Modularity in LaTeX

Matt Swift
59 Brainerd Rd #202
Allston MA 02134-4564
USA
Email: <swift@bu.edu>

Abstract

The author surveys several kinds of desirable modularity in LaTeX and argues the advantages of a system in which sources and macros may inhabit modules called bits and features, respectively, that are independent of their context in a disk file and are identified by names independent of disk file names. The author discusses implementation, and sketches are given of solutions involving extensions, enhancements, front ends, and back ends to TeX. The author has made available code which adds some new modular features to LaTeX.

This paper aims to clarify several issues in the use and development of LaTeX that belong under the general heading of modularity. The issues have arisen during attempts to solve particular problems.

Certain modular features are not worth implementing in LaTeX 2$_\varepsilon$ and should instead be the concern of those who are designing and implementing LaTeX3, ε-TeX, $\mathcal{NTS}$, and the tools that will accompany them. Other modular features can be implemented in LaTeX without a large programming effort or extensive changes in user syntax. The author has implemented some modest modular features in LaTeX with encouraging results.

Some terms will be given a precise definition below, but *modularity* will be used only as a broad descriptive term.

Several concerns addressed here have been addressed earlier in the companion papers "An Object-Oriented Programming System in TeX" (Baxter, 1994) and "Object-Oriented Programming, Descriptive Markup, and TeX" (Ogawa, 1994) published in the TeX Users Group '94 proceedings (*TUGboat* 15, no. 3). I recommend these thoughtful papers to readers interested in this subject. Baxter describes a LaTeX-like markup that realizes many of the benefits of object-oriented program design. Its syntax could for the most part exist simultaneously with standard LaTeX. Ogawa enumerates some ways LaTeX falls short of an ideal object-oriented markup language.

What is a document?

The LaTeX processor and an accompanying suite of programs and hardware devices translate a computer disk file called a *LaTeX source* ("source") into a document. One can reasonably say, in our context, that a *document* is a text-dominated visual presentation of information whose canonical form is an ordered collection of pages.

We are used to calling many things documents that are not documents in this sense. A source is not a document — not a visual presentation of information — nor is it intended to be a complete description of a document (though it is such, trivially, when it is viewed or printed). A source is a partial, abstract description of a document, a meta-description of a document. The LaTeX processor interprets the minimal information in a source by means of implicit conventions and explicit rules, and the resulting interpretation (the `dvi` file) is a complete document description. A subsequent procedure transforms the document description into a real document, a real image on a screen or page.

There are several good reasons to create and maintain sources instead of document descriptions or documents themselves. Source files are much smaller than the files that result from them, since they contain less information. They are human-readable, whereas document descriptions, in the interests of efficiency, are difficult or impossible to read. Worrying about the large amount of extra information necessary to describe a document distracts authors, to whom it is superfluous. Sources are a versatile form because they can be used to produce many different kinds of documents or even presentations of the information that are not documents in the present sense, such as aural presentations (see Raman, 1995).

Our goal is to allow a computer program to present information to our senses in a manner that meets our needs, which can change from time to time and from user to user. Ideally, marked-up text in the source makes explicit all the information we would like the document in any form to convey. Marked text is a very efficient conventional form of information. Typically, we wish to present the information in a manner that most greatly facilitates the reader's understanding by presenting various aspects of its content efficiently to the eye, which can process it extremely quickly. But there is not a single superior scheme for accomplishing this. Also, because perceiving a document is a personal experience like any other, typesetting can aim to be beautiful, or, as in advertising, manipulative. To meet these various needs, a variety of documents can be derived from the source by reprocessing it with different parameters.

Practically, it is very hard to decide what information should be made explicit by markup, and what can be assumed. When we stretch the borders of the present definition of a document, we change our ideas of what elements must be differentiated in the document and consequently discover a need to mark structures that were handled implicitly for more familiar documents. In some languages, for example, the sequence of glyphs in the source must be presented right-to-left, or top-to-bottom, or both, not in the way of written English. The notion of pages is ill-adapted to a computer-screen. And when we change the presentation's form from visual to aural, we discover that some of our markup is visual and not as abstractly informational as perhaps we thought.

One can consider a LaTeX source from several different points of view. Very basically, it is a program written in TeX, which is a "list-based macro language with late binding" (NTS, 1995), and using the LaTeX macro library. From another point of view, as mentioned, it is a meta-description of a visual presentation of text-dominated information. If a certain method of processing the source, such as standard LaTeX, is agreed upon, then it can be agreed that a source is a complete description of a document.

Our present interest suggests another point of view that considers a source to comprise two kinds of elements identified by markup (not including comments). A *block element* is a list of atoms — irreducibles such as font glyphs and spaces — other block elements, and/or anchor elements. An *anchor element* is a signal to the formatter that it should act as if certain atoms or block elements were in the source at that point.

Anchors are used when it is better that the formatter supply data from somewhere at runtime, rather than the user include it in the source document. There is a close correspondence between anchors and HTML entities (see Goossens and Saarela, 1995a). An anchor used for citation, for example, might generate two block elements such as (Hobson 1956) and an entry in a bibliography at the end of the chapter.

From a more abstract point of view, blocks and anchors are markup functions. The difference — not strict — is that blocks take an argument, often a long one, of text to be represented quite directly in the typeset image, whereas anchors do not take an argument of this kind.

A source is therefore itself a single block element. Once anchor references are resolved, it may be considered to be a single list of atoms. Any number of contiguous regions of the list are identified ("tagged") as subordinate block elements. Block elements may nest but not overlap.

In the translation from source to document, the order of the list of atoms is for the most part preserved. Certain classes of block elements, however, are conceptually independent of their contexts in the source, and routinely appear in new contexts in the document. We might call this kind of block element a *float element* as opposed to a *fixed element* whose immediate context does not change in the translation. Float elements include footnotes, LaTeX floats, marginalia, and entries in bibliographies and indices. The existence of float elements is possible exactly because block elements do not overlap in the source.

LaTeX is both the language in which sources are written and the engine which creates a document description from a source. Users must distinguish these capabilities or confusion and inefficiency may result. Regions in the list of atoms that compose the source are ideally tagged by markup that describes what the region is; that is, markup that is not specific to any particular document description derived from it. Some "visual markup", or markup that describes how the region should be actually presented, is inevitably necessary during the creation of complex documents, when the automatic procedures break down and require manual aid. What is most to be regretted, however, is the confusion and encouragement of inefficient habits that results from allowing the same syntax for both the desirable and undesirable kinds of markup.

A new block element is needed

Block elements usually constitute the majority of a source, conceptually and physically. Let us use the term *block modularity* to describe what allows LaTeX to handle block elements independently and abstractly. LaTeX already provides block modularity in several useful ways. An itemized list, for example, is a natural unit of text for which LaTeX provides the `itemize` environment whose appearance in the document is controlled by macro definitions that are in effect only inside the environment. This kind of local definition is made possible by TeX's grouping mechanism. The items within the list are blocks in their own right, and these in turn might be divided further into paragraph blocks or other kinds of blocks, such as emphasized phrases, and so on. The `document` environment is another natural unit of text, a much more general one. Its appearance is controlled by the preamble, which includes `\documentclass` and `\usepackage` commands.

These examples show that block elements are specified in the source file in a broad variety of ways. Some of the causes and consequences of this variety are adduced in another section below.

I propose as a useful concept a block element of intermediate size called a *bit* defined as a *conceptually independent unit of text*, a block element that might reasonably appear in new contexts. If the block could reasonably appear with a null context, that is, could alone generate a reasonable document, it is not only a bit but also a *potential document*. The name is appropriate because any document could conceivably be placed in a new context, though long documents are quite unlikely to appear in any but the null context.

A collection of poems or recipes would consist almost entirely of a sequence of bits. An academic paper would probably contain only one bit, its entire self, but might contain a bit here or there, such as an embedded poem, illustration, or derivation. The standard LaTeX *letter* class provides a clear example of a bit structure. The `document` environment in a source of the *letter* class consists of one or more LaTeX `letter` environments, each of which is functionally modular and conceptually independent.

Bits are similar to sections as we might think of them when we read or write. *Sections*, however, may not be conceptually independent. The LaTeX sectioning commands such as `\chapter` have limited power and serve for the most part as informative markers in a continuous stream of text. Therefore they should be considered not as tagging the long block element that follows them, but rather as an anchor specifying one fixed element (the section heading) and one float element (the entry in the table of contents).

The status of the LaTeX sectioning commands as anchors and not block delimiters is not required by their syntax; that is, not required because they occur singly and do not enclose text. The familiar sectioning commands *could* in fact be implemented to give a programmer control over the following text as a block element, but they are not implemented this way.

When we consider the modularity of sources and documents, bits are the most basic and important unit. They are atomic in the sense that every document has an integral number of them. When we think about the exchange of sources, we should think in terms of the bit.

A generally useful implementation of bits would have several key characteristics. First, there should be an anchor for bits, a command which says "put this bit here". Bits should also be as independent as possible of disk files. By this I primarily mean 1) they should be referred to by names which do not depend on the disk file in which they occur, and 2) as LaTeX program code they should be portable — as independent as possible of their immediate programming context in the disk files that contain them. A bit should have a type and a name — a unique label. To process a source is to process a bit of type "document". This task comprises relatively independent subtasks, one corresponding to each bit type, and one to handle the presentation of the entire sequence of bits and intervening text, the presentation of the document as a whole.

Many users are probably accustomed to implementing bits implicitly by taking advantage of the modularity of disk files. A bit can be put into its own disk file which can be incorporated into new contexts by the action of disk file inclusion. The bit name is the disk file name. Chief among the advantages of identifying bits explicitly in the markup is that disk file structure and disk file names can be freed from this frequent burden of bearing information about bits. This change facilitates several beneficial developments proposed below. Summarily, the introduction of a layer of abstraction between disk files and user syntax makes possible a broad and desirable flexibility in the distribution of source documents among disk files. Disk files might ideally be construed as objects which can export bits.

Many things said so far about source documents and disk files apply also to code libraries and disk files. In particular, there are similar advantages in recognizing functionally independent segments of

code through markup, rather than disk file structure, and to introducing a layer of abstraction between disk files and programmer syntax. These segments, which we can call *features* after GNU Emacs, are exact analogs of the bit in a context of programming code. The breaking up of the kernel and the standard document classes into functionally independent modules is the intended first stage of the $\mathcal{NTS}$ project (NTS, 1995).

Present limitations on LaTeX disk file structure

The TeX primitives `\input` and `\endinput` determine the possible structures of LaTeX disk files. The `\input` command is an anchor for the contents of a disk file. When the contents of a disk file are being processed at the request of an `\input` command, the `\endinput` command signals the formatter to act as if the end of the file was encountered at the end of the current line.[1]

Though many disk file configurations are possible, only one seems practical, namely, a *principal source* that contains a single `document` environment, and any number of *auxiliary sources* that contribute material to the principal source or to other auxiliary sources by disk file inclusion.

LaTeX provides two more sophisticated interfaces to the `\input` primitive. The `\include` and `\includeonly` commands implement a convenient way to enable and disable the inclusion of files. The price is that `\include` commands cannot be nested like `\input` commands. The group of class and package commands new in LaTeX 2_ε keep track of what files have been included, perform some checking to ensure that the right file was included, and implement a system of user options — a system of passing certain information to the included files.

Because both extensions are founded on the primitive `\input`, the basic modular unit is still the disk file, identified by its name. Several alternative ways to identify bits and features may now be suggested. One difficulty to keep in mind is that file names are often exactly how users choose to uniquely identify their information. Consider two versions of the same composition residing in two files in the same directory, distinguished only by a few changed words.

The possibilities seem to be:

1. Mandate a new TeX primitive

   `\inbit{`⟨*bit-name*⟩`}`

   or primitives upon which such a command could be constructed. One can imagine a `kpathsea`-like system library which handles requests for both files and bits. When a search is ambiguous, the engine could issue a warning.
2. Bend the rules and allow the back end (system-dependent implementation) of `\input` to search for bits if its argument does not match a file name. This change would not break the `trip` test, and need not be considered an change to TeX.

   The search method for bits could be path searching like `kpathsea`or involve other schemes like file stamp attributes or a catalog of cross references like the SGML Open HTML catalog file (see Appendix D of Goossens and Saarela, 1995b). When a search is ambiguous, the engine could issue a warning.

   This is the most promising solution to pursue.
3. Develop a sophisticated front-end to LaTeX that will hide from the user a preprocessing stage that assembles a conventional source document for compilation.

A back-end solution is going to be more stable and portable than a front-end solution, but it must inevitably be less ambitious and slower to appear.

Experiments in LaTeX

The author is in the process of implementing bits in a limited way within LaTeX. The ongoing project, called `Frankenstein`, will provide several bit types as well as generic routines for creating new ones. It will be a poor-man's Object-Oriented Processor (see Baxter, 1994). The anchor for bits, available in the *newclude* package, has the following syntax:

`\includebit{`⟨*bit name*⟩`}{`⟨*file name*⟩`}`

And a bit is tagged as follows:

```
\begin{⟨bit type⟩}{⟨bit name⟩}{⟨init code⟩}
...
\end{⟨bit type⟩}
```

For example, the command

```
\includebit{Ode to a Lion}{safari}
```

would include the bit that is declared in the file `safari.tex` with

```
\begin{poem}{Ode to a Lion}
    {\numberstanzas
    \newcommand\growl{{\large Gr\"owl!}}}
```

The new interface to `\input` provided by the *newclude* package implements two new basic commands, a command like `\include` but without the enclosing `\clearpage`s, and a command to include

[1] Suppressing the continuation until the end of the line is a surprisingly complex task; here, to my mind, is a good candidate for a change to TeX.

a delimited section of a disk file. The omission of the `\clearpage`s is achieved by writing out a single `aux` file per document (eliminating the additional ones for each file included with `\include`). None of the features of the old `\include` command are lost. The task of including only a delimited section of an included file is accomplished by generalizing the *verbatim* package so that all irrelevant parts of the the file can be ignored.

The success of the system so far is promising. It is now possible with LaTeX to create a document whose contents are LaTeX environments distributed among an arbitrary number of disk files. In this case these files are used as auxiliary sources, but nothing prevents them from being principal source files that can be processed independently by LaTeX. That is, they may have the usual `\documentclass` declaration and `document` environment or not, it is irrelevant.

One can imagine the following application: a teacher's 25 students turn in (or, better, make available over a network) 25 LaTeX sources containing three book reviews each. The teacher can prepare a document consisting of all the reports on a single book, or a portfolio of the best reviews of each student, or a snapshot of their work in progress. It remains possible for the students to format their sources in their own preferred manner, completely independent of the formatting the teacher chooses for the composite documents. Since the sources for each are the same, version control does not involve more than the usual task of keeping document images up to date with principal sources.

Further applications suggest themselves readily. A lecturer can extract derivations or figures or abstracts or quotations from scholarly papers and incorporate them into slide presentations. If any of the source material is revised in the future, it need only be revised in one place. Selected parts of a LaTeX source can be exported and published on the World Wide Web using `latex2html`. This is already possible using that program's conditional inclusion facilities, but the bit solution allows the same source to export different parts of itself to different web documents without need to alter it (see Goossens and Saarela, 1995b).

The limitations encountered in this system have prompted theoretical contemplation of more powerful improvements. The shortcomings include, notably, the continuing dependence of bit names on disk file names. The method of including regions of files accepts only a strict verbatim syntax and verbatim matches for the delimiters. TeX inserts `\par` at an `\input`, so that bit boundaries must correspond with paragraph boundaries.[2]

These are significant limitations, and the way forward in LaTeX is difficult. The *verbatim* and *doc* packages are evidence of how complex is the task of getting TeX to perform the offices of even a simple inflexible text stream editor. Moreover, it seems foolhardy to attempt to emulate in LaTeX what can be done in UNIX with minimal effort, and probably with no more effort on other TeX platforms. I believe these experiments in LaTeX are going to be useful, but the most satisfactory way forward must be toward one of the solutions suggested at the end of the previous section.

The implementation of block modularity

I observed above that block elements are specified in a variety of ways. Some (though not all) of the differences can be justified by an appeal to natural user syntax. It would certainly be inconvenient most of the time, for example, to have to type

```
\beginparagraph
...
\endparagraph.
```

The differences present a problem, however, to the LaTeX developer for whom, as a programmer, standards are always an advantage.

TeX's internals confront us with basic differences between those units handled by an `every*` token variable (paragraphs, lines, etc.), those handled by TeX's grouping mechanism (environments, `\item`s), and those handled by macro arguments (e.g., `\emph`). A good programmer interface would hide these differences as much as possible.

The number and format of optional and mandatory arguments to environments is nonstandard. In the present LaTeX environment, one needs to rely on a convention, such as the syntax given above for bits, the syntax suggested by Baxter (1994) of a series of command sequences each taking a single mandatory argument, or a single argument parsed by the *keyval* package.

The `document` "environment" looks like an environment and should be parallel to one. The preamble is just a special kind of argument that is serving to instantiate a block element, a bit of type "document".

The `list` environment is an attempt to provide a standard programmer interface to defining a

[2] This behavior seems to be another good candidate for a change to TeX.

block element made up of a single kind of subelement separated by dividing commands. This is the right idea and can profitably be made more general: the inheritance (nesting) of lists is ad hoc, and only one kind of subelement is allowed (that is, `\item`s). In typesetting a play, for example, you would like at least two kinds of subelements: speeches and stage directions. This of course could be implemented using environments, but there are many situations in which using dividing commands rather than enclosing commands is preferable (e.g., ease of use, importing or converting source material).

Intra-package modularity

While developing the `Frankenstein` system, I ran into an interesting problem which led to the idea of *intra-package modularity*. I had a large package which accomplished a number of things that I preferred to use in constellation but others were going to use singly or in arbitrary combinations. The problem was to find a system that allowed me to share my code with the LaTeX community in a way that both provided efficient code and allowed me to maintain the code easily. Using the vocabulary introduced in this paper, the problem was how to get a single disk file to efficiently provide more than one feature.

If I broke up my large package into a number of smaller independent packages, then each package would be efficient when used alone but I would have to maintain several different packages that shared common code and documentation. In some cases, the shared code was so brief that it seemed inefficient and confusing to separate it into its own disk file. There would also be an efficiency problem with TeX's resources such as command names and counters, since each package would have to reserve its own resources. A way was desired to let this group of packages share resources, while preventing conflicts with other packages.

With a tool like `noweb` (see Bzyl, 1995), the *doc* package and `docstrip` program, or an implementation of features as discussed above, a macro written only once in the source can end up in any number of extracted packages. None of the standard defining commands, however, are suitable for defining such a macro. If `\def` is used, all the extracted packages are vulnerable to name conflicts with other packages (not to mention self-conflict during development). But `\newcommand` is also wrong because then the extracted packages could not be used together — the second package to define the command would fail. If `\renewcommand` were used, the first would fail. The command `\providecommand`, which defines a macro only if it is not already defined, was added to LaTeX 2_ε. But using this, one assumes that if the macro was already defined it has an acceptable definition. This level of checking might be sufficient in some circumstances, but a command is desired that *guarantees* a macro will subsequently have a particular definition. To this end I define `\guaranteecommand` which calls `\newcommand` if its first argument is undefined and `\CheckCommand` if it is already defined.

In the `Frankenstein` source, any macro which will end up in more than one package, but which I do not want to install into a separate package required by the others, is defined using `\guaranteecommand`. The definition of `\guaranteecommand` occurs in the *safedefs* package, which all the other packages require (though it is not hard to bootstrap this one command, to get it to guarantee itself).

Disk files and copyrights

The LaTeX developers have put a lot of effort into making it easy to create sources which reliably produce identical document descriptions (identical `dvi` files) at different sites. Such uniformity in the documents derived from a single source is very important in many situations, especially when a longer document is assembled from multiple sources.

One of the ways in which the developers have sought to establish and maintain this standard is by placing conditions on the distribution of the LaTeX system files that require disk file names to serve as unique labels for segments of code. Because commands like `\documentclass` and `\usepackage` cause LaTeX to load files with particular names, there is one and only one (legally-produced) document description that can be generated from a source which invokes standard LaTeX classes and packages.

No one doubts the usefulness of a good standard for deriving documents from sources, but this standard has been achieved at the cost of abstraction. The unique labels which serve to establish the standard should not of necessity coincide with the labels by which a feature or document class is identified in the source. It should never be necessary to alter a source to derive different document descriptions from it. Altering sources is time-consuming. It causes timestamps to be updated and confuses version control systems. It can introduce errors, and it encourages a proliferation of not-quite-identical copies.

On the other hand, it should always remain obvious how to generate the standard document from

a source, so that standard documents are generally available and convenient.

In response to these two concerns, the author has developed the AlaTeX system. The *A* may be understood to stand for *alternate* or *abstract*, or to be the indefinite article, which emphasizes that fact that documents derived from sources with AlaTeX are just one of an indefinite number of possibilities.

AlaTeX is a simple system. It is a slightly-modified clone of the standard LaTeX format, which when used as distributed behaves exactly like LaTeX, except that it displays its own identification banner. (Only in the most perverse situations will the two formats not produce identical output.) Internally, the behavior is not quite the same. The `\documentclass` command in AlaTeX parses its arguments and passes them to a file of code called `metaclas.cfg`. As its name implies, the file is a meta-class which determines how the class specification in the source should be interpreted. And just as important, the file may be altered and distributed with no restrictions. The meta-class distributed with AlaTeX emulates LaTeX's behavior, but this can easily be overridden by changing the file, or putting a different file with the same name earlier in TeX's search path. Code that enables either of two convenient mechanisms of overriding are provided in comments. Using one of these sample mechanisms restores a useful level of abstraction to LaTeX sources, because if full abstract control is available at the first line of the source, it is available for the whole source.

Because it is very difficult to have a working AlaTeX without also having a working LaTeX, no one is likely to find it inconvenient to create, from the same source, either a standard LaTeX `dvi` file (by invoking the LaTeX format) to facilitate seamless exchange of sources, or a not-standard-LaTeX `dvi` file (by invoking the AlaTeX format with an appropriate meta-class).

It should be emphasized that there is no reason at all to *compose* sources while previewing with AlaTeX. Doing so could compromise the portability of the source if a strange meta-class is used. AlaTeX is useful only to change the look of already-existing sources from the standard appearance to a nonstandard appearance. Even in this case, using AlaTeX is not always necessary, since it may be possible to legally modify the appropriate style files, or not too inconvenient to modify the source.

Conclusion

I have argued for the advantages of several kinds of modularity in the LaTeX user and developer environments that do not presently exist to a satisfactory degree. A primary difficulty not resolved is the choice of the domain in which to improve modularity. Are the `Frankenstein`, *newclude*, and AlaTeX solutions adequate? If not, should solutions be effected in LaTeX, in LaTeX3, at the back end of TeX in platform-dependent TeX distributions, at the front end of LaTeX in platform-dependent integrated LaTeX user and developer environments, in extensions to TeX (ε-TeX), or in enhancements to TeX ($\mathcal{NTS}$)? In any case I hope I have won interest and enthusiasm for discussing and working on changes in a certain direction.

The `Frankenstein` system, the *safedefs* and *newclude* packages, and the AlaTeX format should be available on CTAN by the time of publication, if they are not in the meantime adopted in some form into the standard LaTeX distribution.

References

NTS. "Frequently asked questions of NTS-L". 1995. 5th edition, available as CTAN`:info/nts-faq`, maintained by Jörg Knappen `<knappen@vkpmzd.kph.uni-mainz.de>`.

W. E. Baxter. "An object-oriented programming system in TeX". *TUGboat* **15**(3), 331–338, 1994.

W. Bzyl. "Literate `plain` source is available!". *TUGboat* **16**(3), 297–299, 1995.

M. Goossens and Saarela, Janne. "A practical introduction to SGML". *TUGboat* **16**(2), 103–145, 1995a.

M. Goossens and Saarela, Janne. "TeX to HTML and back". *TUGboat* **16**(2), 174–214, 1995b.

A. Ogawa. "Object-oriented programming, descriptive markup, and TeX". *TUGboat* **15**(3), 325–330, 1994.

T. V. Raman. "An Audio View of (LA)TeX Documents — Part II". *TUGboat* **16**(3), 310–314, 1995.