# GAPDoc

( Version 0.99 )

**Frank Lübeck**
**Max Neunhöffer**

**Frank Lübeck** — Email: `Frank.Luebeck@Math.RWTH-Aachen.De`
— Homepage: `http://www.math.rwth-aachen.de/~Frank.Luebeck`
**Max Neunhöffer** — Email: `Max.Neunhoeffer@Math.RWTH-Aachen.De`
— Homepage: `http://www.math.rwth-aachen.de/~Max.Neunhoeffer`

## 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. 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 `http://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="ISO-8859-1"?>
```

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 ISO-8859-1, also called ISO-latin1 character set. This is a nowadays widely used extension of the ASCII character set which contains all special characters of Western European languages (e.g., German umlauts and French accented characters).

from 3k+1.xml
```
<!--   A complete "fake package" documentation
   $Id: intro.xml,v 1.5 2001/11/16 15:20:47 gap Exp $
-->
```

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 (as in 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, in particular certain "special characters" must be typed via entities, for example "<", ">" and "&" to avoid a misinterpretation as XML markup. But also the special characters in LaTeX are written with entities, since they need a different handling in a LaTeX and a text output format, see 2.1.10 and 2.2.1 for more details. 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><b>...</a></b>` 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 \N</M> be a natural number. We consider the sequence
      <M>n(i,  k), i  \in  \N,</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 typed as in LATEX. 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 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 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 parse and lots of tools are already available and there will be more in the future. (Our experience was however, that only a few of them are usable currently.)

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:

`http://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="ISO-8859-1"?>
<!DOCTYPE Book SYSTEM "gapdoc.dtd">
```

See 2.1.12 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 characters “`<`” and “`&`”.

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. Important entities for GAPDoc are described in 2.1.10, 2.2.1 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 “`&lt;`” to get “<” and “`&amp;`” to get “`&`”. Furthermore “`&gt;`” should sometimes be used to get “>”.

Another possibility is to use a `CDATA` statement explained in 2.1.11.

### 2.1.11 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 “`]]&gt;`” outside the `CDATA` statement.

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

### 2.1.12 Encoding of an XML document

We suggest to use the ISO-8859-1 or ISO-latin1 encoding for writing GAPDoc XML documents. This character set contains the ASCII characters and all special characters from Western European languages like German umlauts or French accented characters. Text in this character set can be used directly with LaTeX and many current default terminal fonts support this character set.

### 2.1.13 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 More Special Characters

Since one purpose of GAPDoc documents is to produce a high quality LaTeX output version we have to pay attention to characters with a special meaning in LaTeX or in XML. These are the following characters:

“&”, “<”, “>”, “#”, “$”, “%”, “~”, “\”, “{”, “}”, “_”, “^” and “ ” (the last one is a non-breakable space, similar to LaTeX’s “~” character).

The right way to access these symbols is by using “entities”, see 2.1.9. The following table shows what to type to get these characters in the output text of the document.

| | |
|---|---|
| & | `&tamp;` |
| < | `&tlt;` |
| > | `&tgt;` |
| # | `&hash;` |
| $ | `&dollar;` |
| % | `&percent;` |
| ~ | `&tilde;` |
| \ | `&bslash;` |
| { | `&obrace;` |
| } | `&cbrace;` |
| _ | `&uscore;` |
| ^ | `&circum;` |
| | `&nbsp;` |

**Table:** What to type for special characters in character data

Note that the first three have an extra “t” at the beginning in comparison with the standard entities of XML described in 2.1.10. The difference is necessary because for example “`&tamp;`” produces “\&” for LaTeX to actually get an ampersand character in the printed version. Use “`&amp;`” if you want to pass an ampersand character without a backslash in front directly to LaTeX.

Inside attribute values you should *not* use these entities. Instead use the corresponding characters directly. The reason is that attribute values are often used as labels in LaTeX and it is easier to process this properly with the direct input of the characters.

Also, these entities are *not* used inside mathematical formulae, see 2.2.2 below.

### 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 remarks about special characters in 2.2.1 do not apply to the content of these elements. 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 2.1.11.

### 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

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="ISO-8859-1"?>

<!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 an extra document with book name `GAPDocExample` (also accessible from the GAP online help). 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 2.1.13.

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:

`http://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?, Author+, Date?, 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 |
                      Keyword | K | Arg | A | Quoted | Q | Code | C |
                      File | F | Button | B | Package |
                      M | Math | Display |
                      Example | Listing | Log | Verb |
                      URL | Email | Homepage | 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 <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 elements 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.

### 3.2.7 <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.

### 3.2.8 <Abstract>

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

This element contains an abstract of the whole book.

### 3.2.9 <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.

### 3.2.10 <Acknowledgements>

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

This element contains the acknowledgements.

### 3.2.11 <Colophon>

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

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

### 3.2.12 <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

From gapdoc.dtd
```
<TableOfContents/>
```

to denote this empty element.

### 3.2.13 <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 and refers to BibTeX databases. The databases must be separated by commas and must *not* have a .bib extension. 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.

The reference for the format of BibTeX database files is [1, Appendix B].

### 3.2.14 <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

`ManSection`s 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 ( ((Func, Returns?) | (Oper, Returns?) |
                        (Meth, Returns?) | (Filt, Returns?) |
                        (Prop, Returns?) | (Attr, 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`), 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. A possible heading is generated automatically from the first element.

### 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 arguments are separated by whitespace or commas.

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 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 3.2.14 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 3.4.4) which also use the same name. For reference purposes these must be distinguished by different `Label` attributes.

### 3.4.4 <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).

Due to the fact that it often happens that many methods are installed for the same operation it seems to be interesting to document them independently. This is possible by using the same method name in different `ManSection`s. It is however required that these subsections and those describing the corresponding operation are distinguished by different `Label` attributes.

### 3.4.5 <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`”.

### 3.4.6 <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).

### 3.4.7 <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).

### 3.4.8 <**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.

### 3.4.9 <**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.

### 3.4.10 <**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
              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`, `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).

Optionally an external reference into another book can be specified by using the `BookName` attribute. In this case the `Label` attribute *must* be specified and refers to a search string as in the GAP help system. It is guaranteed that the reference points to the position in the other book, that the GAP help system finds as first match if one types the value of the `Label` element after a question mark.

The optional attribute `Style` can take only the values `Text` and `Number`. It can be used with references to sectioning units and it controls, 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 3.2.13). The value of the `Where` attribute specifies the position in the document as in the corresponding LaTeX syntax `\cite[...]{...}`.

### 3.5.4 <Index>

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

This element generates an index entry. The text within the element is typeset in the index entry, which is sorted under the value, that is specified in the `Key` and `Subkey` attributes. If they are not specified, the typeset text itself is used as the key.

Note that all `Func` and similar elements automatically generate index entries. If the `TheIndex` element (3.2.14) is not present in the document all `Index` elements are ignored.

### 3.5.5 <URL>

```
From gapdoc.dtd
<!ELEMENT URL (#PCDATA)>    <!-- Can we define this better? -->
<!ATTLIST URL Text CDATA #IMPLIED>   <!-- This is for output formats
                                          that have links like HTML -->
```

This element is for references into the internet. The text within the element should be a valid URL. It is typeset in the document. For the case of an output document format that supports links the value of the attribute `Text` is typeset as visible text for the link.

### 3.5.6 <Email>

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

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)>
```

This element type is the special case of an URL specifying a WWW-homepage. The content of the element should be the valid URL specifying a world wide web page. In comparison with the `URL` element the address is visible in all output formats.

## 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)|(BigMark,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 identically to the `List` element (see 3.6.1) except that the items may 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 `Description`s in `ManSection`s 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|Alt)*>    <!-- GAP code -->
<!ELEMENT C (#PCDATA|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. No further markup elements are allowed within this element.

### 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 characters 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.

### 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 `Table`s.

## 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)*>
```

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 [1].)

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.

In text and HTML output these formula are shown as LaTeX source code. For simple formulae (and you should try to make all your formulae simple!) there is the element `M` (see 3.8.2) 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` and `Display` elements 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 ASCII 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.

The following LaTeX macros have a sensible ASCII translation and are guaranteed to be translated accordingly by text (and HTML) converters:

| | |
|---|---|
| \ldots | ... |
| \mid | \| |
| \left | |
| \right | |
| \mathbb | |
| \mathop | |
| \limits | |
| \cdot | * |
| \ast | * |
| \geq | >= |
| \leq | <= |
| \pmod | mod |
| \equiv | = |
| \rightarrow | -> |
| \hookrightarrow | -> |
| \to | -> |
| \longrightarrow | --> |
| \Rightarrow | => |
| \Longrightarrow | ==> |
| \Leftarrow | <= |
| \iff | <=> |
| \mapsto | -> |
| \leftarrow | <- |
| \langle | < |
| \rangle | > |
| \setminus | \ |

**Table:** LaTeX macros with special text translation

In all other macros only the backslash is removed. 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 following values are allowed as of now: “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.

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>      <!-- this is intentionally empty! -->
```

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.)

Chapter 4

# Distributing a Document into Several Files

In GAPDoc there are facilities to distribute a document into 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. Basically, there is only a set of conventions how this is done and some tools to collect the text for further processing.

## 4.1 The Conventions

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="`” 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 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 `ComposedXMLString` (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 an XML document.

## 4.2 A Tool for Collecting a Document

### 4.2.1 ComposedXMLString

◇ `ComposedXMLString( path, main, source )` (function)

**Returns:** XML document as string

This function returns a string containing a GAPDoc XML document constructed from several source files.

Here `path` must be a path to some directory (as string or directory object), `main` the name of a file in this directory and `source` a list of file names, all of these relative to `path`. The document is constructed via the mechanism described in Section 4.1.

First the files given in `source` are scanned for chunks of GAPDoc-documentation marked by `<#GAPDoc Label="...">` and `</#GAPDoc>` 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.

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

Chapter 5

# The Converters

The GAPDoc package contains a set of programs which allow 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-, `dvi`-, `pdf-`, postscript- and HTML-version of the document.

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 `ComposedXMLString` (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> str := ComposedXMLString(path, main, files);;
```

Then 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` (5.2.4).

Example
```
gap> r := ParseTreeXMLString(str);;
gap> CheckAndCleanGapDocTree(r);
true
```

We first produce a LaTeX version of the document. `GAPDoc2LaTeX` (5.3.1) returns a string containing the LaTeX source. The utility function `FileString` (5.6.5) writes the content of a string to a file, we choose `MyBook.tex`.

Example
```
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.

Example
```
cd /my/book/path
latex MyBook
bibtex MyBook
makeindex MyBook
latex MyBook
latex MyBook
mv MyBook.dvi manual.dvi
dvips -o manual.ps manual.dvi
rm MyBook.aux
pdflatex MyBook
pdflatex MyBook
mv MyBook.pdf manual.pdf
```

After this we have a `dvi`-, `pdf`- and postscript version of the document in the files `manual.dvi`, `manual.pdf` and `manual.ps`. The first two versions contain hyperlink information which can be used with appropriate browsers for convenient reading of the document on screen (e.g., current versions of `xdvi`, respectively `xpdf` or `acroread`.

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 will use this later.

Next we produce a text version 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 color markup using ANSI escape sequences. One can use this with a terminal which interprets these sequences appropriately after setting the GAP variable `ANSI_COLORS` to `true`. For the bibliography we have to tell `GAPDoc2Text` (5.3.2) the location of the BibTeX database by specifying a `path` as second argument.

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

This command constructs all parts of the document including table of contents, bibliography and index. The functions `FormatParagraph` (5.5.3) for formatting text paragraphs and `ParseBibFiles` (5.4.1) 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. We can add the page number information from the LaTeX version of the document and then print a `manual.six` file which is read by GAP when the document is loaded. This is done with `AddPageNumbersToSix` (5.3.4) and `PrintSixFile` (5.3.5).

Example
```
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` (5.3.6) and `GAPDoc2HTMLPrintHTMLFiles` (5.3.7). We also add a link from `manual.html` to `chap0.html`. You may also add a file `manual.css`, see `GAPDoc2HTMLPrintHTMLFiles` (5.3.7) for more details.

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

### 5.1.1 MakeGAPDocDoc

◊ `MakeGAPDocDoc( path, main, files, bookname[, gaproot] )` (function)

This function collects all the commands for producing a `dvi`-, `pdf`-, postscript-, text- and HTML-version of a GAPDoc document as described in Section 5.1.

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 GAP root directory. If this is given, the HTML files are produced with relative paths to external books.

## 5.2 Parsing XML Documents

Arbitrary well-formed XML documents can be parsed and browsed by the following functions.

### 5.2.1 ParseTreeXMLString

◊ `ParseTreeXMLString( str )` (function)

**Returns:** a record which is root of a tree structure

This function parses an XML-document stored in string `str` and returns the document in form of a tree.

A node in this tree looks corresponds either to an XML element, or 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 2.1.13 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` (5.2.4) for some validation and additional checking of the document structure.

### 5.2.2 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**).

### 5.2.3 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` which is assigned to the top node `tree`. These components can be removed afterwards with `RemoveRootParseTree`.

And here are utilities for processing GAPDoc XML documents.

### 5.2.4 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**).

### 5.2.5 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.

## 5.3 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

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` (5.2.4)). 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: `a4wide`, `amssymb`, `isolatin1`, `makeidx`, `color`, `fancyvrb`, `pslatex` and `hyperref`. These are for example provided by the teTeX-1.0 distribution of TeX (which in turn is used for most TeX packages of current Linux distributions); see `http://www.tug.org/tetex/`.

In particular, the resulting `dvi`- or `pdf`-output contains (internal and external) hyperlinks which can be very useful for online 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). There is support for two types or XML processing instructions which allow to change the options used for the document class or to add some extra lines to the preamble of the LaTeX document. They can be specified as in the following examples:

```
in top level of XML document
<?LaTeX Options="12pt"?>
<?LaTeX ExtraPreamble="\usepackage{blabla}
\newcommand{\bla}{blabla}
"?>
```

A hint for large documents: In many TeX installations one can easily reach some memory limitations with documents which contain many (cross-)references. In teTeX you can look for a file `texmf.cnf` which allows to enlarge certain memory sizes.

This function works by running recursively through the document tree and calling a handler function for each GAPDoc XML element. These handler functions are all quite easy 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.

### 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` (5.2.4)). 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 BibTeX databases. 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 also a record with 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 color markup via ANSI escape sequences, see `TextAttr` (5.5.2). To view the colored text you need a terminal which interprets these escape sequences correctly and you have to set the variable `ANSI_COLORS` to `true` (a good place for doing this is your `.gaprc` file).

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 and use the name of this directory as an argument for one of the commands `DeclarePackageDocumentation` (**Reference: DeclarePackageDocumentation**) or `DeclarePackageAutoDocumentation` (**Reference: DeclarePackageAutoDocumentation**). Furthermore you need to put the file `manual.six` into this directory, see `PrintSixFile` (5.3.5).

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` (5.3.6) to this directory, see `GAPDoc2HTMLPrintHTMLFiles` (5.3.7). 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` (5.3.1)`(tree)` produces a file `pnrfile` (with extension `.pnr`). The command `GAPDoc2Text` (5.3.2)`(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 GAPDoc2HTML

◊ `GAPDoc2HTML( tree[, bibpath[, gaproot]] )` (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` (5.2.4)). This function produces an HTML version of the document which can be read with any Web-browser and also 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 the third argument `gaproot` is given and is a string then this string is interpreted as relative path to GAP's 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 also a record with components The HTML code produced with this converter conforms to the W3C specification HTML 4.01 strict, see `http://www.w3.org/TR/html401`. This means in particular that the code doesn't contain any explicit font or color information. The layout information for a browser should be specified in a cascading style sheet (CSS) file. The GAPDoc package contains an example of such a style sheet, see the file `gapdoc.css` in the root directory of the package. This file conforms to the W3C specification CSS 2.0, see `http://www.w3.org/TR/REC-CSS2`. You may just copy that file as `manual.css` into the directory which contains the HTML version of your documentation. But, of course, you are free to adjust it for your package, e.g., change colors or other layout details, add a background image, ... Each of the HTML files produced by the converters contains a link to this local style sheet file called `manual.css`.

**`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)

The result can be written into files with the command `GAPDoc2HTMLPrintHTMLFiles` (5.3.7).

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.

### 5.3.7 GAPDoc2HTMLPrintHTMLFiles

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

**Returns:** nothing

The first argument must be a result returned by `GAPDoc2HTML` (5.3.6). 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`.

You can make these files accessible via the GAP online help by putting them into a directory and using this as an argument for one of the commands `DeclarePackageDocumentation` (**Reference: DeclarePackageDocumentation**) or `DeclarePackageAutoDocumentation` (**Reference: DeclarePackageAutoDocumentation**). To tell GAP that the HTML version is accessible you have to add a file `manual.html` which is a link to or a copy of `chap0.html`. You may also want to put a file `manual.css` into that directory, see `GAPDoc2HTML` (5.3.6).

## 5.4 Parsing BibTeX Files

Here are functions for parsing, normalizing and printing reference lists in BibTeX format. The reference describing this format is [1, Appendix B].

### 5.4.1 ParseBibFiles

◊ `ParseBibFiles( bibfile )` (function)

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

This function parses a file `bibfile` (if this 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 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`.

As an example consider the following BibTeX file.

my.bib
```
@string{ j  = "Important Journal" }
@article{ AX2000, Author=  "Fritz A. First and Sec, X. Y.",
TITLE="Short", journal = j, year = 2000 }
```

Example
```
gap> bib := ParseBibFiles("my.bib");
[ [ rec( Type := "article", Label := "AB2000",
          author := "Fritz A. First and Sec, X. Y.", title := "Short",
          journal := "Important Journal", year := "2000" ) ],
  [ "j" ],
  [ "Important Journal" ] ]
```

### 5.4.2 NormalizeNameAndKey

◊ `NormalizeNameAndKey( r )` (function)

**Returns:** nothing

This function normalizes in a record describing a BibTeX reference (see `ParseBibFiles` (5.4.1)) the `author` and editor fields using the description in [1, Appendix B 1.2]. The original entries are stored in `.authororig` and `.editororig`.

Furthermore a short and a long citation key is generated.

We continue the example from `ParseBibFiles` (5.4.1).

Example
```
gap> bib[1][1];
rec( Type := "article", Label := "AB2000",
  author := "First, F. A. and Sec, X. Y. ", title := "Short",
  journal := "Important Journal", year := "2000",
  authororig := "Fritz A. First and Sec, X. Y.", key := "FS00",
  keylong := "firstsec2000" )
```

### 5.4.3 WriteBibFile

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

**Returns:** nothing

This is the converse of `ParseBibFiles` (5.4.1). Here `bib` must have a format as it is returned by `ParseBibFiles` (5.4.1). 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` (5.4.2). 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},
  key =             {FS00},
  authororig =      {Fritz A. First and Sec, X. Y.},
  keylong =         {firstsec2000}
}
```

## 5.5 Text Utilities

This section describes some utility functions for handling texts within GAP. They are used by other 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.

### 5.5.1 WHITESPACE

◊ `WHITESPACE` (global variable)
◊ `CAPITALLETTERS` (global variable)
◊ `SMALLLETTERS` (global variable)
◊ `LETTERS` (global variable)
◊ `DIGITS` (global variable)
◊ `HEXDIGITS` (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"`

### 5.5.2 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 the variable `ANSI_COLORS` has the value `true`.

### 5.5.3 FormatParagraph

◊ `FormatParagraph( str[, len[, flush[, attr]]] )` (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 resulting text (default is `78`)

**`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 `["", ""]`)

This function tries to handle markup with the escape sequences explained in `TextAttr` (5.5.2) 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. */
```

### 5.5.4 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"
```

### 5.5.5 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.

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

### 5.5.6 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` (5.5.2). If `str` does not contain such a sequence then `str` itself is returned.

### 5.5.7 RepeatedString

◊ `RepeatedString( 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`.

```
Example
gap> RepeatedString('=',51);
"==================================================="
gap> RepeatedString("*=",51);
"*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*"
```

### 5.5.8 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"
```

### 5.5.9 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
```

### 5.5.10 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.

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

## 5.6 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.

### 5.6.1 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’
```

### 5.6.2 StringPrint

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

These functions return a string containing the output of a `Print` or `ViewObj` call 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.

### 5.6.3 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` (5.5.2), where lines may have more characters than *visible characters*.

### 5.6.4 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));
```

### 5.6.5 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.

# 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="ISO-8859-1"?>

<!--   A complete "fake package" documentation
   $Id: app3k1.xml,v 1.1.1.1 2001/01/05 13:37:49 gap Exp $
-->

<!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 \N</M> be a natural number. We consider the sequence
      <M>n(i,  k), i  \in  \N,</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 \N</M>
      there is an <M>m \in \N</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>http://mathsrv.ku-eichstaett.de/MGF/homes/wirsching/</URL>
      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="ISO-8859-1"?>
<!-- ==================================================================
     gapdoc.dtd - XML Document type definition for GAP documentation
     By Frank Lübeck and Max Neunhöffer
     ================================================================== -->

<!-- $Id: gapdoc.dtd,v 1.5 2002/05/02 20:39:58 gap Exp $ -->

<!-- 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 are necessary because these characters have special
     meanings in either XML or LaTeX: -->

<!ENTITY tamp
  "<Alt Only='LaTeX'>\&amp;</Alt><Alt Not='LaTeX'><Alt Only='HTML'>&amp;amp;</Alt><Alt Not='HTML'>&amp
<!ENTITY tlt
  "<Alt Only='LaTeX'>{\textless}</Alt><Alt Not='LaTeX'><Alt Only='HTML'>&amp;lt;</Alt><Alt Not='HTML'>
<!ENTITY tgt
  "<Alt Only='LaTeX'>{\textgreater}</Alt><Alt Not='LaTeX'><Alt Only='HTML'>&amp;gt;</Alt><Alt Not='HTM
<!ENTITY hash "<Alt Only='LaTeX'>\#</Alt><Alt Not='LaTeX'>#</Alt>">
<!ENTITY dollar "<Alt Only='LaTeX'>\$</Alt><Alt Not='LaTeX'>$</Alt>">
```

```
<!ENTITY percent
  "<Alt Only='LaTeX'>\&#37;</Alt><Alt Not='LaTeX'>&#37;</Alt>">
<!ENTITY tilde
  "<Alt Only='LaTeX'>{\textasciitilde}</Alt><Alt Not='LaTeX'>~</Alt>">
<!ENTITY bslash
  "<Alt Only='LaTeX'>{\textbackslash}</Alt><Alt Not='LaTeX'>\</Alt>">
<!ENTITY obrace "<Alt Only='LaTeX'>\{</Alt><Alt Not='LaTeX'>{</Alt>">
<!ENTITY cbrace "<Alt Only='LaTeX'>\}</Alt><Alt Not='LaTeX'>}</Alt>">
<!ENTITY uscore
  "<Alt Only='LaTeX'>{\textunderscore}</Alt><Alt Not='LaTeX'>_</Alt>">
<!ENTITY circum
  "<Alt Only='LaTeX'>{\textasciicircum}</Alt><Alt Not='LaTeX'>^</Alt>">
<!ENTITY nbsp "<Alt Only='LaTeX'>~</Alt><Alt Not='LaTeX'>&#160;</Alt>">

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

<!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
  "<Alt Only='LaTeX'>{\copyright}</Alt><Alt Not='LaTeX'>(C)</Alt>">


<!-- ==================================================================
     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 |
                      Keyword | K | Arg | A | Quoted | Q | Code | C |
                      File | F | Button | B | Package |
                      M | Math | Display |
                      Example | Listing | Log | Verb |
                      URL | Email | Homepage | Cite | Label |
                      Ref | Index" >


<!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>      <!-- this is intentionally empty! -->


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


<!-- 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)*>    <!-- Can we define this better? -->
<!ATTLIST URL Text CDATA #IMPLIED>   <!-- This is for output formats
                                          that have links like HTML -->

<!-- The following two are actually URLs, but the element name determines
     the type. -->

<!ELEMENT Email (#PCDATA|Alt)*>
<!ELEMENT Homepage (#PCDATA|Alt)*>

<!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
              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. -->

<!ELEMENT Index (%InnerText;)*>
<!ATTLIST Index Key    CDATA #IMPLIED
                Subkey CDATA #IMPLIED>


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

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

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

<!ELEMENT List ( ((Mark,Item)|(BigMark,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?, Author+, Date?, Abstract?,
                     Copyright? , Acknowledgements? , Colophon? )>

<!ELEMENT Title (%Text;)*>
<!ELEMENT Subtitle (%Text;)*>
<!ELEMENT Version (%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>


<!-- ==================================================================
     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 (((Func, Returns?) | (Oper, Returns?) |
                       (Meth, Returns?) | (Filt, Returns?) |
                       (Prop, Returns?) | (Attr, 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  even if 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 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.
     -->

<!-- ================================================================== -->
```

# References

[1] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, 1985. 18, 30, 42, 43

# Index