1987

The Design and implementation of a system for processing documents described in generalized markup languages

Frank Cost

The Design and Implementation of a System for Processing Documents Described in Generalized Markup Languages

by

Frank J. Cost

A thesis

Submitted to the faculty of the
School of Computer Science and Technology

at

Rochester Institute of Technology

In partial fulfillment of the
Requirements for the
Degree of

Master of Science in Computer Science

May 28, 1987

Approvals:

_____

Professor Guy Johnson Thesis Advisor

_____

Professor Michael Kleper

_____

Professor Jeffrey Lasky

Title of Thesis: The Design and Implementation of A System for Processing Documents Described in Generalized Markup Languages.

I, Frank J. Cost prefer to be contacted each time a request for reproduction is made. I can be reached at the following address:

12 Partridgeberry Way
W. Henrietta, NY 14586

Date: June 2, 1987

The Design and Implementation of a System for Processing Documents Described in Generalized Markup Languages

by

Frank J. Cost

A thesis

Submitted to the faculty of the
School of Computer Science and Technology

at

Rochester Institute of Technology

In partial fulfillment of the
Requirements for the
Degree of

Master of Science in Computer Science

May 28, 1987

ACKNOWLEDGEMENTS

I would like to acknowledge the help of the following people: Patty Cost, Guy Johnson, Jeff Lasky, Mike Kleper, Ken Reek, Jim Hamilton, Emery Schneider, Herb Johnson, Robert Hacker, Archie Provan and Miles Southworth.

CONTENTS

APPENDICES

LIST OF FIGURES

ABSTRACT

This thesis proposes an architecture for a system capable of transforming the work of an author into printed form, and builds a partial implementation based upon that architecture. It allows an author to describe a document as a hierarchical structure and remain unconcerned about how his work will ultimately appear in printed form. The author is provided with a special markup language which can be used to tag the various logical parts of a document. The author must also be aware of how the various tagged parts fit together. The system is able to read the author's manuscript as input, and produce a typeset object as output.

1. INTRODUCTION AND BACKGROUND

This thesis is concerned with the mechanisms used to transform manuscripts into print. Ever since the invention of printing from movable types more than five centuries ago it has been necessary to supply detailed information along with a manuscript indicating how it should appear in final form. A printer would use this information to determine how to assemble individual pieces of type so as to create the requisite printing surfaces. These surfaces would then be used to transfer a light-absorbing substance called "ink" to a light reflecting surface called "paper" to create a printed artifact.

At first, information of this kind, called "markup," was in the form of instructions to be followed by human beings who were responsible for hand composing the assemblages of type. When the process of composition was automated beginning in the late 19th century, markup had to be translated into specific instructions which would cause a particular machine to set type accordingly. The expertise which had once been required of the hand compositor was now required of the person preparing input for the machine.

An author thinks of a manuscript as a logical construct. A book, for instance, is best seen as a hierarchical collection of parts. The graph of figure 1.1 shows the structure of a book as an author might imagine it. A book consists of a set of elements called "front matter" followed by another set of elements called "body" followed by a third set called "rear matter." The front matter consists of a "title page" followed by a "dedication page" followed by a "table of contents" followed by one or more optional "prefaces" followed by an optional "acknowledgements"

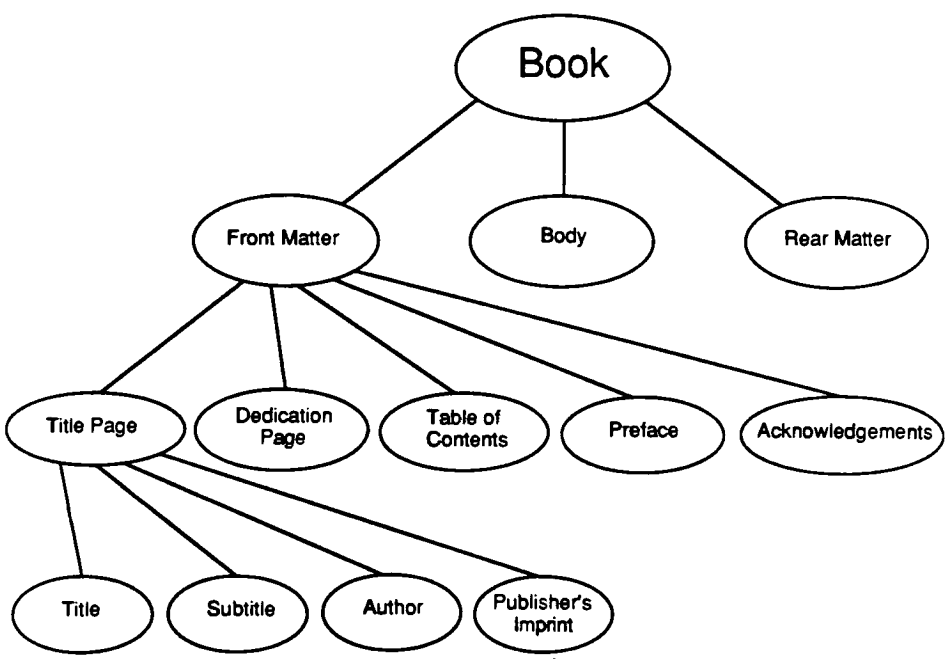


Figure 1.1 Partial Representation of a Book Structure.

section. Each of these elements can in turn be broken into yet smaller elements. A title page, for instance, consists of a "title" followed by a "subtitle" followed by an "author's name" followed by a "publisher's imprint."

This thesis proposes an architecture for a system capable of transforming the work of an author into printed form, and builds a partial implementation based upon that architecture. The system allows the author to describe a document as a hierarchical structure without any concern for how it will appear in print. The author is provided with a special markup language which can be used to tag the various logical parts of a document, along with an idea of how the various tagged parts can fit together. The system is able to read the author's manuscript as input, and produce a typeset object as output.

1.1 THE INVENTION AND DEVELOPMENT OF TYPOGRAPHY

Prior to the invention of printing from movable types, the composition of written images was accomplished either by manually applying ink to paper with a brush or pen, or by preparing a carved relief surface which was then used to transfer ink to paper. The former technique is called "calligraphy" and the latter is called "xylography."* Xylography and calligraphy differ fundamentally in that xylography allows multiple copies of the same image to be reproduced whereas calligraphy does not.

It is useful, however, to consider calligraphy and xylography as constituting a single paradigm. Both allow the freehand placement of light-absorbing material on a surface to form an image, and while it may be technically difficult to duplicate a calligraphic form by carving an image into a block of wood, it is conceptually possible to do so.

The creation of textual images from movable types is called "typography." Typography is not part of the same paradigm as calligraphy and xylography. It involves the assembly of images from discrete elements which can only fit together in certain ways. Figure 1.2 is a drawing of a single type. The face of the type is that part which transfers ink to paper in the form of a

---

*Calligraphy means literally "beautiful writing" (from the Greek kallos, meaning beauty, and graphein, meaning writing). Xylography means writing from wood (xylos) which can be carved to produce a relief printing surface.

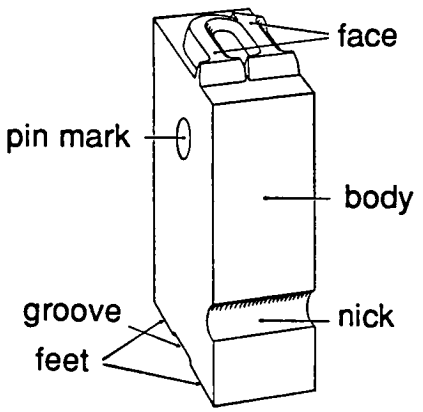


Figure 1.2 A single Piece of Metal Type.

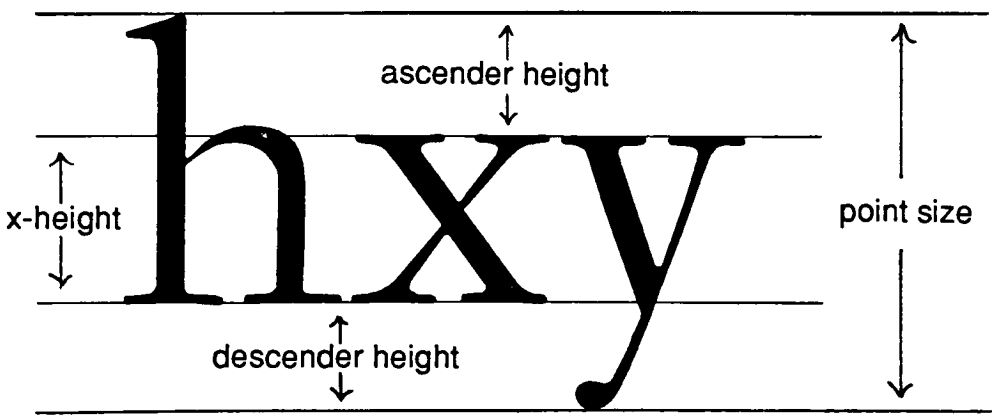


Figure 1.3 Some Important Dimensions of Type.

letter. It is situated upon a rectangular body that is slightly wider than the widest part of the letter. This dimension is called the "set width" of the type, and is variable within a particular alphabet or font.* The "point size" of a type is the vertical measure perpendicular to the set width in a plane parallel to the face. The point size of any given font is roughly the vertical distance from a line tangential to the top of the upper case letters, to a line tangential to the bottom of the lower case letters which descend below the baseline such as "y" and "p."** Figure 1.3 illustrates these dimensions of type.

Johannes Gutenberg, the inventor of typography, intended for the product of his printing press to closely resemble the style of calligraphy known as "Northern Gothic" or "blackletter." Another manuscript style known as "littera antiqua" or "whiteletter" was to become the model for typefaces used to reproduce secular works in most parts of Europe [ROSEN, p.8]. Both styles of handwriting incorporated many attached pairs of characters known as ligatures, and the early printers had the formidable task of creating fonts of type that included all of

---

*The "M" being the widest, and the "i" the narrowest in most fonts. In every font, which is a complete set of characters of the same size and type style, the width of the "M" is called an "em," and is the basis for a system of measurement of objects called "spaces" which are used to separate types from one another. A space with a width of one em is called an "em quad." A space 1/3 em wide is called a "3-em space," and a space 3 ems wide is called a "3-em quad."

**The American point system was developed as a way of standardizing typographic measurements. In this system, one point is equal to 0.0138 inches and 72 points is equal to 0.996 inches. On "pica" is equal to 12 points.

them. The font created by Gutenberg, which he used to print his famous Bible*, consisted of 299 elements, most of them two-character ligatures [RUPPELL]. Figure 1.4 is an illustration of Gutenberg's font. Only a few such ligatures survive in most modern fonts. These include ff, fi, fl, ffi, ffl, ae and oe [STEINBERG, p.30].

The elimination of ligatures served to reduce the number of characters in a font, and thus to reduce the effort required to produce a full set of characters. This also resulted in the rapid divergence of typography from its calligraphic foundations during the first century of printing. In the emerging typographic model, individual characters situated on separate pieces of type were assembled into words, lines, pages and groups of pages in that order.

It is not possible here to survey all of the relevant literature about typography. There is simply too much of it. However, a few classic works should be mentioned. The first is Joseph Moxon's Mechanick Exercises on the Whole Art of Printing, which was published in London in installments beginning in 1683 [MOXON]. For more than two centuries before Moxon's book appeared, the techniques of printing from movable types had been carefully-protected guild secrets. Moxon was the first writer to tell these secrets to the world. His book became the pattern for many others to follow. In 1825, a book entitled Typographia: An

*The first book printed from movable type was a Latin Bible now called the "42-line Bible" because it was composed of pages with 42 lines of text in two columns.

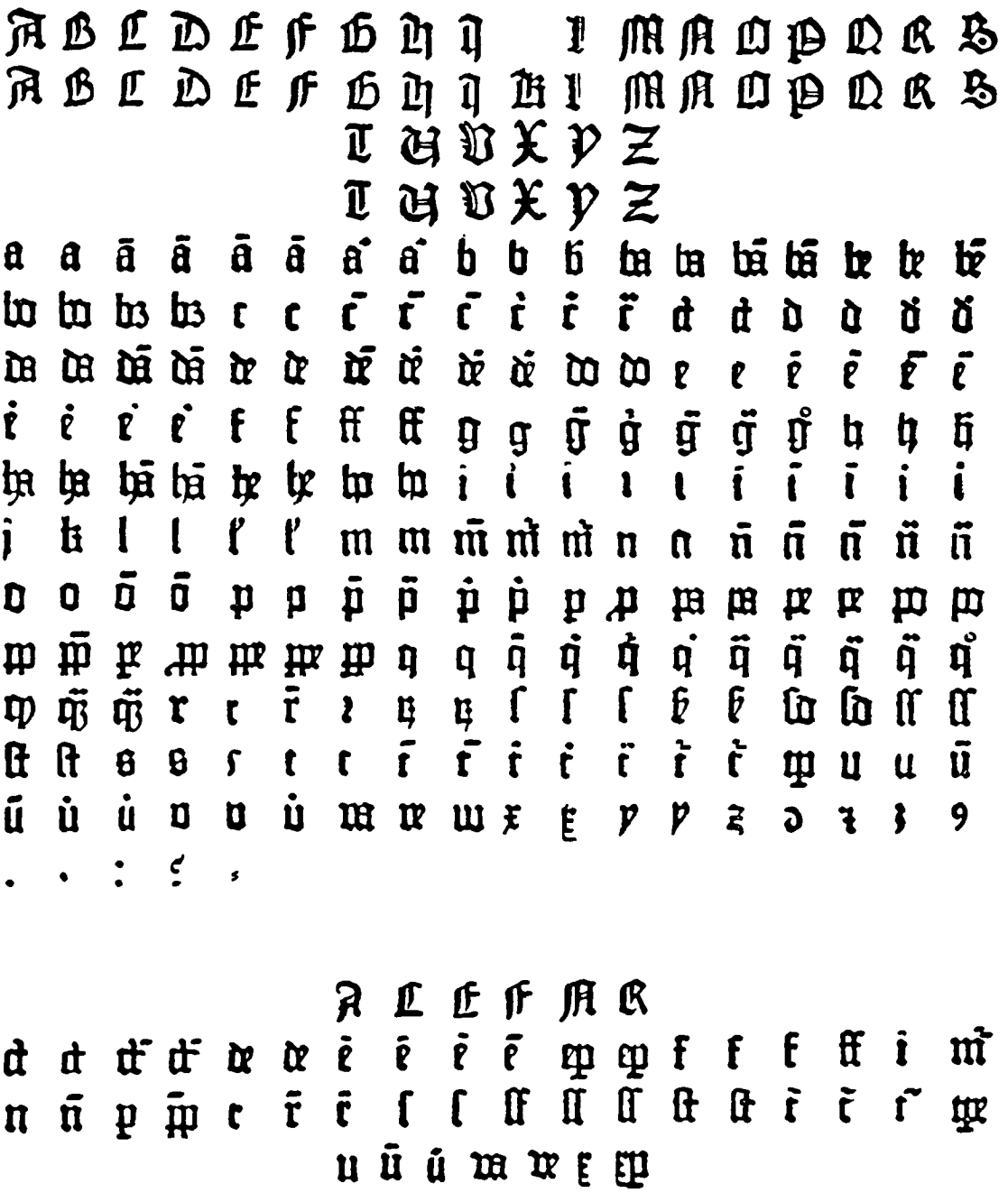


Figure 1.4 Gutenberg's Font containing 299 characters.

Historical Sketch of the Origin and Progress of the Art of Printing, by T.C. Hansard, was published in London [HANSARD]. This is considered a classic by most typographers, and is still valued as a reference on certain topics. Toward the end of the nineteenth century, in 1882, another book was published in London entitled Practical Printing: A Handbook of the Art of Typography, by John Southward [SOUTHWARD]. This was only four years before the debut of the first commercially successful type composition machine, the linotype.

From the time of Gutenberg's invention to Southward, a period of more than four centuries, not much had changed in the way pages of type were composed. Therefore, we can safely regard the whole body of work between Moxon and Southward as comprising a single resource, describing a single composition paradigm called "hand composition," or "hand typography."

1.2 MACHINE COMPOSITION

The first successful attempt to mechanize the composition process was the work of Ottmar Merganthaler [THOMPSON]. His machine came to be known as the "linotype" because it produced whole lines of letters cast on a single body called a "slug." A keyboard operator would enter a sequence of characters to be typeset. Each keystroke would release an appropriate brass letter mold, called a "matrix," from a magazine containing matrices for all the characters in a particular font. The spaces between words were provided by wedge-shaped adjustable "spacebands." When the line was almost full, the operator would decide where to break the line* and then the casting mechanism would adjust the spacebands to expand the line to its full measure by inserting extra space between all the words before casting the slug. A number of these slugs were then stacked together to form a column of type. A good source of information about the linotype mechanism is The Manual of Linotype Typography, published by the Merganthaler Linotype Company in 1923.

Less than a decade after the introduction of the linotype, a machine called the "monotype" was introduced. The machine used a punched paper tape to drive the casting unit. The system consisted of two machines--a keyboard and a casting mechanism--

---

*The line could be ended by breaking at the end of a word, or by breaking a word at an appropriate point and inserting a hyphen. On the original linotype machine, these so-called "end of line decisions" were made manually by the operator.

each requiring an operator. The keyboard operator would prepare a perforated tape which would then be used to control the casting mechanism. The tape was fed into a pneumatic "reader" which would in effect decode the character information on the tape, and cast a series of types accordingly. This was accomplished by positioning a two-dimensional array of brass matrices containing the entire font of characters over a cavity into which molten type metal was injected. The dimensions of the cavity were adjusted for each letter to provide for variable set widths within a font. The individually cast letters were then assembled in a channel until a line was completed. The monotype was a more flexible mechanism than the linotype because it preserved the separation of individual characters. A hand compositor, working with the output of the monotype, was still able to make subtle changes to the word and letter spacing to improve the appearance of the type. The linotype, on the other hand, had a larger granularity. The solid line of type, once cast, could not be altered.

The first photocomposition machines employed the mechanisms of the monotype and the linotype respectively. The array of brass matrices in the monotype was replaced by an array of film negative images of letters in the "monophoto." The casting unit was replaced by an exposure unit. Paper tape was still used to control the movement of the array of film images. Photographic paper was transported through the machine in such a way that selected letters would be projected onto successive positions on

the paper to form lines of text. The linecaster mechanism was employed by a machine called a "Fotosetter." The brass matrices of the original linotype were now used as holders for film masters of all the letters in a font. These two machines together comprise what is called the "first generation" of photo-composition technology.

Eventually new mechanisms were devised departing from the linotype and monotype designs altogether. The "second generation" of machines still stored their fonts as film images, usually on rapidly rotating disks or drums. These machines were designed to read input from punched paper tape or from magnetic storage media, and produce output on photographic paper or film. "Third generation" typesetters are machines that store their fonts in either optical (film master) or digital form, and use special high-resolution cathode ray tubes to output images onto film or paper. "Fourth generation" machines use a scanning laser beam in place of the cathode ray tube.*

As new technologies have been applied to typography, the aim has always been to produce images which could not be distinguished from those produced by previous methods. Printing itself was born out of a desire to mass-produce manuscripts--to build a machine capable of doing calligraphy. Often, new technologies exert subtle but insistent pressures on graphic conventions that had been associated with the older ways. The

---

* An excellent source for descriptions of typesetting machines and composition systems from earliest days to present is The World of Digital Typesetting, by John Seybold [SEYBOLD].

desire to reduce the number of elements in a type font, for example, resulted in the elimination of most ligatures soon after the invention of typography. Ligatures had originally been employed by calligraphers to reduce the number of pen strokes necessary to write a given number of words. Typography exerted pressure in the opposite direction. Production of a font of type was extremely expensive. Fewer characters in a font meant less expense. Ligatures were nice, but ultimately not nice enough to justify their cost.

Stylistic changes that happen as a result of technological developments have almost always been resisted vigorously upon their initial proposal. This is because of the extreme conservatism of the human eye regarding the written word.* Modern typographic composition systems are still heavily influenced by the model of composition that emerged in the first century of printing from movable types. We must therefore turn to a discussion of the traditional model, and to how it has been adapted and changed by new technologies.

---

*The output of 9-pin dot matrix printers is not allowed for final copies of M.S. Theses in the School of Printing at Rochester Institute of Technology because it departs too radically from accepted form. There has even been some discussion as to the acceptability of typeset copy. Masters theses have traditionally been typewritten, and therefore do not look "authentic" if produced in any other way. Needless to say, very strong pressure is being brought to bear on these kinds of restrictions.

1.3 THE TRADITIONAL MODEL OF TYPOGRAPHIC COMPOSITION

A font* of type is a full character set** in a given typeface and point size. A metal font includes duplicate characters in proportion to frequency of use (more e's than z's), and therefore is not a "set" in the formal sense of the word.*** Henceforth we shall use the term "font" to mean a formal set of characters in a certain typeface in a certain point size. We shall name a font by giving both the size and the name (e.g. 12 point Helvetica Medium).**** The font name includes a designation of weight if the typeface comes in more than one. Weight is the relative boldness of the typeface. Typefaces may vary in weight from "extralight" to "ultrabold" but there is no absolute meaning to any of these terms. Generally, the heavier the weight of a

---

*Originally called a "found" or "fount" which means literally "something made in a foundry." [MCLEAN, p. 72]

**A character set includes upper and lower case characters, small capital letters (upper case letters which are as tall as or slightly taller than the x-height of the lower case characters), accented letters, ligatures, figures (0-9), punctuation marks and special symbols such as &, $, % etc.. In addition, spaces are included in a font. These range in thickness from a fraction of the thickness to several times the thickness of the letter "M."

***The concept of a digital font is different in two important ways from the traditional meaning of the word. First, digital fonts require only one master for each character in the set. In this regard a digital "font" has more of a relation to a set of matrices or letter molds than to the letters themselves. Second, a digital font may be used to create letters in more than one point size. This , of course, is impossible with metal type or matrices.

****In practice a font is named also by its manufacturer, since there are subtle but visible differences among fonts of the same name created by different manufacturers.

typeface, the darker its appearance on the page will be. Figure 1.5 is an illustration of a font of 36-point type.

An important characteristic of a type face is the "set width" of the face. Each character in a font has a width expressible as a whole fraction of the width of the letter "M" or "m." A common system divides the width of the M (or "em") into 18 equal units. Each character in the font has an integer width less than or equal to 18 of these units.* The width of the em in a typeface of "normal" width is the same as the point size. A 12-point "M" in such a typeface will be 12 points wide.

Every typeface has an important number associated with it called "characters per pica." This is the average number of characters that will fit into one pica of horizontal space based on normal English usage. Within a given point size "narrower" typefaces will have a larger character per pica measure than "broader" typefaces. By obtaining an accurate count of characters in a manuscript, character per pica information can be used to determine the minimum number of lines of a given line measure required to set the manuscript in a particular typeface. This number, divided by the number of lines of text on a page, will yield the number of full pages of text in the typeset version of the manuscript. The number of lines of text that will fit on a

---

*In a typical font, characters such as "A", "D" or "G" might be 16 units wide, "B", "C" and "E" might be 14 units wide, "b" and "d" might be 12 units wide, etc.. In a normal 12 point typeface, the width of the em is 12 points. If the set widths of the font are based on an 18-unit system, the value of a single unit is 12/18 of a point or 0.0092 inches.

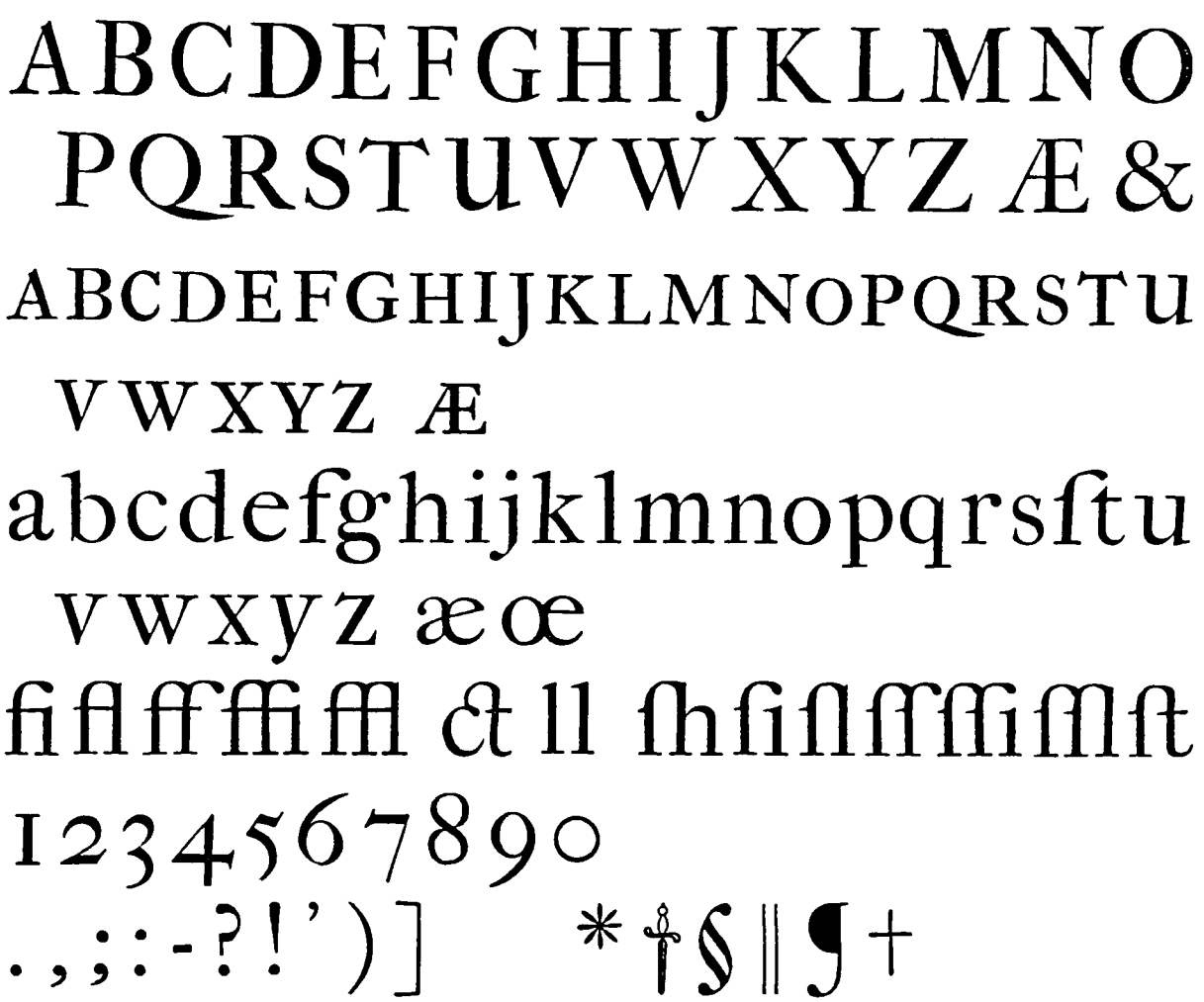


Figure 1.5 A font of a 36-point type called "Canon" dating from the 17th Century.

page is dependent upon three factors; point size of the typeface, interlinear spacing or "leading" and the vertical dimension of the space on the page to be occupied by text and/or other image elements.* This vertical dimension is called the "depth" of a page.

---

*The designer makes this decision. As an example, if a designer determines that the vertical dimension of print on a page will be 6 inches, and if the type chosen for the text is 11 point Times Roman with one point of leading, 36 lines of text will fit on the page.

1.3.1 HIERARCHICAL NATURE OF TYPOGRAPHIC STRUCTURES

It is useful to view the relationships among the various elements of a typographic object in a hierarchical manner. Let us take as an example a paragraph structure and analyze it in this way. We will assume that the paragraph exists as a unified entity on one page. On the lowest level, individual characters are bound together to form words. If we assume that the entire paragraph is set in the same typeface and point size, the only variable in the inter-character intra-word relationship is a linear measure called "letter spacing." The lower limit to this value in traditional composition is established by the widths of the metal bodies upon which the characters are situated, although in special cases it is possible to further reduce the spacing between characters by "mitering" the bodies of adjacent characters. In modern systems which store their fonts optically or digitally it is much easier to reduce the inter-character spacing. This practice is called "minus setting" or "kerning," depending respectively upon whether it is done globally or locally.* In some systems it is possible to introduce such a degree of minus leading that adjacent characters overlap.

The upper limit to spacing between characters is related to the amount of spacing between adjacent words. If the inter-

---

*Minus setting is the reduction of all inter-character spacing uniformly in a text. Kerning is the selective adjustment of spacing between specific adjacent pairs of letters. Kerning, therefore, requires more intelligence.

character spacing is equal to or exceeds the inter-word spacing, words cannot be distinguished from one another. Thus inter-character spacing must always be strictly less than inter-word spacing. The spacing between words is the next level in the hierarchy. This value is usually established globally within a line of text, but may vary slightly from one line to the next.

The variation in line length within a paragraph is the next level in the hierarchy. If the paragraph is set "justified" this value is zero. Each line must be the same length. If the paragraph is set "ragged" this value has an upper limit of the total line measure, and a lower limit established by the degree of raggedness allowed. In a line measure of 27 picas (approximately 4.5 inches) a lower limit of 25 picas will result in a less ragged setting than a lower limit of 23 picas. A justified setting can be viewed as a ragged setting with the lower limit equal to the line measure.

At the highest level in the structural hierarchy of the paragraph is the interlinear spacing or leading. In the traditional model leading is the amount of additional space inserted between adjacent lines of text. Lines set with no leading between them are said to be "set solid." In modern electronic composition systems leading is the vertical measure from baseline to baseline of two adjacent lines of text. The lower limit in this case is the point size of the typeface. In some systems it is also possible to use "minus leading" which is

leading less than the point size, but this is rarely done because ascenders and descenders begin to overlap.

Looking at this structural hierarchy from the top down we can define a paragraph as one or more lines of text separated by a constant amount of leading between each line. Each line has a length less than or equal to the line measure of the paragraph. A line is defined as one or more words* separated by word spaces. A word is defined as one or more characters (not including blanks) separated by letter spaces.

An underlying assumption in the above discussion has been that a paragraph exists in a unified form on a single page. But a paragraph may take other physical forms. For example, a paragraph may be interrupted at the end of a page and continued at the top of the next page. Such a paragraph would not conform to the definition given above. The problem here is that a paragraph is a "logical" structure, and we have attempted to define it as a "physical" structure. Understanding the differences between the logical and physical structures of a document and how they are related is the central task of this thesis.

When a page or part of a page is hand-composed in metal type, the compositor is acting as a language interpreter. His input is called "marked-up copy." The copy combines the actual text to be set along with directions that specify the design. Perfect copy specifies almost completely how the finished page or page segment

---

*A "word" in this context may be a hyphenated part of a word if it occurs at the end of a line.

is to appear. The compositor is left a few decisions--primarily end of line decisions (how to fill out a line and where to break words). The marked up copy is both code (how to set) and data (what to set) combined.

What sort of information is conveyed by markup? Let us again refer to our example of a single paragraph. The minimum information a compositor needs to typeset a paragraph is as follows:

1. The amount of indentation of the first line.*

2. The measure or maximum length of each line.*

3. Whether each line is to be set to the full measure or whether some degree (and if so, how much) of raggedness or variability in line length is allowable.**

4. Whether or not words can be broken (hyphenated) at the end of a line.

5. The choice of typeface.

6. The point size of the typeface. Typeface and point size

---

*This is normally expressed in picas, but could also be expressed in centimeters or inches.

**The last line in a paragraph is never expanded in length to fill the measure regardless of the degree of raggedness called for.

have traditionally been viewed as a unified parameter.*

7. The leading or interlinear spacing.

The compositor takes this information along with the text to be set and builds a typographic structure.

> "The fundamental task of hand typesetting is the assembly of types. Each individual character or symbol is picked by hand from the case and placed into a composing stick. When a take of perhaps a half dozen lines has been assembled it is carefully removed from the stick and deposited into the galley--an oblong metal tray with an edge around it about a half-inch high" [SEYBOLD, p. 31].

The compositor builds rectangular assemblages of type which can be "locked up" together into a "chase" that can then be used as a printing surface. The chase itself is the highest level in the structural hierarchy of a composed page or set of pages. It is built out of lower level rectangular structures. A four-page form consists of four single-page rectangular forms. Each of these in turn consists of several lower level rectangular forms. A page, for example, might consist of a rectangular form containing the header of the page which might contain the title of the book on

---

*This is because in metal, certain typefaces may not be available in certain sizes. In modern systems with digital font masters, it is more appropriate to conceptually separate a typeface from its point size, because all sizes within a range are normally available.

even numbered folios or the title of the chapter on odd numbered folios, along with a folio number. Other rectangular elements on the page might consist of paragraphs, section headings, footers, etc. Each of these is built of smaller rectangular elements (lines of type).

Donald Knuth has created a typesetting language called "TeX" built on these principles. TeX [KNUTH] is a procedural (block structured) typesetting language that is useful especially for typesetting difficult mathematical texts. Letters, words, paragraphs, etc., are viewed as nested "boxes" which are "glued" together to form pages. TeX parameterizes the glue between the various levels of boxes in such a way that more or less variability in the spacing between boxes at each level is allowed. This has allowed the design of intelligent composition systems using the TeX model as the underlying mechanism. A system was defined by Michael Plass for determining the optimal pagination scheme for any given set of text and pictures (also treated as rectangular boxes). Plass defines a methematical function called the "badness function" which depends upon how a particular text is broken into pages. "This function ideally has the property that the less readable the book is, the greater the value of the function will be. Of course, such a function cannot hope to capture all the nuances of readability.... The badness function may depend on the distribution of white space, the placement of illustrations relative to their citations in the text, and whether the page breaks come in logically desirable

places" [PLASS, p. 8]. Plass then defines an algorithm which is guaranteed to minimize the value of this function in a "reasonable" amount of time. He also shows how pagination problems can grow in complexity to the point of being provably unsolvable in polynomial time (NP-Complete pagination problems) [PLASS, p. 42].

Pagination problems are at the highest level in the structural hierarchy of a document. We will now turn to a discussion of those parts of the composition process where artificial or machine intelligence has been successfully applied, starting at the lowest level and working upward in the hierarchy.

1.6a

1.6b

Figure 1.6 Example of kerning between the first two characters in the word "Typography." 1.6a is unkerned and 1.6b is kerned.

1.4 AUTOMATION OF COMPOSITION FUNCTIONS

Each character or symbol in a typographic composition occupies a rectangular space or "box" in Knuth's terminology. These are the smallest elements or "atoms" in the structural hierarchy.* The spacing of these atoms in relation to one another is variable. Space can be added (inserted) or subtracted between any two characters. In the traditional model the "default" spacing was set by the widths of the types themselves. It could be increased by inserting metal or paper spacers between the types, or, in some cases, decreased by cutting away the adjacent type bodies.

1.4.1 KERNING

The reduction of space between selected pairs of types requires that intelligence be applied at the lowest level in the hierarchy. This means that a decision can sometimes be made to reduce the space between a pair of types without reference to any outside structural elements (other characters in a word, etc.). Figure 1.6 shows the word "Typography" before and after the space between the initial two letters is reduced. The "Ty" letter pair

---

*In the traditional model, a single character is indivisible. Bitmapped characters in digital composition systems are in fact composed of smaller elements called "pixels," and are therefore no longer indivisible in the strictest sense of the word.

can almost always be "kerned" in this way, without regard for the surrounding context. Thus "Type," "Typical," "Typographic," etc. would all benefit from an identical treatment.

The kerning of letter pairs as explained above only requires that an algorithm have the capabilities of looking ahead in the input one character.* The addition of space between letters is, on the other hand, a retrospective decision made to resolve the problem of justifying a line of type.

1.4.2 JUSTIFICATION AND HYPHENATION

With each line of text a compositor sets, a decision must be made as to how the line will end. This is called the "end of line decision" and involves a more or less complex weighing of factors. The simplest algorithm is as follows:

1. Start setting the line, keeping track of the accumulated values of the character widths and spaces.

2. When this value exceeds the measure (maximum line length) move the current word to the beginning of the next line and continue.

---

*Kerning can be more sophisticated than this. Take, for example, the words "WATER" and "WATT." The amount of space removed between the "WA" and the "AT" pairs in either case might also depend upon the remaining letters in the word. "WATER" requires a greater degree of kerning because of the tightness of the "TER" combination.

This algorithm yields a ragged setting with a degree of raggedness dependent upon the width of the longest word encountered (If the last character in a very long word overflows the measure, the entire word is moved to the next line). If a maximum degree of raggedness (minimum line length) is imposed, the algorithm must be altered. When the measure is exceeded, the current word can be moved to the next line, and the current line spaced out so that it is equal to or exceeds the minimum line length. This spacing can be accomplished by inserting spaces between the remaining words on the line. As the minimum line length approaches the full measure (as the degree of allowable raggedness is reduced), the variability in the word spacing from one line to the next will increase.* As this variability increases, legibility suffers. Although it is possible to insert spaces between individual letters (letterspacing) to reduce the size of interword spaces, noticeable letterspacing is itself considered a detriment to legibility. The solution to this problem is to allow words to be divided at the end of a line. This is called "hyphenation."

There are two approaches to hyphenation. One is to look up a word in a dictionary containing information about where the word

---

*The minimum line length is increased, and therefore the amount of space inserted between words to cause the line to reach the minimum will also increase. Variability in interword spacing among lines also increases as the measure decreases. Short lines with no raggedness allowed vary the most in terms of word spacing.

can be divided with a hyphen.* The other approach is to determine hyphenation points algorithmically.

> "As a rule such algorithms do not examine the entire word, but only inspect a portion of it. In fact, one of the most common algorithmic tests is to ascertain whether a hyphen may or may not be placed between the second and third of a string of four characters. The analysis may not begin at the back end of the word, or the front either, for that matter. The program may start with a group of four characters including the last two which, when added to the value of a hyphen, would barely fit on the line, and adding to the string the following two "fall-off" characters." [SEYBOLD, p. 204]

The hyphenation point occurring in the middle of this character string is then examined. If it is determined that a hyphen can occur at that point, the hyphen is inserted and the remainder of the word is carried to the beginning of the next line. If a hyphen cannot occur at that point, the four-character window is moved to the left one character in the input and the same analysis is done. An excellent discussion of this subject occurs in [SEYBOLD, pp. 204-214].

---

*In American English, these "hyphenation points" are based upon pronunciation, whereas in British English, they are more likely to be etymological.

1.4.3 PAGINATION

Pagination is the process of making up fully composed pages of text and other elements. Each page is a unit area, and a composition system capable of pagination or "area composition" will fill the area to create a page. The simplest case is the setting of straight sequential text. Each page can hold a certain number of lines, and when the last line on a given page is set, the system starts a new page.

Pagination becomes complex when elements whose position is variable in the text such as illustrations, examples and footnotes must be placed close to the point in the text where they are referenced. There have been successful attempts to formalize this process, although as we have previously seen, the complexity of some pagination problems makes them virtually unsolvable [PLASS]. All require an ability to look ahead several pages in the input before they are actually typeset to determine the optimal placement of the "floating" elements.

1.5 THE EVOLUTION OF MARKUP LANGUAGES

The traditional meaning of the term "markup" is information added to the content of a document specifying how the document should appear on the printed page. The following is an excerpt from the University of Chicago Manual of Style regarding the way type is to be specified in a manuscript:

> "In many publishing houses...the editor marks the manuscript with type specifications indicated on the designer's layout or from a list of specifications furnished by the designer. Care must be exercised to follow the specifications exactly and to mark like parts alike. The layout does not show all parts of a book but does give a sample of each type size to be used in text and display matter....The type size, leading, typeface, and type width to be used in the text proper should be written in the margin of the first page of each chapter or other division of text. The type size and leading and amount of indentation, if any, should be placed beside the first extract in each chapter. Poetry extracts, unless the design specifies otherwise, are generally marked "center on longest line." If all material to be set as extracts is marked carefully with a red vertical line at the left, it is not necessary to mark type sizes for extracts after the first in each chapter...." [CHICAGO, p. 53]

The emphasis in this style manual, as in all others, is on consistency and unambiguity. Markup is superimposed on the text and runs in parallel with it. A universal example of this

parallel relationship between a text and its markup is the wavy line underneath a portion of text to be set in a boldface version of the text font.

When typesetting is to be performed by a machine operating in a batch mode, the text and markup must enter the machine as a single serial stream of input. In the following chapter a batch-oriented typesetting language will be described. This language will be used as the input language of a typesetting system in the experimental portion of this thesis.

2. A BATCH-ORIENTED PROCEDURAL TYPESETTING LANGUAGE

Typographic output for this thesis project was produced on a Compugraphic Modular Composition system 10/100 (MCS) driving a Compugraphic 8600 ImageSetter phototypesetter.

> "The 8600 Imagesetter uses stored digitized font information to produce character images on photosensitive materials....The MCS front-end system transmits files to the 8600 ImageSetter as coded numbers. The information is translated into program instructions for the photo unit, which uses an electronic scanning beam to construct the type characters on the face of a cathode ray tube (CRT).* Photosensitive paper or film is moved across the faceplate of the CRT, exposing its emulsion-coated surface to the constructed character images, one at a time. The exposed material is then fed into a take-up cassette. At the completion of typesetting the file, extra material is advanced, then the cassette is removed and carried to a processor for development." [COMPUGRAPHIC]

---

*Each character is stored in the memory of the computer as a bitmapped image which is referenced by the system to control the movement of the scanning beam on the faceplate of the CRT.

Since the files input into the typesetting system were produced on a Pyramid 90X computer that was physically removed from the typesetting site, a bridge between the two systems had to be established. Files from the Pyramid were transmitted over a modem to an IBM PC/MS-DOS system. A program called TRANSFER* running under MS-DOS was then used to move files (as well as to perform some translation) to floppy disks formatted for use on the MCS system. The MCS typesetting command language is partially hidden from a user of the TRANSFER program, which has its own command language. The TRANSFER commands are mapped to MCS commands when the files are moved from the MS-DOS format to the MCS format. Both command languages are functionally equivalent, and therefore this translation function is not significant. Henceforth this discussion will refer to the TRANSFER command language as the MCS command language.

---

*TRANSFER copyright 1985 by Bryan D.K. Biggers and the 'Puter Group.

2.1 ELEMENTS OF THE MCS COMMAND LANGUAGE

We will now turn to a discussion of the MCS command language. A full description of these commands can be found in appendix A. Here we will concentrate on an explanation of the main categories of commands and give some examples of each.

2.1.1 PARAMETERS

Parameters include typeface, point size, leading and line measure. When the value of a parameter is set, it remains thus until changed. Parameters are delimited by open and close angle brackets. <FTn> (or <ftn>) informs the system that characters are, until further notice, to be taken from font n. <PSn> informs the system that the point size of characters will be n points. <LSm.n> establishes the leading between baselines of contiguous lines of text set as m and n/4 points. <LLm.n> establishes the line measure as m picas, n points.

This is a demonstration of the independence of the *point size* of a typeface and the *leading* or interlinear spacing. The point size is 14 and the leading is 12. This results in an overlap of lines.

Figure 2.1 A "minus-leading" setting showing overlap.

It should be noted that the selection of point size and the selection of leading are independent of one another. A selection of point size and leading of <PSm> and <LSn> where m > n will result in overlapping of successive lines. Figure 2.1 shows the effects of setting a 14 point typeface with only 12 points of leading. The four parameter changes "<FT21> <PS12> <LS13> <LL25.6>" have the following four effects:

1. **<FT21>** indicates characters are to be selected from font 21, whatever typeface that may be.

2. **<PS12>** indicates the point size is 12 points.

3. **<LS13.3>** indicates the leading from baseline to baseline is 13.75 points.

4. **<LL25.6>** indicates the line measure is 25 picas and 6 points (or 25.5 picas).

2.1.2 MNEMONIC CODES

Mnemonic codes allow functions or characters not available as singular keystrokes on a keyboard to be accessed. These include such typographic elements as fixed spaces, quad commands*,

---

*Quadding refers to the filling out of lines of type with spaces to bring the line to the full measure. Quadding left refers to the addition of space to the right hand side of a line (and thus creating a line that is pushed up against the left hand margin). Quadding center refers to the addition of equal space on

baseline rules, tab commands, etc. Figure 2.2 shows the effects of quadding a line left right and center. Most of the mnemonic codes are preceded by a single percent sign. Some examples of mnemonic codes are:

%1 = thin space. A space with the same width as most punctuation marks such as a period, comma, colon, etc..

%n = en space. A space with half the width of the letter "M."

%m = em space. Twice the width of the en space.

%c = quad center.

%r = quad right.

%% = quad left.

This line is quad left.

This line is quad center.

This line is quad right.

Figure 2.2 Demonstration of the effects of quadding.

---

the left and right sides of a line to place the line in the center of the measure. Quadding right is the opposite of quadding left.

A few mnemonic codes are delimited by open and close angle brackets in the same way that parameters are. An example is the "<IS>" command which inserts space at the point where it appears in a line. This provides a way of pushing elements in the line out to the full measure by inserting space between them.* The line "WORD1<IS>WORD2" would be typeset with WORD1 pushed against the left margin and WORD2 pushed against the right margin.

2.1.3 REFINEMENTS

Refinements are switches for some special features of the MCS typesetting system. These include turning hyphenation on and off, placing discretionary hyphenation points within words, causing the typesetter to slant characters to a certain degree to produce pseudo italic type, and changing the set size of the type to condense or expand a particular typeface. They can occur anywhere in the input stream.

2.1.4 POSITION COMMANDS

Position commands allow control over the setting of complex tabular material. These include such things as indents, tab settings, and auto quadding (which result in every subsequent

---

*sometimes referred to as "quadding middle."

line of text being quadded accordingly). Position commands are delimited by open and close angle brackets and can be in upper or lower case. Some examples of position commands are as follow:

<IBm.n> Indent both sides m picas and n points.

<ILm.n> or **<IRm.n>** Indent left or right sides.

<IHm.n> Hanging indent. Every line except the first line is indented from the left until a carriage return is encountered.

<AL> Automatic quad left. Every subsequent carriage return results in a quad left command.

<AR> Automatic quad right.

<AJ> Automatic justify. This is the normal typesetting mode, and is used to turn off previous automatic quad commands.

Other features of the system can be found in Appendix A, and will not be discussed here.

The MCS command language is a classic markup language for automatic typesetting applications. It is strongly patterned after conventions of markup originally developed for hand composition. This type of markup describes the physical structure of a document to the output device, and is sometimes called "procedural markup." Since it is only concerned with physical

appearances, and since appearances often deceive, procedural markup is not designed to clearly demarcate the various elements of a document in logical terms. Let us consider the following piece of marked up text:

<PS18><LL45><LS20><FT21>THE QUICK FOX JUMPED%c

The point size is larger than a text size, the content is set in upper case, and the line is centered in the measure. This may be some kind of title, but there is no way to know for sure. The markup does not describe what it is, only how it will appear. Furthermore, if other elements in the same document are marked up in the same way, there is no way to know whether they are related to this one. The best one could hope to do would be to make some intelligent guesses about how each element relates to the others based on presumptions about how documents are normally structured.

> "Procedural markup is also inflexible. If the user decides to change the [typographic] style of his document (perhaps because he is using a different output device), he will need to repeat the markup process to reflect the changes. This will prevent him, for example, from producing double-spaced draft copies on an inexpensive computer line printer while still obtaining a high quality finished copy on an expensive photocomposer. And if he wishes to accept competitive bids for the typesetting of his document, he will be restricted to those vendors that use the identical text

> processing system, unless he is willing to pay the cost of repeating the markup process." [SGML, p. 60]

In the next section we will investigate another approach to the marking up of documents generally known as "descriptive markup." Instead of describing how the various physical pieces of a document will appear when output, descriptive markup treats a document as a collection of logical entities. It is not concerned with how these entities are to appear in typographic form.

3. DESCRIPTIVE OR GENERALIZED MARKUP

The need for descriptive markup* arises out of the inability of device-specific procedural markup to convey information about the logical structure of documents. A document marked up in a procedural manner has no other use, unless altered in some way, than as an input string to the particular composition system for which it was created. Descriptive or "generalized" markup, on the other hand, describes the logical structure of a document without saying anything about how that structure is to be displayed. A paragraph, therefore, might be described as follows:

<P> This is a paragraph....

Nothing is said about the font, line measure, point size or leading. All of this information is considered to be application specific, and is not part of the logical document. In one application, this paragraph might be printed in ten-point

---

*Descriptive or generalized markup is also called "content structure coding" in some sources [DOEBLER, p.30].

Helvetica Medium with two points of additional leading in a line measure of 32 picas with a paragraph indent of one em. In another application, these parameters might be completely different, but the object would still be a paragraph. So the physical structure can change without affecting the logical structure.

3.1 A SIMPLE PROCESSOR FOR DESCRIPTIVELY MARKED UP DOCUMENTS

In this section, a simple processor for translating a descriptively marked up or "tagged" document into a form which can serve as input into a conventional typesetting system will be proposed. We can conceive of this processor as a simple translation filter. Input to the translation filter is a single stream of characters consisting of tags and content elements. Tags are delimited by less-than and greater-than symbols. Everything not within these delimiters is content. The translation filter reads the input stream character by character and passes content elements directly to output. When a tag is encountered, the tag is processed and the result passed to output.

The character string "<P>" in the above example serves as a label or tag for the content that follows.* A simple program could be designed to map markup in this form to device-specific procedural typesetting commands. Each occurrence of a tag would be replaced by a specific string of typesetting commands. Each tag would serve as a form of macro-instruction and would be expanded by the processor into a set of appropriate device instructions. The relationship between some tags and their expansions** is shown in figure 3.1.

---

*The less-than symbol is being used as a "tag-open" symbol and the greater-than symbol as a "tag-close" symbol.

**Typesetting commands are in the MCS command language described in section 2.

TAG	REPLACEMENT STRING
<P1>	<LL32><PS12><LS13><FT21>%m
<P2>	<LL32><PS10><LS11><FT21>%m%m
<P3>	<LL32><PS8><LS9><FT22>
<T1>	<LL32><PS24><LS28><FT41>

Figure 3.1 Replacement table for markup expansions.

Each of these tags, labelled P1, P2 and P3, is a paragraph tag which when expanded will result in paragraphs formatted in three different ways. The formatting of P1 paragraphs might be applied to the first paragraph in a chapter of a book. Subsequent paragraphs might use the formats associated with P2. Paragraphs tagged "P3" are formatted in yet another manner.

There is a subtle problem with this scheme. Let us assume that the tag "T1" marks the beginning of a title somewhere in a document. In most typesetting languages, in order to typeset this title quad center (centered in the measure) it is necessary to insert the quad center command after the content of the title and not before. For example, in the MCS command language, to quad center an element labelled "T1," the full expansion of the string "<T1>THIS IS A TITLE" must be "<LL32><PS24><LS28><FT41>THIS IS A TITLE%C<RETURN>". The "%C" is the MCS command to set the preceding line quad center, and "<RETURN>" is the carriage return.

The expansion of tagged document elements into procedural typesetting command structures requires the process responsible for doing so to be able to recognize the beginning and end of the element to be typeset. This can be accomplished most easily by placing a tag at both ends of the content. Each tag is then expanded into the appropriate commands independently. Alternatively, the beginning of any tag other than the first tag in a document can be used as a marker for the end of the preceding element, although this precludes the nesting of elements inside other elements. This method also requires a more powerful translation mechanism than if the end of each element is tagged. Since some elements will require end processing and others will not, the simple substitution table of figure 3.1 must be modified to handle the conditional aspects of the expansion. Figure 3.2 shows a modified table that would work with a translation algorithm capable of processing a document with tags only at the beginnings of content elements. This table divides the expansion of each tag into that which precedes and that

TAG	PRE-CONTENT EXPANSION	POST-CONTENT
<T1>	<LL32><PS24><LS28><FT41>	%C<RETURN>
<T2>	<LL32><PS18><LS20><FT31>	%L<RETURN>
<X1>	<LL32><Ps10><LS10><FT33>	null

Figure 3.2 Modified translation table for markup expansion.

which follows the content of the tagged element. This table, along with a global variable to hold the value of the current tag could be used by a processing algorithm that would operate as follows:

1. Read input until a tag is encountered passing input to output.

2. When a tag is encountered do the following (A through D):

    A. get the current tag value.
    B. write the post-content commands associated with that tag to output.
    C. update the current tag value.
    D. write the pre-content commands associated with the new current tag to output.

3. Continue with step 1.

This added complexity eliminates the need to explicitly tag the end of each content element, but results in a less powerful mechanism because of its inability to handle nested elements.

The class of typesetting commands referred to as "parameters" in the MCS command language discussed in the previous section establish values for such things as line measure, point size and typeface. Parameter typesetting commands have the effect of assigning values to global variables which, when taken as a set, define the state of the typesetting system at any time. The

value assigned by a parameter typesetting command to a particular variable remains thus until changed by another assignment. Before the first such assignment occurs in a document, each variable has a value by default. The processor described above, based upon simple macro substitutions of typesetting commands for tags, introduces redundancies into the output stream whenever a tag expansion results in changes to only a proper subset of the full set of parameters. These redundancies can be eliminated if the system responsible for expanding the tagged document into a suitable form for input to the typesetting system maintains its own set of variables corresponding to the typesetting parameters. Each parameter assignment in the pre-content tag expansions can then be compared with an existing value in this variable set and written to output only if different.

3.2 WEAKNESSES OF THE DESIGN

Each of the descriptive tags serves as a simple label for the content that follows. In the case where only the beginning of each element is tagged (and therefore no nesting of elements allowed) any sequence of elements in a document is legal. In the case where both the beginning and end of each content element is tagged, any arbitrary nesting of elements is also legal. Thus it is perfectly legal for a title, for instance, to contain within it a nested paragraph structure.

Since there are no rules defining the legality of relationships among the various elements of a document, a person using such a markup scheme must build a reasonable structure from scratch when creating a document. Some external notion of how elements can and cannot be related to one another must be brought into the process. In other words, rules are needed.

3.3 DOCUMENT STRUCTURE

The structure of any document can be described using a common abstract notation. If we take a typical book as an example, it is possible to describe the structure of the book as follows: the book consists of a set of elements called "front matter" followed by another set of elements called the "body" followed by a third set called "rear matter." The front matter consists of a "title page" followed by a "dedication page" followed by a "table of contents" followed by one or more optional "prefaces" followed by an optional "acknowledgements" section. Each of these elements can in turn be broken into yet smaller elements. A title page, for instance, consists of a "title" followed by a "subtitle" followed by an "author's name" followed by a "publisher's imprint." In this way, each element in a book (except for the individual characters) can be seen as a collection of subelements and (except for the element which is the book itself) as part of some superelement.

A language called "Standard Generalized Markup Language" or "SGML" has been developed to allow a structure to be specified for any type of document, and for documents to then be created which adhere to the abstract specification. Document structure is defined by a formal construct known as a "document type definition." A computer program can be designed which reads this type definition and constructs a parser. The parser embodies the logical structure of a generalized document of the type specified

by the type definition. Actual documents can now be "marked up" using a notation which clearly identifies the location of each element in the structure. The resulting marked-up document can be processed in a variety of ways.

As an example, let us consider a single paragraph occurring somewhere within a particular document. The relationship of the paragraph to the rest of the document is determinable by referring to its position in the structure. If the processor has this information, it can then determine (perhaps by looking it up in a table) how the paragraph is to be formatted for output. Since the formatting instructions for each element in a particular type of document are variable, the identical marked-up document can be output in a number of different forms.

The use of some type of generalized markup is essential to the creation of an expert system capable of doing typographic design work. A person responsible for transforming a manuscript into type must first know its underlying structure. Generalized markup provides a way of describing this structure to a machine. Ideally, an expert system should be capable of taking a document marked up in this manner, and output a complete typographic description in "camera ready" form. In the following section we will discuss the specifics of Standard Generalized Markup Language.

4. STANDARD GENERALIZED MARKUP LANGUAGE

Standard Generalized Markup Language or SGML is a formal language for describing generalized markup languages. It is not itself a markup language, but rather a markup metalanguage. Markup languages which can be described by SGML are far more powerful than the simple generalized markup languages described in the previous section. SGML is defined in the International Standard ISO-DIS 8879 [SGML].

Figure 4.1a shows the extent to which a generalized markup language based solely on tags at the beginning of document elements can describe the relationships among them. There is no structural hierarchy. Elements can occur in any order. The structure of a document is the result of the arbitrary concatenation of elements by the user of the system. By including tags at the end of each element, nesting of elements within other elements is allowed. This relationship among elements is illustrated in figure 4.1b. Document structure is again arbitrarily decided by the user of the system. Since in both cases tags are processed without reference to other tags, the

system does not even have the power to determine whether the user has properly indicated nested structures with markup. It is perfectly legal, for example, to insert an end tag for a higher level structure before properly ending a nested substructure. The inclusion of a simple push-down stack would provide for a check against such an occurrence. Start tags could then be pushed onto the stack, and end tags made to cause the top of the stack to be popped. If an occurrence of an end tag caused a non-matching start tag to be popped, an error condition could be generated. This would provide a necessary safeguard against illegal nesting of subelements. It would still not prevent a user from creating a legal, but highly unworkable structure such as a paragraph nested within a title.

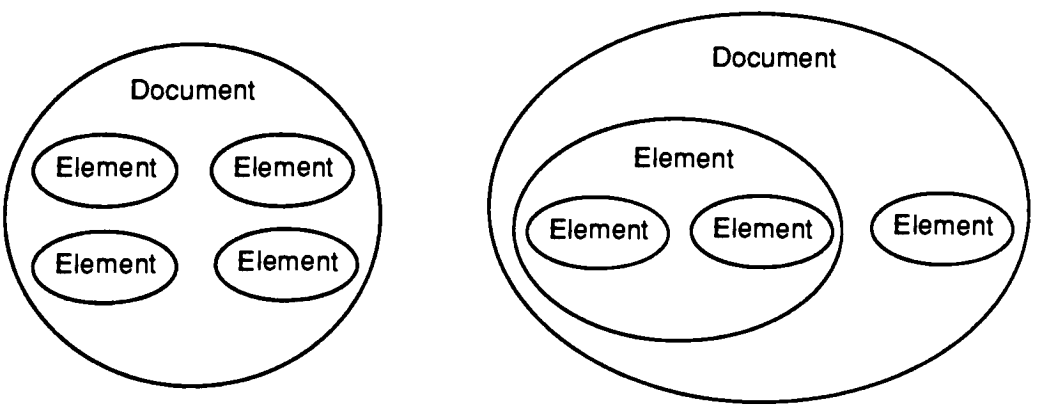


Figure 4.1 Graphic representation of document structures.

4.1 MODELLING DOCUMENT STRUCTURE IN SGML

Standard Generalized Markup Language provides a mechanism for building an unambiguous hierarchical representation of document structure to which marked up documents must conform. This representation is called a "document type definition."

> "In generalized markup, the term "document" does not refer to a physical construct such as a file or set of printed pages. Instead, a document is a logical construct called a "document element," the top node of a tree of elements that make up the document's content. A book, for example, could contain chapter elements that in turn contain "paragraph" elements and "picture" elements....Eventually, the terminal nodes of this document tree are reached and the actual characters or other data are encountered. If paragraphs, for example, were terminal, their content would be characters, rather than other elements. If photographs were terminal they would contain neither elements nor characters, but some noncharacter data that represents an image....The elements are distinguished from one another by additional information, called "markup," that is added to the data content. A document thus consists of two kinds of information: data and markup." [SGML, p. 68]

The structure of each element is defined in the document type definition. Two types of declarations can occur within a type definition. These are 1. entity declarations, and 2. element

declarations. Entity declarations are macro definitions which can be referred to within a document conforming to the type definition. They take the form

```
<!ENTITY entity_name "entity expansion">
```

where "<!" indicates the beginning of a markup declaration, "ENTITY" indicates that what follows is an entity declaration, "entity_name" is the macro instruction, and "entity expansion" is the replacement string. If the string "&entity_name;" is encountered within a conforming document, it is replaced with the entity expansion. Entity declarations are not relevent to this thesis, and will not be discussed further.

The element declarations in a type definition form a grammar consisting of productions with a single element name or "generic identifier" on the left hand side, and a content model for that element on the right hand side. The content model defines the structure of the element. The form of an element declaration is as follows:

```
<!ELEMENT element_name (content model)>
```

where "<!" indicates the beginning of a markup declaration, "ELEMENT" indicates that what follows is an element declaration, "element_name" is the left-hand side of the production and "content model" is the right-hand side. An example of an element

declaration is as follows:

<!ELEMENT essay (title, author, body)>.

This indicates that the document element called "essay" consists of three subelements--"title," "author" and "body"--concatenated together. Concatenation is indicated by the comma or "sequence" connector.* In this case, title, author and body must all occur in that sequence to form an essay element. In addition to concatenation, the binary operators for union and alternation are also available. If, for example, an essay consisted of a title and author in arbitrary order followed by a body, the proper element declaration would be

<!ELEMENT essay ((title & author), body)>.

The "&" (and) connector indicates union, and is used when both operands are required, but sequence is not important. The "|" (or) connector indicates alternation, and is used when one or the other but not both operands is required. Union and alternation operators can be used to connect more than two operands in a group. Thus

<!ELEMENT essay (title & author & body)>

---

*Connectors can also be referred to as binary operators.

indicates that an essay consists of a title, and an author and a body in any order, and

```
<!ELEMENT essay (title ¦ author ¦ body)>
```

indicates that an essay consists of a title or an author or a body. Binary operators or connectors are summarized in the table in figure 4.2

OPERATOR	NAME	FUNCTION
,	SEQ	connected elements must occur in the indicated order
¦	OR	only one of the connected elements must occur
&	AND	all of the connected elements must occur, but in any order

Figure 4.2 Summary of binary operators and their functions.

A class of unary operators called "occurrence indicators" is also included in SGML. These operators are attached as suffixes to elements to modify their meaning. The absense of any occurrence indicator is itself an indicator that the element must occur (is required) but may not occur more than once (is not

repeatable). The "?" operator indicates that the element is optional but may only occur once if it does. The "*" operator indicates that the element is optional but repeatable. The "+" operator indicates that the element is required but repeatable. These unary operators are summarized in figure 4.3.

OPERATOR	NAME	FUNCTION
?	opt	indicates element is optional but not repeatable
+	plus	indicates element is required and repeatable
*	rep	indicates element is optional and repeatable
null	-	the absence of an operator indicates element is required and not repeatable

Figure 4.3 Summary of unary operators and their functions.

The effects of occurrence indicators can also be summarized in a slightly different way. This is shown in figure 4.4. The set of element declarations in a document type definition forms a grammar which represents the structure of a document of that type. An example of such a grammar is shown in figure 4.5.

REQUIRED	REPEATABLE	OPERATOR
false	false	?
false	true	*
true	false	no suffix
true	true	+

Figure 4.4 Truth table representing the effect of unary operators.

```
<!ELEMENT story (title, author, body)>
<!ELEMENT title (#CDATA)>
<!ELEMENT author (#CDATA)>
<!ELEMENT body (leadparagraph, ((paragraph ¦ figure)*)>
<!ELEMENT leadparagraph (#CDATA)>
<!ELEMENT paragraph (#CDATA ¦ footnote)*>
<!ELEMENT figure NDATA>
<!ELEMENT footnote (#CDATA)>
```

Figure 4.5 Grammar representing the structure of a document in SGML.

This grammar represents the structure of a simple document type in SGML. At the top level of the document structure is the element "story" which consists of a "title" element followed by an "author" element followed by a "body" element. Both the title and the author elements consist of collections of zero or more characters. All the actual character data in a document is represented by using the reserved name "#CDATA," which means literally "zero or more data characters." The "*" or "rep" unary

operator is built in to the definition of #CDATA. So a title element and an author element both consist of zero or more characters.

The content model for a body element is defined as a "leadparagraph" followed by zero or more occurrences of the content model "(paragraph ¦ figure)." The "¦" operator indicates that either a paragraph or a figure can occur, and the "*" operator indicates that zero or more (paragraph ¦ figure)'s can occur. A paragraph is in turn defined as zero or more occurrences of (#CDATA ¦ footnote). This construction allows a footnote to occur at any point in a paragraph. The footnote itself is defined simply as zero or more characters. The keyword "NDATA" indicates data that is not defined by SGML. In this case, a figure might be a binary file representing a bitmapped graphic element.

4.2 FORMAT OF AN SGML DOCUMENT FILE

Each element name in an SGML document type definition is associated with two tags that may occur in a document conforming to that particular type definition. These are used to mark the beginning and the end of the element. In the default* SGML syntax, the tags used to mark the beginning and end of an element such as a "title" in the above example are "<title>" and "</title>" respectively. The "<" (less-than) symbol is called the "start-tag open" and the "</" pair of symbols is called the "end-tag open." The ">" (greater-than) symbol is called the "tag close." A string of characters beginning with a start-tag open and ending with a tag close is called a "start tag," and a string beginning with an end-tag open and ending with a tag close is called an "end tag."

The occurrence of a "title" element in a document conforming to the above grammar might look like this:

```
<title>Roger Takes a Bath</title>.
```

The characters "Roger Takes a Bath" are modelled in the type definition by "#CDATA." This is the only allowable content of a title element. In the type definition the element "title" is part

---

*The characters used as delimiters in the following discussion are part of the SGML "reference concrete syntax." The standard also allows the definition and use of an "alternative concrete syntax." [SGML, p. 71]

of the element "story." Therefore it must occur in proper sequence with other elements in the document to conform to the structure of the type definition. By reference to the type definition it is clear that the only element that can legally follow a "title" element is an "author" element. This is because in the rule

```
<!ELEMENT story (title, author, body)>
```

"author" follows "title" in sequence and is required. If the rule defining the structure of "story" were

```
<!ELEMENT story (title, author?, body)>,
```

then either the "author" start tag or the "body" start tag would be legal next tags in the document.

4.3 CONVERSION OF SGML TYPE DEFINITIONS TO LITERAL GRAMMARS

The grammar shown in figure 4.5 is not a literal grammar for strings in the language defined by it. Rather it is an abstract grammar. Fortunately it can easily be rewritten in a form that literally defines the language. If we take as an example the rule

<!ELEMENT story (title, author, body)>,

the string in the language that would satisfy this rule will take the following form:

<story> title author body </story>

where "title," "author" and "body" are non-terminals, and the tags "<story>" and "</story>" are terminals.

Figure 4.6a is an SGML type definition incorporating all the binary and unary operators available in the language. Figure 4.6b is a literal grammar for the language defined by the type definition. Terminal elements in figure 4.6b are in single quotes. The notations "#CDATA" and "NDATA" have been carried over from the type definition and have the same meaning. #CDATA represents any string of zero or more characters, and NDATA represents non-character data. The first production in 4.6b is a rule defining the structure of an element called "story." Since the SGML content model for "story" includes the "&" operator

```
<!ELEMENT story ((title & author+), body)>
<!ELEMENT title (#CDATA)>
<!ELEMENT author (#CDATA)>
<!ELEMENT body (leadpar, ((par | figure)*)>
<!ELEMENT leadpar (#CDATA)>
<!ELEMENT par (#CDATA)>
<!ELEMENT figure (picture, caption?)>
<!ELEMENT picture NDATA>
<!ELEMENT caption (#CDATA)>
```

4.6a

```
 1. story   --> '<story>' stuff1 body '</story>'
 2. stuff1  --> title author
 3. stuff1  --> author title
 4. title   --> '<title>' #CDATA  '</title>'
 5. author  --> '<author>' #CDATA '</author>'
 6. author  --> author author
 7. body    --> '<body>' leadpar stuff2 '</body>'
 8. leadpar --> '<leadpar>' #CDATA '</leadpar>'
 9. stuff2  --> par
10. stuff2  --> figure
11. stuff2  --> stuff2 stuff2
12. stuff2  --> EMPTY
13. par     --> '<par>' #CDATA '</par>'
14. figure  --> '<figure>' picture caption '</figure>'
15. picture --> '<picture>' NDATA '</picture>'
16. caption --> '<caption>' #CDATA '</caption>'
17. caption --> EMPTY
```

4.6b

Figure 4.6 Two versions of the same grammar. Figure 4.6a is in the form of an SGML type definition and figure 4.6b is in a form which is a literal grammar describing the language defined by the type definition.

between the elements "title" and "author+," both must occur, but order is arbitrary. Thus the first production in the grammar of 4.6b inserts the new name "stuff1" to represent "(title &

author+)." Productions 2 and 3 are rules defining "stuff1" as either a title followed by an author or an author followed by a title.

Since "author" is modified by the "+" operator in the type definition, it is required and repeatable. This is expressed in rules 5 and 6. A string such as "<author> John Doe </author> <author> John Smith </author> <author> John Jones </author>" can be derived using the grammar in 4.6b by applying rule 6 twice and rule 5 three times:

```
author --> author author (by rule 6)
       --> author author author (by rule 6)
       --> <author> ... </author> author author (by
            rule 5)
       -->  etc.
```

The content model for the element "body" in the type definition of 4.6a is "(leadpar, ((par | figure)*)." This construction allows any number (zero or more) of paragraphs and figures to appear in any order in the body of the document. This is expressed by rules 9 through 12 in the grammar of 4.6b. The content model "(par | figure)*" is represented by the name "stuff2" in rule 7. The "*" operator demands that one rule define "stuff" to be empty, hence rule 12. Rule 9 defines stuff as a paragraph, rule 10 defines stuff as a figure, and rule 11 allows for repetition. Thus a string consisting of n occurrences of

figures and paragraphs in any order can be derived by applying rule 11 n-1 times, then applying rule 9 to derive each occurrence of a paragraph and rule 10 to derive each occurrence of a figure.

Documents conforming to a particular SGML type definition are strings in the language defined by the grammar derived from the type definition. If a string is not in the language, it is not a conforming document. In the following section the derivation of a literal grammar from a type definition will be formalized.

4.4 FORMALIZATION OF THE DERIVATION OF GRAMMARS FROM SGML TYPE DEFINITIONS

In the following discussion SGML element names (called "generic identifiers" or "GI's" in the SGML standard) will be represented by single lower case letters. In the examples of derived grammars, terminals will be contained within single quotes, and non-terminals will be represented by single lower case letters. The SGML reserved word "#CDATA" will be used in both cases to mean zero or more characters, not including those used as markup delimiters.

The rules for rewriting SGML element declarations as literal grammar rules are as follow:

1. If the content model for an element "e" is simply zero or more characters designated by "#CDATA" in the element declaration, the right-hand side of the grammar rule is created by adding the prefix "'<e>'" and the suffix "'</e>'" to the string "#CDATA." Thus the SGML element declaration "<!ELEMENT e (#CDATA)>" will be rewritten as the grammar rule "e --> '<e>'#CDATA '</e>'."

2. If the content model for an element "e" is a single non-terminal "a," add the prefix "'<e>'" and the suffix "'</e>'" to the string "a" to form the rule "e --> '<e>'a'</e>'."

3. If the content model for an element "e" consists of two or more non-terminals concatenated together such as in the element declaration "<!ELEMENT e (a, b, c)>," add

the prefix "'<e>'" and the suffix "'</e>'" to the string "abc" to form the rule "e --> '<e>'abc'</e>'."

4. If two elements "a" and "b" in a content model are joined by the "&" operator such as in the declaration "<!ELEMENT e (a & b)>," use a new name "x" and write a rule defining "e" as follows:

   production 1: e --> '<e>'x'</e>'

   the write two rules defining "x" as follow:

   production 2: x --> ab

   production 3: x --> ba

5. If two elements "a" and "b" in a content model are joined by the "|" operator such as in the declaration "<!ELEMENT e (a | b)>," use a new name "x" and write a rule defining "e" as follows:

   production 1: e --> '<e>'x'</e>'

   then write two rules defining "x" as follow:

   production 2: x --> a

   production 3: x --> b

6. If an element "a" in a content model is modified by the unary operator "?" such as in the declaration "<!ELEMENT e (a?)>," write two productions for the rules defining "a" as follow:

   production 1: a --> '<a>'...''

   production 2: a --> EMPTY

7. If an element "a" in a content model is modified by the unary operator "+" such as in the declaration "<!ELEMENT e (a+)>," write two productions for the rules defining "a" as follow:

   production 1: a --> '<a>'...''

   production 2: a --> aa

8. If an element "a" in a content model is modified by the unary operator "*" such as in the declaration "<!ELEMENT e (a*)>," write three productions for the rules defining "a" as follow:

```
production 1: a --> '<a>'...'</a>'
production 2: a --> aa
production 3: a --> EMPTY
```

By using combinations of these eight rules, SGML type definition element declarations can be rewritten as productions in a literal grammar for documents of the same type. A few examples are shown below:

Example 4.1.

SGML declaration:

```
<!ELEMENT a ((b,c) & d)>
```

Derived grammar:

```
a --> '<a>'x'</a>'        using rule 4
x --> bcd                 using rule 4
x --> dbc                 using rule 4
```

Example 4.2.

SGML declaration:

```
<!ELEMENT a ((b & c) | d)>
```

Derived grammar:

```
a --> '<a>'x'</a>'        using rule 5
x --> y                   using rules 4 and 5
x --> d                   using rule 5
y --> bc                  using rule 4
y --> cb                  using rule 4
```

Example 4.3.

SGML declaration:

```
<!ELEMENT a ((((b & c) | d) & e)*)>
```

Derived grammar:

```
a --> '<a>'x'</a>'        using rule 4
x --> ye                  using rule 4
x --> ey                  using rule 4
x --> xx                  using rule 8
x --> EMPTY               using rule 8
y --> z                   using rules 4 and 5
y --> d                   using rule 5
z --> bc                  using rule 4
z --> cb                  using rule 4
```

```
<!ELEMENT a (b,c,d)>
<!ELEMENT b (#CDATA)>
<!ELEMENT c ((e | f)*)>
<!ELEMENT e (#CDATA)>
<!ELEMENT f (#CDATA)>
<!ELEMENT d (g? & h*)>
<!ELEMENT g (#CDATA)>
<!ELEMENT h (#CDATA)>
```

4.7a

```
 1. a --> '<a>'bcd'</a>'
 2. b --> '<b>'#CDATA'</b>'
 3. c --> '<c>'x'</c>'
 4. x --> e
 5. x --> f
 6. x --> xx
 7. x --> EMPTY
 8. e --> '<e>'#CDATA'</e>'
 9. f --> '<f>'#CDATA'</f>'
10. d --> '<d>'y'</d>'
11. y --> gh
12. y --> hg
13. g --> '<g>'#CDATA'</g>'
14. g --> EMPTY
15. h --> '<h>'#CDATA'</h>'
16. h --> hh
17. h --> EMPTY
```

4.7b

Figure 4.7 A type definition and its corresponding grammar.

The grammars derived from SGML type definitions in the previous section are context free grammars.* This is because only

*A complete discussion of the theory of context-free languages can be found in chapter 3 of [LEWIS, PAPA].

single non-terminals are allowed on the left-hand side of each production, and therefore the context of the non-terminal (the surrounding strings in the input stream) do not affect the rules of replacement of the non-terminal. In order for a document to conform to a type definition, it must be a string in the language defined by the derived grammar. In other words, by applying some finite combination of rules of the grammar, it must be possible to derive the document string.

The process of deriving a document as a string in the language defined by a context free grammar will be illustrated by reference to the SGML type definition shown in figure 4.7a. Figure 4.7b is the context free grammar for the same language. Let us take as an example the following "document" which conforms to the type definition in 4.7a and derive it using the grammar of figure 4.7b:

```
<a><b>Here are some characters</b><c><e>and some more
characters.</e><f>the "*" operator allows for
repetition</f><e>and this could go on for a long
time.</e></c><d><h>notice that there is no 'g' tag in
this document.</h></d></a>
```

One possible derivation for this document is as follows:

```
a -->     '<a>'bcd'</a>'

  -->     '<a><b>Here are some characters</b>'cd'</a>'
```

```
-->     '<a><b>Here are some characters</b><c>'x
        '</c>'d'</a>'

-->     '<a><b>Here are some characters</b><c>'xx
        '</c>'d'</a>'

-->     '<a><b>Here are some characters</b><c>'xxx
        '</c>'d'</a>'

-->     '<a><b>Here are some characters</b><c>'efe
        '</c>'d'</a>'

-->     '<a><b>Here are some characters</b><c><e>
        and some more characters.</e>'fe
        '</c>'d'</a>'

-->     '<a><b>Here are some characters</b><c><e>
        and some more characters.</e><f>the "*"
        operator allows for repetition</f>'e
        '</c>'d'</a>'

-->     '<a><b>Here are some characters</b><c><e>
        and some more characters.</e><f>the "*"
        operator allows for repetition</f><e>and
        this could go on for a long time</e></c>'
        d'</a>'

-->     '<a><b>Here are some characters</b><c><e>
        and some more characters.</e><f>the "*"
        operator allows for repetition</f><e>and
        this could go on for a long time</e></c>
        <d>'y'</d></a>'

-->     '<a><b>Here are some characters</b><c><e>
        and some more characters.</e><f>the "*"
```

operator allows for repetition</f><e>and
this could go on for a long time</e></c>
<d>'gh'</d>'

--> '<a>Here are some characters<c><e>
and some more characters.</e><f>the "*"
operator allows for repetition</f><e>and
this could go on for a long time</e></c>
<d>'g'<h>notice that there is no 'g' tag
in this document</h></d>'

--> '<a>Here are some characters<c><e>
and some more characters.</e><f>the "*"
operator allows for repetition</f><e>and
this could go on for a long time</e></c>
<d><h>notice that there is no 'g' tag
in this document</h></d>'

There are other derivations of the same document string. For example, the element declaration "<!ELEMENT d (g? & h*)>" allows two alternative content models for "d." These are "g?h*" or "h*g?" and since there are no occurrences of a "g" element in the document string, it does not matter whether production 11 or production 12 are used to define "y" in the grammar of 4.6b. Either way the document string can be derived. The possibility of the existence of multiple derivations of the same string is characteristic of context free grammars [LEWIS, p. 102].

5. CONSTRUCTION OF A DOCUMENT PARSER

In the previous chapter, a method was described for transforming an SGML type definition, which is an abstract definition of a markup language, into a literal context-free grammar for the language. It was also shown that in order for a document to conform to a particular type definition it must be derivable from the related context-free grammar. The most obvious task to be performed by an SGML document parser, then, is to verify the syntactical correctness of the document. A parser must also be capable of performing certain actions when structures are encountered in the document string. In the following discussion we will refer to the type definition and grammar of figure 4.6. A document file which conforms to this grammar is shown is shown in figure 5.1.

We will now turn our efforts to the creation of a parser to perform the functions of 1. verifying the conformance of a document to a particular type definition* and 2. translating the

---

*in other words, of verifying that the document string is in the language defined by the grammar derived from the type definition.

descriptive markup to device-specific markup for a particular typesetting system. These functions can certainly be performed separately, and it makes good sense to verify conformance of a document before an input file for a typesetting system is constructed. However, it is also possible to perform both functions concurrently. It will now be demonstrated that a parser can be constructed using the Unix tools Lex* and Yacc** for any context-free grammar derived from an SGML type definition.

<story><title>Why Johnny Can't Play the Harpsichord **</title><author>**Frank Cost**</author><author>**John Smith**</author>**
<body><leadpar>The problem of musical illiteracy among young people in this country is currently...**</leadpar><par>**The fear of ivory or "tuskophobia" has been investigated as a possible cause...**</par><par>**...**</par>**
</body></story>

Figure 5.1 A document conforming to the type definition and grammar of figure 4.6. Markup is printed in bold.

---

*Lex is a lexical analyzer generator running under the Unix operating system. A complete description of it can be found in [LESK].

**Yacc is a parser generator running under Unix. A description of Yacc can be found in [JOHNSON].

5.1 THE LEX LEXICAL ANALYZER GENERATOR

Lex is a program running under the Unix operating system which can be used to generate programs capable of lexical processing of character strings.

> "It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed."[LESK, p.388]

A regular expression is one that is built from finite formal languages using the operations of union, concatenation and Kleene star. The following, taken from [LEWIS, p. 35], is a formal definition:

> "The regular expressions over an alphabet Σ are the strings over the alphabet Σ union {), (, φ, U, *} such that the following hold:

1. φ and each member of Σ is a regular expression.

2. if α and β are regular expressions then so is $(\alpha\beta)$.

3. if α and β are regular expressions then so is $(\alpha \cup \beta)$.

4. if α is a regular expression, the so is α^*.

5. Nothing is a regular expression unless it follows from (1) to (4).

Every regular expression represents a language, according to the interpretation of the symbols U and * as set union and Kleene star."

So if $\Sigma = \{\alpha, \beta\}$, then the following are all regular expressions over Σ: $\alpha\beta\beta\beta\alpha$, $\alpha^*\beta^*$, $\alpha \cup \beta\alpha\beta^*$, $(\alpha \cup \beta)\alpha\beta^*$, etc. A program generated by Lex is capable of finding the longest string in an input stream matching a given regular expression in the Lex specification file. Any part of the input not matching a regular expression in the specification file is passed to output by the Lex-generated program.

Which parts of an SGML document file do we want a Lex-generated program to recognize? There are, remember, two types of things to be found in such a file--markup and content. The markup is responsible for connoting the structure of the

document. The content is not important. In fact it is possible to imagine a document consisting of markup only. This notion of an "empty" document is important, because whether or not there is any content should not bear on the syntactic "correctness" of a document. This is because content (character content) is always defined as "#CDATA," which simply means "zero or more characters." The document of figure 5.1 could be wrung of all its content and still conform to the type definition of 4.6. Figure 5.2 shows a document with the same structure as that of 5.1, but without any content.

```
<story><title></title><author></author><author>
</author><body><leadpar></leadpar><par></par>
<par></par><par></par></body></story>
```

Figure 5.2 A content-less document conforming to the grammar of 4.6.

It is only necessary that a lexical analyzer find markup tags in an input document. There are two types of tags--start tags and end tags. Start tags are delimited by the prefix "<" and the suffix ">" and end tags are delimited by the prefix "</" and the suffix ">." It is a simple matter to write regular expressions to match any start tag or end tag. In the Lex specification language, classes of characters can be denoted using the operator pair "[]." The character class "[αβ]" matches a single character,

which may be α or β. The "-" character is used to indicate ranges.* Thus the class "[a-z]" matches a single lower case character, and "[0-9]" matches a single decimal digit. If tag names were restricted to the lower case alphabet and digits, the following two regular expressions would match arbitrary start tags and end tags respectively:

```
<[a-z0-9]*>     matches any start tag
</[a-z0-9]*>    matches any end tag
```

A lexical analyzer based solely upon these two regular expressions would be able to recognize markup in a document conforming to any SGML type definition, but would not be able to distinguish among various start tags and end tags.

It it also possible to build a lexical analyzer more specific to a given document type. Regular expressions can be written to match specific tags in the markup. Figure 5.3 shows a list of regular expressions which will be matched by tags contained within documents conforming to the type definition of 4.6. When a Lex-generated program finds a string that matches a regular expression in its specification, it can be made to return a value to the calling program. If the calling program is designed to interpret this value as an indication that a tag has been located, and it is capable of analyzing the structure of the

---

*based on a particular collating sequence such as ASCII or EBCDIC.

stream of returned values from the Lex-generated program, the conformance of the document to a type definition can be determined.

1. <story>	7. <body>	13. <figure>
2. </story>	8. </body>	14. </figure>
3. <title>	9. <leadpar>	15. <picture>
4. </title>	10. </leadpar>	16. </picture>
5. <author>	11. <par>	17. <caption>
6. </author>	12. </par>	18. </caption>

Figure 5.3 A list of regular expressions matching tags associated with documents conforming to the type definition in 4.6.

5.1.1 THE STRUCTURE OF A LEX SOURCE FILE

The creation of a Lex source file for our purposes is a simple matter. The general format of a Lex source file is:

```
{definitions}
%%
{rules}
%%
{user subprograms}
```

The definitions and user subprograms are not required. "The rules represent the user's control decisions; they are a table, in

which the left column contains regular expressions...and the right column contains actions, program fagments to be executed when the expressions are recognized. Thus an individual rule might appear

```
integer  printf("found keyword INT");
```

to look for the string "integer" in the input stream and print the message "found keyword INT" whenever it appears. In this example the host procedural language is C and the C library function "printf" is used to print the string." [LESK, p. 390]

The actions which can be performed include returning simple integer values to the calling program. These values can be defined in the definitions section of the Lex source specification. For example, the definitions section of a Lex source file used to generate a lexical analyzer for documents conforming to the grammar of 4.6 is shown in figure 5.4. The rules section of the Lex source consists of regular expressions with corresponding actions. The actions in this case are simply to return the correct integer values to the calling program when rules are matched in the input stream.

The lexical analyzer of 5.5 produces an output stream consisting of data unmatched by any rule interspersed with tokens representing markup tags. The calling program must be able to determine whether the stream of tokens from first to last corresponds to a derived string in the grammar. It must also be

capable of recognizing substructures in the stream of tokens so that appropriate actions can be taken to process their content.

```
# define STORYTOKEN 257
# define ENDSTORYTOKEN 258
# define TITLETOKEN 259
# define ENDTITLETOKEN 260
# define AUTHORTOKEN 261
# define ENDAUTHORTOKEN 262
# define BODYTOKEN 263
# define ENDBODYTOKEN 264
# define LEADPARTOKEN 265
# define ENDLEADPARTOKEN 266
# define PARTOKEN 267
# define ENDPARTOKEN 268
# define FIGURETOKEN 269
# define ENDFIGURETOKEN 270
# define PICTURETOKEN 271
# define ENDPICTURETOKEN 272
# define CAPTIONTOKEN 273
# define ENDCAPTIONTOKEN 274
```

Figure 5.4 The definition section of a Lex source for a lexical analyzer for documents conforming to the grammar of 4.6.

```
"<story>"       return (STORYTOKEN);
"</story>"      return (ENDSTORYTOKEN);
"<title>"       return (TITLETOKEN);
"</title>"      return (ENDTITLETOKEN);
"<author>"      return (AUTHORTOKEN);
"</author>"     return (ENDAUTHORTOKEN);
"<body>"        return (BODYTOKEN);
"</body>"       return (ENDBODYTOKEN);
"<leadpar>"     return (LEADPARTOKEN);
"</leadpar>"    return (ENDLEADPARTOKEN);
"<par>"         return (PARTOKEN);
"</par>"        return (ENDPARTOKEN);
"<figure>"      return (FIGURETOKEN);
"</figure>"     return (ENDFIGURETOKEN);
"<picture>"     return (PICTURETOKEN);
"</picture>"    return (ENDPICTURETOKEN);
"<caption>"     return (CAPTIONTOKEN);
"</caption>"    return (ENDCAPTIONTOKEN);
```

Figure 5.5 The rules section of a Lex source for a lexical analyzer for documents conforming to the grammar of 4.6.

5.2 THE YACC PARSER GENERATOR

The following description of the Yacc parser generator is taken from the introduction of [JOHNSON, p. 354]:

> "Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when the rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a 'parser,' calls the user-supplied low-level input routine (the 'lexical analyzer') to pick up the basic input items (called 'tokens') from the input stream. These tokens are organized according to the input structure rules, called 'grammar rules;' when one of these rules has been recognized, the user code supplied for this rule, an 'action,' is invoked; actions have the ability to return values and make use of the values of other actions...An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a 'terminal symbol,' while the structure recognized by the parser is called a 'nonterminal symbol.' To avoid confusion, terminal symbols will usually be referred to as 'tokens.'"

A Yacc specification file consists of three sections: declarations, grammar rules and programs. The sections are separated from one another by the double percent "%%" marks as in a Lex specification file. The full Yacc specification file has the following structure:

```
declarations
%%
grammar rules
%%
programs
```

Both the declarations and programs sections may be omitted, so the minimal Yacc specification file structure is

```
%%
grammar rules
```

The rules section consists of one or more grammar rules having the following form:

```
NAME : DEFINITION;
```

where NAME is a nonterminal, and DEFINITION represents a sequence of zero or more nonterminals and terminals. If a nonterminal has

more than one definition, it is possible to indicate this by including more than one rule as in

```
NAME : DEFINITION 1;
NAME : DEFINITION 2;
NAME : DEFINITION 3;
```

or by using the "¦" symbol to indicate alternation as in

```
NAME : DEFINITION 1
     ¦ DEFINITION 2
     ¦ DEFINITION 3;
```

If a nonterminal is matched by the empty string, this can be indicated by the rule

```
NAME : ;
```

5.2.1 THE YACC DECLARATIONS SECTION

Names representing tokens must be declared as such in the declarations section of the Yacc specification file. This is done by simply listing the token names following the string "%token." The declarations section must also include the name of the start symbol for the grammar. This is preceded by the string "%start."

Figure 5.6 shows the declarations section for a Yacc specification file to be used to generate a parser for documents conforming to the grammar of 4.6.

```
%token     STORYTOKEN
           ENDSTORYTOKEN
           TITLETOKEN
           ENDTITLETOKEN
           AUTHORTOKEN
           ENDAUTHORTOKEN
           BODYTOKEN
           ENDBODYTOKEN
           LEADPARTOKEN
           ENDLEADPARTOKEN
           PARTOKEN
           ENDPARTOKEN
           FIGURETOKEN
           ENDFIGURETOKEN
           PICTURETOKEN
           ENDPICTURETOKEN
           CAPTIONTOKEN
           ENDCAPTIONTOKEN

% start    story
```

Figure 5.6 The declarations section of a Yacc specification file used to generate a parser for documents conforming to the grammar of 4.6.

The Yacc program creates a table of definitions for each of the tokens declared in the declarations section of the specification file. This table can then be included in the declarations section of a Lex specification file, or, if the Lex-generated program is included in the Yacc specification file, serve as shared definitions between them. When the Lex-generated program finds a

string matching one of its regular expressions, it passes the appropriate token to the Yacc-generated program.

5.2.2 THE YACC RULES SECTION

The grammar rules in the Yacc specification file defines the structure of the input. The starting nonterminal in the grammar of 4.6 is the name "story." If we picture the structure of a document conforming to this grammar as a tree, story is the root of the tree. A Yacc rule defining the structure of a story is

```
story : STORYTOKEN stuff1 body ENDSTORYTOKEN;
```

Terminals are written in upper case letters, and non-terminals are written in lower case. This rule must be matched by the entire input string in order for the string to conform to the grammar. The first token found in the input must be "STORYTOKEN" and the last token must be "ENDSTORYTOKEN." Since "stuff1" and "body" are nonterminals, they must in turn be defined. The two definitions for "stuff1" are

```
stuff1 : title author;
stuff1 : author title;
```

because in the type definition of 4.6 the element "story is defined as

```
story ((title & author+), body).
```

The "&" operator means both operands are required but in either order, and therefore the grammar must allow for either possibility. The complete rules section of a Yacc specification file reflecting the grammar of 4.6 is shown in figure 5.7

```
%%
story     : STORYTOKEN stuff1 body ENDSTORYTOKEN;
stuff1    : title author
          | author title
          ;
title     : TITLETOKEN ENDTITLETOKEN;
author    : AUTHORTOKEN ENDAUTHORTOKEN
          | author author
          ;
body      : BODYTOKEN leadpar stuff2 ENDBODYTOKEN;
leadpar   : LEADPARTOKEN ENDLEADPARTOKEN;
stuff2    : par
          | figure
          | stuff2 stuff2
          |
          ;
par       : PARTOKEN ENDPARTOKEN;
figure    : FIGURETOKEN picture caption ENDFIGURETOKEN;
picture   : PICTURETOKEN ENDPICTURETOKEN;
caption   : CAPTIONTOKEN ENDCAPTIONTOKEN;
          |
          ;
```

Figure 5.7 The rules section of a Yacc specification file representing the grammar of 4.6.

The program that Yacc writes using this grammar will call upon the program written by Lex to read through a document conforming to the grammar returning a stream of tokens, and will parse the document. Since no actions have been specified, the only output from the parser will be a stream of symbols representing the content of the document (the parts of the data stream which the lexical analyzer fails to match). If the parse is successful, the entire content of the document will be output. Output in this form is rather useless, however, since it will have been stripped of all its markup.

The Yacc-generated program can be modified to perform actions while it is parsing the document. When a rule in the grammar is matched by an input structure, a program segment associated with that rule can be invoked. For example, a simple message might be placed in the output stream indicating that a rule has been matched. Let us include an action associated with the top level rule in the Yacc specification of 5.7. This action is a simple print statement:

```
story     : STORYTOKEN stuff1 body ENDSTORYTOKEN
                {
                printf("Parse Successful!!\n");
                }
          ;
```

If the rule is matched, the message "Parse Successful!!" will be

inserted in the output stream. It is possible to include similar program segments for each rule in the grammar. But these are only invoked after the associated structure is recognized. We also need to make insertions into the output stream before some structures are recognized. Let us take as an example the structure

```
par       : PARTOKEN ENDPARTOKEN;
```

The Lexical analyzer will pass the token "PARTOKEN" to the parser when it encounters a string matching the regular expression "<par>," which is a tag marking the beginning of a paragraph. It will then read through the content of the paragraph until it encounters a string matching the regular expression "</par>" marking the end of the paragraph. At this point it will pass the token "ENDPARTOKEN" to the parser, which will recognize the structure of the nonterminal "par."

By altering the rule describing the structure of a "par" in the following way, the parser will be able to perform actions before the content of a paragraph is encountered:

```
par       : startpar ENDPARTOKEN;
startpar  : PARTOKEN;
```

A new nonterminal called "startpar" is added to the grammar. It is defined as the single terminal "PARTOKEN." When this rule is

matched, actions can be performed. By rewriting the grammar rules defining structures which have content in this way, the parser is able to make insertions preceding the content. Figure 5.8 is the rules section of 5.7 rewritten in this manner.

```
%%
story     : STORYTOKEN stuff1 body ENDSTORYTOKEN;
stuff1    : title author
          | author title
          ;
title     : stitle ENDTITLETOKEN;
stitle    : TITLETOKEN;
author    : sauthor ENDAUTHORTOKEN
          | author author
          ;
sauthor   : AUTHORTOKEN;
body      : BODYTOKEN leadpar stuff2 ENDBODYTOKEN;
leadpar   : sleadpar ENDLEADPARTOKEN;
sleadpar  : LEADPARTOKEN;
stuff2    : par
          | figure
          | stuff2 stuff2
          |
          ;
par       : spar ENDPARTOKEN;
spar      : PARTOKEN;
figure    : FIGURETOKEN picture caption ENDFIGURETOKEN;
picture   : spicture ENDPICTURETOKEN;
spicture  : PICTURETOKEN;
caption   : scaption ENDCAPTIONTOKEN;
          |
          ;
scaption  : CAPTIONTOKEN;
```

Figure 5.8 Modified rules section of a Yacc specification file to allow for pre-content actions.

5.3 USING YACC TO GENERATE A PARSER TO CREATE INPUT FILES FOR A TYPESETTING SYSTEM

Let us reexamine the nature of the input expected by an automated typesetting system such as the MCS system described in chapter 2. In order to typeset a string of characters, the system must be informed of parameter values for line measure, typeface, point size and leading. These parameter values must precede the text to be typeset. Once a value is set for any parameter, it remains thus until explicitly changed. Other commands must follow strings of characters to be typeset. These include such things as quadding commands and extra leading. So the general format of each structure in a typesetting input file is

PRE-CONTENT COMMANDS content POST-CONTENT COMMANDS.

If we take as an example a paragraph structure conforming to the grammar of 4.6, it exists in the SGML document file in the following form:

<par>content...</par>,

and needs to be transformed into something resembling this before it can serve as input to a typesetting system:

<LL35><FT3><PS12><LS13>content...%L.

Since a Yacc specification can be designed so that the resulting parser is capable of performing actions both before and after a particular structure is encountered in the document file, such a transformation of an element from generalized to specialized form is simple. Figure 5.8 shows Yacc rules which will result in the creation of that part of a parser capable of producing output in the necessary form:

```
par         : spar ENDPARTOKEN
                  {
                  printf("%L");
                  }
            ;
spar        : PARTOKEN;
                  {
                  printf("<LL35><FT3><PS12><LS13>");
                  }
            ;
```

Figure 5.9 Yacc rules with actions to insert typesetting commands into the data stream.

When the parser recognizes the start tag for the paragraph structure in the input, the parameter commands are inserted into the output stream. The contents of the paragraph are then passed directly from input to output. When the parser gets the token from the lexical analyzer representing the end tag for the paragraph structure, the structure is matched, and the parser inserts the appropriate quadding command. For every structure

described in the original type definition it is possible to specify an appropriate Yacc-generated mechanism for recognizing the beginning and the end of the content of that structure and performing actions accordingly.

Thus far we have only described actions which make insertions into the data stream. These are the simplest class of actions needed. There are cases where more complex actions must be taken. If we consider the following element declarations:

```
<!ELEMENT paragraph ((#CDATA ¦ footnote)*)>
<!ELEMENT footnote (#CDATA)>
```

the structure will allow a footnote to occur at any point in a paragraph. A Yacc specification corresponding to these declarations is shown in figure 5.9. The part of the Yacc-generated parser specified by the grammar of 5.9 will stream the contents of footnotes to output when they are found. But since

```
paragraph : startpar stuff1 ENDPARTOKEN;
startpar  : PARTOKEN;
stuff1    : startnote ENDFOOTNOTETOKEN
          ¦ stuff1 stuff1
          ¦
          ;
startnote : FOOTNOTETOKEN;
```

Figure 5.10 Yacc grammar for a paragraph structure including footnotes.

footnotes normally occur out of sequence in a text (at the bottom of the page where the reference occurs, for example), the input file for a typesetter cannot simply include the footnote where it occurs in the original document.

This problem indicates the need for additional processing of a file before it can be input to a typesetter. Footnote placement is an extreme high-level consideration in the formatting of a document. Footnotes must normally occur on the same page as the reference.* The amount of space left at the bottom of a page for notes must be calculated on the basis of how much vertical space or depth will be occupied by the footnotes. This means that the depth of footnotes must be calculated before they are actually typeset.** If all footnotes must occur in their entirety on the same page, it is relatively simple to design an algorithm to handle the pagination.

Such an algorithm would work as follows: As each line of text on a page is typeset, the depth of the page is updated until the full page depth is reached. When a footnote is encountered in the input stream, it is "soft-typeset" to determine its depth. This depth will include space separating the last line of text and the first line of the first footnote, as well as a rule or line

---

*The rules governing the placement of footnotes vary from place to place. In some cases, footnotes are all placed on the bottom of the right hand folio. In other cases, they occur at the bottom of each page, and sometimes they occur in the margins adjacent to the references.

**in effect, "soft-typesetting" the notes to determine their dimensions so that enough space is left for them.

separating the footnotes from the text above. As this is happening, the footnote is streamed into a temporary buffer. The calculated depth of the footnote is then subtracted from the total depth of the page to yield a new page depth, and if the remaining difference between the current depth of the text and the new depth of the page is positive, the system continues to typeset the text until the new depth is reached. The process is repeated for each additional footnote encountered. When the last line of text is set, the appropriate font changes are made, and the footnote is typeset. If the calculated depth of a footnote subtracted from the total remaining depth of the page is negative, the line containing the footnote reference is moved to the top of the next page. The footnotes previously encountered on the page are then typeset at the bottom of the page.*

The foregoing discussion is meant to raise the issue of content that cannot be typeset in the same sequence as it is encountered in the original marked up document. If we deal only with document structures which present their content in the order they will be typeset, the mechanisms described above allow sufficient flexibility to produce complete input files for a typesetting system without the need for further processing. In

---

*Such a scheme will work well if footnotes are few and far between. The worst case, however, is the occurrence of several footnote references in the first line of text on a page which, when collectively typeset, have a depth greater than the total page depth. The system will then generate an unending number of blank pages, because the offending line will be moved to the top of the next page where the system will again fail to typeset the line.

the next chapter, we will develop a document type definition, write a specification for a parser, and, by altering the actions associated with the Yacc rules comprising the parser specification, demonstrate the ability of the resulting filters to generate variable input files for a typesetting system.

6. CREATING A TEST PARSER

A document conforming to a relatively simple type definition was created and then filtered by Yacc/Lex-generated programs to produce input files for the MCS typesetting system. The type definition is for a document type called "essay." It is shown in figure 6.1. A graphic representation of the highest two levels of this structure is shown in figure 6.2a. An essay consists of three elements: frontmatter, body and rearmatter. Each of these in turn consist of subelements.

The structure of frontmatter is shown in figure 6.2b. Frontmatter is composed of three subelements. The title consists of a required first title and an optional second title (indicated in the type definition and in the figure by the operator "?"). There can be one or more authors (indicated by the "+" operator). The abstract is optional.

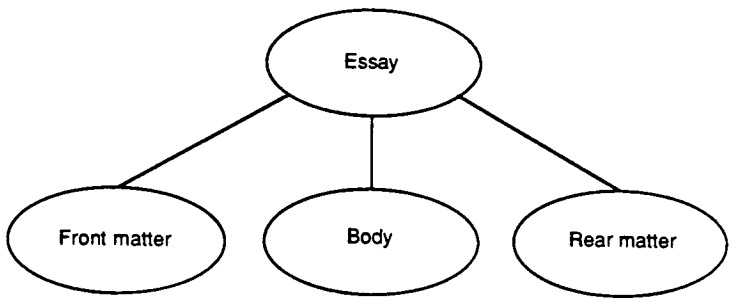
The structure of body is shown in figure 6.2c. A body consists of an introduction followed by one or more sections. An introduction consists of a lead paragraph followed by zero or more paragraphs. A section consists of a section title followed

```
<!ELEMENT essay          (frontmatter, body, rearmatter)>
<!ELEMENT frontmatter    (title, author+, abstract?)>
<!ELEMENT title          (firsttitle, secondtitle?)>
<!ELEMENT author         (#CDATA)>
<!ELEMENT abstract       (abpar+)>
<!ELEMENT abpar          (#CDATA)>
<!ELEMENT firsttitle     (#CDATA)>
<!ELEMENT secondtitle    (#CDATA)>
<!ELEMENT body           (intro, section+)>
<!ELEMENT intro          (introleadpar, intropar*)>
<!ELEMENT introleadpar   (#CDATA)>
<!ELEMENT intropar       (#CDATA)>
<!ELEMENT section        (sectiontitle, sectionbody)>
<!ELEMENT sectiontitle   (#CDATA)>
<!ELEMENT sectionbody    (sectionleadpar, sectionpar*)>
<!ELEMENT sectionleadpar (#CDATA)>
<!ELEMENT sectionpar     (#CDATA)>
<!ELEMENT rearmatter     (biblio?, acknowledge?)>
<!ELEMENT biblio         (ref+)>
<!ELEMENT ref            ((reftitle & refauthor+), refinfo)>
<!ELEMENT acknowledge    (#CDATA)>
<!ELEMENT reftitle       (#CDATA)>
<!ELEMENT refauthor      (#CDATA)>
<!ELEMENT refinfo        (#CDATA)>
```

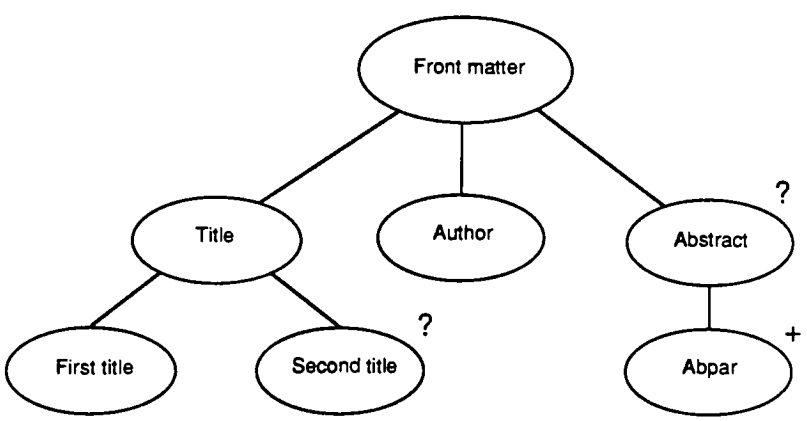
Figure 6.1 Type definition for an essay document type.

by a section body. A sectionbody consists of a lead paragraph (not the same as the leadparagraph in the introduction) followed by zero or more section paragraphs (also different).

The structure of rearmatter is shown in figure 6.2d. Rearmatter consists of an optional bibliography followed by an optional acknowledgements section. The bibliography is a collection of one or more references. Each reference has a title and an author (one or more) in either order, followed by some reference information.

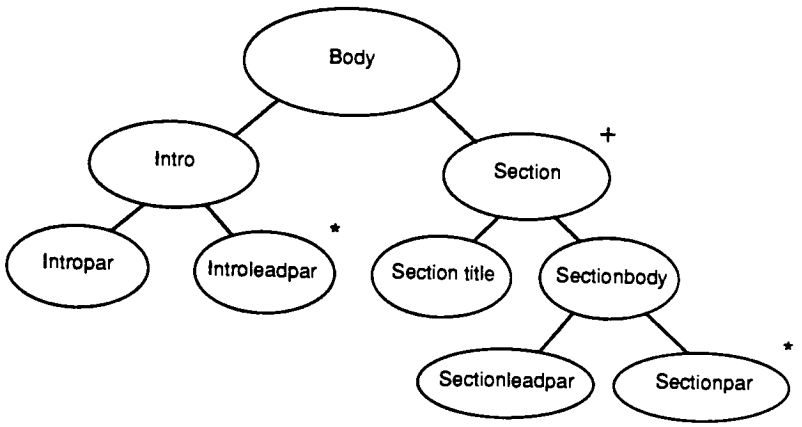


6.2a

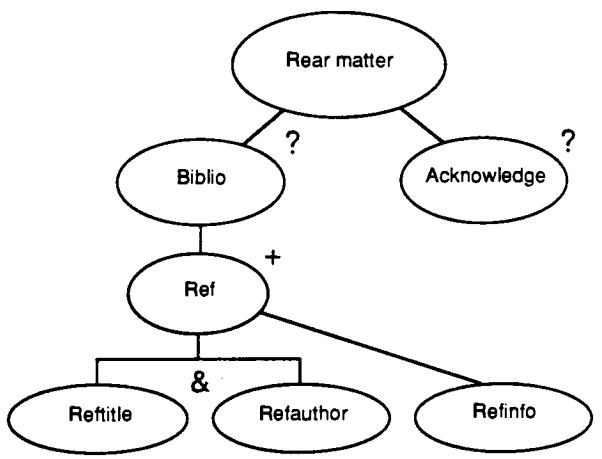


6.2b

Figure 6.2 Graphic representation of a document conforming to the type definition of 6.1.



6.2c



6.2d

Figure 6.2 (continued) Graphic representation of a document conforming to the type definition of 6.1.

A Lex source file specifying a lexical analyzer for documents conforming to the grammar of 6.1 can be found in appendix B and a Yacc source file specifying a parser for documents conforming to the same grammar can be found in Appendix C. This Yacc source file does not include any actions to be performed when structures are encountered in the input. The program generated by Yacc is called "yyparse," and may be called from within a small program which can be designed to write a message indicating the success or failure of a parse. The C source program generated by Yacc is called "y.tab.c." It can be included in the program calling yyparse. This program is shown in figure 6.3.

```
#include "y.tab.c"
main()
{
     if( yyparse() )
               printf( "Parse unsuccessful\n" );
     else
               printf( "Parse successful\n" );
}
```

Figure 6.3 Program calling yyparse() and indicating success or failure of a parse.

Since the program generated by Lex is included* in the program generated by Yacc, definitions for the tokens listed in the

*This include statement occurs in the program section of the Yacc source file.

decarations section of the Yacc file are shared. Since the program of 6.3 includes the Yacc-generated program which in turn includes the Lex-generated program, the entire parsing mechanism is compiled together.

The Lexical analyzer generated by Lex scans the document file looking for strings matching regular expressions. Input which does not match any expression is passed without alteration to output. When a match is found, an integer token is passed to the calling program, which in this case is the Yacc-generated parser. This parser consists of a simple finite state machine with a stack [JOHNSON, p. 360]. The parser has the ability to look ahead one token in the input. The current state is kept on the top of the stack. States are represented by integers. When processing begins, the machine is in state 0 (the stack contains only state 0), and no lookahead token has been read.

> "The machine has only four actions available to it, called shift, reduce, accept and error. A move of the parser is done as follows: 1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls yylex to obtain the next token. 2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone." [JOHNSON, p. 360]

The "shift" action is the way the parser changes states given a present state and the next token in the input. If the parser is in a state n, the action "IF shift m" will cause the machine to enter state m if the lookahead token is "IF." In other words, an integer representing state m is pushed onto the stack. The lookahead token is then cleared.

The "reduce" action reduces the contents of the stack when the parser has recognized the right hand side of a grammar rule, and is intended to prevent the stack from growing without bounds.

> "In effect, the reduce action 'turns back the clock' in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off the stack: the uncovered state is in fact the current state." [JOHNSON, p. 361]

The "accept" action is taken when the entire input has been seen, and the input has conformed to the grammar. The "error" action is taken if the parser reaches a point in the input where it can no longer continue parsing. This happens if the input tokens already seen as well as the lookahead token cannot be followed by any input that would result in a rule being matched. A human-readable representation of the parser can be generated when Yacc is run. This is obtained by invoking Yacc with the -v option. A file called "y.output" is produced automatically. The

file corresponding to the Yacc specification in appendix C is included in appendix D.

The strings in the input file unmatched by the lexical analyzer (i.e. the actual characters to be typeset corresponding to #CDATA in the original type definition) are passed directly as input to the parser, which in turn passes them directly to output without alteration. Matched strings in the input are not passed to output by the lexical analyzer. Instead, tokens are passed to the parser which assembles them and attempts to match rules in its own specification. When the parser recognizes a string of tokens matching a rule, it takes an action. All the actions in this case have the effect of inserting codes for typesetting. The resulting output file is a procedurally-marked-up version of the original generically-marked-up document.

A document conforming to the type definition of figure 6.1 can be found in appendix E. Two parsers for documents of this type were created using different Yacc source files. Both source files contain the same set of grammar rules, but the respective sets of actions associated with the rules (i.e. the insertion of typesetting codes) differ. Both Yacc source files are included in appendix F. The document of appendix E was streamed through the two parsers, and the resulting files were input to the MCS typesetting system via the translation program described in chapter 2. The two input files and their respective typeset results can be found in appendix G.

7. CONCLUSIONS AND RECOMMENDATIONS

The mechanism described in the previous chapter makes direct use of a Yacc-generated parser to create an input file for a typesetting system. The Lex-generated lexical analyzer looks for markup as it scans the document file, and only returns tokens to the parser when it has found a markup tag. The parser collects these tokens and attempts to match grammar rules in its specification. When certain rules are matched, the parser takes actions. These actions have the effect of inserting procedural typesetting commands into the data stream. The resulting file consists of the content (non-markup) of the original document file interspersed with typesetting commands, and can be input directly to a typesetting machine. In addition, the parser will fail to produce a complete output file if a document does not conform exactly to the type definition. Thus the parser simultaneously checks the structure of a document while preparing an input file for the typesetting system.

The two typeset examples in Appendix G are two incarnations of the document of Appendix E. They differ quite markedly in

typographic appearance. The parameters associated with each element in the original document were varied to produce these results. In the hands of a knowledgeable designer, this mechanism should be able to produce some fairly attractive documents.

The mechanism is quite limited, however. A few of these limitations are listed below:

1. Since the parser makes a single pass over the input file, and has no capability of buffering the contents of a document element, it can only deal with text that is to be typeset sequentially. There is no way for the parser to temporarily store the contents of a footnote, for example, for later output.

2. All of the typesetting parameters are buried within the Yacc specification file. To selectively alter these parameters the Yacc file must be altered.

3. The author is completely responsible for preparing a document that conforms to the type definition. In the example given in chapter 6, this is a relatively simple task because the type definition is simple. More complex type definitions will impose a heavier burden on the author to properly mark up a document.

4. The movement of files created on the Pyramid computer to the MCS typesetting system is difficult since a direct interface does not exist. This is not so much a conceptual problem as a practical one.

5. Straight batch processing of the typesetting files sometimes results in undesirable typographic effects. An example is the hyphenation of a word in a title.

With the MCS system, this can be remedied by disallowing hyphenation while such elements are being typeset, but that may in turn create wider than desirable interword spaces.

6. The Yacc specification must be written by hand from the type definition.

Each of these limitations must be addressed if a useable system is to be created.

Rather than being responsible for creating typesetting input files, the function of a parser ought to be to verify the conformance of a document to a type definition, and create an intermediate file which could then be further processed. Subsequent processing would deal with non-sequential elements and perform pagination functions.

The embedding of typesetting formats within the Yacc source makes them rather inaccessible. This could be remedied by locating the typesetting parameters for a given document type in a table. The parser would fetch values from this table. Other programs would also be able to manipulate the table. One program might allow a user to interactively establish typesetting formats for all of the elements in a document. Another might suggest a number of pre-defined formats and fill the table with values based on the user's choice.

The user's responsibility to produce a conforming document is a clear weakness of the system. A possible remedy would be to create a menu-driven interface which would help the user

structure a document. Such a program might begin with a data structure representing an empty document, and allow the user to fill each empty element with content. At any point in the document a menu would inform the user of the possible next legal elements specified by the type definition. All tags would be inserted automatically when a particular path were taken through the document tree.

In the examples shown in Appendix G, the files produced directly by the parser were not altered in any way before they were typeset. The MCS system does provide a sophisticated editing and preview facility at the front end of the typesetting system. The problem of hyphenated words in titles and other subtle violations can be fixed at this stage. To anticipate and fix all such problems before the file is input to the typesetting system would require such a powerful set of mechanisms that it does not seem worth pursuing.

It might be useful to have a program capable of reading an SGML type definition and building a Yacc specification automatically. But for type definitions with sufficient flexibility to handle all or most occurrences of a given document type such as books, essays, resumes, etc., the Yacc specification needs only to be written once. The hand coding of the Yacc specification file, therefore, is not a significant problem if the grammar is for a standard document structure that will be used repeatedly. If, on the other hand, arbitrary type definitions are created on the fly, and Yacc specifications are

needed instantaneously, an automatic Yacc program generator would be useful. The end user of a system would most likely never need such a facility.

The proper representation of document structures is the necessary first step in the development of any system capable of automatically formatting and typesetting the work of an author who has no knowledge about such matters. This thesis has shown that with the mechanism of generalized markup languages, it is relatively simple to translate a document from an author's logical form into a form that makes sense to a composition system.

8. REFERENCES

[CHICAGO] The University of Chicago Press, A Manual of Style, Chicago, 1969.

[COMPUGRAPHIC] Compugraphic 8600 Imagesetter Manual, Compugraphic Corporation Part Number 201033-002, January 1986.

[DOEBLER] Paul Doebler, 'Users Vs. Machines: Improving Communications,' EP&P87, February/March 1987.

[HANSARD] T.C. Hansard, Typographia: An Historical Sketch of the Origin and Progress of the Art of Printing, London, 1825.

[JOHNSON] Stephen C. Johnson, Yacc: Yet Another Compiler Compiler, Bell Laboratories, Murray Hill, New Jersey, 1978.

[KNUTH] Donald E. Knuth, The TeXbook, Reading MA, 1984.

[LESK] M.E. Lesk and E. Schmidt, Lex - A Lexical Analyser Generator, Bell Laboratories, Murray Hill, New Jersey, 1978.

[LEWIS] Harry R. Lewis and Christos H. Papadimitriou, Elements of the Theory of Computation, Englewood Cliffs, New Jersey, 1981.

[MCLEAN] Rauri McLean, The Thames and Hudson Manual of Typography, London, 1980.

[MOXON] Joseph Moxon, Mechanick Exercises on the Whole Art of Printing, London 1958.

[PLASS] Michael Frederick Plass, Optimal Pagination Techniques for Automatic Typesetting Systems, PH.D. Dissertation, Stanford University, 1981.

[ROSEN] Ben Rosen, TYPE&!.,-:;?", New York, 1976.

[RUPPELL] Aloys Ruppell, Johann Gutenberg, New York, 1947.

[SEYBOLD] John Seybold, The World of Digital Typesetting, Media, PA, 1984.

[SOUTHWARD] John Southward, Practical Printing: A Handbook of the Art of Typography, London, 1882.

[STEINBERG] S.H. Steinberg, Five Hundred Years of Printing, New York, 1979.

APPENDIX A

SUMMARY OF

TYPESETTING COMMANDS

The following commands represent just about everything you will need to know to produce professional looking typesetting. Typesetting commands fall into four categories: parameters, mnemonic codes, refinements and position commands.

Parameters include point size, line spacing, column width and typestyle. Mnemonic codes access functions or characters that are not on your computer keyboard. For instance, you need to type %st when you want a star (★). The various refinements or distortions of type covered by the third category include pseudo italic and slanted type. Position commands allow you to set the more complicated tabular material and therefore require a little more concentration.

Note that codes preceded by the percent sign (%) are entered using lower case. You may use upper or lower case when entering codes preceded by the less-than or command-on symbol (<).

1. PARAMETERS:

<LL__> or <ll__> = line length in picas and points (i.e., <LL21> is 21 picas and <LL14.6> is 14½ picas)

<PS__> or <ps__> = point size (4 to 72 points) (i.e., <PS12> is 12 point or <ps9> is 9 point)

<PSMAX> = inserts maximum point size. Prevents headline from breaking into two lines.

<LS__> or <ls__> = line space in points (i.e., <LS12> is 12 points of leading from baseline to baseline.) Quarter points of leading can be entered by adding .1 for ¼, .2 for ½ and .3 for ¾ point. (i.e., <LS12.2> is 12½ points of leading.

<FT__> or <ft__> = font or typestyle number as shown in typebook. In most typestyles the regular or book style ends in a "1", the italic ends in a "2", the bold ends in a "3" and the bold italic ends in a "4"

2. MNEMONIC CODES:

Each typeface has a font number and a keyboard number. Most typefaces use either the 500 or 501 keyboard. Having different keyboards gives you a wider variety of pi characters but it also requires that you know which keyboard you are using when you type a mnemonic code. For instance, if you want a copyright symbol (©), you need to switch to a font using the 501 keyboard. Usually this means changing to italic or bold italic and then switching back to whatever typestyle you were using. The following mnemonic codes are grouped by keyboard.

Use either the 500 or the 501 keyboard:

-- = em dash (—)
... = . . .
.... = ...
" " = quotes — on most computers it is necessary to hit the quote key twice for opening quotes and the apostrophe key twice for closing quotes.
%d = dot leader — Example: (Title%dpage 12) = Title.....page 12
%bx = box (□)
%st = star (★)
%z = slashed zero (Ø)
%ck = check mark (✓)
%rg = registered trademark (®)
%(= open bracket ([)
%) = closed bracket (])
%1/2, %1/4, %1/3, %3/4, %2/3 = ½, ¼, ⅓, ¾, ⅔

Use the 500 keyboard:

%b = bullet (•)
%p = percent sign (%)
%ct = cent sign (¢)
#$, #1, #2, #3, #4 = superior numbers (i.e., ^{1234})

Use the 501 keyboard:

-n- = en dash (–)
%+ = plus sign (+)
%= = equal sign (=)
%– = minus sign (–)
%x = multiply sign (×)
%/ = divide sign (÷)
%dg = degree (°)
%# = pound symbol (#)
%cpr = copyright (©)
%a = "at" symbol (@)
%bbx = black box (■)
%xb = extra big bullet (●)

General Mnemonic Typesetting Codes (All Keyboards)

► *FIXED SPACES:*

%i = thin space (same width value as most punctuation such as periods, decimal points and commas)
%n = en space (same width value as numbers, dollar signs, bullets, and asterisks)
%m = em space (use this for paragraph indent)

► *QUAD COMMANDS:*

%c = quad center (follow with a return)
%r = quad right (follow with a return)
%% = quad left (follow with a return)

► *BASELINE RULES:*

%l = baseline rule — Rules entire line length or portion remaining. It may be entered more than once in a line to create longer or shorter rules, for example: (City %l%l%l<is>State %l<is>Zip %l) = City_____ State_____ Zip_____
Notice the insert space commands. Any extra space will be inserted at these locations. This will prevent an irregular right margin.

%sl = single baseline rule (_) — six of these equal about one inch in 12 point type

► *END LINE COMMANDS:*

%el = end line — this followed by a return will return you to the left margin without leading down. If you are in a tab, it will return you to the left margin of that tab without leading down. (%c%el = quad center end line; %r%el = quad right end line; %%el = quad left end line)

%cr = carriage reset — this is similar to the end line command except it always takes you out of tab and returns you to the "true" left hand margin (%c%cr = quad center carriage reset; %r%cr = quad right carriage reset; %%cr = quad left carriage reset)

► *INSERT SPACE OR CHARACTERS:*

<IS> Insert space fills remaining space on a line with spaces; can be entered more than once on a line.
Example: (Vol 1<is>NEWS<is>JUNE) =
Vol 1 NEWS JUNE

<IC> Insert character fills remaining space on a line with the character immediately following this command.
Example: (<ic>%st) = ★

► *MNEMONIC TAB COMMANDS:*

%t = tab forward to next consecutive tab position.
%tr = tab return. This command takes you out of tab and back to the left margin. Be sure to enter a return after this code.

Note: You do not need to specify quading when using a tab forward or a tab return because a default was entered when you defined your tabs However, if you want to override the default, you can attach the appropriate quad command — i.e., %c%t, %r%t, %%t, %c%tr, %r%tr, %%tr.

3. REFINEMENTS:

<HYO> or <HYX> Hyphenation on or off. (Normally on).
<DH> discretionary hyphen — placed in front of a word, it will prevent that word from being hyphenated; placed in a word, it indicates a preferred hyphenation spot
<CC__> Set character compensation (i.e., <cc8>)
<CCO> and <CCX> Compensation on and off
<PI__> Pseudo italic, slants type to the right 1 to 31 degrees If no value is entered it defaults to 12°. Examples: *12-18-24-31 degrees.*
<PI__L> Pseudo italic, slants type to the left 1 to 31 degrees
<PIO> or <PIX>turns pseudo italic on or off
<SS__>Set size: Making the set size smaller than the point size condenses the type and making it larger than the point size expands the characters. The set size automatically resets to normal when the point size is changed.

4. POSITION COMMANDS:

► *INDENTS:*

<IB__> Indent both sides (i.e., <ib1.6> = indent one and a half picas on both sides of a column)
<IL__> or <IR__> Indent left or right side of a column
<IH__> Hanging indent; i.e., everything except the first line is indented from the left until a return is entered. This copy was done using a hanging indent of one pica.
<IO> or <IX> Turns previously defined indent on or off.

► *TABS:*

<TB__, __, __, __> Tab set (i.e., <tb1,12,14.6,C> defines tab 1 which is 12 picas from the left margin and 14.6 picas wide. Everything will be centered unless a different quad command is entered in the copy)
<AT__,__> Auto tab set (i.e., <at1,r> defines start of auto tab 1 with a default quad = quad right)
<TF__,__,__> Floating tab set (i.e., <tf1,4,j> defines a 4 pica wide floating tab with a default quad = justify)
<CT__> Call tab 1 through 19 (i.e., <ct1> calls tab 1)

► *DIRECTION:*

<ALD__> or <RLD__> Advance or reverse lead. Example: (NAME<rld4<ps4>TM<ald4<ps9> = NAME™)
<FP__> or <BP__> Forward or backward points (i.e., <fp72> moves forward 72 points or one inch)
<FU__> or <BU__> Forward and back units

► *MISCELLANEOUS:*

<NS> Exposes the next character but does not move forward (i.e., <ns>%bx<rld1<fu13>x = ☒(Note that additional vertical and forward positioning was necessary)
<NFO> or <NFX> Flash on or off (when off, typesetter will move forward as characters are entered but nothing will be exposed)

► *AUTO QUAD:*

<AL> Auto left (referred to as ragged right)
<AR> Auto right
<AC> Auto center
<AJ> Auto justify is the normal typesetting mode (entering this code turns off previous auto quad commands and returns you to normal justified copy)

► *MULTI COLUMN COMMANDS:*

<MCO__,__,__> = multi column. Enter the depth in picas (84 picas maximum), the first tab number and the number of tabs.
<MCO> sets baseline counter to zero
<MCR> returns you to the last "MCO" baseline
<MCA__> positions you at specified number of points from the last MCO baseline
<MCX> turns off the MCO and advance leads 9 points

► *REGULAR KEYBOARD CHARACTERS:*

The following characters can be entered without using a special code:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
1 2 3 4 5 6 7 8 9 0 $ & ? ! " ; : . , / ( ) – '

► *FORMATS:*

Formats allow you to creat short mnemonic codes representing repetitious codes or phrases. Before typing anything that will be used later, type an @ = followed by an alpha character and a single digit. This tells our computer that you want to define a format. After typing the codes or words, indicate the end of your format by typing @ ★. Later you can insert this copy by typing @ followed by the alpha character and the single digit number.

Example of formats being defined:

@ = s4<ft3<ps14<ls15<al<hyx<ll14>@ ★ This is a subhead
@ = t1<ps10<ls11<ft1<hyo<aj<ll14>@ ★ This is text . .

Using formats:

@s4This is a subhead. .
@t1This is regular text. . .

► *REMINDER:*

Typesetting commands like <AJ>, <AC>, <AR>, <AL>, <HYX>, <CCO>, <IO>, and <PIO> remain on until you turn them off or enter a different command. For example, once you enter <AL>, you will remain in the ragged right mode until you enter <AJ>, which returns you to justified text. The "no space" command (<NS>) stays on for only one character. The <NFO> command turns the flash unit off. It will turn back on as soon as a return key is entered. You may turn it on sooner by typing: <NFX>.

LEX SOURCE FILE FOR ESSAY DOCUMENT TYPE

```
%%
"<\t\n>"          ;
"<ESSAY>"         return (ESSAYTOKEN);
"</ESSAY>"        return (ENDESSAYTOKEN);
"<FM>"            return (FRONTTOKEN);
"</FM>"           return (ENDFRONTTOKEN);
"<BODY>"          return (BODYTOKEN);
"</BODY>"         return (ENDBODYTOKEN);
"<RM>"            return (REARTOKEN);
"</RM>"           return (ENDREARTOKEN);
"<T>"             return (TITLETOKEN);
"</T>"            return (ENDTITLETOKEN);
"<AU>"            return (AUTHORTOKEN);
"</AU>"           return (ENDAUTHORTOKEN);
"<AB>"            return (ABSTRACTTOKEN);
"</AB>"           return (ENDABSTRACTTOKEN);
"<T1>"            return (FIRSTTITLETOKEN);
"</T1>"           return (ENDFIRSTTITLETOKEN);
"<T2>"            return (SECONDTITLETOKEN);
"</T2>"           return (ENDSECONDTITLETOKEN);
"<INTRO>"         return (INTROTOKEN);
"</INTRO>"        return (ENDINTROTOKEN);
"<SEC>"           return (SECTOKEN);
"</SEC>"          return (ENDSECTOKEN);
"<ILP>"           return (INTROLEADPARTOKEN);
"</ILP>"          return (ENDINTROLEADPARTOKEN);
"<IP>"            return (INTROPARTOKEN);
"</IP>"           return (ENDINTROPARTOKEN);
"<ST>"            return (SECTITLETOKEN);
"</ST>"           return (ENDSECTITLETOKEN);
"<SB>"            return (SECBODYTOKEN);
"</SB>"           return (ENDSECBODYTOKEN);
"<SLP>"           return (SECLEADPARTOKEN);
"</SLP>"          return (ENDSECLEADPARTOKEN);
"<SP>"            return (SECPARTOKEN);
"</SP>"           return (ENDSECPARTOKEN);
"<BIB>"           return (BIBLIOTOKEN);
"</BIB>"          return (ENDBIBLIOTOKEN);
"<ACK>"           return (ACKNOWTOKEN);
"</ACK>"          return (ENDACKNOWTOKEN);
"<REF>"           return (REFTOKEN);
"</REF>"          return (ENDREFTOKEN);
"<REFTIT>"        return (REFTITLETOKEN);
"</REFTIT>"       return (ENDREFTITLETOKEN);
"<REFAU>"         return (REFAUTHORTOKEN);
"</REFAU>"        return (ENDREFAUTHORTOKEN);
"<REFINFO>"       return (REFINFOTOKEN);
"</REFINFO>"      return (ENDREFINFOTOKEN);
```

YACC SOURCE FILE FOR ESSAY DOCUMENT TYPE

```
%token  ESSAYTOKEN ENDESSAYTOKEN
        FRONTTOKEN ENDFRONTTOKEN
        BODYTOKEN ENDBODYTOKEN
        REARTOKEN ENDREARTOKEN
        TITLETOKEN ENDTITLETOKEN
        AUTHORTOKEN ENDAUTHORTOKEN
        ABSTRACTTOKEN ENDABSTRACTTOKEN
        FIRSTTITLETOKEN ENDFIRSTTITLETOKEN
        SECONDTITLETOKEN ENDSECONDTITLETOKEN
        INTROTOKEN ENDINTROTOKEN
        SECTOKEN ENDSECTOKEN
        INTROLEADPARTOKEN ENDINTROLEADPARTOKEN
        INTROPARTOKEN ENDINTROPARTOKEN
        FOOTNOTETOKEN ENDFOOTNOTETOKEN
        SECTITLETOKEN ENDSECTITLETOKEN
        SECBODYTOKEN ENDSECBODYTOKEN
        SECLEADPARTOKEN ENDSECLEADPARTOKEN
        SECPARTOKEN ENDSECPARTOKEN
        BIBLIOTOKEN ENDBIBLIOTOKEN
        ACKNOWTOKEN ENDACKNOWTOKEN
        REFTOKEN ENDREFTOKEN
        REFTITLETOKEN ENDREFTITLETOKEN
        REFAUTHORTOKEN ENDREFAUTHORTOKEN
        REFINFOTOKEN ENDREFINFOTOKEN


%start essay

%%

essay               :   ESSAYTOKEN frontmatter body rearmatter
                        ENDESSAYTOKEN
                    ;

frontmatter         :   FRONTTOKEN title author abstract
                        ENDFRONTTOKEN
                    ;

title               :   TITLETOKEN firsttitle secondtitle
                        ENDTITLETOKEN
                    ;

firsttitle          :   startfirsttitle ENDFIRSTTITLETOKEN
                    ;

startfirsttitle     :   FIRSTTITLETOKEN
                    ;
```

```
secondtitle          :    startsecondtitle ENDSECONDTITLETOKEN
                     |
                     ;

startsecondtitle     :    SECONDTITLETOKEN
                     ;

author               :    startauthor ENDAUTHORTOKEN
                     |    author author
                     ;

startauthor          :    AUTHORTOKEN
                     ;

abstract             :    startabstract ENDABSTRACTTOKEN
                     |
                     ;

startabstract        :    ABSTRACTTOKEN
                     ;

body                 :    BODYTOKEN intro section ENDBODYTOKEN
                     ;

intro                :    INTROTOKEN introleadpar intropar
                          ENDINTROTOKEN
                     ;

introleadpar         :    startilp ENDINTROLEADPARTOKEN
                     ;

startilp             :    INTROLEADPARTOKEN
                     ;

intropar             :    startip ENDINTROPARTOKEN
                     |    intropar intropar
                     |
                     ;

startip              :    INTROPARTOKEN
                     ;

section              :    SECTOKEN sectitle secbody ENDSECTOKEN
                     |    section section
                     ;

sectitle             :    startsectitle ENDSECTITLETOKEN
                     ;

startsectitle        :    SECTITLETOKEN
                     ;
```

```
secbody          :    SECBODYTOKEN secleadpar secpar
                      ENDSECBODYTOKEN
                 ;

secleadpar       :    startslp ENDSECLEADPARTOKEN
                 ;

startslp         :    SECLEADPARTOKEN
                 ;

secpar           :    startsp ENDSECPARTOKEN
                 |    secpar secpar
                 |
                 ;

startsp          :    SECPARTOKEN
                 ;

rearmatter       :    REARTOKEN biblio acknow ENDREARTOKEN
                 |
                 ;

biblio           :    startbiblio ref ENDBIBLIOTOKEN
                 |
                 ;

startbiblio      :    BIBLIOTOKEN
                 ;

ref              :    REFTOKEN refstuff refinfo ENDREFTOKEN
                 |    ref ref
                 ;

refstuff         :    reftitle refauthor
                 |    refauthor reftitle
                 ;

reftitle         :    startreftitle ENDREFTITLETOKEN
                 ;

startreftitle    :    REFTITLETOKEN
                 ;

refauthor        :    startrefauthor ENDREFAUTHORTOKEN
                 |    refauthor refauthor
                 ;

startrefauthor   :    REFAUTHORTOKEN
                 ;

refinfo          :    startrefinfo ENDREFINFOTOKEN
                 ;
```

```
startrefinfo        :    REFINFOTOKEN
                    ;

acknow              :    startacknow ENDACKNOWTOKEN
                    |
                    ;

startacknow         :    ACKNOWTOKEN
                    ;

%%
# include "lex.yy.c"
```

Y.OUTPUT FILE REPRESENTING THE PARSER CREATED BY YACC FROM THE SPECIFICATION IN APPENDIX C

```
state 0
        $accept : _essay $end

        ESSAYTOKEN  shift 2
        .  error

        essay  goto 1


state 1
        $accept :  essay_$end

        $end  accept
        .  error


state 2
        essay :  ESSAYTOKEN_frontmatter body rearmatter
                 ENDESSAYTOKEN

        FRONTTOKEN  shift 4
        .  error

        frontmatter  goto 3


state 3
        essay :  ESSAYTOKEN frontmatter_body rearmatter
                 ENDESSAYTOKEN

        BODYTOKEN  shift 6
        .  error

        body  goto 5


state 4
        frontmatter :  FRONTTOKEN_title author abstract
                       ENDFRONTTOKEN

        TITLETOKEN  shift 8
        .  error

        title  goto 7
```

```
state 5
        essay :  ESSAYTOKEN frontmatter body_rearmatter
                 ENDESSAYTOKEN
        rearmatter : _    (38)

        REARTOKEN  shift 10
        .  reduce 38

        rearmatter  goto 9

state 6
        body :  BODYTOKEN_intro section ENDBODYTOKEN

        INTROTOKEN  shift 12
        .  error

        intro  goto 11

state 7
        frontmatter :  FRONTTOKEN title_author abstract
                       ENDFRONTTOKEN

        AUTHORTOKEN  shift 15
        .  error

        author  goto 13
        startauthor  goto 14

state 8
        title :   TITLETOKEN_firsttitle secondtitle ENDTITLETOKEN


        FIRSTTITLETOKEN  shift 18
        .  error

        firsttitle  goto 16
        startfirsttitle  goto 17

state 9
        essay :  ESSAYTOKEN frontmatter body
                 rearmatter_ENDESSAYTOKEN

        ENDESSAYTOKEN  shift 19
        .  error
```

```
state 10
        rearmatter :  REARTOKEN_biblio acknow ENDREARTOKEN
        biblio : _    (40)

        BIBLIOTOKEN  shift 22
        .  reduce 40

        biblio  goto 20
        startbiblio  goto 21

state 11
        body :  BODYTOKEN intro_section ENDBODYTOKEN

        SECTOKEN  shift 24
        .  error

        section  goto 23

state 12
        intro :  INTROTOKEN_introleadpar intropar ENDINTROTOKEN

        INTROLEADPARTOKEN  shift 27
        .  error

        introleadpar  goto 25
        startilp  goto 26

state 13
        frontmatter :  FRONTTOKEN title author_abstract
                       ENDFRONTTOKEN
        author :  author_author
        abstract : __    (13)

        AUTHORTOKEN  shift 15
        ABSTRACTTOKEN  shift 31
        .  reduce 13

        author  goto 29
        abstract  goto 28
        startauthor  goto 14
        startabstract  goto 30

state 14
        author :  startauthor_ENDAUTHORTOKEN

        ENDAUTHORTOKEN  shift 32
        .  error

state 15
        startauthor :  AUTHORTOKEN_    (11)

        .  reduce 11
```

```
state 16
        title :   TITLETOKEN firsttitle_secondtitle ENDTITLETOKEN

        secondtitle : _    (7)

        SECONDTITLETOKEN  shift 35
        .  reduce 7

        secondtitle  goto 33
        startsecondtitle  goto 34

state 17
        firsttitle :  startfirsttitle_ENDFIRSTTITLETOKEN

        ENDFIRSTTITLETOKEN  shift 36
        .  error


state 18
        startfirsttitle :  FIRSTTITLETOKEN_    (5)

        .  reduce 5


state 19
        essay :  ESSAYTOKEN frontmatter body rearmatter
ENDESSAYTOKEN_    (1)

        .  reduce 1


state 20
        rearmatter :  REARTOKEN biblio_acknow ENDREARTOKEN
        acknow : _    (54)

        ACKNOWTOKEN  shift 39
        .  reduce 54

        acknow  goto 37
        startacknow  goto 38

state 21
        biblio :  startbiblio_ref ENDBIBLIOTOKEN

        REFTOKEN  shift 41
        .  error

        ref  goto 40
```

```
state 22
        startbiblio :  BIBLIOTOKEN_    (41)

        .  reduce 41


state 23
        body :  BODYTOKEN intro section_ENDBODYTOKEN
        section :  section_section

        ENDBODYTOKEN  shift 42
        SECTOKEN  shift 24
        .  error

        section  goto 43

state 24
        section :  SECTOKEN_sectitle secbody ENDSECTOKEN

        SECTITLETOKEN  shift 46
        .  error

        sectitle  goto 44
        startsectitle  goto 45

25: shift/reduce conflict (shift 49, red'n 24) on INTROPARTOKEN

state 25
        intro :  INTROTOKEN introleadpar_intropar ENDINTROTOKEN
        intropar : _    (24)

        INTROPARTOKEN  shift 49
        .  reduce 24

        intropar  goto 47
        startip  goto 48

state 26
        introleadpar :  startilp_ENDINTROLEADPARTOKEN

        ENDINTROLEADPARTOKEN  shift 50
        .  error


state 27
        startilp :  INTROLEADPARTOKEN_    (21)

        .  reduce 21
```

```
state 28
        frontmatter :  FRONTTOKEN title author
                       abstract_ENDFRONTTOKEN

        ENDFRONTTOKEN  shift 51
        .  error


29: shift/reduce conflict (shift 15, red'n 10) on AUTHORTOKEN

state 29
        author :  author_author
        author :  author author_    (10)

        AUTHORTOKEN  shift 15
        .  reduce 10

        author  goto 29
        startauthor  goto 14

state 30
        abstract :  startabstract_abpar ENDABSTRACTTOKEN

        ABPARTOKEN  shift 54
        .  error

        abpar  goto 52
        startabpar  goto 53

state 31
        startabstract :  ABSTRACTTOKEN_    (14)

        .  reduce 14


state 32
        author :  startauthor ENDAUTHORTOKEN_    (9)

        .  reduce 9


state 33
        title :   TITLETOKEN firsttitle secondtitle_ENDTITLETOKEN


        ENDTITLETOKEN  shift 55
        .  error
```

```
state 34
        secondtitle :  startsecondtitle_ENDSECONDTITLETOKEN

        ENDSECONDTITLETOKEN  shift 56
        .  error


state 35
        startsecondtitle :  SECONDTITLETOKEN_    (8)

        .  reduce 8


state 36
        firsttitle :  startfirsttitle ENDFIRSTTITLETOKEN_    (4)

        .  reduce 4


state 37
        rearmatter :  REARTOKEN biblio acknow_ENDREARTOKEN

        ENDREARTOKEN  shift 57
        .  error

state 38
        acknow :  startacknow_ENDACKNOWTOKEN

        ENDACKNOWTOKEN  shift 58
        .  error


state 39
        startacknow :  ACKNOWTOKEN_    (55)

        .  reduce 55


state 40
        biblio :  startbiblio ref_ENDBIBLIOTOKEN
        ref :  ref_ref

        ENDBIBLIOTOKEN  shift 59
        REFTOKEN  shift 41
        .  error

        ref  goto 60
```

```
state 41
        ref :  REFTOKEN_refstuff refinfo ENDREFTOKEN

        REFTITLETOKEN  shift 66
        REFAUTHORTOKEN  shift 67
        .  error

        refstuff  goto 61
        reftitle  goto 62
        refauthor  goto 63
        startreftitle  goto 64
        startrefauthor  goto 65

state 42
        body :  BODYTOKEN intro section ENDBODYTOKEN_    (18)

        .  reduce 18


43: shift/reduce conflict (shift 24, red'n 27) on SECTOKEN

state 43
        section :  section_section
        section :  section section_    (27)

        SECTOKEN  shift 24
        .  reduce 27

        section  goto 43


state 44
        section :  SECTOKEN sectitle_secbody ENDSECTOKEN

        SECBODYTOKEN  shift 69
        .  error

        secbody  goto 68

state 45
        sectitle :  startsectitle_ENDSECTITLETOKEN

        ENDSECTITLETOKEN  shift 70
        .  error


state 46
        startsectitle :  SECTITLETOKEN_    (29)

        .  reduce 29
```

```
47: shift/reduce conflict (shift 71, red'n 24) on ENDINTROTOKEN
47: shift/reduce conflict (shift 49, red'n 24) on INTROPARTOKEN

state 47
        intro :  INTROTOKEN introleadpar intropar_ENDINTROTOKEN
        intropar :  intropar_intropar
        intropar : _    (24)

        ENDINTROTOKEN  shift 71
        INTROPARTOKEN  shift 49
        .  error

        intropar  goto 72
        startip  goto 48

state 48
        intropar :  startip_ENDINTROPARTOKEN

        ENDINTROPARTOKEN  shift 73
        .  error


state 49
        startip :  INTROPARTOKEN_    (25)

        .  reduce 25


state 50
        introleadpar :  startilp ENDINTROLEADPARTOKEN_    (20)

        .  reduce 20

state 51
        frontmatter :  FRONTTOKEN title author abstract
                       ENDFRONTTOKEN_    (2)

        .  reduce 2


state 52
        abstract :  startabstract abpar_ENDABSTRACTTOKEN
        abpar :  abpar_abpar

        ENDABSTRACTTOKEN  shift 74
        ABPARTOKEN  shift 54
        .  error

        abpar  goto 75
        startabpar  goto 53
```

```
state 53
        abpar :  startabpar_ENDABPARTOKEN

        ENDABPARTOKEN  shift 76
        .  error


state 54
        startabpar :  ABPARTOKEN_    (17)

        .  reduce 17


state 55
        title :  TITLETOKEN firsttitle secondtitle ENDTITLETOKEN_
                 (3)

        .  reduce 3


state 56
        secondtitle :  startsecondtitle ENDSECONDTITLETOKEN_
                       (6)

        .  reduce 6


state 57
        rearmatter :  REARTOKEN biblio acknow ENDREARTOKEN_
                      (37)

        .  reduce 37



state 58
        acknow :  startacknow ENDACKNOWTOKEN_    (53)

        .  reduce 53


state 59
        biblio :  startbiblio ref ENDBIBLIOTOKEN_    (39)

        .  reduce 39


60: shift/reduce conflict (shift 41, red'n 43) on REFTOKEN
```

```
state 60
        ref :  ref_ref
        ref :  ref ref_    (43)

        REFTOKEN  shift 41
        .  reduce 43

        ref  goto 60

state 61
        ref :  REFTOKEN refstuff_refinfo ENDREFTOKEN

        REFINFOTOKEN  shift 79
        .  error

        refinfo  goto 77
        startrefinfo  goto 78

state 62
        refstuff :  reftitle_refauthor

        REFAUTHORTOKEN  shift 67
        .  error

        refauthor  goto 80
        startrefauthor  goto 65

state 63
        refstuff :  refauthor_reftitle
        refauthor :  refauthor_refauthor

        REFTITLETOKEN  shift 66
        REFAUTHORTOKEN  shift 67
        .  error

        reftitle  goto 81
        refauthor  goto 82
        startreftitle  goto 64
        startrefauthor  goto 65

state 64
        reftitle :  startreftitle_ENDREFTITLETOKEN

        ENDREFTITLETOKEN  shift 83
        .  error


state 65
        refauthor :  startrefauthor_ENDREFAUTHORTOKEN

        ENDREFAUTHORTOKEN  shift 84
        .  error
```

```
state 66
        startreftitle :  REFTITLETOKEN_    (47)

        .  reduce 47


state 67
        startrefauthor :  REFAUTHORTOKEN_    (50)

        .  reduce 50


state 68
        section :  SECTOKEN sectitle secbody_ENDSECTOKEN

        ENDSECTOKEN  shift 85
        .  error


state 69
        secbody :  SECBODYTOKEN_secleadpar secpar ENDSECBODYTOKEN


        SECLEADPARTOKEN  shift 88
        .  error

        secleadpar  goto 86
        startslp  goto 87

state 70
        sectitle :  startsectitle ENDSECTITLETOKEN_    (28)

        .  reduce 28


state 71
        intro :  INTROTOKEN  introleadpar intropar ENDINTROTOKEN_
                 (19)

        .  reduce 19
72: shift/reduce conflict (shift 49, red'n 23) on INTROPARTOKEN
72: reduce/reduce conflict (red'ns 23 and 24 ) on ENDINTROTOKEN
72: shift/reduce conflict (shift 49, red'n 24) on INTROPARTOKEN
```

```
state 72
        intropar :  intropar_intropar
        intropar :  intropar intropar_    (23)
        intropar : _    (24)

        INTROPARTOKEN  shift 49
        .  reduce 23

        intropar  goto 72
        startip  goto 48

state 73
        intropar :  startip ENDINTROPARTOKEN_    (22)

        .  reduce 22


state 74
        abstract :  startabstract abpar ENDABSTRACTTOKEN_    (12)


        .  reduce 12


75: shift/reduce conflict (shift 54, red'n 16) on ABPARTOKEN

state 75
        abpar :  abpar_abpar
        abpar :  abpar abpar_    (16)

        ABPARTOKEN  shift 54
        .  reduce 16

        abpar  goto 75
        startabpar  goto 53

state 76
        abpar :  startabpar ENDABPARTOKEN_    (15)

        .  reduce 15


state 77
        ref :  REFTOKEN refstuff refinfo_ENDREFTOKEN

        ENDREFTOKEN  shift 89
        .  error
```

```
state 78
        refinfo :  startrefinfo_ENDREFINFOTOKEN

        ENDREFINFOTOKEN  shift 90
        .  error


state 79
        startrefinfo :  REFINFOTOKEN_    (52)

        .  reduce 52


state 80
        refstuff :  reftitle refauthor_    (44)
        refauthor :  refauthor_refauthor

        REFAUTHORTOKEN  shift 67
        .  reduce 44

        refauthor  goto 82
        startrefauthor  goto 65

state 81
        refstuff :  refauthor reftitle_    (45)

        .  reduce 45


82: shift/reduce conflict (shift 67, red'n 49) on REFAUTHORTOKEN

state 82
        refauthor :  refauthor_refauthor
        refauthor :  refauthor refauthor_    (49)

        REFAUTHORTOKEN  shift 67
        .  reduce 49

        refauthor  goto 82
        startrefauthor  goto 65

state 83
        reftitle :  startreftitle ENDREFTITLETOKEN_    (46)

        .  reduce 46


state 84
        refauthor :  startrefauthor ENDREFAUTHORTOKEN_    (48)

        .  reduce 48
```

```
state 85
        section :  SECTOKEN sectitle secbody ENDSECTOKEN_     (26)

        .  reduce 26

86: shift/reduce conflict (shift 93, red'n 35) on SECPARTOKEN

state 86
        secbody :  SECBODYTOKEN secleadpar_secpar ENDSECBODYTOKEN

        secpar : _     (35)

        SECPARTOKEN  shift 93
        .  reduce 35

        secpar  goto 91
        startsp  goto 92

state 87
        secleadpar :  startslp_ENDSECLEADPARTOKEN

        ENDSECLEADPARTOKEN  shift 94
        .  error

state 88
        startslp :  SECLEADPARTOKEN_     (32)

        .  reduce 32

state 89
        ref :  REFTOKEN refstuff refinfo ENDREFTOKEN_     (42)

        .  reduce 42

state 90
        refinfo :  startrefinfo ENDREFINFOTOKEN_     (51)

        .  reduce 51

91: shift/reduce conflict (shift 95, red'n 35) on ENDSECBODYTOKEN

91: shift/reduce conflict (shift 93, red'n 35) on SECPARTOKEN
```

```
state 91
        secbody :  SECBODYTOKEN secleadpar secpar_ENDSECBODYTOKEN

        secpar :  secpar_secpar
        secpar : _    (35)

        ENDSECBODYTOKEN  shift 95
        SECPARTOKEN  shift 93
        .  error

        secpar  goto 96
        startsp  goto 92

state 92
        secpar :  startsp_ENDSECPARTOKEN

        ENDSECPARTOKEN  shift 97
        .  error


state 93
        startsp :  SECPARTOKEN_    (36)

        .  reduce 36


state 94
        secleadpar :  startslp ENDSECLEADPARTOKEN_    (31)

        .  reduce 31


state 95
        secbody :  SECBODYTOKEN secleadpar secpar
                   ENDSECBODYTOKEN_    (30)

        .  reduce 30


96: shift/reduce conflict (shift 93, red'n 34) on SECPARTOKEN
96: reduce/reduce conflict (red'ns 34 and 35 ) on ENDSECBODYTOKEN
96: shift/reduce conflict (shift 93, red'n 35) on SECPARTOKEN
```

```
state 96
        secpar :  secpar_secpar
        secpar :  secpar secpar_    (34)
        secpar : _    (35)

        SECPARTOKEN  shift 93
        .  reduce 34

        secpar  goto 96
        startsp  goto 92
state 97
        secpar :  startsp ENDSECPARTOKEN_    (33)

        .  reduce 33


52/600 terminals, 40/700 nonterminals
56/1000 grammar rules, 98/2000 states
15 shift/reduce, 2 reduce/reduce conflicts reported
40/700 working sets used
memory: states,etc. 447/24000, parser 56/24000
57/1000 distinct lookahead sets
0 extra closures
65 shift entries, 1 exceptions
49 goto entries
21 entries saved by goto default
Optimizer space used: input 190/24000, output 143/24000
143 table entries, 39 zero
maximum spread: 306, maximum offset: 306
```

DOCUMENT CONFORMING TO THE ESSAY TYPE DEFINITION

<ESSAY>
<FM>
<T>
<T1>
The Design and Implementation of a System for Processing Documents Described in Generalized Markup Languages
</T1>
</T>
<AU>
Frank J. Cost
</AU>
<AB>
<AP>
This thesis proposes an architecture for an expert system capable of transforming the work of an author into printed form, and builds a partial implementation based upon that architecture. The system allows the author to describe a document as a hierarchical structure without any concern for how it will be transformed into print. The author is provided with a special markup language which can be used to tag the various logical parts of a document, along with an idea of how the various tagged parts can fit together. The system is able to read the author's manuscript as input, and produce a typeset object as output.
</AP>
</AB>
</FM>
<BODY>
<INTRO>
<ILP>
This thesis is concerned with the mechanisms used to transform manuscripts into print. Ever since the invention of printing from movable types more than five centuries ago it has been necessary to supply detailed information along with a manuscript indicating how it should appear in final form. A printer would use this information to determine how to assemble individual pieces of type so as to create the requisite printing surfaces. These surfaces would then be used to transfer a light-absorbing substance called ink to a light-reflecting surface called paper to create a printed artifact.
</ILP>
<IP>
At first, information of this kind was in the form of instructions to be followed by human beings who were responsible for hand composing the assemblages of type. When the process of composition was automated beginning in the late 19th century, markup had to be translated into specific instructions which would cause a particular machine to set type accordingly. The

expertise which had once been required of the hand compositor was now required of the person preparing input for the machine.
</IP>
<IP>
An author thinks of a manuscript as a logical construct. A book, for example, is best seen as a hierarchical collection of parts. The graph of figure 1.1 shows the structure of a book as an author might imagine it. A book consists of a set of elements called front matter followed by another set of elements called body, followed by a third set called rear matter. The front matter consists of a title page followed by a dedication page followed by a table of contents followed by one or more optional prefaces. Each of these elements can in turn be broken into yet smaller elements. A title page, for instance, consists of a title followed by a subtitle followed by an author's name followed by a publisher's imprint.
</IP>
</INTRO>
<SEC>
<ST>
The Invention and Development of Typography
</ST>
<SB>
<SLP>
Prior to the invention of printing from movable types, the composition of written pages images was accomplished either by manually applying ink to paper with a brush or pen, or by preparing a carved relief surface which was then used to transfer ink to paper. The former technique is called calligraphy, and the latter is called xylography. Calligraphy and xylography differ fundamentally in that xylography allows multiple copies of the same image to be reproduced whereas calligraphy does not.
</SLP>
<SP>
It is useful, however, to consider calligraphy and xylography as constituting a single paradigm. Both allow the freehand placement of light-absorbing material on a surfcace to form an image, and while it may be technically difficult to duplicate a calligraphic form by carving an image into a block of wood, it is conceptually possible to do so.</SP>
<SP>
The creation of textual images from movable types is called typography. Typography is not part of the same paradigm as calligraphy and xylography. It involves the assembly of images from discrete elements which can only fit together in certain ways....
</SP>
<SP>
Johannes Gutenberg, the inventor of typography, intended for the product of his printing press to closely resemble the style of writing called....
</SP>

</SB>
</SEC>
<SEC>
<ST>
Machine Composition
</ST>
<SB>
<SLP>
The first successful attempt to mechanize the composition process was the work of Ottmar Merganthaler. His Machine came to be known as the linotype because it produced whole lines of letters cast on a single body called a slug. A keyboard operator would enter the a sequence of characters to be typeset. Each keystroke would release an appropriate brass matrix from a magazine containing matrices for all the characters in a particular font....
</SLP>
<SP>
Less than a decade after the introduction of the linotype, a machine called the monotype was introduced. The machine used a punched paper tape to drive the casting unit. The system consisted of two machines....
</SP>
</SB>
</SEC>
</BODY>
<RM>
<BIB>
<REF>
<REFAU>
McLean, Rauri
</REFAU>
<REFTIT>
The Thames and Hudson Manual of Typography
</REFTIT>
<REFINFO>
Thames and Hudson Publishing Company, London, 1980
</REFINFO>
</REF>
<REF>
<REFAU>
Moxon, Joseph
</REFAU>
<REFTIT>
Mechanick Exercises on the Whole Art of Printing
</REFTIT>
<REFINFO>
London, 1958
</REFINFO>
</REF>
<REF>
<REFAU>

Ruppell, Aloys
</REFAU>
<REFTIT>
Johann Gutenberg
</REFTIT>
<REFINFO>
New York, 1947
</REFINFO>
</REF>
</BIB>
<ACK>
I would like to acknowledge the help of the following people: Patricia A. Cost, Roger P. Cost, Guy Johnson, Jeff Lasky, Michael Kleper, James Hamilton, Emery Schneider and Darrin Ward.
</ACK>
</RM>
</ESSAY>

YACC SOURCE FILE 1

```
%start essay
%%
essay               : ESSAYTOKEN frontmatter body rearmatter
                      ENDESSAYTOKEN
                    ;
frontmatter         : FRONTTOKEN title author abstract
                      ENDFRONTTOKEN
                    ;
title               : TITLETOKEN firsttitle secondtitle
                      ENDTITLETOKEN
                            {
                            printf("<ALD36>");
                            }
                    ;
firsttitle          : startfirsttitle ENDFIRSTTITLETOKEN
                            {
                            printf("%%%%\n");
                            }
                    ;
startfirsttitle     : FIRSTTITLETOKEN
                            {
                            printf("<LL30><FT1860><PS16><LS20>");
                            }
                    ;
secondtitle         : startsecondtitle ENDSECONDTITLETOKEN
                            {
                            printf("%%%%\n");
                            }
                    |
                    ;
startsecondtitle    : SECONDTITLETOKEN
                            {
                            printf("<FT1740><PS18><LS28>");
                            }
                    ;
author              : startauthor ENDAUTHORTOKEN
                            {
                            printf("%%c\n");
                            }
                    | author author
                    ;
startauthor         : AUTHORTOKEN
                            {
                            printf("<FT1700><PS12><LS24>");
                            printf("by%%c\n");
                            printf("<PS14><LS36>");
                            }
                    ;
```

```
abstract        : startabstract abpar ENDABSTRACTTOKEN
                        {
                        printf("<ix>");
                        printf("<ALD48>");
                        }
                |
                        {
                        printf("<ALD48>");
                        }
                ;
startabstract   : ABSTRACTTOKEN
                        {
                        printf("<ALD42>");
                        printf("<FT2660><PS14><LS24>");
                        printf("ABSTRACT%%c\n");
                        printf("<FT2640><PS12><LS13>");
                        }
                ;
abpar           : startabpar ENDABPARTOKEN
                        {
                        printf("%%%%\n");
                        }
                | abpar abpar
                ;
startabpar      : ABPARTOKEN
                        {
                        printf("<ib2><io>%%m");
                        }
                ;
body            : BODYTOKEN intro section ENDBODYTOKEN
                        {
                        printf("<ALD72>");
                        }
                ;
intro           : INTROTOKEN introleadpar intropar
                  ENDINTROTOKEN
                        {
                        printf("<ALD24>");
                        }
                ;
introleadpar    : startilp ENDINTROLEADPARTOKEN
                        {
                        printf("%%%%\n");
                        }
                ;
startilp        : INTROLEADPARTOKEN
                        {
                        printf("<PS14><FT0641><LS24>");
                        printf("INTRODUCTION%%c\n");
                        printf("<PS12><FT0640><LS13>");
                        }
                ;
```

```
intropar        : startip ENDINTROPARTOKEN
                        {
                        printf("%%%%\n");
                        }
                | intropar intropar
                |
                ;
startip         : INTROPARTOKEN
                        {
                        printf("<PS10><LS11>%%m");
                        }
                ;
section         : SECTOKEN sectitle secbody ENDSECTOKEN

                        {
                        printf("<ALD24>");
                        }
                | section section
                ;
sectitle        : startsectitle ENDSECTITLETOKEN
                        {
                        printf("%%%%\n");
                        printf("<AJ>");
                        }
                ;
startsectitle   : SECTITLETOKEN
                        {
                        printf("<PS14><LS18>");
                        printf("<AL>");
                        }
                ;
secbody         : SECBODYTOKEN secleadpar secpar
                  ENDSECBODYTOKEN
                ;
secleadpar      : startslp ENDSECLEADPARTOKEN
                        {
                        printf("%%%%\n");
                        }
                ;
startslp        : SECLEADPARTOKEN
                        {
                        printf("<PS10><LS11>");
                        }
                ;
secpar          : startsp ENDSECPARTOKEN
                        {
                        printf("%%%%\n");
                        }
                | secpar secpar
                |
                ;
```

```
startsp          : SECPARTOKEN
                         {
                         printf("%%m");
                         }
                 ;
rearmatter       : REARTOKEN biblio acknow ENDREARTOKEN
                 |
                 ;
biblio           : startbiblio ref ENDBIBLIOTOKEN
                 |
                 ;
startbiblio      : BIBLIOTOKEN
                         {
                         printf("<FT2660><PS14><LS24>");
                         printf("BIBLIOGRAPHY%%c\n");
                         printf("<FT1700><PS10><LS11>");
                         }
                 ;
ref              : REFTOKEN refstuff refinfo ENDREFTOKEN
                         {
                         printf("%%%%\n\n");
                         }
                 | ref ref
                 ;
refstuff         : reftitle refauthor
                 | refauthor reftitle
                 ;
reftitle         : startreftitle ENDREFTITLETOKEN
                         {
                         printf(", ");
                         }
                 ;
startreftitle    : REFTITLETOKEN
                         {
                         printf("<FT0641>");
                         }
                 ;
refauthor        : startrefauthor ENDREFAUTHORTOKEN
                         {
                         printf(", ");
                         }
                 | refauthor refauthor
                 ;
startrefauthor   : REFAUTHORTOKEN
                         {
                         printf("<FT0640>");
                         }
                 ;
refinfo          : startrefinfo ENDREFINFOTOKEN
                 ;
```

```
startrefinfo      : REFINFOTOKEN
                        {
                        printf("<FT0640>");
                        }
                  ;
acknow            : startacknow ENDACKNOWTOKEN
                        {
                        printf("%%%%\n");
                        }
                  |
                  ;
startacknow       : ACKNOWTOKEN
                        {
                        printf("<ALD72>");
                        printf("<FT2660><PS14><LS24>");
                        printf("ACKNOWLEDGEMENTS%%c\n");
                        printf("<AL><FT1700><PS12><LS13>");
                        }
                  ;


%%
# include "lex.yy.c"
```

YACC SOURCE FILE 2

```
%start essay
%%
essay               :ESSAYTOKEN frontmatter body rearmatter
                     ENDESSAYTOKEN;
frontmatter         :FRONTTOKEN title author abstract ENDFRONTTOKEN
                            {
                            printf("<AL>");
                            }
                    ;
title               :TITLETOKEN firsttitle secondtitle
                     ENDTITLETOKEN
                            {
                            printf("<ALD36>");
                            }
                    ;
firsttitle          :startfirsttitle ENDFIRSTTITLETOKEN
                            {
                            printf("%%c\n");
                            }
                    ;
startfirsttitle     :FIRSTTITLETOKEN
                            {
                            printf("<LL32><FT4061><PS14><LS16>");
                            }
                    ;
secondtitle         :startsecondtitle ENDSECONDTITLETOKEN
                            {
                            printf("%%%%\n");
                            }
                    |
                    ;
startsecondtitle    :SECONDTITLETOKEN
                            {
                            printf("<FT4041><PS14><LS16>");
                            }
                    ;
author              :startauthor ENDAUTHORTOKEN
                            {
                            printf("%%c\n");
                            }
                    |author author
                    ;
```

```
startauthor     :AUTHORTOKEN
                        {
                        printf("<FT4010><PS14><LS24>");
                        printf("by%%c\n");
                        printf("<PS14><LS36>");
                        }
                ;
abstract        :startabstract abpar ENDABSTRACTTOKEN
                        {
                        printf("<ix>");
                        printf("<ALD48>");
                        }
                |
                        {
                        printf("<ALD48>");
                        }
                ;
startabstract   :ABSTRACTTOKEN
                        {
                        printf("<ALD42>");
                        printf("<FT4060><PS12><LS20>");
                        printf("ABSTRACT%%c\n");
                        printf("<FT4041><PS10><LS12>");
                        }
                ;
abpar           :startabpar ENDABPARTOKEN
                        {
                        printf("%%%%\n");
                        }
                |abpar abpar
                ;
startabpar      :ABPARTOKEN
                        {
                        printf("<ib3><io>");
                        }
                ;
body            :BODYTOKEN intro section ENDBODYTOKEN
                        {
                        printf("<ALD72>");
                        }
                ;
intro           :INTROTOKEN introleadpar intropar ENDINTROTOKEN
                        {
                        printf("<ALD24>");
                        }
                ;
introleadpar    :startilp ENDINTROLEADPARTOKEN
                        {
                        printf("%%%%\n");
                        }
                ;
```

```
startilp        :INTROLEADPARTOKEN
                        {
                        printf("<PS14><FT4060><LS24>");
                        printf("INTRODUCTION &
                                BACKGROUND%%%%\n");
                        printf("<PS12><FT4040><LS14>");
                        }
                ;
intropar        :startip ENDINTROPARTOKEN
                        {
                        printf("%%%%\n");
                        }
                |intropar intropar
                |
                ;
startip         :INTROPARTOKEN
                        {
                        printf("<PS10><LS12>%%m");
                        }
                ;
section         :SECTOKEN sectitle secbody ENDSECTOKEN

                        {
                        printf("<ALD24>");
                        }
                |section section
                ;
sectitle        :startsectitle ENDSECTITLETOKEN
                        {
                        printf("%%%%\n");
                        printf("<FT4040>");
                        }
                ;
startsectitle   :SECTITLETOKEN
                        {
                        printf("<FT4061><PS12><LS16>");
                        }
                ;
secbody         :SECBODYTOKEN secleadpar secpar
                 ENDSECBODYTOKEN;
secleadpar      :startslp ENDSECLEADPARTOKEN
                        {
                        printf("%%%%\n");
                        }
                ;
startslp        :SECLEADPARTOKEN
                        {
                        printf("<PS10><LS12>");
                        }
                ;
```

```
secpar          :startsp ENDSECPARTOKEN
                        {
                        printf("%%%%\n");
                        }
                |secpar secpar
                |
                ;
startsp         :SECPARTOKEN
                        {
                        printf("%%m");
                        }
                ;
rearmatter      :REARTOKEN biblio acknow ENDREARTOKEN
                |
                ;
biblio          :startbiblio ref ENDBIBLIOTOKEN
                |
                ;
startbiblio     :BIBLIOTOKEN
                        {
                        printf("<FT4060><PS14><LS24>");
                        printf("BIBLIOGRAPHY%%%%\n");
                        printf("<FT4040><PS10><LS11>");
                        }
                ;
ref             :REFTOKEN refstuff refinfo ENDREFTOKEN
                        {
                        printf("%%%%\n\n");
                        }
                |ref ref
                ;
refstuff        :reftitle refauthor
                |refauthor reftitle
                ;
reftitle        :startreftitle ENDREFTITLETOKEN
                        {
                        printf(", ");
                        }
                ;
startreftitle   :REFTITLETOKEN
                        {
                        printf("<FT4041>");
                        }
                ;
refauthor       :startrefauthor ENDREFAUTHORTOKEN
                        {
                        printf(", ");
                        }
                |refauthor refauthor
                ;
```

```
startrefauthor      :REFAUTHORTOKEN
                            {
                            printf("<FT4040>");
                            }
                    ;
refinfo             :startrefinfo ENDREFINFOTOKEN
                    ;
startrefinfo        :REFINFOTOKEN
                            {
                            printf("<FT4040>");
                            }
                    ;



acknow              :startacknow ENDACKNOWTOKEN
                            {
                            printf("%%%%\n");
                            }
                    |
                    ;
startacknow         :ACKNOWTOKEN
                            {
                            printf("<ALD72>");
                            printf("<FT4060><PS14><LS24>");
                            printf("ACKNOWLEDGEMENTS%%%%\n");
                            printf("<AL><FT4041><PS12><LS14>");
                            }
                    ;
%% # include "lex.yy.c"
```

MCS INPUT FILE 1

<LL30><FT1860><PS16><LS20>The Design and Implementation of a System for Processing Documents Described in Generalized Markup Languages%%
<ALD36><FT1700><PS12><LS24>by%c
<PS14><LS36>Frank J. Cost%c
<ALD42><FT2660><PS14><LS24>ABSTRACT%c
<FT2640><PS12><LS13><ib2><io>%mThis thesis proposes an architecture for a system capable of transforming the work of an author into printed form, and builds a partial implementation based upon that architecture. The system allows the author to describe a document as a hierarchical structure without any concern for how it will be transformed into print. The author is provided with a special markup language which can be used to tag the various logical parts of a document, along with an idea of how the various tagged parts can fit together. The system is able to read the author's manuscript as input, and produce a typeset object as output.%%
<ix><ALD48><PS14><FT0641><LS24>INTRODUCTION%c
<PS12><FT0640><LS13>This thesis is concerned with the mechanisms used to transformmanuscripts into print. Ever since the invention of printing from movable types more than five centuries ago it has been necessary to supply detailed information along with a manuscript indicating how it should appear in final form. A printer would use this information to determine how to assemble individual pieces of type so as to create the requisite printing surfaces. These surfaces would then be used to transfer a light-absorbing substance called ink to a light-reflecting surface called paper to create a printed artifact.%%
<PS10><LS11>%mAt first, information of this kind was in the form of instructions to be followed by human beings who were responsible for hand composing the assemblages of type. When the process of composition was automated beginning in the late 19th century, markup had to be translated into specific instructions which would cause a particular machine to set type accordingly. The expertise which had once been required of the hand compositor was now required of the person preparing input for the machine.%%

<PS10><LS11>%mAn author thinks of a manuscript as a logical construct. A book , for example, is best seen as a hierarchical collection of parts. The graph of figure 1.1 shows the structure of a book as an author might imagine it. A book consists of a set of elements called front matter followed by another set of elements called body, followed by a third set called rear matter. The front matter consists of a title page followed by a dedication page followed by a table of contents followed by one or more optional prefaces. Each of these elements can in turn be broken into yet smaller elements. A title page, for instance,

consists of a title followed by a subtitle followed by an author's name followed by a publisher's imprint.%%
<ALD24><PS14><LS18><AL>The Invention and Development of Typography%%
<AJ><PS10><LS11>Prior to the invention of printing from movable types, the composition of written pages images was accomplished either by manually applying ink to paper with a brush or pen, or by preparing a carved relief surface which was then used to transfer ink to paper. The former technique is called calligraphy, and the latter is called xylography. Calligraphy and xylography differ fundamentally in that xylography allows multiple copies of the same image to be reproduced whereas calligraphy does not.%%
%mIt is useful, however, to consider calligraphy and xylography as constituting a single paradigm. Both allow the freehand placement of light-absorbing material on a surfcace to form an image, and while it may be technically difficult to duplicate a calligraphic form by carving an image into a block of wood, it is conceptually possible to do so.%%
%mThe creation of textual images from movable types is called typography. Typography is not part of the same paradignm as calligraphy and xylography. It involves the assembly of images from discrete elements which can only fit together in certain ways....%%
%mJohannes Gutenberg, the inventor of typography, intended for the product of his printing press to closely resemble the style of writing called....%%
<ALD24><PS14><LS18><AL>Machine Composition%%
<AJ><PS10><LS11>The first successful attempt to mechanize the composition process was the work of Ottmar Merganthaler. His Machine came to be known as the linotype because it produced whole lines of letters cast on a single body called a slug. A keyboard operator would enter the a sequence of characters to be typeset. Each keystroke would release an appropriate brass matrix from a magazine containing matrices for all the characters in a particular font....%%
%mLess than a decade after the introduction of the linotype, a machine called the monotype was introduced. The machine used a punched paper tape to drive the casting unit. The system consisted of two machines....%%
<ALD24><ALD72><FT2660><PS14><LS24>BIBLIOGRAPHY%c
<FT1700><PS10><LS11><FT0640>McLean, Rauri, <FT0641>The Thames and Hudson Manual of Typography, <FT0640>Thames and Hudson Publishing Company, London, 1980%%

<FT0640>Moxon, Joseph, <FT0641>Mechanick Exercises on the Whole Art of Printing, <FT0640>London, 1958%%

<FT0640>Ruppell, Aloys, <FT0641>Johann Gutenberg, <FT0640>New York, 1947%%

<ALD72><FT2660><PS14><LS24>ACKNOWLEDGEMENTS%c

<AL><FT1700><PS12><LS13>I would like to acknowledge the help of the following people: Patricia A. Cost, Roger P. Cost, Guy Johnson, Jeff Lasky, Michael Kleper, James Hamilton, Emery Schneider and Darrin Ward.%%
Parse successful

TYPESET FILE 1

The Design and Implementation of a System for Processing Documents Described in Generalized Markup Languages

BY

FRANK J. COST

ABSTRACT

This thesis proposes an architecture for a system capable of transforming the work of an author into printed form, and builds a partial implementation based upon that architecture. The system allows the author to describe a document as a hierarchical structure without any concern for how it will be transformed into print. The author is provided with a special markup language which can be used to tag the various logical parts of a document, along with an idea of how the various tagged parts can fit together. The system is able to read the author's manuscript as input, and produce a typeset object as output.

INTRODUCTION

This thesis is concerned with the mechanisms used to transform-manuscripts into print. Ever since the invention of printing from movable types more than five centuries ago it has been necessary to supply detailed information along with a manuscript indicating how it should appear in final form. A printer would use this information to determine how to assemble individual pieces of type so as to create the requisite printing surfaces. These surfaces would then be used to transfer a light-absorbing substance called ink to a light-reflecting surface called paper tocreate a printed artifact.

At first, information of this kind was in the form of instructions to be followed by human beings who were responsible for hand composing the assemblages of type. When the process of composition was automated beginning in the late 19th century, markup had to be translated into specific instructions which would cause a particular machine to set type accordingly. The expertise which had once been required of the hand compositor was now required of the person preparing input for the machine.

An author thinks of a manuscript as a logical construct. A book , for example, is best seen as a hierarchical collection of parts. The graph of figure 1.1 shows the structure of a book as an author might imagine it. A book consists of a set of elements called front matter followed by another set of elements called body, followed by a third set called rear matter. The front matter consists of a title page followed by a dedication page followed by a table of contents followed by one or more optional prefaces. Each of these elements can in turn be broken into yet smaller elements. A title page, for instance, consists of a title followed by a subtitle followed by an author's name followed by a publisher's imprint.

The Invention and Development of Typography

Prior to the invention of printing from movable types, the composition of written pages images was accomplished either by manually applying ink to paper with a brush or pen, or by preparing a carved relief surface which was then used to transfer ink to paper. The former technique is called calligraphy, and the latter is called xylography. Calligraphy and xylography differ fundamentally in that xylography allows multiple copies of the same image to be reproduced whereas calligraphy does not.

It is useful, however, to consider calligraphy and xylography as constituting a single paradigm. Both allow the freehand placement of light-absorbing material on a surfcace to form an image, and while it may be technically difficult to duplicate a calligraphic form by carving an image into a block of wood, it is conceptually possible to do so.

The creation of textual images from movable types is called typography. Typography is not part of the same paradignm as calligraphy and xylography. It involves the assembly of images from discrete elements which can only fit together in certain ways. . .

Johannes Gutenberg, the inventor of typography, intended for the product of his printing press to closely resemble the style of writing called. . .

Machine Composition

The first successful attempt to mechanize the composition process was the work of Ottmar Merganthaler. His Machine came to be known as the linotype because it produced whole lines of letters cast on a single body called a slug. A keyboard operator would enter the a sequence of characters to be typeset. Each keystroke would release an appropriate brass matrix from a magazine containing matrices for all the characters in a particular font. . . .

Less than a decade after the introduction of the linotype, a machine called the monotype was introduced. The machine used a punched paper tape to drive the casting unit. The system consisted of two machines. . . .

BIBLIOGRAPHY

McLean, Rauri, *The Thames and Hudson Manual of Typography*, Thames and Hudson Publishing Company, London, 1980

Moxon, Joseph, *Mechanick Exercises on the Whole Art of Printing*, London, 1958

Ruppell, Aloys, *Johann Gutenberg*, New York, 1947

ACKNOWLEDGEMENTS

I WOULD LIKE TO ACKNOWLEDGE THE HELP OF THE FOLLOWING PEOPLE: PATRICIA A. COST, ROGER P. COST, GUY JOHNSON, JEFF LASKY, MICHAEL KLEPER, JAMES HAMILTON, EMERY SCHNEIDER AND DARRIN WARD.
PARSE SUCCESSFUL
ISIS!FJC1524«6»

MCS INPUT FILE 2

<LL32><FT4061><PS14><LS16>The Design and Implementation of a System for Processing Documents Described in Generalized Markup Languages%c
<ALD36><FT4010><PS12><LS24>by%c
<PS14><LS36>Frank J. Cost%c
<ALD42><FT4060><PS12><LS20>ABSTRACT%c
<FT4041><PS10><LS12><ib3><io>This thesis proposes an architecture for a system capable of transforming the work of an author into printed form, and builds a partial implementation based upon that architecture. The system allows the author to describe a document as a hierarchical structure without any concern for how it will be transformed into print. The author is provided with a special markup language which can be used to tag the various logical parts of a document, along with an idea of how the various tagged parts can fit together. The system is able to read the author's manuscript as input, and produce a typeset object as output.%%
<ix><ALD48><AL><PS14><FT4060><LS24>INTRODUCTION & BACKGROUND%%
<PS12><FT4040><LS14>This thesis is concerned with the mechanisms used to transformmanuscripts into print. Ever since the invention of printing from movable types more than five centuries ago it has been necessary to supply detailed information along with a manuscript indicating how it should appear in final form. A printer would use this information to determine how to assemble individual pieces of type so as to create the requisite printing surfaces. These surfaces would then be used to transfer a light-absorbing substance called ink to a light-reflecting surface called paper to create a printed artifact.%%
<PS10><LS12>%mAt first, information of this kind was in the form of instructions to be followed by human beings who were responsible for hand composing the assemblages of type. When the process of composition was automated beginning in the late 19th century, markup had to be translated into specific instructions which would cause a particular machine to set type accordingly. The expertise which had once been required of the hand compositor was now required of the person preparing input for the machine.%%

<PS10><LS12>%mAn author thinks of a manuscript as a logical construct. A book , for example, is best seen as a hierarchical collection of parts. The graph of figure 1.1 shows the structure of a book as an author might imagine it. A book consists of a set of elements called front matter followed by another set of elements called body, followed by a third set called rear matter. The front matter consists of a title page followed by a dedication page followed by a table of contents followed by one or more optional prefaces. Each of these elements can in turn be broken into yet smaller elements. A title page, for instance, consists of a title followed by a subtitle followed by an

author's name followed by a publisher's imprint.%%
<ALD24><FT4061><PS12><LS16>The Invention and Development of Typography%%
<FT4040><PS10><LS12>Prior to the invention of printing from movable types, the composition of written pages images was accomplished either by manually applying ink to paper with a brush or pen, or by preparing a carved relief surface which was then used to transfer ink to paper. The former technique is called calligraphy, and the latter is called xylography. Calligraphy and xylography differ fundamentally in that xylography allows multiple copies of the same image to be reproduced whereas calligraphy does not.%%
%mIt is useful, however, to consider calligraphy and xylography as constituting a single paradigm. Both allow the freehand placement of light-absorbing material on a surfcace to form an image, and while it may be technically difficult to duplicate a calligraphic form by carving an image into a block of wood, it is conceptually possible to do so.%%
%mThe creation of textual images from movable types is called typography. Typography is not part of the same paradignm as calligraphy and xylography. It involves the assembly of images from discrete elements which can only fit together in certain ways....%%
%mJohannes Gutenberg, the inventor of typography, intended for the product of his printing press to closely resemble the style of writing called....%%
<ALD24><FT4061><PS12><LS16>Machine Composition%%
<FT4040><PS10><LS12>The first successful attempt to mechanize the composition process was the work of Ottmar Merganthaler. His Machine came to be known as the linotype because it produced whole lines of letters cast on a single body called a slug. A keyboard operator would enter the a sequence of characters to be typeset. Each keystroke would release an appropriate brass matrix from a magazine containing matrices for all the characters in a particular font....%%
%mLess than a decade after the introduction of the linotype, a machine called the monotype was introduced. The machine used a punched paper tape to drive the casting unit. The system consisted of two machines....%%
<ALD24><ALD72><FT4060><PS14><LS24>BIBLIOGRAPHY%%
<FT4040><PS10><LS11><FT4040>McLean, Rauri, <FT4041>The Thames and Hudson Manual of Typography, <FT4040>Thames and Hudson Publishing Company, London, 1980%%

<FT4040>Moxon, Joseph, <FT4041>Mechanick Exercises on the Whole Art of Printing, <FT4040>London, 1958%%

<FT4040>Ruppell, Aloys, <FT4041>Johann Gutenberg, <FT4040>New York, 1947%%

<ALD72><FT4060><PS14><LS24>ACKNOWLEDGEMENTS%%
<AL><FT4041><PS12><LS14>I would like to acknowledge the help of

the following people: Patricia A. Cost, Roger P. Cost, Guy Johnson, Jeff Lasky, Michael Kleper, James Hamilton, Emery Schneider and Darrin Ward.%%
Parse successful

TYPESET FILE 2

The Design and Implementation of a System for Processing Documents Described in Generalized Markup Languages

by

Frank J. Cost

ABSTRACT

This thesis proposes an architecture for a system capable of transforming the work of an author into printed form, and builds a partial implementation based upon that architecture. The system allows the author to describe a document as a hierarchical structure without any concern for how it will be transformed into print. The author is provided with a special markup language which can be used to tag the various logical parts of a document, along with an idea of how the various tagged parts can fit together. The system is able to read the author's manuscript as input, and produce a typeset object as output.

INTRODUCTION & BACKGROUND

This thesis is concerned with the mechanisms used to transform-manuscripts into print. Ever since the invention of printing from movable types more than five centuries ago it has been necessary to supply detailed information along with a manuscript indicating how it should appear in final form. A printer would use this information to determine how to assemble individual pieces of type so as to create the requisite printing surfaces. These surfaces would then be used to transfer a light-absorbing substance called ink to a light-reflecting surface called paper to create a printed artifact.

At first, information of this kind was in the form of instructions to be followed by human beings who were responsible for hand composing the assemblages of type. When the process of composition was automated beginning in the late 19th century, markup had to be translated into specific instructions which would cause a particular machine to set type accordingly. The expertise which had once been required of the hand compositor was now required of the person preparing input for the machine.

An author thinks of a manuscript as a logical construct. A book , for example, is best seen as a hierarchical collection of parts. The graph of figure 1.1 shows the structure of a book as an author might imagine it. A book consists of a set of elements called front matter followed by another set of elements called body, followed by a third set called rear matter. The front matter consists of a title page followed by a dedication page followed by a table of contents followed by one or more optional prefaces. Each of these elements can in turn be broken into yet smaller elements. A title page, for instance, consists of a title followed by a subtitle followed by an author's name followed by a publisher's imprint.

The Invention and Development of Typography

Prior to the invention of printing from movable types, the composition of written pages images was accomplished either by manually applying ink to paper with a brush or pen, or by preparing a carved relief surface which was then used to transfer ink to paper. The former technique is called calligraphy, and the latter is called xylography. Calligraphy and xylography differ fundamentally in that xylography allows multiple copies of the same image to be reproduced whereas calligraphy does not.

It is useful, however, to consider calligraphy and xylography as constituting a single paradigm. Both allow the freehand placement of light-absorbing material on a surfcace to form an image, and while it may be technically difficult to duplicate a calligraphic form by carving an image into a block of wood, it is conceptually possible to do so.

The creation of textual images from movable types is called typography. Typography is not part of the same paradignm as calligraphy and xylography. It involves the assembly of images from discrete elements which can only fit together in certain ways....

Johannes Gutenberg, the inventor of typography, intended for the product of his printing press to closely resemble the style of writing called....

Machine Composition

The first successful attempt to mechanize the composition process was the work of Ottmar Merganthaler. His Machine came to be known as the linotype because it produced whole lines of letters cast on a single body called a slug. A keyboard operator would enter the a sequence of characters to be typeset. Each keystroke would release an appropriate brass matrix from a magazine containing matrices for all the characters in a particular font. ..

Less than a decade after the introduction of the linotype, a machine called the monotype was introduced. The machine used a punched paper tape to drive the casting unit. The system consisted of two machines....

BIBLIOGRAPHY

McLean, Rauri, *The Thames and Hudson Manual of Typography,* Thames and Hudson Publishing Company, London, 1980

Moxon, Joseph, *Mechanick Exercises on the Whole Art of Printing,* London, 1958

Ruppell, Aloys, *Johann Gutenberg,* New York, 1947

ACKNOWLEDGEMENTS

I would like to acknowledge the help of the following people: Patricia A. Cost, Roger P. Cost, Guy Johnson, Jeff Lasky, Michael Kleper, James Hamilton, Emery Schneider and Darrin Ward.
Parse successful
isis!fjc1524[5]