Gilbert W. King

Table Look-up Procedures in Language Processing Part I The Raw Text

Abstract: A method of addressing memories is described which is very powerful in the processing of natural languages, where the arithmetic or logical operations are either nonexistent or do not lend themselves to algorithmic description. The main feature is the guarantee of initiation of an exhaustive search for a linguistic word at a point just beyond the desired address. Sequential search backwards not only locates an address if it is there but also provides identification of a longest match first. The method is further extended to provide "conditional" addressing by prefixing subsequent addresses from information obtained in earlier searches.

Introduction

It is safe to say that the processing of languages for information retrieval or automatic translation will always require a table look-up procedure, i.e., reference to a dictionary of some sort. There can never be an algorithm, or stored program, which can give the meaning of a word from its spelling, however encoded. In a subsequent paper of this series we shall be able to show that more sophisticated processing, to discern the grammatical structure of sentences and resolve the ambiguities of meaning, will not in general be resolved by algorithms, but will also require reference to tables in which the properties of the language are listed. The art of data processing with tables is as old as computation, but in recent years has been neglected, at least in scientific computation.

This paper is concerned primarily with the first look-up in a dictionary. Now this operation is characterized by the fact that dictionaries are exceedingly large, of the order of a billion bits. Tables of this size and character have certain characteristics which require novel approaches to the procedure of search, and indeed an appropriate method of addressing lends itself to the execution, by table look-up, of so-called "logical" operations which are normally considered to require stored programs.[1]

Table look-up

The function of a very large memory in a machine organization is to provide table look-up; that is, given an argument x, to find the value of a function of x, $f(x)$. The necessity for table look-up arises when $f(x)$ cannot be computed from x by an arithmetic or logical algorithm, or when the computation is too long. In the early days of computing, almost all functions were obtained by table look-up. (This was done even for reciprocals, $f(x)=1/x$, before automatic division was available.) To obtain $f(x)$ for x with many digits, a straightforward table would, of course, have to be of impractical size. However, values could be obtained from tables of restricted values of x, and $f(x)$ constructed with the aid of interpolation tables or by further calculation (addition theorems, et cetera).

In a stored-program machine a function is computed by an algorithm, such as a continued fraction. As long as computation of algorithms is faster than the necessary sequence of table look-ups, the latter is of no interest for the calculation of most mathematical functions.

In non-numerical work the situation is quite different. For example, it is impossible to calculate the meaning of a word from its spelling by an algorithm in the equivalent of a computer arithmetic unit. Thus, if words are to be processed as they occur in natural languages, a dictionary storage and look-up must be provided in a machine.

The functions $f(x)$ which cannot be calculated, such as meanings of words in a dictionary, telephone numbers of subscribers, prices and inventory of merchandise, et cetera, have other concomitant properties. The set of val-

ues of x, for which $f(x)$ is desired, is very large, and large in quite a different way from the set of x's in numerical work. The latter are generally a finite set, say a subset of all ten-decimal digit numbers. It is interesting to note that there are, however, numerical tables of the type considered here, namely, optimum interval tables with variable-length addresses (to stabilize precision).

The x's in non-numerical work (and in the case just mentioned) are, if not arbitrarily restricted, essentially an unbounded set from an infinite set which is everywhere dense. That is, given an x_1, and another nearby x_2, we find that in general the number of digits in x_1 and x_2 are of no particular fixed length, and may be of indefinite length. Moreover, one cannot tell, unless one in fact looks in the table, whether there is not another x which lies between x_1 and x_2, since x, and the set of x's, belong essentially to the continuum. For example, in a dictionary with the ordinary alphabetic order, there may be the words "bee" and "beef." Between these can be "beech," "beeches," and any idiomatic phrases beginning with either of these words.

Integral address

It is evident that in lexical processing we have in general the requirement of table look-up in memories which have to be extremely large, and in which we cannot tell beforehand what entries, hence addresses, exist. The method of addressing, so current in present-day computers, by absolute addresses is impossible. Absolute addressing is impossible for another reason. For the type of addresses, x, used in dictionaries, directories and catalogs, there is no way to compute from x the page and line number on which x will be found. Instead, an integral addressing method is required in which the address determining the location of the value, $f(x)$, is found by locating x itself.[2]

The combination $x{:}f(x)$ will be called an *entry* and both x and $f(x)$ are stored. The memory stores a sequence of entries. In general, x will be made up of digits or characters, and can be ordered. Since both x and $f(x)$ will be of variable length, it is most economical not to store an entry in a fixed cell, but to store the entries in a continuous stream. This means that the address of an entry, namely x, must be recognized by a first symbol α, meaning "begin entry," or more specifically "begin address," which also acts as "end $f(x)$" for the previous entry.

Because we cannot tell beforehand whether or not x is indeed an entry, nor its length, the location of an address, hence an entry, must, in the final analysis, be by sequential search. However, this does not mean that the whole memory must be exhaustively searched. The entries can be grouped in blocks, corresponding to pages of a dictionary or directory. The division may be systematic, say, by the first letter, or pair of letters; or the groups may contain a fixed number of entries, or of characters, or in any arbitrary way. If the information is stored on a drum or disc, the pages correspond to a track; page turning corresponds to jumping tracks; and sequential search corresponds to sitting in a track, reading everything that passes by. Similar analogies could be made in core or other memories.

There is another instruction symbol which we have represented above by a colon, between the address x and value $f(x)$. To avoid confusion with a colon as a symbol in the text, and possibly in an address, let us call this instruction symbol τ. Its function is to indicate end of the address, x, and beginning of the function value, $f(x)$; i.e., it ends matching on x and initiates readout of $f(x)$. The entry thus is composed of a series of symbols $\alpha x \tau f(x)$.

Search procedure

The search may now proceed by essentially random access to a page. Consider the look-up of an address x. Assume the search mechanism, after the last search, is at an address x_1. The desired address x and x_1 may be immediately compared. Since in general x and x_1 may be of different length, this comparison is serial, bit by bit. Within a few binary characters it can be determined whether $x<x_1$ or $x>x_1$ and hence whether to look back or ahead. Let us first take the case $x>x_1$. The next page will be examined, and an entry x_i picked, perhaps at random, then a new comparison made. This procedure is followed page by page until $x<x_i$. This condition will first arise within one page after the location of the desired address x (if it exists). (If the first comparison resulted in $x<x_1$, pages are turned back until $x>x_i$, then the routine for the former case is followed, giving only one page advance.) Thus, by sampling from successive pages, the search mechanism reaches a point in the store a short distance beyond the desired address. If comparison and turning pages can be made very fast, as seems to be the case in practice, the search mechanism has made essentially random access to a neighborhood just beyond the desired entry.[3]

One can conceive of other methods of arriving at the neighborhood of the desired address. For example, the first digit or two of x could be used as an absolute address. However, since in general the number of addresses beginning with the first one or two characters are not uniformly distributed, a small table (or matrix) has to be consulted.[4] These techniques require additional hardware, and do not present the feature, which we shall see is really valuable, of arriving just beyond x.

At this point, sequential search is resorted to, as it must be. The search now proceeds in descending order of x. A matching address may be found on the page during this sequential search in the decreasing direction (i.e., toward the "top of the page"). However, it may be necessary to turn to the previous page. If, in the page-turning process, the time to turn a page and sample is short, the sampling on successive pages occurs at essentially the same "eye level" or distance down a page. Consequently, in the sequential-search phase it will not be necessary to continue backwards for a distance appreciably longer than a page. The average distance will be half a page.

If the initiation of the search starts at some arbitrary point in the memory x_i (corresponding to the last address found), the distance (in terms of number of pages turned) to a new address x_j is on the average $|x_i - x_j|$. The average value of this when all addresses are equally probable is one-third the number of pages in the dictionary. Minimum access time is achieved when the time to sequentially scan half a page equals the time to turn one-third of the number of pages in the dictionary. This condition in fact determines the optimum size of a page.

Principle of longest match

As we are considering a function $f(x)$ which may exist for any value x from the continuum, we are assuming that addresses x, for which $f(x)$ is desired, arise in some processing, and are samples from the continuum. If ζ_i are individual characters, binary or otherwise, an address is some indefinite sequence of them, $x = \zeta_1 \zeta_2 \ldots$, which for the moment we shall assume lie in a semi-infinite Input Register.

A consequence of the above search procedure, especially the final stage of sequential search in descending order, provides a valuable feature, in that the longest match of an address with the contents of the Input Register is obtained in one search, without programming an assembly of subroutines.

For example, assume the numerical equivalent for the contents of the Input Register is .14768 . . . , and that the table contains the addresses and function values given in Table 1.

Table 1

Page n		*Page n + 1*	
		.12	$f(.12)$
.0015	$f(.0015)$	.13	$f(.13)$
.085	$f(.085)$	.14	$f(.14)$
.09	$f(.09)$	.146	$f(.146)$
.09999	$f(.09999)$	.147	$f(.147)$
.117	$f(.117)$	.15	$f(.15)$
.1181	$f(.1181)$	.15123	$f(.15123)$
.1192	$f(.1192)$	.159	$f(.159)$

Assume the search started at some page $< n$. Sampling on these pages up to and including page n gives x greater than the sampled address. On page $n+1$ assume the sample address is $x_i = .15123$. Now $x < x_i$, i.e., a number slightly larger than the .14768 . . . in the register, is found by random access; then the addresses .15123, .15, .147, .146 in descending order are scanned. The largest sequence in the Input Register for which a complete match can be made is .147, and the value for this would be read out. If, however, .147 was not an entry, then the value for .14 would be located in the search.

If the desired address had been $x = .117$, and the sampling had been high on the pages, page $n+1$ would have been reached before $x < x_i$. The sequential search would then have been up through page $n+1$, without success. But near the bottom of page n, a match would have been found. The total distance scanned would have been less than one page.

An application

An application of this method of search occurs in the preliminaries of mechanical translation of languages. Another very similar application is the processing of text of natural language into a canonical form for information retrieval. The input text about which lexical information is desired for the translation process is composed of an indefinite sequence of characters. Chains of these characters constitute addresses to the memory, or dictionary. The simplest case is when the chain is a single word. It is readily seen that despite their variable length, words can be used as addresses in the above scheme and identified as addresses to entries in the table, provided they are, indeed, listed in the table.

In automatic translation, individual words are not necessarily the linguistic units to deal with.[5] It is, in fact, often desirable to treat groups of contiguous words or idiomatic phrases as units. Clearly, if the space between the words be given a numerical code and treated as a character, there is no difficulty at all in having groups of words as addresses. The principle of longest match, however, offers an interesting feature. It is not necessary to determine beforehand whether a word group is going to exist in the table—perhaps only the individual words occur there. Since the longest sequence is matched first, a word group will always be recognized if it is an entry. Otherwise, its components are looked up individually. Note that this decision is made automatically in the one search.

Conversely, the entries in the table may correspond to parts of words, e.g., stems and endings, prefixes and suffixes, or the dissections of compound words. Again no difficulty is encountered. For example, assume "bag" is an entry, but "bags" is not. If "bags . ." were in the Input Register, the match would be on "bag." The residue "s#" would be found as the next address. (The symbol # indicates the character assigned to the linguistic item "space.") This allows a dictionary to be made of stems and endings only, and avoids the wastefulness of having to enter all the inflectional forms of every word of the language.

For example, assume the Input Register contained the text: "he singed the beard of the . . ."

The first look-up would find "he" as an address, and on the match this would be shifted out, as would be the space. The register would now contain

"singed the beard of the . . ."

(Three new characters would be inserted at the right to make up for those shifted out.) Now a second search is made. If "singed the beard" had been considered an idiomatic phrase, to be translated as a whole, an entry with this as an address would exist. A match would be found, and the words "singed the beard" shifted out, so that the register would now contain

"of the . . ."

If the phrase "singed the beard" had not been considered particularly idiomatic, and not made an entry, a shorter match would have been found in this search. If the word "singed" was an entry, it would be found. Then the successive conditions in the register would be

"the beard of the . . ."

"beard of the . . ."

On the other hand, it may have been decided that "singe" was a straightforward word so that its various inflections need not have been entered specifically. Then the longest match would be on "singe," giving the information about this stem. Then we would have

"d the beard of the . . ."

The next address found would be "d#," giving the information that this is a verbal past tense ending.

Note that "sing" would not be found, giving rise to the incorrect dissection "sing-ed."

The ability to obtain matches all the way from parts of words to indefinitely long groups of words makes it unnecessary to process the text before addressing the memory. (This could not be done anyway in the case of newly constructed compound words, like "leptonuclear," whose composition cannot be anticipated.)

The dynamic dissection of words, however, can lead to errors. The word "needless" might not be in the table, because it is a simple compound of "need" and "less." But it would be split on the longest match, "needle," leaving "ss" which would not be found. Or if "needles" were an entry, this would leave "s" which is an entry (an English ending to nouns and verbs). The remedy is to anticipate these peculiar cases, and enter the whole word.

A significant feature of the longest-match principle is that one can simultaneously play both the strategy of stems and endings, and of whole inflected forms, adopting whichever is of most assistance, with absolutely no modifications of hardware, control or initial decision.

When a match has been found, the value $f(x)$ is read out of memory, and the contents of the Input Register shifted forward by the corresponding number of characters in the address just found. (Since in practice the Register is of finite length, new text is read in to fill it at the tail end.) Then a new search is initiated, scanning down the contents of the Register with its shifted contents.

Break points

If no match were found, the scheme as outlined so far would allow the search to proceed in descending order to the beginning of the dictionary. The search may be stopped, however, soon after the place is passed where the address would have been. This is done by means of break points. These are one-letter addresses corresponding to all characters used, with $f_T(x)$ being x itself or a "transliteration" of x. A match is always obtained on a break point, $f_T(x)$ is read out, and the contents of the Input Register are shifted one character. The point here is that transliteration is initiated in the original search procedure, without further ado.

In practice, break points with more than one letter may be used, to speed up the search. At least one should appear on each page.

This is the general procedure for handling addresses, or sequences of characters, in the Input Register which are not in the memory. In the application to dictionary look-up of running text, this condition will be produced by proper nouns or words not in the dictionary. In this situation, the complete word should be transliterated. The function value of the break points, $f_T(x)$, will then be the transliterations of the break-point symbol x. The method by which the transliteration routine is pursued through the whole word is described below.

In the event some of the single characters are legitimate addresses, as in single-letter words like "a" in English, this procedure, without modification, would give false values (in the example $f(a)$) and the unidentifiable word incorrectly dissected. The same situation arises if the single-letter word is a suffix such as "s." In languages, this difficulty can be avoided because single (or double) letter words may be recognized by the preceding and following space (#). That is, the entry in the table would be not "a," but "#a#." Extra entries "##¢a#" identify "A" when it begins a sentence. (The symbol ¢ indicates capitalization in the text of the following letter.)

This example shows another detail which must be accommodated; namely, that words may occur at the beginning of sentences, or, as is common in Russian or German, may be capitalized within the sentence when used as a title, e.g., Doctor. Obviously, assigning entries for every word that might be capitalized is impossible. Capitalization is indicated in the incoming text by symbol (such as ¢), with value "space-cap," which then will be identified independently. The word sought then resides in the register in uncapitalized form.

Proper nouns, with initial letter capitalized, will be transliterated with the capitalization. Proper names, which happen to be capitalized common nouns, such as "White," thus would be capitalized but translated. To overcome this, entries with the capitalization can be made with the correct proper-name transliteration.

Principle of address modification

With addition of very little hardware, the above addressing scheme can be extended to provide a conditional sequence of events, which enormously increases the power of table look-up.

An additional scanning register (B-box), with a capacity of one or more characters, is placed in front of the Input Register. The entries now may be composed of three parts: the address x, the function $f(x)$, and a prefix ρ. A new instruction μ is introduced to replace τ, so that the entry is $x\mu\rho f(x)$. When μ is recognized, the end of address is recognized as it was with τ, $f(x)$ read out from memory, and x shifted out of the Input Reg-

ister, exactly as before, but two new events happen. The information ρ immediately following μ is read into the B-box, and the next search starts, not at the beginning of the Input Register, but at the B-box, and then proceeds normally through the Input Register. In this way the sequence of characters in the Input Register immediately following the address x just found is prefixed by ρ. We shall now give some examples of how this address modification can be used.

• *Long address*

The principle of longest-match was illustrated with the concept that the Input Register is semi-infinite. In practice, it need not be longer than the longest address in the memory. This is not very economical if long addresses are infrequent, as is usually the case. Addresses longer than the Input Register can easily be handled by the principle of address modification.

Consider a long address x. Split it into parts

$$x=y_1+y_2+\ldots.y_n$$

such that the number of characters in each y_i are to be parts of addresses of new entries in the memory. The entry for y_1 has the structure $y_1\mu\rho_1$ with no value $f(y_1)$, but which will prefix y_2. There are $n-2$ entries of the structure $\rho_{i-1}y_i\mu\rho_i$, and finally one which is normal, $\rho_{n-1}y_n\tau f(x)$, giving the value of the whole chain of y_i's. In this way indefinitely long addresses can be accommodated.

The sequence of events is that in attempting to find x, the longest match is only on y_1. This entry gives no output, but stuffs the prefix ρ_1 in front of the remaining contents of the Input Register. The next scan is then to find not $y_2y_3\ldots$, but $\rho_1y_2y_3\ldots$, and ρ_1y_2 is located, stuffing ρ_2 to give ρ_2y_3, et cetera, until finally $\rho_{n-1}y_n$ gives $f(x)$. Diagrammatically, the sequence of events[6] is given in Table 2.

Table 2

	B-box	*Input Register*	*Match*	*Output*
		$y_1y_2\ldots$		
1st search			y_1	none
state after 1st search	ρ_1	$y_2y_3\ldots$		
2nd search			ρ_1y_2	none
state after 2nd search	ρ_2	$y_3y\ldots$		
3rd search			ρ_2y_3	none
	⋮	⋮	⋮	
state after $n-1^{\text{th}}$ search	ρ_{n-1}	y_n		
n^{th} search			$\rho_{n-1}y_n$	$f(x)$

• *Transliteration*

Another example is the use of address modification in transliteration. We have already mentioned that if a match with a legitimate address cannot be found, the search is terminated by a break point. This allows a "match" on the first character. If now the entry for the break point, at z say, be $z\mu\rho_Tf_T(z)$, where $f_T(z)$ is the transliteration of z, the latter is sent to the output, and the symbol ρ_T is stuffed in front of the next character. If, in addition, we have a set of entries $\rho_Tz_i\mu\rho_Tf_T(z_i)$ for each character, z_i, the transliteration proceeds character by character through the word in the Input Register. Finally, ρ_T is stuffed in front of the symbol for "space" (#). A normal entry $\rho_T\#\tau\#$ gives an output, "space," and terminates the transliteration. It is to be noted that no programming is required. Transliteration is initiated by a break point, and automatically proceeds until a space occurs.

• *Syntactic clues*

The same technique can be used to transfer, in a forward direction, syntactical and semantic information. For example, in English most nouns have a simple plural ending, e.g., "electron-electrons." It would be wasteful to enter every plural form in the dictionary. Only the stem (which in English is merely the singular form) is entered. An entry $s\#\tau f(s)$ could be listed. Here $f(s)$ supplies the information that a plural ending must be supplied in the output language. However, in English verbs also have an ending s, e.g., "evaporate-evaporates." In this case $f(s)$ would have to mean "third person singular." In order to distinguish these two meanings of the endings, all nouns are stuffed with ρ_n, all verbs with ρ_v, and we have not one entry for s but two, $\rho_ns\tau f_n(s)$ and $\rho_vs\tau f_v(s)$, where $f_n(s)$ indicates the nominal plural form and $f_v(s)$ indicates the third-person-singular verbal form.

One can go further and modify the output value $f(x)$ by the previous address. More precisely, if a sequence xx' corresponds to two addresses giving $f(x)$ and $f(x')$, there may be a variety of functions $f_i(x')$ depending on the nature of $f(x)$. For example, if x is a verb stem and x' is an ending indicating past participle, then $f(x)$ is the English equivalent of x, say the verb stem "box" and $f(x')$ is normally the suffix "-ed." The combined output would be "boxed." But if $f(x)$ were say "drag," $f(x')$ now should be not "-ed" but "-ged" to give, not "draged," but "dragged." Stems requiring doubling of the last letter stuff ρ_k, there being one ρ_κ for each consonant κ occurring in English verb forms which require doubling before "-ed." There are correspondingly a series of entries $\rho_\kappa x'$ giving $f(\rho_\kappa x')=_\kappa ed$.

Similarly, plurals such as "economy-economies" can be correctly spelt.

• *Semantic clues*

The ρ-stuffing technique can be used to transfer semantic clues from one word to the next. Let two successive words be $A\#B$, and assume that B has two meanings

$f_1(B)$ and $f_2(B)$. Suppose the word A selects one meaning $f_1(B)$. Then, of course, we could have the pair $A\#B$ as an address with value $f(A)\#f_1(B)$. But it often happens that there is a class of words $\{A\}$ which select the one meaning for B. Instead of listing all pairs $A_i\#B$, for $A_i\varepsilon\{A\}$, we can stuff a symbol ρ_A after each A_i of the class. Once the A_i has been looked up, the next address is no longer B but $\rho_A B$. If now we have an entry $\rho_A B\tau f_1(B)$, we get the correct selection of meaning. Then if we can identify another class $\{C\}$ which determines the other meaning of B, all words $C_i\varepsilon\ \{C\}$ will have ρ_C, and there will be an entry $\rho_C B\tau f_2(B)$, giving the other meaning.

- *Relation to programming*

It is noteworthy that the sequence of prefixing with ρ's and successive searches goes on without any programming. Further elaborations of table look-up with address modification, showing how any kind of data processing can be executed, will be presented in later papers of this series. The assignment of ρ's and assembly of the table define the basic requirements of the processing, and correspond to the "decisive" elements of programming.[7] Administrative program steps are eliminated by the use of the single instruction, "look up in table."

Partial matching

A desirable feature in any addressing system is to recognize a partial match, under some rule. Take the example, used above, of the expression "singe the beard." If this is to be translated idiomatically, the whole phrase may be used as an address. But this implies all the inflected forms, "singes the beard," "singed the beard," etc. must be recognized as a group, too. In order to avoid the enumeration of all internally inflected forms (many of which are very unlikely ever to arise) we introduce another instruction symbol, ν. This may be used as a character anywhere in an address. The result of comparison of the binary digits of this symbol with the corresponding digits in the Input Register are ignored.

In this way, an entry like "singeν the beard" allows a match with any one-letter inflection of the phrase. Similarly, "singe$\nu\nu$ . . ." allows matches with two-letter inflections. In this example, the information contained in the skipped ending is lost as, on the completion of the match, the whole phrase is shifted out.

A subsequent paper in this series will elaborate the use of partial matching and show how the skipped information can be retrieved. Basically, skipping provides all the functions of an "associative memory." The fact that comparisons are made sequentially is irrelevant provided they are done at high speed.[8]

Error correction

In numerical data processing, the addresses in general are a selection from a finite set of numbers; e.g., an address "2456" implies that the complete set 0000 to 9999 is possible. An error of one bit will change an address to another possible one, and, if undetected, the computation proceeds with the erroneous data from the incorrect address.

In lexical processing the addresses are a selection from an infinite set which is everywhere dense. For example, if we assign numbers to the letters of the alphabet, the word "bee" is .255000 . . . and "beef" is .2556. Between these there is an indefinite number of other words and idiomatic phrases, e.g., "beech" .25538. . . . Because of this, the chances that an error in the input or in the search will convert an address to another which exists in the memory is negligible. The worst case is for short words. For example, an error changing an "o" to a "u" would convert "pot" to "put." By choosing the code assignment to the vowels, so that several bits would have to be changed to convert one into another, the probability of this happening can be reduced to well below the frequency of typographical and other errors.

A consequence of these observations is that error-correcting codes are not necessary in addressing large memories of lexical material, although they certainly can be included for extreme safety. Error detection is afforded by the "no match" signal. This institutes a repeat of the search. If the error was statistical, with low frequency, the second trial would be very likely to be successful. Three or more trials may have to be made if very high reliability is desired. Typographical errors in the input, however, remain a problem. A partial solution is to add entries with typical spelling errors.

Alternative methods of achieving high reliability are available when the memory is very large, on the basis of the principle that half of a very large number is still a very large number. This principle is exploited in several ways.

Local repetition

Some addresses are very important. For example, there should be at least one break point on each page, and these entries are repeated to ensure recognition. In this kind of memory addressing, repetition does not interfere with the search procedure. In order to include possible errors in scanning, the first entry in the dictionary is $\nu\mu\rho\tau()$. Here, the () indicates the machine would not recognize the character, and ν is the special character which allows a match to be admitted in all cases. This entry is duplicated for complete safety.

There are many other critical entries which are duplicated, whose details need not be described here.

Replication

A crude but effective way to increase reliability when storage capacity is available is to replicate the entire contents in a separate region of the memory. In a memory of the type envisaged here, replication is extremely economical, as only the additional amount of storage medium has to be supplied, but no hardware such as drivers and address counters. After a failure to match,

the search is repeated in a different region. An automatic way of doing this, requiring no additional hardware, is to have the "no match" signal (e.g., a break point) stuff a new symbol ρ_D, which is assigned a code larger (or smaller) than any character (for example zz). Then the contents of the Input Register are preceded by ρ_D, so that the next search automatically is routed to the end (or beginning) of the table, where the table is duplicated with ρ_D prefixing every address. Break points in the ρ_D region initiate transliteration, which is continued in the normal region.

In some applications, such as translation, the waste in capacity due to repetition of the whole dictionary can be compensated for in increased access time. The dictionary can be broken up into microglossaries for various disciplines (general, mathematics, chemistry, et cetera). Every word occurs at least twice. Failure to find a word initiates a search in successive microglossaries, beginning with the most likely.

Summary

The novel features of the table look-up procedures described here are:

1. The use of integral addressing, as in conventional dictionaries.

2. Random access to a point in a table "beyond" the contents of the Input Register being scanned.

3. Exhaustive backward search over a "page" or so to a breakpoint.

4. Optional skipping over part of the address.

5. Address modification by the immediately preceding look-up.

Features 1, 2, and 3 provide automatic isolation of sequences in the Input Register which in general cannot be predetermined. In the processing of languages this permits automatic dissection of compound words or identification of word groups as a whole. Features 3, 4, and 5 provide much more than work-for-word look-up. Indeed they provide means of executing all the "logic" of a stored program.

The use of tables provides more power than is available in a stored program because one can list functions whose values are not computable.

Programming is essentially eliminated. The decisive steps in the processing are described in the table entries. The administrative elements are reduced to the single operation of looking in a table.

Symbols

n = page number in Input Register
x = address
y_i = portion of an address
ζ = individual character in register
ν = skipping or masking character
ρ = address prefix
z = breakpoint
α = instruction indicating "begin address"
μ = instruction initiating prefixing
τ = instruction symbol indicating end of address and beginning of readout
$\#$ = space between words
ϕ = capitalization of following letter

References and footnotes

1. Table look-up processing is not practical unless random access is reasonably fast. This paper is based on actual operations with the AN/GSQ-16 Photostore, with 30 msec random access.
2. This method is, of course, the ancient one used in dictionaries, logarithm tables, et cetera. It was proposed for a computer memory by G. W. King, G. W. Brown and L. N. Ridenour, "Photographic Techniques for Information Storage," *Proc. IRE* **41,** 1421 (1953).
3. The above basic routine is described in U. S. Patent 2,843,841, G. W. King, E. L. Hughes, G. W. Brown and L. N. Ridenour (Application, Sept. 20, 1954).
4. W. W. Peterson, in his paper "Addressing for Random Access Storage," which appeared in the *IBM Journal* **1,** 130 (1957), has proposed some precomputation on x or part of x to make an absolute address, provided x can be isolated before look-up, i.e., without reference to the table, as for instance a catalog number.
5. Previous work on dictionary look-up has assumed that the linguistic units (presumed to be words) have been isolated beforehand in the text. See for example, M. Taube, "Automatic Dictionaries for Machine Translation," *Proc. IRE* **45,** 1020 (1957). Also R. W. Bemer, "Do it by the Numbers," *Communications of the ACM* **3,** No. 10, 530 (1960).
6. This scheme for handling long addresses by "ρ-stuffing" was introduced by I. Wieselman. See reports of Contract AF 30(602)1566 between USAF and International Telemeter Corp.
7. The distinction between "decisive" and "administrative" instructions in a program has been suggested by Andrew Gleason.
8. In the AN/GSA-16 equipment, the rate is three million bits per second.

Received August 19, 1960