OMNICODE—A COMMON LANGUAGE PROGRAMMING SYSTEM

By
RUSSELL C. McGEE

Reprinted from Monograph No. 3
JOURNAL OF THE FRANKLIN INSTITUTE Series
April, 1957

Third Session

OMNICODE—A COMMON LANGUAGE PROGRAMMING SYSTEM

BY

RUSSELL C. McGEE[1]

The crux of what has been done in the past has been the introduction of a third language into programming—the first two being the language of the machine and the language in which the problem is formulated. (The following analogy is illustrated in Fig. 1.) Until very recently programmers have been like an American who can speak German who finds himself with a Frenchman who can speak Russian. In order to communicate with the Frenchman, the American must find a German who can speak Russian. It would be simpler if the American would learn to speak French or the Frenchman English but, of course, the American would rather have the Frenchman learn to speak English than learn French himself. Similarly, programmers would like computers and data-processing machines to understand the language in which their problems are formulated.

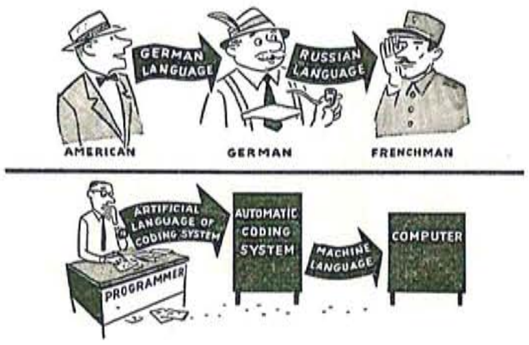


FIG. 1.

It is toward this end that we are working at Hanford. And so for purposes of our discussion today, I should like to define automatic programming as follows:

1. The development of a way of writing programs (a language) which is closely akin to the language in which problems are formulated.
2. An assembly routine for translating this language to machine language.
3. Accomplish 1 and 2 so that the expenditure in time, money and manpower from problem origination to solution is minimized.

[1] Analyst, Automatic Programming, General Electric Company, Richland, Washington.

Now these objectives have to be evaluated in terms of the types of problems one wishes to solve. A system which is useful in one installation is not necessarily useful in another because of differences in applications. Let use see how these objectives have been evaluated at Hanford.

All scientific and commercial calculations and data processing are performed at Hanford by a service organization—the Data Processing Operation. Since June of 1955 we have operated an IBM 702 and in August of last year we acquired a "650." The "702" is equipped with 20,000 characters of core memory, a 60,000-character magnetic drum, nine tape units, a card reader, card punch and two printers. The "650" is used exclusively for scientific calculations and hence does not have the alphabetic device or other optional features.

We handle a wide variety of applications—from payroll and inventory control to reactor design calculations and meteorological data reduction. Figure 2 shows the distribution of programs by program type on the "702" during a typical month.

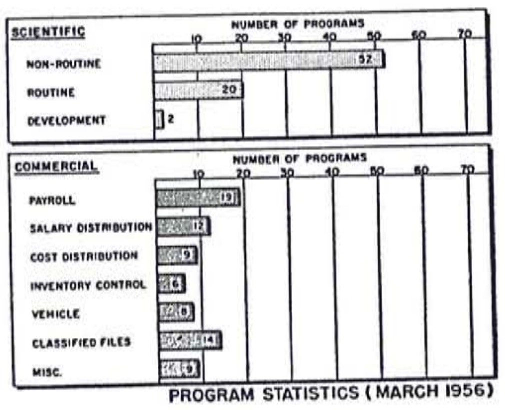
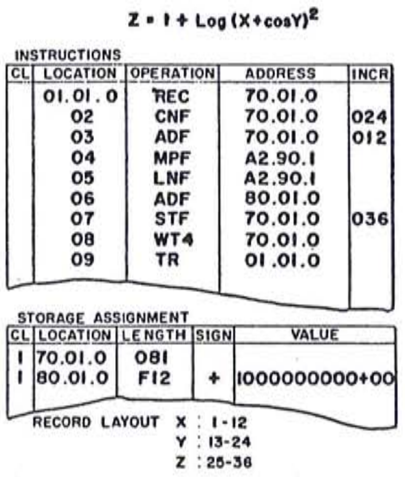


FIG. 2.

In view of the dual nature of our applications, it might appear that we need two automatic programming systems—one for scientific and the other for commercial problems. However, it has been proven to our satisfaction that there are noteworthy advantages in making a system sufficiently comprehensive that it may be used for all the applications in the installation.

This proof was arrived at through the use of SCRIPT—a system developed in 1955 by Mr. C. E. Thompson of General Electric Company and Mr. John Jackson of IBM. Figure 3 is a sample of a SCRIPT program. The name SCRIPT is an acronym derived from the title Scientific and Commercial Sub-Routine Interpreter and Program Translator. As the name implies, the system was developed for use on a wide range of applications. Although SCRIPT has been a successful system, one is still aware when using it of the artificial nature of the symbolic code. Symbolic addresses have no intrinsic "personality" and although provision is made for describing the function of each instruction and storage assignment, the description is often inadequate or omitted entirely in the haste of writing the program. Therefore, after SCRIPT had been in use for several months we turned again to the problem of finding a language better suited for writing programs at Hanford.

The work of designing a language was started shortly after the conversion of our work from Electric Accounting Machines to the "702" was completed. It is not surprising then that we soon started thinking of the language as one we would use for writing programs for machines other than the "702." If we ever changed machines again, conversion would consist of re-writing one routine—*viz.*, the assembly routine. Then all of the old programs could be reassembled on the new machine with a relatively trivial



$Z = 1 + \text{Log}(X + \cos Y)^2$

INSTRUCTIONS

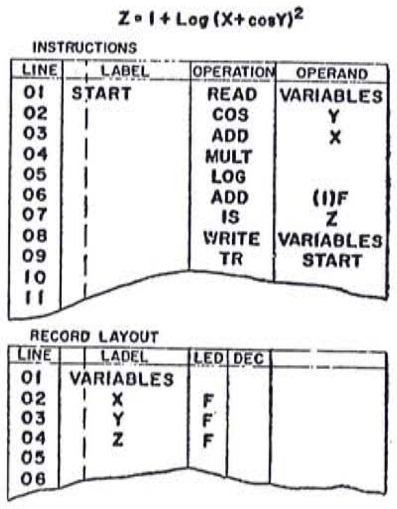
CL	LOCATION	OPERATION	ADDRESS	INCR
	01.01.0	REC	70.01.0	
	02	CNF	70.01.0	024
	03	ADF	70.01.0	012
	04	MPF	A2.90.1	
	05	LNF	A2.90.1	
	06	ADF	80.01.0	
	07	STF	70.01.0	036
	08	WT4	70.01.0	
	09	TR	01.01.0	

STORAGE ASSIGNMENT

CL	LOCATION	LENGTH	SIGN	VALUE
1	70.01.0	081		
1	80.01.0	F12	+	1000000000+00

RECORD LAYOUT X : 1-12
Y : 13-24
Z : 25-36

FIG. 3.



$Z = 1 + \text{Log}(X + \cos Y)^2$

INSTRUCTIONS

LINE	LABEL	OPERATION	OPERAND
01	START	READ	VARIABLES
02		COS	Y
03		ADD	X
04		MULT	
05		LOG	
06		ADD	(1)F
07		IS	Z
08		WRITE	VARIABLES
09		TR	START
10			
11			

RECORD LAYOUT

LINE	LADEL	LED	DEC	
01	VARIABLES			
02	X	F		
03	Y	F		
04	Z	F		
05				
06				

FIG. 4.

amount of effort. Also, if we ever acquired additional machines, work could be shifted from one machine to another as the schedules on the individual machines dictated. Training is also substantially reduced by using a common programming language. One teaches a programming system instead of a machine.

In view of these considerations, our objective became to develop a common language for writing programs for solving scientific and commercial problems for any present and future machines at Hanford. This work was begun in earnest about May of last year. What we came up with was OMNICODE—something closely akin to ordinary English. Figure 4 shows the evaluation of a simple equation in OMNICODE in its original version.

The record layout form describes a record named VARIABLES which is being introduced into the program. The form tells us that each record by this name contains fields called X, Y and Z, each of which is a floating point number. The program says:

> *Read* the record *Variables*
> Obtain the *Cosine* of Y
> *Add X*
> *Multiply* the previous result by itself
> Obtain the *Log* of the previous result
> *Add* a floating point *1*
> The result *IS Z*
> Then *Write* the record *Variables*
> and *Transfer* to the *Start*

The example illustrates the method of introducing constants into a program. When we want to add 1, we merely enter the numeral in parentheses in the operand column and follow it by an F indicating floating point.

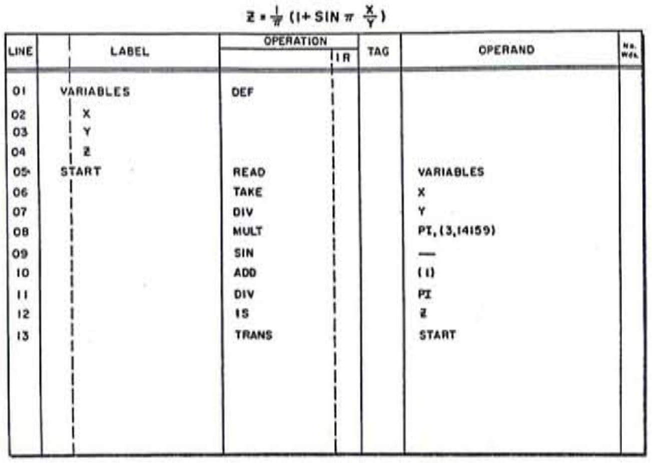
Note also that the only instruction given a label is the one referred to. Others could be given labels but it is not necessary. Instructions are given names just like values and referred to by these names.

This example is just a glimpse of what we planned in the original version of OMNICODE. In summary we proposed the following:

1. Functional storage layout forms for the three basic input/output media—cards, tape and printed reports.
2. Assignment of English language names to operations, instructions and storage.
3. A functional operation vocabulary designed to satisfy the programmer's needs.
4. A single operation vocabulary for both fixed and floating point commands.
5. Provision for defining constant and working storage within the program.
6. Provision for automatic restarts and check points as a part of every assembled program.
7. Provision for automatic address modification by simulating index registers.
8. Automatic decimal control for fixed decimal arithmetic.
9. Multiple levels of labeling to assist in the preparation of large programs which normally require partitioning.

This work was interrupted in June of 1956 when it was decided that a "650" would be obtained to handle some of Hanford's non-routine scientific calculations. Because our own staff of programmers was already working at full capacity and because non-routine scientific calculations occupied a disproportionate amount of our manpower, it was decided that the "650" would be customer programmed and operated. This meant we had in the first place to make this prospect attractive to our customers and then we had to train them. In view of this and our over-all objective of having a common language system we decided to prepare a "watered down" version of OMNICODE for the "650."

The "watering down" of the original version was not an unwelcome prospect. The over-all project had assumed such proporations that it was difficult for one person to comprehend in its entirety. So we were actually grateful for the opportunity to discard some of the "bells and whistles" and concentrate upon a system which would be a pilot model of OMNICODE. This would allow us to test our original hypotheses without the complicating influences of variable word length and multiple input/output units. However, it was clear from the outset that the data-processing ability of the "702" would be needed to perform the assemblies.



$$Z = \frac{1}{\pi}\left(1 + \text{SIN}\,\pi\,\frac{X}{Y}\right)$$

LINE	LABEL	OPERATION	IR	TAG	OPERAND	No. Wds.
01	VARIABLES	DEF				
02	X					
03	Y					
04	Z					
05	START	READ			VARIABLES	
06		TAKE			X	
07		DIV			Y	
08		MULT			PI, (3,14159)	
09		SIN			—	
10		ADD			(1)	
11		DIV			PI	
12		IS			Z	
13		TRANS			START	

FIG. 5.

Figure 5 is a sample of 650 OMNICODE—the pilot version of OMNICODE. The system has been in use since October. It is a floating decimal interpretive system with simulated index registers. Through its use one has at his disposal a vocabulary of 83 commands as compared with the 35 operations built into the "650."

One notices immediately the similarity between this and the original version of OMNICODE. However, we have done away with auxiliary forms. The entire program including record definition is written on the Programming Form. The DEFINE instruction has been included to make this possible. The LABEL portion of the form is used for the introduction of names—names of records and fields as well as instructions. In our example, we define the record VARIABLES by writing VARIABLES against the left margin and entering DEF in the operation column. Fields are written one space to the right of the left margin and have no operation.

On line 08 we have an example of introducing a constant by its name. The first time we see the constant PI, we enter it with its value. From that point on, we merely write PI as on Line 11. One should note that we do not specify values as floating point since this has been specified in the definition of the system. Also, we have adopted the convention of drawing a line when an operation refers to a previous result. The line is ignored during keypunching but it serves to distinguish valid operands from unfinished instructions while the program is being written.

The use of index registers is illustrated in the vector by matrix multiplication of Fig. 6. Here we are dealing with records all of whose contents are referred to by the same name. This is quite acceptable. All we need do is indicate the number of words to be given this label so that adequate storage will be reserved. In the third instruction, the value zero has been

$$Y_J = \sum_{I=1} A_I X_{IJ}$$

LINE		LABEL	OPERATION	IR	TAG	OPERAND	No. Wds
01	A		DEF				3
02	X		DEF				3
03	Y		DEF			(0)	5
04			READ			A	
05			INIT	J	1		
06			READ			X	
07			INIT	I	1		
08			TAKE		I	A	
09			MULT		I	X	
10			ACC		J	Y	
11			TEST		I	3.	
12			TEST		J	5	
13			PUNCH			Y	
14			HALT				

FIG. 6.

entered in the operand column. This causes all the storage assigned to this record to be set to zero when the program is loaded. Any other value could have been used as well.

A general remark seems in order regarding the tag column. It is used to specify that one in a sequence of things to which reference is being made. So in the fifth instruction we intend to say, "As of the time this instruction is performed, consider only the first in the sequences of things tagged with J," even though in plain English it is easier to say, "Initialize J to 1."

The program of Fig. 6 can be read, then, as follows:

Read a record *A*
*Init*ialize *J* to *1*
Read a record *X*
*Init*ialize *I* to *1* (*Continued*)

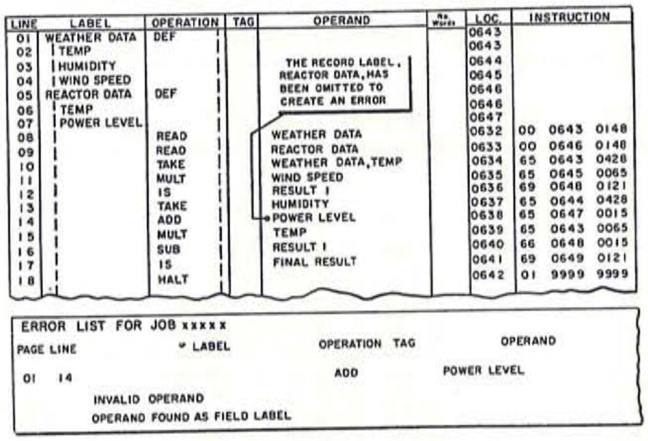
```
Take the Ith A
Multiply by the Ith X
Accumulate the result with the Jth Y
Test to see if I equals 3
     If it does not, increment I by 1
     and transfer to the instruction
     following the last INIT I.
     If it does, proceed to the next
     instruction.
Test to see if J equals 5
     If it does not, increment J by 1
     and transfer to the instruction
     following the last INIT J.
     If it does, proceed to the next
     instruction.
Punch the record Y
Halt
```

LINE	LABEL	OPERATION		TAG	OPERAND	No. Wds
			IR			
01	WEATHER DATA	DEF				
02	TEMPERATURE					
03	HUMIDITY					
04	WIND SPEED					
05	REACTOR DATA	DEF				
06	TEMPERATURE					
07	POWER LEVEL					
08		READ			WEATHER DATA	
09		READ			REACTOR DATA	
10		TAKE			WEATHER DATA, TEMP	
11		MULT			WIND SPEED	
12		IS			RESULT I	
13		TAKE			HUMIDITY	
14		ADD			REACTOR DATA, POWER LEVEL	
15		MULT			TEMPERATURE	
16		SUB			RESULT I	
17		IS			FINAL RESULT	
18		HALT				

FIG. 7. Illustration of two-level reference.

Figure 7 illustrates the two level method used in OMNICODE for referring to operands. One must give names to the fields of a record which are unique within that record, but he may use the same name in different records as often as he chooses. This brings us to one of the cardinal rules of OMNICODE—*viz.,* whenever reference is made to a field, the field label must be preceded by its record label if the field does not belong to the most recently referred to record. Hence, on Line 10 we must refer to the record Weather Data as well as the field Temp. If we did not do this we would take the field Temp. from the record Reactor Data since Reactor Data is the most recently referred to record. However, note that once we have made reference to the record Weather Data it is not necessary to refer to a record label again until Line 14 where reference is made to Power level—a field in the record Reactor Data. This figure also illustrates how one introduces working storage in OMNICODE. In line 12, the label Result 1 is seen for the first time. Since this is an IS instruction OMNICODE will reserve a word of storage and associate with it the name Result 1.

Now let us suppose that the record label had been omitted in this example. Figure 8 shows what the assembly routine would produce if this error were committed. One will observe that the correct data address, 0647, was provided in the erroneous instruction. The assembly routine was able to make this substitution correctly because Power Level is a unique field label. However, notice that the incorrect temperature is now being referred to on Line 15. Because errors of this type can result in incorrect addresses being assigned and because in many cases the assembly routine is never



```
LINE  LABEL           OPERATION  TAG  OPERAND               No.    LOC.  INSTRUCTION
                                                            Words
 01   WEATHER DATA    DEF                                          0643
 02    TEMP                                                        0643
 03    HUMIDITY                                                    0644
 04    WIND SPEED                                                  0645
 05   REACTOR DATA    DEF                                          0646
 06    TEMP                                                        0646
 07    POWER LEVEL                                                 0647
 08                   READ            WEATHER DATA                 0632  00 0643 0148
 09                   READ            REACTOR DATA                 0633  00 0646 0148
 10                   TAKE            WEATHER DATA,TEMP            0634  65 0643 0428
 11                   MULT            WIND SPEED                   0635  65 0645 0065
 12                   IS              RESULT 1                     0636  69 0648 0121
 13                   TAKE            HUMIDITY                     0637  65 0644 0428
 14                   ADD             POWER LEVEL                  0638  65 0647 0015
 15                   MULT            TEMP                         0639  65 0643 0065
 16                   SUB             RESULT 1                     0640  66 0648 0015
 17                   IS              FINAL RESULT                 0641  69 0649 0121
 18                   HALT                                         0642  01 9999 9999
```

THE RECORD LABEL, REACTOR DATA, HAS BEEN OMITTED TO CREATE AN ERROR

```
ERROR LIST FOR JOB xxxxx
PAGE LINE           * LABEL          OPERATION  TAG          OPERAND

 01   14                             ADD              POWER LEVEL

          INVALID OPERAND
          OPERAND FOUND AS FIELD LABEL
```

FIG. 8. Example of an erroneous program.

able to find the specified operand, an error entry is printed for each detected error following the assembly listing. A sample error list with the entry for the instant case is shown at the bottom of Fig. 8. The point we wish to make here is that even though a person violates the rules of OMNICODE, the assembly routine will do what it can to produce a correct program and then print a message describing any difficulties it has encountered. The error we have described in this example is just one of thirteen different error types that the assembly routine is able to detect. As we gain experience with the system we will probably find additional programming errors that can be detected during assembly. We feel that if a problem is written in OMNICODE and after assembly all the errors indicated on the error list are corrected, then the only errors left in the the program should be logical errors. So far, our experience has borne this out.

Before looking to the future of OMNICODE let us summarize what we have accomplished with this pilot model:

1. We have arrived at a language which we feel is well suited for writing both scientific and commercial applications. We can take very little credit for this; for, after all, the language we are using is ordinary English with certain constraints applied to it by a programming form.
   While we are on the subject of language, it might be of interest to digress long enough to point out that with the exception of the operation, one can write both symbolic and actual programs in OMNICODE. That is, the names that one gives to instructions and values are arbitrary and could hence be symbolic or actual addresses as well as English Words. However, this procedure is not recommended.
2. We have developed an assembly routine for the "702" that will translate OMNICODE programs to "650" programs. If I may be forgiven for endowing the machine with intelligence, we may say that we have taught the "702" to understand English.
3. We have substantially reduced the problem of training new personnel. We have trained 45 engineers, physicists and mathematicians in 36 hours of classroom instruction. Most of these people had never seen a computer before taking our courses but were able to write, debug and run their own problems on the "650" after this small amount of training. The training period will probably be reduced when we become more familiar with OMNICODE and as more comprehensive literature on the system becomes available.
4. Debugging has been reduced to finding the logical errors in a program.
5. The problems of documentation and communication that were present in symbolic and actual codes have been greatly lessened because OMNICODE programs are self-descriptive. Now one has no choice in the matter of describing each instruction, because an instruction and its description are synonymous. For the same reason, it is a simple matter for one programmer to read another's code.
6. Perhaps the most important thing we have done has been to lay the foundation for 702 OMNICODE. We have shown that assembly of completely literal programs is a practical process and, in fact, uses very little more machine time than it takes to assemble the same programs written in a symbolic code.

This brings us to the topic: Where do we go from here? Well we certainly will return to the project we started out to accomplish—*viz.*, OMNICODE for the "702." So far as we can now see, there is nothing about the language we are using that would make it unsuitable for any present or future problems. Hence, we will look forward to writing OMNICODE assembly routines for machines we may acquire in the future as well as those we now have.

However, assuming that future versions of OMNICODE are completely successful, we will have solved only half of the problem of communicating with the machine in English. Although we are confident that debugging will be reduced in future versions of OMNICODE, as it has been in the present version, we are not so naive that we think debugging will become a thing of the past. Hence, in order to complete the picture, we should be able to derive diagnostic routines whose output is in the same language in which the program is written. Or (if I may again be excused for over-endowing the machine with intelligence), we might say: We have taught the machine to understand English, now let's teach it to write English. This is perhaps a very ambitious proposal but it deserves a feasibility study at least.

Indeed, we have little choice but to give serious consideration to such methods. As machines become faster and are equipped with larger memories and more comprehensive instruction vocabularies, debugging programs in machine language becomes increasingly difficult. This is true partly because machines with greater capabilities will work on more complex problems and also because the increased complexity of their addressing and operation code structures makes the machine language codes more difficult to read and understand. This argument is particularly pertinent if a common programming language is to be adapted. The benefits of the common language will be only partially realized if programmers must also learn the languages of every machine they work on. Hence, we at Hanford are looking forward to the day when we will use a truly common language coding system—a system common to all applications, common to all machines and finally, common to debugging and program writing.

DISCUSSION

Moderator John W. Carr[1]: After hearing a definition of automatic programming, I should like to give you an extension of the automatic programming definition in rather crude terms, as to the direction in which I think the trend will go from here.

First, the computer must, in addition to the job you have heard about today, be able to do the job of the automatic programmer. It must be able to manipulate its own language; it must be able to understand its own language; it must be able to add symbols to its own vocabulary that it will find of use to itself. Beyond this, it must be able to make, of its own volition, comparisons between elements or structures within its own memory, and therefore it must be able to search through its own memory with what will turn out probably to be some sort of inductive manipulation. I think the entire information which is available to the system now consisting of programmer and computer must be made available to the machine alone. That is something that will come.

If any of you are interested, I think there are two papers that will give ideas about possible directions: first a paper by Newell and Simons, in the I.R.E. *PGIT Transactions,* with respect to a program for proving the theorems inductively. It is not a regular deterministic procedure in the classical sense of the word; it is a sort of hit-and-miss, but nevertheless, successful heuristic program, being done on the Johnniac. This procedure makes use of the Rand machine at Santa Monica in a novel fashion, an un-standard fashion, from the point of view of programmers. I understand the machine has now proved some 47 theorems from *Principia Mathematica.*

The second paper is one by Solomonoff, dealing with "Artificial Intelligence," which has not yet been published except in a private format. It will be given at the I.R.E. Convention this spring. This is not advanced learning, but it is interesting that somebody has tried to do this. I think those papers are worth noting.

Mr. Robert Serrell[2]: Am I correct in understanding that no part of OMNICODE as at present constituted can be used on the "650" without a "702" attachment?

Mr. McGee: That is correct—and I might say we are not in "cahoots" with IBM, trying to sell the "702" s. It is just the way the situation worked out.

[1] Mathematics Department, University of Michigan, Ann Arbor, Mich.

[2] Research Staff Member in Charge of Computation Laboratory, Radio Corporation of America, RCA Laboratories, Princeton, N. J.

MR. DAN C. WILKINS[3]: I really have a two-part question. I was wondering how much storage is actually required on your "702" for the "650" symbolizer. The reason for that question is this—do you think it will be feasible with a full-blown "650" with tapes, etc., on it, to be able to symbolize its own program without going to the "702"?

MR. McGEE: Our "650" is a stripped-down model.

MR. WILKINS: What I mean is, with what you know of how you use the "702" to do your symbolizing, do you think that with a tape attachment on the "650" it could do its own, without the "702"?

MR. McGEE: I think you could, probably without great difficulty. Of course, the problem is to find sufficient storage for the tables of literal *versus* actual addresses. That is, literal names *versus* actual addresses would have to be built up. We use our drums for that purpose. However, I should imagine tapes would work out all right, because there are, generally speaking, not as many English names introduced into the program as one might expect. I think the answer to your first question would be yes.

DR. CARR: I might comment here that you will hear a talk this afternoon on how the Carnegie Tech compiler for the type "650" does its job. Because the type "650" has a restricted storage, this compiler doesn't have literal addresses. You have Y_1, Y_2, etc.; you can have as many subscripts as you want. One cannot use all the letters of the alphabet in symbolic addresses, since the group that wrote the compiler didn't have room to use all letters if they had wanted to.

MR. FRANC A. LANDEE[4]: I was curious about the vector matrix multiplication, which is the usual way of solving simultaneous equations. Why don't you have just one command, saying "multiply"?

MR. McGEE: That would be the thing we would do if we had a lot of matrices or matrix work, at Hanford. Even if we had, I think we would still use this example to illustrate the use of index registers. Certainly, otherwise, we would have a set-up of subroutines.

MR. LANDEE: When you go to completely symbolic addressing, have you any idea with what efficiency you use the storage? The storage on the "650" is only 2,000 memory spaces. I have an idea that completely symbolic addressing using your storage is somewhat less efficient than if you actually wrote out the address locations.

MR. McGEE: In the first place, a literal name will never get into the "650." You understand that. In other words, only actual "650" instructions will ever get into the "650." Corresponding to every OMNICODE

[3] Mathematics Analyst, Temco Aircraft Corporation, Dallas, Texas.

[4] Head of Computation Laboratory, The Dow Chemical Company, Midland, Mich.

instruction that one writes, you will get one "650" instruction, generally; in some cases more—so I don't see that instruction storage would be any greater than if you were writing in actual. Now, number storage, that is to say, input-output storage, would not be any greater except for the fact that we don't allow overlapping of records. That is to say, we couldn't read a record *B* on top of a record *A*. They must be in separate storage locations.

Now, as to working storage: if you wanted to put something away, called Result 1, as we did in the example, it would be assigned one storage location. If you had a Result 2, it would be a different one. But it is only in these minor areas where the planner might have overlapped things that you would use storage, I feel.

MR. A. M. PEISER[5]: I wanted to raise another question concerning efficiency. We have talked a lot about how automatic coding is improving the efficiencies of programming, but we have said little about how it is affecting the efficiency of the computer. I am thinking primarily of computer speed. I would like to ask whether you feel OMNICODE is using the "650" efficiently and if not—what balance should we aim for between programming efficiency and computer efficiency?

MR. McGEE: I can give you some quantitative information. First, we feel if we are going to be doing scientific calculations, we are going to be working in floating point; hence, we will have to use floating point subroutines. Most of the time will be spent in floating point subroutines. These will be optimized, etc. In OMNICODE we use an interpretive system, as mentioned, and since we are doing that, no attempt is made to optimize the locations of pseudo-instructions.

However, notwithstanding all of this, we spend less than 20 per cent of our time doing non-productive things such as interpretation, normalizing results of operation, and so forth. So, I really don't feel that this is a great price to pay in view of the fact that we have saved so much in training. Actually in our location—as I mentioned—it was only through the use of a system like this that it was possible for us to have customer programming without additional expense in training. In addition, we have had very good success in having correct programs written. Most people have one or two debugs, and in some cases none. I feel of course that the balance is in favor of letting the programming system do the work.

MR. MARVIN SENDROW[6]: You spoke of OMNICODE as for both scientific and business-handling problems. The whole lecture was taken up with scientific matters. I wondered if you would say more of the business systems.

[5] The M. W. Kellogg Company, New York, N. Y.
[6] Engineer, Radio Corporation of America, Philadelphia, Pa.

MR. McGEE: Well, we will probably continue to do what we are now doing. We will not do any business applications on the "650." Right now we do on the "702" a 7000-man payroll, and inventory controls. We have an inventory-control system on the machine, a cost system, and we accounted for all the classified documents at Hanford. The latter is quite a large application. I am not too familiar with the particular applications. These are just some I can think of.

MR. SENDROW: What I meant was—can you give examples of the language you are going to use in OMNICODE for the business applications?

MR. McGEE: We have planned to use the same or a very similiar language. Of course, I think every time you do something for the first time, having done it you want to start over again. You now have found how you wanted to do it in the first place. In our work in OMNICODE I don't think it is any different. There are changes we would like to make. However, we feel in general that we have arrived at the language we would like to use. If I may answer one of the questions asked yesterday with regard to today's topic:

Questions were asked about mis-spelling and mispunctuating. We are doing what I think may be helpful along these lines. Whenever we encounter a new literal operand, a new name, we remove all the blanks and all the periods from the name before we store it in a table. So that if during key-punching, a girl misunderstands the way something should be spaced, or if the person writing the program writes "TEMP" one place and "TEMP." (with a period) somewhere else, these common differences in form will not give the assembly system any trouble. I think this might be a useful thought for helping to solve this problem of mis-spacing, mis-spelling, and mispunctuation.