BIJLAGE N

Two faces of text

TEX as a programming tool for advanced document retrieval systems

Joop van Gent

Institute for Language Technology and Artificial Intelligence (ITK)
Tilburg University
PoBox 90153
Tilburg, The Netherlands
gent@kub.nl

Introduction

It is a common misconception to think that a scientific book about earthquakes is a book about earthquakes. If this would be the case the book could be replaced by a database containing the *information about earthquakes* exposed in the book. All scientific books about earthquakes could be replaced by one such database and — still more improbable —all books in the world could be replaced by one huge knowledge base containing all ideas exposed in all books. This is not the case.

A scientific book about earthquakes is also a book about the author, the time of writing, the origins of the ideas and linked ideas that are stored in other books. It's a book about other books about earthquakes and a book about itself. The book presents a lot of different sorts of information, the organisation of which is taken care of by historically developed and commonly accepted methods of referencing (bibliographies, index tables, thesauri) and methods of division (logically splitting up information in parts like nesting paragraphs in sections, sections in chapters, etc.). People find their way in libraries and books by search strategies based on these finegrained organisation systems.

The basis of the main ideas presented in this article is that — *as long as hard copy documents exist parallelly to electronic ones* — document retrieval systems should not only make use of these advanced organisation methods but should also have user interfaces that allow search strategies that people are acquainted with. Document retrieval systems should pair the advantages of fast search performance of computers with 'traditional' bumming around in libraries and 'sniffing' in books.[1] This implies that users should have electronic documents available in a typeset format that looks like a corresponding hard copy as much as possible. It also implies that these systems allow users to query the database in *natural language.*

For the former statement there are several arguments. The most important one is probably the following. In contrast with searching in large databases and quickly overviewing pages, reading large pieces of text is in many ways more comfortable on paper than on a computer screen. Evidently, hard copies are also more easily carried in a handbag. It is therefore to be expected that hard copies (especially large ones) will keep playing a role together with electronic documents for quite a long time. To allow a user to comfortably switch from hard copy to screen and vice versa is therefore a prerequisite for a document retrieval system. This can be achieved by an 'isomorphism' of typography.[2] Another argument is that besides using referencing and division a user often finds his way by *recognition of patterns* without really reading text, by just running a thumb through the pages of a book. This 'method' is evidently supported by having pages in the same typesetting on the screen.

I don't have strong arguments for qualifying interrogation in natural language as an important feature of document retrieval systems. Nevertheless I think document retrieval is one of the 'discourse domains' where a natural language interface is pretty efficient in comparison with alternatives like menu structured interaction or command languages. This is so, because the objects by their nature have an extremely deep hierarchical structure. Therefore menu structuring makes search actions last unnecessary long and command languages — impopular as they are anyway — too complex.

In this article I want to show the important role Knuth's typesetting language TEX — as defined in [6] — can play in the kind of document retrieval systems introduced above (from now on I will call these systems *advan-*

[1] And, of course, with the advantages of new strategies, like those developed in Hypertext(-like) environments (see [3] for a short introduction & survey).

[2] In some sense the physical sectioning of documents in numbered pages can be seen as part of the typesetting. Switching between hard copy and screen is evidently aided by a correspondence in paging and pagenumbering.

ced document retrieval systems). But before doing this I will give a more detailed description of these systems.

Advanced document retrieval systems

Until recently document library systems have been limited by the fact that most documents are only available in (good old) hard copy format. For this reason it was only possible to *automatically* retrieve information about a document by looking at its *description* in a database. These descriptions were (and still are) stated in terms of attributes like author, date, editor, title, keywords etc., like the features one finds on the little paper cards in a library catalog. In other words: only the library *catalog* could be automatized. Most conventional database systems are designed to query 'things' that are not really in the database. For example a database of second hand boats does not really contain second hand boats, it only contains *descriptions* of them.

Since more and more documents become available in (some) electronic format, it becomes interesting to develop library systems with databases containing electronic documents. What makes a database of electronic documents so special is that it does really *contain* electronic documents! Retrieval systems therefore can have the property of being able to *show* the objects in the database and we probably want to use this ability: we want to be able to treat a lot of queries by showing document components in some way on the screen. The system allows us to look *into* a document instead of just looking *at* it.

While containing documents, these systems allow a lot of new kinds of queries. For example queries about the semantic content of a document or about all kinds of properties of the *document components*. [3]

But how do we set up such a system? What exactly will be the objects in the database, what structure do they have and how should they be stored? And secondly: how should information be retrieved from them? To answer these questions we have to look at the 'nature' of documents and the way people find information in them.

Logical structure and search strategy

Documents basically have two types of structures: a logical structure (the organisation of document components) and a typographical structure (the way the document is actually presented on paper), both mirroring on a very abstract level the organisation of the semantic content of the document. In hard copy documents only the typographical structure is 'available', the logical structure has to be abstracted by the reader by pattern recognition.[4] For example chapters can be recognized by their highlighted titles, paragraphs by indentation etc. When we are looking for some information in a hard copy document we often make use of the logical structure of the document reflected in its typography.

It is this natural kind of search strategy that we would like to imitate in our retrieval system. An example will illustrate this. Suppose we have an electronic document about the sales of a certain series of computers and we want to know how many computers were actually sold. No system today will be able to retrieve this information directly from the document by *semantic interpretation* of the text (or figures). But if we know or presume that the information we are looking for is contained in some *table*, we can ask the system to look for a document component that has the property of being a *table*. In this way we can shortcut our search actions.[5]

How could we get the system to find a table? More generally, how can the system find document components? By extracting the logical structure from the typography like humans? Recognition of a logical structure on the basis of typography would request computable methods of *pattern recognition*. In addition to the wellknown problems in this field (algorithm complexity and noise, see for example the problems with current OCR systems) The many different kinds of typography used by different authors and editors make any attempt along these lines hopelessly futile. A better and faster way to achieve our goal is to explicitly impose a logical structure on a document.

In what way? It will be clear that for these purposes storage of documents in *plain text format* will not suffice.[6] We will need some *format* to express attributes of the components of a document. If — for example — we want to know the title of a particular chapter of some document, the system will have to know how to recognize a chapter and a title. This can be done by using a markup language.

The idea of imposing a logical structure on electronic documents by using a markup language is not new of course. SGML, ODA and, in a way, LaTeX have been designed for these purposes, although they were primarily intended for improvement of document interchangability (by separation of logical and typographical structure).

So documents need to be stored in two different formats — a logical structure and some typographical represen-

[3] I use the term 'document components' here to indicate the pieces of text (and figures) a document consists of, like paragraphs, sections, footnotes, etc. Document components are pieces of a document that form somehow a logical unit.

[4] And of course by semantic interpretation...

[5] Sometimes the attributes of document components themselves will be the final target of our query. For example we might want to know the exact title of a particular chapter without being interested in its meaning. Or maybe we want to list the particular components of a document that mention Shakespeare, without being interested in the contents of these components.

[6] With *plain text format* I mean a plain ascii format that contains no codes indicating the logical or typographical attributes of document components.

tation — in order to be able to treat queries about logical components that should result in showing typeset pages. If we drop one of them we are either not capable of representing figures or typeset text, or we have to make inferences about the logical structure by pattern recognition (as typeset formats do not contain codes indicating the logical structure information). Moreover if we we want to see documents in a format that *resembles a printed document as closely as possible*, we will probably want them to be sectioned in *pages*.

The role of TEX

If we choose for a pagewise presentation of typeset documents, the two 'faces' of the document now somehow have to be linked by a function assigning pages to document components. This is so because search actions will always take place on the logical face. Whenever a query is meant to get typeset information on the screen, the system will have to know where to find it. Therefore every component in the logical presentation of a document has to contain a *label* — or *adress* — indicating a position in the corresponding typeset representation. In many cases the typeset version is calculated from the logically formatted version on the basis of *font properties* and other graphical information, like grid, size of figures, style, etc.[7] But, as stated before, the typeset version never contains codes indicating the logical structure of a document. So what kind of computer program should take care of the linking? At this point TEX smiles at us.

Because it is the program that does the typesetting that also has to take care of the right pointers from logical document components to physical positions in the typeset version. If we would try to make this preparation module with an ordinary programming language like C or PASCAL we would have to solve typesetting calculation problems that took Donald Knuth ten years while implementing his TEX.

The TEX language and compilers were originally designed to bring professional typesetting within the reach of (scientific) authors. LATEX was meant to combine typesetting facilities of TEX with an advanced strategy of logically structuring documents, as is stated clearly by Leslie Lamport in [7]. I think most of the popularity of both products is (besides — of course — their quality) due to the elegance of this combination.

In my opinion TEX has another great merit: it can be used to elegantly *prepare* documents for databases of intelligent document retrieval systems as described above. The elegance is due to the parallel processing of semantic representations of documents and corresponding DVI-files. More precisely: The TEX compiler can be used to convert a document to a set of two files, one of which is the standard DVI-format, the other one is a file containing all information about the logical structure of a document and about the logical and typographical properties of the document components. But that's not all. The latter file also contains information about the *physical positions* of all components. With physical positions is meant the positions of the DVI-representations of these components in the DVI-file.

In the next section I will sketch briefly a TEX macro set that creates database objects from LATEX documents. This program has been succesfully implemented and can be extended to other types of document (input) formats like WordPerfect or SGML. The program is now incorporated in a prototype document retrieval system that can be combined elegantly with a natural language interface. I need to say at this point that I owe many suggestions and ideas for implementation of the TEX macro set to Huub Mulders, my ex collegue (and the best TEXnician I know) from the Computer Centre of Tilburg University.

A document preparation module in TEX

The main components of the system we designed are a document *preparation module* and a *document retrieval* module. The former is a *format translator*. It is meant to translate formats used by authors (e.g. WordPerfect, WordStar, LATEX) to formats that are convenient for retrieval. The latter is meant to translate queries of a user to operations on a database.[8]

The preparation module is a TEX macro set consisting of a parser and a generator. In the current implementation only LATEX documents can be parsed. From every document (besides the usual DVI-files and auxiliary files[9]) a set of two files is generated: an LST-file and an LSP-file, which will be examined in depth in the next section. For the moment I only want to make the following remarks about them. The former contains the document representation as a nested list of document components and also all text, the latter a set of statements about the sublists of this list (i.e. about the document components). Both files are in plain ascii format and can be directly interpreted by a Prolog interpreter.

Input and output routines

Only TEX's standard input routines are used. To be able to open extra files and write tokens to them I used TEX's `\openout` and `\write` instructions. To split up the information for the DVI-file and for both logical format files I used a recursive macro like:
`\eat#1 {#1 ...\swallow{#1}..... \eat}`.

[7] E.g. LaTeX , Ventura, Interleaf.

[8] At the moment, only a very limited set of query types can be treated.

[9] For those readers not acquinted with TEX: DVI files are typeset images. Auxiliary files are used for several purposes among which calculating logical references like table of contents. A nice introduction in TEX, though in Dutch is to be found in [4]

This macro reads strings from space character to space character. It takes care for both a direct 'consumption' of `#1` by TEX's standard typesetting mechanism and a 'rechewing' `#1` by a (recursive) macro `\swallow`. The latter reads `#1` token by token. Suppose for example `#1` is the string:

(i) `congres.\footnote{...}.This`.

Then `\swallow` makes:
`c o n g r e s \footnote {...} T h i s`
out of it. `\swallow` passes these 13 tokens to another set of macro's that splits up text and logical information.

It will be clear that the `\begin{document}` macro had to be adapted to initiate the `\eat` macro. To properly exit `eat` and other recursive macro's and to account for a proper nesting of document components in the LST file the `\end{document}` macro was also adapted.[10]

These redefinitions are very trivial:

```
\let\begindoc\document%
\def\document#1{%
   ......(instantiations)....\eat#1}%
\let\enddoc\enddocument%
\def\enddocument{%
   .........(handle nestings).....%
            \enddoc}%
```

For the interpretation of tokens routines were needed to check whether two tokens are the same. This is handled by the folowing macro:

```
\newif\ifsame%
\long\def\testsametoken#1#2{%
      \edef\partwo{\string#2}%
      \edef\parone{\string#1}%
      \ifx\parone\partwo\sametrue%
      \else\samefalse%
      \fi}%
```

Because of the token by token reading I needed a macro to append characters to a string. This could be done by the following macro:

```
\newtoks\ta\newtoks\tb%
\long\def\rightappend#1\to#2{%
         \ta={#1}%
         \tb=\expandafter{#2}%
         \edef#2{\the\tb\the\ta}}%
```

Another problem that had to be handled has to do with the linking of the logical and typographical image. Pagenumbers are generated at the beginning of a logical document component. As TEXnicians know TEX does a lot of recalculating at the end of document components. For this reason sometimes a wrong pagenumber was generated. This problem was solved by `\relax`-ing the pagenumber calculation:

```
\def\tolspfile#1{{\let\thepage\relax%
    \xdef\temppar{%
      \write11{\string#1}}}%
    \temppar}%
```

The rest of the macro set comes down to a partly recursive handling of LATEX's component delimiters.

The output format

It is often suggestes that documents should be stored in SGML format ,or a similar markup language type. From a document retrieval point of view this is not very attractive because it implies parsing of SGML documents while treating a query. We cannot avoid 'precompiling' SGML-documents to some other format that allows faster search strategies.[11]

I chose to take a format that has at least two attractive properties. First it is a format that can be *interpreted* by any proper Prolog interpreter and therefore it can also be compiled. Compiling this format results in a low level pointer structure that allows fast searching. Second it is a format that facilitates the interpretation of (natural language) queries. The latter point needs some further explication.

Interpretation of queries

As in any other query language the evaluation of a natural language query comes down to checking relations between objects in a database. In the case of documents and their components these relations are often very particular types of relations that have to do with the *logical positions* of document components, for example the *precedence* relation ($\alpha\ precedes\ \beta$) and the *sublist* relation ($\alpha\ is\_a\_sublist\_of\ \beta$). If we say — for example — that some chapter contains some footnote we say in fact that a list of a certain type is a sublist of another type.[12] What is so special about this? Well, nested lists (like for example *sets*) are in fact *algabraic* entities with a lot of welldefined properties. Basic operations on lists can be implemented in a very trivial way. The idea is then that we can facilitate the evaluation of queries when the database is a nested list and when we define the semantics of our query language in such a way that it can make use

[10] None of LaTeX macro's are redefined in the system files. All redefinitions take place in a distinct top level macro set in a trivial way by temporarily renaming the LaTeX instructions.

[11] A way to avoid parsing SGML bulk files is to make use of SGML's file pointer mechanism. This comes down to putting markups in files that are interpreted as pointers to other files (for details see [1]). In this way sets of files (document components) can be structured in any desired way. In this way we loose an advantage of SGML, namely its direct interchangability with electronic publishing environments.

[12] The term 'type' should not be taken literally here. It is to be discussed whether being a chapter or a footnote is to be taken as a type or as a predicate.

of the basic operations on nested lists when necessary.[13] One might argue that a lot of queries don't request mathematical operations on lists! I think this is indeed the case with queries about the *contents* of a document. A lot of queries however contain subexpressions (for example definite descriptions) that refer to document *components*. Part of the evaluation of the query therefore comes down to finding the correct referents for these subexpressions. Finding a referent on its turn implies an *extraction* of one or more elements from a list, which is in fact a mathematical operation.

EL/DR, a document representation language

EL/DR is a shorthand for *Ensemble Language for Document Representation.* The term 'Ensemble Language' refers to a type of semantic representation languages proposed by Harry Bunt [2]. EL/DR is defined within this framework. The language is used both for document representation and for the semantics of natural language queries. I will only introduce the sublanguage of EL/DR that is used for document representation here. For details about EL/DR the reader be referred to [5].

The language called EL/DR is based on the idea that documents are in fact *tree structures* or *nested lists* of logical components that are themselves also nested lists. For example, a chapter consists of a list of sections. A section consists of a list of subsections or paragraphs and so on. I call those components *DLS* (Document Logical Structure). *Words*, *figures* and (for the moment) *formula's* and *tables* are considered to be atomic. DLS's are syntactically represented in a trivial way. Consider the next (very small) document:

```
Yesterday I bought:
    - two appels
    - a beer
    - milk
    - some bread
```

Suppose we say that this document consists of a paragraph and an enumeration, the latter consisting of four items. This document can now be logically presented as a nested list as follows:[14]

```
[[Yesterday,I,bought],[[two,appels],
      [a,beer],[milk],[some,bread]]]
```

To avoid ambiguities — every word can occur more than once in a document — occurrences of words have to be labelled in a way different from the syntactic representations of the words themselves, for example:

```
[[w1,w2,w3],[[w4,w5],
         [w6,w7],[w8],[w9,w10]]]
```

In fact this is done in EL/DR, but from an implementation point of view it is more convenient to have labels for DLS's (nodes in a tree) too. Therefore, all nodes in a tree structure (atoms included) are implicitly numbered starting from 0 (the whole document) topdown, leftright depth-first, to n, the rightmost atomic element of the tree.

All properties of these DLS's can now very simply be stated in terms of predicate-argument constructions (or for those acquinted with Prolog, in terms of *horn clauses*). For example:

```
paragraph([w1,w2,w3]).
enumeration([[w4,w5],[w6,w7],
             [w8],[w9,w10]]).
item([w4,w5]).
item([w6,w7]).
item([w8]).
item([w9,w10]).
word(w4).
word(w5).
...
```

Or when using the implicitly imposed labels:

```
paragraph(1).
enumeration(5).
item(6).
item(9).
item(12).
item(14).
word(7).
word(8).
...
```

Chapter titles and crossreferences are two-place predicates, for example:

```
title(100,101).
crossref(221,457).
```

The first clause in this example states that the DLS labeled '101' is the title of the DLS labeled '100'. The second clause states that the DLS labeled '221' refers to the DLS labeled '457'.[15]

Finally, the linking of logical components to pagenumbers is done in a similar way, for example:
`pagenumber(221,23).`

The first argument in this example is a label for a logical component, the second argument refers to a pagenumber, which is a physical position in the typeset image.

[13] If we think of representing documents as nested lists we can also make use of their properties in other ways. For example we can use them to *reason* about documents.

[14] For the sake of clarity interpunction is left out.

[15] Note that crossreferences are not seen as attaching *physical positions* in a text to document components, but rather as combining two (or more) logical document components.

Conclusions and final remarks

In this paper I have tried to show that TEX can be used to prepare suitable database objects for document retrieval systems and that alternatives must have particular qualities that are scarcely found in other programming languages. Current software supporting TEX has its limitations though. One of these is that not all graphic information can be represented in DVI-files. This is particularly the case with bitmaps. It seems therefore worthwile investigating alternatives like PostScript.

I have made a lot of presuppositions that are or should be subject of discussion, specifically:

1. advanced document retrieval systems should allow search strategies users have historically got acquinted with (but also — of course — new strategies),
2. hard copies will exist for a long time *together with* their electronic 'sisters'; for this reason a correspondence between the graphic representations of both sisters is desirable,
3. queries that make use of a document's logical structure play an important role in document retrieval.

Still I hope to have made clear that the preparation module for a document retrieval system should output documents as nested lists to make its sublists accessable by mathematical operations in the retrieval module. I also hope to have stimulated the TEX community to think about TEX's hidden qualities in the field of document retrieval.

References

[1] Martin Bryan: *SGML An author's guide to the Standard Generalised Markup Language*, Addison-Wesley Publishing Company, 1988.

[2] Harry Bunt: *Mass Terms and Model-theoretic Semantics*, Cambridge: Cambridge University Press, 1985.

[3] Jeff Conklin: *Hypertext; An introduction and survey*, in: Computer, september 1987.

[4] Victor Eijkhout & Nico Poppelier: *Wat is TEX?*, in: NTG Report 4, november 1989.

[5] Joop van Gent: *A formal language for describing document structures*, forthcoming, 1990.

[6] Donald E. Knuth: *Computers & Typesetting*, Addison-Wesley Publishing Company, 1986.

[7] Leslie Lamport: *LATEX user's guide & reference manual*, Addison-Wesley Publishing Company, 1986.