A Structured Document Preparation System — *AutoLayouter* Version 2.0 — An Enhancement for Handling Multiple Document Types

Takashi Kakiuchi, Yuki Kusumi, Yoshiyuki Miyabe, and Kazu Tsuga
Information and Communications Research Center
Matsushita Electric Industrial Co., Ltd
1006 Kadoma, Kadoma-shi, Osaka 571 Japan
+81 6 906 4873; FAX: +81 6 906 8148
Internet: `kakiuchi@isl.mei.co.jp`

Abstract

AutoLayouter is a structured document preparation system used to increase efficiency in creating and reusing designed documents in offices. *AutoLayouter* consists of an easy-to-use structured editor and a Japanese LaTeX-based formatter. With a structured editor, the user need not be concerned with page layout, and can concentrate on creating the contents of the document. Because these documents are structured logically, they can be easily reused or processed further by other systems.

At the 1990 TUG meeting, we presented *AutoLayouter* version 1.0. Since then we have been improving the system to handle more complicated document structures, such as are defined in SGML. In this paper, we describe 1) new document structures, and 2) ALTeX, which directly formats structured documents.

Introduction

Recent research projects on document processing have been directed at structured document representations, such as SGML. The basic idea of a structured document is to separate a document into structure and content; its contents are then extracted in terms of its structure. In an SGML document, the structure is defined explicitly as a DTD (Document Type Definition), so that documents created with the same DTD are interchangable. Such a structure can also be used by a document processing system to retrieve the required information: for instance, the title, author, and date of technical reports can be retrieved through their structure and merged into a summary table.

The structured document representation, especially the logically structured one, is essential to making the best use of electronic documents. We can store documents in electronic format, and load and print them on paper, using conventional word processor or desktop publishing systems. These documents cannot be processed by other systems, however, unless the logical meanings of their contents are preserved, because there is no other way to identify the contents. Because of its abstract, declarative language, LaTeX is often referred to as an example of a text formatter for logically structured documents. LaTeX is used as a document preparation tool by computer software engineers because they can use any editor and can concentrate on a document's content and structure without paying any attention to its physical appearance.

In Japan, the advance of word processing technology has meant that business documents are prepared and stored electronically, but they must also be kept in printed form. The format of most Japanese business documents separates items with rule lines. This standardizes the items to be written and determines the text area available for each item. Japanese word processors possess some characteristics for editing these forms: they draw ruled lines and insert text in the area surrounded by the rules. However, this augmentation of rule-line functions has made it too complex to manage document files and to reuse document contents. As a result, a document must still be managed in the printed form, even though it is stored in an electronic format.

To solve these problems, we have developed a structured document preparation system, *AutoLayouter*, whose objective is to increase efficiency in creating and reusing preformed documents. *AutoLayouter* consists of a structured editor for creating SGML-like documents, and a Japanese LaTeX-based formatter called ALTEX. In the subsequent sections of this paper, we mainly describe the document structures of *AutoLayouter* and implementation issues of ALTEX formatter.

Document Structure

Model for document structures. The *AutoLayouter* document is represented as a tree structure (like an SGML document). Each node of the document tree, except the leaves, has a unique label associated with it. Each leaf of the document tree contains a text segment, which is represented as a sequence of characters. Any node may have an arbitrary number of attributes, represented as name–value pairs.

A major difference between the document structure of *AutoLayouter* and SGML is that the *AutoLayouter* document has two structure layers, namely the *logical structure* and the *generic structure*. The logical structure presents the logical meaning of the subsidiary structures, such as a sender's address in a letter, which is specific to the document type. Meanwhile, the generic structure presents such document elements as itemization, enumeration, and centering; these are common to all document types. The generic structure is already predefined in the system. When defining a document structure, we need only specify the logical structure.

The whole document structure is organized as follows: the root node of the document belongs to the logical structure, and its descendents can belong to either the logical structure or the generic structure, according to the document definition. A node in the generic structure cannot be a parent of any nodes in the logical structure; furthermore, siblings belong to the same structure. In the rest of the paper, we shall call nodes in the logical structure the *logical element*, and nodes in the generic structure the *generic element*. Each leaf of the document is a special generic element that has only a text segment with no children.

A model for structured documents should be well designed so as to make it easy to define document structures and maintain consistencies between them, and also to make its editor easy to use. In SGML, the whole document structure must be defined explicitly, using the fully expressive description language. This means that to use the contents of one document in another document, the structure definitions of both must be strictly consistent with each other; such consistency requires as much effort as does designing database schemes. Furthermore,

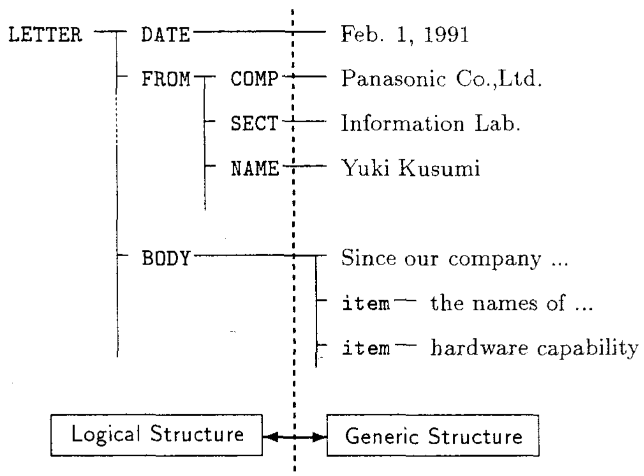


Figure 1: Document structure in *AutoLayouter*

the user interface of a structured editor tends to be awkward because of the flexibility required to handle all document structures as generated from their definition. This is analogous to the trade-off between functionality and ease of use involved with most systems, namely, easy-to-use tools can be achieved at the expense of their restricted flexibility.

In *AutoLayouter*, the generic structure is predefined in the system and only the logical structure needs to be defined; thus, only the logical part of document structures should be designed to be consistent. Moreover, we can build in the easy-to-use, dedicated user interface for editing the generic structure; this contributes to efficiency in preparing documents. A user often manipulates a document's generic structure rather than its logical structure, because most of the logical structure can be generated automatically by the system and need not be modified so frequently, whereas the generic structure contains the text segments to be typed and the layout directives that have been left to the user.

Example 1: *In Fig. 1, a whole document structure is divided into two structures. The document definition specifies only the logical structure, shown on the left side.*

By using these two-layered structures, the design of a new document type is accomplished by defining a logical part of its structure and specifying how to present each element on paper (layout definition).

Structure definition. The structure definition of a document type is a generic specification of its logical structures. This is expressed in a grammar format that specifies the logical elements and the order

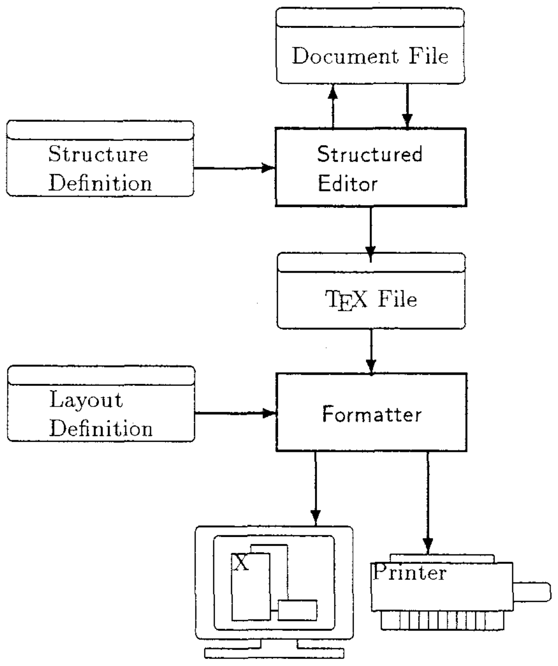


Figure 2: System diagram of *AutoLayouter*

in which they will be generated. Each rule consists of a left-hand side, which corresponds to a node, and a right-hand side, which is a restricted regular expression that specifies occurrences of its children.

System Structure of *AutoLayouter*

System overview. As shown in Fig. 2, *AutoLayouter* consists of two subsystems: a structured editor and a formatter.

The structured editor interactively performs the following tasks:

- interprets a structure definition;
- edits documents, showing the structure elements to be inserted and checking illegal structure modifications;
- loads and saves structured document files; and
- converts documents into TEX files.

Meanwhile, the formatter completes the following tasks:

- typesets the document in accordance with the layout definition (style file) provided; and
- converts formatted documents (`dvi` file) to a specified device such as a bitmap display or a PostScript printer.

In the rest of this section, we describe various file formats used by subsystems, to clarify their roles.

File formats. The data files used in the *AutoLayouter* are the following:

- a structure definition file (for input);
- a structured document file (for input and output);
- a TEX file (for output).
- a layout definition file (for input); and

A structure definition file. In order to define document structures (see the Model for Document Structures subsection on previous page), we use the following three syntaxes in the structure definition file.

1. A node having children of logical elements is defined using the following syntax:

   `<!node` *node_name*,
   regular_expression`>`

   This implies that if a node is a logical element, then its siblings are also logical elements.
2. A node having children of generic elements is defined using the following syntax:

   `<!leaf` *node_name*, *type*`>`

   A *type* field, which can be `general`, `string`, or `integer`, and so on, specifies a selection of the subsidiary structures that are allowed to appear; `general` allows any kind of generic elements, including any nested sub-tree of a generic structure; `string` allows only a string in a text segment; and `integer` allows only an integer in a text segment.
3. Attributes associated with a node are defined using the following syntax:

   `<!attribute` *node_name*,
   {*attr_type*
   attr_name = *initial_value*}*>

   An attribute, which may be used for any purpose, is typically used to define layout parameters, such as paper size or column layout.

In addition to the syntax above, we provide a syntax just for the structured editor; this is used to define help information for each logical element, such as a label string shown in the editor.

Example 2: *The following is the structure definition of the document shown in Fig. 1.*

```
<!rootnode LETTER, DATE.FROM.BODY...>
<!leaf DATE, date>
<!node FROM, COMP.SECT.NAME...>
<!leaf COMP, string>
...
<!leaf BODY, general>
```

The structured editor reads the structure definition file in two situations: when selecting a document style to create a new document, or when starting to edit an already existing document.

A structured document file. We directly represent a tree structure of an *AutoLayouter* document as a block structure of the document file. A node n, whose children are $m_1, m_2, \ldots, m_k$, is expressed in the document file as follows:

```
<n attribute_list>
    <m_1 attribute_list>
    ...
    </m_1>
...
    <m_k attribute_list>
    ...
    </m_k>
</n>
```

A TEX file. The structured editor outputs a TEX file to be input by the formatter. The TEX file represents the tree structure of the *AutoLayouter* document directly, converting a node <n>,...,</n> in the document file into a TEX command `\beginnode{n}`,...,`\endnode{n}`, and replacing all special TEX characters with TEX commands that generate the characters literally.

```
\beginnode{n}[attribute_list]{
  \beginnode{m_1}[attribute_list]{
  ...
  }\endnode{m_1} ...
  \beginnode{m_k}[attribute_list]{
  ...
  }\endnode{m_k}
}\endnode{n}
```

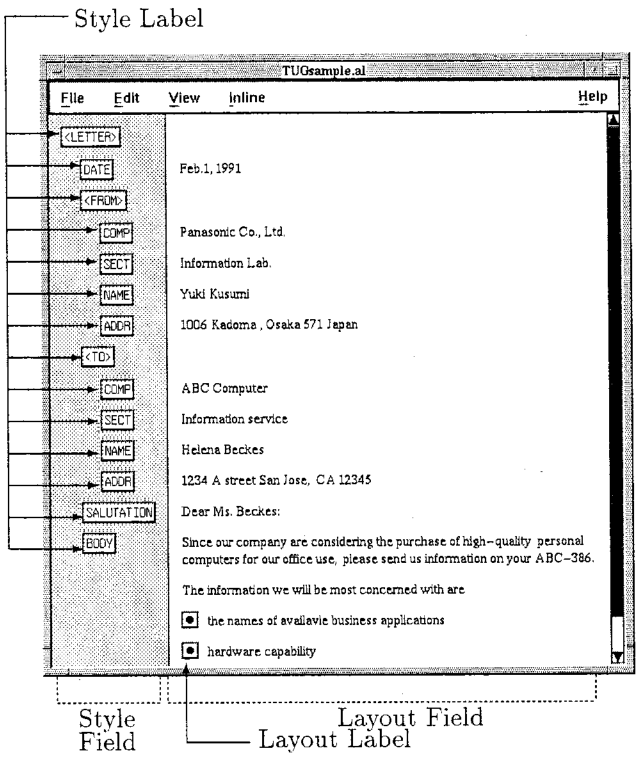
The name of the root node that appears at the top of the TEX file identifies the style file.

A layout definition file. The layout definition file is a TEX style file. This will be discussed later.

Editing the Structured Document

As shown in Fig. 3, the editing field of the structured editor is divided into two areas, a style field and a layout field, that represent the logical structure and generic structure, respectively. Usually we use different labels in different structures, such as text labels in *style field* and graphical labels in *layout field*. This makes it easy for users to see the whole document structure. In each field, we use indentations to show substructures.

When creating a new document, one selects the document type, such as `letter` or `report`. The editor reads the structure definition file of the specified document type and generates a mandatory and minimum structure according to the definition rules.[1] Since the mandatory structure has already been generated, one completes the document by simply typing text into each text segment.



Feb.1, 1991

Panasonic Co., Ltd.

Information Lab.

Yuki Kusumi

1006 Kadoma , Osaka 571 Japan

ABC Computer

Information service

Helena Beckes

1234 A street San Jose, CA 12345

Dear Ms. Beckes:

Since our company are considering the purchase of high-quality personal computers for our office use, please send us information on your ABC-386.

The information we will be most concerned with are

● the names of avallavie business applications

● hardware capability

Figure 3: Snap shot of editor screen

One may insert a logical element, such as a report date field, as needed, whenever it has been defined as optional or is repeated in a regular expression. When the insert command is selected for the layout field, the editor displays candidates for the logical elements that can be inserted at the specified position. One only needs to select a candidate to insert it. Since only valid candidates are shown, an illegal structure can never be generated. When deleting a node, the editor checks whether this violates a rule; if it does, the editor displays an error message and ignores the user's operation.

In the layout field, one can insert any generic element at any position, as long as the type of its ancestral logical element is declared as `general` in the definition. When the insert command on the layout field is selected, the editor shows a label list containing all generic elements.

The editor also has additional features listed below:

Motif as Graphical User Interface. Motif provides a consistent look and feel in different applications.

[1] Each leaf of the logical element has a generic element for a text segment.

Japanese Kana-to-Kanji conversion. We developed Japanese input as a front-end processor. Communication between this and the text editor realizes in-line conversion of Japanese.
Operations with keys. Most commands can be operated with either a mouse or a keyboard. This satisfies a wide range of users, from novice to expert.
Browsing functions. Moving around labels that have a keyboard focus switches the contents of the panels that display the attributes and the help messages.

Formatting with ALTEX

AutoLayouter formats its structured documents using an original typesetter called "ALTEX", which has the following features:

- handles a tree-structured document directly; and
- provides ready-to-use macros to support layout abstraction.

ALTEX is implemented in LATEX.[2] Therefore, not only can LATEX users include their LATEX documents within an ALTEX document, but LATEX experts can easily describe a layout definition by using LATEX commands.

We will describe our ALTEX in detail with respect to these features in this section.

Formatting tree-structured documents. First, we will explain the mechanism for mapping a structure to its layout. As we mentioned in the section System Structure of *AutoLayouter*, a structure element in a document is represented in the form

```
\beginnode{..},...,\endnode{..}
```

in an ALTEX file produced by the structured editor. ALTEX expands the two control sequences `\beginnode` and `\endnode` in the same way that it is used in the LATEX environment, namely `\begin{..}`,...,`\end{..}`. For instance, a structure

\beginnode{*foo*}[*attribute list*]{
...
}\endnode{*foo*}

is expanded to the following:

\begingroup\nodefoo{*attribute list*}{
...
}\endnodefoo\endgroup

This expansion indicates that the layout for a structure *foo* is based on the definition of two control sequences, `\nodefoo` and `\endnodefoo`.

In this mechanism, it should be noted that the text segment of a structure is enclosed with the grouping symbols { and }. The braces allow the text segment to be processed as an argument to a TEX macro in some cases, or to be laid out as text as soon as it appears in other cases. To be more specific, in the case where the text segment is to be placed directly into the main vertical list, one can define the control sequence `\nodefoo` as

```
\def\nodefoo#1{...}
```

In this case, `\nodefoo` works as a pre-processor before the text segment is laid out on the page. If, on the other hand, the text segment needs processing, or it should be saved once and laid out later, one defines `\nodefoo` as

```
\def\nodefoo#1#2{...}
```

This form of definition enables us to describe any operations on the text segment (i.e., argument #2) in the replacement text of the macro definition. However, note that the former form is recommended wherever possible, because the latter form consumes more TEX memory.

Example 3: *Let us consider a definition for a declaration of the author of an article, similar to the* `\author` *command in LATEX. In the LATEX* `article.sty` *file, the* `\author` *command is defined as:*

```
\def\author#1{\def\@author{#1}}
```

i.e., the `\author` *command saves its argument into a macro* `\@author`. *In order to implement the same function as the* `\author` *command in ALTEX, we define a* `\def\nodeAUTHOR` *macro for a logical structure AUTHOR as:*

```
\def\nodeAUTHOR#1#2{\gdef\@AUTHOR{#2}}
```

The mechanism mentioned above is not applied to the outermost structure, namely \beginnode{*root*} and \endnode{*root*}, which represents the root node of the document, because it requires extra tasks. The \beginnode{*root*} command should load a layout definition file and set up miscellaneous parameters, and the \endnode{*root*} command should flush out the main vertical list and process cross-references.

Incidentally, ALTEX expands attribute lists in a uniform fashion. For instance, if an attribute list of the structure *foo* appears as:

```
\beginnode{foo}[attra=vala;
                attrb=valb]{
```

then each "*attribute*=*value*" pair is expanded into a command \foo@*attribute*{*value*}, i.e.:

```
\foo@attra{vala}
\foo@attrb{valb}
```

[2] Japanese LATEX (ASCII version), to be exact.

To process the expanded attribute list, we must prepare control sequences that have one argument "`\foo@`*attribute*" for each attribute associated with a node *foo* in a layout definition file.

Layout model and layout definition. When considering a practical usage for a document preparation system that is based on a structured document, providing a toolkit to facilitate layout definitions is indispensable. When using a document preparation system with WYSIWYG and direct manipulation features, we can perform any page layouts with some cumbersome efforts. Obviously, *AutoLayouter*'s automatic layout feature does not work without a layout definition. This becomes the most critical bottleneck in practical use.

To keep the toolkit from being complex and confusing, it should be based on a well-designed and simple layout model. In ALTEX, we provided two layout models, a *paragraph layout model* and a *form layout model*. Each tool is an abstraction of a layout based on these models.

In the rest of this section, we present these two layout models, as well as the way to use the toolkit to map the logical structure element to the physical layout.

Layout model. The sequence of words in a text segment is broken into lines with the paragraph layout model. The result of paragraph layout is a *box* that might either be put into the main vertical list directly, or aligned vertically or horizontally together with other boxes before being put into the main vertical list. In the latter case, the alignment is performed on the form layout model. Now, let us see each model in detail.

Paragraph layout model. This model is provided for the sake of putting the contents of a structure element into the heap of lines. Each text segment in the leaf elements contains logical paragraphs. These are put into the physical layout of the paragraphs, whose shapes vary according to the parameters shown in Fig. 4. We utilized TEX's line-breaking mechanism in implementing this model; itemizing, centering, and flushing, for example, can be represented with this model.

Roughly speaking, this model corresponds to LATEX's `list` environment with only one `\item`. However, our model has such extended features that we can set labels on top of the second and subsequent paragraphs, as well as the first one, and we can set the arbitrary shape of any hanging indent, and so on.

Furthermore, when both a node and its children are laid out with this model, the margin of the parent node is inherited by the children. This is why the layout of nested items is guaranteed, as is expected.

Incidentally, we furnished ALTEX with a command to define a structure as this model. Assume structure *foo* is defined as a node laid out with this model, then the result of `\beginnode{`*foo*`}` ,..., `\endnode{`*foo*`}` is put into a `\vbox`, such as the main vertical list, after the text segment in the structure has been broken up into lines.

Form layout model. This model is provided to make forms in which boxes are aligned with each other. In this model, the alignment of boxes is modeled as the tree structure shown in Fig. 5(a). Each node of the tree aligns its children either horizontally or vertically. As our approach is based on TEX, this model is implemented as nested `\vbox`es and `\hbox`es.

ALTEX also provides commands for making various boxes, as well commands to align the boxes. For example,

- a command to make a box with specified width and height: the layout of the inside of the box can be also specified, along with centering, flushing, paragraph shape, and so on. (See Fig. 5(b).)
- the commands to make a box for the title and to specify the contents for it: the same layout commands have the same function as above. (See Fig. 5(c).)

In plain TEX, it is not easy to make a box with a specified width and height, which is why we decided to provide these commands at the system level.

In addition, we created some commands, used instead of `\vbox` and `\hbox`, to improve the readability of the layout definition. Using *AutoLayouter*, one can describe a vertical box with

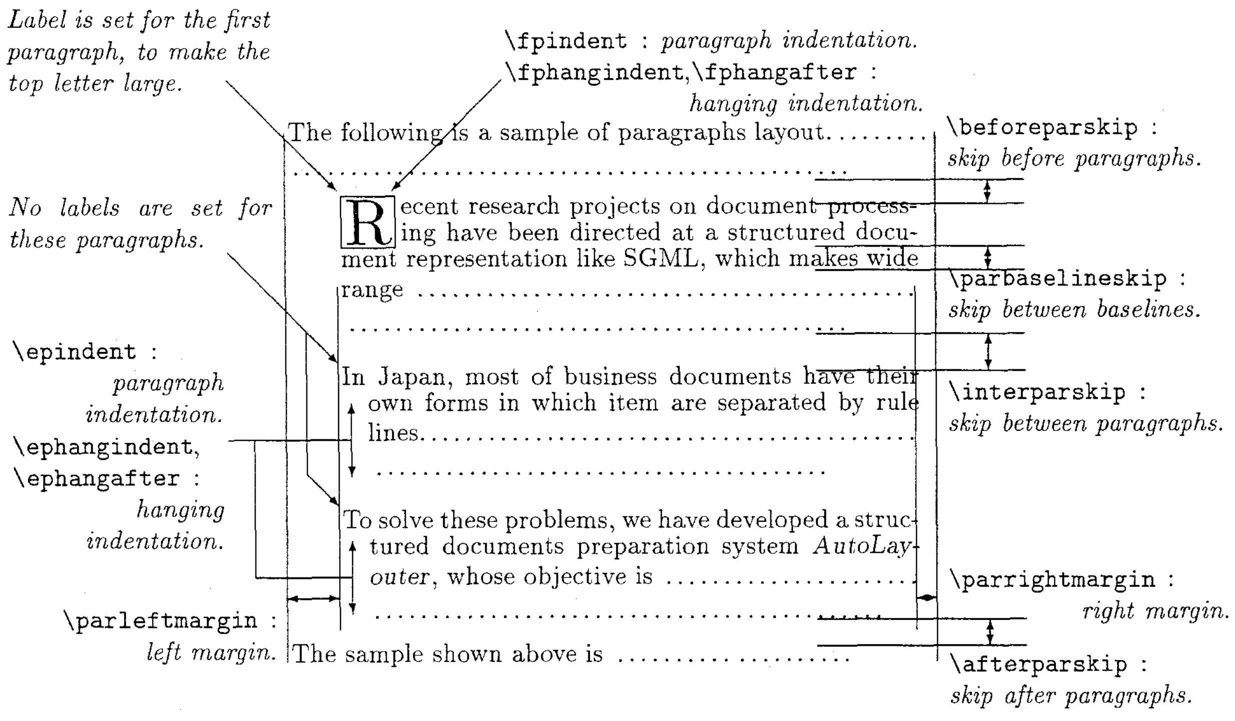
```
\BeginV
   ....
\EndV
```

instead of with

```
\vbox{\offinterlineskip
  ...
}
```

Two ways to map a structure to its layout. There are two ways of mapping a logical structure element to its physical layout, namely *direct mapping* and *indirect mapping*, depending on how the occurrence of the element corresponds to its layout.

Direct mapping. In the case of the `letter` or `article` style, most of the logical elements are laid out in the same order as they appear in a document.



Label is set for the first paragraph, to make the top letter large.
\fpindent : paragraph indentation.
\fphangindent,\fphangafter : hanging indentation.
The following is a sample of paragraphs layout.........
\beforeparskip : skip before paragraphs.
No labels are set for these paragraphs.
Recent research projects on document processing have been directed at a structured document representation like SGML, which makes wide range
\parbaselineskip : skip between baselines.
\epindent : paragraph indentation.
In Japan, most of business documents have their own forms in which item are separated by rule lines
\interparskip : skip between paragraphs.
\ephangindent, \ephangafter : hanging indentation.
To solve these problems, we have developed a structured documents preparation system AutoLayouter, whose objective is
\parrightmargin : right margin.
\parleftmargin : left margin.
The sample shown above is
\afterparskip : skip after paragraphs.

Figure 4: Paragraph Layout Model

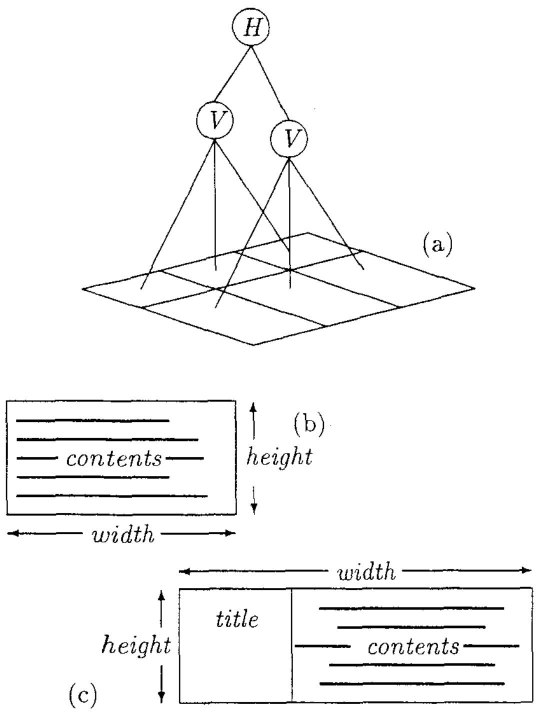


Figure 5: Form Layout Model

For these elements, we can put their contents into the main vertical list as they appear, using *paragraph layout*. In this case, assuming the name of the node is *foo*, mapping is performed simply by declaring the command \nodefoo and \endnodefoo for paragraph layout. We call this *direct mapping*.

Example 4: *Let us consider the case where one wants to define the layout of the structure element to provide an* `agreement` *style:*

(1) A member should notify the consortium as soon as possible after modifying *AutoLayouter*.

Assume that the name of this structure element is "PROVISION". All that must be done is to specify the parameters to the paragraph layout model for `PROVISION`,

```
\par@nodedef{PROVISION}%
  {\fpindent\z@%
   \afterparskip=.7ex plus .2ex%
   \interparskip=.3ex plus .02ex}%
  {increment=1;ctrlayout=hang;%
   before=\bf (;after=)}% counter
  {}% use default fonts
  {showctr}% at the top of 1st pararaph
  {default}% at the top of the others
```

where \par@nodedef *is the command to define a structure element using the paragraph layout model. This definition* directly *maps the logical element "PROVISION" to its layout.*

Most generic elements, such as itemizing, enumerating, and flushing, are also directly mapped with the `\par@nodedef` command.

Indirect mapping. In the case where the contents of each structure element are laid out irrespective of the order of their appearance, we can save the contents once and lay them out later. We call this type of mapping *indirect mapping*, and it applies to most forms, the *title* structure of `article`, and the *heading* of `letter`, for instance.

Now, let us consider this mapping with respect to macro definitions. Assume that an element *foo* is mapped indirectly, then the command `\nodefoo` should be defined with the form (see subsection Formatting Tree Structured Documents):

```
\def\nodefoo#1#2{...}
```

In the replacement text of this definition, argument `#2`, which contains a text segment, would be saved instead of being put out into the main vertical list. Only later would it be put into the main vertical list.

Example 5: *Let us consider Example 3 again. ALTEX's toolkit provides the command that directs an element to save the contents of a text segment using a macro definition. With this command, the node AUTHOR can be defined as:*

```
\def@nodedef{AUTHOR}{10}{}
```

where the first argument is the name of the element, the second argument specifies how many occurrences of the element can be allowed, and the last argument holds the initial value for the element.

For each occurrence of the element *AUTHOR*, `\beginnode{`*AUTHOR*`}`,..., `\endnode{`*AUTHOR*`}` is expanded. In this expansion, the text segment is defined as the macros `\@AUTHORi`, `\@AUTHORii`,`\@AUTHORiii`..., and so on. The roman numerals `i`, `ii`, and `iii` in the name of the control sequences stand for the order of occurrence of the element.

Now, assume that *HEAD* is the parent node of *AUTHOR*, then one should define `\endnodeHEAD` as

```
\def\endnodeHEAD{...
                  \@AUTHORi
                  ...}
```

in order to lay out the contents of the *AUTHOR* element.

Conclusion

In this paper, we have described *AutoLayouter*, a structured documents preparation system that uses TEX and LATEX commands for structuring and formatting documents. By dividing a document structure into two layers, each of which contains logical elements and generic elements, respectively, we can easily define the structure and layout of documents. Furthermore, we built-in an easy-to-use, dedicated user interface for editing the generic structure; this contributes to efficiency in document preparation.

In a future version, we plan to develop tools for defining the document's structure and layout, and also document management facilities.

Acknowledgment

The authors would like to thank T. Ohno and R. Kurasawa, who developed Japanese TEX.

Bibliography

Adobe Systems Incorporated. *PostScript Language Reference Manual, Second Edition.* Reading, Mass.: Addison-Wesley, 1990.

ISO 8879, "Information Processing — Text And Office Systems — Standard Markup Language (SGML)." Geneva ISO, 1987.

Knuth, Donald E. *The TEXbook*. Reading, Mass.: Addison-Wesley, 1984.

Kurasawa, Ryoichi. "Japanese TEX at ASCII Corporation" (*in Japanese*). Proceedings of TEX Users Group Japan, TX-97-5, September 1987.

Kusumi, Yuki, Takashi Kakiuchi, Yoshiyuki Miyabe, and Kazu Tsuga. "Structured Document Preparation System *AutoLayouter* — Design and Implementation," IEICE Technical Report OS90-23, 1990.

Lamport, Leslie. *LATEX: A Document Preparation System*. Reading, Mass.: Addison-Wesley, 1983.

Miyabe, Yoshiyuki, Hiroshi Ohta, and Kazu Tsuga. "Structured Document Preparation System: *AutoLayouter*." *TUGboat*, **11**#3, 353–358, September, 1990.