The representation of verbal information as single numbers using APL functions can optimize main storage, peripheral storage, and data transmission.

Presented in tutorial form are the concepts of the encoding and decoding process. Applications including text processing and instructional systems are also discussed.

Encoding verbal information as unique numbers

by W. D. Hagamen, D. J. Linden, H. S. Long, and J. C. Weber

Were it not for some sort of encoding of the input and decoding of the output, neither people nor computers would be able to communicate in a meaningful manner.

As we attempt to write this paper, there are certain ideas and experiences we wish to share with you. Just how these are stored is not at all clear. However, it is apparent that they are not stored as sentences or paragraphs, and even groping for appropriate words involves a certain amount of seeking or trial and error behavior.

Thus the process in which we are now mutually engaged involves encoding and decoding of information. We, the authors, are taking information stored in one (unknown) form and representing it in another, the English language. As the reader, you take this output and translate it into a form in which you can efficiently process and store it. Which of these processes in this case should be called encoding, and which should be called decoding, is neither clear nor important. However, it is important to realize that the sender and receiver must have certain things in common. Not only must we use the same alphabet, vocabulary, and general rules of syntax, but we must to a large extent have had common experiences. To the extent that we draw upon data (whether this be words or experiences) that the reader does not share with us, then we are not understood.

As the reader is well aware, information is commonly stored within most computers as strings of 0s and 1s. Such binary strings become numbers because of their allocation into storage units of predetermined size (computer words). Thus when we speak of character strings within a computer, we really are talking about a series of numbers already encoded from their peripheral representation. It is assumed that everyone is familiar with the existence of this level of encoding, and we shall talk about characters as though they actually existed as such in the computer.

The encoding or representation of words as single numbers is the essence of a relatively new programming concept. In certain situations groups of words (now numbers) are further encoded so that phrases, sentences, and whole paragraphs are represented as single numbers.

This concept yields three tangible benefits:

- It packs the data and thus conserves storage space
- Response time and CPU time are reduced
- Natural language processing, at least on a conceptual level, is greatly facilitated

One cannot do anything he could not do before, given enough space, processing time, and programming patience. However, by simplifying the task, one finds that he does do things he did not think he could do before.

Each of these benefits derives chiefly from the fact that the result of encoding is the representation of strings of varying lengths as single elements or units. Each of these units (single numbers) occupies the same amount of space and can be addressed as a single entity. Such *unitizing* is perhaps the most important concept to be presented.

conservation of space

The levels of encoding we want to consider involve the creation of multiple hierarchies of lists. The starting point is the alphabet of characters. A number is assigned to each character according to its position in the alphabet we are using. Thus we might indicate the space by a 0, the letter A by a 1, B by a 2, and so on. The input string is then scanned and a series of numbers or a *map* of the string is produced, each element of which indicates the position of the character in the alphabet. Using the spaces (0s) as word delimiters, we then encode the successive groups of non-zero elements as single numbers. These numbers represent words.

If we want to encode groups of words as single numbers, we first create a new, higher-level list or alphabet in which each

word (now a number) is represented only once. After encoding the input string to the word level, we then find the position of each word in this numeric wordlist or second-order alphabet. Groups of elements of the resulting map are then represented as single numbers. Thus, for each level of encoding there must be an underlying numeric list or alphabet. However, just as there is no need to represent the letter A more than once in the alphabet, there is no need to store a given word, or grouping of words, more than once in any of the supporting lists.

The same reasoning applies to each of the higher levels of encoding. Certain combinations of letters form patterns people recognize as words. Various combinations of words form patterns we call sentences. Sentences may be combined into paragraphs, and so forth.

As a result of encoding, each of these exists as a single unit or number which is a word pattern conveying concepts or ideas and, except at the highest level, is stored only once. Because there is no need to store a concept more than one way as long as we have rules whereby it can be restored in its variety, redundancy in storage is avoided. Thus, the efficiency of storage achieved is largely dependent on the amount of redundancy in the text. The more frequently an underlying unit is used, the greater the saving.

processing time

Encoding and decoding both require a certain amount of time. If all one wanted to do were to input data at one point and output it at another without modification or comparison, CPU time would be increased.

However, generally one wants to process and modify the data. Most manipulations are simplified and require fewer iterations when the strings (words, phrases, or paragraphs) are represented as single numbers. The greater and more complex the manipulations required, the greater the saving in processing time.

Consider such a problem as searching for the second "THE" and changing it to "THAT" in the following sentence:

THE MAN IS A PROGRAMMER OF THE COMPUTER

One way to approach this would be to write a program that would scan the text, character by character, utilizing a counter. Using the spaces as word delimiters, you would initialize your counter. You would examine the first character to see if it were a "T". If yes, you would determine if the next character were an "H", then an "E", then a space. If at any point in the search the answer were no, you would continue to loop until you encountered a space, reinitialize the counter and begin again. After

finding the first "THE", you would go on to find the second. If the text were long, such iterations would consume a lot of time.

Once you had finished you would have the positions in the string representing the word you wanted to replace. However, since the word you want to substitute contains more characters than the one you are replacing, there is not enough space in the string.

It is precisely because of such storage considerations that you would probably store your words in blocks, each equal in length to your longest word. Thus we might represent the string as follows, where we have used "#" to indicate a blank:

```
#######THE
#######MAN
########IS
#########A
PROGRAMMER
########OF
#######THE
##COMPUTER
```

This would considerably reduce the number of iterations necessary to find the word, since you would essentially compare each of what we have represented as rows of a matrix with the word you were trying to find. However, if the text contained one very long word, there would be a tremendous wastage of storage space.

By contrast, if each word were represented as a single, four-byte number, your program would simply have to find the second occurrence of that number, and replace it with the new number representing "THAT." You would then have the best of both situations—minimum iterations or CPU time, and minimum storage allocation.

We shall speak of three types of encoding:

ease of manipulation

- Representing each word as a single number, using the spaces as word delimiters
- Simple superencoding where groups of words, such as paragraphs, are represented as single numbers
- Selective superencoding where the text is selectively scanned for specific word groupings which are then encoded as single numbers.

A simple application of representing words as single numbers has already been given. It facilitates locating the words in a

string, matching the input with those words in storage, and word substitution. We have a text-editing program, for example, where if one knows a word has been misspelled one or more times, a single command will correct every instance of the misspelling.

Simple superencoding finds application in our computer-mediated tutorials. Here one deals with lists, the elements of which may be as large as paragraphs and include:

- The questions the author wants to ask the student
- The various combinations of key words the author defines which he hopes will extract the essential meaning from the student's answer
- Comments the author may want to make in response to a given student's answer
- A branch which then leads the student to the next question that should be presented

Each of these varied units exists as a single number. Therefore it becomes a relatively simple matter to store these interrelated elements so that the program can easily find the appropriate question to ask, the list of anticipated answers associated with each question, the corresponding comment the author may have defined, and the subsequent branch.

In selective superencoding, the text, already encoded to the single-word level, is scanned for the presence or location of certain key words or logical delimiters. This is somewhat analogous to the use of the spaces as word delimiters at the simpler level. Then, each of the delimiters as well as the groups of words between the delimiters are encoded into single numbers. Such *parsing* of the input comes closer to representing concepts as single numbers than does simple superencoding.

The techniques presented in this paper were developed to solve specific problems encountered by the authors in trying to store and process large amounts of verbal information in a 32K APL\360 workspace, and will be described in that context. The fact that all our applications are essentially conversational and require a response time consistent with human discourse, plus the fact that in our interpretive implementation one is heavily penalized for looping or iteration within an APL function, may help explain our preoccupation with these terms. Conservation of space is of equal importance, since the system we are using has no file capacity, and all the functions and data must be stored within the limited workspace. Aside from this, however, the reader need have no knowledge of the APL system, and the basic concepts should have applications in a variety of programming situations.

The text that follows is divided into two major parts—a formal description of the encoding and decoding algorithms in non-APL terms, and various applications of the techniques. The latter includes:

- A consideration of possible applications to the storage and transmittal of verbal data
- A text formatting and editing program
- A brief description of an authoring and teaching system the authors have developed called A Tutorial System (ATS)
- The application of these algorithms to an APL version of ELIZA[1]
- A proposed extension of ATS in which a logical analyzer will be interposed between the program and the author or student

Each of these applications has been included to illustrate specific points. The discussion of possible use in the storage and transmitting of data describes in greater detail the situations in which encoding does, and does not, conserve space. The text formatting and editing program illustrates a simple application of encoding to the single-word level. The authors' original need for developing a method of superencoding was to implement a system for authoring and supervising computer-mediated tutorials, and is the most fully documented phase of their work. Both ELIZA and the proposed extension of ATS illustrate the use of selective superencoding. It is hoped that in the variety, the individual reader will find something that is related to his own experiences and needs.[2]

There are two other situations in which the authors have used these encoding techniques. One is in the mechanical translation of languages where encoding most clearly facilitates the translation process when a group of words in one language must be substituted for a single word or another group of words in the other language. A prime example is idiomatic translation.

The second is in the area of encoding graphic or pictorial information. Just as people seldom analyze sentences letter by letter, we seldom analyze pictures dot by dot. The animate organism has the ability to extract and store patterns in some coded form that permits easy recall and manipulation, with minimal storage demands. One area where we have applied this technique is in a chess-playing program. When people play chess, they look for certain visual patterns or board positions. Our program encodes the total board pattern existing at any moment as a single number. This is used for comparison against the possible partial board positions it is seeking in determining its next move.

Neither the language translation nor the chess-playing programs will be described beyond this, because of lack of space for ade-

quate description, and because the authors hope for considerable further development of both.

The basic encoding and decoding processes

Encoding, within the context of this discussion, involves two rather simple processes. Decoding involves the inverse of these two operations.

Because one must understand these basic principles in order to follow the remaining discussion, they will be explained at the simplest level. The more computer-literate reader may simply scan or even skip this section.

encoding

The first step in encoding is mapping the positions of each element of one string in another. The map of the word FACE in the alphabet A through Z is the numeric vector 6 1 3 5. The same process can be applied between two numeric vectors. The positions of the numbers 6 1 3 5 in the vector 16 6 2 1 33 3 45 5 are given by the vector 2 4 6 8.

One way of representing the literal vector FACE as a single number is 6135, which is the value of the vector 6 1 3 5 evaluated in the base 10 number system. The algorithm for performing the evaluation of a mapping vector follows. If we let B stand for the new base, and B*N means B to the Nth power, the vector 6 1 3 5 becomes:

$$(6 \times B*3) + (1 \times B*2) + (3 \times B*1) + (5 \times B*0)$$

Given the alphabet:

A B C D E F G H I
1 2 3 4 5 6 7 8 9

we can obtain a unique base 10 value for the map of any word composed from this alphabet:

EDGE → 5 4 7 5 → 5475
HIDE → 8 9 4 5 → 8945

However, the base 10 will not serve beyond 9 characters, as:

BAKE → 2 1 11 5 → 2215

would be indistinguishable from:

BBAE → 2 2 1 5 → 2215

To obtain a unique value for any such mapping vector in a given

number system, the elements of the vector must all be representable as digits of that number system. In other words, the value of the new base must be at least one greater than the size of the alphabet or list to be mapped.

Because most of us are more accustomed to thinking in terms of base 10 numbers, let us first evaluate the vector 6 1 3 5 in the base 10 number system:

```
(6 × 10*3) + (1 × 10*2) + (3 × 10*1) + (5 × 10*0)
(6 × 1000) + (1 × 100) + (3 × 10) + (5 × 1)
6000 + 100 + 30 + 5
6135
```

The evaluation of the vector using base 129 is:

```
(6 × 129*3) + (1 × 129*2) + (3 × 129*1) + (5 × 129*0)
(6 × 2146689) + (1 × 16641) + (3 × 129) + (5 × 1)
12880134 + 16641 + 387 + 5
12897167
```

The evaluation of the vector using base 2000 is:

```
(6 × 2000*3) + (1 × 2000*2) + (3 × 2000*1) + (5 × 2000*0)
(6 × 8000000000) + (1 × 4000000) + (3 × 2000) + (5 × 1)
48000000000 + 4000000 + 6000 + 5
48004006005
```

significant digits

Using APL one can store very large and very small numbers (as many as 75 decimal positions) in exponential form. However, the number of significant digits that can be stored is only 16. This places a limitation on the number of digits that can be represented as a single number in a new base system. With base 10 this number is 16 digits. Thus, if we tried to convert 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 (17 digits) to a single base 10 number, 12345678912345678 would be rounded off to 1234567891234568 or 1.234567891234568E16 . On reconversion this would yield 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 8 and fail to serve its purpose.

The maximum number of base 129 digits that can be converted to a single base 129 number is eight; with a higher base, 2000 for example, it reduces to five.

decoding

Decoding, the inverse of encoding, may be described as the process of extracting successive remainders. Again we start by illustrating with base 10:

```
6135 ÷ 10 = 613    and a remainder of    5
613 ÷ 10 = 61      and a remainder of    3
```

$61 \div 10 = 6$ and a remainder of 1
the remainder of $6 \div 10$ is 6

Thus, we obtain the original vector: 6 1 3 5. If we use base 129:

$12897167 \div 129 = 99978$ and a remainder of 5
$99978 \div 129 = 775$ and a remainder of 3
$775 \div 129 = 6$ and a remainder of 1
the remainder of $6 \div 129$ is 6
6 1 3 5

With base 2000:

$48004006005 \div 2000 = 24002003$ and a remainder of 5
$24002003 \div 2000 = 12001$ and a remainder of 3
$12001 \div 2000 = 6$ and a remainder of 1
the remainder of $6 \div 2000$ is 6
6 1 3 5

Thus, no matter in what base we evaluate the vector 6 1 3 5, this algorithm returns the original vector.

The final step in the decoding process is a simple indexing or subscripting operation. Take the 6th, 1st, 3rd, and 5th elements of the alphabet to produce FACE.

Representing words as single numbers

Although the basic algorithms underlying the encoding and decoding are quite simple, when actually handling verbal input, certain decisions must be made. Some of these involve deciding what is a word; some are formatting considerations; and others arise as a result of the limitations on the number of significant digits that can be represented.

We call the APL function that does the basic encoding up to the level of single words IN and the inverse decoding function, OUT. The major steps involved in the functions IN and OUT are now described in general, non-APL, terms.

To start with, we define three short pieces of text which we will call TEXT1, TEXT2, and TEXT3. These samples of text are devoid of meaning, but illustrate the special points we want to discuss. We will then use these samples in dissecting the various stages of the encoding process.

```
      TEXT1
ONE TWO THREE FOUR FIVE
      TEXT2
ABCDEFGHIJKLMNOPQRSTUVWXYZ'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
      TEXT3
ONE, TWO,THREE , FOUR ,FIVE
```

The functions IN and OUT are dyadic in that each requires both a right and a left argument. The right argument of IN is a character string; its output is a numeric vector. The right argument of OUT is a numeric string representing the same level of encoding as the output of IN; its output is a character string.

words less than nine characters

In the case of both IN and OUT, the left argument is a binary switch (that is, either 0 or 1), whose purpose is to modify the manner in which the function operates upon its right argument. 0 OUT means that the entire right argument is decoded at once and appears as the literal output which can then be stored as a variable. 1 OUT means that the right argument is decoded only a line at a time—that is, the number of words that will evenly fit on a line whose width is set by the user. The latter greatly improves the apparent response time of the function, since while one line is being printed, the next is being decoded.

The left argument of IN determines whether a distinction will be made between upper versus lower case, or underlined versus non-underlined characters. This feature is used in the application, A Tutorial System, in which the encoding and decoding functions play a vital role. Where it is desired to encode a piece of text for fidelity of subsequent reproduction, we use 1 IN; but where our purpose is to compare a word stored in the computer with input from the keyboard, we use 0 IN, which converts all upper case characters to their lower case equivalents. Thus, the words "Cat," "CAT," and "cat" are encoded equivalently for purposes of matching.

The first step in encoding words into single numbers is the mapping operation of finding the positions of all elements of the incoming character string in the alphabet. All spaces are represented in this map, MAP1, by zeros.

```
      TEXT1
ONE TWO THREE FOUR FIVE
      1 IN TEXT1
1   2   3   4   5
      MAP1
15   14   5   0   20   23   15   0   20   8   18   5   5
0   6   15   21   18   0   6   9   22   5
```

In the above example, disregard the explicit output of IN and focus only on the intermediate variable MAP1 where the character O is the 15th letter of the alphabet, N is the 14th, E is the 5th, and so forth.

Using zeros as word delimiters, the next logical step in encoding is the evaluation in the new base number system of each of the segments of MAP1 which represent single words. We use a variety of alphabets, depending on the typeball and application.

However, we never exceed 128 characters and thus we use base 129. If this conversion is done on MAP1 we obtain:

```
251426    335802    5555931320    13132476    13032746
```

where 251426 represents the conversion of 15 14 5; 335802 the conversion of 20 23 15, and so forth. This string of five numbers could be decoded, by the method previously described, to reproduce TEXT1.

word fragments

Since a maximum of only eight characters can be represented as a single number using base 129, we need a means of permitting longer words. Thus, we are forced to consider that the numbers resulting from this initial evaluation may potentially represent word fragments as well as complete words.

Although TEXT1 contains no words longer than eight characters, the function IN has to make provisions for this situation. This involves a further encoding. Base 2000 is used since we can evaluate up to five elements of MAP2 as a single number without exceeding the limitation of 16 significant digits.

The result of the initial base 129 evaluation is stored in a temporary, local variable or Temporary Word List (TEMPWL) called TEMPWL0. There is also a permanent, global variable or Word List (WL) called WL0 which, originally an empty vector, will eventually contain all the numbers that have been used in TEMPWL0.

TEMPWL0 is a one-to-one representation containing one element or number for every word in the text. Therefore, if the text contains a word more than once, TEMPWL0 will contain the corresponding numeric element an equal number of times. WL0, on the other hand, contains each number only once, and these appear in the order in which they were entered.

Although WL0 is a numeric vector, it is comparable to our alphabet since the elements comprising WL0 will be used to obtain a map, MAP2, for the next level of encoding. Just as there was no need to have the letter A appear more than once in the alphabet, there is no need to have the number 251426 occur more than once in WL0.

In the following sequence we show WL0 in its initial state followed by TEMPWL0. After the formation of TEMPWL0, WL0 is displayed followed by MAP2. Originally WL0 is an empty vector. Hence, nothing is displayed. TEMPWL0 is then compared to WL0 and any elements of TEMPWL0 not contained in WL0 are catenated to it. MAP2 results from finding the position of each element of TEMPWL0 in the updated version of WL0.

```
      WL0    (empty)
      TEMPWL0
251426    335802    5555931320    13132476    13032746
      WL0
251426    335802    5555931320    13132476    13032746
      MAP2
1 2 3 4 5
```

TEMPWL0 and the updated version of WL0 are identical in this instance because WL0 was empty and because TEMPWL0 contains no number more than once. In this situation TEMPWL0 was effectively substituted for WL0. That is, TEMPWL0 was catenated into the empty vector WL0. Both of these represent the character positions, MAP1, evaluated to base 129 in units determined by the occurrence of the delimiting zeros. MAP2 has an orderly appearance because the first element of TEMPWL0 corresponds to the first element of WL0, the second element of TEMPWL0 corresponds to the second element of WL0, and so forth.

The next step in the encoding process is the evaluation of elements of MAP2 in a new base. If we use base 2000 we can evaluate as many as five elements of MAP2 as a single number without exceeding the 16 significant digits.The purpose of the additional encoding is to combine word fragments into actual words. Since each word fragment or element of WL0 may be eight characters in length and we may further encode as many as five of these together, the maximum length of a word we can represent without hyphenation is 40 characters.

The result of this evaluation of the elements of MAP2 to base 2000 is the explicit output of IN which has already been shown. In this instance, MAP2 and the output of IN are identical (1 2 3 4 5). To understand why, the reader should recall the explanation of the conversion process. There is no combining of elements since no word in TEXT1 exceeded eight characters. The representation of a single element or digit in one base numbering system is unchanged when evaluated to a higher base because it is multiplied by the new base to the 0 power which is always 1.

If we use the output of IN as the right argument or input of OUT, we obtain the original text:

```
      0 OUT 1 IN TEXT1
ONE TWO THREE FOUR FIVE
```

Next we shall go through each of these steps with TEXT2, the first 63 characters of the alphabet we are using at the moment.

words longer than eight characters

```
      TEXT2
ABCDEFGHIJKLMNOPQRSTUVWXYZ'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
      1 IN TEXT2
96056032018010
```

```
      MAP1
1   2   3   4   5   6   7   8   9   10   11   12   13   14
15  16  17  18  19  20  21  22  23  24   25   26
27  28  29  30  31  32  33  34  35  36   37   38
39  40  41  42  43  44  45  46  47  48   49   50
51  52  53  54  55  56  57  58  59  60   61   62
63
      WL0
251426   335802   5555931320   13132476   13032746
      TEMPWL0
¯603792137497124   ¯5396684763830884   ¯1.018957739016464E16
¯1.4982470016498 4E16   1.977536264283216E16
      WL0
251426   335802   5555931320   13132476   13032746
¯603792137497124   ¯5396684763830884   ¯1.018957739016464E16
¯1.4982470016498 4E16   1.977536264283216E16
      MAP2
6 7 8 9 10
      0 OUT 1 IN TEXT2
ABCDEFGHIJKLMNOPQRSTUVWXYZ'0123456789ABC
```

Notice that when we encode and then decode TEXT2, only the first 40 of these characters are reproduced. This is because the longest word we permit without hyphenation is 40 characters. If we were to define TEXT4 as the same string hyphenated between the 40th and 41st characters, the reproduction would be faithful as shown below.

```
      TEXT4
ABCDEFGHIJKLMNOPQRSTUVWXYZ'0123456789ABC-DEFGHIJKLMNOPQRSTUVWXYZ
      0 OUT 1 IN TEXT4
ABCDEFGHIJKLMNOPQRSTUVWXYZ'0123456789ABC-DEFGHIJKLMNOPQRSTUVWXYZ
```

The primary reason for using TEXT2 is to illustrate how words with more than eight characters are handled. We need some means of determining which elements of WL0 represent word fragments and which represent words. That is, when do we encode elements of MAP2 together, and when do we let them stand alone? This question resolves itself into deciding when do we follow a word fragment by a space, and when do we follow it by another word fragment.

This question is a formatting problem. The fidelity between input and output should be as good as possible. Thus, we would not like CHARACTERS to become CHARACTE RS nor ONE TWO THREE to become ONETWOTHREE. The sign bit is used to accomplish this formatting. Numbers representing word fragments that do not complete a word are given a negative sign; those that do end a word or represent the entire word are given a positive sign.

MAP1, displayed above, has the ordered appearance it does because it results from comparing one string with itself. It contains no zeros since there are no spaces. Therefore, IN tries to treat it as a single word, but truncates it after the 40th character for the reasons previously explained.

Again, TEMPWL0 contains no elements already in WL0 and contains no numbers more than once. Therefore, it is simply catenated to the existing WL0 to form the new WL0. MAP2 contains the positions of all elements of TEMPWL0 in the updated WL0. The first four elements of TEMPWL0 are negative. The positions, MAP2, of each sequence of elements of TEMPWL0 ending in a positive number are evaluated together to form a single number in base 2000. This is the explicit output of IN (96056032018010). Any sequence of four consecutive negative elements must be followed by a positive number, because the truncation to 40 characters occurs before this stage.

In this example, MAP2 bears no resemblance to the output of IN since the five elements of MAP2 have been combined to form the single element of output of IN.

punctuation

Punctuation and special characters present an additional formatting problem. Words are not always followed by a space, since they may be followed by a punctuation mark or special character. TEXT3 illustrates the various possibilities. The same steps by way of illustration are again presented.

```
      TEXT3
ONE, TWO,THREE , FOUR ,FIVE
      1 IN TEXT3
1    11    2    12    3    13    4    14    5
      MAP1
15    14    5    64    0    20    23    15    64    20    8    18    5
5    0    64    0    6    15    21    18    0    64    6    9    22    5
      WL0
251426    335802    5555931320    13132476    13032746
¯603792137497124    ¯5396684763830884    ¯1.018957739016464E16
¯1.49824700164984E16    1.977536264283216E16
      TEMPWL0
251426    8256    335802    ¯8256    5555931320    64    13132476    ¯64
13032746
      WL0
251426    335802    5555931320    13132476    13032746
¯603792137497124    ¯5396684763830884    ¯1.018957739016464E16
¯1.49824700164984E16    1.977536264283216E16    8256    ¯8256    64
¯64
      MAP2
1    11    2    12    3    13    4    14    5
      0 OUT 1 IN TEXT3
ONE, TWO,THREE , FOUR ,FIVE
```

TEMPWL0 contains some elements that are already in WL0 and some that are not. Only the new elements become catenated onto WL0 to form the new WL0. MAP2 represents the position of each element of TEMPWL0 in the updated WL0. Again, MAP2 and the explicit output of IN are identical because none of the words in TEXT3 exceed eight characters. The new feature is the formatting related to the punctuation.

The comma is the 64th element in the alphabet we are using.

129 times 64 is 8256. The number representing the comma appears in four different forms because it appears in four different formatting contexts in TEXT3. If the comma is preceded by a space, the number representing it is its position in the alphabet; if it is preceded by another character, this number is multiplied by 129, the base value. If the comma is followed by a space, the number representing it is positive; otherwise it is negative. Thus the first comma becomes 8256; the second, ¯8256; the third, 64; and the fourth, ¯64. The same logic applies to any of the characters of our alphabet occupying a position greater than 63.

Multiplication by base 129 does not distort the significance of these numbers. 8256 divided by 129 is 64 with a remainder of 0. The 129 remainder of 64 is 64. Thus, all this does is introduce extra zeros into the numeric string. All zeros are eliminated before indexing.

Encoding groups of words as single numbers

The alphabets we use vary, depending on the typeball and application. They all include the upper case letters A–Z, the digits 0–9, either lower case a–z or underlined A–Z, plus punctuation, and special characters. In our applications, the alphabet never exceeds 128 characters, and is encoded to base 129. Thus it has an arbitrary, but predetermined, order and length.

WL0 represents word fragments. It consists of numbers resulting from the evaluation of elements of MAP1 (as many as eight at a time) as single base 129 integers. The same number never appears more than once in WL0. In this sense WL0 is comparable to an alphabet, the elements of which represent groups of characters, rather than single characters. It differs from an alphabet, however, in that its elements are neither arbitrary nor predetermined. Both the order and the length are determined by the evolution or history of the input.

Although WL0 is not predetermined, once any part of it has evolved, this order must be permanently stored. TEMPWL0, on the other hand, is a transient phenomenon. It is formed by the same rules as WL0 except that redundancies are permitted. MAP2 represents the positions of elements of TEMPWL0 in WL0. If the output of IN is stored, both TEMPWL0 and MAP2 can be recreated during the process of decoding. The output of IN consists of elements of MAP2 evaluated as single base 2000 numbers (as many as five at a time).

The output of IN is comparable to TEMPWL1 in that redundancies are retained. The next logical step is to create a permanent WL1 from TEMPWL1, the output of IN, just as we created WL0 from TEMPWL0. Again, if we use base 2000, we can encode up

to five elements at a time. This process of superencoding may be repeated as often as desired to form WL1, WL2, WL3, and so forth.

Recall that for each Word List, WL, there is a Temporary Word List, TEMPWL, and that the WLS must be permanently stored. Each one is comparable to an alphabet as WL0. By mapping elements of the permanent WL0, we produce a TEMPWL1. Similarly, by mapping elements of the permanent WL3, we must create a TEMPWL4, the output of the function that creates WL3. These superencoding functions are called W12, W13, and W14.

MAP1 represents the positions of each character of the literal input string in the alphabet. MAP2 contains the positions of each element of TEMPWL0 in WL0. We shall go on to define a MAP3 which represents the positions of each element of TEMPWL1 in WL1, a MAP4 for TEMPWL2 and WL2, and a MAP5 for TEMPWL3 and WL3. There is a need to store permanently only the result of the final level of encoding, plus the various underlying WLS and the alphabet; each of the TEMPWLS and MAPS can be reconstructed during the decoding process.

The explicit output of W14 is a vector of numbers each one of which could be decoded to form as many as 125 words each containing as many as 40 characters.

simple superencoding

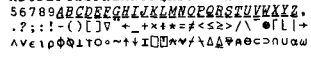
The simplest form of superencoding is encoding the maximum number of elements in successive steps according to the sequence of words in the input string regardless of context. Thus, if the input string were a paragraph of less than 126 words, it could be encoded to form a single number.

As our example text, TEXT5, let us use one of the paragraphs from this discussion. We shall use the alphabet shown in Figure 1, although any alphabet could be used.

TEXT5

ALTHOUGH WL0 IS NOT PREDETERMINED, ONCE ANY PART OF IT HAS EVOLVED, THIS ORDER MUST BE PERMANENTLY STORED. TEMPWL0, ON THE OTHER HAND, IS A TRANSIENT PHENOMENON. IT IS FORMED BY THE SAME RULES AS WL0 EXCEPT THAT REDUNDANCIES ARE PERMITTED. MAP2 REPRESENTS THE POSITIONS OF ELEMENTS OF TEMPWL0 IN WL0. IF THE OUTPUT OF IN IS STORED, BOTH TEMPWL0 AND MAP2 CAN BE RECREATED DURING THE PROCESS OF DECODING. THE OUTPUT OF IN CONSISTS OF ELEMENTS OF WL0 EVALUATED AS SINGLE BASE 2000 NUMBERS (AS MANY AS FIVE AT A TIME).

Figure 1 Alphabet used in simple superencoding examples



Simple superencoding of TEXT5 yields one number:

W14 1 IN TEXT5

16016012008005

We are already familiar with the function of 1 IN. W14 takes the output of IN and superencodes it to a level such that its output corresponds to TEMPWL4.

The output of 1 IN TEXT 5 is:

```
24    25    17    26    54028    11    29    30    31    21    32    33
34    11    35    36    37    38    78040    41    42    43    11
44    15    45    46    11    17    18    94048    98044    42    32
17    50    51    15    52    53    54    25    55    23    112057
58    118060    42    61    124063    15    128065    21    66    21
43    67    25    42    68    15    69    21    67    17    41    11
70    43    71    61    72    38    146060    74    15    75    21
76    42    15    69    21    67    77    21    66    21    25
156060    54    79    80    81    82    83    54    84    54    5    85
18    86    87    88
```

The various vectors that result are now presented. There are a lot of numbers, but we will try to point out the simple patterns and logic of the lists. In most cases the reader is advised to first read what is said immediately following a long list, and then look at the list itself only if he wishes to verify the point that was made.

*MAP*1

```
1    12    20    8    15    21    7    8    0    23    12    28    0
9    19    0    14    15    20    0    16    18    5    4    5    20
5    18    13    9    14    5    4    64    0    15    14    3    5    0
1    14    25    0    16    1    18    20    0    15    6    0    9
20    0    8    1    19    0    5    22    15    12    22    5    4
64    0    20    8    9    19    0    15    18    4    5    18    0
13    21    19    20    0    2    5    0    16    5    18    13    1
14    5    14    20    12    25    0    19    20    15    18    5    4
65    0    20    5    13    16    23    12    28    64    0    15    14
0    20    8    5    0    15    20    8    5    18    0    8    1    14
4    64    0    9    19    0    1    0    20    18    1    14    19    9
5    14    20    0    16    8    5    14    15    13    5    14    15
14    65    0    9    20    0    9    19    0    6    15    18    13
5    4    0    2    25    0    20    8    5    0    19    1    13    5
0    18    21    12    5    19    0    1    19    0    23    12    28
0    5    24    3    5    16    20    0    20    8    1    20    0    18
5    4    21    14    4    1    14    3    9    5    19    0    1    18
5    0    16    5    18    13    9    20    20    5    4    65    0
13    1    16    30    0    18    5    16    18    5    19    5    14
20    19    0    20    8    5    0    16    15    19    9    20    9
15    14    19    0    15    6    0    5    12    5    13    5    14
20    19    0    15    6    0    20    5    13    16    23    12    28
0    9    14    0    23    12    28    65    0    9    6    0    20    8
5    0    15    21    20    16    21    20    0    15    6    0    9
14    0    9    19    0    19    20    15    18    5    4    64    0
2    15    20    8    0    20    5    13    16    23    12    28    0
1    14    4    0    13    1    16    30    0    3    1    14    0    2
5    0    18    5    3    18    5    1    20    5    4    0    4    21
18    9    14    7    0    20    8    5    0    16    18    15    3    5
19    19    0    15    6    0    4    5    3    15    4    9    14    7
65    0    20    8    5    0    15    21    20    16    21    20    0
15    6    0    9    14    0    3    15    14    19    9    19    20
19    0    15    6    0    5    12    5    13    5    14    20    19
0    15    6    0    23    12    28    0    5    22    1    12    21
1    20    5    4    0    1    19    0    19    9    14    7    12    5
0    2    1    19    5    0    30    28    28    28    0    14    21
13    2    5    18    19    0    71    1    19    0    13    1    14
25    0    1    19    0    6    9    22    5    0    1    20    0    1
0    20    9    13    5    72    65
```

TEXT5 contains 514 characters including the spaces. Therefore, MAP1 contains 514 elements.

```
      WL0
251426    335802    5555931320    13132476    13032746
¯603792137497124    ¯5396684763830884    ¯1.018957739016464E16
¯1.49824700164984E16    1.977536264283216E16    8256    ¯8256
64    ¯64    333857    216476    1180    1    ¯9594963588678757    663
1941    1852994888265327    43067057
```

WL0, before the action of IN, already contains 23 numbers, because of what has been previously encoded.

```
      TEMPWL0
650483295410396    384319    1180    234929    ¯9594605499801819
3619551277    8256    32433701    18472    34366007    1941    1181
133276    23831455420117    8256    43068088    4192550844    28258889
263    ¯9535164825477206    334393    684308939473    8385
92347723241461    8256    1949    333857    4196910786    17191963
8256    1180    1    ¯1.197233461665866E16    20    ¯9548525553753818
1949    8385    1181    1180    218531010493    283    333857
40805414    5029892683    148    384319    185267932745    43067057
¯1.072360155085922E16    6590500    18968    ¯9535164842752490    4
8385    27925692    ¯1.072402937764029E16    2599    333857
¯9581282218169717    19    1941    3027818022632413    1941
92347723241461    1175    384319    8385    1167    333857
541704358001    1941    1175    1180    684308939473    8256    4545581
92347723241461    18451    27925692    50066    263
¯1.072356497767124E16    4    148746379081    333857    74379553902175
1941    2401021910014141    8385    333857    541704358001    1941
1175    1853031416310646    1941    3027818022632413    1941    384319
¯3073757624260823    4    148    681260458946    4312475    64870258
65269619739100    ¯71    148    27925429    148    13032746    149    1
43085231    ¯9288    65
      WL0
251426    335802    5555931320    13132476    13032746
¯603792137497124    ¯5396684763830884    ¯1.018957739016464E16
¯1.49824700164984E16    1.977536264283216E16    8256    ¯8256    64
¯64    333857    216476    1180    1    ¯9594963588678757    663    1941
1852994888265327    43067057    650483295410396    384319    234929
¯9594605499801819    3619551277    32433701    18472    34366007
1181    133276    23831455420117    43068088    4192550844    28258889
263    ¯9535164825477206    334393    684308939473    8385
92347723241461    1949    4196910786    17191963
¯1.197233461665866E16    20    ¯9548525553753818    218531010493    283
40805414    5029892683    148    185267932745    ¯1.072360155085922E16
6590500    18968    ¯9535164842752490    4    27925692
¯1.072402937764029E16    2599    ¯9581282218169717    19
3027818022632413    1175    1167    541704358001    4545581    18451
50066    ¯1.072356497767124E16    148746379081    74379553902175
2401021910014141    1853031416310646    ¯3073757624260823
681260458946    4312475    64870258    65269619739100    ¯71    27925429
149    43085231    ¯9288    65
```

TEMPWL0 contains some elements already in WL0, and contains some elements more than once. Thus the new WL0 is not a simple catenation of TEMPWL0 to the old WL0. The fact that there are 115 elements in TEMPWL0 and only 88 in the updated ver-

sion of WL0 attests to this overlap. Actually 43 elements were dropped because of repetition within TEMPWL0 and 7 were dropped because they already existed in WL0 (43 + 7 = 50; 115 − 50 = 65; 65 + 23 = 88).

MAP2

```
24  25  17  26  27  28  11  29  30  31  21  32
33  34  11  35  36  37  38  39  40  41  42  43
11  44  15  45  46  11  17  18  47  48  49  44
42  32  17  50  51  15  52  53  54  25  55  23
56  57  58  59  60  42  61  62  63  15  64  65
21  66  21  43  67  25  42  68  15  69  21  67
17  41  11  70  43  71  61  72  38  73  60  74
15  75  21  76  42  15  69  21  67  77  21  66
21  25  78  60  54  79  80  81  82  83  54  84
54  5   85  18  86  87  88
```

MAP2 has 115 elements, the same number as TEMPWL0, since it represents the positions or map of each element of TEMPWL0 in the new WL0.

MAP2 and the output of IN at first glance may look the same, but are not. The output of IN has 105 elements, 10 less than MAP2. If you examine the output of IN, you can quickly spot the numbers greater than 88. These represent those ten elements of MAP2 that have been combined in the evaluation to base 2000.

The number of negative elements in TEMPWL0 is 12. None of these are adjacent to each other. You might expect that these are the ones that are combined with the next positive element when evaluated to base 2000. Thus 115 minus 12 should equal the number of elements in the output of IN. However, it does not (115 − 12 = 103, not 105). This discrepancy of two is explained by the fact that two of the negative numbers, ¯71 and ¯9288, represent the punctuation marks (and) respectively. Although punctuation marks whose representation carries a minus sign are formatted by OUT without an intervening space, they are considered to be separate words. This is the reason the hyphenation in TEXT4 permitted the reproduction of the entire text.

We now consider the superencoding function W14 which calls W12 and W13 as subfunctions:

WL1 (empty)
TEMPWL1

```
24  25  17  26  54028  11  29  30  31  21  32  33
34  11  35  36  37  38  78040  41  42  43  11  44
15  45  46  11  17  18  94048  98044  42  32  17
50  51  15  52  53  54  25  55  23  112057  58
118060  42  61  124063  15  128065  21  66  21  43
67  25  42  68  15  69  21  67  17  41  11  70
43  71  61  72  38  146060  74  15  75  21  76
42  15  69  21  67  77  21  66  21  25  156060  54
79  80  81  82  83  54  84  54  5  85  18  86
87  88
```

WL1

24 25 17 26 54028 11 29 30 31 21 32 33
34 35 36 37 38 78040 41 42 43 44 15 45
46 18 94048 98044 50 51 52 53 54 55 23
112057 58 118060 61 124063 128065 66 67 68 69
70 71 72 146060 74 75 76 77 156060 79 80
81 82 83 84 5 85 86 87 88

MAP3

1 2 3 4 5 6 7 8 9 10 11 12 13 6
14 15 16 17 18 19 20 21 6 22 23 24
25 6 3 26 27 28 20 11 3 29 30 23 31
32 33 2 34 35 36 37 38 20 39 40 23
41 10 42 10 21 43 2 20 44 23 45 10
43 3 19 6 46 21 47 39 48 17 49 50 23
51 10 52 20 23 45 10 43 53 10 42 10
2 54 33 55 56 57 58 59 33 60 33 61
62 26 63 64 65

W12 builds up the permanent, global variable WL1. This begins as an empty vector since we have no previous examples of encoding beyond the IN level.

TEMPWL1 is identical to the output of IN in that it serves as the **input** to W14, W12. Those elements of TEMPWL1 not already in WL1, minus any redundancies in TEMPWL1 itself, become catenated to WL1. Since there are only 65 elements in WL1, there must have been 105-65 or 40 repetitions in TEMPWL1.
MAP3 represents the positions of each element of TEMPWL1 in the updated WL1. It also is the output of W12.

W13 performs an identical operation at the next higher level. MAP3 may be considered the input of W13.

WL2 **(empty)**

TEMPWL2

16016012008005 96056032018010 176096052012014 240128068036019
320168024044023 384200024006026 432224080022003 464240092062032
528016136070036 592304080078040 368328040084010 336344008040044
368360040086003 304048184042047 624384068098050 368408040104020
368360040086053 160336040004054 528440224114058 944264240066061
992208252128065

WL2

16016012008005 96056032018010 176096052012014 240128068036019
320168024044023 384200024006026 432224080022003 464240092062032
528016136070036 592304080078040 368328040084010 336344008040044
368360040086003 304048184042047 624384068098050 368408040104020
368360040086053 160336040004054 528440224114058 944264240066061
992208252128065

MAP4

1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21

Again, the corresponding Word List, WL2, starts as an empty vector. TEMPWL2 results from evaluating the elements of

MAP3, five at a time, to base 2000. Since there were 105 members of TEMPWL1, there are 21 elements in TEMPWL2.

We are now dealing with small enough lists that you can see that the new WL2 and TEMPWL2 are identical. Remembering that WL2 was empty to start with and, given the orderly enough arrangement of MAP3 to see that no consecutive groups of five elements are identical, this would necessarily be so.

MAP4 results from finding the positions of each member of TEMPWL2 in WL2. Since this constitutes mapping one list against itself, MAP4 consists of the sequence 1 2 3. . . .21. This is the output of W13.

```
      WL3 (empty)
      TEMPWL3
16016012008005     96056032018010     176096052028015     256136072038020
21
      WL3
16016012008005     96056032018010     176096052028015     256136072038020
21
      MAP5
1    2    3    4    5
```

The superencoding function, W14, then takes MAP4 and, encoding it five elements at a time to base 2000, forms TEMPWL3. TEMPWL3 has five elements, the last one being identical to the last element of MAP4 (the 5 remainder of 21 is 1).

TEMPWL3 and the second WL3 are identical for reasons already explained. The orderly appearance of MAP5 is also understandable.

Thus the simple superencoding of the paragraph, TEXT5, produces a single number:

```
      W14 1 IN TEXT5
16016012008005
```

The output of W14 is a single number resulting from the evaluation of the five elements of MAP5 to the base 2000.

The output of W14 and the first element of TEMPWL3 and TEMPWL2 are identical (16016012008005). This is because they each resulted from evaluating the sequence 1 2 3 4 5 to the base 2000.

selective superencoding

Selective superencoding involves parsing the input into its various component parts. This will be discussed later in the applications discussion since the rules employed for such parsing depend on the particular application. The input is first encoded to the single-word level. Then, a new APL function scans this en-

coded string for the presence of certain word groupings that divide it into its component parts. Each part is then selectively superencoded through one of the W functions, W12, W13, W14, just described.

decoding back to literal form

The process of decoding is the exact inverse of the processes just described. The basic decoding function, other than OUT is called U2.

U2 operates on the principle of extracting successive remainders which was explained in detail earlier. It assumes the base 2000. If we use the output of W14 1 IN TEXT5 as the input to U2, we obtain the following:

```
      U2 16016012008005
1    2    3    4    5
```

If we then use this to index WL3, we obtain the first five elements or, in this case, all of WL3.

```
      WL3[U2 16016012008005]
16016012008005    96056032018010    176096052028015    256136072038020
21
```

If we then, in turn, U2 this, we take it down another level and obtain MAP4.

```
      U2 WL3[U2 16016012008005]
1    2    3    4    5    6    7    8    9    10    11    12    13    14
15    16    17    18    19    20    21
```

This may be used to index WL2 and returns TEMPWL2, which in this case is identical to WL2.

```
      WL2[U2 WL3[U2 16016012008005]]
16016012008005    96056032018010    176096052012014    240128068036019
320168024044023    384200024006026    432224080022003    464240092062032
528016136070036    592304080078040    368328040084010    336344008040044
368360040086003    304048184042047    624384068098050    368408040104020
368360040086053    160336040004054    528440224114058    944264240066061
992208252128065
```

U2 TEMPWL2 yields MAP3. When MAP3 is used to index WL1, the result is TEMPWL1. This, identical to the output of IN, then becomes the input for the function OUT.

OUT itself calls U2 once as a subfunction to produce MAP2. When MAP2 is used to index WL0, the result is TEMPWL0. We cannot perform U2 TEMPWL0 because U2 assumes the base 2000 and we now need the base 129. This is simply a coding decision on our part. Execution is faster by making this an integral part of the function OUT than by giving the subfunction U2

a left argument. However, the logic is the same. Thus, decoding TEMPWL0 to base 129 produces MAP1 which is then used to index our alphabet. The final result is identical to TEXT5:

```
      1 OUT WL[U2 WL2[U2 WL3[U2 160160120080005]]]
ALTHOUGH WL0 IS NOT PREDETERMINED, ONCE ANY PART OF IT HAS
EVOLVED, THIS ORDER MUST BE PERMANENTLY STORED. TEMPWL0, ON
THE OTHER HAND, IS A TRANSIENT PHENOMENON. IT IS FORMED BY
THE SAME RULES AS WL0 EXCEPT THAT REDUNDANCIES ARE
PERMITTED. MAP2 REPRESENTS THE POSITIONS OF ELEMENTS OF
TEMPWL0 IN WL0. IF THE OUTPUT OF IN IS STORED, BOTH TEMPWL0
AND MAP2 CAN BE RECREATED DURING THE PROCESS OF DECODING.
THE OUTPUT OF IN CONSISTS OF ELEMENTS OF WL0 EVALUATED AS
SINGLE BASE 2000 NUMBERS (AS MANY AS FIVE AT A TIME).
```

Some applications

The function IN can encode words, each containing as many as 40 characters, into single numbers. W12 can represent in one single number as many as 5 words; W13, as many as 25 words; and W14, as many as 125 words at a time. This is as far as we have gone in our own applications. However, this cut-off point is arbitrary. A W17 will permit representation of as many as 15,-625 words as a single number which is the size of some books. A W20 will encode 1,953,125 words as one number.

If one could represent an entire book as a single number, one should consider the implications this has in relieving data transmission telephone line loads or in reducing the physical requirements for the storage of data.

In the introduction we suggested that human discourse involves encoding and decoding, and that the sender and recipient must have certain data and experiences in common. We also made the points that our alphabets were arbitrary and predetermined, but that the wordlists reflected the evolution or historical sequence of the input. Both the sender and receiver use the same alphabet. This is true of any form of data transmission, but it takes us only to the word-fragment level. The further encoding of groups of word fragments involves the creation of WL0. Both parties would also have to have this "experience" in common. If we were to go to W14, they would need WL0, WL1, WL2, and WL3.

Thus, it would seem apparent that representation of text as a single 16-digit number is misleading, since it would have to be accompanied by each of its underlying wordlists. However, there are data on word frequency which might ease this restriction somewhat.

Meier,[3] in counting eleven million words of German text which included a vocabulary of 258,000 different words, found that

200 of these words accounted for 54 percent of the total text. If it were assumed that the encoder and decoder shared this basic vocabulary, as well as the alphabet, the burden would be significantly reduced. A basic vocabulary of 1000 words would represent 69 percent of the text. In other words, approximately 70 percent of WL0 and WL1 would already exist, and only those words not included would have to be stored with the data.

The amount of space that is saved is a function of the redundancy of the text. It may help to illustrate this by first picking two extremes. If we were to choose two pieces of text, each consisting of 15,625 words with each word being exactly 40 characters in length, this would cost 640,624 bytes of storage without encoding for each piece of text. If in the first instance each word were different, then encoding to a single number would cost 781,248 bytes, or an efficiency of 82 percent. On the other hand, if each word were identical, encoding to a single number would cost only 96 bytes, or an efficiency of 667,317 percent.

We have performed measurements on a manuscript stored in our workspace consisting of 3,420 total words in the text and a vocabulary of 906 words. WL0 was 958 and WL1 was 906. This ratio of 1.1 to 1.0 between the two is relatively constant in our applications. The average word length, including the space, was 7.76 characters.

When stored in literal form, this text occupied 26,540 bytes. When stored as the output of IN, with WL0 also retained, it cost 21,340 bytes. This then was 1.24 times as efficient a form of storage. However, when stored as the output of W12, with WL0 and WL1 retained, it cost only 16,760 bytes—that is, 1.58 times as efficient. If we took it up through W17 which resulted in a single number plus WL0, WL1, WL2, WL3, WL4, WL5, and WL6, it required 18,152 bytes—reducing the efficiency to a factor of 1.46.

With our particular method of encoding we gain efficiency through the W12 level, then begin to lose efficiency. This is because along with IN one has to store WL0 which consists of 8-byte numbers, plus the output of IN which is only 4 bytes per number. However, the number is equal to the number of words of text. When we go one level higher we have WL0, plus WL1 consisting of four-byte numbers, plus the output of W12. The output of W12 consists of eight-byte numbers, but there are only one-fifth as many elements as words of text. Also, the length of WL1 is also usually much less than the number of words of text depending on the amount of redundancy.

The amount of this redundancy in our example is 3420 to 906 or 3.77. In Meier's much larger text, the ratio is 10,910,777 to

258,173, or 42.26. If we use his ratio in our data—that is, if we multiply our WL0 and WL1 by .09, then our efficiency through W12 is 4.01 instead of 1.58.

The reason WL1 and the output of IN consist of four-byte numbers is because we have yet to encounter a word of more than 24 characters. If we did, then this would introduce an eight-byte number and thus convert the whole vector into eight-byte elements.

It should be remembered that our goals for a terminal-based conversational system involved not only conservation of space, but also ease of manipulation and rapid response time. The latter two factors are enhanced when we encode a greater number of elements at a time. When working in a 32K workspace, it might be to our advantage to move up to six words at a time. This would limit the length of each wordlist to 645 (rather than 2352) which we seldom exceed in our tutorials.

However, if one were primarily concerned with conservation of space and relatively unconcerned with response time, he would do better with selective superencoding where certain patterns of words not necessarily adjacent to one another were sought out, or with the reducing of the number of words encoded at a time to two. The latter would not only have the advantage of keeping the elements to four-byte numbers, but a considerable redundancy might occur in the couplets that were produced. This redundancy is the greatest single space-conserving factor.

text editing

The authors have a text formatting and editing program written in APL. It incorporates the usual features of permitting the typist to enter the text casually, and then allowing the specification of the width, length per page, centering, various forms and amounts of indentation, tabulation, and so forth. Right margin justification is also a feature and variable spacing is selectively random in that preference is given to inserting the extra space following certain types of punctuation.

There are two reasons for discussing this application within the context of this paper. Despite the foregoing discussion, the text is stored in literal form and is encoded only a line at a time for editing purposes. Secondly, the editing facilities illustrate one of the basic advantages of simple encoding.

The basic reason for storing such text in literal form is that we do not feel we would save enough storage space to pay for the cost in response time entailed by the encoding and decoding. As a result of the preceding calculations, in the future we may investigate this further. There are several other advantages in storing the text in numeric form, such as being able to replace every in-

stance of a misspelled word with a single command, or being able to find a piece of text without knowing its exact location. These are discussed further in the tutorial application that follows.

The typist initiates editing by calling for a known paragraph. He is then asked which line he wants to edit. The editing options provided are:

- Cutting after a word (erasing either the entire line or that portion that follows the given word)
- Inserting one or more words into the line
- Erasing a word or words
- Replacing any word or series of words with any number of other words

The line to be edited is the only existing text that is encoded. Any insertions are subsequently encoded. It makes little difference to the program whether it is a single word that is to be erased or replaced, or a number of words, since they exist as single numbers. Similarly, it does not matter whether a single word is to be replaced by another word of the same or different length, or by a series of words, since in any case it involves replacement of one number by another. Erasure involves the insertion of 0s. If a word in question occurs more than once, the typist is asked which occurrence of the word should be erased, replaced, followed by an insertion, and so forth. After a line has been edited, all the encoded wordlists are destroyed.

A Tutorial System

A Tutorial System (ATS) provided the initial impetus for developing the concept of superencoding verbal data, and perhaps best illustrates the interweaving of the conservation of space with ease of manipulation. The details of the system are documented elsewhere.[4-6] We shall first briefly describe some of the highlights of the system to give the reader some idea of the magnitude of the problem, and then show how encoding aids in solving it. In evaluating this, the reader should remember that both the functions described and the data that comprise the content of each tutorial reside in a 32K APL\360 workspace without file capacity, and that we must maintain a response time consistent with human discourse. ATS is designed for experienced teachers who want to create sophisticated computer-mediated tutorials, but who neither know nor have the desire to know anything about computer programming. ATS comprises two main parts—an author-interrogation program and a tutorial supervisor with author feedback.

The author-interrogation program converses with the teacher in ordinary English, telling him what type of information it needs at any point and formatting his tutorial for him. It is often possible

for an author to create a tutorial at the terminal in less time than it would take to type it if it were already written out in longhand. This is the result of certain special features such as:

- a FIND command that locates a block of text containing one or more words the author may remember, obviating much of the need for producing time-consuming course listings
- a CHANGE command that changes every occurrence of a word or block of text
- a COPY command together with various text editing functions which permit him to copy and edit an already existing unit of text

The tutorial supervisor program runs the tutorial, interfaces with the student, and collects certain data for the author's use in updating his tutorial on the basis of student interaction. This supervisor program contains certain functions which provide each tutorial with an aura of intelligent behavior without the author's awareness of them. It not only recognizes when a student is asking a question, rather than answering one, based on the syntax of his response, but also chooses the most logical route along which to lead the student to reason through the answer to his own question—without anticipation or intervention on the part of the author. If a student asks an ambiguous question, either because he is too verbose or because he is not specific enough, the program recognizes this and attempts to keep him in normal conversation until his question can be understood. Because the program has the ability to keep track of the subject under discussion and, if necessary, to assign a hierarchy of meanings to the indefinite pronoun according to the context of the student's question, it permits the students to use pronouns whose meaning is indefinite unless considered in the context in which they are used. Thus when a student asks "What does that mean?" or "What is its action?", the meaning of the pronoun is updated not only on the basis of his location in the program, but also on the basis of what he himself has said. The supervisor program also handles words which would negate the meaning of an otherwise correct answer. It does not accept words that are outside the domain of the discussion, thereby shielding the author from many of the normal pitfalls of key-word analysis.

The supervisor program not only provides a record of the student's route through the tutorial, but provides the following four types of information verbatim and in the context in which they occurred:

- Any words used by the student that the program did not understand
- Any questions the student asked that the program could not answer

- Every answer a student gave that was not anticipated by the author
- Any comments the student made during the course of the discussion

The data comprising any ATS tutorial exists as a series of named numeric vectors, most of which represent literal data encoded to various levels. This data is created by the author-interrogation program, and it is with this data that the students interact. This data, or information regarding his particular subject, is all that the author provides. The APL functions are common to every tutorial and are independent of the data.

We now single out four types of data to illustrate how these are interrelated, and the role that simple superencoding plays in their manipulation.

Each tutorial must contain a series of questions which the program is prepared to ask the student. We shall call this list the Vector of Author Questions (VAQ). Each element of VAQ consists of a single number which is the output of W14—that is, the number can represent as many as 125 words of text.

Associated with each element of VAQ there are several other numbers which, in order to conserve space, we store in vector form by binary compression but which are more easily conceived and described as a series of matrices. We call these matrices the Matrix of Anticipated Answers (MAA), the Matrix of Author Responses (MAR), and the Matrix of Branches (MBR).

For each question or element of VAQ the author provides, he is asked to supply up to 25 key words which he feels will define the essential meaning of each of the various answers he anticipates a student may give. The key words representing each of these anticipated answers are encoded into a single number via W13. Within this framework he is given many additional options such as treating certain phrases as single words and specifying whether the order or sequence between words and phrases is important. He is then asked whether there are any equivalent or synonymous answers he would anticipate. Equivalent answers are those that the author would respond to in an identical manner. As many as five such equivalent answers are encoded one level further, corresponding to the output of W14, and these numbers then become the elements of MAA.

For any given question, the author may provide as many anticipated answers as he desires. The number of rows in MAA corresponds to the number of questions or elements of VAQ and the number of columns corresponds to the largest number of anticipated answers he provides for any question. It is precisely be-

cause he may have 30 anticipated answers for one question and only 1 for another that we actually represent this in vector, rather than matrix form. Empty boxes in the matrix cost as much as full ones, but we shall continue with the description as though this were the way the storage actually occurs. The last column of MAA may be conceived as containing all zeros, representing the unanticipated answers to each question. No matter how many answers an author provides, a student may come up with one he did not anticipate and thus fail to obtain a match. We call this the unanticipated answer.

For each anticipated answer he provides as well as for the unanticipated answer, the author is given the option of providing a comment which will be shown to the student before being given another question. These are stored as single numbers, the output of W14, in MAR. The dimensions of MAR are identical to MAA, and the absence of a response is represented by a zero.

After a student has been asked a question and his answer has been analyzed, he is then presented with the author's comment, if any, and an appropriate branch is chosen. Thus MBR consists of a matrix of numbers corresponding to the location or index of the appropriate question in VAQ. MBR has the same dimensions as MAA and MAR.

Therefore, a question is selected on the basis of a number which indicates the position of the question in the vector VAQ. This element of VAQ is decoded and presented to the student as his question. The student's answer is then compared against the elements of the corresponding row of MAA. The particular match, or lack of a match, specifies the intersection of the row with a certain column of MAR. This number is then decoded and presented to the student as the author's comment (a zero decodes as an empty line), and the corresponding box in MBR determines what the student will be asked next.

This brief description illustrates the basic contribution of encoding to the compactness of storage and retrievability of the data. Each question, response, and anticipated answer (as many as five equivalent answers) exist as single numbers, each of which requires the same amount of storage allocation and is easily indexed by its simple position in the vector. If there are 200 questions, the largest list we have to search contains 200 numbers.

Consider the alternatives. If one question contained 125 words (as many as 5125 characters), either an equal 5125 bytes would have to be assigned to every other question or some means of subdividing the text into questions, other than numeric indexing, would have to be devised. The reader should see the implications without the presentation of a detailed numerical analysis.

There tends to be more redundancy in words in a tutorial than in a manuscript. Thus, in a typical tutorial we get more than ten times the reduction in storage requirements by encoding to the W12 level, compared with the storing of the data in literal form with only a space separating the words. However, this represents only a part of the total compression actually achieved. One must also consider the storage space that would be consumed by the storing of units of text in blocks, each element of which is equal to the size of the largest unit, were it not for the representation of these units as single numbers. By direct comparison of identical tutorials written in ATS and certain other authoring languages, we have found that the actual reduction is often in excess of thirty-fold.

simulation of human discourse

Perhaps the best known attempt to simulate human discourse is Joseph Weizenbaum's original DOCTOR script, ELIZA. This was written in MAD-Slip, a list-processing language, for the IBM 7094.[1] We have written a literal translation of his program in APL\360. Discussing it here allows us to illustrate the use of selective superencoding.

The ELIZA program has a list of key words and each key word is given a relative rank or preference. The program scans the input and selects the key word with the highest rank. If no key word is found, it goes, as described by Weizenbaum, to its "memory" and may request more discussion regarding something the "patient" said earlier. If its "memory" is empty, it will respond with a relatively noncommittal remark such as PLEASE CONTINUE.

Certain routine functions such as the transformation of personal pronouns such as YOU = I or MY = YOUR are performed on the encoded input before any analysis.

The selected key word leads the program to a list containing one or more Decomposition Rules (DR). Thus, if the subject used the word MY, this would be transformed into YOUR. If YOUR was the highest ranking key word in the input, it would go to the associated DR. There happen to be two of these associated with the key word YOUR. In the DOCTOR script some key words have as many as 13 associated DRs. The way we print these out for the author is as follows:

```
YOUR
155 156 157 158
40
(0 YOUR 0 (*MOTHER MOM DAD FATHER SISTER BROTHER WIFE CHILDREN) 0)
```

```
YOUR
159 160 161
41
(0 YOUR 0)
```

Associated with every DR there are a number of Reassembly Rules (RR). In the listing for the author the key word is shown, followed by the locations or indices of the RR associated with this DR, the number of this DR in the list, and the DR itself.

DR 40, which heads the list associated with YOUR, may be interpreted as follows. Those words following the * are not key words. Rather they are the members of a set (in this case, family or relatives). The DR says that one or more of these words must occur in the input in order to match. If this match occurs, then the input is parsed into five parts:

- Any words that precede YOUR
- YOUR
- Any words that occur between YOUR and the first word of the list of relatives that may be present
- The family word selected
- Any words that may follow this
- If a match occurs on this DR, the program goes to the list of RRs associated with the DR. There are an average of four RRs for each DR. The ones associated with this DR are shown below:

  RR 155 TELL ME MORE ABOUT YOUR FAMILY.
  RR 156 WHO ELSE IN YOUR FAMILY (5) ?
  RR 157 YOUR (4)
  RR 158 WHAT ELSE COMES TO MIND WHEN YOU THINK OF YOUR (4) ?

Once the program gets to this point, there is no further question of matching. It selects whatever RR is at the top of the list. That RR is then shifted to the bottom of the stack and will not be called again until all the others have been used.

Assume the actual input were:

BUT MY MEAN BROTHER IS A STUPID BULLY

This first is transformed to:

BUT YOUR MEAN BROTHER IS A STUPID BULLY

It then is parsed into five parts:

(1) BUT
(2) YOUR
(3) MEAN
(4) BROTHER
(5) IS A STUPID BULLY

If RR 155 were first, the program responds with:

TELL ME MORE ABOUT YOUR FAMILY

If RR 156 were selected, it would respond:

WHO ELSE IN YOUR FAMILY IS A STUPID BULLY?

Thus the fifth element of the parsed input becomes the second element of the output.

With RR 158 the response is:

WHAT ELSE COMES TO MIND WHEN YOU THINK OF YOUR BROTHER?

Here the fourth element of the parsed input becomes substituted as the second element of the output.

It should be pointed out that, failing to match on DR 40, any input containing MY (= YOUR) cannot fail to match on DR 41 (the word YOUR, preceded and followed by none or any number or words).

selective superencoding

The literal parts of the RRs exist as single numbers – the output of W13, which can decode into as many as 25 words. Thus in RR 156, WHO ELSE IN YOUR FAMILY would be one number, and the ? would be another.

Each of the five elements resulting from the parsing of the input are then selectively or individually encoded using W13. Therefore the input, in this case, is represented as five numbers. Using RR 156, the first element of the RR, the fifth element of the input, and the third element of the RR are then catenated and decoded to produce the literal response.

The important point to be made here about selective superencoding is that it is analogous to our encoding to the word level. Between single words there are natural word delimiters – spaces and punctuation. In this application, there are logical delimiters – specific words – that are the basis for the parsing and subsequent selective superencoding.

extension of A Tutorial System

In one sense, ATS is a very logical system. A completely debugged computer program that converses with both the author and student in English, it has a very formal set of rules it follows to cope with every contingency. However, it does not apply even the simplest rules of logic to the meaning of the input. Its intelligence is only an illusion. Thus, if an author says that two plus two equals four in one part of his tutorial, and two plus two

equals three in another part, the author-interrogation program will not recognize the contradiction.

Furthermore, the student may ask questions of his own and get into a dialogue resembling a debate, but there is no way for him to win or lose, except as predetermined by the author. If he disagrees with the author's conclusions, the student cannot apply his reasoning powers to prove the author wrong to the satisfaction of the program.

A third problem is that a good teacher may produce a poor tutorial because of inexperience or lack of patience with the medium. Therefore, aside from some basic intelligence and information written into the program to recognize what the student is doing, the tutorial is only as good as the information the author may provide.

Recently we have begun to correct these deficiencies in what will become a new version of ATS. It is important for the reader to realize the distinction between writing a special-purpose program in which the subject matter is known beforehand, and writing an author-interrogation program to produce tutorials on an unlimited range of topics. The fact that this fits into the latter category means that we cannot rigidly restrict the rules of syntax. The program does not use any form of matrix, but the techniques we want to describe are perhaps better understood by referring to Figure 2.

Figure 2 depicts a matrix of 13 rows and 8 columns. It is really two matrices – the upper one being 5 by 8 and the lower being 8 by 8. The headings for the columns are identical in the two matrices. On the left there is literal text, presented as such so you can read it, but actually stored as single numbers.

The words specifying the five rows of the top matrix are key words the author supplies with each of his questions. These essentially define the meaning of the question. The words that specify the eight rows of the lower matrix and the columns of both matrices are the key words the author defines to extract the meaning of a student's response. Since these eight entries specify the columns as well as the rows, they are shown both in their literal and encoded form, the latter only being shown above each column.

This representation in binary matrix form shows the relationship between the key words that are used to extract the meaning of a response and those that define the meaning of a question. Thus, from the upper matrix the reader can extract the fact that for the question: WHAT IS THE ACTION OF THE SUPERIOR RECTUS?, the author expects the student's answer to contain the key words

Figure 2 Relationship between key words in questions and responses

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
ACTION +SUPERIOR RECTUS+	1	0	0	1	1	0	0	0
TEST +SUPERIOR RECTUS+ (O)	0	1	0	1	0	0	0	0
+SUPERIOR RECTUS+ =INSERT	0	0	0	1	0	0	0	0
+SUPERIOR RECTUS+ =APPROACH	1	0	0	0	0	0	0	0
+SUPERIOR RECTUS+ EQUATOR	0	0	0	0	0	0	0	1
=MEDIAL (1)	0	1	0	0	0	0	0	0
=LATERAL (2)	1	0	0	0	0	0	0	0
=INFERIOR (3)	0	0	0	1	0	0	0	0
=SUPERIOR (4)	0	0	1	0	0	0	0	0
**=ROTAT =INTERNAL* (5)*	0	0	0	0	0	1	0	0
**=ROTAT =EXTERNAL* (6)*	0	0	0	0	1	0	0	0
=POSTERIOR (7)	0	0	0	0	0	0	0	1
=ANTERIOR (8)	0	0	0	0	0	0	1	0

=MEDIAL, =SUPERIOR, and *=ROTAT =INTERNAL*. The sequence of appearance of these three word groupings does not matter. From the lower matrix he can determine that none of these key words is contradictory.

There is a certain amount of special notation that we use in ATS which must be explained to the reader. The words enclosed between the plus signs (+), such as +SUPERIOR RECTUS+, are treated as though they were single words. That is, they must

occur in juxtaposition and in the prescribed sequence. The letter O means that the order of the key words that define the anticipated answer does matter. The equal sign (=) preceding any key word means that there are a number of other words or phrases that may be substituted for it. This representation is stored in a dictionary. Thus, =ROTAT may stand for ROTATE, ROTATES, or ROTATOR.

The asterisks (*) enclosing a group of words mean that they must be considered as a unit in analyzing the key word components. This is one of only two additional pieces of information that the author-interrogation program needs that it does not already receive in the current version of ATS. The key words are still stored as they were before. Anticipated answers are selectively encoded through W13; then as many as five equivalent anticipated answers are further encoded through W14. However, for the additional analyses the program will perform, it must know what the individual units of each anticipated answer are. The program will assume that each key word represents such a unit unless (1) two or more words are enclosed between plus signs which indicates that the words must occur in that sequence with nothing intervening, or (2) they are enclosed between asterisks which implies nothing about juxtaposition or sequence, but that the author considers that grouping to constitute a unit.

The other information the program requires is which elements making up the total list of key word units are mutually contradictory. The author is interrogated regarding this before his tutorial is finished.

The reader is reminded that for every anticipated answer the author provides, as well as for the unanticipated answer or null response, he must supply a branch pointing to the question the student should be asked next. The author may also insert a response or comment between the student's answer and the next question.

In the present version of ATS if the student were asked WHAT IS THE ACTION OF THE SUPERIOR RECTUS? and the author had provided only the single anticipated answer indicated (MEDIAL, SUPERIOR, INTERNAL ROTATION), then an answer such as IT MOVES THE EYE MEDIALLY AND SUPERIORLY would not match and the student would take the branch corresponding to the unanticipated answer. In the version we are describing he would be told: MEDIALLY AND SUPERIORLY IS CORRECT, BUT INCOMPLETE. On first occurrence of this, he is given another chance. If it recurs, he is taken to the branch corresponding to an unanticipated answer. In other words, if the author fails to anticipate each possible answer, the student still is treated selectively as long as the information is present. By using the asterisks the

program knows that this group of words constitutes a single element of the anticipated answer, and thus, a single entry in our hypothetical matrix.

If the student were to answer the same question with: IT MOVES THE EYE MEDIALLY AND LATERALLY, the program says: MEDIALLY AND LATERALLY ARE CONTRADICTORY. This is the meaning of the 1's in the lower matrix in Figure 2.

If the student answers the question, HOW WOULD YOU TEST THE SUPERIOR RECTUS? with the response, LOOK SUPERIORLY, THEN LATERALLY, the program responds with YOUR SEQUENCE IS WRONG. This information is obtained from the (O) associated with the author's AQ. No conflict arises here because, if you recall, we do not actually employ a matrix. The reason for this is that we already have this information encoded in our various vectors, and a glance at the matrix shows the large percentage of zeros or meaningless data.

If the author contradicts himself by asking the same question in another place in the tutorial and specifying a different anticipated answer, the program does not accept this entry until he resolves the discrepancy. For example, the action of the superior rectus muscle varies depending on the starting position of the eye. Thus, in this example, the author would have to go back and redefine the key words or subject for the original question to include something such as STRAIGHT AHEAD OR MEDIALLY to define the starting position, before he could enter his second version. This, in our opinion, is a significant monitor on the author's activity. It is unlikely that an author would not know this, but it is likely that he might overlook it.

A second place where the author might contradict himself is in lack of agreement between his response or comment and the anticipated answer he specified. Thus, the author's responses are treated by the new version of the author-interrogation program the same as a student answer, and an author's comment or response to this question that said: YES, THE SUPERIOR RECTUS MOVES THE EYE MEDIALLY, SUPERIORLY, AND ROTATES IT EXTERNALLY, is likewise rejected until the author resolves the discrepancy.

Thus we have made a start towards correcting two of the deficiencies we cited at the outset—being able to provide the student with more individualized instruction than the author might have the patience to provide and being able to prevent some of the self-contradictions the author might introduce.

We also have the technical facility to give the student the ability to debate. However, it is impossible for the student to debate on

the basis of the author's failure to see the full implications of something he has said because the program automatically fills in the missing pieces. Therefore, an argument such as: if A is true, and B is true, therefore C must be true does not arise because the program already knows that C is true and would have said so. However, the situation could arise where the student either disagrees with the factual information the program has, or has additional information that supersedes it.

In such a situation we can give the student the ability to act temporarily as an author and alter the information in the matrix. There are two restrictions. The changes he makes are deleted when he exits from the program. Also, the fact that he is making a change is flagged and any response based on this change is qualified by: IF YOUR INFORMATION IS CORRECT. The latter reservation is often made in discussions when one is confronted with new information that he has not personally verified. However, we have learned from tests of an experimental variation of ATS which permitted the student to define new words that most students have little patience for this sort of exercise.

Concluding remarks

The technique of encoding verbal information as unique numbers arose from the authors' need for the capability to store and process large quantities of data in a 32K APL workspace. The algorithms, programmed in APL\360, support several levels or hierarchies of encoding and decoding where associated with each level is its alphabet or wordlist. The first level results in the representation of words as single numbers. Higher levels provide single numbers which can represent phrases, sentences, or paragraphs. Because the encoding process has no finite limit, the implication is that very large quantities of information, such as the amount contained in books or data files, can be represented as single numbers. Perhaps one of the greatest implications for the future is the concept of data compaction where main storage, peripheral storage, and data transmission are optimized.

ACKNOWLEDGMENT

This research was supported by Grant No. 50-68 from the National Fund For Medical Education and Grant No. MH01386 from the National Institutes of Mental Health, U.S. Public Health Service. The computing facilities of the Integrated Data System Laboratory and the IBM Systems Research Institute were made available to the authors as Research Fellows of the Institute. In particular, the authors wish to acknowledge the assistance of S. W. Dunwell, Field Engineering Division, and Dr. E. S. Kopley, Data Processing Division.

CITED REFERENCES

1. J. Weizenbaum, "ELIZA—a computer program for the study of natural language communication between man and machine," *Communications of the ACM* **9**, No. 1, 36-45 (1966).

2. Further information on the encoding and decoding algorithms and on the applications discussed in this paper may be obtained from Dr. J. C. Weber, Cornell University Medical College, Department of Anatomy, 1300 York Avenue, New York, New York 10021.
3. H. Meier, "Deutsche Sprachstatistik," Hildesheim: G. Olms, (1964).
4. W. D. Hagamen, J. C. Weber, D. J. Linden, and S. S. Murphy, *ATS (Version I) Author's Manual*, may be obtained from the authors at Cornell University Medical College, New York, New York, (1971).
5. W. D. Hagamen, J. C. Weber, D. J. Linden, and S. S. Murphy, *A Tutorial System (ATS): Concepts and Facilities Manual*, may be obtained from the authors at Cornell University Medical College, New York, New York, (1971).
6. J. C. Weber and W. D. Hagamen, "A new system for computer mediated tutorials in medical education," *The Journal of Medical Education*, **47**, 637-644 (1972).