The structure of the Cambridge algebra system

D. Barton*, S. R. Bourne† and J. R. Horton*

* University Mathematical Laboratory, Cambridge
† Trinity College, Cambridge

This paper describes the implementation of an algebra system under development in Cambridge.
(Received October 1969)

The algebra system under construction in Cambridge consists of a programming language together with a package of programs to perform certain types of algebraic manipulation. The development of this work gives rise to two fundamental classes of problem. The first class is directly associated with the nature of the manipulative algebra to be attempted and includes such problems as the integration of arbitrary expressions and the so-called 'simplification problem'. The second class of problems arises from the fact that elementary operations on mathematical expressions, such as addition, may well take an inordinately long time when programmed for a computer, and that algebraic expressions themselves frequently occupy very large quantities of the machine's store. The two classes are not completely distinct since successful simplification techniques will substantially reduce the machine facilities required by subsequent manipulation. However, it is useful to consider the second class of problem independently of the first and discover how to construct an algebra system that makes economical use of the computer. Once this system exists it may then be used to conduct experiments that will assist in the attack on the more difficult problems of manipulative algebra that we have placed in the former class.

Let us consider now the representation of the user's basic data element by the system and consider how this may be achieved efficiently. It is obvious that while it is possible to consider an ordinary floating point number as a degenerate polynomial the overheads implied by such a representation prohibit the use of a polynomial manipulation system for ordinary numerical calculations. This type of false generality may easily arise in a manipulative algebra system, when a simple expression is treated as a degenerate example of a more complicated entity, and is perhaps the major cause of diseconomy arising in such systems. This difficulty is partially overcome by constructing the Cambridge system from a variety of packages that each compute with different types of expression. Then, given a particular problem, the user may select a combination of packages that represents with least redundancy the expressions occurring in the problem.

The second source of inefficiency in an algebra system arises through the choice of data structure used to represent particular types of algebraic expression. A polynomial might be considered simply as a string of characters and manipulated by editing operations or alternatively represented by a list structure and manipulated by a list processing language. Other representations are also possible. The selection of a particular type of representation should be based, in the main, on considerations of machine efficiency at *runtime* while the availability of a particular language in which to program the manipulative system itself should be of secondary importance. The type of data structure that it is economic to use depends, in part, on the nature of the object to be represented and may well differ substantially for different classes of mathematical entity. Thus the system should be programmed in a language appropriate to the processing of the type of data structure used to represent mathematical expressions. The latter remarks indicate that the design of the language used will be dictated by the broad requirements of the algebra system and that the use of an established language may constrain the choice of data structure in an undesirable manner. It is with the above points in mind that the Cambridge Algebra System has been designed.

The hierarchical structure of the manipulative system

We now describe the way in which the complete manipulative system has been divided into self-contained component subsystems known as *runtime systems*. The divisions made in the system correspond to the divisions that occur in mathematical expressions themselves and are probably best illustrated by example. Consider the composition of a manipulative system to compute in the ring P of polynomials in n variables over a field G. The addition of two polynomials takes place in P and this will cause arithmetic between the coefficients of the polynomials that must be carried out in G. Thus we provide a runtime system $\bar{P}$ to perform the manipulation of the data structure representing a polynomial. This data structure will contain references to the elements of G and, during an addition, the runtime system $\bar{P}$ makes calls to another runtime system $\bar{G}$ to carry out arithmetic on the coefficients in G of the polynomials.

This structure can be simply generalised. In order to manipulate the ring P' of polynomials over a new ground field G' we need only to provide a runtime system $\bar{G}'$ manipulating the elements of G' and couple this with the $\bar{P}$ manipulator. In both of these examples we say that the user makes *direct* use of the manipulator $\bar{P}$ and *indirect* use of the manipulators $\bar{G}$ or $\bar{G}'$. We may, of course, use the $\bar{P}$ manipulator indirectly. Suppose that we wish to compute with square matrices whose elements are drawn from a ring. The matrices themselves form a ring M for which we must provide a runtime system $\bar{M}$.

Arithmetic between the members of M will involve arithmetic between the elements of the matrices and we may provide this in several ways.

1. $\bar{M}$ and $\bar{G}$ will allow the manipulation of matrices whose elements are in G.
2. $\bar{M}$ and $\bar{G}'$ manipulate matrices with elements G'.
3. $\bar{M}$, $\bar{P}$ and $\bar{G}$ manipulate matrices with polynomial elements whose coefficients are drawn from G.
4. $\bar{M}$, $\bar{P}$ and $\bar{G}'$ manipulate matrices with polynomial elements whose coefficients are drawn from G'.

We see that in cases 3 and 4 above $\bar{P}$ is used indirectly. Direct use of either G or G' would provide simply for manipulation of field elements in G or G' respectively.

Provided that a system is built as indicated above and a suitable set of basic runtime systems is provided it will be possible for the user to select the appropriate combination of runtime systems to obtain a manipulative package that satisfies the requirements of his problem with least redundancy. Provided also that a runtime system is well defined, it is possible for the user to write a new member system for inclusion in the hierarchy if no suitable combination of systems already exists. Further, when the system is subdivided in the above manner each runtime system is concerned only with the manipulation of those parts of the overall data structure known to it. No other runtime system requires direct access to this part of the data structure, which may therefore be arranged in the machine's store as considerations of space and time dictate. A runtime system may always be replaced by a system manipulating a different representation of the same mathematical entity and hence there is room for experimentation to determine the most economic data structures.

The structure of a runtime system

We shall now discuss in greater detail the set of subroutines of which a manipulative runtime system is composed. Broadly speaking, a runtime system is the set of all subroutines that deal with a particular aspect of the complete data structure. Each routine takes as input head pointers* or integers, and either modifies an existing data structure, creates a new structure, or returns an integer or head pointer as output. This output is passed to the calling program that is either a user program, if the runtime system is being used directly, or the runtime system immediately above it in the hierarchy if it is used indirectly. As an example we describe a minimal set of subroutines that would comprise the manipulator $\bar{P}$ for polynomials in P. These subroutines may be divided into four classes, A to D.

Class A subroutines are housekeeping programs. They are not directly connected with the arithmetic operations necessary in P but are required to administer space and arrange type conversion between numerical data and algebraic data. Two routines are provided, one to copy an existing polynomial into new space obtained from the space administration system and one to return the space occupied by a polynomial to the pool of vacant space. Further routines are necessary to convert a number into a polynomial and a polynomial into a number if this latter operation is possible.

*A head pointer is the store address of the beginning of a data structure.

Class B subroutines are those that perform the arithmetic and other fundamental operations on the polynomials. This class of subroutines frequently contains programs that are unique to a particular runtime system and depend on the nature of the entity represented. However, we have found that the following subroutines occur in most non-numerical systems.

In this case α and β represent polynomials contained in P and the Class B routines included in the $\bar{P}$ manipulator are:

1. $\alpha = -\alpha$ (negate α)
2. $\alpha = \partial\alpha/\partial x_i$ (differentiate α w.r.t. x_i)
3. $\alpha = \int \alpha dx_i$ (integrate α w.r.t. x_i)
4. $\alpha = \alpha + \beta$ (add α to β)
5. $\alpha = \alpha - \beta$ (subtract β from α)
6. $\alpha = \alpha\beta$ (multiply α by β)
7. $\alpha = \alpha/\beta$ (divide α by β)
8. $\alpha = \alpha \uparrow \beta$ (raise α to the power β)
9. $\alpha = \alpha(x_0, ..., \beta, ..., x_n)$ (this routine substitutes the polynomial β into the polynomial α for one of the variables x_i)
10. Print α on an output device.
11. Read α from an input device.

Class C subroutines are intended to allow the user to make his own economies with the mathematical formalism. In any particular instance the Class C subroutines simply produce the result of an arithmetic operation modified according to some prescribed rules, the exact nature of which is not relevant to this discussion. To illustrate the use of these routines, suppose that we are multiplying two polynomials in P and that only the part of the result linear in a particular variable is required. It is clear that there are more economical means of obtaining this result than simply to compute the entire product and subsequently to discard unwanted terms, and if a Class C subroutine is provided for this purpose in $\bar{P}$ then considerable economies are possible at runtime. We have found that Class C subroutines of $\bar{P}$ include versions of all the Class B routines involving multiplication of polynomials. These subroutines carry out the formal manipulation of polynomials in the ordinary manner but dynamically apply certain rules to the partial result of the computation. Finally, they produce a result, modified according to the rules imposed, and are thus able to make substantial savings in both space and computation time compared with that required to construct and then edit the complete result.

In general the specification of the subroutines in Class C for a complicated mathematical entity is not a trivial matter since the type of formal mathematical device for which it is useful to provide a special facility is heavily dependent on the nature of the object under consideration. It is our experience that these are best discovered by experiment and by reference to user programs. To provide a Class C subroutine for a particular purpose, as in the case of the linear example above, is usually very easy. The difficulty lies in the specification of a set of such routines that provides comprehensive facilities while not leading to a proliferation of special purpose subroutines, although in the case of polynomials this latter problem leads to very little trouble.

The above three classes of subroutines are adequate to allow the construction of a runtime system for use

directly by an object program, but if it is to be used indirectly by another runtime system a further class must be provided. The subroutines in this final class, D, provide for communication between the several runtime systems and they are intended to allow the calling program to arrange its activities more economically in the light of the subordinate system's requirements. A typical such routine calculates the space occupied by a section of data structure or alternatively estimates the computation time required for a particular operation. Such statistics are of considerable use to simplification programs whose object is to try to contain the user's calculations within the available machine store.

The object programming language

Once a set of runtime systems exists the user may call any combination of systems appropriate to his problem. Each combination manipulates a distinct mathematical entity and consequently, when designing a programming language for users of the system, it is convenient to draw the distinction, discussed by Wilkes (1968), between the inner and outer syntax of the potential language. It is clear that if this distinction is not made then a separate language and compiler must be provided corresponding to each mathematical entity manipulated by the system. Further, the provision of a new runtime system, giving rise to several new combinations of systems, would also require the production of an equivalent number of new compilers. To avoid this proliferation of compilers the programming language is provided with a type *expression* to which no explicit meaning is attached. The translation of syntactic constructions involving expressions only specifies the operations to be performed on them, without reference to their particular mathematical significance. At runtime, these operations can be passed to the appropriate runtime system for interpretation. It is then possible for the same programming language to be used with any combination of runtime systems and consequently only one compiler need be produced.

The programming language in use with the Cambridge Algebra System contains two types, *index* and *expression*. The language is very similar to Titan Autocode, and consequently the *index* and *expression* variables are given single letter names. We next describe these variable types:

Index variables Referred to by the names I, J, ..., T alternatively as arrays I[m, n, ...], ..., T[m, n, ...]. An *index* variable may take integral values in the range $|n| < 10^7$.

Expression variables Called by the names A, B, ... H, U, ... , Z or alternatively as arrays A[m, n, ...], ..., H[m, n, ...], U[m, n, ...], ..., Z[m, n, ...]. An *expression* variable is used to denote a mathematical entity upon which certain operations such as addition and multiplication are possible.

Program control in the language is provided by means of *For* loops, conditional and unconditional jumps to labels and a closed subroutine facility.

As we have indicated above, constructions involving the type *expression* must be translated into a sequence of calls to a runtime system which may subsequently be obeyed causing the appropriate runtime system to be entered. For example the statement

$$A = B + C$$

causes the addition routine of the top level runtime system to be entered with pointers to the structures named B and C as arguments, and the pointer to the resulting sum to be assigned to the variable A. In order to reduce the space occupied by the user program, it is compiled into a compact interpretive code rather than machine code. This device provides a 50% saving in space for an average program with a negligible increase in execution time since the time spent in the user program is small in comparison with that spent in the various runtime systems.

Apart from control of arithmetic and other operations on mathematical objects the programming language also contains constructions to allow control of the interface between the Algebra System and the Titan operating system. Thus the user has control of all input/output by the system and he may use magnetic tape or disc backing store for the storage of both numerical and algebraic data. Since the same compiler is used with all combinations of runtime systems it is necessary for the user to specify the combination of systems to be loaded at the end of the compilation.

The programming language for the writer of a runtime system

We have described earlier the classes of subroutines that compose a runtime system and have indicated that in general these subroutines manipulate a section of data structure and make calls to runtime systems below them in the hierarchy. It is, therefore, possible to write a runtime system in the user's programming language provided that the manipulation of the data structure is programmed with *index* variables and *index* arrays, while the calls to the lower level are initiated by the use of *expression* variables. A number of problems arise if this procedure is adopted. Firstly, a runtime system will in general spend a large proportion of its time on the administration of its own data structure with comparatively infrequent calls to the subordinate system. It is, therefore, no longer reasonable to interpret the index arithmetic and hence those parts of the program must be compiled into efficient machine code. Secondly, the user programming language does not allow the programmer access to an arbitrary quantity of machine store for arrays of index variables, since the size of such arrays is fixed at compile time. The programmer of a runtime system clearly requires access to a variable amount of space for his data structures drawn from the system's central space routines and consequently the programming language that he uses must provide facilities to obtain and relinquish this space. However, experience shows that the runtime system programmer does not require control over input/output facilities or backing store.

The above considerations imply that it would be more convenient to use a separate version of the programming language that contains instructions to access arbitrary areas of store with the freedom that would normally be available only to a machine code programmer, and a separate compiler that generates both interpretive and

direct codes. Such a variant of the language has been produced as part of the Algebra System and a program written in this language is compiled into machine code and interpretive code with the latter type of code resulting only from constructions involving *expressions*. The machine code generated by the compiler is approximately 125% the length of the equivalent optimal hand-coded program and could if necessary be further improved by a code optimisation process.

The algebra system compilers

We have indicated earlier in this paper that, by delaying the interpretation of the type *expression* until runtime, it is possible to design a programming language for both users and runtime system programmers that will operate with any combination of runtime systems. While this is broadly true there are circumstances in which some knowledge of the nature of the mathematical object under consideration at runtime is required at compile time. For example, the language must contain a syntactic construction to denote differentiation if the system is to be used with polynomials. However, when the language is used to manipulate matrices whose elements are numbers, the construction referring to differentiation is no longer relevant. Thus, associated with each syntactic construction of the language that refers to the expressions, there is a list of the possible combinations of subsystems for which that construction is meaningful.

The addition of a new runtime system to the algebra system means, in general, that a number of new combinations of systems become available. For these it will be necessary to add syntactic constructions to the language that have not previously appeared and it is also necessary to label the existing constructions of the language that remain relevant to the new combinations. It is desirable that this complicated operation should be carried out easily and the two compilers for the Algebra System are therefore held in separate syntax tables. A new compiler is generated by presenting its syntax table to the compiler-compiler Psyco (Irons, 1961, Matthewman, 1965).

Let us now examine in greater detail the syntax tables that represent the two compilers. The syntactic constructions of the language may be divided into two groups; those that are meaningful for all combinations of runtime systems and the remainder. The first of these groups is by far the greater and contains all constructions that do not refer to the explicit, detailed structure of the type *expression*. It includes therefore all constructions referring to:

(*a*) Program control.
(*b*) Index arithmetic.
(*c*) Communication with the Titan operating system.
(*d*) Communication with the Algebra System space administration routines.
(*e*) All Input/Output.
(*f*) Arithmetic between *expressions*, e.g. B + C.

The remaining constructions are those that are only meaningful for some combinations of runtime systems, i.e. those which call the specialised routines in Class B and all the Class C routines.

The constructions of the first group have the same translation for all combinations of runtime system. Those of the second group have the same translations for all combinations of runtime system for which they have meaning but are rejected as syntax errors if they occur in a program using an inappropriate combination of systems. The possibility of distinguishing between separate translations for constructions in the first group according to the combination of systems to be used was discarded to reduce compiler size. It is this decision that causes us to maintain two separate compilers for users and runtime system programmers since otherwise the two syntax tables could clearly be merged.

General remarks upon the operation of the system

It has been explained above that the Cambridge Algebra System consists of a set of runtime systems together with a compiler for users' programs and a compiler for runtime system programs. A program to be run using the system is presented to the compiler, and if successfully compiled, the resulting interpretive code is loaded into store together with the relevant runtime systems. Finally, the interpreter and a set of service routines are loaded. The service routines comprise:

1. A space administration system.
2. Magnetic tape and disc control system.
3. Various print routines.
4. A runtime trace facility.
5. A post mortem system.

It will be noted that the machine dependent parts of the whole algebra system are almost entirely included in the first four of these. Thus at runtime there is present in store

1. The user's compiled program.
2. The interpreter and service routines.
3. Such runtime systems as are required.

The remaining store is initialised as the free store and is administered by the space routines.

The space occupied by the several components of the system is listed below in units of 1 block = 512 Titan* words of core (a Titan word contains 48 bits).

1. *Interpreter*	1 Block
2. *Service routines*	3 Blocks
3. *Runtime systems*	
(*a*) Arithmetic	1 Block
(*b*) Polynomials	3 Blocks (contains (*a*))
(*c*) Fourier series	6 Blocks (contains (*a*) & (*b*))
(*d*) Elementary functions	10 Blocks
(*e*) Tensors	7 Blocks

System (*a*) provides Integer, Rational, Real, and Complex Arithmetic, in a single manipulator since the individual systems are too small to make their separation useful. The above runtime systems are frequently used in one of the following combinations to give complete manipulative schemes.

1. Polynomials + Arithmetic	7 Blocks
2. Fourier series + Arithmetic	10 Blocks (3)

* Titan is the prototype Atlas II.

3. Elementary functions with polynomial arguments. The polynomials having numerical coefficients. 17 Blocks (4)
4. Tensor manipulator with polynomial elements. 14 Blocks
5. Tensor manipulator with elementary functions as elements. 24 Blocks

Further details of the systems numbered 2 and 3 above may be obtained from Barton, Bourne and Burgess (1968) and Barton, Bourne and Fitch (1969).

Assuming that an average user program occupies 2 Blocks we have only 26 Blocks allocated to the program using even the most complicated combination of runtime systems available, namely 5, while the program to manipulate polynomials will occupy just 9 Blocks. In practice both of these figures are reduced by two blocks since the service routines are overlaid. During initialisation it is necessary to have one block of service routines in store. These are over-written by more service routines for use during the actual running of the algebra program and finally the post mortem part of the service routines overwrites this latter set of routines in the event of a failure in the user's program. The above estimates of space do not of course include that required by the data structures that may extend to the capacity of the machine if necessary.

In conclusion it is once again a pleasure to express our thanks to the Director and Staff of the Mathematical Laboratory for their continued interest in this project and for extensive use of the computing facilities available in Cambridge. Our thanks are also due to Mr. J. P. Fitch for his assistance throughout the development of the system.

References

BARTON, D., BOURNE, S. R., and BURGESS, C. J. (1968). A simple algebra system, *The Computer Journal*, Vol. 11, No. 3.
BARTON, D., BOURNE, S. R., and FITCH, J. P. (1970). An algebra system, *The Computer Journal*, Vol. 13, No. 1.
IRONS, E. T. (1961). A syntax directed compiler for ALGOL 60, *CACM*, Vol. 4, p. 51.
MATTHEWMAN, J. H. (1965). Syntax Directed Compilers, Ph.D. Thesis, Cambridge.
WILKES, M. V. (1968). The inner and outer syntax of a programming language, *The Computer Journal*, Vol. 11, No. 3.

Book Review

Memory and Attention, by Donald A. Norman, 1969; 201 pages (*London: John Wiley & Sons Ltd.*, £4·20. cloth, £2·20 paper)

This book is divided up into nine chapters which start with a statement of the problem and finish with the conclusions. In between we consider a number of important and related topics which are built up, in seminar fashion, around the writings of various scientists. In a sense this is a book with a difference, since it does not collect papers written by other people but rather abstracts them and suggests them as a text and then with references and sections on suggested reading, builds up a picture of each of the many central features of what is connected to human information processing. It should be noted that the sub-title of the book is 'An introduction to human information processing'. Let us say right away that the book is an extremely readable one and is very well produced physically, and that it is insightful in its analysis of some of the thornier problems involved in human information processing. In fact, the problems considered are central to the whole issue, since they include such major subjects as attention, recognition, particularly pattern recognition, memory and the computer simulation of these various cognitive features.

The author divides memory into primary and secondary components, and in so doing is well in step with later views on this subject. He presents a number of models which purport to provide some picture of the memory process. This includes well-known models such as those of Miller, Galanter and Pribram who worked on plans and totes. Less familiar models such as that of mnemonics in the context of the 'Greek art of memory' are also included. One of the points that is brought out quite clearly here is the fact that much of what we now understand about memory was understood a very long time ago. However, the computer simulation of many of these processes holds out high hopes for considerable future development in the field.

One of the most important issues raised by Dr. Norman is that between the 'active' and 'passive' theories of pattern recognition. In the first place he draws attention to the vital importance of attention. Attention is the process which is clearly linked to perception, conception, discrimination, and remembering. This process of extracting he compares, in Broadbent's writing, to that of having an active filter. This is also a process of selecting, and Dr. Norman sees that the sort of stimulus analysing mechanism suggested by Sutherland is appropriate to the process. The main point he is making here is that the process of remembering and pattern recognition are active reconstructional processes rather than passive acceptance processes. This distinction between active and passive cognitive processes, and Dr. Norman like most other people in the field at the moment accepts that both are probably available to the human processing systems, is reminiscent of the difference between merely looking up, say, log tables in a book, as opposed to generating log tables as they are needed.

By and large, most people will accept the broad sweep and philosophy of this book and would accept it as yet another link in a chain which is gradually building up showing fairly clearly and explicitly how the cognitive processes operate. I would regard the book as thoroughly worthwhile. Certainly it is very readable and I think it is fair to encourage all people interested in the central cognitive activities to buy a copy.

F. H. GEORGE (Uxbridge)