stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)   IW 191/82   FEBRUARI

H. NOOT

STRUCTURED TEXT FORMATTING

Preprint

kruislaan 413   1098 SJ   amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

1982 CR Categories: I.7.2

STRUCTURED TEXT FORMATTING†

by

H.Noot

ABSTRACT

A Document Formatting system is presented which provides a highly structured user interface. In particular, the system incorporates an elaborate error detection- and recovery mechanism, whereas format directives are implemented by the user in a high-level language, instead of in the rather low-level macro languages generally used for this purpose. Furthermore, a systematic way of dealing with problems, like the avoidance of 'widows' etc. is discussed, as well as a concatenation mechanism for 2-dimensional strings. Finally, all major design decisions taken for this system, are extensively motivated.

KEY WORDS & PHRASES: text formatting, text markup, string concatenation, constraints, error detection

†This report will be submitted for publication elsewhere.

AN OVERVIEW

Introduction

In the past decade, a host of computer-based text formatting systems have emerged. They fall into three broad categories:

1. Systems for use in an office environment. These systems can only be used for the processing of simple plain text.
2. (Semi) automated systems for use in the publishing industry. These systems, intended for the production of 'printers quality' documents, require attention from specialized professionals, and vast quantities of (dedicated) equipment.
3. Layout programs like NROFF [2] and TEX. [3] They are often intended for the processing of both simple and complicated (e.g. mathematical) texts. These programs, which can run on general purpose computers, are almost exclusively used in technical or scientific milieus.

This paper deals with systems of the last kind only. A typical organization of such a layout system is the following: The basic system supports low-level format directives. On top of this, there is a possibility to define macros, implementing higher level instructions in terms of those primitives. For instance, the systems TEX, [3] NROFF, [2] PUB [11] and TEXTURE [7] all have this structure. In such a system, we can distinguish two types of users:

1. Those which mark up a text with high-level directives and then have that text formatted.
2. Those that specify and implement high-level directives.

We will call those users end user and layout designer, respectively.

In our view, to day's layout programs are still too difficult to use, because end user and layout designer must face the following problems:

1. End user problems:
    1. The lack of conciseness of the high-level directives, especially for complicated (e.g. mathematical) texts.
    2. The absence of error detection and recovery.
2. Layout designer problems:
    1. The low level of the languages which are provided for the implementation of high-level directives.
    2. The (sometimes extreme) difficulty of describing complicated page formats, i.e. of defining a page as a composition of arbitrarily placed blocks of various kinds of text.

It should be noted, that these problem area's do not include things like line-breaking, formula composition or table construction. There are a number of good programs that can put adjusted text or a formula into a box, but the problems we discuss are caused by shortcommings of the user interfaces to these programs.

In this paper, we will present an approach to text formatting, which alleviates these difficulties. It takes the form of a proposal for a user interface to the already existent line-breakers etc. In our system, the end user has at his diposal very concise format directives and the support of an error handling mechanism. The layout designer has access to high level tools for the implementation of format directives. Most (but not all) of the discussed system has been implemented (see the last chapter). The design presented here takes into account experience gained through this implementation.

Design objectives

The aim of our formatting system is, to remedy the problems mentioned above. To satisfy the needs of the end user and the layout designer, the system must meet some criteria which we will state now.

For the end user we require:

1. Descriptive markup, instead of procedural markup.[5] Ideally, this means that the end user only has to indicate which input text goes into for instance footnotes, which into headings etc, while all decisions about the layout are made by the system.
2. Error detection and whenever possible, recovery.

As far as possible, the end user needs only to mark the beginning and end of different kinds of text (such as a footnote or numbered list). The number of other directives (such as font changes or line length specifications) that has to be given must be kept to the logically possible minimum. To assist with markup, the system must know about restrictions for the type of text at hand. For instance, a numbered list as part of a title may be forbidden. In this case, the user must get an elaborate error message. Furthermore, recovery measures must be taken to generate as much meaningful output as possible.

For the layout designer we demand:

1. A simple model of the formatting process, i.e. a clear framework in which a document can be thought of in terms of its constituent parts and operations thereon.
2. An implementation of this model. By a somewhat loose analogy, this implementation can be viewed as to realize a microprogrammable format machine, analogous to a normal microprogrammable computer. The input text mixed with format directives which goes to the format machine corresponds to a machine level program plus data for a microprogrammable computer. The format directives are implemented through programs in a special language (see below); those programs correspond to the microprograms, and their statements to the microinstructions of the microprogrammable computer. Finally there is -just as in genuine microprogramming- a strict separation between the levels of use of format directives (machine instructions) and statements in the language in which they are implemented (micro instructions).

3. A high level programming language for the implementation of format directives. (The languages used in most systems are not of a sufficiently high level.)
4. Special instruments for string analysis and construction.

It should be emphasised, that we presume a system which runs in a working environment, where end use and layout design are distinct. Hence, definitions of new formatting directives are not part of the normal textual input to the formatter, but are made known in a different way.

END USERS VIEW

Document classes

Documents with different purpose can have quite different formats, as is the case with letters, the plain text of a novel, an article in a technical journal, etc. A group of documents with a characteristic format, will be called a document class. To minimize the number and complexity of the format directives to be given by the end user, the following strategy is used:

1. For every document class there exists a specific set of format directives.
2. The form of similar directives of different classes is as identical as possible (e.g. the markup of a section heading is the same in a novel and in a technical text).
3. The details of the effect of the directive on the formatted output depends on the document class. In one class, a footnote reference may be an automatically generated number, while in another class asterixes may be used.

It should be noted that, although conceived independently, this concept of document classes bears a very close resemblance to ideas incorporated in the SCRIBE [10] system.

Text types

A formatted document is thought of as being composed of blocks of different kinds of text (such as footnote or title). Those blocks are distinguished according to:

1. Content, e.g. plain text, illustrations, footnotes, formula's, etc.
2. Positioning with respect to the page (e.g. header at the top, footers at the bottom etc.).

3. Positioning with respect to each other (see the examples in fig. 1a and 1b).
4. The values of bounds on the sizes of blocks of the type.

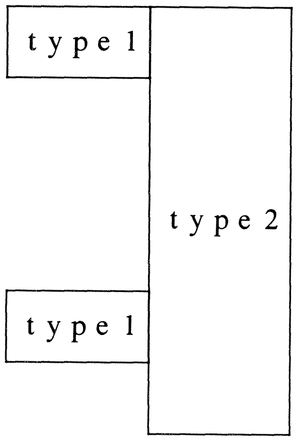

figure 1a.

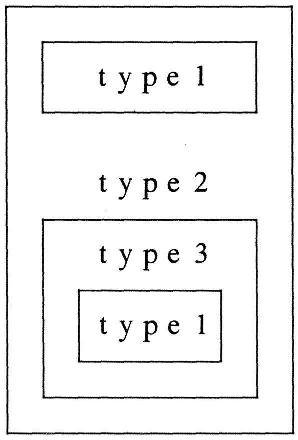

figure 1b.

Blocks differing in one or more of these characteristics are said to be of different type. In this context we will use the term 'block type'. At the input side, text elements can be distinguished according to the type of the block in which they will be placed upon output. Hence, with every block type there corresponds an input text type to which we will refer as 'text type'. This type of a piece of input text is specified by format directives indicating its begin and end (see the next section). We will call such a piece of text a 'marked text piece', or 'marked text' for short.

As seen from the example of fig. 1b, blocks of certain types may be embedded within other blocks. This is reflected in the nesting of the corresponding marked text pieces in the input. Not every conceivable nesting is allowed, however. It seems meaningful, for instance, to permit blocks of type 'numbered list' within blocks of type 'plaintext', but not within blocks of type 'title'.

Now we can define the concept of document class more formally. A document class is characterized by the specification of the properties of the block types of the text blocks that may be used to construct a document from the class. This implies, that every document class has its own set of nesting permissions for marked text pieces.

Markup

In this section we discuss the nature of the format directives an end user may place in an unformatted document. Directives fall into four different catagories:

1. Character level directives.
2. Text type demarcation directives.
3. Special action directives.
4. Parameter setting directives.

Character level directives serve to put accents over characters, to generate overprints and so on.

Demarcation directives indicate the start and end of marked text pieces. They must always occur in pairs, because we do not allow a begin directive to imply an end directive for a preceding marked text piece.

When the unformatted document is scanned from begin to end 'current text type' is defined as follows. When a begin marker is encountered, the current text type becomes the one implied by this marker. When the corresponding end marker is seen, the current text type is reset to the type of the text surrounding the marker pair. To make this work, we assume the presence of an implicit pair of outermost demarcation directives, marking the start and end of the document. Text belonging to the outermost level is discarded by the system; only explicitly marked up text filters through into the output.

Action directives are used to generate output not corresponding to **textual** input. Obvious examples are 'generate a blank line' and 'jump to the next tab position'.

Another aspect of actions is that they reduce the amount of redundant markup that would result from the obligation to mark both the beginning and end of every marked text piece. For example, in a sequence of paragraphs which all have to start with say a blank line, we do not need to mark both the beginning and end of every paragraph. The whole sequence is marked with one pair of directives (e.g. 'begin plaintext' and 'end plaintext'). Every single paragraph within the sequence is begun with a 'start paragraph' action, which will generate a blank line. The redundancy is further reduced through the nesting of text types. Consider a formula embedded in plaintext. The marker 'end of formula' implies the resumption of plaintext. In a system where only the beginning of text blocks must be marked, the 'end of formula' directive would merely be replaced by a 'begin of plaintext' directive. All in all, in spite of the use of directive pairs, the number of directives needed in our markup scheme, is close to the logically possible minimum.

Parameter setting directives are used to change the values of numeric or boolean variables that influence the format process. Examples are 'page size' and 'margin width' (numeric) or 'adjust on' and 'hyphenate on' (boolean). Thefull set of parameters is duplicated for every text type, hence for instance every text type has its own margin width. The scope of the effect of a parameter setting is given by the following rules:

1. When a parameter setting directive occurs between the demarcation directives of a marked text, it only has effect for text of the type of that marked piece.
2. This effect is carried over from the marked text in which the setting took place to subsequent marked text pieces of the same type.
3. When a parameter setting occurs outside an outermost demarcated text block, it is valid for all text types.
4. For every parameter, there is a (text type- and document class dependent) default value with which we start off.
5. There is a 'reset' directive, which resets all parameters to their default values. Just as with the parameter setting directives, reset can either have a global, or a local effect.

The most common use of parameter setting directives is to temporarily switch off hyphenation or change the typefont, but it is also possible to change, for instance, the line length in footnotes.

Parameter settings are so-called procedural directives.[5] It is a trend in current formatting systems to eliminate them as much as possible. (In the SCRIBE[10] system, they are eliminated altogether.) In our system there is a tradeoff between the number of document classes and text types to be dealt with and the frequency of use of parameter setting directives. As will be seen below, it is up to the layout designer to decide what balance he will strike between the two extremes possible.

Error handling

The markup of a document is subject to the following (class dependent) restrictions:

1. The range of permissible parameter values is constrained.
2. The amount of text in a marked text piece may be limited.
3. Proper nesting of marked texts is required.
4. Only certain text types and actions may be nested within a given other text type.

When the system detects a violation of these restrictions, the user will be notified. Whenever meaningful, the error message will contain a suggestion for a remedy. In that case, the system proceeds as if the suggestion has been followed. In that way, the amount of meaningful output per run is maximized.

Enforcing these restrictions has turned out worthwile. Without introducing redundant markup, a lot of missing or misplaced directives are spotted by the system. (Unfortunately, we have no statistics on this subject.)

In the remainder of this section we will discuss the restrictions in more detail.

Ad 1. The domain of values of an individual parameter is restricted. For instance, an indentation depth larger than the maximal width of the physical output device is considered an error. Furthermore, parameters can have conflicting values; consider an indentation depth bigger than the line length.

Ad 2. While for instance a paragraph might be arbitrarily long, more than one page of text in a single footnote may seem undesirable.
Ad 3. Nesting of chunks of text could be accomplished by giving every text type its own begin marker and providing one common end marker. Providing private end markers for every text type gives more redundancy, however. In that way some of the cases in which the user has lost track of the nesting are detected. In the strategy we have adopted, it is checked whether an end marker is in accordance with the current text type.
Ad 4. With every document class goes an internal system table which describes allowed nestings of directives and actions. It is shown in fig. 2.

		encountered directive					
		type 1	...	type n	action 1	...	action m
current	type 1	0 or 1	...	0 or 1	0 or 1		0 or 1
text	⋮	⋮		⋮	⋮		⋮
type	type n	0 or 1	...	0 or 1	0 or 1	...	0 or 1

figure 2.

The rows of the table correspond with the text types of the document class, the columns with the text types and actions. Suppose, there are n types and m actions. When entry (i,j), $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant n$ contains a '1', text type j may be nested in text type i. When entry (i,j), $1 \leqslant i \leqslant n$, $n < j \leqslant n+m$ contains a '1', action $j-n$ may be used when text type i is the current one. All other nestings are forbidden.

Furthermore, there is an additional table that tells which text types may occur at the outhermost level. (Example: a footnote as the outermost text type makes no sense.)

LAYOUT DESIGNERS VIEW

What does a layout designer do?

It is the task of a layout designer to specify a document class and to implement the format directives pertinent to it. This task encompasses the following:

1. Specification of how the text from a marked text piece is to be placed in a block.

2. Specifications of the effect of actions.
3. Specification of the denotation for the format directives.
4. Specification of the nesting permissions.
5. Specification of how the text blocks are to be placed on a page.
6. Implementation of the specifications, i.e. 'microprogramming' the format machine. This includes:
    1. Writing procedures for text types and actions.
    2. Writing procedures for the assembling of a page from already formatted blocks of text.
    3. Adding these procedures to the formatter.
    4. Notifying the formatter of nesting permissions.

    The last two of these tasks are accomplished by specifying entries in various system tables.

Again we mention that these activities are above the level where lines are broken or a formula is formatted.

Language tools

In this section we discuss requirements to be imposed on the programming language used for the implementation of format directives.

In general, the language should be of a much higher level than the (macro) languages often used in text formatting systems. In spite of its prime importance, we will not attempt to go into this subject any further. We will only discuss those language aspects, which are of specific importance to text formatting.

In the first place, we need string manipulation facilities of the following kind:

1. A string construction mechanism.
2. Operators or functions returning the dimensions of constructed strings.
3. A string analysis mechanism.

We will discuss these elements one by one.

The string construction mechanism has the aim of providing a tool with which one can assemble blocks of text into bigger blocks. As an example, think of the construction of a newspaper page, build up from headlines, columns etc. Hence, the language must provide a data type 'string' with much more powerful properties than those of the conventional one-dimensional string. As usual, a string is made up of characters, but different characters can have different sizes and can belong to different typographic fonts. A character is an n-tuple of the form (fontnr, charnr, typographic data). Typographic data include at least character width, height, the height of ascenders or descenders and a measure for slantedness. Furthermore, there is one special character which behaves as a two-dimensional rubber band. It starts out with zero dimensions, but can stretch and shrink with uniform elasticity in the horizontal and vertical directions. This character is a generalization of the rubber bands discussed by Gimpel,[4] but a restricted form of Knuth's glue.[3] For a character to be part of a string, its position in a two dimensional quarter plane (the string plane) must be specified. A string is any set of characters distributed across such a plane. How this distribution comes about will be discussed below. (In this discussion, we assume that the string plane is the lower righthand quarter plane, with the positive y-axis pointing downward.)

A general string construction mechanism for '2-D' strings could be realized by two dedicated language constructs: a string labeling-, and a concatenation construct. The labeling construct is used to mark positions in the string-plane. In an improvised notation, it could look like:

```
LABEL string1 WITH {
      lab1 AT pos1,
      lab2 AT pos2,
              .
              .
              .
      labn AT posn
};
```

figure 3a.

Here, *string1* is a string identifier, *lab1,...,lab2* are label identifiers and *pos1,...,posn* denote expressions indicating a position in the string plane. A concatenation construct is used to paste together two labeled strings, i.e. a new string plane is generated, containing images of the original strings. An image arizes through scaling and translation.
Concatenation could look like:

```
   CONCATENATE string1 AND string2 PLACING {
      string1.labi ON string2.labj,
      string1.labk ON string2.labl,
              .
              .
              .
};
```

figure 3b.

Here, the points in *string1,* labeled *labi* and *labk,* are placed in the new string plane in the same position as the points *labj* and *labl* of *string2.* The upper left-hand corner of *string1* is placed at the origin. When the strings do not contain rubber band characters, one label in each string will do, otherwise more labels can be used. It should be noted that this type of concatenation may fail. It is quite possible to specify sets of points in both strings, all of which cannot be brought to coincide pairwise.

Although quite general, this mechanism is much to cumbersome for frequent use. We would probably want to use notions like 'upper left-hand corner' to take the place of explicit labels. Furthermore we would want different compact notations (e.g. operator notations) for concatenating two strings in the various ways suggested in the diagram below:

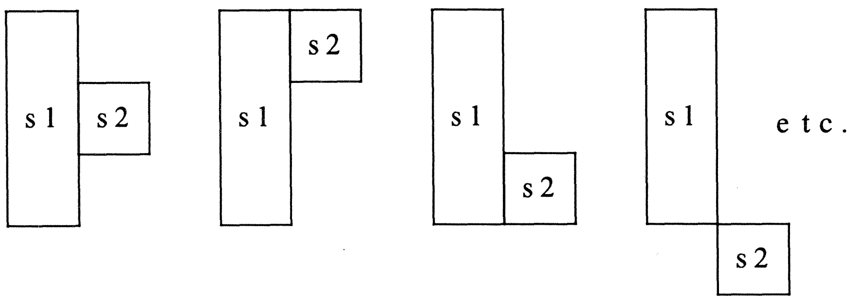

figure 4a.

There are infinitely many of these concatenation configurations. A way out would be a language in which a user can declare his own operators. He would then construct concatenation operators for his favorite string constructions, using the general mechanism outlined above. An operator definition for the second concatenation diagrammed in fig. 4, could look something like:

```
STRING OPERATOR conc2 (s1, s2)
STRING s1, s2 ; {
        LABEL s1 WITH Ls1 AT (width(s1), 0);
        LABEL s2 WITH Ls2 AT (0, 0);
        CONCATENATE s1 AND s2 PLACING s1.Ls1 ON s2.Ls2
};
```

figure 4b.

In this example, *width()* is a function returning the width of its string argument. So, *(width(s1), 0)* stands for the coordinates of the upper righthand corner of string *s1*. As a complement to the construction operators, we need a string decomposition device. It must be possible to specify a window in a string plane and to perform an operation which returns the characters of the string that can be viewed through this window.

The dimension returning functions are quite trivial. An instance of their use can be seen in the example of an operator definition. We only need the functions *width* and *height* which take a string as argument. They return the x and y coordinate, respectively, of the lower right-hand corner of the smallest rectangle whose upper left-hand corner lies at the origin and which contains the characters in the string plane. (The sides of the rectangle are parallel to the coordinate axis.)

Though it may come as a surprise, the string analysis functions we have in mind only deal with conventional one-dimensional strings. They are mainly used for analysis of the unformatted input. Although most of this work is automatically done by the system on a level below the one where the advanced user operates (see the subsection on input fetching), the availability of a string pattern matching mechanism is nevertheless of great value. The demands made on the mechanism are not so specific. Pattern matchers as diverse as those in SNOBOL4, [13] SPRING [1] and its successor SUMMER [14] are suited for the task.

Finally, a question related to the programming language used, is whether there is a compiler for that language supporting incremental compilation. This would come in handy, because the layout designer is supposed to add pieces of code to the already existing formatter code. A detailed treatment of this subject for an application comparable to ours can be found in the discussion of the extensible text editor EMACS. [6]

A model for text formatting

Let us view formatting as the mapping of marked up unformatted input to formatted output, where the mapping is restricted by constraints.
Examples of constraints are:

1. A text of a certain type must fit in a block of 2 by 5 inch.
2. A page must start with text of type 'header'.
3. The last two lines of a paragraph should not be placed at the top of a page (i.e. 'widows' should be avoided).

We distinguish between static- and dynamic formulation of constraints: The examples above are formulated in a static way. A dynamic formulation of the third constraint would be:
'When a page is about to be ended, check if there are at most two lines of the current paragraph left. If so, postpone the termination of the page until the remaining lines are included.'

In this view, the central problem is how constraints are presented to the formatter. In a system as NROFF, [2] both static and dynamic constraints are used; examples are the specification of line length and the setting of a trap, respectively. These constraints are given as part of the unformatted input. In TEX, [3] there is a special form of static constraints which arize through penalties associated with, for instance, potential line breaking points. Certain penalties are preset by the system, additional ones can be specified by the user as part of the unformatted input. We feel that an ideal system should:

1. Strictly separate constraint specification from textual input.
2. Employ a static formulation of constraints.
3. Provide a more natural way to formulate constraints than through the sole use of penalties.

Unfortunately, this is too far fetched, because:

1. Some tasks must be left to the end user; otherwise the number of document classes and text types would explode.
2. We would need a powerful formalism for specifying predicates on text types.

The last problem can be alleviated by using dynamic constraints; they can be stated in existing algorithmic languages. Hence we settle for a hybrid approach; some constraints are formulated statically, others dynamically. We will discuss these in the following two sections.

Static aspects

The source code of the formatter consists of three sections:

1. The fixed code of the format machine.
2. A collection of tables, whose entries are supplied by the layout designer.
3. Code for the 'microprograms', again supplied by the layout designer.

Form of the layout directives

The first thing a layout designer must do, is to specify the text types and actions needed in the document class for which he is about to 'microprogram' the formatter. Next, he must decide what the begin- and end directives and action directives, respectively, will look like. Their denotations will serve as keys in the formatter tables.

Text types and buffers

When text is formatted, certain text blocks can only be positioned long after they have been constructed (e.g. footnotes), while the position of others with respect to surrounding blocks is immediately known (e.g. a numbered list). So, for the first kind we need temporary storage space. For the second, we can use the private storage of the first surrounding text type which has such storage. When a text type needs a private buffer, this buffer must be declared. This is done by making an entry in a formatter table. The key in the table is the denotation for the begin marker for the text type. When the text block is to be limited in size, the size restrictions are specified in the same table.

Buffers have a formatted text part and an unfinished text part. The formatted text part contains text that has been brought in its final form, the unfinished text part is a temporary storage area for text elements which still need further processing.

Filling of text buffers

For every text type, the layout designer must supply a procedure (which we will call a text type procedure), which places text of that type in the appropriate buffer. These procedures take as input unformatted text, or the content of buffers filled by other text type procedures. Their names are entered in a table having as keys the begin marker of the type. Often text type procedures are easy to construct. The system provides built-in routines, which can do most of the work. Examples are: *addword(wrd, bf), addline(bf), plaintext (wrd, bf).*

Procedure *plaintext* is the built-in line breaker. Whenever it is called, it stores the text-word argument *wrd* (a word from the input or from the text part of another buffer), in the unfinished text part of the plaintext buffer. From time to time the line breaker decides to add the words stored sofar to the formatted text of the buffer. Whether this proceeds line wise, or for instance paragraph wise, is immaterial at this level of use. When the layout designer needs explicit control at the line level, he can use procedure *addword* to put a word in the unfinished text part. From time to time, he may call *addline* to construct a line from those words and to append it to the text part of the buffer.

Actions

Actions are implemented through procedures, supplied by the layout designer. A procedure is invoked when the corresponding action directive is encountered in the unformatted input. As usual, this is done through a table. Actions manipulate, or contribute, to the text in the buffer in which the current text type procedure places its output. Again, there are some built-in procedures to assist in action writing.

Nesting

Another table to be filled by the layout designer is the nesting permission table (see fig. 2). There is one complication to be aware of here. Text types which own a private text buffer cannot be invoked recursively, because the buffers are allocated statically. This must be reflected in the nesting permission table. This restriction may seem quite severe, but we are not aware of practical situations * where it really causes problems. Meanwhile, the restriction greatly simplifies the format machine, and more important, the model of text formatting is kept straightforward.

* It turns out, that text types which must be used recursively, (notably numbered lists), can do without private buffers.

Page construction

At certain moments, a page must be constructed. When this is the case, procedure *buildpage* is invoked. This procedure is supplied by the layout designer. It describes what the page will look like in terms of its constituent text blocks. The invocation moments are determined by dynamic constraints to be discussed later.
Example: Suppose a document class contains the text types header, footer, footnote, plaintext and title. All text types, except title, have a private buffer. Title puts its output text in the plaintext buffer. We will denote the text types by *HR, FR, FN, PT* and *TI.* The (text parts of) buffers are denoted by *buff[HR]* etc. A page is built up from a header at the top, followed by plaintext, followed by footnotes if any are present, and is terminated by a footer at the bottom. In a crude form, procedure *buildpage* looks as shown * in fig. 5.

```
PROC buildpage {
        "generate page number, make it part of the header";
        print(lpile(buff[HR], vspace, buff[PT], vspace,
                        IF height(buff[FN] > 0 THEN
                                buff[FN], vspace, buff[FR]
                        ELSE
                                buff[FR]
                        FI
                )
        );
        buff[PT]:=buff[FN]:=empty
}
```

figure 5.

Here, *lpile* is a concatenation function with a variable number of arguments. It returns a string, constructed form its arguments, by placing them above each other (the first argument comes at the top), with their left sides alligned. The variable *vspace* denotes a string of blank space and *empty* denotes the empty string.

The text blocks used here resemble the ones in TEXTURE [7] and bear a resemblance to the objects from which galley-beds are constructed in JANUS. [9] They are quite different from the boxes in TEX. [3] The use of TEX's boxes ranges from encompassing single characters till whole pages.

* In this and the following program examples, we use the language SPRING [1], apart from the fact that we represent keywords in capitals.

Dynamic aspects

The purpose of this section is, to discuss dynamic constraints in greater depth. To this end we must describe the flow of control in the format machine and some of the fixed (not microprogrammable) tasks it performs. This is the subject of the following three subsections. Thereafter, dynamic constraints themselves will be discussed.

The formatting process

The operation of the format machine is sketched in figure 6. The format process has the form of a loop. On every iteration a piece of input is fetched and returned by procedure *getinp*. It is a self-contained unit which is either a format directive, or a piece of text. If it is a piece of text it either is a chunck of input text (preprocessed by *getinp)* or it is a block of text which is already formatted, but which must be incorporated in another block (see the next subsection).

```
WHILE input:=getinp(flag) "not equal end of input" DO
        IF input "is a begin of text type directive" THEN          (1)
                "check nesting permission";
                "do text type switching";
                dynconst3[oldtype][curtype]()                      (2)
        ELIF input "is an end of text type directive" THEN
                ............................
        ELIF input "is a parameter setting directive" THEN
                "check parameter value";
                "set parameter";
        ELIF input "is an action directive" THEN                   (3)
                "check nesting permission";
                dynconst1[input]();                                (4)
                result:=acctable[input]();                         (5)
                dynconst2[input][result]();                        (6)
        ELIF input "is a textual unit" THEN
                report:=texttypetable[curtype](input);             (7)
                dynconst6[curtype][report]();                      (8)
        FI
OD;
```

figure 6.

When the unit is a textual unit, it is passed to the routine for the current text type. The name of this routine is found in table *texttypetable,* (see statement 7 from fig. 6) using the key *curtype.* The routine itself must have been supplied by the layout designer.

When the input is a format directive, further processing depends on whether it is demarcation directive which denotes the begin or end of a text type, an action, or a parameter setting directive. If it concerns a demarcation directive, (statement 1) text type switching (see below) occurs. If it concerns an action, (statement 3) a procedure supplied by the layout designer from the table *acttable* (statement 5) is invoked. In all these cases, checks are made to see whether the required operation is allowed. Aspects which have not yet been discussed are: input fetching, text type switching, and dynamic constraint specification. We will turn to these matters in the next sections.

Input fetching

Procedure *getinp* delivers input units to the main formatter loop. They can be fetched from two different sources: from the unformatted text or from the private buffer for a text type. In the last case, such a unit is a block containing fully formatted graphical material. We will discuss this case first.

Suppose that, during formatting of plaintext, an equation must be processed. 'Equation' will probably be a text type with a private buffer. The formatted equation will be put in this buffer. When the processing of plaintext is resumed, the first thing that must be done is to pick up the equation from the equation buffer and to place it in the plaintext buffer. This fetching is done by *getinp,* the placement by the normal plaintext routine. This mode of operation is not standard; a formatted footnote must be left in its buffer until a page is constructed and must not be further processed by the plaintext routine. There is a table (given by the layout designer) which tells which text type procedures generate output that must still be processed by the procedure dealing with the surrounding text type. Output from the other text type routines is left untouched until the page is constructed.

The other (and more common) mode of operation of *getinp* is when units of input from the unformatted document are fetched. What constitutes a unit is determined by the value of *flag,* which is passed as a parameter to *getunit.* It is intended that *flag* is set to the appropriate value by the text type procedures. A unit can be:

1. The next word from the input, i.e. a string of characters not containing blanks or newlines.
2. The next input line, unless this line contains the end marker for the current text type. In that case, the string preceding the marker (if any) is returned first followed by the marker itself.
3. All input until, but not including, the end marker of the current text type. Thereafter the end marker itself is returned.

Case 1 is the most common one and it is intended for plaintext. Case 2 is needed when, for instance, lines from the unformatted input must be copied literaly to the formatted output. Case 3 will, among other things, be used by a formula processing routine because it needs the whole formula before its layout can be determined. When the second or third modes of operation are used, it is possible that layout directives hidden in the input units returned by *getinp* are not automatically recognized by the formatter. So these modes must be used with care. The end marker of the current text type is always returned as a separate unit and hence is always recognized.

Finally, *getinp* performs operations on the character level like accenting or overprinting. This is a built-in facility not to be tampered with by the layout designer.

Text type switching

In this section, we outline what happens when a text type is superceded by another one. Part of the process sketched in fig. 6, is shown in more detail in fig. 7.

```
IF input "is a begin of text type directive" THEN
        "check nesting permission";                              (1)
        newtype:="type of text, about to begin";
        report:=texttypeinit1[newtype]();                         (2)
        dynconst6[newtype][report]();                             (3)
        IF "curtype may be called recursively" THEN               (4)
                stack(stackedstorage[curtype])
        FI;
        stack(curtype);                                           (5)
        curtype:=newtype;                                         (6)
        IF "curtype may be called recursively" THEN               (7)
                stackedstorage[curtype]:=default[curtype]
        FI;
        "set parameters";                                         (8)
        report:=textypeinit2[curtype]();                          (9)
        dynconst6[curtype][report]()                              (10)
FI;
```

figure 7.

The crucial statement is statement 6. After it is executed, the current text type is the one whose begin directive has just been recognized. Before its execution, the type of the more global marked text piece is still current.

Before and after the text type switch, initialization routines are invoked (see lines 2 and 9). These routines are addressed through tables *texttypeinit1* and *texttypeinit2,* respectively. The first operates in the environment (buffer, parameter values) valid before, and the second in the environment valid after the text type switch. An example of their use is the following: Suppose a footnote text is encountered in the middle of plaintext. Through *texttypeinit1* a routine will be invoked which puts a reference to the footnote in the private buffer for plaintext. A routine from *texttypeinit2* will attach a corresponding label to the footnote text, this time in the footnote buffer.

Now, we turn to statements 4 and 7. Certain text types can be nested recursively. Hence, storage space for variables may be needed, which is properly initialized, stacked and unstacked. The address of the storage area for text of type *curtype* is found in the table element *stackedstorage[curtype].* It is put there by the layout designer. The storage area itself is declared by the layout designer as part of the 'microprogram section' of the formatter. In statement 4, the storage area (if any) of the more global text type gets copied to the stack. In statement 7, the storage for the text type that has just been made current, is initialized.

In statement 5, the designation for the more global text type is stacked, while in statement 9, the parameters are set to values pertinent for the new text type. Statements 3 and 10 will be discussed in the next section. Finally we remark that, when the end marker of a text type occurs, an analogous sequence of steps is taken. The main difference is, that text type identification and storage contents, are restored from the stack.

Dynamic constraints

The aim of dynamic constraints is, to provide the layout designer with the tools to make the formatter react as desired when certain conditions pertinent to the document layout become true. Again, this must be done through 'microprogramming' of the formatter, without modifying its fixed code.

First we observe that the kind of conditions we have in mind can only change at very specific moments during the format process. We call those moments synchronization points. They are:

1. Just before the invocation of an action procedure.
2. Just after the invocation of an action procedure.
3. When the begin delimiter for a text type is encountered.
4. When the end delimiter is recognized.
5. When a text type is made current again, after the termination of a type nested within it.
6. When a text type routine or its initialization or termination routines may have put text in a text type buffer.

Furthermore, a routine which may put text in a buffer must return a report describing what it has done. Possible reports are:

1. Word added (to the unfinished text of the buffer).
2. Line added (to the bottom of the text part of the buffer).
3. Block added (to the bottom of the text part of the buffer).
4. Buffer full (i.e. last added text element filled up the buffer).
5. Error.

These standard reports are believed to be sufficient but, if needed, the layout designer can extend the list.

With every synchronization point, a different table of functions is associated from which a function is invoked whenever that point is reached. The functions are supplied by the layout designer. Which function is chosen from the table depends on the most recent demarcation- or action directive and may further depend on the last report returned by a text type- or action procedure. By way of an example, see statements 3 and 10 from figure 7, or statements 2, 4, 6, and 8 from fig. 6.

More precicely, the six tables containing contraint routines are indexed as follows:

tabel type	indices
1	action designation, result-report
2	"
3	type of text about to begin
4	type of terminated text
5	type of terminated text , type of text about to resume activity
6	current text type, result-report

figure 8.

It should be noted that through the introduction of fixed synchronization points and standard reports, a mechanism is provided to partially specify the conditions pertinent to dynamic constraints in a systematic way. When a dynamic constraint procedure is invoked, it is guaranteed that certain conditions belonging to a fixed, limited, set are met. (For instance, a line has been added to a buffer by an action procedure.) In general, the procedure will perform additional tests of course.

Example: We extend the example of fig. 5 and sketch two dynamic constraint routines (see fig. 9b). The first, *PTlineadded*, is invoked whenever the plaintext routine reports that a line has been added to *buf[PT]*. It is a routine from the sixth constraint table. The other, *TIstart,* checks, before a title is added to *buff[PT],* if there is enough space left on the current page for the title plus three text lines. If not, a page is constructed first, and the title will be placed at the top of the next one. *TIstart* is a routine from the third constraint table.

```
PROC texth {
        return(height(buff[HR])+height(buff[FR])+height(buff[PT])+
                height(buff[FN]+2*height(vspace)+
                IF height(buff[FN]) > 0 THEN
                        height(vspace)
                ELSE
                        0
                FI;
        )
};
```

figure 9a.

In the example, *pageh* (see fig. 9b) stands for the height of a page and *texth* (see fig. 9a) is an auxiliary function provided by the layout designer. It returns the height the page would have if it were constructed from the contents of the text buffers as they are at this moment.

```
PROCEDURE PTlineadded {
        IF texth = pageh THEN buildpage FI
};


PROCEDURE TIstart {
        IF pageh - texth < "space for title + 3 lines" THEN
                buildpage
        FI
};
```

figure 9b.

At this point, a comparison with traps, as occuring in NROFF, [2] is in place. In that system, traps are used to implement what we have called dynamic constraints. A trap is set to indicate that at specific moments during the format process, specific macros must be invoked. These moments can be:

1. When line N of the current page is reached.
2. When line N of the current diversion is reached. (Text that cannot immediately be placed on the current page, for instance footnote text, is temporarily diverted to a buffer, called a diversion.)
3. When line N from the input is reached.

The main difference between this mechanism and our one is that in the latter we differentiate between much more synchronization moments. In particular, information about the nesting of text types and actions at a specific synchronization moment is implicitely provided, a feature absent in NROFF. This information is needed for instance, if we want to decide whether an action which normally generates a blank line, will really place a blank line in the output, or cause a new page to be started. (Note that a blank line near the bottom of a page should be avoided in plaintext, but not in a footnote.)

AN IMPLEMENTATION

A large subset of the system discussed in the previous chapters has been implemented and has been in use for about two years. The system that is proposed here is in some aspects an improvement on the implemented system. Experience gained through the implementation has been taken into account. The differences are not modifications of the original design; they have the nature of extensions.
The most important are:

1. The string construction mechanism is much more powerful than the one implemented.

2. Procedure *getinp,* which analyses the unformatted input, is slightly more powerful than the one implemented.
3. The 'reset' directive was absent in the initial design.

Apart from these modifications, the implementation realizes a subset of the original design in that:

1. The implementation only deals with fixed size characters (i.e. typewriter fonts).
2. Only a simple line-breaker and no mathematics formatting primitives have been provided.

The system has been implemented on a PDP11/45 under the UNIX [12] operating system. The language used is SPRING. [1] The format machine is implemented in this language; it is also the language to be used by the layout designer. The choice of SPRING brings about a string construction mechanism less poweful than the one desired and precludes the use of variable sized characters. The choice ot this language is nevertheless considered fortunate. It meets our other requirements and, on top of that, a data type 'table' is available.

The most important problems encountered were those of memory requirements and speed of execution. The SPRING compiler generates code that is interpreted. On the PDP11/45 the code of the interpreter plus the interpreted formatter code does not leave much space for data. The result is that the system spend most of its time doing garbage collection, especially when large pages are formatted (which are wholly kept in core). The execution time does critically rise with page size. For a page of seventy lines, a formatting time of up to thirty seconds may well result.

In spite of these severe restrictions, the system provides a working environment, which was up to the expectations.

REFERENCES

1. P. Klint, *From SPRING to SUMMER: Design, defintion and implementation of programming languages for string manipulation and pattern matching,* Stichting Mathematisch Centrum, Amsterdam, (1982)
2. J.F. Ossanna, *NROFF/TROFF User's Manual,* Bell Laboratories Computing Science Technical Report 54, 1976
3. D.E. Knuth, *TEX and METAFONT,* Digital Press, 1979
4. J.F. Gimpel, *Algorithms in SNOBOL4,* John Wiley & Sons, New York 1976
5. C.F. Goldfarb, *A Generalized Approach to Document Markup,* Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, 68-73 (1981)
6. R.M. Stallman, *EMACS, the extensible, customizable self-documenting display editor,* Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, 147-156 (1981)
7. M. Gorlick, V. Manis, T. Rushworth, P. van den Bosch, T. Venema, *Texture, A document Processor,* Department of Computer Science, University of Britisch Columbia Vancouver B.C, Technical Report 76-1 (1976)
8. J.O. Achugbue, *On the line breaking problem in text formatting,* Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, 117-122 (1981)
9. D.D. Chamberlin, J.C. King, D.R. Slute, S.J.P. Todd, B.W. Wade, *JANUS: an interactive system for document composition,* Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, 82-99 (1981)
10. B.K. Reid, *A High-Level Approach to Computer Document Formatting,* Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, 24-31 (1980)
11. L. Tesler, *PUB The Document Compiler,* Stanford Artificial Intelligence Project, Operating Note 70, (1973)
12. D.M. Ritchie and K. Thomson, *The UNIX Time-Sharing System,* CACM, 17, No7, 365-375, (July 1974)
13. R.E. Griswold, J.F. Poage, I.P. Polonsky, *The SNOBOL4 Programming Language,* Second Edition, Prentice-Hall, Englewood Cliffs, N.J, (1971)
14. P. Klint, *An Overview of the SUMMER Programming Language,* Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, 47-55 (1980)