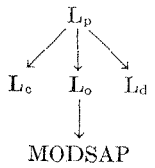
The CL-I System has proved to be very successful, not only in dealing with large-scale information-processing problems but in providing intercommunication between programmers and/or programs. As a result of our experience with CL-I, we can make the following statements about how well it fulfills the requirements of the "programming system" set forth in this paper:

1. The decoupling of data descriptions and programs has proved extremely valuable, not only in the initial design and coding of programs, but also in their subsequent extension and modification. For example, there have been several occasions when production programs built with CL-I have required several extensions and modifications which, had we been using conventional techniques, would have taken several man-months of effort. These were accomplished literally overnight with CL-I.

2. The file for programs and data has been very valuable in solving many of the logistic problems inherent in the building of large programs where many programmers are involved.

3. One feature of CL-I is not adapted for certain types of problems, in particular, the manner in which the system itself is organized. That is, CL-I is organized in a hierarchal fashion in the sense that there are "levels" of languages. Further, it is not possible to intermix languages in any code sequence, except that MODSAP segments can be imbedded in an L_0 code under certain conditions. Because of its hierarchized organization, the CL-I System is awkward to extend.



In short, we feel that the CL-I System represents a big step forward in the implementation of large-scale information-processing problems. However, CL-I does not constitute our ultimate goal. Indeed, with the experience gained both in the construction of CL-I and in nearly two years of use, we feel that we can now make another long step forward. Thus, a successor to CL-I, the CL-II System, is currently being designed.[3]

[3] See CHEATHAM, T. E. JR., and COLLINS, G. O. Programmed parallel computation sequence, ACM National Meeting, 22 August 1960; CHEATHAM, T. E. JR., Preliminary design specification for the CL-II programming system, Section II, Technical Operations, Inc., TO-B 60-23, June, 1960.

The Internal Organization of the MAD Translator

B. W. Arden, B. A. Galler, R. M. Graham

The University of Michigan, Ann Arbor, Michigan

Introduction

This paper describes the internal structure of a translator for the MAD language. The language is very similar to that of ALGOL 60, although it is more restricted in some respects and more general in others. We shall assume a familiarity with the MAD language, although many of the remarks made below will be clear even to those with only a slight knowledge of the language. Throughout any discussion of the organization of a translator, it is important to understand the primary objectives that have been set for its behavior.

This translator was designed to have the following properties (in the order given): (1) high speed of translation, (2) generality (i.e., as few restrictions on the user as possible), and (3) object program efficiency. Many decisions were made by means of these criteria which might have been made differently if there were other objectives. The design was also influenced to some extent, although not very much, by the organization of the computer (IBM 704) on which the translator was to operate.

Overall Structure

The translator divides quite naturally into three parts, to be referred to hereafter as I, II, and III. In I, statements of the source program are read, identified as to statement type, decomposed into "matrix form", and transmitted to III. In I, also, several tables of information such as dimension, mode, etc., are developed and passed on to II. These tables are digested by II to produce the storage allocation, which is in turn passed on to III. This storage assignment information is used by III together with the matrix form of the input statements to produce a final program.

Statement Decomposition (Part I)

The heart of the decomposition is the "expression scanner" subroutine which uses the now familiar right-to-left precedence-hierarchy scheme. In this procedure, the expression is scanned from right to left, placing each operand on a list, and placing on the list as well each operation which is not of lower precedence than its predecessor. (The

standard precedence table for arithmetic operations is

$$\begin{matrix} *, & / \\ +, & - \end{matrix}$$

One can also add to this table mathematical relations, logical operations, and so on.) Whenever an operation is encountered with precedence lower than that of the preceding operation on the list, a row of the "matrix form of the program" is generated for the preceding operation. Thus, the expression A + B − C/D would cause the following matrix to be generated:

$$\begin{matrix} / & C & D \\ + & A & B \\ - & R_2 & R_1 \end{matrix}$$

where R_i is the value of the result of the computation given in row i.

Before this can be applied, however, the statement type must be identified, and any expression which needs to be "scanned" must be isolated. It is also necessary to replace names of variables and functions, as well as numerical and alphabetical constants, by a more manageable code (which includes an indication of the kind of object it represents). Thus, when a statement is read into the computer, the first seven characters consist only of letters or digits, indicating an identifiable special statement such as EXECUTE . . . , or WHENEVER . . . , or there is some special character such as an equal sign, or a left parenthesis, which indicates an ordinary substitution statement. During the left-to-right scan of the statement thus begun, each operation is replaced by a simple numeric code, and each operand (variable name, function name, or constant) is saved on an "occurrence list" (i.e., each occurrence is saved as a separate entry). The operand is replaced in the expression by a code which indicates its index in the occurrence list and its type, such as "integer constant".

During the right-to-left scan which follows, the matrix form of the statement is developed. It should be noted that the entire handling of the type of statement is done at this time by I. For example, when an iteration statement is encountered, its entire structure is carried along quite simply by the matrix form of the program and by the use of "floating addresses". Consider the simple example:

```
          THROUGH ALPHA, FOR
         1J = X, H, .ABS. F.(X) .L. EPSLON
          ⋮
ALPHA     Y = Z − 1
```

This is translated as if the following statements had been written instead:

```
          J = X
          TRANSFER TO F1
F2        J = J + H
F1        WHENEVER .ABS. F.(X) .L. EPSLON,
         1TRANSFER TO F3
          ⋮
ALPHA     Y = Z − 1
          TRANSFER TO F2
F3        - - -
```

This entire set of statements is easily translated if provision is made for the assignment of true addresses instead of the floating addresses F1, F2, and F3. This is done in III. A similar method involving the use of floating addresses is used for compound conditional statements such as the following:

```
WHENEVER X .G. − 1 .AND. X .LE. 3
          Y = X .P.5 − 3.14
OR WHENEVER X .LE. −1
          Y = .ABS. (X .P. 3) + 4
OTHERWISE
          Y = X
END OF CONDITIONAL
```

Thus, the entire treatment of flow through the program, loop control, etc., occurs in I during the initial scanning.

As each row of the matrix is developed by I, three complete (cumulative) occurrence tables are constructed, making use of the local occurrence list for the current statement which was described above. The three tables designated MST, TT, and ABC, contain occurrences of variable names, function names, and constants, respectively.

A complete description of all tables developed in I and transmitted to II is given in Figure 1 (p. 30).

Storage Allocation (Part II)

The input to II is the collection of tables listed in Figure 1. The output is the collection of tables listed in Figure 2.

In order to produce these tables storage assignment is handled in the following way. The tables DIM, MST, and TT are first condensed. Since PCT, ERS, and ET represent the programmer's wishes with regard to storage assignment, they are processed immediately in that order, and the addresses thus assigned are entered in CMST. Since DIM is available, it is at all times possible to allocate sufficient storage for vectors and arrays. The ET assignment is accomplished by establishing equivalence classes containing, along with any variable name, any other variable name which occurs with it, or with any other member of the same equivalence class, in an equivalence declaration. Thus, in the declaration

```
EQUIVALENCE (A,B(1)), (B(1), C), (C,E), (D,F,G)
```

one equivalence class would contain A,B,C, and E, while the other would contain D,F, and G. Once a base address is assigned (temporarily), addresses relative to that base may be computed for each member of the class. Then, either an absolute address is indicated for the class because some element in the class has already been assigned to PROGRAM COMMON or ERASABLE, or the whole class needs to be assigned. The rest of the handling of the ET table is straight forward.

At this point it is possible to assign addresses to all heretofore unassigned variables. Before this is done, however, the transfer table TT is processed, since the transfer vector which results must be at the low end of the program, just above PROGRAM COMMON, as the object program is presently laid out. By means of the IFT table, the lowest

Name	*Code*	*Form of Entry*
Main Symbol Table	MST	Name of every variable, by occurrence.
Absolute Constant Table	ABC	Value of every constant, by occurrence.
Transfer Table	TT	Name of every function, by occurrence.
PROGRAM COMMON Table	PCT	Name of every variable to be assigned to the PROGRAM COMMON area.
ERASABLE Storage Table	ERS	Name of every variable to be assigned to the ERASABLE area. All names occurring in the same ERASABLE declaration are preserved as a group.
EQUIVALENCE Table	ET	Names of entries in equivalence groups, together with linear subscripts, if any.
Mode Table	MT	Name and mode entered for each mode declaration. Some entries generated by I, such as statement labels.
DIMENSION Table	DIM	Name of array, highest subscript, name of dimension vector, subscript of dimension vector.
Parameter List Table	PLT	Names of parameters for all function definition programs. Grouped by definition program.
Internal Function Table	IFT	Name of each entry to an internal function.
Scope Table	SCPT	Name of each statement label that occurs as the scope of some iteration statement.

FIG. 1

addresses are assigned in CTT table to "external functions", and the "internal functions" are assigned in CTT table just above the transfer vector, which contains only the external functions. Then all remaining variables are assigned, thus completing the CMST table. Now the mode table MT is condensed, and, by combining the mode information with the newly assigned addresses, both the modified MST and modified TT tables are formed. Replacement of names in the PLT table by modes is now easily accomplished, and finally the ABC table is condensed to the CABC table, and sequential addresses are assigned in CABC, which are in turn used to modify the ABC table.

An interesting sidelight to this process is the list that is produced at the end of II of variables which have not occurred in DIMENSION, PROGRAM COMMON, ERASABLE, or EQUIVALENCE declarations and which have only one reference in the program. (This is not considered an error, and the list is printed as a guide to the programmer only. It does not stop the compilation. The SCPT table is used here to delete names from the list which are used *only* as scope indicators for iteration statements.) This list has become a very valuable debugging aid, since a single reference to a variable usually does indicate an error. In particular, misspelled names will invariably show up on this list.

Name	*Code*	*Form of Entry*
Main Symbol Table (Modified)	MST	In place of each occurrence of a variable name is found the address assigned that variable, its mode, and in the case of an array, the address of the dimension vector, and a flag indicating that it is an array.
Transfer Table (Modified)	TT	In place of each occurrence of a function name is found the address assigned that function in the transfer vector, together with the mode of the value of the function.
Absolute Constant Table (Modified)	ABC	In place of each occurrence of of a constant is found the address assigned that constant.
Condensed Main Symbol Table	CMST	Each distinct variable name and its address.
Condensed Transfer Table	CTT	Each distinct function name and its address.
Condensed Absolute Constant Table	CABC	Each distinct constant and its address.
Parameter List Table (Modified)	PLT	Each parameter name has been replaced by its mode.

FIG. 2

Generation of the Object Program (Part III)

The tables of Figure 2, together with the matrix form of the source program from I, form the input to III. For each code representing a variable, function name, or constant in the matrix there is a corresponding entry in MST, TT, or ABC which now contains the address and mode. Thus, as III works its way through the matrix, it forms for each row a "mode context" for the indicated operation. For example, the first row of the example matrix given in section 3 above would represent the mode context "/, floating point, floating point" if both C and D were of floating-point mode, and the mode context "/, integer, floating point" if C were of integer mode and D floating-point. All legitimate mode contexts are listed in a mode context table (MCT) which contains, with each mode context, the address of the generator program for the mode context, and the mode of the result. The context "/, floating point, floating point" would give the address of a generator sequence (to be explained below) which generates instructions for the object program to carry out the indicated floating point division. On the other hand, the context "/, integer, floating point" would give the address of the generator sequence for integer to floating point conversion, and cause the context to be changed to "/, floating point, floating point" and reprocessed. Thus, mode conversions are handled by the same machinery as straight computation.

The generator sequence referred to above consists of a mixture of machine instructions and pseudo-instructions which is interpreted by III. Whenever a machine instruction is encountered, an instruction is generated in the ob-

ject program with the same operation code and appropriate addresses. Whenever one of the pseudo-instructions is encountered, it indicates a jump to one of two places in the sequence depending on the answer to some question. The four questions allowed in MAD are:

(AT) Is the first operand the result of the preceding computation?
(BT) Is the second operand the result of the preceding computation?
(AC) Is the result of the preceding computation in the accumulator?
(MQ) Is the result of the preceding computation in the multiplier-quotient register (on the IBM 704)?

A typical sequence, then, for some mode context, say "+, floating point, floating point", would be:

```
JMP *+1,MQ,*+5      Test: (MQ)
STQ T               Save previous result
CLA A               Generator code
FAD B
OUT AC              Exit with result in AC
JMP *+1,AC,*-3      Test: (AC)
JMP *-3,AT,*+1      Test: (AT)
JMP *+1,BT,*+3      Test: (BT)
FAD A               Generator code
OUT AC              Exit with result in AC
STO T               Save previous result
JMP *-9
```

where * means "this location" and the address part of each pseudo-operation indicates the jump to be made if the answer to the question is "true". Note that in the case in which the operand A is the result of the preceding computation and is in the accumulator, this sequence generates only the instruction FAD B (floating add B).

Thus, when III has worked its way through the matrix, the object program has been generated. In the process, however, floating addresses are usually encountered which were generated by I and which refer to parts of the program which occur earlier or later than the point at which the reference appears. These have not been completely processed at this point, so III makes one additional pass over the program and assigns floating addresses, generates actual binary card-images, and produce the printed version of the object program.

Definition of New Operations

There has been a great emphasis here on the storage of information in tables. One of the most important reasons for this was the desire to allow the user to incorporate definitions of new operations and modes into his program and then use these operations and modes directly as if they had already been in the language. This facility has recently been added to MAD, and it consists of making additions at "translation time" to such tables as the precedence table of I and the mode context table of III. It is felt that this facility will lead to experimentation which will point the way toward those operations and modes which would be useful in future languages.

MADCAP: A Scientific Compiler for a Displayed Formula Textbook Language*

Mark B. Wells
Los Alamos Scientific Laboratory of the University of California, Los Alamos, New Mexico

1. Introduction

This paper discusses some of the unique features of a compiler for scientific problems that was designed at Los Alamos for the MANIAC II computer. The compiler's name is MADCAP. An independent compiler project, under the name of COLASL, is now under way at Los Alamos for the STRETCH computer. The language goals of the two projects are quite similar and much of this paper, particularly the discussion in section 3, applies equally as well to the COLASL project as to the MADCAP project.

Actually, the MADCAP as discussed here is the third, and not the last, of a series of compilers resulting from a serious effort by the MANIAC group to eliminate the disparity between the "textbook" presentation of a problem and that presentation which serves as input for a compiler. Ideally, for example, one might like this source program to be able to pass for the problem report. In striving for such a goal there appear to be two essentially independent aspects of algorithm presentation to be considered. The first, which will be called the *local* problem, concerns the method of writing the contents of a formula or statement, that is, the expression of numeric and literal symbols, of English words and names, and of arithmetic operations. The second, which will be called the *global* problem, concerns the method of presenting the sequence of statements to be performed so that the algorithm will be solved, that is, the logical control of the problem. Of the two, the first is more easily attacked since it is a much simpler task to say what constitutes standard formula notation than what constitutes standard logical control notation. Few scientists would misinterpret

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a},$$

* This work was performed under the auspices of the U. S. Atomic Energy Commission.