negation, or/and, not, equals, less than, if—otherwise. For example, concatenation is defined as follows: Concatenation (o) is a binary operator with properties that:

(*a*) If $f_1 \circ f_2 = f_3$, then, for any $L, f_3(L)$ is the couple $f_1(L), f_2(L)$.
(*b*) If $f_1 \circ (f_2 \circ f_3) = f_4$, then $f_4(L)$ is the triple $f_1(L), f_2(L), f_3(L)$.
(*c*) $f_1 \circ (f_2 \circ f_3) = (f_1 \circ f_2) \circ f_3 = f_1 \circ f_2 \circ f_3$.
(*d*) In general $f_1 \circ f_2 \neq f_2 \circ f_1$.

The concatenation function is used for adjoining values of distinct properties, such as considering several sort keys side by side as one larger sort key. The Language Structure Group has used the Information Algebra in its present form to state several simplified data-processing problems, such as a payroll problem, and has also used the Algebra to define the concept of updating a file by several other files. It has noted that the information contained in a file may be represented in a property space in different ways depending on the choice of entities, and has shown that in certain cases functions of glumps and functions of bundles may be used to represent one area set in terms of another, and vice versa. The Language Structure Group plans to search for classes of problems which can be stated in terms of the Information Algebra, for which it can visualize automatic programs which could invent solutions to the problem and generate machine codes to influence the solution.

Discussion
Session 4: 18 April 1962

The Chairman (Dr. M. V. Wilkes, *University Mathematical Laboratory, Cambridge*): I have been asked to report briefly on the International Symposium on Symbolic Languages for Data Processing which was held recently in Rome (26–31 March 1962). This Symposium was organized by the International Computation Centre, which has just come into official existence after existing for some years in provisional form. The treaty establishing the centre was negotiated some years ago, but it has only recently been ratified by the minimum of ten Governments necessary for it to come into force.

The general situation in programming languages reminds me very strongly of the situation in computers themselves just ten years ago. At that time a few digital computers were working and others were in course of construction. Many people were fascinated by the new ideas behind the logical design of computers. It sometimes seemed that there was more talk than progress. So it is now with programming languages. A number of pioneering systems are in operation, but work on really sophisticated systems is only beginning and it will be some years before they are fully developed.

The Symposium was much dominated by ALGOL—perhaps too much so—and there was plenty of that type of arguing that one has come to associate with gatherings of ALGOL enthusiasts. The result was a background of controversy which resulted in there being, I thought, surprisingly little free discussion of programming languages in general, what features they should have, and what they are going to be like in the future.

It was clear that in one respect some advance in thinking had taken place since the ALGOL report was written, namely in regard to how a language should be specified. This morning Mr. Glennie referred to the fact that when FORTRAN existed for one machine only, the compiler constituted in effect the definition of the language, but that more formal treatment was necessary when the language came to be implemented for a number of machines. This brings out what I think has not always been clearly realized, that the defining of a language and the writing of a compiler are, in a way, similar operations. Thus, while the definition of a language should not be a compiler for any actual machine, there is much to be said for the view that a definition should be constructed along the same lines as a compiler. In ALGOL this was not done, and a sharp distinction was made between syntax and semantics. Many people now feel that there should not be two separate procedures, one for finding out whether a given piece of discourse is legal, and the other for finding out what it means, but that there should be one procedure only; this would either lead to the statement that the discourse means nothing, or would yield its meaning.

The discussions about ALGOL brought out two things quite clearly. In the first place ALGOL is incomplete in a number of respects. It lacks input and output facilities, and is not sufficiently rich in means for data description; one cannot, for example, declare double-length numbers, or manipulate sets. This need for extension is, I think, generally recognized. The other point relates to the efficiency of the object program, and here there is much controversy. Compared with other automatic-programming systems, object programs generated from ALGOL turn out to be rather long, and in view of certain features of the language, there is little that the writer of a compiler can do about this. A number of people, including myself, believe that some changes will have to be made in the language before it is acceptable for general use in a busy computing centre.

Unfortunately, when the ALGOL Committee finished their work in 1960, they did not set up any organization for performing the routine maintenance necessary with any programming language, or for developing and extending the language. For some time there has been in existence a movement to rectify this omission, and the matter had been considered in the week before the Symposium by the Council of the International Federation for Information Processing (IFIP), of which The British Computer Society is a member. The Council agreed that it would offer to set up a working party under IFIP auspices to take charge of the future of ALGOL. This offer was subject to the agreement of the original ALGOL Committee, and it so happened that quite a number of members of this Committee were present in Rome and able to meet together. I am glad to say that the IFIP proposal found acceptance, and the working party has now been set up. It includes the original ALGOL Committee, or as many of them as are still active in the subject.

From the point of view of international organization, the setting up of this working party represents a distinct step forward.

In the meanwhile, a problem faces those who like the idea of ALGOL, but find that it does not fully meet their requirements in its present form. We had an interesting lecture from Dr. Schwartz (System Development Corporation), who told us that his group was originally in very much this position. They decided that they would themselves modify and extend ALGOL, and in this way they created the language JOVIAL. A JOVIAL program looks very much like an ALGOL program, but the language is much richer, particularly in data description. To go ahead independently in this way is one means of achieving progress, and I would say to people who are at the present time considering the implementation of ALGOL, that it would be a mistake to adhere slavishly to a language as laid down in the report; it would be quite proper to extend and adapt it as seemed necessary, and a language so created might well prove to be a more valuable commodity than the original.

One thing that ALGOL has done for us all is to provide a means of communication between programmers. I find that I can now talk about programming to, and even exchange programs with, people who have machines very different from the one we have in Cambridge. This is a decided advance, and one that is not likely to be lost by the changes that may now take place officially in ALGOL, or even by changes that are introduced unofficially to meet local requirements.

I would like to mention two things not connected with ALGOL that were spoken about at the Symposium. One was a new system of list processing described by Dr. Weizenbaum (G.E.). A paper on this system has just appeared in the *Communications of the A.C.M.* It has a number of interesting features, including an original method of "garbage collection."

Dr. Harry Huskey of the University of California, Berkeley, spoke about machine independence in the writing of compilers, and described his experience in transferring NELIAC, a fairly large system of automatic programming, from one computer to another. I would like to try the experiment of transferring NELIAC to EDSAC 2, and I had a number of private discussions with Dr. Huskey about the possibility of doing this.

My overriding general impression of the symposium is that we have in computer languages a field of increasing technicality and increasing abstraction. Subtle ideas are called for, and one must be able to think on several levels at once. Discussions about meta-languages and meta-symbols, and as to whether a given symbol stands for itself or for something else—all these show that the subject is no longer quite the same plain, matter-of-fact one that it used to be. (*Applause.*)

(*Dr. R. D. Clippinger then read his papers.*)

Mrs. M. M. Barritt (*Royal Aircraft Establishment, Farnborough*): Dr. Clippinger, could I ask you to go on from where you just left off. You mentioned that you have this small group of people whose "supervisors" do not know what they are doing. This seemed to be the very thing that we were lacking yesterday. In commercial programming we have this set of manufacturers working extremely hard, under pressure, to produce compilers for a specific range of customers, but there was no mention of any University groups or Government Research Establishments "squandering" their time on a purely experimental approach to a good commercial language, possibly allied to a scientific language with prospects of amalgamation. Now, in the United States, have you found that Universities are, as it were, putting on one side all the commercial work and leaving it to the manufacturers?

Dr. R. D. Clippinger: Broadly speaking, I would say that is the situation. There are universities that are getting into data processing from many different angles. Economics Departments are getting interested in the use of computers for a variety of ideas, and Industrial Management Departments are getting interested in computers from the point of view of dyanmic programming of the systems work of a company. Business schools are starting to think about the use of computers. They are starting to talk about linear programming and instituting courses on linear programming, and giving case studies in the use of linear programming. I do not think you would find a university in the United States that is not in data processing in one way or another. On the other hand, I do not know any university in the United States that directs its attention to the notion of designing a good language for data processing. There are universities working on the problems of language translation, an entirely different kind of thing. It is related, but it is not the same problem.

I guess that is about all I can say, but, of course, the universities are also involved in teaching people to program. This is another way for universities to be involved and it is an important way for the universities to be involved; there is not as much of that as we in business would like to see. In other words, there is not anywhere near as much as there shortly will be, because this is growing very rapidly. Some universities have had computers a long time, and the teaching staffs are teaching the latest techniques of programming; these are growing, but this obtains only in a few universities.

Mr. K. W. Lawrence (*General Post Office*): I have been listening with a great deal of interest during the last two days, trying to discover where we are going. As a user of computers, my reaction is, "what do we want from these languages?" This was stated earlier on as being some form of compatibility of programs between one installation and another, and easier program writing. I do not find us getting anywhere near these. There does not seem to be any encouragement from the speakers that we shall ever get compatibility between installations, or even between the machines of the same manufacturer, unless he does the same as Honeywell have done and builds subsequent machines round the common language already written for the earlier machines.

With regard to easier programming, I was sorry to hear Dr. Clippinger say that in the future COBOL is going to be extended. He has already told us that it is more difficult to learn assembly languages than machine languages, and still more difficult to learn COBOL, although it is easier to use. I have looked through COBOL 60 and in the procedure section alone there are approximately 23 verbs, 68 key words, and 17 reserve words; and most of these are qualified with five different notes of some form or another. To hear that we are now going to have Report Writing and other aspects added to COBOL suggests to me that we will soon have longer compilers, longer compiling runs, and possibly even more inefficient object programs.

This may be acceptable for service centres where the consumer pays for the additional running time caused by inefficient object programs, but what about the person who is buying a computer for his own use? He also pays for the inefficient use of his machine.

I feel that we should go a little bit farther back in this problem and try to see if we cannot get some rationalization at the source. Instead of trying to produce a language to give us some commonality between the diverse monsters which we ourselves have produced, could we not make the machines and the assembly languages more compatible? Then a commercial user operating his own machine and hoping to produce economies may achieve compatibility one installation with another, with relatively simple program writing. This is more than he is ever likely to get with COBOL and the present multiplicity of machines. (*Applause.*)

Dr. R. D. Clippinger: You have expressed a lot of views and I do not *have* to comment on any of them, but I feel the urge to.

I agree with much of what you have said. I would point out that during the past day you have heard a description of several languages, several COBOLS amongst others. It is my contention that these more elaborate languages, albeit more difficult to learn, are more useful, and I should like to put this in a different form. During the next year or two, you are going to see us manufacturers locked in mortal combat, with salesmen out telling prospects that they can do their work using a compiler and that it is easy if a compiler is working. "Why do you not take an example and *you* do it yourself, in the language of machine A, or machine B, or machine C, and draw your own conclusions as to how hard it is and what is the efficiency of the object code? . . . And there will be a lot of complementary equipment you want to use, and we will help you to do it."

Now this is going to be a very different sales atmosphere! The prospective customer is going to be able to draw conclusions, which are valid conclusions, no longer based merely on hopes and distortions of salesmen. Now he will shake down and find out that it is not really so hard to use a complex compiler—or is it easy? He will find out whether the loss in efficiency is something he can support, or whether it is not. He will find out whether you can get enough programs to get the job done with primitive languages—or whether in fact he wants to get the job done? (*Laughter.*) I think it is going to be very interesting what is going to happen during the next few years, after which all this discussion is going to be academic!

Talking once more on the question of learning just a little bit, I think the experience of ALGOL has been very interesting. People take a look at ALGOL and they say that it is impossible to learn it. It is first class, but it is very formal and frightening; but you go through a course with a man who understands it, and you do a few exercises, and pretty quickly you begin to see that there is not anything that you cannot understand; pretty soon you are writing programs and you think that you know *all* about it, which is not true. Your learning process has just begun, but you have now been stimulated to learn, and you learn very fast; it takes hold, and once you have learned, you persuade yourself that it was no trouble at all to learn! What does it matter how hard it was to learn, if you reach this point and are now successfully using it?

Now, Mr. Lawrence talked about machines being more compatible. This is something that frightens the manufacturers and it frightens the users, because they are afraid that it is going to get rid of all diversity, all ingenuity; that it is going to shake down to one machine design, and whoever can produce that most efficiently will win the battle, and we shall all be using that machine. This would be a horrible result. I would hate to see this happen; I do not think it will happen. I do think that machines need to be made more compatible, and will be made more compatible, without the loss of ingenuity on the part of the manufacturers.

Mr. Thompson spoke yesterday* about the ideal language which was concise and extremely easy to use, and the compiler had many lovely properties. I criticized him a little bit, and yet I would have thought that we could move in the direction he has talked about, if we were to simplify our machines a little bit and make them more compatible, and standardize a little bit on how we use them.

For example, one gentleman yesterday spoke of five different conventions of using a card which he had to provide for, and I told you about FACT having 30,000 instructions in the input editor; you were all shocked and horrified, probably partly because with its many different conventions, which we provide, you are being enabled to live with it, because we are in process of adjusting to a world that is, and not a dream world. But we can move a bit towards this dream world by adopting these conventions, and by standardization, and then we shall all benefit very substantially in simplified compilers, more efficient object codes, more useful work done per dollar unit, or however you want to look at it, with a little more discipline and reason brought in regarding the way we do things.

Mr. J. P. Hough (*Robson, Morrow & Co.*): I should like to ask Dr. Clippinger, assuming 70 to 80% of the computers in the United Kingdom are of a power not greater than that represented by a four-tape 1401 or 301, is COBOL a suitable language for such a system?

Dr. R. D. Clippinger: I think that I have already hinted that I do not think so. In my opinion COBOL is much too complex for the smallest machines, and a new language is called for. That is why I spoke in the way I did about Language H, although Language H may not be it. I have not studied Language H, but I think there are more appropriate languages that can be developed for a machine of that level. I am a heretic, however, so this is a heretic's personal view.

Mr. J. P. Hough: Thank you, Sir. Is such a generalized system likely to yield efficient programs on machines of such limited powers?

Dr. R. D. Clippinger: Definitely not. That is why I hold this view that it is almost impossible to achieve efficient programs on a small machine for elaborate languages.

Mr. J. P. Hough: Is commonality in any degree possible on a machine of such limited power?

Dr. R. D. Clippinger: Yes, it is very possible but you presumably mean, is it possible keeping the machines as much different as possible, and in that environment I would say it is very difficult, but I think something can be done in that direction.

Mr. J. P. Hough: In your view, is commonality of practical interest to the majority of commercial E.D.P. users, as opposed to the scientific users?

Dr. R. D. Clippinger: I consider that a very good question. There are many different aspects to the question of commonality. In the United States there are some users of equipment that are gigantic, I think, from your point of view. I am thinking, for example, of the Air Force Logistics Command that has thirty computers of three different makes, and there are other systems where they have several different computers doing the same job. Clearly, commonality is important to them to the extent that it can be achieved,

* See page 164.

delivering the efficiency that is also important to them, and they are some of the prime movers towards commonality.

There is another reason—there are several other reasons—why commonality can be useful, but can be overrated. You program your application because you have a job to be done, and let us say you only have one computer, but some day it becomes obsolete. Then you get another computer and you can then choose let us say from half a dozen computers, each one of which could do that same job much more quickly if you only have that much money, if you could have it programmed for that machine. At this point, if you were to use the same language, and if it ran efficiently on one of those machines and you selected that machine, you now have the job running for less money than you had before, and the change-over did not cost you anything. This is somewhat of a dream, because at the same time machines are changing, the languages are evolving, and the appropriate language of the latter day may not be the one in which the program was originally written, even if you tried to use a language which provides commonality.

Furthermore, the amount of work in taking over a problem that is well defined in one of these compiler languages, and putting it into another language is very much less than the amount of work in taking the series of object programs, writing them in assembly language, and doing them in a new assembly language for another machine. In such a case you may have, for example, with a job which took ten man-years to program in the assembly language used originally, the possibility to reprogram it in the assembly language for another machine in a couple of man-years, because the job is so much better defined than it was originally. It might also be possible to reprogram in compiler language in six man-months, if it were in a compiler language. You could then probably reprogram in a different compiler language in one man-month, so I do not think you should put too much faith in commonality.

Mr. J. P. Hough: Finally, you have talked about program maintenance from the point of view of clarifying definitions. Would you like to speak as a manufacturer on the problem where the traditional pattern seems to be that you put a machine on the market for five to seven years, and to keep it competitive you announce new options which make it more efficient as time goes on?

My experience on one machine is that the languages never caught up with the options, so by the time you got floating-point or indexing registers on a first-generation machine, the language did not give you the facility to make the maximum use of these new options.

Dr. R. D. Clippinger: This is another very good point. The more complex the language is, the more difficult it is to modify it, to adapt it to the different situations. For example, discs are available on Honeywell equipment. We could re-do the FACT compiler to take advantage of discs in such a way that segmentation would be automatic. We have used parallel processing to call in the next segment to the extent of predicting which one is next, and it is segmented automatically, so that the programmer did not know anything about this segmentation. This appeared to remove the restriction of program size to an extent that would open the door to a new way of using computers. If we can do this, in theory it is a tough job, and we are now just about to try to do it. It would be very much tougher for a language as complex as FACT than it would be for a much simpler language, but this is a kind of revolutionary change due to a change in hardware specification which is particularly troublesome. It is so troublesome, that nobody is using discs in my opinion with the flexibility and the power that they could be used, because they have not wanted the trouble to think this out, and because it is a tough job.

But minor changes are added, such as communication equipment or paper tape instead of punched cards, or vice versa, and there is a steady stream of these little things. Sometimes, adjustments to the language can be made which make it convenient and relatively easy to use the additional items without big revolutionary program adjustments. We have been adding on more memories to make the compiler work well. With a small memory, it is a greater effort to make the compiler work than it is with a larger memory.

All of these things cost you human effort, and you have to choose between all the things you could do and pick out those which you feel you and the customers can pay for; it is a real problem.

The Chairman: I should like on your behalf to thank Dr. Clippinger very much for his talk which has led to a most interesting discussion.

We continue our program with two contributions and some discussion. I propose that we hear these two contributions first and then have a discussion on all of them at the end. The first speaker is Dr. Brooker, University of Manchester, and he will speak on compiler techniques.

(*Dr. R. A. Brooker, Manchester University, read his paper.*)

The Chairman: We are grateful to Dr. Brooker for that contribution; and now let us go straight to the final formal contribution by Mr. Strachey on future prospects.

Mr. C. Strachey: Some of the remarks that I was going to make about future prospects have already been made by previous speakers, and one or two of the things I wanted to say have already been said by my Chairman. He was comparing the present state of development of automatic-programming languages with the state of development of computers ten years ago. I should just like to expand that a little bit. It has seemed to me that the last two days have been very similar to one of the early conferences on computers where people were busy describing in—if I may borrow a word from Rapidwrite—what must be considered as talks on a pre-printed format; they were busy describing computers which would compare very favourably and would be working soon. When pressed as to how soon, the speaker would give an estimate (which came to be known later as a *Hartree constant*) of the time which would elapse before his machine was working; unfortunately the Hartree constant for any particular machine was a good deal more constant than most physical constants. Ultimately, of course, we had some machines, and I suppose ultimately we shall have some commercial translators which have been written in England.

The general development of compilers that is likely to take place in the immediate future has been talked about by Dr. Clippinger, and I shall spend most of my time talking about what I think is going to happen in the more remote future, let us say three or four years away. In this field, of course, that is very remote.

In order to do this I want to take off from the ground instead of from somewhere in the air, and I should like first of all to give a slight résumé of the position as I see it at the moment. What I am going to try to do is not to give a catalogue of the existing languages, but to give you what seem to be the primary characteristics of the languages that we already have, and what I think is the way in which they are going to change.

In the first place we have two sorts of language: business languages and mathematical languages. At first sight these seem to be totally different objects, and when you begin to look at them at all closely you find that it is very remarkable how they each are specialized in one field and extremely naïve in another.

Let us first consider the business languages. These are very sophisticated in their dealing with data and the handling of input and output, but they are very naïve indeed on procedure divisions when compared with mathematical languages like ALGOL. The sophistication of procedures involved in ALGOL is enormous compared with what you can do with COBOL, but, of course, the means for handling input and output and data description in ALGOL are naïve to the extent of being non-existent.

Now there is another feature which is woefully lacking in most realizations of ALGOL (and certainly in the early realizations of other mathematical languages), and that is any provision for an operating system for the computer itself. One of the most important developments of the last few years, in America, but, unfortunately, not yet very much over here, is the realization that an operating system for a large computer is an absolute necessity if you are going to make much use of the computer. The reason that it has not been developed over here is that we have very few, if any, large computers.

There are various other characteristics that I might mention to differentiate between business languages and mathematical languages. The business languages are rather imprecise in the definition of their actual language. It is difficult to make a syntactic analysis of a business language, and its terms tend to be somewhat imprecise. Business languages tend to avoid, as far as they can, the use of symbols, in what I think is the mistaken idea that symbols are so frightening that at all costs they are to be avoided. Instead the program sheets are filled up with noise words and *ad hoc* devices. The associated translators are very large and are very much concerned with problems of efficiency in the object program, though not, so far as I can make out, so much concerned with the efficiency of the translation process.

The mathematical languages, on the other hand, are very sophisticated in their handling of procedures in both senses. They are more naïve in their input and output operating systems. Their characteristic format is ALGOL, and those likely to be developed in the future will, I am sure, show this extreme precision of statement and definition of language, and very high degree of symbolism in the language in which they are expressed. They are essentially really suitable only for the mathematician, and for mathematically trained people. People do not like symbols. They are frightened of symbols and will not become happy with them until they have been educated not to be frightened by them.

The people who design the best algebraic languages and the best mathematical programming languages tend to go in for a great deal of abstracting and generalizing. They resist as much as they possibly can the temptation to use *ad hoc* devices. It may be that sometimes they go too far, although some people think that they do not go far enough. The essential thing is that they are continually looking at the problem to find abstract mathematical ideas; that is the kernel of the matter. When they find one they put it into the language and then see whether they can use it more efficiently and whether it increases the power of the language.

The translators which make use of the realization of these languages tend to be considerably smaller than those for the business languages. Sometimes they run to as little as 4,000 words, instead of the 220,000 words needed in FACT. They often produce, however, a rather inefficient object program, but the designers are very much worried about the efficiency of the translation process.

This is the picture, as I see it at the moment, of the differences between the two types of language. What is going to happen in future? Well, first of all, the two types of language must each catch up in the areas where the other type is ahead. It is quite clear, or so it seems to me, that the mathematical languages must extend their range of data description, and that they must be able to become part of a program-operating system. They must also be improved so that it is possible to extend the number of types of object that may be handled within a program.

I think the business languages will have to extend their use of symbols. Their designers will have to get over their silly fright of using the plus sign, and what I can only regard as the sales gimmick of trying to write the program in "English." It is not, of course, English, it is a sort of "computer-ese."

The idea of making simple universal languages which are easy to understand and which make programming easy to the extent that, if you are careful, the whole of programming work becomes unnecessary, is an entirely mistaken idea. It is rather like saying that because we are soon, perhaps, going over to a decimal currency, we shall not then bother to have any accountants; clearly this is complete rubbish.

The real problem about programming is deciding what you want to do and specifying this sufficiently clearly. If you have a good language to program in this merely removes one of the tiresome irritations. There is nothing more tiresome and more likely to cause a number of slips than trying to program in a bad language; it takes a lot of time, and it does not make the real problem any easier.

I think that if the designers of commercial languages were not so frightened of using symbolism and abstract ideas, we should then be developing more towards the sort of thing that Dr. Clippinger was talking about this afternoon. This is obviously the beginning of the process of abstracting the data structure, and this is the interesting thing about commercial languages which is essentially different from anything in mathematical languages.

The next thing I want to suggest is that there are some areas in which each class of language is at the moment weak. I think that the business languages, although they are normally incorporated in operating systems, are rather weak in the specification of operating procedures. That is to say there is considerable confusion when giving a program to a computer and compiling it: sometimes you will wish to load and go, sometimes you will want to run an extra piece of program, or to up date the program by taking in another piece, and sometimes you will wish to put something in and get something else out in the assembly language; all these various stages of actually operating a program, as opposed to sitting down and writing what you want done, are described in very woolly fashion, and generally they are left out of most programming manuals. They form the sort of know-how which you pick up by spending time near the computer and actually operating it. This can cause a great deal of confusion and difficulty in trying to write a programming system.

I think that one of the very important things that the whole of the ALGOL development did, was to clarify and make us think more clearly about what we were writing down in the computer program. I would hope to see in the next ten years or less an equal clarification about what we mean when we instruct a computer to do a job.

Now, on the mathematical side I think there is still a very serious disadvantage and difficulty about ALGOL, and, indeed, all our mathematical programming languages. They are still machine-orientated in the sense that they all assume the sort of machine which has a single control and does one operation at a time, one after the other, generally very fast, and has access to a large, or fairly large, store. It is perfectly possible to conceive a computer with a totally different structure. One can imagine a machine which is a hundred yards long and has fifty processing stations with a loop of magnetic tape circulating continuously between them. Each processing station would be reading the tape, performing some calculations, and putting some results back on the tape, and they would all be operating simultaneously. You could not program a machine like that in ALGOL.

Another feature which I always find very irritating about ALGOL or FORTRAN or Mercury Autocode, or any machine code for that matter, is the fact that I have to specify a lot of wholly unnecessary information. For instance, if I want to do matrix multiplication I have a number of multiplication operations which have to be carried out. The order in which these multiplications are done is totally irrelevant from the mathematical point of view. However, it is not possible to write programs for any computer, in any language I know of, and not have to specify precisely the order in which these multiplications are to be done. If you are faced with translating this program on to a different sort of machine, you may be put to considerable trouble. For example, you may have such a program written for a machine with a large core store, where the natural sequence is to go along a row and down a column. If you are faced with the problem of translating programs like this on to a machine which has either a drum store with access to the matrices by rows, or for which the matrices are much too big so that you have to resort to tapes, you have then to rearrange the order in which the multiplications are done. This has no effect on the result of the program, but it cannot be done automatically in any language I know, because it involves throwing away information which is there in the source language; I think the translator would almost certainly regard any source-language statement as being sacrosanct, and so it is very difficult to remove this information. It seems to me that it is important that some improvements should be made in this respect.

There are, of course, a lot of minor improvements which I shall not bother to go into now, but which would make ALGOL more sensible for operating on existing machines. Some of the present eccentricities were included by mistake, I think, when ALGOL was specified.

There is one other field of use of computers with automatic programming languages which we ought to begin to think about, which is sometimes known as the *real-time* use of computers. This is another way of making a computer look after a number of things at once. My definition of a real-time computer is a machine which spends most of its time doing nothing. It just waits until, at the crucial moment, a demand arises for its attention. It satisfies that demand, and then looks around to see if there is anything else waiting for it to do. A real-time computer can be presented with all sorts of logical problems, not sequentially but in random order. I think this is another feature which will come into programming languages.

I should now like to say something about the problems of writing compilers. We have heard a lot about compilers, how difficult it is to write them accurately and how horrifying it is to find how big they are. Springing from this there is clearly a very strong desire, on the part of people who have to write them, to write as few as possible, to change the languages for which they are writing as rarely as possible, and to change the machines for which they are writing as little as possible, at least in so far as their order codes are concerned.

Now, I do not think that this is more than a temporary state of affairs. I think the problem of how to write compilers is going to get easier, and that we are going to learn more about how to do it, partly perhaps because we shall have languages for writing compilers such as Dr. Brooker was describing, but principally because we shall understand what a compiler does and how it works. When we begin to understand this we shall find it very much easier to specify the job and to do it—although this will still take a little time. What is more, we shall be able to specify the compiler in a language which is more or less machine-independent. You cannot, of course, specify the parts of the compiler which actually produce the machine-code object program independently of the machine on which the object program is to run; the characteristics of the machine must be embedded in the compiler. This is a matter of putting something into the compiler program and not the language you are writing in the compiler program. I think there are some developments already in sight about list-processing which make it very likely that the problem of writing compilers will get considerably easier during the next five years. Again, the problems of changing compilers from one machine language to another, and generally coping with the transformation of language, will, I hope, get a little easier.

I think that perhaps at this point I might say something about standardizing and specifying languages. Commonality between languages and machines keeps cropping up, and I think it is essentially a red herring. Before you can standardize on language in the official sense of the word, you would presumably specify it, and if we are to be able to specify a language we have to think about it. Programming languages are highly intellectual things, and have a very much higher intellectual content than a data-coding system or the specification, say, of a B.A. screw. The content is practically co-terminous with mathematics, and it is a great mistake to think that there is not much in specifying a programming language; it is an extremely difficult operation and a very sophisticated one.

In order to specify a programming language you need informed experts' opinion on the committee which is going to do the specification. I use the word "specification" and not "standardization." Now, there are a number of people who are experts in thinking in ALGOL, and a number of experts in COBOL, but I doubt whether there is anybody who is an expert in both, at least in the sense of being able to make a really informed criticism about the details of each language. I think that it is very unlikely that there is such a thing as *the* programming language, although I think we shall see a number of them. I think it is essential that a language should be specified by people who know what they are talking about. Of course there are experts on standardizing and they are very good at specifying how to standardize, but committees which are chiefly, or wholly, composed of standardizing experts cannot deal with all the sophisticated programming languages; they attempt to do so at their peril.

Much of the pressure for standardizing languages comes from people who have little or no understanding of the problems of using and deciding about programming languages, and who do not really know what they are asking for. Their attitude is "we shall attempt to stick to this in every detail, and we are going to standardize and you had better standardize yourself or get out of the way." This attitude does not impress me very much. I call it the argument from the lemmings—everyone is going to rush down to the sea and drown, and as they are going to do something silly, you had better do it too. (*Laughter.*)

I think it is certainly desirable that all languages which are used should be accurately specified, and this is a very difficult operation. All these committees trying to decide on *the* programming language, are, on the whole, I believe, wasting their time.

The evolution of programming languages will, I think, have a considerable influence on the design of computers, both because of the problem of writing the compilers and also because of the nature of the object program which the compilers produce. One of the characteristics of ALGOL is that the object programs which are produced from it almost invariably use stacks or communication lists between the various procedures that are involved. This is a characteristic of the language, and the natural way to implement it is to use a stack in this sort of way. It is also true, and this is coincidental, that most of the translators for ALGOL also use a stack. This is not the important point. The important thing is that the run-time program uses a stack naturally. I think very probably that this is the sort of thing which will have to be incorporated in computers because the run-time programs require this sort of communication method between their various sections.

Computers will very probably be influenced by the problem of designing compilers to the extent of having more useful instructions for doing list-processing operations. I do not think that this need worry anybody who does not want to write compilers. The only difference it will make for them is that the actual run time of compilers will be shorter.

I think, however, that the whole use of automatic-programming languages will and must influence our attitude towards machines, towards storage space in machines, and to the efficiency and the operating time of machines. I think it is absolutely right and proper to consider that a machine is there in order to save the human beings that use it time and trouble. That is a correct attitude to adopt in considering the use of automatic-programming languages—that they save you time and trouble; your time is worth more to *you* than the machine's, and you have to persuade your boss that really, in the long run, your time is worth more to *him* than the machine's. It seems to me to be absolutely wrong to insist on the last half per cent or the last five per cent of efficiency in run time at the expense of a factor of several times in writing these programs. Writing programs needs genius to save the last order or the last millisecond. It is great fun, but it is a young man's game. You start it with great enthusiasm when you first start programming, but after ten years you get a bit bored with it, and then you turn to automatic-programming languages and use them because they enable you to get to the heart of the problem that you want to do, instead of having to concentrate on the mechanics of getting the program going as fast as you possibly can, which is really nothing more than doing a sort of crossword puzzle.

Automatic programming will have some very important influences on the design of hardware, and the most important ones, I think, will involve the interaction between human beings who are instructing the computer and the computer itself. I think we shall need new methods of using the machines. There has been a very interesting and exciting set of experiments at M.I.T. about using very large computers in a sort of real time-sharing method. In this every user has a Flexowriter directly connected to the machine and can type in his own program at whatever speed he likes. Everyone has the impression that he has the machine all to himself. This is a fascinating idea and when I tell you that the prospect involves, so far as I can make out, a million words of core storage and a machine which operates at one instruction a microsecond, you will see that this gives scope for doing certain things which are rather far beyond the things we are thinking about at the moment.

I think that this is the way in which computers are going to be used in the future, and that languages will have to be adapted to use such methods. Another feature which I think will certainly have to be very considerably improved is the prosaic problem of printing, which involves choice of character sets on printers. If we want to use a complicated and rich language it is very difficult at the moment; some improvement of this is very seriously overdue. I see that this is not the only thing that is overdue; and maybe the end of my talk is overdue, too. (*Applause.*)

The Chairman: Thank you, Mr. Strachey, very much. Well now, the subjects are open for discussion.

Mr. B. Randell (*Atomic Power Division, English Electric*): A question for Mr. Brooker on his phrase-structure language compiler. Could he please tell us something about the state of implementation of this system, and also the experience he has had in using it—for instance, the languages he has translated, or would like to translate, the sort of effectiveness he hopes for with the compiler he produces, and the effectiveness of the object program which such a compiler would itself produce?

Mr. R. A. Brooker (*Manchester University*: *Revised reply May* 1962): The compiler compiler has been written, and at the time of writing a few of its routines have been tested in the 1,024 words of Atlas, which is all the storage that is available at the moment. Eventually we shall have 16K words of core storage and 96K on drums. The compiler compiler language has been used to write a compiler for FORTRAN II and for Mercury Autocode. It is also being used to write an Atlas Autocode, a language which has many of the features of ALGOL, but is rather Atlas-orientated. Harwell have got their own FORTRAN project so we at Manchester may not go ahead with the FORTRAN compiler. The Mercury Autocode translator took about a month to write, including all the facilities for matrix operations and complex number operations.

As regards the efficiency of the compilers written in this way, we must distinguish between the speed of compiling and the speed of operation of the final object program. The latter is quite independent of the former. A compiler compiler is simply a programming system for writing compilers, and just as with any system one can write good programs or bad programs, so with this system one can write good compilers and bad compilers. Generally speaking, however, the compilers will tend to be slower in the speed of translation than conventional tailor-made compilers, because of the recognition-logic employed. The efficiency of the final object program, however, is entirely dependent on the amount of effort put into the compiler, and is not affected in any way by the decision to use a compiler compiler. On the contrary, the facilities of the compiler compiler enable one to test very easily for all kinds of special cases which lend themselves to optimization.

Mr. d'Agapeyeff: I would like first to correct a misconception that my talk has apparently raised, namely that I had suggested that The British Computer Society had defined a standard programming language. In fact, my text read "that The British Computer Society has belatedly set up a committee to consider this problem"—and hence my diatribe against the numerous committees on the subject, since one would have thought it necessary to know what one was aiming at before one began.

Mr. Strachey has gone farther than this and mentioned the dangers of having a standard at all. But in a sense this is a red herring. Manufacturers have got to sell machines in order to build those he wants, and if the salesmen are certain they want and can have something called a standard, then for good or ill that is what we are likely to get.

I should like to congratulate one of the previous speakers, I think he was from the Post Office, because this is the first time we have heard a general user, to my knowledge, get up and ask a real question on automatic programming. The point is the learnability of source language.

In ALGOL you can begin quite gently and treat it as a very simple language. You do not know what a procedure call by name is, so you do not use it. Later you learn a bit more from the manual and gradually you get excited about the language. But with COBOL this is not possible; it is all or nothing. It is very much more difficult with a commercial language to build up slowly, but it was never expected they would get so complicated. The original idea was that they would be simple, and it is very sad to see their current state. Now we must think again.

Finally, I would ask Mr. Brooker what properties of data his system would allow one to declare?

Mr. R. A. Brooker: Well, there is no limit to the types of operations which can be included in a compiler, but as I have already said, the operations which characterize our compiler compiler are those to do with expressions (the internal form of which is a tree), and ordinary red-tape operations with integers.

Mr. G. P. Judd (*English Electric Co. Ltd.*): Would Mr. Brooker consider it possible to write a compiler for ALGOL 60 using his scheme, and if not, what features of ALGOL 60 make this difficult?

Mr. R. A. Brooker. The essential features of the compiler compiler are a means for defining, recognizing, and comparing classes of expressions. Any compiler could be used to write other compilers, the point is that we have thrown emphasis on the above-mentioned facilities in our program, and neglected to include more conventional facilities such as would be used in connection with numerical analysis. It would be perfectly sensible, however, to take any general-purpose compiler and, by including facilities for handling expressions and symbol manipulation, to turn it into a so-called compiler compiler. The short answer is therefore: Yes.

Mr. E. A. Newman (*National Physical Laboratory*): It has always seemed to me that one possible use of a programming aid is as an aid to thinking. I personally find that about as many ideas as I can manipulate at any one time is about ten of varying complexity. Therefore, of course, some sort of programming system is needed to manipulate a limited number of ideas, and this would be a good thing. If the number of ten which I give is the number which I can manipulate, let us assume that most people can cope with not more than a hundred; it would seem on this basis that one does not want too many different sorts of things which you can do, in a given program language, if it is intended to help you to think; so I would say that there is a reasonable case for having a different program language for various different sorts of jobs, and confusion would be avoided in this way.

I wanted to ask Mr. Strachey if this were possible as an idea. One other point I wondered about, is that it always seemed to me that if our bosses thought our time was more valuable than that of the machine, perhaps they would make our salaries higher than that of the machines. I am not sure that they do.

Mr. C. Strachey: I am all in favour of having lots and lots of programming languages, of course. Most people who write a complicated large program for dealing with some kind of group of problems are in fact writing the program language. I think that when we know more about writing compilers, more programs will look like languages instead of multiple-purpose subroutines. I must say that the fact that business languages started off by being ever so simple, and then got unwieldy and complicated, does remind me of the early computing machines which were going to be "ever so simple" to program.

Mr. J. A. Brunt (*English Electric Co. Ltd.*): I want first to take issue with Mr. Strachey, on—really—his whole approach. For example, Mr. Strachey repeatedly uses the phrase "programming language," though I understood this to be a conference on "automatic-programming languages." In other words we are in this conference concerned not with programming as such but with the automation of programming. This is not just a matter of making programming machine-independent. It is a matter of using system languages or procedure-description languages *instead* of programming.

System documentation is the one job that cannot be avoided. The implicit aim of COBOL (perhaps unacknowledged?) is to use the system documentation itself as the automatic-programming input.

This—surely—is why one has all this English floating about in COBOL: because somewhere along the line a system has to be reported in English, and someone has to agree that that is the system. How far one can condense—perhaps on the lines that ICT's COBOL has taken—is a matter to be determined; but even to consider this is an illustration that it is problem documentation which should be at the back of our minds, when we set out to produce an automatic-programming language.

Another point I should like to bring up concerns a specific item of "science fiction" that we have heard, an important matter mentioned by Dr. Clippinger which Mr. Strachey did not take up. I think this is something that we should like to hear more about, and again I will try to put this rather controversially.

What I want to say is this: that linked conditional sentences and compound conditional sentences are Stone Age. These very complex facilities that one has in COBOL 61 just cannot be used, because one cannot get one's mind round them. I would suggest that this is because they try to document, in a sequential manner, logic which just is not sequential. Apart from the strings of instructions which can happily run down the page, there are these other ingredients in problems: what we may call the "decision junctions."

We have heard references to the work of a Systems Committee of CODASYL who are concerned with trying to take this ingredient out of the flow charts and reports, and express the decision junctions in a tabular form. I would say that the use of decision-structure tables is more precise and concise, that the tables are more readable and more readily checked than are these conditional or compound conditional sentences in COBOL. I do not know what other methods may yet be devised, but I do feel that this work on decision-structure tables, and other similar work, is a step in the right direction. It is an acknowledgement that the ingredients in a problem are not all sequential in nature, and cannot all be suitably recorded by conventional flow-chart methods.

Mr. A. G. Fraser (*Ferranti Ltd.*): I should like to ask Mr. Strachey for his opinion on two things. In the first instance, he did mention that he would like to see some information removed from the programs and he gave the order of multiplication as an example. I have found in my work over a very short period that there is indeed a good deal of information which is demanded of the programmer which could possibly be avoided. I find also that there is a large amount of information that the compiler would like to have about the job which it is compiling. One has to decide whether an improvement produced in the object program would justify the extra complexity in the source language. In many cases I find that the compiler has to allow for circumstances which, in the language as it stands now, have in fact never occurred in practice.

I wonder whether he feels that a problem-orientated language would be made simpler by removing program-orientated features, or whether in fact the thing would become more and more complicated because of a necessity to describe the problem in great detail.

The second more trivial point, I think, is that the farther removed that the source language is from the object program, the more work the compiler must do. Compilers themselves may become cheaper to produce but, nevertheless, the amount of work they must do in order to compile a program will increase with the abstraction of programs. I should like to know whether he thinks that it is implied from past experience that work done on a machine will inevitably become cheaper than work done by human beings, or whether there is not a class of work which is unlikely to be done as cheaply by machines.

Mr. C. Strachey: Well, if I may answer the second question first, I am all in favour of more work for computers. I think it is quite possible that there may be types of work which, certainly at the moment and perhaps for a long time in the future, will be more efficiently and cheaply done by human beings. I do not see why we should not envisage a computer system which is a co-operative operation between the compiler and the program writer, so that when the compiler reaches the stage where it wants more information, it asks for it; or it may say that if the programmer can start off by giving the minimum amount of information, it would ask for more if it wanted to be given it. And then, if he cannot be bothered to give it more, it then could make stated assumptions or wait. That I think deals with the first part of the question.

'On the second question: more feedback, not less, between the people who write the programs and the computer would, in my opinion, be a much better thing. At the moment the system of running a machine makes it difficult if you interfere with it or *try* to help it, and I think it ought to be made much easier.

Mr. E. A. Newman: I want to come back a bit on what I was saying before, because of something that has been said since. I submit that human beings are queer fish as far as I understand the concept, because there are some things with which they are very familiar and which they understand as a major concept, whereas they would not understand if there were bits and pieces from which the concept was built up. Anyone in this audience can go along and look at a hundred and fifty women and pick out those that appear attractive. They would understand that perfectly clearly and know what to do, but try to make a break-down of that for a computer program, with all the necessary instructions, and they would not recognize it at all. There are other things which have to be looked at simply as concepts, such as "is two bigger than four?" Now, it seems to me that one of the things that any sort of programming aid has to do, is to find ways and means of making it so that human beings can present the problems they have in some sort of unit concept. If the unit concept happens to be something which is basically very complicated, but with which they are familiar, this has a unit conception. This is particularly true in business languages, I should have thought, in so far as one could make a business language which will let people give their programs in this manner, with an assortment of complex and simple ideas which the computer is going to sort out. This idea for a computer is not only easy for the programmer to use, but it is also one which makes people think in their own programs. This is very important because a lot of us find it very hard to think how to do a job in programming.

Mr. P. Wegner (*London School of Economics*): In connection with the problem posed by the last speaker, I feel that the computer could solve this problem relatively easily if it had a one-track mind.

My question is addressed to Mr. Strachey and is concerned with his statement that ALGOL is machine-orientated. Languages must inevitably be machine-orientated if they are to run on a particular machine. However, we can insulate the language from the machine by including within the language some vocabulary for specifying the characteristics of the computer on which a program is to be run. Such characteristics can include memory size, types of input-output equipment, and the degree of precision of the computer word. Current commercial languages, such as COBOL and NEBULA, have explicit vocabularies for machine description. The best way to achieve machine-independent languages seems to require explicit declarative machine-dependent statements in part on the language, so that the rest of the language is truly machine-independent. I wonder whether Mr. Strachey would agree with this approach to machine-independence, or whether he feels there is an alternative approach?

Mr. C. Strachey: I think maybe I did not make myself quite clear when I said that I thought that ALGOL was a machine-orientated language. I do not mean orientated to a particular machine, or a particular type of computer mechanism. There are languages which are not orientated to any specific one of these, and I think one of the great difficulties about translating languages which are machine-orientated in the ALGOL sense is that it is very difficult to get over more than minor variations in what COBOL calls the environment. As Dr. Clippinger said earlier, the introduction of discs into the Honeywell computer makes it very difficult to cope with the program translator, and the translator does not in fact deal with this. This seems to me to be all that one can say about automatic commonality across machines of very different types, by which is generally meant different organizations of stores. More levels of storage present problems—not merely problems of translation but problems of redesigning. This is usually outside the scope of any translator.

Mr. L. R. Greybourne (*Elliott Brothers* (*London*) *Ltd.*): There are two points I would like to raise with the speaker. First, how is it possible to devise a non-sequential method of use for a device which is inherently sequential in its nature?

Secondly, what you appear to require is a machine, which, for all practical purposes, is a calculating robot with the capacity to evaluate accurately and understand correctly the wishes of its human user. I am sure that the manufacturers of computing machinery would agree that such a device is very desirable. With respect, sir, could you suggest any practicable method of approaching the design problems involved?

Mr. C. Strachey: The problem is that I am not quite sure which of my remarks you are referring to about wanting to have machines which are much easier to use. This is clearly an old saw, and everybody says this. So far as the problem of irrevelant sequencing is concerned, I have in fact a way of dealing with this. It involves using a different form of programming language. This is nothing to do with the manufacturer or the computer; it is simply a question of persuading the people doing programming to do it in suitable language.

Mr. G. A. Gibson (*International Computers and Tabulators Ltd.*): I feel that Mr. Strachey and one or two of the speakers this afternoon have been chasing a bit of a red herring. One of my articles of faith is that a computer can only do what it is told to do. It can do it faster and it can do it more accurately, we hope, but it can only do what it is told, yet some of the speakers today have seemed to me to be seeking for problem-orientated languages. This I feel is impossible. We can get only procedure-orientated languages. We can phrase a procedure-orientated language so that it looks as if it is problem-orientated. For example, in the problem of matrix multiplication we say "multiply two matrices" and the compiler looks at the size of the matrices and decides which way it should be done; but I feel that we must always have such a procedure-orientated language, and problem-orientation is an impossibility.

Mr. C. Strachey: I do not think I ever actually said that I wanted a language which simply stated the problem. I should like to suspend judgement as to whether certain classes of problem might not be stated in problem-orientated language in your sense, and solved satisfactorily by programming systems. All I did actually say was that I wanted methods of describing what we wanted done, which did not involve using a lot of unnecessary things we did not really care about.

Mr. J. Harwell (*Honeywell Controls*): Mr. Strachey seemed to support many different languages, which I think would meet the needs of many different users, and might also meet the case of people using very different machines. Would he solve the standardization problem and support having one standard language for writing compilers?

Mr. C. Strachey: No.

The Chairman: Time is getting on. If the comments are as brief as the answer to that last question we could make the best use of the remaining few minutes.

Mr. F. Duncan (*English Electric Co. Ltd.*): I agree that we ought to have a lot of programming languages. The trouble at the moment is that there just is not even one. (*Laughter.*)

The Chairman: I should like to bring the proceedings to a close by thanking Mr. Strachey and Mr. Brooker for their excellent contributions this afternoon, and all those who have taken part in the discussion. Thank you very much. (*Applause.*)

(The Conference closed at 4.58 p.m.)

Book Review

Computer Handbook, by Harry D. Huskey and Granino A. Korn, 1962; 1220 pages. (London: *McGraw-Hill Publishing Co. Ltd.*, £9 14s. 0d.)

It is difficult to know how to review an encyclopaedia—which is what the present work really is, although the articles are arranged systematically instead of alphabetically. Half the book deals with analogue computers and the other half with digital computers. Evidently the former was primarily the responsibility of Dr. Korn and the latter the responsibility of Dr. Huskey. Neither Dr. Korn nor Dr. Huskey have been content with exercising editorial supervision alone, but they have themselves each contributed to a number of sections. They have enlisted the help of some 70 authors, all of them highly competent in their fields, many being leading figures.

The difficulty of keeping such a large quantity of material up to date during the process of compilation must have been a very serious one, and yet I can only say that it has been successfully solved. Rarely does the material or treatment strike one as being dated—nearly all the circuits, for example, are transistorized, and little space is wasted on obsolete techniques such as the use of mercury delay lines for storage. Material of strictly historical interest is kept to a minimum. All the material appears here for the first time with one or two small exceptions: one of these is an article on ALGOL 60 by M. Woodger, which first appeared in *The Computer Journal* in July 1960 and which is reproduced by permission.

I would not go quite so far as to endorse the authors' claim in the preface that, in both the sections dealing with analogue and digital computing, sufficient detail is presented so that anyone competent in the field can proceed to construct a computer, or having a computer can proceed to use it. This may be more true of the analogue section than of the digital section. In the latter I note, for example, that the important subject of software is not treated at all fully. Nevertheless, the book does contain a large amount of current and well presented information, and its appearance is to be welcomed.

Reading this and other books that have appeared recently one wonders why it is that publishers consider that analogue and digital computers should be treated in the same volume. Surely the two subjects have very little in common once one goes below the surface. Indeed, if you put analogue and digital people in the same room, they usually begin to quarrel, although perhaps not quite so violently as they did a few years ago. The present volume would have been much better published in two separate parts. Each would have been lighter, cheaper, and more likely to come into the personal possession of those engineers who would really use it. There will surely be a second and enlarged edition of this book, and I would seriously urge the publishers to consider issuing it in two volumes, or as two separate books.

M. V. Wilkes.