MEMORANDUM
RM-4920-PR
AUGUST 1966

AD637303

COMPUTER ROUTINES TO READ NATURAL TEXT WITH COMPLEX FORMATS

Patricia A. Graves, David G. Hays,
Martin Kay and Theodore W. Ziehe

This research is sponsored by the United States Air Force under Project RAND—Contract No. AF 49(638)-1700—monitored by the Directorate of Operational Requirements and Development Plans, Deputy Chief of Staff, Research and Development, Hq USAF. Views or conclusions contained in this Memorandum should not be interpreted as representing the official opinion or policy of the United States Air Force.

The RAND Corporation
1700 MAIN ST • SANTA MONICA • CALIFORNIA • 90406

20050218147

PREFACE

Use of the computer as a tool for managing files of text as a part of information systems for management or science - or as part of linguistic research - is more convenient if a standard format is maintained internally. But for the convenience of input typists, the designer must be free to set up keyboard arrangements and special conventions according to his requirements for each job. The present Memorandum describes programs facilitating the connection of an arbitrary input scheme with either a standard internal storage scheme or any arbitrary output scheme.

The present Memorandum is intended for use by system designers. The authors hope that it will be useful to anyone with ordinary text to process and a computer to do the processing.

SUMMARY

This Memorandum describes a set of subroutines for the IBM 7040/44 computers for reading textual material with complex formats and coding conventions--questionnaires, library catalog cards, etc.--from any external medium into the high-speed store of the machine. Different kinds of information in the input are recognized by explicit markers, position on the line or page, or syntactic clues given by other items. Less complex material requires only a portion of the system. Information may be recoded according to the user's conventions before being delivered to his program. The routines may be called from either FORTRAN- or MAP-coded programs.

FOREWORD

The variety of tasks to which digital computers are applied is rapidly increasing, and more and more of the computer's time is being devoted to non-numerical work, much of which involves processing ordinary-language text in one form or another. Research in linguistics, machine translation, and documentation can only proceed if large files of text are available. The Linguistics Project at RAND, in collaboration with the Centre d'Etudes pour la Traduction Automatique of the University of Grenoble, has, over the past two years, developed a method of encoding linguistic data on magnetic tape so as to capture all the information from the most complex printed page.* This format greatly facilitates the exchange of information.

The programs described in this Memorandum are intended for reading texts in other formats so that they can then be processed by standard programs or re-encoded according to the standard conventions. It is expected that the text in its original form will often be most unsuitable for computer processing, having been produced with some quite different purpose--automatic typesetting, library catalog

*For a detailed description of the application of this format to text, see Martin Kay and Theodore Ziehe, Natural Language in Computer Form, The RAND Corporation, RM-4390-PR, February 1965.

card preparation, etc.--in mind, and in these cases a great deal of preliminary work will be required before the material can be used. For this reason the programs described here will sometimes seem cumbersome and unnecessarily complicated. However, every attempt has been made to divide the work up among several autonomous routines, the whole set of which will only rarely have to be used together. If the format is simple and the keys explicit, then the largest and most sophisticated routines can be left out. Conversely, if the format is complex and the distinctions subtle, then the whole system will probably be required, and the labor involved in specifying the parameters for each process will be correspondingly greater.

This Memorandum is divided into two parts, the first of which is an introduction to the system as a whole and to the individual programs. Since the routines are intended to carry out sophisticated processes on almost any kind of textual material, the directions required from the user are very comprehensive and detailed. However, the amount of programming experience required to use the routines is actually very small.

The second part (Appendix) is a detailed programmer's guide, giving all the information necessary to use the routines. The Appendix will only be intelligible to a reader with some programming experience, but this need not be very extensive. These programs and any others developed by the Linguistics Project at RAND are available to others on request.

CONTENTS

COMPUTER ROUTINES TO READ NATURAL TEXT WITH COMPLEX FORMATS

1. INTRODUCTION

In a typical computer job, a program is written first and then data are prepared for it in a form the machine can read. The details of the data format--how the numbers shall be represented, where they will be placed on the card or line, how program options are chosen--are dictated by the program. It is right that they should be because data are most often prepared for a single program, and one format is usually as convenient as another for the person preparing the data, though not necessarily for the programmer. Systems programs provide a minimum of assistance in the form of routines for converting numbers found at a certain place on a line to binary form.

1.1. Shortcomings

However, under some conditions, this way of doing things is intolerable either because the data format exists before the program or because the amount and complexity of the data are such that the burden on the persons preparing the data far outweighs that on the programmer. Linguistic data are a case in point. Most of the raw material used in linguistic, literary, and bibliographic computing consists of text. The amount of material to be processed is often large, and the tedium of keypunching or typing ordinary texts so as to preserve all the typographical distinctions of the original is often very great. This leads to compromises and short cuts that often entail more labor in the long run, for a single text could often be profitably used in other processes than that for which it was first prepared if sufficient information from the original had been preserved. Furthermore, text is now becoming widely available in a form acceptable to a computer from sources outside research and computing, notably as a by-product of the printing process itself. This will save the scholar tedious work if he can convert the by-products into usable form.

It is therefore wrong that linguistic computing should continue in the traditional pattern with a new data format for each new job. There is a pressing need for a standard computer format for textual material and for flexible methods of getting material from widely differing sources into this format. The most profitable direction for linguistics computation to pursue is a method that goes beyond the well-known "format statements" of FORTRAN and the like. The programs that we describe here, used in various combinations, accept data in a wide variety of formats, on diverse media, and leave the data in core storage ready for whatever processing the user requires.

1.2. Desiderata

The person who prepares text for a computer is the machine's eye. His task is to record in a series of codes on a punched card, a role of paper tape, or in whatever medium, everything of significance on the page. This task includes not only encoding sequences of letters and marking differences of type font and style, but also noting the position of each mark on the page. Positions are critical for the proper interpretation of diacritics and underlines and for the identification of titles, subtitles and footnotes. Page positions are equally important for tabular material such as questionnaires, directories and catalogs.

The best way to collect samples of text is to eavesdrop on the process that actually produces the page of print. For example, it is easy to salvage the punched paper tapes used by the printer to drive his typesetting machines. Ideally, nothing should be printed on the page that was not expressly directed to its place by instructions on this tape. In real life, printers often correct the type after it has been set and adjust the machines manually, leaving no mark on the tape. The method therefore falls well short of perfection, though it is certainly the most economical. If we cannot eavesdrop, we must produce a replica of the original and monitor our own process. We might, for example, replicate with a typewriter and monitor with a tape punch.

The programs that interpret the final stream of codes reconstruct from them everything of significance about the page of print they represent. This is not necessarily straight forward. There are often many ways in which the operator of the original machine could have gone about producing a given result on the hard copy, and we dare not suppose that one way is consistently preferred to the rest. By looking at a typescript we cannot tell whether the shift key was pressed immediately after the period at the end of a sentence, between the two spaces separating sentences, or immediately before the capital beginning the second sentence. We cannot tell if the shift key was pressed and instantly released at any place in the document. These are differences that do not change the appearance of the final result; they are therefore matters of no concern to the person at the key-board. However, if there is a paper-tape punch connected to the typewriter, the codes it punches are different in each case. Consider the case of underlining. A typist may underline words one by one, backspacing from the end of the word each time, or she may backspace after every letter, or she may underline only when she has finished everything else on the line, or she may have no consistent policy. So, the underline that goes with a letter can occur before or after it in the stream of codes and can be separated from it by a large or small number of other codes that affect quite different places on the line.

Our programs therefore begin by reconstructing, in coded form, the sequence of characters that were produced on the hard copy. Since we wish to accept input from a number of different machines (keypunches, typewriters, etc.), our programs are capable of disentangling information about the appearance of the page from information about incidental properties of the initial machine. A given machine may return the carriage and advance the page with two control signals or with one; it may or may not have a tabulator control, and so on. The programs use a formalized description of a particular machine as a guide to the interpretation of the input codes; the sequence of characters constructed by our programs to represent a page will be the same no matter what initial machine prepared the input.

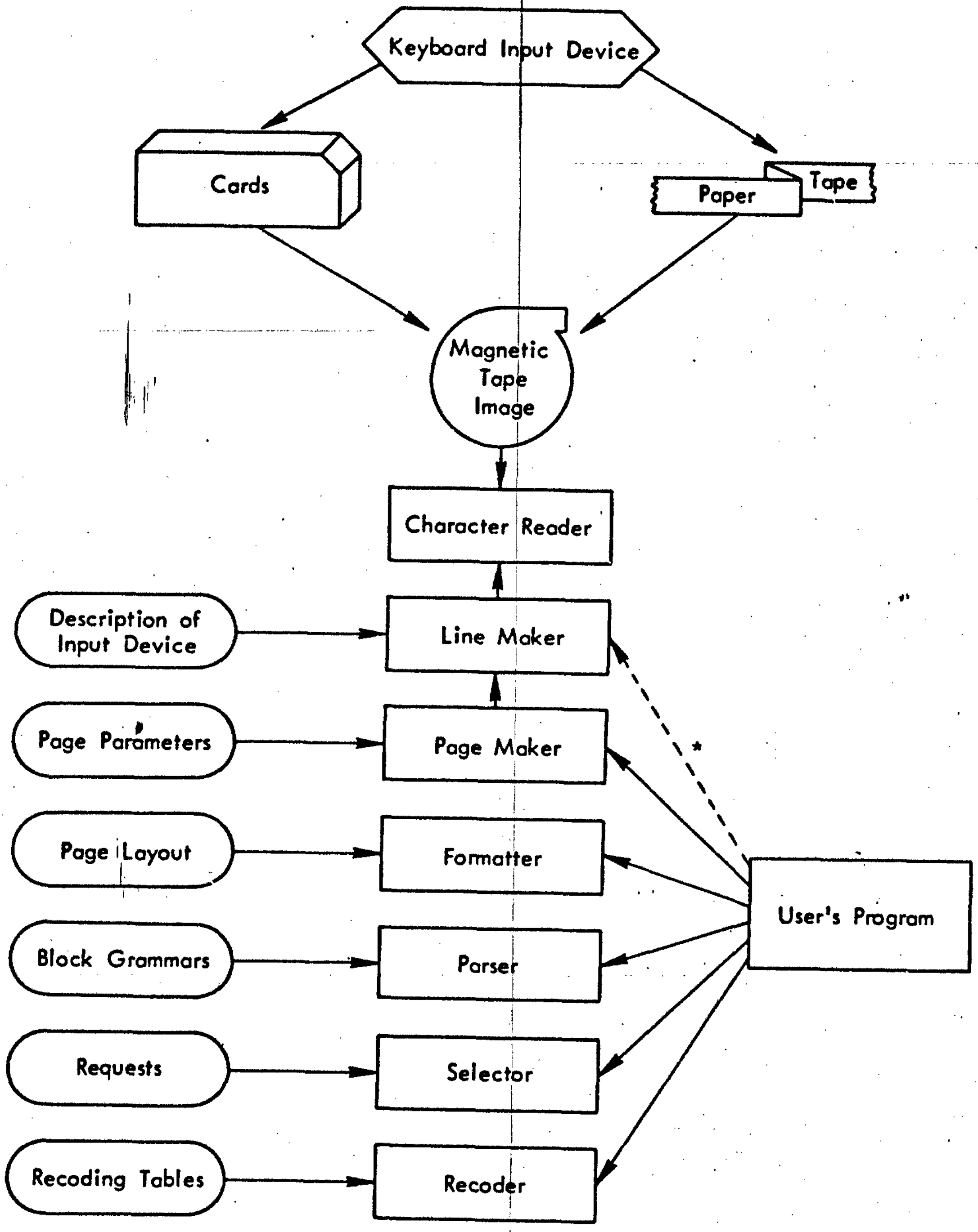
A strategy of much or little generality might be proposed for describing a text-encoding machine to a computer and having the computer use it to decode text. The programs reported in the present Memorandum are capable of development in various directions, but to understand them in their present form the reader needs to be aware of some important limitations that we have accepted.

1. Each character is taken to occupy the same amount of space on the line--there is no allowance for proportional spacing.

2. The system always assumes as much space to be filled on a line as was in fact filled. This means that there can be no means of deleting a letter--for example, by overstriking with "X"--that will cause the character to disapppear from the record. On the other hand, combinations produced by overstriking are permitted.
3. The basic unit of information is the printed line. Once a "line feed" code has been issued, the current line is accepted as it stands, and work begins on the one below. Within a given line, attention may be directed to different character positions by the normal advance that accompanies a character, and by tabs, backspaces and carriage returns. If future machines provide for rolling the platen down as well as up by pressing keys, they will require a modification to our system.

1.3. Overall Design

The overall design of the system is shown in Fig. 1. The magnetic tape image is disentangled by the character reader (characters may be packed together in unhandy, if not irrational, ways on the tape); characters pass one by one to the line maker. Operating in accordance with a formalized description of the input machine, which the user supplies in the form of a set of tables, the line maker assembles a single line of text from the input stream. This routine also



* The user's program refers to the line maker directly only when the page maker is not being used

Fig. 1—The overall design of the system

converts characters from the input code to a code established by the person supplying the tables. The line maker is described in Sec. 2.

The line maker serves as a subroutine to a second routine called a <u>page maker</u> (Sec. 3) whose function is simply to collect batches of lines that will go through subsequent steps together. The logical status of a line in a text sometimes has to be determined by reference to other lines. In a certain context, a line immediately preceded or followed by a line of blanks must perhaps be treated in a special way, or, if the information on the page is arranged in columns, information of a certain kind is contributed by small parts of different lines. The context for operations of this kind must all lie within what the system knows as a page. The system recognizes a page by the line or sequence of lines that ends it or begins the following one; the user must specify how these lines are to be recognized.

A routine called a <u>formatter</u> (Sec. 4) takes each page and does little or much, depending on the complexity of the page layout. Its job is to identify areas on the page that contain different kinds of information. An ordinary page of prose requires almost no work of the formatter. Text arranged in columns is easily handled provided it is known how wide each column is. In more complicated examples, a single column may contain several kinds of material, each of which is either known to occupy a set number of lines, or is distinguishable from its surroundings by more subtle means. Part of a page or a column may be solid, whereas another part may be divided into columns or subcolumns. The formatter divides the page into rectangular blocks by a sequence of horizontal or vertical slices in accordance with instructions supplied by the user of the program.

By this stage, the program is supposed to know as much as can be gleaned from the page layout. However, some blocks may still contain logically heterogeneous information. Suppose a particular block contains a bibliographic reference; we know from the formatter which block it is. If gross features of position on the page do not suffice to distinguish author, title, publisher, etc., more subtle means must be adopted. The system allows the inclusion of a <u>parser</u> capable of applying to the contents of a block grammatical rules tailored to the kind of information in that block. These rules may result in a finer subdivision of the block. A parser is only applied to blocks that need it, and for some purposes no parser is needed. A relatively simple parser is described in Sec. 5.

The blocks delineated by the formatter, and the <u>segments</u> within blocks identified by the parser, can be collected into new combinations in preparation for output. When necessary, the <u>selector</u> described in Sec. 6 is used for this purpose. The internal code produced by the line maker is translated by the <u>recoder</u> (Sec. 7) into the final form required by the user.

A page is stored in the computer as a rectangular array in which each cell represents a character position. The code for a character occupies 24 of the 36 bits in the cell. This may seem prodigal, but it has the great advantage of allowing properties of characters to be represented by individual bits. Thus, one bit can be used to distinguish digits and another punctuation marks. One can be used for italics, one for upper case, one for underlining, and so on. This possibility greatly simplifies the remainder of the system.

A cell containing only zeros represents a blank; other codes are specified by tables supplied by the user. A standard convention is adopted for the blank because the computer must know what code to store for those parts of the page that the original input device never visits and because of the strategy adopted for handling overstruck characters. The characters remain in this array throughout the process to be described. The result of subdividing the page is shown by means of coordinates rather than by moving material to new storage locations. This means that it is possible, in principle, for the blocks into which the page is cut to overlap, though it is not clear that any good can come of this.

What is finally passed to the program using this input system is a page, the result of dividing a page into blocks and segments, or the output of a parser.

It is therefore of some interest to establish a standard form in which to deliver a unit of text. Economy clearly requires that an indication of the part of the page occupied by a unit be given rather than that the characters themselves be moved. If only blocks were involved, it would be sufficient to deliver coordinates--first and last row and column. However a parser may find segments of somewhat odder shapes. Nevertheless, four parameters are probably enough, namely first and last column and first and last character in the region so defined (see Fig. 2). For this purpose, of course, all columns are regarded as running from top to bottom of the page.

Suppose a page accommodates 2000 characters arranged in 50 lines of 40 (see Fig. 2). The formatter has recognized five blocks, A, B, C, D, and E. Block A consists of the top 15 lines of the page and block E of the bottom 24. Lines 16 through 26 are occupied by B, C, and D arranged side by side with widths of 20, 12 and 8 character positions each. Block C can be specified by the numbers 21 and 32 (the character positions that bound it) and the numbers 181 and 312 (the serial numbers of the first and last characters falling within these limits and contained in C). Similarly, the block A can be specified by the quadruple 1, 40, 1, 600. Now, suppose that a parser has been set to work on block C and that it has identified, among other things, the sequence "John Doe, Inc. (New York.)"

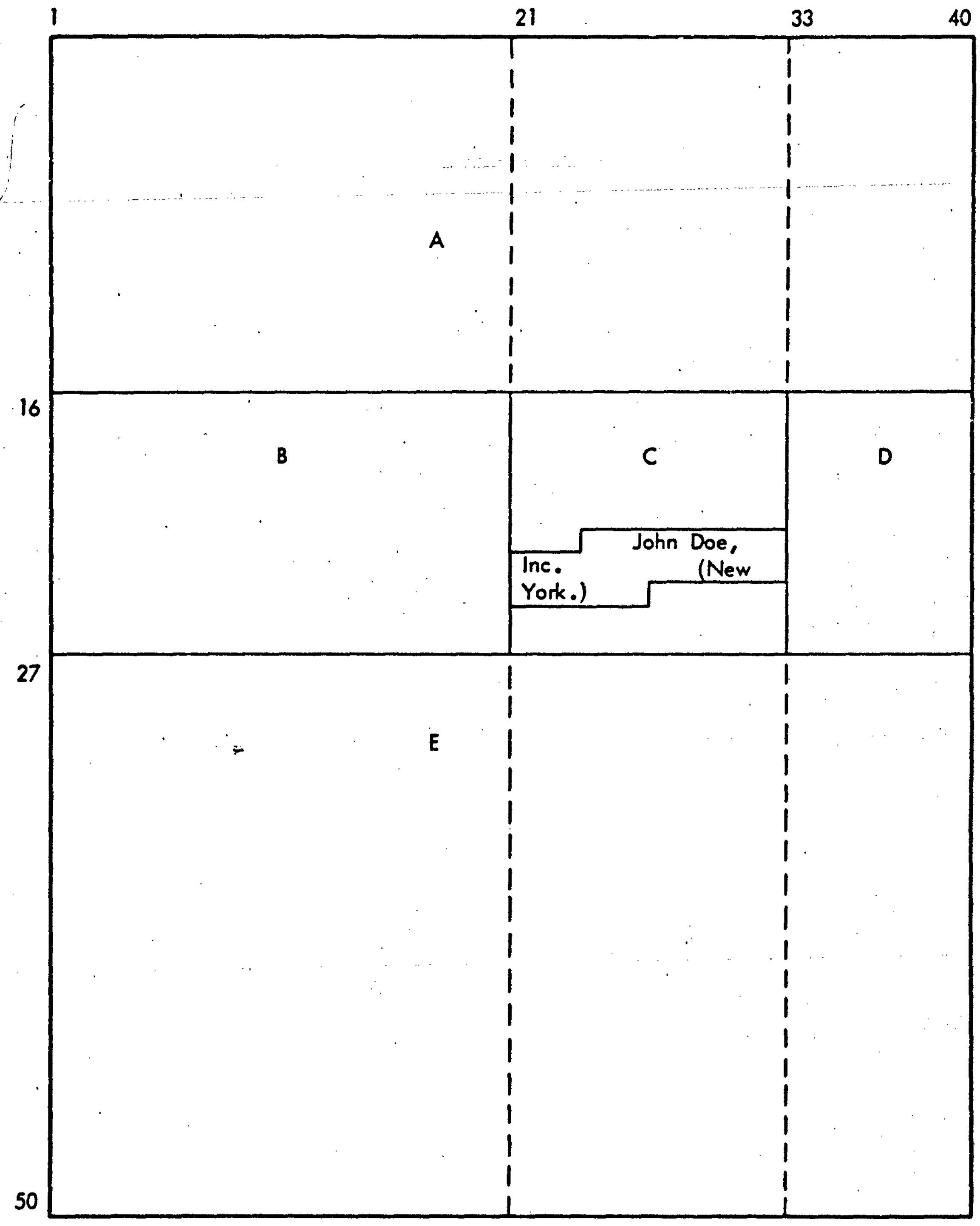


Fig.2—Sample page

beginning with the 3rd character of the 7th line of the block and ending with the 6th character of the 9th line. These are, infact, the 255th and the 282nd characters in the region bounded by the 21st and 32nd columns. The sequence can therefore be uniquely identified by the quadruple 21, 32, 255, 282.

2. THE LINE MAKER

A keyboard input device (keypunch, paper-tape typewriter, etc.) may or may not have shift keys, but whether it does or does not, the user may wish to use certain keys as shifts. He may, for example, wish to bound by asterisks sequences that are to be taken as being in bold face. If the machine he is using already has a built-in upper- and lower-case shift key, this means that he has effectively four shifts. One of his shift keys, the built-in one, does not cause the carriage to move, whereas the other, the asterisk, causes it to move forward one position. The effect of a shift key, whether built-in or conventional, is to change the significance of other keys. After one shift, a given key produces an "a", after another "A", and so on. On a machine with two or more shift keys, the effect of one shift key, or even whether a given key functions as a shift, could in principle be determined by the shifts already made. Thus, if only lower-case boldface type were required, the asterisk could be used to indicate this; in the upper-case shift, the same key might produce say, a dash. In general, then, the description of the device furnished to the line maker consists of a number of major parts, one for each of the shifts that the devic or the user's conventions allow.

For each shift on the input device, the description vides a table with one entry for each permissible input code. As each code is received, it is referred to the tal for the current shift, and four kinds of information are obtained:

(i) A binary number representing a graphic mark. If the incoming code represents a space, backsp tab, shift, or move of the device that does not produce an actual character, then this binary nu is zero. How binary numbers are assigned to cha acters is important, and we return to it shortly

(ii) An indication of any movement of the carriage implied by the code. There are five possibiliti

a. No movement ("dead key")
b. Space forward
c. Backspace
d. Tabulate
e. Return carriage

(iii) An indication of whether the current code makes the input device advance to a new line. A code that does this can have other meanings; it can also produce a carriage return, a shift, or even a graphic mark.

(iv) An indication of any change in shift entailed by the current code.

The description of the input device has one further part, namely a list of tab stops. This list is separate from the remainder of the description since it can be expected to change more often. It has as many entries as there are character positions on the line. The entry for a given character position gives the number of the character position where the next tab stop is set. Thus, if the device is at character position p and the code representing a tab is received, the number in entry p on the list replaces p as current position.

To simplify the detailed description of the line maker we assume that the main part of a device's description (all except the tab list) is stored in the form of four rectangular arrays. In each array, the rows represent codes issued by the device and the columns shifts. The notation "Char[i,j]" will then mean the entry in the "Char" array for character j and shift i. The four arrays are as follows:

(i) "Char" contains the binary numbers representing graphic marks.

(ii) "Move" contains the indications of carriage movement--one of five possibilities for each entry.

(iii) "Feed" may be regarded as a Boolean array in which the entry in a given cell is set to true if the corresponding shift and character represent a line-feed (the last code of a line) and otherwise to false.

(iv) "Shift" contains, for a given current shift and code, what shift the machine now moves into. If the current code is not a shift character, then the entry contains the current shift.

Figure 3 is a flow chart of the line maker. There p is an integer variable giving the current position of the carriage. When the routine is first entered, p is undefined, for there is usually no way of knowing where the carriage was efore the first code was issued. For this reason, it is standard practice to arrange for the first code to be that for a carriage return. The value of s, an integer variable, represents the current shift. Like p, it is apparently undefined on first entry to the routine. However, the user must in fact set s to some reasonable initial value since it is otherwise impossible for the routine to interpret any codes at all.

The line maker prepares one line each time it is entered. Accordingly, it begins (Box 1) by filling the array representing the line with zeros--the agreed representation for blanks. Then a character is read (Box 2). At the beginning of work on a new line, the current position on the line (p) is not necessarily 1 since there may

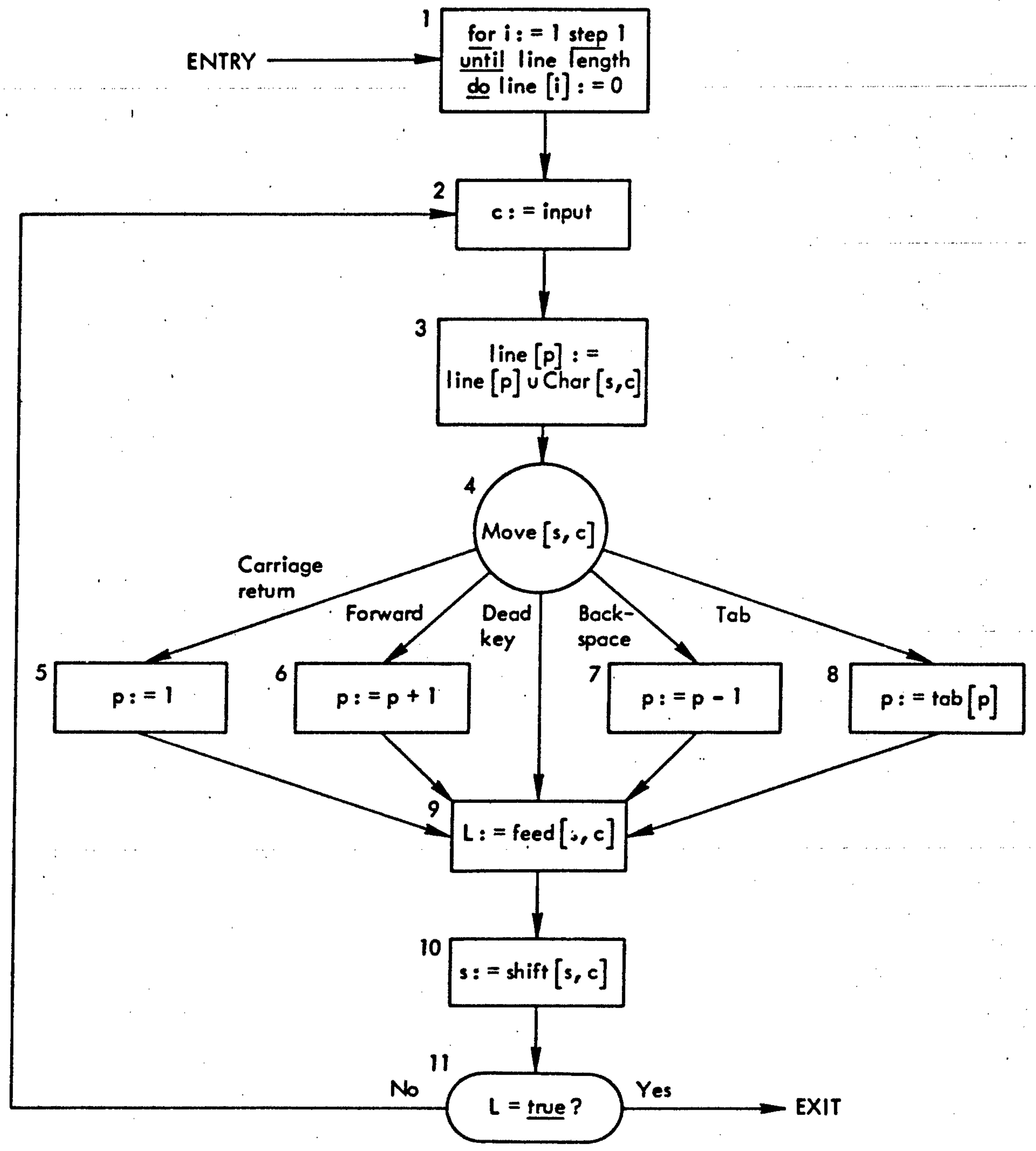


Fig.3—Flow chart of the line maker

not have been a carriage return at the end of the preceding line. In position p in the line array is now placed the logical sum of the binary number of the new character specified by the current code and shift and the previous occupant of that position (Box 3).

We can now see why the choice of binary numbers to represent characters is important. When a character position is first encountered, the operation of taking the logical sum has no effect other than to enter the number for the new character in that position. If no graphic mark is associated with the code, the binary number assigned to that code is zero, and the contents of the current character position remain unchanged. Now, when a character is underlined, the number that ultimately occupies that position in the line array must be derivable by merely combining that character with the underline. One way to do this is to assign a number consisting of zeros except for one bit to the underline and to insure that this bit is zero for all other characters. Similar considerations apply to accent marks. If there is an accent that may combine with any vowel, then the result of taking the logical sum of the corresponding binary numbers must always be a code that can be arrived at in no other way. However, the combination of an accent with a consonant or other character can be allowed to result in a number that represents some other character or no character at all since this would in any case be a mistake. Notice that, with suitably chosen numbers, such overstruck characters as "¢", "ẋ", and "ᵬ" are automatically provided for.

The next thing the line maker does (Box 4) is to take note of the carriage movement specified by the current code and shift. The forward movement accompanying a normal character or a space causes p to be incremented by one (Box 6); a backspace decrements p by 1 (Box 7). "Carriage return" resets p to 1 (Box 5) and "tab" gives p a new value obtained from the table of tab stops in the manner already described (Box 8).

A variable L is now set to <u>true</u> if the current code and shift indicate a line end and otherwise to <u>false</u> (Box 9). This information is obtained from the "Feed" array. Next, the number corresponding to the new shift, or to the current shift if this is unchanged, is obtained from the "Shift" array and made the new value of s (Box 10). Finally, the variable L is examined to see if the line is complete (Box 11). If not, we return to the point where a new code is read (Box 2) and continue processing characters in this manner until a line-feed is encountered.

It is an important feature of this scheme that shift, carriage movement, line-feed, and even graphic marks are regarded as independent entities, any combination of which may be represented by a single code. If the line-feed key on a machine is separate, it has a vacuous mark associated with it, but it takes no more than a simple change in a table entry to make this mark non-vacuous. A machine combining the functions of carriage return and line-feed can likewise be accommodated by appropriate table entries.

3. THE PAGE MAKER

The first major component of the input processing system is a program called the page maker. It reads a sequence of lines, using the line maker subroutine, and identifies a group of one or more lines as marking the bounda between two consecutive pages. When a boundary is recognized, the page maker returns contr:l to the user's program.

To use the pagemaker, one must provide all of the input information needed by the line maker--a description of the input machine and of the conventions followed in typing text. The user also supplies a description of the kind of line or sequence of lines that mark the end of each page. Since pages can end in many ways, the page maker allows the user to specify boundaries in the same fairly rich language that is used in the formatter (Sec. 4). A line can be described as consisting of all blanks, all underscores, blanks except for a designated sequence of characters in fixed or variable position on the line, and so on. A page boundary can consist of a single line of a designated kind, or of several kinds in a fixed sequence. A boundary can be defined as a line of any of several different kinds, or different boundaries can be made up of lines of different kinds in different orders. The user pays a small penalty for this freedom; even if his needs are simple, he cannot avoid a certain amount of complexity.

To obtain a page, then, the user allocates a block of storage space to it, sets aside four cells for parameters that will identify its first and last column numbers and its first and last character numbers, sets a maximum depth for the page, and reserves one cell for reports of any oddities that the page maker finds in doing its work. Since one cell will be used for each character on the page, the amount of space to be reserved for storage of the page itself is the product of maximum width by maximum depth. The user must state maximum width, since the first several lines on the page may not be filled completely; he designates maximum depth in order to avoid the pitfall that would otherwise be created by failure to insert a page boundary in the input.

Of the four parameters that identify boundaries of the page, only one is in fact significant. Every page begins in column 1 and extends through its maximum column. It begins with character number 1, but its last character has to be determined by inspection of the input stream. The four parameters are required for consistency with what follows (as explained in the description of the formatter, Sec. 4).

The description of the page boundary supplied by the user is a short program that examines every line when it is delivered by the line maker. If the line qualifies as a possible last line of a page, the description program goes on to test the preceding line and, if that line qualifies as a possible next to last line, the one before that; boundaries are always tested from the bottom up. The group of lines comprising a boundary can be treated either as the end of a page or the beginning of one; if they end a page, the next line of input will not be constructed by the line maker until the user calls the page maker again; if they mark the beginning of a page, they will be stored until the next page maker call, then moved to the head of the page.

Let us suppose, for example, that each page ends with a blank line below the text matter, followed by a page number that appears on the left when even, on the right when odd, followed by exactly two blank lines. The program tests whether the last two lines delivered by line maker are blank. If they are, the next preceding line is tested against each of two possibilities: a number followed by blanks across the line, or blanks across the line ending with a number. If either of those conditions is satisfied, the next preceding line is tested against a description of an all-blank line, and if it passes the test, the end of the page is recognized. The boundary lines--four lines, blank everywhere except for a page number at the right or left on the second line--are delivered by page maker together with what preceded them. The next line is produced by the line maker whenever the page maker is called.

But suppose that the user's input stream consists of short statements, say two or three paragraphs, each beginning with a caption of some kind. Specifically, suppose that each page begins with two blank lines, and that on the first nonblank line, after a paragraph indentation, there is an underscored word or phrase followed by a period, two spaces, and solid text. Such a boundary cannot be taken as the end of a page, since the caption must stay with what follows. The page maker examines each line to see whether it begins with a caption, then checks for two preceding blank lines. The three-line group is preserved in memory; the page delivered does not include the newly found boundary, but when the page maker is called next it moves those three lines to the top of the page storage area before reading another line from the input stream and proceeding. To avoid error, the user must always remember that the page storage area just below the end of the page he is using can contain the beginning of the page that he will see later. He must not tamper with that area between successive calls of the page maker when his page boundaries are beginnings rather than endings.

A line description is a statement of the characters it can contain, and the order in which they can occur. The lines that participate in boundary markers commonly include long strings of identical characters: blanks, asterisks, and so on. In our program, a line of blanks is described this way:

```
0. Start with the first position on the line.
1. Does the position contain a blank?
   If yes: Move to the next position and
           redo step 1.
   If no:  Hold the same position and per-
           form step 2.
2. Does the position contain a line-feed?
   If yes:  The line is blank.
   If no:   The line is not blank.
```

If a line contains blanks everywhere except for a line-feed in the last position, this description causes the computer to apply step 1 at each position, finally moving to step 2 and recognizing the line. Putting a line-feed in the first position reduces the labor to be performed, since then step 2 is performed after a single execution of step 1.

Step 0 is implicit in every line description; the first step is always applied initially to the first position on the line.

An abbreviated notation is used in actual descriptions. Steps 1 and 2 each consist of three parts. The first names a character; the second and third tell what to do if the character is or is not found. The following form is equivalent to the foregoing, and closer to the real form:

1. (Blank)(Next, 1)(Same, 2)
2. (Line-feed)(True)(False)

Here "same" and "next" refer to positions on the line; "1" and "2" refer to locations in the description; and what is true or false is the statement "the line is blank."

If a page number consists of either one or two digits, we can test for a number flush left.

1. (Digit)(Next, 2)(False)
2. (Digit)(Next, 3)(Same, 3)
3. (Blank)(Next, 3)(Same, 4)
4. (Line-feed)(True)(False)

Of course, "Digit" is not a character; rather, it is a property of several characters. We allow 24 bits per position just for the sake of encoding different properties with individual bits. Here in steps 1 and 2, the program asks whether a certain bit is equal to unity; the input encoder must make that bit zero for all characters except the digits.

A line with page number flush right* is recognized by a different program:

*Flush right in this example means not left adjusted.

```
1.  (Blank)(Next, 1)(False)
2.  (Blank)(Next, 1)(Same, 2)
3.  (Digit)(Next, 3)(False)
4.  (Digit)(Next, 4)(Same, 4)
5.  (Line-feed)(True)(False)
```

The two programs can be combined to accept a line with page numbers at either margin:

```
1.  (Digit)(Next, 2)(Same, 4)
2.  (Digit)(Next, 3)(Same, 3)
3.  (Blank)(Next, 3)(Same, 7)
4.  (Blank)(Next, 4)(Same, 5)
5.  (Digit)(Next, 6)(False)
6.  (Digit)(Next, 7)(Same, 7)
7.  (Line-feed)(True)(False)
```

To recognize a centered page number reliably, we should have to specify that it follow some exact number of blanks, say 40, and this would entail writing 40 essentially identical instructions. But this can be avoided by means of a simple shorthand device allowing us to give as part of an instruction a count of the number of characters to which it must be applied. Thus, for the centered page number, we could write:

```
1.  40(Blank)(Next, 2)(False)
2.  (Digit)(Next, 3)(False
3.  (Digit)(Next, 4)(Same, 4)
4.  (Line-feed)(True)(False)
```

A caption line, as described above, can be recognized by the following program if underscore is a property and an indentation is three spaces:

```
1.  (Blank)(Next, 2)(False)
2.  (Blank)(Next, 3)(False)
3.  (Blank)(Next, 4)(False)
4.  (Underscore)(Next, 5)(False)
5.  (Underscore)(Next, 5)(Same, 6)
6.  (Blank)(Next, 5)(Same, 7)
7.  (Period)(Next, 8)(False)
8.  (Blank)(True)(False)
```

This version of the description has several attributes that may or may not be appropriate depending on the input, but are instructive. Steps 1-3 require exactly three blanks at the left margin. Steps 4 and 5 guarantee at least one underscored character, but not more, although they accept more. Step 6 allows blanks within the otherwise underscored string, permitting multiword captions. A period must follow the caption, and a line-feed is not allowable until after at least one blank, but no further text on the same line is guaranteed.

Each program describing a type or class of types of line is named. Another program uses these names to describe a boundary marker. In this context, let "blank" mean the program defining a blank line. A page end can now be described in this way:

1. Blank, No page
2. Blank, No page
3. Page number, No page
4. Blank, No page
5. End of page

Suppose that the line maker has just delivered the 47th line of a page; the page maker attempts step 1 with line 47. If that line is not blank, there is no page, i.e., line 48 should be constructed. But if step 1 succeeds, step 2 is tried on line 46. Another success leads to application of step 3 to line 45. Here "Page number" is supposed to be the name of a program describing a line that is blank except for a one- or two-digit number, flush left or right. If step 4 is successfully applied to line 44, the end of a page occurs at line 47; the 47-line block is released by the page maker.

If the user decides to write two line descriptions, call them RPN for a page number appearing flush right and LPN for a page number appearing flush left, he uses a different program to describe the boundary sequence:

1. Blank, No page
2. Blank, No page
3. RPN, 6
4. Blank, No page
5. End of page
6. LPN, No page

7. Blank, No page
8. End of page

At step 3, failure causes a jump to step 6.

The beginning of a page is described in the following program:

1. Caption, No page
2. Blank, No page
3. Blank, No page
4. Beginning of page

When the page maker applies step 1 successfully to line 47, and goes on to steps 2 (line 46) and 3 (line 45), searching step 4 causes it to deliver lines 1-44 as a complete page. Lines 45-47 remain in the general page storage area; when the user asks for a new page, the page maker moves line 45 to the normal position of line 1, 46 to line 2, and 47 to line 3; then it calls the line maker to produce line 4. These changes of location go astray if maximum line length changes between successive calls.

Each program describing a boundary type has a name, which appears in a call to the page maker.

A user who wants to read a stream of text arranges for his program to call the page maker whenever it is ready to examine a new page. Each call includes:

(1) The address in storage where the first character on the page is to be stored.

(2) The address where the first of four page parameters is to be stored; the others go in the three following cells.

(3) The number of lines in the deepest page to be read.

(4) The number of columns in the longest line to be read.

(5) The identification of a program that can recognize page boundaries.

(6) The address in which the page maker is to store an indication of its actions. The four condition that it can report are (i) normal completion of its task, (ii) overflow of a maximum line length or page depth, (iii) end of input, (iv) end of input and overflow.

4. THE FORMATTER

The business of the routine called formatter is to div: a box into two subboxes or blocks. The formatter works onl) with rectangles. Ordinarily the user begins with a page anc calls the formatter repeatedly, cutting first perhaps horizontally, then (if he likes) vertically, then vertically again, and so on, until he obtains the boxes he can use.

Given a box to be cut, and a direction, the formatter works by seeking a row (if the cut is to be made horizontall or a column (for vertical cuts) that meets a certain specification. One way to specify a line or position across the page is by number. If the input box is 80 character positions, that is to say, if the line length is 80 characters, and if it is to be divided into two equal columns, then the user can specify a cut after position 40. Counting lines and character positions is sometimes good enough, but not always. The user can also write specifications in the language described in Sec. 3, the one he uses to describe page boundaries.

The formatter can cut <u>before</u>, <u>at</u>, or <u>after</u> a row or column of specified type. A cut before position 41 has the same effect as a cut after position 40; either cut yields a box containing everything in columns 1-40, another containing 41-80, for example. A cut <u>at</u> column 40 effectively deletes the 40th column; if the input box consists of 80-position lines, the left-hand output box gets 1-39, the right-hand 41-80.

When two useful boxes are separated by a gutter consisting of several adjacent blank lines or positions, the user can call for a cut <u>at</u> a blank line or position. He thereby excludes the whole gutter: when a cut is made at a line or position of a certain type, all contiguous lines or positions of the same type are eliminated.

If a page consists of three or more blocks of text, each separated by a gutter from the preceding and following, then the user calls the formatter repeatedly. The first call separates the top (or leftmost) box from the remainder of the page; the second call isolates the second box; and so on, until the last call separates the last two boxes from each other. Each call eliminates one gutter.

The user names each box as he creates it, using single letters or arbitrary strings. Sometimes a box has only momentary interest; in the last example above, the first cut produces a box containing one block of text and another containing two or more. The first box is presumably useful subsequently, but the second is to be recut immediately. In such a case, the user can use one name over and over.

Continuing the example, a formatter instruction sequence could be

> Cut the Input into One and Rest at the first blank line;
> Cut the Rest into Two and Rest at the first blank line;
> Cut the Rest into Three and Four at the first blank line.

Suppose that the input is 47 lines deep, and that lines 13, 14, 21, 22, 23, 31, and 45 are blank. Carrying out the first instruction, formatter identifies lines 1-12 as belongin to a box named One and lines 15-47 as belonging to Rest; the first blank line of the input, 13, and the contiguous blank line, 14, are eliminated since the instruction uses the word at. With Rest defined as comprising lines 15-47, formatter obeys the second instruction by assigning lines 15-20 as the box named Two and redefining Rest as lines 24-47.

According to the new definition, the first blank line of Rest is row 31; box Three receives lines 24-30, and box Four receives lines 32-47. The fact that line 45 is blank does not matter, since the three instructions locate only three gutters.

In some input streams, more complex format conventions are used. For example, the upper portion of a form could consist of several lines, say three or more, with the second line occasionally blank. A gutter, beginning somewhere below the third line, could separate the upper portion from the rest. Here the user wishes to call for a cut at a blank line after row 3.

To provide for such situations, the formatter accepts instructions containing multiple line or position descriptions. Each description consists of a preposition (at, before, or after) and a specification. The descriptions come in the order in which they are to be applied. Thus, for example, if an instruction reads

Cut the Input into One and Rest after line 3 at
the first blank line

and rows 2 and 5 are blank in the Input, the formatter counts off three rows and then begins looking for a blank row.

In a sequence of descriptions within one instruction, the three prepositions have altogether different effects. When after is used, subsequent descriptions apply to lines below the one specified, or to character positions further to the right. When before is used, subsequent descriptions apply to lines above the one specified, or to positions further to the left. Once boundaries are fixed, they are never crossed; even counts are made within the bounded region. For example, consider the following instruction:

> Cut the Input into One and Rest after the first blank line before Asterisks after Dots before blank after line 3

and suppose that the input contains lines meeting the given specifications as follows (see Fig. 4): Blank, 7 and 31; Dots, 15; Asterisks, 43. The formatter starts with line 1 of the input and locates line 7, which satisfies the first specification; subsequent searches begin with line 8, making the contents of lines 1-6 irrelevant. Next, line 43 is found; even if a line meeting the Asterisks specification existed above line 7, the search beginning at line 8 would end at line 43. The search region now comprises lines 8-42; if lines of the required kinds cannot be found there, the formatter gives up. In the example, the third part of the instruction leads to line 15, reducing the search region to lines 16-42; then the first blank line within that region, number 31, is found, and the region left to be searched is composed of lines 16-30. Finally, a cut is made after the third line of the search region

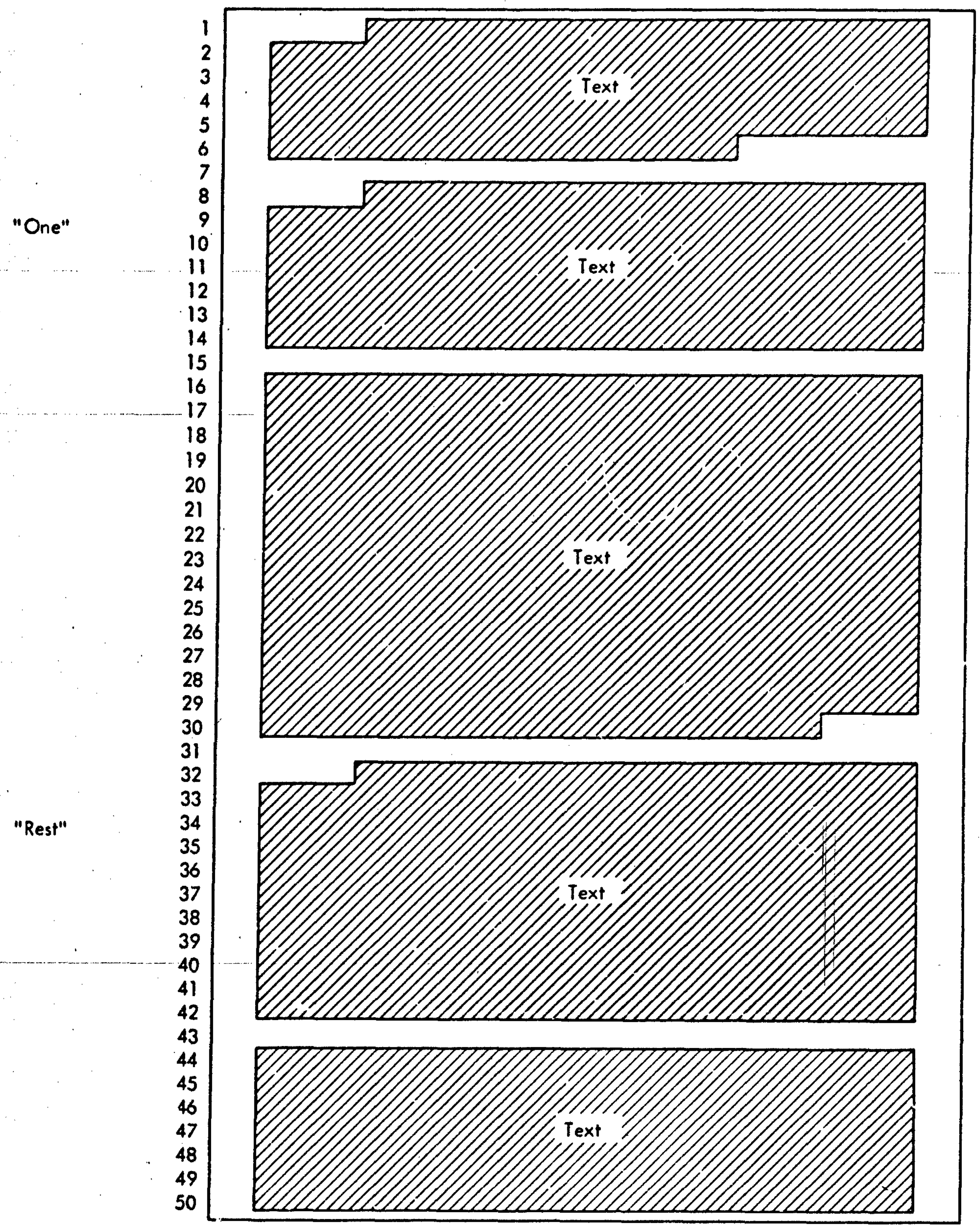


Fig.4—An example of formatting

(i.e., between lines 18 and 19), yielding two boxes: One, lines 1-18; and Rest, lines 19 to the end of Input. Note that if line 17 or 18 had been blank, the third line of a one- or two-line search region would have been sought unsuccessfully.

In a sequence of descriptions, the preposition at can be used only once, in last place; once it is used, no further search region is defined.

The formatter does not move the text it works on; it operates by means of calculations based on page location, page width, column numbers, and character counts. In machine storage, each character occupies a cell, and a page of C columns and R rows occupies C x R consecutive cells. For convenience, suppose that the first cell is number 1; then the first row is in cells 1 through C, the second in C + 1 through 2C, and so on.

The user gives the formatter two locations before his first instruction. One is the location of the first cell assigned to the page. The other is the first of four cells containing the parameters of the page: first column (always 1), last column (C), first character (always 1), and last character (R x C).

Each instruction to the formatter includes three addresses, namely the starting locations for three sets of box parameters: one input and two outputs. These parameters can be interpreted only by reference to the page parameters, as an example demonstrates.

Take C = 50, R = 40; the page parameters are (1, 50, 1, 2000). Assume that previous cuts have yielded a box with parameters (21, 30, 1, 400); this box, with 10 characters per row, runs the depth of the page. If we instruct the formatter to cut this box at its first blank line, the program must examine the following cells, counting from the beginning of the page: 21-30, 71-80, 121-130, ..., 1971-1980. Obviously, the formatter must have row length (page width), page depth, and starting location. Now if cells 271-280 contain blanks, the sixth row of the input box meets the specification, and, assuming that contiguous rows are nonblank, the two sets of output parameters are (21, 30, 1, 50) and (21, 30, 61, 400).

The exact form of a call (or instruction) to the formatter appears in the Appendix (Sec. A.8). Roughly, the user must state whether he wants a <u>row</u> or <u>column</u> cut; where he has placed the parameters of the input box; where he wants the parameters of the output boxes; and how he wants the cut to be made. This last element is a sequence of one or more preposition-specification pairs; in a row-cut instruction, it is assumed that the specification applies to a row, and in a column-cut instruction, to a column.

Application of an instruction can yield an empty box, that is, one containing no rows. It is even possible to obtain two empty boxes, by ordering a cut at a blank row or column and supplying an input box with every character blank; the whole box is excluded as gutter, and both output boxes are vacuous. When an output box is empty, the formatter sets its first parameter at zero.

An empty box results from a successful cut; but an attempt to apply the formatter can also fail. If a blank row is specified, and no row of the input box is blank, the formatter reports a failure; similarly, whenever impossible specifications are imposed, the formatter puts an impossible number in the first parameter location for both output boxes.

5. THE PARSER

The system makes its finest distinctions among different kinds of information on a page by means of a parser. As many parsers may be supplied as there are boxes requiring them. Simple input formats should require none at all.

A parser is supplied with a set of rules equivalent to a context-free grammar. However, the right-hand side of each rule consists not of a string of terminal and nonterminal symbols, but of a regular expression involving both terminals and nonterminals. Those to whom this is a meaningful characterization will need to read little or nothing of what remains in this section. They will find a notation for the rules and directions for keypunching them in the Appendix (Sec. A.9).

The grammar applied to a box consists of a more or less complicated description of a particular kind of object, such as a name, a citation, a date, a chemical formula, or a statement in a computer programming language. The object, since it is assumed to embrace all the material in a box, is not itself the center of attention. More interesting are the smaller objects in terms of which this first one is described.

Suppose that a certain box contains a name, and the parser is intended to distinguish between the first name, initials, and last name. The primary object the parser will investigate is the name itself. In the course of establishing a relationship between the description of names in general given by the rules and the particular string of characters before it, the parser will have cause to investigate the descriptions of other objects, namely, first name, initials, and last name.

The most elementary objects, in terms of which all others must eventually be described, are the characters themselves or, more precisely, the properties assigned to characters by the line maker and referred to from time to time in the preceding sections.

The definition of "Name" might be as follows:

First name   Initial   Last name

The definition of "First name" might be

Upper case letter   Lower case letter string

Whether some part of the material in the box conforms to this description can be verified directly. We check first that the initial character has the property "Upper case". We then check that the next is a letter, and continue collecting characters under this description until the first one that is not a letter--presumably a space--is encountered. The definition of an initial is, if anything, even more straightforward. It might be as follows:

Upper case letter   Period

Finally there would be a definition of "Last name" substantially similar to that for "First name". In fact, it will be instructive to consider the admittedly unlikely possibility that the descriptions of "First name" and "Last name" are identical. In such a case, it would clearly be uneconomical to include separate descriptions; instead we may include a description of an object called, say, "Name part", similar to the one previously proposed for "First name", and amend the description of "Name" to read as follows:

Name part   Initial   Name part

Much of the power of context-free grammars, of which this is an example, comes from the fact that they allow recursive descriptions; in other words, they allow descriptions to refer to themselves. Suppose, for example, we wish to describe an object to be called a "String", consisting of a series of letters (zero or more) and left and right parentheses, provided these occur in matching pairs. One way to describe this would be as follows:

```
   Letter string  [Open parenthesis  String  Close parenthesis]
                                [String]
```

Notice that we use the words "Open parenthesis" and "Close parenthesis" to avoid any possibility of confusion between the objects being described and the syntax of the description itself. In this example, we are using brackets to enclose the part of the description that is optional. Any letter string, and in particular one that includes no parentheses, is acceptable. However, if in the course of examining an actual sequence of characters, an open parenthesis is encountered, the work of checking the string description currently under way is temporarily laid aside while we check that the material from here to the matching close parenthesis itself meets the description of a string. If this turns out to be the case, work is resumed on the original description.

In the course of checking a parenthesized string, another open parenthesis may be encountered at any time. In this case, the checking procedure on the main string and the embedded one are postponed while yet a third is undertaken. If at any stage the attempts to match the given material against the description of a parenthesized string fail because, say, a number or a punctuation mark is encountered, then clearly all the other checking processes that were suspended pending the results of this one also fail.

It is no accident that the word "String" is enclosed in brackets as part of the definition of a string. This must always be the case in a recursive description, that is, a description that refers to itself. Clearly, an object of finite size cannot have another object of the same kind as an obligatory part, for this part would also have to have such a part, and so on indefinitely.

Another requirement on recursive descriptions is that they must always refer to some other object as well as themselves. Clearly, an object meeting a description couched entirely in terms of references to itself has no substance, for the attempt to find out what one such object might be like would involve an indefinite amount of work, but would never get anywhere. The essence of the argument is this: The description of a particular object may involve any finite number of references back to the original description, but, if the process is ever to terminate, then sooner or later the description must be nonvacuously met otherwise than by a reference to itself.

Recursive descriptions can be direct, as in the example we have discussed, or indirect. An equivalent of the string description discussed above would be as follows:

Letter string [Parenthesized expression] [String]

The description of "Parenthesized expression" is as follows:

Open parenthesis   String   Close parenthesis

Here the second description does not refer to itself directly, but each refers directly to the other and therefore to itself indirectly. The same caution must be exercised with indirectly as with directly recursive descriptions. It is, for example, crucial that part of the new description of "String", which leads to indirect recursion, is contained in brackets. It is of course not necessary for the term "String" in the description of "Parenthesized expression" to be in brackets, but only that the recursive loop should be capable of being broken at some point.

We have seen that, in the interests of preserving some symbols for our own use in writing descriptions, it is often expedient to refer to a symbol by writing out its name rather than by using it as a name for itself. In this connection, it is important to make clear when an element of a description is the name of a symbol and when it is a reference to another description. Following the terminology established by grammarians, we shall speak of references to actual symbols as *terminals* and references to other descriptions as *nonterminals*. The object of the parser is to construct a tree with terminals as labels for the extremities and nonterminals at the root and at all other nodes. With these terms at our command we can go on to examine a notation for writing descriptions similar to that used by the computer itself.

The following is the transcription in this notation of our first description of "String" as a sequence of letters incorporating optional parenthesized expressions:

```
          Define         String
A         Terminal       Letter, B
          Goto           A
B         Terminal       Open Parenthesis, C
          Nonterminal    String
          Terminal       Close Parenthesis
          Goto           A
C         Stop
```

The first line identifies what follows as a definition of "String". The second, fourth, and eighth lines have labels on the left-hand side that are arbitrary names distinguishing these lines from all others not only in this definition, but in the whole set of definitions to which this belongs. The second line says that the string may begin with a letter, and to verify whether this is the case for a particular example, we go not to another definition but directly to the data. This is what it means to mark "Letter" as terminal.

If, when the material is checked, a letter is actually found, we move on to the third line of the description which immediately sends us back to the second. A line bearing the word "Goto" is one of the principle ways of modifying the normal reading order of a description, and in this case it has the important effect of causing the specification on Line 2 to be applied an indefinite number of times; exactly the effect required.

Sooner or later the supply of letters will run out. When a specification in a description is not met, we may normally conclude that the description did not fit the material and abandon the attempt. However, in this case, Line 2 contains the name of another line, B, to which reference is to be made if ever this specification fails. Now B is the name of Line 4, and it is therefore to this line that we refer the first character encountered that is not a letter. If that character is an open parenthesis, it is accepted under this description, and we go on to the next line. If not, we find that there is yet another line, C, to which we may refer. Line C, the last one in the definition, carries the single word "Stop", indicating that the description can be regarded as met by the symbols that have been accepted so far, and without including this latest character.

If the first character encountered that is not a letter in fact turns out to be an open parenthesis, our attention passes to Line 5 of the description. Here the question is: Can some sequence of characters be found starting with the character we have now reached in the material that also meets the definition of a string? This is where the process becomes recursive, and we suspend our work on the present description while we go in search of another string that we must know about before we can continue the original job.

Suppose that this attempt to identify an embedded parenthesized expression fails, perhaps because a number or other unacceptable character is encountered. Clearly we cannot now continue where we left off in the original description, and the line calling for an investigation of a nonterminal "String" carries no reference to another line in the description. It would be wrong for it to contain a reference to Line C, for this would lead to the open parenthesis being accepted without any following string or close parenthesis. However, it would also be wrong to abandon the present description as a failure because the parenthesized expression was, in any case, optional. Accordingly, we retrace our steps through the description until we meet a specification with a reference to an alternative line, rejecting previously accepted characters as we go, and take the alternative. In this case, the alternative line bears the single word "Stop", and the material up to, but not including, the open parenthesis is accepted as fitting the description.

This first example has, in fact, the somewhat strange property that it cannot fail no matter what data it is applied to. Suppose the first character in the data is a numeral. Since it is not a letter, the specification on Line 2 will fail, and the line named B, that is Line 4, will be tried next. But since the character is not an open parenthesis, the line named C, that is the last line of the set, will be tried, and this says "Stop", meaning that the definition is to be taken as met by all the characters accepted so far. But no characters have been accepted so far, and the description has therefore been met vacuously--by the null series of characters. There is no rule against definitions of this kind, and, though they can in fact always be avoided, they are sometimes useful.

However, the fact is that this is not quite the description we have in mind for this example. We want somehow to arrange that, however many characters turn out to be covered by the description, there shall always be at least one letter. We shall shortly introduce a device that makes this kind of description particularly easy. But for the sake of the exercise and to demonstrate that, in principle, nothing more is required than we have already introduced, we have in the following lines a description that does in fact fulfill the requirements:

	Define	String
	Terminal	Letter, D
A	Terminal	Letter, B
	Goto	A
B	Terminal	Open Parenthesis, C
	Nonterminal	String
	Terminal	Close Parenthesis
	Goto	A
C	Stop	
D	Terminal	Open Parenthesis
	Nonterminal	String
	Terminal	Close Parenthesis
	Goto	A

The same effect can be achieved using indirect recursion as follows:

	Define	String
	Terminal	Letter, D
A	Terminal	Letter, B

```
            Goto           A
B           Nonterminal    Pstring, C
            Goto           A
C           Stop
D           Nonterminal    Pstring
            Goto           A
            Define         Pstring
            Terminal       Open Parenthesis
            Nonterminal    String
            Terminal       Close Parenthesis
            Stop
```

In order to enrich the language and avoid cumbersome descriptions like the one shown above, we are allowed terminal specifications of the following five kinds:

1. Terminal (A, B, ...)
2. Terminal <u>mult</u> (A, B, ...)
3. Terminal <u>or</u> (A, B, ...)
4. Terminal <u>not</u> (A, B, ...)
5. Terminal <u>anybut</u> (A, B, ...)

Each kind allows a set of character names--often only one--to be included in the parentheses. A sequence of characters meets a specification of the first kind if it consists of just the characters named occurring in the order specified. Thus, the specification "Terminal (C, A, T,)" is met only by the three character string "CAT". A specification of the second kind is met by any string made up entirely of the characters named--or rather characters having the properties named--in the parentheses. Thus the specification "Terminal mult (C, A, T,)" would be met, among indefinitely many others, by the string "CAACAC". A specification of the third kind can be met only by a single character, which is one of those named or a character with at least one of the properties named. The specification "Terminal <u>or</u> (C, A, T)" could be met by a C, an A, or a T. The fourth kind of specification also accepts a single character, which may be any of those not referred to between the parentheses. The fifth kind of specification is similar to the second in that it accepts a string of indefinite length. In this case, however, the string must not include any of the characters referred to.

The definition of "String" can now be as follows:

```
      Define              String
      Terminal            (Open Parenthesis), A
      Nonterminal         String
      Terminal            (Close Parenthesis)
      Goto                B
A     Terminal mult       (Letter)
B     Nonterminal         String, C
C     Stop
```

The program that checks this definition will first look for a parenthesized expression and, if this fails, will follow the direction of line A and seek a string of letters followed by an optional string. If a parenthesized expression is found, the program will also go on to look for another optional string. Thus, every parenthesized expression begins a new string as does the first of every sequence of letters. Figure 5 is the structure of a typical expression according to this definition.

Since the terminal characters on which the parser works can come from a variety of input devices and since, in any case, they are all recoded in an essentially arbitrary way by the line maker, the parser must be told explicitly how to recognize an "a", a "b", an underlined character, a capital, etc. The language in which parser specifications are written therefore contains two other kinds of statement besides those used in definitions. These have the forms:

```
          Term        (a,b,c)
```

and

```
          Terms        (a,b,c),(d,e,f).......
```

The second is simply a shorthand way of writing a series of statements of the first kind. The triples consist of a name by which a character or character property is to be known, and a pair of octal numbers serving to define it.

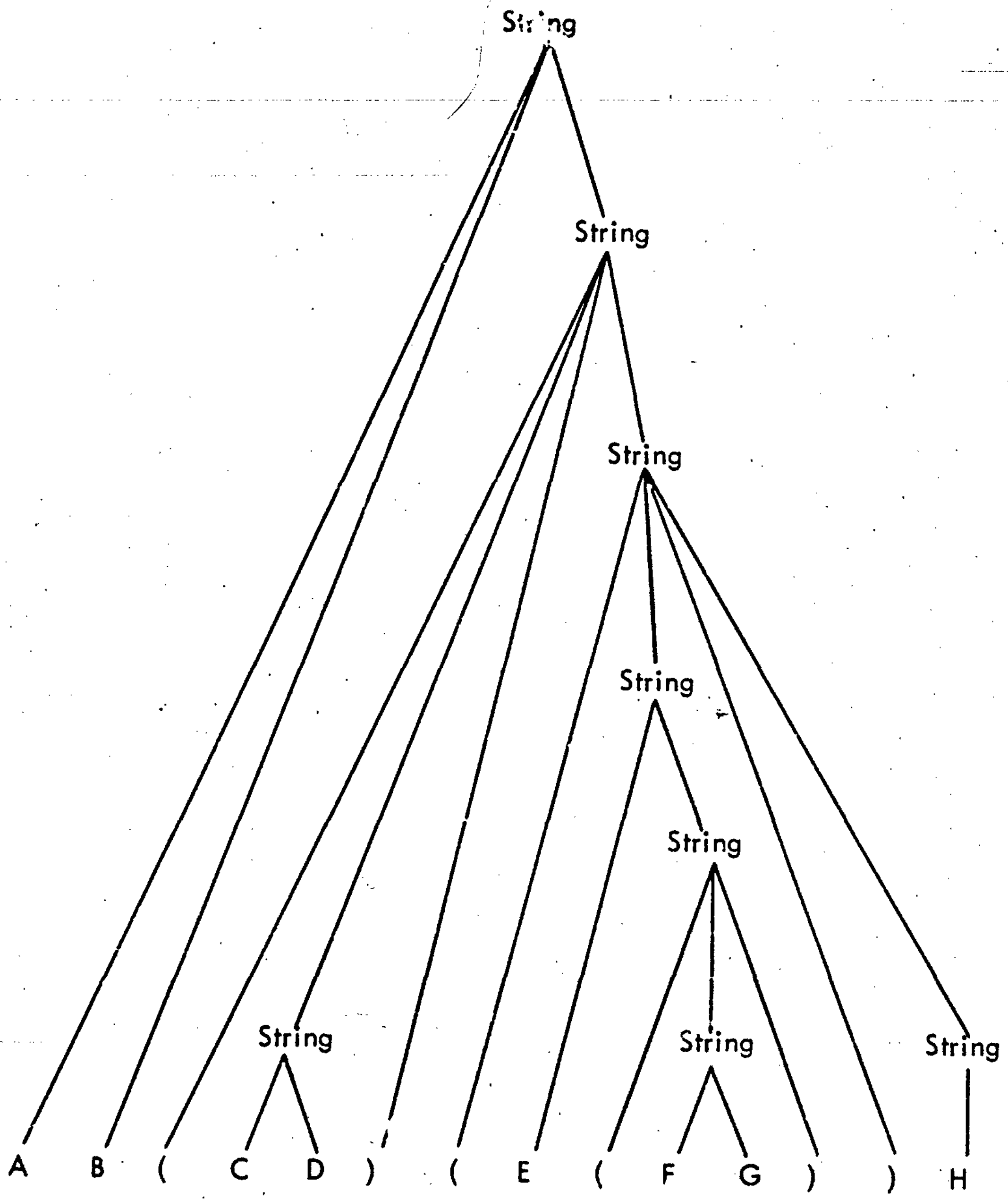


Fig.5—Structure of a string

The octal numbers may contain up to eight digits, which correspond to 24 binary places, the number used to represent a character. The first number shows which of the 24 binary places are significant to recognize it as falling under the definition, and the second shows for each significant place whether it must contain a one or a zero.

We can best make these statements clear with an example. Each of the ten digits except zero could be represented, in the code supplied by the line maker, by its translation into the binary number system. The digit "1" would be "1", "2" would be "10", and so on. Zero cannot be "0" because we have seen that there is very good reason to reserve this for the space character. However, a parser will often have cause to check whether a character is a digit without having any concern for which particular one it is. Let us therefore assume that the line maker supplies a "1" in, say, the fifteenth position of the codes for all digits. This will not only serve to distinguish the digits from other characters but will make it possible for the representation of zero to be consistent with those of the other digits; it will have a "1" in position fifteen and nowhere else.

Table 1 table shows the codes we have set up for digits in both binary and octal notation. Now, there is a rule that numbers cannot be used as names in parser specifications. We shall therefore use "NO" as the name of zero, "N1" as the name of one and so on. We could equally well have used "ZERO", "ONE", etc., instead, but there is something to be said for brevity. The following will introduce these codes to the parser:

Table 1

DECIMAL-BINARY-OCTAL CONVERSION

Digit	Binary	Octal
0	1000000000	1000
1	1000000001	1001
2	1000000010	1002
3	1000000011	1003
4	1000000100	1004
5	1000000101	1005
6	1000000110	1006
7	1000000111	1007
8	1000001000	1010
9	1000001001	1011

```
Terms     (N0,1777,1000),(N1,1777,1001),(N2,1777,1002),
          (N3,1777,1003),(N4,1777,1004),(N5,1777,1005),
          (N6,1777,1006),(N7,1777,1007),(N8,1777,1010),
          (N9,1777,1011),(DIGIT,1000,1000)
```

The last triple defines the word "DIGIT", saying that only the "digit" bit is significant and that it must be a one.

Let us use another bit, say the fourteenth, to mark letters. We can represent the individual letters in a manner analogous to that adopted for the digits. Each can be represented by its serial number in the alphabet. A statement of the following form is in order in the parser:

```
        Terms   (A,2777,2001),(B,2777,2002),......
                ......(Z,2777,32),(LETTER,2000,2000)
```

Notice that letters can perfectly well be used as names for themselves.

Now, let us make the complicating but realistic assumption that we are faced with an input device that does not have separate characters for one and lower-case "L". The symbol used to represent these two should clearly be marked as both letter and digit. As for the rest of the code, it does not matter if we put it with the numbers or the letters or with neither. Let us put it with the letters. This means that the triple defining "1" can be "(N1,1777,1014)", "(N1,2777,2014)", or "(N1,3777,3014)". Which of the "digit" and "letter" bits we choose to make significant does not really matter in this case.

With the definitions so far in force, we can write descriptions with statements like these:

```
        Terminal        (T,H,E)
        Terminal        (LETTER)
        Terminal not    (DIGIT)
        Terminal or     (A,E,I,O,U,N2,N4,N6,N8)
```

But wherever we mention a letter, it will be understood that any form of the letter will do; it does not matter whether it is in upper or lower case or if it is underlined. If the input keyboard device makes these distinctions, then we need some more definitions in order to profit from them.

Suppose that a one in the thirteenth position of the twenty-four-bit code signals an upper-case letter and that the underscore puts a one in the twelfth position. Then we can have these definitions:

```
Terms     (CAP,4000,4000),(UL,12000,12000),(UD,11000,11000)
```

among others. "CAP" is now a general name for any capital letter, "UL" for any underscored letter, "UD" for any underscored digit, and so on. If necessary, we can supply names for the individual letters of the alphabet in upper and lower case. The triple for upper-case "A" will be "(CAPA,6777,6001)"; for upper-case "B", "(CAPB,6777,6002) and so on.

Like numbers, punctuation marks and the space character cannot be used as names for themselves, and we must therefore use definitions like

```
Terms     (SPACE,3777,0000),(PERIOD,3777,0401)....
```

Notice that, even in the definition of "space", not all parts of the code are significant; we leave open the possibility that a space is underscored

We have now covered everything that needs to be said about the parser's principal function of revealing the fine structure of a page. How the information discovered by a parser can be exploited is the subject of the next section. But the parser also has a subsidiary function. We have remarked that some keyboard devices introduce ambiguities in the representation of characters and that allowance must be made for these. If the parser is looking for a number, it will accept a character that just might have been intended for an "L", there being no obviously better policy. It will accept the same character as a letter in another context. In most cases, the rules can, in fact, be written so that these judgments turn out right, and it is therefore appropriate to recode the character so that the ambiguity will not be preserved in the archival form of the material.

The rules for parser specifications allow for a recoding clause to be added to any "Terminal" or "Nonterminal" statement. An example might be

```
Terminal mult          (DIGIT)
Edit                   RD
```

where RD (Recode Digit) refers to a definition introduced in a "Term" or "Terms" statement. That statement might be

```
Term                (RD,11777,0)
```

This time, the interpretation of the two octal numbers must, of course, be different. If a digit is found, then only the bits that match those in the first octal number referred to in the recoding clause are preserved, and any bits specified by the second octal number are added. In the present example, the recoding has the effect of removing the bit indicating a letter, if it is present. Nothing has to be added.

Consider another example. Suppose the parser is set up to recognize hyphens as distinct from minus signs although the input device produces the same code for both and, furthermore, that it recognizes two of these as representing a "dash". The parser might contain the following statements:

```
Terminal       (Minus)
Edit           RMIN
Terminal       (Hyphen)
Edit           RHYPH
Nonterminal    (Dash)
Edit           RDASH
Terms          (Hyphen,777,501),(DASH,777,502),
               (Minus,777,503),(RHYPH,10501,0),
               (RDASH,10500,2),(RMIN,10500,3)
```

This, of course, is only one way to do it. For example, the triples "(RMIN,10501,2)", "(RMIN,10000,503)" and

"(RMIN,10400,103)" among others would have had equivalent effects.

If recoding is specified for a nonterminal, it applies to all the characters covered by the corresponding definition. Recoding of a single character may be specified in more than one place. For example, the "Terminal" instruction referring to a character may be followed by an "Edit" and so may one or more of the "Nonterminals" covering this position. Possible conflicts are resolved as follows: the recoding of characters covered by the most inclusive nonterminal is performed first. The next most inclusive definition is taken next, and so on. The "Edit" following the "Terminal" instruction is applied last.

In describing the way the parser goes to work on a box, we occasionally referred to such things as the current character, the previous character, and the next character. These were to be interpreted relative to the normal left-to-right, top-to-bottom reading order. But the parser can also be instructed to work from the end of the box toward the beginning, that is, from right to left, and from bottom to top. Everything else remains the same. This is useful when the parser is required to work on material only at the bottom of the box.

6. THE SELECTOR

The parser constructs a phrase-structure diagram for the contents of a box or part of a box. This diagram is a tree with labeled nodes. Several direct descendants of a node can bear identical labels, and since recursive grammars are permissible, a node can bear the same label as one or more of its ancestors. In this diagram, the user must seek out nodes covering the terminal strings he needs.

For example, if the input is a citation, the user must find the last name of the authors; perhaps he wants the name of the first, second, ..., last authors in order.

The most convenient plan is probably to select one or a few strings, arrange them as appropriate, and use the result before selecting further input. Otherwise the user would have to provide an intermediate storage format, distinct from the parser's phrase-structure format.

The selector is a set of subroutines for finding nodes and reporting the location in storage of the strings they cover. A node is identified by three properties: (i) It is a descendant of some specifically identified node. (ii) It bears a specified label. (iii) Among the nodes satisfying conditions (i) and (ii), it is the leftmost (<u>or</u> the rightmost) not previously found by the routine. According to condition (iii), it is not possible to select the same node twice.

When the parser finishes working on a box, it sets up the selector, ready for use. The program that called the parser continues by specifying a single node, presumably the origin of the phrase-structure tree. Specification means attaching an index to the node. Now an instruction to find a node can be written; it has four elements:

$$\text{INDEX}_1 = \begin{array}{l}\text{FIRST}\\ \text{LAST}\end{array} \quad (\text{NAME, INDEX}_2$$

Here INDEX_2 points to a known node; to begin with, only the origin qualifies. NAME is a variable that takes as values the labels borne by nodes in the phrase-structure tree. The two options FIRST and LAST yield, respectively, the leftmost and rightmost nodes descendant from the node identified by INDEX_2 and bearing the label given as NAME. At most one node in the tree meets this specification; henceforth it can be pointed out by INDEX_1.

Calls to the FIRST-LAST subroutine can be woven into a program of whatever degree of complexity the user needs. Often enough, the user cannot find a node of interest with a single call. To find the first author's surname, he might begin by attaching an index to the leftmost node labeled author; then, using that index, he could go on to find the descendant node labeled surname. If his program returns to the same instruction to find (for the second or third time) the leftmost author node, it will index the second, third, etc., such nodes.

In a moderately complex situation the user will know that his program has found all nodes meeting a certain set of conditions only by virtue of the fact that a call to FIRST-LAST produces no result. The index, in effect, is attached to no node. The program making the calls must of course test for such a nonresult.

When the user is ready to do so, he can obtain the location of the string covered by a node; he supplies an index to the node and a location (the first of four cells, as usual) for storage of parameters:

PACK (PARAMETERS, INDEX).

The subroutine traces out descendants of the indexed node until it finds terminals, which are still kept in their original place in the page image. It computes the serial number of the first and last characters of the terminal string, relative to the column in which the box lies, and puts the four parameters in the place indicated.

Should a call to the parser result in coverage of only part of a box, the user can select and pack the covered portion and then reduce the box by elimination of what was parsed. The subroutine called REDUCE changes the parameters of the box; after it is applied, the material covered by the parser no longer belongs to any box at all.

When the parser operates backward, the FIRST-LAST subroutine does so as well. FIRST then means rightmost, and LAST means leftmost.

7. THE RECODER

All the processes that have been described except the initial character reader operate on a special intermediate form of the original lines and pages. Each character occupies a whole cell in the memory of the machine, and, in order that the layout of the page is accurately preserved, even extra spaces at the end of a line count as characters. But, against this inefficiency in the use of space we can set the advantages of a coding scheme in which each character is self-sufficient. To discover if the character at a certain place on the page is a letter, a number, a punctuation mark or whatever, whether it is in upper or lower case, or whether it is underlined or carries an accent, it is necessary only to examine the bits stored in a single cell whose location is readily calculable. It is never necessary to search back to see if the shift key was up or down when a given code was produced, or forward to see if an underline produced later was intended to go with this character.

However, when the formatter, the parsers, and the selector have done their work, the usefulness of this extravagant storage and coding scheme will presumably be exhausted.

It is available,* and the user is free to do what he will with it, but for most purposes, and certainly for permanent storage, a more compact representation would be preferable. This is what the recoder is for. It moves data from the page array to a new location provided in the user's program, rearranging them and recasting them according to new coding conventions as it goes.

Just as the parser is an embodiment of a well-established theoretical device, namely a context-free phrase-structure grammar, so also is the recoder; specifically, it is a finite-state transducer. And, once again, there will be those for whom this is an adequate summary of what will be explained in this section, and they can turn straight to the Appendix (Secs. A.11 and A.12).

The recoder examines the character codes in a box or sequence of boxes one by one, moving steadily forward and never going back on its tracks. For each code encountered, it produces a string of new codes--usually one, sometimes none and sometimes more than one. The code or codes produced depend on the old code and also on the history of the recoding process up to that point. Unlike the parser, the recoder cannot go back over ground it has once covered, and

*The FETCH routine (see Appendix, Sec. A.13) can be used to transfer characters one by one from the page array to arbitrary core locations without recoding.

so it cannot easily produce different outputs for a given input depending on other codes coming later in the string. But what has gone before can influence the coding of what follows. This is not a serious restriction precisely becuase of the self-sufficiency of the intermediate codes.

The new codes are six bits (two octal digits) rather than twenty-four bits (eight octal digits) long. This is a restriction that could, on occasion, be serious. However, the physical construction of the IBM 7040/44 computers for which these programs were prepared is such as to make the choice of six-bit codes very natural. All the input and output devices attached to these machines and, in particular, the magnetic tape units, process only six-bit units of code. One cell of machine storage accommodates exactly six such units, and special instructions are provided for manipulating them. In short, while a user of these machines may not always work with codes of six bits, he is very likely to use fundamental units consisting of some multiple of six bits, and these can be readily handled by the system to be described.

The detailed description of the recoder will profit by being tied to an example from the start. Suppose, therefore, that the library catalog card shown in Fig. 6 has been read in to the page array and divided into the boxes A, B, and C by the formatter. Box A contains an accession number, box B a catalog mark, and Box C a citation.

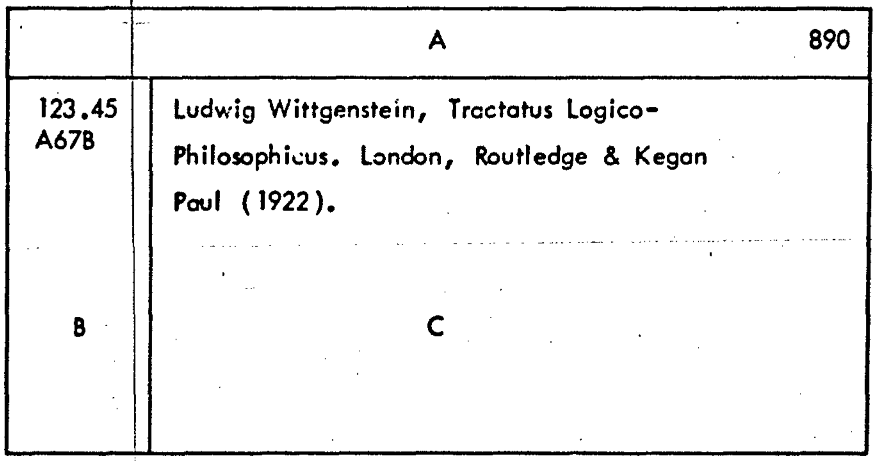

A
890
123.45
A67B
Ludwig Wittgenstein, Tractatus Logico-Philosophicus. London, Routledge & Kegan Paul (1922).
B
C

Fig.6—An input page

Suppose that the immediate concern is with recoding box C.

The program will pick up the first intermediate code in box C and ask if it has certain properties, say those defining the letter "A". If the character has these properties, a new code may be provided in the instruction, which will then be output. In the example, the first letter is an "L", and the instruction gives the location of another instruction to try instead. Sooner or later, an instruction will presumably be encoun·ered specifying properties that the first character in the box does have. It might have the form

Code (L,1,43,done)(tryM)

This can be read "If the character is an 'L', output one six-bit code, namely octal 43, and apply the instruction at location 'done' to the next character. If the character is not an 'L', then go to location 'tryM', and apply that instruction to it". The coding sequence for the Roman alphabet as a whole might begin

```
        done   Code    (A,1,21,done)(tryB)
        try B  Code    (B,1,22,done)(tryC)
        try C  Code    (C,1,23,done)(tryD)
              .              .         .
              .              .         .
              .              .         .
```

Like the parser, the recoder must be told what sets of properties are being referred to by symbols like "A" and "B" in the above instructions, and the method is the same.

Statements of the forms

```
        Term     a,b,c
```

and

```
        Term   (a,b,c),(d,e,f)...
```

are used, the second being simply a shorthand for a string of statements of the first form. The triple consists of a name for the combination of properties being defined and two octal numbers, the first showing which of the 24 bits of the intermediate code are significant and the second giving the values they must have.

It is reasonable to assume that the same bits will always be used for upper case, underlining, diacritics, etc., and that the same set of bits will always be significant for distinguishing letters from one another. Now, the values of these significant bits impose an ordering on the letters of the alphabet that can be exploited in the recoding process Let us assume that there is some subset of the 24-bit field, say the last 6 bits, in which each letter has a distinct representation and, furthermore, that the values found in these bits correspond to the normal alphabetical order. We shall now show how the technique known as binary search[*] can be used in an efficient strategy for recoding the letters.

If the first instruction in the recoding sequence suggested above inquires after an "A" but fails to find it, the possibilities are reduced by one; there are 25 remaining. If the letter is "Z", 26 questions will be asked before the proper output code can be produced, and the average number of questions for a letter will be about 13. If, however, the routine first inquires after an "M" and, failing to find it, notes whether the significant portion of the intermediate code it does find precedes or follows that for "M", the number of possibilities will at once be reduced by

[*]See inter alia Bernard A. Galler, The Language of Computers, McGraw-Hill Book Co., New York, 1962, pp. 97-104, and Kenneth E. Iverson, A Programming Language, John Wiley and Sons, Inc., New York, 1962, pp. 141-144.

half. If the code precedes the one for "M", only the first part of the alphabet need be searched, if it follows, only the last part. The two parts of the alphabet can now be divided into two sections in the same way so that the next character to be sought will in one case be, say, "G" and in the other "T". After two questions have been asked of a given code, it may not have been identified, but less than a quarter of the possibilities still remain open. Therefore, the greatest number of questions that need be asked to identify any letter is five instead of 26.

If we know the language to be recoded, then we can do even better than the simple binary search provided that the significant part of the intermediate codes can be arranged to impose on the letters an order based on their frequency of occurrence in the language. There are many orderings that are about equally good for English, and one is the following:

P D B O C Y A R N M U J E Z L H K F V T Q S I G W X

The first possibility to be tested will be "E", the most frequently occurring letter in the language. When the letter found is not an "E" but precedes it in this sequence, it will be compared with "A", when it follows, with "T". These are the next most frequent letters. The actual order in which the tests are performed for different letters is established with a view not so much to reducing the number of different letters by half at each operation as to making the probability that the letter comes before or after the one tested approximately equal. This probability is estimated from observed frequencies.* A set of instructions for the recoder using the above ordering of the letters is as follows:

```
done   Code   (E,1,25,done)(tryA,tryT)
tryA   Code   (A,1,21,done)(tryO,tryN)
tryO   Code   (O,1,46,done)(tryD,tryY)
tryD   Code   (D,1,24,done)(tryP,tryB)
tryP   Code   (P,1,47,done)
tryB   Code   (B,1,22,done)
tryY   Code   (Y,1,70,done)(tryC)
tryC   Code   (C,1,23,done)
tryN   Code   (N,1,45,done)(tryR,tryU)
tryR   Code   (R,1,51,done)
tryU   Code   (U,1,64,done)(tryM,tryJ)
tryM   Code   (M,1,44,done)
tryJ   Code   (J,1,41,done)
tryT   Code   (T,1,63,done)(tryH,tryI)
```

*The frequencies used here are taken from Lawrence M. Stolurow and Paul I. Jacobs "Tables of Estimated Letter and Letter Combination (Bigram and Trigram) Frequencies in Printed English" which, in its turn, was based on the Lorge Magazine count together with some extra material.

```
tryH  Code   (H,1,30,done)(tryL,tryF)
tryL  Code   (L,1,43,done)(tryZ)
tryZ  Code   (Z,1,71,done)
tryF  Code   (F,1,26,done)(tryK,tryV)
tryK  Code   (K,1,42,done)
tryV  Code   (V,1,65,done)
tryI  Code   (I,1,31,done)(tryS,tryW)
tryS  Code   (S,1,62,done)(tryQ)
tryQ  Code   (Q,1,50,done)
tryW  Code   (W,1,66,done)(tryG,tryX)
tryG  Code   (G,1,27,done)
tryX  Code   (X,1,67,done)
```

This example shows some new forms of the basic "Code" instruction. In the earlier example, the second pair of parentheses enclosed the name of a location from which the next instruction would be taken in case the present one failed. When failure occurs in this example, if the second pair of parentheses contains a pair of location names separated by a comma, the first is used if the character sought comes later than the one found in the sequence established by the 24-bit intermediate codes, the second if the character sought comes earlier than the one found. Thus, in the example, if the character being tested is not an "E" and comes before it in the sequence, "A" is tried next, otherwise "T".

Extending these principles beyond the 26 letters of the Roman alphabet is entirely straightforward so long as the total number of characters to be treated is not greater than 64, the total number of combinations of six bits. But this clearly will happen. A typical typewriter has some 46 keys, each of which produces one of two characters depending on the position of the shift key, giving a total of 92 characters. Furthermore, conventions appointing certain characters as shift markers can be used in the way outlined in Sec. 2 to augment the effective character set still more. In any case, the conventions established by the programming system or the permanent storage format may have to accommodate many more symbols than are produced by any one input device.

There are two standard ways of escaping from the six-bit straight jacket. The simplest is to assign 12 bits to every symbol, thus raising the maximum number of combinations from 64 to 4096. The machinery already described takes this in its stride. It is sufficient to replace instructions like

```
L1    Code    (Char,1,36,done)(L2,L3)
```

with instructions like

```
L1    Code    (Char,2,3536,done)(L2,L3)
```

The other method is to introduce codes that function like shift keys, determining the interpretation of other codes that follow them. This is the solution adopted in the RAND text encoding scheme,* and it requires some extra devices in the recoder.

In the example in Figs. 6 and 7, the first letter is in upper case, and the ones immediately following in lower case. In the RAND scheme, the same six-bit codes are used for the Roman alphabet regardless of case. They are taken to represent lower-case letters except when an upper-case shift character precedes them somewhere in the string. There are also other shift characters for italics, boldface, and the like. The effect of a shift character continues until nullified by a down shift, and in the RAND scheme the only down shift provided is one which nullifies the effect of all shifts currently in force. This is not the place to argue the merits of such a system; it is rather the place to profit from its apparent perversities to demonstrate the power of the recoder.

The recoder must examine the first character of any string presented to it to see if any shift characters need to be issued and then go on to code it as a member of the alphabet. It must examine each subsequent character for

*Martin Kay and Theodore Ziehe, Natural Language in Computer Form, The Rand Corporation, RM-4390-PR, February 1965.

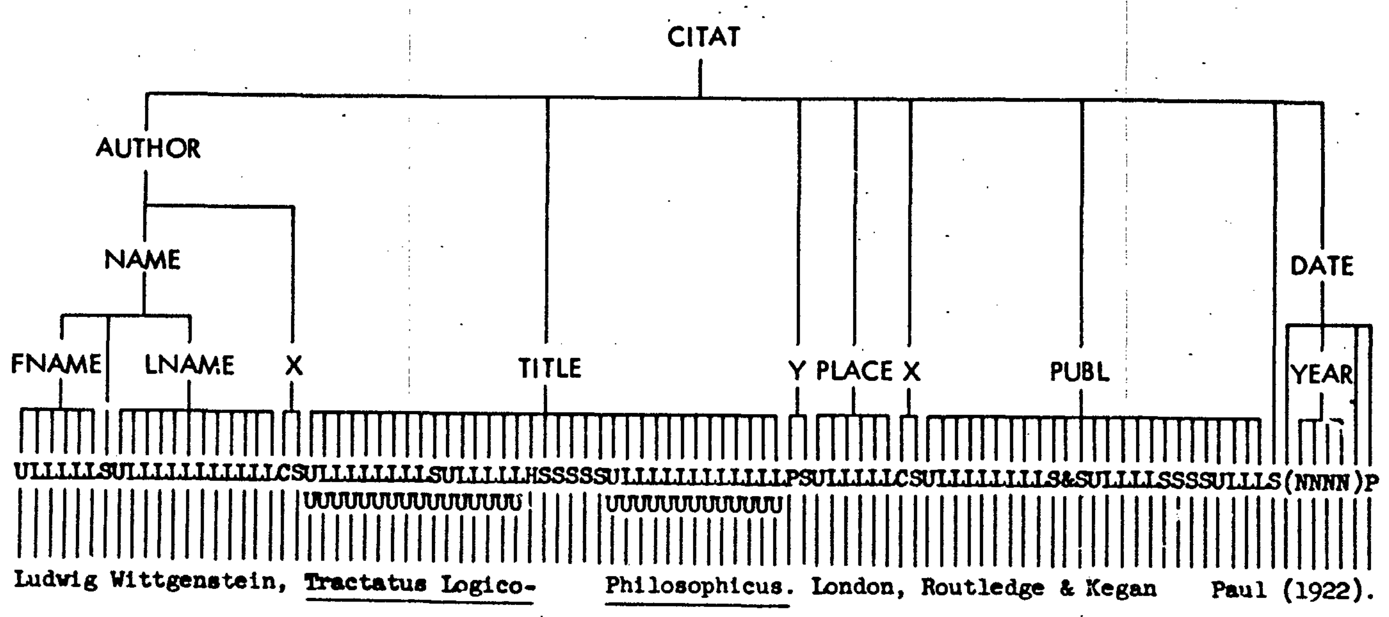

CITAT
AUTHOR
NAME
FNAME
LNAME
X
TITLE
Y
PLACE
X
PUBL
DATE
YEAR
ULLLLLSULLLLLLLLLLLLCSULLLLLLLLLSULLLLLHSSSSSULLLLLLLLLLLLLPSULLLLLCSULLLLLLLLLS&SULLLLSSSSULLLS(NNNN)P
UUUUUUUUUUUUUUUU
UUUUUUUUUUUUUU
Ludwig Wittgenstein, Tractatus Logico- Philosophicus. London, Routledge & Kegan Paul (1922).

Fig.7—Results of parsing

shift changes. For this purpose, it is convenient to be able to apply more than one instruction to the same intermediate code. Consider the following sequence:

```
BEGIN  Code   (Ucase,1,1,S)(LC)
       Code   (Italic,1,2,S)(UNE)
UIE    Code   (UCITE,1,25,UIE)(UIA,UIT)
UIA    Code   (UCITA,1,21,UIE)(UIO,UIN)
UIO    Code   (UCITO,1,46,UIE)(UID,UIY)
UID    Code   (UCITD,1,24,UIE)(UIP,UIB)
UIP    Code   (UCITP,1,47,UIE)(END)
UIB    Code   (UCITB,1,22,UIE)(END)
 .      .              .             .
 .      .              .             .
END    Code   (Any,1,11,S,BEGIN)
UNE    Code   (UCNIE,1,25,UNE)(UNA,UNT)
 .      .              .             .
 .      .              .             .
LC     Code   (Italic,1,2,S)(LND)
LIE    Code   (LCITE,1,25,LIE)(LIA,LIT)
 .      .              .             .
 .      .              .             .
LNE    Code   (LCNIE,1,25,LNE)(LNA,LNT)
 .      .              .             .
 .      .              .             .
```

In some of the instructions, the fourth place in the first set of parentheses is occupied by the letter "S" (for "Same"), which is not to be interpreted as the name of a location from which the next instruction executed is to be taken, but as an indication that, whatever instruction is next used, it must be applied to the same character. The following two sequences are equivalent:

```
          Code   (Ucase,1,1,S)(LC)
          Code   (A,121,NEXT)
```

and

```
          Code   (Ucase,1,1,S,TryA)(LC)
            .         .          .
            .         .          .
   TryA   Code   (A,1,21,NEXT)
```

In the second case, both the "S" and a location to go to in case of success are given.

The table given on p. 75 inquires of the first character presented to it if it is in upper case. If not, it directs attention to the instruction at location LC where the re-coding of lower-case letters begins. If the letter is in upper case, a code (octal 01) is output, and, in the absence of any direction to the contrary, the next instruction in line is taken next. But, since the first instruction contains an "S", the next one will also be applied to the first character. The second instruction works in a similar way, delivering an output code (octal 02) if the character is italic, and otherwise directing that the next instruction be taken from location UNE (Upper-case not italic). Once again, the same input character is retained for processing by the next instruction. Suppose the first character is an upper-case italic "E". The first and second instructions will output octal 01 and 02 respectively, and the instruction at location UIE will contribute octal 25. Only then will work begin on the second character.

The sequence of instructions beginning at UIE constitutes an improved binary search of the letters of the alphabet as already described, but it applies only to upper-case italic letters. Any other letter will fail to be identified and, sooner or later, will cause the instruction at location END to be used. This produces an output character, namely the universal down shift, regardless of what input it is applied to. The name "Any" can conveniently be defined as follows:

```
TERM   Any,0,0
```

Since no bits are significant for recognizing an "Any", any pattern of bits will do. Once the instruction at END has been carried out, the program goes back to the head of the list to output any shift characters necessary.

Each of the four styles of alphabet (upper case, lower case, upper-case italic and lower-case italic) has its own set of binary search instructions. These continue to be used for character after character until one in a new style is encountered. This leads to the instruction at END, which produces a down shift, issues any shift characters needed for the new character style, and causes a transfer to a new set of binary search instructions. The only exception is the sequence that codes lower-case non-italic characters; it does not transfer to END on failure since, in this case, no down shift is required.

The binary-search sequences are only trivially different from one style to another, ard some means of conflating them would be welcome. There are, in fact, two methods, and the first is illustrated in the following sequence:

```
BEGIN  Code    (Ucase,1,1,S)(LC)
ITAL   Code    (Italic, 1,2,S)(UNI)
UI     Do      Letter
       Code    (Ucase,S)(DOWN)
       Code    (Italic,S,UI)(DOWN)
LC     Code    (Italic,1,2,S)(LCN1)
LI1    Do      Letter
       Code    (Italic,S)(DOWN)
       Code    (Ucase,1,1,S,UI)(LI1)
LCN1   Do      Letter
       Code    (Ucase,1,1,S,ITAL)(LCN2)
LCN2   Code    (Italic,1,2,S,LI1)(LCN1)
UNI    Do      Letter
       Code    (Ucase,S)(DOWN)
       Code    (Italic,1,2,S,UI)(UNI)
DOWN   Code    (Any,1,11,S,BEGIN)
       Define  Letter
       Code    (E,1,25,END)(TryA,TryT)
TryA   Code    (A,1,21,END)(TryO,TryN)
         .           .           .
         .           .           .
END    Stop
```

In this sequence, which achieves the same effect as the previous one, and by a fundamentally similar strategy, three new instruction types, "Do", "Define" and "Stop", have been introduced. The instructions included between a "Define" and a "Stop" make up a package that can be inserted by means of a "Do" at any number of other places in the stream. In this case, the package refers only to those bits in the intermediate code that distinguish the letters of the alphabet from one another and not to those that distinguish type styles, and it performs the binary search. As soon as a letter has been identified, the program goes to the instruction at END, which is a "Stop". This causes the machine to take next the instruction immediately following the "Do" that called the package of instructions into use. The instruction "Do" occurs four times in this example, once for each type style. Each "Do" is followed by a pair of instructions that check that the type style of the next character is the same as that of the previous one and, if it is, return to the "Do". Many of these instructions do not cause any output to be produced but merely cause a transfer to the instruction at DOWN, which issues the universal down shift in appropriate circumstances. The reader may find it worthwhile to work through Box C of the example in Fig.6 using these instructions.

The other way of economizing in recoding specifications is to use special forms of the "Code" instruction, which permit glances at characters to the left and right of the current one. The instruction

```
Code    (FROM,Ucase,1,11,S,NEXT)(FAIL)
```

will output the six-bit code octal 11 just in case the previous non-blank character was in upper case and the current character is not; otherwise the program will take the instruction at FAIL. This could be used to output the downshift code in case of a change of type style from upper case. Similarly, the instruction

```
Code   (TO,Ucase,1,1,S,NEXT)(FAIL)
```

will output the code octal 1 in case the previous non-blank character was not in upper case and the current character is, that is, if there is a change to upper case.

When an ordinary "Code" instruction fails, we say that it is sometimes appropriate to distinguish two possibilities, one in which the code found preceded the one sought and one in which it followed. Two possibilities can also usefully be distinguished for instructions involving "FROM" and "TO", one in which the property mentioned in the instruction is not manifested by the character that would have to have it for the instruction to succeed, and one in which both characters show this property so that the change needed for the instruction to succeed is not found. Consider the instruction

```
Code    (FROM,A,1,21,NEXT)(FAIL1,FAIL2)
```

If this is applied to a character that does not follow an "A", it fails in the most conspicuous way and sends the machine to FAIL1 for its next instruction. But, if it is applied to a character that does follow an "A" but that is itself an "A", then it also fails, but this time the next instruction is taken from location FAIL2. Similarly, if the instruction

```
Code    (TO,A,1,21,NEXT)(FAIL1,FAIL2)
```

is applied to a character that is not an "A", it fails and points to FAIL1; if it is applied to an "A" that, however, follows an "A", it fails and points to FAIL2.

In a final example, we shall demonstrate all the facilities available in the recoder. The RAND encoding scheme has a special set of 15 codes called alphabet flags. These function rather like the shift codes already mentioned. They condition the interpretation of all codes following them up to the next alphabet flag. They are different from shifts in that their effect is not cumulative so that no downshift is required from them; the effect of one flag ends with the beginning of another. The characters whose codes follow a given alphabet flag are said to belong to the corresponding alphabet. Provision is made in this way for a Roman, a Cyrillic and a punctuation alphabet among others--these will be enough for the example. Within each alphabet, an arbitrary selection of codes can, in principle, be chosen as shift markers; it happens that the upper-case and italic shifts as well as the universal down shift are represented by the same codes in the Roman and Cyrillic alphabets, and we shall make use of this fact. The recoding sequence is as follows:

```
             Recode  RAND,SP
      S1     Code    (SP,S1)(S3)
      S2     Code    (SP,1,60,S1)(S3)
      S3     Code    (TO,Roman,1,35,S,RAB)(S4,RAA)
      S4     Code    (TO,Cyrillic,1,36,S,CAB)(S5,CAA)
      S5     Code    (TO,Punctuation,1,15,S,PA)(,PA)
      RAG    Do      (Gamma,RomE)
      RAA    Do      (Alpha)(RAG)
      RAB    Do      (Beta)
      RomE   Code    (E,1,25,S2)(RomA,RomT)
               .           .          .
               .           .          .
      CAG    Do      (Gamma,CyrE)
      CAA    Do      (Alpha)(CAG)
      CAB    Do      (Beta)
      CyrE   Code    (E,1,25,S2)(CyrI,CyrY)
               .           .          .
               .           .          .
      PP     Code    (Period,1,33,S2)(PCom,PSemi)
               .           .          .
               .           .          .
```

```
          Define Alpha
          Code   (FROM,Ucas,1,11,S,A1)(A2)
A2        Code   (FROM,Italic,1,11,S)
A1        Stop
          Define Beta
          Code   (Ucase,1,1,S)(B1)
B1        Code   (Italic,1,2,S)(B2)
B2        Stop
          Define Gamma
          Code   (TO,Ucase,1,1,S)(G1)
G1        Code   (TO,Italic,1,2,S)(G2)
G2        Stop
```

The first instruction in this sequence is of a type that still remains to be explained. Its purpose is to give a name to this particular sequence of recoding instructions so that it may be distinguished from others that may be used in the same program and to tell the program the intermediate code for a space. In this case, the recoding sequence is to be known as "RAND", and "SP" is the name of the space character that must be defined somewhere in a TERM or TERMS statement--these statements are not shown in the example. The program must know how to recognize a space in order to make "Code" instructions involving "FROM" and "TO" operate in the way described. Recall that they must be able to identify the nearest non-blank preceding any given character.

The next two instructions, S1 and S2, handle the recoc of spaces. Instruction S1 returns to itself if it identifies a space; otherwise it causes a transfer to S3. In an) case, it does not produce any output and serves simply to delete any spaces from the beginning of the character strir submitted for recoding. We shall see that a space encountered later in the process will be handled by the instruction at S2, which does produce an output code (octal (and then causes a transfer back to S1. The effect of this will be that a sequence of spaces will be closed up and represented by a single space in the result.

The next three instructions (S3, S4, and S5) are the only ones in the sequence that concern alphabet flags. They are used at the very beginning to produce the first alphabet flag, and, when the coding of any character is complete, the program returns to these instructions to chec for a change of alphabet. In order to see how they will op ate when applied to the first character in a string, it is necessary to have a convention for the predecessor of the first character. According to the conventions used by the program, the character whose intermediate code is all zero is deemed to precede the first character in any string so that, provided the bits that are significant for recognizing the alphabets each contain at least one "1", the desire effect is obtained. When the condition specified in one of these instructions is met, that is, when a character belonging to the given alphabet is found following one belonging to another alphabet, then the necessary alphabet flag is output. If the instruction fails because the current character does not belong to the specified alphabet, the next instruction is tried, or, if the character found does not belong to any of the three alphabets, the recoding operation as a whole fails. If, on the other hand, an instruction in this group fails because the current character is indeed in that alphabet, but so was the previous one, then the program transfers to a place in the sequence of instructions that simply continues the recoding of characters in that alphabet (RAA, CAA, or PA).

The instructions for the Roman and Cyrillic alphabets begin with three "Do" instructions, and they illustrate the ways in which the format of this instruction can be generalized on the analogy of the "Code" instruction. When there are two symbols in the first set of parentheses, the first is the name of the package of instructions to be used at that point, and the second is the location of an instruction to be used if that package ends successfully at a "Stop". If there is a second pair of parentheses, they contain the location of the instruction to be taken if one of the instructions in the package fails and does not indicate an alternative.

The instruction package called "Alpha" produces the universal down shift at appropriate places; "Beta" produces any shift codes appropriate to the current character regardless of what has gone before; "Gamma" produces shift codes for new shifts when they are first encountered. "Alpha" is defined in such a way as to succeed, that is, to reach the "Stop" instruction, only if it issues a down-shift code. The "Do" that invokes "Alpha" directs the machine to do "Gamma" in case of failure and "Beta" in case of success. The "Beta" and "Gamma" packages are both needed because, after a down shift, any shifts that remain in force must be reissued whereas, in other places, only new shifts need be issued. The three "Do" instructions do not figure in the instructions for the punctuation alphabet where they are presumably irrelevant.

Each alphabet has a sequence of binary-search instructions (beginning at RomE, CyrE and PP), and when any one of these instructions finds its character, it directs the machine back to the instruction at S2, which we have already discussed.

Once again, the reader is urged to apply these instructions to specific examples to verify that they produce the effects claimed for them.

A program using the recoder must supply it with a name specifying the sequence of instructions to be used, a name that appears in a "Recode" instruction at the head of that sequence. It must also supply parameters specifying boxes in the page array that contain the information to be recoded, and a storage area in the computer in which to store the results. The boxes are taken in the specified order, and the recoded characters follow one another without breaks so that distinctions brought out by the formatter and parsers may once again be submerged. Thus, in the example in Fig. 7, the name "Ludwig Wittgenstein" could be recoded as a single unit, but if the user desired the last name could be placed first. The standard format that places last names first also normally requires them to be followed by commas. The recoder has facilities for introducing constant information like this comma directly from the program. The main program might call for recoding of the author's name somewhat as follows:

```
        Encode   RAND, LNAME, COMMA, FNAME
```

RAND is the name given to the particular sequence of recoding instructions in a RECODE instruction. LNAME and FNAME are names of boxes found on the page, in this case, by the parser. COMMA is the name of a string of intermediate codes provided in the user's program and consisting, in this case, only of a comma.

Any amount of constant material can be introduced in the output sequence. For example, the string "Routledge & Kegan Paul (1922)" from Fig. 7 could be made to contribute "Published by Routledge & Kegan Paul in 1922" in the output.

Appendix

PROGRAMMER'S REFERENCE MANUAL

A.1. INTRODUCTION

Table 2 gives a list of the routines and tables that make up the input system, and the dependency relations among them. The routines whose names appear at the left margin are those normally called by the user's program. The formatter and the parser may be left out, or the formatter left out and the parser applied directly to a page. The recoder can be left out if the user arranges to use the material in the page array directly or provides his own recoding facilities. The line maker can be used in place of the page maker for simple line-by-line input, but single lines must be made to appear as pages to subsequent routines. A simple way of doing this is given in Sec. A.3.

A program using all the capabilities of the input system could be outlined as follows:

1) Call the page maker to obtain a page and its parameters.
2) Call the formatter repeatedly to slice the page into rectangular blocks and obtain their parameters.
3) Call the parser to identify meaningful subsections of a block. Then call the selector repeatedly to obtain the parameters of the smaller blocks.
4) Repeat step 3 for as many blocks as necessary.

Table 2

ROUTINES AND TABLES

- Page Maker
    - Line Maker
        - Character Reader*
        - Input Device Description**
    - Page Tester**
        - Stringer Checker
        - Specification Tables**
- Formatter
    - String Checker
        - Specification Tables**
- Parser**
- Selector
- Recoder
    - Recoding Rules**

*Supplied by the user.

**Constructed by the user as a set of macro-instructions with the system-provided package of macro-definitions.

5) Call the recoder repeatedly to collect and recode the data in blocks.
6) Output the recoded data.
7) Repeat steps 1 through 6 until all of the data are processed.

Table 3 contains a list of deck names, entries, and externs. Those decks that must be provided (character reader) or supplemented (page tester, parser, etc.) are not assigned names. Examination of the entries and externs reveals the logical relationships between the segments of the system.

The routines are written in MAP language for the IBM 7040/7044. They can be called from either MAP or FORTRAN IV programs. All arguments are core locations, all integers are stored right-adjusted in words, and all arrays are stored forward in core.

The calling sequences given are as from a MAP program. The only changes the FORTRAN user need make concern (i) the transmission of constant arguments and (ii) results returned in the accumulator. In MAP, the location of a constant or literal must be supplied, whereas in FORTRAN the constant may be written directly in the argument list and the compiler left to provide a location for it. Thus, MAP

```
CALL PGEMAK(PAGE,PGEPAR,=50,=80,TEST,IND)
```

Table 3

DECKS, ENTRIES AND EXTERNS

ROUTINE/TABLE	DECK NAME	ENTRIES	EXTERNS	COMMENTS
Character reader		CINPUT		Supplied by user
Input device description		INDD, TABS		Written by user with system macro-definitions
Line maker	LNEMAK	LINMAK, PRELIN	CINPUT, INDD, TABS	
Specification tables		One for each table		Written by user with system macro-definitions
String checker	STNCHK	STRCHK		
Page tester		One for each routine	STRCHK One for each specification table	Written by user with system macro-definitions
Page maker	PAGMAK	PGEMAK, CLRPGE	LINMAK	
Formatter	FORMT	FORMAT, ROW, COL, BEFORE, AFTER, AT	STRCHK	
Parser		P.1, P.4, P.L	PICK	Written by user with system macro-definitions. May be composed of separate subroutine decks given appropriate ENTRIES and EXTERNS
Selector	SELECT	PICK, FIRST, LAST, PACK, REDUCE	P.1, P.4, P.L	
Recoder	RECOD	RECODE		
Recoding rules		One for each set		Written by user with system macro-definitions

corresponds to FORTRAN

```
          CALL PGEMAK(PAGE,PGEPAR,50,80,TEST,IND)
```

Routines that return results in the accumulator appear as <u>functions</u> in a FORTRAN program. Thus MAP

```
                    CALL FIRST(NAME,I)
                    STO  K
```

corresponds to FORTRAN

```
                    K = FIRST(NAME,I)
```

FIRST and LAST are the only routines in which Hollerith literals may be communicated as literals. A typical pair would be MAP

```
                    CALL FIRST(=HTITLE ,I)
                    STO  K
```

corresponding to FORTRAN

```
                    CALL ATHRUZ (A,5HTITLE)
                    K = FIRST (A,I)
```

Certain routines and tables must be constructed by the user with macro-instructions provided as part of the system. These all have the following format, familiar to MAP programmers:

Col's 1-6:	the symbolic location, if any, assigned to the statement.
Col's 8-14:	one of the macro-operations: SHIFTS, SHIFT, ENTER, TABS, PGTEST, NOPGE, ENDPGE, BEGPGE, TEST, SPECS, SPEC, GRAM, DEFINE, N, T, GOTO STOP, TERM, RECODE, CODE, DO
Col's 16-72:	the parameters for the statement.

A.2. INPUT DEVICE DESCRIPTION

The line maker refers each input code received to one of the input tables that make up the input device description. The first code is normally referred to the first table and the remainder to tables determined by shifts implied by preceding codes. Each entry of an input table contains the character to be stored in the line array, an indication of any carriage movement, an indication of whether or not there is a change of shift, i.e., a change of input table for the next code, and a line-feed flag.

The input device description consists of macro-instructions. The set is headed by the statement:

SHIFTS $(T_1,T_2,\ldots,T_n)$

where $T_1,T_2,\ldots,T_n$ are the symbolic names of the input tables. The input table names can be any distinct MAP symbols with less than six characters.

The leading statement is followed by the subsets of macro-instructions, each subset describing a shift table. The order in which the tables are described need not correspond to the order of their names in the above list. Each input table description begins with the identifying statement:

```
SHIFT     b
```

where b is the name of the table. Each entry in an input table has the form:

```
ENTER   Inchar, Char, Move, Feed, Shift
```

where

Inchar	=	An octal number, no greater than octal 30000, which is the code supplied by the input device. If this parameter is null, it is taken to be zero.
Char	=	A number of up to eight octal digits, which is the code to be stored in the line array. If this parameter is null, it is treated as zero.

Move	=	"BACK" for backspace;
		"TAB" for tabulate;
		"FORWD" for forward space;
		"CARR" for carriage return;
		Null or zero for no movement.

Feed	= "FEED" for line-feed; Null or zero for no line-feed.
Shift	= Symbolic name of the shift table to be used for the next character. If this parameter is null, it is interpreted as implying no change in shift.

The last one, two, or three parameters may be missing from the list, in which case they will be treated as null.

Each ENTER assembles as one word with the following format:

Bit 0	= 1 for line-feed; 0 for no line-feed.
Bits 1-5	= 0 for no movement; 1 for backspace; 2 for tabulate; 3 for forward space; 4 for carriage return.
Bits 6-11	= position in shift table directory for next shift table to use.
Bits 12-35	= character to be stored in line array.

The ENTER macro-instructions within an input table description may be in any order. The assembled table will be in order of increasing "Inchar" values. Also, there will be an entry in the assembled table for every integral value between zero and the largest Inchar value. For example, if the largest Inchar value is 200, there will be 201 entries in the table. All entries not resulting from an explicit ENTER macro will be zero.

Tab stop settings are specified by the statement:

TABS $(t_1, t_2 \ldots t_n$, length)

where

$t_1, t_2, \ldots t_n$ = the positions of the tab stops relative to the first character position on the line.

length = the total number of character positions on a line.

For example, a line with 20 character positions and tab stops set at positions 5 and 10 would be described by the statement:

TABS (5,10,20)

This statement assembles as an array of "length" cells, with each entry containing the character position of the next tab stop on the line. Thus, in the above example, if the current character position on the line were 6 and the current character's input table entry contained the tab indicator, the next character would go into position 10. Also, any tab indicator at position 10 or thereafter would move the carriage to the end of the line.

A.3. THE LINE MAKER

The calling sequence for this routine is:

```
CALL   LINMAK (Line, Linlen, Ind)
```

where

Line	= location of line array.
Linlen	= location containing line length (maximum number of characters on a line).
Ind	= location for return conditions.

When the line maker is called by the page maker, the line array will be a suitably chosen sequence of cells in the page array. When the routine is entered, each cell is set to zero, the internal code for blank, and the cell for any position for which no character is supplied by the input device will remain zero on exit. The normal return from the line maker occurs when a "line-feed" is received from the input device.

A character reader, supplied by the user, feeds the line maker one character, right-adjusted in the accumulator, in response to the calling sequence:

```
CALL   CINPUT
```

The character reader returns a negative number, whose exact value is insignificant, after the last character in the input stream. If the input is being read from a multi-reel tape file, the negative accumulator can be used to signal an end of reel, the line maker then resuming normal operation when a new reel is supplied.

The first character input from the reader will be looked up in the first input table (Sec. A.2.) and the appropriate character stored in the first cell of the line array. Thereafter, these positions are determined by the information obtained from the input device description for the last character received. However, both settings can be altered by the user's program at any time with the call:

```
CALL   PRELIN (Stpos,Chpos)
```

where

Stpos = location containing the serial number of an input table in the input device description.

Chpos = location containing character position in the line array.

The line maker checks each indicated movement of the carriage for line overflow. If a forward space or tabulate would cause the line to overflow on the right, the routine resets the current character position to "1", sets a bit in IND, and returns the current line to the page maker as it is without testing for a line-feed indication. Any characters remaining between this point and the "line-feed" will be treated as a separate line, which will be delivered when the routine is next called.

The return conditions set in IND are as follows:

2 = Line overflow

1 = Input exhausted

0 = Normal return

Normally, the user's program calls the page maker (Sec. A.7 and this calls the line maker. But if the page maker is omitted from the package because the input format is not sufficiently complex to require it, but other routines in the system (e.g., the parser) are required, then the user's program must provide a set of four parameters that will make each line appear as a page in its own right. In MAP, this can be done simply as follows:

```
              CALL    LINMAK(LINE,LEN,IND)
                .
                .
                .
       LEN    PZE     LENGTH
      PPAR    PZE     1
              PZE     LENGTH
              PZE     1
              PZE     LENGTH
      LINE    BSS     LENGTH
```

where the value of LENGTH is the number of characters in the longest line expected, and the four locations beginning with PPAR are the required page parameters.

A.4. SPECIFICATION TABLES

Specification tables are used to give more or less complete descriptions of character strings for use by the page tester (Sec. A.6.) in recognizing page boundaries and the formatter (Sec. A.8.) in recognizing boxes. In both cases, the "string checker" routine (Sec. A.5.) is called to interpret the tables. Each table consists of a sequence of macro-instructions headed by

```
SPECS    Name
```

where

"Name" is a MAP symbol that will be used to refer to the table.

The table itself is constructed from statements of the form:

```
SPEC    n(Char,Mask)(Match)(No Match)
```

where

n = number of string characters to be compared with "Char." If this argument is missing, it is assumed to be 1.

Char = A number of up to eight octal digits to be compared with the string character(s).

Mask = A number of up to eight octal digits to be used as a mask in the comparison. When a string character is compared with Char, only those bit positions that contain a 1 in Mask will be considered significant. If this argument is missing, it is interpreted as octal 77777777.

"Match" and "No Match" have one of the following forms:

1) "F" meaning false or failure.
2) "T" meaning true or success.
3) "NEXT, Loc" meaning check the next character in the string against the specification at location 'Loc'. If 'Loc' is missing, it is taken to be the immediately following statement.
4) "SAME, Loc" meaning check the same character against the specification at 'Loc'.

The SPEC statements describe a row from left to right or a column from top to bottom. The meaning of a statement is: "Compare n string characters with 'Char', using 'Mask'. If all n match, proceed as indicated by the 'Match' alternative. If any one of the n does not match, proceed as indicated by the 'No Match' alternative".

For example, consider the statement:

```
SPEC    (310,770)(T)(SAME, L1)
```

The string character (current at the time the string checker reaches this specification in the table) will be compared with octal 310, taking account only of the six bits coinciding with octal 770. If it matches, the string satisfies the specification table. If it does not match, the same string character will be compared with the specification at L1.

Since the string checker routine will return a failure indication if it exhausts the string before encountering a success or failure indication in the table, it is possible for a table to consist of one statement such as:

```
          L1    SPEC    (60)(T)(NEXT, L1)
```

This specification table will be satisfied only by a string that contains at least one character equal to octal 60.

To specify a string consisting of 99 characters, all of which are octal 60, the table could consist of the statement:

```
          SPEC    99(60)(T)(F)
```

This one statement is equivalent to the 99 statements:

```
          SPEC    (60)(NEXT)(F)
          SPEC    (60)(NEXT)(F)
           .         .
           .         .
           .         .
           .         .
          SPEC    (60)(T)(F)
```

Notice that if only, say, 35 characters are found meeting the specification, the next character is deemed to be the 36th and not the 2nd.

Each spec statement assembles as two words:

First Word

Bits 0-11	= Match field
Bits 12-35	= Char

Second Word

Bits 0-11	= No Match field
Bits 12-35	= Mask

Match and No Match Fields

Bits 0-1	= 0 for failure
	= 1 for success
	= 2 for check next string character
	= 3 for check same string character
Bits 2-11	= 0 for failure or success
	= Index of next entry to use otherwise

A.5. THE STRING CHECKER

This routine is used by the page tester (Sec. A.6.) and formatter (Sec. A.8.) routines. On each entry it compares a string of characters from a row or column with a specification table and returns a success or failure indication. The calling sequence is:

```
CALL   STRCHK(String,Inc,Strlen,Spec)
```

where

String = location of first character in the string.

Inc = location containing difference of memory locations between two successive characters in the string.

Strlen = location containing number of characters in the string.

Spec = location of the specification table (Sec. A.4.)

When checking rows of a page, Inc will contain "1"; when checking columns, it contains the number of columns on the page.

The routine returns in the accumulator:

1 for success.

0 for failure.

If the routine checks all of the characters in the string without encountering an explicit success or failure indication in the specification table, it returns the failure indication.

A.6. THE PAGE TESTER

Each time the page maker (Sec. A.7.) receives a line from the line maker (Sec. A.3.), the page tester is entered to determine if a complete page has been constructed. The page tester first examines the current line, i.e., the one most recently constructed by the line maker. If this line passes the test, and the specifications involve more than one line, the previous line is examined, then the line previous to that, and so on. Therefore, the user constructs the page tester so that it examines lines from the bottom upward.

A page tester routine is written by the user as a set of macro-instructions. The macro-instruction package provides three types of statements:

1) The identifying statement

   PGTEST   Name

where

"Name" is any valid MAP symbol with no more than five characters. This is the first statement in a routine, and its function is to assign a name to the routine. The name is used in the page maker calling sequence to identify the particular page tester.

2) The terminating statements

   NOPGE
   ENDPGE
   BEGPGE

These macro-instructions, which have no parameters, are used to terminate the test routine. A routine may contain as many such statements as desired. The ENDPGE terminator indicates that the page ends with the last line constructed. The BEGPGE terminator indicates that a new page begins with the last line tested by the routine (i.e., the first constructed among those tested). The NOPGE terminator indicates that a page has not yet been completed.

3) The test statement

```
        TEST   Spec, Alt
```

where

Spec = Name of a specification table (Sec. A.4.) describing a line. All names must be listed in a MAP EXTERN statement when the routines are assembled.

Alt = location of another TEST statement in the sequence or one of the words "NOPGE," "ENDPGE," "BEGPGE."

The first test statement in a program applies to the most recently constructed line. The line is matched with the description in the specification table. If it matches, the next statement in the sequence is executed. If that is a test statement, the previous line on the page is matched with the indicated specification. Whenever a line fails to match, and "Alt" is the location of another test statement, the same line is matched with the indicated specification.

Using one of the terminating words for "Alt" is equivalent to making "Alt" the location of a terminating statement. That is, the statements:

```
          TEST   BLANKS, A1
            .
            .
            .
      A1   BEGPGE
```

are equivalent to:

```
          TEST   BLANKS, BEGPGE
```

The line returned as a page beginning is always the last line tested, regardless of the results of the test. In the above example, a line that does not match the line description in the table called BLANKS will be considered the first line of a new page. In the example:

```
          TEST   BLANKS, NOPGE
          BEGPGE
```

a line that does match the line description is considered the beginning of a new page. A line that does not match causes the routine to return the indication that a page is not complete.

The calling sequence for a page tester routine is:

```
          CALL   Test(Line,Lneno,Cols)
```

where

Test = name assigned to the routine by the PGTEST statement.

Line = location of the line.

Lneno = location containing the line number of the current line (first, second, third, etc.).

Cols = location containing the length of a line.

The routine returns in the accumulator:

1) zero for no page.
2) minus one for end of page.
3) line number of first line for beginning of page.

A.7. THE PAGE MAKER

The calling sequence is:

```
CALL PGEMAK(Page,Pgepar,Depth,Width,Test,Ind)
```

where

Page = location of the page array of dimension C(Depth) x C(Width).

Pgepar = location for page (boundary) parameters--a four-word array.

Depth = location containing the maximum number of lines on the page.

Width = location containing the maximum number of character positions on a line.

Test = name of a page tester routine (Sec. A.6.).
Ind = location for return conditions.

The name of the page tester routine must be an external symbol.

The page maker constructs a page by alternately calling the line maker (Sec. A.3.) and the page tester (Sec. A.6.) until one or more of the following conditions is met:

1) the page tester indicates that the page is complete.
2) the specified maximum number of lines has been constructed, i.e., the page array is full.
3) the line maker indicates that the input is exhausted.

The line maker stores characters directly into the page array. Regardless of the return conditions, the routine stores the page parameters as right-adjusted integers in the four-word array.

The order of the boundary parameters is:

1) First character position across the page.
2) Last character position across the page.
3) Sequence number of first character within the region bounded by 1) and 2).
4) Sequence number of last character within the region bounded by 1) and 2).

The first and third parameters of a page are always 1, and the second parameter is the number of columns on the page. The fourth parameter, number of columns times number of rows, is computed by the routine.

The page maker calls the line maker (Sec. A.3.) with the calling sequence:

```
CALL   LINMAK(Page+n*Width,Cols,Ind)
```

with n initially zero and incremented by one on each successive call. The page tester (Sec. A.6.) routine is called by:

```
CALL   Test(Page+n*Width,n+1,Cols)
```

The three arguments of the calling sequence are the line location, the line number (the first line of the page is line number 1, etc.) and the length of the current line on the page.

The page maker calls on the page tester to tell it when the end of a page or the beginning of a new page has been encountered. When pages are being recognized by their beginnings rather than their ends, one or more lines of the next page will already have been set up in the page array before the parameters of the current one can be constructed and passed to the calling program. For this reason, it is important that cells of the page array should not be tampered with between successive calls to the page maker even if they lie beyond the end of the current page, and that the number of characters on a line should not be varied from one call to the next (see below: Return Condition Bit 34).

The user can cause left-over lines to be ignored on the next entry to the page maker with the calling sequence:

```
CALL   CLRPGE
```

Any combination of the following bits in IND may be set to 1 when a page is returned to the user's program:

Bit 35 - Input is exhausted, and the current page is the last.

Bit 34 - Either a line has been encountered that is longer than the specified maximum, or that maximum has been changed since the last call.

Bit 33 - The end of the current page was not signalled by the page tester either because it overflowed the page array or because the input is exhausted.

A.8. THE FORMATTER

The formatter is initiated by the calling sequence:

```
CALL   FORMAT(Page,Pgepar)
```

where

Page = location of page array.

Pgepar = location of page (boundary) parameters.

This entry to the routine specifies a particular page as current. The calling program can format any number of pages in parallel with appropriate calls of this form to indicate the current page. However, this flexibility is of limited use since the other routines do not allow for parallel input from several devices.

Once a page is specified and FORMAT has been called, a box on the page is subdivided with the calling sequence:

```
        CALL    Sub(a,b,c,Cut_i,Spec_i,...,Cut_n,Spec_n)
```

where

Sub = "ROW" or "COL".

a = location of the input box parameters.

b = location for the first output box parameters.

c = location for the second output box parameters.

Cut_i = "AFTER", "BEFORE" or "AT".

$Spec_i$ = location of a specification table (Sec. A.4.) describing a row or column, or a location containing a row or column number.

"AFTER", "BEFORE" and "AT" are the names of entries in the formatter and must be identified as external symbols by the EXTERN pseudo-op in MAP or the EXTERNAL statement in FORTRAN calling programs. Also, all specification table names must be identified as external symbols.

When "ROW" is used, the formatter divides the input box horizontally. The upper box becomes the first output box and the lower box the second. When "COL" is used, the division is vertical, with the left box the first output box and the right the second. The first time ROW or COL is used for a page, the input box is of course the page itself, and therefore "a" will be the "Pgepar" of the initiati call. It is permissible for either output box parameter location to be the input box parameter location, i.e., b=a or c=a. But "Pgepar" should never be used as an output box parameter location.

All boxes produced by the formatter are rectangular, and the routine assumes that all input boxes are also.

Any number of cut indicators may appear in a single call. They are interpreted from left to right. Each is applied to those rows or columns that satisfy the previous indicators. A cut is completely defined when there is only one division of the input box that will satisfy the indicators. For example, the cut indicator "after row three" following "before row four" completely defines a cut. Whenever this point is reached by the formatter, the routine ignores all further indicators and makes the division.

On the other hand, if all the indicators are processed and the cut is not completely defined, the division is made after/before the last row/column specified. The AT indicator defines a cut completely.

The following conditions result in a return to the calling program <u>without</u> a cut and with the first parameter of both output boxes set to -1:

1) An input box with the first parameter equal to -1.
2) A specification for a row or column that does not exist, encountered by the formatter when the cut is not yet completely defined.

Either or both output boxes may be empty. An empty box is represented by the first box parameter set to zero. If the formatter receives an empty input box, it produces empty output boxes.

A.9. THE PARSER

The program using the parser calls it with the following calling sequence:

```
CALL   Name(Page,Pgepar,Boxpar,Flag)
```

where

Name =	name of the grammar appearing in a GRAM statement (see below).
Page =	location of the page array.
Pgepar =	location of the page parameters.
Boxpar =	location of the parameters of the box on the page to be parsed.
Flag =	location containing plus one if the box is to be parsed forward; minus one if backward.

A box is parsed backward by examining the string of characters in it in reverse order, i.e., starting with the last character and ending with the first.

The parser returns an indication of whether or not the rules of the grammar were successfully applied to the box. The accumulator contains the total number of phrases--successfully applied T and N statements--which is zero when the whole parsing fails.

The set of rules constituting a grammar are constructed from the macro-instructions listed below. As many different grammars as desired may be included in the same assembly as long as all symbols are uniquely defined. Objects define in one grammar may be referred to freely in others. In particular, one grammar can completely contain another.

The following macro-instructions are used in building parsers:

```
        1)  GRAM    Grname,Poname
```

where

Grname = name of the grammar.

Poname = name of the primary object described by the grammar.

There is one statement of this form in every grammar. "Grnam is the name used by the program calling the parser to specif the desired grammar. "Poname" appears in the variable field of a DEFINE statement in the grammar. These names may be any valid MAP symbols.

2) DEFINE Name

where

Name = the name of the nonterminal object described by the statements immediately following this one. It may be any valid MAP symbol. Any number of names may be associated with the same description by listing them in the statement, separated by commas and all enclosed in a pair of parentheses.

3) N (Name,c,Loc)

where

Name = the name of a nonterminal appearing in a DEFINE statement somewhere in the parser.

c = the number of characters in the string claimed by this statement. If the parameter is null or missing, it is interpreted as zero.

Loc = the location of an alternative statement to this one. If there is no alternative in the description, the parameter is omitted.

Both "Name" and "Loc" may be common to more than one grammar provided they are assembled together.

4) T $((t_1,\ldots,t_n),c,Loc)Op$

where

$t_1,\ldots,t_n$ = terminal symbols defined by the TERM (or

TERMS) statement (see 7 below). When n=1, the inside set of parentheses may be omitted.

Op = "OR", "NOT", "MULT", "ANYBUT" or null.

"c" and "Loc" have the same meanings as above.

Any number (except zero) of terminal symbols may be listed in this statement. The statement describes one or more characters, depending on the operator Op, as follows:

Op	Description
"OR"	A single character that is any one of those listed.
"NOT"	A single character that is not any one of those listed.
"MULT"	Any number (greater than zero) of characters, each of which is any one of those listed.
"ANYBUT"	Any number (greater than zero) of characters, each of which is not in the set listed.
Null	All of the characters listed in the listed order.

When the operators MULT and ANYBUT are used, the number of characters claimed, 'c', refers to each application of the description to the string. Usually, then, c = 1 for these operators. For all others, 'c' refers to the number of characters claimed by the entire statement.

5) GOTO Loc

where

Loc = the location of a statement that is to be taken next.

6) STOP

This statement, which contains no parameter list, logically terminates an object description.

7) TERM (Name,Mask,Char)

TERMS ((Name,Mask,Char),...)

where

Name = A MAP symbol used to designate a terminal symbol or class of symbols.

Mask = Up to eight octal digits specifying the bits in the internal 24-bit code that are significant for the identification of this symbol or class of symbols.

Char = Up to eight octal digits giving the values required of each of the bits specified in "Mask". "Char" <u>must</u> have zero for all bits where "Mask" is zero.

A TERMS statement is equivalent to a sequence of TERM statements.

The following is an example of a grammar written in this language:

```
               TERMS        ((SP,77,00),(LET,200,200))
               TERMS        (PRD,77,33),(HY,77,40)
               GRAM         CITGR,CIT
                 .
                 .
                 .
               DEFINE       NAME
               N            (FNAME,2)
               T            (SP,1)MULT
        NM1    N            (MNAME,1,NM2)
               T            (SP,1)MULT
               GOTO         NM1
        NM2    N            LNAME
               STOP

               DEFINE       (FNAME,MNAME)
               T            (LET,1)
               T            (PRD,1,FAM1)
               STOP
        FAM1   T            ((LET,HY),1,FAM2)MULT
        FAM2   STOP

               DEFINE       LNAME
               T            ((LET,HY),1)MULT
               STOP
```

The grammar is named CITGR, and the primary object is CIT. NAME is an element in the description of CIT. NAME consists of a first name called FNAME, any number (including zero) of middle names called MNAME, and a last name called LNAME.

Each of these is separated by at least one terminal called SP, a space. The first name and middle name have the same description: a letter followed by a period or a string of letters and hyphens. The last name is a string of letters and hyphens.

The terminals are based on an input device description in which the low-order two of the eight octal digits of the intermediate representation identify the character. The remaining positions are flags, with the 17th bit position from the left as a letter flag.

The first statement in the description of NAME claims two characters, one for the first name and one for the last. Both of these descriptions can be satisfied by one character; additional characters, if any, are claimed within the subordinate descriptions. Since middle names are optional, no characters are claimed for them in the first statement. Since there must be at least one space, between the first and last name, a third character could have been claimed in the first statement. But then, to avoid claiming that space a second time, the second statement would have to be replaced by:

```
T      SP
T      (SP,1,NM1)MULT
```

A.10. THE SELECTOR

Once a box on a page has been parsed, the user obtains the results of the parsing with repeated calls to the selector routines. Since the parser has only one storage area for its results, the user must select all those desired before parsing another box. Before returning to the main program, the parser activates the selector by giving it the parameters of the parsed box and the parsing direction, forward or backward (via an entry called PICK). The location of the current page and its parameters are not required by the selector.

The user selects a node with one of the calling sequences:

```
          CALL    FIRST(Name,i)
          CALL    LAST(Name,i)
```

where

Name = a location containing the BCD name of the node to be selected. The characters are stored left-adjusted in the word; unused positions are filled with blanks.

i = a location containing a pointer (index) to a node that is an ancestor of the one to be selected.

The routines return, in the accumulator, the pointer (index) to the selected node.

The pointer to the origin is always one, and initially this is the only pointer the user knows. Therefore the first time a node is selected after a box has been parsed, the ancestor referred to will be the primary object. Thereafter it may be any previously selected node.

The routine "FIRST" selects, from the parser's results, a node called NAME that has not been previously selected and that is the first such descendant of the node indicated by i. The routine "LAST" selects the last such descendant. If no such descendant exists, the routines return a pointer of zero. By testing this pointer the user determines whether he has selected all the descendants of a particular node. The terms "first" and "last" are relative to the direction of parsing. If the box was parsed in a forward direction, first means left-most and last means right-most. If the box was parsed in a backward direction, the meanings are reversed.

Once a node has been selected, its parameters can be obtained with the calling sequence:

```
CALL    PACK(Par,i)
```

where

Par = the location of the four-word array in which the new box parameters are to be stored.

i = location containing the pointer to the desired node.

If the value of i is zero, the routine will indicate an empty box (first parameter set to zero). If the desired node represents a character string of zero length, the routine will also indicate an empty box. This arises from an object description in the grammar that is satisfied by the null string of characters.

Once all desired nodes have been selected and packed, the user can reduce the initial box to exclude the portion that was parsed by:

```
CALL   REDUCE
```

This routine calculates the parameters of the portion of the box not parsed and stores them as the parameters of the original box. If the entire box was parsed, the new parameters will indicate an empty box.

Below is a segment of a FORTRAN program using the selection routines. It is based on the grammar example given in the last section.

```
             .
             .
             .
             IND=CITGR(PAGE,PGEPAR,BOXPAR,1)
             IF  (IND)  90,2000,90
   90        CIT=1
             NAME=ATHRUZ(NAME,4HNAME)
  100        NAME=FIRST(NAME,CIT)
```

```
          IF   (NAME)  110,1000,110
110       CALL ATHRUZ  (FNAME,5HFNAME)
          FNAME=FIRST(FNAME,NAME)
          CALL   PACK(FNPAR,FNAME)
          .
          .
          .
200       CALL   ATHRUZ(MNAME,5HMNAME)
          MNAME=FIRST(MNAME,NAME)
          IF  (MNAME)   210,300,210
210       CALL   PACK(MNPAR,MNAME)
          .
          .
          .
          GOTO   200
300       LNAME=FIRST(5HLNAME,NAME)
          CALL   PACK(LNPAR,LNAME)
          .
          .
          .
          GOTO   100
          .
          .
          .
```

The first statement shown calls the parser. The next determines whether the box was successfully parsed. A block of statements to deal with a parsing failure begins with the statement numbered 2000. When control reaches statement 1000, all of the names have been selected from the parser's results.

A.11. RECODING RULES

The six macro-instructions used to write recoding rules are as follows:

1) RECODE Name,Space,Filler

where

"Name" = A MAP symbol to be used in referring to the set of recoding instructions of which this is the first. It will become an external symbol and must therefore be chosen so as not to conflict with other external symbols.

"Space" = A MAP symbol defined in a TERM or TERMS macro-instruction, giving the properties by which the internal code for a space can be recognized.

"Filler" = An octal number of one or two digits to be used as the filler character when the last output word is partially filled.

2) CODE (Type,Name,n,Output,s,Loc1)(Loc2,Loc3)

where

"Type" = "FROM" or "TO" or is omitted. The effect of this parameter is described fully in Sec. 7 of the text and is summarized below.

"Name" = A MAP symbol defined in a TERM or TERMS macro-instruction, giving the properties (bits) required of an intermediate code to satisfy this instruction. This parameter must be present.

"n" = The number of six-bit bytes to be output when this instruction is satisfied. This may be omitted if "Output" is omitted.

"Output" = An octal number of 12 digits or less giving the output to be produced when the instruction is satisfied. If less than 2n digits are given, the code will be right-adjusted in the field. If more than 2n digits are given, the 2n least significant digits are used. This parameter may be omitted if "n" is omitted.

"s" = "S" if present and specifies that, if the instruction is satisfied, the following instruction will be applied to the same intermediate code. If this parameter is omitted, the next instruction will apply to the following code.

"Loc1" = A MAP symbol (other than "S") naming the location from which the next instruction is to be taken after this one is satisfied. It may be omitted, in which case the instruction immediately following this one will be used.

"Loc2 and Loc3" = Locations from which the next instruction will be taken if this one is not satisfied.

If the instruction fails and "Loc2" and "Loc3" are both missing, then the whole recoding process or the current subroutine fails, and an error condition is signalled in the main program. (See Sec. A.12.) If only one location is given and no comma, the instruction at that location is used in all cases of failure. If a single location is given together with a comma to show whether it is intended to fill the position of "Loc2" or "Loc3", then the whole recoding process fails in cases where the missing location would normally be used.

If "Type" is omitted, "Loc2" is used if the significant part of the intermediate code found is algebraically less than the one sought; if it is greater, "Loc3" is used. If "Type" = "TO", "Loc2" is used if the current code does not have the specified properties; "Loc3" is used if both the current code and the previous non-blank one have the specified properties. If "Type" = "FROM", "Loc2" is used if the previous non-blank code does not have the properties specified in the instruction; "Loc3" is used if both the previous non-blank and the current code have the specified properties.

```
        3)  DO    Loc1
            DO    (Loc1,Loc2)(Loc3)
```

where

"Loc1", "Loc2", and "Loc3" = MAP symbols

The symbol "Loc1" must be defined in a "DEFINE" macro-instruction as the name of a subroutine. The subroutine is called by this instruction, and if all its instructions are satisfied, the program takes its next instruction from "Loc2". If "Loc2" is omitted, the next following location is used. The symbol "Loc3" is used if the last instruction to be executed in the subroutine was not satisfied, and no alternative location was provided. There is no provision for recursive subroutines.

4) STOP

This macro-instruction, which has no parameters, marks the successful conclusion of a subroutine.

5) DEFINE Name

where

"Name" = A MAP symbol naming a subroutine, i.e., a self-contained package of recoding instructions called by DO instructions elsewhere in the program and ending in a STOP instruction. The first instruction of a subroutine must immediately follow the DEFINE.

6) TERM (Name,Mask,Char)
TERMS((Name,Mask,Char)......)

where

"Name" = A MAP symbol used to designate a class of one or more interm

"Mask" = Up to twelve octal digits specifying the bits that are significant for the recognition of this class of codes.

"Char" = Up to twelve octal digits giving the values required of each of the bits specified in "Mask".

A.12. THE RECODER

The calling sequence to the recoder is:

```
CALL  RECODE(Output,Length,Table,Page,Pgepar,a1,a2,....an
```

where

"Output"	=	The location of an array where the recoder output is to be stored.
"Length"	=	The number of words in the "output" array.
"Table"	=	The name of the sequence of recoding instructions to be used. This must appear in a RECODE instruction in the recoding rules (Sec. A.11.).
"Page"	=	The location of the page array.
"Pgepar"	=	The location of the page parameters.
"a1,..an"	=	Location of box parameters or constant arrays.

RECODE is used as a function in FORTRAN and must be declared to be of type INTEGER. A constant array is distinguished by the fact that it contains a negative number in its first cell. An array headed by the number -n contains n intermediate character codes in the immediately following cells and therefore consists of n+1 cells altogether.

The following return codes are transmitted through the accumulator:

-1	= An ill-formed set of box parameters was offered for recoding (first word = octal 777777777777).
-2	= Failure to recode--an intermediate code was encountered for which no equivalent was provided. The last instruction applied therefore failed, but pointed to no alternative location.
-3	= The output array overflowed.
+n	= The input was successfully recoded, and n output words were produced.

A.13. FETCH

The FETCH routine is used internally by the system to extract individual intermediate codes from the page array, and it can be called by the user who does not wish to include a recoding step. There are two calling sequences, one to provide parameters to the routine and one to obtain the codes. These are:

```
1)   CALL   PREFET(Page,Pgepar,Boxpar,Flag,Error)
```

where

"Page"	= The location of the page array.
"Pgepar"	= Location of the page parameters.

"Boxpar" = Location of the parameters of the box from which codes are to be fetched.

"Flag" = 1 if the character codes in the box are to be considered numbered from top left to bottom right and -1 if they are to be numbered from bottom right to top left.

"Error" = A location to which return should be made from "CALL FETCH" if the serial number of the character called for is such as to place it outside the specified box.

On return the accumulator contains the number of characters in the box (zero for an empty box) or octal 777777777777 if the first box parameter is that.

2) CALL FETCH

The ith intermediate code in the box, where i is given in index register 1, is returned in the accumulator. The sign bit is positive unless the code is the first in the box on a line, i.e., unless it begins a new line in the box in which case both sign and p-bit are set to one. If the number in index 1 is greater than the number of characters in the box, the return is to the error location.

This routine may not be used between a call to the parser and the last corresponding call to the selector. Also, FETCH cannot be called from a FORTRAN program.

UMENT CONTROL DATA

IGINATING ACTIVITY THE RAND CORPORATION	2a REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP

EPORT TITLE

COMPUTER ROUTINES TO READ NATURAL TEXT WITH COMPLEX FORMATS

UTHOR(S) (Last name, first name, initial)

Graves, Patricia A, David G. Hays, Martin Kay and Theodore W. Ziehe

EPORT DATE ~~July 1966~~ August 1966	6a TOTAL No. OF PAGES 143	6b. No. OF REFS. ---
ONTRACT OR GRANT No. AF49(638)-1700	8. ORIGINATOR'S REPORT No. RM-4920-PR	
AVAILABILITY/LIMITATION NOTICES DDC 1	9b. SPONSORING AGENCY United States Air Force Project RAND	

ABSTRACT

A description of a system of IBM 7040/
subroutines that will accept natural-
nguage input with complex formats--e.g.,
om books, journals, questionnaires,
ippings, library catalog cards, etc.--
epared by any typesetting device or
her machine (typewriter, keypunch, etc.).
puts are transcribed by the computer
to a standard code for machine process-
g and can be rearranged into any desired
rmat for storage or output. Different
nds of information are recognized by
plicit markers, position on the line or
ge, or syntactic clues given by other
ems. The subroutines can be used singly
together; they may be called from
ther FORTRAN or MAP programs. A detailed
ogrammers' guide is included.

11. KEY WORDS

Algorithms
Bibliography
Catalogs
Data processing
Dictionaries
Documents
FORTRAN
Grammar
Indexes
Information storage and
retrieval
Language
Linguistics
Library science
Computer languages
Computer programs
IBM 7040/7044
MAP language