`)` (function)

Returns: nothing

Here *tree* must be the XML tree of a GAPDoc document, returned by ParseTreeXMLString (5.2.1). Running latex on the result of GAPDoc2LaTeX(*tree*) produces a file *pnrfile* (with extension .pnr). The command GAPDoc2Text(*tree*) creates a component *tree*.six which contains all information about the document for the GAP online help, except the page numbers in the .dvi, .ps, .pdf versions of the document. This command adds the missing page number information to *tree*.six.

5.3.5 PrintSixFile

▷ PrintSixFile(*tree, bookname, fname*) (function)

Returns: nothing

This function prints the .six file *fname* for a GAPDoc document stored in *tree* with name *bookname*. Such a file contains all information about the book which is needed by the GAP online help. This information must first be created by calls of GAPDoc2Text (5.3.2) and AddPageNumbersToSix (5.3.4).

5.3.6 SetGAPDocTextTheme

▷ SetGAPDocTextTheme(*[optrec1[, optrec2], ...]*) (function)

Returns: nothing

This utility function is for readers of the screen version of GAP manuals which are generated by the GAPDoc package. It allows to configure the color and attribute layout of the displayed text. There is a default which can be reset by calling this function without argument.

As an abbreviation the arguments *optrec1* and so on can be strings for the known name of a theme. Information about valid names is shown with SetGAPDocTextTheme("");.

Otherwise, *optrec1* and so on must be a record. Its entries overwrite the corresponding entries in the default and in previous arguments. To construct valid markup you can use TextAttr (6.1.2). Entries must be either pairs of strings, which are put before and after the corresponding text, or as an abbreviation it can be a single string. In the latter case, the second string is implied; if the string contains an escape sequence the second string is TextAttr.reset, otherwise the given string is used. The following components are recognized:

flush
: "both" for left-right justified paragraphs, and "left" for ragged right ones

Heading
: chapter and (sub-)section headings

Func
: function, operation, ... names

Arg
: argument names in descriptions

Example
: example code

Package
: package names

`Returns`
: Returns-line in descriptions

`URL`
: URLs

`Mark`
: Marks in description lists

`K`
: GAP keywords

`C`
: code or text to type

`F`
: file names

`B`
: buttons

`M`
: simplified math elements

`Math`
: normal math elements

`Display`
: displayed math elements

`Emph`
: emphasized text

`Q`
: quoted text

`Ref`
: reference text

`Prompt`
: GAP prompt in examples

`BrkPrompt`
: GAP break prompt in examples

`GAPInput`
: GAP input in examples

`reset`
: reset to default, don't change this

`BibAuthor`
: author names in bibliography

`BibTitle`
: titles in bibliography

`BibJournal`
: journal names in bibliography

`BibVolume`
: volume number in bibliography

`BibLabel`
: labels for bibliography entries

`BibReset`
: reset for bibliography, don't change

`ListBullet`
: bullet for simple lists (2 visible characters long)

`EnumMarks`
: one visible character before and after the number in enumerated lists

`DefLineMarker`
: marker before function and variable definitions (2 visible characters long)

`FillString`
: for filling in definitions and example separator lines

Example
```
gap> # use no colors for GAP examples and
gap> # change display of headings to bold green
gap> SetGAPDocTextTheme("noColorPrompt",
>            rec(Heading:=Concatenation(TextAttr.bold, TextAttr.2)));
```

5.3.7 GAPDoc2HTML

▷ `GAPDoc2HTML(`*`tree[, bibpath[, gaproot]][, mtrans]`*`)` (function)

Returns: record containing HTML files as strings and other information

The argument *`tree`* for this function is a tree describing a GAPDoc XML document as returned by `ParseTreeXMLString` (5.2.1) (probably also checked with `CheckAndCleanGapDocTree` (5.2.8)). Without an *`mtrans`* argument this function produces an HTML version of the document which can be read with any Web-browser and also be used with GAP's online help (see `SetHelpViewer` (**Reference: SetHelpViewer**)). It includes title page, bibliography, and index. The bibliography is made from BibTeX databases. Their location must be given with the argument *`bibpath`* (as string or directory object, if not given the current directory is used). If the third argument *`gaproot`* is given and is a string then this string is interpreted as relative path to GAP's main root directory. Reference-URLs to external HTML-books which begin with the GAP root path are then rewritten to start with the given relative path. This makes the HTML-documentation portable provided a package is installed in some standard location below the GAP root.

The output is a record with one component for each chapter (with names `"0"`, `"1"`, ..., `"Bib"`, and `"Ind"`). Each such component is again a record with the following components:

`text`
: the text of an HTML file containing the whole chapter (as a string)

`ssnr`
: list of subsection numbers in this chapter (like `[3, 2, 1]` for chapter 3, section 2, subsection 1)

Standard output format without `mtrans` argument

The HTML code produced with this converter conforms to the W3C specification "XHTML 1.0 strict", see `https://www.w3.org/TR/xhtml1`. First, this means that the HTML files are valid XML files. Secondly, the extension "strict" says in particular that the code doesn't contain any explicit font or color information.

Mathematical formulae are handled as in the text converter `GAPDoc2Text` (5.3.2). We don't want to assume that the browser can use symbol fonts. Some GAP users like to browse the online help with `lynx`, see `SetHelpViewer` (**Reference: SetHelpViewer**), which runs inside the same terminal windows as GAP.

To view the generated files in graphical browsers, stylesheet files with layout configuration should be copied into the directory with the generated HTML files, see 5.3.9.

Output format with `mtrans` argument

Currently, there are three variants of this converter available which handle mathematical formulae differently. They are accessed via the optional last *mtrans* argument.

If *mtrans* is set to `"MathJax"` the formulae are essentially translated as for LaTeX documents (there is no processing of `<M>` elements as decribed in 3.8.2). Inline formulae are delimited by `\(` and `\)` and displayed formulae by `\[` and `\]`. With MathJax webpages can contain nicely formatted scalable and searchable formulae. The resulting files link by default to https://www.jsdelivr.com/ to get the MathJax script and fonts. This means that they can only be used on computers with internet access. An alternative URL can be set by overwriting `GAPDoc2HTMLProcs.MathJaxURL` before building the HTML version of a manual. This way a local installation of MathJax could be used. See https://www.mathjax.org/ for more details.

The following possibilities for *mtrans* are still supported, but since the MathJax approach seems much better, their use is deprecated.

If the argument *mtrans* is set to `"Tth"` it is assumed that you have installed the LaTeX to HTML translation program `tth`. This is used to translate the contents of the `M`, `Math` and `Display` elements into HTML code. Note that the resulting code is not compliant with any standard. Formally it is "XHTML 1.0 Transitional", it contains explicit font specifications and the characters of mathematical symbols are included via their position in a "Symbol" font. Some graphical browsers can be configured to display this in a useful manner, check the Tth homepage for more details.

This function works by running recursively through the document tree and calling a handler function for each GAPDoc XML element. Many of these handler functions (usually in `GAPDoc2TextProcs.<ElementName>`) are not difficult to understand (the greatest complications are some commands for index entries, labels or the output of page number information). So it should be easy to adjust certain details to your own taste by slight modifications of the program.

The result of this converter can be written to files with the command `GAPDoc2HTMLPrintHTMLFiles` (5.3.8).

There are two user preferences for reading the HTML manuals produced by GAPDoc. A user can choose among several style files which determine the appearance of the manual pages with `SetUserPreference("GAPDoc", "HTMLStyle", [...]);` where the list in the third argument are arguments for `SetGAPDocHTMLStyle` (5.3.11). The second preference is set by `SetUserPreference("GAPDoc", "UseMathJax", ...);` where the third argument is `true` or `false` (default). If this is set to `true`, the GAP help system displays the MathJax version of the HTML manuals.

5.3.8 GAPDoc2HTMLPrintHTMLFiles

▷ `GAPDoc2HTMLPrintHTMLFiles(t[, path])` (function)

Returns: nothing

The first argument must be a result returned by GAPDoc2HTML (5.3.7). The second argument is a path for the files to write, it can be given as string or directory object. The text of each chapter is written into a separate file with name chap0.html, chap1.html, ..., chapBib.html, and chapInd.html.

The MathJax versions are written to files chap0_mj.html, ..., chapInd_mj.html.

The experimental version which is produced with tth uses different names for the files, namely chap0_sym.html, and so on for files which need symbol fonts.

You should also add stylesheet files to the directory with the HTML files, see 5.3.9.

5.3.9 Stylesheet files

For graphical browsers the layout of the generated HTML manuals can be highly configured by cascading stylesheet (CSS) and javascript files. Such files are provided in the styles directory of the GAPDoc package.

We recommend that these files are copied into each manual directory (such that each of them is selfcontained). There is a utility function CopyHTMLStyleFiles (5.3.10) which does this. Of course, these files may be changed or new styles may be added. New styles may also be sent to the GAPDoc authors for possible inclusion in future versions.

The generated HTML files refer to the file manual.css which conforms to the W3C specification CSS 2.0, see https://www.w3.org/TR/REC-CSS2, and the javascript file manual.js (only in browsers which support CSS or javascript, respectively; but the HTML files are also readable without any of them). To add a style mystyle one or both of mystyle.css and mystyle.js must be provided; these can overwrite default settings and add new javascript functions. For more details see the comments in manual.js.

5.3.10 CopyHTMLStyleFiles

▷ CopyHTMLStyleFiles(*dir*) (function)

Returns: nothing

This utility function copies the *.css and *.js files from the styles directory of the GAPDoc package into the directory *dir*.

5.3.11 SetGAPDocHTMLStyle

▷ SetGAPDocHTMLStyle(*[style1[, style2], ...]*) (function)

Returns: nothing

This utility function is for readers of the HTML version of GAP manuals which are generated by the GAPDoc package. It allows to configure the display style of the manuals. This will only have an effect if you are using a browser that supports javascript. There is a default which can be reset by calling this function without argument.

The arguments *style1* and so on must be strings. You can find out about the valid strings by following the [STYLE] link on top of any manual page. (Going back to the original page, its address has a setting for GAPDocStyle which is the list of strings, separated by commas, you want to use here.)

Example

```
gap> # show/hide subsections in tables on contents only after click,
gap> # and don't use colors in GAP examples
gap> SetGAPDocHTMLStyle("toggless", "nocolorprompt");
```

5.3.12 InfoGAPDoc

▷ `InfoGAPDoc` (info class)

The default level of this info class is 1. The converter functions for GAPDoc documents are then printing some information. You can suppress this by setting the level of `InfoGAPDoc` to 0. With level 2 there may be some more information for debugging purposes.

5.3.13 SetGapDocLanguage

▷ `SetGapDocLanguage([lang])` (function)

Returns: nothing

The GAPDoc converter programs sometimes produce text which is not explicit in the document, e.g., headers like “Abstract”, “Appendix”, links to “Next Chapter”, variable types “function” and so on.

With `SetGapDocLanguage` the language for these texts can be changed. The argument *lang* must be a string. Calling without argument or with a language name for which no translations are available is the same as using the default `"english"`.

If your language *lang* is not yet available, look at the record `GAPDocTexts.english` and translate all the strings to *lang*. Then assign this record to `GAPDocTexts.(lang)` and send it to the GAPDoc authors for inclusion in future versions of GAPDoc. (Currently, there are translations for `english`, `german`, `russian` and `ukrainian`.)

Further hints: To get strings produced by LaTeX right you will probably use the `babel` package with option *lang*, see `SetGapDocLaTeXOptions` (5.3.1). If *lang* cannot be encoded in `latin1` encoding you can consider the use of `"utf8"` with `SetGapDocLaTeXOptions` (5.3.1).

5.4 Testing Manual Examples

We also provide some tools to check and adjust the examples given in `<Example>`-elements.

Former versions of GAPDoc provided functions `ManualExamples` and `TestManualExamples`. These functions are still available, but no longer documented. Their use is deprecated.

5.4.1 ExtractExamples

▷ `ExtractExamples(path, main, files, units[, withLog])` (function)

Returns: a list of lists

▷ `ExtractExamplesXMLTree(tree, units[, withLog])` (function)

Returns: a list of lists

The argument *tree* must be a parse tree of a GAPDoc document, see `ParseTreeXMLFile` (5.2.1). The function `ExtractExamplesXMLTree` returns a data structure representing the `<Example>` elements of the document. The return value can be used with `RunExamples` (5.4.2) to check and optionally update the examples of the document.

Depending on the argument *units* several examples are collected in one list. Recognized values for *units* are `"Chapter"`, `"Section"`, `"Subsection"` or `"Single"`. The latter means that each example is in a separate list. For all other value of *units* just one list with all examples is returned.

The arguments *path*, *main* and *files* of `ExtractExamples` are the same as for `ComposedDocument` (4.2.1). This function first contructs and parses the GAPDoc document and then applies `ExtractExamplesXMLTree`.

If the optional argument *withLog* is given and `true` then `<Log>` elements are handled like `<Example>` elements. This allows to put examples which can only run under certain conditions, e.g., when certain external programs are available, into `<Log>` elements. (Put example code which should also not be included by this variant into `<Listing>` elements.)

5.4.2 RunExamples

▷ `RunExamples(`*exmpls[, optrec]*`)` (function)

Returns: `true` or `false`

The argument *exmpls* must be the output of a call to `ExtractExamples` (5.4.1) or `ExtractExamplesXMLTree` (5.4.1). The optional argument *optrec* must be a record, its components can change the default behaviour of this function.

By default this function runs the GAP input of all examples and compares the actual output with the output given in the examples. If differences occur these are displayed together with information on the location of the source code of that example. Before running the examples in each unit (entry of *exmpls*) the function `START_TEST` (**Reference: START_TEST**) is called and the screen width is set to 72 characters.

This function returns `true` if no differences are found and `false` otherwise.

If the argument *optrec* is given, the following components are recognized:

`showDiffs`
: The default value is `true`, if set to something else found differences in the examples are not displayed.

`width`
: The value must be a positive integer which is used as screen width when running the examples. As mentioned above, the default is 72 which is a sensible value for the text version of the GAPDoc document used in a 80 character wide terminal.

`ignoreComments`
: The default is `false`.
If set to `true` comments in the input will be ignored (as in the default behaviour of the `Test` (**Reference: Test**) function).

`changeSources`
: If this is set to `true` then the source code of all manual examples which show differences is adjusted to the current outputs. The default is `false`.
Use this feature with care. Note that sometimes differences can indicate a bug, and in such a case it is more appropriate to fix the bug instead of changing the example output.

`compareFunction`
: The function used to compare the output shown in the example and the current output. See `Test` (**Reference: Test**) for more details.

`checkWidth`
: If this option is a positive integer `n` the function prints warnings if an example contains any line

with more than `n` characters (input and output lines are considered). By default this option is set to `false`.

Chapter 6

String and Text Utilities

6.1 Text Utilities

This section describes some utility functions for handling texts within GAP. They are used by the functions in the GAPDoc package but may be useful for other purposes as well. We start with some variables containing useful strings and go on with functions for parsing and reformatting text.

6.1.1 WHITESPACE

▷ `WHITESPACE` (global variable)
▷ `CAPITALLETTERS` (global variable)
▷ `SMALLLETTERS` (global variable)
▷ `LETTERS` (global variable)
▷ `DIGITS` (global variable)
▷ `HEXDIGITS` (global variable)
▷ `BOXCHARS` (global variable)

These variables contain sets of characters which are useful for text processing. They are defined as follows.

`WHITESPACE`
: `" \n\t\r"`

`CAPITALLETTERS`
: `"ABCDEFGHIJKLMNOPQRSTUVWXYZ"`

`SMALLLETTERS`
: `"abcdefghijklmnopqrstuvwxyz"`

`LETTERS`
: concatenation of `CAPITALLETTERS` and `SMALLLETTERS`

`DIGITS`
: `"0123456789"`

`HEXDIGITS`
: `"0123456789ABCDEFabcdef"`

`BOXCHARS`
: `Encode(Unicode(9472 + [ 0, 2, 12, 44, 16, 28, 60, 36, 20, 52, 24, 1, 3, 15, 51, 19, 35, 75, 43, 23, 59, 27, 80, 81, 84, 102, 87, 96, 108, 99, 90, 105, 93 ]), "UTF-8")`, these are in UTF-8 encoding, the `i`-th unicode character is `BOXCHARS{[3*i-2..3*i]}`.

6.1.2 TextAttr

▷ `TextAttr` (global variable)

The record `TextAttr` contains strings which can be printed to change the terminal attribute for the following characters. This only works with terminals which understand basic ANSI escape sequences. Try the following example to see if this is the case for the terminal you are using. It shows the effect of the foreground and background color attributes and of the `.bold`, `.blink`, `.normal`, `.reverse` and `.underscore` which can partly be mixed.

Example
```
extra := ["CSI", "reset", "delline", "home"];;
for t in Difference(RecNames(TextAttr), extra) do
  Print(TextAttr.(t), "TextAttr.", t, TextAttr.reset,"\n");
od;
```

The suggested defaults for colors `0..7` are black, red, green, brown, blue, magenta, cyan, white. But this may be different for your terminal configuration.

The escape sequence `.delline` deletes the content of the current line and `.home` moves the cursor to the beginning of the current line.

Example
```
for i in [1..5] do
  Print(TextAttr.home, TextAttr.delline, String(i,-6), "\c");
  Sleep(1);
od;
```

Whenever you use this in some printing routines you should make it optional. Use these attributes only when `UserPreference("UseColorsInTerminal");` returns `true`.

6.1.3 WrapTextAttribute

▷ `WrapTextAttribute(`*`str, attr`*`)` (function)

Returns: a string with markup

The argument *`str`* must be a text as GAP string, possibly with markup by escape sequences as in `TextAttr` (6.1.2). This function returns a string which is wrapped by the escape sequences *`attr`* and `TextAttr.reset`. It takes care of markup in the given string by appending *`attr`* also after each given `TextAttr.reset` in *`str`*.

Example
```
gap> str := Concatenation("XXX",TextAttr.2, "BLUB", TextAttr.reset,"YYY");
"XXX\033[32mBLUB\033[0mYYY"
gap> str2 := WrapTextAttribute(str, TextAttr.1);
"\033[31mXXX\033[32mBLUB\033[0m\033[31m\027YYY\033[0m"
gap> str3 := WrapTextAttribute(str, TextAttr.underscore);
"\033[4mXXX\033[32mBLUB\033[0m\033[4m\027YYY\033[0m"
gap> # use Print(str); and so on to see how it looks like.
```

6.1.4 FormatParagraph

▷ `FormatParagraph(str[, len][, flush][, attr][, widthfun])` (function)

Returns: the formatted paragraph as string

This function formats a text given in the string *str* as a paragraph. The optional arguments have the following meaning:

len the length of the lines of the formatted text, default is `78` (counted without a visible length of the strings specified in the *attr* argument)

flush
can be `"left"`, `"right"`, `"center"` or `"both"`, telling that lines should be flushed left, flushed right, centered or left-right justified, respectively, default is `"both"`

attr
is a list of two strings; the first is prepended and the second appended to each line of the result (can for example be used for indenting, `["  ", ""]`, or some markup, `[TextAttr.bold, TextAttr.reset]`, default is `["", ""]`)

widthfun
must be a function which returns the display width of text in *str*. The default is `Length` assuming that each byte corresponds to a character of width one. If *str* is given in `UTF-8` encoding one can use `WidthUTF8String` (6.2.3) here.

This function tries to handle markup with the escape sequences explained in `TextAttr` (6.1.2) correctly.

Example
```
gap> str := "One two three four five six seven eight nine ten eleven.";;
gap> Print(FormatParagraph(str, 25, "left", ["/* ", " */"]));
/* One two three four five */
/* six seven eight nine ten */
/* eleven. */
```

6.1.5 SubstitutionSublist

▷ `SubstitutionSublist(list, sublist, new[, flag])` (function)

Returns: the changed list

This function looks for (non-overlapping) occurrences of a sublist *sublist* in a list *list* (compare `PositionSublist` (**Reference: PositionSublist**)) and returns a list where these are substituted with the list *new*.

The optional argument *flag* can either be `"all"` (this is the default if not given) or `"one"`. In the second case only the first occurrence of *sublist* is substituted.

If *sublist* does not occur in *list* then *list* itself is returned (and not a `ShallowCopy(list)`).

Example
```
gap> SubstitutionSublist("xababx", "ab", "a");
"xaax"
```

6.1.6 StripBeginEnd

▷ `StripBeginEnd(list, strip)` (function)

Returns: changed string

Here *list* and *strip* must be lists. This function returns the sublist of list which does not contain the leading and trailing entries which are entries of *strip*. If the result is equal to *list* then *list* itself is returned.

```
gap> StripBeginEnd(" ,a, b,c,   ", ", ");
"a, b,c"
```

6.1.7 StripEscapeSequences

▷ `StripEscapeSequences(str)` (function)

Returns: string without escape sequences

This function returns the string one gets from the string *str* by removing all escape sequences which are explained in `TextAttr` (6.1.2). If *str* does not contain such a sequence then *str* itself is returned.

6.1.8 RepeatedString

▷ `RepeatedString(c, len)` (function)
▷ `RepeatedUTF8String(c, len)` (function)

Here *c* must be either a character or a string and *len* is a non-negative number. Then `RepeatedString` returns a string of length *len* consisting of copies of *c*.

In the variant `RepeatedUTF8String` the argument *c* is considered as string in UTF-8 encoding, and it can also be specified as unicode string or character, see `Unicode` (6.2.1). The result is a string in UTF-8 encoding which has visible width *len* as explained in `WidthUTF8String` (6.2.3).

```
gap> RepeatedString('=',51);
"==================================================="
gap> RepeatedString("*=",51);
"*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*"
gap> s := "bäh";;
gap> enc := GAPInfo.TermEncoding;;
gap> if enc <> "UTF-8" then s := Encode(Unicode(s, enc), "UTF-8"); fi;
gap> l := RepeatedUTF8String(s, 8);;
gap> u := Unicode(l, "UTF-8");;
gap> Print(Encode(u, enc), "\n");
bähbähbä
```

6.1.9 NumberDigits

▷ `NumberDigits(str, base)` (function)

Returns: integer

▷ `DigitsNumber(n, base)` (function)

Returns: string

The argument *str* of `NumberDigits` must be a string consisting only of an optional leading `'-'` and characters in `0123456789abcdefABCDEF`, describing an integer in base *base* with $2 \leq base \leq 16$. This function returns the corresponding integer.

The function `DigitsNumber` does the reverse.

Example
```
gap> NumberDigits("1A3F",16);
6719
gap> DigitsNumber(6719, 16);
"1A3F"
```

6.1.10 LabelInt

▷ `LabelInt(n, type, pre, post)` (function)

Returns: string

The argument *n* must be an integer in the range from 1 to 5000, while *pre* and *post* must be strings.

The argument *type* can be one of `"Decimal"`, `"Roman"`, `"roman"`, `"Alpha"`, `"alpha"`.

The function returns a string that starts with *pre*, followed by a decimal, respectively roman number or alphanumerical number literal (capital, respectively small letters), followed by *post*.

Example
```
gap> List([1,2,3,4,5,691], i-> LabelInt(i,"Decimal","","."));
[ "1.", "2.", "3.", "4.", "5.", "691." ]
gap> List([1,2,3,4,5,691], i-> LabelInt(i,"alpha","(",")"));
[ "(a)", "(b)", "(c)", "(d)", "(e)", "(zo)" ]
gap> List([1,2,3,4,5,691], i-> LabelInt(i,"alpha","(",")"));
[ "(a)", "(b)", "(c)", "(d)", "(e)", "(zo)" ]
gap> List([1,2,3,4,5,691], i-> LabelInt(i,"Alpha","",".)"));
[ "A.)", "B.)", "C.)", "D.)", "E.)", "ZO.)" ]
gap> List([1,2,3,4,5,691], i-> LabelInt(i,"roman","","."));
[ "i.", "ii.", "iii.", "iv.", "v.", "dcxci." ]
gap> List([1,2,3,4,5,691], i-> LabelInt(i,"Roman","",""));
[ "I", "II", "III", "IV", "V", "DCXCI" ]
```

6.1.11 PositionMatchingDelimiter

▷ `PositionMatchingDelimiter(str, delim, pos)` (function)

Returns: position as integer or `fail`

Here *str* must be a string and *delim* a string with two different characters. This function searches the smallest position `r` of the character *delim*`[2]` in *str* such that the number of occurrences of *delim*`[2]` in *str* between positions *pos*`+1` and `r` is by one greater than the corresponding number of occurrences of *delim*`[1]`.

If such an `r` exists, it is returned. Otherwise `fail` is returned.

Example
```
gap> PositionMatchingDelimiter("{}x{ab{c}d}", "{}", 0);
fail
gap> PositionMatchingDelimiter("{}x{ab{c}d}", "{}", 1);
2
gap> PositionMatchingDelimiter("{}x{ab{c}d}", "{}", 6);
11
```

6.1.12 WordsString

▷ `WordsString(str)` (function)

Returns: list of strings containing the words

This returns the list of words of a text stored in the string *str*. All non-letters are considered as word boundaries and are removed.

```
gap> WordsString("one_two \n    three!?");
[ "one", "two", "three" ]
```

6.1.13 Base64String

▷ `Base64String(str)` (function)

▷ `StringBase64(bstr)` (function)

Returns: a string

The first function translates arbitrary binary data given as a GAP string into a *base 64* encoded string. This encoded string contains only printable ASCII characters and is used in various data transfer protocols (`MIME` encoded emails, weak password encryption, ...). We use the specification in RFCÂ 2045.

The second function has the reverse functionality. Here we also accept the characters `-_` instead of `+/` as last two characters. Whitespace is ignored.

```
gap> b := Base64String("This is a secret!");
"VGhpcyBpcyBhIHNlY3JldCEA="
gap> StringBase64(b);
"This is a secret!"
```

6.2 Unicode Strings

The GAPDoc package provides some tools to deal with unicode characters and strings. These can be used for recoding text strings between various encodings.

6.2.1 Unicode Strings and Characters

▷ `Unicode(list[, encoding])` (operation)

▷ `UChar(num)` (operation)

▷ `IsUnicodeString` (filter)

▷ `IsUnicodeCharacter` (filter)

▷ `IntListUnicodeString(ustr)` (function)

Unicode characters are described by their *codepoint*, an integer in the range from 0 to $2^{21}-1$. For details about unicode, see `https://www.unicode.org`.

The function `UChar` wraps an integer *num* into a GAP object lying in the filter `IsUnicodeCharacter`. Use `Int` to get the codepoint back. The argument *num* can also be a GAP character which is then translated to an integer via `IntChar` (**Reference: IntChar**).

`Unicode` produces a GAP object in the filter `IsUnicodeString`. This is a wrapped list of integers for the unicode characters in the string. The function `IntListUnicodeString` gives access to this list of integers. Basic list functionality is available for `IsUnicodeString` elements. The entries are in `IsUnicodeCharacter`. The argument *list* for `Unicode` is either a list of integers or a GAP string. In the latter case an *encoding* can be specified as string, its default is `"UTF-8"`.

Currently supported encodings can be found in `UNICODE_RECODE.NormalizedEncodings` (ASCII, ISO-8859-X, UTF-8 and aliases). The encoding `"XML"` means an ASCII encoding in which non-ASCII characters are specified by XML character entities. The encoding `"URL"` is for URL-encoded (also called percent-encoded strings, as specified in RFC 3986 (see here). The listed encodings `"LaTeX"` and aliases cannot be used with `Unicode`. See the operation `Encode` (6.2.2) for mapping a unicode string to a GAP string.

Example
```
gap> ustr := Unicode("a and \366", "latin1");
Unicode("a and ö")
gap> ustr = Unicode("a and &#246;", "XML");
true
gap> IntListUnicodeString(ustr);
[ 97, 32, 97, 110, 100, 32, 246 ]
gap> ustr[7];
'ö'
```

6.2.2 Encode

▷ `Encode(`*ustr[, encoding]*`)` (operation)

Returns: a GAP string

▷ `SimplifiedUnicodeString(`*ustr[, encoding][, "single"]*`)` (function)

Returns: a unicode string

▷ `LowercaseUnicodeString(`*ustr*`)` (function)

Returns: a unicode string

▷ `UppercaseUnicodeString(`*ustr*`)` (function)

Returns: a unicode string

▷ `LaTeXUnicodeTable` (global variable)

▷ `SimplifiedUnicodeTable` (global variable)

▷ `LowercaseUnicodeTable` (global variable)

The operation `Encode` translates a unicode string *ustr* into a GAP string in some specified *encoding*. The default encoding is `"UTF-8"`.

Supported encodings can be found in `UNICODE_RECODE.NormalizedEncodings`. Except for some cases mentioned below characters which are not available in the target encoding are substituted by '?' characters.

If the *encoding* is `"URL"` (see `Unicode` (6.2.1)) then an optional argument *encreserved* can be given, it must be a list of reserved characters which should be percent encoded; the default is to encode only the `%` character.

The encoding `"LaTeX"` substitutes non-ASCII characters and LaTeX special characters by LaTeX code as given in an ordered list `LaTeXUnicodeTable` of pairs [codepoint, string]. If you have a unicode character for which no substitution is contained in that list, you will get a warning and the translation is `Unicode(nr)`. In this case find a substitution and add a corresponding [codepoint, string] pair to `LaTeXUnicodeTable` using `AddSet` (**Reference: AddSet**). Also, please, tell the GAPDoc authors about your addition, such that we can extend the list `LaTeXUnicodeTable`. (Most of the initial entries were generated from lists in the TeX projects encTeX and `ucs`.) There are some variants of this encoding:

`"LaTeXleavemarkup"` does the same translations for non-ASCII characters but leaves the LaTeX special characters (e.g., any LaTeX commands) as they are.

`"LaTeXUTF8"` does not give a warning about unicode characters without explicit translation, instead it translates the character to its `UTF-8` encoding. Make sure to setup your LaTeX document such that all these characters are understood.

`"LaTeXUTF8leavemarkup"` is a combination of the last two variants.

Note that the `"LaTeX"` encoding can only be used with `Encode` but not for the opposite translation with `Unicode` (6.2.1) (which would need far too complicated heuristics).

The function `SimplifiedUnicodeString` can be used to substitute many non-ASCII characters by related ASCII characters or strings (e.g., by a corresponding character without accents). The argument *ustr* and the result are unicode strings, if *encoding* is `"ASCII"` then all non-ASCII characters are translated, otherwise only the non-latin1 characters. If the string `"single"` in an argument then only substitutions are considered which don't make the result string longer. The translations are stored in a sorted list `SimplifiedUnicodeTable`. Its entries are of the form `[codepoint, trans1, trans2, ...]`. Here `trans1` and so on is either an integer for the codepoint of a substitution character or it is a list of codepoint integers. If you are missing characters in this list and know a sensible ASCII approximation, then add an entry (with `AddSet` (**Reference: AddSet**)) and tell the GAPDoc authors about it. (The initial content of `SimplifiedUnicodeTable` was mainly generated from the "`transtab`" tables by Markus Kuhn.)

The function `LowercaseUnicodeString` gets and returns a unicode string and translates each uppercase character to its corresponding lowercase version. This function uses a list `LowercaseUnicodeTable` of pairs of codepoint integers. This list was generated using the file `UnicodeData.txt` from the unicode definition (field 14 in each row).

The function `UppercaseUnicodeString` does the similar translation to uppercase characters.

Example
```
gap> ustr := Unicode("a and &#246;", "XML");
Unicode("a and ö")
gap> SimplifiedUnicodeString(ustr, "ASCII");
Unicode("a and oe")
gap> SimplifiedUnicodeString(ustr, "ASCII", "single");
Unicode("a and o")
gap> ustr2 := UppercaseUnicodeString(ustr);;
gap> Print(Encode(ustr2, GAPInfo.TermEncoding), "\n");
A AND Ö
```

6.2.3 Lengths of UTF-8 strings

▷ `WidthUTF8String(`*str*`)` (function)

▷ `NrCharsUTF8String(`*str*`)` (function)

Returns: an integer

Let *str* be a GAP string with text in UTF-8 encoding. There are three "lengths" of such a string which must be distinguished. The operation `Length` (**Reference: Length**) returns the number of bytes and so the memory occupied by *str*. The function `NrCharsUTF8String` returns the number of unicode characters in *str*, that is the length of `Unicode(`*str*`)`.

In many applications the function `WidthUTF8String` is more interesting, it returns the number of columns needed by the string if printed to a terminal. This takes into account that some unicode characters are combining characters and that there are wide characters which need two columns (e.g., for Chinese or Japanese). (To be precise: This implementation assumes that there are no control characters in *str* and uses the character width returned by the `wcwidth` function in the GNU C-library called with UTF-8 locale.)

Example
```
gap> # A, German umlaut u, B, zero width space, C, newline
gap> str := Encode( Unicode( "A&#xFC;B&#x200B;C\n", "XML" ) );;
gap> Print(str);
AüBC
gap> # umlaut u needs two bytes and the zero width space three
gap> Length(str);
9
gap> NrCharsUTF8String(str);
6
gap> # zero width space and newline don't contribute to width
gap> WidthUTF8String(str);
4
```

6.2.4 InitialSubstringUTF8String

▷ `InitialSubstringUTF8String(`*str, maxwidth[, suf]*`)` (function)

Returns: UTF-8 encoded string

The arguments *str* and *suf* each must be a GAP string with text in UTF-8 encoding or a unicode string. The argument *suf* is optional and its default value is the empty string. If the visible width of *str* is at most *maxwidth* then *str* is returned as UTF-8 encoded GAP string. Otherwise, *suf* is appended to the maximal initial substring of *str* such that the total visible width of the result is at most *maxwidth*.

Example
```
gap> # A, German umlaut u, B, zero width space, C, newline
gap> str := Encode( Unicode( "A&#xFC;B&#x200B;C\n", "XML" ) );;
gap> ini := InitialSubstringUTF8String(str, 3);;
gap> WidthUTF8String(ini);
3
gap> IntListUnicodeString(Unicode(ini));
[ 65, 252, 66, 8203 ]
gap> l := Unicode([ 23380, 22827, 23376 ] );; # three chars of width 2
gap> s := InitialSubstringUTF8String(l, 4, "*");;
gap> WidthUTF8String(s);
3
```

6.3 Print Utilities

The following printing utilities turned out to be useful for interactive work with texts in GAP. But they are more general and so we document them here.

6.3.1 PrintTo1

▷ `PrintTo1(`*filename, fun*`)` (function)

▷ `AppendTo1(`*filename, fun*`)` (function)

The argument *fun* must be a function without arguments. Everything which is printed by a call *fun()* is printed into the file *filename*. As with `PrintTo` (**Reference: PrintTo**) and `AppendTo` (**Reference: AppendTo**) this overwrites or appends to, respectively, a previous content of *filename*.

These functions can be particularly efficient when many small pieces of text shall be written to a file, because no multiple reopening of the file is necessary.

Example
```
gap> f := function() local i;
>   for i in [1..100000] do Print(i, "\n"); od; end;;
gap> PrintTo1("nonsense", f); # now check the local file ‘nonsense’
```

6.3.2 StringPrint

▷ `StringPrint(obj1[, obj2[, ...]])` (function)
▷ `StringView(obj)` (function)
▷ `StringDisplay(obj)` (function)

These functions return a string containing the output of a `Print`, `ViewObj` or `Display` call, respectively, with the same arguments.

This should be considered as a (temporary?) hack. It would be better to have `String` (**Reference: String**) methods for all GAP objects and to have a generic `Print` (**Reference: Print**)-function which just interprets these strings.

6.3.3 PrintFormattedString

▷ `PrintFormattedString(str)` (function)

This function prints a string *str*. The difference to `Print(str);` is that no additional line breaks are introduced by GAP's standard printing mechanism. This can be used to print lines which are longer than the current screen width. In particular one can print text which contains escape sequences like those explained in `TextAttr` (6.1.2), where lines may have more characters than *visible characters*.

6.3.4 Page

▷ `Page(...)` (function)
▷ `PageDisplay(obj)` (function)

These functions are similar to `Print` (**Reference: Print**) and `Display` (**Reference: Display**), respectively. The difference is that the output is not sent directly to the screen, but is piped into the current pager; see `Pager` (**Reference: Pager**).

Example
```
gap> Page([1..1421]+0);
gap> PageDisplay(CharacterTable("Symmetric", 14));
```

6.3.5 StringFile

▷ `StringFile(filename)` (function)
▷ `FileString(filename, str[, append])` (function)

The function `StringFile` returns the content of file *`filename`* as a string. This works efficiently with arbitrary (binary or text) files. If something went wrong, this function returns `fail`.

Conversely the function `FileString` writes the content of a string *`str`* into the file *`filename`*. If the optional third argument *`append`* is given and equals `true` then the content of *`str`* is appended to the file. Otherwise previous content of the file is deleted. This function returns the number of bytes written or `fail` if something went wrong.

Both functions are quite efficient, even with large files.

Chapter 7

Utilities for Bibliographies

A standard for collecting references (in particular to mathematical texts) is BibTeX (https://tug.org/bibtex/). A disadvantage of BibTeX is that the format of the data is specified with the use by LaTeX in mind. The data format is less suited for conversion to other document types like plain text or HTML.

In the first section we describe utilities for using data from BibTeX files in GAP.

In the second section we introduce a new XML based data format BibXMLext for bibliographies which seems better suited for other tasks than using it with LaTeX.

Another section will describe utilities to deal with BibXMLext data in GAP.

7.1 Parsing BibTeX Files

Here are functions for parsing, normalizing and printing reference lists in BibTeX format. The reference describing this format is [Lam85, Appendix B].

7.1.1 ParseBibFiles

▷ `ParseBibFiles(`*`bibfile1[, bibfile2[, ...]]`*`)` (function)

▷ `ParseBibStrings(`*`str1[, str2[, ...]]`*`)` (function)

Returns: list `[list of bib-records, list of abbrevs, list of expansions]`

The first function parses the files *`bibfile1`* and so on (if a file does not exist the extension `.bib` is appended) in BibTeX format and returns a list as follows: `[entries, strings, texts]`. Here `entries` is a list of records, one record for each reference contained in *`bibfile`*. Then `strings` is a list of abbreviations defined by `@string`-entries in *`bibfile`* and `texts` is a list which contains in the corresponding position the full text for such an abbreviation.

The second function does the same, but the input is given as GAP strings *`str1`* and so on.

The records in `entries` store key-value pairs of a BibTeX reference in the form `rec(key1 = value1, ...)`. The names of the keys are converted to lower case. The type of the reference (i.e., book, article, ...) and the citation key are stored as components `.Type` and `.Label`. The records also have a `.From` field that says that the data are read from a BibTeX source.

As an example consider the following BibTeX file.

doc/test.bib
```
@string{ j  = "Important Journal" }
@article{ AB2000, Author=  "Fritz A. First and Sec, X. Y.",
TITLE="Short", journal = j, year = 2000 }
```

Example
```
gap> gddirs := DirectoriesPackageLibrary("gapdoc","doc");;
gap> f := Filename(gddirs, "test.bib");;
gap> bib := ParseBibFiles(f);
[ [ rec( From := rec( BibTeX := true ), Label := "AB2000",
          Type := "article", author := "Fritz A. First and Sec, X. Y."
            , journal := "Important Journal", title := "Short",
          year := "2000" ) ], [ "j" ], [ "Important Journal" ] ]
```

7.1.2 NormalizedNameAndKey

▷ `NormalizedNameAndKey(namestr)` (function)

Returns: list of strings and names as lists

▷ `NormalizeNameAndKey(r)` (function)

Returns: nothing

The argument *namestr* must be a string describing an author or a list of authors as described in the BibTEX documentation in [Lam85, Appendix B 1.2]. The function `NormalizedNameAndKey` returns a list of the form [normalized name string, short key, long key, names as lists]. The first entry is a normalized form of the input where names are written as “lastname, first name initials”. The second and third entry are the name parts of a short and long key for the bibliography entry, formed from the (initials of) last names. The fourth entry is a list of lists, one for each name, where a name is described by three strings for the last name, the first name initials and the first name(s) as given in the input.

The function `NormalizeNameAndKey` gets as argument *r* a record for a bibliography entry as returned by `ParseBibFiles` (7.1.1). It substitutes `.author` and `.editor` fields of *r* by their normalized form, the original versions are stored in fields `.authororig` and `.editororig`.

Furthermore a short and a long citation key is generated and stored in components `.printedkey` (only if no `.key` is already bound) and `.keylong`.

We continue the example from `ParseBibFiles` (7.1.1).

Example
```
gap> gddirs := DirectoriesPackageLibrary("gapdoc","doc");;
gap> f := Filename(gddirs, "test.bib");;
gap> bib := ParseBibFiles(f);;
gap> NormalizedNameAndKey(bib[1][1].author);
[ "First, F. A. and Sec, X. Y.", "FS", "firstsec",
  [ [ "First", "F. A.", "Fritz A." ], [ "Sec", "X. Y.", "X. Y." ] ] ]
gap> NormalizeNameAndKey(bib[1][1]);
gap> bib[1][1];
rec( From := rec( BibTeX := true ), Label := "AB2000",
  Type := "article", author := "First, F. A. and Sec, X. Y.",
  authororig := "Fritz A. First and Sec, X. Y.",
  journal := "Important Journal", keylong := "firstsec2000",
  printedkey := "FS00", title := "Short", year := "2000" )
```

7.1.3 WriteBibFile

▷ `WriteBibFile(bibfile, bib)` (function)

Returns: nothing

This is the converse of `ParseBibFiles` (7.1.1). Here *bib* either must have a format as list of three lists as it is returned by `ParseBibFiles` (7.1.1). Or *bib* can be a record as returned by

`ParseBibXMLextFiles` (7.3.4). A BibTEX file *bibfile* is written and the entries are formatted in a uniform way. All given abbreviations are used while writing this file.

We continue the example from `NormalizeNameAndKey` (7.1.2). The command

Example
```
gap> WriteBibFile("nicer.bib", bib);
```

produces a file `nicer.bib` as follows:

nicer.bib
```
@string{j = "Important Journal" }

@article{ AB2000,
  author =          {First, F. A. and Sec, X. Y.},
  title =           {Short},
  journal =         j,
  year =            {2000},
  authororig =      {Fritz A. First and Sec, X. Y.},
  keylong =         {firstsec2000},
  printedkey =      {FS00}
}
```

7.1.4 LabelsFromBibTeX

▷ `LabelsFromBibTeX(`*path, keys, bibfiles, style*`)` (function)

Returns: a list of pairs of strings `[key, label]`

This function uses `bibtex` to determine the ordering of a list of references and a label for each entry which is typeset in a document citing these references.

The argument *path* is a directory specified as string or directory object. The argument *bibfiles* must be a list of files in BibTEX format, each specified by a path relative to the first argument, or an absolute path (starting with `'/'`) or relative to the GAP roots (starting with `"gap://"`). The list *keys* must contain strings which occur as keys in the given BibTEX files. Finally the string *style* must be the name of a bibliography style (like `"alpha"`).

The list returned by this function contains pairs `[key, label]` where `key` is one of the entries of *keys* and `label` is a string used for citations of the bibliography entry in a document. These pairs are ordered as the reference list produced by BibTEX.

Example
```
gap> f := Filename(DirectoriesPackageLibrary("gapdoc","doc"), "test.bib");;
gap> LabelsFromBibTeX(".", ["AB2000"], [f], "alpha");
[ [ "AB2000", "FS00" ] ]
```

7.1.5 InfoBibTools

▷ `InfoBibTools` (info class)

The default level of this info class is 1. Functions like `ParseBibFiles` (7.1.1), `StringBibAs...` are then printing some information. You can suppress it by setting the level of `InfoBibTools` to 0. With level 2 there may be some more information for debugging purposes.

7.2 The BibXMLext Format

Bibliographical data in BibTeX files have the disadvantage that the actual data are given in LaTeX syntax. This makes it difficult to use the data for anything but for LaTeX, say for representations of the data as plain text or HTML. For example: mathematical formulae are in LaTeX `$` environments, non-ASCII characters can be specified in many strange ways, and how to specify URLs for links if the output format allows them?

Here we propose an XML data format for bibliographical data which addresses these problems, it is called BibXMLext. In the next section we describe some tools for generating (an approximation to) this data format from BibTeX data, and for using data given in BibXMLext format for various purposes.

The first motivation for this development was the handling of bibliographical data in GAPDoc, but the format and the tools are certainly useful for other purposes as well.

We started from a DTD `bibxml.dtd` which is publicly available, say from `https://bibtexml.sf.net/`. This is essentially a reformulation of the definition of the BibTeX format, including several of some widely used further fields. This has already the advantage that a generic XML parser can check the validity of the data entries, for example for missing compulsary fields in entries. We applied the following changes and extensions to define the DTD for BibXMLext, stored in the file `bibxmlext.dtd` which can be found in the root directory of this GAPDoc package (and in Appendix C):

names
: Lists of names in the `author` and `editor` fields in BibTeX are difficult to parse. Here they must be given by a sequence of `<name>`-elements which each contain an optional `<first>`- and a `<last>`-element for the first and last names, respectively.

`<M>` **and** `<Math>`
: These elements enclose mathematical formulae, the content is LaTeX code (without the `$`). These should be handled in the same way as the elements with the same names in GAPDoc, see 3.8.2 and 3.8.1. In particular, simple formulae which have a well defined plain text representation can be given in `<M>`-elements.

Encoding
: Note that in XML files we can use the full range of unicode characters, see `https://www.unicode.org/`. All non-ASCII characters should be specified as unicode characters. This makes dealing with special characters easy for plain text or HTML, only for use with LaTeX some sort of translation is necessary.

`<URL>`
: These elements are allowed everywhere in the text and should be represented by links in converted formats which allow this. It is used in the same way as the element with the same name in GAPDoc, see 3.5.5.

`<Alt Only="...">` **and** `<Alt Not="...">`
: Sometimes information should be given in different ways, depending on the output format of the data. This is possible with the `<Alt>`-elements with the same definition as in GAPDoc, see 3.9.1.

`<C>`
: This element should be used to protect text from case changes by converters (the extra `{}` characters in BibTeX title fields).

`<string key="..." value="..."/>` **and** `<value key="..."/>`
: The `<string>`-element defines key-value pairs which can be used in any field via the `<value>`-element (not only for whole fields but also parts of the text).

`<other type="...">`
: This is a generic element for fields which are otherwise not supported. An arbitrary number of them is allowed for each entry, so any kind of additional data can be added to entries.

`<Wrap Name="...">`
: This generic element is allowed inside all fields. This markup will be just ignored (but not the element content) by our standard tools. But it can be a useful hook for introducing arbitrary further markup (and our tools can easily be extended to handle it).

Extra entities
: The DTD defines the standard XML entities (2.1.10 and the entities ` ` (non-breakable space), `–` and `&copyright;`. Use `–` in page ranges.

For further details of the DTD we refer to the file `bibxmlext.dtd` itself which is shown in appendix C. That file also recalls some information from the BibTEX documentation on how the standard fields of entries should be used. Which entry types and which fields are supported (and the ordering of the fields which is fixed by a DTD) can be either read off the DTD, or within GAP one can use the function `TemplateBibXML` (7.3.10) to get templates for the various entry types.

Here is an example of a BibXMLext document:

doc/testbib.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE file SYSTEM "bibxmlext.dtd">
<file>
<string key="j" value="Important Journal"/>
<entry id="AB2000"><article>
  <author>
    <name><first>Fritz A.</first><last>First</last></name>
    <name><first>X. Y.</first><last>Sec&#x0151;nd</last></name>
  </author>
  <title>The <Wrap Name="Package"> <C>F</C>ritz</Wrap> package for the
         formula <M>x^y - l_{{i+1}} \rightarrow \mathbb{R}</M></title>
  <journal><value key="j"/></journal>
  <year>2000</year>
  <number>13</number>
  <pages>13&ndash;25</pages>
  <note>Online data at <URL Text="Bla Bla Publisher">
                 http://www.publish.com/~ImpJ/123#data</URL></note>
  <other type="mycomment">very useful</other>
</article></entry>
</file>
```

There is a standard XML header and a `DOCTYPE` declaration referring to the `bibxmlext.dtd` DTD mentioned above. Local entities could be defined in the `DOCTYPE` tag as shown in the example in 2.2.3. The actual content of the document is inside a `<file>`-element, it consists of `<string>`- and `<entry>`-elements. Several of the BibXMLext markup features are shown. We will use this input document for some examples below.

7.3 Utilities for BibXMLext data

7.3.1 Translating BibTeX to BibXMLext

First we describe a tool which can translate bibliography entries from BibTeX data to BibXMLext `<entry>`-elements. It also does some validation of the data. In some cases it is desirable to improve the result by hand afterwards (editing formulae, adding `<URL>`-elements, translating non-ASCII characters to unicode, ...).

See `WriteBibXMLextFile` (7.3.5) below for how to write the results to a BibXMLext file.

7.3.2 HeuristicTranslationsLaTeX2XML.Apply

▷ `HeuristicTranslationsLaTeX2XML.Apply(`*`str`*`)` (function)

Returns: a string

▷ `HeuristicTranslationsLaTeX2XML.ApplyToFile(`*`fnam[, outnam]`*`)` (function)

Returns: nothing

These utilities translate some LaTeX code into text in UTF-8 encoding. The input is given as a string *`str`*, or a file name *`fnam`*, respectively. The first function returns the translated string. The second function with one argument overwrites the given file with the translated text. Optionally, the translated file content can be written to another file, if its name is given as second argument *`outnam`*.

The record `HeuristicTranslationsLaTeX2XML` mainly contains translations of LaTeX macros for special characters which were found in hundreds of BibTeX entries from MathSciNet. Just look at this record if you want to know how it works. It is easy to extend, and if you have improvements which may be of general interest, please send them to the GAPDoc author.

Example
```
gap> s := "\\\"u\\'{e}\\`e{\\ss}";;
gap> Print(s, "\n");
\"u\'{e}\`e{\ss}
gap> Print(HeuristicTranslationsLaTeX2XML.Apply(s),"\n");
üéèß
```

7.3.3 StringBibAsXMLext

▷ `StringBibAsXMLext(`*`bibentry[, abbrvs, vals][, encoding]`*`)` (function)

Returns: a string with XML code, or `fail`

The argument *`bibentry`* is a record representing an entry from a BibTeX file, as returned in the first list of the result of `ParseBibFiles` (7.1.1). The optional two arguments *`abbrvs`* and *`vals`* can be lists of abbreviations and substitution strings, as returned as second and third list element in the result of `ParseBibFiles` (7.1.1). The optional argument *`encoding`* specifies the character encoding of the string components of *`bibentry`*. If this is not given it is checked if all strings are valid UTF-8 encoded strings, in that case it is assumed that the encoding is UTF-8, otherwise the latin1 encoding is assumed.

The function `StringBibAsXMLext` creates XML code of an `<entry>`-element in `BibXMLext` format. The result is in UTF-8 encoding and contains some heuristic translations, like splitting name lists, finding places for `<C>`-elements, putting formulae in `<M>`-elements, substituting some characters. The result should always be checked and maybe improved by hand. Some validity checks are applied to the given data, for example if all non-optional fields are given. If this check fails the function returns `fail`.

If your BibTEX input contains LaTeX markup for special characters, it can be convenient to translate this input with `HeuristicTranslationsLaTeX2XML.Apply` (7.3.2) or `HeuristicTranslationsLaTeX2XML.ApplyToFile` (7.3.2) before parsing it as BibTEX.

As an example we consider again the short BibTEX file `doc/test.bib` shown in the example for `ParseBibFiles` (7.1.1).

Example
```
gap> gddirs := DirectoriesPackageLibrary("gapdoc","doc");;
gap> f := Filename(gddirs, "test.bib");;
gap> bib := ParseBibFiles(f);;
gap> str := StringBibAsXMLext(bib[1][1], bib[2], bib[3]);;
gap> Print(str, "\n");
<entry id="AB2000"><article>
  <author>
    <name><first>Fritz A.</first><last>First</last></name>
    <name><first>X. Y.</first><last>Sec</last></name>
  </author>
  <title>Short</title>
  <journal><value key="j"/></journal>
  <year>2000</year>
</article></entry>
```

The following functions allow parsing of data which are already in BibXMLext format.

7.3.4 ParseBibXMLextString

▷ `ParseBibXMLextString(str[, res])` (function)

▷ `ParseBibXMLextFiles(fname1[, fname2[, ...]])` (function)

Returns: a record with fields `.entries`, `.strings` and `.entities`

The first function gets a string *str* containing a `BibXMLext` document or a part of it. It returns a record with the three mentioned fields. Here `.entries` is a list of partial XML parse trees for the `<entry>`-elements in *str*. The field `.strings` is a list of key-value pairs from the `<string>`-elements in *str*. And `.strings` is a list of name-value pairs of the named entities which were used during the parsing.

The optional argument *res* can be the result of a former call of this function, in that case the newly parsed entries are added to this data structure.

The second function `ParseBibXMLextFiles` uses the first on the content of all files given by filenames *fname1* and so on. It collects the results in a single record.

As an example we parse the file `testbib.xml` shown in 7.2.

Example
```
gap> gddirs := DirectoriesPackageLibrary("gapdoc","doc");;
gap> f := Filename(gddirs, "testbib.xml");;
gap> bib := ParseBibXMLextFiles(f);;
gap> Set(RecNames(bib));
[ "entities", "entries", "strings" ]
gap> bib.entries;
[ <BibXMLext entry: AB2000> ]
gap> bib.strings;
[ [ "j", "Important Journal" ] ]
gap> bib.entities[1];
[ "amp", "&#38;#38;" ]
```

7.3.5 WriteBibXMLextFile

▷ `WriteBibXMLextFile(fname, bib)` (function)

Returns: nothing

This function writes a BibXMLext file with name *fname*.

There are three possibilities to specify the bibliography entries in the argument *bib*. It can be a list of three lists as returned by `ParseBibFiles` (7.1.1). Or it can be just the first of such three lists in which case the other two lists are assumed to be empty. To all entries of the (first) list the function `StringBibAsXMLext` (7.3.3) is applied and the resulting strings are written to the result file.

The third possibility is that *bib* is a record in the format as returned by `ParseBibXMLextString` (7.3.4) and `ParseBibXMLextFiles` (7.3.4). In this case the entries for the BibXMLext file are produced with `StringXMLElement` (5.2.2), and if *bib*`.entities` is bound then it is tried to resubstitute parts of the string by the given entities with `EntitySubstitution` (5.2.3).

As an example we write back the result of the example shown for `ParseBibXMLextFiles` (7.3.4) to an equivalent XML file.

Example
```
gap> gddirs := DirectoriesPackageLibrary("gapdoc","doc");;
gap> f := Filename(gddirs, "testbib.xml");;
gap> bib := ParseBibXMLextFiles(f);;
gap> WriteBibXMLextFile("test.xml", bib);
```

7.3.6 Bibliography Entries as Records

For working with BibXMLext entries we find it convenient to first translate the parse tree of an entry, as returned by `ParseBibXMLextFiles` (7.3.4), to a record with the field names of the entry as components whose value is the content of the field as string. These strings are generated with respect to a result type. The records are generated by the following function which can be customized by the user.

7.3.7 RecBibXMLEntry

▷ `RecBibXMLEntry(entry[, restype][, strings][, options])` (function)

Returns: a record with fields as strings

This function generates a content string for each field of a bibliography entry and assigns them to record components. This content may depend on the requested result type and possibly some given options.

The arguments are as follows: *entry* is the parse tree of an `<entry>` element as returned by `ParseBibXMLextString` (7.3.4) or `ParseBibXMLextFiles` (7.3.4). The optional argument *restype* describes the type of the result. This package supports currently the types `"BibTeX"`, `"Text"` and `"HTML"`. The default is `"BibTeX"`. The optional argument *strings* must be a list of key-value pairs as returned in the component `.strings` in the result of `ParseBibXMLextString` (7.3.4). The argument *options* must be a record.

If the entry contains an `author` field then the result will also contain a component `.authorAsList` which is a list containing for each author a list with three entries of the form `[last name, first name initials, first name]` (the third entry means the first name as given in the data). Similarly, an `editor` field is accompanied by a component `.editorAsList`.

The following *options* are currently supported.

If `options.fullname` is bound and set to `true` then the full given first names for authors and editors will be used, the default is to use the initials of the first names. Also, if `options.namefirstlast` is bound and set to `true` then the names are written in the form “first-name(s) last-name”, the default is the form “last-name, first-name(s)”.

If `options.href` is bound and set to `false` then the `"BibTeX"` type result will not use `\href` commands. The default is to produce `\href` commands from `<URL>`-elements such that LaTeX with the `hyperref` package can produce links for them.

The content of an `<Alt>`-element with `Only`-attribute is included if *restype* is given in the attribute and ignored otherwise, and vice versa in case of a `Not`-attribute. If `options.useAlt` is bound, it must be a list of strings to which *restype* is added. Then an `<Alt>`-element with `Only`-attribute is evaluated if the intersection of `options.useAlt` and the types given in the attribute is not empty. In case of a `Not`-attribute the element is evaluated if this intersection is empty.

If *restype* is `"BibTeX"` then the string fields in the result will be recoded with `Encode` (6.2.2) and target `"LaTeX"`. If `options.hasLaTeXmarkup` is bound and set to `true` (for example, because the data are originally read from BibTeX files), then the target `"LaTeXleavemarkup"` will be used.

We use again the file shown in the example for `ParseBibXMLextFiles` (7.3.4).

Example
```
gap> gddirs := DirectoriesPackageLibrary("gapdoc","doc");;
gap> f := Filename(gddirs, "testbib.xml");;
gap> bib := ParseBibXMLextFiles(f);;
gap> e := bib.entries[1];; strs := bib.strings;;
gap> Print(RecBibXMLEntry(e, "BibTeX", strs), "\n");
rec(
  From := rec(
      BibXML := true,
      options := rec(
           ),
      type := "BibTeX" ),
  Label := "AB2000",
  Type := "article",
  author := "First, F. A. and Sec{\\H o}nd, X. Y.",
  authorAsList :=
   [ [ "First", "F. A.", "Fritz A." ],
      [ "Sec\305\221nd", "X. Y.", "X. Y." ] ],
  journal := "Important Journal",
  mycomment := "very useful",
  note :=
   "Online data at \\href {http://www.publish.com/~ImpJ/123#data} {Bla\
 Bla Publisher}",
  number := "13",
  pages := "13{\\textendash}25",
  printedkey := "FS00",
  title :=
   "The  {F}ritz package for the \n         formula $x^y - l_{{i+1}} \
\\rightarrow \\mathbb{R}$",
  year := "2000" )
gap> Print(RecBibXMLEntry(e, "HTML", strs).note, "\n");
Online data at <a href="http://www.publish.com/~ImpJ/123#data">Bla Bla\
 Publisher</a>
```

7.3.8 AddHandlerBuildRecBibXMLEntry

▷ `AddHandlerBuildRecBibXMLEntry(`*elementname, restype, handler*`)` (function)

Returns: nothing

The argument *elementname* must be the name of an entry field supported by the BibXMLext format, the name of one of the special elements `"C"`, `"M"`, `"Math"`, `"URL"` or of the form `"Wrap:myname"` or any string `"mytype"` (which then corresponds to entry fields `<other type="mytype">`). The string `"Finish"` has an exceptional meaning, see below.

restype is a string describing the result type for which the handler is installed, see `RecBibXMLEntry` (7.3.7).

For both arguments, *elementname* and *restype*, it is also possible to give lists of the described ones for installing several handler at once.

The argument *handler* must be a function with five arguments of the form *handler*`(entry, r, restype, strings, options)`. Here *entry* is a parse tree of a BibXMLext `<entry>`-element, *r* is a node in this tree for an element *elementname*, and *restype*, *strings* and *options* are as explained in `RecBibXMLEntry` (7.3.7). The function should return a string representing the content of the node *r*. If *elementname* is of the form `"Wrap:myname"` the handler is used for elements of form `<Wrap Name="myname">...</Wrap>`.

If *elementname* is `"Finish"` the handler should look like above except that now *r* is the record generated by `RecBibXMLEntry` (7.3.7) just before it is returned. Here the handler should return nothing. It can be used to manipulate the record *r*, for example for changing the encoding of the strings or for adding some more components.

The installed handler is called by `BuildRecBibXMLEntry(`*entry*, *r*, *restype*, *strings*, *options*`)`. The string for the whole content of an element can be generated by `ContentBuildRecBibXMLEntry(`*entry*, *r*, *restype*, *strings*, *options*`)`.

We continue the example from `RecBibXMLEntry` (7.3.7) and install a handler for the `<Wrap Name="Package">`-element such that LaTeX puts its content in a sans serif font.

Example
```
gap> AddHandlerBuildRecBibXMLEntry("Wrap:Package", "BibTeX",
> function(entry,  r, restype,  strings, options)
>   return Concatenation("\\textsf{", ContentBuildRecBibXMLEntry(
>             entry, r, restype,  strings, options), "}");
> end);
gap>
gap> Print(RecBibXMLEntry(e, "BibTeX", strs).title, "\n");
The \textsf{ {F}ritz} package for the
         formula $x^y - l_{{i+1}} \rightarrow \mathbb{R}$
gap> Print(RecBibXMLEntry(e, "Text", strs).title, "\n");
The  Fritz package for the
         formula x^y - l_{i+1} -> R
gap> AddHandlerBuildRecBibXMLEntry("Wrap:Package", "BibTeX", "Ignore");
```

7.3.9 StringBibXMLEntry

▷ `StringBibXMLEntry(`*entry[, restype][, strings][, options]*`)` (function)

Returns: a string

The arguments of this function have the same meaning as in `RecBibXMLEntry` (7.3.7) but the return value is a string representing the bibliography entry in a format specified by *restype* (default is `"BibTeX"`).

Currently, the following cases for *restype* are supported:

`"BibTeX"`
: A string with BibTEX source code is generated.

`"Text"`
: A text representation of the text is returned. If `options.ansi` is bound it must be a record. The components must have names `Bib_Label`, `Bib_author`, and so on for all fieldnames. The value of each component is a pair of strings which will enclose the content of the field in the result or the first of these strings in which case the default for the second is `TextAttr.reset` (see `TextAttr` (6.1.2)). If you give an empty record here, some default ANSI color markup will be used.

`"HTML"`
: An HTML representation of the bibliography entry is returned. The text from each field is enclosed in markup (mostly `<span>`-elements) with the `class` attribute set to the field name. This allows a detailed layout of the code via a style sheet file. If `options.MathJax` is bound and has the value `true` then formulae are encoded for display on pages with MathJax support.

`"Markdown"`
: A representation of the bibliography entry in Markdown format is returned. If `options.markup` is bound it must be a record which is used in the same way as `options.ansi` for the `"Text"` version.

We use again the file shown in the example for `ParseBibXMLextFiles` (7.3.4).

Example

```
   edited for readability
gap> gddirs := DirectoriesPackageLibrary("gapdoc","doc");;
gap> f := Filename(gddirs, "testbib.xml");;
gap> bib := ParseBibXMLextFiles(f);;
gap> e := bib.entries[1];; strs := bib.strings;;
gap> ebib := StringBibXMLEntry(e, "BibTeX", strs);;
gap> PrintFormattedString(ebib);
@article{ AB2000,
  author =           {First, F. A. and Sec{\H o}nd, X. Y.},
  title =            {The  {F}ritz  package  for  the formula $x^y -
                      l_{{i+1}} \rightarrow \mathbb{R}$},
  journal =          {Important Journal},
  number =           {13},
  year =             {2000},
  pages =            {13{\textendash}25},
  note =             {Online          data          at         \href
                      {http://www.publish.com/~ImpJ/123#data}   {Bla
                      Bla Publisher}},
  mycomment =        {very useful},
  printedkey =       {FS00}
}
gap> etxt := StringBibXMLEntry(e, "Text", strs);;
gap> etxt := SimplifiedUnicodeString(Unicode(etxt), "latin1", "single");;
gap> etxt := Encode(etxt, GAPInfo.TermEncoding);;
gap> PrintFormattedString(etxt);
[FS00]  First,  F.  A.  and Second, X. Y., The Fritz package for the
```

```
formula  x^y  -  l_{i+1}  ?  R, Important Journal, 13 (2000), 13-25,
(Online       data       at       Bla       Bla       Publisher
(http://www.publish.com/~ImpJ/123#data)).

gap> ehtml := StringBibXMLEntry(e, "HTML", strs, rec(MathJax := true));;
gap> ehtml := Encode(Unicode(ehtml), GAPInfo.TermEncoding);;
gap> PrintFormattedString(ehtml);
<p class='BibEntry'>
[<span class='BibKey'>FS00</span>]
 <b class='BibAuthor'>First, F. A. and Second, X. Y.</b>,
 <i class='BibTitle'>The  Fritz package for the
         formula \(x^y - l_{{i+1}} \rightarrow \mathbb{R}\)</i>,
 <span class='BibJournal'>Important Journal</span>
(<span class='BibNumber'>13</span>)
 (<span class='BibYear'>2000</span>),
 <span class='BibPages'>13-25</span><br />
(<span class='BibNote'>Online data at
<a href="http://www.publish.com/~ImpJ/123#data">Bla Bla
Publisher</a></span>).
</p>
```

The following command may be useful to generate completly new bibliography entries in BibXMLext format. It also informs about the supported entry types and field names.

7.3.10 TemplateBibXML

▷ `TemplateBibXML([type])` (function)

Returns: list of types or string

Without an argument this function returns a list of the supported entry types in BibXMLext documents.

With an argument *type* of one of the supported types the function returns a string which is a template for a corresponding BibXMLext entry. Optional field elements have a `*` appended. If an element has the word `OR` appended, then either this element or the next must/can be given, not both. If `AND/OR` is appended then this and/or the next can/must be given. Elements which can appear several times have a + appended. Places to fill are marked by an `X`.

Example
```
gap> TemplateBibXML();
[ "article", "book", "booklet", "conference", "inbook",
  "incollection", "inproceedings", "manual", "mastersthesis", "misc",
  "phdthesis", "proceedings", "techreport", "unpublished" ]
gap> Print(TemplateBibXML("inbook"));
<entry id="X"><inbook>
  <author>
    <name><first>X</first><last>X</last></name>+
  </author>OR
  <editor>
    <name><first>X</first><last>X</last></name>+
  </editor>
  <title>X</title>
  <chapter>X</chapter>AND/OR
```

```
    <pages>X</pages>
    <publisher>X</publisher>
    <year>X</year>
    <volume>X</volume>*OR
    <number>X</number>*
    <series>X</series>*
    <type>X</type>*
    <address>X</address>*
    <edition>X</edition>*
    <month>X</month>*
    <note>X</note>*
    <key>X</key>*
    <annotate>X</annotate>*
    <crossref>X</crossref>*
    <abstract>X</abstract>*
    <affiliation>X</affiliation>*
    <contents>X</contents>*
    <copyright>X</copyright>*
    <isbn>X</isbn>*OR
    <issn>X</issn>*
    <keywords>X</keywords>*
    <language>X</language>*
    <lccn>X</lccn>*
    <location>X</location>*
    <mrnumber>X</mrnumber>*
    <mrclass>X</mrclass>*
    <mrreviewer>X</mrreviewer>*
    <price>X</price>*
    <size>X</size>*
    <url>X</url>*
    <category>X</category>*
    <other type="X">X</other>*+
  </inbook></entry>
```

7.4 Getting BibTEX entries from MathSciNet

We provide utilities to access the MathSciNet data base from within GAP. The first condition for this to work is that one of the programs `wget` or `curl` is installed on your system. The second is, of course, that you use these functions from a computer which has access to MathSciNet.

Please note, that the usual license for MathSciNet access does not allow for automated searches in the database. Therefore, only use the `SearchMR` (7.4.1) function for single queries, as you would do using your webbrowser.

7.4.1 SearchMR

▷ `SearchMR(`*qurec*`)` (function)
▷ `SearchMRBib(`*bib*`)` (function)

Returns: a list of strings, a string or `fail`

The first function `SearchMR` provides the same functionality as the Web interface MathSciNet. The query strings must be given as a record, and the following components of this record are recognized: `Author`, `AuthorRelated`, `Title`, `ReviewText`, `Journal`, `InstitutionCode`, `Series`, `MSCPrimSec`, `MSCPrimary`, `MRNumber`, `Anywhere`, `References` and `Year`.

Furthermore, the component `type` can be specified. It can be one of `"bibtex"` (the default if not given), `"pdf"`, `"html"` and probably others. In the last cases the function returns a string with the content of the web page returned by MathSciNet. In the first case the MathSciNet interface returns a web page with BibTEX entries, for convenience this function returns a list of strings, each containing the BibTEX text for a single result entry.

If a component `uri` is bound and set to `true` the function does not actually send a request to MathSciNet but returns a string with the URI that can be called for the request.

The format of a `.Year` component can be either a four digit number, optionally preceded by one of the characters `'<'`, `'>'` or `'='`, or it can be two four digit numbers separated by a `-` to specify a year range.

The function `SearchMRBib` gets a record of a parsed BibTEX entry as input as returned by `ParseBibFiles` (7.1.1) or `ParseBibStrings` (7.1.1). It tries to generate some sensible input from this information for `SearchMR` and calls that function.

Example
```
gap> ll := SearchMR(rec(Author:="Gauss", Title:="Disquisitiones"));;
gap> ll2 := List(ll, HeuristicTranslationsLaTeX2XML.Apply);;
gap> bib := ParseBibStrings(Concatenation(ll2));;
gap> bibxml := List(bib[1], StringBibAsXMLext);;
gap> bib2 := ParseBibXMLextString(Concatenation(bibxml));;
gap> for b in bib2.entries do
>          PrintFormattedString(StringBibXMLEntry(b, "Text")); od;
[Gau95]  Gauss,  C.  F.,  Disquisitiones  arithmeticae,  Academia
Colombiana  de  Ciencias  Exactas,  Físicas  y  Naturales,  Bogotá,
Colección  Enrique  Pérez  Arbeláez  [Enrique  Pérez  Arbeláez
Collection],  10  (1995), xliv+495 pages, (Translated from the Latin
by  Hugo  Barrantes  Campos,  Michael Josephy and Ángel Ruiz Zúñiga,
With a preface by Ruiz Zúñiga).

[Gau86]  Gauss, C. F., Disquisitiones arithmeticae, Springer-Verlag,
New  York  (1986),  xx+472  pages, (Translated and with a preface by
Arthur  A.  Clarke,  Revised  by  William  C.  Waterhouse, Cornelius
Greither and A. W. Grootendorst and with a preface by Waterhouse).

[Gau66]  Gauss,  C. F., Disquisitiones arithmeticae, Yale University
Press, New Haven, Conn.-London, Translated into English by Arthur A.
Clarke, S. J (1966), xx+472 pages.
```

Appendix A

The File 3k+1.xml

Here is the complete source of the example GAPDoc document 3k+1.xml discussed in Section 1.2.

3k+1.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<!--   A complete "fake package" documentation
-->

<!DOCTYPE Book SYSTEM "gapdoc.dtd">

<Book Name="3k+1">

<TitlePage>
  <Title>The <Package>ThreeKPlusOne</Package> Package</Title>
  <Version>Version 42</Version>
  <Author>Dummy Authör
    <Email>3kplusone@dev.null</Email>
  </Author>

  <Copyright>&copyright; 2000 The Author. <P/>
    You can do with this package what you want.<P/> Really.
  </Copyright>
</TitlePage>

<TableOfContents/>

<Body>
  <Chapter> <Heading>The <M>3k+1</M> Problem</Heading>
    <Section Label="sec:theory"> <Heading>Theory</Heading>
      Let  <M>k \in  &NN;</M> be  a  natural number.  We consider  the
      sequence <M>n(i, k), i \in &NN;,</M> with <M>n(1, k) = k</M> and
      else <M>n(i+1,  k) = n(i, k)  / 2</M> if <M>n(i,  k)</M> is even
      and <M>n(i+1, k) =  3 n(i, k) + 1</M> if  <M>n(i, k)</M> is odd.
      <P/> It  is not known  whether for  any natural number  <M>k \in
      &NN;</M> there is an <M>m \in &NN;</M> with <M>n(m, k) = 1</M>.
      <P/>
      <Package>ThreeKPlusOne</Package>  provides   the  function  <Ref
      Func="ThreeKPlusOneSequence"/>   to  explore   this  for   given
      <M>n</M>.  If  you really  want  to  know something  about  this
```

```
      problem, see <Cite Key="Wi98"/> or
      <URL Text="this homepage">https://www.ku.de/mgf/mathematik/statistik/personen-des-lehrstuhl
      for more details (and forget this package).
    </Section>

    <Section> <Heading>Program</Heading>
      In this section we describe the main function of this package.
      <ManSection>
        <Func Name="ThreeKPlusOneSequence" Arg="k[, max]"/>
        <Description>
          This  function computes  for a  natural number  <A>k</A> the
          beginning of the sequence  <M>n(i, k)</M> defined in section
          <Ref Sect="sec:theory"/>.  The sequence  stops at  the first
          <M>1</M>  or at  <M>n(<A>max</A>, k)</M>,  if <A>max</A>  is
          given.
<Example>
gap> ThreeKPlusOneSequence(101);
"Sorry, not yet implemented. Wait for Version 84 of the package"
</Example>
        </Description>
      </ManSection>
    </Section>
  </Chapter>
</Body>

<Bibliography Databases="3k+1" />
<TheIndex/>

</Book>
```

Appendix B

The File gapdoc.dtd

For easier reference we repeat here the complete content of the file gapdoc.dtd.

gapdoc.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ===================================================================
     gapdoc.dtd - XML Document type definition for GAP documentation
     By Frank Lübeck and Max Neunhöffer
     =================================================================== -->


<!-- Note that this definition goes "bottom-up" because entities can only
     be used after their definition in the file. -->


<!-- ===================================================================
     Some entities:
     =================================================================== -->

<!-- The standard XML entities: -->

<!ENTITY lt     "&#38;#60;">
<!ENTITY gt     "&#62;">
<!ENTITY amp    "&#38;#38;">
<!ENTITY apos   "&#39;">
<!ENTITY quot   "&#34;">


<!-- The following were introduced in GAPDoc version < 1.0, it is no longer
     necessary to take care of LaTeX special characters
     (we keep the entities with simplified definitions for compatibility) -->

<!ENTITY tamp "&amp;">
<!ENTITY tlt "&lt;">
<!ENTITY tgt "&gt;">
<!ENTITY hash "#">
<!ENTITY dollar "$">
<!ENTITY percent "&#37;">
<!ENTITY tilde "~">
<!ENTITY bslash "\\">
```

```
<!ENTITY obrace "{">
<!ENTITY cbrace "}">
<!ENTITY uscore "_">
<!ENTITY circum "^">

<!-- ==================================================================
     Our predefined entities:
     ================================================================== -->

<!ENTITY nbsp " ">
<!ENTITY ndash "&#x2013;">
<!ENTITY GAP    "<Package>GAP</Package>">
<!ENTITY GAPDoc "<Package>GAPDoc</Package>">
<!ENTITY TeX
  "<Alt Only='LaTeX'>{\TeX}</Alt><Alt Not='LaTeX'>TeX</Alt>">
<!ENTITY LaTeX
  "<Alt Only='LaTeX'>{\LaTeX}</Alt><Alt Not='LaTeX'>LaTeX</Alt>">
<!ENTITY BibTeX
  "<Alt Only='LaTeX'>{Bib\TeX}</Alt><Alt Not='LaTeX'>BibTeX</Alt>">
<!ENTITY MeatAxe "<Package>MeatAxe</Package>">
<!ENTITY XGAP   "<Package>XGAP</Package>">
<!ENTITY copyright "&#169;">

<!-- and unicode math symbols -->
<!ENTITY CC "&#x2102;" > <!-- double struck -->
<!ENTITY ZZ "&#x2124;" >
<!ENTITY NN "&#x2115;" >
<!ENTITY PP "&#x2119;" >
<!ENTITY QQ "&#x211a;" >
<!ENTITY HH "&#x210d;" >
<!ENTITY RR "&#x211d;" >


<!-- ==================================================================
     The following describes the "innermost" documentation text which
     can occur at various places in the document like for example
     section headings. It does neither contain further sectioning
     elements nor environments like Enums or Lists.
     ================================================================== -->

<!ENTITY % InnerText "#PCDATA |
                      Alt |
                      Emph | E |
                      Par | P | Br |
                      Keyword | K | Arg | A | Quoted | Q | Code | C |
                      File | F | Button | B | Package |
                      M | Math | Display |
                      Example | Listing | Log | Verb |
                      URL | Email | Homepage | Address | Cite | Label |
                      Ref | Index |
                      Ignore" >
```

```
<!ELEMENT Alt (%InnerText;)*>     <!-- This is only to allow "Only" and
                                       "Not" attributes for normal text -->
<!ATTLIST Alt Only CDATA #IMPLIED
              Not  CDATA #IMPLIED>

<!-- The following elements declare a certain block of InnerText to
     have a certain property. They are non-terminal and can contain
     any InnerText recursively. -->

<!ELEMENT Emph (%InnerText;)*>    <!-- Emphasize something -->
<!ELEMENT E    (%InnerText;)*>    <!-- the same as shortcut -->


<!-- The following is an empty element marking a paragraph boundary. -->

<!ELEMENT Par EMPTY>    <!-- this is intentionally empty! -->
<!ELEMENT P EMPTY>      <!-- the same as shortcut  -->

<!-- And here is an element for forcing a line break, not starting
     a new paragraph. -->

<!ELEMENT Br EMPTY>     <!-- a forced line break  -->

<!-- The following elements mark a word or sentence to be of a certain
     kind, such that it can  be typeset differently. They are terminal
     elements that should only contain  character data. But we have to
     allow  Alt elements  for handling  special characters.  For these
     elements we introduce  a long name - which is  easy to remember -
     and a  short name - which  you may prefer because  of the shorter
     markup. -->

<!ELEMENT Keyword (#PCDATA|Alt)*>  <!-- Keyword -->
<!ELEMENT K (#PCDATA|Alt)*>        <!-- Keyword (shortcut) -->

<!ELEMENT Arg (#PCDATA|Alt)*>      <!-- Argument -->
<!ELEMENT A (#PCDATA|Alt)*>        <!-- Argument (shortcut) -->

<!ELEMENT Code (#PCDATA|Alt|A|Arg)*> <!-- GAP code -->
<!ELEMENT C (#PCDATA|Alt|A|Arg)*>    <!-- GAP code (shortcut) -->

<!ELEMENT File (#PCDATA|Alt)*>     <!-- Filename -->
<!ELEMENT F (#PCDATA|Alt)*>        <!-- Filename (shortcut) -->

<!ELEMENT Button (#PCDATA|Alt)*>   <!-- "Button" (also Menu, Key) -->
<!ELEMENT B (#PCDATA|Alt)*>        <!-- "Button" (shortcut) -->

<!ELEMENT Package (#PCDATA|Alt)*>  <!-- A package name -->

<!ELEMENT Quoted (%InnerText;)*>   <!-- Quoted (in quotes) text -->
<!ELEMENT Q (%InnerText;)*>        <!-- Quoted text (shortcut) -->
```

```
<!-- The following elements contain mathematical formulae. They are
     terminal elements that contain character data in TeX notation. -->

<!-- Math with well defined translation to text output -->
<!ELEMENT M (#PCDATA|A|Arg|Alt)*>
<!-- Normal TeX math mode formula -->
<!ELEMENT Math (#PCDATA|A|Arg|Alt)*>
<!-- TeX displayed math mode formula -->
<!ELEMENT Display (#PCDATA|A|Arg|Alt)*>
<!-- Mode="M" causes <M>-style formatting -->
<!ATTLIST Display Mode CDATA #IMPLIED>


<!-- The  following  elements  contain  GAP related  text  like  code,
     session  logs or  examples. They  are all  terminal elements  and
     consist of character data which is normally typeset verbatim. The
     different  types  of  the  elements only  control  how  they  are
     treated. -->

<!ELEMENT Example (#PCDATA)>  <!-- This is subject to the automatic
                                   example checking mechanism -->
<!ELEMENT Log (#PCDATA)>      <!-- This not -->
<!ELEMENT Listing (#PCDATA)>  <!-- This is just for code listings -->
<!ATTLIST Listing Type CDATA #IMPLIED> <!-- a comment about the type of
                                            listed code, may appear in
                                            output -->

<!-- One  further  verbatim element,  this is truely  verbatim without
     any processing and intended  for ASCII substitutes of complicated
     displayed formulae or tables. -->

<!ELEMENT Verb  (#PCDATA)>

<!-- The following  elements are  for cross-referencing  purposes like
     URLs, citations,  references, and  the index. All  these elements
     are  terminal and  need special  methods  to make  up the  actual
     output during document generation. -->

<!ELEMENT URL (#PCDATA|Alt|Link|LinkText)*>  <!-- Link, LinkText
     variant for case where text needs further markup -->
<!ATTLIST URL Text CDATA #IMPLIED>   <!-- This is for output formats
                                          that have links like HTML -->
<!ELEMENT Link     (%InnerText;)*> <!-- the URL -->
<!ELEMENT LinkText (%InnerText;)*> <!-- text for links, can contain markup -->
<!-- The following two are actually URLs, but the element name determines
     the type. -->
<!ELEMENT Email (#PCDATA|Alt|Link|LinkText)*>
<!ELEMENT Homepage (#PCDATA|Alt|Link|LinkText)*>

<!-- Those who still want to give postal addresses can use the following
     element. Use <Br/> for specifying typical line breaks  -->
```

```
<!ELEMENT Address (#PCDATA|Alt|Br)*>

<!ELEMENT Cite EMPTY>
<!ATTLIST Cite Key CDATA #REQUIRED
               Where CDATA #IMPLIED>

<!ELEMENT Label EMPTY>
<!ATTLIST Label Name CDATA #REQUIRED>

<!ELEMENT Ref EMPTY>
<!ATTLIST Ref Func      CDATA #IMPLIED
              Oper      CDATA #IMPLIED
              Constr    CDATA #IMPLIED
              Meth      CDATA #IMPLIED
              Filt      CDATA #IMPLIED
              Prop      CDATA #IMPLIED
              Attr      CDATA #IMPLIED
              Var       CDATA #IMPLIED
              Fam       CDATA #IMPLIED
              InfoClass CDATA #IMPLIED
              Chap      CDATA #IMPLIED
              Sect      CDATA #IMPLIED
              Subsect   CDATA #IMPLIED
              Appendix  CDATA #IMPLIED
              Text      CDATA #IMPLIED

              Label     CDATA #IMPLIED
              BookName  CDATA #IMPLIED
              Style (Text|Number) #IMPLIED>  <!-- normally automatic -->

<!-- Note that  only one attribute  of Ref is used  normally. BookName
     and  Style  can  be  specified in  addition  to  handle  external
     references and the typesetting style of the reference. -->

<!-- For explicit index entries (Func and so on should cause an
     automatically generated index entry). Use the attributes Key,
     Subkey for sorting (simplified, without markup). The Subkey value
     also gets printed. Use the optional Subkey element if the printed
     version needs some markup.                                       -->
<!ELEMENT Index (%InnerText;|Subkey)*>
<!ATTLIST Index Key    CDATA #IMPLIED
                Subkey CDATA #IMPLIED>
<!ELEMENT Subkey (%InnerText;)*>


<!-- ==================================================================
     The following  describes the normal documentation  text which can
     occur  at various  places in  the document.  It does  not contain
     further sectioning elements. In addition to InnerText it can contain
     environments like enumerations, lists, and such.
     ================================================================== -->
```

```
<!ENTITY % Text "%InnerText; | List | Enum | Table">

<!ELEMENT Item ( %Text;)*>
<!ELEMENT Mark ( %InnerText;)*>

<!ELEMENT List ( ((Mark,Item)|Item)+ )>
<!ATTLIST List Only CDATA #IMPLIED
               Not  CDATA #IMPLIED>
<!ELEMENT Enum ( Item+ )>
<!ATTLIST Enum Only CDATA #IMPLIED
               Not  CDATA #IMPLIED>

<!ELEMENT Table ( Caption?, (Row | HorLine)+ )>
<!ATTLIST Table Label   CDATA #IMPLIED
                Only    CDATA #IMPLIED
                Not     CDATA #IMPLIED
                Align   CDATA #REQUIRED>    <!-- A TeX tabular string -->
                <!-- We allow | and l,c,r, nothing else -->
<!ELEMENT Row   ( Item+ )>
<!ELEMENT HorLine EMPTY>
<!ELEMENT Caption ( %InnerText;)*>

<!-- ==================================================================
     We start defining some things within the overall structure:
     ================================================================== -->

<!-- The TitlePage consists of several sub-elements: -->

<!ELEMENT TitlePage (Title, Subtitle?, Version?, TitleComment?,
                     Author+, Date?, Address?, Abstract?, Copyright?,
                     Acknowledgements? , Colophon? )>

<!ELEMENT Title (%Text;)*>
<!ELEMENT Subtitle (%Text;)*>
<!ELEMENT Version (%Text;)*>
<!ELEMENT TitleComment (%Text;)*>
<!ELEMENT Author (%Text;)*>    <!-- There may be more than one Author! -->
<!ELEMENT Date (%Text;)*>
<!ELEMENT Abstract (%Text;)*>
<!ELEMENT Copyright (%Text;)*>
<!ELEMENT Acknowledgements (%Text;)*>
<!ELEMENT Colophon (%Text;)*>


<!-- The following things just specify some information about the
     corresponding parts of the Book: -->

<!ELEMENT TableOfContents EMPTY>
<!ELEMENT Bibliography EMPTY>
<!ATTLIST Bibliography Databases CDATA #REQUIRED
                       Style CDATA #IMPLIED>
```

```
<!ELEMENT TheIndex EMPTY>

<!-- ================================================================
     The Ignore element can be used everywhere to include further
     information in a GAPDoc document which is not intended for the
     standard converters (e.g., source code, not yet finished stuff,
     and so on. This information can be extracted by special converter
     routines, more precise information about the content of an Ignore
     element can be given by the "Remark" attribute.
     ================================================================ -->

<!ELEMENT Ignore (%Text;| Chapter | Section | Subsection | ManSection |
                  Heading)*>
<!ATTLIST Ignore Remark CDATA #IMPLIED>

<!-- ================================================================
     Now we go on with the overall structure by defining the sectioning
     structure, which includes the Synopsis element:
     ================================================================ -->


<!ELEMENT Subsection (%Text;| Heading)*>
<!ATTLIST Subsection Label CDATA #IMPLIED> <!-- For reference purposes -->

<!ELEMENT ManSection ( Heading?,
                      ((Func, Returns?) | (Oper, Returns?) |
                       (Meth, Returns?) | (Filt, Returns?) |
                       (Prop, Returns?) | (Attr, Returns?) |
                       (Constr, Returns?) |
                       Var | Fam | InfoClass)+, Description )>
<!ATTLIST ManSection Label CDATA #IMPLIED> <!-- For reference purposes -->

<!ELEMENT Returns (%Text;)*>
<!ELEMENT Description (%Text;)*>


<!-- Note that  the ManSection element  is actually a  subsection with
     respect  to labelling,  referencing, and  counting of  sectioning
     elements. -->

<!ELEMENT Func EMPTY>
<!ATTLIST Func Name  CDATA #REQUIRED
               Label CDATA #IMPLIED
               Arg   CDATA #REQUIRED
               Comm  CDATA #IMPLIED>

<!-- Note  that Arg  contains the  full list  of arguments,  including
     optional  parts,  which  are   denoted  by  square  brackets  [].
     Arguments   are  separated   by  whitespace,   commas  count   as
     whitespace. -->

<!-- Note further that although Name and Label are  CDATA (and not ID)
```

```
     Label must make up a unique identifier. -->

<!ELEMENT Oper EMPTY>
<!ATTLIST Oper Name  CDATA #REQUIRED
               Label CDATA #IMPLIED
               Arg   CDATA #REQUIRED
               Comm  CDATA #IMPLIED>

<!ELEMENT Constr EMPTY>
<!ATTLIST Constr Name  CDATA #REQUIRED
                 Label CDATA #IMPLIED
                 Arg   CDATA #REQUIRED
                 Comm  CDATA #IMPLIED>

<!ELEMENT Meth EMPTY>
<!ATTLIST Meth Name  CDATA #REQUIRED
               Label CDATA #IMPLIED
               Arg   CDATA #REQUIRED
               Comm  CDATA #IMPLIED>

<!ELEMENT Filt EMPTY>
<!ATTLIST Filt Name  CDATA #REQUIRED
               Label CDATA #IMPLIED
               Arg   CDATA #IMPLIED
               Comm  CDATA #IMPLIED
               Type  CDATA #IMPLIED>

<!ELEMENT Prop EMPTY>
<!ATTLIST Prop Name  CDATA #REQUIRED
               Label CDATA #IMPLIED
               Arg   CDATA #REQUIRED
               Comm  CDATA #IMPLIED>

<!ELEMENT Attr EMPTY>
<!ATTLIST Attr Name  CDATA #REQUIRED
               Label CDATA #IMPLIED
               Arg   CDATA #REQUIRED
               Comm  CDATA #IMPLIED>

<!ELEMENT Var  EMPTY>
<!ATTLIST Var  Name  CDATA #REQUIRED
               Label CDATA #IMPLIED
               Comm  CDATA #IMPLIED>

<!ELEMENT Fam  EMPTY>
<!ATTLIST Fam  Name  CDATA #REQUIRED
               Label CDATA #IMPLIED
               Comm  CDATA #IMPLIED>

<!ELEMENT InfoClass EMPTY>
<!ATTLIST InfoClass Name  CDATA #REQUIRED
                    Label CDATA #IMPLIED
```

```
                    Comm  CDATA #IMPLIED>


<!ELEMENT Heading (%InnerText;)*>

<!ELEMENT Section (%Text;| Heading | Subsection | ManSection)*>
<!ATTLIST Section Label CDATA #IMPLIED>    <!-- For reference purposes -->


<!ELEMENT Chapter (%Text;| Heading | Section)*>
<!ATTLIST Chapter Label CDATA #IMPLIED>    <!-- For reference purposes -->


<!-- Note that  the entity %InnerText; is  documentation that contains
     neither sectioning  elements nor environments  like enumerations,
     but  only  formulae,  labels, references,  citations,  and  other
     terminal elements. -->

<!ELEMENT Appendix (%Text;| Heading | Section)*>
<!ATTLIST Appendix Label CDATA #IMPLIED>   <!-- For reference purposes -->

<!-- Note that  an Appendix  is exactly  the same  as a  Chapter. They
     differ only in the numbering. -->

<!-- ==================================================================
     At last we define the overall structure of a gapdoc Book:
     ================================================================== -->

<!ELEMENT Body  ( %Text;| Chapter | Section )*>

<!ELEMENT Book (TitlePage,
               TableOfContents?,
               Body,
               Appendix*,
               Bibliography?,
               TheIndex?)>
<!ATTLIST Book Name CDATA #REQUIRED>

<!-- Note  that  the  entity  %Text; is  documentation  that  contains
     no  further sectioning  elements but  possibly environments  like
     enumerations,  and formulae,  labels, references,  and citations.
     -->

<!-- ============================================================== -->
```

Appendix C

The File `bibxmlext.dtd`

For easier reference we repeat here the complete content of the file `bibxmlext.dtd` which is explained in 7.2.

bibxmlext.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
  - (C) Frank Lübeck (https://www.math.rwth-aachen.de/~Frank.Luebeck)
  -
  - The BibXMLext data format.
  -
  - This DTD expresses XML markup similar to the BibTeX language
  - specified for LaTeX, or actually its content model.
  -
  - It is a variation of a file bibxml.dtd developed by the project
  -   https://bibtexml.sf.net/
  -
  - For documentation on BibTeX, see
  -   https://tug.org/bibtex/
  -
  - A previous version of the code originally developed by
  - Vidar Bronken Gundersen, https://bibtexml.sf.net/
  - Reuse and repurposing is approved as long as this
  - notification appears with the code.
  -
-->

<!-- ..................................................................... -->
<!-- Main structure -->

<!-- key-value pairs as in BibTeX @string entries are put in empty elements
     (but here they can be used for parts of an entry field as well)      -->
<!ELEMENT string EMPTY>
<!ATTLIST string
   key        CDATA     #REQUIRED
   value      CDATA     #REQUIRED >

<!-- entry may contain one of the bibliographic types. -->
<!ELEMENT entry ( article | book | booklet |
                         manual | techreport |
```

```
                         mastersthesis | phdthesis |
                         inbook | incollection |
                         proceedings | inproceedings |
                         conference |
                         unpublished | misc ) >
<!ATTLIST entry
   id         CDATA     #REQUIRED >

<!-- file is the documents top element. -->
<!ELEMENT file  ( string | entry )* >


<!-- ................................................................ -->
<!-- Parameter entities -->

<!-- these are additional elements often used, but not included in the
     standard BibTeX distribution, these must be added to the
     bibliography styles, otherwise these fields will be omitted by
     the formatter, we allow an arbitrary number of 'other' elements
     to specify any further information   -->

<!ENTITY   %  n.user " abstract?, affiliation?,
                        contents?, copyright?,
                        (isbn | issn)?,
                        keywords?, language?, lccn?,
                        location?, mrnumber?, mrclass?, mrreviewer?,
                        price?, size?, url?, category?, other* ">

<!ENTITY   %  n.common "key?, annotate?, crossref?,
                        %n.user;">

<!-- content model used more than once -->

<!ENTITY   %  n.InProceedings "author, title, booktitle,
                    year, editor?,
                    (volume | number)?,
                    series?, pages?, address?,
                    month?, organization?, publisher?,
                    note?, %n.common;">

<!ENTITY   %  n.PHDThesis "author, title, school,
                    year, type?, address?, month?,
                    note?, %n.common;">

<!-- ................................................................ -->
<!-- Entries in the BibTeX database -->

<!-- [article] An article from a journal or magazine.
  -  Required fields: author, title, journal, year.
  -  Optional fields: volume, number, pages, month, note. -->
<!ELEMENT   article    (author, title, journal,
               year, volume?, number?, pages?,
```

```
              month?, note?, %n.common;)
>

<!-- [book] A book with an explicit publisher.
  -  Required fields: author or editor, title, publisher, year.
  -  Optional fields: volume or number, series, address,
  -     edition, month, note. -->
<!ELEMENT   book    ((author | editor), title,
              publisher, year, (volume | number)?,
              series?, address?, edition?, month?,
              note?, %n.common;)
>

<!-- [booklet] A work that is printed and bound, but without a named
  -  publisher or sponsoring institution
  -  Required field: title.
  -  Optional fields: author, howpublished, address, month, year, note. -->
<!ELEMENT   booklet    (author?, title,
              howpublished?, address?, month?,
              year?, note?, %n.common;)
>

<!-- [conference] The same as INPROCEEDINGS,
  -  included for Scribe compatibility. -->
<!ELEMENT   conference      (%n.InProceedings;)
>

<!-- [inbook] A part of a book, which may be a chapter (or section or
  -  whatever) and/or a range of pages.
  -  Required fields: author or editor, title, chapter and/or pages,
  -     publisher, year.
  -  Optional fields: volume or number, series, type, address,
  -     edition, month, note. -->
<!ELEMENT   inbook    ((author | editor), title,
              ((chapter, pages?) | pages),
              publisher, year, (volume |
              number)?, series?, type?,
              address?, edition?, month?,
              note?, %n.common;)
>

<!--
  - > I want to express that the elements a and/or b are legal that is one
  - > of them or both must be present in the document instance (see the
  - > element content for BibTeX entry ‘InBook’).
  - > How do I specify this in my DTD?
  -
  - Dave Peterson:
  -  in content model:   ((a , b?) | b)          if order matters
  -                      ((a , b?) | (b , a?))   otherwise
-->
```

```
<!-- [incollection] A part of a book having its own title.
  -  Required fields: author, title, booktitle, publisher, year.
  -  Optional fields: editor, volume or number, series, type,
  -     chapter, pages, address, edition, month, note. -->
<!ELEMENT   incollection    (author, title,
              booktitle, publisher, year,
              editor?, (volume | number)?,
              series?, type?, chapter?,
              pages?, address?, edition?,
              month?, note?,
              %n.common;)
>

<!-- [inproceedings] An article in a conference proceedings.
  -  Required fields: author, title, booktitle, year.
  -  Optional fields: editor, volume or number, series, pages,
  -     address, month, organization, publisher, note. -->
<!ELEMENT   inproceedings      (%n.InProceedings;)
>

<!-- [manual] Technical documentation
  -  Required field: title.
  -  Optional fields: author, organization, address,
  -     edition, month, year, note. -->
<!ELEMENT   manual    (author?, title,
              organization?, address?, edition?,
              month?, year?, note?, %n.common;)
>

<!-- [mastersthesis] A Master's thesis.
  -  Required fields: author, title, school, year.
  -  Optional fields: type, address, month, note. -->
<!ELEMENT   mastersthesis      (%n.PHDThesis;)
>

<!-- [misc] Use this type when nothing else fits.
  -  Required fields: none.
  -  Optional fields: author, title, howpublished, month, year, note. -->
<!ELEMENT   misc    (author?, title?,
              howpublished?, month?, year?, note?,
              %n.common;)
>

<!-- [phdthesis] A PhD thesis.
  -  Required fields: author, title, school, year.
  -  Optional fields: type, address, month, note. -->
<!ELEMENT   phdthesis      (%n.PHDThesis;)
>

<!-- [proceedings] The proceedings of a conference.
  -  Required fields: title, year.
  -  Optional fields: editor, volume or number, series,
```

```
  -     address, month, organization, publisher, note. -->
<!ELEMENT   proceedings    (editor?, title, year,
               (volume | number)?, series?,
               address?, month?, organization?,
               publisher?, note?, %n.common;)
>

<!-- [techreport] A report published by a school or other institution,
  -  usually numbered within a series.
  -  Required fields: author, title, institution, year.
  -  Optional fields: type, number, address, month, note. -->
<!ELEMENT   techreport    (author, title,
               institution, year, type?, number?,
               address?, month?, note?, %n.common;)
>

<!-- [unpublished] A document having an author and title, but not
  -  formally published.
  -  Required fields: author, title, note.
  -  Optional fields: month, year. -->
<!ELEMENT   unpublished    (author, title, note,
               month?, year?, %n.common;)
>

<!-- ............................................................. -->
<!-- Fields from the standard bibliography styles -->

<!--
  - Below is a description of all fields recognized by the standard
  - bibliography styles.  An entry can also contain other fields, which
  - are ignored by those styles.
  -
  - [address] Usually the address of the publisher or other type of
  - institution  For major publishing houses, van~Leunen recommends
  - omitting the information entirely.  For small publishers, on the other
  - hand, you can help the reader by giving the complete address.
  -
  - [annote] An annotation  It is not used by the standard bibliography
  - styles, but may be used by others that produce an annotated
  - bibliography.
  -
  - [author] The name(s) of the author(s), here *not* in the format
  - described in the LaTeX book. Contains elements <name> which in turn
  - contains elements <first>, <last> for the first name (or first names,
  - fully written or as initials, and including middle initials) and
  - the last name.
  -
  - [booktitle] Title of a book, part of which is being cited.  See the
  - LaTeX book for how to type titles.  For book entries, use the title
  - field instead.
  -
  - [chapter] A chapter (or section or whatever) number.
```

```
-
- [crossref] The database key of the entry being cross referenced.
-
- [edition] The edition of a book-for example, ‘‘Second’’.  This
- should be an ordinal, and should have the first letter capitalized, as
- shown here; the standard styles convert to lower case when necessary.
-
- [editor] Name(s) of editor(s), typed as indicated in the LaTeX book.
- If there is also an author field, then the editor field gives the
- editor of the book or collection in which the reference appears.
-
- [howpublished] How something strange has been published.  The first
- word should be capitalized.
-
- [institution] The sponsoring institution of a technical report.
-
- [journal] A journal name.  Abbreviations are provided for many
- journals; see the Local Guide.
-
- [key] Used for alphabetizing, cross referencing, and creating a label
- when the ‘‘author’’ information (described in Section [ref: ] is
- missing. This field should not be confused with the key that appears
- in the \cite command and at the beginning of the database entry.
-
- [month] The month in which the work was published or, for an
- unpublished work, in which it was written. You should use the
- standard three-letter abbreviation, as described in Appendix B.1.3 of
- the LaTeX book.
-
- [note] Any additional information that can help the reader.  The first
- word should be capitalized.
-
- [number] The number of a journal, magazine, technical report, or of a
- work in a series.  An issue of a journal or magazine is usually
- identified by its volume and number; the organization that issues a
- technical report usually gives it a number; and sometimes books are
- given numbers in a named series.
-
- [organization] The organization that sponsors a conference or that
- publishes a manual.
-
- [pages] One or more page numbers or range of numbers, such as 42-111
- or 7,41,73-97 or 43+ (the ‘+’ in this last example indicates pages
- following that don’t form a simple range).  To make it easier to
- maintain Scribe-compatible databases, the standard styles convert a
- single dash (as in 7-33) to the double dash used in TeX to denote
- number ranges (as in 7-33). Here, we suggest to use the entity
- &ndash; for a dash in page ranges.
-
- [publisher] The publisher’s name.
-
- [school] The name of the school where a thesis was written.
```

```
  -
  - [series] The name of a series or set of books.  When citing an entire
  - book, the the title field gives its title and an optional series field
  - gives the name of a series or multi-volume set in which the book is
  - published.
  -
  - [title] The work's title. For mathematical formulae use the <M> or
  - <Math> elements explained below (and LaTeX code in the content, without
  - surrounding '$').
  -
  - [type] The type of a technical report-for example, ``Research
  - Note''.
  -
  - [volume] The volume of a journal or multivolume book.
  -
  - [year] The year of publication or, for an unpublished work, the year
  - it was written.  Generally it should consist of four numerals, such as
  - 1984, although the standard styles can handle any year whose last four
  - nonpunctuation characters are numerals, such as `(about 1984)'.
-->

<!-- Here is the main extension compared to the original BibXML definition
     from which is DTD is derived: We want to allow more markup in some
     elements such that we can use the bibliography for high quality
     output in other formats than LaTeX.

     - <M> and <Math>, mathematical formulae: Specify LaTeX code for "simple"
       formulae as content of <M> elements; "simple" means that they can be
       translated to a fairly readable ASCII representation as explained in
       the GAPDoc documentation on "<M>".
       More complicated formulae are given as content of <Math> elements.
       (Think about an <Alt> alternative for text or HTML representations.)

     - <URL>: use these elements to specify URLs, they can be properly
       converted to links if possible in an output format (in that case
       the Text attribute is used for the visible text).

     - <value key="..."/>:  substituted by the value-attribute specified
       in a <string key="..." value="..."/> element. Can be used anywhere,
       not only for complete fields as in BibTeX.

     - <C> protect case changes: should be used instead of {}'s which are
       used in BibTeX title fields to protect the case of letters from
       changes.

     - <Alt Only="...">, <Alt Not="...">, alternatives for different
       output formats:  Use this to specify alternatives, the GAPDoc
       utilities will do some special handling for "Text", "HTML",
       and "BibTeX" as output type.

     - <Wrap Name="...">, generic wrapper for other markup:
       Use this for any other type of markup you are interested in. The
```

```
      GAPDoc utilities will ignore the markup, but provide a hook
      to do install handler functions for them.
-->
<!ELEMENT   M               (#PCDATA | Alt)* > <!-- math with simple text
                                                representation, in LaTeX -->
<!ELEMENT   Math            (#PCDATA | Alt)* > <!-- other math in LaTeX -->
<!ELEMENT   URL             (#PCDATA | Alt | Link | LinkText)* > <!-- an URL -->
<!ATTLIST   URL Text CDATA #IMPLIED>    <!-- text to be printed
                                              (default is content) -->
<!ELEMENT   value             EMPTY   > <!-- placeholder for value given .. -->
<!ATTLIST   value key CDATA #REQUIRED > <!-- .. by key, defined in a string
                                              element -->
<!ELEMENT   C    (#PCDATA | value | Alt |
                  M | Math | Wrap | URL)* >  <!-- protect from case changes -->
<!ELEMENT   Alt  (#PCDATA | value | C | Alt |
                  M | Math | Wrap | URL)* > <!-- specify alternatives for
                                              various types of output -->
<!ATTLIST   Alt  Only CDATA #IMPLIED
                 Not  CDATA #IMPLIED  > <!-- specify output types in comma and
                  whitespace separated list (use exactly one of Only or Not) -->

<!ENTITY % withMURL "(#PCDATA | value | M | Math | Wrap | URL | C | Alt )*" >

<!ELEMENT   Wrap           %withMURL; > <!-- a generic wrapper  -->
<!ATTLIST   Wrap Name CDATA #REQUIRED > <!-- needs a 'Name' attribute  -->

<!ELEMENT   address         %withMURL; >
<!-- here we don't want the complicated definition from the LaTeX book,
     use markup for first/last name(s): a <name> element for each
     author which contains <first> (optional), <last> elements:  -->
<!ELEMENT   author          (name)* >
<!ELEMENT   name            (first?, last) >
<!ELEMENT   first           (#PCDATA) >
<!ELEMENT   last            (#PCDATA) >

<!ELEMENT   booktitle       %withMURL; >
<!ELEMENT   chapter         %withMURL; >
<!ELEMENT   edition         %withMURL; >
<!-- same as for author field -->
<!ELEMENT   editor          (name)* >
<!ELEMENT   howpublished    %withMURL; >
<!ELEMENT   institution     %withMURL; >
<!ELEMENT   journal         %withMURL; >
<!ELEMENT   month           %withMURL; >
<!ELEMENT   note            %withMURL; >
<!ELEMENT   number          %withMURL; >
<!ELEMENT   organization    %withMURL; >
<!ELEMENT   pages           %withMURL; >
<!ELEMENT   publisher       %withMURL; >
<!ELEMENT   school          %withMURL; >
<!ELEMENT   series          %withMURL; >
<!ELEMENT   title           %withMURL; >
```

```
<!ELEMENT   type            %withMURL; >
<!ELEMENT   volume          %withMURL; >
<!ELEMENT   year            (#PCDATA) >

<!-- These were not listed in the documentation for entry content, but
  -  appeared in the list of fields in the BibTeX documentation -->

<!ELEMENT   annotate        %withMURL; >
<!ELEMENT   crossref        %withMURL; >
<!ELEMENT   key             (#PCDATA) >


<!-- ................................................................ -->
<!-- Other popular fields
  -
  - From: http://www.ecst.csuchico.edu/~jacobsd/bib/formats/bibtex.html
  - BibTeX is extremely popular, and many people have used it to store
  - information. Here is a list of some of the more common fields:
  -
  - [affiliation]  The authors affiliation.
  - [abstract]  An abstract of the work.
  - [contents]  A Table of Contents
  - [copyright]  Copyright information.
  - [ISBN]  The International Standard Book Number.
  - [ISSN]  The International Standard Serial Number.
  -         Used to identify a journal.
  - [keywords]  Key words used for searching or possibly for annotation.
  - [language]  The language the document is in.
  - [location]  A location associated with the entry,
  -             such as the city in which a conference took place.
  - [LCCN]  The Library of Congress Call Number.
  -         I've also seen this as lib-congress.
  - [mrnumber]  The Mathematical Reviews number.
  - [mrclass]  The Mathematical Reviews class.
  - [mrreviewer]  The Mathematical Reviews reviewer.
  - [price]  The price of the document.
  - [size]  The physical dimensions of a work.
  - [URL] The WWW Universal Resource Locator that points to the item being
  -       referenced. This often is used for technical reports to point to the
  -       ftp site where the postscript source of the report is located.
  -
  - When using BibTeX with LaTeX you need
  - BibTeX style files to print these data.
-->

<!ELEMENT   abstract        %withMURL; >
<!ELEMENT   affiliation     %withMURL; >
<!ELEMENT   contents        %withMURL; >
<!ELEMENT   copyright       %withMURL; >
<!ELEMENT   isbn            (#PCDATA) >
<!ELEMENT   issn            (#PCDATA) >
<!ELEMENT   keywords        %withMURL; >
```

```
<!ELEMENT   language        %withMURL; >
<!ELEMENT   lccn            (#PCDATA) >
<!ELEMENT   location        %withMURL; >
<!ELEMENT   mrnumber        %withMURL; >
<!ELEMENT   mrclass         %withMURL; >
<!ELEMENT   mrreviewer      %withMURL; >
<!ELEMENT   price           %withMURL; >
<!ELEMENT   size            %withMURL; >
<!ELEMENT   url             %withMURL; >


<!-- Added by Zeger W. Hendrikse
  - [category]  Category of this bibitem
-->
<!ELEMENT   category      %withMURL; >

<!-- A container element [other] for any further information, a description
   - of the type of data must be given in the attribute 'type'
-->
<!ELEMENT   other      %withMURL; >
<!ATTLIST   other
    type      CDATA   #REQUIRED >


<!-- ............................................................... -->
<!-- Predefined/reserved character entities -->

<!ENTITY amp    "&#38;#38;">
<!ENTITY lt     "&#38;#60;">
<!ENTITY gt     "&#62;">
<!ENTITY apos   "&#39;">
<!ENTITY quot   "&#34;">


<!-- Some more generally useful entities -->
<!ENTITY nbsp " ">
<!ENTITY copyright "&#169;">
<!ENTITY ndash "&#x2013;">


<!-- ............................................................... -->
<!-- End of BibXMLext dtd -->
```

References

[GAP06] The GAP Group, Aachen, St Andrews. *GAP -- Groups, Algorithms, and Programming, Version 4.4.9*, 2006. `https://www.gap-system.org`. 5

[Lam85] L. Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, 1985. 19, 33, 68, 69

Index