LADC-5901

SEP 12 1963

Facsimile Price $ 1.60
Microfilm Price $ .80

Available from the
Office of Technical Services
Department of Commerce
Washington 25, D. C.

79849

MASTER

Recent Improvements in MADCAP*

Mark B. Wells

Los Alamos Scientific Laboratory

of the

University of California

Los Alamos, New Mexico

This paper was submitted for publication in the open literature at least 6 months prior to the issuance date of this Microcard. Since the U.S.A.E.C. has no evidence that it has been published, the paper is being distributed in Microcard form as a preprint.

LEGAL NOTICE

This report was prepared as an account of Government sponsored work. Neither the United States, nor the Commission, nor any person acting on behalf of the Commission:

A. Makes any warranty or representation, expressed or implied, with respect to the accuracy, completeness, or usefulness of the information contained in this report, or that the use of any information, apparatus, method, or process disclosed in this report may not infringe privately owned rights; or

B. Assumes any liabilities with respect to the use of, or for damages resulting from the use of any information, apparatus, method, or process disclosed in this report.

As used in the above, "person acting on behalf of the Commission" includes any employee or contractor of the Commission, or employee of such contractor, to the extent that such employee or contractor of the Commission, or employee of such contractor prepares, disseminates, or provides access to, any information pursuant to his employment or contract with the Commission, or his employment with such contractor.

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

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

Recent Improvements in MADCAP

Mark B. Wells

ABSTRACT

MADCAP is a programming language admitting subscripts, superscripts and certain forms of displayed formulae. The basic implementation of this language was described in a previous paper [MADCAP: A scientific compiler for a displayed formula textbook language, Comm. ACM, Vol. 4 (Jan. 61), 31-36]. This paper discusses recent improvements in the language in three areas: complex display, logical control, and sub-programming. In the area of complex display, the most prominent improvements are a notation for integration and for the binomial coefficients. In the area of logical control the chief new feature is a notation for variably nested looping. The discussion of sub-programming is focused on MADCAP's notation for and use of "procedures".

1. Introduction

Authors of programming languages for scientific problems (FORTRAN, ALGOL, etc.) generally place the burden of "linearizing" mathematical formulae (that is, reducing scripting and display to a one-line notation) upon the programmer. Historically, this state of affairs arose because it was (mechanically) difficult to present normal mathematical formulae as input to a compiler. Many people in the computer field seem to be content with this situation, but, in this author's opinion, the linearization of formulae is just as burdensome as translating from mnemonics to machine code, assigning storage, or any of the other numerous mechanical tasks of coding from which compilers are intended to liberate the scientist. Furthermore, the legibility of a "textbook" presentation of a problem is extremely important to a less computer oriented colleague called upon to understand and verify a calculation.

MADCAP is a programming language admitting general displayed formulae [3]. With the help of a "scripting" Flexowriter for preparing compiler input it has been successfully implemented for the MANIAC II computer at Los Alamos. MADCAP is a "growing" language [1, 2, 3], and this paper discusses some of the newer features of the language.

The object of this exposition is twofold. First, by exhibiting the types of mathematical formulae that can be implemented easily once a one-line notation is abandoned, we hope to induce a measure of envy among scientist-programmers. Perhaps in this way we will speed the day when the input of general display will be made available to everyone.

Second, we wish to present some results of programming language research which have evolved independently of the massive FORTRAN and ALGOL projects. In this author's opinion, a truly acceptable "universal" programming language can only evolve (as mathematical language has and still is) by natural selection from many mutations.

Section 2 gives an abstract of the MADCAP language. Section 3 discusses new features involving complex display. Section 4 presents some extensions to the notation for expressing iterations and Section 5 gives MADCAP's approach to some sub-program questions (procedures, non-library functions, and blocks).

2. The Basic Language

As with most programming languages, MADCAP is a statement language. That is, the source program consists of a sequence of statements — formulae (defining equations, e.g., $a = \sqrt{b} + 2$), control statements ("if . . .", "for . . .", etc.), and information statements ("subscript range . . .", "format . . .", etc.) which outline in detail the solution to the problem at hand. This code is prepared on a full-keyboard, modified (to allow key-controlled platen rotation) Friden Flexowriter Typewriter, which records the sequence of keystrokes on a paper tape (for use as input to the compiler) as well as on the typed page. The individual characters which may be printed (in red as well as black) are as follows:

a through z A through Z Δ

0 through 9 π

\+ - × / _ $\sqrt{\ }$ = → < >

. , : ( ) [ ] | "

* '

In addition, the following typewriter control keys are available and also record their function on the paper tape when struck: space, backspace, upper case, lower case, red, black, carriage return, tabular, stop, delete, superscript, subscript. The last two keys represent the modification mentioned above. Their function is to rotate the platen one half-line down or one half-line up, respectively. They allow the direct typing of superscripts, subscripts, and displayed formulae. The flexibility of the Flexowriter allows new symbols to be formed by typing certain characters on top of or closely adjacent to one another. Those which have a known meaning to MADCAP are as follows:

$\not<$ $\not>$ $\neq$ (or $\neq$) $\leq$ (or $\underline{<}$) $\geq$ (or $\underline{>}$) ;

Σ $\prod$ $\int$!

The formation and use of the last four symbols is discussed in section 3.

With respect to the writing of individual formulae, the MADCAP language differs very little from common mathematical notation. Scripting for name (identifier) formation, indexing and exponentiation, displayed division, and implied multiplication by juxtaposition of operands are all part of the language. There are, of course, some notational

restrictions imposed to keep the compiler and compiling within reasonable bounds of space and time. In most cases, however, a restriction is imposed on the language only when the frequency of occurrence of that being disallowed is extremely small. For example, at the present, display within exponents or subscripts is not allowed. On the other hand, the common notation for exponentiation of a functional value, e.g., $\cos^2 x$, is permitted as it arises quite frequently in our work.

Although not particularly common in mathematical notation, the use of "word-names" is very popular in computer work. In the MADCAP language such identifiers are capitalized to distinguish them from function mnemonics, keywords, or the product of variables. For example, "csc" is a mnemonic for "cosecant", while "Csc" would be an identifier, and "cCs" would be "c times Cs".

Program control is handled by the basic "go" and "if" statements, "for" statements and their extensions (see Section 4), and sub-programs (procedures - see Section 5). The MADCAP language also contains statements for array storage assignment, number type declaration, input-output handling, and other jobs associated with program preparation. Details of the language may be found in the MANUAL [2].

3. Complex Displayed Formulae

Two of the early notational improvements to the basic language were the incorporation of a summation and a product notation. Motivation for their inclusion was as much the flaunt of our new capability (see, for

fun, the cover of Datamation, December 1961) as it was the utility of the notation, though both are now an active part of the language. The limited size of the Flexowriter keyboard did not allow the inclusion of a large sigma and a large pi, but the composed facsimiles we were able to construct were surprisingly acceptable. A sigma is formed from two "less than" symbols one half-line apart, and a pi consists of adjacent vertical bars topped with a pair of underscores three half-lines above the line of the bars. The limits appear as subscripts and superscripts on these symbols.

Since summations and products may appear as part of a more involved formula, it is necessary to require the summand (or factor) to be in parentheses. Examples of legitimate MADCAP statements involving these notations are:

$$I_p = \sum_{i=1}^{m} (u_i v_i)$$

$$\phi_r = \prod_{i=1}^{r} \left[1 - \frac{1}{p_i}\right] + b_r$$

With the limits treated as normal scripts, the linearization of such statements is straightforward. The translation of this one-line form into symbolic machine code is of interest as it involves a useful compiler recursion. A summation (or product) is rewritten by MADCAP into a series of simpler statements and then these new statements are analyzed just as if they had been prepared by the programmer. Compilation of a complicated statement (for example a multiple summation) involves

iteration of this process (see [3] for further discussion).

A related, and more valuable, notation that is part of the language is that for numerical integration. Standard textbook notation for definite integration has been adopted, for example:

$$Y = \int_{1.0}^{\pi/2} \frac{\sin^3 t}{t}\, dt$$

The integral sign is not typable directly, but a left parenthesis one half-line above a right parenthesis is quite satisfactory. As before, the limits appear as scripts on this symbol. The integrand may be any expression and is followed by the differential which defines the dummy variable of integration (any legal name [3]). Language restrictions that must be imposed on the use of the integration notation are few and natural. As an example, a lower case d may not be used within an integrand. To accomplish the integration, MADCAP inserts a call to a (Simpson's rule) library subroutine. The subroutine has facility for returning temporarily to the main program in order to compute the functional values it needs. Also, communication of data between the subroutine and main program is by means of a data-table assigned by MADCAP for each particular integral. Thus, multiple integrals are possible. The subroutine generally uses a preassigned number of intervals (thirty-two) for its calculation, although it is possible for the programmer to specify the number to be used by altering the quantity "IGN$\Delta\#$" in a statement preceeding the integration.

It is possible, and often convenient, to use this notation to perform an integration where part of the integrand is a tabular function. Thus, for example, one might have

$$Q = \int_a^b \text{fct}(X)\sin X \, dX$$

where fct(X) is a procedure (see Section 5) whose job is table lookup, and perhaps interpolation, from a given tabular function.

Integrations are always performed in floating point arithmetic.

A recent addition to the language is the common display notation for the binomial coefficients. This notation greatly improves the readability of many combinatorial formulae.

No new symbols are required for this notation although we would occasionally prefer to have larger parentheses. The following examples illustrate the notation:

$$\binom{n}{r} \quad \binom{n+r-1}{r} \quad \binom{q+k}{q+j-1} \quad \binom{\frac{A+B}{2}}{R_i+1}$$

Note that the "numerator" and/or the "denominator" of the coefficient may themselves be displayed (this is seldom called for, however).

The linearization of a coefficient is accomplished in a manner similar to that for displayed division [3]. In this case, however, the dividing "line" between "numerator" and "denominator" is the blank half-line between parentheses. The presence of a binomial coefficient is detected during the map analysis by the change of main-line following a

left parenthesis. Then, the linearization results in, say, "binomial(m, r)", where "binomial" is the title of a two-argument library subroutine.

For the sake of efficiency, the binomial coefficient subroutine is merely a table lookup. MADCAP arranges for the computation of the table as part of, and at the beginning of, the target code.

A related, but simpler, new notation is that for factorials. The exclamation point is constructed from a prime and a period. There is no linearization required, but an expression, say, "n!", would be rewritten as "factorial(n)" during the linearization pass. As with "binomial","factorial" is a library subroutine which merely finds its value from a precomputed table.

For both the binomial coefficient and factorial cases, the arguments may be either real (floating point) or whole (integer) numbers. MADCAP selects the appropriate subroutine (including table computation) to yield the desired result; that is, real arguments will yield a real answer, and integer arguments will yield an integer result.

4. Extensions to Looping Notation

Notation for expressing the logical control of an involved calculation is generally far less developed than notation for expressing the individual mathematical formulae (statements) used in the calculation. The use of a flow diagram has always been (and still is) an excellent way to describe logical control. However, the difficulty in presenting

a flow diagram as input to a compiler tends to frustrate the incorporation of such notation into a programming language (in this regard, see [4]). Consequently, programming languages generally require a "linear" presentation of the statements composing a program, with allowance for conditional (and unconditional) branching of control by means of "if" (and "go") statements and iteration by means of "for" statements. In the MADCAP language the requisite bracketing of statements to indicate dependence under such control statements is accomplished with indentation [3] rather than by statement number reference [FORTRAN, for instance] or "begin-end" bracketing [ALGOL]. This simple use of a second dimension in program presentation greatly improves the readability of programmed logical control. (It is interesting to note how natural this language feature must be as most ALGOL programs are now written in this form although, regrettably, the "begin" and "end" must still be there.) The implementation of indentation-bracketing is quite simple using the tabular key of the Flexowriter.

The introduction of a notation for iteration ("for" statements) into a programming language is really the first step in the sophistication of a programming language with respect to logical control. Such a notation is introduced, of course, because it is burdensome to be required to write statements to initialize, increment and test a loop index everytime an iteration is desired. Other common control tasks for which the expression in terms of several basic statements rapidly becomes tiresome are frequently encountered. (See [5], for instance,

which presents some language extensions to FORTRAN for expressing complex searching and testing jobs useful in simulation problems.) Two such tasks have recently been assigned a more compact notation in the MADCAP language.

The first and simpler allows a loop to have more than one (dummy) index. For example:

for i = 1 to I; j = 0, 2, ... 2m; X = X_o, $X_o + \Delta X$, ...

BODY OF LOOP

C = 0

is equivalent to

```
      i = 1
      j = 0
      X = X_o
      go to #1
#2    i + 1 → i
      j + 2 → j
      X + ΔX → X
#1    if i > I, go to #3
      if j > 2m, go to #3
      BODY OF LOOP
      go to #2
#3    C = 0
```

The exit from the loop occurs when any of the loop indices has completed its range. This "parallel incrementing" feature should be contrasted

with the similarly appearing ALGOL notation, for example:

for i := 1 step s until n; n + 1 step 1 until m

which assigns a succession of values to a single dummy index. There has been less demand for this latter "for" statement extension, but such facility will soon be available in MADCAP via a general notation for the manipulation and use of "sets" of integers.

The second new notation allows variably or indefinitely nested iterations ("for" statements) to be written compactly and quite naturally. Demand for such nesting arises in many problems but particularly often in the form of "backtracking" in combinatorial investigations [6, 7]. As a simple example, consider the task of counting from 0 to 242 ($= 3^5-1$) using base 3 arithmetic where the 5 individual "digits" are what is needed. In terms of nested iterations this job would be written as follows:

```
for d1 = 0 to 2
    for d2 = 0 to 2
        .
        .
        .
            for d5 = 0 to 2
                USE DIGITS d1, d2, ..., d5
```

The MADCAP notation for this job is:

```
with i = 1, 2, ... 5
    .
    .
    .
    for di = 0 to 2
        .
        .
        .
            USE DIGITS d1, d2, ..., d5
```

More realistic examples would have the "5" a variable (or unexpressed), the "2" a function of the index i, and statements preceding and following each iteration, for example:

```
with i = 1 to I
  .
    .
      STATEMENTS
      for q_1 = 0 to Q_i
          STATEMENTS
            .
              .
                STATEMENTS
          STATEMENTS
ADDITIONAL PROGRAM
```

The form for the expression of the range of the dummy index in a "with" statement is the same as allowed in a "for" statement [1].

It is seen from the above example that three indentation levels are required to express normal "with" statement dependency. A "with" statement may operate upon another "with" statement, rather than on a "for" statement, and each such application adds (at least) two indentation levels of dependency.

As mentioned above, the logical control expressed by "with" statements occurs especially often in "enumerative" jobs. A good

percentage of the programs written in MADCAP are of this type and further language development in this direction (e.g. set manipulation) is being made. These developments will be reported elsewhere.

5. Procedures

A MADCAP "procedure" is a section of program written in MADCAP language and referenced by means of a mnemonic title and list of arguments. Thus MADCAP procedures are quite similar to ALGOL procedures; yet, the MADCAP notation arose independently and is generally more concise than ALGOL notation. In what follows it will be convenient to explain certain features by comparing with ALGOL or by using ALGOL terminology [8], since that language is so widely publicized.

There are two basic reasons for incorporating such sub-program notation into a language, each of which presents somewhat different notational demands. One, it is desirable to allow the programmer to define his own specialized function routines and be able to refer to them in the usual fashion (as he refers to library functions such as sin, log, etc.), and two, it is efficient to allow pre-coded and pre-debugged jobs to be incorporated easily into another program. Associated with reason two is the useful programming practice of splitting a large problem into several nearly independent sections, each section prepared by a different

programmer, and where communication between sections derives from reference to a few common indentifiers. A feature suggested by reason one above is that provision for "transitory" arguments (in ALGOL, arguments called by value) be made. Suggested by reason two is provision for "stationary" arguments (in ALGOL, arguments called by name). In giving a list of arguments, the transitory arguments are separated from the stationary arguments by a semi-colon.

It seems reasonable that all labels, and identifiers not in the argument list, which appear within the body of the procedure code should be local to that code. Thus, as with ALGOL, a procedure is essentially an independent sub-program, a block. It is interesting to note that there has been no motivation to define a block independent of a procedure; that is, sub-programming by means of procedures has, so far, proved both natural and sufficient.

A procedure is called by giving the procedure title (a mnemonic devised by the programmer and consisting of three or more lower case letters) followed by the list of arguments enclosed in parenthesis. This may appear within a statement as in

$$y = gud(x)/[e^{X} + \theta]$$

or as a statement by itself as

hydro(n, k; P, R, E, V, g).

A procedure is declared by heading the sub-program with a statement giving the title and argument list (as used by the procedure code) and bracketing it with the symbols "(..." and "...)", as for example:

```
(...   hydro(a, b; p, r, E, V, h)
       BODY OF PROCEDURE
...)
```

If the result of the procedure is a single quantity and the procedure is to be called within an arithmetic statement, then the last line of the body of the procedure should define this result, for example:

```
(...   diff(X, Y)
       a = X - Y
       if a < 0.0
              a = 0.0
       diff(X, Y) = a
...)
```

If the body of the procedure producing a single result can be written on a single line, then so may the entire procedure declaration. For example:

$$(\ldots \quad gud(x) = 2\arctan(e^x) - \frac{\pi}{2} \quad \ldots)$$

As usual in the MADCAP language, transitory arguments and identifiers internal to a procedure are assigned real number-type unless specifically declared to be integers the first time they appear. Stationary arguments, of course, adopt the number-type of the "main-problem" identifier to which they are equivalent. The number-type of the procedure result itself is defined either by the last line of the procedure body (see last two examples) or in a specific declaration.

It was noted above that the transitory and stationary arguments in an argument list are separated by a semi-colon. Thus, in "hydro(n, k; P, R, E, V, g)", "n" and "k" are transitory arguments while the rest are stationary arguments. If there are no stationary arguments, then the semi-colon may be omitted as in the "diff" and "gud" examples above. If there are no transitory arguments, then the semi-colon must still be there, but for esthetic reasons the word "none" may be placed before it, as for example, "proc(none; A, B, C)". More often than not the identifiers of a procedure body which are stationary arguments are identical to the indentifiers of the "main code" (actually one procedure level back, since procedures may be nested) to which they are being made equivalent. In this case the stationary arguments in the argument list of the procedure call may be omitted. This is especially useful when a pre-coded functional procedure, such as "lstsq" (least squares), must refer to an as yet uncoded function with an unknown number of arguments. The arguments are then given in the heading of the function procedure declaration as stationary arguments and refer to main problem identifiers. Occasionally, it is desirable (when a procedure is one part of an arbitrarily sectioned problem, for instance) to make no distinction between identifiers internal or immediately external to a procedure body. This may be done by making all identifiers within the procedure body (except identifiers which are transitory arguments, of course) stationary arguments, by placing the word "all" after the semi-colon in the argument list of the heading, for example:

(... fct(n, m; all)

As indicated above, procedure bodies may be nested, much as are blocks in ALGOL. Reference to such procedures is restricted only in that there be no closed cycles, the simplest case of which is a procedure body containing a reference to itself. So far, no practical example requiring that this restriction against "recursive procedures" be lifted has arisen in our work.

6. Conclusion

This paper has presented some of the newer features of the "scientist-oriented" language MADCAP. The presentation is, of course, informal and incomplete. In fact, the language itself will never be completely or formally defined since, from a practical point of view, it makes no sense to attempt to do so. The expedition of problem solving is the primary motivation for MADCAP language development.

The current expanding tide of research in formal algorithmic languages is a little frightening in some respects. The danger that this tide will drown out research on the more mundane problem of bridging the notational gap between men and machine is not negligible. Programming language authors should not forget that compiler languages are intended to help make the computer a working tool of the everyday scientist and not merely to present the scientist with a new (and "foreign"), albeit efficient, language in which to state his problems.

REFERENCES

1. Bradford, D. H. and Wells, M. B. MADCAP II. In Annual Review in Automatic Programming, Vol. 2, Pergamon Press, 1961, 115-140.

2. Bradford, D. H. and Wells, M. B. MADCAP III (Manual). LAMS-2601, Los Alamos Scientific Laboratory, 1961.

3. Wells, M. B. MADCAP: A scientific compiler for a displayed formula textbook language. Comm. ACM 4 (Jan. 61), 31-36.

4. Voorhees, E. A. Algebraic formulation of flow diagrams. Comm ACM 1 (June 58), 4-8.

5. Buxton, J. N. and Laski, J. G. Control and simulation language. Comput. J. 5 (Oct. 62), 194-200.

6. Walker, R. J. An enumerative technique for a class of combinatorial problems. In Proceedings of Symposia in Applied Mathematics, Vol. 10, Amer. Math Soc., 1960, 91-94.

7. Lehmer, D. H. Teaching combinatorial tricks to a computer. In Proceedings of Symposia in Applied Mathematics, Vol. 10, Amer. Math Soc., 1960, 179-193.

8. Naur, P. and others. Revised report on the algorithmic language ALGOL 60. Comm. ACM 6 (Jan. 63), 1-17.