Gummi Projekt-Spezifikation

Torsten Bronger

18. August 2007

Inhaltsverzeichnis

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 makred 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 irgnored 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 replecements 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

Kapitel 2

Referenzimplementierung

2.1 Infrastruktur

The Projekthomepage ist `http://bobcat.origo.ethz.ch/`.

Die Mailingliste kann auf `https://lists.sourceforge.net/lists/listinfo/latex-bronger-gummi` gefunden werden. Die Mailingliste ist außerdem auf Gmane gelistet unter dem Gruppennamen `gmane.text.formats.gummi`.

Der Quellcode ist als SVN-Repository realisiert. Angaben dazu finden sich auf `http://bobcat.origo.ethz.ch/wiki/development`.

Ein Beispieldokument gibt es unter `http://latex-bronger.sourceforge.net/gummi/gummi-example.html`.

2.2 Name

Der Name „Gummi“ ist vorläufig. Bessere Vorschläge sind herzlich willkommen. Der Name sollte knuffig sein und ein tierisches Maskottchen zulassen.

2.3 Projektziel

Ziel ist, in Python einen Prototypen zu entwickeln, der die Textrepräsentierung von Gummi einliest, in einen Abstract Syntax Tree (AST) umwandelt und an Backends für mindestens LaTeX (PDF) und HTML weitergibt.

Darüberhinaus könnte es sinnvoll weil recht einfach machbar sein, eine GUI mit Editor und Syntax-Highlighting drumherum zu stricken. Meine Erfahrungen mit wxPython haben gezeigt, daß das sehr wenig Aufwand wäre (< 500 Zeilen). Es soll ja keine vollständige Gummi-Entwicklungsumgebung werden, sondern nur etwas bedienungstechnischer Zucker.

2.4 Design-Philosophie des Textformats

Das Textformat und das Umwandlungsprogramm sollen folgenden Ansprüchen genügen:

- Es soll intuitiv vom Benutzer einzugeben sein.
- Es soll in der ursprünglichen Form bereits gut lesbar sein.
- Es muß eine eindeutige Syntax haben. Verstöße gegen diese Syntax führen dazu, daß das Dokument nicht (auch nicht teilweise) verarbeitet wird. Es gibt eine zweite Gruppe von „Ungereimtheiten“ in der Datei, die Warnungen auswirft.
- Es ist *nicht* wichtig, daß das Textformat leicht zu parsen ist. Hauptsache, wir Hobbyprogrammierer haben überhaupt eine Chance, diesen Parser zu schreiben. Insbesondere muß die Syntax nicht systematisch sein im Sinne von kontextfreier Grammatik o. ä. Entscheidend ist, daß sie praktisch ist.
- Logisches Markup vor visuellem Markup, aber nicht um jeden Preis. Klare Trennung von Inhalt (Gummi-Datei) und Layout (Themes), aber nicht um jeden Preis.
- Zielgruppe sind alle, die nicht-triviale Dokumente mit dem Computer eingeben wollen. Es geht nicht nur um die Cracks. LaTeX-Benutzer können uns erstmal ruhig verhöhnen für das, was wir so treiben.

Noch ein paar weitere Design-Eckpfeiler, inspiriert vom *Zen of Python* von Tim Peters:

- Mache das Häufige einfach, auf Kosten des Seltenen.
- Würde das Seltene das Häufige nicht mehr ganz so einfach machen, erkläre es zum Unmöglichen.
- Inhalt und Struktur gehören zum Dokument, aber das Layout gehört zum Theme. Beides darf nicht in derselben Datei stehen.
- Unicode ist gut, Encodings sind böse.
- Ein Dokument wird geschrieben, nicht programmiert. Präambeln à la LaTeX sind böse.
- Syntax muß dem Autor dienen, nicht dem Compiler.
- Es sollte immer genau einen offensichtlichen Weg geben, etwas zu tun.
- Außer bei Tabellen.
- Lokale Layoutanpassung ist eine schlechte Idee, globale Layoutanpassung kann eine gute Idee sein.
- Kleine systembedingte Layout-Schwächen sind verzeihlich, wenn der Gewinn bei Benutzbarkeit und Verarbeitbarkeit der Dokumente sie rechtfertigt.
- Erweiterungsmechanismen sind unvermeidbar, aber problematisch. Sie dürfen nicht dazu führen, daß Dokumente nur unter bestimmten Umständen funktionieren.
- Syntax muß viele Sprachen sprechen können. Groß-/Kleinschreibung ist wichtig.
- Gib dem Autor nicht 1001 Layout-Knöpfe zum herumspielen. Der Autor darf zu seinem Layout-Glück genötigt werden.

- Flach ist besser als verschachtelt.
- Verarbeite nur Dokumente ohne technische Mängel. Versuche Probleme automatisch zu lösen, aber nicht ohne Rückmeldung.
- Es ist erlaubt, den Quelltext automatisch zu verändern, aber nicht ohne Rückmeldung.
- Autoren lieben Themes.

2.5 Programmaufbau

Das Programm ist sowohl als Stand-Alone-Umwandlungspramm auf der Kommandozeile, als auch als Python-Modul zu gebrauchen. Es gliedert sich in drei Teile:

1. Der Parser für die Text-Repräsentation
2. Der AST, bestehend aus Klasseninstanzen
3. Die Backends, realisiert als einzelne Module

Technisch gesehen sind die ersten beiden Punkte allerdings miteinander verwoben, d. h. die Objekt-Klassen enthalten auch den Code, um sich selber zu parsen. Ich glaube allerdings nicht, daß das ein Problem ist.

2.5.1 Der Abstract Syntax Tree (AST)

Die AST ist eine Art DOM (Document Object Model), d. h. eine baumartige Struktur aus Klassenobjekten. Man kann sie sich wie eine XML-Datei vorstellen:

```
<document>
  <section>
    <paragraph>Dies ist der <emph>erste</emph> Absatz.</paragraph>
  <section>
<document>
```

Man beachte, daß „`Dies ist der `“, „`erste`“ und „ `Absatz.`“ drei sogenannte Textnodes sind, also eigene Elemente im Baum. Im AST von Gummi wird das genauso gehandhabt, d. h. es gibt Text-Elemente, die keine Kinder mehr enthalten können (und insbesondere frei von Formatierung sind; also reine Buchstabensequenzen).

Alle Klassen sind von der Klasse `Node` abgeleitet. Gemeinsam ist allen, daß sie Kinder haben können (die Liste kann selbstredend leer sein), und daß im Konstruktor die Zeilennummern des Quelltextes übergeben werden, in denen das jeweilige Element seinen Inhalt findet, den es parsen muß und sich und seine direkten Kinder erzeugen muß.

Die Element im AST sollten ruhig üppig ausgestattet sein, um den Backends das Leben möglichst einfach zu machen. So enthält die `Section`-Klasse sowohl die Schachtelungstiefe, als auch die Numerierung des aktuellen Abschnitts. Theoretisch kann das das Backend zwar selber berechnen, aber nicht nur die Autoren von Gummi-Dokumenten, sondern auch die von Backends sollten so weit wie möglich entlastet werden.

2.5.2 Präprozessor

Dem Parser ist der Präprozessor vorgeschaltet, der den Quelltext im richtigen Encoding einliest und die Ersetzungen der Input Method durchführt. Gleichzeitig führt er Buch, in welcher Zeile und in welcher Spalte dafür Buchstaben gelöscht wurden, damit man nachher die Original-Stelle wieder rekonstruieren kann.

Das ganze wird realisiert durch einen speziellen Datentyp `Excerpt` (von `unicode` abgeleitet), welcher nicht nur den Text selber sowie den Dateinamen und die originale Zeilennummer enthält, sondern auch Angaben dazu, wo escapierende Backslashs standen, und wie die orginale Zeilen- und Spaltennummer im Falle eines Fehlers wieder rekonstruiert werden kann.

2.5.3 Parser

Danach kommt dann der Parser dran. Der Parser ist eine Menge von Klassen, die die einzelnen Elemente des Textes repräsentieren, also z.B. `Document`, `Section`, `Paragraph` und `Emphasize`.

Alle Querverweise, also auch solche auf Textbausteine, Fußnoten und annotierte Hyperlinks, werden erstmal als Referenzen eingefügt. Nachdem das Dokument vollständig prozessiert wurde, werden Textbausteine, Fußnoten und annotierte Hyperlinks rekursiv aufgelöst, wobei zyklische Inklusionen natürlich einen Fehler verursachen. Aufgelöst bedeuet in diesem Fall, daß sie an die referenzierenden Stellen im AST hineinkopiert werden.

Literaturstellen und Meta-Daten werden an einer wurzelnaher Stelle des AST gesammelt.

2.5.4 Backends

Ein Backend ist für genau ein Ausgabeformat zuständig. Das Backend wird irgendwann aufgefordert, seine Callback-Funktionen in den AST zu injizieren. Das bedeutet, daß es Methoden in allen Klassen setzt bzw. von einem Vorgänger-Run u. U. eines anderen Backends überschreibt. Gummi ruft dann die Callback-Methode der Top-Node auf, die dann verschachtelt den ganzen Baum abarbeitet und so die Ausgabe erzeugt.

Das Backend kann dann auch noch weitere Arbeiten erfüllen, z. B. pdfLaTeX, xindy etc. aufrufen.

2.6 Themes

Benutzer lieben Themes. Bei Gummi sind Themes Gruppen von Backends. Ein Theme „Doktorarbeit WZL“ kann also aus einem LaTeX/PDF und HTML-Backend bestehen. Ist ein Ausgabeformat in einem Theme nicht vorhanden, gibt's ein Fallback mit einer Warnung bzw. einen Fehler, wenn ein Ausgabeformat von keinem registrierten Backend abgedeckt wird.

Für bestimmte Ausgabeformate wird es Helfer-Module geben, die Code enthalten, der für alle Themes, die dieses Ausgabeformat unterstützen, nützlich sein können. Außerdem kann ein Backend ein anderes Backend sich erstmal installieren lassen und dann noch ein paar eigene Dinge drüberbügeln. Gerade für LaTeX werden die Backends ja zu 90 % gleiche Dinge tun, im Zweifel wird nur eine andere Präambel benutzt.

Das Standard-Theme heißt „Standard“.

Man kann dem Theme – bzw. letztlich dem Backend – Optionen übergeben, genau wie im optionalen Parameter von `\includegraphics`. Damit kann man Papiergröße, Schriftart etc. einstellen. Es wird

Empfehlungen für die wichtigen Optionsnamen geben, damit nicht ein Backend die Papiergröße mit „paper size“ und ein anderes mit „papersize“ einstellen läßt. Auch hier gilt wieder: Englisch geht immer, außerdem eventuell die aktuell eingestellte Dokumentsprache.

2.7 Writebacks

Etwas, was in LaTeX verpönt wäre, ist in Gummi möglich, nämlich daß der Umwandler bzw. der Editor den Quellcode verändert. Zur Zeit passiert das an zwei Stellen, aber nur, wenn der Benutzer dies ausdrücklich wünscht:

1. Die Numerierung der Abschnitte, Gleitumgebungen, Listen etc. wird angepaßt.
2. Die Unicode-Ersetzungen aus der Input-Method werden in den Quellcode hineingeschrieben.

Man könnte sich auch vorstellen, daß beim Writeback Fußnoten, Linklisten und Literaturstellen an bessere Stellen im Quelltext verschoben werden.