T H E U N I V E R S I T Y O F M I C H I G A N

Memorandum 31

DEFAULTS AND BLOCK STRUCTURE IN THE MAD/I LANGUAGE

Allen Springer

CONCOMP: Research in Conversational Use of Computers
ORA Project 07449
F.H. Westervelt, Director

supported by:

DEPARTMENT OF DEFENSE
ADVANCED RESEARCH PROJECTS AGENCY
WASHINGTON, D.C.

CONTRACT NO. DA-49-083 OSA-3050
ARPA ORDER NO. 716

administered through:

OFFICE OF RESEARCH ADMINISTRATION ANN ARBOR

July 1970

ACKNOWLEDGMENTS

The author would like to acknowledge the support of the CONCOMP Project; IBM, who sent the author on an IBM Resident Study Program; and especially his co-workers on the MAD/I compiler, Bruce Bolas, Ronald Srodawa, Charles Engle, David Mills, Fred Swartz; and the MAD/I coordinators, Profs. Bernard Galler and Bruce Arden.

TABLE OF CONTENTS

1. INTRODUCTION

This paper describes the default and block structure mechanisms of MAD/I, a PL/I-like language, and the interaction of these mechanisms with the three types of MAD/I declarations: explicit declarations, default declarations, and conditional declarations. MAD/I allows the programmer extraordinary control over the default assignment of data types to variables, and also allows the programmer more than usual control over the scope of variable names in block structure. The interaction of these two facilities can make the handling of declaration information a difficult problem. This paper outlines an algorithm in which this information is processed "on the fly" in the first pass of the compiler over the source program, and then the symbol table is processed to assign defaults and allocate storage. A simple second pass over a transformed version of the source text resolves the scope and interpretation of variable names.

MAD/I is a computer language under development at the University of Michigan Computing Center, sponsored by the CONCOMP Project. It can be thought of as a remote descendant of 7090 MAD and ALGOL 60, with PL/I being a not-too-distant relative. However, MAD/I and its compiler have some unusual features that aid language modification and extendibility, although these features are

beyond the scope of this paper. Except for block structure scope facilities and the default setting facilities, then, MAD/I may be regarded as simply another representative of the class of procedural languages which includes ALGOL 60 and PL/I.

Briefly, MAD/I has blocks, as in PL/I and ALGOL 60. Like PL/I (but unlike ALGOL 60), declarations may occur anywhere within a block, and are not required for all variables in the program. If some attributes of a variable are not declared then they are given "default" values. Such attributes include storage class (e.g., static, based, etc.) and data type. The facilities for specifying the defaults are very different from those of PL/I, and are a generalization of those of 7090 MAD. The scope of a variable is determined in much the same spirit as in ALGOL 60 and PL/I, but the programmer has more control over the specification of scope, including the scope of variables which are not declared. This makes determining scope and determining defaults a complicated problem.

2. DEFAULTS IN MAD/I

The default assignment of data types is done in a very systematic and general manner. At any point within the program there is defined a current default data type. This default data type may be declared by the programmer on a block basis. A special symbol, 'DEFAULT', is used to carry the default information, and is treated like a variable when in the context of declarations, but otherwise it is not written by the programmer.

The default data type is given to any variable for which no data type has been explicitly declared. For some data types one can declare a "sub-data-type," such as the component data type of an array, the data type of the result returned from a subroutine, or the data type of a component of a structure. If such a "sub-data-type" is not specified then it is given the default data type. For example, assume that the default has been declared as follows:

```
'DECLARE' 'DEFAULT' 'INTEGER'
```

Then assume the following declaration:

```
'DECLARE' A 'FIXEDARRAY'(4,4)'FLOATING',
          B 'FIXEDARRAY'(4,4);
```

The mode of both A and B is 'FIXEDARRAY', with dimension 4 x 4. The component data type of A was explicitly declared to be 'FLOATING'. Since the component data

type of B is not explicitly declared, it is taken to be the default, 'INTEGER'. If some other variable that belonged to that block were referenced in the block but no declaration made about its data type then it would also be assigned 'INTEGER' data type.

There are other cases where default actions occur in MAD/I. They will be mentioned briefly here although they are not involved in the rest of this paper. The dimension information given in the above example is specified in a declaration "suffix." If such a suffix is omitted for declarations where they are normally expected, then a default set of information is assumed for the missing information. For example, in the case of an item of 'CHARACTER' mode, the suffix specifies how many characters the variable has. If the suffix is omitted, the number of characters is assumed to be one. If the dimension information were omitted above, a warning message would be issued, and an array which has one dimension and one component would result. The lexical class of constants specifies an implicit data type which they are assigned, unless a declaration is explicitly written which specifies some other data type. As an example, the constant 5 will be assigned 'INTEGER' data type (32 bits long on the IBM 360), whereas 5@('INTEGERSHORT') produces a constant 16-bit integer.

These two types of default operations are presently not controllable any further by the programmer. For suffixes the default is associated with the mode involved. For constants the default is associated with the lexical class of the symbol. It would be possible to have these defaults also controllable by the programmer by adding special declarations to the language, but this has not yet been done.

The default data type in the outermost block is 'FLOATING' unless it is explicitly declared to be something else. For any other block the default is the same as for the next outer block unless it was explicitly declared in the inner block. If a default mode is declared, but not completely, then the remainder of the default is taken from the default of the next outer block. This is done in exactly the same manner in which defaults are applied to a variable whose "sub-data-type" may not have been declared. As an example, assume that in the outer block the default is 'BOOLEAN'. Assume that the default is then declared as follows in the inner block: 'DECLARE' 'DEFAULT' 'FIXEDARRAY'(10); Thus the component of the inner block's default is not explicitly specified. It will be made the data type of the default of the next outer block, 'BOOLEAN'. Generally the default propagates inward from the next outer block, in a manner similar to the propagation of scope of variables.

3. BLOCK STRUCTURE IN MAD/I

There are three concepts embodied in block structure as it is traditionally specified in ALGOL 60 and similar languages. Typically the block is denoted by a beginning keyword and an ending keyword. In ALGOL a block has three functions: (1) to specify scope of variables, (2) to specify the dynamic nature of storage allocation for certain classes of variables, and (3) to group statements. In PL/I the grouping effect can also be obtained with a DO statement as well as with a BEGIN statement. In MAD/I the 'BEGIN' statement is used for simple grouping of statements, and the other two facilities are specified by 'BLOCK' or 'PROCEDURE', corresponding to BEGIN and PROCEDURE in PL/I. Thus MAD/I has facilities similar to those of PL/I, although with different names.

The scope in which a variable is known is determined rather simply in ALGOL. If a variable is declared in a given block, that variable's name represents a different variable from one of the same name in the next outer block. If a variable is used in an inner block but not declared there, then it is the same variable as one of the same name in the next outer block. Finally, in ALGOL 60 all variables must be explicitly or implicitly declared in the outermost block in which they are to be known.

In MAD/I the "naming" or scope rules are similar to those of ALGOL 60, but there are additional rules allowing the programmer more control over the "naming" facility. In MAD/I the user does not have to declare a variable at all; therefore he needs conventions in order to know in which block an undeclared variable belongs. In PL/I the rule apparently is that a declared variable belongs to the outermost block in which it is declared. If it is not declared, then it belongs to the outermost block.

Let us motivate the additional rules for assigning defaults to symbols. By writing a large block and specifying the default within that block, the user can avoid writing a large number of individual declarations for variables in that block. But if the block is an inner one, then, following the PL/I rule, variables that are not declared in that block would belong to the next outer block and would not be affected by the default. What is desired, in some cases, is that unless otherwise specified, any variable used in a block is declared in that block implicitly. In other cases we would want to have the PL/I rule. Thus we have modified the scope rules for MAD/I as follows:

(1) If no default is declared for a block then the only symbols that belong to that block

are those that are declared in the block.

(2) If a default is declared in a block, but 'NEW' has not been declared for that default symbol, then symbols that have not been declared in the block are treated as if they were referenced in the next outer block.

(3) If default was declared for the block and 'NEW' was also declared for the default, then, unless otherwise specified (by rules below), all symbols referenced in the block are implicitly declared in the block.

Note that under these rules a block with default declared 'NEW' would not be able to access any variables outside that block. Therefore, we have devised additional rules, which apply irrespective of any default declarations or 'NEW' declarations currently in effect:

(1) If a symbol is declared 'NOTNEW' in a given block, then it is treated as if it were referenced in the next outer block.

(2) If a symbol is declared 'GLOBAL', then it is treated as if it were declared 'NOTNEW' in that block and each surrounding block.

(3) If there is no next outer block as stated

in (1) and (2) above, then the variable belongs to the outermost block.

Although MAD/I has not yet been used extensively, most of these rules have proved useful and have eliminated much writing of declarations in some cases. Typically, the scope rules of block structure are used to allow the writing of relatively independent sections of program which are to be part of the same compilation. The block structure allows the user to write the sections without worrying that two variables in different sections may accidentally have the same name. In ALGOL the variables in the two blocks would be declared in their own blocks, and those that are intended to be common would be declared in the next outer block. In MAD/I the programmer has the freedom of not declaring all variables in such blocks; instead he declares 'NEW' 'DEFAULT' in each independently written block. Then all variables referenced in each block belong to that block unless declared 'NOTNEW'. This combination of rules gives the user the advantages of both the block structure and the default declaration facility.

In the left-hand column below are several blocks representing the skeleton of a complete MAD/I program. All references are indicated by occurrences of variable names. All declarations are indicated. The right-hand column contains comments about items to the left.

```
'PROCEDURE' MAIN;                  Block 1 begins.  Main is im-
                                   plicitly declared 'ENTRYPOINT'
                                   mode.
... A ...                          This variable A is not de-
                                   clared,so it belongs to the
                                   outermost block and has de-
                                   fault mode of 'FLOATING'.
'BLOCK'                            The beginning of block 2.
                                   This block has no default
                                   declared for it.
... A ...                          This is the same A as in
                                   block 1.
'DECLARE' B;                       B is new to this block, and
                                   will have the default mode,
                                   'FLOATING'.
'DECLARE' C;                       Despite the declaration, C
'NOTNEW' C;                        is not new to block 2, but
                                   belongs to block 1, and has
                                   default mode.
'BLOCK'                            Block 3 begins.
'DECLARE''DEFAULT''INTEGER';       There is a new default for
                                   this block, but 'DEFAULT'
                                   has not been declared 'NEW'.
... A ...                          Since it is only referenced
                                   here, this A is the same as
                                   in block 1 and 2.
... B ...                          Since it is not declared in
                                   this block, B is the same as
                                   in block 2.
'DECLARE' C;                       C is new to this block and
                                   has default mode of 'INTEGER'.
'END';                             The end of block 3.

'BLOCK'                            The beginning of block 4.
'DECLARE''DEFAULT''NEW'            This block does have its de-
  'CHARACTER'(256);                fault declared 'NEW'.
... A ...                          As a result, this A is a
                                   new variable even though it
                                   is only referenced in the
                                   block; it has the default
                                   mode of 'CHARACTER'(256).
'DECLARE' B 'FLOATING';            B is new to this block.
'DECLARE' D 'NOTNEW''BOOLEAN';     D is the only variable
                                   referenced in this block
                                   which does not belong to the
                                   block.  It belongs to the
                                   next outer block.
'END';                             The end of block 4.
'DECLARE' D;                       This is the same D as in
                                   block 4.  D belongs in this
                                   block instead of the next
```

```
                                outer one because of this
                                declaration.
'END';                          End of block 2.

'END';                          End of block 1.
```

In this example have two distinct As, two distinct Bs, and two distinct Cs. Of course this example looks rather complicated, because no other program details are supplied to make it look more natural, and because it attempts to illustrate many rules with one example.

It is interesting to point out, for procedures in MAD/I, that entry points to a procedure fall inside the 'PROCEDURE' ... 'END' brackets, and are implicitly declared to be 'ENTRYPOINT' mode. According to the strict rules specified above, these entry points would be "new" variables in the block and thus not known outside the block, definitely an undesirable situation! Thus there is also an implicit 'NOTNEW' declaration on each entry point specified in the prefix of a 'PROCEDURE' statement.

4. THE ORGANIZATION OF THE COMPILER

This section discusses the organization of the compiler so that the algorithm given in the next section will be seen in the proper context. The compiler makes two passes over the source program, in which it collects all declaration information, parses the source text, resolves all default information for symbols referenced by the programmer, and straightens out all block structure information. Between the two passes there is a symbol-table-processing phase.

The first pass does most of the work. Briefly, it parses the input character stream into "symbols," parses the program in symbol form, and expands the parsed symbols into "n-tuples" of the form of an operator followed by zero or more operands. The n-tuples become a new representation of the source program. For example, A:=B+C might be transformed into

```
+,%T1,B,C;
:=,%T2,A,%T1;
```

where the percent symbols are user-generated temporary symbols. The algorithm described below assigns data types to the symbols A, B, and C (but not to the temporary symbols). Also, if several variables named A are declared, the algorithm will determine which variable named A is represented by any given instance of the Symbol A.

The major problem encountered in scanning the input text is that after a symbol has been found which could represent a variable, nothing more may be known about it until the end of the block is encountered. This is because declarations about a variable, if there are any at all, may occur anywhere in the block. By the end of the block it is possible to determine whether a given symbol referenced in the block represents a variable belonging to the block. To solve this problem, we need to know (1) what, if anything, has been explicitly declared about the symbol, and (2) whether a 'DEFAULT' has been declared 'NEW' for the block. A second problem is that attributes cannot be completely assigned for any variable until all the attributes of the default for that block are known. But the attributes of the default cannot always be known until they are known for the default of the next outer block. Thus, since the last statement of the program might be the declaration of the default of the outermost block, the whole program has to be scanned before defaults can be applied to the variables.

Let us examine in more detail what happens to a specific symbol during the processing of the program. When a symbol which can represent a variable is first encountered, all that can be done is to save its name and note that it was referenced in the block currently

being scanned. We cannot know whether it represents a variable belonging to that block until the end of the block has been found. Furthermore, we cannot know whether it belongs to that block even if a declaration occurs for it, since a subsequent 'NOTNEW' or 'GLOBAL' declaration might occur for it in that block. More particularly, we cannot know whether it represents the same variable or a different variable from the symbol of the same name found in the next outer block. Note that if we are to produce n-tuples while parsing the input text, we must represent a variable in the n-tuple by a pointer to the symbol for that variable, at the very least. We cannot include which block it belongs to, however, since that is not known yet. Therefore we must either (1) assume which block it belongs to, and correct that assumption later if it is incorrect, or (2) not bother to assume which block it belongs to, and correct the n-tuples some way later. No matter what is done initially, however, the n-tuples must somehow be corrected later. The method of doing so, of course, depends upon how the symbol is represented in that n-tuple. In the first implementation of MAD/I we have chosen to have the representation of the symbol in the n-tuple always point to the same "main symbol table" entry for that symbol. Then, in the second pass, the n-tuple is made to point to some

other symbol table entry, if necessary.

Let us assume that something was declared about A in an outer block and then something else was declared about A in the next inner block. If A is subsequently declared 'NOTNEW' in the inner block, then the two declarations must refer to the same variable. If the 'NOTNEW' does not occur, then the declarations refer to two different variables. In the present implementation of MAD/I, the symbol table entries carry the declaration information. We thus need a way of keeping separated the information of the two declarations about A until it can be determined definitely whether they should be separated or not. (Note that it is not illegal to have several declarations about the same variable in MAD/I. Requiring all information about a variable to be made in the same declaration statement might simplify some of the declaration-processing problems but it would lessen the convenience to the user.)

5. THE BLOCK STRUCTURE ALGORITHM

Several routines can be called upon to perform various functions when the compiler is scanning the descriptors before and during parsing. The algorithm will describe these routines and the circumstances under which they are called. A particular symbol can have associated with it several variables whose names are the same, but at most one variable per block. The job of this algorithm is to determine to which blocks such variables belong, and then to map the symbols in the n-tuples which result from the parse into the proper variables for that point in the program.

At any point during the scan of the input descriptors, a symbol can be in one of four states with respect to a block: "unreferenced," "referenced," "declared," and "not new." For the declared state there is a variable associated with that block. This is not true for the other three states except when the block is the outermost block. The outermost block is a special case, of course, since it is not surrounded by another block. In the outermost block a variable must be in one of the first three states; i.e., it cannot be in the "not new" state.

Note that in PL/I-like languages, a symbol like IF can represent either a variable or a statement keyword,

depending upon context, and the dilemma must be resolved before this algorithm will work. In MAD/I this is not a problem, since keywords and variables are represented by distinct lexical classes. Subsequently we will assume that this problem has been solved for any given language, and we are considering only symbols which represent variables.

A "referenced" symbol in a block is one which has been encountered in that block but for which no declarations of any type have occurred, including 'NOTNEW' and 'GLOBAL'. A symbol is termed "declared" when it has been declared in the block but not declared 'NOTNEW' or 'GLOBAL'. A variable is created for it which is a carrier of mode and other declared information. A symbol in a "not new" state has been declared 'NOTNEW' or 'GLOBAL' in that block, and its status with respect to that block cannot be further altered. A symbol cannot have "not new" status with respect to the outermost block since that status indicates that it is a symbol which belongs to a block surrounding the one under consideration, and which cannot be declared to belong to that block.

When the beginning of a block is encountered a routine called BEGINBLOCK is called which pushes down the status of all symbols of the current (old) block, if any, and sets the status of all variables to

"unreferenced." Further processing of the new block may then proceed.

When a symbol is encountered in a block it is passed to a routine called SETREF. If the symbol is in "unreferenced" status it is set to "referenced" status for that block, otherwise nothing is done.

When a symbol is declared in a block, except for a declaration of 'NOTNEW' or 'GLOBAL', it is passed to the SETDECL routine. If the symbol is "declared" in the block nothing is done. If the symbol is "unreferenced" or "referenced" in the block, then a variable is created for that block with the name of the symbol, and the symbol is set to "declared" status. Note that declaration information is always applied to the first variable encountered for the symbol when the search is made outward from the current block to surrounding blocks. Thus the declaration information, if any, which is associated with the declared symbol is to be applied only after SETDECL has been called. If the symbol is in "not new" status, a search is made outward, successively through surrounding blocks until the symbol is found in other than "not new" status. Then the symbol is treated with respect to that block in the same manner as an "unreferenced," "referenced," or "declared" symbol would be for the current block. The variable that results from the SETDECL operation is the one to which the original

declaration information was assigned.

When a symbol is declared 'NOTNEW' in a block it is passed to a routine called SETNOTNEW. At this point, one of three situations will occur:

1. If the symbol is already "not new" or if the symbol is 'DEFAULT', or if the current block is the outermost block, nothing is done.
2. If the symbol is "unreferenced" or "referenced" it is set "not new" in the current block. A search is then made outward through the containing blocks and the status of the symbol is determined for each block, until the symbol is found in other than "not new" status. If that status is "unreferenced" then it is set to "referenced."
3. If the symbol was in "declared" status when SETNOTNEW was called then it is set to "not new" status in the current block. A search is made outward through all the surrounding blocks until the symbol is found in other than "not new" status. If that status was "unreferenced" or "referenced" it is changed to "declared" status, and the variable of the the symbol for the current block is used as the variable for the symbol in the outer block where the search ended. If the search ended

on a variable with a "declared" status then we have an interesting situation of two variables in existence which should be replaced by a single variable for the outer block. These variables will have to be "merged." Any declarations declared on the inner variable must be copied over to the outer variable, with appropriate error comments if conflicts are discovered. In the case of MAD/I a variable may be declared only once with mode information; an attempt to do so more than once causes an immediate error comment, except in the case where the two variables are being merged into one, as above, due to the 'NOTNEW' declaration. If modes were declared for each of the variables before they were merged the conflict will not cause an error comment until the 'NOTNEW' declaration is encountered.

When a symbol is declared 'GLOBAL', SETNOTNEW is called for that symbol for the current block and for each surrounding block. Hence, SETNOTNEW has a block as one of its arguments.

The actions of the above-described routines determine, as closely as possible, the status of symbols within a block by the time the end of that block is

reached. The end of the block triggers a call on the routine ENDBLOCK which will complete the determination of the status of all symbols which have variables in the block. In the outermost block, the action is simple: the symbols referenced in the block are made into variables with "declared" status. If it is not the outermost block, then the symbols "referenced" in the block are treated in one of two ways:

(1) If 'DEFAULT' is declared for the block, including the attribute 'NEW', then all the "referenced" symbols are made "declared" symbols for the block, and variables are created for each such symbol.

(2) If 'DEFAULT' was not declared 'NEW' in the block, the status of "referenced" symbols is checked against the next outer block, since these symbols belong there.

If a symbol is "unreferenced" in the next outer block, it is set to "referenced." After one of these two actions is done, we are finished with the inner block, and the status of all symbols is "popped" back to that of the next outer block.

When a program has been completely scanned, each variable will have been assigned to its appropriate block. It is then possible to go through the blocks from the outermost to the innermost to supply default

information for those variables. It is also possible to go through the parsed form of the program, replacing each occurrence of a symbol with the appropriate variable for that block. This process is called the "remap" phase in the current MAD/I compiler. There is no restriction on whether remapping or default assignment is done first as far as the algorithm is concerned. The method of default assignment on a block and variable basis was described in Section 2. The method of remapping is described below in fairly general terms.

Many "tricks" could be used for implementing the described routines, for the method of representing symbols and variables, for remapping and for default assignment, but these tricks all depend upon the representation of descriptors, the method of parsing, the sort of declarations to be stored and the method of storing, etc. These details, in turn, depend on the language and the particular compiler implementation. In the case of MAD/I, as the present implementation of the compiler and language evolved, it almost always increased rather, than decreased, in complexity, and hence it is presently difficult to debug.

The remap phase assumes that the beginning and end of each block are easily spotted in the parsed form, and that there is an easy way to search through the parsed form such that the beginning and end of each block are

encountered in the same order as in the scan to produce calls on BEGINBLOCK and ENDBLOCK, and such that the symbols previously encountered within a given block are again encountered in that same block. Assume that if we have a symbol representing a variable then we can easily find a place to look for the current variable representing that symbol. Let us assume that there is a field within the symbol which can point to the variable. Also assume that there is a similar field associated with each variable of each block, and that the field initially points to itself. At the beginning of each block, for each variable belonging to that block, exchange the symbol field with the variable field. At that point, the symbol points to the current variable. At the end of the block perform the same exchange. As we progress through the parsed program the symbols will effectively be pushed and popped properly so that whenever a symbol is encountered we can replace it with the variable it represents at that point. This is a workable alternative to the one of keeping a pushdown stack for each symbol, the current variable being at the top of the stack. Both of these approaches assume that remapping would be more expensive if we simply searched the blocks outward from the current block until a variable of the right name is found.

6. CONDITIONAL DECLARATION HANDLING IN MAD/I

A conditional declaration is one which is applied when a variable appears in a certain context, unless that variable has had an explicit declaration which would conflict. There are a number of such declarations in PL/I. In MAD/I there is presently only one. If the "." operator has been used on a variable (the function call operator), then there is a conditional declaration applied to the variable. The declaration says that if no mode or storage class information has been declared about the variable, then it is to be taken as an 'EXTERNAL' 'ENTRYPOINT', which returns a value of default mode when called. Thus it is the name of an externally compiled subroutine. Notice that the explicit declarations are applied first, then conditional declarations, if any, and finally default declarations. This is also the usual order in PL/I.

Conditional declarations are handled somewhat differently from other declarations because of the convention that a conditional declaration does not imply that a variable has been declared "new" to a block. Conditional declarations have no influence in determining what block the variable resides in. This is contrary to the effects of all other declarations. Therefore the previously described algorithms do not work for conditional declarations.

Such declarations are easily handled, however, in the following manner:

When a conditional declaration is discovered, it is saved in some way on a list associated with the current block, as is the symbol to which it will conditionally apply. After the end of the block is encountered and it is closed out (so that the variables that belong to the block are known), the list of conditional declarations is searched. For each symbol on the list there is either an associated variable that now belongs to the block, or else there is not. If there is such a variable, then the conditional declaration belongs with it. Otherwise, the symbol and conditional declaration are put on the list for the next outer block.

7. SOME IMPLEMENTATION DETAILS

This section discusses some details of the implementation of the block structure algorithm in the present version of the MAD/I compiler.

In the MAD/I compiler, symbols and variables have the same form. The symbols themselves are used as variables for the outermost block, thus economizing storage. Associated with any block are three lists, one each for symbols which are "referenced," "declared," and "not new." In addition, in each symbol there is a two-bit field which specifies which of the four states the symbol is in. Each block points to the next outer block, thus facilitating popping back to or searching to the next outer block. At any given time the symbol carries the current information declared about the current variable associated with that symbol. Any previously specified variable associated with that symbol in an outer block has its information pushed down in some fashion. Thus the symbols which are declared in a block must have their contents appropriately popped at the end of each block. In any case, as long as a call is made on SETDECL before applying the declaration information, the declaration information associated with a symbol can be stored with the symbol itself, and it is not necessary to search for an associated variable at that time. This

is carried out according to how pushing and popping are done; however, such details are outside the scope of this paper.

However, the method of copying attribute information from default symbols is relevant. Associated with each mode are two routines, each having two arguments, a "from" variable and a "to" variable. The intention is to copy information from one to the other under certain circumstances. One routine is used when the "to" variable has no mode information, in which case the information is copied to it from the other symbol (which may be the default symbol or a subtype of the default symbol). The other routine is used to copy mode information when the "to" symbol already has mode information. In that case the "from" symbol may be used to copy information to a subtype of the "to" symbol which does not have mode information. Needless to say, the routines are recursive. Another routine, let us call it COPYATRS, selects one of the two routines just described and decides which mode to use. The routines are also used to set information (associated with a mode) which was not explicitly declared, such as length or dimension information.

The action of COPYATRS is described here, with two arguments, FROM and TO.

1. If TO has a mode, then select the "to" routine for that mode and pass to it FROM and TO as its arguments. Exit upon return of control from the

"to" routine.

2. If TO has no mode set, then select the "from" routine associated with the mode of the FROM symbol. Exit upon return from the "from" routine. Note that there will always be a mode on the FROM symbol (if everything is properly debugged).

Consider the "from" routine for an array mode, and call the routine FROMARRAY. This routine is called when its TO symbol has no mode, and thus the job of FROMARRAY is to copy the FROM symbol information to the TO symbol. Therefore it will copy the array mode, the dimension information, and any other mode-associated information to the TO symbol. It will create a symbol-like construct associated with the TO symbol which is to carry mode information for the component mode of TO. Let us denote the carrier by component-of-TO; there is a similar carrier for component-of-FROM. Then a call on COPYATRS (component-of-FROM, component-of-TO) is made to copy the component information. It will always work out that the data type information for the FROM symbol will be complete.

Consider the "to" routine for an array mode, and call the routine TOARRAY. This routine is called when the TO symbol already has a mode, and that mode is array mode. The job of TOARRAY is to set any remaining undeclared information about the TO symbol. For example, if

suffix (i.e. dimension), information was omitted from the array declaration, then default information would be set for it. (In MAD/I such information is associated with the mode and not taken from a default which can be declared about arrays. There is no way presently in MAD/I to say that the default dimensions of an array are to be 3 x 3, for example. In principle this would be possible, however.) Next the TO symbol is examined to see if it has a component mode carrier, and if not, it is attached to the TO symbol. Whether or not the carrier was there before, a call is made on COPYATRS (FROM, component-of-TO). This will cause defaults to be set on the component-of-TO symbol, if needed.

Obviously, if the modes involved had no components, then the associated "from" and "to" routines would be simpler. The "from" and "to" routines may do other jobs also, such as returning length and alignment information to their caller. Thus the initial caller of COPYATRS would call it with a symbol to be allocated, as the TO symbol, and the 'DEFAULT' symbol as the FROM symbol. It would get in return the length and alignment of the TO symbol. Notice that the COPYATRS routine is also used to set the information on the default symbol itself. When starting allocation of variables in a block, first COPYATRS is called with FROM being the default of the next outer block (which has already been taken care of),

and TO being the default of the current block. For the outermost block there is no next outer block, so a special FROM symbol is used which has the "default default," i.e., the default mode which is used if none is declared in the outermost block.

8. CONCLUSION

Everything described in this paper has been successfully implemented in a compiler for the MAD/I language which runs on an IBM/360 model 67 under the University of Michigan timesharing system, MTS. For various reasons which are not relevant here it is a very large compiler. Since the system provides very large virtual memory for execution (about four million bytes), the compiler is written to take advantage of a large virtual memory. MAD/I was also written mostly in an experimental compiler implementation "macro" language, which allows easy modification of the compiler, even at run time, for those who know the incredible intricacies of the compiler. These factors, of course, have influenced the implementation of the block structure and default facilities. Nevertheless, it is felt that what we have learned about these facilities may be useful to compiler implementers whose design requirements impose very different constraints on their compilers.

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R & D
(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
THE UNIVERSITY OF MICHIGAN
CONCOMP PROJECT

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP

3. REPORT TITLE
DEFAULTS AND BLOCK STRUCTURE IN THE MAD/I LANGUAGE

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Memorandum 31

5. AUTHOR(S) *(First name, middle initial, last name)*
Allen Springer

6. REPORT DATE
July 1970

7a. TOTAL NO. OF PAGES
32

7b. NO. OF REFS
0

8a. CONTRACT OR GRANT NO.
DA-49-083 OSA-3050

b. PROJECT NO.

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)
Memorandum 31

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. DISTRIBUTION STATEMENT
Qualified requesters may obtain copies of this report from DDC.

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY
Advanced Research Projects Agency

13. ABSTRACT

This paper describes the default and block structure mechanisms of MAD/I, a PL/I-like language, and the interaction of these mechanisms with the three types of MAD/I declarations: explicit declarations, default declarations, and conditional declarations. MAD/I allows the programmer extraordinary control over the default assignment of data types to variables, and also allows the programmer more than usual control over the scope of variable names in block structure. The interaction of these two facilities can make the handling of declaration information a difficult problem. This paper outlines an algorithm in which this information is processed "on the fly" in the first pass of the compiler over the source program, and then the symbol table is processed to assign defaults and allocate storage. A simple second pass over a transformed version of the source text resolves the scope and interpretation of variable names.

DD FORM 1 NOV 65 1473

UNCLASSIFIED
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
MAD/I PL/I defaults declarations block structure						