Numero 27
Aprile 2019

ArsTEXnica

Rivista italiana di TEX e LATEX

GUIT

http://www.guitex.org/arstexnica/

GUIT – Gruppo Utilizzatori Italiani di TEX

ArsTEXnica è la pubblicazione ufficiale del GUIT

ArsTEXnica è la prima rivista italiana dedicata a TEX, a LATEX ed alla tipografia digitale. Lo scopo che la rivista si prefigge è quello di diventare uno dei principali canali italiani di diffusione di informazioni e conoscenze sul programma ideato quasi trent'anni fa da Donald Knuth.

Le uscite avranno, almeno inizialmente, cadenza semestrale e verranno pubblicate nei mesi di Aprile e Ottobre. In particolare, la seconda uscita dell'anno conterrà gli Atti del Convegno Annuale del GUIT, che si tiene in quel periodo.

La rivista è aperta al contributo di tutti coloro che vogliano partecipare con un proprio articolo. Questo dovrà essere inviato alla redazione di ArsTEXnica, per essere sottoposto alla valutazione di recensori. È necessario che gli autori utilizzino la classe di documento ufficiale della rivista; l'autore troverà raccomandazioni e istruzioni più dettagliate all'interno del file di esempio (`.tex`). Tutto il materiale è reperibile all'indirizzo web della rivista.

Gli articoli potranno trattare di qualsiasi argomento inerente al mondo di TEX e LATEX e non dovranno necessariamente essere indirizzati ad un pubblico esperto. In particolare tutorials, rassegne e analisi comparate di pacchetti di uso comune, studi di applicazioni reali, saranno bene accetti, così come articoli riguardanti l'interazione con altre tecnologie correlate.

Di volta in volta verrà fissato, e reso pubblico sulla pagina web, un termine di scadenza per la presentazione degli articoli da pubblicare nel numero in preparazione della rivista. Tuttavia gli articoli potranno essere inviati in qualsiasi momento e troveranno collocazione, eventualmente, nei numeri seguenti.

Chiunque, poi, volesse collaborare con la rivista a qualsiasi titolo (recensore, revisore di bozze, grafico, etc.) può contattare la redazione all'indirizzo:

arstexnica@guitex.org.

Nota sul Copyright

Associarsi a GUIT

Fornire il tuo contributo a quest'iniziativa come membro, e non solo come semplice utente, è un presupposto fondamentale per aiutare la diffusione di TEX e LATEX anche nel nostro paese. L'adesione al Gruppo prevede una quota di iscrizione annuale diversificata: 30,00 € soci ordinari, 20,00 (12,00) € studenti (junior), 75,00 € Enti e Istituzioni.

Indirizzi

Gruppo Utilizzatori Italiani di TEX
c/o Università degli Studi di Napoli Federico II
Dipartimento di Ingegneria Industriale
Via Claudio 21
80125 Napoli – Italia
`http://www.guitex.org`
guit@guitex.org

Redazione ArsTEXnica:
`http://www.guitex.org/arstexnica/`
arstexnica@guitex.org

Codice ISSN 1828-2369

15 Aprile 2019

ArsTEXnica

Rivista italiana di TEX e LATEX

Numero 27, Aprile 2019

Gruppo Utilizzatori Italiani di TEX

Editoriale

Claudio Beccari

Dopo le decisioni del Consiglio direttivo i numeri di ArsTEXnica sono pubblicati con le solite scadenze, ad aprile, per i numeri dispari, e a ottobre per i numeri pari. Ogni fascicolo diventa però disponibile per tutti in formato PDF quando viene pubblicato il numero successivo, non più dopo due numeri. Per i soci la disponibilità è immediata come sempre. La versione stampata, quella di ottobre, che contiene sia il numero dispari sia il numero pari, è disponibile per i soci del GUIT già in occasione del Meeting del GUIT.

In questo numero 27 di ArsTEXnica il lettore trova alcuni articoli originali scritti da membri del GUIT, e alcuni articoli tradotti in italiano, ma già pubblicati su *TUGboat* la cui traduzione è stata autorizzata dagli autori. Va da sé che eventuali deviazioni dal senso espresso dagli autori originali è di responsabilità dei traduttori, non degli autori, che di solito conoscono poco o per niente l'italiano, quindi non sono in grado di rivedere le traduzioni per indicare eventualmente alcune correzioni nella traduzione. Gli articoli tradotti sono già stati pubblicati su *TUGboat* o su altre riviste o sono apparsi nelle pubblicazioni di altri *User Groups* nazionali; per esempio l'articolo di Jean-Michel Hufflen è stato pubblicato in inglese su *TUGboat*, ma è stato presentato in polacco sul bollettino del GUST in occasione del BachoTEX, i cui atti, appunto, sono stati pubblicati sul *TUGboat*.

Gli argomenti sono vari e disparati

Io stesso ho scritto qualcosa che è un po' più di un tutorial in merito alla gestione delle lingue e ai famosi *patterns*, a cui si fa spesso riferimento, ma che pochi sanno davvero che cosa siano. L'articolo vorrebbe essere di carattere generale, ma ovviamente ho parlato solo delle lingue e dei pattern di cui ho qualche competenza; nello stesso tempo mi è piaciuto indicare lo schema da seguire per creare una serie di pattern per una lingua locale poco conosciuta, il *cimbro*, perché ho cercato di immedesimarmi in un linguista che stia svolgendo uno studio universitario su una lingua rara e in via di estinzione.

Ho scritto anche questo breve articolo sul disegno della bandiera europea; ho preso spunto dall'articolo di Peter Flynn, pubblicato su *TUGboat*, ma, diversamente da lui, ho cercato di fare uso solo delle funzionalità di LATEX, in particolare del suo ambiente nativo `picture` esteso con il pacchetto `pict2e` ed ulteriormente esteso con il pacchetto `curve2e`. Infatti le dodici stelline che formano la bandiera europea sono inscritte in un pentagono; questa figura geometrica ha stretti legami con la sezione aurea, cosicché non è difficile far eseguire i calcoli necessari al programma di composizione stesso, ottenendo così un codice che non dipende in nessun modo dai font usati nel documento. Ne è venuto fuori un codice più semplice, ma il risultato, per quel che si può giudicare ad occhio, sembra lo stesso. O meglio La mia soluzione è indipendente dai font usati e non è legata a nessuna delle idiosincrasie che il sistema TEX ha nel gestire i font, sia pure quelli vettoriali, alcuni dei quali non sono selezionabili per qualsiasi corpo si desideri.

Roberto Giacomelli ci mette al corrente dei suoi progressi nel gestire le opzioni dei file di classe o dei pacchetti di estensione mediante l'uso del linguaggio Lua. Come è noto agli assidui lettori di ArsTEXnica, Giacomelli si occupa da tempo di usare LuaTEX e LuaLATEX per comporre la documentazione formale del suo lavoro di professionista; egli, infatti, deve poter includere nei suoi documenti informazioni che sono raccolte in file di diverse origini, codificati in diversi modi, impaginati con software diversi, eccetera. In questo processo egli ha bisogno di richiamare comandi o interi pacchetti che accettano delle opzioni; egli ritiene che l'uso del software Lua sia particolarmente adatto a questo scopo e presenta lo stato dell'arte a cui è arrivato. Egli, primo fra i tanti autori che hanno collaborato ad ArsTEXnica, fornisce anche del software che i lettori si possono scaricare per sperimentare la funzionalità del suo codice Lua.

Tommaso Gordini, da musicista qual è, si è preso l'impegno di tradurre in italiano l'articolo di Jean-Michel Hufflen, già pubblicato su *TUGboat* e sugli atti del BachoTeX. Quanto egli scrive sulla notazione gregoriana è un punto di vista mediato fra quello di chi usa programmi di vario genere per tipocomporre gli spartiti e quello di chi conosce la musica (in cui è diplomato). Quanto scrive va quindi visto con questi occhi, non con quelli dello storico o del musicologo, che probabilmente non condivide la stessa visione, o addirittura la considera errata. Lo scopo di presentare questo articolo tradotto non è quello di pubblicare un punto di vista musicologico e storiografico della notazione gregoriana, ma serve per portare a conoscenza dei lettori il genere di articoli che vengono pubblicati sulla rivista internazionale *TUGboat*

Nello stesso spirito, Tommaso Gordini ha tradotto in italiano l'articolo di Joseph Wright, che parla del pacchetto `siunitx` dalla prima versione, fino alla terza, ancora in via di implementazione; al

momento di scrivere queste note, della versione 3 esiste solo una variante *alpha*, che chiunque può scaricare e testare ricavandola dal sito di Wright. Il vantaggio della nuova versione dovrebbe essere quello di eliminare certe inconsistenze della versione precedente e di aggiungere funzionalità nuove; nel fare questo, Wright ha deciso di riscrivere tutto in linguaggio L3.

Io ho tradotto anche l'articolo di Frank Mittelbach sull'evoluzione di LaTeX. Questo linguaggio è restato retrocompatibile nei limiti del possibile, con la versione precedente fino alla versione del 2017. Poi ha cominciato ad arricchirsi di nuove funzionalità, precedentemente confinate in pacchetti di estensione che aggiungevano poco o quasi niente al codice, ma nello stesso tempo rendevano la nuova versione incompatibile con il recente passato, pertanto il nucleo (kernel) di LaTeX è stato dotato di una serie di comandi per ritornare al passato, cioè per ripristinare provvisoriamente le funzionalità che il kernel aveva prima di una certa data scelta dall'utente. Con questo articolo Mittelbach informa i lettori che queste funzionalità di ritorno al passato sono disponibili anche per gli autori dei numerosi pacchetti che accompagnano LaTeX, ma che negli anni hanno subito upgrade tali da modificare le funzionalità delle precedenti versioni.

Ringrazio tutti coloro che hanno collaborato per la realizzazione di questo numero di ArsTEXnica, la Redazione, il Consiglio Scientifico, i numerosi revisori editoriali. Sta a voi lettori giudicare se quanto esposto in questo numero di ArsTEXnica è di livello corrispondente alle vostre aspettative.

▷ Claudio Beccari
Professore emerito
Politecnico di Torino
`claudio dot beccari at gmail dot com`

Language management and patterns for line breaking

Claudio Beccari

Abstract

Language management is supported by different files according to the language manager babel or polyglossia; they are similar to a certain extent, but differ in the way they handle the language patterns. There are also small differences when using XeLaTeX compared to LuaLaTeX. Obviously patterns are different from language to language, but there are also some languages with variants. Therefore the language-supporting compiler-structure has to manage a variety of situations.

Sommario

La gestione delle lingue è sostenuta da file differenti a seconda che si usi il gestore babel o polyglossia; gli specifici file di ogni lingua sono abbastanza simili ma differiscono specialmente nel modo di gestire i pattern per la cesura in fin di riga. Ci sono anche delle piccole differenze se si usa XeLaTeX o LuaLaTeX. I pattern, ovviamente, sono diversi da lingua a lingua, ma ci sono anche lingue che hanno anche delle varianti. Pertanto il meccanismo di sostegno linguistico dei compilatori deve essere in grado di gestire una varietà di situazioni.

1 Introduction

This paper is half way between a technical document and a tutorial. Some parts are quite professional, and others are for any LaTeX user.

Any LaTeX user knows very well that an important part of the preamble of any source `.tex` main file is dedicated to the language management of the only language used in the document, or to the specification of the various languages that are used in the document; very often, depending on the language(s), it is necessary to specify the font collections to be used in order to use different alphabets. The only users who do not need to specify anything used to be the American ones, because the default language is the American variety of the English language; of course, if they have to cite any stretch of text in a language different from English, they have to do as anybody else. The British, Australian, Canadian, New Zealander English authors have to specify their language variety because their varieties require a little different handling compared to the American one.

The situation is quite different when the document is typeset with pdfLaTeX, compared to what is possible to do with XeLaTeX and LuaLaTeX. In facts pdfLaTeX can use only the babel package, while when XeLaTeX or LuaLaTeX are used it is possible to require either the polyglossia or the babel package.

The babel package, (Braams e Bezos, 2018) has its origin in the early nineties when Johannes Braams invented its mechanism and became its maintainer. After some twenty years the maintenance was taken over by Javier Bezos who introduced many extensions and functionalities.

The polyglossia package (Charette e Reutenauer, 2018) was created with the advent of XeLaTeX and was refined a little bit when LuaLaTeX was stabilised. Now polyglossia can be considered quite stable, and any addition or modification is mostly related to the numerous specific language files.

babel and polyglossia handle a large number of languages, more than eighty, although for some rare ones there might be some differences. According to my experience, the language handling by these two packages offers some pros and cons, so that it is difficult to say which is the most performant. Personally I prefer polyglossia, but I use pdfLaTeX quite often, therefore, possibly, I use babel more often than polyglossia. The differences are described in detail in the following sections.

Both handlers have to resort to language specific files in order to define (*a*) the infix words necessary to typeset language dependent headings, the date, and some other language specific structures; and (*b*) they have to select the proper *hyphenation patterns* in order to let the typesetting engine split in the best way the input strings so as to create perfectly justified paragraphs, with the least number of hyphenated line breaks and, with the support of the microtype package (Schlicht, 2018), the most homogeneous inter word spaces.

The handling of patterns is different for the three LaTeX based typesetting engines. The pdfLaTeX and XeLaTeX typesetting engines require that the patterns are preloaded into their format files, while LuaLaTeX resorts to a different mechanism so that the format file knows only the list of all the available pattern files, but, except for American English, none is preloaded, and the typesetting engine loads at run time the patterns it needs for the only languages that are specified for the document.

In this paper we show how to create a *language description file* for a generic language, and how to

create a *pattern file* for that language. Moreover we shortly describe the service files through which the typesetting engines are specified what to preload, where to find it, what to replace, and so on.

Language description files are pretty easy to create; on the opposite pattern files are pretty difficult, and different languages require different approaches. In the sequel I will cite some of my own work without the purpose of showing an achievement of mine, but to make examples of what and how to do it in order to have the necessary structure for language handling.

I have created pattern files for more than a dozen languages, mainly romance ones, and I followed the same method that, according to my experience and my capabilities, was suitable for all of them. For example I created the patterns for Latin in these varieties: modern (as it is being used in Italy), medieval, classical, ecclesiastical, and liturgical. At the moment these patterns are supported by another team composed of several people with different linguistic backgrounds and well supported by Latinist scholars; they are doing a marvellous work; they started from my patterns, and they are modifying and extending them using another very different approach that will lead them to a much better final result.

2 The required files

We have seen that we need three file levels for handling any language.

1. The language description language files; there are two kinds of such files, because there are two different packages that are used to handle a given language.
   (a) The ⟨*language*⟩`.ldf` to be used with `babel` is generally matched with a documentation file named ⟨*language*⟩`.pdf` resident in a `doc/generic/babel-`⟨*language*⟩ folder. The latter file describes the features and the functionalities associated to the specific ⟨*language*⟩, such as user settings, commands and/or shortcuts to perform certain actions when a specific text is typeset in that ⟨*language*⟩. The user can access this documentation by means of the terminal command `texdoc babel-`⟨*language*⟩ command.
   (b) The file `gloss-`⟨*language*⟩`.ldf` to be used with `polyglossia`; it is not matched with a documentation file; actually a short text concerning this ⟨*language*⟩ is included into the `polyglossia` documentation (Charette e Reutenauer, 2018); the user can access this documentation with the terminal command `texdoc polyglossia`.
2. The pattern files have different extensions: `.tex` for use with pdfLaTeX and XeLaTeX; `.txt` for use with LuaLaTeX. Their names are of the form `hyph-`⟨*language ISO code*⟩⟨*-modifiers*⟩. The ⟨*language ISO code*⟩ is a two or three letter standard name established by the ISO regulations, such as `en` for English, `it` for Italian, `la` for Latin, `kmr` for Kurmanji (Northern Kurdish), and so on. The ⟨*modifiers*⟩ are short strings that specify a specific variety of the ⟨*language*⟩: for example US English file has a full name of `hyph-en-us` while British English one has a full name of `hyph-en-br`; other languages may have longer modifier strings, but the idea is the same. The pattern file may be accompanied by other specific files; for example all languages that use the apostrophe as an elision marker (for example, French, Italian, Catalan,...) as well as a single or (ligated) double closed quotation marks have another matching file named `hyph-quote-`⟨*language ISO code*⟩ containing specific patterns to handle such writing elements.
3. The service macros are basically the ones that list the files described above; these service macros are used only when a format file has to be (re)created. These files have names such as `languages.`⟨*specific extension*⟩ where the ⟨*specific extension*⟩ is different depending on which typesetting engine the format file is created for. These files are maintained by the TeX-hyphen Working Group that takes care of every detail (including licences, since the pattern files are used also by other free or commercial external programs).

The user should not care about these "official" files; s/he needs these latter files only for a local installation that s/he uses for testing the other files s/he created; eventually s/he sends his/her language description and the pattern files to the TeX-hyphen Working Group that, in turn, takes care of their conformance with the TUG regulations in this matter and possibly adds them to the official TeX system distribution.

A partial description of such three kinds of files appears in both the TeX Live documentation (Berry, 2018a) and that of the `tlmgr` program, (Berry, 2018b) that handles the whole management of the TeX Live distribution. For MiKTeX there might be different ways of managing these language details, but in general the end user does not need to access such functionalities for his/her everyday typesetting chores.

3 Syllabification vs. hyphenation

The hyphenation patterns were invented by Frank Liang at the very beginning of the existence of TeX and were described in his doctoral thesis defended

a little later at Stanford University (Liang, 1983). At that time it was still TEX 78, a beta version of the future TEX. But he was essentially concerned with American English hyphenation because at that time this was the only language the TEX program could handle. The idea was a winning one; it was extended to other languages handled by the TEX system; nowadays it has been ported to a number of widespread free and commercial programs; just to name a couple, the Open/Libre Office word processors and the InDesign program.

But patterns deal with hyphenation, not with syllabification; the difference is essential; linguists deal with syllabification with certain criteria; phonologists deal with syllabification with other criteria. Linguistic syllabification deals with grammar rules; phonetic syllabification deals with the pronunciation of a specific language. Both approaches do no directly concern typography. Typography is mostly concerned in line breaking and with splitting words in a way that the reader is not confused while reading his/her text. Of course hyphenation should not infringe grammatical rules, but it is subject to stricter needs that force using as hyphenation break points only a subset of the grammatical ones.

This difference sometimes is confusing anybody who has to deal with line breaking. But it must be clear that the purpose of typography is not to take over the expertise of a linguist; typography main purpose is the reader's comfort in reading.

Actually even the grammar rules are not so strict; on one side every language contains words that are exceptions to the rules; on the other side there are certain texts that presume a different way of syllabification. Some of these texts are poems: the verses must follow a certain syllabic rhythm so that sometimes certain diphthongs are split so as to become hiathus. Other texts are the scripts that a speaker should read in a formal way; the speaker certainly emphasises certain words again by pronouncing some diphthongs as if they were hiathus. These particular situations are not taken into account by hyphenation.

Another point: hyphenation should not take place when the break point is too close to the start or the end of a word; the TEX system typesetting engines at the moment take into consideration such minimum distances by means of two numerical parameters `\lefthyphenmin` and `\righthyphenmin`. For most languages such parameters are preset to the values 2 and 3; these numbers measure the distance from the start and the end of a word in terms of "number of characters"; the TEX-hyphen Working Group is studying how to exclude from this count the self combining characters available with OpenType fonts; actually the `luatex` engine is already capable of doing this "correction".

The TEX-hyphen Working Group is also studying a way to assign different penalties to every possible hyphen point so as to reduce the number of short word fragments; short words just over the minimum length established by the `...hyphenmin` parameters may have just one hyphen point to use, but longer words have more hyphen points and their choice may be subjected to the optimisation algorithm (based on penalties) that the TEX system typesetting engines use to process paragraphs. Therefore in the future there will be more new functionalities associated to hyphenation.

4 The patterns

The patterns are short letter sequences that represent word fragments; these fragments may be of any length, from one letter up. The letters are interspersed with digits from 0 to 9. The value 0 is optional, because it is assumed when no digit is explicitly shown. Any position occupied by an even digit is where break points are forbidden, while on the opposite a position where an odd digit is present implies a possible break point; when in different patterns the same two letters are separated by different digits the highest value prevails in deciding if it is a legal break point or no break is allowed.

4.1 Word strings

The above introduction to patterns is a simplified one. It is better to specify in a better way how the typesetting engines deal with words to hyphenate.

The typesetting engine receives from previous steps of text processing each paragraph in the form of a single long string of tokens; the aim of the paragraph processing module is to divide such long line into lines of equal length, except possibly the last one. The string to process is made of words, inline math, penalties, kernings, punctuation marks, footnotes, macros that shall be expanded when the paragraph is shipped to the output file, and other such objects.

Penalties are used for both line breaking or page breaking; kerns are used to set adjacent characters in a proper way, but they might influence line breaking; the same is true for ligatures, such as the "f" ones, that might be interrupted by a hyphen in case a line break occurs at that point.

Other macros, such as `\footnote`, may influence hyphenation in the sense that they interfere with the process of finding a word end. But it is not up to the program to "correct" these situations; it is up to the user to provide some way of avoiding them; the simplest one is to put a zero width glob of glue just before the `\footnote` macro (or before any similar offending macro), so that this glue lets the program identify the word end.

In any case the program must identify each word for possible hyphenation; the program considers a "word" any string of consecutive characters that have a positive value of their `\lccode`. In general

any alphabetic letter has already been classified as being of category 11 while any non alphabetic character has been classified as being of category 12.

Characters of category 11 have two associated codes: the "lowercase" and the "uppercase" codes; respectively assigned with the native commands `\lccode`⟨*character*⟩=⟨*value*⟩ and `\uccode`⟨*character*⟩=⟨*value*⟩. Here ⟨*character*⟩ means the internal numerical code used by the engine to address a specific character in a given font; ⟨*value*⟩ is another numerical value to address a character in the same font. Such associations are mostly useful for lower/upper case transformations. Generally speaking users don't set and/or reset lower and upper case codes; they also generally do not know the glyph addresses in a given encoding; therefore a special grammar is used in order to access to such addresses. As an example let us see the situation with characters "a" and "A":

```
\lccode‘\a=‘\a \uccode‘\a=‘\A
\lccode‘\A=‘\a \uccode‘\A=‘\A
```

This means that when lowercasing "a" the same glyph is used, while when uppercasing it, "A" is used; the opposite takes place with "A".

But for what concerns hyphenation the program considers as valid symbols also non alphabetic signs provided they have a positive lower case code. Therefore for languages that use the elision apostrophe, such glyph must receive a positive `\lccode`; the language description file should therefore set this value but it has to reset it to zero, upon changing language. Assigning a category code to glyphs is generally done by the LaTeX kernel and/or by the encoding packages used for input and/or output and/or by the classes and packages that are being used to typeset the document; this action takes place also when a specific language is being typeset with an alphabet different from the Latin one.

All patterns are written with lowercase letters; this implies that in order to find the possible break points, the typesetting engine must deal only with the lowercase codes of the glyphs forming the words. Since other glyphs in a string do not have lowercase positive codes, the engine is capable of isolating single words formed by glyphs of any category, but with positive lowercase codes.

4.2 How patterns work

Let us explain how patterns work with a simple English word: "electricity". The typesetting engine module searches in the English pattern-list all the patterns that can be found into the given word, at its beginning, at its end, and anywhere in the middle. Let us show the various patterns found in the diagram of table 1; for simplicity the first line contains the letter string that the typesetting engine recognises as a word. Here we neglect some details, but the description is correct.

The `hyph-en-us.tex` file contains the original Knuthian 4938 hyphenation patterns for American English. In table 1 line 1 contains the original word; the dots before and after the word are just place holders to show the word boundaries; such dots are used also in the pattern files to mark the pattern strings that may be found at the beginning or at the end of a word. Line 2 contains the default digit 0 between every couple of letters; if no patterns were found in the pattern file, these zeros would imply that there are no legal break points in the word. The pattern file contains only the following patterns `2c1t`, `2c1it`, `2ici`, `4rici` that are made of substrings that can be found in the given word; they are copied in the table lines 3, 4, 5, and 6 with their letters in column with the letters of the original word and their digits in the position they are in the patterns. Eventually line 7 contain the letters in their columns and the highest digit in each numerical column. The only odd digits occur between the couples `ct` and `ci`, therefore the word may be hyphenated as such: *elec-tric-ity*. The grammar might allow other break points, but the preset `...hyphenmin` parameters are set to 2 and 3, therefore the first word fragment must be at least two characters long and the last one must be at least three characters long.

4.3 Grammatical rules

Not all languages have grammatical rules for syllabification and therefore for hyphenation; as we said, even if grammatical rules do exist, every language has several words that form a set of exceptions to "the rule".

In general a syllable contains one vowel, or one diphthong, or one triphthong. The set of vowels is specific of every language; we are mostly used to the Latin script, and we assume the vowels are in the set "aeiou"; some other languages have larger sets, such as "aäåeioöuy" or "aàáâeèéêioôu", and so on. Some languages contain among their vowels also some characters other languages consider consonants: for example in some Slavic grammars the letter "r" is labelled as a vowel; matter of fact how could anybody pronounce "Trst" (the city of Trieste) or "smrt" (death) if the letter "r" did not contain some "voice" in itself?

Diphthongs and triphthongs are made of two or three vowels, respectively; for diphthongs the couple must contain an unstressed semivowel "i", or "u", or "j", or "y", or "w", the last three of which are considered semivowels only in certain languages. Triphthongs must contain two unstressed semivowels either close to one another or sandwiching the third vowel.

One consonant or a cluster of consonants may be part of a syllable when they precede the vocalic part; when these clusters start with specific consonants the latter might remain appended to the vocalic part of the preceding syllable; they gener-

Table 1: The process of analysing the patterns that are found in the word "electricity" and of finding the legal break points

.	e		l		e		c		t		r		i		c		i		t		y	.
.	e	0	l	0	e	0	c	0	t	0	r	0	i	0	c	0	i	0	t	0	y	.
						2	c	1	t													
												2	i	0	c	0	i					
														2	c	1	i	0	t			
										4	r	0	i	0	c	0	i					
.	e	0	l	0	e	2	c	1	t	4	r	2	i	2	c	1	i	0	t	0	y	.

ally are "l, r", and "m, n", but such consonants that remain appended to the previous syllable are specific of every language; for example, in French the consonant "s" remains appended to the preceding syllable, while in Italian it remains at the start of the current syllable. The consonants at the end of a word remain appended the previous vocalic part, as well as all the initial word consonants remain with the vocalic part they precede, irrespective if they start with the special consonants mentioned above. In some languages reinforced (double) consonants are split between syllables.

Compound words may violate the above rules, in the sense that the grammar of the language includes a further rule that a legal break point is located also at the junction of two component words. Some languages, on the opposite, treat compound words as if they were a single word. Italian is one of such languages but the Italian UNI regulation dealing with hyphenation, (UNI 6461, 1969), does not forbid hyphenation at a compound word boundary; this is why a word such as "transatlantico" (compound with "trans" and "atlantico") may be divided either way: *tran-sat-lan-ti-co* and *trans-at-lan-ti-co*. While creating patterns one of these choices must be coherently selected in order to treat all compound words in the same way.

In some languages the break point at a compound word boundary might require also a change of spelling: for example the German word "Bettuch"[1] (bed sheet) gets hyphenated in the form "Bett-tuch". These situations are not dealt with by patterns and the language description files must provide shorthands to handle them.

4.4 Syllabification dictionaries

Some languages are so complicated in their spelling and/or syllabification rules that it is almost impossible to define any reasonably sized set of rules; it is therefore impossible to create a reliable set of patterns so that they provide the greatest part of the work, without recurse to large exception files that list the words that do not follow the rules.

One such language is English, at least the American variety; we are aware that British English has been subjected to a reform in syllabification but we haven't seen yet any pattern file that implements the new standard.

In any case such languages must be treated with the `patgen` program (Liang e Breitenlohner, 1991), that is part of the TEX Live and MiKTEX distributions, together with some of its siblings; the original `patgen` was created at the very beginning of the TEX system on purpose, because the American variety was so complicated that it was impossible to derive from the grammar rules the hyphenation patterns.

The `patgen` solution consists in processing a large list of already hyphenated words in order to extract from them a pattern set that *minimises the probability* of producing wrong break points; may be such set might prove to be insufficient to hyphenate correctly almost any word in the list, and therefore in the language, but it is better than nothing. The original Knuthian patterns generated with `patgen` had an error rate of about 10%; the list of exceptions contains some thousands of words, so that the error rate diminishes significantly but it is not zero. In the past years the list of exceptions was added to the format file, so that nowadays it is difficult but not impossible to get wrong break points.

One detail that makes it impossible to correctly hyphenate English words, is that certain homographs get hyphenated in a different way according to their pronunciation; take for example "he analyses" and "the analyses", where the first "analyses" string gets hyphenated as *a-na-lys-es* while the second one becomes "a-nal-y-ses". Actually these break points are those that might be obtained if both `...hyphenmins` were set to 1; actually they are set by default to 2 and 3 respectively, so that the actual break points are just *ana-lyses* and *analy-ses*. The typesetting engines can "read" the text but they cannot "pronounce" the words; they work on spelling, not on sound.

It should be obvious that the quality of the patterns depends very much on the number of words contained in the input list to `patgen`; Frank Liang started with a list of 25 000 words; eventually the actual American English patterns are built with a list of 100 000 words, but the problem shown in the previous paragraph cannot be solved with patterns or exception lists; the problem could be

1. This is the "old spelling" prior to 1996; with the "new spelling" it is Betttuch.

solved if the typesetting engines contained also a *semantic* analysis module for each language, but this would be too heavy on the typesetting process.

Provided one has available a hyphenated word list, or is keen to manually hyphenate several myriad words (remember: one myriad contains 10 000 items) it is difficult to create language patterns for all languages, even those that have clear grammar rules. We are not discouraging the use of `patgen`; we simple underline the difficulties implied with its use.

4.5 Creating patterns by hand

When clear grammar rules are available, hyphenation patterns may be manually created. I followed this approach for more than a dozen languages; I compared my patterns with those that I could derive by using `patgen` and generally I found that I was able with fewer patterns to get a better success percentage (close to 100%) compared to that obtainable with `patgen`. I would not generalise this conclusion, but this is what I found out in my experiments. I agree that with languages such as English the `patgen` approach would be the only affordable.

I created pattern files for a dozen languages that had all clear and simple rules for syllabification; I generally created the patterns so that if the `...hyphenmin` parameters are both set to 1, the hyphenated words are always correct; well, some people might argue that this is an overstatement; the point is that I do not consider wrong a word division that does not contain *all* possible break points. The reasoning is that the reader is uncomfortable if the break point takes place just after or just before a vowel that might behave as a semivowel. In Italian for example the word "paura" (fear) is grammatically syllabified as *pa-u-ra* but I managed that patterns divide it as *pau-ra*; in this word the group "au" is a hiatus, because the vowel "u" is tonic and therefore the vowels don't form a descending diphthong so that they may be grammatically divided; but by doing so, the reader is uncomfortable. When accents are used such situations should not take place, but even so the reader is uncomfortable.

How to create patterns: the simplest way is to follow a procedure used also by the TEX-hyphen Team; name with capital letters sets of letters that have the same behaviour for what concerns hyphenation; for example O denoting the open or semi open vowels, C to denote the closed vowels that may form diphthongs and triphthongs; K to denote "normal" consonants; L to denote those consonants that may be appended to the preceding syllable; and so on.

Write down the grammatical rules in terms of these classes, i.e. create "macro patterns" in terms of these classes; then use some external software to expand such macro patterns to create the patterns with actual vowels and consonants. You get a large number of patterns many of them might be superfluous, because some combinations do not exist in a specific language; nevertheless, if a searchable dictionary file is available for that language, try to find specific words that contain unusual letter combinations. Most of the time such words can be found and one realises that most of those unusual patterns were not useless. In any case they might not be part of the normal words of a language because they are derivatives forms from a foreign root; therefore some of those unusual patterns are not to be thrown out.

4.5.1 The Italian example

These foreign roots may give some problems to any language; grammars of Italian state that any consonant cluster belongs to the syllable that contains the following vocalic group if and only if there is another Italian word that starts with that consonant cluster. This works fine most of the time, but it is an imprecise rule: it should be completed with the clause of "... Italian word, *not of Greek etymology*, that starts with...". The difficulty for the user is to know the etymology of every word; this is why the UNI regulation explicitly specifies that the groups `bd, gm, ps, pn, tm,…`, must be divided between two adjacent syllables. This of course excludes such groups at the very beginning of specific words (of Greek etymology) but interferes with compound words containing such words; "psicologia" is definitely of Greek etymology, but what do you do with the Italian word "parapsicologia"? It is necessary to create a special pattern for words containing the string `psic`; what to do with "pneuma" and "apnea"? It is necessary to do the same[2].

This is why the intelligence of the human being comes into play to solve these special situations that require manual intervention. The same intervention is necessary to handle Italian words that derive from foreign names such as "newyorkese", "maxwelliano", "wagneriano", "massmediologo", "leishmaniosi", "lewisite", "wahhabbita", and the list might go on for a long list of other such words.

Just to remain with Italian as an example, the few patterns needed to correctly hyphenate all Italian words without foreign roots (but also many such mixed etymology words) are shown in code listing 2.

It is worth noting what follows.

1. The dot that precedes or follows a given pattern marks that it may be used only at the beginning or, respectively, at the end of a word.

2. Actually in "apnea" the first "a" is a prefix, therefore some dictionaries prefer to use the compound word division.

Code 2: The 355 patterns needed to hyphenate the Italian language

```
\patterns{% Pattern start
.a3p2n
.anti1 .anti3m2n
.bio1
.ca4p3s .circu2m1 contro1
.di2s3cine
.e2x1eu
.fran2k3 .free3
.li3p2sa
.narco1
.opto1 .orto3p2 .para1
.ph2l .ph2r
.poli3p2 .pre1 .p2s
.re1i2scr
.sha2re3
.tran2s3c .tran2s3d .tran2s3l .tran2s3n .tran2s3p .tran2s3r .tran2s3t
.su2b3lu .su2b3r
.wa2g3n .wel2t1
2'2
a1ia a1ie a1io a1iu a1uo a1ya
2at.
e1iu e2w
o1ia o1ie o1io o1iu
1b 2bb 2bc 2bd
2bf 2bm 2bn 2bp 2bs 2bt 2bv b2l b2r 2b. 2b'
1c 2cb 2cc 2cd 2cf 2ck 2cm 2cn 2cq 2cs 2ct 2cz
  2chh c2h 2ch. 2ch'. 2ch''. 2chb ch2r 2chn
  c2l c2r 2c. 2c' .c2
1d 2db 2dd 2dg 2dl 2dm 2dn 2dp d2r 2ds 2dt 2dv 2dw 2d. 2d' .d2
1f 2fb 2fg 2ff 2fn f2l f2r 2fs 2ft 2f. 2f'
1g 2gb 2gd 2gf 2gg g2h g2l 2gm g2n 2gp g2r 2gs 2gt 2gv 2gw
  2gz 2gh2t 2g. 2g'
.h2 1h 2hb 2hd 2hh hi3p2n h2l 2hm 2hn 2hr 2hv 2h. 2h'
.j2 1j 2j. 2j'
.k2 1k 2kg 2kf k2h 2kk k2l 2km k2r 2ks 2kt 2k. 2k'
1l 2lb 2lc 2ld 2l3f2 2lg l2h l2j 2lk 2ll 2lm 2ln 2lp 2lq
  2lr 2ls 2lt 2lv 2lw 2lz 2l. 2l'. 2l''
1m 2mb 2mc 2mf 2ml 2mm 2mn 2mp 2mq 2mr 2ms 2mt
  2mv 2mw 2m.2m'
1n 2nb 2nc 2nd 2nf 2ng 2nk 2nl 2nm 2nn 2np 2nq 2nr 2ns
  n2s3fer 2nt 2nv 2nz n2g3n 2nheit 2n. 2n'
1p 2pd p2h p2l 2pn 3p2ne 2pp p2r 2ps 3p2sic 2pt 2pz 2p. 2p'
1q 2qq 2q. 2q'
1r 2rb 2rc 2rd 2rf r2h 2rg 2rk 2rl 2rm 2rn 2rp 2rq 2rr 2rs
  2rt r2t2s3 2rv 2rx 2rw 2rz 2r. 2r'
1s2 2shm 2sh. 2sh' 2s3s s4s3m 2s3p2n 2stb 2stc 2std 2stf
  2stg 2stm 2stn 2stp 2sts 2stt 2stv 2sz 4s. 4s'. 4s''
.t2 1t 2tb 2tc 2td 2tf 2tg t2h 2th. t2l 2tm 2tn 2tp
  t2r t2s 3t2sch 2tt t2t3s 2tv 2tw t2z 2tzk tz2s 2t. 2t'. 2t''
1v 2vc v2l v2r 2vv 2v. 2v'. 2v''
1w w2h wa2r 2w1y 2w. 2w'
1x 2xb 2xc 2xf 2xh 2xm 2xp 2xt 2xw 2x. 2x'
y1ou y1i 1z
2zb 2zd 2zl 2zn 2zp 2zt 2zs 2zv 2zz 2z. 2z'. 2z'' .z2
}% Pattern end
```

2. The patterns that involve the letters "j, k, w, x, y" involve only words with foreign roots, because in strict Italian they never appear[3].
3. The numerous patterns involving the "h" line deal mostly with foreign root words, because in Italian the "h" glyph is used more as a diacritical mark rather than a pronunceable sound.
4. The fist part of the pattern set concerns only etymological hyphenation of a few prefixes; the other prefixed words may be treated as regular words.
5. The reader can see that, except for a few triphthongs, most patterns do not contain vowels; in this way patterns containing accented vowels are unnecessary.
6. The reader also may notice that the above described scheme of combining the couples of consonants emerges from the pattern set; the patterns containing couples that never appear have been deleted.
7. At the same time the reader may see the difference of the liquid and nasal consonants "l, m, n, r" that may remain appended to the preceding syllable, opposite to the behaviour of the other consonants.
8. Also the "s" unique property at the beginning of a consonant cluster, shows that it remains attached to the cluster, instead of being separated as it happens in all other romance languages.
9. All consonants may be reinforced by doubling; in this case the first occurrence remains attached to the preceding syllable, at least in words with Latin etymology.

4.5.2 The Latin case

Latin is an emblematic situation where there are several variants: modern, medieval, classical, ecclesiastic, liturgical, with prosodic marks, and so on. They differ in the alphabets they use and in the hyphenation rules; infix words may also be different or have a different spelling according to the specific alphabet.

Let us see the main differences.

Modern Latin uses the Latin alphabet; no surprise; but here with Latin alphabet the whole 26 letters alphabet is meant; without the Latin ligatures æ and œ. No accents are used; modern Latin hyphenation rules in Italy are very similar to the Italian ones, but in Germany they use slightly different rules. It is mainly used for schoolbooks, grammars and literature collections, in high schools and partially in universities. But most important, it is the official language of the Holy See. It is not the official language of the Vatican City State, because its official language is Italian. The Pope is the monarch of both entities, but the formal writings signed by the Pope as the Head of the Roman Catholic Christianity are written with this Latin variety.

Modern Latin with prosodic marks uses the macron and the breve diacritical marks and is used in school grammars and dictionaries; the length of each vowel is marked as "long" with the macron, or "short" with the breve; it helps students when reading the hexametres and pentametres of the Latin poems.

Medieval Latin uses a shorter alphabet:

```
aæbcdefghiklmnoœpqrstuyz
AÆBCDEFGHIKLMNOŒPQRSTVYZ
```

where the Latin ligatures æ and œ more often than not are contracted to their pronunciation "e"; the use of "I" and "Y" is generally inconsistent; I have seen the spellings Italia and Ytalia; Iesus and Yesus even in the same text. The only sign "u" was used also for the consonant "v", and viceversa for the capital variant. Actually the differentiation between the two signs, not only in Latin, took place by the end of the XII century. It does not require much, except changing Novembris to Nouembris, and adding few patterns to the same pattern file of modern Latin; such peculiar patterns involve "u" just where it plays the role of a "v".

Classical Latin uses "u" and "V" the same way as medieval Latin, but it does not use the Latin ligatures "æ" and "œ". Hyphenation is completely different from the modern and medieval varieties; prefixes and suffixes are provided breakpoints in terms of etymology, instead of grammatical rules only; "i" and "u" never play the role of semivowels, therefore there are no diphthongs; even "ae" and "oe" are pronounced separately by classical latinists, but are not divided. The guttural consonants "c" and "g" remain guttural and never become palatal sounds; they never form digraphs or trigraphs as in other variants of Latin. Moreover all words with a Greek etymology are hyphenated with the Greek rules. As one can imagine, classical Latin requires a completely different management, especially with hyphenation, because the inflexional nature of the language requires not only a strict relationship with etymology but also the analysis of all the inflected forms of verbs, nouns, and adjectives. The French team that is working on this problem is doing a beautiful work thanks, also, to the support of latinist scholars.

The fact that classical Latin does not use the medieval æ and œ ligatures renders the prob-

3. Actually the letters "k" and "x" may be part of words of Greek etymology; "j" is an old fashioned spelling for the semivowel "i"; the five of them appear in neologisms of foreign origin and in proper names.

lem of homographs quite more complicated; for example the inflected form *aeris* comes from the nouns *aer* (air) and *aes* (bronze); the first requires to be hyphenated as *a-e-ris*, the second as *ae-ris*; if the second were written with the diphthong ligature *æs, æris* there would not be homographs any more. In ecclesiastic Latin, on the opposite, the first one would be spelled *aer,..., áeris* and, again there would not be homography any more.

Ecclesiastic Latin is used in clergy texts, such as missals and breviaries. It is mainly modern Latin with (acute) accents; such accents are used to mark the "rhythm" in word pronunciation. It is done in such a way that clergy of every national mother language background and used to a different rhythmic approach may pray together pronouncing the same words with the same stress. The accents are missing from all words where the stress falls on the penultimate syllable; nevertheless in view of uniform pronunciation, the clergy with different mother language backgrounds may have difficulties to perceive the difference between diphthongs and hiathus, so that if the penultimate syllable contains a group of vowels the tonic one is marked with the accent.

There are very little differences in the infix words, but accents introduce many complications in hyphenation, therefore it is necessary to provide for such differences.

Liturgical Latin is someway between ecclesiastic and classical Latin; it is pronounced the same as ecclesiastic, but it is hyphenated as if it were classical Latin, with minor variations due to the fact that the pronunciation is different from the classical one. It is mainly used in liturgical singing; nowadays, that the national languages are used in all Roman Catholic services, Latin remains in use only when singing the Gregorian plain chant. Matter of fact it is the GregorioTeX Team that is taking care of the hyphenation patterns for liturgical Latin. Meanwhile they revise also the other varieties of Latin.

German style modern Latin is a variant of modern Latin. The spelling is the same, but hyphenation is somewhere between Italian style modern latin, and classical Latin. Evidently in Germany Latin is taught with a different approach than in Italy. Possibly the revision that the GregorioTeX Team is doing will produce a different set of hyphenation patterns even for modern Latin, so that it is not excluded that in the future the actual Italian style hyphenation is replaced by the German style one.

I can report that at the moment the `hyph-la.tex` file contains the patterns for (Italian style) modern and medieval Latin; prosodic marks are dealt with special macros defined in the language description file `latin.ldf` for babel; they do not need any particular functionality when using OpenType fonts with LuaTEX and XeLaTEX. The number of patterns in the above pattern file amounts to 334 items. They are valid for both the Italian style modern Latin and the medieval one.

Just as the patterns for Italian were shown before, those for modern and medieval Latin are shown in pattern list 3.

At the moment of writing the patterns for classical Latin are in file `hyph-la-x-classic.tex` and amount to 658 items.

Similarly the patterns for liturgical Latin are stored in file `hyph-la-x-liturgic.tex` and amount to 1730 items. It is not excluded that after the revision by the GregorioTeX Team both such patterns files may double in size.

4.5.3 The Greek case

In Greek we have a similar situation as in Latin. There are some main differences: modern Greek has two spellings: monotonic and polytonic, but in spite of their different accent system they have the same lexicon; modern and ancient Greek are written in a different alphabet than Latin; modern and ancient Greek use a lot of diacritics; the lexicon of ancient Greek is rather different from the modern one. In facts modern Greek lexicon contains groups of consonants that never occur in ancient Greek. Some are special combinations to render sounds absent from Greek but present in loan words.

The language has inflection for verbs, nouns and adjectives and the diacritical marks quite often change and/or do not remain on the same vowel in the inflected forms.

Hyphenation therefore is further complicated by the presence of diacritical marks.

But the modern versions of the language follow simple grammar rules and do not take in consideration prefixes and suffixes. At the time of writing this document the `...hyphenmin` values are both preset to 1, but many typographers in Greece set them to 2 and 3.

Modern monotonic Greek is the most used version of Greek in Greece. The alphabet is the usual 25 lowercase and 24 uppercase letters:

αβγδεζηθικλμνξοπρσςτυφχψω
ΑΒΓΔΕΖΗΘΙΚΛΜΝΞΟΠΡΣ ΤΥΦΧΨΩ

It uses just two accents, the "tonos" (in practice the acute accent), and the "dialytika" (the diaeresis). The tonos is used on the tonic syllable of polysyllabic words, and the dialytica when it is necessary to split an apparent diphthong; sometimes the dialytica and the tonos fall on the same vowel; all vowels including iota and upsilon may get the tonos, but the

Code 3: The 334 patterns for modern and medieval Latin

```
\patterns{% Pattern start
.a2b3l .anti1  .anti3m2n .circu2m1 .co2n1iun
.di2s3cine .e2x1 .o2b3
.para1i .para1u .su2b3lu .su2b3r 2s3que.  2s3dem. 3p2sic 3p2neu
æ1 œ1 a1ia a1ie a1io a1iu ae1a ae1o ae1u e1iu io1i o1ia o1ie o1io o1iu uo3u
1b   2bb   2bc   2bd   b2l  2bm  2bn  b2r  2bt  2bs  2b.
1c   2cc   c2h2  c2l   2cm  2cn  2cq  c2r  2cs  2ct  2cz  2c.
1d   2dd   2dg   2dm   d2r  2ds  2dv  2d.
1f   2ff   f2l   2fn   f2r  2ft  2f.
1g   2gg   2gd   2gf   g2l  2gm  g2n  g2r  2gs  2gv  2g.
1h   2hp   2ht   2h.
1j
1k   2kk   k2h2
1l   2lb   2lc   2ld   2lf  l3f2t 2lg 2lk  2ll  2lm  2ln  2lp  2lq  2lr
     2ls   2lt   2lv   2l.
1m   2mm   2mb   2mp   2ml  2mn  2mq  2mr  2mv  2m.
1n   2nb   2nc   2nd   2nf  2ng  2nl  2nm  2nn  2np  2nq  2nr  2ns
     n2s3m n2s3f 2nt   2nv  2nx  2n.
1p   p2h   p2l   2pn   2pp  p2r  2ps  2pt  2pz  2php 2pht 2p.
1qu2
1r   2rb   2rc   2rd   2rf  2rg  r2h  2rl  2rm  2rn  2rp  2rq  2rr  2rs  2rt
     2rv   2rz   2r.
1s2  2s3ph 2s3s  2stb  2stc 2std 2stf 2stg 2st3l     2stm 2stn 2stp 2stq
     2sts  2stt  2stv  2s.  2st.
1t   2tb   2tc   2td   2tf  2tg  t2h  t2l  t2r  2tm  2tn  2tp  2tq  2tt
     2tv   2t.
1v   v2l   v2r   2vv
1x   2xt   2xx   2x.
1z   2z.
% Additional patterns for medieval Latin
a1ua a1ue a1ui a1uo a1uu e1ua e1ue e1ui e1uo e1uu
i1ua i1ue i1ui i1uo i1uu o1ua o1ue o1ui o1uo o1uu
u1ua u1ue u1ui u1uo u1uu
%
a2l1ua a2l1ue a2l1ui a2l1uo a2l1uu e2l1ua e2l1ue e2l1ui e2l1uo e2l1uu
i2l1ua i2l1ue i2l1ui i2l1uo i2l1uu o2l1ua o2l1ue o2l1ui o2l1uo o2l1uu
u2l1ua u2l1ue u2l1ui u2l1uo u2l1uu
%
a2m1ua a2m1ue a2m1ui a2m1uo a2m1uu e2m1ua e2m1ue e2m1ui e2m1uo e2m1uu
i2m1ua i2m1ue i2m1ui i2m1uo i2m1uu o2m1ua o2m1ue o2m1ui o2m1uo o2m1uu
u2m1ua u2m1ue u2m1ui u2m1uo u2m1uu
%
a2n1ua a2n1ue a2n1ui a2n1uo a2n1uu e2n1ua e2n1ue e2n1ui e2n1uo e2n1uu
i2n1ua i2n1ue i2n1ui i2n1uo i2n1uu o2n1ua o2n1ue o2n1ui o2n1uo o2n1uu
u2n1ua u2n1ue u2n1ui u2n1uo u2n1uu
%
a2r1ua a2r1ue a2r1ui a2r1uo a2r1uu e2r1ua e2r1ue e2r1ui e2r1uo e2r1uu
i2r1ua i2r1ue i2r1ui i2r1uo i2r1uu o2r1ua o2r1ue o2r1ui o2r1uo o2r1uu
u2r1ua u2r1ue u2r1ui u2r1uo u2r1uu
}% Pattern end
```

last two ones can get the dialytica or both diacritical marks.

Modern polytonic Greek uses the same lexicon of monotonic Greek but uses the full set of accents as ancient Greek, that is acute, grave and circumflex, plus three diacritical marks: the smooth breath, the rough breath and the dialytica; they may be combined in couples, therefore there are 15 different combinations (not available on all vowels, but some vowels may have that full variety of diacritical marks). It is obvious that hyphenation becomes very complicated, but at least hyphenation follows simple grammar rules.

Ancient Greek of course uses the ancient lexicon; it uses the same diacritical marks as modern polytonic Greek, plus the iota subscripted or adscripted. Hyphenation is further complicated since break points must take into consideration prefixes and suffixes.

The use of a different alphabet sets forth some problems with fonts. As it is well known pdfLaTeX handles only fonts the glyphs of which are coded with just one byte. On the opposite, LuaLaTeX and XeLaTeX use OpenType fonts that may contain glyphs for dozens of languages and require the Unicode encoding that is a multibyte one. Two different approaches are therefore required for such different typesetting engines.

For pdfLaTeX it is necessary to use specially coded Greek fonts where each glyph has a one byte address, while with OpenType fonts LuaLaTeX and XeLaTeX can fetch the glyphs they need directly from the many alphabets coded in those fonts with their multibyte encoding; but, caution: not all fonts contain the Greek script (in particular the OpenType Latin Modern default fonts), therefore it is important to check that the desired font contains it.

With pdfLaTeX a *Local GReek* (LGR) encoding is defined so that with a regular Latin keyboard it is possible to write Greek using a Latin transliteration; with OpenType fonts it is possible to enter the Greek text directly in Greek without any transliteration. Actually the `greek.ldf` language description file (for babel) uses a LICR (LaTeX Internal Character Representation) technique such that also with pdfLaTeX it is possible to enter Greek text in Greek. The problem, therefore, is not the input text, but the keyboard itself. Often modern computers have available a virtual keyboard to be acted upon with the mouse or by tapping a touchscreen, so that a real keyboard is now relatively superfluous.

In any case the LGR encoding maps the Latin characters to the Greek ones as shown in table 4. The various diacritical marks are prefixed to the letters, except the iota sub- or ad-scripted, that follows the letter; they are represented by the ASCII chracters ` ' " > < '' | (respectively for grave, acute, circumflex, soft breath, rough breath, diaeresis, iota sub- or ad-scripted); they may also be paired; they do not need macros, but if macros are used a better kerned text is obtained and hyphenations should work better (but see below); therefore one can enter a transliterated text as `a>ut~h|` or `a\>ut\~h|` where the second input form is recommended; but, if a Greek real or virtual keyboard is available, it is possible to enter directly αὐτῆ. The internal LICR character representation fetches directly the marked letters and uses the correct kernings.

Table 4: Mapping of the Latin alphabet to the Greek one

```
abcdefghijklmnopqrstuvwxyz
αβςδεφγηιθκλμνοπχρστυ ωξψζ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
AB ΔΕΦΓΗΙΘΚΛΜΝΟΠΧΡΣΤΥ ΩΞΨΖ
```

But all this influences the hyphenation patterns; at the moment of writing the patterns to be included into the format file for `pdftex` work only with the Latin transliteration. With XeLaTeX there are no problems when the user specifies the Greek language and optionally selects one of its variants; with LuaLaTeX at the moment only the hyphenation for modern monotonic Greek is available; further on it will be shown a patch in order to extend the functionality of LuaLaTeX in connection with the varieties of Greek (of course, it is not necessary to patch anything if meanwhile the patch we suggested to the maintainers has already been applied).

With pdfLaTeX and its Latin transliteration, the Latin "v" key is not useless: it types an invisible strut character just as high as lowercase letters in order to have something to put diacritics on; it is more a service trick; it is used also to hide the actual word termination in order to type an isolated σ; in facts the ligature mechanism embedded into the LGR encoded Greek fonts is so strong that typing "s" at the end of a word gets transformed into a final sigma ς, so that non Greek users do not have to remember to insert "c" at the end of the words when using the Latin transliteration.

All these peculiarities imply a complicated pattern file; at the moment of writing this paper the existing pattern files to be used with pdfLaTeX work correctly only with Latin transliteration, not when the LICR codes or the direct Greek script input are used. To be precise, they do work also in these situations but, when accented glyphs are encountered, some break points may be missing.

The pattern files presently available when using pdfLaTeX were prepared by Dimitri Filippou several years ago, when only the Latin transliteration was available. I tried to upgrade those patterns, but without a specific testing made by actual Greek writers, I can't upload such new pattern files. In

the pattern list 5 I show these patterns for the monotonic variety; those for the polytonic and ancient varieties are too long to be shown here. In any case neither the patterns of table 5 (shown just to underline the special codes that are needed to identify the single byte 8-bit codes, from 128 to 255 in the LGR encoding) nor those for the polytonic variety is yet publicly available. Dimitri Filippou already provided the patterns for the three varieties to be used with LuaLaTeX and XeLaTeX, but nothing has been done to update the situation with pdfLaTeX; it is understandable: with the availability of XeLaTeX and LuaLaTeX, the importance of pdfLaTeX is strongly diminished for the Greek users.

The modification consists in replicating the patterns that involve ligatures between diacritics and vowels by replacing the accent-vowel pair with the upper half address of the LGR encoding; to do this it is necessary to code such letters with their double caret notation hexadecimal address; this means that the two (lowercase) characters that follow a couple of carets `^^`, form the hexadecimal address of the accented character; this is clearly evident in the seven lines that follow the comment `vowels`. The vowels iota and upsilon can receive the tonos alone, the dialytika alone, and both the dialytika and tonos.

Most of the patterns that are included in the pattern list 5 come from the pattern file `grmhyph5.tex` created by Dimitri Filippou; the upper half addresses of the LGR encoded Greek fonts have been added, and a very small number of consonant patterns has been added in order to handle a certain number of particular words, especially loan words from foreign languages and/or toponyms.

For modern polytonic and ancient Greek patterns the one byte situation is similar but it is much more complicated because of the multitude of diacritical marks and in both variants the inflection may move and/or change the accents, so that the number of patterns may be to six times larger than for monotonic modern Greek.

When using XeLaTeX and LuaLaTeX Dimitri Filippou created the `hyph-grc.tex` for ancient Greek that contains about 2000 patterns in Greek script. He also coded Greek patterns in Greek script for monotonic and polytonic Greek; their names are `hyph-el-monoton.tex` and `hyph-el-polyton.tex`.

4.5.4 General considerations

From the above it may seam that preparing pattern files requires an excellent ability in using "strange" codes; but it is not that difficult, since anybody with a good deal of patience and attentions can do the coding. The difficult part is to have a good knowledge of the language in order to extend the initial codes set forth by some scripting language, say Python or Java, to perform all the combinations required by the "macro patterns" discussed above. The pattern creator must be aware of the multitude of exceptions to the general rule without needing to create also the list of exceptions; for American English the *TUGboat* editor, Barbara Beeton, has been in charge of collecting and maintaining the list of exceptions that cannot be handled by the various pattern sets available for US English. But, as it has already been emphasised, American English has so many and "strange" rules that it is virtually impossible to manually create any set of patterns, so that recourse to `patgen` is the only practicable way to do the job. In any case with or without `patgen` homographs remain a big problem.

It is worth noting that there are some languages, especially typeset with other scripts, that do not use hyphenation; oriental languages that use ideographs do not hyphenate anything; their line breaking is on word boundaries or "anywhere". Arabic scripts do not use hyphenation but produce justified texts by horizontally stretching the "kashida", a curved line that decorates the ending forms of most letters.

5 The language description file

Hyphenation by itself is a necessary part while managing typesetting in different languages. Nevertheless some typesetters prefer typesetting ragged right; others produce block justification avoiding hyphenation of any kind; in order to avoid white meandering lines through the text, due to wide inter *word* spaces, they use tracking, that consists in enlarging the space between *characters*. For most book designers slight tracking is something to be used only when typesetting in all caps and/or small caps. With regular lowercase letters, with or without serifs, they claim that those who use tracking should be treated as those who steal sheep; this very strong sentence is understandable if one recalls that in the past centuries sheep thieves used to be condemned to death by hanging.

Instead of tracking, the best way to reduce to a minimum the number of hyphenated line breaks consists in using the microtype package. The TeX system typesetting engines follow an algorithm that minimises the badness of each paragraph; this is not the right place to discuss the details, but every LaTeX user knows how well paragraphs typeset by these programs are done. In any case package microtype does a very fine work in further improving this paragraph typesetting excellency.

The best results are obtained with pdfLaTeX and LuaLaTeX by applying two techniques: *protrusion* and *character expansion*. Protrusion lets small parts of the first and the last characters in a line to protrude in the adjacent margin. Character expansion consists in very slight stretching or shrinking of all the characters in a line so as to ren-

Code 5: The extended list of monotonic Greek hyphenation patterns for use with pdfLaTeX

```
\patterns{% Pattern start
%%%% vowels
a1 ^^881
e1 ^^e81
h1 ^^a01
i1 ^^d01 ^^f21 ^^f01
o1 ^^ec1
u1 ^^f61 ^^f41 ^^d41
w1 ^^b8
%%%%
a2i a2’i a2^^d0 a2u a2’u a2^^d4 ’a3u ^^883u
e2i e2’i e2^^d0 e2u e2’u e2^^d4 ’e3u ^^e83u
h2u h2’u h2^^d4 ’h3u ^^a03u
o2i o2’i o2^^d0 o2u o2’u o2^^d4 ’o3u ^^ec3u
u2i u2’i u2^^d0 ’u3i ^^d43i
%%%%
a2h ’a3h. ^^883h. a2’’i a2^^f0 ’a3i. ^^883i. a2’’u a2^^f4
e2’’i e2^^f0 e2’’u e2^^f4 o2ei o2h ’o3h. ^^ec3h. o2’’i o2^^f0 ’o3i. ^^ec3i.
%%%%
i2a i2’a i2^^88 i2e i2’e i2^^e8 i2o i2’o
o3’’i3’o o3^^f03^^ec
i2w i2’w i2^^b8 .i3 ’i3
h2a h2’a h2^^88 h2e h2’e h2^^e8 h2o h2’o h2^^ec h2w h2’w h2^^b8 .h3 .’h3 .^^a03
u2a u2’a u2^^88 u2o u2’o u2^^ec u2w u2’w u2^^b8 .u3 .’u3 .^^d43
%%%%
4b. 4g. 4gk. 4d.  4z.  4j.  4k. 4l. 4m. 4mp. 4n. 4nt. 4x. 4p. 4r.
4s. 4c. 4t.  4tz. 4ts. 4tc. 4f. 4q. 4y.
%%%%
4’’  4b’’  4g’’ 4gk’’ 4d’’ 4z’’ 4j’’ 4k’’  4l’’  4m’’ 4mp’’
4n’’ 4nt’’ 4x’’ 4p’’  4r’’ 4s’’ 4t’’ 4tz’’ 4ts’’ 4f’’ 4q’’ 4y’’
%%%%
.b4 .g4 .d4 .z4 .j4 .k4 .l4 .m4 .n4 .x4 .p4 .r4 .s4 .t4 .f4 .q4 .y4
%%%%
4b1b 4g1g 4d1d 4z1z 4j1j 4k1k 4l1l 4m1m 4n1n 4p1p 4r1r
4s1s 4t1t 4f1f 4q1q 4y1y
%%%%
4b1z 4b1j 4b1k 4b1m 4b1n 4b1x 4b1p 4b1s 4b1t 4b1f 4b1q 4b1y
4g1b 4g1z 4g1j 4g1m 4r5g2m 4g1x 4g1p 4g1s 4g1t 4g1f 4g1q 4g1y
4d1b 4d1g 4d1z 4d1j 4d1k 4d1l 4d1x 4d1p 4d1s 4d1t 4d1f 4d1q 4d1y
4z1b 4z1g 4z1d 4z1j 4z1k 4z1l 4z1m tz2m 4z1n 4z1x 4z1p 4z1r 4z1s 4z1t 4z1f 4z1q 4z1y
4j1b 4j1g 4j1d 4j1z 4j1k 4j1m 4r5j2m sj2m 4j1x 4j1p 4j1s 4j1t 4j1f 4j1q 4j1y
4k1b 4k1g 4k1d 4k1z 4k1j 4k1m 4l5k2m  4r5k2m  4k1x 4k1p 4k1s 4k1f 4n5k2f  4k1q 4k1y
4l1b 4l1g 4l1d 4l1z 4l1j 4l1k 4l1m 4l1n 4l1x 4l1p 4l1r 4l1s 4l1t 4l1f 4l1q 4l1y
4m1b 4m1g 4m1d 4m1z 4m1j 4m1k 4m1l 4m1x 4m1r 4m1s 4m1t 4m1f 4m1q 4m1y
4n1b 4n1g 4n1d 4n1z 4n1j 4n1k 4n1l 4n1m 4n1x 4n1p 4n1r 4n1s 4n1f 4n1q 4n1y
4x1b 4x1g 4x1d 4x1z 4x1j 4x1k 4x1l 4x1m 4x1n 4x1p 4x1r 4x1s 4x1t 4g5x2t 4r5x2t 4x1f 4x1q 4x1y
4p1b 4p1g 4p1d 4p1z 4p1j 4p1k 4p1m 4p1x 4p1s 4p1f 4p1q 4p1y
4r1b 4r1g 4r1d 4r1z 4r1j 4r1k 4r1l 4r1m 4r1n  4r1p 4r1s 4r1t 4r1f 4r1q 4r1y
4s1d 4s1z 4s1n 4s1x 4s1r 4s1y
4t1b 4t1g 4t1d 4t1j 4t1k 4t1n 4t1x 4t1p 4t1f st2f 4t1q 4t1y
4f1b 4f1g 4f1d 4f1z 4f1k 4f1m 4f1n 4r5f2n 4f1x 4f1p 4f1s 4f1q 4f1y
4q1b 4q1g 4q1d 4q1z 4q1k 4q1m 4r5q2m 4q1x 4q1p 4q1s 4q1f 4q1y
4y1b 4y1g 4y1d 4y1z 4y1j 4y1k 4y1l 4y1m 4y1n 4y1x 4y1p 4y1r 4y1s 4y1t 4m5y2t 4y1f 4y1q
%%%%
4g5k2f 4g1kt 4m1pt 4n1tz 4n1ts
%%%%
4br. 4gl. 4kl. 4kt. 4gkc. 4gks. 4kc. 4ks. 4lc. 4ls.
4mpl. 4mpn. 4mpr. 4mc.  4ms. 4nc. 4ns. 4rc. 4rs.
4sk. 4st. 4tl. 4tr. 4ntc. 4nts. 4ft. 4qt.
%%%%
4gk1mp 4gk1nt 4gk1tz 4gk1ts 4mp1nt 4mp1tz
4mp1ts 4nt1mp 4ts1gk 4ts1mp 4ts1nt
}% Pattern end
```

der more homogeneous the inter word spaces. The results are visible in this very document, where the narrow columns still require some hyphenated line breaks but the inter word spaces do not let any meandering rivulets appear trough the paragraphs. XeLaTeX cannot use character expansion, but even by using only protrusion it does a fine work.

The choice of using microtype may be one of the tasks assigned to the language description files. The interested reader can examine the babel documentation Braams e Bezos (2018) that describes a model or a template for creating language description files; here we summarise the tasks that are generally assigned to such files.

Any language description file should do the following.

1. It makes sure the file is read just once.
2. It examines the presence of options, language modifiers and similar user choices in order to execute the proper settings.
3. For the language it describes, or for a language variant, it controls if suitable hyphenation pattern files have been loaded in the format file or can be loaded at run time (only LuaLaTeX). If no patterns are available a suitable set of alternate patterns are chosen; very often the default US English patterns are selected; sometimes those of another language; sometimes those of the "pseudolanguage" `nohyphenation` so that the almost empty, patternless file `zerohyph.tex` is loaded.
4. It checks if the `\captions`⟨*language*⟩ macro is defined or if the infix word macro definitions should be let equivalent to similar macros for another variant of the language. The above macro contains the definitions of "name" macros that contain the infix words such as "Chapter", "Table of contents", and the like. In this way the LaTeX kernel macros or the class macros use such "name" macros without the need to have different definitions for each language; these different definitions are necessary, but are included in each language definition file.
5. It defines how to set the date; in particular it redefines the macro `\today` to adapt its format and month name to the specific language. Everybody knows that even in English the American-style date follows the format "⟨*month*⟩ ⟨*day number*⟩, ⟨*year*⟩", while the British style follows the format "⟨*day number with ordinal suffix*⟩ ⟨*month*⟩ ⟨*year*⟩". Other languages may follow different formats. The Scandinavian countries use the numerical ISO format "⟨*year*⟩-⟨*month*⟩-⟨*day*⟩"; others may use their ordinal suffixes and prepositions before the month name and/or year number. In addition some languages may require other date formats, such as the roman numbering or other kinds of numbering. Historical eras generally are omitted, since the current era is assumed; but some languages accept negative year numbers so that special macros have to be created for them.
6. Some languages require shortcuts in order to perform frequent tasks; for this purpose they generally use *active characters* to be defined as it is necessary for any specific task. Active characters are of category 13. The most common one is the tilde, "~", that is a LaTeX kernel specific command used to insert a non breakable space between two words; it is generally named "tie". Most languages define other active characters, the most frequent of which is the straight double quote "; but according to the specific needs of each language other active characters may be necessary. In French the "high" punctuation marks are defined as active characters in order to automatically insert an unbreakable space before the regular punctuation glyphs; similarly an unbreakable space is necessary after open guillemets[4] and before closed guillemets. In Italian such spaces indadvertedly inserted in the source file must be eliminated. In ecclesiastic Latin such guillemet spaces are optional, but if used they must have a thinner unbreakable space than in French. In German quotation marks are different from those used in other western languages and require special shortcuts.
   The actual German shortcuts are shown in table 6 that replicates the table contained in the `ngerman.ldf` file.
7. Some languages define also characters that are active only in math mode; in Italian, for example, the comma is optionally defined in a specific way so that in math mode it recognises if it is followed by a digit, so as to behave as a decimal separator, else it behaves as a punctuation mark.
8. Other settings may be optionally activated or deactivated at user command; fore example, when typesetting Latin text, prosodic marks can be turned on or off with the specific user commands `\ProsodicMaksOn` and `\ProsodicMarksOff`. In Italian even the straight double quotes can be activated or deactivated.
9. In any case each language definition file has to activate its macros and general settings, because the situation must be restored upon changing language.
10. All language definition files for babel must be pure ASCII texts; no accented characters are allowed, because the file must be "encoding

4. The babel documentation uses the french word *guillemet*, but the code uses commands such as `\guillemotleft` and `\guillemotright`.

Table 6: Shortcuts defined for the German Language with the new orthography

''a	Umlaut ⟨ä⟩ (shorthand for \''a). Similar shorthands are available for all other lower- and uppercase vowels (umlauts: ''a, ''o, ''u, ''A, ''O, ''U; tremata: ''e, ''i, ''E, ''I).
''s	German ⟨ß⟩ (shorthand for \ss {}).
''z	German ⟨ß⟩ (shorthand for \ss {}). Differs to ''s in uppercase version.
''S	\uppercase {''s}, typeset as ⟨SS⟩(⟨ß⟩ must be written as ⟨SS⟩ in uppercase writing).
''Z	\uppercase {''z}, typeset as ⟨SZ⟩. In traditional spelling, ⟨ß⟩ could also be written as ⟨SZ⟩ instead of ⟨SS⟩ in uppercase writing. Note that, with reformed orthography, the ⟨SZ⟩ variant has been deprecated in favour of ⟨SS⟩ only.
''|	Disable ligature at this position (e. g., at morpheme boundaries, as in Auf''|lage).
''-	An additional breakpoint that does still allow for hyphenation at the breakpoints preset in the hyphenation patterns (as opposed to \-).
''=	An explicit hyphen with a breakpoint, allowing for hyphenation at the other points preset in the hyphenation patterns (as opposed to plain -); useful for long compounds such as IT''=Dienstleisterinnen.
''~	An explicit hyphen without a breakpoint. Useful for cases where the hyphen should stick at the following syllable, e. g., bergauf und ''~ab .
''"	A breakpoint that does not output a hyphen if the line break is performed (consider parenthetical extensions as in (pseudo''~)''"wissenschaftlich).
''/	A slash that allows for a line break. As opposed to \slash {}, hyphenation at the breakpoints preset in the hyphenation patterns is still allowed.
''‘	German left double quotes ⟨„⟩.
'''	German right double quotes ⟨“⟩.
''<	French/Swiss left double quotes ⟨«⟩.
''>	French/Swiss right double quotes ⟨»⟩.

neutral". This is true even for LGR encoded Greek fonts, where the specific LICR macros must be used to spell the infix Greek words. The situation is different for polyglossia because this package works only with LuaLaTeX and XeLaTeX that can use OpenType fonts in order to manage many different scripts and certainly also non-ASCII glyphs.

We describe a language definition file prepared for polyglossia and an unusual language: Occitan. This choice is just to show a possible situation to face when a user has to deal with a rare language; there are dozens of minority languages that might be the object of essays or linguistic research. Of course the linguist who is doing research in this field must know the minority langue s/he is researching on, therefore s/he should not have difficulties creating the pattern files and even less difficulty creating the relative language description file.

5.1 The Greek case with LuaLaTeX

The differences between LuaLaTeX and the other typesetting programs consist mainly in the fact that pattern files are loaded at run time, and only for the languages explicitly named through the calls by the \setmainlanguage, \setotherlanguages, and \setotherlanguage in the preamble of the main source file. In the past it used to be impossible to load different pattern files for the language variants, at least if was difficult and no user commands were available. The gloss-greek.ldf language description file worked correctly with polyglossia when the document was typeset with XeLaTeX, but could not correctly work with LuaLaTeX.

Recently the package luahyphenrules (Bezos, 2016) was added to the TeX Live distribution. By means of this package it is possible to let LuaLaTeX behave as XeLaTeX by loading at run time the pattern files for the referenced languages and their variants as they are listed in the service file language.dat used by XeLaTeX; therefore even LuaLaTeX, by using this package, has available the hyphenation pattern files for all variants of Greek. I therefore created for myself a patched gloss-greek.ldf that contains such enhancement; the patch is discussed further on. Please remember, though, that the greek language name is used in the \setmainlanguage and \setotherlanguage mandatory arguments, while in the language changing specific commands the name used in the \HyphenRules argument must be used to refer to a specific variant; therefore in a document body written in English, a Greek citation in ancient Greek requires the Greek text to be enclosed in a normal greek environment, but within its body any command that refers to the language should use the ancientgreek argument, as it it shown with code 7.

The patch shown starting on page 20 is relatively simple and certainly should be improved. Notice that the code contained between the lines that mark the start and end of the pack are partially additions (from lines 19 to 40) and partially modifications of some existing code in the unpatched

CODE 7: The document to test the patched `gloss-greek.ldf`

```
% !TEX encoding = UTF-8 Unicode
% !TEX TS-program = LuaLaTeX

\documentclass[12pt]{article}

\usepackage{fontspec}
\setmainfont{CMU Serif}

\usepackage{polyglossia}
\setmainlanguage{italian}
\setotherlanguage{english}
\setotherlanguage[variant=ancient]{greek}

\begin{document}
\begin{otherlanguage}{greek}
Language: \the\language\ \languagename
\iflanguage{ancientgreek}{\ αὐτῇ}{}
\end{otherlanguage}

\end{document}
```

`gloss-greek.ldf` file. The dots in line 102 represent the remaining untouched part of the original `gloss-greek.ldf` file.

The name of the modified file remains the same, because polyglossia reads that very file for that language; in order to avoid conflicts this file may be saved in the personal `texmf` tree, so that the typesetting languages find this personal copy with precedence to the standard one; as mentioned in this paper, this personal copy should be renamed or deleted if and when a new release of official one contains a similar or better functionality.

The patch may exhibit innocuous error messages, but I always got out the correctly typeset document. I tested this patch with the document shown in the code listing 7 with both LuaLaTeX and XeLaTeX (with an "intelligent" shell editor capable to interpret the autoconfiguration initial comments, this amounts to change the prefix `Lua` with `Xe` in line 2 of the code shown in the code listing 7), by replacing the option `ancient` with `poly` or `mono`, and, correspondingly, by replacing the language name `ancientgreek` with `polygreek` or `monogreek` in line 17. If in line 15 the environment `otherlanguage` argument is set to `italian` or `english` the current language number and name is printed, but the Greek word is omitted. The attentive reader who is going to test the given code will notice that with both typesetting engines the language number for `english` is always 0 (zero), while for the other languages the number shown when using XeLaTeX is one unity lower than that shown when using LuaLaTeX; it is not an error, but it is implicit in the background macros that are being used to do the job; this is not the right place to discuss such details.

5.2 The Occitan language description file for polyglossia

Let us comment the code shown in appendix B. Lines 1 and 2 contain the usual *incipit* that every file should have; language definition files for babel have the almost equivalent statement `\ProvideLanguage`.

Lines from 3 to 9 contain the general settings for the language, in particular the parameters `hyphenmins` whose values represent the minimum lengths of the first and respectively the last word fragments when hyphenation can take place. The boolean flag `frenchspacing` set to `true` abolishes the different space factor to use for the full stop and the other "high" punctuation marks (as it is customary in English, but not in French and many other languages). The other boolean parameter `indentfirst` set to `true` requires that also the first paragraph of a sectional unit is indented as the paragraphs that follow it. Finally the boolean flag `fontsetup` set to `true` allows polyglossia to associate a specific font to a specific language.

Line 10 authorises to use and define those babel commands that define shorthands (shortcuts) for this specific language. Lines 12 to 16 load the specific babel module to define shorthands.

From line 21 to line 31 the straight double quote character is defined active and is given the main definition to distinguish its role in math vs text mode. Some service macros and the specific alternatives that the active character can execute, together with the macro to turn the active char off, are defined in lines 32 to 62.

From line 63 to line 85 the infix names are defined; since this code is to be used with polyglossia, accented characters are freely used. The Occitan date requires a special way to prefix month names and to write day numbers; actually the cardinal number 1 is substituted with its ordinal numeral, while the other day numbers are typeset with digits.

From line 76 to the end of the file there are the "housekeeping" macros to be executed when the language is set and when it is reset. Notice the reference to the two byte code point `''2019` that refers to the apostrofe; when setting Occitan the apostrophe is given its specific address, a positive number, as the lower case code, so that it is treated as a legal word character; when resetting its `lccode` is given the value zero (`\z@` stands for 0).

5.3 The Cimbrian language definition file

Let us assume that a philologist/linguist wants to write an essay on the Cimbrian language. This is a (rare) germanic language spoken in northern Italy by a small community estimated to two thousand people; of course this number is below the critical mass, so that UNESCO classifies it as an endangered language (WIKIPEDIA, 2018). It is reasonable

that linguists would like to record its lexicon, its pronunciation, its grammar, and so on. A sample of Cimbrian, at least one of its varieties, taken from Wikipedia (2018), is shown in page 22 together with its German and English translations.

But in order to use the TEX system programs our linguist should create the necessary equipment for typesetting anything in Cimbrian. Therefore he has to create the Cimbrian pattern file, and the Cimbrian language description file.

In a first step our linguist could concentrate on the language description file. The model offered from the Occitan file can be used; all occurrences of the string `occitan` should be changed to `cimbrian` and the short string `oc` appearing in some macros should be changed to `cim`; according to the ISO 639-3 regulation, `cim` is the official letter code for this language. Our linguist should not have any problems replacing the Occitan infix words and the date strings in order to comply with the Cimbrian language. He should pay attention to the apostrophe, possibly specifying the same settings used for Occitan.

But missing a Cimbrian pattern file, our linguist could temporarily specify among the `\PolyglossiaSetup` command arguments `hyphenation = germanb`; after all Cimbrian is a far relative of German. The short Cinmbrian text in page 22 has been typeset with the German hyphen patterns and apparently none of the few line breaks appears to be wrong. In his experiments at the beginning he will find wrong hyphen points, but while experimenting with various texts he collected from the local Cimbrian speaker area, he might classify a certain number of corrections to add to the German patterns. He therefore copies the `hyph-de-1901.txt` or `hyph-de-1996.txt` file to a new file `hyph-cim.txt` (the `.txt` extension is specific for LuaLATEX); he will replace the prologue of the file with suitable information and licence statement. He will gradually replace or add new patterns according to the rules specified in previous sections and such that the hyphenation errors in his Cimbrian text vanish.

In order to speed up this work our linguist is recommended to use the testhyphens package (Beccari, 2015). He creates himself a working directory, say, CimbrianTestFolder, where he saves the above named `hyph-cim.txt` new file. In this directory he creates and saves the following source `.tex` file, naming it, say, `TestCimbrianHyphens.tex`:

```
% !TEX TS-program = LuaLaTeX
% !TEX encoding = UTF-8 Unicode
\documentclass[12pt]{article}
 \usepackage{fontspec}
 \defaultfontfeatures{Ligatures={NoCommon,
 NoDiscretionary, NoHistoric, NoRequired,
 NoContextual}}
 \setmainfont{CMU serif}
\usepackage{luacode}
\usepackage{testhyphens}
\usepackage{multicol}

\begin{luacode}
local patfile = io.open('./hyph-cim.txt')
langobject = lang.new()
lang.patterns(langobject, patfile:read('*all'))
patfile:close()
\end{luacode}

\language=%
   \directlua{tex.sprint(lang.id(langobject))}
\begin{documment}

Language \the\language

\begin{multicols}{2}
\lefthyphenmin=3
\righthyphenmin=2
\lccode`\'=\`\' % possibly needed
                % to handle apostrophes
\begin{checkhyphens}
% Cimbrian word list
...
\end{checkhyphens}
\end{multicols}
```

He will change the values of the `...hyphenmin` macros to the values he thinks to be adequate to the Cimbrian language; the numbers shown are the most widely used among the more than eighty languages dealt with by the TEX system. He will delete the comment line `% Cimbrian word list` and the line of dots that immediately follows, and replace them with a list of Cimbrian words; for example the words that form his quoted Cimbrian texts; for his convenience he will copy his whole text and globally put after each inter word space a new line character so as to have one word followed by one space per line; he will resort to a suitable text editor that can sort the words; in the worst case he can use a command line command to sort the file lines. By so doing he can easily delete duplicate words.

At this point he processes the document with LuaLATEX. The program loads the specified pattern file and the specified packages and produce the PDF output file. The format is a two column document containing all the input words, one per line, fully hyphenated. In this way it is very easy to control the result, find possible errors, correct the `hyph-cim.txt` file accordingly, and restart the process.

When there will be no more errors our linguist finds another Cimbrian text, adds it to the word list; sorts the words one per line, eliminates the duplicates and restarts the process.

There is some work to do, but having available a decent number of Cimbrian texts, the process may last few hours in total; it requires a lot of intelligence in order to analyse every hyphenated word so as to decide if the hyphens are correct;

Cimbrian

Pan khriage dar forte vo Lusern hat se gebeart gerecht. Di earstn tage von khriage, dar kommandant a Tschechoslowako hebat in forte gebelt augem un hat ausgezoget di bais bandiara un is vongant pin soldan. A trunkhantar soldado alua is no gestant sem in forte. Bia da soin zuakhent di Balischan zo giana drin in forte, is se darbkeht dar trunkhante soldado un hat agehevt z'schiasa.

German

Während des Krieges wehrte sich die Festung von Lusern vortrefflich. Die ersten Tage wollte sie ein tschechischer Kommandant aufgeben, indem er die weiße Fahne hisste und mit der Besatzung abzog. Nur ein betrunkener Soldat blieb zurück in der Festung. Als die anstürmenden Italiener in die Festung eindringen wollten, um sie in Besitz zu nehmen, erwachte der betrunkene Soldat von seinem Rausch und fing an, das Maschinengewehr knattern zu lassen.

English

During the war, the fort of Lusern resisted superbly. In the first few days a Czech commander wanted to give up, hoisting the white flag and withdrawing the garrison. Only one drunken soldier remained in the fort. When the Italians came storming into the fort to occupy it, the drunken soldier awoke from his intoxication and began to let the machine gun rattle.

in case they are not, intelligence comes again to rescue to decide which patterns to correct or which new patterns to add to the pattern file.

Once the work is done our linguist is very satisfied; he can write his paper in/on Cimbrian, but as a LaTeX user he feels necessary to share his work with other TeXies. He therefore sends his `hyph-cim.txt` file to the TeX-hyphen Team. On turn the Team processes the file, transforms it to the other formats needed by pdfLaTeX, XeLaTeX and pLaTeX (the version that is used in Japan to handle Japanese fonts but also for writing texts in western languages on any subject); the Japanese TeXies are very active in every discipline and may be interested even in studying minority European languages. The Team provides also to edit the various service macros necessary to configure the format file initialisation; not only, but they maintain the whole procedure running smoothly for the benefit of the whole community.

6 The service files

Anyone who creates pattern files and/or language description files does not need only a structure to test the patterns as described above. That person probably wants to use those files for his/her documents. Therefore s/he has to install those files so as to make them visible to the TeX system even before the Teams working at TUG installs them in an update for the general TeX user community.

It is therefore time to use a personal TeX directory tree. With TeX Live this tree is rooted in the user `$HOME` directory, where that symbol means different things with different operating systems: for Linux platforms it is `~/`; for Mac it is `~/Library/`; for recent Windows operating systems it is `C:\Users\`⟨*username*⟩`\`. The tree base directory is named `texmf` and its ramification should be a subset of the main TeX system tree one; therefore it has a ramification such as `tex/latex/` to which another directory such as `MyTeXLiveFiles` may be added. This directory shall contain the language description files; with the TeX Live installation it is not necessary to update the file name database.

Things are more complicated with pattern files; they should be saved into other branches of the personal tree, but the really important files are the `language-local.dat`, `language-local.def`, and `language-local.dat.lua` ones; such files must contain only the personal additions or deletions to the already existing files; by running

```
tlmgr generate language
```

with administrator or root privileges. This action regenerates the various `language.*` needed to rebuild the format files.

Eventually, having created the proper files and saved them in the proper folders, the final touch is to run, with administrator or root privileges, the `fmtutil-sys` program specifying the format files where it is desired to install the new language functionalities; in general they will be pdfLaTeX, LuaLaTeX and XeLaTeX. Personally I prefer to recreate only the pdfLaTeX and LuaLaTeX formats.

Caution: if and when the new functionalities are added to the TeX Live distribution, the branches of the personal tree just discussed should be eliminated from the computer. Any upgrade of the TeX Live distribution never touches the personal tree so that they are not upgraded any more unless the user provides by him/herself.

7 Conclusion

What has been described in this paper describes what there is behind the language processing done by the various typesetting programs based on LaTeX.

The processing steps needed to extend such language processing tasks are all pretty delicate, but after all they are not so complicated. They require a lot of work and we, the LaTeX users, are grateful to the various contributors of pattern and language description files. They saved us a lot of work. But if we have a sufficient knowledge of a language, we

too can contribute our work to save a lot of work to the members of the TEX community.

Acknowledgements

A special grateful thank-you is due to the TEX-hyphen Working Group; they do a marvellous work, and without them language management would be at the level it was in the mid-nineties.

References

Beccari, Claudio (2015). *The `testhyphens` package*. TUG. Readable with `texdoc testhyphens`.

Berry, Karl (2018a). *The TEX Live Guide – 2018*. TUG. Readable with `texdoc texlive`.

— (2018b). *TLMGR(1)*. TUG. Readable with `texdoc tlmgr`.

Bezos, Javier (2016). *luahyphenrules –Loading patterns in LuaLATEX with language.dat*. TUG. Readable with `texdoc luahyphenrules`.

Braams, Johannes e Javier Bezos (2018). *Babel*. TUG. Readable with `texdoc babel`.

Charette, François e Arthur Reutenauer (2018). *Polyglossia*. TUG. Readable with `texdoc polyglossia`.

Liang, Frank (1983). «Word Hy-phen-a-tion by Com-put-er». The original thesis, defended at Standford University, was scanned and made available at www.tug.org/docs/liang/liang-thesis-hires.pdf.

Liang, Frank e Peter Breitenlohner (1991). *PATGEN(1)*. TUG. This is the PDF version of the manual revision after the upgrade of 1991 to adapt the program to TEX 3. Readable with `texdoc patgen-man1`.

Schlicht, R. (2018). *The `microtype` package –Subliminal refinements towards typographical perfection*. TUG. Readable with `texdoc microtype`.

UNI 6461 (1969). *Divisione delle parole in fin di linea*. Ente Italiano di Unificazione, Milano.

Wikipedia (2018). «The Cimbrian Language». https://en.wikipedia.org/wiki/Cimbrian_language. Last checked 2018-11-10.

A Patch to modify the `gloss-greek.ldf` in order to have LuaLaTeX load the pattern files for the three Greek variants

```
\ProvidesFile{gloss-greek.ldf}[polyglossia: patched module for Greek]
\PolyglossiaSetup{greek}{
  script=Greek,
  scripttag=grek,
  frenchspacing=true,
  indentfirst=true,
  fontsetup=true,
}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% The code in this file was initially adapted from the antomega
%% module for Greek. Currently large parts of it derive from the
%% package xgreek.sty (c) Apostolos Syropoulos
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This file imported from xgreek fixes the \lccode and \uccode
% of Greek letters:
\input{xgreek-fixes.def}
%  ------------------------------------------------- BEGIN PATCH
\ifx\directlua\undefined\else\RequirePackage{luahyphenrules}\fi

\define@key{greek}{variant}{monotonic}{%
   \ifx\directlua\undefined\language\l@monogreek
   \else\HyphenRules{monogreek}\fi%
}
\define@key{greek}{variant}{mono}{%
   \ifx\directlua\undefined\language\l@monogreek
   \else\HyphenRules{monogreek}\fi%
}
\define@key{greek}{variant}{polytonic}{%
   \ifx\directlua\undefined\language\l@polygreek
   \else\HyphenRules{greek}\fi%
}
\define@key{greek}{variant}{poly}{%
   \ifx\directlua\undefined\language\l@polygreek
   \else\HyphenRules{greek}\fi%
}
\define@key{greek}{variant}{ancient}{%
   \ifx\directlua\undefined\language\l@ancientgreek
   \else\HyphenRules{ancientgreek}\fi%
}

%TODO: set these in \define@key instead:
\ifx\l@greek\@undefined
  \ifx\l@polygreek\@undefined
    \xpg@nopatterns{Greek}%
    \adddialect\l@greek\l@nohyphenation
  \else
    \let\l@greek\l@polygreek
  \fi
\fi
\ifx\l@monogreek\@undefined
  \xpg@warning{No hyphenation patterns were loaded for Monotonic Greek\MessageBreak
         I will use the patterns loaded for \string\l@greek instead}
  \adddialect\l@monogreek\l@greek
\fi
\ifx\l@ancientgreek\@undefined
  \xpg@warning{No hyphenation patterns were loaded for Ancient Greek\MessageBreak
```

```
        I will use the patterns loaded for \string\l@greek instead}
  \adddialect\l@ancientgreek\l@greek

%set monotonic as default
\def\greek@variant{\l@monogreek}%                monotonic
\def\captionsgreek{\monogreekcaptions}%
\def\dategreek{\datemonogreek}%
\fi

\def\tmp@mono{mono}
\def\tmp@monotonic{monotonic}
\def\tmp@poly{poly}
\def\tmp@polytonic{polytonic}
\def\tmp@ancient{ancient}
\def\tmp@ancientgreek{ancientgreek}

\define@key{greek}{variant}[monotonic]{%
  \def\@tmpa{#1}%
  \ifx\@tmpa\tmp@mono\def\@tmpa{monotonic}\fi
  \ifx\@tmpa\tmp@poly\def\@tmpa{polytonic}\fi
  \ifx\@tmpa\tmp@ancientgreek\def\@tmpa{ancient}\fi
  \ifx\@tmpa\tmp@polytonic%                       polytonic
    \def\greek@variant{\l@greek}%
    \def\captionsgreek{\polygreekcaptions}%
    \def\dategreek{\datepolygreek}%
    \edef\greek@language{\noexpand\language=\greek@variant}
    \xpg@info{Option: Polytonic Greek}%
  \else
    \ifx\@tmpa\tmp@ancient%                       ancient
      \def\greek@variant{\l@ancientgreek}%
      \def\captionsgreek{\ancientgreekcaptions}%
      \def\dategreek{\dateancientgreek}%
      \edef\greek@language{\noexpand\language=\greek@variant}
      \xpg@info{Option: Ancient Greek}%
    \else %                                       monotonic
      \def\greek@variant{\l@monogreek}% monotonic
      \def\captionsgreek{\monogreekcaptions}%
      \def\dategreek{\datemonogreek}%
      \edef\greek@language{\noexpand\language=\greek@variant}
      \xpg@info{Option: Monotonic Greek}%
    \fi
  \fi}

% ------------------------------------------------------ END PATCH
% The original code continues hereafter
...
\endinput
```

B The Occitan language definition file

```
\ProvidesFile{gloss-occitan.ldf}[2016/02/04 v0.3 polyglossia:
     module for Occitan]
\PolyglossiaSetup{occitan}{
  hyphennames={occitan},
  hyphenmins={2,2},
  frenchspacing=true,
  indentfirst=true,
  fontsetup=true,
}
```

```
\define@boolkey{occitan}[occitan@]{babelshorthands}[true]{}

\ifsystem@babelshorthands
  \setkeys{occitan}{babelshorthands=true}
\else
  \setkeys{occitan}{babelshorthands=false}
\fi
\ifcsundef{initiate@active@char}{%
\input{babelsh.def}%
\initiate@active@char{''}%
}{}
\def\occitan@shorthands{%
  \bbl@activate{''}%
  \def\language@group{occitan}%
  \declare@shorthand{occitan}{''}{%
    \relax\ifmmode
      \def\xpgoc@next{''}%
    \else
      \def\xpgoc@next{\futurelet\xpgoc@temp\xpgoc@cwm}%
    \fi
  \xpgoc@next}%
}
\def\xpgoc@@cwm{\nobreak\discretionary{-}{}{}\nobreak\hskip\z@skip}
\def\xpgoc@ponchinterior{%
      \nobreak\discretionary{-}{}{\mbox{$\cdot$}}\nobreak\hskip\z@skip}
\def\xpgoc@cwm{\let\xpgoc@@next\relax
  \ifcat\noexpand\xpgoc@temp a%
                          \def\xpgoc@@next{\xpgoc@@cwm}%
  \else
    \if\noexpand\xpgoc@temp \string|%
      \def\xpgoc@@next##1{\xpgoc@@cwm}%
    \else
      \if\noexpand\xpgoc@temp \string<%
        \def\xpgoc@@next##1{«\ignorespaces}%
      \else
        \if\noexpand\xpgoc@temp \string>%
          \def\xpgoc@@next##1{\unskip»}%
        \else
          \if\noexpand\xpgoc@temp\string/%
            \def\xpgoc@@next##1{\slash}%
          \else
            \if\noexpand\xpgoc@temp\string.%
              \def\xpgoc@@next##1{\xpgoc@ponchinterior}%
            \fi
          \fi
        \fi
      \fi
    \fi
  \fi
  \xpgoc@@next}
\def\nooccitan@shorthands{%
  \@ifundefined{initiate@active@char}{}{\bbl@deactivate{''}}%
}
\def\captionsoccitan{%
```

```
    \def\refname{Referéncias}%
    \def\abstractname{Resumit}%
    \def\bibname{Bibliografia}%
    \def\prefacename{Prefaci}%
    \def\chaptername{Capítol}%
    \def\appendixname{Annèx}%
    \def\contentsname{Ensenhador}%
    \def\listfigurename{Taula de las figuras}%
    \def\listtablename{Taula dels tablèus}%
    \def\indexname{Indèx}%
    \def\figurename{Figura}%
    \def\tablename{Tablèu}%
    \def\partname{Partida}%
    \def\pagename{Pagina}%
    \def\seename{vejatz}%
    \def\alsoname{vejatz tanben}%
    \def\enclname{Pèça junta}%
    \def\ccname{còpia a}%
    \def\headtoname{A}%
    \def\proofname{Demostracion}%
    \def\glossaryname{Glossari}%
}
\def\dateoccitan{%
    \def\occitanmonth{\ifcase\month\or
       de~genièr\or
       de~febrièr\or
       de~març\or
       d'abril\or
       de~mai\or
       de~junh\or
       de~julhet\or
       d'agost\or
       de~setembre\or
       d'octobre\or
       de~novembre\or
       de~decembre\fi
    }%
    \def\occitanday{\ifcase\day\or
       1èr\else% primièr
       \number\day\fi% all other numbers
    }%
    \def\today{\occitanday\space \occitanmonth\space de~\number\year}%
}
\let\xpgoc@savedvalues\empty
\AtEndPreamble{% the user or the class might define different values
  \edef\xpgoc@savedvalues{%
    \clubpenalty=\the\clubpenalty\space
    \@clubpenalty=\the\@clubpenalty\space
    \widowpenalty=\the\widowpenalty\space
    \finalhyphendemerits=\the\finalhyphendemerits}
}
\def\noextras@occitan{%
    \lccode\string'2019=\z@
    \nooccitan@shorthands
```

```
    \xpgoc@savedvalues
}
\def\blockextras@occitan{%
    \lccode\string'2019=\string'2019
    \clubpenalty=3000 \@clubpenalty=3000 \widowpenalty=3000
    \finalhyphendemerits=50000000
    \ifoccitan@babelshorthands\occitan@shorthands\fi
}

\def\inlineextras@occitan{%
    \lccode\string'2019=\string'2019
    \ifoccitan@babelshorthands\occitan@shorthands\fi
}
```

▷ Claudio Beccari
claudio dot beccari at gmail dot com

La bandiera europea e la sezione aurea

Claudio Beccari

Sommario

La bandiera europea contiene una corona di dodici stelle a cinque punte; ogni stella è inscritta in un pentagono e il pentagono è legato alla sezione aurea. In questo breve articolo si mostra come il normale ambiente *picture* sia in grado di disegnare la stella e sia in grado di metterla in posizione correttamente.

Abstract

The European flag contain a circle of five-pointed stars distributed at the vertices of a regular dodecagon. This article shows how it is possible to draw the flag by using only the *picture* environment to draw the stars and to put them in the correct position.

Introduzione

Nell'ultimo aggiornamento di TEX Live che ho eseguito, ho notato il nuovo pacchetto euflag di Flynn (2019). Leggendone la documentazione, ho notato che l'autore sottolinea la validità del pacchetto spiegando che la bandiera viene disegnata semplicemente mediante il normale ambiente *picture*, nativo di LATEX, perché garantisce che non siano importati file grafici di nessun genere, visto che latex accetta solo il formato EPS; pdflatex accetta solo i formati PNG, JPG e PDF, ma gestisce anche le immagini EPS provvedendo da solo a convertirle in immagini PDF; invece, lualatex e xelatex accettano senza battere ciglio direttamente i formati PNG, JPG, PDF e EPS/PS.

Peter Flynn ha studiato con attenzione i decreti dell'Unione Europea per determinare sia le caratteristiche per il disegno della bandiera e delle sue stelle, sia dei colori; questi sono fissati con certi codici Pantone, che, trasformati in codici HTML, sono il `003399` per il blu di sfondo e `FFCC00` per il giallo dorato delle stelle.

Leggendo il suo codice, ho visto che in effetti usa l'ambiente *picture* con il comando `\put` per mettere in posizione le stelline in punti le cui coordinate sono calcolate a mano. Inoltre, le stelle sono ricavate dalla polizza di caratteri della American Mathematical Society, accessibili tramite il pacchetto amssymb che definisce il comando `\bigstar`.

Le dimensioni del disegno della bandiera e delle stelle sono legate alle dimensioni del font corrente, precisamente al valore di `1em` nel font corrente, tanto che, per mostrare nella documentazione la bandiera più grande, è costretto a specificare un font immenso con il comando:

```
{\fontsize{120}{0}\selectfont\euflag}
```

Mi pare chiaro che questa scelta insolita di dimensionare un disegno dipende dal fatto che è necessario dimensionare anche il font con cui rappresentare la stella.

Il codice contenuto nel pacchetto di Peter Flynn è riportato (con qualche modifica tipografica per farlo rientrare nella giustezza di questo documento) nel codice 1.

Mi sono domandato se non fosse possibile fare tutto con il solo ambiente *picture* senza ricorrere al dimensionamento dei caratteri. Dubito che la bandiera dell'Unione Europea sia da disegnare in linea con il testo; d'altra parte mi rendo conto che, se la bandiera deve essere disegnata in un frontespizio o in un poster dove si usano caratteri cubitali, è importante avere la possibilità di disegnare la bandiera delle dimensioni opportune, per esempio quelle delle altezze dei caratteri cubitali.

Il problema, comunque, rimane il disegno della stella a cinque punte.

Mi è venuto in mente che questa stella ha i vertici coincidenti con quelle del pentagono circoscritto e che il pentagono è alla base della definizione del numero aureo e della sezione aurea.

Così ho riscritto il codice che presento in questo articolo; mi è servito molto per ripassare i concetti relativi ai pentagoni e alla sezione aurea che sono alla base di questo codice.

1 L'ambiente *picture* esteso

Come è noto, l'ambiente *picture*, definito nel kernel di LATEX fin dalle origini, è in disuso; la maggior parte degli utenti di LATEX non lo conosce per niente, oppure pensa che non serva assolutamente a nulla, visto che esistono sia il pacchetto PSTricks, completo di decine di moduli per applicazioni speciali che permettono di disegnare praticamente tutto, sia il pacchetto TikZ, che rivaleggia con PSTricks. Quest'ultimo ha il "difetto" di servirsi del linguaggio PostScript, che non può venire interpretato direttamente da nessuno dei programmi di composizione basati sul mark up LATEX, perciò richiede il passaggio attraverso il *postprocessing* eseguito con il programma dvips e, eventualmente, dall'ulteriore elaborazione con il programma ps2pdf. Il pacchetto

```
\NeedsTeXFormat{LaTeX2e}[2016/02/01]
\ProvidesPackage{euflag}[2019/02/02 v0.4
 European Union Flag]
\def\CPK@thispackage{euflag}
\edef\CPK@thispackage{%
   \meaning\CPK@thispackage}
\edef\CPK@thisjob{\jobname}
\edef\CPK@thisjob{\meaning\CPK@thisjob}
\ifx\CPK@thispackage\CPK@thisjob
%
\message{Option svgnames %
         not being passed to package xcolor}
\else
%
\message{Option svgnames %
         being passed to package xcolor}
\PassOptionsToPackage{svgnames}{xcolor}
\fi
%% Provide color.
\RequirePackage[svgnames]{xcolor}
  \@ifundefined{T}{%
    \newcommand{\T}[2]{{%
      \fontencoding{T1}\selectfont#2}}}{}
%% Provide for graphics
\RequirePackage{graphicx}
%% Provide for the AMS's symbols
\RequirePackage{amssymb}
\definecolor{PantoneReflexBlue}{HTML}{003399}
\definecolor{PantoneYellow}{HTML}{FFCC00}
\newcommand{\eustar}{%
   \scalebox{0.1}{\ensuremath{\bigstar}}}
\newcommand{\euflag}{{%
    \fboxsep0pt
    \colorbox{PantoneReflexBlue}{%
      \vbox to1em{%
        \hsize1.5em
        \parskip0pt
        \parindent0pt
        \centering
        \color{PantoneYellow}%
        \setlength{\unitlength}{1em}
        \divide\unitlength by18
        \begin{picture}(6,6)(-2,3.5)
          \put(6,0){\eustar}
          \put(5.196,3){\eustar}
          \put(3,5.196){\eustar}
          \put(0,6){\eustar}
          \put(-3,5.196){\eustar}
          \put(-5.196,3){\eustar}
          \put(-6,0){\eustar}
          \put(-5.196,-3){\eustar}
          \put(-3,-5.196){\eustar}
          \put(0,-6){\eustar}
          \put(3,-5.196){\eustar}
          \put(5.196,-3){\eustar}
        \end{picture}%
      }% end vbox
    }% end colorbox
  }% end environment
}% end command

\endinput
```

Codice 1: Il codice del pacchetto euflag di Peter Flynn

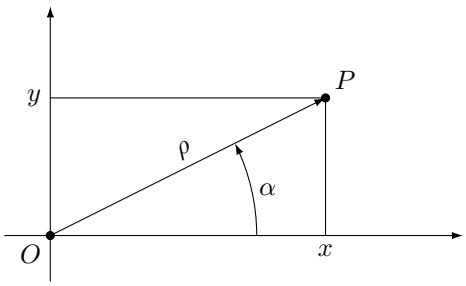


Figura 2: Coordinate cartesiane e polari

TikZ non ha questa limitazione, ma si avvicina, senza però raggiungerle, alle potenzialità strepitose di PSTricks.

Nello stesso tempo, la versione originale dell'ambiente *picture* (del vecchio LaTeX 209, defunto nel 1994 quando è stato sostituito da LaTeX 2ε) è stata estesa secondo quanto indicato dal suo creatore Lamport (1994) nella seconda edizione del suo manuale. Molte restrizioni sono state eliminate, ma prima che esistesse il pacchetto pict2e nel 2003, e successivamente ampliato fino all'edizione odierna, (Gäßlein *et al.*, 2016), non solo l'ambiente era rimasto quello di prima del 1994, ma era talmente insoddisfacente che, durante la lunga attesa del pacchetto di estensione citato, questo ambiente è caduto nel dimenticatoio.

Nel frattempo, avendo avuto bisogno di disegnarmi dei circuiti elettronici, ancor prima che comparisse pict2e, ho ritenuto opportuno estendere le funzionalità di quell'ambiente con un ulteriore pacchetto, curve2e, (Beccari, 2019), aggiornato via via che anche pict2e i suoi autori vi introducevano alcune funzionalità già presenti nel mio curve2e.

Con il pacchetto curve2e si possono specificare le coordinate dei punti in cui collocare gli oggetti, o che ne definiscono le coordinate, non solo mediante le solite coordinate cartesiane, ma anche con le quelle polari: in sostanza le coordinate cartesiane $\langle x\rangle, \langle y\rangle$ possono anche essere scritte nella forma $\langle\alpha\rangle:\langle\rho\rangle$, avendo presente che:

$$x + \mathrm{i}y = \rho\,\mathrm{e}^{\mathrm{i}\alpha}$$

La precedente espressione in termini di numeri complessi dice in modo estremamente sintetico che l'ascissa, x e l'ordinata y del punto P sono legate a ρ, la distanza del punto P dall'origine O degli assi cartesiani, e all'angolo α mediante la relazione mostrata nella figura 2.

2 La sezione aurea e il pentagono

La sezione aurea (figura 3) è il segmento $\overline{\mathsf{AC}}$, parte di un segmento $\overline{\mathsf{AB}}$, tale che

$$\frac{\overline{\mathsf{AC}}}{\overline{\mathsf{AB}}} = \frac{\overline{\mathsf{CB}}}{\overline{\mathsf{AC}}} \tag{1}$$

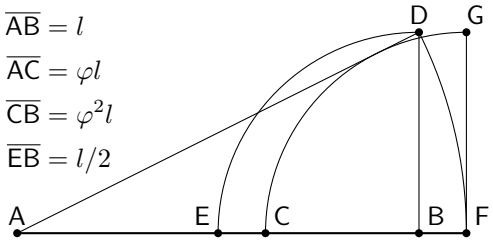


Figura 3: La sezione aurea

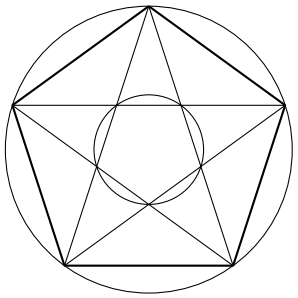


Figura 4: Il pentagono e le sue diagonali

Questo rapporto φ, minore di 1, prende il nome di *sezione aurea*; se si mette il tutto in formule, l'equazione (1), che a parole si esprime con la frase "la parte piccola sta alla parte grande come la parte grande sta al tutto", diventa:

$$(1-\varphi):\varphi=\varphi:1 \quad \text{cioè} \quad 1+\varphi=\frac{1}{\varphi} \qquad (2)$$

Risolvendo e scartando la soluzione negativa, si trova che

$$\varphi=\frac{\sqrt{5}-1}{2}\approx 0{,}618\,034\ldots \qquad (3)$$

La figura 3 mostra anche la costruzione geometrica con riga e compasso: nel segmento $\overline{\mathsf{AB}}$ di lunghezza l si determina il centro E; con raggio $\overline{\mathsf{EB}}$ si intercetta sulla perpendicolare in B il punto D e lo si congiunge con il punto A, formando il triangolo rettangolo con i cateti lunghi rispettivamente l e $l/2$, per cui l'ipotenusa risulta lunga $\sqrt{5}/2$; con centro in A si riporta la lunghezza dell'ipotenusa sul prolungamento del segmento $\overline{\mathsf{AB}}$ intercettando il punto F; con centro in quest'ultimo punto, si riporta indietro l'estremità della quantità $l/2$ intercettando il punto C; ecco quindi che la lunghezza del segmento $\overline{\mathsf{AC}}$ vale proprio $(\sqrt{5}-1)l/2$, cioè, per la formula (3), proprio φl. Per l'equazione (1), il restante segmento $\overline{\mathsf{CB}}$ è lungo $\varphi^2 l$.

Il reciproco della sezione aurea si chiama *numero aureo*; la proprietà di questi due numeri è che sono, appunto, uno il reciproco dell'altro, ma la mantissa (la parte fratta) del numero aureo è uguale alla sezione aurea.

Nel passato si è discusso a lungo sulla perfezione di questo rapporto geometrico; gli architetti dell'antichità classica erano piuttosto famosi nel proporzionare le loro opere basandosi sulla proporzione aurea; Leonardo da Vinci stesso, nel disegnare l'*Uomo di Vitruvio*, ha identificato diverse proporzioni del corpo umano legate alla sezione aurea; Luca Pacioli, l'aio di Guidubaldo di Montefeltro, scrisse all'inizio del 1500 un libro, *De divina proportione*, sulla sezione aurea; inoltre, Leonardo fornì a Luca Pacioli i disegni prospettici dei poliedri platonici e di altri solidi particolari con facce formate da soli poligoni regolari. Fra questi primeggia il pentagono che ha notevoli legami con la sezione aurea.

Con riferimento alla figura 4 si possono dimostrare le seguenti proprietà.

1. Il lato del pentagono è φ volte la lunghezza della corda maggiore.
2. Le corde maggiori del pentagono si incrociano in 5 punti che definiscono un piccolo pentagono interno a un cerchio il cui raggio è φ^2 volte il raggio del cerchio circoscritto al tutto.
3. Gli angoli al centro sottesi da ogni lato del pentagono ammontano a $72^\circ = 4\times 18^\circ$ e il $\sin 18^\circ = \varphi/2$.
4. Il rapporto fra la lunghezza della corda maggiore e il diametro del cerchio circoscritto ammonta a $\cos 18^\circ$.

3 La stella a cinque punte

Le diagonali maggiori del pentagono formano una figura che, se si tolgono i lati del pentagono interno, delimitano proprio la stella a cinque punte, quella che ci interessa per disegnare la bandiera europea.

Le coordinate polari dei vertici concavi e di quelli convessi della stella, riferite a un cerchio circoscritto di raggio unitario, hanno alternativamente il valore 1 e il valore $\varphi^2 = 0{,}381\,966\ldots$; i loro angoli polari differiscono tutti l'uno dall'altro di 36°.

Il contorno della stella, dunque può essere descritto dal "poligono" definito dal codice seguente:

```
\edef\euroflag@r{% $\varphi^2$
  \fpeval{round((sqrt(5)-1)^2/4,5)}}
\polygon(54:\euroflag@r)(90:1)%
        (126:\euroflag@r)(162:1)%
        (198:\euroflag@r)(234:1)%
        (270:\euroflag@r)(306:1)%
        (342:\euroflag@r)(378:1)}%
```

che genera la stella riportata in grande nella figura 5 all'interno del suo cerchio circoscritto. Va notato che il comando `\fpeval`, definito tramite il linguaggio L3 dal pacchetto `xfp`, è in grado di calcolare una espressione mediante aritmetica in virgola mobile (`fp`: *floating point*) e nel nostro caso l'espressione è proprio quella di φ^2 arrotondata a cinque cifre.[1]

1. Ringrazio vivamente Enrico Gregorio che mi ha segnalato il pacchetto `xfp` e mi ha indicato come usarlo in pratica.

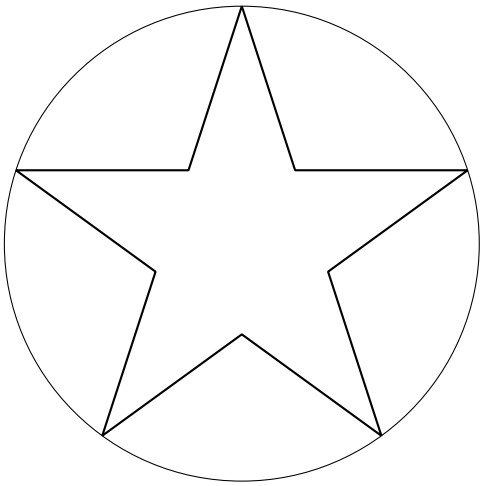


Figura 5: Il contorno della stella

Il comando `\polygon` è definito all'interno del pacchetto pict2e, ma richiede le coordinate cartesiane per i suoi vertici; nel pacchetto curve2e esso è ridefinito in modo da accettare anche le coordinate polari. Ovviamente, questo vale anche per il corrispondente comando `\polygon*`, che disegna un poligono senza bordo ma con l'interno colorato.

Non è difficile definire un comando `\euroflag@star` che contenga il codice precedente, nel quale sia aggiunta la specificazione per il colore giallo e sia aggiunto l'asterisco al comando `\polygon`. I dettagli sono contenuti nel codice completo esposto nel paragrafo 5.

4 Il disegno della bandiera

Le norme per il disegno della bandiera richiedono l'impostazione dei colori prescritti; si possono definire mediante le funzionalità pacchetto xcolor, che accetta il modello HTML uguale a quello prescritto per i colori Pantone.

```
\definecolor{BluEU}{HTML}{003399}%
\definecolor{GialloEU}{HTML}{FFCC00}%
```

La bandiera ha un rapporto base/altezza pari a 3/2; il raggio del cerchio circoscritto ad ogni stella deve essere 1/18 dell'altezza della bandiera; è quindi conveniente definire per il disegno da eseguire all'interno dell'ambiente *picture* l'unità di lunghezza pari a 1/18 dell'altezza della bandiera, così tutto il disegno viene in proporzione a tale altezza, che può essere liberamente specificata dall'utente. Conviene impostarne il valore di default pari ad 1 em, in modo che il comando per disegnare la bandiera possa essere proporzionato alle dimensioni dei font (cubitali) usati in determinati contesti. Si vedano i dettagli nel paragrafo 5.

Per ora ci basta sapere che il disegno da eseguire con l'ambiente *picture* deve avere le dimensioni di 27 unità per 18 unità. Conviene anche che l'origine degli assi cartesiani del disegno sia posta al centro del rettangolo.

Per le norme dell'Unione Europea, le stelline devono essere disposte ai vertici di un dodecagono regolare iscritto in un cerchio di raggio pari a sei unità; Peter Flynn, usando l'ambiente *picture* nativo del kernel di LaTeX, ha dovuto calcolarsi le coordinate cartesiane dei dodici vertici del dodecagono, come si vede nelle righe 44–55 del suo codice 1; usando le coordinate polari e il pacchetto curve2e, è invece facilissimo partire dal primo vertice e ripetere la stessa posizione aggiungendo via via 30° all'angolo di ciascun punto; detto `\euroflag@star` il comando che disegna la stella, è semplice usare un ciclo "while" per eseguire questa operazione.

Il ciclo "while" è già definito nel kernel di LaTeX, ma è protetto con il carattere `@` nel suo nome; inoltre è un po' più ruspante degli analoghi cicli definiti dai pacchetti calc e etoolbox, quindi gli utenti non sono soliti servirsene direttamente. Costruendo il pacchetto euroflag, come ho fatto io, non è un problema servirsi del comando del kernel, perché nei file `.sty` il carattere `@` è considerato alla stregua di una lettera.

Per mettere in posizione le dodici stelle basta usare il semplice brano di codice seguente:

```
\def\A{0}%
\@whilenum\A<360 \do{%
  \put(\A:6){\euroflag@star}%
  \edef\A{\the\numexpr\A+30}%
}%
```

Non è il caso di preoccuparsi delle definizioni di basso livello usate, perché queste definizioni sono usate all'interno di un ambiente e quindi rimangono locali e, eventualmente, riprendono i loro significati originali nel momento in cui l'ambiente viene chiuso.

5 Il codice completo

Mettendo insieme la varie parti descritte nei paragrafi precedenti, ho creato un piccolo file di estensione, `euroflag.sty`, che contiene il codice seguente.

```
\ProvidesPackage{euroflag}[2019-03-03 v.1.1
  Draws the European flag]
\NeedsTeXFormat{LaTeX2e}[2018-01-01]
\RequirePackage{curve2e,graphicx,xcolor,
                xparse,xfp}
%
\definecolor{BluEU}{HTML}{003399}
\definecolor{GialloEU}{HTML}{FFCC00}
%
\edef\euroflag@r{% $\varphi^2$
  \fpeval{round((sqrt(5)-1)^2/4,5)}}
%
\newcommand{\euroflag@star}{%
  \color{GialloEU}%
  \polygon*(54:\euroflag@r)(90:1)%
           (126:\euroflag@r)(162:1)%
           (198:\euroflag@r)(234:1)%
```

```
            (270:\euroflag@r)(306:1)%
            (342:\euroflag@r)(378:1)%
}
%
\NewDocumentCommand\EUflag{O{1em} O{0}}{{%
  \unitlength=\dimexpr#1/18\relax
  \raisebox{#2\unitlength}{%
  \begin{picture}(27,18)(-13.5,-9)
  \put(-13.5,-9){\color{BluEU}%
    \rule{27\unitlength}{18\unitlength}}
  \def\A{0}%
  \@whilenum\A<360 \do{%
    \put(\A:6){\euroflag@star}%
    \edef\A{\the\numexpr\A+30}%
  }%
  \end{picture}%
  }}}
%
\endinput
```

Lo sfondo della bandiera è ottenuto con un comando `\rule` che ha esattamente le dimensioni della bandiera. I colori sono specificati globalmente; le stelle sono messe in posizione con il comando `\put`.

La decisione di usare le funzionalità del pacchetto xparse (Beccari, 2018) per la definizione del comando `\EUflag` non è strettamente indispensabile, ma semplifica la gestione del codice; consente infatti di descrivere i due argomenti del comando come facoltativi, da racchiudere fra parentesi quadre e a cui assegnare i valori di default rispettivamente pari a 1 em e a zero unità grafiche.

Le sintassi possibili sono pertanto solo le tre seguenti:

`\EUflag`
`\EUflag[`⟨*altezza*⟩`]`
`\EUflag[`⟨*altezza*⟩`][`⟨*sollevamento*⟩`]`

La prima forma disegna una bandiera alta come 1 em del font corrente; la seconda forma disegna una bandiera di ⟨*altezza*⟩ specificata; la terza forma permette di abbassare il disegno (specificando un ⟨*sollevamento*⟩ negativo) in modo, per esempio, che la stella più in basso giaccia sulla linea di base del testo circostante; in questo caso il ⟨*sollevamento*⟩ deve essere pari a `-2`.

Non ci sono, insomma, cose misteriose; il codice è chiaro. Non voglio fare nessun confronto con il codice del pacchetto euflag, se non notare che, tolti tutti, o quasi tutti, i commenti, le righe di codice (leggermente editate) di euflag sono circa 60, mentre quelle di euroflag sono solo circa 30. Peter Flynn aveva le sue ragioni per scrivere un codice così ampio e sono descritte nel file di documentazione di euflag. Forse merita notare anche che per sollevare o abbassare la bandiera rispetto alla linea di base, con il codice di Peter Flynn bisogna usare il comando `\raisebox` esplicitamente, mentre con il mio codice basta esprimere un'opzione. Non è sicuramente una differenza sostanziale, ma è facile da ottenere con la sintassi dei comandi di definizione di xparse; è invece impossibile da ottenere con un solo comando del kernel di LaTeX.

Io non ho incontrato con il mio codice nessuno degli inconvenienti descritti da Peter Flynn per il suo codice. Tuttavia, è possibile che io non abbia usato il mio pacchetto tanto quanto lui ha usato il suo, magari in circostanze nelle quali io non mi sono mai trovato.

Merita anche notare che tutti i disegni presentati in questo articolo sono stati eseguiti mediante l'ambiente *picture* esteso con le funzionalità di curve2e. Il che dimostra che, adeguatamente esteso, il vecchio ambiente *picture* non è quell'inutile giocattolino di cui, più o meno, ci siamo dimenticati tutti.

6 La bandiera conclusiva

Essa è ottenuta semplicemente inserendo il comando `\EUflag[50mm]` dopo il titolo del paragrafo.

Riferimenti bibliografici

Beccari, Claudio (2018). «Introduzione al pacchetto `xparse`». *ArsTEXnica*, (26), pp. 16–29.

— (2019). *The extension package curve2e*. TUG. Leggibile con `texdoc curve2e`.

Flynn, Peter (2019). *The euflag LaTeX 2ε package*. TUG. Versione 0.4β; Leggibile con `texdoc euflag`.

Gässlein, Hubert, Rolf Niepraschk e Josef Tkadlec (2016). *The pict2e package*. TUG. Leggibile con `texdoc pict2e`.

Lamport, Leslie (1994). *A document preparation system — LaTeX — User's guide and reference manual*. Addison Wesley, Reading, Mass., 2ª edizione.

▷ Claudio Beccari
`claudio dot beccari at gmail dot com`

Parsing di opzioni in LuaTEX

Roberto Giacomelli

Sommario

In questo lavoro esploreremo come implementare un parser in Lua, il linguaggio di programmazione elegante e facile da imparare incluso in LuaTEX, per una lista di opzioni nel formato ⟨*chiave*⟩=⟨*valore*⟩.

Un simile componente è utile nello sviluppo di pacchetti, perché fornisce un sistema di opzioni progettato per l'efficienza e l'espressività della sintassi.

Abstract

This paper explains how to implement a parser in Lua—the elegant and easy to use programming language included in LuaTEX—for a list of options in the ⟨*key*⟩=⟨*value*⟩ format.

A similar parser is useful in the package development allowing an option system specifically designed for efficiency and syntax expressiveness.

1 Motivazione

Lentamente sto acquisendo una maggiore consapevolezza delle potenzialità del motore di composizione LuaTEX rispetto al tradizionale pdfTEX. Ho scoperto dapprima la tecnologia dei nodi, ovvero tutte quelle funzionalità che fanno capo alla libreria `node` e che consentono di costruire oggetti tipografici nativi programmando in Lua. Ho poi approfondito le questioni, se vogliamo più tecnologiche, del caricamento di librerie esterne non specifiche del sistema TEX e capaci di leggere le informazioni presenti in database anche di livello enterprise, ovvero ciò che rende l'utente in grado di costruire splendidi report preparando un sorgente TEX.

Ogni volta ne ho dato conto con un articolo su ArsTEXnica, la prestigiosa rivista del Gruppo Utilizzatori Italiani di TEX, sia per condividere e ricambiare contributi ricevuti, sia per mettere nero su bianco le cose a mia migliore comprensione e a futuro riferimento.

Oggi, sto gradualmente spostando l'attenzione verso lo sviluppo applicativo, cercando di dare concretezza a un progetto a medio termine. L'idea è impegnativa, ma di grande soddisfazione: creare funzioni avanzate in LuaTEX che anche gli altri utenti possano utilizzare componendo i loro documenti nel formato LATEX o ConTEXt.

Alla pagina web `https://github.com/robitex/barracuda` trovate la casa virtuale del progetto, che ho chiamato `Barracuda` per assonanza con il termine *barcode* e che implementa il disegno di alcune di questi simboli, che compaiono praticamente sull'etichetta di ogni prodotto.

Uno dei componenti del progetto sarà una classe di documento chiamata `labelreport` per la generazione di etichette, un elemento tipografico molto particolare perché comprende sia la grafica che la gestione dati.

In questo articolo non mi soffermerò sui dettagli del codice, quasi tutto scritto in Lua, ma illustrerò come poter risolvere proprio con questo linguaggio un problema molto importante, perché determina la qualità dell'esperienza utente in termini di semplicità di configurazione tipografica: l'insieme di opzioni ⟨*chiave*⟩=⟨*valore*⟩.

Lo scopo del presente breve articolo sarà quello di illustrare lo sviluppo in Lua di un modulo per la valutazione di questi elenchi di opzioni. Proveremo a sperimentare il setup di opzioni di un comando come il seguente, analogamente a quanto già è in grado di fare il pacchetto per LATEX `xkeyval`, per la definizione geometrica della pagina di etichette:

```
\setupGrid{
    xsep = 12mm,
    ysep = 5mm,
    showgrid,
    name = label21,
    shape = rect(68mm, 38mm)
}
```

Molti pacchetti offrono agli utenti una sintassi di configurazione simile, per esempio `siunitx` per la corretta composizione di numeri e unità di misura del Sistema Internazionale, o `pgf-tikz`, notissimo per la produzione della grafica, solo per citarne un paio. A volte gli Autori implementano il codice di gestione delle opzioni per proprio conto, a volte invece contano sulle librerie LATEX3.

Queste ultime offrono la soluzione migliore perché se tutti usassero L3 per fornire funzionalità di configurazione delle macro dei propri pacchetti, gli utenti si troverebbero a utilizzare un'interfaccia consistente e già nota. Tuttavia, è anche probabile che chi scrive pacchetti in Lua per LuaTEX desideri ricevere i parametri utente come variabili Lua.

La facilità di scrivere codice in Lua rispetto a TEX determina comunque un vantaggio importante per una libreria che dovesse fornire funzioni di lettura di sistemi di opzioni, in particolare perché è più semplice implementare nuove idee, a tutto vantaggio degli utenti.

Cercherò di illustrare alcune idee in Lua con l'obiettivo sperimentale di valutare l'impegno neces-

sario per implementare una libreria per la lettura di valori di configurazione utente secondo una sintassi ⟨*chiave*⟩=⟨*valore*⟩, nonché valutarne la semplicità d'uso dal punto di vista di chi scrive pacchetti.

1.1 Note e requisiti

Per la comprensione del codice riportato nelle prossime sezioni è preferibile che il lettore abbia un minimo di dimestichezza col linguaggio Lua, che si acquista senza particolari difficoltà grazie alla sua semplicità.

Troverete inoltre spesso la macro `\directlua`. Si tratta di una primitiva espandibile di LuaTEX che invoca l'interprete Lua con il codice rappresentato dal proprio argomento. Alcuni articoli apparsi su questa rivista, inoltre, contengono l'illustrazione di base delle caratteristiche di Lua e del tipo di dato tabella, l'unico predefinito a essere strutturato e che implementa allo stesso tempo un array e un dizionario.[1]

Saranno quindi utili riferimenti sia il testo ufficiale su Lua (Ierusalimschy, 2016), sia il manuale di LuaTEX (The LuaTEX development team, 2018). Rimando tuttavia al forum GUIT per le domande che eventualmente fossero ancora rimaste senza risposta.

2 Sviluppo

La sintassi prevista per un'opzione consiste nello specificare un nome seguito dal segno di uguale, seguito a sua volta dal valore. Ogni coppia ⟨*chiave*⟩=⟨*valore*⟩ deve essere separata da una virgola.

Non c'è alcuna necessità di implementare codice di parsing per un tale elenco di opzioni, poiché Lua stesso ne implementa già uno: il costruttore di tabelle. All'ipotetico comando della sezione precedente `\setupGrid` potremo passare opzioni racchiudendo i valori di tipo stringa tra doppi apici e assegnando alle alternative valori del tipo vero (`true`) o falso `false`:

```
\setupGrid{
    xsep = 12,
    ysep = 5,
    showgrid = true,
    name = "label21",
    shape = {rect = {68, 38}}
}
```

Per catturare queste opzioni è sufficiente il codice di questo sorgente minimo:

```
% !TeX program = LuaTeX
\def\setupGrid#1{\directlua{
    local t = {#1}
    % print some options
    local par = string.char(92).."par"
    tex.print("xsep = " .. t.xsep)
    tex.print(par)
    tex.print("name = " .. t.name)
    tex.print(par)
    tex.print("shape.rect[1] = " ..
    t.shape.rect[1])
}}
\setupGrid{
    xsep = 12,
    ysep = 5,
    showgrid = true,
    name = "label21",
    shape = {rect = {68, 38}}
}
\bye
```

Tuttavia, l'utente LaTeX non è abituato a racchiudere testo tra doppi apici; non si possono specificare unità di misura per le dimensioni né esprimere enumerazioni come quella dell'opzione `shape`, con la sintassi del costruttore di tabelle Lua.

Inoltre, definendo particolari sintassi per le opzioni potremo rendere più intuitivo il sistema, se solo fosse disponibile una funzione di parsing corrispondente.

2.1 Un parser byte per byte

Un *parser* è una funzione che ricava dati strutturati dalla lettura di un testo. Per implementarne uno possiamo usare apposite librerie, per esempio LPEG[2] inclusa in LuaTEX oppure affidarci alle espressioni regolari.

Preferisco, tuttavia, usare un algoritmo *byte per byte*, che consiste nel leggere un carattere alla volta dall'input e dedurne i dati, esercitando così un controllo minuzioso sulla sintassi e senza dover utilizzare le funzioni di LPEG, per me poco intuitive. In un secondo momento, poi, è sempre possibile definire grammatiche con questa libreria per un codice più veloce.

Per l'opzione semplice dove esiste la chiave e il valore è un semplice testo, la funzione potrebbe essere:

```
local function parseOption(s) --> key, val
    local isKeyParse = true -- state: key
    local key, val
    for c in s:gmatch(".") do
        if isKeyParse then -- read a key
            if c ~= " " then
                if c == "=" then
                    isKeyParse = false
                else
                    key = (key or "") .. c
                end
            end
        else -- read a value
            val = (val or "") .. c
        end
    end
    return key, val
end

print(parseOption "xsep = 12mm")
```

1. La struttura dati dizionario è detta anche 'array associativo'.

2. http://www.inf.puc-rio.br/~roberto/lpeg/lpeg.html.

La funzione utilizza un iteratore sul singolo carattere della stringa in ingresso e il valore booleano di stato `isKeyParse` è vero se stiamo leggendo il nome dell'opzione, falso se invece ne stiamo leggendo il valore. Nel primo caso, il carattere viene concatenato alla stringa `key`, altrimenti alla stringa `val`.

In particolare, quando incontriamo il carattere di uguale (`=`) lo stato `isKeyParse` è impostato su `false`, mentre ignoriamo completamente i caratteri spazio solo se stiamo leggendo il nome dell'opzione.

Passiamo ora alla lista di opzioni modificando il codice in questo modo, inserendo il codice Lua in una definizione di comando tramite la primitiva ormai nota `\directlua`:

```
% !TeX program = LuaTeX
\long\def\setupGrid#1{\directlua{
local function parseOption(s) % -> topt, err
    local isKeyParse = true % state: key parsing
    local topt = {}
    local i = 0 % state: option list number
    for c in s:gmatch(”.”) do
        if c == ”,” then
            if i == 0 then
                return nil, ”Empty first option”
            end
            i = i + 1 % new option
            topt[i] = {}
            isKeyParse = true
        elseif c == ”=” then
            isKeyParse = false
        else
            if isKeyParse then % read a key
                if i == 0 then
                    i = 1
                    topt[i] = {}
                end
                if not (c == ” ”) then
                    local t = topt[i]
                    t.key = (t.key or ””) .. c
                end
            else % read a value
                local t = topt[i]
                t.val = (t.val or ””) .. c
            end
        end
    end
    return topt
end
local s = [===[#1]===]
local topt, err = parseOption(s)
if err then error(err) end
for _, t in ipairs(topt) do
    tex.print(t.key..” = ”..t.val..”, ”)
end
}}

\setupGrid{
    xsep = 12mm,
    ysep = 5mm,
    showgrid = true,
    name = label21,
    shape = rect(68mm 38mm)
}
\bye
```

Si tratta ancora di codice grezzo, per esempio non possiamo usare la virgola per separare i vari valori associati all'enumerazione `rect` in `shape`, ma stampa correttamente i nomi e i valori corrispondenti delle opzioni. Se si compila il sorgente con LuaTEX, lo si può verificare.

2.2 Definizione di opzioni

A questo punto diventa chiaro che abbiamo la necessità di associare ulteriori informazioni al sistema delle opzioni, per esempio, per far sì che il parser possa segnalare nomi di opzioni non valide, o, cosa più importante, implementare sintassi più intuitive.

Con una tabella Lua è possibile descrivere la struttura di un'opzione in modo semplice e completo. Al campo `default` potremo far corrispondere il valore da assegnare all'opzione se l'utente non lo esplicita, e interpretarne l'assenza con la sua obbligatorietà.

Il campo `optype` potrebbe definire il tipo del valore associato all'opzione, per esempio una dimensione, e il campo `fncheck` potrebbe contenere una funzione Lua da applicare al valore specificato dall'utente per il controllo di validità e prima della memorizzazione.

Per l'opzione `xsep` la descrizione precedente si concretizza nel codice:

```
xsep = {
    default = 0.0,
    optype = ”dim”,
    fncheck = function (v)
        return v >= 0
    end,
}
```

Per le enumerazioni, che definirò compiutamente nella prossima sezione, la struttura di definizione potrebbe essere la seguente:

```
shape = {
    optype = ”enum”,
    enum = {
        rect = {
            {arg = ”x”, optype = ”dim”},
            {arg = ”y”, optype = ”dim”},
        },
        roundrect = {
            {arg = ”x”, optype = ”dim”},
            {arg = ”y”, optype = ”dim”},
            {arg = ”r”, optype = ”dim”},
        },
        circle = {
            {arg = ”d”, optype = ”dim”},
        },
    },
}
```

Si noti come in Lua sia possibile assegnare a campi di tabella valori che corrispondono a funzioni

per mezzo della sintassi anonima. Anzi, in Lua le funzioni sono valori di prima classe, perciò non hanno un nome, poiché la sintassi tradizionale è convertita dall'interprete in sintassi anonima per far sì che le definizioni seguenti siano equivalenti:

```
function nome (arg)
    -- body
end

nome = function (arg)
    -- body
end
```

Una volta ottenuto il nome dell'opzione leggendola dalla stringa di input inserita dall'utente, il parser può ottenerne la corrispondente definizione e proseguire con la lettura del valore con la specifica funzione.

2.3 Interazione tra opzioni

Possiamo ora comprendere più chiaramente quello che abbiamo chiamato il 'sistema delle opzioni', composto dai seguenti tre distinti componenti:

1. definizione delle opzioni in base al tipo;
2. parsing del testo con validazione e ritorno dei valori associati;
3. memorizzazione dei valori.

Non rimane quindi che soffermarci sulla terza fase, quella della memorizzazione. Ottenuti i parametri, è compito del pacchetto applicativo rielaborarli in una fase successiva e indipendente da quella del parsing, acquisendoli all'interno del proprio stato.

Questo, oltre alla memorizzazione, può comprendere anche una rielaborazione, per esempio per tenere conto di *interazioni* tra parametri. In una prima fase di implementazione del parsing (l'interazione tra parametri) può essere vantaggiosamente demandata al pacchetto applicativo per mantenere un contesto più semplice. In seguito, potrebbe essere inclusa nel parsing stesso attraverso funzioni da aggiungere alla definizioni delle opzioni.

Un esempio d'interazione tra opzioni è dato dal voler includere l'unità di misura delle dimensioni in modo che l'utente possa omettere quest'informazione nel caso che le misure da inseririre fossero molte. L'ulteriore opzione `unit`, disponibile per esempio in `pstricks`, tra i parametri della griglia consentirebbe di scrivere il seguente codice utente:

```
\setupGrid{
    unit = mm,
    xsep = 12,
    ysep = 5,
    shape = rect(68, 38)
}
\bye
```

L'interazione potrebbe anche essere più complessa, per esempio quando il valore dato dall'utente a un parametro, che chiameremo principale, determina la validità o meno di quello assegnato a un secondo parametro che chiameremo secondario. Questa dipendenza può essere implementata senza alcun problema in Lua modificando le funzioni `fncheck` nelle definizioni delle opzioni e aggiungendo un campo intero che determini l'*ordine* di chiamata delle stesse funzioni di verifica.

In questo modo, il sistema di opzioni passa da un insieme di parametri indipendenti a un grafo di nodi dipendenti che deve essere risolvibile, ovvero deve esistere un percorso di visita del grafo stesso che connetta tutti i parametri, da quelli principali a quelli secondari.

L'ordine di verifica stabilisce a tutti gli effetti la risoluzione del grafo di opzioni solamente nella procedura di verifica.

2.4 Meta opzioni

Ulteriore idea è quella di consentire all'utente di influenzare il parser di opzioni con specifici parametri. Diventerebbe possibile per esempio specificare il carattere che funge da separatore per le coppie ⟨*chiave*⟩=⟨*valore*⟩, indicare come considerare i caratteri spazio per il valore o applicare un eventuale arrotondamento ai valori numerici.

Questo tipo di regolazione influenza la sintassi con cui il parser legge le opzioni, nella maggioranza dei casi per evitare i casi in cui i caratteri di separazione confliggono con i valori.

Rimane da valutare con ulteriori approfondimenti se queste metaopzioni debbano essere impostate dal pacchetto oppure dall'utente. Infatti, è sempre possibile pensare che il pacchetto costruisca delle opzioni utente che in realtà corrispondono dietro le quinte a una o più metaopzioni per il parser.

3 Sintassi ⟨*chiave*⟩=⟨*valore*⟩

Sono previsti dal sistema diversi tipi di opzioni che l'utente dovrà conoscere e lo sviluppatore implementare. Nelle prossime sezioni proverò a definirne alcuni.

Una regola generale prevede che una data opzione debba obbligatoriamente essere valorizzata dall'utente, a meno che nella definizione non ne esista un valore di default. Quanto al carattere di separazione delle coppie ⟨*chiave*⟩=⟨*valore*⟩, la virgola, si ammette che dopo l'ultima opzione possa essercene una facoltativa.

3.1 Chiavi

L'elemento sintattico delle chiavi è definito semplicemente come un nome, inteso come una o più parole separate da spazi. Dei caratteri spazio possono essere inseriti sia prima che dopo la sequenza di parole, mentre tra le parole possono essere inseriti spazi a piacere, che saranno comunque considerati come uno solo.

Con questa definizione ammettiamo che le chiavi possano contenere spazi, come accade per le opzioni del pacchetto tikz o pgfplots. Così le seguenti sono tutte chiavi valide e diverse una dall'altra:

```
show grid
showgrid
show-grid
show_grid
ShowGrid
SHOWgrid
```

mentre le seguenti rappresentano sempre la stessa chiave:

```
show grid
   show grid
show    grid
```

Nell'ipotetico parser che legge i caratteri uno dopo l'altro, introduciamo delle funzioni Lua specializzate per leggere un dato elemento sintattico, facendo sì che il codice sia ben strutturato. Queste funzioni avranno sempre la stessa firma: gli argomenti saranno l'array con i caratteri da esaminare e l'indice di partenza, mentre il risultato restituito sarà la coppia di valori della posizione raggiunta dell'indice e il valore atteso.

È da notare che, essendo le tabelle Lua degli oggetti, le variabili ne contengono un riferimento, perciò passare tabelle a una funzione è efficiente e quindi lo è anche la soluzione di leggere l'array di caratteri anziché direttamente la stringa.

La funzione che legge dall'array una chiave di opzione in base alle regole stabilite può essere la seguente:

```
local function parseKey(t, i) --> i, key
    local k = {}
    local isInnerChar = false
    local isLastChar = true
    while true do
        if i > #t then
            break
        end
        local c = t[i]
        if c == ”,” or c == ”=” then
            break
        end
        if c == ” ” then
            if isInnerChar and isLastChar then
                isLastChar = false
                k[#k + 1] = c
            end
        else
            isLastChar = true
            isInnerChar = true
            k[#k + 1] = c
        end
        i = i + 1
    end
    local key
    local len = #k
    if len > 0 then
        if not isLastChar then
            k[len] = nil
        end
        key = table.concat(k)
    end
    return i, key
end
```

L'indice finale sarà la posizione del carattere da cui occorrerà leggere il valore dell'opzione a seconda del tipo corrispondente.

3.2 Opzioni booleane

L'opzione più semplice è quella che assume un valore vero oppure falso, acceso o spento. Seguendo il nostro sistema di esempio per la definizione della geometria delle etichette, stiamo parlando per esempio dell'opzione `showgrid`. Essa indica se la stampa del disegno della griglia regolare di etichette — che corrisponde ai loro contorni — debba o meno essere eseguita.

La sintassi base prevede che l'utente assegni il valore `true` o `false` all'opzione oppure che ne indichi solamente la chiave per attivarla. In altre parole, assegnare il valore `true` è opzionale.

Una funzione di lettura del valore è la seguente:

```
local function parseBool(t, i) --> index, bool
    local c = t[i]
    local len = #t
    if i > len or c == ”,” then
        return i, true
    end
    if c == ”=” then
        i = i + 1
        while i<= len do
            local b = t[i]
            if b ~= ” ” then
                break
            end
            i = i + 1
        end
        local res
        local concat = table.concat
        if i + 4 <= len and
            concat(t, ””, i, i+4) == ”false”
            then
            i = i + 5
            res = false
        elseif i + 3 <= len and
            concat(t, ””, i, i+3) == ”true”
            then
            i = i + 4
            res = true
        end
        while i <= len do
            local c = t[i]
            if c == ”,” then
                return i + 1,  res
            end
            if c ~= ” ” then
                return i, nil
            end
            i = i + 1
        end
```

```
        return i, res
    end
    return i, nil
end
```

Nel leggere il codice della funzione `parseBool()`, si deve tener presente che l'indice nell'array passato come argomento sarà posizionato su un segno di uguale o su una virgola, oppure alla fine dell'array, per effetto della funzione di parsing delle chiavi. L'idea, infatti, è quella di chiamare la funzione di lettura del valore subito dopo la chiamata della funzione di lettura della chiave, determinando l'avanzamento dell'indice.

3.3 Opzioni numeriche

Di base, qualsiasi parametro intero o frazionario come parti o dimensioni è rappresentato dal tipo numerico. Possiamo quindi intendere che una lunghezza come la distanza `xsep`, che separa orizzontalmente le etichette affiancate, appartenga a questo tipo anche se entrano in gioco le unità di misura.

Una funzione di lettura valore potrebbe essere la seguente:

```
local function parseDim(t, i) --> index, sp
    local c = t[i]
    local len = #t
    if i > len or c ~= "=" then
        return i, nil
    end
    i = i + 1
    local k = i
    while k < len and t[k] ~= "," do
        k = k + 1
    end
    local pos = k
    if t[k] == "," then
        pos = k - 1
    end
    local dim = table.concat(t, "", i, pos)
    return k+1, tex.sp(dim)
end
```

che molto semplicemente ricerca il carattere di separazione '`,`' oppure la fine dell'array determinando l'indice `k`, e usa la funzione LuaTEX `tex.sp()` per tradurre la stringa tra l'indice corrente e l'indice `pos` in punti scalati `sp`, l'unità di misura delle lunghezze interna a TEX (che corrisponde a 1/65536) punti tipografici.

3.4 Opzioni enumerazione

Questo tipo può assumere solamente i valori che appartengono a un dato insieme i cui elementi sono di tipo stringa. Per esempio, l'opzione `shape` può assumere i valori `rect` per rettangolo, `roundrect` per rettangolo con angoli arrotondati e `circle` per la forma circolare.

A un dato valore è anche possibile associare argomenti racchiudendoli tra parentesi tonde. Questi argomenti se previsti sono obbligatori. Così, nell'esempio pratico della forma dell'etichetta, è possibile definire larghezza e altezza del rettangolo o diametro del cerchio.

Assieme alle dimensioni di pagina e alle distanze orizzontali e verticali che separano le etichette, espresse dalle opzioni `xsep` e `ysep` rispettivamente, questi dati sono sufficienti per definire in modo completo l'esatta posizione sulla pagina di ciascuna etichetta, sapendo che la griglia è regolare e sulle righe e sulle colonne ci sono il massimo numero di etichette possibili in posizione simmetrica rispetto agli assi della pagina stessa.

Associare valori a un'enumerazione non è una novità: alcuni moderni linguaggi di programmazione (Rust per esempio) ne sono capaci attribuendovi il nome di tipi algebrici.[3]

A ben vedere le opzioni booleane sono opzioni enumerazione dove i possibili valori sono solamente `true` oppure `false`, se non fosse che la sintassi preveda opzionale specificare un valore `true`.

Mentre si lascia al lettore l'implementazione della funzione `parseEnum()`, perché simile a quello delle funzioni precedenti, nella prossima sezione presenterò un listato completo per la lettura dei valori di un dato sistema di opzioni.

3.5 Limitazioni

Una limitazione che forse potrebbe essere risolta è la defininizione di opzioni tramite espansione di macro. Non è possibile, infatti, inserire al posto di un valore una macro, perché il codice Lua non la espanderebbe a meno di non modificare opportunamente il comando TEX del parser.

4 Il parser principale

Una volta trasformata la stringa d'ingresso in array di caratteri, il codice del parser dovrà svolgere la lettura delle opzioni ⟨*chiave*⟩=⟨*valore*⟩ con i seguenti passi:

1. leggere dall'array la chiave dell'opzione;
2. determinare dalla definizione associata il tipo di dato;
3. leggere dall'array il valore per quel tipo;
4. tornare al passo iniziale con l'indice posizionato sul successivo carattere separatore oppure terminare, se si è raggiunta la fine dell'array.

Innanzi tutto predisponiamo il codice come modulo Lua contenuto in un file: dovremo memorizzare tutte le funzioni in una tabella locale e, come ultima istruzione, restituirne il riferimento. Lo schema è:

```
local lib = {}
lib.defopt = {
    ...
}
```

3. https://www.rust-lang.org/.

```
local function parseKey(t, i) --> index, key
    ...
end

local fnparselib = {}
fnparselib.bool = function (t, i) --> i, bool
    ...
end

fnparselib.dim = function (t, i) --> i, sp
    ...
end

fnparselib.enum = function (t, i) --> i, e
    ...
end

function lib:parseOption(s)
    ..
end
return lib
```

La chiave `defopt` conterrà le definzioni delle opzioni in forma di tabella, segue la dichiarazione della funzione `parseKey()` che vale per le chiavi di tutte le opzioni possibili. Le funzioni di parsing di valori, già introdotte precedentemente tranne per il tipo `enum`, sono invece incluse in una tabella indicizzata con i nomi dei tipi, in modo da facilitare l'implementazione della funzione principale di parsing chiamata `parseOption()` e che riporto di seguito:

```
function lib:parseOption(s) --> topt, err
    local t = {} -- option table
    for c in s:gmatch(”.”) do
        t[#t + 1] = c
    end
    local i = 1
    local len = #t
    local opt = {}
    local defopt = self.defopt
    while i <= len do
        local key
        i, key = parseKey(t, i)
        if not key then
            error(”Error at index ” .. i)
        end
        if not defopt[key] then
            error(”Key not found ” .. key)
        end
        local def = defopt[key]
        local optype = def.optype
        local fnparse = fnparselib[optype]
        local val
        i, val = fnparse(t, i)
        if not val then
            error(”Invalid value for ”.. key)
        end
        local fncheck = def.fncheck
        if fncheck then
            local ok = fncheck(val)
            if not ok then
                error(
                    ”Invalid value for ”..
                    key
                )
            end
        end
        opt[key] = val
    end
    return opt
end
```

Si noti nello schema che le funzioni di parsing ausiliarie sono tutte dichiarate come locali. Questo le renderà non visibili all'utente della libreria ma non alle altre funzioni grazie alla tecnologia delle *closure* di Lua.

4.1 Verifica

Proviamo la libreria con il seguente file LuaTEX che stampa i valori raccolti dalle opzioni:

```
% !TeX program = LuaTeX
\directlua{
local parselib = require ”parseopt”
local opt = parselib:parseOption([[
    ysep = 1pt  ,
    showgrid, xsep=12pt,]]
)
print()
for k, v in pairs(opt) do
    print(k, v)
end
}
\bye
```

Se compilato, il sorgente stampa in console le righe seguenti:

```
showgrid    true
xsep    786432
ysep    65536
```

Le opzioni possono essere stampate con un diverso ordine poiché la tabella di Lua intesa come dizionario non è ordinata, perciò l'iteratore `pairs()` non garantisce sempre lo stesso ordine anche a parità di contenuto della tabella. Provate a compilare più volte di seguito per constatare questa proprietà.

Se vogliamo usare una macro, potremo scrivere:

```
% !TeX program = LuaTeX
\long\def\setupGrid#1{\directlua{
local parselib = require ”parseopt”
local opt = parselib:parseOption([===[#1]===])
print()
for k, v in pairs(opt) do
    print(k, v)
end
}}

\setupGrid{
    ysep = 1pt  ,
    showgrid, xsep=12pt,}
\bye
```

4.2 Famiglie di opzioni

La libreria di parsing delle opzioni ha così raggiunto il livello minimo di implementazione con la struttura e i concetti fino a ora illustrati. Il prossimo passo è l'introduzione di una funzione che memorizzi internamente le definizioni delle opzioni.

A questo scopo, come già nel pacchetto xkeyval, è possibile introdurre sia il concetto di *famiglia*, la raccolta di un gruppo di opzioni, sia quello di *namespace*, un gruppo di famiglie.

In questo modo, il parser può organizzare le opzioni necessarie alla classe di documento o al singolo pacchetto che occupa un proprio namespace.

5 Conclusioni

Creare una chiara e intuitiva sintassi per la definizione di un sistema di opzioni basato sul formato chiave/valore può aiutare sia gli sviluppatori di pacchetti sia gli utenti. In questo lavoro introduttivo ho illustrato l'idea di *tipo* per le opzioni e presentato il codice Lua per implementare una libreria di parsing per LuaTEX generale perché in grado di utilizzare le *definizioni* delle singole opzioni.

Ho anche accennato ai problemi d'interazione tra le opzioni quando vi sono valori la cui correttezza dipende da altri parametri utente. Queste considerazioni e i brevi listati di codice mostrano come il problema di come facilitare l'utente nella configurazione di parametri sia molto più interessante di quello che ci si potrebbe attendere.

Il codice del parser qui presentato si basa sulla scansione sequenziale dei caratteri digitati dall'utente. Pur trattandosi di un problema interessante come esercizio di programmazione utile ad acquisire dimestichezza con Lua, potrebbe essere sostituito da sorgenti predisposti per l'uso di LPEG, abbandonando algoritmi che possiamo definire di basso livello, e senza che cambi l'interfaccia.

I soci GUIT potranno scaricare i file dei sorgenti dal sito dell'associazione con le modalità previste dagli amministratori e potranno sperimentare agevolmente il codice eseguendo gli script con `texlua` e LuaTEX.

Chi è interessato al progetto `Barracuda` che includerà la libreria di parsing di un sistema di opzioni così come accennato in questo lavoro, visiti periodicamente la pagina web `https://github.com/robitex/barracuda`.

6 Ringraziamenti

Ringrazio la redazione della rivista ArsTEXnica per l'opera di revisione che ha migliorato questo lavoro, oltre che, naturalmente, tutti gli altri membri dello staff. Ringrazio anche Luigi Scarso per il suo costante appoggio nello sviluppo di nuovi progetti basati su LuaTEX.

Riferimenti bibliografici

Ierusalimschy, Roberto (2016). *Programming in Lua*. Lua.org, 4ª edizione.

The LuaTEX development team (2018). *LuaTEX Reference Manual*, 1.0.7 edizione.

▷ Roberto Giacomelli
Carrara
`giaconet dot mailbox at gmail dot com`

Antichi sistemi di notazione musicale

Jean-Michel Hufflen[*]

Sommario

Alcuni programmi per la composizione in notazione musicale — MusiXTEX o LilyPond, per esempio — permettono di rappresentare la notazione quadrata del canto gregoriano su tetragrammi. Nella presente guida introduttiva si spiega come sono organizzati gli spartiti che adoperano questa notazione, della quale si mostra lo sviluppo fino a quelle in uso agli inizi dell'epoca barocca.

Abstract

Some music engraving programs — e.g., MusiXTEX or LilyPond — allow Gregorian chant's square notation on four-line staves to be rendered. In this tutorial we explain how scores using this notation are organised. Then we show how it developed until the notations used at the early baroque era.

1 Introduzione

Come si legge in (Hufflen, 2011), la composizione degli spartiti offre agli sviluppatori dei programmi d'incisione musicale l'opportunità di ingaggiare sfide stimolanti. Da un punto di vista storico, lo studio degli spartiti di *canto gregoriano* è interessante, dato che essi non miravano a ottenere una potenza espressiva comparabile con quella delle moderne partiture. Eppure, nonostante appartengano agli inizi dello sviluppo della musica nei Paesi occidentali, sollevano problemi tipografici tutt'altro che irrilevanti. Inoltre, alcuni programmi musicali — MusiXTEX (Taupin *et al.*, 2011), Finale (Finale, 2017), LilyPond (LilyPond, 2017), OpusTEX (Opus, 2011) e Gregorio TEX (Gregorio, 2016), per esempio — permettono di comporre spartiti con le notazione gregoriane. Ed è proprio alla loro organizzazione, dunque, che la presente guida si propone di introdurre il lettore.

La sezione 2 tratta il posto occupato dal canto gregoriano nella storia della musica. Nella sezione 3, poi, si descrivono nel dettaglio le diverse notazioni impiegate negli anni. Alcune di quelle introdotte allora vengono adoperate ancora oggi, anche se qualcuna molto di rado: questo punto viene esplorato brevemente nella sezione 4. La sezione 5, infine, illustra le caratteristiche principali di alcuni programmi di notazione musicale in grado di comporre spartiti gregoriani.

La lettura di quest'articolo richiede una conoscenza musicale appena elementare, ma i lettori interessati a definizioni musicologiche più precise possono consultare (Arnold, 1983; Jacobs, 1988). Chi comprende il francese, invece, può trovare la storia completa delle prime forme musicali in (Chailley, 1967, 1972), mentre la relazione tra le forme musicali successive e gli eventi storici è descritta brillantemente in (Chailley, 1984, 1985).

2 Che cos'è il canto gregoriano?

Da un punto di vista terminologico, l'espressione 'canto piano'[1] indica complessivamente i canti adoperati nelle liturgie della Chiesa occidentale, in particolare di quella romana cattolica.[2] Il canto piano è *monofonico* e consisteva di linee melodiche singole e prive di accompagnamento musicale — cioè era *a cappella*, come si dice tecnicamente. Il *canto gregoriano* è la varietà di canto piano più diffusa, ma non l'unica: prima di esso, infatti, esistevano almeno quelle elencate di seguito.

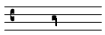
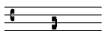
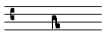
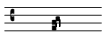
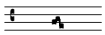
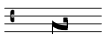
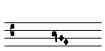
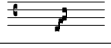
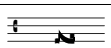
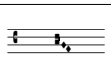
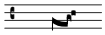
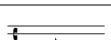
- Il *canto romano antico*, cioè il repertorio del rito romano della chiesa paleocristiana, a partire dal IV secolo.
- Il *canto gallicano*, adoperato nelle terre dei Franchi prima del regno di Carlo Magno, a partire dall'VIII secolo.
- Il *canto ambrosiano*, dal nome del vescovo Ambrogio (340–397 circa), originario di Milano e giunto alla maturità a partire dall'VIII secolo.
- Il *canto beneventano*, simile al canto precedente, sviluppatosi nei secoli VII–VIII e praticato nel ducato di Benevento, a nord-est di Napoli.
- Il *canto mozarabico*, adoperato dai cristiani in Spagna quando il paese era sotto il dominio arabo (dopo il 711), include elementi relativi alla Chiesa visigotica e si basa principalmente su vocalizzazioni simili a improvvisazioni.[3]

Le differenze tra queste varietà sono individuabili nei diversi stili musicali di ciascun canto, nei testi e nella loro posizione all'interno dei servizi

[*] Quest'articolo costituisce il contributo di Jean-Michel Hufflen al BachoTEX 2018. È stato tradotto da Tommaso Gordini con il permesso dell'autore. Eventuali errori o fraintendimenti del testo originale sono del traduttore. (*N.d.R.*)

1. Dal latino *cantus planus*.
2. Al contrario, il *canto bizantino* — proveniente dall'Impero bizantino e adoperato nelle liturgie della Chiesa ortodossa orientale — non è classificato come canto piano.
3. In effetti, 'mozarabico' è un termine improprio, dato che questo canto non rivela alcuna influenza araba.

Tabella 1: Neumi fondamentali.

Note	Dir.	Neuma	Esempio	Codice LilyPond
1	–	*punctum*		`si`
		virga		`\[ \virga si \]`
2	→	*bipunctum*		`si si`
	↑	*podatus*		`\[ sol \pes si \]`
	↓	*clivis*		`\[ si \flexa sol \]`
3	↑↑	*scandicus*		`\[ sol \pes la \virga si \]`
	↑↓	*torculus*		`\[ la \pes si \flexa sol \]`
	↓↑	*porrectus*		`\[ la \flexa sol \pes si \]`
	↓↓	*climacus*		`\[ \virga si \inclinatum la \inclinatum sol \]`
4	↑↑↑	*virga praetripunctis*		`\[ mi \pes sol \pes la \pes do' \]`
	↑↑↓	*scandus flexus*		`\[ sol \pes la \virga si \pes \auctum \descendens sol \]`
	↑↓↑	*torculus resupinus*		`\[ fa \pes la \flexa fa \pes sol \]`
	↑↓↓	*pes subtripunctis*		`\[ si \pes do' \inclinatum la \inclinatum sol \]`
	↓↑↑	*porrectus resupinus*		`\[ la \flexa sol \pes \auctum \descendens si do' \]`
	↓↑↓	*porrectus flexus*		`\[ la \flexa fa \pes sol \auctum \descendens mi \]`
	↓↓↑	*climacus resupinus*		`\[ \virga si \inclinatum la \inclinatum sol la \]`
	↓↓↓	*virga subtripunctis*		`\[ \virga la \inclinatum sol \inclinatum fa \inclinatum mi \]`

liturgici. Tutti questi stili vennero sostituiti dal canto gregoriano, con l'unica eccezione del canto ambrosiano.

Al canto gregoriano è succeduto il *chant messin*.[4] Il canto gregoriano fu chiamato così in onore di papa Gregorio I 'il Grande' (540–604 circa), ma fece la sua comparsa solo alla fine dell'VIII secolo, epoca in cui vennero scritte le prime fonti provviste di notazione musicale, e si sviluppò in Europa occidentale e centrale nei secoli IX e X. Nell'XI secolo si è evoluto gradualmente nell'*organum*, la prima fase dello sviluppo della *polifonia*.[5]

3 Notazioni gregoriane

Ricordiamo che il canto gregoriano si diffuse in molti paesi e per diversi secoli, quindi ne esistono alcune varianti. In particolare, sono state adoperate notazioni simili all'antica musica greca o al canto bizantino — le cosiddette notazioni di San Gallo — messe sopra i testi da cantare. Quella usata più di frequente per il canto gregoriano si basa sui *neumi*.[6] Un neuma è una formula melodica e ritmica applicata a una sillaba. Le note di un neuma si raffigurano mediante una notazione quadrata, e note successive appartenenti allo stesso neuma vengono raggruppate ricorrendo ad altri effetti grafici. Si ricorda che i neumi non avevano lo scopo di specificare né la musica strumentale, né gli accordi, né i ritmi paralleli.

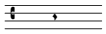
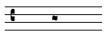
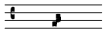
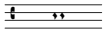
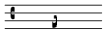
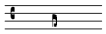
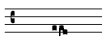
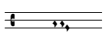
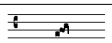
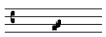
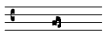
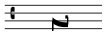
Nelle prossime sezioni si descriveranno prima i modi gregoriani, quindi i fondamenti della notazione neumatica e i neumi principali. Alcuni neumi aggiuntivi vengono adoperati molto raramente o sono stati esclusi dalle edizioni critiche del canto gregoriano. Si possono trovare ulteriori dettagli, in particolare sulla relazione tra le notazioni di San Gallo e i neumi quadrati, in (Martignac e Pistone, 1981; Viret, 1999).

4. Letteralmente, 'canto di Metz'. All'epoca, Metz — nella Francia orientale — era la capitale del regno franco.

5. Per la verità, alcuni precoci tentativi di *organum* risalgono al IX secolo, ma questa tecnica si è sviluppata compiutamente solo nel XI secolo.

6. Dal greco νεῦμα, 'segno'.

Tabella 2: Neumi speciali.

Note	Dir.	Neuma	Esempio	Codice LilyPond
1	–	*stropha* (R)		`\[ \stropha si \]`
		oriscus		`\[ \oriscus si \]`
		quilisma		`\[ sol \pes \quilisma la \pes si \]`
2	→	*distropha* (R)		`\[ \stropha si \stropha si \]`
	↑	*epiphonus* (L)		`\[ sol \pes \deminutum si \]`
	↓	*cephalicus* (L)		`\[ si \flexa \deminutum sol \]`
3	→	*tristropha* (R)		`\[ \stropha si \stropha si \stropha si \]`
	→↓	*pressus*		`\[ sol sol \flexa fa \]`
		trigon		`\[ \stropha si \stropha si \stropha la \]`
	↓↓	*salicus*		`\[ sol \oriscus la \pes \virga si \]`
		scandicus liquescens (L)		`\[ sol \pes la \pes \deminutum si \]`
	↑↓	*pinnosa* (L)		`\[ la \pes si \flexa \deminutum sol \]`
	↓↑	*porrectus liquescens* (L)		`\[ la \flexa sol \pes \deminutum si \]`

3.1 Modi gregoriani

In (Hufflen, 2017a), si osserva che nell'alto Medioevo i *modi* musicali non si possono considerare come le scale 'moderne' comprendenti sette gradi, ma erano organizzati in gruppi di sei note. In seguito, i modi gregoriani si basarono sulle sette note che conosciamo oggi: do, re, mi, fa, sol, la, si. La nota 'si' poteva essere *bemollizzata* in si♭. Più esattamente in si♭, se adoperiamo il simbolo dell'epoca. Poiché ogni nota naturale può essere la base — la *tonica* — di un modo, teoricamente si ottengono sette modi, anche se, nella pratica, se ne adoperavano quattro soltanto. Da un lato, nessun modo si basava sulla nota si, soggetta com'era a variazione: talvolta si♮, talaltra si♭. Per di più, una scala basata su si♮ sarebbe stata molto instabile. Dall'altro, l'uso del si♭ permetteva di vedere alcune coppie come uguali, fino a una trasposizione completa:

Modo la ⟸ Modo re con si♭
Modo do ⟸ Modo fa .....

Così, i modi effettivamente in uso erano basati sulle note re, mi, fa, sol. Ciascuno di essi era a propria volta suddiviso in due 'sotto'-modi.

- *Modo autentico*: l'intervallo coperto dal canto si trova *sopra* la tonica, fino a un'ottava, e può anche comprendere la nota immediatamente sotto la tonica.
- *Modo plagale*: la tonica si trova *a metà* dell'intervallo coperto dal canto.

Consideriamo, per esempio, il modo di re. Ecco gli intervalli di ciascun sotto-modo:

$$\underset{\text{Modo plagale}}{\underline{\text{la si do } \boxed{\text{re}} \text{ mi fa sol la}}}\ \overset{(\downarrow)}{\overline{\text{si do re}}}\ \text{Modo autentico}$$

In entrambi i casi, la tonica (la nota finale) è riquadrata. Il modo autentico può comprendere anche la nota contrassegnata con '(↓)'. Si possono trovare ulteriori dettagli in proposito in (Chailley, 1967) e (Chailley, 1972).

3.2 Righi e chiavi

Il canto gregoriano era scritto su *tetragrammi*, come mostra la figura 2. Si adoperavano due *chiavi*:

- la chiave di fa 𝄢 era per le voci gravi e poteva essere scritta sulla seconda o sulla terza linea;[7]
- la chiave di do 𝄡 era per le voci intermedie e poteva essere scritta sulle linee diverse dalla prima.

La forma di queste chiavi è correlata alle lettere gotiche 'F' e 'C' rispettivamente. Non c'è *armatura di chiave*, se si esclude l'indicazione del si♭, adoperato peraltro molto di rado. Da ultimo, ma non meno importante, osserviamo che le altezze sono piuttosto approssimative, dato che che all'epoca non esisteva un diapason ufficiale.

3.3 Neumi

Come si è detto più sopra, un neuma è associato a una sillaba. I neumi consistono di una o più note, fino a quattro. Le note vengono raffigurate mediante quadrati o losanghe; ad alcune note viene apposto un gambo (𝅘𝅥), a destra della testa per una

7. Le linee di un rigo musicale — 'classico' o gregoriano che sia — sono numerate dal basso verso l'alto.

```
\version "2.19.82"

\language "italiano"
\include "gregorian.ly"

\score {
 \new VaticanaStaff = "cantus" {
  \override Staff.StaffSymbol.color =
    #(x11-color 'grey15)
  \clef "vaticana-do2"
  si \virgula
  \[ \inclinatum si \]
  \caesura
  \[ \deminutum si \]
  \divisioMinima
  \[ \stropha si \]
  \divisioMaior
  \[ \oriscus si \]
  \divisioMaxima
  \[ \quilisma si \]
  \finalis
 }
}
```

Figura 1: Codice LilyPond per la figura 2.

nota isolata, a sinistra in alcuni altri casi. Quando più note sono allineate verticalmente, vanno lette dal basso verso l'alto. Nel caso in cui sia tracciato un tratto obliquo, le prime due note del neuma corrispondono ai suoi punti iniziale e finale.

La tabella 1 mostra i neumi adoperati più di frequente, classificati in base al numero delle note di cui sono composti, indicate nella prima colonna. Se più d'una, la o le *direzioni* da una nota a quella successiva ('Dir.', nella tabella) sono indicate mediante frecce, visibili nella seconda colonna. Ciascun neuma porta un nome di origine latina.

Il *podatus* ( ) — o *pes* ('piede') — ha due note ascendenti, la *clivis* ( ) — o *flexa* ('discesa') — due note discendenti.

Per quanto riguarda i neumi di tre note, lo *scandicus* (, 'alto'), viene adoperato per tre note ascendenti, mentre il *climacus* ( ), originato dalla parola latina *climax* ('scala'), viene adoperato per tre note discendenti; il *torculus* (, 'attorcigliato'), è adoperato quando la seconda nota è più acuta. Quando questa nota è più grave, si scrive un *porrectus* (, 'allungato').

I nomi dei neumi di quattro note sono composti: l'aggettivo *resupinus* ('inverso'), significa che l'ultima nota è più acuta di quella precedente. Quando questa nota è più grave, si adopera l'aggettivo *flexus* ('piegato').

La tabella 2 raccoglie alcuni neumi specializzati, che includono informazioni aggiuntive su espressione o ritmo. 'R' sta per neumi con note leggere e rapide. Spesso questi ultimi sono impiegati nelle ripetizioni veloci, al contrario del *bipunctum*, che esprime una durata doppia.

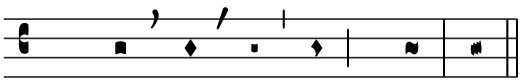


Figura 2: Teste di note, respiri e divisioni (il codice relativo è mostrato nella figura 1).

Nei neumi *liquescenti* ('L', nella tabella), l'ultima nota è di piccole dimensioni. Essi si applicano a una sillaba che termina con una consonante, da eseguire rapidamente e discretamente. L'*epiphonus* può essere visto come un *podatus* liquescente, il *cephalicus* come una *clivis* liquescente, la *pinnosa* come un *torculus* liquescente.

Il *pressus* è una specie di neuma 'ornamentale' adoperato quando la nota finale di un gruppo è identica alla nota iniziale del gruppo successivo. L'*oriscus* significa che la testa della nota corrispondente non può essere unita alla precedente. Il *quilisma* è contrassegnato da una linea frastagliata: la nota precedente è tenuta un po' più a lungo.

Potrebbe essere difficile distinguere graficamente le diverse teste di nota. Per aiutare i lettori, la figura 2 mostra un'immagine di grandi dimensioni in cui si vedono, da sinistra a destra, una nota quadrata, una nota 'losangata', una nota adoperata alla fine di un neuma liquescente e una nota rapida.[8]

Terminiamo questa sezione con il segno di *custos*: questo piccolo simbolo — una nota piccolissima con un gambo sottile ( ) — veniva messa alla fine del rigo per segnalare al cantore quale sarebbe stata la prima nota del rigo seguente.

3.4 Notazione del ritmo

Ritmicamente parlando, non c'era differenza tra un *punctum* e una *virga*: in origine erano adoperati solo per effetti grafici. Il *bipunctum* o una nota puntata esprimono una durata doppia. Un altro simbolo, l'*episema*,[9] è collegato all'*accentazione ritmica*. Quando è nella forma di un piccolo tratto orizzontale sopra un neuma, esprime una durata leggermente più lunga; quando è un piccolo tratto verticale sopra una nota, indica un accento ritmico.

Non esistono pause, nel senso di durate precise associate a tali segni, come accade nella musica classica con le teste e i gambi delle note. Le notazioni gregoriane contemplano *segni di respiro*, come mostra la figura 2: da sinistra a destra, la *virgula* sta per un segno di respiro breve, la *caesura* per un segno di respiro un po' più lungo. Questi due segni sono spesso sostituiti da *divisioni* — anch'esse presenti nella stessa figura — che sembrano stanghette di battuta: la *divisio minima* e la *divisio maior*. Il segno di *finalis*, infine, assomiglia a una doppia stanghetta e segnala la fine del canto. Si possono

8. La sequenza mostrata nella figura 2 non ha senso nel canto gregoriano; ha il solo scopo di facilitare un confronto tra le diverse forme.

9. Dal greco ἐπίσημον, 'segno distintivo'.

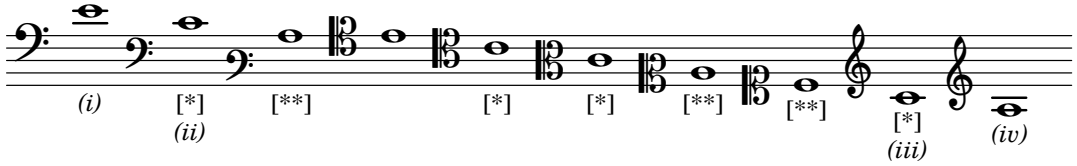


Figura 3: Chiavi classiche e moderne.

trovare ulteriori dettagli sul ritmo nel canto gregoriano in (Martignac e Pistone, 1981).

4 Verso la musica preclassica

Le successive fasi dell'evoluzione della musica occidentale dal gregoriano alle forme del XVIII secolo sono descritte più nel dettaglio in (Gut e Pistone, 1985). Qui di seguito proponiamo un breve collegamento tra il canto gregoriano e alcune notazioni moderne.

Righi con cinque o sei (!) linee comparvero nel XIII secolo, mentre l'uso normalizzato dei pentagrammi è entrato in vigore solo dal XVII secolo. La chiave di sol, per le voci acute, comparve dopo le altre due chiavi, di do e di fa.[10] La figura 3 raccoglie tutte le chiavi adoperate durante il periodo preclassico. Oggi si adoperano solo quelle contrassegnate con '[*]', nel senso che gli interpreti devono leggerle negli spartiti dei propri strumenti: per esempio, la chiave di do sulla terza linea è adoperata per le viole, la chiave di do sulla quarta linea è adoperata per il registro intermedio di strumenti come il violoncello o il fagotto. Il segno '[**]' significa che la chiave è ancora in uso, ma solo a scopi di trasporto o didattici (negli esercizi di armonia, per esempio). Tutte le note presenti nella figura 3 si trovano alla stessa altezza. Così possiamo notare l'equivalenza tra la chiave di fa sulla terza linea e la chiave di do sulla quinta.[11] Un'altra relazione è più sottile: '*(i)*' legge come '*(iii)*', ma suona due ottave sotto; la stessa relazione lega '*(ii)*' e '*(iv)*'.[12] La figura 4 mostra un esempio nel quale si adoperano le vecchie chiavi.

I gambi e le diverse forme delle teste di nota vennero introdotti progressivamente per rappresentare il *ritmo*. Ricordiamoci che la moderna notazione ritmica negli spartiti si basa su una suddivisione per due, se non consideriamo le note puntate: una nota intera (𝅝) è divisa in due metà (𝅗𝅥), una metà è divisa in due note da un quarto (♩), e così via. Questo *modus operandi* non era universale, nel Medioevo: secondo il *tempus perfectum*, una nota lunga era divisa in *tre* note più brevi, mentre una divisione per due era correlata al *tempus imperfectum*. Si può notare un uso simile nelle opere moderne strettamente collegate alla musica medievale da intenzioni arcaistiche, come mostra la figura 5. L'interpretazione dei neumi di due e tre note nello stesso canto — cioè ♫ e ♬ — è piuttosto controversa: i cantanti dovrebbero basarsi su un ritmo regolare e cantare *duine* e *terzine*, com'è stato fatto da Arthur Honegger quando ha adattato un brano gregoriano nella propria opera *Jeanne au bûcher* (figura 6)? Dovrebbero associare la stessa durata a ogni nota? O dovrebbero basarsi sul *tempus perfectum* e interpretare i gruppi binari ♫ come ♩ 𝅘𝅥𝅯? Attualmente la seconda interpretazione è la più diffusa, ma, secondo alcuni studi recenti, è la terza quella più probabile.

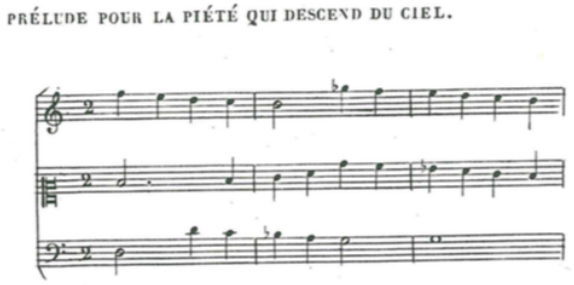
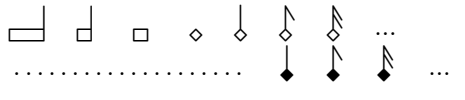


Figura 4: Uso delle chiavi antiche (Moreau, 1902, inizio).

L'idea di mettere in relazione i ritmi con l'aspetto delle note risale alla fine del XIII secolo. A quel tempo, la suddivisione per due aveva soppiantato quella per tre e i tre segni fondamentali erano interpretati in modo diverso: una *virga* (𝅋) per due *puncti*, un *punctum* per due *semibreves* (◆). Alla fine del XVII secolo, comparve la seguente suddivisione:



cioè, 𝅘𝅥𝅯 era sinonimo di 𝅘𝅥, mentre 𝅘𝅥𝅰 lo era di 𝅘𝅥𝅯. In seguito, ◇ si trasformò in 𝅝, la testa delle note con il gambo divenne più tonda, le note bianche con il gambo scomparvero, tranne la nota che indica una metà (𝅗𝅥). Oggi, le durate più lunghe di quella di una nota di intero (𝅝) sono adoperate molto

10. Quando la chiave di sol fece capolino, le chiavi di do, in uso per le voci intermedie, vennero impiegate sempre meno, perché l'uso delle altre due (fa e sol) permetteva di coprire un intervallo significativo dal registro grave a quello acuto.

11. Queste due chiavi erano le cosiddette 'chiavi di baritono'. Nella pratica, la prima ha soppiantato la seconda molto rapidamente. Oggi la chiave di fa sulla terza linea è adoperata solo per il trasporto.

12. La chiave di fa sulla quinta linea, che può essere vista come una chiave di 'sub-basso', è stata abbandonata presto. Quella di sol sulla prima linea è chiamata *chiave di violino francese*, perché era adoperata nella musica per violino in Francia nei secoli XVII e XVIII.

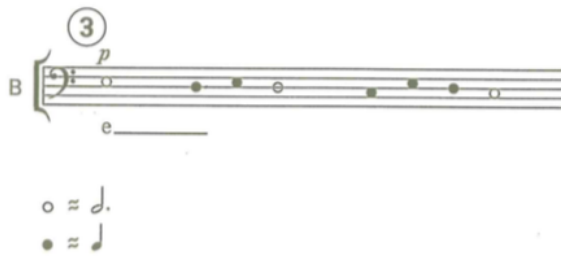


Figura 5: Notazione ritmica antica (Pärt, 1984).

di rado; □ si è trasformata nella *doppia nota intera* (𝅜). Le due figure più lunghe sono rimaste esattamente dov'erano, ora sono conosciute rispettivamente come *maxima* e *longa*. Le figure 7 e 8 ne mostrano degli esempi: nella seconda figura, possiamo notare che il gambo di una *longa* è sempre a destra della testa, com'è sempre stato il caso delle aste adoperate nel canto gregoriano; inoltre, una *longa* può essere puntata.[13]

5 Comporre il canto gregoriano

In questa sezione, esaminiamo alcuni programmi musicali in grado di comporre il canto gregoriano adoperando i neumi originali. Non pretendiamo di essere esaurienti, intendiamo solo orientare i lettori interessati ad approfondire questa ricerca.

5.1 MusiXTEX

MusiXTEX (Taupin *et al.*, 2011) fornisce alcune estensioni per comporre i neumi gregoriani su tetragrammi. Tra di esse, segnaliamo `musixgre` (Taupin *et al.*, 2011), implementata da Andreas Egler, che si basa principalmente su comandi simili a quelli di TEX. Per esempio, il neuma di un *torculus* — che ricordiamo essere di tre note — è implementato per mezzo di un comando `\torculus` con tre argomenti. Abbiamo già espresso la nostra opinione generale su MusiXTEX in (Hufflen, 2011, 2017b): il programma produce bei documenti, ma la sua sintassi è difficile da gestire e mantenerne i sorgenti può essere noioso. Si possono evitare alcuni inconvenienti di questo tipo ricorrendo a *preprocessori*, ma, per quanto ne sappiamo, non esiste un preprocessore specifico per il canto gregoriano. Inoltre, alcuni neumi non sono stati ancora implementati.

5.2 OpusTEX

Anche OpusTEX (Opus, 2011) è esterno a TEX ed è stato implementato da Andreas Egler. Quando uscì, questo programma venne indicato come lo strumento migliore per comporre il gregoriano. Tuttavia non è libero, sembra incompleto e da diversi anni non viene più mantenuto.

13. Nella *Josquiniana*, Charles Wuorinen riarrangiò le opere vocali di Josquin des Prés (1450–1521 circa) per un quartetto d'archi adoperando le figure di ritmo originali.

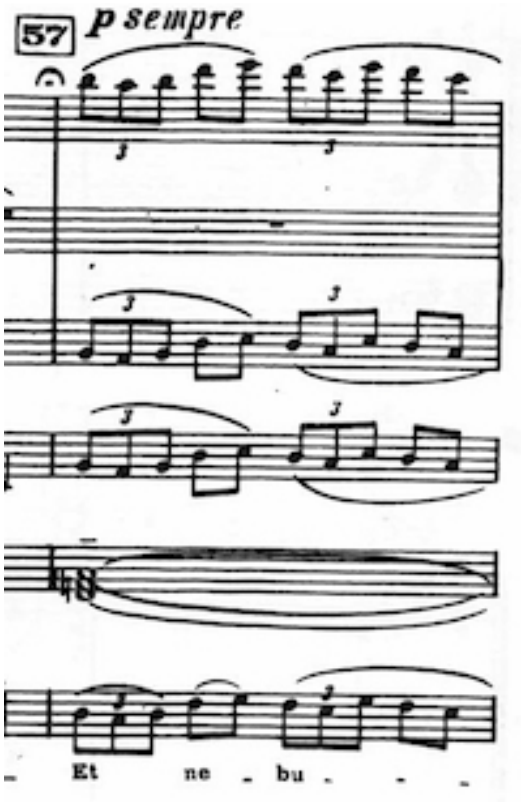


Figura 6: Adattamento moderno (Honegger, 1938, n. 57).

5.3 LilyPond

LilyPond (LilyPond, 2017) permette di comporre il canto gregoriano. Come mostrano le tabelle 1 e 2, i *comandi* di LilyPond[14] vengono adoperati per neumi di una nota — `\virga` e `\stropha`, per esempio — e per cambiare l'aspetto di una nota — `\inclinatum` e `\quilisma`, per esempio — mentre i neumi di più note sono costruiti tramite *legature*, specificate con comandi *infissi* — `\pes` e `\flexa`, per esempio. Sempre nelle stesse tabelle abbiamo messo i codici LilyPond per ottenere i neumi: possiamo notare che quelli di più note sono racchiusi da '`\[`' e '`\]`'.[15] Analogamente, il codice Lilypond della figura 2 si vede nella figura 1.

LilyPond produce spartiti molto gradevoli alla vista, anche se l'allineamento orizzontale delle note dovrebbe essere migliorato.[16] Parimenti, note e testi non sono sempre perfettamente allineati. Il vantaggio principale del programma è che è molto personalizzabile. Per esempio, sono disponibili numerosi stili storici per le chiavi:[17] stile *Vatica-*

14. LilyPond è un programma WYSISWYM (W*hat* Y*ou* S*ee* I*s* W*hat* Y*ou* M*ean*), come TEX. I suoi comandi non sono paragonabili a quelli di TEX, ma ne hanno l'aspetto.

15. Il codice nelle tabelle assume che si imposti il programma per la lingua italiana. (*N.d.T.*)

16. Questo difetto può essere parzialmente risolto adoperando una *pausa fittizia*, come abbiamo fatto nella figura 10 (con `s2`).

17. Osservando la figura 1, si può notare che la specificazione di una chiave in LilyPond segue la numerazione delle linee dal basso verso l'alto, ma partendo da zero.

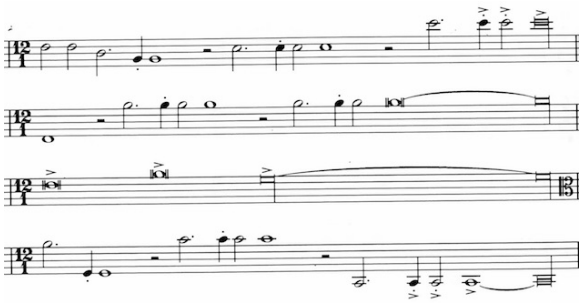


Figura 7: Uso di *breves* e *longae* (Wuorinen, 2001, misura 143).

na,[18] stile *Hufnagel*, stile *Editio Medicaea*. Un altro esempio è offerto dai righi: spesso le linee di un rigo gregoriano erano disegnate in rosso, ma questo colore predefinito può essere ridefinito, come si vede nelle figure 1 e 10.

5.4 Gregorio TEX

Gregorio TEX è parte del *Gregorio project* (Gregorio, 2016). Esso fornisce `gabc`, una notazione per rappresentare il canto gregoriano adoperando unicamente caratteri ASCII,[19] e Gregorio TEX, uno stile TEX per comporre spartiti. Gregorio TEX produce bei documenti e fornisce alcune interessanti integrazioni con altre funzionalità di TEX, ma il suo *modus operandi* non è veramente WYSISWYM: il linguaggio `gabc` non è stato progettato per l'introduzione diretta di testi da parte degli utenti finali. I testi sorgente, inoltre, sono creati mediante un'interfaccia grafica.

5.5 Finale

Oggi Finale (Finale, 2017) è probabilmente il programma più largamente adoperato dai tipografi musicali professionisti. Non è libero ed è WYSIWYG.[20] Permette di installare il plug-in Medieval 2 (Piéchaud, 2017) per gestire le notazioni musicali antiche, comprese quelle del canto gregoriano. Nemmeno i font che adopera sono liberi.

6 Conclusioni

Rimanendo alla pratica musicale, oggi numerosi cantanti interpretano brani di canto gregoriano. Alcuni ne adoperano trascrizioni realizzate con notazioni moderne, altri, per ragioni di autenticità, preferiscono leggere gli originali. La buona composizione degli spartiti gregoriani, dunque possibilmente con l'indicazione delle diverse interpretazioni dei neumi o delle divergenze tra le differenti edizioni critiche, potrebbe interessare a un certo numero di utenti. Anche se ci si accosta a tali spartiti principalmente per una questione di interesse storico,

18. È lo stile adoperato nelle figure di quest'articolo tranne che nella figura 9, realizzata con lo stile *Editio Medicaea*.
19. *American Standard Coding for Information Interchange*.
20. *What You See Is What You Get*.

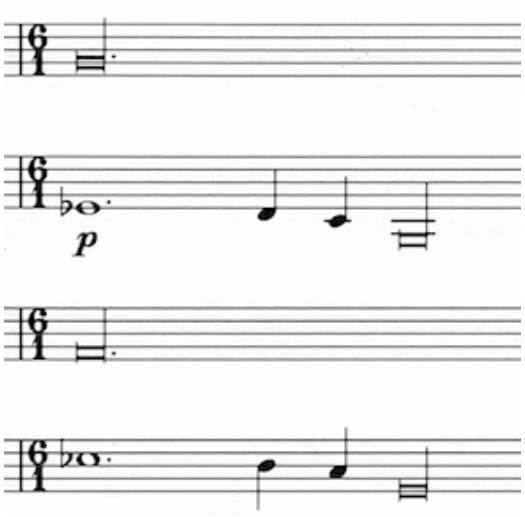


Figura 8: (Wuorinen, 2001, misura 109).

essi sono la testimonianza di un'altra percezione della musica, e la loro tipografia riflette la messa per iscritto di questa percezione organizzando lo spazio in modo diverso da come si fa con la notazione musicale moderna.

A Influenza del canto gregoriano

Il canto gregoriano ha avuto un'influenza significativa sulla musica, nel senso che alcuni canti popolari derivano da quelli gregoriani. Analogamente, alcuni canti gregoriani sono stati adoperati nella musica classica. Un ottimo esempio è fornito dal *Dies irae*,[21] che descrive il Giudizio universale. Ecco un elenco non completo — ordinato cronologicamente per data di composizione — di brani di musica romantica e moderna che lo citano o che si basano su temi da esso derivati:

- *Symphonie fantastique* di Hector Berlioz (Berlioz, 1830);
- *Danse macabre* (Saint-Saëns, 1874) e Sinfonia n. 3 per organo (Saint-Saëns, 1885) di Camille Saint-Saëns;
- Sinfonia n. 6 di Nikolaj Mjaskovskij[22] (Mjaskovskij, 1923);
- *Butantan* di Ottorino Respighi (in *Impressioni Brasiliane* (Respighi, 1928));
- Sinfonia n. 1 (Rachmaninov, 1895) e n. 3 (Rachmaninov, 1938) di Sergej Rachmaninov;
- *Dies Irae* di Krzysztof Penderecki (Penderecki, 1967).

21. 'Giorno dell'ira', in latino
22. Nonostante che fosse nato a Modlin, in Polonia, Mjaskovskij era considerato russo.

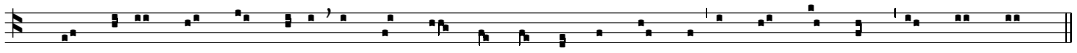


Figura 9: *Alieni insurrexerunt in me* (il codice relativo è mostrato nella figura 10).

```
\version "2.19.82"

\language "italiano"
\include "gregorian.ly"

\score {
 \new VaticanaStaff = "cantus" {
  \override Staff.StaffSymbol.color =
    #(x11-color 'grey15)
  \clef "medicaea-do2"
  sol s2 la
  \[ do' \pes re' \]
  re' re' do' re' mi' re'
  \[ do' \pes re' \]
  re' \virgula re' la re' do'
  \[ do' \flexa si \]
  \[ la \flexa sol \]
  \[ la \flexa sol \]
  \[ fa \pes sol \]
  la do' la la
  \divisioMinima
  re' do' re' fa' do'
  \[ la \pes do' \]
  \divisioMinima
  re' do' re' re' re'
  \finalis
 }
}
```

Figura 10: Codice LilyPond per la figura 9.

B *Farewell Song*

A volte, nelle mie composizioni musicali mi sono ispirato a melodie gregoriane o ambrosiane. Le ho sviluppate aggiungendo un po' d'armonia e cercando di combinare accordi antichi e moderni, dato che il mio personale stile compositivo è caratterizzato da contaminazioni di questo tipo. Penso che il repertorio gregoriano si presti alla cosa, perché non è musica tonale come la musica classica. In particolare, i canti gregoriani mi hanno offerto buoni punti di partenza per le parti tristi di alcuni canti della liturgia del Venerdì Santo.

Quando ho saputo della morte accidentale di Stanisław Wawrykiewicz, mi è venuto in mente che il tema di *Alieni insurrexerunt in me*[23] (Anonimo, 1230) sarebbe potuto essere un buon punto di partenza per un canto d'addio.

Questo adattamento personale — una specie di 'ricomposizione' — è mostrato *in extenso* nelle figure 11 e 12. Possiamo considerarlo come un *decrescendo*, come qualcuno che si allontana. Ho mescolato il modo di re con la 'classica' tonalità di re minore fino alla fine del brano.

Farewell Song è scritta per un quartetto di ottoni: due trombe, un trombone e una tuba. La parte delle trombe è in do,[24] ma nulla vieta di adoperare trombe in si♭. Infatti, i quattro strumenti potrebbero essere sostituti, a patto che l'insieme sia omogeneo: fiati, archi, eccetera.

Per scrivere la partitura ho adoperato MuseScore (MuseScore, 2017). Il canto gregoriano originale è mostrato nella figura 9, mentre il codice LilyPond usato per produrla si vede nella figura figura 10.

Riferimenti bibliografici

Anonimo (1230). «Alieni insurrexerunt in me». http://gregorianik.uni-regensburg.de/cdb/1321. Manoscritto.

Arnold, Denis (a cura di) (1983). *The New Oxford Companion to Music*. Oxford University Press.

Berlioz, Hector (1830). *Symphonie fantastique, Op. 14*. No. 442. Edition Ernst Eulenburg, Mainz.

Chailley, Jacques (1967). *Cours d'histoire de la musique*. Alphonse Leduc, Paris. Tome 1, *Des origines à la fin du 17e siècle*. 1er Volume, *Cours d'Historie*.

— (1972). *Cours d'histoire de la musique*. Alphonse Leduc, Paris. Tome 1, *Des origines à la fin du 17e siècle*. 2e Volume, *Exemples musicaux*.

— (1984). *Histoire musicale du Moyen Age*. Quadrige / Presses Universitaires de France, Paris, 3ª edizione.

— (1985). *La musique et le signe*. Les Introuvables. Éditions d'Aujourd'hui.

Finale (2017). «Music Notation Software». https://www.finalemusic.com.

Gregorio (2016). «The Gregorio project». http://gregorio-project.github.io/index.html.

Gut, Serge e Danièle Pistone (1985). *Le commentaire musicologique du grégorien à 1700. Principles et exemples*. Librairie Honoré Champion, Paris.

23. 'Gli stranieri sono insorti contro di me'

24. In questo modo, chi legge non deve preoccuparsi di trasportare.

Honegger, Arthur (1938). *Jeanne d'Arc au bûcher*. Éditions Salabert, Paris.

Hufflen, Jean-Michel (2011). «Some experience with MusiXTEX». In *EuroBachoTEX 2011 Conference Proceedings*, a cura di Karl Berry, Jerzy Ludwichowski e Tomasz Przechlewski. GUST, Bachotek, pp. 19–30.

— (2017a). «History of accidentals in music». In *Premises, Predilections, Predictions*, a cura di Karl Berry, Jerzy Ludwichowski e Tomasz Przechlewski. GUST, Bachotek, pp. 42–49. Atti del convegno TUG@BachoTEX 2017.

— (2017b). «Requirements for a Music Engraving Program: a Composer's Point of View». *ArsTEXnica*, (24), pp. 32–36. https://www.guitex.org/home/images/ArsTeXnica/AT024/hufflen-venice.pdf.

Jacobs, Arthur (a cura di) (1988). *The New Penguin Dictionary of Music*. Penguin Books, 4ª edizione.

LilyPond (2017). «...music notation for everyone». http://lilypond.org/index.html.

Martignac, André G. e Danièle Pistone (1981). *Le chant grégorien. Historique et pratique*. Librairie Honoré Champion, Paris.

Mjaskovskij, Nikolaj Jakovlevič (1923). *Sinfonia n. 6 in mi bemolle minore, Op. 23*. Universal Edition, Wien.

Moreau, Jean-Baptiste (1902). *Esther. Intermèdes en musique de la Tragédie de Jean Racine*. Èdition de la Schola Cantorum, Paris. Remis au jour d'après les éditions de 1689 et 1696, avec réalisation de la basse chiffrée, nuances et indications d'exécution et préface de Charles Bordes.

MuseScore (2017). «MuseScore Handbook». https://musescore.org/.

Opus (2011). «Installation and Use of OpusTEX». https://www.muri-gries.ch/OpusTeX/OpusTeX.html.

Penderecki, Krzystof (1967). *Dies Irae. Oratorium ob memoriam in perniciei castris in Oświęcim necatorum inexstinguibilem reddendam*. ED 20162. Schott Music, Mainz. Na sopran, tenor, bas, chórmieszany (SATB) i orkiestrę.

Piéchaud, Robert (2017). «Medieval. The smart and unique solution devoted to early music notation». https://www.klemm-music.de/notation/medieval/en/index.php.

Pärt, Arvo (1984). *An den Wassern zu Babel saßen wir und weinten (Psalm 137)*. Universal Edition, Wien.

Rachmaninov, Sergej Vasil'evič (1895). *Sinfonia n. 1 in re minore, Op. 13*. SIK2318. Internationale Musikverlage Hans Sikorski, Hamburg.

— (1938). *Symphony No. 3 in A minor, Op. 44*. Boosey & Hawkes, London.

Respighi, Ottorino (1928). *Impressioni brasiliane*. P.R. 478. Ricordi, Milano.

Saint-Saëns, Camille (1874). *Danse macabre, Op. 40*. No. 442. Durand & C.ie Éditeurs, Paris.

— (1885). *Symphonie n⁰ 3 en ut mineur, Op. 78 «avec orgue»*. No. 442. Durand & C.ie Éditeurs, Paris.

Taupin, Daniel, Ross Mitchell e Andreas Egler (2011). *MusiXTEX. Using TEX to write polyphonic or instrumental music*. https://ctan.org/pkg/musixtex. Version 1.15.

Viret, Jacques (1999). «La notation du chant grégorien. Écriture et oralité, des rapports problématiques». In *Cahiers d'éthnomusicologie*, Ateliers d'ethnomusicologie, Genève, 12, pp. 75–93. http://journals.openedition.org/ethnomusicologie/679.

Wuorinen, Charles (2001). *Josquiniana*. PE.EP68012. Edition Peters, Leipzig. For String Quartet.

▷ Jean-Michel Hufflen
Institut de Recherche FEMTO-ST
Université de Franche-Comté
Besançon cedex – France
jmhuffle@femto-st.fr

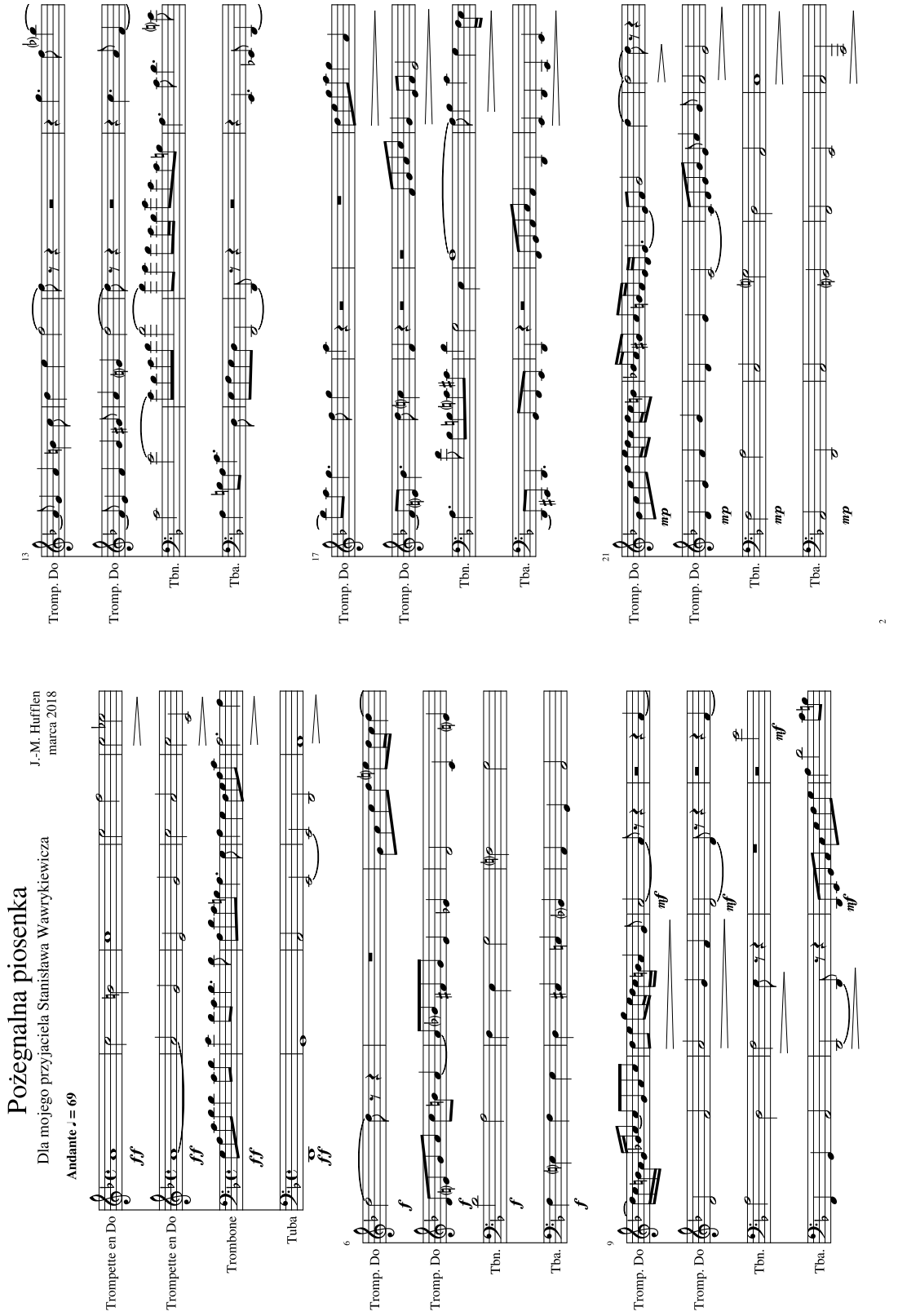


Figura 11: *Farewell Song* (inizio).

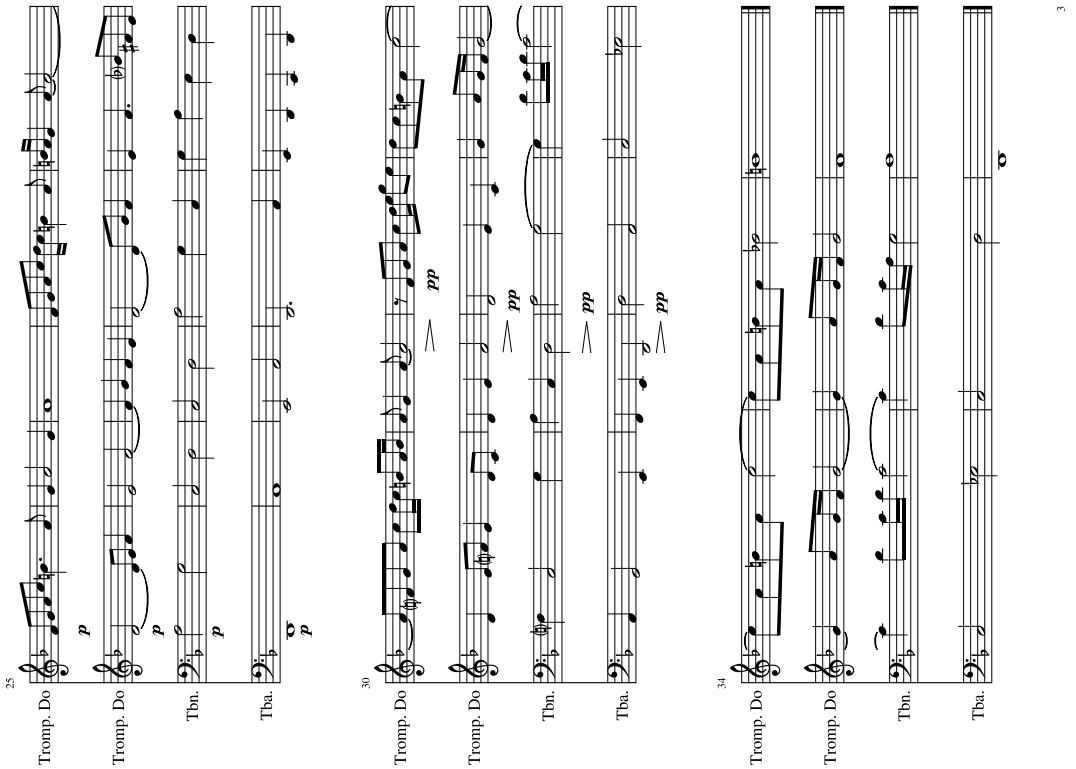


Figura 12: *Farewell Song* (fine).

siunitx: passato, presente e futuro

*Joseph Wright**

Sommario

Il pacchetto siunitx fornisce una potente serie di strumenti per la composizione di numeri e unità in LaTeX. Incorporando i dettagli sulle regole comunemente accettate per la presentazione dei dati scientifici, siunitx permette agli autori di concentrarsi sul contenuto e il significato del proprio contributo, accollandosi *in toto* l'onere della formattazione. In questa sede getterò uno sguardo sul retroterra del pacchetto, sulle motivazioni che mi hanno portato dalla versione 1 alla versione 2 e sul perché ora è in fase di sviluppo la terza versione.

Abstract

The siunitx package provides a powerful toolkit for typesetting numbers and units in LaTeX. By incorporating detail about the agreed rules for presenting scientific data, siunitx enables authors to concentrate on the meaning of their input and leave the package to deal with the formatting. Here, I look at the background to the package, what led me from version 1 to version 2, and why version 3 is now under development.

1 Introduzione

In TeX, la composizione delle unità si presta naturalmente a essere supportata da macro. Le regole formali per le unità SI (BIPM, 2006) mettono in relazione i nomi delle unità con i relativi simboli e non sorprende che numerosi autori abbiano scritto pacchetti che affrontano alcune o tutte le sottigliezze coinvolte. Tempo fa, l'ultima volta che ho esaminato siunitx per *TUGboat* (Wright, 2011), ho descritto alcuni esempi degli uni e delle altre. Qui riassumerò brevemente alcuni di questi punti chiave, poi analizzerò le motivazioni trainanti verso la terza versione del pacchetto.

2 I primi tempi

Prima di siunitx esisteva un certo assortimento di pacchetti dedicati alla composizione delle unità, tra i quali uno, chiamato SIunits (Heldoorn e Wright, 2007), che definiva macro di tipo semantico. Il mio coinvolgimento nella faccenda è incominciato quando, nel novembre del 2007, ho risposto a un messaggio apparentemente innocuo inviato da Stephen Pinnow a `comp.text.tex`, nel quale si lamentava un bug inequivocabile:

> `I want to report that \reciprocal, \rpsquare, \rpcubic, etc. output is written as "-1" instead of a $-1$, when the package option "textstyle" is used. I tried to contact Mr. Heldoorn, but he didn't answer until now. Does anyone have an idea what to do?`
>
> «Vorrei segnalare che quando l'opzione textstyle è attiva, il risultato di `\reciprocal`, `\rpsquare`, `\rpcubic`, eccetera è stampato come `-1` anziché come `$-1$`. Ho provato a contattare il signor Heldoorn, ma finora non mi ha risposto. Qualcuno ha idea di che cosa fare?»

Essendo giovane e incosciente, dopo aver esaminato il bug e aver scoperto che SIunits non era più mantenuto, mi sono offerto volontario per rilevare il pacchetto. Ancora più incautamente, sempre nello stesso mese ho replicato come segue:

> `As some of you may have noticed, following a recent bug report concerning the SIunits package, I have taken over as the package maintainer. I have uploaded a bug fix for the specific issue to CTAN, and so hopefully it will appear within a day or two.`
> `It has been suggested by the maintainer of the SIstyle package that integration of the two be would worth considering. Other suggestions have also been made in the newsgroup and by private mail. I am therefore planning to review the existing situation and see what improvements are needed/desirable. As well as SIunits and SIstyle, I am going to look at numprint, units, unitsdef and hepunits for inspiration/points to consider/etc. So far, I have some outline ideas, for example: ...`
>
> «Come alcuni di voi avranno notato, in seguito a una recente segnalazione di bug relativa al pacchetto SIunits mi sono assunto l'incarico di mantenere il pacchetto. Ho caricato su CTAN una correzione per il problema in questione, quindi spero che nel giro di uno o due giorni sarà visibile. Il manutentore del pacchetto SIstyle ha

*Questo articolo è stato pubblicato originariamente su *TUGboat*, vol. 39, n. 2 (2018), p. 119–121. È stato tradotto da Tommaso Gordini con il permesso dell'autore. Eventuali errori o fraintendimenti del testo originale sono del traduttore. (*N.d.R.*)

> consigliato di prendere in considerazione una possibile integrazione dei due pacchetti. Altri suggerimenti sono stati avanzati anche nel newsgroup e per posta privata. Pertanto, ho intenzione di rivedere la situazione esistente e valutare quali miglioramenti siano necessari o desiderabili. Oltre a SIunits e SIstyle, esaminerò numprint, units, unitdef e hepunits per ispirarmi, isolare i punti da considerare, eccetera. Per il momento ho solo abbozzato qualche idea, per esempio: ...»

In poco tempo ha preso forma un elenco piuttosto corposo di punti a cui mettere mano e il lavoro al nuovo pacchetto è incominciato sul serio; nel febbraio del 2008 ne è uscita la prima versione di test. Dopo averci lavorato ancora un po' e aver scelto un nuovo nome, il 16 aprile 2008 è approdata a CTAN la prima versione ufficiale di siunitx (Wright, 2018).

3 Caratteristiche chiave

siunitx presenta le proprie caratteristiche fondamentali fin da quella prima versione e sono piuttosto conosciute. Le riassumerei come segue.

- Formattazione automatica e semantica delle quantità (numeri con unità).
- Analisi e gestione dei numeri.
- Controllo della stampa di numeri, unità e quantità.
- Allineamento dei numeri nelle tabelle.
- Interfaccia ⟨*chiave*⟩-⟨*valore*⟩ unificata per il controllo delle opzioni.

Per quanto mi riguarda, il pacchetto ha sempre riguardato le *unità* ed è rimasto fedele all'idea fondamentale per cui un input come

```
\joule\per\mole\per\kelvin
```

possa restituire $\mathrm{J\,mol^{-1}\,K^{-1}}$, $\frac{\mathrm{J}}{\mathrm{mol\,K}}$ o $\mathrm{J/(mol\,K)}$ a seconda delle opzioni attive. Quest'idea c'è fin dal primo giorno e in questo senso il codice è rimasto più o meno immutato.

4 Dalla versione 1 alla versione 2

Fintanto che si trattava di garantire quelle caratteristiche chiave, la versione 1 di siunitx ha funzionato bene. I successivi rilasci si sono succeduti rapidamente culminando nella versione 1.4c del febbraio 2010. Tuttavia, aggiungere nuove funzionalità era un problema: internamente, infatti, il codice era un po' disastroso. Se si osserva il vecchio codice, per esempio, si vedranno:

- comandi interni *diversi da quelli dedicati all'analisi delle unità* presi da pacchetti esistenti e piuttosto a caso;
- scelte ⟨*chiave*⟩-⟨*valore*⟩ non ottimali;
- assenza quasi totale di API interne;
- cicli realizzati con codice personale di dubbia qualità;
- ...

Più o meno in questo periodo, Will Robertson mi contattò per chiedermi che cosa pensavo del linguaggio expl3 di LaTeX3 (LaTeX3 Project, 2019). Questo accadeva prima della mia entrata nel LaTeX Team, e expl3 sembrava un po' diverso da com'è oggi, anche se le idee centrali erano già tutte lì. Le idee mi piacevano, ma per il momento ero un po' cauto nel caricare una libreria esterna (e quindi una dipendenza). Così ho incominciato a selezionare le idee e a ricodificarle nella mia configurazione di sviluppo per la versione 2. Presto fu chiaro che avevo bisogno di *molte* idee e mi resi conto che sarei stato molto meglio semplicemente passando a expl3.

Lavorare alla seconda versione di siunitx mi ha portato a programmare in expl3 e a chiedere al team molte funzionalità. In particolare, piuttosto che dover ricorrere a un pacchetto esterno, volevo avere un supporto integrato per le opzioni del tipo ⟨*chiave*⟩-⟨*valore*⟩. Così, per risolvere il problema ho scritto del codice che ho chiamato keys3. La cosa si sarebbe rivelata essere la mia parte per entrare a far parte del team: oggi, infatti, lo stesso l3keys è compreso in expl3!

La riscrittura mi ha dato la possibilità di rivedere in modo significativo gli aspetti interni delle API e di migliorarne sensibilmente le prestazioni. Inoltre, sono state introdotte nuove funzionalità per gli utenti e nomi completamente nuovi per l'interfaccia ⟨*chiave*⟩-⟨*valore*⟩. Nella versione 2.0 non ho incluso la retrocompatibilità: ho imparato alla svelta, e questa caratteristica è stata presente fin da pochi giorni dopo il suo rilascio.

5 Dalla versione 2 alla versione 3

La versione 2 di siunitx conserva la maggior parte delle funzionalità presenti in quella precedente, ma, oltre a quelle buone, ne mantiene anche alcune di cattive! In particolare:

- ipotesi sui font: OpenType, eccetera;
- nessuna API a livello di codice expl3;
- comandi interni ancora troppo disordinati;
- test della sola documentazione in PDF;
- codice sorgente monolitico;
- lentezza ancora eccessiva.

5.1 Controllo dei font

Le ipotesi sui font, che ho modificato solo leggermente, sono state prese in blocco da SIstyle (Els, 2008). L'approccio attualmente seguito è il seguente.

1. Rilevamento del carattere corrente mediante i dati interni di LaTeX.
2. Inserimento del tutto all'interno di `\text`.
3. Applicazione di `\ensuremath` dentro la scatola.
4. Probabilmente, ulteriore applicazione di `\text` (per l'output in modo testo).
5. Forzatura del font con `\mathrm` o `\rmfamily`, per esempio.

Il tutto richiede molto lavoro e, cosa più importante, è inaffidabile: non è sempre facile ottenere il carattere giusto 'dentro' la sezione di output. Inoltre, il procedimento fallisce malamente con i font OpenType in modo matematico, dove le idee del TeX tradizionale sulle famiglie matematiche semplicemente non si applicano. Quindi, la versione 3 prevede un nuovo approccio.

1. Rilevamento del carattere corrente mediante i dati interni di LaTeX.
2. Impostazione di tutti gli aspetti *necessari*.
3. Uso di `\mbox` solo se è necessario modificare la versione matematica.

Questo approccio 'a cambiamento minimo' è molto più veloce di quello attuale e molto più efficace nel rispettare i cambiamenti del font. Sto ancora ultimando la compatibilità per le attuali configurazioni dei casi limite, ma credo che, nel complesso, il nuovo codice sia di molto preferibile al precedente.

5.2 Il codice delle API e il test

Lo sviluppo della versione 2 di siunitx è molto simile a quello di expl3 come linguaggio completamente utilizzabile: nel periodo in cui l'ho adoperato, expl3 è passato dall'essere una serie di esperimenti intelligenti a costituire un approccio consolidato alla scrittura di codice TeX. Ma mantenere ogni cosa di siunitx al passo con le idee è stato complicato.

Il problema più grande è che quando ho scritto il codice dell'attuale versione, c'erano solo comandi per l'utente come `\num` e relativa implementazione interna. Tuttavia, ora è chiaro che ogni comando per l'utente dovrebbe avere un'interfaccia *documentata* a livello di codice. Inoltre, queste interfacce dovrebbero essere testate a dovere: il team ha creato l3build (LaTeX3 Project, 2018) proprio per questo. Combinando queste idee, il nuovo codice avrà come input

```
\siunitx_unit_format:nN
  { \joule \per \mole }
  \l_tmpa_tl
\tl_show:N \l_tmpa_tl
```

e restituirà

```
> \l_tmpa_tl=
  \mathrm {J}\,\mathrm {mol}^{-1}
```

Si noti che questo secondo codice è facile da testare e mette in evidenza un'altra nuova idea: la fase di analisi dovrebbe produrre gli stessi risultati ottenuti da un utente che scriva la formattazione 'a mano'.

6 Prima versione alfa

Mentre sto scrivendo, lo sviluppo della versione 3 ha raggiunto il primo stadio alfa: il codice è adoperabile ma ci sono autentiche lacune. Attualmente funziona tutto quanto si elenca di seguito:

- funzioni di base:
  - analisi e formattazione delle unità;
  - formattazione dei numeri reali;
  - colonne nelle tabelle;
- API esistenti: `\num`, `\SI`, `\si`, colonna `S`;
- nuove API (sperimentali): `\unit`, `\qty`.

Ci sono alcune ampie sezioni ancora da coprire, come numeri complessi, intervalli, elenchi e, soprattutto, il livello di compatibilità per lavorare con i documenti già esistenti. Tuttavia, il tutto è relativamente gestibile e mi aspetto di terminare il lavoro verso la fine dell'anno.[1] Il 2019, dunque, dovrebbe vedere siunitx raggiungere la versione 3.0.0, sperando che mantenga il proprio posto come *il* pacchetto di LaTeX per le unità.

A Esempi

Formattazione semplice dei numeri:

123	`\num{123}`
1234	`\num{1234}`
12 345	`\num{12345}`
0,123	`\num{0,123}`
0,1234	`\num{0,1234}`
0,123 45	`\num{,12345}`
$3{,}45 \times 10^{-4}$	`\num{3,45d-4}`
-10^{10}	`\num{-e10}`

Angoli:

10°	`\ang{10}`
12,3°	`\ang{12,3}`
4,5°	`\ang{4,5}`
1°2′3″	`\ang{1;2;3}`
1″	`\ang{;;1}`
10°	`\ang{+10;;}`
−0°1′	`\ang{-0;1;}`

1. 2018. (*N.d.T.*)

Unità come macro:

$\mathrm{kg\,m\,s^{-2}}$	`\si{\kilo\gram\metre\per\square\second}`
$\mathrm{g\,cm^{-3}}$	`\si{\gram\per\cubic\centi\metre}`
$\mathrm{V^2\,lm^3\,F^{-1}}$	`\si{\square\volt\cubic\lumen\per\farad}`
$\mathrm{m^2\,Gy^{-1}\,lx^3}$	`\si{\metre\squared\per\gray\cubic\lux}`
$\mathrm{H\,s}$	`\si{\henry\second}`

Quantità:

$\mathrm{1{,}23\,J\,mol^{-1}\,K^{-1}}$	`\SI[mode=text]{1,23}{J.mol^{-1}.K^{-1}}`
$0{,}23 \times 10^7\,\mathrm{cd}$	`\SI{,23e7}{\candela}`
$1{,}99/\mathrm{kg}$	`\SI[per-mode=symbol]{1,99}{\per\kilogram}`
$1{,}345\,\frac{\mathrm{C}}{\mathrm{mol}}$	`\SI[per-mode=fraction]{1,345}{\coulomb\per\mole}`

Riferimenti bibliografici

LaTeX3 Project (2018). *The l3build package. Checking and building packages*. https://ctan.org/pkg/l3build.

— (2019). *The expl3 package and LaTeX3 programming*. https://ctan.org/pkg/expl3.

BIPM (2006). «The International System of Units (SI)». Technical report, Bureau International des Poids et Mesures.

Els, Daniel (2008). *The SIstyle package*. https://ctan.org/pkg/sistyle.

Heldoorn, Marcel e Joseph Wright (2007). *The SIunits package. Consistent application of SI units*. https://ctan.org/pkg/siunits.

Wright, Joseph (2011). «siunitx: A comprehensive (SI) units package». *TUGboat*, **32** (1), pp. 95–98. http://tug.org/TUGboat/tb32-1/tb100wright-siunitx.pdf.

— (2018). *siunitx — A comprehensive (SI) units package*. https://ctan.org/pkg/siunitx.

▷ Joseph Wright
Morning Star
2, Dowthorpe End
Earls Barton
Northampton NN6 0NH
United Kingdom
joseph dot wright (at)
morningstar2.co.uk

Il concetto di ritorno al passato per le classi e i pacchetti*

Frank Mittelbach

Sommario

Nel 2015 è stato introdotto il concetto di "ritorno al passato" per il nucleo di LaTeX. Questa nuova funzionalità ci permise di apportare correzioni al software (cosa che più o meno non avvenne mai per quasi due decadi) mantenendo però la compatibilità col passato al massimo grado.

In questo articolo spieghiamo come ora abbiamo esteso questo concetto, che inizialmente non era stato implementato, al mondo dei pacchetti e delle classi. Siccome le classi e i pacchetti di estensione hanno requisiti diversi rispetto al nucleo di LaTeX, l'approccio è diverso (e più semplice). Ciò dovrebbe rendere più facile agli sviluppatori applicare la nuova tecnica ai loro pacchetti, e agli autori usarla quando è necessario.

Abstract

In 2015 a rollback concept for the LaTeX kernel was introduced. Providing this feature allowed us to make corrections to the software (which more or less didn't happen for nearly two decades) while continuing to maintain backward compatibility to the highest degree.

In this paper we explain how we have now extended this concept to the world of packages and classes, which was not covered initially. As classes and extension packages have different requirements compared to the kernel, the approach is different (and simplified). This should make it easy for package developers to apply it to their packages and authors to use when necessary.

1 Introduzione

Nel 2015 abbiamo introdotto il concetto di "ritorno al passato" per il nucleo di LaTeX,[1] che permette all'utente di richiedere il ritorno a una versione del kernel precedente rispetto a una specifica data mediante l'uso del pacchetto latexrelease (The LaTeX Team, 2018b). Per esempio

```
\RequirePackage[2016-01-01]{latexrelease}
```

serve per disfare tutte le modifiche del kernel (correzioni ed estensioni) apportate tra il primo di gennaio 2016 e la data corrente.[2] 'Disfare' significa reinstallare le definizioni correnti alla data richiesta e anche rimuovere nuovi comandi dalla memoria di TeX, cosicché `\newcommand` e dichiarazioni simili non falliscano per il fatto che un nome di comando è già stato definito.

Questo meccanismo aiuta nell'elaborare correttamente vecchi documenti che contengono rappezzi per cose assenti nei vecchi kernel, ma che nel frattempo sono state sistemate e che quindi farebbero fallire la compilazione del vecchio documento, oppure produrrebbero file di uscita diversi, se compilati con il nuovo kernel aggiornato.

Se necessario, il pacchetto latexrelease consente anche un "ritorno al futuro" senza bisogno di installare il nuovo formato. Per esempio, se l'installazione corrente è quella del 2016-04-01, ma si dispone di un documento che richiede un kernel datato 2018-01-01, allora questo si può ottenere richiedendo

```
\RequirePackage[2018-01-01]{latexrelease}
```

purché si disponga di una versione del pacchetto latexrelease che conosca le modifiche del kernel tra la data di quello di cui si dispone e la data richiesta. Procurarsi questa versione del pacchetto è semplice, poiché quella più recente può sempre essere scaricata da CTAN. Perciò si può elaborare il documento correttamente anche quando aggiornare l'installazione completa del sistema TeX non è consigliabile o è impossibile per una ragione qualsiasi.

Tuttavia, ritornare al passato con il kernel è soltanto metà dell'opera: l'universo LaTeX consiste di moltissimi pacchetti aggiuntivi, e questi non sono stati coinvolti nel ritorno al passato del kernel. Noi ora stiamo estendendo questo concetto mediante un metodo più semplice da usare con i pacchetti e le classi; un metodo che riteniamo di uso più diretto per gli sviluppatori e anche d'uso più facile per gli autori.

Differentemente dal metodo usato per il kernel, che rintraccia ogni modifica individualmente ed è capace di tornare al preciso stato che esso aveva a ogni specifica data, il nuovo metodo per i

*Questa è la traduzione in italiano, eseguita da Claudio Beccari, dell'articolo di Frank Mittelbach, *A rollback concept for packages and classes*, pubblicato su *TUGboat*, vol. 39, n. 2 (2018), p. 107-112. Eventuali errori sono da attribuire al traduttore.

1. D'ora in poi, sostituirò la locuzione 'nucleo di LaTeX' con la parola *kernel*. *N.d.T*

2. Ci sono delle eccezioni, in quanto alcune modifiche sono conservate: per esempio, l'abilità di accettare date nel formato ISO (p.es. 2016-01-01) in aggiunta alla precedente convenzione (p.es. 2016/01/01). Queste non sono riportate indietro perché rimuovere queste funzionalità potrebbe portare a inutili manchevolezze.

pacchetti e le classi dovrebbe applicarsi solo ai cambiamenti di maggiore importanza, per esempio, l'introduzione di nuove funzionalità o di modifiche (incompatibili) nella sintassi o nelle interfacce.[3]

Siccome avremo solo pochi punti di "ritorno" per ogni pacchetto o classe, le differenti versioni sono conservate in file diversi. Il file principale, dunque, necessita di una sola dichiarazione per ogni versione per consentire il ritorno al passato. L'inconveniente, perciò, consiste nel fatto che per ogni nuova versione con cambi importanti bisogna salvare l'intero file, invece di gestire solo le differenze fra versioni. Questo è il motivo per il quale questo approccio dovrebbe essere usato solo per modifiche importanti, cioè poche lungo la vita di un pacchetto.

Da un punto di vista tecnico è anche possibile usare il metodo introdotto con latexrelease, consistente, cioè, nel marcare le modifiche, con i comandi `\IncludeInRelease` e `\EndIncludeInRelease` — la documentazione del pacchetto (The LaTeX Team, 2018b) indica come applicarli nello scenario di un pacchetto — ma il loro uso nel codice del pacchetto è complicato e produce un codice difficile da leggere, specialmente quando vi sono molti cambiamenti di minore importanza. Questo prezzo da pagare è accettabile per un codice piuttosto stabile, come lo stesso kernel, poiché consente un controllo completo sul ritorno a qualsiasi data, ma non è davvero pratico nello sviluppo di un pacchetto o di una classe, cosicché, a nostra conoscenza, fino a oggi non è stato molto usato. Il paragrafo 5.4 dà alcuni consigli su come ottenere un controllo fine in modo più semplice.

2 Gli scenari tipici

Un esempio tipico, per il quale questa funzionalità di ritorno avrebbe prodotto grandi benefici (e lo farà per i pacchetti in futuro), è dato dal pacchetto caption di Axel Sommerfeldt. Questo pacchetto cominciò sotto il nome di caption con una certa interfaccia per l'utente. Dopo un po' di tempo, fu chiaro che in essa erano presenti delle manchevolezze nell'interfaccia per l'utente: per aggiustarle evitando di compromettere documenti già composti, Axel produsse il pacchetto caption2. Successivamente anche la sintassi di quello stesso pacchetto venne cambiata e nacque caption3, che alla fine venne rinominato caption. Ora i vecchi documenti che usavano il primo caption non possono più essere compilati senza modifiche, mentre i documenti del periodo intermedio richiedono ancora caption2 (che è dichiarato superato su CTAN, ma viene ancora distribuito con le distribuzioni più importanti). Di conseguenza, gli utenti abituati a copiare il preambolo da un loro documento al successivo, continuano ad usarlo senza notare che stanno usando un pacchetto difettoso con un'interfaccia limitata.

Un altro esempio è dato dal pacchetto fixltx2e, che per molti anni ha fornito delle correzioni al kernel. Nel 2015 queste correzioni sono state introdotte nel kernel, cosicché questo pacchetto è diventato un involucro quasi vuoto, che si limita a informare l'utente che non ha più bisogno di essere usato. Tuttavia se si compila un documento antecedente al 2015 che carica fixltx2e, le correzioni che quel pacchetto forniva (come, per esempio, le correzioni all'algoritmo di gestione degli oggetti flottanti) verrebbero persi se anche fixltx2e non ritornasse al passato — fortunatamente lo fa e l'ha fatto sempre, così che in effetti non è un involucro vuoto.

Un altro esempio, un po' differente, è dato da amsmath, che per circa un decennio non ha subito alcuna correzione, anche se diversi problemi erano emersi durante quel periodo di tempo. Se quelle manchevolezze venissero finalmente corrette, esse produrrebbero effetti su molti documenti prodotti dal 2000 in poi, in quanto i loro autori avrebbero provveduto a correggere a mano questa o quella manchevolezza. Naturalmente, come per il pacchetto caption, si sarebbero potuti introdurre i pacchetti amsmath2, amsmath3, ..., ma ciò caricherebbe inutilmente l'utente, costretto a specificare sempre l'ultima versione (invece di usare automaticamente la versione più recente, a meno che non sia espressamente necessaria una versione precedente).

3 L'interfaccia del documento

Per impostazione predefinit,a LaTeX usa automaticamente la versione corrente di qualunque classe o pacchetto — e prima di offrire il concetto di ritorno al passato, ha fatto sempre così, a meno che il pacchetto o la classe non disponesse un suo proprio meccanismo per fornire la versione giusta, o usando nomi alternativi, o selezionando a mano determinate opzioni.

3.1 Ritorno al passato globale

Con il nuovo concetto di ritorno al passato, tutto ciò che l'utente deve fare (se desidera compilare il proprio documento con una specifica versione del kernel o dei pacchetti) è aggiungere la chiamata al pacchetto latexrelease all'inizio del suo preambolo specificando la data nelle opzioni, come mostrato nel primo esempio:

```
\RequirePackage[2018-01-01]{latexrelease}
```

Questo produce il ritorno al passato del kernel allo stato in cui era quel giorno (come descritto in precedenza) e per ogni pacchetto o classe controlla se ci sono versioni alternative e sceglie quella più appropriata per quel pacchetto o classe in rapporto alla data specificata.

3. Le versioni contenenti tali importanti modifiche verrano definite nel seguito *versioni maggiori. N.d.T*

3.2 Ritorno al passato specifico

C'è un altro possibile aggiustamento fine: sia `\documentclass` sia `\usepackage` dispongono di un secondo argomento (poco conosciuto e quindi poco usato) che fino ad oggi poteva consentire di specificare una "data minima". Per esempio, specificando

```
\usepackage[colaction]
           {multicol}[2018-01-01]
```

si richiede che il pacchetto multicol non sia più vecchio dell'inizio del 2018. Se è disponibile solo una versione precedente, allora la compilazione del documento produce un avviso:

```
LaTeX\ Warning: You have requested, on input
   line 12, version '2028-01-01' of package
   multicol, but only version
   '2017/04/11 v1.8q multicolumn formatting
   (FMi)' is available.
```

L'idea dietro questo approccio è che i pacchetti raramente cambiano sintassi in un modo incompatibile, ma più spesso aggiungono nuove funzionalità: con una dichiarazione di questo genere vuol dire che si richiede una versione che offre certe funzionalità.

Il nuovo concetto del ritorno ora estende l'uso di questo argomento facoltativo permettendo di fornire una data precisa per il ritorno. Questo effetto si ottiene premettendo il segno di 'uguale' alla stringa della data. Per esempio:

```
\usepackage{multicol}[=2017-06-01]
```

richiede una versione di multicol corrispondente a quella che aveva nel giungo 2017.

Perciò, supponendo che in futuro avvenga una modifica importante di questo pacchetto che cambia il modo di bilanciare le colonne, la specifica precedente richiederebbe un ritorno del pacchetto alla versione di oggi, che ha la data 2017-04-11. Il vecchio modo di usare quell'argomento facoltativo è ancora disponibile perché è determinato dalla presenza o dall'assenza del segno `=`.

Lo stesso meccanismo è disponibile per le classi di documento attraverso la dichiarazione `\documentclass` e per `\RequirePackage`, nel caso in cui fosse necessario usarlo.

3.3 Specificare la versione invece della data

Specificare la data di ritorno è particolarmente adeguato se si vuole assicurare che il comportamento del programma di composizione (cioè il kernel e tutti i pacchetti) corrisponda a quella data specifica. Infatti, non appena si è finito di editare un documento, lo si può preservare per i posteri semplicemente aggiungendo questa riga di codice:

```
\RequirePackage[⟨data di oggi⟩]
               {latexrelease}
```

Questo significa che la sua compilazione sarà eseguita un po' più lentamente (poiché il kernel potrebbe dover ritornare al passato e ogni pacchetto viene controllato per versioni alternative), ma avrebbe il vantaggio che la compilazione, anche in un lontano futuro, probabilmente funzionerà ancora, senza che sia necessario aggiungere quella linea di codice in quel lontano momento.

Tuttavia, in un caso come quello di caption o, per esempio, di longtable, che potrebbe ricevere un aggiornamento importante dopo diversi anni, sarebbe bello poter specificare una versione per nome, invece che attraverso una data: per esempio, di quest'ultimo un utente potrebbe voler usare la versione 4, invece della versione 5, qualora queste due versioni abbiano sintassi differenti e incompatibili o producano risultati diversi.

Ciò è possibile anche ora se lo sviluppatore definisce per i pacchetti o per le classi le loro versioni per nome: l'utente può allora richiedere una versione per nome semplicemente usando questo secondo argomento con il nome preceduto da una segno di 'uguale'. Per esempio, se esisterà una nuova versione di longtable e la vecchia versione (quella che oggi è corrente) è etichettata "`v4`", allora per selezionare la vecchia versione sarà possibile usare semplicemente

```
\usepackage{longtable}[=v4]
```

Si noti che non è necessario sapere che la nuova versione ha la data 2018-04-01 (nemmeno richiedere una data precedente a questa) per usare ancora quella specifica vecchia versione.

Il nome della versione è una stringa arbitraria a discrezione dell'autore del pacchetto — ma, attenzione!, non deve essere scambiata con una data, cioè non deve contenere trattini o barre, poiché questi segni confonderebbero la routine di analisi della stringa che dovrebbe indicare una data.[4]

3.4 Dati errati

L'interfaccia per l'utente è piuttosto semplice e, per mantenere alta la velocità di elaborazione, l'analisi sintattica è molto leggera. Di base viene usata la stessa routine di parsing del kernel, che è piuttosto intollerante se trova dati inattesi.

In sostanza, ogni stringa che contiene un trattino e una barra fa partire l'algoritmo per analizzare una data che, quindi, si aspetta due trattini (per il caso di una data ISO) o due barre (altrimenti) e, oltre questi separatori, solamente cifre. Se trova qualsiasi altra cosa, è possibile che l'utente riceva il messaggio d'errore "`Missing \begin{document}`" o, magari, potrebbe avere luogo un riconoscimento strano e potrebbe essere eseguita una strana selezione. Per esempio, una data del tipo `2011/02` può

4. Ovviamente, un'analisi più sofisticata della stringa potrebbe sistemare questo aspetto, ma noi usiamo una routine semplice e veloce che cerca solo barre e trattini senza nessuna ulteriore analisi.

voler dire per noi "febbraio 2011", ma per la routine di analisi è un qualche giorno dell'anno 20 d.C. In altre parole, la stringa viene convertita nel solo numero `201102`, così che quando questo numero viene confrontato con il numero 20000101, sarà minore, e quindi precedente, anche se il secondo è la rappresentazione numerica del 1º gennaio 2000.

Pertanto, ultimo avviso: non si scrivano date ortograficamente errate e tutto andrà bene. Non è stato un problema in passato, perciò si spera che continui ad andar bene con questo modo semplice di analizzare le stringhe. Se non fosse così, saremmo costretti ad eseguire controlli più estesi durante l'analisi della stringa.

3.5 Consigli per i nuovi utenti

Se un documento usa le nuove funzionalità di ritorno al passato globale, dovrebbe essere compilabile con qualsiasi installazione più recente dell'inizio del 2015, quando è stato reso disponibile per la prima volta il pacchetto latexrelease. Se l'installazione è più datata, allora è necessario un aggiornamento, oppure aggiungere la versione corrente di latexrelease.

Tuttavia, se il documento usa il nuovo concetto del ritorno individuale per i pacchetti e le classi (cioè con la sintassi `=...` nell'argomento facoltativo) è necessario usare un'installazione del 2018 o successiva.[5] Le distribuzioni precedenti si bloccherebbero appena trovano il segno di uguale, perché si aspettano di trovare solo una stringa che corrisponde a una data valida.

4 L'interfaccia della classe o del pacchetto

Il meccanismo di ritorno al passato per i pacchetti e le classi viene reso possibile mettendo, all'inizio del file di codice, una sezione di dichiarazioni che informano il kernel in merito all'esistenza di versioni alternative.

Queste dichiarazioni devono precedere tutto il codice e devono essere in ordine di data perché il meccanismo di caricamento le deve valutare una ad una e, quando una versione adeguata è stata trovata, viene caricata e l'elaborazione del file principale del pacchetto o delle classe può terminare. Se non ci sono queste dichiarazioni, o se le vecchie versioni sono state cancellate per qualsiasi motivo, l'elaborazione continua leggendo tutto il file principale.

Le versioni precedenti sono contenute in file distinti, uno per ciascuna versione, e suggeriamo di usare uno schema del tipo ⟨*nome del pacchetto*⟩`-`⟨*data*⟩`.sty` perché questo è facile da capire e viene ordinato facilmente per data in una cartella. Tuttavia, qualsiasi altro schema va altrettanto bene, perché il nome fa parte della dichiarazione.

5. In alternativa, si potrebbe tentare il "ritorno al futuro" usando una versione corrente di latexrelease e specificando una data adeguata.

Il contenuto di questa versione di file è semplicemente il pacchetto o la classe in uso precedentemente. Questo significa che prima di fare una nuova versione, tutto ciò che bisogna fare consiste nel disporre di una versione verbatim del file corrente e nel darle un nuovo nome adeguato.[6]

In questo modo è anche immediato includere vecchie versioni dopo quanto si è fatto; in altre parole, riprendendo l'esempio di caption, Axel potrebbe assegnare alla sua primissima versione il nome caption-⟨*una data*⟩, a caption2 il nome caption-⟨*un'altra data*⟩, oltre a fornire le necessarie dichiarazioni alla versione corrente.

Le dichiarazioni necessarie nel file principale sono fornite dai comandi `\DeclareRelease` e `\DeclareCurrentRelease`, che devono essere usati nella sezione *selezione delle versioni* all'inizio del file. Per ciascuna precedente versione si può specificare un ⟨*nome*⟩, la ⟨*data*⟩ di quando quella versione era diventata disponibile e il ⟨*file esterno*⟩ che contiene il codice.

La sintassi per dichiarare le precedenti versioni è la seguente:

`\DeclareRelease`
`{`⟨*nome*⟩`}{`⟨*data*⟩`}{`⟨*file esterno*⟩`}`

Degli argomenti ⟨*nome*⟩ e ⟨*data*⟩, o l'uno o l'altro può essere vuoto, ma non tutti e due contemporaneamente. Se non si specifica la ⟨*data*⟩, è per riferirsi a versioni "beta", che gli utenti possono esplicitamente richiamare ma che non dovrebbero giocare alcun ruolo nel ritorno al passato.

Anche la versione corrente riceve una dichiarazione, ma con due soli argomenti: un ⟨*nome*⟩ (che di nuovo può essere vuoto) e una ⟨*data*⟩, perché il codice per questa versione costituisce il resto del file corrente:

`\DeclareCurrentRelease{`⟨*nome*⟩`}{`⟨*data*⟩`}`

Questa dichiarazione deve essere l'ultima nella sequenza delle dichiarazioni e termina l'elaborazione della *selezione delle versioni*.

L'ordine delle altre versioni deve iniziare con la più vecchia e procedere verso la più recente, perché il meccanismo di caricamento confronta ogni dichiarazione di versione con la data specificata per il ritorno al passato e finisce nel momento in cui ne trova una più recente della data specificata. Esso quindi seleziona la precedente versione, cioè una che abbia una data precedente o uguale a quella desiderata. Poiché le dichiarazioni eseguite con `\DeclareRelease` con l'argomento ⟨*data*⟩

6. Invece di fare una copia verbatim, si potrebbe aggiustare il commento aggiunto dal pacchetto docstrip all'inizio del file. Sebbene la cosa sia tecnicamente corretta, può ingenerare confusione se il file contiene la frase "was generate from ...", visto che ora si tratta di una versione congelata che rappresenta una particolare situazione nel tempo, piuttosto che una situazione che riguarda un file generato in un certo momento ma che può venire rigenerato quandunque si voglia se è necessario.

vuoto non giocano nessun ruolo nel ritorno al passato, esse possono essere messe dove si vuole nella sequenza delle dichiarazioni.

Un tipico esempio di *sezione delle versioni* è all'inizio del pacchetto multicol; ha l'aspetto seguente perché vi è stato un aggiornamento importante nell'aprile 2018. Si noti che a causa di alcune piccole modifiche eseguite dopo quella data, la data effettiva del pacchetto è già quella di giugno.

```
\NeedsTeXFormat{LaTeX2e}[2018-04-01]
\DeclareRelease{}{2017-04-01}
                 {multicol-2017-04-01.sty}
\DeclareCurrentRelease{}{2018-04-01}
\ProvidesPackage{multicol}[2018/06/26 v1.8t
              multicolumn formatting (FMi)]
```

Se il ritorno al passato si affida a un nome invece che a una data, il meccanismo lavora nello stesso modo, salvo che viene scelta una versione in base al nome che sia coincidente con quello richiesto. Se nessun nome coincide con quello richiesto viene emesso un avviso d'errore e l'elaborazione prosegue usando la versione corrente.

5 Considerazioni speciali per gli sviluppatori

Quando viene caricata una precedente versione di un pacchetto o una classe, entrambi i comandi di dichiarazione di versione sono cambiati in comandi *no-op*, inerti, così che, nel caso in cui i file che contengono il codice abbiano altre simili dichiarazioni, esse non producano alcun effetto. Questo permette di copiare semplicemente il codice di una versione da sostituire con una più recente, senza toccarne affatto il contenuto. Naturalmente, togliendo quelle dichiarazioni dai file delle versioni precedenti non si produce alcun danno e si rende l'elaborazione e il caricamento leggermente più veloce.

Come detto in precedenza, il miglior modo per specificare i nomi di versione è quello di appendere la data di versione al nome del pacchetto o della classe, ma l'argomento *file esterno* consente anche altri schemi.

Ci si può domandare perché si debba eseguire una dichiarazione per la versione corrente, visto che è poi seguita dalla dichiarazione `\Provides...`, che contiene anch'essa una data e una stringa descrittiva, e potrebbe segnalare essa stessa la fine della sezione delle versioni. Il motivo è questo: se si vuol dare alla versione corrente un nome, è bene che questo sia una cosa semplice, come `v4` (da mantenere così) anche se la versione corrente tecnicamente è già alla versione `v4.2c` e viene indicata così nella dichiarazione `\ProvidesPackage`. Per lo stesso motivo (visto che non necessariamente ogni versione con modifiche minori viene indicata come una versione diversa alla quale si possa applicare il ritorno al passato), la ⟨*data*⟩ nella dichiarazione `\DeclareCurrentRelease` riflette quella in cui è stata introdotta la versione con cambiamenti importanti. Pertanto, dopo un po' quella data potrebbe essere precedente a quella del pacchetto corrente.

5.1 Utenti dei primi tempi dall'introduzione del nuovo metodo

Per un periodo di uno o due anni dall'introduzione di questo nuovo metodo, c'è il pericolo che gli utenti con installazioni precedenti scelgano una forma specifica di un pacchetto, per esempio da CTAN, che contenga una dichiarazione con la quale il kernel (del 2017 o precedente) è del tutto incompatibile. Perciò, può essere raccomandabile che gli sviluppatori aggiungano anche le righe seguenti all'inizio dei pacchetti e delle classi quando usano questa funzionalità di ritorno al passato:

```
\providecommand\DeclareRelease[3]{}
\providecommand\declareCurrrentRelease[2]{}
```

In questo modo le dichiarazioni vengono ignorate nel caso che il vecchio kernel non sappia cosa farsene.

Una alternativa potrebbe essere quella di aggiungere una riga che stabilisce una data minima per la versione del kernel, cioè

```
\NeedsTeXFormat{LaTeX2e}[2018-04-01]
```

cosicché gli utenti ricevono un chiaro messaggio d'errore, cioè che essi devono aggiornare la loro installazione se vogliono usare il file corrente.

5.2 Una nuova versione maggiore a livello beta

Se si sta lavorando su una nuova versione maggiore del proprio pacchetto o della propria classe, si potrebbe metterla a disposizione anche per il pubblico, cosi che la si possa collaudare e si possa ricevere il necessario feedback. In questo caso la versione corrente è quella ufficiale da usare come predefinita, mentre la versione "beta" potrebbe essere selezionata solo espressamente. Per ottenere questo risultato, si può aggiungere

```
\DeclareRelease{beta}{}{⟨file esterno⟩}
```

prima di

```
\DeclareCurrentRelease{}{⟨data⟩}
```

cosicché i collaudatori possono esplicitamente accedere alla nuova versione richiedendola mediante

```
\usepackage[⟨opzioni⟩]{⟨pacchetto⟩}[=beta]
```

mentre gli altri, che caricano il pacchetto senza l'argomento supplementare, caricano la versione corrente.

5.3 Due nuove versioni già in uso

Uno scenario particolare per il quale questo metodo è adatto solo parzialmente è quello in cui ci sono due versioni maggiori entrambe continuamente in uso e mantenute attivamente (cioè ricevono correzioni e altri aggiornamenti di tanto in tanto). In questo caso è necessario considerare una versione come primaria e consentire all'altra (e i suoi aggiornamenti) di essere accessibile mediante i nomi: il ritorno al passato, ovviamente, funziona solo con la linea principale di sviluppo.

Per esempio, se entrambe le versioni `v4` e `v5` del pacchetto foo sono in uso, e se si considera la versione `v5` quella da portare avanti (anche se si sta ancora facendo manutenzione alla versione `v4`), allora si può seguire una strategia come la seguente:

```
% last v4 only release:
  \DeclareRelease{}{2017-06-23}
                   {foo-2017-06-23.sty}
% first v5 release:
  \DeclareRelease{}{2017-08-01}
                  {foo-2017-08-01.sty}
% patch to v4 after v5 got introduced:
  \DeclareRelease{v4.1}{}
                   {foo-v4-2017-09-20.sty}
% patch to v5:
  \DeclareRelease{}{2017-08-25}
                   {foo=2017-08-25.sty}
% another patch to v4:
  \DeclareRelease{v4.2}{}
                   {foo-v4-2017-10-01.sty}
% nickname for the latest v4 if you want
% users to have simple access via a name
  \DeclareRelease{v4}{}
                   {foo-v4-2017-10-01.sty}
% current v5 with further patches:
  \DeclareCurrentRelease{v5}{2018-01-01}
```

In questo modo, gli utenti possono usare `\usepackage{foo}[=v4]` per accedere all'ultima versione `v4`, oppure usano il nome più specifico come `[v4.1]` per accedere a una versione particolare. Questo significa che se si richiede il pacchetto foo nella versione `v4` (o una delle sue sotto versioni), esso non sarà cambiato, anche se esiste una specificazione di ritorno al passato attraverso latexrelease.

Questo normalmente dovrebbe andare bene, ma se veramente si chiede un ritorno al passato automatico per entrambe queste versioni maggiori, perché sono in effetti dello stesso rango, allora si sta essenzialmente dicendo che sono due lavori distinti con qualche cosa in comune. In questo caso, si dovrebbero usare due nomi separati, per esempio, chiamando la versione precedente foo-v4, quando si introduce la versione 5 di foo e da quel momento in poi le si gestisce in modo indipendente.[7]

7. Sebbene in rari casi questo potrebbe essere l'approccio migliore, si cerchi di evitarlo, perché nel lungo termine la gestione potrebbe diventare, come minimo, problematica.

5.4 Controllo fine (se necessario)

Come detto prima, l'interfaccia è deliberatamente progettata per essere semplice e facile da usare. Come prezzo da pagare, ogni ritorno al passato (per impostazione predefinita) punta a un file separato. L'idea dietro a tutto ciò è che non sembra una gran cosa stare dietro ad ogni singolo minimo cambiamento come punto di ritorno al passato, ma che questi siano solo quelli che possono alterare il comportamento di un pacchetto nell'elaborazione di un documento cosi che, quando si compilano documenti più vecchi, sia importante essere capaci di andare indietro a una data precedente.

Tuttavia, se ci si trova in una situazione in cui si hanno molti file per tornare indietro con differenze minori, e lo si consideri insoddisfacente, allora si può ricorrere a un altro comando che permette di combinare diversi file in un unico file. Dentro un file esterno, nominato in una dichiarazione `\DeclareRelease`, si può usare

`\IfTargetDateBefore{`⟨*data*⟩`}`
`{`⟨*codice prima di*⟩`}{`⟨*codice da*⟩`}`

Questo va usato dopo la sezione *selezione delle versioni* (se presente) e ha il seguente effetto: se l'utente ha richiesto, per esempio, `[=2017-06-01]`, il meccanismo prima seleziona il file che era corrente a quella data, cioè la versione che era stata resa disponibile in quella data o prima di quella data, poi la ⟨*data*⟩ viene confrontata con quella specificata, `2017-06-01` e, a seconda dell'esito del confronto, esegue il ⟨*codice prima di*⟩ quella data, oppure il ⟨*codice da*⟩ quella data in poi.

In questo modo un singolo file esterno può contenere informazioni per il ritorno al passato per diversi aggiustamenti eseguiti in date diverse; ma, naturalmente, il carico di lavoro ricade sullo sviluppatore che deve aggiungere i comandi appropriati, e questo è un po' più oneroso che non semplicemente copiare un file cambiandogli il nome.

L'alternativa consiste nell'usare `\IncludeInRelease` e `\EndIncludeInRelease`. La documentazione di latexrelease (The LaTeX Team, 2018b) dà consigli su come usare in pratica questi comandi.

5.5 L'uso di l3build per la gestione dei sorgenti

Se si usa l3build, (The LaTeX Team, 2018a), per gestire i propri file sorgente, è necessario essere certi che i file per le versioni precedenti siano copiati nella distribuzione; per consentire questo, la configurazione di default per l3build specifica

```
sourcefiles = {"*.dtx", ".ins",
               "*-????-??-??.sty"}
```

cioè che tutti i file `.dtx` e `.ins`, insieme a quelli con estensione `.sty`, secondo la convenzione consigliata in questo articolo per rinominare le versioni

precedenti, siano automaticamente inclusi nella costruzione del pacchetto di distribuzione dei file.

Se si preferisce una convenzione diversa, bisogna aggiustare le impostazioni nel file `build.lua` del proprio progetto. Altrimenti, bisogna andare avanti senza aggiustamenti.

6 Sommario dei comandi

6.1 L'interfaccia del documento per gli utenti

Per specificare un ritorno al passato di tipo globale si usi

`\RequirePackage[`⟨*data*⟩`]{latexrelease}`

all'inizio del proprio documento.

Per un ritorno al passato di singoli pacchetti o classi, si usi il secondo argomento facoltativo precedendo la data con il segno di uguale, cioè

`\documentclass[`⟨*opzioni*⟩`]{`⟨*classe*⟩`}[=`⟨*data*⟩`]`
`\usepackage[`⟨*opzioni*⟩`]{`⟨*pacchetto*⟩`}[=`⟨*data*⟩`]`

6.2 L'interfaccia di classi e pacchetti per gli sviluppatori

Per dichiarare una versione precedente o speciale si usi

`\DeclareRelease{`⟨*nome*⟩`}{`⟨*data*⟩`}`
`{`⟨*file esterno*⟩`}`

Si lasci ⟨*nome*⟩ vuoto se si desidera che il ritorno al passato avvenga solo attraverso le date. Si lasci ⟨*data*⟩ vuoto se questo file speciale deve essere accessibile solo attraverso il ⟨*nome*⟩.

In ogni caso si termini la *selezione delle versioni* con la dichiarazione relativa alla versione corrente

`\DeclareCurrentRelease{`⟨*nome*⟩`}{`⟨*data*⟩`}`

In questa dichiarazione bisogna specificare la ⟨*data*⟩ ma il ⟨*nome*⟩ può essere lasciato vuoto (cosa che rappresenta il caso normale).

Nel file corrispondente all'ultima versione (ma dopo la sezione *selezione delle versioni*), si può specificare del codice condizionale da eseguire per un ritorno al passato con una data specifica, usando

`\IfTargetDateBefore{`⟨*data*⟩`}`
`{`⟨*codice prima di*⟩`}{`⟨*codice da*⟩`}`

Riferimenti bibliografici

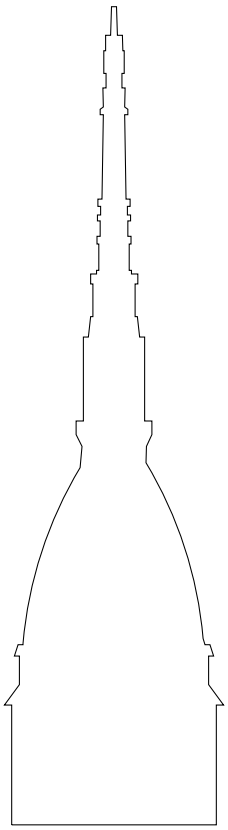
The LaTeX Team (2018a). *The l3build package — Checking and building packages*. Leggibile con `texdoc l3build`. La versione del 2015 si trova in https://ctan.org/pkg/l3build.

— (2018b). *The latexrelease package*. Leggibile con `texdoc latexrelease`. Altrimenti disponibile in https://www/latex-project.org/help/documentation.

▷ Frank Mittelbach
https://www.latex-project.org

Questa rivista è stata prodotta
dal Gruppo Utilizzatori Italiani di TEX
usando esclusivamente software libero.

Versione elettronica per la diffusione via web.



GUIT meeting 2019

Sedicesimo convegno nazionale su TEX, LATEX e tipografia digitale

Torino, 26 ottobre 2019

Sabato 26 ottobre a Torino si terrà il sedicesimo Convegno annuale su TEX, LATEX e tipografia digitale organizzato dal Gruppo Utilizzatori Italiani di TEX. Il Convegno sarà un momento di ritrovo e di confronto per la comunità LATEX italiana, tramite una serie di interventi atti sia a contribuire all'arricchimento sia a supportarne lo sviluppo. Maggiori informazioni sul Convegno e sulle modalità di presentazione degli interventi saranno disponibili all'indirizzo:

`https://www.guitex.org/home/guit-meeting-2019/`

ArsTEXnica

Rivista italiana di TEX e LATEX

Numero 27, Aprile 2019

90027
9 771828 236001