Gummi project specification

Torsten Bronger

18. August 2007

Contents

1 Gummi 1.0 specification **4**

1.1 Gummi file format . . . 4

1.1.1 Header . . . 4

1.1.2 Comment lines . . . 5

1.1.3 The input method . . . 5

1.1.4 Escaping . . . 5

1.1.5 Structuring . . . 6

1.1.6 Cross referencing . . . 6

1.1.7 Inline markup . . . 7

1.1.8 Footnotes, hyperlinks, and references . . . 7

1.1.8.1 Footnotes . . . 7

1.1.8.2 Hyperlinks . . . 7

1.1.8.3 References . . . 7

1.1.9 Source code excerpts . . . 7

1.1.10 Tables . . . 7

1.1.11 Mathematical material . . . 7

1.1.11.1 Physical units . . . 7

1.1.12 Images and floating material . . . 7

1.1.13 Directives and roles . . . 7

1.1.13.1 Directives . . . 7

1.1.13.2 Meta information . . . 8

1.1.13.3 Roles . . . 8

1.2 Input method file format . . . 8

1.3 Additional implementation requirements . . . 9

1.4 API for backends . . . 9

1.4.1 The content models . . . 9

1.5 Backend requirements . . . 9

2 Reference implementation **10**

2.1 Infrastructure . . . 10

2.2 Name . . . 10

2.3 Goals . . . 10

2.4 Design guidelines of the Gummi text format . . . 10

2.5 Structure of the program code . . . 12

2.5.1 The abstract syntax tree (AST) . . . 12

2.5.2 Preprocessor 12
2.5.3 Parser 13
2.5.4 Backends 13
2.6 Themes 13
2.7 Writebacks 13

Chapter 1

Gummi 1.0 specification

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

1.1 Gummi file format

Gummi files are pure text files in an octet-based encoding. This encoding SHOULD be supported by current Python implementations, see `http://docs.python.org/lib/standard-encodings.html`. The encoding MUST have ASCII as its 7-bit subset.

To ensure interoperability, line endings may be those of Unix (LF), Windows (CR+LF), or Macintosh (CR) systems. They may even be mixed in one file. Therefore, Gummi treats line endings like XML does: LF, CR+LF, and any CR not followed by LF are regarded as one line ending. (LF: ASCII `0Ah`, CR: ASCII `0Dh`)

If not stated otherwise, whitespace characters are SPACE (ASCII: `20h`) and TAB (ASCII: `09h`). If not stated otherwise, a sequence of whitespace characters is treated as one whitespace character.

1.1.1 Header

A Gummi file MAY declare Emacs-like local variables in its very first line. This line MUST have the following format:

```
.. -*- keyword1: value1; keyword2: value2 -*-
```

All whitespace is OPTIONAL except for the very beginning of the file: It must be "`..␣`".

Both, keywords and values, MUST consist only of the following characters: `[A-Za-z0-9_-]`. The values may additionally contain the comma but *not* whitespace. Both keywords and values are treated case-insensitively. The following keywords are used by Gummi:

`coding`	encoding of the file (default depends on the implementation)
`input-method`	input method(s) of the file, comma-separated. Default: `minimal`

Both are OPTIONAL. Other keyword–value pairs MAY be declared in the line.

Immediately after the local variables in the second line of the file – or in the very first line if no local variables were declared – there MAY be a Gummi version line. Its format is as follows:

`.. Gummi` ⟨*version number*⟩

The version line MUST start with "`..␣`". The version number of this specification is "`1.0`".

1.1.2 Comment lines

Comment lines match the following regular expression:

```
^\.\.([ \t].*)?$
```

If not stated otherwise, this is meant by "comment line" throughout this specification.

1.1.3 The input method

The input method is a mechanism for entering characters that are not easily accessible to the user, e. g. Greek letters, mathematical operators, or special typographic characters. The substitution rules of the input method are applied before the parser sees the text. However, the input method MUST NOT remove syntactically relevant characters used for markup, including LF or CR. It may add them, though. The replacement MUST be a single unicode character which is not LF or CR.

The default input method is called "`minimal`". It is used if no input method is given in the local variables line. The input method "`minimal`" is specified by the reference implementation and MUST be provided by any Gummi implementation. The special input method name "`none`" means that no input method is applied.

The text is searched from the beginning to the end. If a portion of the text matches an item of the input method, the portion is replaced with the replacement of that item.

The following three points determine which substitution rule is applied. If a point means equality for two or more rules, the next point is tried.

1. earliest match
2. longest match
3. match declared last in the input method file(s)

1.1.4 Escaping

Gummi uses the backslash »\« for escaping. Escaping takes place on two levels, namely the preprocessor and the parser level. In order to avoid having two different escaping characters, things are a little bit more complicated than usual.

Preprocessor

On the preprocessor level, escaping hinder input method matches from being applied. If a backslash is immediately before a match, the corresponding substitution is not realised. However, scanning for matches re-starts at the second character of the (ignored) match. Similarly, a backslash within a match spoils the substitution. However, the prepending characters may correspond to a shorter match which is then applied.

A double backslash generates one backslash that is printed as is and that is not used in any escaping. It is only marked as escaped if deferred escaping was used, see next section. Furthermore, "`[[`" and "`]]`" are transformed to *escaped* "`[`" and "`]`", respectively

A character may be given by its decimal Unicode number of the form `\#1234;` or its hexadecimal Unicode number of the form `\0x123f;`. This has precedence over all input method matches.

Parser

On the parser level, a backlash immediately before a character avoids using this character for syntactic structuring. In other words, an escaped character is printed as is and is not used for any kind of markup. Characters which a used for markup in certain contexts needn't be escaped in other contexts. Superfluous escaping is ignored.

Things are getting somewhat cumbersome if an input method match should be escaped on the parser level. For example, "→" is the syntactic symbol for cross referencing. If you enter it as `-->`, the minimal input method replaces it with "→". However, if you want to have the arrow as is, i. e. not as a syntactic command, you can't just write `\-->` because this prints as "-->".

To overcome this problem, there is the so-called "deferred escaping": A backslash before a match, with any amount of whitespace, including up to one linebreak inbetween, escapes the character that is inserted by the substitution. The whitespace (not the linebreak) is removed from the input. Thus you could write "`\␣-->`" to get an arrow which does not begin a cross reference. Note that you can also simply write "\→" if your editor lets you do this.

If no substitution has taken place (i. e., the deferred escaping was unnecessary), the deferred escaping is just like an ordinary parser-level escaping. The inner whitespace is still removed, though.

Source code excerpts

Within source code excerpts, escaping is radically different. For escaping, »`\‘`« instead of »`\`« is used. However, it's only meaningful to escape »`‘‘‘`« and »`\‘`« itself. Apart from this, the whole excerpt is copied as is to the output until the next »`‘‘‘`«. This is to ensure that escaping is barely necessary in source code excerpts.

1.1.5 Structuring

1.1.6 Cross referencing

1.1.7 Inline markup

1.1.8 Footnotes, hyperlinks, and references

1.1.8.1 Footnotes

1.1.8.2 Hyperlinks

1.1.8.3 References

1.1.9 Source code excerpts

1.1.10 Tables

1.1.11 Mathematical material

1.1.11.1 Physical units

1.1.12 Images and floating material

1.1.13 Directives and roles

1.1.13.1 Directives

1.1.13.2 Meta information

1.1.13.3 Roles

1.2 Input method file format

In the first line of an input file, local variables are set analogously to the Gummi file itself. However, in input method files, this line is required. It may be for example:

```
.. -*- input-method-name: my_method; parental-input-method: minimal -*-
```

The following fields are known to Gummi, however, further field may be added, they are ignored by Gummi.

`input-method-name`	REQUIRED; name of the input method of the file
`coding`	encoding of the file. Default: `utf-8`
`parental-input-method`	name(s) of input method(s) this method bases upon; comma-separated

The second line MUST be the exactly following (only additional trailing whitespace is allowed):

```
.. Gummi input method
```

From there on, all lines only consisting of whitespace or starting with "`..␣`" are ignored. All other lines are interpreted as one substitution rule. Every substitution rule starts with the match, followed by one or more tabs, followed by one single character which is the replacement. After that, the line may contain an arbitrary comment separated by whitespace. The replacement character may also be given by its decimal Unicode number of the form `#1234` or its hexadecimal Unicode number of the form `0x123f`.

By default, the match is a simple string. With the prefix `REGEX::`, everything after this prefix is a Python regular expression according to `http://docs.python.org/lib/re-syntax.html`. The regular expression MUST NOT contain groups, i. e. "`(...)`" is forbidden, while "`(?:...)`" is allowed, for example.

By default, the substitutions are done before the parser sees the input. In rare cases this is unfortunate, the most prominent being "--" which is transformed to "–". This breaks the parsing of tables with their horizontal lines, as well as dashed lines which generate skips between paragraphs. In order to avoid this, an input method rule may be prepended by `POST::` which means that it is applied *after* the parser has done its work. Note that this means that it cannot generate syntactic relevant information of any kind. Additionally, POST rules may use replacement characters of the first pass as part of their match, unless these replacements were escaped.

The matches (normal strings or regexps) are matched against the Gummi source text on string level, i. e. only the proper encoding is already applied to both the input method match and the source code.

The order of the substitution rules is significant. The longest match wins. If two rules match with the same length, the latest match in the Gummi file is used. Therefore, rules which are defined later in the input method file may override earlier ones (or those in a parental input method), unless their match is shorter, as explained in section 1.1.3.

If the local variable `parental-input-method` is set, all input methods mentioned in this variable are read in the order given in this variable. Then, the rules in the file are read.

The file name of an input method must be the name of the input method itself plus the extension "`.gim`".

1.3 Additional implementation requirements

1.4 API for backends

1.4.1 The content models

`Document`: `*block*, Section*`

`Section`: `Heading, *block*, Section*`

`Heading`: `*inline*`

`inline`: `*Text* | Emphasize`

`Emphasize`: `*inline*`

1.5 Backend requirements

Chapter 2

Reference implementation

2.1 Infrastructure

The project's home page is at `http://bobcat.origo.ethz.ch/`.

The mailing list is on `https://lists.sourceforge.net/lists/listinfo/latex-bronger-gummi`. It is also listed on Gmane with the group name `gmane.text.formats.gummi`.

The SVN repository of the source code is at `http://bobcat.origo.ethz.ch/wiki/development`. You can also browse through the interfaces at `http://latex-bronger.sourceforge.net/gummi/epydoc/`.

A sample document – which is at the moment the only definitive specification – is at `http://latex-bronger.sourceforge.net/gummi/gummi-example.html`.

2.2 Name

The Name "Gummi" is provisional. Better suggestions are welcomed. The name should be cute and allow for a nice mascot.

2.3 Goals

The primary goal is to generate a prototype in Python that reads a Gummi source code, converts it internally into an Abstract Source Tree (AST), and pass this to backends for at least PDF (via LaTeX) and HTML.

Moreover, it may be feasible and sensible to develop a GUI with editor and syntax highlighting. My experiences with wxPython showed that this can be done with less than 1000 lines of code, at least for the basic stuff.

2.4 Design guidelines of the Gummi text format

The text format is designed along the following guidelines:

- It is supposted to be entered intuitively by the author.
- It is supposed to be well-legible in its original form.

- It must have an unambiguous syntax. Syntax errors are fatal, however, there is a second class of inconsistencies which generates warnings only.
- It is *not* important that the format can be parsed easily. Well, it must be realistic that amateur programmers can write a parser for it, but that's all. The important thing is that the syntax is practical for most authors.
- Semantic markup takes precedence over visual markup but not at all costs. Clear separation between contents (Gummi file) and layout (themes) but not at all costs.
- The target group are all authors who want to write non-trivial documents with the computer. It's not for the geeks.

Yet another set of design guidelines, inspired from Tim Peter's *Zen of Python*:

- Make the frequent things simple at the expense of the rare things.
- If the rare thing made the frequent thing less simple, declare it impossible.
- Contents and structure belong to the document but the layout belongs to the theme. Both must not be kept within the same file.
- Unicode is good, encoding are evil.
- A document is written and not programmed. Preambles à la LaTeX are evil.
- Syntax must serve the author, not the parser.
- There should be one obvious way to do it.
- Except for tables.
- Local layout adjustments are a bad idea. Global layout adjustments may be a good idea.
- Minor systematic weaknesses in the layout are acceptable if justified by the gain in usability and processability of the documents.
- Extension mechanisms are unavoidable but problematic. They must not make some documents work under certain conditions only.
- Syntax must be multi-lingual. Case sensitivity is important.
- Don't give the author 1001 layout knobs to play with. Good layout may be gently forced.
- Flat is better than nested.
- Process only documents without syntax errors. Try to solve problems automatically but not without asking the author.
- It is allowed to change documents with the autor's consent.
- People love themes.

2.5 Structure of the program code

The program should be useable as both a stand-alone command line tool and a Python module. It consists of three parts:

1. The parser,
2. the AST, consisting of class instances,
3. the backends, realised as Python modules.

Technically, the first two points are intertwined, i.e., the classes also contain the code to parse them and their children.

2.5.1 The abstract syntax tree (AST)

The AST is some kind of DOM (document object model), i.e. a tree-like structure made of class instances. You may see them as somethin like an XML file:

```
<document>
  <section>
    <para>This is the <emph>first</emph> paragraph.</para>
  <section>
<document>
```

Note that "`This is the `", "`first`", and "` paragraph.`" are three so-called text nodes in XML, which are separate elements in the tree. This is the same in Gummi: There are text nodes, which can't contain any children any more. They are pure character sequences.

All classes are derived from the class `Node`. They have in common to be able to have children (this list may be empty of course), and some methods for parsing and backend-processing.

The elements in the AST should have a large set of attributes and special method so that the backend gets a rich and convenient environment. For example, the `Section` class can give its nesting level although the backend could find it out itself by parsing the section number. However, not only Gummi authors but also backend authors should live as comfortable as possible.

2.5.2 Preprocessor

There is a preprocessor before any parsing. Its purpose is twofold:

1. Read in the source code and transform it to a Unicode-string-like data structure (called `Excerpt`) that also records escaped characters, and the original file positions of every character (for generating verbose error messages).
2. Perform the substitutions of the input method.

2.5.3 Parser

All cross references, including those to text blocks, footnotes and annoted hyperlinks are included as references at first. After the document has been processed fully, these references are resolved, i.e. they are copied to the referencing elements in the AST.

Bibliographic references and meta data is collected in the root node of the AST, an instance of the class `Document`.

2.5.4 Backends

A backend is responsible for exactly one output file format. The backend's routines are injected into the classes of the AST, so they override dummy methods in the AST classes. Then, the process method of the root node is called which then calls recursively all process methods of the tree. By this, the final output (e.g. the LaTeX file) is generated.

The backend may contain additional code, e.g. for generating PDF from the LaTeX file, or for calling BibTeX or xindy, or for cleaning up the working directory.

2.6 Themes

Users love themes. In Gummi, themes are groups of backends. Thus, a theme "PhD thesis IEF-5" may consist of a LaTeX/PDF and an HTML backend. Is a certain output format not available in a theme, there's a fallback with a warning.

For some backends, there will be auxiliary modules that contain code which is useful for all themes that implement the respective backend. Moreover, a backend can install another backend first and then just override some things. For example, the LaTeX backends will differ only very slightly, maybe even in the preamble.

The standard theme is called "Standard".

You may pass options to the theme – effectively to the backend of that theme – in the Gummi source document. So some things can be set to personal preferences, like paper size, font etc. There will be a core set of options with specified semantics, however, each backend may support additional options. English option names work always, the current document language may also work.

2.7 Writebacks

In the Gummi world, the converter or the Gummi editor is allowed to change the source text but only if the user doesn't object to it. There are two purposes for this: First, the section, enumeration etc numbers may be adjusted in order to have the same numbers in the document and in the PDF, and the input method substitutions may be written back to the source file.

Additionally, in case of a missing encoding information, the Gummi converter will perform an autodetect and may write the found encoding into the source.