

A Galley and Page Formatter Based on Relations

by

Lok, Shien-Wai (Frank)

**School of Computer Science
McGill University**

© July 1985

**A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Masters of Science (Computer Science)**

ABSTRACT

This thesis deals with a means of representing text based on a set of relations. A relational formatter was developed and implemented. A subset of the "Generalized Markup Language" (GML) was adapted for the purpose of describing the text structure. The line breaking algorithm developed by Knuth and Plass was modified to break pages as well as lines. The method of storing data in relational form in secondary storage is explained and the major requirements and components of the formatter examined.

The experimental formatter runs under the UNIX operating system on the CADMUS 9700 micro-computer and has been written in the programming language "C".

Résumé

Ce mémoire a pour sujet une méthode de représentation de texte, basée sur un ensemble de relations. Nous avons développé un système relationnel de traitement de texte. Dans ce but, nous avons adapté un sous-ensemble du "Generalized Markup Language" (GML) pour décrire la structure du texte. Nous avons modifié l'algorithme de bris de ligne, proposé par Knuth et Plass pour inclure le bris de page. Nous expliquons la méthode de mise en mémoire secondaire des données sous forme relationnelle, et nous examinons les prérequis et les composantes les plus importants de ce système.

Nous avons implanté ce système expérimental en "C", sur micro-ordinateur CADMUS 9700 sous le système d'exploitation UNIX.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Prof. T. H. Merrett, for his supervision and valuable advice during the development of this thesis. The financial support I received through Prof. T. H. Merrett's NSERC and FCAC research grants was highly appreciated.

Many thanks to Wendy Marston for her help in editing and proofreading. I am indebted to the following persons for their aid and moral support: Jennifer Jones, Ryan Hayward, Roger Cormier, Sylvie Sadones, Claudia Wagner, Charles Snow, and my colleagues at McGill. I wish to acknowledge the School of Computer Science at McGill for the financial aid which enabled me to complete my studies.

Table of Contents

1 Introduction	1
1.1 Relations	3
1.2 Relational Algebra	5
1.3 Conventional text representation	7
1.4 Text represented in relational form	8
1.5 Thesis Outline	10
 2 Survey of Existing Text Formatting Systems	 11
2.1 Procedural Approach Systems	12
2.1.1 RUNOFF	12
2.1.2 FORMAT	13
2.1.3 PUB	15
2.1.4 TEX	16
2.1.5 NROFF/TROFF	18
2.2 Declarative Approach Systems	21
2.2.1 SCRIBE	22
2.2.2 GML	23
2.2.3 JANUS	28
2.3 Conclusion	30
 3 System Design	 31
3.1 Basic System Requirements	31
3.2 Formatting Language	32
3.3 Logical Design of the System	33
3.3.1 Editor Based on Relations	34
3.3.2 STRUC and COPY Relations	34
3.3.3 Formatter Based on Relations : COPY and STRUC	38
3.3.3.1 User Interface	39
3.3.3.2 Line/Page Break Module	40
3.3.3.3 Generate Galley Modules : line-galley and page-galley	60
3.3.3.4 Display Driver	62

4 Details of Implementation	63
4.1 Host System Configuration	63
4.2 Programming Language Used	64
4.3 Implementation Goal	64
4.4 Output Device	65
4.5 Simple Description of the Implementation	66
4.5.1 Storage Structure of Relations	67
4.5.2 Pre-Processor for the Formatter	70
4.5.3 User-Interface Module	74
4.5.4 Line/Page Break Module	75
4.5.5 First Pass Module : galley formatting	77
4.5.6 Second Pass Module : page formatting	81
4.5.7 Display Driver Module	82
 5 Tutorial Introduction	 87
5.1 How to Create Input Data File	87
5.2 How to use the Pre-Processor	90
5.3 User Interface with the Formatter	91
5.4 How to use the Display Driver	96
5.5 Formatting Commands	97
 6 Conclusion	 103
6.1 Summary and Advantages	103
6.2 Limitations and Drawbacks	104
6.3 Further Work	105
 Appendix A	 107
 References	 113

CHAPTER ONE

Introduction

In the field of text processing, the common practice has been to build a special system to handle each special task. These tasks have ranged from simple editing to complex linguistic analysis. While this method works well, special systems are usually costly and inflexible as well as complicated. Also, the user must spend an inordinate amount of time learning how to use these systems. If a system could be developed to handle most of the work needed to be done on text, the burden on the user would be alleviated. Moreover, processing and storing text in secondary storage could be unified.

Over the past 15 years, the relational algebra has become well established as a successful mechanism for processing formatted data modelled as relations.

Two aspects of the relational algebra make it practical for use in computer applications. These aspects are "atomicity" and "closure". Atomicity refers to the fact that data objects are undecomposable; the internal structures of these objects are not investigated. This aspect permits a suitable level of abstraction while at the same time allowing implementations to make optimum use of the secondary storage.

Closure refers to the fact that algebraic systems are complete in themselves; the result of an operation in relational algebra is a relation. This aspect allows operands of any given operation to be specified either as simple relation names or as expressions that resolve into relations. As a result of the closure properties, only one set of rules is required for a given operation. There are no exceptions and each execution of an operation is carried out in the same manner. This greatly simplifies the application of relational algebra.

[Merrett 84a] goes beyond common practice in his investigation of operations on text data. These operations include: concordance building, dictionary searching, cryptography, linguistic analysis, editing and formatting. One of the simplest examples shows the advantage of using relational algebra to build a concordance as follows:

Assume we have the following two relations:

ENGLISH (WORD	SEQ)	STOP (WORD)
	about	14		about
	because	8		because
	being	12		being
	Centre	7		I
	code	16		my
	comical	13		the
	coming	3		to
	computer	10		was
	Computing	6		
	I	1		
	my	15		
	the	5		
	the	9		
	to	4		
	was	2		
	was	11		

the command " CONCORDANCE <-- ENGLISH djoin STOP "

will produce the following relation:

CONCORDANCE (EWORD	SEQ)
coming	3
Computing	6
Centre	7
computer	10
comical	13
code	16

As can be seen from the above example, if text data is stored in the relational form then relational operations can be used to process it.

Although Merrett's studies were theoretically innovative, two fundamental text processing problems still existed. An editor was needed to adapt data into relational form, and a formatter was required that would accept data in relational form. The purpose of this thesis was to develop an experimental formatter based on the relational model.

1.1 Relations

Relation is a term used in computer science to specify a particular method of representing data. The formal definition of a relation is:

Given a collection of sets D_1, D_2, \dots, D_n (not necessarily distinct),

R is a relation on those n sets if it is a set of ordered n -tuples $\langle d_1,$

$d_2, \dots, d_n \rangle$ such that d_1 belongs to D_1 , d_2 belongs to D_2 , ..., d_n

belongs to D_n . Sets D_1, D_2, \dots, D_n are the domains of R . The value

n is the degree of R . [Date 81]

It is convenient to represent a relation as a table with the table name as the relation name. Each column of the table corresponds to an attribute and the column name is the attribute name. Each column takes values from one domain only, but more than one column can take values from the same domain. Each row of the table represents one n-tuple or simply one tuple of the relation. The number of tuples in a relation is called the cardinality of the relation. For example, if someone is asked to prepare a class-list with the following information: student's names, id numbers and phone numbers, the most straightforward way would be the following:

CLASS-85

name	id-number	phone-number
GAUTHIER RICHARD	8304829	876 - 8597
CORBET URSULA	7804103	332 - 9772
GERMAN MARK	8106567	392 - 5189
LICZNER RITA	7514040	848 - 8029
GERMAN MARK	49710	392 - 5189

The method presented above is a relation called "Class-85". It contains three attributes: name, id-number and phone-number. Each column represents one attribute and each row represents one tuple of the relation "Class-85".

A relational table has the following properties:

1. All rows (tuples) are distinct
2. The row order is insignificant
3. The column order is insignificant
4. Every value in a relation is atomic (i.e. nondecomposable)

1.2 Relational Algebra

The relational Algebra is a set of operators on relation. In this section, a few basic operators will be described in order to show readers the characteristics of the relational algebra.

- a) Projection - used to get a relation in which the attributes are a subset of the initial set (vertical selection). For example, if one is only interested in the students' names and phone numbers in the relation "Class-85", the projection operator can be used to get a new relation containing only these two attributes.

The projection command:

```
PHONEBOOK <-- name, phone-number in CLASS-85
```

will produce the following relation:

PHONEBOOK

name	phone - number
CORBET URSULA	332 - 9772
GAUTHIER RICHARD	876 - 8597
GERMAN MARK	392 - 5189
LICZNER RITA	848 - 8029

- b) Selection - used to get a relation in which the attributes will remain the same and the output will be a subset of the initial set (horizontal selection). For example, if one is interested in the student "GERMAN MARK", the selection operator can be used to get a relation combining the tuples with the value of the name attribute equal to "GERMAN MARK".

The selection command:

```
INFOFILE <-- where name = "GERMAN MARK" in CLASS-85
```

will produce the following relation:

INFOFILE

name	id-number	phone-number
GERMAN MARK	8106567	392 - 5189
GERMAN MARK	49710	392 - 5189

- c) Natural Join - used to get a relation combining the attributes and tuples of two relations. The set of attributes is the union of the attributes of each relation. The tuples are selected according to the common attribute(s). For example, assume one has another relation, "TESTMARK", containing students' names and test marks:

TESTMARK

name	mark
CORBET URSULA	73
LICZNER RITA	67
CORBET URSULA	65
GERMAN MARK	84
GERMAN MARK	95

The natural join can be used to combine the relations "TESTMARK" and "CLASS-85" to form another relation "STUDENTDATA" with 4 attributes.

The natural join command:

STUDENTDATA <-- CLASS-85 ljoin TESTMARK

will produce the following relation:

STUDENTDATA

name	id-number	phone-number	mark
CORBET URSULA	7804103	332 - 9772	73
LICZNER RITA	7514040	848 - 8020	67
CORBET URSULA	7804103	332 - 9772	65
GERMAN MARK	8106567	392 - 5189	84
GERMAN MARK	8106567	392 - 5189	95
GERMAN MARK	49710	392 - 5189	84
GERMAN MARK	49710	392 - 5189	95

d) **Difference Join** - used to get the difference between two relations "A" and "B" (in that order). The result is the set of all tuples belonging to "A" and not to "B". For example, assume we want to find the phone number of the students who has no marks. The difference join command can be used on the relations "PHONEBOOK" and "TESTMARK" to form another relation "ABSENTLIST" with 2 attributes.

The difference join command:

```
ABSENTLIST <-- PHONEBOOK djoin TESTMARK
```

will produce the following relation

ABSENTLIST

name	phone-number
GAUTHIER RICHARD	876 - 8597

1.3 Conventional text representation

In a conventional system, text is stored exactly as it has been typed into the computer. There are no restrictions on text elements and text can be represented as a sequence of characters without reference to their logical structure or meaning (here, logical structure refers to the organization of the document elements.). In other words, the system has only to preserve the order of the characters and the contents of the text itself.

1.4 Text represented in relational form

Unlike conventional systems, restrictions are imposed on text elements that are to be stored in relational form. These restrictions are the result of the properties of a relation, for example, all data is simple, no repetition of tuples is allowed, etc. Making the user responsible for adjustments in the new internal data representation works against the purposes of our system. As a result, the system was designed to let the user enter the input document as in the conventional method; the system takes care of all the necessary conversions. In this way, the process will remain simple for the user, who can nevertheless benefit from the advantages of relational representation.

Because of the properties of a relation, text stored in relational form is very different from the conventional fashion. In order to store text elements in terms of relations there are three problems that must be resolved: (1) how to separate the text elements into tuples; (2) how to preserve the order of the text elements; (3) how to deal with the repetition of text elements. There are various ways to separate the text into tuples. The text can be separated according to its physical size (i.e. n character(s) per tuple, n line(s) per tuple, n page(s) per tuple; $n > 0$) or according to the logical meaning of the text elements (i.e. n word(s) per tuple, n sentence(s) per tuple, n paragraph(s) per tuple, etc.; $n > 0$). It was decided to separate text according to its logical meaning. Text itself does not include physical size constraints. For example; line length and page size may vary between two different printed versions or according to different display mediums such as screen, typewriter, and phototypesetter. Representing text by line per tuple or

page per tuple adds irrelevant physical constraints, such as line length, which have nothing to do with text itself and which complicate processing. In keeping with the "logical meaning method" it was decided to use one word per tuple. This allows a high amount of control and facilitates the manipulation of the text elements, while using less memory space than a single character. In terms of preserving the order of the text elements, and making each tuple unique, at least two attributes are needed. These attributes are "word" and "wordsequence". "Wordsequence" contains real numbers and serves as an id-number to each word, the value of which determines the order of the text element. For example, if ascending order is used words with smaller values in the "wordsequence" field will precede words with larger values. The relation containing "word" and "wordsequence" will be referred to in this thesis as "FIRST". An example of this relation is as follows:

FIRST

word	wordsequence
	18
just	19
an	20
example	21
	22

Thus, the relation "FIRST" preserves the information and supports all the functions (updating, viewing, storing, etc.) as the conventional method. Ultimately, however, a more advanced method of storing text in the relational model was used, as will be described in chapter 3.

1.5 Thesis Outline

This chapter has been an explanation of the purpose of this thesis and a brief introduction to the concepts of relation and relational algebra. Both the conventional and relational methods of storing text in secondary storage were described. Chapter Two contains a survey of some of the existing text formatting systems. Formatter system requirements are explained, a formatting language selected and a brief discussion about the system design is presented in Chapter Three. Chapter Four is an explanation of the configuration of the host system and programming language used to build the formatter. The implementation of the system is also discussed. Chapter Five is a user's manual. It explains the use of the system without knowledge of the technical details of the implementation. Finally, in Chapter Six, comments on the accomplishments of the system are discussed and suggestions for future work are proposed.

CHAPTER TWO

Survey of Existing Text Formatting Systems

A text formatting system is a computer program designed to deal with the physical layout of a document on a specific medium. Since the first appearance of the text formatting system RUNOFF in the early 1960's, a considerable amount of research has been done in this area. The advantages of using a text formatting system are obvious: It saves time, reduces production costs, is easier for updating and costs less for reformatting. Different formats can easily be achieved, and output can be produced on different devices. The increasing cost of manually produced documents versus the decreasing cost of computer hardware and software further contributes to the popularity and desirability of text formatting systems.

In this survey, we are interested in the following issues:

1. the formatting power of the formatter,
2. the user interface,
3. other features related to the document, such as table of contents, indices, footnotes and cross references.

Systems using the "procedural approach" (low-level approach) will be compared and contrasted to the "declarative approach" (high-level approach), and the advantages and drawbacks of each system discussed.

2.1 Procedural Approach Systems

Systems using the procedural approach to text formatting problems are based on the assumption that the user of the system will want to design the final appearance of the document. The objective of the system is therefore to provide its user with a set of tools (commands) to manipulate the physical layout of the document. In other words, the users of the system are fully responsible for the formatted output, as long as the system provides all the necessary tools.

In the following sections we shall discuss a few systems based on this approach. The systems are ordered by their date of appearance and their formatting capability.

2.1.1 RUNOFF

RUNOFF was one of the pioneer text formatting systems. Appearing in 1964 on the Compatible Time Sharing System (CTSS) at MIT [Furuta 82], it was designed to deal with its input and output on a typewriter-like device. Since RUNOFF was developed when computer technology was in its infancy, its formatting capabilities are limited by the output device. Also, since RUNOFF was one of the first text formatters, it provides relatively few features compared to text formatting systems commercially available today. Essentially, anything produced by RUNOFF can be produced similarly by a typewriter if sufficient time is provided. However, RUNOFF proved that using a computer to perform tedious work results in a great reduction of man hours.

Following are some of the RUNOFF commands:

- a) .center - place the object in the center of a line
- b) .space # - skip # of lines (used to produce a vertical spacing)
- c) .indent # - skip # of spaces (used to produce horizontal spacing)
- d) .undent # - unskip # of spaces (used to reduce the horizontal spacing)
- e) .adjust - start left and right justification
- f) .noadjust - no justification


RUNOFF is obviously easy to use. There are few commands, all command names are self explanatory, and all deal with the simple problem of object placement. However, RUNOFF's simplicity results in several drawbacks. The small set of commands limits the formatting capabilities (this problem is caused by the limitation of the output device). Command names are usually long, leading to a greater chance of typing errors, and costing more time for expert users. Also, since all commands deal with the physical layout of the page, a small change in the input text may require the whole document to be re-organized.

2.1.2 FORMAT

FORMAT appeared in the late 1960's. It was developed by IBM for use on IBM S/360 computers and was designed to accept input text from punched cards and to produce its output to a line printer with both lower-case and upper-case letters.

Since FORMAT was developed a few years later than RUNOFF, it is no surprise to find that its capabilities exceed those of the earlier system. In addition, to simple object placement, FORMAT allows input text to be alternated with formatting commands. Also, a help facility for the indexing process is provided.

There are three types of command in FORMAT, namely: character-level commands, phrase-level commands and paragraph-level commands. Character-level commands are reserved characters that affect only the single letter immediately following the command. For example, "&" is a character-level command [Bernes 69]. If "&" precedes the character 'a', it will change 'a' into the character 'A'. Phrase-level commands are single character and may be grouped together to specify some particular formatting actions. They are initialized by the character ")" and terminated by a blank. The effects of phrase-level commands are valid until the end of the line/sentence unless there is another phrase-level command. For example, ")M& " is a phrase-level command which specifies that the following input text is to be centered and capitalized. Paragraph-level commands are used to define the general formatting rules for the whole document. For example, paragraph-level commands define left margins, page length, meaning of the special characters. These commands are started with ")" and terminated by "GO".



Format obviously exceeds RUNOFF in terms of formatting capability, however, Format is more difficult to use. Since each command consists of only one character, much memorization is involved for the user, making the system harder to learn. Also, the input method and the formatting commands are designed in such a way that it is extremely difficult to make corrections and/or changes in the input description.

2.1.3 PUB

PUB was developed at the Stanford Artificial Intelligence Laboratory in 1971 and was designed to be used on the PDP-10 computer [Furuta 82].

Considered in terms of the physical layout of commands, PUB closely resembles RUNOFF. Both use self-explanatory commands with a period in front to indicate the command line status.

However, PUB introduced several new ideas into the formatting field. First, it borrowed some of the programming language features such as block structures, variables and if....then....else statements, so that its user could have both global and local control of the document layout descriptions and conditional compilations. Second, it provided a macro facility, allowing the user to group the frequently used formatting commands together thus avoiding retyping. Finally, and probably most importantly, the designer of PUB made an effort to eliminate widow lines (isolated single lines).

PUB exceeds both RUNOFF and FORMAT in terms of formatting capability. In addition to providing most of the formatting features of RUNOFF and FORMAT, it provides commands for multiple columns, footnotes, the automatic numbering of sections and subsections, construction of a table of contents, and creation of new characters (by over-striking). In terms of user interface, PUB not only has all the advantages of RUNOFF (excluding, of course, a small set of commands), but it also goes one step further by providing the macro facility and eliminating the undesirable breaking of paragraphs (bad breaks for the user). Moreover, PUB's user is able to redefine most of the formatting actions specified

by control characters. One of the drawbacks in PUB is that since the system provides more features than RUNOFF and FORMAT, it also requires the user to spend more time learning it.

2.1.4 TEX

One of the most powerful formatting systems available is TEX, developed by D.E. Knuth at Stanford University in the late 1970's [Furuta 82]. The system was designed to deal with text interspersed with mathematical and tabular materials. The input to TEX may be entered through any conventional CRT; the output is produced on a photo-typesetter type device. The TEX formatting command system was designed to allow the user full control of the output device. In fact, D.E. Knuth used TEX to typeset the entire second volume of his book: "The Art of Computer Programming" [Knuth 81]. His test was intended to prove that TEX produces the highest possible quality product for its user. Since Knuth emphasized high quality output most of the commands in TEX deal with problems of physical layout. New concepts (box, penalty, glue) and algorithms (e.g. line break) are also implemented in the system.

In TEX, each object to be formatted is called a box. A box can be a character, a word, a table, a graph, or even consist of a number of other boxes. The system need only concern itself with the information carried by a particular box, such as width or height. The spacing between boxes (i.e. between lines, between words, etc.) is called "glue". Glue acts like a spring: it can be stretched, compressed or maintained at its normal width in order to fit the justification requirements [Knuth 84]. The box and glue concept turned out to be a very

useful tool. It simplified the formatting problem and allowed for the development of more efficient algorithms. This concept has been adapted to other formatting systems such as ETUDE [Hammer et al 81], JANUS [Chamberlin et al 81] and YALE'S PEN [Allen 81].

One of the major factors which determines the physically attractive output of the TEX system is the routine which breaks each paragraph into lines. In fact, TEX's designers developed one of the best algorithms, known in this area. According to Knuth, this algorithm is so powerful and sophisticated, it can do a better job than a skilled typesetter [Knuth and Plass 81].

Briefly, the line breaking algorithm uses discrete dynamic programming techniques to build a linked list network. Consider the network as a directed graph with a relative undesirability value as arc value and each feasible break point as vertex. The question of the best way to break a particular paragraph is then equivalent to the question of finding the shortest path in the graph. The difference between the linked list network and the graph is that, in the network model, each node (vertex in graph) remembers its parent and total undesirability from the beginning up to this node. The undesirability value is assigned according to the amount of adjustments the glue has to make, i.e. the extent it has to stretch or compress. If the glue has to adjust a lot, then the undesirability value will be large; since the undesirability value and the amount of adjustments vary in the same direction. Another factor which affects the undesirability value is the penalty item [Knuth and Plass 81]. A penalty item is a value the user can assign manually if a line break occurs at a particular point. For example, if the user

wants a line break to occur at a certain point, he can assign a value of negative infinity to the penalty value (i.e. forced break). Similarly, if he does not want a break at that point, he can assign a positive infinity as penalty value. A penalty value can also be assigned by the system. For example, if a line must end half-way through the word, the system will insert the hyphenation. Since the rule is to have as little hyphenation as possible, a penalty value will also be assigned at that break point by the system. For information on other factors which cause the insertion of penalty values, consult the details in the references [Knuth and Plass 81].

One of the only apparent drawbacks to TEX is in the user interface. To learn how to use the system well involves a considerable amount of time and memorization (even though there are macro facilities in TEX). The materials TEX deals with involve a lot of mathematical and tabular items. These two fields are usually difficult to represent by simple formatting commands. As a result there are more than 300 predefined control sequences and 6 operation modes in TEX [Knuth 84]. Another reason for the enormous amount of control sequences and operation modes is that the designers of TEX wanted to make the formatting commands as general as possible. Generality increases the complexity of designing the commands.

2.1.5 NROFF/TROFF

NROFF ("en-roff") and TROFF ("tee-roff") are the basic formatting systems on the UNIX time-sharing system [Bell Laboratories 83]. In order to understand the UNIX documentation system better, the main ideas behind the UNIX

time-sharing system will be discussed briefly. The designers of the UNIX time-sharing began with the assumption that problem solving would be facilitated by breaking a major problem into several subproblems. The process of subdividing could be repeated until the resulting subproblems were very simple and straightforward. Since these more simple problems were basic to many problems, the same methods could be used to solve a number of larger problems. Thus UNIX evolved the important idea of building programs on top of other programs. For the idea of problem solving by repeated problem decomposition to be worth considering, the UNIX time-sharing system had to provide some means for reconstructing the solutions of the many sub-problems in order to produce the final solution. The mechanism that combines all the smaller solutions into one is called the "pipe" in UNIX.

NROFF and TROFF are basically the same program and will accept the same input description language. The reason for having two systems is that their output devices are different. NROFF is designed for typewriter-like devices and TROFF is designed for phototypesetter-like devices. They accept most of the same input language although NROFF ignores TROFF commands which it cannot honour (e.g. change font size). Though this discussion is based on TROFF, most of the comments apply to NROFF as well.

TROFF is the most difficult formatting language discussed thus far. It is a very low-level language, and some of the layout commands are so tedious and hard to remember that they are not intended for human use. Despite the difficulty and complexity, TROFF is still being used because it provides all the

tools (formatting commands) needed to format almost any kind of document.

Since TROFF is difficult to use, there are several macro packages created to help the user. They are the "ms", "mm" and "me" etc. macro packages. All macro packages are intended for user friendliness. They allow the user to describe the document in terms of its logical part rather than in terms of its physical layout details. For example, using a macro package, a user can describe title, heading, subheading, paragraph, author, etc., instead of specifying the spacing, font type and font size of the document. Moreover, the main ideas (solving the problem by small parts) of UNIX time-sharing system also apply when such problems as formatting tables and mathematical materials appear. Several individual programs are created to handle different problems, for example "eqn" for mathematical materials and "tbl" for tables. The UNIX "pipe" mechanism is used to combine all the partial solutions. We call this type of program "pre-processors" of TROFF.

In summary, the UNIX documentation system is a very successful one. It is also the most powerful formatting system known. The macro packages make it user friendly and its "pipe" mechanism makes it flexible enough to change or adjust to new demands (For instance, new pre-processor can be created).

TROFF's major drawback is that its users have to learn several formatting languages instead of one. In addition, they also have to learn how to use the UNIX time-sharing system, which is quite unfriendly to new-comers. The mixing of macro commands with TROFF commands also creates some potential formatting errors for its users. If several pre-processors are involved, formatting time

can be lengthy, since it means several passes of the input documents.

2.2 Declarative Approach Systems

Declarative approach systems are based on the assumption that most of their users will have no time or interest in specifying details of the formatting of their documents. In addition, it is assumed that users will be satisfied as long as a certain quality and formatting standards are guaranteed. Normally, a user of these kinds of systems will only be required to define the type of document and specify its structure (here, structure means the logical meaning of the text, e.g. paragraph, heading, footnote). The system will then produce the finished formatted product on a specified output device. In contrast to the procedural approach, the system, rather than the user is the designer of the ultimate appearance and style of the formatted document. As a result, the system is fully responsible for the quality of the finished product.

Declarative approaches are being used in text formatting systems such as SCRIBE, GML, ETUDE, JANUS, XEROX's BRAVO and ANDRA. In the following sections SCRIBE, GML and JANUS will be briefly discussed. SCRIBE and GML were selected since they are the only two systems completed and currently in use (at the time of this survey). The others are still under development. JANUS is worth mentioning because it is a system which is being built on top of GML.

2.2.1 SCRIBE

SCRIBE was developed by Brian K. Reid at Carnegie-Mellon University in the late 1970's [Reid 80]. In SCRIBE, the formatting actions are invoked by declaring the formatting environment. For example, if a word is to be printed in italic font, the current environment is changed to an italic environment through formatting command "@i" [Reid 81]. (The symbol "@" is used to indicate a command to the formatter)

Ideally, users of the SCRIBE system only have to specify the type and the logical structure of the document, and the system will do the layout by following predefined environment commands in the SCRIBE database. This lets the system do the tedious layout work. If the users are not satisfied with the result, SCRIBE allows them to create new environment commands (new ways to format the object) and to change the formatting action of existing environment commands (modify some of the formatting actions). The created environment command is denoted as "@define.....", and the command to modify the existing environment is denoted as "@modify.....". Moreover, the changes can apply to a local region of text by using "@begin" and "@end" to mark the boundaries of the region for which the commands are valid.

Generally speaking, the number of different types of documents that can be stored in the SCRIBE database is quite limited and the style of formatting a particular text type is based on common practice. Formatting in low-level layout commands is more difficult than formatting in high-level commands. However, if a document type is not in the SCRIBE database, users will be in a situation simi-

lar to low-level language formatting. They will be required to define a new environment and then use the environment attributes to make it work. If they are not satisfied with the style of any existing text type, the degree of difficulty involved in alteration will depend on the amount of changes required to convert from one style to another. Sometimes procedural systems are even more complicated to use than low-level systems. Another drawback of SCRIBE in particular is that it can only handle simple text type materials. Features such as mathematical and table formatting are not available. Despite all the disadvantages, SCRIBE's mechanism is easy to use. It also provides flexibility when the environment or style has to be changed or created. The simple formatting mechanism provides machine independence (portability) and a uniform formatting style. Last but not least, the writer's workbench features in SCRIBE, such as table of contents, indexing, section numbering, cross references, bibliography management facility, are very helpful for its users.

In summary, SCRIBE is a very successful text formatting system. It provides an alternative to users who do not want to spend much time for detailing physical layout.

2.2.2 GML (Generalized Markup Language)

GML was developed by C.F. Goldfarb in the late 1970's [Goldfarb 81]. It is part of the IBM document composition facility [Goldfarb 80]. Like SCRIBE, GML is a text formatting system using the declarative approach. It is based on IBM's SCRIPT formatter (SCRIPT is a RUNOFF-like formatter developed in the late 1960's). Like SCRIBE, GML has some pre-defined document types. Unlike

SCRIBE, however, GML's major concern is to provide methods to deal with simple (pure text) document types. The way GML deals with a general document is quite appealing. It classifies all possible document elements into different categories, empty categories to be disregarded in any particular document. Figure 2.1a and 2.1b [Goldfarb 80] shows the document categories GML includes in its language.

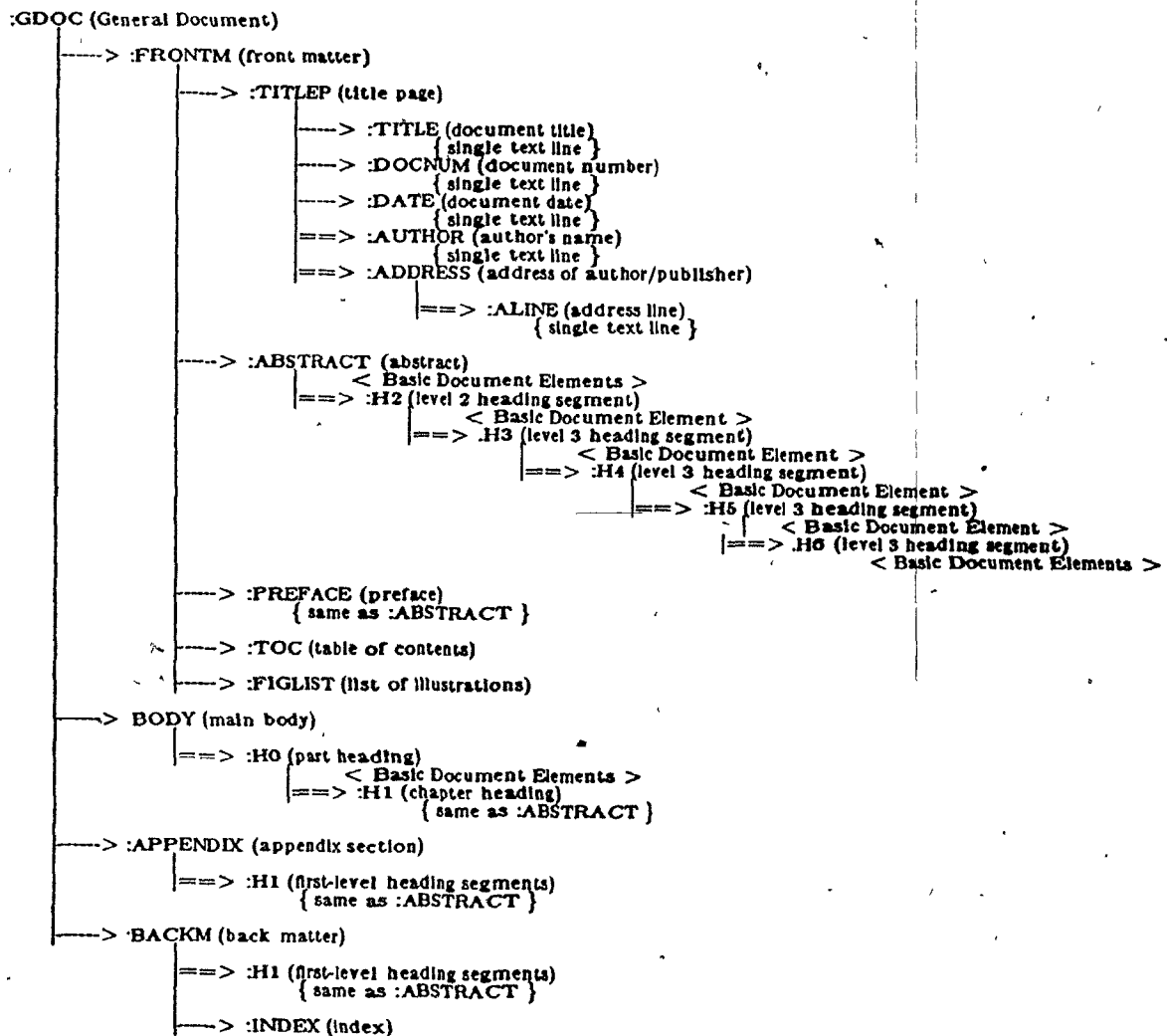


Figure 2.1a Overall structure of a General Document

< Basic Document Elements >

```

==> :ADDRESS (address)
    ==> :ALINE (address line)
        { single text line }

==> :DL (definition list)
    ==> :DT (definition term)
        { single text line }
    -----> :DD (definition description)
        -----> < Implied paragraph >
        -----> < Basic Document Elements >
    ==> :LP (list part)
        -----> < Implied paragraph >
        -----> < Basic Document Elements >

==> :XMP (example)
    -----> < Basic Document Elements >
    -----> < line elements >
        { normal text }

==> < extended paragraph >
    -----> < P (paragraph) or Implied paragraph >
        { normal text }
    -----> < Basic Document Elements >
    ==> :PC (paragraph continuation)
        { normal text }

==> :FIG (figure)
    -----> < figure body >
        -----> < Basic Document Elements >
        -----> < line elements >
            { normal text }
    -----> :FIGCAP (figure caption)
        { single text line }
    -----> :FIGDESC (figure description)
        -----> < Implied paragraph >
        -----> < Basic Document Elements >

==> :OL, :UL, :SL (ordered, unordered and simple lists)
    ==> :LI (list item)
        -----> < Implied paragraph >
        -----> < Basic Document Elements >
    ==> :LP (list part)
        -----> < Implied paragraph >
        -----> < Basic Document Elements >

==> :NOTE (note)
    < Implied paragraph >

==> :P (paragraph)
    { normal text }
    ==> :Q (quote)
        { normal text }
    ==> :HP0 (highlight type 1)
        { normal text }
    ==> :HP1 (highlight type 2)
        { normal text }
    ==> :HP2 (highlight type 3)
        { normal text }
    ==> :HP3 (highlight type 4)
        { normal text }

==> :LQ (long quote)
    < Basic Document Elements >
  
```

Figure 2.1b Basic Document Elements

In the above figures, some tags indicate structure rather than text. For instance, GDOC (general document) indicates the basic structure of the entire document, beginning with front matter and concluding with back matter. Only tags containing "single text line", "normal text", or "implied paragraph" may contain text (here "text" means a sequence of words). The tags in figure 2.1a are ordered according to a general document format. For example, "title page" always appears before the abstract, the abstract before the preface, and so on. The user must enter tags according to document sequence and all tags can be specified only after the parent tag is specified, eg., document type "FRONTM" can only be specified after tag "GDOC" is specified. No ordering is imposed on basic document elements (figure 2.1b), since paragraphs, quotes, etc., may appear a number of times within a document and in any order. A single arrow (--->) indicates those tags which can only be specified once. For instance, "Appendix" can only appear once in a document. A double arrow (==>) implies that the tag can be repeated, eg: within the appendix structure, H1 (first level heading) can appear any number of times.

Following, are two examples (taken from [Goldfarb 80]) used to illustrate how the GML tags work:

Example A

:lq
:p. Four score and seven years
ago our fathers brought forth on
this continent a new nation ...
:elq

formatted output will be following:

Four score and seven years ago our
fathers brought forth on this con-
tinent a new nation ...

Example B

:q.The observation :q.The
coldest winter I ever
experienced was one summer
in San Francisco:eq. is widely
attributed to Sam Clemens:eq..

formatted output will be following:

"The observation 'The coldest win-
ter I ever experienced was one sum-
mer in San Francisco' is widely
attributed to Sam Clemens."

Another goal of GML is to ensure that any kind of document need only be marked up once. In other words, the document is no longer limited to a single application, formatting style or processing system. If a user has already

identified the document elements properly, there is no reason for him to repeat the markup processes in order to achieve a different effect. GML made this possible by the use of profiles prepared by APFs (application processing functions). A profile is a mapping mechanism such that the same markup tag can have different group of low-level layout commands (SCRIPT commands).

Generally speaking, GML is less complicated than SCRIBBE but has all the advantages of the latter system. For example, it is easy to use and has a powerful writer's work bench. On the other hand, GML also has the same basic disadvantage as SCRIBE, in that it can only handle simple forms of text documents. It cannot handle mathematical materials and simple line drawings.

2.2.3 JANUS

JANUS is an interactive formatter under development at the IBM research laboratory in San Jose [Nievergelt 82]. While based on GML's concepts, the system has some other interesting features. JANUS displays formatting results and corresponding input text on two different screens. The formatted output is displayed one page at a time. If users are not satisfied they can use the joystick to rearrange the objects in the formatted output. Once satisfied, they can then get the hardcopy from the phototypesetter device. The interactive aspect of JANUS greatly simplifies the reformatting process for the users. All batch formatters mentioned earlier require users to fix the original input text and reformat it.

Another of JANUS' interesting features is its internal representation of documents. Documents are specified as a collection of galley. A galley member can be

a table, a justified line, a picture or any other element ready to be put onto the page. The only restriction on galley members is that they be indivisible and have an ordered sequence in relation to other galley members. The idea of galleys simplifies the formatting process, because it breaks the formatting problem into two smaller problems: a horizontal and a vertical problem. The horizontal problem deals with the page width. Its main concern is with dividing texts into galley members to fit the page width. The vertical problem deals with the page height. Its main concern is with arranging the line galley members into pages. Another advantage of the galley concept is that it is easy to mix tables and figures, because the system can treat them like any ordinary galley member with different heights and widths.

Questions remain as to how JANUS will deal with the changes effected through the use of the joystick. The basic question is whether the system must change the original input document file or create a new document file. While JANUS has a processing power equal to GML, it has many advantages over the latter in terms of user friendliness.

2.3 Conclusion

Through the evolution stages of text formatting systems, there has been an obvious switch from low-level (procedural) approach systems to high-level (declarative) approach systems. High-level systems are easier to learn and use: syntax errors are less likely to occur, thus reducing both cost and time in producing the document. In terms of updating, high-level systems are less costly, since they are easier to reformat than low-level systems. Different formatting styles can easily be achieved in high-level approach systems by simply changing the markup tag definition. While in low-level approach systems, users have to redo the whole document. Since high-level systems have so many advantages over low-level systems, it is expected that in the near future more sophisticated high-level approach systems will be developed to satisfy the needs of users.

JANUS is the prototype of the next generation of formatting systems. Although the idea of interactive text formatting is fascinating, the problems discussed above remain open.

Finally, there are still problems as how to enable sophisticated high-level approach systems such as TROFF or TEX, to deal with tables and mathematical materials without losing their declarative characteristics.

After having evaluated the above three approaches it was decided that a high-level approach would be used for the experimental formatting system. As has been discussed, the high-level system is more user friendly than a low-level system, but does not have the "ad hoc" problems experienced in interactive formatting.

CHAPTER THREE

System Design

This chapter will discuss the requirements set for the experimental system, as well as the formatting language used in the system. Following this will be a short discussion of the basic design of the system.

3.1 Basic System Requirements

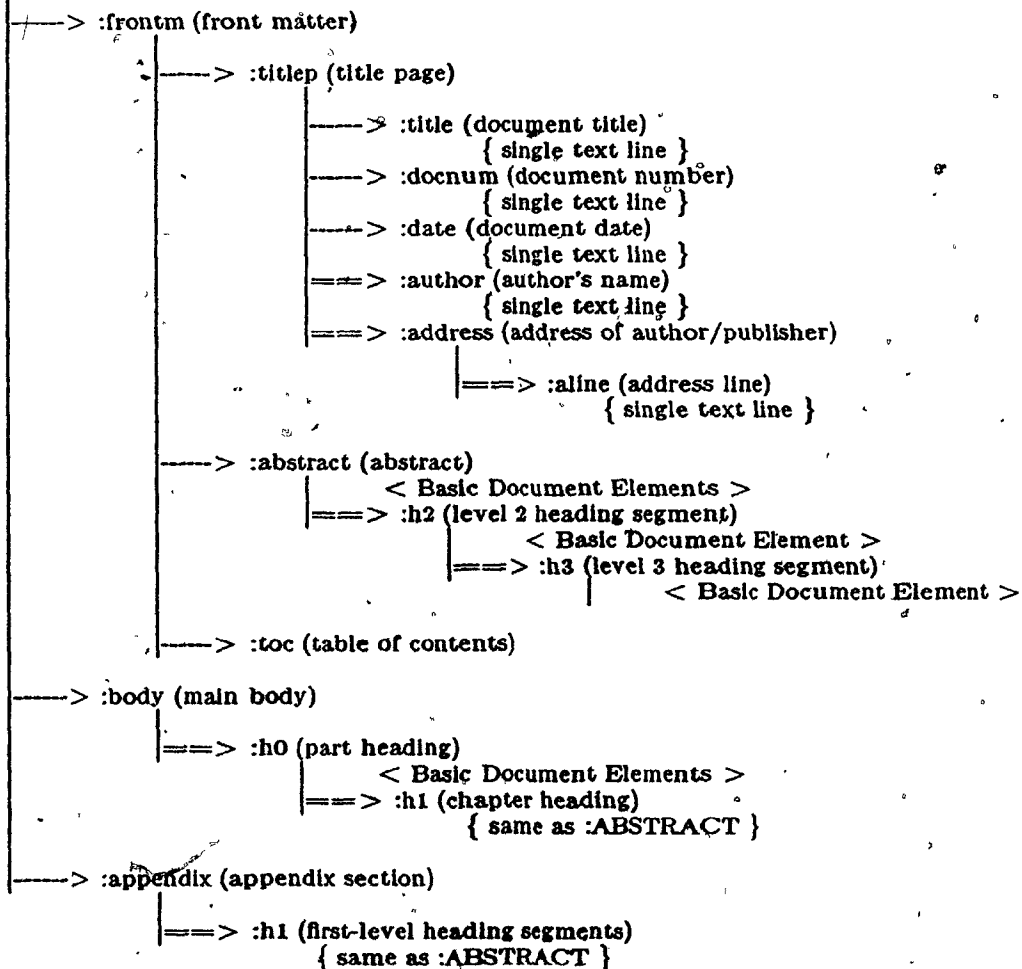
The major goal of the experimental formatting system was to evaluate whether a text formatting system based on a relational model was flexible or not. The prime requirement of the system was to be able to deal with relational files without violating the closure properties [Merrett 84a] of algebraic systems. The user should be able to use it not only to format text but also to take advantage of the relational model, using relational algebra to solve some other text processing problems such as indexing, spelling correction, and word frequency count.

After assessing the procedural and declarative methods, it was found that the declarative approach was more suitable to the experimental system. Formatting declarative commands is less complex and more systematic than formatting procedural commands. This makes implementation of a relational based formatting system easier. Finally, the flexibility and user friendliness of the declarative approach also makes it more appealing than the procedural method.

3.2 Formatting Language

Due to time constraints, and since the major concern was not how to design a declarative formatting language, it was decided to adopt a subset of GML's general document definition. GML's formatting language was chosen because it is well defined and is easy to extend to the complete set in the future. The subset of GML's general document language was adopted into the experimental system as shown in figure 3.1:

:gdoc (General Document)



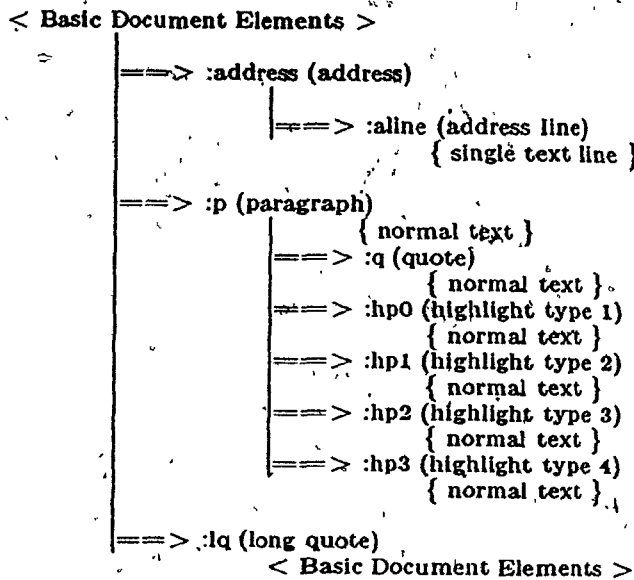


Figure 3.1 Subset of GML Formatting Definition

note: the same rules apply as in the discussion of the GML system in section 2.2.2

3.3 Logical Design of the System

The experimental relational text formatting system is divided into three major components. These are: editor, formatter, and display driver, as shown in figure 3.2.

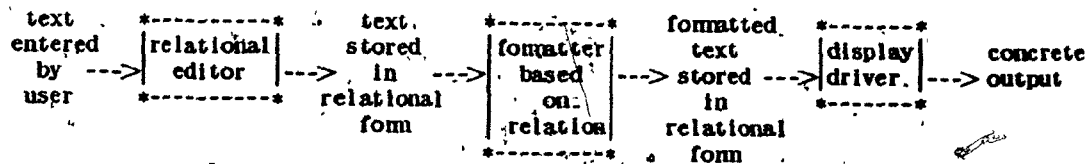


Figure 3.2 Relational Text Formatting System

3.3.1 Editor Based on Relations

The relational editor is somewhat different from the normal editor. Besides providing normal editing functions, it also provides a mapping function between relations and normal text files (a string of characters). The mapping function is essential, since the editor actually stores the input text in two relations ("STRUC" and "COPY"). Another distinguishing character of this editor is that it is syntax-directed. The syntax of the formatting commands of high-level systems is simple and systematic and eliminates unnecessary formatting errors. This editor can also use "template" techniques [Telteibaum 81a]. The "template" method eliminates typing errors and is more user-friendly than other methods, since the user has simply to jump into the right place-hole and start typing without actually typing the formatting commands. The editor also monitors the whole editing process and thus minimizes the amount of work for the user. Since it is not the major concern of this thesis, readers who are interested in the relational editor should refer to [Fayerman 84].

3.3.2 STRUC and COPY Relations

As explained in chapter one, section 1.4, a relation 'FIRST' composed of two attributes: 'word' and 'wordseq' (word sequence), would be able to preserve the information and support all the functions (updating, viewing, storing etc.) as the conventional method. However, relation 'FIRST' is not the optimum way to store text in the relational form. 'FIRST' does not take advantage of the possibility of storing text according to its logical structure. If one more attribute, 'type' (text type) is added, the type of text elements can also be stored.

e.g. SECOND (type , wordseq , word)

```

      :
      :
p      12.0      This
p      13.0      is
p      14.0      an
p      15.0      example.
      :
      :

```

(note : In section 3.2 it was mentioned that GML formatting language was used as the formatting language for the experimental system. Therefore in the examples GML text element tag will be used as the text type (i.e. p --> paragraph).)

Since the major goal of the system is formatting text, relation 'SECOND', while an improvement on relation 'FIRST', is still not good enough. Problems such as distinguishing one paragraph from another, or knowing that a paragraph belongs to one part of a long quote, are solved by adding three more attributes, namely: 'ptag' (parent text type), 'p_id' (parent text type id) and 'c_id' (child text type id). Since the GML tags are being used in the example, 'ptag' or 'ctag' are used as the names of the text type attributes. The relation proposed would look like the following example:

THIRD (ptag , p_id , ctag , c_id , wordseq , word)

```

      :
      :
lq      6      p      10      0      This
lq      6      p      10      1      is
lq      6      p      10      2      an
lq      6      p      10      3      example.
      :
      :

```

Relation 'THIRD' contains all the information needed. However, repetition of the same data value (in ptag, p_id, ctag and c_id) caused it to be rather

cumbersome. It was decided to break relation 'THIRD' into two relations: 'STRUC' and 'COPY', which are the input relations (relations we mentioned in section 3.3.1) of the experimenting formatter. The STRUC relation, as the name indicates, is used to store the logical structure information of the input document.

The STRUC relation has the following attributes:

- a) ptag - parent GML tag
- b) pseq - parent tag-id
- c) ctag - child GML tag
- d) cseq - child tag-id

The tag-id is an unique value in ascending order, which serves two purposes:

- i). It is used to distinguish the document elements with the same GML tag and
- ii). It is used to preserve the natural ordering of the document elements; e.g. the element with smaller tag-id should be placed before the element with bigger tag-id.

The copy relation is used to store the actual document. It has the following attributes:

- a) ctag - GML tag
- b) cseq - tag-id
- c) wordseq - word-id
- d) word - actual word

The linkage of STRUC and COPY is made by the attribute CSEQ. For example, the beginning of this section will be stored as the following relation segments:

STRUC (PTAG, PSEQ, CTAG, CSEQ)

h1	10	h2	18
h2	18	p	19

COPY (CTAG, CSEQ, WORDSEQ, WORD)

h2	18	0	Logical
h2	18	1	Design
h2	18	2	of
h2	18	3	the
h2	18	4	System
p	19	0	The
p	19	1	experimental
p	19	2	relational

note :- sequence number was chosen arbitrarily

3.3.3 Formatter Based on the Relations : COPY and STRUC

The relational formatter is further broken down into the following major parts as shown in Figure 3.3

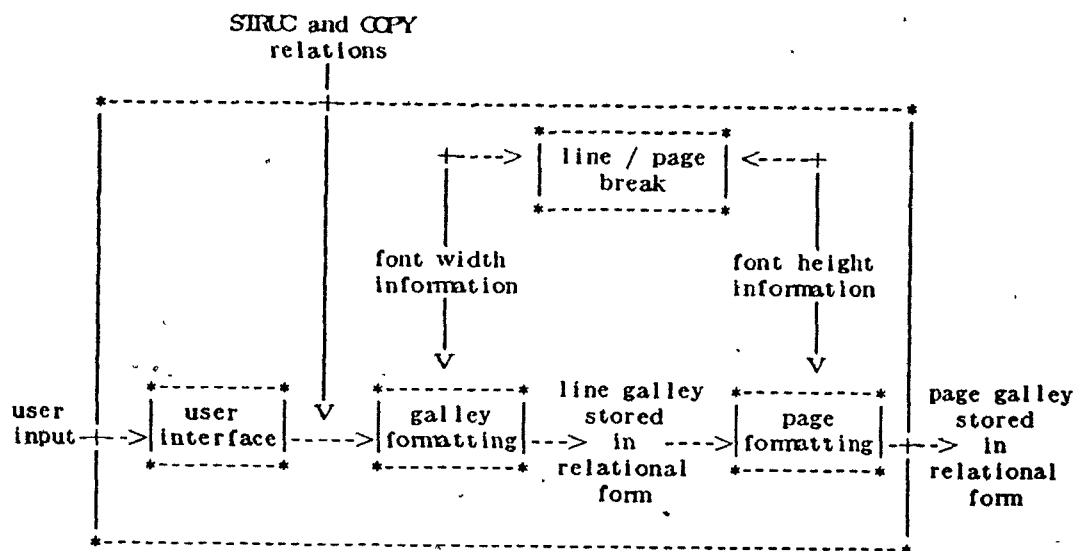


Figure 3.3 Major Components of the Formatter

3.3.3.1 User Interface

The user interface part was designed to allow the user to modify the formatting options and is the only communication bridge between the user and the formatter. The options the user may change are: page size, font size, formatting rules, document type, formatting style, and output device. In order to satisfy different needs and to make the system as flexible as possible, a "switch board" was used. Each GML formatting command has a set of formatting rules. The user has the choice of using the default option or selecting the one he/she needs. The default option is decided according to the document type specified by the user. The interface routine is designed with the novice user in mind and many options are involved. It was decided to use the "template" method instead of the conventional line-oriented method or menu driven type technique. Ideally, the user has a full page of options with the default setting. Options are ordered according to their natural dependence. For example, since document type will decide the default choice of the switch board, the option of document type should go before the option of switch board. The interface mechanism works in the following fashion: If the user is satisfied with the default settings, he/she can jump to the next page of options by simply hitting the return key. Otherwise, he/she can jump to the place-hole and make changes. This method makes the user interface part extremely fast and user friendly.

3.3.3.2 Line/Page Break Module

One of the major tasks of text formatting is to break down paragraphs into lines and then into a "rigid area" with left and right margins being aligned. This "breaking process" is essential for any text longer than the available line width. The effectiveness of this process has a tremendous impact on the appearance and quality of the finished document. Since a poorly broken document will distract the attention of the reader from the document, it is clear that the breaking routine is vital for the development of a document preparation system. The possible solutions to this problem are the following :

1). line-by-line approach

The conventional way to solve the line breaking problem is by assigning one word at a time to the current line. The word is accepted if the additional word width does not exceed the preassigned line width. Otherwise, this word is removed from the current line and placed at the beginning of the next following line. The whole process keeps on until the last word of the paragraph. To summarize the line by line approach we present the following algorithm.

LINE BY LINE ALGORITHM

Input

n = total number of words in the paragraph.

size = desired line width.

word = array used to store each word width.

output

line = array used to store the break points.

line[0] = total number of lines generated.

time complexity

$O(n)$

Step 1 /* Initialize */

l := 1; line[0] := 1; width := word[l];

Step 2 /* calculate line width */

while ((width < size) and (l <= n)) do

begin

l := l + 1;

width := width + word[l] ~~+ 1;~~

end;

If (width > size)

then goto 3

else If (l > n)

then

begin

line[0] := l - 1;

terminate;

end

else

begin

l := l + 1;

goto 3;

end;

Step 3 /* need a new line */

line[line[0]] := l - 1;

width := word[l];

line[0] := line[0] + 1;

goto 2;

Example 1

The paragraph below is for demonstration purposes only:

"The conventional way to solve the line breaking problem is by assigning one word at a time to the current line."

The above paragraph is translated into an array "word" which is the input to the line by line algorithm. Array "word" contains the width of each word in the paragraph.

Input to the algorithm are :-

word = 3,12,3,2,5,3,4,8,7,2,2,9,3,4,2,1,4,2,3,7,5

n = 21

size = 35

The algorithm performs the following calculation.

i = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21
word = 3, 12, 3, 2, 5, 3, 4, 8, 7, 2, 2, 9, 3, 4, 2, 1, 4, 2, 3, 7, 5
width = 3, 16, 20, 23, 29, 33, 38
 4, 13, 21, 24, 27, 37
 9, 13, 18, 21, 23, 28, 31, 35
 7, 13

The output array 'line' has the following values.

```
line[0] = 4      /* total number of lines */
line[1] = 6      /******
line[2] = 11     /*      last word of      */
line[3] = 19     /*      each line      */
line[4] = 21     /******
```


Now the production of the formatted output is trivial. Another routine can be used which takes the contents of array 'line' as its input and produces the following output with left and right justification and distributes the extra spaces in a random manner:

The conventional way to solve the line breaking problem is by assigning one word at a time to the current line.

2). dynamic programming approach

The next possible choice for the line breaking problem is to use the so-called dynamic programming approach [Sedgewick 83]. When a sequence of words has to be broken into two or more lines, the dynamic programming approach can help to break the lines in such a way that they are equally used up or very nearly so. The concept is simple : all possible permutations and combinations of ending words for every line are calculated and the results stored in a table (cost matrix). Once all possible solutions are found it is simple to determine the best way to break the lines (according to the lowest values in the cost matrix). The cost value for each possible ending word is strictly a matter of personal choice. For example, if the user dislikes hyphenation he/she can assign a large cost value for it. The following algorithm summarizes the dynamic programming approach:

DYNAMIC PROGRAMMING ALGORITHM

Input

n = total number of words in the paragraph.

size = desired line width.

word = array used to store each word width.

output

end_word = array used to store the optimal break points.

time complexity

The time complexity involved in step 1 is n^2 . However, the rest of the algorithm requires a time complexity of order n .

Step 1 /* Initialize */

cost[1,j] := ∞ ; $1 = j = 0, 1, \dots, n$

$i := 1$; $j := 1$;

top := 0;

prewidth := size;

precost := 0;

width := word[i];

Step 2 /* test finish all the words or not */

If $j = n$

then

begin

If cost[i,j] > precost

then cost[i,j] := precost;

If not emptystack

then

begin

width := word[stack[top].i];

$i := \text{stack}[\text{top}].i$;

$j := i$;

prewidth := stack[top].width;

precost := stack[top].cost;

top := top - 1;

goto 3

end

else goto 6

end

else goto 3;

Step 3 /* calculate current line width */

```
J := J + 1;  
width := width + 1 + word[J];  
If size < width  
    then goto 4  
else If (size - width) <= (size / 5)  
    then goto 5  
    else goto 2;
```

Step 4 /* calculate cost value */

```
If cost[l,J-1] = ∞  
    then  
        begin  
            width := width - 1 - word[J];  
            cost[l,J-1] := (size - width)3 + precost + abs(width - prewidth)3;  
            top := top + 1;  
            stack[top].l := J;  
            stack[top].cost := cost;  
            stack[top].width := width;  
        end  
        width := word[stack[top].l];  
        l := stack[top].l;  
        J := 1;  
        prewidth := stack[top].width;  
        precost := stack[top].cost;  
        top := top - 1;  
        goto 2;
```

Step 5 /* new line */

```
cost := (size - width)3 + precost + abs(width - prewidth)3;  
If cost < cost[l,J]  
    then  
        begin  
            cost[l,J] := cost;  
            top := top + 1;  
            stack[top].l := J + 1;  
            stack[top].cost := cost;  
            stack[top].width := width;  
        end  
        goto 2;
```

Step 6 /* find best sequence of break points */

```
<use the cost matrix to find the optimal break point>;  
terminate.
```

In the above algorithm the "cost" is derived from a function which itself is defined as:

$d1$ = difference between desired line width and actual line width

$d2$ = difference between current line width and previous line width

$$\text{cost} = (d1^2 + d2^2)$$

Example 2

With the same input as in example 1, step 1 through step 5 of the algorithm will produce a two dimensional matrix that is given below (note : blanks in the cost matrix represent infinity and only the smallest cost values will be stored):

i \ j	1	..	5	6	...	10	11	...	17	18	19	..	21
1			432	16									
6						776	504						
7							744						
11									867	993			
12									874	568	568		
18												867	
19												568	
20												568	
21													

i = starting word of the current line

j = ending word of the current line

Cost Matrix

In order to produce the above matrix, the algorithm will do the following calculations.

j = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21
word = 3, 12, 3, 2, 5, 3, 4, 8, 7, 2, 2, 9, 3, 4, 2, 1, 4, 2, 3, 7, 5
width = 3, 16, 20, 23, 29, 33, 38 prewidth = 35 precost

					0
					16
					744
				7, 13	1256
				3, 11, 17	872
				2, 6, 14, 20	1088
					432
					504
				7, 13	568
				3, 11, 17	568
				2, 6, 14, 20	874
					776
				3, 11, 17	993
				2, 6, 14, 20	867

Stack used in the calculation.

19	993	34	13	12
18	867	31	12	13
20	568	35	11	8
19	568	31	10	9
18	874	28	9	10
12	504	31	8	7
11	776	28	7	11
20	1256	35	6	3
19	872	31	5	4
18	1088	28	4	5
12	744	27	3	2
7	16	33	2	1
6	432	29	1	6

i	cost	width	push	pop
---	------	-------	------	-----

Each row of the matrix indicates the starting word number of a particular line, whereas the column indicates the ending word number of that line. Since the total number of words "n" in the paragraph is known, the minimum cost is found from the n^{th} column. The corresponding row indicates the starting word number of the last line. The preceding line's ending word number is equal to the current row's starting word number - 1. The same process is then repeated until the first line is reached.

A routine accepting the "cost" matrix as input can be used to produce the following formatted output:

The conventional way to solve the line breaking problem is by assigning one word at a time to the current line.

By comparing this with the output from the first algorithm, it is clear that the dynamic programming has produced a better result.

3). The Heuristic Approach [Knuth and Plass 81]

The line breaking algorithm developed by Knuth and Plass is also based on the dynamic programming approach. The basic idea of the algorithm is the same as the dynamic programming approach : calculate the cost value of each feasible ending word (break point); store the cost values; when all the feasible ending words are calculated, pick the best sequence of the ending words according to the pre-stored cost values. However, the algorithm developed by Knuth and Plass is

somewhat more complex than the dynamic programming approach. Their algorithm eliminates the most unlikely combinations, such as extremely loose or tight lines. It also discourages "widow lines" (lines with only one word). Rather than using a matrix to store the cost value, the system uses a network structure, which avoids the unnecessary waste of memory space. Finally, their system gives the user more control by allowing him/her to add additional information (control factors) based on personal choice. For example, the user can assign a larger penalty value for undesirable break points.

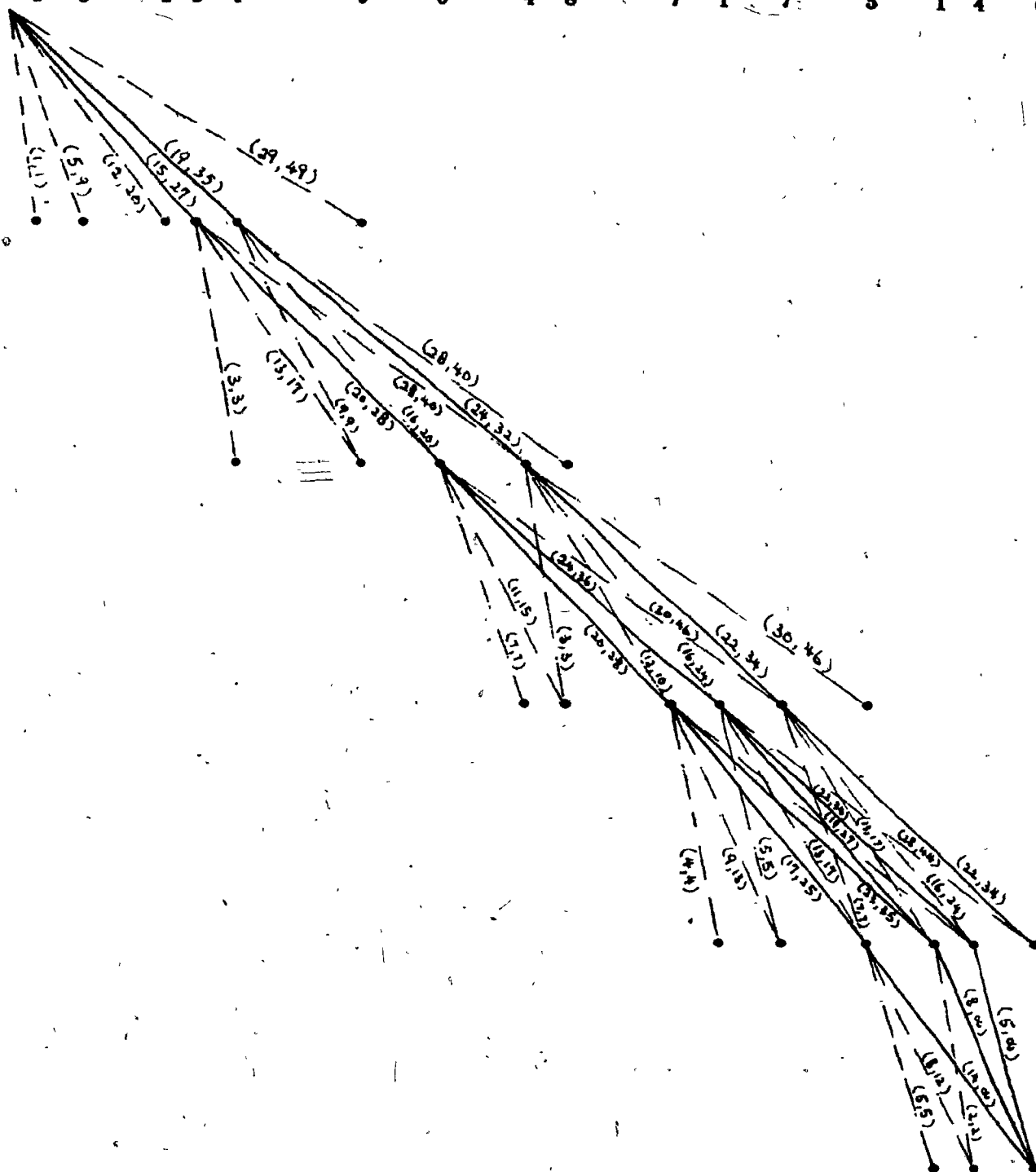
The following example will illustrate how the line breaking algorithm works. For simplicity's sake the example will assume the following conditions:

- 1) no automatic hyphenation involved
- 2) all character widths are equal to 1
- 3) maximum blanks allowed (between words) are four.
- 4) desired line length is 25

with the following input paragraph

"I was coming to the Computer Centre because the computer was being comical about my code."

		1	1	1		2		3		4	4		5	6	6		7		8	8		9	
1	5		2	5	9		9		6		4	8		7	1	7		5		1	4		0



O

Figures 3.4a and 3.4b should be read from left to right and from top to bottom. In figure 3.4a the first number refers to the actual number of characters and spaces in a line. Since it is possible to insert another four spaces between words, the second number represents the maximum width of the line. For example, one of the feasible lines runs from "I" to "to". The first number at this break point (15) equals the actual number of characters and spaces. The second number (27) equals the first number plus 4 times the total number of interval spaces in that line ($4 \times 3 = 12$). In figure 3.4b the adjustment ratios (r_j) are calculated by the following equation:

$$\text{adjustment ratio } (r_j) \equiv (l_j - L_j) / \sum str$$

where l_j is the desired length; L_j is the actual length; $\sum str$ is the stretchability of line j .

The badness values (β_j) are calculated by the following equations:

$$\text{badness } (\beta_j) \equiv \begin{cases} \infty & \text{if } r_j \text{ undefined or } < -1 \\ 100|r_j|^3 & \text{otherwise} \end{cases}$$

The demerits (δ_j) are calculated by the following equation:

$$\text{demerits } (\delta_j) \equiv \begin{cases} (1+\beta_j+\pi_j)^2 + \alpha_j & \text{if } \pi_j \geq 0 \\ (1+\beta_j)^2 - \pi_j^2 + \alpha_j & \text{if } -\infty < \pi_j < 0 \\ (1+\beta_j)^2 + \alpha_j & \text{if } \pi_j = -\infty \end{cases}$$

where π and α arise from hyphenation

Using the same feasible line example as above (the line running from "I" to "to")

$$r_1 = (25-15) / (3 \times 4) = 0.83$$

$$\beta_1 = 100 |0.83|^3 = 57.9$$

$$\delta_1 = (1+57.9+0)^2 + 0 = 3465.7$$

Finally, to find the best breaking sequence figure 3.4b should be read from the bottom to find the least $\sum \delta_j$ over all lines j and then follow the path backward (right to left; bottom to top) to get all the break points (ending words) for each line.

I was coming to the Computing Centre because the computer was being comical about my code.

1	5	1	1	1	2	3	4	4	5	6	6	7	8	8	9
1	5	2	5	9	9	6	4	8	7	1	7	5	1	4	0

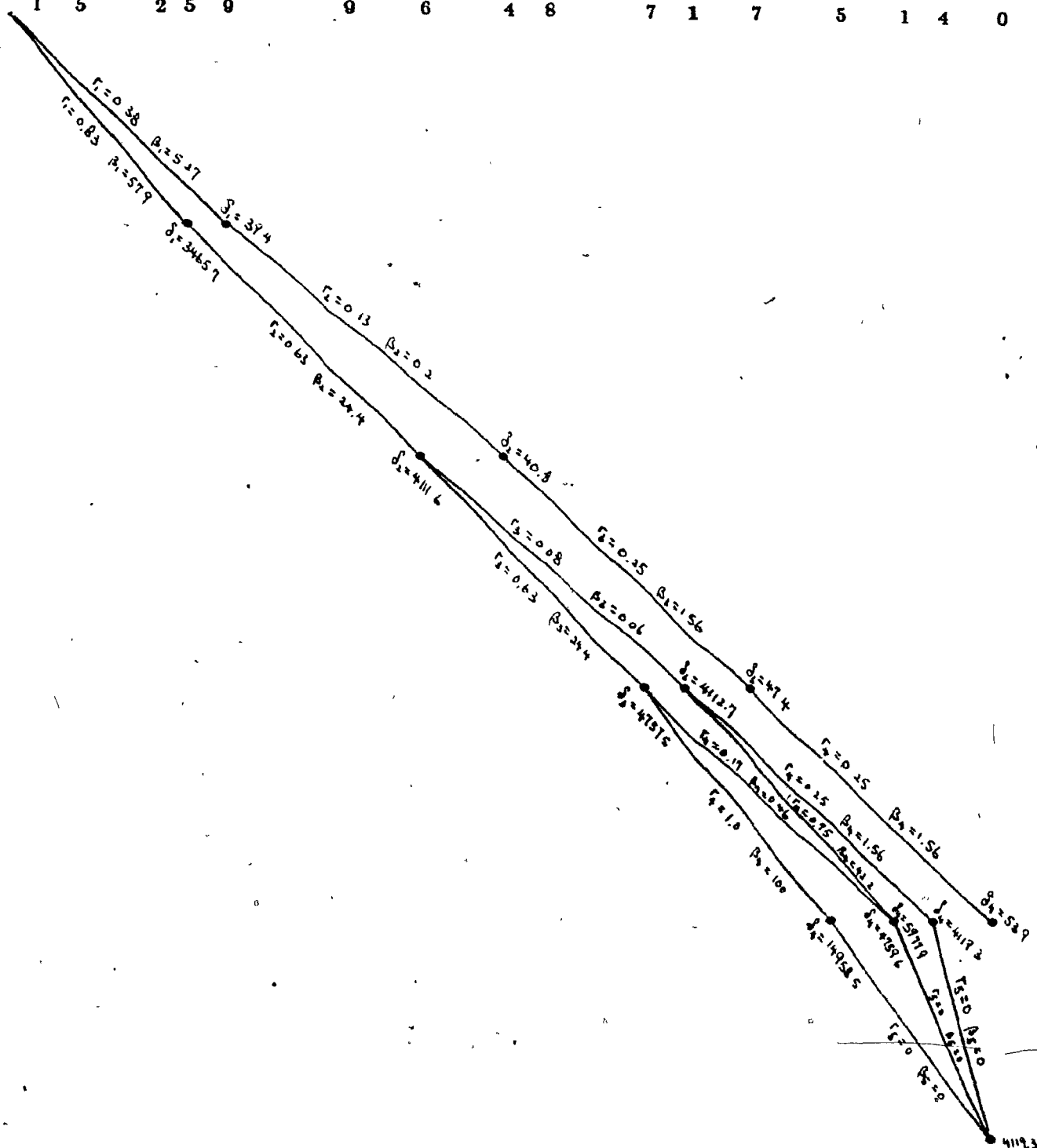


Figure 3.4b: The important information calculated according to ending word (break point). The values are: adjustment ratio (r_j), badness (β_j) and demerits (δ_j).

7
The formatted result (least demerit) will be following:

I was coming to the
Computing Centre because
the computer was being
comical about my code.

The line breaking algorithm developed by Knuth and Plass is as follows:

THE LINE BREAKING ALGORITHM (general outline) [Knuth and Plass 81]

Time complexity

$O(n)$

```
<create an active node representing the beginning of the paragraph>;
for b := 1 to m do
  <If b is a legal breakpoint>
    then
      begin
        <Initialize the feasible breaks at b to the empty set>;
        <for each active node a> do
          begin
            <compute the adjustment ratio r from a to b>;
            If  $r < -1$  or <b is a forced break>
              then <deactive node a>;
            If  $-1 \leq r < p$ 
              then <record a feasible break from a to b>;
          end;
        <If there is a feasible break at b>
          then
            <append the best such breaks as active nodes>;
      end;
  <choose the active node with fewest total demerits>;
  If  $q < 0$ 
    then <choose the appropriate active node>;
  <use the chosen node to determine the optimum breakpoint sequence>
```

<create an active node representing the beginning of the paragraph>

begin

A := new node(position = 0, line = 0, fitness = 1,
totalwidth = 0, totalstretch = 0, totalshrink = 0,
totaldemerits = 0, previous = A, link = A);

P := A;

end.

¶

(for b := 1 to m do <if b is a legal breakpoint> then <main loop>.)

$\Sigma W := \Sigma Y := \Sigma Z := 0;$

for b := 1 to m do

if $t_b = \text{'box'}$ then $\Sigma W := \Sigma W + w_b$

else if $t_b = \text{'glue'}$ then

begin

if $t_{b-1} = \text{'box'}$ then <main loop>;

$\Sigma W := \Sigma W + w_b;$

$\Sigma Y := \Sigma Y + y_b;$

$\Sigma Z := \Sigma Z + z_b;$

end

else if $p_b < +\infty$ then <main loop>.

<main loop>

begin

a := A;

preva := A;

loop

$D_0 := D_1 := D_2 := D_3 := D := +\infty;$

loop

nexta := link(a);

<compute the adjustment ratio r from a to b>;

if $r < -1$ or $p_b = -\infty$

```

    then <deactivate node a>
    else preva := a;
    if -1 <= r <= p
    then
        begin
            <compute demerits d and fitness class c>;
            if d < Dc
            then
                begin
                    Dc := d;
                    Ac := a;
                    if d < D then D := d;
                end;
            end;
        end;
    a := nexta;
    if a = A then exit loop;
    if line(a) >= j and j < j0 then exit loop;
    repeat;
    if D < ∞
    then <insert new active nodes for breaks from Ac to b>;
    if a = A then exit loop;
    repeat;
    if A = A
    then <do something drastic since there is no feasible solution>;
end.

```

<compute the adjustment ratio r from a to b>

```

L := ΣW - totalwidth(a);
if tb = 'penalty' then L := L + wb;
j := line(a) + 1;
if L < lj
then
    begin
        Y := ΣY - totalstretch(a);
        if Y > 0 then r := (lj - L) / Y;
        else r := ∞;
    end
else if L > lj
then
    begin

```

```

    Z :=  $\Sigma Z$  - totalshrink(a);
    if Z > 0
        then r := (Ij - L) / Z
        else r :=  $\infty$ ;
    end
    else r := 0.

```

<deactive node a>

```

begin
    if preva = A
        then A := nexta
        else link(preva) := nexta;
    link(a) := P;
    P := a;
end;

```

<compute demerits d and fitness class c>

```

begin
    if pb >= 0
        then d := (1 + 100|r|3 + pb)2
    else if pb <> -infinite
        then d := (1 + 100|r|3)2 - p2
        else d := (1 + 100|r|3)2;
    d := d +  $\alpha$  * fb * fposition(a);
    if r < -0.5 then c := 0
    else if r <= 0.5 then c := 1
    else if r <= 1 then c := 2 else c := 3;
    if |c - fitness(a)| > 1 then d := d +  $\gamma$ ;
    d := d + totaldemerits(a);
end;

```

<insert new active nodes for breaks from A_c to b >

begin

<compute $tw = (\Sigma w)_{af_{ter}(b)}$, $ty = (\Sigma y)_{af_{ter}(b)}$, and $tz = (\Sigma z)_{af_{ter}(b)}$ >

for $c := 0$ to 3 do

if $D_c \leq D + \gamma$

then

begin

$s := \text{new node}(\text{position} = b, \text{line} = \text{line}(A_c) + 1,$

$\text{fitness} = c, \text{totalwidth} = tw, \text{totalstretch} = ty,$

$\text{totalshrink} = tz, \text{totaldemerits} = D_c,$

$\text{previous} = A_c, \text{link} = a);$

if $\text{preva} = A$

then $A = d$

else $\text{link}(\text{preva}) := s;$

$\text{preva} := s;$

end;

end;

<compute $tw = (\Sigma w)_{af_{ter}(b)}$, $ty = (\Sigma y)_{af_{ter}(b)}$, and $tz = (\Sigma z)_{af_{ter}(b)}$ >

begin

$tw := \Sigma W; ty := \Sigma Y; tz := \Sigma Z; l := b;$

loop

if $l > m$ then exit loop;

if $t_i = \text{'box'}$ then exit loop;

if $t_i = \text{'glue'}$

then

begin

$tw := tw + w_i;$

$ty := ty + y_i;$

$tz := tz + z_i;$

end

else if $p_i = -\infty$ and $l > b$

then exit loop;

$l := l + 1;$

repeat;

end;

The comparison of the three approaches has clearly indicated that the heuristic algorithm is superior. Of the three methodologies being discussed, the line by line approach is the simplest one to implement. However, the output produced by the line by line approach usually can be improved by re-arranging certain words in the paragraph so that each line is equally used up. This implies that the different interword space of the various lines can be minimized and the number of the output lines remains the same as those of the line by line approach. This improvement is actually performed by the dynamic programming approach. However, the dynamic programming approach is also imperfect. It will produce "bad breaking points", "widow lines" and allows no option for breaking a line at the point desired by the user. Here, "bad breaking point" refers to a situation such as the breaking of "figure 3.1" into two separate lines. "Widow line" refers to the case when the last line only contains a few characters (say 5% of total available space). Certainly, widow lines and bad breaks are not desirable. Most of these bad breaks and widow lines can be overcome by using the heuristic approach. In fact, this approach provides several advantages, such as: (a) option to break at any point (b) different line widths can be specified within one paragraph (c) total number of lines generated can be made greater or shorter. Also, hyphens can be generated automatically because the algorithm is carried out on a character by character basis. It was decided to adopt the heuristic approach because of its greater power and sophistication, as well as the fact that it provides some means to help eliminate bad breaks which are not provided by other approaches. (Since page breaking problems are similar to those of line breaking, the algorithm developed by Knuth and Plass also applies)

3.3.3.3 Generate Galley Modules : line-galley and page-galley

As shown in figure 3.3, section 3.3.3, the line breaking process (the production of line galley) and the page breaking process (the production of page galley) are separated into two different modules. The reasons for the separation are obvious: the two tasks are practically unrelated, and the separation simplifies the formatting process. Finally, if the dynamic programming approach or the heuristic approach were to be used, then the amount of data (text and its formatting information) needed to be stored in the core memory would be enormously large (practically unfeasible). The basic tasks of these two modules are the same :

- 1). accept input relations
- 2). call line/page break routines
- 3). create galley relations

The output relation, "line-galley", from the line-galley module contains all the information needed to display a line. (note : the line-galley relation will be discarded at the end of the formatting process.)

The relation "line-galley" will have the following attributes :

0) partname --> used to distinguish a different part of the text

e.g. 'f' --> means front matter

'b' --> means main body

'a' --> means appendix

1) tag --> GML tag

2) leftspace --> left space is needed before printing a word

3) wordseq --> word sequence number

4) lineseq --> line sequence number

5) maxheight --> maximum vertical space needed for a line

6) maxup --> maximum upper vertical space needed for a line

e.g. vertical size of a character is defined in two parts 'up' and 'down', which are used for line-up purposes.

7) word --> actual word

The output relation, "page-galley", from the page-galley module will contain all the information needed to produce the formatted output (all the information contained in the line-galley plus information needed to display a page).

The relation "page-galley" will have all the attributes contained in the relation "line-galley" plus the following two attributes :

1) pagenum --> page number

2) topspace --> upper vertical space needed before printing a line

3.3.3.4 Display Driver

Since the display driver is machine dependant, the module was designed with the output device in mind. In order to make the system easier to adjust to other output devices, the functions of this module were kept as general and simple as possible. They are:

- 1). to request the input relation
- 2). to process the input relation (page galley)
- 3). to translate the logical formatting commands into machine readable form
- 4). to send the machine readable form to the output device

CHAPTER FOUR

Details of Implementation

This chapter deals with a description of the host system configuration which was used to build the experimental formatter. The programming language used to build the system will be discussed. Then the implementation goal and the output device for the system will be described. Finally, a brief discussion of the implementation of each important module will be presented.

4.1 Host System Configuration

The experimental relation-based formatter is built on a small Cadmus work station. The work station consists of one console, three normal CRTs and one black and white raster graphic terminal with mouse interface. The system is run by two Motorola Mc68000 series microprocessors: one for the operating system and the other for the graphic controller for the graphic terminal. The memory of the work station is three mega bytes of RAM. Two 65 mega byte winchester hard disks are also included as a secondary memory. A streamer tape and a floppy disk drive unit are equipped for back up purposes. The graphics terminal is able to address 1024×1024 pixels, the display screen can display 1024×800 pixels. The rest of the 1024×224 pixels are designed for pattern storage purposes. The operating system used on this work station is a multi-user UNIX operating system.

4.2 Programming Language Used

The "C" programming language was used to build the experimental system. "C" was chosen because it is the major programming language for UNIX as well as the fact that it allows its user to deal with both low level objects (masking words, shifting bits, etc.) and high level objects (records, files, pointers, etc.). These features make this programming language extremely powerful and appealing. "C" also has a simple, yet extremely efficient filing mechanism. There are no complicated file structures. All files are flat files and no special definitions are imposed on them.

4.3 Implementation Goal

Due to time constraint, it was decided to concentrate on one document type, namely book. Only one formatting style was implemented for that type. In other words, instead of an incomplete system with formatting choices, it was decided to build one complete formatter (based on relations). Even though only one document type was used for the implementation part, the system was developed to be as general as possible, so that in the future multiple choice options could be easily added. For example, though there were no other formatting options, the switch board mechanism was still implemented for each formatting command. The default choice of each formatting command on the switch board was the one implemented and was the only workable one.

4.4 Output Device

The output device chosen for the experimental system was the Cadmus graphic terminal. This terminal is able to display one page per screen, and the major goal was to determine whether or not a formatter based on relations was feasible. For our purpose it was inconsequential whether the formatted results were printed on paper or displayed on a screen. However, the use of the graphic screen as the output device proved that the formatter could be device independent. Furthermore, in the future the system can easily be changed to a "What you see, is what you get" type of formatter. This is of interest since many find it more appealing to see the formatted output before it is printed on paper.

4.5 Simple Description of the Implementation

In this section, the physical storage structure (methods) used to store the relations will be described. After this, the implementations of each important module described in figure 4.1 will be discussed briefly.

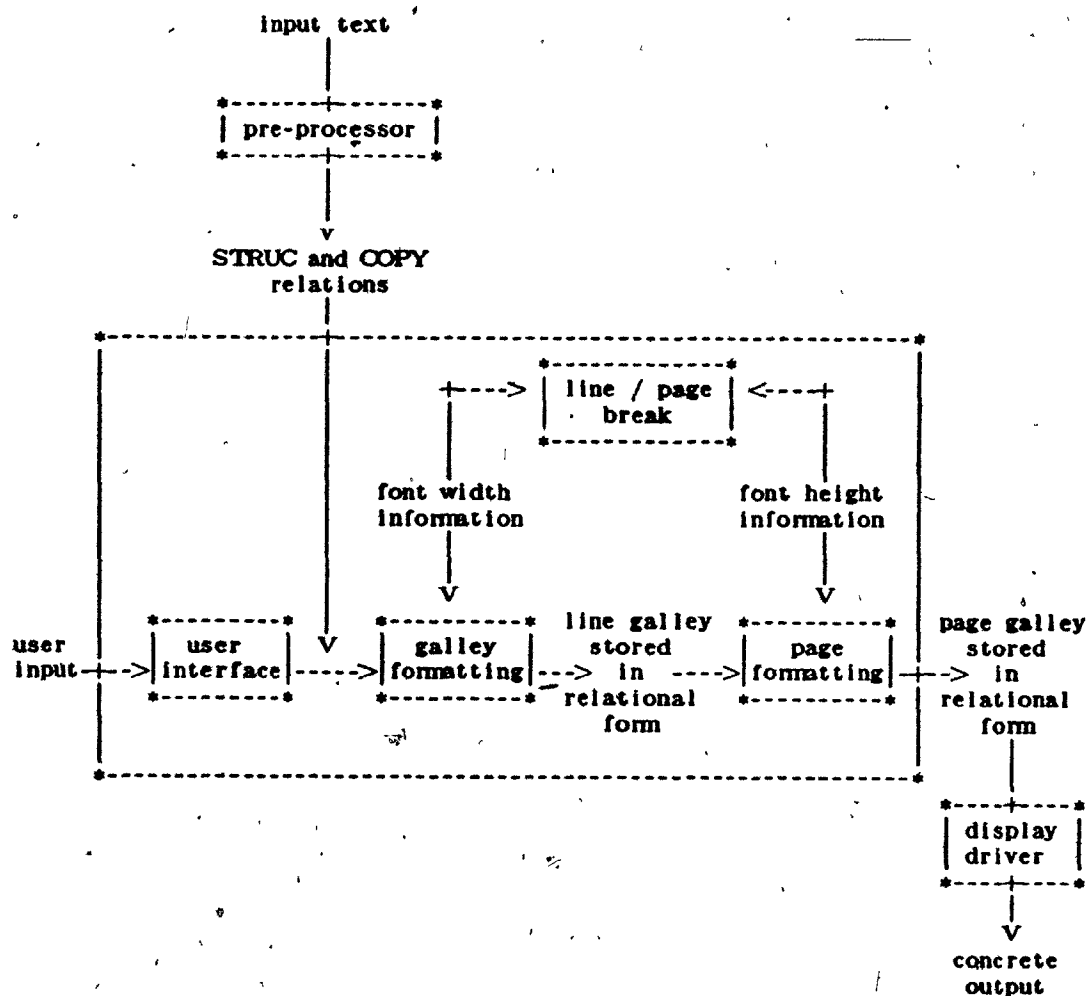


Figure 4.1 Major components of the relational formatter

4.5.1 Storage Structure of Relations

In this experimental formatter, relations are stored in n-tuple form in secondary storage. For the purpose of this thesis, n-tuple form means that each tuple was stored exactly as is, without going through other forms of operations before storing or retrieving. In other words, if a tuple will take 20 bytes in RAM then the attempt was made to use the exact memory (20 bytes) to store it in secondary storage. However, since the structure of the memory devices are one dimensional, in order to preserve the tabular structure of a relation some kind of indicator was needed to signify the end of one tuple and the beginning of another. The indicator chosen for this purpose is "/". In the n-tuple form, if a tuple will take 20 bytes in RAM, it will take 21 bytes in secondary storage.

The major advantage of the indicator is that it allows the tuple to be stored in various lengths. For example, in a COPY relation each tuple has the following attributes: CTAG, CSEQ, WORDSEQ and WORD. When the tuple is stored the attributes are stored in the forms of an integer (2 bytes), a float (4 bytes), an integer (2 bytes), and n characters (n bytes) respectively. In addition, 1 byte will be reserved for the separator. In this case, the ordering of the attributes is immaterial and the size of each attribute is fixed except for the word size. The only place an indicator is needed is immediately after the word. Once the size of the word is known the end of the current tuple is also known. The only disadvantage of this kind of file structure is that the file which contains the tuples cannot be processed with a normal text editor.

While this may be a disadvantage for processing it is an advantage for file security. For an unauthorized user who does not know the ordering of the attributes or the indicator symbol, there is no way he/she can figure out the contents from that file. In other words, the file is encoded by itself, and the key to decoding it is knowledge of the ordering of the attributes and the indicator symbol. Besides providing document security there are other advantages to this system.

The n-tuple form saves memory space. In the ASCII form (common practice), for every non character type attribute (i.e. Integer, real) with a value greater than or equal to 10, at least one byte more will be needed for storage than in the n-tuple form. For example, to store the Integer 10 in the ASCII form then one byte will be needed for the character '1', one byte for the character '0' and one byte for the indicator (used to indicate the end of the attribute). However, to store 10 in n-tuple form only 2 bytes (Integer size on CADMUS) will be needed. The indicator is not required, because the size of an Integer type element will always be 2 bytes.

In terms of processing time at run time, the n-tuple form is much more efficient than the ASCII form. Following are two examples, one for ASCII and another for n-tuple form, used to illustrate the required statements to read in the value of one Integer type attribute.

Example 1 (data stored in ASCII form)

```
number = 0;
read("data", chartemp, 1); /* read one character from file */
do
{
    number = number * 10 + (chartemp - '0');
    read("data", chartemp, 1);
}
while (chartemp != '/');
```

Example 2 (data stored in n-tuple form)

```
read("data", &number, 2);
```

In example 1, at least 2 assignments, 1 comparison, 1 multiply, 1 addition, 1 subtraction and 2 read statements are needed. However, in example 2, only 1 read statement is needed. The same situation will be true for the storage of a number.

4.5.2 Pre-Processor for the Formatter

As discussed in chapter three, the input relations to the formatter, namely COPY and STRUC, are generated by the relational editor. Since the editor is not the major concern in this thesis, but the two relations for our experimental formatter were still needed it was decided to build a pre-processor. The job of the pre-processor is firstly to read a normal file containing the GML commands and the document; and secondly to produce the two relations. The implementation of the pre-processor is quite simple. Essentially, it works on two tables: the "command-stack" and "word". The "command-stack" is a two dimensional table with the following attributes: 'PTAG', 'PSEQ', 'CTAG', 'CSEQ'; where 'PTAG' contains parent "tag name" (GML tag), 'PSEQ' contains the parent tag-id, 'CTAG' contains child "tag name", and 'CSEQ' contains child tag-id. The "word" is a one dimensional table, which is used to store one word or the "tag name". The algorithm presented on the following page is the simplified version of the pre-processor.

Algorithm

- 1). open the file needed to be formatted;
 $l = 1$; $top = 0$; $wordseq = 0$; $tagseq = 0$;
 $command_stack[CTAG][top] = "gdoc"$; $command_stack[CSEQ][top] = 0$;
- 2). read one character from the file put into $word[0]$;
 if end of file goto 13;
- 3). if $word[0] = " "$ goto 2;
- 4). if $word[0] = ":"$ goto 10
- 5). read one character from file put into $word[l]$;
- 6). if $word[l] = " "$ or $word[l] = ":"$ goto 8;
- 7). $l = l + 1$; goto 5
- 8). write one COPY tuple;
 $wordseq = wordseq + 1$;
 if $word[l] = ":"$ goto 10;
- 9). $l = 1$; goto 2;
- 10). look ahead one character;
 if character = "e" goto 12;
- 11). read in the new tag name; $tagseq = tagseq + 1$;
 $top = top + 1$;
 $command_stack[PTAG][top] = command_stack[CTAG][top - 1]$;
 $command_stack[PSEQ][top] = command_stack[CSEQ][top - 1]$;
 $command_stack[CTAG][top] = \text{new tag name}$;
 $command_stack[CSEQ][top] = tagseq$;
 write one tuple of STRUC file from $command_stack[*][top]$;
 $wordseq = 0$; goto 2;
- 12). read in the tag name;
 $top = top - 1$; /* pop the $command_stack[top]$ */
 goto 2;
- 13). close all files; terminate.

Example

Assume we have the following input text file (taken from chapter two of this thesis)

```
:body.  
:h0.  
D E M O  
:h1.  
Empty Chapter  
:p.  
Empty paragraph.  
:ep.  
:eh1.  
:h1.  
Survey of  
:eh1.  
:h1.  
Existing  
:eh1.  
:h1.  
Text Formatting Systems ,  
:p.  
A text formatting system ... program designed to deal  
with ... document on a specific medium.  
:  
:  
of computer hardware and software ... the popularity and  
desirability of text formatting systems.  
:ep.  
:  
:  
:h2.  
Procedural Approach Systems  
:p.  
Systems using ... text formatting problems are ... of the  
system will ... document. The objective ... is therefore  
:  
:  
necessary tools.  
:ep.  
:  
:  
:eh0.  
:ebody.  
:egdoc.
```

The output from the pre-processor will be the following two relations:

STRUC (PTAG , PSEQ , CTAG , CSEQ)

gdoc	0	body	1
body	1	h0	2
h0	2	h1	3
h1	3	p	4
h0	2	h1	5
h0	2	h1	6
h0	2	h1	7
h1	7	p	8
:	:	:	:
h1	7	h2	16
h2	16	p	17
:	:	:	:

COPY (CTAG , CSEQ , WORDSEQ , WORD)

h0	2	0	D
h0	2	1	E
h0	2	2	M
h0	2	3	O
h1	3	0	Empty
h1	3	1	Chapter
p	4	0	Empty
p	4	1	paragraph.
h1	5	0	Survey
h1	5	1	of
h1	6	0	Existing
h1	7	0	Text
h1	7	1	Formatting
h1	7	2	Systems
p	8	0	A
p	8	1	text
p	8	2	formatting
:	:	:	:
p	8	8	designed
p	8	9	to
p	8	10	deal
:	:	:	:
p	8	106	popularity
p	8	107	and
p	8	108	desirability
:	:	:	:
p	17	18	the
p	17	19	system
p	17	20	will
:	:	:	:
p	17	29	document.
p	17	30	The
p	17	31	objective
:	:	:	:
p	17	78	necessary
p	17	79	tools.

4.5.3 User-Interface Module

The user-interface module was implemented as mentioned in the previous chapter. The "Screen Updating and Cursor Movement Optimization: A Library Package" [Bell Laboratories 83] (a utility program on UNIX operating system) was used to build the user interface module. This package provides facilities for the "C" programmer to deal with the CRT screen display and updating. Details of each page template will be discussed in Chapter five, section 5.3.

Following is a general outline of the algorithm of this module:

- 1) clear the screen
- 2) display one screen of default options;
move cursor to the "change" place-hole;
/* default place-hole value equal to NO */
- 3) accept input;
If input equal to <CR> goto 12
clear the place-hole;
echo input; accept input;
while (input != <CR>)
{
 echo input;
 accept input;
}
- 4) move cursor to the first place-hole of the options
- 5) accept input;
If input equal to <CR> goto 10
- 6) clear place-hole
- 7) echo input
- 8) accept input;
If input not equal to <CR> goto 7

- 9) change the current option's value
- 10) move cursor to the next place-hole;
if current place-hole equal to "change" place-hole goto 3
- 11) goto 5
- 12) if all default options not displayed goto 1
- 13) terminate.

4.5.4 Line/Page Break Module

As mentioned in chapter 3, section 3.3.3b, the line breaking algorithm developed by Knuth and Plass was implemented. The algorithm had to be modified so as to handle our basic input data structure. (Since our application handled a paragraph word by word rather than character by character). Only information about word size, word type and total number of words in the paragraph is required, since the application routine is designed to produce the optimal break points for each paragraph. The information of word size and word type is stored in two separate arrays. In contrast to the line breaking algorithm, the word type only indicates penalty type and box type. There are only two types because the input paragraph is stored in a relational form and the inter-word space has been assumed to be unique. The information on the inter-word space is not necessary for our application. Without the glue type, it is assumed that there is a legal break point after each word. Further, our application allows freedom for the use of phototypesetter, CRT and normal typewriter devices for the formatted output. A listing of our line breaking procedures is also included in appendix A.

Example

With the same input paragraph as in Example 1 and Example 2 (in chapter 3, section 3.3.3b), our line breaking routine will have the following inputs:

ptype :- array used to store word type
 psize :- array used to store word size
 index :- (index - 1) pointing to the last word

GLUEWIDTH = 1.0
 GLUESTRETCH = 1.5
 GLUESHRINK = 0.0
 GLUE = '0'
 BOX = '1'
 PENALTY = '2'
 size = 35

Figure 4.2 illustrates the result of our line breaking routine.

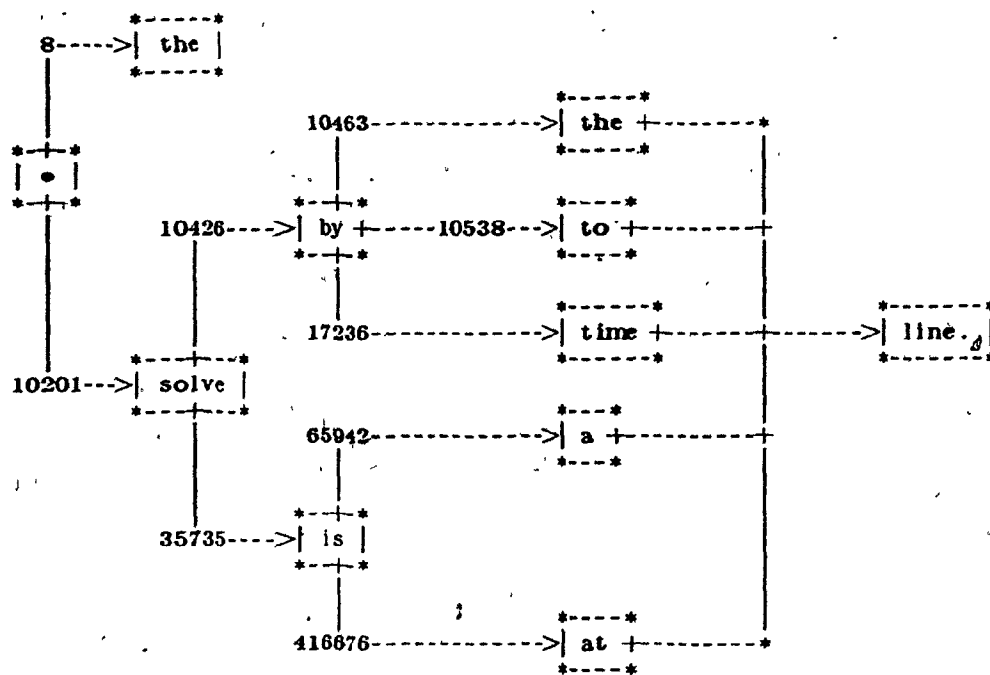


Figure 4.2. This network is constructed by the program using the algorithm developed by Knuth and Plass. It shows the feasible breakpoints (in box) and number of demerits charged (numbers) when proceeding from one breakpoint to another. Note: this example has been represented in a network structure as compared to the shortest path graph in figure 3.4b but represents the same process.

Once the network is established, it is trivial to find the best sequence to break the paragraph. Finally, the same routine as in example 1 (chapter 3, section 3.3.3b) can be used to produce the following formatted output.

The conventional way to solve the line breaking problem is by assigning one word at a time to the current line.

4.5.5 First Pass Module : galley formatting

The major goal of this module is to generate the temporary line-galley relation. As mentioned above, (figure 4.1) the input to this module will be the COPY and STRUC relations. Since the only instructions contained in the COPY and STRUC relations are document structure and types of document elements, this module requires an understanding of a number of GML tags, presented in section 3.2. For instance, the module must understand the hierarchical structure of the GML syntax and the logical meaning of the GML tags. A switchboard mechanism must be implemented in order to allow for different formatting styles for each adopted GML tag. In order for this module to meet the above requirements, it was decided to implement each of the adopted GML tags as individual units (modules). In other words, each tag will have a unique procedure (subroutine) to perform the necessary tasks. These procedures process the STRUC and the COPY relations and store the information into arrays, which are the input of the line/page break module. At the end of each procedure, if the current document type is to be treated as a complete unit, then the corresponding procedure will

pass the information arrays to the line/page break module to break the stream of text into lines and store the line-galley tuples according to the result returned by the line/page break module. For example, "p" (paragraph) and "lq" (long quote) are complete units, but "q" (quote) and "hp2" (bold face font) are not considered complete units. This form of implementation was necessary in order to make the switchboard mechanism possible as well as to facilitate the adherence to the GML hierarchical schema.

From relations COPY and STRUC a new relation "LINEGALLEY" can be generated with the following new attributes: partname, leftspace, llneseq, max-height, maxup (see section 3.3.3.3).

The algorithm presented on the following page is the simplified version of the implementation of the first pass.

Algorithm (general outline)

- 1) open STRUC and COPY files
- 2) read in one tuple from STRUC relation;
push the tuple into STRUC-STACK
- 3) If current tag (CTAG) has no text elements goto 2
look ahead the CSEQ from COPY relation;
If ((current (CSEQ) not equal to the CSEQ from COPY relation)
and (CSEQ from COPY relation not in STRUC-STACK)) goto 2
/* e.g. tag "toc" has no text elements */
- 4) read in one tuple from COPY relation;
If COPY relation finished goto 9
- 5) If current tag-ld (CSEQ) from COPY relation not equal to
tag-ld (CSEQ) from STRUC-STACK[top] goto 7
- 6) calculate the size information of the word;
store information into arrays;
goto 4
- 7) push back the COPY file pointer one tuple;
If CTAG of STRUC-STACK[top] is a complete unit goto 8;
look ahead one tuple from STRUC file;
If ((STRUC file finished) or (PSEQ from STRUC file not equal to
CSEQ from STRUC-STACK[top])) pop STRUC-STACK;
If CSEQ from STRUC-STACK[top] equal to CSEQ from COPY
goto 4;
goto 2
- 8) call line/page break module;
store the line-galley tuples according to the results
returned by line/page break module;
look ahead one tuple from STRUC file;
If ((STRUC file finished) or (PSEQ from STRUC file not equal to
CSEQ from STRUC-STACK[top])) pop STRUC-STACK;
If CSEQ from STRUC-STACK[top] equal to CSEQ from COPY
goto 4;
goto 2
- 9) call line/page break module;
store the line-galley tuples;
close files; terminate.

Example

The output relation ("line-galley") from this module, for the COPY and STRUC relations shown in section 4.5.2 will look like the following:

line-galley (PART , TAG , LEFTSPACE , WORDSEQ , LINESEQ , MAXHEIGHT , MAXP , WORD)

b	p	74.75	0	9	19.0	17.0	A
b	p	9.56	1	9	19.0	17.0	text
b	p	9.56	2	9	19.0	17.0	formatting
:	:	:	:	:	:	:	:
b	p	9.56	7	9	19.0	17.0	program
b	p	9.56	8	9	19.0	17.0	designed
b	p	74.75	1	10	10.0	8.0	to
b	p	10.77	2	10	10.0	8.0	deal
:	:	:	:	:	:	:	:
b	p	7.81	8	19	10.0	8.0	popularity
b	p	7.81	9	19	10.0	8.0	and
b	p	74.75	1	20	10.0	8.0	desirability
:	:	:	:	:	:	:	:
b	p	12.05	10	32	10.0	8.0	of
b	p	12.05	11	32	10.0	8.0	the
b	p	74.75	1	33	10.0	8.0	system
b	p	7.45	2	33	10.0	8.0	will
:	:	:	:	:	:	:	:
b	p	7.45	11	33	10.0	8.0	document
b	p	74.75	1	34	11.0	8.0	The
b	p	10.65	2	34	11.0	8.0	objective
:	:	:	:	:	:	:	:

4.5.6 Second Pass Module : page formatting

This module is used to generate the page-galley relation. The implementation of this module is similar to the first pass module, the major difference being the problems dealt with. In this module, the implementation part is concerned with making the page attractive. Thus, there is more design dependency than in the first pass module. The implementation part is also structured according to the formatting rules, such as starting a new page when text type is "h1" (chapter heading). The page-galley relation generated by this module has all the attributes of the line-galley relation. In addition, there are two more attributes: page_number (indicate the page number) and topspace (top space needed before printing the line). The simplified version of the implementation can be summarized in the following algorithm.

Algorithm (general outline)

- 1) read in one tuple from the line-galley file;
if line-galley file finished goto 5
- 2) if current document type should begin on a new page goto 4
- 3) store the vertical information into arrays;
skip all tuples belonging to this line;
goto 1
- 4) call line/page break module;
store the page-galley tuples according to the result returned
by the line/page break module;
goto 3
- 5) call line/page break module;
store the page-galley tuples;
terminate.

Example

The output relation ("page-galley") from this module, for the "line-galley" relation shown in section 4.5.3 will look like the following:

page-galley (PART , TAG , PAGENUM , TOPSPACE , MAXHEIGHT , MAXUP , LEFTSPACE , LINESQ , WORDSEQ , WORD)

b	p	5	55.35	19.0	17.0	74.75	9.0	0	A
b	p	5	55.35	19.0	17.0	9.58	9.0	1	text
b	p	5	55.35	19.0	17.0	9.58	9.0	2	formatting
...									
b	p	5	55.35	19.0	17.0	9.58	9.0	7	program
b	p	5	55.35	19.0	17.0	9.58	9.0	8	designed
b	p	5	4.0	10.0	8.0	74.75	10.0	1	to
b	p	5	4.0	10.0	8.0	10.77	10.0	2	deal
...									
b	p	5	4.0	10.0	8.0	7.81	19.0	8	popularity
b	p	5	4.0	10.0	8.0	7.81	19.0	9	and
b	p	5	4.0	10.0	8.0	74.75	20.0	1	desirability
...									
b	p	5	4.0	10.0	8.0	12.05	32.0	10	of
b	p	5	4.0	10.0	8.0	12.05	32.0	11	the
b	p	5	4.0	10.0	8.0	74.75	33.0	1	system
b	p	5	4.0	10.0	8.0	7.45	33.0	2	will
...									
b	p	5	4.0	10.0	8.0	7.45	33.0	11	document.
b	p	5	22.33	11.0	8.0	74.75	34.0	1	The
b	p	5	22.33	11.0	8.0	10.65	34.0	2	objective

4.5.7 Display Driver Module

The implementation of this module is the easiest one in the system. The system was designed so that the display driver only has to do the following: a) use the right character font, b) move to the right place, and c) display the character. It is easy for the display driver to meet the requirements. In the page tuple, attribute "tag" indicates which character font to use, attributes "maxheight", "maxup", "leftspace" and "topspace" indicate where to print the character. Finally, to display the character knowledge of the control sequence is needed for

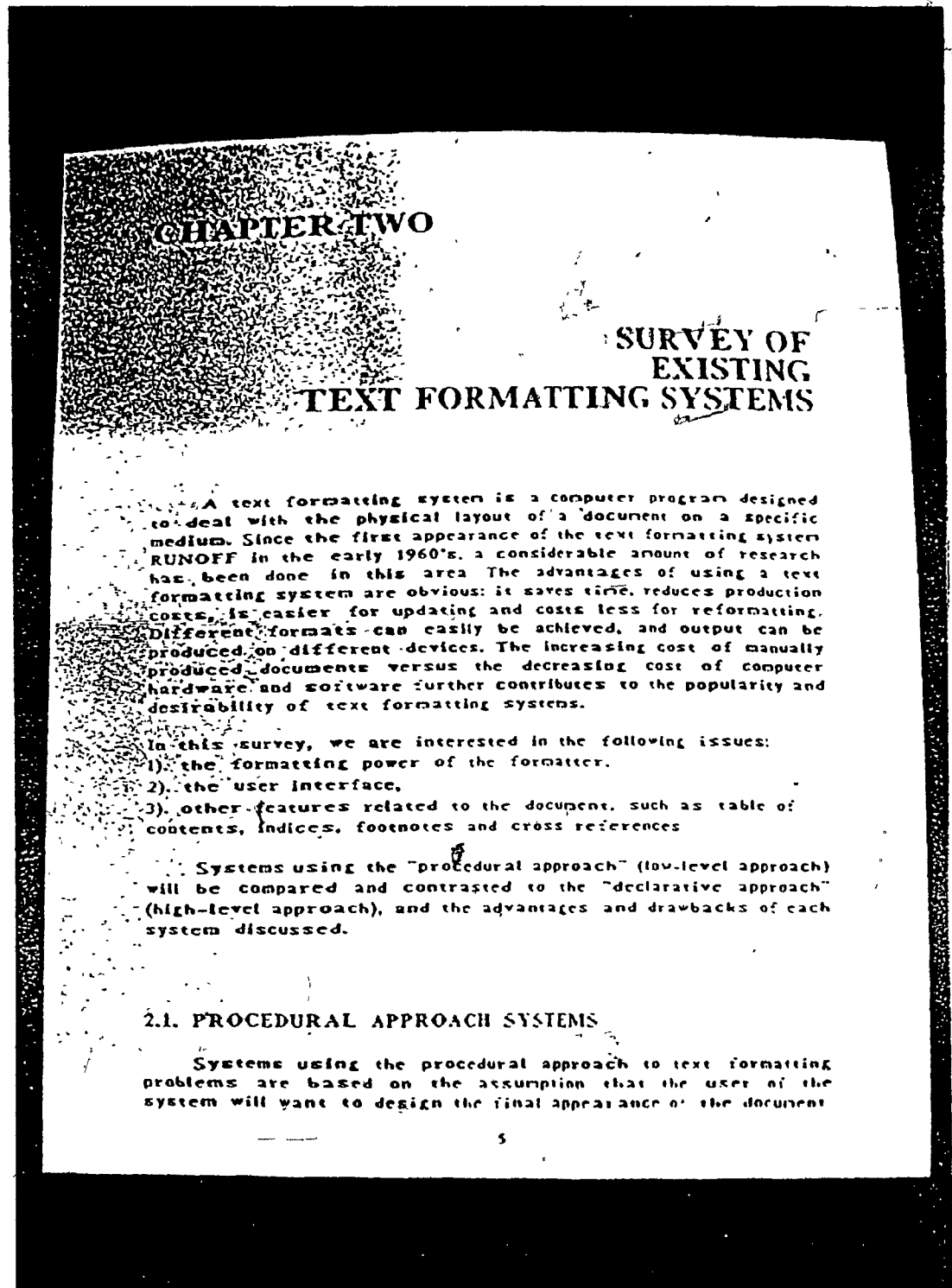
the particular output device. As mentioned in the beginning of the chapter, it was decided to use the Cadmus graphic terminal as the output device for our experimental system. Since the output is shown on the screen instead of on paper, one more function must be added into the display driver. This is the wait function which is added to allow the users the time to view any particular formatted page displayed on the graphic screen.

Following is a general outline of the algorithm of this module:

- 1) ~~open the page-galley file~~
- 2) clear the graphic screen;
display one blank page template;
current_page = 1
- 3) read in one tuple from the page-galley;
if page-galley finished goto 7
- 4) if pagenum not equal to current_page goto 6
- 5) move the cursor to the right location;
display the word;
goto 3
- 6) accept input; /* wait for input */
if input character equal to "s" goto 7;
current_page = pagenum;
clear page template;
goto 5
- 7) close file;
terminate

Example

For the "page-galley" relation shown in the previous section, the output from the display driver will look like the following:



CHAPTER TWO

The objective of the system is therefore to provide its user with a set of tools (commands) to manipulate the physical layout of the document. In other words, the users of the system are fully responsible for the formatted output, as long as the system provides all the necessary tools.

In the following sections we shall discuss a few systems based on this approach. The systems are ordered by their date of appearance and their formatting capability.

2.1.1 RUNOFF

RUNOFF was one of the pioneer text formatting systems, appearing in 1964 on the Compatible Time-Sharing System (CTSS) at MIT [Kurata 1982]. It was designed to deal with its input and output on a typewriter-like device. Since RUNOFF was developed when computer technology was in its infancy, its formatting capabilities are limited by the output device. Also, since RUNOFF was one of the first text formatters, it provides relatively few features compared to text formatting systems commercially available today. Essentially, anything produced by RUNOFF can be produced similarly by a typewriter if sufficient time is provided. However, RUNOFF proved that using a computer to perform tedious work results in a great reduction of man hours. Following are some of the RUNOFF commands:

- a) .center - place the object in the center of a line
- b) .space $\#$ - skip $\#$ of lines (produce vertical spacing)
- c) .indent $\#$ - skip $\#$ of spaces (produce horizontal spacing)
- d) .undent $\#$ - unskip $\#$ of spaces (reduce horizontal spacing)
- e) .adjust - start left and right justification
- f) .noadjust - no justification

RUNOFF is obviously easy to use. There are few commands, all command names are self explanatory, and all deal with the simple problem of object placement. However, RUNOFF's simplicity results in several drawbacks. The small set of commands limits the formatting capabilities (this problem is caused by the limitation of the output device). Command names are usually long, leading to a greater chance of typing errors, and costing more time for expert users. Also, since all commands deal with the physical layout of the page, a small change in the input text may require the whole document to be re-organized.

CHAPTER FIVE

Tutorial Introduction

This chapter will describe the use of the experimental formatter. First the creation of the input data file will be discussed, followed by an explanation of the use of the pre-processor to generate the COPY and STRUC relations. Then the use of the formatter and the display driver will be explained. Finally, a list of implemented GML formatting commands will be provided and the user will be told how to use the commands collectively.

5.1 How to Create Input Data File

As mentioned in the previous chapter, the relational editor is not available at the present time. As an alternative, a normal text editor was used to enter the documents and subsequently the pre-processor was used to generate the required relations, namely COPY and STRUC, to the formatter. The text editors available on the UNIX operating system are "ed", "ex" and "vi" editors. The "vi" editor was chosen not only because "vi" is a full screen editor, but also because it is easy to use. In the following paragraphs a few "vi" commands will be provided, and the way to use "vi" to create a file will be discussed.

The insert and command mode of "vi" is sufficient for our purposes. The insert mode, which is triggered by typing the "i" once, will allow its user to enter

any printable text. To exit the insert mode, the user must press down the `<ESC>` key once. The command mode is the home mode of "vi", which is designed for modifying text.

The following is a partial list of "vi" command mode commands, sufficient for the new user.

- 1) `h` - move cursor to the left
- 2) `j` - move cursor down one line
- 3) `k` - move cursor up one line
- 4) `l` - move cursor to the right
- 5) `x` - delete the character pointed to by cursor
- 6) `dd` - delete the line pointed to by cursor
- 7) `u` - undo previous operation
- 8) `i` - enter insert mode
- 9) `:wq` - save the file and then exit the "vi" editor

The following is a demonstration of the creation of a new file:

```

UNIX -> login:
user -> frank
UNIX -> password:
user -> /* not echoed */
UNIX -> /* prompt from the system */
user -> %
vi example

"example" [New file]

user -> <i>
user -> :gdoc.
:frontm.

:efrontm.
:body

:h1.
Tutorial Introduction
:p.
In this chapter ..... use the command collectively.
:ep.
:h2.
How to Create the Input Data File
:p.
As mentioned in the ..... skip to next section.
:ep.
:p.
The editor "vi" ..... commands.
:ep.

:eh2.

:eh1.

:ebdy.
:appendix.

:eappendix.
:egdoc.

user -> <ESC> /* get out of the insert mode */
user -> :wq <cr> /* save the file and terminate the editor */
UNIX -> "example" [New file] #lines, #characters
UNIX -> %

```

Once a file is created, the "vi" editor can be used to modify it. To modify an existing file, a user has merely to type "vi filename <CR>" and the "vi" editor will be activated. The user will then be in the "vi" command mode in which he can use the cursor movement commands to move the cursor to the right position to make the desired changes.

5.2 How to use the Pre-Processor

The pre-processor is simple to use. After signing on to the system, the user has merely to run the pre-processor. The pre-processor is activated by typing the "pre-processor's name <CR>". Of course, the user has to make sure that the pre-processor is in his/her current directory or is in the system directory. Once the pre-processor has been activated, it will ask for the input file name of the document needed to be formatted. The pre-processor will produce the COPY and STRUC relations in the current directory under the file name COPY and STRUC.

Example:

```
UNIX -> %
user -> gen_copy<cr> /* activate the pre-processor */

Pre-Processor -> /* clear the screen */

Pre-Processor -> enter input text file name ->
user -> example<cr> /* document needed to be formatted */

UNIX -> % /* return from the pre-processor */
```

5.3 User Interface with the Formatter

Once the relations COPY and STRUC are ready, the user can activate the formatter by typing "formatter's name <CR>". The formatter will first clear the screen and then print out the first page of the available default options. One page of the screen is assumed to contain 24 lines vertically and 80 columns horizontally. It is assumed that a normal CRT is being used with the graphics terminal beside it. However, if the user wants to use the graphics terminal as the only communication device, the formatter will still accept it. In such a case, however, even though the graphics terminal can display 66 lines per screen, the formatter will only use the first 24 lines. Similarly, for each line the formatter will only use the first 80 columns (note: this is only for displaying the options, it is the display driver which displays the formatted text).

The interface always starts at the "CHANGE" place-hole which asks the user whether he/she wants to make changes to the current page, or not. If no changes are required, the user simply hits the return key once. The interface routine will go to the next page of default options. If the user does want to make some changes, he/she has to type "y" or "Y" in the first place-hole and then use the return key to jump to the place-hole where the change is desired.

The following are examples of the actual screen display of the four pages of default options:

```

*****
* Defaults for the Relations                                     CHANGE --> NO
*
* Input Relations
* -----
* COPY --> copy
* STRUC --> struc
*
* Output Relations
* -----
* LINE galley --> line_galley
* PAGE galley --> page_galley
* TABLE of CONTENTS --> tableofcontent
*
* Output device is bip
* -----
*****

```

Page 1

```

*****
* Text Type --> 0; { 0 - for book; 1 - for paper; 2 - others }
*
* DEFAULT for PHYSICAL OUTPUT SIZES
*
* width height units
* -----
* Page Size --> 650 * 797 pixels
*
* MARGINS
* -----
* a) left 11.5 % of the page width
* b) right 11.5 % of the page width
* c) top 5.6 % of the page height (include page number)
* d) bottom 5.6 % of the page height (include page number)
*
* CHANGE --> NO
*****

```

Page 2

Page 3

```

*****
Switch Board (Default Options)
*****
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*

```

CHANGE --> NO

Page 4

The first page of the default options is concerned with the input and output relations as well as the display device. The user should check the screen for the file names (relation names) and the output device name.

The second page of the default options contains the default setting for the text type, page size and margins. The text type will be "book", which is the one implemented and which should not be changed. The units of the page size will be in pixels, the default value is equivalent to the paper size 8.5" X 11". This value may be changed by the user, but it is the largest size the graphics screen can handle for one page per screen. The other default settings on this page are for margins. Margins include the left, right, top and bottom margins, which users are able to adjust.

Page three of the default options contains the default formatting rules, which are the general formatting rules for the document. For example, there may be questions such as "Should the system start a new page at heading level 2 ?", or "Should the formatter number the headings?", etc.

Finally, the last page of the default options is the default setting for the switch board. This page is concerned with the formatting style. As mentioned in the previous chapters, the switch board combines different routines to achieve the final document style desired by the user. Different choices of each routine for a particular formatting command is defined by a number. Number 0 is the default option. For the time being, the default setting is the only choice provided to its users.

Once the user has selected the options, the system will start formatting the document accordingly. Unless changed by the user, the output file name of the page-galley will be "page-galley". When execution is completed, the formatter will return control to the UNIX operating system. If an error occurs during the formatting process, the formatter will print the error message on the CRT and terminate itself.

Example:

```
UNIX --> %  
user --> formatter<cr> /* activate the formatter */
```

Once the formatter is activated the user will begin with the options given above. When the formatting task is finished the formatter will return the control to the UNIX operating system.

5.4 How to use the Display Driver

The display driver is activated when the user types in the "display driver's name <CR>". The routine begins by asking the user for the input page-galley's file name. If it can find the page-galley file and has permission to access it, the display routine will clear the Cadmus graphics screen and display one blank page. It will then display the first page of the document. The display routine will wait for the user command before it will display the next page. The command is "s" for terminate; other characters are taken as a request for the next page. After displaying all the pages, the routine will automatically return the control to the host system (UNIX operating system).

Example:

```
UNIX --> %
user --> bldriver<cr> /* activate the display driver */

display driver --> /* clear the screen */

display driver --> enter the page-galley name -->
user --> page_galley<cr>

display driver --> /* start display */
```

5.5 Formatting Commands

This section contains a list of the implemented GML commands. The use of each command will be explained briefly and the syntax rule provided. The syntax is provided in abstract form in order to show the complete syntax under the command and to save space. Two notations are being used to help to express the syntax structure of the formatting language. The first notation introduced is the square bracket ("[.....]"), which means zero or more. In other words, the items in the square bracket can be repeated as many times as necessary or not used at all. If there is more than one square bracket under a particular formatting command, there is no ordering imposed on them. For example, assume we have the following syntax rule under a particular formatting command:-

```
<formatting command>
  [ "a" ]
  [ "b" ]
  [ "c" ]
<end of formatting command>
```

The items might be arranged in the following order:-

```
<formatting command>
  "a"
  "a"
  "c"
  "a"
  "b"
  "b"
  "c"
<end of formatting command>
```

The second notation introduced is the string of "x", which is the symbol for text.

The command list on the following page is arranged in alphabetical order.

1. abstract - use to identify a summary of the document

```
:abstract.  
[ :p. .... :ep. ]  
[ :lq. .... :elq. ]  
[ :h2. .... :eh2. ]  
[ :address. .... :eaddress. ]  
:eabstract.
```

2. address - use to identify the beginning of an address

```
:address.  
[ :allne. xxxxxxxxxxxx :eallne. ]  
:eaddress.
```

3. allne - use to identify a single line of text

```
:allne.  
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  
:eallne.
```

4. appendix - use to identify text materials helpful to the reader, but not essential to the main text.

```
:appendix.  
[ :h1. .... :eh1. ]  
:eappendix.
```

5. author - use to identify the writer of the document

```
:author.  
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  
:eauthor.
```

6. body - use to identify the beginning of the major elements of the document.

```
:body.  
[ :h0. .... :eh0. ]  
:ebody.
```

7. date - use to identify the date associated with the document
- :date.
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
:edate.
8. docnum - (document number) use to identify the number associated with the document
- :docnum.
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
:edocnum.
9. frontm - (front matter) use to identify the starting of the guide line and the introduction part of a document (e.g. abstract, title page, table of contents etc.)
- :frontm.
[:titlep. :etitlep.]
[:abstract. :eabstract.]
[:toc. :etoc.]
:efrontm.
10. gdoc - (general document) use to identify the beginning of the general document
- :gdoc.
[:frontm. :efrontm.]
[:body. :ebody.]
[:appendix. :eappendix.]
:egdoc.
11. hp0 - (highlighted phrase) use to identify a phrase which will be printed in normal text style
- :hp0.
[XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX]
[:hp0. :ehp0.]
[:hp1. :ehp1.]
[:hp2. :ehp2.]
[:hp3. :ehp3.]
[:hp4. :ehp4.]
:ehp0.

12. hp1

(highlighted phrase) identifies a phrase which will be printed in underline form

(same as hp0)

13. hp2

(highlighted phrase) identifies a phrase which will be printed in bold form

(same as hp0)

14. hp3

(highlighted phrase) identifies a phrase which will be printed in bold and underline form

(same as hp0)

15. hp4

(highlighted phrase) identifies a phrase which will be printed in italic form

(same as hp0)

16. h0

(heading level 0) use to identify the beginning of a group of elements. (e.g. a group of consecutive chapters)

```
:h0.  
[ :address. .... :eaddress. ]  
[ :p. .... :ep. ]  
[ :lq. .... :elq. ]  
[ :h1. .... :eh1. ]  
:eh0.
```

17. h1

(heading level 1) use to identify chapter heading, appendix, etc.

```
:h1.  
[ :address. .... :eaddress. ]  
[ :p. .... :ep. ]  
[ :lq. .... :elq. ]  
[ :h2. .... :eh2. ]  
:eh1.
```

18. h2

(heading level 2) use to identify sections

```
:h2.  
[ :address. .... :address. ]  
[ :p. .... :ep. ]  
[ :lq. .... :elq. ]  
[ :h3. .... :eh3. ]  
:eh2.
```

19. h3

(heading level 3) use to identify subsections

```
:h3.  
[ :address. .... :address. ]  
[ :p. .... :ep. ]  
[ :lq. .... :elq. ]  
:eh3.
```

20. lq

(long quotation) use to identify a block of text which is quoted from another source.

```
:lq.  
[ :address. .... :address. ]  
[ :p. .... :ep. ]  
[ :lq. .... :elq. ]  
:elq.
```

21. p

(paragraph) use to identify a paragraph

```
:p.  
[ xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ]  
[ :q. .... :eq. ]  
[ :hp0. .... :ehp0. ]  
[ :hp1. .... :ehp1. ]  
[ :hp2. .... :ehp2. ]  
[ :hp3. .... :ehp3. ]  
[ :hp4. .... :ehp4. ]  
:ep.
```

22. preface

use to identify preface

```
:preface.  
[ :address. .... :address. ]  
[ :p. .... :ep. ]  
[ :lq. .... :elq. ]  
[ :h2. .... :eh2. ]  
:epreface.
```

23. q

(quote) use to identify a phrase cited from a person or text.

:q.

xx

:eq.

24. title

use to identify the name of a document

:title.

xx

:etitle.

25. titlep

(title page) use to identify the beginning of a title page

:titlep.

[:title. :etitle.]

[:docnum. :edocnum.]

[:date. :edate.]

[:author. :eauthor.]

[:address. :eaddress.]

:etitlep.

26. toc

use to request the "table of contents" generated by the system

:toc.

:etoc

CHAPTER SIX

Conclusion

6.1 Summary and Advantages

This thesis was a discussion of the theoretical and practical aspects of the development of an experimental formatter based on the relational model. A brief introduction to the relational algebra was given in order to explain how relations can be manipulated by means of algebraic operations. This was done in order to illustrate the ways in which algebraic operations are applicable to the text processing problem.

Existing text formatting systems were studied in order to find the most suitable approach to our experimental system. Both high and low level systems were discussed. High level systems were found to be easier to implement and were more user friendly. Finally, the GML formatting language was chosen because it is a high level system with a well defined syntax structure.

The most important task of a formatter is to solve the line and page breaking problems in order to produce attractive text. Three different methods were examined and compared. Knuth's and Plass' heuristic approach was found to be superior to both the conventional (line by line) and the dynamic programming approaches. The heuristic approach was somewhat modified in order to adapt it

to the page breaking problem.

The system was designed with the user in mind. A switch board mechanism was implemented in order to provide a wide variety of formatting styles. New styles can easily be created through the combination of different styles. A template mechanism was implemented in order to facilitate changes or corrections in formatting rules. Finally, a modular approach was adopted in order to maximize the system's flexibility. For example, new formatting styles can easily be added to the switch board.

The experimental formatter produced results that were highly satisfactory. It generated a balanced, attractive output that was free from loose or widow lines. It also avoided bad page breaks. Its success proved the feasibility of formatting text from data stored in relational form. In addition, it demonstrated the possibility of applying relational algebra to text processing tasks. It also unified the methods of text data storage.

6.2 Limitations and Drawbacks

At present, there are some minor limitations to the formatting system. The first drawback is the fact that in order for the system to produce the final formatted output, the document must be complete. The user can format individual chapters in order to check layout style (see example, section 4.5.2). However, the formatter will not assign the correct page numbers until the document is complete. This problem can easily be overcome with the addition of another formatting option for the initial page number.

The second limitation is that presently there is no means of printing output on paper. The formatted output is only registered on the CADMUS raster graphics terminal. Also, at present there is no means of incorporating table and figures into the system. This problem must be solved by further research.

There is a maximum limit to the number of lines in a paragraph and to the number of pages in a chapter (approximately fifty in both cases). This is because a fixed array type data structure was used to implement the algorithm developed by Knuth and Plass. This problem can be overcome by a program change which uses the pointer type data structure instead. However, the processing time will be slowed as a result.

Presently, the total number of chapters per document is limited to twenty five, the total number of appendices is limited to fifteen, and headings are limited to eighty characters. These limits can easily be extended. For example, chapter heading limits can be extended by changing the value of the constant "MAX-CHARHEAD".

Since the system is designed to do most of the layout details for the user, and because of the extra I/O operations involved in extremely high quality formatting, this formatter tends to run slower than conventional systems (approximately half the speed of TROFF).

6.3 Further Work

While the experiment was successful, further research would increase the system's appeal and practicality. At present, the switch board has been provided

with only one formatting style. Obviously, the addition of more formatting styles would allow the user a wider variety of choices.

As mentioned in Chapter 3, not all of the GML formatting commands have been implemented. These remaining commands must be added in order to make the system complete. This project was an experiment only, therefore, as noted earlier all formatted output is only registered on the CADMUS raster graphics terminal. There is no means of printing output on paper. In order to make the system practical, a new display routine must be developed.

More research is needed in order to fully implement the relational editor (used to generate COPY and STRUC relations). In addition, further research must be done in order to develop a means of incorporating tables and figures into the high level system.

Finally, new algorithms should be developed in order to use relational algebra to generate indexes, cross references, word frequency counts, linguistic analyses, etc.

APPENDIX A

The "C" program listing on the following page is a modified version of the algorithm developed by Knuth and Plass.


```

1  #define MAXNODE      150
2  #define NIL          -1
3  #define TRUE         1
4  #define FALSE        0
5  #define GLUE         '0'
6  #define BOX          '1'
7  #define PENALTY      '2'
8  #define GLUEWIDTH    1.0
9  #define GLUESHRINK   0.0
10 #define GLUESTRETCH  1.5
11 #define INFINITE     999999999.0
12 #define CLASS        4
13 #define MAXRATIO      2
14 #define P_DCLASS     1000.0 /* penalty for different class */
15 #define EQ            =
16 #define MAX_LINES     50
17 #define MAX_P_SIZE    2560
18 #define size          35 /* input data */
19 #define lineindex     0 /* input data */
20
21
22
23 int psize[MAX_P_SIZE]; /* store the word size */
24 char ptype[MAX_P_SIZE]; /* store word type */
25
26 int linenum [MAXNODE]; /* ..... */
27 fitness [MAXNODE]; /* ..... */
28 position [MAXNODE]; /* ..... */
29 float t_width [MAXNODE]; /* ..... */
30 t_stretch [MAXNODE]; /* ..... */
31 t_shrink [MAXNODE]; /* ..... */
32 t_demerit [MAXNODE]; /* ..... */
33 adj_ratio [MAXNODE]; /* ..... */
34 int previous [MAXNODE]; /* ..... */
35 next [MAXNODE]; /* ..... */
36 active;
37 passive;
38 avnode;
39 more;
40 now;
41 future;
42 before;
43 class;
44 ptractive[CLASS];
45 lnum;
46 num;
47 pt;
48 index;
49 float twidth;
50 tshrink;
51 tstretch;
52 width;
53 shrink;
54 stretch;
55 penalty;
56 bestfit;
57 fitclass[CLASS];
58 adj_rat[CLASS];
59 adjustr;
60 demerit;
61 tv;
62 ty;
63 tz;
64
65 char type;
66 pretype;

```

```

67
68
69
70
71 choosenode()
72
73 /*****
74  */
75 /* choose the best node */
76 /* input :- size,passive */
77 /* output :- active */
78  */
79 /*****
80
81 {
82     int i,
83         first,
84         second;
85
86     float temp1,
87           temp2;
88
89     i = next[passive];
90     first = passive;
91     second = passive;
92     while((t_width[passive] - t_width[i]) <= size)
93     {
94         if (t_demerit[i] <= t_demerit[first])
95         {
96             second = first;
97             first = i;
98         }
99         else if (t_demerit[i] < t_demerit[second])
100         {
101             second = i;
102         }
103     }
104     if (((t_demerit[second] - t_demerit[first]) < 100) &&
105         (second != first))
106     {
107         temp1 = adj_ratio[first] - adj_ratio[previous[first]];
108         if (temp1 < 0)
109             temp1 = 0 - temp1;
110         temp2 = adj_ratio[second] - adj_ratio[previous[second]];
111         if (temp2 < 0)
112             temp2 = 0 - temp2;
113         if (temp1 <= temp2)
114             active = first;
115         else
116             active = second;
117     }
118     else
119         active = first;
120 }
121
122 help()
123
124 /* the max ratio is not big enough */
125
126 {
127     write(efd, "\n\n noway to break the lines.\n\n", 30);
128 }
129
130
131
132

```

```

133 getnode(ptr)
134
135     int *ptr;
136
137     {
138         if (avnode <= MAXNODE)
139             *ptr = avnode++;
140         else
141             write(efd, "\n\n out of nodes.\n";17);
142     }
143
144
145 loadword()
146
147     {
148         type = ptype[pt];
149         width = psize[pt++];
150         if (type EQ PENALTY)
151             penalty = 100; /* penalty for '-' */
152         else penalty = 0;
153     }
154
155
156 setup(indent)
157
158     int indent;
159
160     {
161         pt = 0;
162         avnode = 0;
163         pretype = GLUE;
164         more = TRUE;
165         if (indent EQ 0)
166             loadword();
167         else
168             {
169                 type = BOX;
170                 width = indent;
171             }
172     } /* setup */
173
174
175 firstnode()
176
177     /*
178     ****
179     ** create an active node representing the beginning of the paragraph.
180     **
181     ****
182
183     {
184         getnode(&active);
185         position[active] = 0;
186         linenum[active] = 0;
187         fitness[active] = 1;
188         adj_ratio[active] = 0;
189         t_width[active] = 0;
190         t_stretch[active] = 0;
191         t_shrink[active] = 0;
192         t_demerit[active] = 0;
193         previous[active] = NIL;
194         next[active] = NIL;
195         passive = NIL;
196     }
197
198

```

```

199
200
201
202 getstart()
203
204     /*
205     ****
206     ** for b := 1 to m do <if b is a legal breakpoint> then <mainloop>
207     **
208     ****
209
210     {
211         twidth = 0;
212         tstretch = 0;
213         tshrink = 0;
214         while (more)
215             {
216                 if (type EQ GLUE)
217                     {
218                         if (pretype EQ BOX)
219                             mainloop();
220                         if (pt EQ index)
221                             {
222                                 type = PENALTY;
223                                 penalty = -INFINITE;
224                             }
225                         else
226                             {
227                                 twidth = twidth + GLUEWIDTH;
228                                 tstretch = tstretch + GLUESTRETCH;
229                                 tshrink = tshrink + GLUESHRINK;
230                                 loadword();
231                                 pretype = GLUE;
232                             }
233                     }
234                 else if (type EQ BOX)
235                     {
236                         twidth = twidth + width;
237                         type = GLUE;
238                         pretype = BOX;
239                     }
240                 else if (penalty != INFINITE)
241                     {
242                         mainloop();
243                         if (pt EQ index)
244                             more = FALSE;
245                         else
246                             {
247                                 type = GLUE;
248                                 pretype = PENALTY;
249                             }
250                     }
251             }
252     }
253
254
255
256
257 mainloop()
258
259     /*
260     ****
261     ** main loop
262     **
263     ****
264

```

```

265 {
266     int i;
267     now = active;
268     before = NIL;
269     do
270     {
271         bestfit = INFINITE;
272         for (i = 0; i <= CLASS; i++)
273         {
274             fitclass[i] = INFINITE;
275             adj_rat[i] = INFINITE;
276         }
277         while (TRUE)
278         {
279             future = next[now];
280             ratio();
281             if ((adjustr < -1) || (penalty <= -INFINITE))
282                 freenode();
283             else
284             {
285                 before = now;
286                 if ((-1 <= adjustr) && (adjustr <= MAXRATIO))
287                 {
288                     demerits();
289                     adj_rat[class] = adjustr;
290                     if (demerit < fitclass[class])
291                     {
292                         fitclass[class] = demerit;
293                         ptractive[class] = now;
294                         if (demerit < bestfit)
295                             bestfit = demerit;
296                     }
297                 }
298             }
299             now = future;
300             if (now EQ NIL)
301                 break;
302             if ((linenum[now] >= lnum) && (lnum < lineindex))
303                 break;
304         }
305         if (bestfit < INFINITE)
306             insertnode();
307     }
308     while (now != NIL);
309     if (active EQ NIL)
310         help();
311 }
312
313
314
315 ratio()
316
317 /******
318 /**
319 /** compute the adjustment ratio r from a to b
320 /**
321 /**
322
323 {
324     float length,
325         difference;
326
327     length = twidth - t_width[now];
328     if (type EQ PENALTY)
329     length = length + < width of '-' >;
330     lnum = linenum[now] + 1;

```

```

331 if (length < size)
332 {
333     difference = tstretch - t_stretch[now];
334     if (difference > 0)
335         adjustr = (size - length) / difference;
336     else
337         adjustr = INFINITE;
338 }
339 else if (length > size)
340 {
341     difference = tshrink - t_shrink[now];
342     if (difference > 0)
343         adjustr = (size - length) / difference;
344     else
345         adjustr = INFINITE;
346 }
347 else
348     adjustr = 0;
349 }
350
351
352 freenode()
353
354 /******
355 /**
356 /** deactivate node a
357 /**
358 /**
359 /**
360
361 {
362     if (before EQ NIL)
363         active = future;
364     else
365         next[before] = future;
366     next[now] = passive;
367     passive = now;
368 }
369
370
371 demerits()
372
373 /******
374 /**
375 /** compute demerits d and fitness class c
376 /**
377 /**
378 /**
379
380 {
381     float temp1,
382         temp2;
383
384     if (adjustr < 0)
385         temp1 = 0 - adjustr;
386     else
387         temp1 = adjustr;
388     temp2 = temp1 * temp1 * temp1;
389     demerit = 1 + 100 * temp2;
390     if (penalty >= 0)
391         demerit = (demerit + penalty) * (demerit + penalty);
392     else if (penalty != -INFINITE)
393         demerit = demerit * demerit - penalty * penalty;
394     else
395         demerit = demerit * demerit;
396     demerit = demerit + < d * fb * fa>;

```

```

397     if (adjustr < -0.5)
398         class = 0;
399     else if (adjustr <= 0.5)
400         class = 1;
401     else if (adjustr <= 1)
402         class = 2;
403     else
404         class = 3;
405     templ = class - fitness[now];
406     if (templ < 0)
407         templ = 0 - templ;
408     if (templ > 1)
409         demerit = demerit + P_DCLASS;
410     demerit = demerit + t_demerit[now];
411 }
412
413
414
415 insertnode()
416
417 /******
418 /*
419 /*  insert new active nodes for breaks from Ac to b
420 /*
421 /******
422
423 {
424     int i;
425
426     findtwyz();
427     for (i = 0; i < CLASS; i++)
428     {
429         if (fitclass[i] <= (bestfit + P_DCLASS))
430         {
431             getnode(&num);
432             linenum[num] = linenum[ptractive[i]] + 1;
433             fitness[num] = i;
434             position[num] = pt;
435             t_width[num] = tw;
436             t_stretch[num] = ty;
437             t_shrink[num] = tz;
438             t_demerit[num] = fitclass[i];
439             adj_ratio[num] = adj_rat[i];
440             previous[num] = ptractive[i];
441             next[num] = now;
442             if (before EQ NIL)
443                 active = num;
444             else
445                 next[before] = num;
446             before = num;
447         }
448     }
449 }
450
451
452
453 findtwyz()
454
455 /******
456 /*
457 /*  compute tw = (sum w) after (b);
458 /*  ty = (sum y) after (b);
459 /*  tz = (sum z) after (b);
460 /*
461 /******
462

```

```

463 {
464     tw = twidth;
465     ty = tstretch;
466     tz = tshrink;
467     if (type EQ GLUE)
468     {
469         tw = tw + GLUEWIDTH;
470         ty = ty + GLUESTRETCH;
471         tz = tz + GLUESHRINK;
472     }
473 }

```

References

- [Achugbne 81] James O. Achugbne
On the lines breaking problem in text formatting
Department of Mathematical and Computer Science
Michigan Technological University, Houghton MI
- [Allen 81] Todd Allen, Robert Nix and Alan Perlis
Pen : a hierarchical document editor
SIGPLAN/SIGOA Symposium on Text Manipulation, Portland (1981)
- [Bell Laboratories 83] Bell Laboratories
UNIX Programmer's Manual Vol. 1 and Vol. 2
CBS College Publishing, (1983)
- [Bernes 1969] Gerald M. Bernes
Description of FORMAT, a text-processing system
CACM Vol. 12, March (1969)
- [Chamberlin et al 81] Chamberlin, D.D., et al.
JANUS: an interactive system for document composition
SIGPLAN/SIGOA Symposium on Text Manipulation, Portland (1981)
- [Chamberlin et al 82] Chamberlin, D.D., et al.
JANUS: an interactive document formatter based on declarative tags
IBM Research Laboratory
San Jose RJ3366 (40402), (1982)
- [Date 81] Date, C.J.
An Introduction to Database Systems
Addison-Wesley Publishing Company, Inc. (1981)
- [Fayerman 84] Brenda Fayerman
A Text Editor Based on Relations
MSc Thesis, McGill University, (Aug. 1984)
- [Furuta 82] Richard Furuta, Jeffrey Scofield, and Alan Shaw
Document Formatting Systems: Survey, Concepts and Issues
ACM Computing Survey, (Sept. 1982)
- [Goldfarb 80] Goldfarb, C.F.
Document Composition Facility GML: concepts and design guide
Form no. SH20-0188-0, IBM, (1980)

- [Goldfarb 81] Goldfarb, C.F.
A Generalized Approach to Document Markup
SIGPLAN/SIGOA Symposium on Text Manipulation, Portland (1981)
- [Good 81] Michael Good
Etude and the Folklore of User Interface Design
SIGPLAN/SIGOA Symposium on Text Manipulation, Portland (1981)
- [Gutknecht 84] J. Gutknecht and W. Winiger
Andra: The Document Preparation System of the Personal Workstation Lillith
Software-Practice and Experience, vol. 14, (1984), 73-100
- [Hammer et al 81] Hammer, M., et al.
The implementation of Etude,
an integrated and interactive document production system
SIGPLAN/SIGOA Symposium on Text Manipulation, Portland (1981)
- [Han 83] Han Noot
Structured Text Formatting
Software-Practice and Experience, vol. 13, (1983), 79-94
- [Hayes 81] Phil Hayes, Eugene Ball, and Raj Reddy
Breaking the Man-Machine Communication Barrier
IEEE Computer, Vol. 14, No 3, (1981), 19-30
- [Horowitz 1976] Ellis Horowitz, and Sartaj Sahni
Fundamentals of Data Structures
Computer Science Press, Inc. (1976)
- [Knuth 81] Knuth, D.E.
The Art of Computer Programming, vol. 2: Seminumerical Algorithms
Addison-Wesley, Reading Massachusetts, (1981)
- [Knuth 84] Knuth, D.E.
The TEXbook
Addison-Wesley Publishing Company, (1984)
- [Knuth and Plass 81] Donald E. Knuth and Michael F. Plass
Breaking Paragraphs into lines
Software-Practice and Experience, vol. 11, (1981), 1119-1184
- [Merrett 84a] Merrett, T.H.
First steps to algebraic processing of text
Gardarin, G., Gelenbe, E., eds.
New Applications of Databases
Academic Press, (1984)

- [Merrett 84b] Merrett, T.H.
Relational Information Systems
Reston Publishing Co., Reston Va., (1984).
- [Newman 82] P.S. Newman
Towards an Integrated Development Environment
IBM System Journal, Vol. 21, No 1, (1982)
- [Nievergelt 82] Nievergelt, J., et al, eds.
Document Preparation Systems
North-Holland Publishing Co., Amsterdam, (1982)
- [Reid 80] Brian K. Reid
Scribe: a document specification language and its compiler
Technical Report Carnegie-Mellon University, Oct., (1980)
- [Reid 81] Brian K. Reid
A high level approach to computer document formatting
7th ACM Symposium on Principle of Programming Language, (1981)
- [Sedgewick 83] Sedgewick, Robert
Algorithms
Addison-Wesley Publishing Company, (1983)
- [Teitelbaum 81a] Teitelbaum, T.
The why and wherefore of the Cornell program synthesizer
SIGPLAN/SIGOA Symposium on Text Manipulation, Portland (1981)
- [Teitelbaum 81b] Teitelbaum, T., Reps, T.
The Cornell Program Synthesizer: A syntax-directed programming environment
Commun. ACM 24, 9 (Sept. 1981)
- [Walker 81] Janet H. Walker
The Document Editor:
A Supporting Environment for Preparing Technical Documents
SIGPLAN/SIGOA Symposium on Text Manipulation, Portland (1981)

note: This thesis was formatted by a TROFF system on a VAX/750 computer. The output printer was a laser IMAGEN 8/300.