The Law and Computers

by

B. C. BROSNAHAN

PREFACE

The following article is based on a year's research by the writer on this topic in 1969. It is not intended to be comprehensive, but on the other hand detail has not been avoided. The writer has tried not to assume any great knowledge of the subject on the part of the reader. The article is intended for lawyers and law students who might know little or nothing about computers.

I. GENERAL INTRODUCTION

The heading "The Law and Computers" might awake in the minds of some the idea that this article will discuss the prospects of a computer conducting the trials of criminals, or deciding disputes presented to it. With this idea, there is probably also the assumption that some day a computer will be able to perform both tasks with a certain and all-pervasive knowledge of the law, and a wisdom beyond that of mere humans. However, this would be a misconception of the article's subject matter; and it may be said, too, that the prospects mentioned above are real only in science fiction.

A computer is only an electronic device, consisting of a highly complex combination of electrical circuits and components whose magnetic state can be altered. The computer's peculiarity lies in the fact that a pattern of electrical activity can be set up in its circuits to produce an end result, and that the particular pattern of activity can be specified at will. There must be someone to set up a pattern of activity initially; otherwise it will sit there, a useless maze of circuits, doing nothing even though the power is on.

The person who writes the instructions determining the particular pattern of electrical activity that is to occur in the computer is called a "programmer". The set of instructions given to a computer which, together, are sufficient to make it perform a certain operation, is called a "program"; this word is not to be confused with the word "programme". The computer is confined to the pattern created by the programmer. For example, the computer cannot of itself decide that it is required by the sense of the program to perform an addition, if the program only makes provision for subtractions to be performed. Therefore, any error or lack of foresight on the programmer's part will cause the computer to produce an incorrect result. In fact it can be said with justification that a computer is only as clever or as stupid as its programmer.

Now, if a computer is strictly confined within the logic of the program it is executing ("to execute" a program means in computing language "to perform" it, or "to carry it out"), then, no matter how many contingencies the program may be designed to cope with, a computer can never outdo humans and perform activities that cannot be analysed into logical patterns. Thus it would seem that the judicial function will remain the prerogative of human beings; for it would not be feasible to program a computer to evaluate the truthfulness of a witness's testimony, or take into account policy considerations in its decision, or exercise a power of discretion.

Nor is it envisaged that, some day, it will be possible for a person to go into a Government agency and request by means of a device linked into a centralised computer, the computer's opinion on his latest legal problem. For similar reasons to those given in the paragraph above, it would seem to be impossible to program the computer to isolate the issues in the problem confronting it.

It would appear, then, that the courts and practising lawyers are safe from the encroachments of the computer.

But what if it is envisaged that the person would request from the computer not an opinion, but merely information on the law bearing on his particular problem? This would definitely be possible. The computer could serve as a very useful device for the retrieval of information from legal documents such as statutes and the law reports. This operation is termed "data retrieval".

The operation the computer would perform in data retrieval would have certain similarities to that performed by a librarian when she sets out to find a book that has been requested. If someone were to request, for instance, a book on geology written by I. M. Hardistone (fictitious name), it could be expected that the librarian would understand the term "geology". But her understanding of the term would not be necessary for her to find the book, nor would she even have to know that I. M. Hardistone was an author's name. For her to be able to find the book, it would be sufficient if she knew

(a) that she has to look in an index for a word with the same letters, in the same order, as in "geology";

(b) that she has to check beside each occurrence of that word in the index for the group of letters that make up "I. M. Hardistone"; and

(c) that she has to use a code given in the index to locate the book in the library.

The fact that the human librarian does understand what she is about enables her to be of help to inquirers, and so we normally ensure that our librarians are fairly well educated. However, when a computer performing data retrieval acts on principles very similar to those followed by a librarian, it does so in a purely mechanical way. In fact, when we say that we made a request to the computer for information, and it duly gave us the information required, we are really quite mistaken if we regarded it as understanding us and giving us what seemed to it to be a sensible reply—that would be to speak of it in human terms.

A purely mechanistic description of its operations is all that is needed to explain how it acted. The person's request would trigger electronic switches, and the specific headings (in electro-magnetic patterns) on which data is required would be compared with electromagnetic patterns in the computer's data banks, such as disks and tapes. Whenever this physical comparison of magnetic pattern with magnetic pattern revealed that both were identical, further electronic switches would be triggered and that piece of data, along with its context, would be transmitted to the printer from the data bank. The printer, acting on the electrical impulses received, would select the correct lettering and print out the retrieved data.

It is from the aspect of data retrieval that this article will examine the uses of computers in the law.

II. THE PROBLEM CREATED BY THE SOURCES OF LAW

The employment of computers seems an obvious way of making the administration of law and the use made of the sources of law more effective than it is. To put this another way, if we take Roscoe Pound's view of law, namely that "law is social engineering",[1] the use of computers could markedly improve the efficacy of this social engineering.

Just as Caligula, a Roman emperor, made law-making a play exercise by putting his laws at the top of columns in the forum where

[1] Pound, R., *Philosophy of Law* (1953) 47.

none could read them, so too might modern conditions have the same effect. Because of the cumbersome nature of the vast mass of materials from which the law has to be collated, the credibility of the present-day legal system can but diminish if the law cannot be found.

The main sources of law are statutes and the decisions of judges as reported in the law reports; and all would agree that both are very difficult to use. What is worse is that the situation is not improving. Consider the situation New Zealand lawyers will be faced with in 2000 A.D. As well as having to try and find his way through the hundreds of volumes of law reports and statutes already existing, a lawyer graduating now will also by then have to cope with (the following figures are approximate)

120 volumes of Statutes (assuming there will continue to be only four volumes per annum);

30 volumes of New Zealand Law Reports (one volume per annum);

30 volumes of the Law Reports, Appeal Cases (one volume per annum);

90 volumes of the Law Reports, Queen's Bench Division (three volumes per annum);

30 volumes of the Law Reports, Chancery Division (one volume per annum);

30 volumes of the Law Reports, Probate Division (one volume per annum);

30 volumes of Volume 1, Weekly Law Reports, which is an official Law Report;

Numerous volumes of the various unofficial English Law reports;

120 volumes of the Commonwealth Law Reports (three volumes per annum);

160 volumes of the Dominion Law Reports (four volumes per annum).

The first seven above are particularly relevant to the needs of the New Zealand lawyer. In our courts probably as many English cases as New Zealand ones are cited. The last two, the Commonwealth Law Reports and the Dominion Law Reports, are little used because the burden is great enough already. Moreover, in addition to the above, there will be several hundred volumes of Australian State Reports.

The tremendous rate at which legislation is being produced now is a relatively modern development, and the systematic reporting of cases was only really under way by the middle of last century. Yet, while it is true that nine-tenths of human knowledge has undergone radical change since 1900, the law has not yet developed any effective tools for coping with the information explosion in the sources of law.

Indexes and other old-fashioned devices can never efficiently classify the law because penetration in any great depth into the subject matter of the materials is never really achieved. Moreover they are always out of date.

How can a computer help? The human brain has two important disadvantages. First, it has an inefficient memory system that allows only limited data to be ready for immediate recall without recourse to prompters. Secondly, the electrical impulses in the brain move quite slowly through the nerves. For certain applications, the computer can be regarded as being an artificial appendage of the human brain that overcomes the latter's disadvantages. It can store enormous amounts of data on disks and tapes, which it can quickly recover on request. Moreover, the electrical current travels in its circuits at the speed of light, i.e. 186,000 miles per second, as against only a few feet per second in the brain.

Obviously the increased speed of performance would be fantastic. For example, a simple subtraction that would take a person, say, five seconds, would be done by the IBM 1130 computer in twelve micro seconds (or twelve millionths of a second). It is the fact that it can work at such a speed that enables it to cope with its large data banks.

A project in the United States provides a good illustration of the way computers are already helping. In that project, all the Pennsylvanian Statutes have been stored in computer data banks, and considerable success is being attained in the retrieval of the provisions in statutes relevant to a particular question. Its use in this field would be of great help to the practising lawyer.

However, in his research last year, the writer concentrated on case law rather than statute law; though this is of little moment, as conclusions reached for one are by and large applicable to the other. But it is important that the mass of case law should be made more manageable.

With the development of organised reporting of cases last century, the doctrine of *stare decisis* took firm root and flourished. The key reasons for the existence of the doctrine, it may be said, are that the more heads that consider a problem, the better the solution is likely to be, and that consistency is of the essence of justice. But if this doctrine is to remain workable, and if the practice of consulting "persuasive" authorities is to be encouraged, then it must be practical for lawyers and courts to seek out the relevant earlier decisions. That they should do so will only be practical and feasible if the earlier decisions are readily accessible.

At present, it would be fair comment to say that the doctrine has lost its classical meaning, and is rapidly coming to mean that, of the cases that are theoretically binding on a court, only those cases that *happen to be found* are in fact binding on that court. The fact that the doctrine is having to accommodate a large element of chance could well bring it into complete disrepute.

III. THE COMPUTER ITSELF

So far the writer has tried to outline and delimit some of the possible ways computers could be used in relation to the law, and the problems they could rectify. It is now proposed to discuss the computer itself, as it is hoped that the discussion will give a better idea of the capabilities of computers.

The particular computer the writer worked with was called the IBM 1130. It is a moderately sized machine, but has the disadvantage of lacking very extensive data storage facilities. The essential feature it has in common with most other computers is that it has components that represent, or symbolise, numbers in binary notation.

Before it is explained how this representation of numbers is done, the term "binary notation" might require clarification. The way in which we ordinarily represent numbers, i.e. the ordinary *notation* for numbers, is called decimal notation: for example, two hundred and fifty written as '250' would be said to be written in decimal notation. We could indicate that '250' is a number in decimal notation by writing it as '250_{10}', the word description for which is 'two hundred and fifty to the base ten'.

Decimal notation means that each digit in '250', i.e. '2', '5' and '0', really represents a value multiplied by a power of ten. The power is '0' for the right-most digit in a number, and increases by one as we move each place to the left: for example, in '129800_{10}' the digit '9' is in the fourth place from the right and therefore represents 9×10^3 or 9000. So, in '250', '0' represents 0 multiplied by 10^0, '5' represents 5 multiplied by 10^1, and '2' represents 2 multiplied by 10^2. In mathematical form, this is:

$$\begin{aligned} 250_{10} &= \underline{2} \times 10^2 + \underline{5} \times 10^1 + \underline{0} \times 10^0 \\ (\text{But } 10^2 &= 100,\ 10^1 = 10, \text{ and } 10^0 = 0) \\ \therefore\ 250_{10} &= 200 + 50 + 0 \\ &= 250_{10}. \end{aligned}$$

It is to be noted how the numbers underlined in the first line are brought together in '250' to represent the whole value of two hundred and fifty.

It is called *decimal* notation because in each digit place in a number there are ten possible values for each place, namely 0,1,2,3,4,5,6,7,8,9.

Thus we could take the equation:

$$Z_{10} = W \times 10^2 + X \times 10^1 + Y \times 10^0,$$

and if we were to let the variables W, X, and Y be any of those ten values, Z could be any number up to 1000. For example,

$$\begin{aligned} &\text{let W be '4', X be '9', and Y be '2'} \\ &\text{then } Z_{10} = 4 \times 10^2 + 9 \times 10^1 + 2 \times 10^0 \\ &\qquad\quad = 492_{10} \end{aligned}$$

In *binary* notation, however, only two values for every digit place are allowed, namely '0' and '1'. Because there are only two values and not ten, we multiply the value in each digit place by a power of 2, instead of by a power of 10. That this must be so can be seen from the following examples.

Eleven in binary notation is '1011', and to indicate that '1011' does not represent one thousand and eleven, but eleven, we could write it as '1011_2' (i.e. eleven to the base two). In decimal notation eleven is '11_{10}', therefore

$$11_{10} = 1011_2.$$

This equality can be demonstrated mathematically:

$$\begin{aligned} 1011_2 &= \underline{1} \times 2^3 + \underline{0} \times 2^2 + \underline{1} \times 2^1 + \underline{1} \times 2^0 \\ (\text{but } 2^3 &= 8, 2^2 = 4, 2^1 = 2, \text{and } 2^0 = 1) \\ \therefore 1011_2 &= 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 \\ &= 8 + 0 + 2 + 1 \\ &= 11_{10}. \end{aligned}$$

It is noticed how in the case of binary notation also the underlined values in the first line are brought together in '1011' to represent the number eleven.

Again '250_{10}' would be written in binary notation as

$$11111010_2.$$

'11111010_2' is readily enough obtained from '250_{10}'. Ascertain the highest power of two that will divide into '250_{10}' once, and write a '1' (this '1' will be the left-most digit in the binary number). Ascertain the next highest power of two that will divide into the remainder from the previous division. If that power is immediately below the previous power, i.e. is less than it by no more than one, then write '1' to the immediate right of the first '1' we wrote; otherwise write '0' for each of the intervening powers that do not occur and then '1' for the power of '2' that does. This process is continued until there is no remainder left, or it is one; if the remainder is one, write '1' (it will be the right-most digit). Mathematically, the process is as follows (great detail deliberate):

$$\begin{aligned}250_{10} &= 128_{10} + 122_{10}\\ &= 128_{10} + (64_{10} + 58_{10})\\ &= 128_{10} + 64_{10} + (32_{10} + 26_{10})\\ &= 128_{10} + 64_{10} + 32_{10} + (16_{10} + 10_{10})\\ &= 128_{10} + 64_{10} + 32_{10} + 16_{10} + 8_{10} + 2_{10}\\ &= \underline{1} \times 2^7 + \underline{1} \times 2^6 + \underline{1} \times 2^5 + \underline{1} \times 2^4 + \underline{1} \times 2^3 + \underline{0} \times 2^2\\ &\quad + \underline{1} \times 2^1 + \underline{0} \times 2^0\end{aligned}$$

(take the '1's' and '0's' underlined, and bring them together)
$= 11111010_2$.

Because of the nature of its components, the computer must represent numbers in binary notation. For example, the IBM 1130 computer, in the memory area used during processing and called "core-storage", uses tiny metallic cores or rings that can hold a magnetic state to symbolise whatever is stored there. By altering the direction of flow of an electric current, these cores can be magnetised in two directions, clockwise and anti-clockwise. Since only two physical states are possible, it can be specified that each core is to symbolise the two digits '0' and '1', as shown in Figure 1.

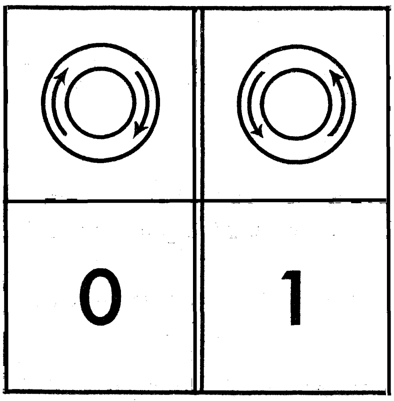

Figure 1. Magnetic symbols for binary symbols '0' and '1'.

Each core is called a "bit", and these are stacked several high to form a composite unit called a "word". It can be seen therefore that each word is suitable only for symbolising binary numbers, since each bit can only represent the two digits '0' and '1'. Each bit making up a word would represent one digit place of the binary number; in Figure 2, a ten-bit word symbolises two hundred and fifty in binary notation, i.e. 11111010_2. The arrows indicate the direction of the magnetic charge.

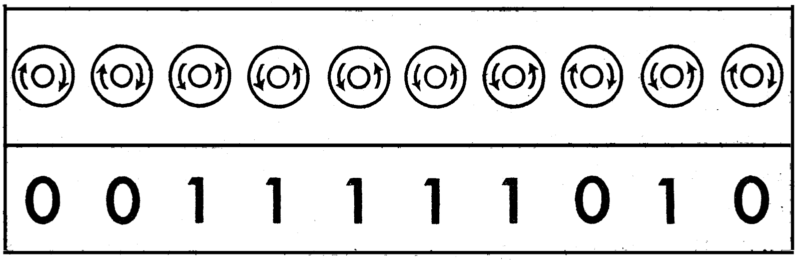

Figure 2. Two hundred and fifty symbolised by a ten-bit word.

In the IBM 1130, it takes sixteen bits to constitute one "word" of core-storage. It should be noted here also that the principles described above for core-storage apply as well to devices used by binary computers for their permanent memory, namely magnetic tapes, disks and drums.

However, the computer is not confined to numbers so that it can only be used as a complicated adding machine. It can symbolize the alphabet, and it does so by means of binary numbers which serve as a code for the letters of the alphabet. The magnetic words symbolise these binary numbers in the computer by means of bit patterns, just as they do for other numbers. But on any one occasion the computer knows when it is dealing with these binary numbers, not as numbers, but as a code for the letters of the alphabet, because it is given an instruction to that effect.

Table 1 is an illustration of a binary code in a six-bit word for the

Alphabet	Binary Code	Alphabet	Binary Code
(space)	010000	N	100101
A	010001	O	100110
B	010010	P	100111
C	010011	Q	101000
D	010100	R	101001
E	010101	S	110010
F	010110	T	110011
G	010111	U	110100
H	011000	V	110101
I	011001	W	110110
J	100001	X	110111
K	100010	Y	111000
L	100011	Z	111001
M	100100		

Table 1. A six-bit binary code for the alphabet.

letters of the alphabet. The space which occurs between words, i.e. ordinary words, is included as a letter of the alphabet.

Using such a code we can then construct symbols for strings of characters which form English words. It is possible to form symbols for sentences, paragraphs, or even books by using a binary code. By adding a few binary codes to those in Table 1 to symbolise commas, etc., punctuation can also be included.

To illustrate this six-bit binary code, T. H. Crowley[2] gives the lines

"How do I love thee?
Let me count the ways."

from one of E. Browning's sonnets in the form shown in Figure 3. In Figure 3 the meaning is just the same as the English original, though any warmth is banished.

```
Line 1  011000  100110  110110  010000  010100  100110
        010000  011001  010000  100011  100110  110101
        010101  010000  110011  011000  010101  010101  001111
Line 2  100011  010101  110011  010000  100100  110111
        010000  010011  100110  110100  100101  110011
        010000  110011  011000  010101  010000  110110
         01001  111000  110010  011011
```

The two underlined above are punctuation:
001111 = '?' (question mark)
011011 = '.' (full stop).

Figure 3. Two lines of a sonnet in a computer language.

The IBM 1130 uses an eight-bit code for the letters of the alphabet and other special characters; and as it has a sixteen-bit word, two characters can be packed into each word if it is desired. So, assuming that in Figure 3 an eight-bit code had been used, either forty-one or twenty-one words would be needed to symbolise or "store" the two lines in memory, depending on whether there were one or two characters per word.

In a computer's core-storage memory area, therefore, the two lines above would become physical symbols; i.e. groups of cores magnetised in certain directions. However, core-storage would not provide a permanent memory or record of these two lines. It is usually only large enough for the storage of the program to be executed (i.e. carried out) and any data that might be needed for processing in the course

[2] Crowley, T. H., *Understanding Computers* (1967) 25–26.

of the program's operation. It "forgets" everything stored there as soon as another program is executed. In the IBM 1130, core-storage would probably not be larger than 8,000 words. For permanent storage of the two lines of poetry, they would have to be stored on devices such as magnetic disks or tapes; these latter serve as the permanent memory of a computer. Obviously permanent storage of data would be essential for any data retrieval system.

In the IBM 1130 that the writer used there is only one disk on line at a time for permanent storage; so it was not a very suitable machine for, in any data retrieval system, large banks of data are required. The disk itself looks like an L.P. record, and uses small magnetic particles on its surface for bits instead of the ferro-magnetic rings used in core-storage.

Table 2 gives the amount of permanent storage available on the various permanent-storage devices, and the length of time it would take to have access, i.e. store in or retrieve from, any location in these types of memory.

Memory Type	Amount of Storage	Access Time
Disk	200–350 pages of data	Varies: 2 millionths to half a second.
Tape	7–8 complete 1,000 page	Varies: 10 millionths of a second to 10 seconds
Drum	Slightly larger than that given for the tape	Varies: 2 millionths to 30 thousandths of a second.

Table 2. Volume and access-time of memory devices.

The reason for the greater variation in access time for the tape is that each time it is searched it has to be unwound and re-wound.

However, before we could permanently store the two lines of poetry in Figure 3, we have to first get them into the computer. The process of getting data into the computer is called "input" of data. For input into the IBM 1130, the data has to be punched on to cards (or for some types of data, paper tape is suitable). Each character has a specific punch arrangement that symbolises it. The punch machine has a keyboard like that of an ordinary typewriter, but when a key is pressed the machine punches holes in one column of a card to form the punch code for the character designated by that key. Each card has a maximum of eighty columns that can be used. Just over forty columns would be used to represent the two lines of poetry. It is to be noted that the punch machine is not linked to the computer. Also it is a weak point in the use of a computer because it can be operated only at manual speed, which would be that of the particular typist using it.

Another piece of equipment, the *card reader*, which is linked to the computer, reads the data on the punched cards into the computer at speeds as fast as 1,000 cards per minute, i.e. 80,000 columns of punches per minute. The IBM 1130 card reader can read up to 300 cards per minute. The latter uses photo-electric cells which emit an electrical impulse whenever a punched hole is encountered. As the data is initially stored in core-storage, the electrical impulses emitted are transmitted to the word of core-storage which is to symbolise the character represented by the punched holes. Which bits in a word are magnetised in an anticlockwise direction, and which are not, depends on the pattern of impulses transmitted from the card reader.

To store the two lines permanently, the computer would need further instructions to the effect that it has to transmit the symbols in core-storage to the disk, and then move a protective "barrier" to prevent them being over-written with data later transmitted to the disk. The lines would remain there until the computer was instructed to delete them.

This idea of "transmission" of symbols might need clarification. It was said above that the bits in a word, when magnetised in particular directions during input, together constituted the physical symbol of a character. These bits are used to generate pulses of electrical current that travel in a direction corresponding to the way each bit is magnetised. These pulses, travelling at a 1,000 feet in 1,000,000th of a second, cause the bits of a word on the disk to be magnetised in directions that are the same as those of the bits in core-storage which generated the pulses. Thus the transmission of a symbol means that a symbol at one location is used to generate the same symbol at another location, and that the symbol in the first location is not in fact shifted at all.

Data retrieved from a computer's memory, or the results obtained from an operation that the computer has just performed, is called the "output". In the IBM 1130 computing system, a very fast *printer* which can print 320 lines per minute is used to produce the output. It is essential that the printer be extremely fast, because the faster the printer the faster are the speeds at which the computer can process. It usually happens that the computer has to pause in its operations while the printer catches up.

To illustrate the printer's speed, the printer gave a print-out of some programmes the writer had written, which took a typist 10–14 hours to punch out on cards, in less than ten minutes. The printer attains these speeds by printing whole lines at a time. To do this, the printer is geared so that the particular characters whose symbols were transmitted to the printer for output are arranged side by side in the consecutive order in which their symbols were received. When a special control character which acts like a typewriter carriage-return is received and indicates that the line is complete, the whole line is printed in one action. Control characters are specified in the course of the actual programming, and the use of them enables the spacing to be varied, or indentations to be made for the start of a new paragraph, etc.

IV. PROGRAMMING

The last section described in outline how a computer works. It is proposed now to explain briefly how a computer is programmed, i.e. instructed what to do. It is thought that this will help the reader to understand how the computer is adapted to suit the particular programmer's wishes.

The point has been made above that a computer is useless without programs, i.e. a set of detailed instructions to be executed by the computer in sequence. It is the programmer's task to analyse the work it is desired to have done by a computer, and then to draw up a set of instructions sufficient to make it carry out that work. To write a good program, the programmer must first acquire a deep understanding of the problem. In passing, it would seem therefore that the person most suited to adapt computers to solve the problems presented by the sources of law would be a programmer with a legal training.

In its most elementary form a program consists of a list of binary numbers. Each binary number is a code, firstly, for an operation the computer is to perform, and secondly, for where it is to find the data in core-storage to perform the operation with. The binary codes would tell it, for example, when to read a card, where to store the data read in from a card, or else what switches to set to carry out some processing.

To illustrate this, if we were using a sixteen-bit word, we could use the first five bits to form the code that specifies the operation to be performed. The rest of the bits could be used to represent the number of the word, or "location" as it is often called, in core-storage to or from which symbol transmission is to take place. For instance, we could let '01001' be the code for the instruction "Read a card", in which case the binary code in Figure 4 would show the actual instruction that could be framed.

01001	00000001001
READ a card and transmit data into	location, i.e. word 9 in core-storage

Figure 4. An instruction in machine language and its explanation.

That the last eleven bits would in fact represent location 9 is shown mathematically as follows:

$$
\begin{aligned}
00000001001_2 &\text{ is the same as } 1001_2 \\
1001_2 &= 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
&= 8 + 0 + 0 + 1 \\
&= 9_{10}.
\end{aligned}
$$

All the instructions that a computer actually executes are in fact in the form of binary codes somewhat similar in principle to the illustration given above. They are called "machine language". If we assumed a machine language such as that in Table 3, we could write a program such as that in Table 4. The X's in Table 3 fill the bits which would make up the binary number for a location in core-storage.

Binary Code	Description
11001XXXXXXXXXXX	An "ADD" instruction
01001XXXXXXXXXXX	A "READ" instruction
01010XXXXXXXXXXX	A "WRITE" instruction
01011XXXXXXXXXXX	A "STORE" instruction

Table 3. A machine language for illustration purposes.

The programme in Table 4 will add two numbers read into core-storage, and will print out the sum. The "accumulator" mentioned in the explanation in Table 4 is the part of the computer where adding, etc., are done.

Instruction	Explanation
0100100000001001	READ a number into location 9
0100100000001010	READ a number into location 10
1100100000001001	ADD number in location 9 into accumulator
1100100000001010	ADD number in location 10 into accumulator
0101100000001011	STORE sum in location 11
0101000000001011	WRITE sum from location 11

Table 4. Program for adding two numbers

The programmed instructions, stored in core-storage in the form of these binary codes, are transmitted to a part of the computer called "control". The control unit is analogous to a telephone operator in a manual exchange—it sets switches, selects circuits, and controls the processing generally. Since the control unit is an electrical circuit similar to all others in the computer, it too must receive the data transmitted to it in the form of electrical pulses (cf. "symbol transmission" discussed above in the previous section). The control unit has been designed so that it reacts in a pre-determined way to the particular pattern of pulses received; in this way the necessary switches are set so that the specified instruction can be performed. Thus, the binary instruction in Figure 4, on transmission to the control unit, would cause:

1. The circuit linking the card reader to core-storage to close for symbol transmission to location nine;
2. the card reader to read a card.

One could then say that "0100100000001001" had been executed. The control unit acts in response only to the codes it is designed for, i.e. the legitimate codes.

Equally important is the fact that the control unit incorporates an instruction counter. The binary codes for instructions are stored in consecutive order in core-storage. Let us suppose that the instructions for the program in Table 4 are stored in locations one to six. However, the control unit, of itself, would have no way of knowing which instruction to execute first, and which next. The instruction counter serves this purpose.

When execution of a program begins, the instruction counter is set to one, and the control unit goes to location one in core-storage for its first instruction. On completing the first instruction, the instruction counter is set to two, and the control unit goes to location two for its next instruction, and so on.

Turning to the instructions themselves again, the reader can probably appreciate the feat of memory that would be required to remember dozens of codes like those in Table 3, the high likelihood of error, and the tediousness of having to frame programs in machine language. It would be much easier if we could write instructions in familiar language and notation.

A way to do this would be to set up a table of equivalences such as those in Table 5, so that the programmer need only to write what is on the left when he wishes to specify the equivalent instruction in binary code on the right. 'N' in Table 5 is any location number the programmer happens to specify (cf. the use of the 'X's').

Form of instructions to be used by the programmer	Equivalent instruction in binary code
ADD N	11001XXXXXXXXXXX
READ N	01001XXXXXXXXXXX
WRITE N	01010XXXXXXXXXXX
STORE N	01011XXXXXXXXXXX

Table 5. A set of equivalences.

The instruction in the form on the right is solely for the benefit of the programmer. In fact a programmer would not need to know any of the instructions in machine language at all. When the computer has read the punched cards for the instructions in Table 6 (Table 4 translated), it would execute a specially written translater program. This

Instructions in a computer user's language
READ 9
READ 10
ADD 9
ADD 10
STORE 11
WRITE 11

Table 6. Using Table 5, a translated version of Table 4.

program would check the symbols from the instruction cards, and if it "recognised" legitimate instructions, it would create a program with equivalent instructions in binary codes. Thus the program the computer would actually execute would still be one in machine language, and only the computer would need to "know" machine language programming.

There is an actual language in common usage quite similar to that shown in Table 5, and illustrated in Table 6; it is called "assembly language". The specially written program which converts it into machine language is called the "assembler". But it is a higher level, or more abstracted language, because the programmer does not have to specify the actual locations in core-storage to or from which symbol transmission is to occur. When using the language in Table 5, it was said that the programmer had to specify what 'N' was to be; and this was done in Table 6. But, in assembly language, the programmer only has to write a different variable whenever he wishes a different location in core-storage to be used; the assembler is designed so that it chooses the actual location to be used when it translates assembly language into machine language. Table 7 illustrates this. 'N1', 'N2' and 'SUM' are the variables used.

Assembly language instructions
READ N1
READ N2
ADD N1
ADD N2
STORE SUM
WRITE SUM

Table 7. The program in Table 6 re-written in assembly language.

However, assembly language, although it has advantages of flexibility for certain purposes, is also very tedious and difficult to use. As in the use of machine language, programs written in assembly language are very detailed and every step is precisely defined; so much so that even a programmer can have difficulty following his own program. Error detection can also be very difficult.

But advantage has been taken of the fact that the same clusters of assembly language instructions re-occur whenever a similar operation is being performed, so that even more abstracted languages can be written. In such languages one instruction is used to specify a whole cluster of assembly language instructions; and the composition and notation of their instruction make it easier to mentally picture the operation the computer will perform.

Even though an assembly language is inherently more efficient for data handling, ease of programming makes a language such as FORTRAN a more suitable choice for this study. The equivalent of the assembly language program in Table 7 in a FORTRAN-like language would be the instructions, or "statements" as they are often called, in Table 8. FORTRAN IV statements themselves are not used as too much extra and unnecessary explanation would be required to explain further details.

Instructions, or "statements"
READ N1, N2
SUM = N1 + N2
WRITE SUM

Table 8. Program in table written in a FORTRAN-like language.

Yet another program is used to translate the FORTRAN IV statements into assembly language instructions. It is called the "compiler". As in the case of assembly language, 'N1', 'N2' and 'SUM' are variables used to indicate that different locations in core-storage are to be used, and the computer is left to select these different locations. Figure 5,[3] in diagramatic form, shows the complete process that occurs before a program in machine language that the computer can execute is arrived at. A program written in FORTRAN IV is called the "source program".

The IBM 1130 computer has the compiler and the assembler stored in its memory on the disk. Whenever a source program is read into its core-storage, the above programs are summoned in turn into core-storage for execution. Once the translation into the machine language program is completed, that program can then be executed immediately or, if required, be itself stored on the disk to await a summons into core-storage for execution on some future occasion. The translation of the source program into a machine language program is called "compilation".

[3] *Ibid.*, 94. A modified version is given here. This book is an excellent introduction to computers.

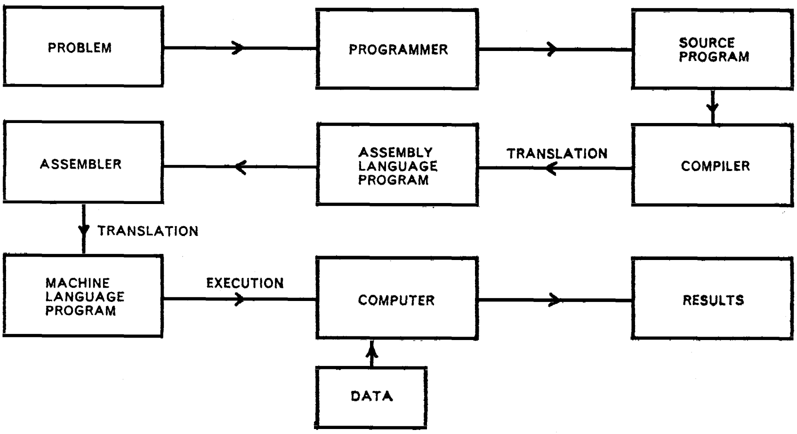

Figure 5. Diagrammatic representation of the preceding explanation.

It is hoped that the above discussion has brought out the respective roles played by the programmer and the computer when any application of the computer is being developed. It can probably be seen that the real burden of adapting the computer to solve the problems created by the sources of law would be on the programmer, and that the computer would only be a willing horse.

V. OUTLINE OF THE WRITER'S RESEARCH

The writer decided to choose the topic "Law and Computers" as a research subject for a seminar in 1969. There were two possible ways in which the topic could have been tackled. The first would have been to conduct a fairly academic study, i.e. confine any research to what had been written on the subject, and to then give an account of what had been achieved up to date. The other would have been to learn about computers and how to use them, and then to discover, by actually attempting to program a data retrieval system, the difficulties and problems that would have to be overcome.

The writer was discouraged from the first option because, when he started, he did not know of any work being done elsewhere.[4] Moreover, a natural curiosity about computers themselves prompted the latter course.

So, the first thing to do was to learn how to program. It did not take long to grasp the essential principles of FORTRAN IV programming,

[4] As it turned out, a lot of work has been done overseas. However, the writer does not know of any research done in New Zealand to date.

and the writer was directed to much material of great assistance.[5] He ran numerous programs through the computer, incorporating more and more of the FORTRAN IV statements as he learnt them. More seems to have been learnt about the statements from the "bungles" made, and the arduous task of working out what had gone wrong, than was ever learnt from bald explanations of them in a textbook. A useful by-product of several months of practising programming, whenever time could be found for it, was that several false notions of how data could be handled by the computer were destroyed.

Time had to be spent, too, learning how to use the disk. The disk has a set of instructions for its use quite apart from those that must be learnt in order to use the computer.

Gradually it was realised that the computer could not be bent to suit a specific problem, but that the problem had to be bent and manipulated into a form the computer would find acceptable. For one thing, it might be easy to get the computer to store data on the disk, but it has to be kept in mind how particular parts of that data are to be found on future occasions. The secret lies in organisation. A piece of unorganised data, e.g. several pages of written material, would prove completely unsuitable for useful and efficient access later to its contents, if it were deposited as it stood on to the disk.

It was quickly apparent that a lot of work would be involved to achieve anything worthwhile in the form of practical work in one year. It was found necessary to limit severely time spent reading about other projects at this stage.

However, some interesting pamphlets about an international conference of jurists in July 1967 at Geneva, at which the question of using computers within the law seems to have featured prominently, were brought to the writer's attention later in the year. They gave some insight into the surprising amount of research already under way in this field, and gave a list of centres overseas where it is being done. A very useful article in which much of this research was mentioned was that by Aviezri S. Fraenkel.[6] It appears that some projects are dormant, that any systems working are still largely only tentative schemes, and that the field is still wide open. However, the article does show that most of the main difficulties in developing a legal data retrieval system have been averted to, even if not solved.

With the help of the above materials it became clear that to carry

[5] McCracken, P. D., *A Guide to FORTRAN IV PROGRAMMING.*
I.B.M. Systems Reference Library:
I.B.M. 1130/1800 BASIC FORTRAN IV Language, File No. 1130/1800—25 Form C26—3715—2.
I.B.M. Computing System Users Guide, Manual No. C20—1690—0.
Robert K. Louden, *Programming the IBM 1130 and 1800*, Prentice-Hall.

[6] *Legal Information Retrieval* (1968) 9 Advances in Computers, 113–178.

out an analysis of the subject matter of cases so that they could be grouped under the appropriate heading in an index of legal classifications would be wholly impractical. To explain what is meant, when a lawyer is using a traditional index to find cases on equitable easements, he would first look for the legal classification REAL PROPERTY, then EASEMENTS; and EASEMENTS would have the sub-headings Legal and Equitable.

The reasons why such a procedure would be impractical are several. Firstly, it would require the use of lawyers to analyse each case to determine its classification. As such a method would be very time-consuming, and as lawyers would command high wages, the cost would be prohibitive. Secondly, such headings would not tell us about many things that would have been discussed in the case. In short, it is hard to compress several pages of a judge's decision into a summary a paragraph in length, let alone a few words. Things such as *obiter dicta*, etc., are not covered. It might be asked: what if the analysis were comprehensive enough? But, again, the tremendous effort required to do this to, say, a 1,000 cases, would bring us back to the considerations again of time and cost.

Thirdly, the analysis of a case would depend very much on the opinion of the lawyer doing it; and proof of this is the amount of disagreement that often arises as to what a case really decided. Moreover, it is unavoidably rigid and presupposes that future lawyers would regard the case in the same way as present-day lawyers do.

Finally, legal classifications would be too general. An immense number of cases would probably be produced if an inquirer were to ask the computer for cases on "Equitable Easements". Since most of them would probably be irrelevant, the whole purpose of a legal data retrieval system would be defeated.

A method that has proved suitable is that known as KWIC, or the key-word-in-context method. Assuming that data banks had already been created, a search for relevant data is carried out by selecting a small number of words which are then used to find all the cases which contain them. The words selected are fed into the computer as data for a request and search program. The citation of the cases which contained the words are then printed out.

An excellent illustration of this system is a commercial project operated in the United States by an organisation called "Automated Law Searching". The following is a quotation explaining how their system for searching statutes works:[7]

"The user conducts a search by indicating to the computer those words whose presence in a section, in the relationship he specifies,

[7] This is taken from advertising material published by the Automated Law Searching Co.

> would probably signal its relevance to his problem. Here [an actual example is referred to], in a search for sections dealing with the rights of parents with respect to abandoned children up for adoption, the user specifies four concepts which he believed would be expressed in any applicable statute.
>
> The instructions are to find every section containing the word CHILD or one of its synonyms, and ABANDON or one of its grammatical variations. The computer is to search the Pennsylvania statutes vocabulary listing[8] for these sections and store them. The computer is then told to find every section containing PARENT or PARENTS and ADOPT or one of its variations. These sections are to be identified and stored. The computer is then told to compare the contents of the two groups and print only those sections which appear in both [groups]. The final result is that only those sections will be printed which contain a word or phrase expressive of each of the four concepts."

Many variations of the KWIC method are possible, but the essential idea behind all is the same. It is that the very words of any written material should be used as the content of any index, as those words can best serve as the keys to the written material's subject matter. The context of the word in any material retrieved would reveal whether the sense in which it was used in that particular context was relevant or not.

The following is another way in which the writer feels computers could help to tap the wealth of information in the law reports. It would involve taking advantage of the doctrine of judicial precedent.

Since precedent is very important in the common law, earlier cases are constantly being cited in later cases. As can be expected, the later case would have dealt to some degree or at length with the same subject matter as that dealt with in the earlier case. For this reason, a useful technique often used to find out more information on a problem is to look up cases in which a known case has been cited. For example, if a lawyer has a problem that has flavours of *Hedley Byrne* v. *Heller*,[9] much light would be shed on the problem before him if he could quickly obtain a run-down of the cases in which *Hedley Byrne's* case was cited. This should not be unexpected, as precedent does embody the commonsense principle that new situations that arise are often to a large extent variations of situations that have arisen before.

The computer would be a very efficient assistant here. What is pro-

[8] The vocabulary contains all the words which can be legitimately specified in a search out of all those that occur in the statutes, e.g. "the", "or", "which" are not words that can be specified. Each word in the vocabulary has the citation of the sections in which it appears stored with it.

[9] *Hedley Byrne & Co. Ltd.* v. *Heller & Partners Ltd.* [1963] 2 All E.R. 575.

posed is that the computer would have on storage in its memory banks distinct units of data, i.e. records, each of which would contain the following:

1. The case citing;
2. the case cited;
3. how the cited case was treated;
4. from what aspect the cited case was discussed; and
5. what the citing case was about.

To enlarge on 3 above, a case cited does not generally emerge unaffected by the experience. It can, for example, emerge approved, distinguished, overruled, doubted, questioned, or as an authority for a principle.

Thus, when the computer is asked for the annotations of a particular case, the five pieces of information above would be printed out for each case found that cites the specified case.

VI. THE PROBLEMS OF A CASE LAW RETRIEVAL SYSTEM

Most of the problems involved in organising case law data so that it would be a suitable subject of an information retrieval system, and so that there would be reasonable certainty of most of the relevant information being retrieved, are faced by any system dealing with language and written materials.

The basic cause of the problems is that, unfortunately, the computer does not understand what is being fed into it, nor what it is doing. It is simply a symbol storer, and works by the transmission of symbols.

A. The Problems

The problems are threefold: unsuitable words, grammatical variations of words, and synonyms.

1. *Unsuitable words*

The bulk of a case is made up of words unsuitable for indexing. The idea of an index is to have only those words in it that are "concept storers", and it is pointless to include words that would never be requested; for example, "of", "it", "they", "we", "a", "the", "was", "were", and adverbs, connectives and conjunctions.

There is the danger, too, of repetitive indexing because a word can re-occur many times in a case; for example, in a case on trusts the word "beneficiary" or "trustee" would appear many times. Also, some words are so general that they would occur in thousands of cases; examples are "contract", "trust", "crime", "offence".

How is the computer to be told
- (a) not to index certain words on the ground that they are unimportant;
- (b) to index each different word in a case once, and not repetitively;
- (c) not to index certain general words.

2. *Grammatical variations*

Most words have grammatical variations such as various verb forms, and a noun form, e.g. abandon, abandons, abandoned, abandoning, and abandonment. If a person were to specify the word "abandonment" as being one of the words relevant cases would be likely to contain, how is the computer to be told that the words "abandons", "abandoned", "abandoning", or simply "abandon" also embody the same concept?

3. *Synonyms*

Most words have synonyms, and in some cases they are numerous; for example, "automobile" has the synonyms "car", "motor-car", "motor vehicle" and "vehicle". How is the computer to be told that "motor-car" means the same thing as "automobile"?

Moreover, in some instances, words are synonyms when they are in certain contexts, but not when they are in others. Consider, for example, the underlined words in the following three sentences:
- (a) There are specific legal requirements for the signing of wills.
- (b) Two persons must witness the execution of a will.
- (c) The execution of murderers has been abolished in New Zealand.

  "Execution" in the sense it has in (b) is the synonym of "signing", but not when it has the sense it has in (c). How is the computer to be told, assuming that a way of telling it that certain words are synonyms has been worked out, that "execution" is the synonym of "signing" in some instances but not in others?

B. Possible Solutions to the Above Problems

1. *Unsuitable words*

(a) *Unnecessary words*

It would be possible to select, say, a hundred cases at random and let the computer digest them to build up a word frequency table. It would probably be found that the words that occurred most often would be those that it would be unnecessary to include in an index; e.g. "it", "was", "were", etc. The programs for the input and digesting of cases could then eliminate these words, and any other undesirables that might be decided upon either *a priori* or by experience.

(b) *Repetition of words*

This problem could easily enough be overcome. Some instructions could be included in the input and digesting programmes to the effect that a word is to be indexed on the first occasion only that it is found in a case, and is to be ignored on subsequent occasions. Alternatively, a counter could be incremented by one on each occasion the word is encountered, and this number could be entered after the word in the index. This number could be useful, because, if a key word occurred particularly frequently in a case, it would show that the chances of that case being relevant would be pretty high.

(c) *General words*

General words would prove a little more awkward. How are the words which are to be ignored for indexing purposes (cf. unnecessary words above) to be selected? Words that are too general, unlike unnecessary words, have meaningful content. Yet it is suggested that they might have to be ignored on the ground that they occur much too frequently. Perhaps a simple frequency test might be sufficient to decide which words are not to be indexed.

On the other hand, it is felt that this is a problem better suited to examination in the light of experience with an operating system.

2. *Grammatical variations*

This problem could be solved by using a bank of grammatical variations of words. When a word is specified in a search request, the computer could check this bank for its grammatical variations (if any) and include these automatically as words to be searched for.

3. *Synonyms*

This problem would require somewhat more ingenuity in order to alleviate the effects that flow from it. It would never be open to complete solution because language is a living thing and the nuances of meaning are enormous. However, law does have the advantage that words are used precisely.

A "thesaurus" bank could be built up similar to the bank suggested for grammatical variations above. To illustrate how it would work, if the word "child" were specified in a search request, the computer would go to the thesaurus and would find there the synonyms "infant" and "minor". The computer would then include these two words in the key words it would search for.

However in building the thesaurus it would be difficult not to overlook that a word has synonyms, or to give only an incomplete listing of the possible synonyms.

To turn to the second aspect of the problem of synonyms; namely, what to do about words that are only sometimes synonyms. One possible way to prevent irrelevant cases being retrieved because of this lack of consistency in language might be to allow the inquirer to make negative specifications of key words. Using the example above to explain what is meant, the inquirer could specify "murderer" as a key word that must *not* occur in any cases the computer retrieves. In this way, a case using "execution" in the sense it has in sentence (c) would not be retrieved if it also contained the word "murderer". This would help to cut down the number of irrelevant retrievals.

VII. AN ATTEMPTED CASE LAW RETRIEVAL SYSTEM

As noted earlier, for the purposes of research for his seminar the writer preferred to examine the problems involved in case law retrieval by doing some actual practical work rather than by academic study. As a result the work handed in for the seminar at the end of 1969 consisted in large mainly of a tentative case law data retrieval system. It was a rather unsophisticated first attempt and it did not go very far by way of incorporating solutions to the problems discussed in the last section.

As designed, the system consists of approximately twenty programs and subprograms. A "subprogram", it should be explained, is an auxiliary program to the main program that uses it; it is called by the main program to perform a specific task as an integral part of that main program. The print-out for the programs amounted to fifty pages. However there are about ten to fifteen pages to be added as two of the programs were not ready for inclusion at the time the research for the seminar had to be completed. The print-out consists of the complete listing given by the computer (via the printer) of what was on each of the cards that made up the FORTRAN IV source program.

In writing the programs, comments were liberally used to explain what the FORTRAN IV statements meant and for other purposes. It was a simple matter to include comments in the source program without interfering with its compilation. Each comment card has a "C" in its first column, and the computer knows that when such cards occur it is not to include the symbols they contain for compilation, but only to include them in the print-out.

A. The Two Sub-systems

The Case Law Data Retrieval System has two sub-systems: the Annotation Retrieval System, and the "KWIC" Case Retrieval System.

In the case of the first, the Annotation Retrieval System, the inquirer would specify a case whose annotations he wanted and the computer would print out the following information for each case it found that cited the given case:

1. The citation of the case that cited the specified case;
2. how the cited case was treated; e.g. was it overruled, questioned, approved, applied, etc.;
3. the page in the cited case which was quoted from or referred to as being of particular interest;
4. the page(s) where the cited case was discussed in the citing case;
5. and lastly, approximately eight words[10] about the particular subject matter in reference to which the case was cited.

In the case of the second syb-system, the "KWIC" Case Retrieval System, the inquirer would specify a few words embodying concepts which he would expect to appear in any relevant case. Where all or, at least, most of the words specified were found in a particular context, the computer will supply the following information:

1. The citation of the case in which the words specified appeared; and,
2. a limited sampling of the context of that case.

The twenty programs and subprograms would be divided between these two sub-systems, and some of them would serve in both as well.

B. The Setting-up of the Whole System

Once the programs are all ready, and the errors in them have been eliminated, they are fed into the computer in the form of a deck of cards. Approximately 2,000 cards were needed for the programs here. When the cards for each program have been read in, the FORTRAN IV source program on those cards is compiled into a machine language program. By means of two cards following each program, the computer is given an order to immediately store the program just compiled into permanent storage on the disk. The programs remain there until deleted by further instructions.

Each program in the system is referenced by a name. For example, in the first program of the system to be stored on the disk, it is stated that the program is to have the name 'MASTR'. The statement, or instruction,

```
*NAME MASTR
```

makes this specification. The name that references a program is useful because by simply naming the program in the "execution" statement as follows,

```
//XEQ MASTR
```

the program can be summoned from the disk, executed, and be returned

[10] "Word" is not used in the technical sense (see the discussion in Section III), but in the ordinary sense; namely, words of language.

to the disk when execution is complete. Thus with one card we can, in effect, make the computer perform any operation however complicated the program required for it may be.

There is a special technique available that enabled all the individual programs to be formed into a co-ordinated system; it also makes use of the fact that programs can be referenced with names. Instructions can be included in a program to the effect that, if a certain condition is fulfilled, the computer is to cease execution of that program, summon another program from the disk into core-storage, and commence its execution. The other program is named in the calling program. This is known as the "CALL LINK" statement, and the form of the statement if it appeared in a program calling link to program 'CASES', for example, would be

```
CALL LINK (CASES)
```

C. Classification of the Programs

Each program (and subprogram) has its own particular function to perform, but nevertheless because of similarity of function they can be classified under four main headings:

1. The master-control program.

Its function is to control the whole system. It selects which of the other programs are to be executed, and the order of their execution. This program is called 'MASTR'. It has a subprogram called 'CLERK' that it can call on to help with its task of selection.

2. The data-bank-creation programs.

Their function is to organise data into a suitably referenced form so that it can be found later, and to store the referenced data into files on the disk. The "files" are the data banks and each entry into them is called a "record". These programs are called 'CASES, 'ANNOT', and 'KWIC1'.

3. The clerical programs.

There are four; they are called 'SORT1', 'SORT2', 'SORT3', 'SAME'. The function of the first three is to organise the data in the files into alphabetical or numerical order, depending on the type of data in the file. The function of the fourth program is to check one of the files for identical entries and eliminate them.

The writer did not have to develop the three sorting programs mentioned above. A research graduate in physics allowed him to use an extremely fast program he had developed himself for sorting the records in a file on the disk. It can sort 1,000 records in fifteen minutes; i.e. faster than one a second. All he needed to adapt his program were the particulars about each file to be sorted. Another program of his then used these particulars to generate modified versions of his sorting program. Three sort programs were generated in this way, since the particulars for each of the three files to be sorted varied. The programs generated were punched out on to cards, ready for use by the computer.

4. The data retrieval programs.

Their function is to retrieve data from the files (or data banks) created by 'CASES', 'ANNOT' and 'KWICI'. They are called 'FIND' and 'KWIC2'. The former retrieves data for the Annotation Retrieval System, and the latter retrieves data for the 'KWIC' Case Retrieval System. 'KWIC2' has yet to be developed—it will probably require difficult and complicated programming.

In addition to the programs mentioned above, there are numerous subprograms which are called by them, and therefore fit into one or other of the classifications above.

VIII. CONTROL OF THE WHOLE SYSTEM, AND ITS OPERATION ILLUSTRATED

The whole Case Law Data Retrieval System is controlled, as said above, by the Master Control program 'MASTR'. This program receives directions by the use of code cards. The code cards are read into the computer as ordinary data cards, which in fact they really are; they supply data on the basis of which the computer decides which of the possible paths of operation in program 'MASTR' it will take. Program 'MASTR' is supposed to read only code cards, and has instructions to ignore everything on a card except what is punched in columns 79–80 of the card. The code cards for program 'MASTR' have 'AA', 'BB', 'CC', 'DD', 'EE', or 'LS' punched in columns 79–80. If a card with anything else in columns 79–80 is read, it is ignored and the next card is read.

Further code cards are used to give directions on the path of operation to be followed in programs that 'MASTR' has brought into execution; some have the code in columns 1–2 of the card. However, since on every occasion that the system is used 'MASTR' is always the first program to be executed, one of the six code cards that give 'MASTR' its directions *must* always be the first card in the deck of data cards to be read into the computer. One of these six code cards must also occur as the next card to be read in, whenever program 'MASTR' is executed after another program has called link to it in the course of the system's operation.

In Table 9, there is a complete listing under four headings of all the code cards that were used to control the operation of the Case Law Data Retrieval System. There is typical data included in the table to illustrate the system's operation; it is derived from *Curtis* v.

INDEX	Columns 1–2 CARDS	Columns 79–80
A.	*Code Cards for Data-Bank-Creation Program, and Typical Data*	
1		AA
2		BB
3	CURTIS *v.* CHEM. CO./1951/1 K.B. 805 (C.A.)	
4 (a) (i)	L'ESTRANGE *v.* GRAUCOB /1934/ 2 K.B. 394	
(ii)	APPLIED 403, 807, 808	
(iii)	WRITTEN TERMS — SIGNED UNREAD — *INNOCENT MISREPR. — 810 RECISSION POSS.	
(b) (i)	OLLEY *v.* MARLBOROUGH /1949/ 1 K.B. 532	
(ii)	APPLIED 808	
(iii)	COMMON LAW LIABILITIES — REQUIREMENTS *FOR EFFECTIVE EXEMPTION — NOTICE	
(c) (i)	REX *v.* KYLSANT /1932/ 1 K.B. 442	
(ii)	APPLIED 809	
(iii)	FRAUDULENT + INNOCENT MISREPR. COMPARED *— VOID EXEMPTIONS — PRINTED FORM	
5	YY	
6 (a)	((CONTRACT)) — b ((NEGLIGENCE)) — bDRESS *LEFT FOR bb((CLEANING)) — b((DAMAGE)) *-b ((CONDIT	
(b)	IONS))//ONbb ((RECEPTb)) ((EXEMPTINGb)) *CLEANER FROM b ((LIABILITYb)) *FOR DAMAGE//HO	
(c)	WEVER CAUSED — bRECEIPTb ((SIGNED)) bBY *PLAINTIFFb- b((INNOCENT))// *((MISREPRESENTATIO	
(d)	))NbBY SHOP ASSISTANT. b((EXEMPTIONbCLAUSE)) *bb((FAILS))	
7		99
8	VV	
B.	*Code Cards for the Clerical Programs*	
1		CC
2		S1
3		S2
4		S3
5		SM
C.	*Code Cards for Retrieval Programs, and Typical Request Data*	
1		DD
2	L'ESTRANGE *v.* GRAUCOB /1934/ 2 K.B. 394	
3	(A blank card would occur here as a code card)	
4		EE
5	EXEMPTION CLAUSE, INNOCENT, MISREPRESENTATION, RECEIPT.	
D.	*Code Card to Terminate the Operation of the Whole System*	
1		LS

Table 9. Code cards and typical data for the Case Law Data Retrieval System.

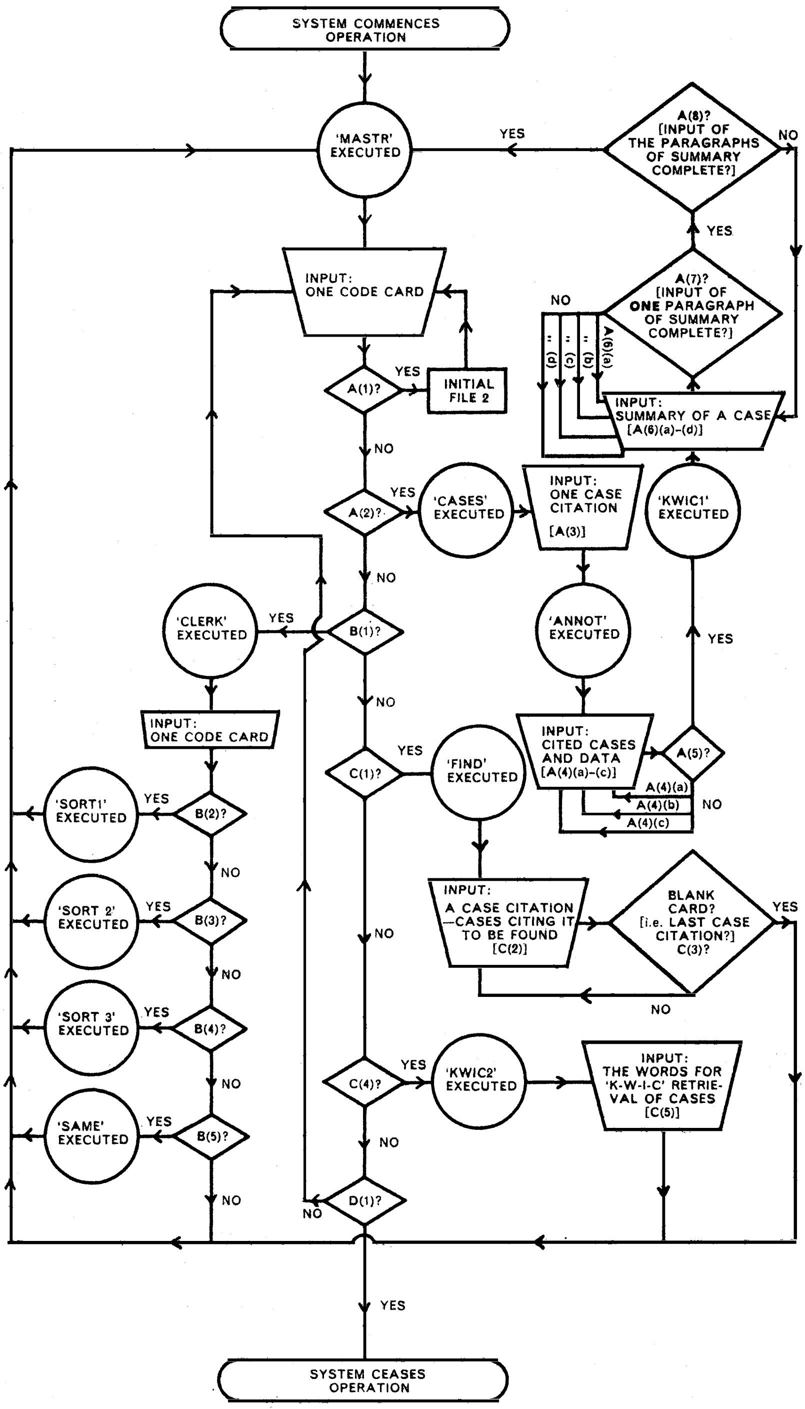

Figure 6. The flow-chart for the Case Law Data Retrieval System, showing the co-ordination of the programs, and method of control.

Chemical Dry Cleaning Co.,[11] and is in the form or "format" it would need be to meet the requirements of one of the data input programs. Each line of the table represents one card; where an asterisk occurs it is not a new card but merely a continuation from the preceding line. The index is used to reference the cards for discussion purposes in the text; the table is discussed in four subdivisions. It would be best to imagine the table as a deck of data cards about to be fed into the computer in the order they are in. The 'b' written frequently in lines 6(a)–(d) is there to indicate a space where it is not obvious; 'b' equals 'blank', and does not appear on the actual cards. The double slashes in 6(a)–(d) are a code for the retrieval program 'KWIC2'.

Figure 6 preceding is a flow-chart showing how the programs in the Case Law Data Retrieval System are co-ordinated, and how the code cards control the operation of the system; this diagram, or flow-chart, will help the reader to understand the following explanation. All references in Figure 6, for example 'A(2)', 'C(5)', 'A(4)(a)–(c)', 'A(6)(a)–(d)', etc., are to the index in Table 9. The circles indicate programs; the trapeziums indicate that a card appearing in Table 9 is being read; and the diamonds indicate that a card read in was a code card and that a decision has to be made between two paths.

The following four subdivisions explain what the programs do as each of the cards in Table 9 is fed into the computer. The explanation follows the order of the cards in the table. The index in Table 9 has been used to refer to the cards in the course of the discussion. For example, 'A(1)' in "If A(1)......" refers to the first card in Table 9, and "If A(1)......" therefore means: "If the card fed into the computer has 'AA' in columns 79–80 of the card, then ......". "If A(1)" is used as a useful abbreviation.

A. Code Cards for Data-Bank-Creation Programs, and Typical Data

Assuming that the system has been called into operation by the instruction

```
//XEQ MASTR
```

and that the cards in Table 9 are now being fed into the computer, then:

If A(1), then 'MASTR' is directed to set up a special file, File 2, on the disk containing the number of records, and the number of *sorted* records in each of the other files. After the first occasion the system is operated, the use of this card is *never* repeated.

If A(2), then 'MASTR' calls link to 'CASES'.

'CASES' reads in card A(3) which contains the citation of a case to be digested. The citation is stored in File 1 on the disk. 'CASES' then automatically calls link to 'ANNOT'.

'ANNOT' reads in cards A(4)(a)(i)–(iii), which is the citation of a case cited in the case on card A(3) and other data.[12] All of this is stored in File 3. The next card is not A(5), therefore the same process is repeated for cards A(4)(b)(i)–(iii), and ditto for cards A(4)(c)(i)–(iii). The citation in each of these cases is also stored in File 1.

If A(5), then 'ANNOT' is directed to call link to 'KWIC1'.

'KWIC1' reads in cards A(6)(a)–(d) which is a paragraph summarising the subject matter of the case on card A(3). 'KWIC1' puts the words inside the brackets into File 5, which contains key-words that reference the paragraph (and hence the case) they were extracted from. 'KWIC1' then eliminates the brackets and stores the *whole* paragraph into File 4, which contains only paragraphs.

[At present 'KWIC1' is unsatisfactory because obviously the process of preparing the paragraph into the form it is here on cards A(6)(a)–(d) is very laborious and slow: the case has to be summarised and the key words have to be bracketed manually to suit 'KWIC1' as it is programmed now. Also, with a summary, there is always information lost; and it is fairly subjective. The computer would be much faster if it were doing all the work. But on what criteria it is to choose the key words. Again, how would it summarise a case and at the same time minimise information loss. If the entire case were to be stored as it is in data banks, they would have to be extremely vast and therefore the system would be very expensive. Moreover it would merely be duplicating the Law Reports in the library. Such then are the problems that must be overcome to improve 'KWIC1'.]

If A(7), then 'KWIC1' is directed that it has finished reading in the cards containing the paragraph on cards A6(a)–(d). This card is needed because the paragraphs are of variable length, and the computer keeps on reading in the cards for the paragraph until card A(7) is encountered.

If A(8), then 'KWIC1' is directed that there are no more paragraphs summarising the case on card A(3) to be read in, and that it is to call link to 'MASTR'.

[The system allows for the summary of a case to consist of more than one paragraph; each paragraph would deal with different aspects of the case.]

'MASTR' then reads the next card, which *must* be one of its six code

[12] See the Discussion of the Annotation Retrieval System in Section VIIA above. The particulars of what data the computer would need when executing 'ANNOT' are given there.

cards. For the purposes of this explanation, it is assumed that the next card is not the code card that directs 'MASTR' to cease execution and return to the disk. So, to continue:

B. Code Cards for the Clerical Programs

If B(1), then 'MASTR' is instructed to call on its auxiliary, subprogram 'CLERK', to perform the selection of the next program to be executed.

Subprogram 'CLERK' reads the next card, which must be one of the code cards with 'S1', 'S2', 'S3', 'SM' in columns 79–80.

If B(2), then subprogram 'CLERK' calls link to 'SORT1'.

'SORT1' sorts the records in File 1 on the disk. It calls link back to 'MASTR' when it has finished sorting. Therefore, card B(1) must preface any occurrence of cards B(3), B(4), or B(5).

If B(3), then 'CLERK' calls link to 'SORT2'.

'SORT2' sorts File 3 and *also* calls link back to 'MASTR' when it has finished sorting.

If B(4), then 'CLERK' calls link to 'SORT 3'.

'SORT3' sorts File 5, which contains key-words, into alphabetical order. It also calls link back to 'MASTR' when it has finished sorting.

If B(5), then 'CLERK' calls link to 'SAME'.

'SAME' eliminates identical entries that could occur in File 1 for this reason: the case on card A(3) would be stored in File 1 by 'CASES', and when it occurred (if ever) as a cited case in a later case 'ANNOT' would store its citation in File 1 too. Hence there could be two entries of the same citation in File 1. 'SAME' calls link back to 'MASTR' when it has finished checking File 1.

Again the next card in the deck of cards being fed into the compute *must* be one of the six code cards for program 'MASTR', So, to continue:

C. Code Cards for Retrieval Programs, and Typical Request Data

If C(1), then 'MASTR' is directed to call link to 'FIND'.

'FIND' will read from card C(2) the citation of the case for which it has to find annotations. 'FIND' then searches File 3 on the disk for entries where the case on card C(2) is given as having been cited. However, in order to save storage space in File 3, instead of the full citation of citing cases being contained therein, only a key referencing the relevant full citation in File 1 is given. Therefore it is necessary for 'FIND' to search File 1 as well for the full citation of a citing case whose key it found in File 3. 'FIND' then writes out on the printer all the annotations it found for the case on card C(2).

If C(3), then 'FIND' is directed that for the present there are no more cases for which annotations have to be found, and that it has to call link to 'MASTR'.

If C(4), then 'MASTR' calls link to 'KWIC2'.

'KWIC2' reads card C(5) which contains words all or some of which must occur in any paragraph the computer prints out as being relevant to the inquirer's problem. 'KWIC2' first searches File 5 to see if the words specified on card C(5) occur there as key-words. After each key-word in File 5 there is a key referencing the paragraph that the key-word came from. 'KWIC2' compares the keys after the words given on card C(5) if they occur in File 5. Whenever it finds that the key after all, or most, of the key-words is the same, it then knows that the paragraph that the key-words came from is one that should be printed out. Hence 'FIND' would print out the paragraph (in its edited form) that was fed into the computer on cards A(6)(1)–(d) if the key-words given by an inquirer were those on card C(5). 'FIND' would also supply the citation of the case from which the paragraph came. When all the paragraphs that have been found to be relevant by this *gormal*[13] test have been printed out, 'FIND' calls link back to 'MASTR'.

D. Code Card to Terminate the Operation of the Whole System

This code card would always be the last card to occur in any deck of data cards being fed into the computer. But to continue with the explanation of the way the code cards control the system:

If D(1), then 'MASTR' is instructed that it must cease execution and return to storage on the disk.

Thus the whole Case Law Data Retrieval System will lie dormant on the disk, ready to come into operation whenever the instruction card

//XEQ MASTR

is again fed into the computer.

Table 10 following shows how the information retrieved by "FIND" would appear in a typical print-out from the computer, assuming that 'FIND' had been searching for the annotations of *L'Estrange* v. *Graucob.*[14] Whatever occurs to the left of the double dashes in Table 10 is standard for every print-out from the Annotation Retrieval System.

[13] The *actual* test for relevance will be when the inquirer reads the paragraphs printed out by the computer to see if they have any relevance to his problem.
[14] [1934] 2 K.B. 394.

```
THE CITED CASE - - L'ESTRANGE v. GRAUCOB/1934/2 K.B. 394
    PARTICULAR PAGE CITED (IF ANY) - - 403
THE CITING CASE - - CURTIS v. CHEM CO./1951/1 K.B. 805
                                    (C.A.)
  THE CITED CASE WAS - - APPLIED
    THE CITED CASE IS FOUND AT PAGE(S) - - 807, 808
BRIEF NOTE TO INDICATE SUBJECT MATTER - -
    WRITTEN TERMS - SIGNED UNREAD - INNOCENT
    MISREPR. - 810 RECISSION POSS.
```

Table 10. Typical print-out from program 'FIND'.

Table 11 following shows how a typical print-out from program 'KWIC2' would appear, assuming that the paragraph summarising *Curtis* v. *Chemical Dry-cleaning Co.*[11] had been retrieved as being relevant subsequent to an inquiry specifying the words:

EXEMPTION CLAUSE, INNOCENT, MISREPRESENTATION, RECEIPT.

```
CONTRACT — NEGLIGENCE — DRESS LEFT FOR CLEANING
— DAMAGE — CONDITIONS ON RECEIPT EXEMPTING
CLEANERS FROM LIABILITY FOR DAMAGE HOWEVER
CAUSED — RECEIPT SIGNED BY PLAINTIFF — INNOCENT
MISREPRESENTATION BY SHOP ASSISTANT. EXEMPTION
CLAUSE FAILS.
    - - CURTIS v. CHEM. CO./1951/1 K.B. 805 (C.A.).
```

Table 11. Typical print-out from program 'KWIC2'.