transition table which contains lists of all predecessors and successors of each statement in the routine.

Table Generator

The table generator is called into action by the parser whenever a new item of information is to be stored. The information is organized in a set of tables (described in [5]). Depending on the class of information to be stored (variable occurrence, COMMON block declaration, subroutine call, label reference, etc.), the table generator decides where and how the item should be stored and alters the tables accordingly.

For example, in the statement

```
A = B(I)
```

the table generator would be called three times. First the parser would recognize that variable A was being used as "output" to an assignment statement (appears on the left side of an"="). The table generator would first check to see if A had been stored in the symbol table. If it had not, the table generator would store it there along with information about its class (variable) and type (floating point). An entry would then be made in the usage table showing that A had been used as output to an assignment statement. Next the parser would call the table generator telling it that B was being used as a function (a guess by the parser). The table generator would then check to see

if B had been stored in the symbol table. If it had not, it would make an entry for it and assume it was a function call. If B had already been stored in the symbol table, it would check to see if it had been declared an array. If so it would send this information back to the parser. As for variable A, an entry would be made in the usage table for the occurrence of B, this time as "input" to an assignment statement. The parser would then call the table generator telling it that I was being used as either a parameter or subscript (depending on whether B was a function or an array). The table generator would again make sure that I had been stored in the symbol table and would create a new entry in the usage table.

After each routine of the user's source program has been completely parsed, the table generator writes the set of local tables for that routine to a permanent file to allow access to the tables later by the diagnostic routines. After all routines have been parsed, the global tables are written on the file also. The information in the tables can also be printed for user examination.

## Diagnostic Routines

Each of the diagnostic routines of FACES is written as a separate module and can be executed independently of the other routines. Each routine, however, requires information from some of the tables produced by the FORTRAN Front

End. For example, the common block alignment routine needs access to the storage allocation table which contains information about the way in which all common blocks are set up. It also must reference the symbol tables from each of the routines to be analyzed. From the combined analysis of both the tables, the routine can compare the structure of common blocks declared in different routines.

The parameter alignment routine needs information about the occurrence of parameter lists from the parameter table and information about the type and length of each parameter from the symbol tables. By analyzing the tables, the routine can make comparisons between actual parameter lists and formal parameter lists to insure that type and length match appropriately.

The variable initialization routine finds all occurrences of variables used in a way that assumes the variable has a value (such as input to an assignment statement). This information is obtained from analysis of the usage table and the symbol table. Then the routine traces a backward path (using the graph of the program which can be derived from the node table, successor table, and predecessor table) from each of the noted variable occurrences to insure that the variable is initialized somewhere before its use.

The variable trace routine uses a procedure similar to the variable initialization routine. The routine takes a certain variable in a specific statement and traces either

its history or future to find variables that have affected its value or variables whose value will be affected by the certain variable in question. (See previous section.) Here again information from the node table, successor table, and predecessor table is used to trace the logical flow of the program.

The entire structure of FACES has been designed in a modular fashion to allow as much flexibility as possible. The FORTRAN Front End has been written as a completely separate entity to allow its use as a general purpose data base collector. The data base formed from it could be used for a variety of purposes (to be described in the next section) other than the diagnostic routines that have been implemented. The FORTRAN Front End has been written in three basic modules--the scanner, parser, and table generator--to fascillitate redesign when necessary. For example, recognition of a new statement type could be added to the parser with no change being made to the scanner. Each of the diagnostic routines has also been designed as a separate entity to allow the user to execute only the ones needed by him.

## PROBLEMS WITH FACES

The major problems with FACES lie more in what it does not do than in what it does do. As the system currently exists, only four diagnostic routines are available. These four routines may be very helpful to a programmer but they

do not begin to cover all the possibilities for program analysis. No dynamic analysis has been implemented, and the data base collected by the FORTRAN Front End has not been used to the fullest extent. We shall not dwell here though on those features not included in the system. The next section will discuss possibilities for extensions to FACES that could solve this basic problem of limited scope.

Another problem with the FACES system is the large amount of memory it uses. This problem arises from the large tables used to store the data base for the source program. The tables have been designed with tightly packed fields in an attempt to use a minimum amount of space, but the amount of information to be stored is vast in scope. There are two possible solutions to this problem, neither of which seems desirable for the purposes for which the system was intended. One solution would be to decrease the amount of information stored and to choose a subset of functions of the entire system to be implemented. This of course would limit the system's ability to serve as a general-purpose data base collector. Another solution would be to limit the size of the programs that could be analyzed by FACES. The table sizes could then be decreased because of the decreased upper limit of information to be stored. This solution, however, does not seem desirable since large programs are often in need of automatic analysis more than small ones. Thus it seems that the large amount of memory used will remain a problem of FACES.

Another problem of the system is that the diagnostic routines require a large amount of auxilary storage input-output when analyzing the tables. This problem stems mostly from the basic organization of the tables. Since each routine has its own set of local tables, a diagnostic routine which performs analysis on a global level (such as the common block alignment routine) must read a local table from one module, use it as needed, and then read the same local table from a different routine into the same space. This problem could be solved only be reserving a large enough space in memory to hold all local tables. As can be seen from the preceding paragraph, the space problem is already critical in the FACES system; thus a solution to the file input-output problem seems unlikely.

The FACES system, like most large software systems, has problems in its design. The three largest problems of FACES seem to be its present limited scope, the large amount of core space it uses, and the large amount of file input-output it requires. The first problem has many possible solutions to be discussed in the next section. The other problems, although somewhat undesirable, do not seem to be serious hinderances to the system's usefulness.

## POSSIBLE EXTENSIONS TO FACES

The FACES system could be extended to a great degree by making additions to the already existing diagnostic routines.

These additions should require no changes to the FORTRAN Front End. Since the diagnostic routines are written as separate modules, the changes should require only a small amount of effort.

An obvious improvement to the system would be to extend the variable trace to the global level. This would mean a programmer could trace the history or future of a variable not only within subprogram boundaries but over the entire range of a program. This would be a much more effective means of tracing value errors in large programs with many subprograms, since it often happens that a mistake in one subprogram affects other subprograms also.

A similar improvement would be to extend the variable initialization routine to the global level. This would enable the programmer to verify that all variables in the program are initialized at some point. The extension would involve a considerable amount of change in the present local-oriented diagnostic routine since all variables in common and in parameter lists would have to be traced. The new global-oriented routine would, however, eliminate all guesswork in verifying variable initialization.

Another diagnostic routine that could be added to the present system is a static path trace. With this routine, a programmer could find all possible paths leading to a certain point in a program or all possible paths stemming from a certain point in a program. The routine would act in a way similar to the variable trace, but would give the

programmer a list of statement numbers (for each path) rather than a set of variables. The routine could be designed for use on a local or global level.

Three minor additions could be made to the FACES system that might prove useful to programmers. Routines could be added to list all user undefined functions and subroutines, statement numbers never referenced, and segments of unreachable code. The list of undefined functions and subroutines would help the user verify that he has included all necessary subprograms. Statement numbers that are defined but never referenced are not illegal in FORTRAN but may often indicate unintentional programming slips. Sections of unreachable code (segments of code to which there is no possible path) are also often indications of errors or garbage left from previous versions of the program. These three possible additions to FACES are relatively minor features that some compilers already include. However, these additions could be made to FACES with very little effort and would prove helpful in installations where the compiler does not include all three features.

Another possibility for improving the usefulness of FACES would be to design a routine that could answer general information-retrieval questions. The routine might be equipped to answer questions such as:

1. In what statements does variable A appear?
2. From which subprograms and where in those subprograms is function F called?

3. In which subprograms does common block C1 appear?

The list could go on. With this facility, a programmer could find out details about his program without tracing through the code. This feature would be particularly applicable to use on an interactive system.

Questions such as those listed above might also be useful for documentation purposes. A documentation routine could be added to FACES that would extract from the data base information such as common block structure, variable usage, and subroutine structure. The information could be printed in a report-like form to serve as a supplement to the programmer's documentation.

Another possibility for an additional diagnostic routine is to perform checks on array subscript ranges to insure that references to the array do not exceed the limits of the declared dimensions. A preliminary investigation of the problem has already been done to check ranges on array references whose subscripts are constant integers. For example, in the statement

```
ARRAY(20) = 0
```

the routine would check to see if 20 is less than or equal to the declared dimension of ARRAY. The real problem arises though when array subscripts have variable values. The investigation of this problem in fact would be of much more value to program analysis since most array subscripts do at some time have variable values. The solution to this problem would have to involve some type of interval arithmetic

in which all possible values of the variable subscript are traced to verify that they fall within a certain range. The information for this type of analysis is available in the data base, but the construction of a diagnostic routine to perform the analysis would involve a considerable amount of effort.

The previously mentioned suggestions for extensions to FACES have been those that would involve additions and alterations to the diagnostic routines only. In other words, the information is already available in the data base collected by the FORTRAN Front End, but routines are needed to organize and extract the information appropriate to the specific purpose.

The FACES system could be expanded even further if additions were made to the FORTRAN Front End. Since the Front End already scans and parses the FORTRAN code, these additions would not have to be major overhauls but could hopefully be added in a modular fashion to the present Front End system. The following paragraphs will describe the possibilities for such extensions.

At present, the FACES system does not recognize FORTRAN FORMAT statements. The addition of FORMAT processing routines to the Front End could provide the basis for further expansion of the diagnostic routines to include analysis of input and output. A useful feature might be to compare input/output lists with their corresponding FORMAT statements to insure that each I/O item has a reasonable I/O specification

in the FORMAT statement. For instance, the routine could check to see that an integer variable is output with an integer FORMAT specification. This feature would be especially helpful in preventing input errors that can cause the failure of an entire program.

Another possible extension to FACES would be to incorporate into the system the ability to statically estimate memory size and execution time requirements--that is, to analyze a FORTRAN program before it is executed to estimate the amount of memory space and execution time it will require. Of course this feature would require knowledge of the particular computer on which the program would be run. Information about the way in which each FORTRAN statement is implemented in machine language would also be necessary. From this information and that already collected by the FORTRAN Front End, reasonable estimates could be made about size and time requirements. This might help the programmer decide if he needs to overlay part of a program or optimize code to cut down on time requirements. The feature could be especially helpful to programmers writing programs for computers to which they do not have immediate access.

Dynamic analysis is another feature that could be added to the FACES system. At present, the FORTRAN source code is only analyzed statically, but several dynamic analysis methods could be added without great trouble. A dynamic variable trace is one possibility that is often found useful, but many compilers already include this fea-

ture. A dynamic trace of path execution might be a helpful addition also. The frequency of execution of each path in the program could be monitored to help the programmer find those areas of the program that use most of the execution time. This information would be helpful in program optimization. The dynamic path trace feature would also be helpful in testing the program to show the programmer areas of code that have never been executed. The programmer could then devise data that would cause those sections to be executed.

A final suggestion for the expansion of FACES is to include a method of test data generation. The most ideal solution would be to automatically generate test data cases that together would cause execution of all branches in the program being tested. This procedure would involve two basic steps:

1. Determine a minimum set of program paths that would cause execution of all branches in the program.
2. Generate test data sets that would cause execution of each of those paths determined by 1.

The first step could be done by a method similar to that described by Krause and Smith [7]. (See chapter 2 for explanation of their method.) The automatic generation of test data sets would require a considerable amount of additional effort. An interval arithmetic processor would have to be employed to trace variable values back to their input

points. The advantages of this feature would probably be well worth the effort, though, since program testing time would be greatly reduced.

It can be seen that FACES has many possibilities for useful extensions to its present set of tasks. The broad-ranged data base collected by the FORTRAN Front End provides the information to allow these extensions to become practical possibilities. Although FACES is somewhat limited without these extensions, the addition of these features could make it a truly general-purpose FORTRAN analyzer.

## CONCLUSION

The FORTRAN Automatic Code Evaluation System has been described with attention given to its purposes, the tasks it has accomplished, methods used in the system, problems in its design, and possible extensions to its present abilities. The system was designed to be used in validation of large FORTRAN programs but provides a data base that can be used for other purposes. At present the system has four diagnostic routines that may help a programmer find and correct errors in a program. The FACES system has several problems in its design, the major one being that it does not at present contain the wide variety of features which can be derived from the data base. Suggestions for extensions have been made which would enable FACES to aid in not only program validation but also in documentation, maintenance, performance evaluation, optimization, and structural and timing analysis.

# IV. EXAMPLE OUTPUT FROM THE FORTRAN AUTOMATIC CODE EVALUATION SYSTEM

The FORTRAN Automatic Code Evaluation System at present has four diagnostic routines which extract information about a user's program from the tables produced by the FORTRAN Front End. (As mentioned in previous chapters, the FORTRAN Front End scans and parses the user's FORTRAN code and builds tables of information about the program which are stored on file for later reference by the diagnostic routines.) The four presently available diagnostic routines are the following:

1. Common block alignment routine
2. Parameter alignment routine
3. Variable initialization routine
4. Variable trace routine

Example output from each routine will be discussed in this chapter.

For the purpose of obtaining illustrative examples, a set of three FORTRAN routines has been designed which have troublesome constructs that can be detected by the diagnostic routines. (These routines appear in Figure 4.) The FORTRAN Front End was used to analyze these routines and build tables of information. Each of the diagnostic routines was then executed to extract the appropriate information from the tables. (See Appendix for a listing of the tables produced by the FORTRAN Front End in this example.)

```
MODULE 201

 1                      SUBROUTINE TEST1
 2                      DIMENSION KWORDS(10),LWORDS(12)
 3                      COMMON/BLOCK1/ALPHA,BETA,GAMMA,DELTA,TOTA
 4                      COMMON/BLOCK2/IARRAY(20),VAR1,VAR2
 5                      COMMON/BLOCK3/MATRIX(10,30),INDEXM
 6                      IF(TOTA.EQ.0) GO TO 25
 8                      DO 20 K=1,20
 9           20         IARRAY(K)=0
10           25         BETA=VAR1 * VAR2 + FUNCT1(GAMMA,DELTA)
11                      CALL SUB2(BETA,KWORDS)
12                      IF(BETA.GT.0) GO TO 30
14                      ALPHA=VAR1/BETA
15                      CALL SUB2(ALPHA,KWORDS,1)
16                      RETURN
17           30         CALL SUB2(ALPHA,LWORDS)
18                      INDEXM=1
19                      RETURN
20                      END

MODULE 202

 1                      SUBROUTINE SUB2(PARAM1,JARRAY)
 2                      COMMON/BLOCK1/ALPHA,BETA,GAMMA,DELTA,TOTA
 3                      COMMON/BLOCK2/IARRAY(20),VAR1
 4                      COMMON/BLOCK3/MATRIX(10,30),INDEXM
 5                      DIMENSION JARRAY(10),ARRAY(2,3)
 6                      DO 30 I=1,10
 7                      MATRIX(I,1)=0
 8           30         JARRAY(I)=PARAM1 * I
 9                      X=GAMMA * DELTA
10                      Y=ARRAY(1,1)
11                      Z1=FUNCT1(X,Y) + VAR2
12                      Z2=FUNCT1(X,0)
13                      PARAM1=Z1 * Z2
14                      RETURN
15                      END

MODULE 203

 1                      FUNCTION FUNCT1(X1,X2)
 2                      COMMON/BLOCK1/ALPHA,BETA,GAMMA,DELTA,EPSILON
 3                      COMMON/BLOCK2/IARRAY(20),VAR1,VAR2
 4                      COMMON/BLOCK3/MATRIX(10,30),INDEXM
 5                      IF(EPSILON.LT.0) GO TO 50
 7                      Y=ALPHA + BETA + GAMMA * VAR1
 8                      GO TO 60
 9           50         Y=ALPHA + BETA + DELTA * VAR1
10                      IF(X1.LT.X2) GO TO 60
12           60         DO 70 I=1,20
13                      TEMP=(IARRAY(I) + X1)/X2 + Y
14           70         FTEMP=BETA+TEMP
15                      IF(INDEXM.EQ.0) GO TO 80
17                      DO 75 K=1,INDEXM
18           75         MATRIX(1,K)=Y
19                      INDEXM=FTEMP
20           80         FUNCT1=FTEMP
21                      RETURN
22                      END
```

Figure 4: FORTRAN Test Routines

## RESULTS OF COMMON BLOCK ALIGNMENT ROUTINE

The output from the common block alignment routine appears in Figure 5. The purpose of this routine is to check the alignment of each common block declaration to insure that each declaration of a specific common block is the same length, that corresponding elements within the declarations are of the same type, and that corresponding elements within the declarations have the same name (if the user so chooses).

A separate listing appears for each common block in the program being analyzed. The module number is listed for each routine in which the common block appears. If an error appears in the common block declaration of a specific module, the type of error found is printed beside the module number. (Errors are derived by comparing each common block declaration with a "standard" common block. The "standard" is chosen by analyzing the first three declarations of a common block that are encountered. If two out of the three match exactly, then the declaration for the matching common blocks is chosen to be the standard.)

Three types of errors, called "size", "name", and "type" can appear. In the results of the alignment check for common block BLOCK1 (see Figure 5), a "type" error appears in module 303. By looking at the example routines, one can see that in function FUNCT1, the sixth variable of BLOCK1 is called EPSILON (floating point variable), whereas the corresponding variable in the standard BLOCK1 is called

```
RESULTS OF ALIGNMENT TEST FOR COMMON BLOCK LABELED *BLOCK1  *
  FIRST MODULE LISTED WITHOUT ERROR CONTAINS THE STANDARD COMMON BLOCK

  MODULE NUMBER          ERROR
       201
       202
       303               TYPE


RESULTS OF ALIGNMENT TEST FOR COMMON BLOCK LABELED *BLOCK2  *
  FIRST MODULE LISTED WITHOUT ERROR CONTAINS THE STANDARD COMMON BLOCK

  MODULE NUMBER          ERROR
       201
       202               SIZE
       303


RESULTS OF ALIGNMENT TEST FOR COMMON BLOCK LABELED *BLOCK3  *
  FIRST MODULE LISTED WITHOUT ERROR CONTAINS THE STANDARD COMMON BLOCK

  MODULE NUMBER          ERROR
       201
       202
       303               NAME
```

Figure 5: Results of Common Block Alignment Routine

IOTA (an integer variable). Thus a "type" error exists. (If both variables had been of the same type but had different names, a "name" error would have appeared.)

In the results of the alignment test for common block BLOCK2, a "size" error appears in module 202. From the example routines, one can see that in the declaration for BLOCK2 in subroutine SUB2, VAR2 has been left off the end. This indicates a "size" error since the common blocks are not the same length. A "size" error would also appear if the common blocks were the same length but arrays within the block were of differing lengths.

In the results of the alignment test for common block BLOCK3, a "name" error appears in module 303. This results from the spelling of "MATRX" in the declaration of BLOCK3 in function FUNCT1. In the other modules the variable is called "MATRIX". (The name check is an optional feature of the common block alignment routine, since some programmers do not intend for common block elements in different routines to have the same names.)

## RESULTS OF PARAMETER ALIGNMENT ROUTINE

The results of the parameter alignment routine are shown in Figure 6. The purpose of this routine is to check the parameter lists of calls to all subroutine and functions to insure that the actual parameter lists are the same size as the formal parameter list and to insure that each actual

```
PARAMETER ALIGNMENT MESSAGES

TOO MANY PARAMETERS IN CALL TO SUB2      FROM MODULE 201

ARRAY LWORDS  IN CALL TO SUB2    FROM MODULE 201 DIFFERS IN DIMENSIONS
     FROM THE CORRESPONDING FORMAL PARAMETER.
PARAMETER NO.  2 IN CALL TO FUNCT1  FROM MODULE 202 IS OF WRONG TYPE.

END OF PARAMETER ALIGNMENT CHECK
```

Figure 6: Results of Parameter Alignment Routine

parameter is of the same type as its corresponding formal parameter.

In the results of this example test, three errors appear. The first message indicates that there are too many parameters in a call to subroutine SUB2 from subroutine TEST1. One can see that in line 15 of TEST1, the call to SUB2 has three parameters. The formal parameter list, however, only has two parameters. A similar message would be printed if the formal parameter list had more parameters than the actual parameter list.

The second message warns that the array "LWORDS", in a call to subroutine SUB2 from subroutine TEST1, differs in dimensions from its corresponding formal parameter. In the formal parameter list for SUB2, the second parameter "JARRAY" is a linear array of length 10. The actual parameter "LWORDS" in line 17 of TEST1 is a linear array of length 12. This does not constitute a FORTRAN error, but may be an unintentional slip by the programmer. The message serves as a warning rather than an error.

The third parameter alignment message warns that the second parameter in a call to function FUNCT1 from subroutine SUB2 is not the same type as its corresponding formal parameter. The actual parameter "0" in line 12 of SUB2 is an integer constant, whereas the corresponding formal parameter "X2" is a floating point variable.

## RESULTS OF VARIABLE INITIALIZATION ROUTINE

The variable initialization routine checks the initialization of variables on a modular level. That is, for each module requested by the user, the variable initialization routine will check all variables in that routine to see if the variables are properly initialized within the bounds of that specific routine. Variables which appear as formal parameters or as common block elements are assumed to be initialized.

Two types of messages may appear. One message indicates that a variable is never initialized; that is, the variable is used in the routine but is never assigned a value. The second type of message indicates that a variable may be referenced before being initialized. This message is caused by two situations--one, when the variable appears as an actual parameter without being initialized, and second, when a path exists through the routine in which the variable may be referenced before being assigned a value. In the first case, it is not known whether the variable is being used as an input or output parameter; thus the message is meant as a warning to the programmer to make sure the parameter is being used as an output parameter. In the second case, the variable is given a value somewhere within the routine, but a possible path exists through the program in which the variable would be used before being assigned the value.

The results of the variable initialization check

```
RESULTS OF INITIALIZATION CHECK FOR MODULE TEST1

THE VARIABLE * ZETA      * IS NEVER INITIALIZED
THE VARIABLE * I         * MAY BE REFERENCED BEFORE BEING INITIALIZED.
THE VARIABLE * LWORDS    * MAY BE REFERENCED BEFORE BEING INITIALIZED.
THE VARIABLE * KWORDS    * MAY BE REFERENCED BEFORE BEING INITIALIZED.

INITIALIZATION CHECK HAS BEEN MADE ON EACH VARIABLE
```

```
RESULTS OF INITIALIZATION CHECK FOR MODULE FUNCT1

THE VARIABLE * FTEMP     * MAY BE REFERENCED BEFORE BEING INITIALIZED.

INITIALIZATION CHECK HAS BEEN MADE ON EACH VARIABLE
```

Figure 7: Results of Variable Initialization Check

appear in Figure 7. For module TEST1, four messages appear. The first message indicates that variable "ZETA" is never initialized. One can see that in line 14 of TEST1, "ZETA" is used as input to an assignment statement. However, it is never assigned a value. The other three messages produced for subroutine TEST1 warn that variables "I", "LWORDS", and "KWORDS" may be referenced before being initialized. Since the variables are used as actual parameters to subroutine SUB2, it is not known whether they should have been assigned a value previous to being used as parameters.

A single message is produced for module FUNCT1 warning that variable "FTEMP" may be referenced before being initialized. This message is generated even though "FTEMP" is assigned a value in line 13, since a path exists from line 10 to line 20 in which "FTEMP" is referenced without being assigned a value.

## RESULTS OF VARIABLE TRACE ROUTINE

The variable trace routine is used to trace the history (backward trace) or future (forward trace) of a variable at a particular point in a routine. In the examples listed in Figure 8, a backward trace is shown for variable "PARAM1" in subroutine SUB2 and for variable "TEMP" in function FUNCT1. In the backward trace, all variables that could have possibly affected the value of the variable prior to the execution of a particular node are listed. A forward trace

TARGET MODULE FOR PATHS = SUB2
THE VALUE OF *PARAM1 * AT NODE 13 CAN BE AFFECTED BY THE FOLLOWING

| VARIABLE | NODE OF USE |
|---|---|
| ARRAY | 12 |
| DELTA | 9 |
| FUNCT1 | 11 |
| FUNCT1 | 12 |
| GAMMA | 9 |
| VAR2 | 11 |
| X | 11 |
| X | 12 |
| Y | 11 |
| Z1 | 13 |
| Z2 | 13 |

TARGET MODULE FOR PATHS = FUNCT1
THE VALUE OF *TEMP * AT NODE 13 CAN BE AFFECTED BY THE FOLLOWING

| VARIABLE | NODE OF USE |
|---|---|
| ALPHA | 7 |
| ALPHA | 9 |
| BETA | 7 |
| BETA | 9 |
| DELTA | 9 |
| GAMMA | 7 |
| IARRAY | 13 |
| I | 13 |
| VAR1 | 7 |
| VAR1 | 9 |
| X1 | 13 |
| X2 | 13 |
| Y | 13 |

TARGET MODULE FOR PATHS = FUNCT1
FROM NODE 7 THE VARIABLE *GAMMA * CAN AFFECT THE FOLLOWING VARIABLES

| VARIABLE | NODE OF USE |
|---|---|
| FTEMP | 14 |
| FUNCT1 | 20 |
| INDEX1 | 19 |
| MATRIX | 18 |
| TEMP | 13 |
| Y | 7 |

Figure 8: Results of Variable Trace Routine

is shown for variable "GAMMA" beginning at line 7 in function FUNCT1. All variables that may be affected by that variable after execution of line 7 are listed.

These examples have served merely as an illustration of the types of messages that the diagnostic routines can produce. Although the errors at times seem obvious in these three small routines, it can be seen that the output from FACES may prove to be very helpful in analyzing large programs in which dangerous constructs can be camouflaged in pages of FORTRAN code.

# V. CONCLUSION

The complexity of large software systems has brought about the need for some type of automatic program analysis. The out-dated method of searching through pages of code trying to find errors, verify program correctness, document programs, and modify existing programs is no longer effective. Automatic program analysis seems to be a plausible alternative to aid in this process.

Automatic program analysis can help a programmer in several ways, such as program validation, structural and timing analysis, documentation, performance evaluation, and maintenance. An automatic aid can be useful to these purposes by extracting and displaying constructs in a program that might not be obvious to the human observer. Both dynamic and static methods of analysis can be used, depending on the particular analysis function.

The problem of automatic program analysis can be approached in two ways. One way is to develop specific analysis programs that perform a limited service for the programmer. A second approach is to develop a general purpose analysis system to provide a variety of services at the option of the user. The specific analysis programs have the advantage that they can be easily written and can be executed with a small amount of time and memory requirements. If a programmer has a limited need for a particular service, the limited analysis system can be used with little trouble.

The disadvantage of a limited system is that a programmer who needs a wide variety of analysis functions must learn how to use many different systems and use each one separately.

A general purpose analysis system can offer a variety of services to a user in one package. The system is likely to be bulky (in terms of memory and time requirements), but its total size and time requirements will likely be less than the sum of all the specific analysis systems that would be needed to perform all of the services of the general purpose analyzer. The general purpose analyzer is built around a data base which contains a large amount of information about the program being analyzed. The data base need only be collected once, though, and can be referenced for a wide variety of purposes.

Some previous work has been done in the area of automatic program analysis. Most of it, however, has been directed toward program validation, particularly in the area of automatic aid to testing programs. Most of the previous work that has been done has made use of dynamic analysis for a specific function.

A FORTRAN Automatic Code Evaluation System has been developed which involves static analysis of programs through the collection of a large data base. FACES is designed to provide a broad scope of analysis functions using both static and dynamic methods, building on the idea of providing one general purpose package to aid in program analysis.

Automatic program analyzers at present are still in

the early stages of development and experimentation. One of the major problems in developing them lies in defining goals, because there are no set rules or patterns that make a program "correct". In other words, we are not always sure what to analyze, much less how to do it automatically. Because the human environment in which programs are run is always subject to an infinite supply of data, many of the questions which we wish to ask about a program become open-ended. We find that we can only begin to insure a program's correctness under a given, and of course limited, situation. Once a clear set of goals is established, more progress can be made, even though we are still faced with the problem of discovering methods of satisfying the specifications. The author hopes that this thesis has served to organize some of the basic goals with which one might begin to develop an automatic aid to program evaluation, documentation, and maintenance and to discuss some of the methods by which the goals may be reached.

# APPENDIX

## Tables Produced by FORTRAN Front End for Test Routines

LOCAL TABLES GENERATED FOR MODULE 201

## SYMBOL TABLE

THIS TABLE CONTAINS ALL NAMES AND LABELS THAT APPEAR IN EACH MODULE.
NAMES ARE HASH-CODED AND APPEAR WITH FOLLOWING CODES AND POINTERS

TYPE CODES
0--INTEGER
1--FLOATING POINT
2--DOUBLE PRECISION
3--COMPLEX
4--LOGICAL
5--NEUTRAL

CLASS CODES
0--PROGRAM NAME
1--SUBROUTINE NAME
2--STATEMENT FUNCTION NAME
3--ARRAY NAME
4--FUNCTION NAME
5--LABEL
6--VARIABLE
7--COMMON BLOCK NAME

USETAB TOP POINTER--POINTS TO FIRST ENTRY OF NAME IN USAGE TABLE
USETAB BOTTOM POINTER--POINTS TO FIRST ENTRY OF NAME IN USAGE TABLE
STALTH PTR.--POINTS TO LOCATION OF NAME IN STORAGE ALLOCATION TABLE IF NAME IS A COMMON VARIABLE.
OR TO ENTRY IN PARTAB IF SUB. OR FUNCTION NAME
ALTAB PTR.--POINTS TO LOCATION OF NAME IN ARRAY LENGTH TABLE IF NAME IS AN ARRAY.

| INDEX | NAME | TYPE | CLASS | USETAB TOP PTR. | USETAB BOT. PTR. | STALTH PTR. | ALTAB PTR. |
|---|---|---|---|---|---|---|---|
| 20 | VAR1 | 1 | 6 | 13 | 38 | 8 | 0 |
| 21 | VAR2 | 1 | 6 | 14 | 28 | 9 | 0 |
| 47 | ZETA | 1 | 6 | 39 | 39 | 0 | 0 |
| 93 | BLOCK3 | 5 | 7 | 15 | 15 | 0 | 0 |
| 185 | 20 | 5 | 10009 | 20 | 22 | 0 | 0 |
| 262 | K | 0 | 6 | 21 | 24 | 0 | 0 |
| 288 | FUNCT1 | 1 | 4 | 29 | 29 | 2 | 0 |
| 439 | DELTA | 1 | 6 | 9 | 31 | 5 | 0 |
| 492 | TEST1 | 5 | 1 | 2 | 2 | 0 | 0 |
| 506 | GAMMA | 1 | 6 | 8 | 30 | 4 | 0 |
| 575 | IARRAY | 0 | 3 | 12 | 23 | 7 | 4 |
| 770 | BETA | 1 | 6 | 7 | 35 | 3 | 0 |
| 772 | I | 0 | 6 | 43 | 43 | 0 | 0 |
| 928 | MATRIX | 0 | 3 | 16 | 14 | 10 | 5 |
| 1134 | LWORDS | 0 | 3 | 4 | 47 | 0 | 3 |
| 1270 | ALPHA | 1 | 6 | 6 | 46 | 2 | 0 |
| 1349 | KWORDS | 0 | 3 | 3 | 42 | 0 | 2 |
| 1396 | BLOCK1 | 5 | 7 | 5 | 5 | 0 | 0 |
| 1430 | 25 | 5 | 10011 | 19 | 25 | 0 | 0 |
| 1475 | IOTA | 0 | 6 | 10 | 18 | 6 | 0 |
| 1643 | BLOCK2 | 5 | 7 | 11 | 11 | 0 | 0 |
| 1656 | SUB3 | 5 | 1 | 32 | 45 | 6 | 0 |
| 1701 | INTEAM | 0 | 6 | 17 | 44 | 11 | 0 |
| 1729 | 30 | 5 | 10017 | 36 | 44 | 0 | 0 |

## USAGE TABLE

THIS TABLE CONTAINS ONE ENTRY FOR EACH APPEARANCE OF A NAME IN A MODULE. EACH ENTRY CONTAINS THE STATEMENT NO. IN WHICH THE NAME APPEARS IN THE SOURCE LISTING AND A USE CODE INDICATING HOW THE NAME IS USED IN THAT STATEMENT. EACH ENTRY IS LINKED TO THE PREVIOUS AND NEXT ENTRIES OF THE SAME NAME BY BACKWARD AND FORWARD POINTERS. THE FIRST ENTRY OF A NAME CONTAINS A POINTER TO THE NAME IN THE SYMBOL TABLE.

BACK POINTER--POINTS TO PREVIOUS ENTRY OF NAME IN USAGE TABLE OR TO NAME IN SYMBOL TABLE
FORWARD POINTER--POINTS TO NEXT ENTRY OF NAME IN USAGE TABLE. FOR LAST ENTRY, FORWARD POINTER IS 0.
BP-CODE--1 WHEN BACK PTR. POINTS TO SYMBOL TABLE
0 WHEN BACK POINTER POINTS TO USAGE TABLE

| INDEX | STATEMT NO. | USE CODE | BP-CODE | BACK PTR. | FORWARD PTR. |
|---|---|---|---|---|---|
| 1 | 0 | 48 | 0 | 0 | 0 |
| 2 | 1 | 0 | 1 | 492 | 0 |
| 3 | 2 | 13 | 1 | 1389 | 34 |
| 4 | 2 | 13 | 1 | 1134 | 47 |
| 5 | 3 | 0 | 1 | 1394 | 0 |
| 6 | 3 | 11 | 1 | 1270 | 37 |
| 7 | 3 | 11 | 1 | 770 | 26 |
| 8 | 3 | 11 | 1 | 506 | 30 |
| 9 | 3 | 11 | 1 | 439 | 31 |
| 10 | 3 | 11 | 1 | 1475 | 18 |
| 11 | 4 | 0 | 1 | 1643 | 0 |
| 12 | 4 | 11 | 1 | 575 | 23 |
| 13 | 4 | 11 | 1 | 20 | 27 |
| 14 | 4 | 11 | 1 | 21 | 28 |
| 15 | 5 | 0 | 1 | 93 | 0 |
| 16 | 5 | 11 | 1 | 928 | 0 |
| 17 | 5 | 11 | 1 | 1701 | 48 |
| 18 | 6 | 20 | 0 | 10 | 0 |
| 19 | 7 | 10 | 1 | 1430 | 25 |
| 20 | 8 | 25 | 1 | 185 | 22 |
| 21 | 8 | 5 | 1 | 262 | 24 |
| 22 | 9 | 9 | 0 | 20 | 0 |
| 23 | 9 | 1 | 0 | 19 | 0 |
| 24 | 9 | 10575 | 0 | 21 | 0 |
| 25 | 10 | 9 | 0 | 19 | 0 |
| 26 | 10 | 1 | 0 | 7 | 33 |
| 27 | 10 | 2 | 0 | 13 | 38 |
| 28 | 10 | 2 | 0 | 14 | 0 |
| 29 | 10 | 2 | 1 | 288 | 0 |
| 30 | 10 | 18 | 0 | 8 | 0 |
| 31 | 10 | 18 | 0 | 9 | 0 |
| 32 | 11 | 21 | 1 | 1654 | 40 |
| 33 | 11 | 19 | 0 | 26 | 35 |
| 34 | 11 | 19 | 0 | 7 | 42 |
| 35 | 12 | 20 | 0 | 33 | 0 |
| 36 | 13 | 10 | 1 | 1729 | 44 |
| 37 | 14 | 1 | 0 | 6 | 41 |
| 38 | 14 | 2 | 0 | 27 | 0 |
| 39 | 14 | 2 | 1 | 47 | 0 |
| 40 | 15 | 21 | 0 | 32 | 45 |
| 41 | 15 | 19 | 0 | 37 | 46 |
| 42 | 15 | 19 | 0 | 34 | 0 |
| 43 | 15 | 19 | 1 | 772 | 0 |
| 44 | 17 | 9 | 0 | 36 | 0 |
| 45 | 17 | 21 | 0 | 40 | 0 |
| 46 | 17 | 19 | 0 | 41 | 0 |
| 47 | 17 | 19 | 0 | 4 | 0 |
| 48 | 18 | 1 | 0 | 17 | 0 |

ARRAY LENGTH TABLE

THIS TABLE CONTAINS DIMENSIONS FOR ALL ARRAYS THAT APPEAR IN A MODULE. EACH ENTRY IN THIS TABLE IS POINTED TO BY THE ALIAS POINTER OF THE ARRAY NAME IN THE SYMBOL TABLE.

| INDEX | DIM. 1 | DIM. 2 | DIM. 3 |
|---|---|---|---|
| 1 | 5 | 1 | 1 |
| 2 | 10 | 1 | 1 |
| 3 | 12 | 1 | 1 |
| 4 | 20 | 1 | 1 |
| 5 | 10 | 30 | 1 |

TRANSITION PAIRS TABLE

| | |
|---|---|
| 50000 | 1 |
| 6 | 7 |
| 6 | 8 |
| 9 | 8 |
| 7 | 10 |
| 9 | 10 |
| 11 | 12 |
| 12 | 13 |
| 12 | 14 |
| 15 | 16 |
| 13 | 17 |
| 17 | 18 |
| 11 | 10000 |
| 15 | 10000 |
| 17 | 10000 |
| 16 | 20000 |
| 19 | 20000 |
| 20 | 30000 |

NODE TABLE

| STATEMT NO. | STATEMT TYPE | USETAB PTR. | SUCCESS. PTR. | SUCCESS. NO. | PRED. PTR. | PRED. NO. |
|---|---|---|---|---|---|---|
| 1 | 30 | 2 | 2 | 1 | 2 | 1 |
| 2 | 28 | 3 | 3 | 1 | 3 | 1 |
| 3 | 36 | 5 | 4 | 1 | 4 | 1 |
| 4 | 36 | 11 | 5 | 1 | 5 | 1 |
| 5 | 36 | 15 | 6 | 1 | 6 | 1 |
| 6 | 10 | 18 | 7 | 2 | 7 | 1 |
| 7 | 45 | 19 | 9 | 1 | 8 | 1 |
| 8 | 53 | 20 | 10 | 1 | 9 | 2 |
| 9 | 52 | 22 | 11 | 2 | 11 | 1 |
| 10 | 52 | 25 | 13 | 1 | 12 | 2 |
| 11 | 34 | 32 | 14 | 2 | 14 | 1 |
| 12 | 10 | 35 | 16 | 2 | 15 | 1 |
| 13 | 45 | 36 | 18 | 1 | 16 | 1 |
| 14 | 52 | 37 | 19 | 1 | 17 | 1 |
| 15 | 34 | 40 | 20 | 2 | 18 | 1 |
| 16 | 51 | 0 | 22 | 1 | 19 | 1 |
| 17 | 34 | 44 | 23 | 2 | 20 | 1 |
| 18 | 52 | 48 | 25 | 1 | 21 | 1 |
| 19 | 51 | 0 | 26 | 1 | 22 | 1 |
| 20 | 14 | 0 | 27 | 1 | 0 | 0 |

| INDEX | SUCCESSOR TABLE | PREDECESSOR TABLE |
|---|---|---|
| 1 | 27 | 22 |
| 2 | 2 | 50000 |
| 3 | 3 | 1 |
| 4 | 4 | 2 |
| 5 | 5 | 3 |
| 6 | 6 | 4 |
| 7 | 7 | 5 |
| 8 | 8 | 6 |
| 9 | 10 | 6 |
| 10 | 9 | 9 |
| 11 | 8 | 8 |
| 12 | 10 | 7 |
| 13 | 11 | 9 |
| 14 | 10000 | 10 |
| 15 | 12 | 11 |
| 16 | 13 | 12 |
| 17 | 14 | 12 |
| 18 | 17 | 14 |
| 19 | 15 | 15 |
| 20 | 10000 | 13 |
| 21 | 16 | 17 |
| 22 | 20000 | 18 |
| 23 | 10000 | 0 |
| 24 | 18 | 0 |
| 25 | 19 | 0 |
| 26 | 20000 | 0 |
| 27 | 30000 | 0 |

DO LOOP TABLE

| ST. NO. | LABEL PTR. | INDEX PTR. | VP-CODE | INIT. VALUE | VP-CODE | TERM. VALUE | VP-CODE | INCREMENT |
|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 185 | 262 | 0 | 1 | 0 | 20 | 0 | 1 |

STORAGE ALLOCATION TABLE

THIS TABLE CONTAINS LISTS OF ALL COMMON VARIABLES IN ALL MODULES AND THEIR ASSOCIATED WORD LENGTHS. ONE ENTRY IS MADE EACH TIME A COMMON VAR. IS DECLARED IN A COMMON BLOCK.

CNTAB PTR.--POINTS TO NAME OF COMMON BLOCK IN COMMON NAME TABLE
SYMTAB PTR.--POINTS TO VARIABLE NAME IN SYMBOL TABLE
STARTING LOCATION--INDICATES STORAGE STARTING LOCATION RELATIVE TO CURRENT COMMON BLOCK
NO. OF WORDS--TOTAL DIMENSION OF VARIABLE

| INDEX | CNTAB PTR. | SYMTAB PTR. | START LOC. | NO. OF WDS. |
|---|---|---|---|---|
| 1 | 2 | 11 | 0 | 0 |
| 2 | 2 | 1270 | 1 | 1 |
| 3 | 2 | 770 | 2 | 1 |
| 4 | 2 | 506 | 3 | 1 |
| 5 | 2 | 439 | 4 | 1 |
| 6 | 2 | 1475 | 5 | 1 |
| 7 | 3 | 575 | 1 | 20 |
| 8 | 3 | 20 | 21 | 1 |
| 9 | 3 | 21 | 22 | 1 |
| 10 | 4 | 928 | 1 | 300 |
| 11 | 4 | 1701 | 301 | 1 |

LOCAL TABLES GENERATED FOR MODULE 2^?

## SYMBOL TABLE

THIS TABLE CONTAINS ALL NAMES AND LABELS THAT APPEAR IN EACH MODULE.
NAMES ARE HASH-CODED AND APPEAR WITH FOLLOWING CODES AND POINTERS

TYPE CODES
0--INTEGER
1--FLOATING POINT
2--DOUBLE PRECISION
3--COMPLEX
4--LOGICAL
5--NEUTRAL

CLASS CODES
0--PROGRAM NAME
1--SUBROUTINE NAME
2--STATEMENT FUNCTION NAME
3--ARRAY NAME
4--FUNCTION NAME
5--LABEL
6--VARIABLE
7--COMMON BLOCK NAME

USETAB TOP POINTER--POINTS TO FIRST ENTRY OF NAME IN USAGE TABLE
USETAB BOTTOM POINTER--POINTS TO FIRST ENTRY OF NAME IN USAGE TABLE
STALTR PTR.--POINTS TO LOCATION OF NAME IN STORAGE ALLOCATION TABLE IF NAME IS A COMMON VARIABLE.
OR TO ENTRY IN PARTAB IF SUB. OR FUNCTION NAME
ALTAB PTR.--POINTS TO LOCATION OF NAME IN ARRAY LENGTH TABLE IF NAME IS AN ARRAY.

| INDEX | NAME | TYPE | CLASS | USETAB TOP PTR. | USETAB BOT. PTR. | STALTR PTR. | ALTAB PTR. |
|---|---|---|---|---|---|---|---|
| 20 | VAR1 | 1 | 6 | 13 | 13 | 8 | 0 |
| 21 | VAR2 | 1 | 6 | 37 | 37 | 0 | 0 |
| 67 | PARAM1 | 1 | 6 | 3 | 41 | 0 | 0 |
| 93 | BLOCK2 | 5 | 7 | 14 | 14 | 0 | 0 |
| 288 | FUNCT1 | 1 | 4 | 34 | 39 | 23 | 0 |
| 290 | Y | 1 | 6 | 31 | 35 | 0 | 0 |
| 303 | ARRAY | 1 | 3 | 18 | 32 | 0 | 5 |
| 320 | JARRAY | 0 | 3 | 4 | 24 | 0 | 4 |
| 439 | DELTA | 1 | 6 | 9 | 30 | 5 | 0 |
| 50A | GAMMA | 1 | 6 | 8 | 29 | 4 | 0 |
| 545 | X | 1 | 6 | 28 | 40 | 0 | 0 |
| 575 | IARRAY | 0 | 3 | 12 | 12 | 7 | 2 |
| 770 | BETA | 1 | 6 | 7 | 7 | 3 | 0 |
| 772 | I | 0 | 6 | 20 | 27 | 0 | 0 |
| 928 | MATRIX | 0 | 3 | 15 | 21 | 9 | 3 |
| 1199 | Z1 | 1 | 6 | 33 | 42 | 0 | 0 |
| 127C | ALPHA | 1 | 6 | 6 | 6 | 2 | 0 |
| 1394 | BLOCK1 | 5 | 7 | 5 | 5 | 0 | 0 |
| 144B | Z2 | 1 | 6 | 38 | 43 | 0 | 0 |
| 1475 | IOTA | 0 | 6 | 10 | 10 | 6 | 0 |
| 1643 | BLOCK3 | 5 | 7 | 11 | 11 | 0 | 0 |
| 165A | SUB3 | 5 | 1 | 2 | 2 | 19 | 0 |
| 1701 | INDEX4 | 0 | 6 | 16 | 16 | 10 | 0 |
| 1729 | J0 | 5 | 10008 | 19 | 23 | 0 | 0 |

## USAGE TABLE

THIS TABLE CONTAINS ONE ENTRY FOR EACH APPEARANCE OF A NAME IN A MODULE. EACH ENTRY CONTAINS THE STATEMENT NO. IN WHICH THE NAME APPEARS IN THE SOURCE LISTING AND A USE CODE INDICATING HOW THE NAME IS USED IN THAT STATEMENT. EACH ENTRY IS LINKED TO THE PREVIOUS AND NEXT ENTRIES OF THE SAME NAME BY BACKWARD AND FORWARD POINTERS. THE FIRST ENTRY OF A NAME CONTAINS A POINTER TO THE NAME IN THE SYMBOL TABLE.

BACK POINTER--POINTS TO PREVIOUS ENTRY OF NAME IN USAGE TABLE OR TO NAME IN SYMBOL TABLE
FORWARD POINTER--POINTS TO NEXT ENTRY OF NAME IN USAGE TABLE. FOR LAST ENTRY, FORWARD POINTER IS 0.
BP-CODE--1 WHEN BACK PTR. POINTS TO SYMBOL TABLE
0 WHEN BACK POINTER POINTS TO USAGE TABLE

| INDEX | STATEMT NO. | USE CODE | BP-CODE | BACK PTR. | FORWARD PTR. |
|---|---|---|---|---|---|
| 1 | 0 | 43 | 0 | 0 | 0 |
| 2 | 1 | 0 | 1 | 1656 | 0 |
| 3 | 1 | 17 | 1 | 67 | 24 |
| 4 | 1 | 17 | 1 | 320 | 17 |
| 5 | 2 | 0 | 1 | 1394 | 0 |
| 6 | 2 | 11 | 1 | 1270 | 0 |
| 7 | 2 | 11 | 1 | 770 | 0 |
| 8 | 2 | 11 | 1 | 504 | 29 |
| 9 | 2 | 11 | 1 | 439 | 30 |
| 10 | 2 | 11 | 1 | 1475 | 0 |
| 11 | 3 | 0 | 1 | 1649 | 0 |
| 12 | 3 | 11 | 1 | 575 | 0 |
| 13 | 3 | 11 | 1 | 20 | 0 |
| 14 | 4 | 0 | 1 | 93 | 0 |
| 15 | 4 | 11 | 1 | 928 | 21 |
| 16 | 4 | 11 | 1 | 1701 | 0 |
| 17 | 5 | 13 | 0 | 4 | 24 |
| 18 | 5 | 13 | 1 | 303 | 32 |
| 19 | 6 | 25 | 1 | 1729 | 23 |
| 20 | 6 | 5 | 1 | 772 | 22 |
| 21 | 7 | 1 | 0 | 15 | 0 |
| 22 | 7 | 10928 | 0 | 20 | 25 |
| 23 | 8 | 9 | 0 | 19 | 0 |
| 24 | 8 | 1 | 0 | 17 | 0 |
| 25 | 8 | 10320 | 0 | 22 | 27 |
| 26 | 8 | 2 | 0 | 3 | 41 |
| 27 | 8 | 2 | 0 | 25 | 0 |
| 28 | 9 | 1 | 1 | 545 | 35 |
| 29 | 9 | 2 | 0 | 8 | 0 |
| 30 | 9 | 2 | 0 | 9 | 0 |
| 31 | 10 | 1 | 1 | 290 | 34 |
| 32 | 10 | 2 | 0 | 18 | 0 |
| 33 | 11 | 1 | 1 | 1199 | 42 |
| 34 | 11 | 2 | 1 | 288 | 39 |
| 35 | 11 | 18 | 0 | 28 | 40 |
| 36 | 11 | 18 | 0 | 31 | 0 |
| 37 | 11 | 2 | 1 | 21 | 0 |
| 38 | 12 | 1 | 1 | 1448 | 43 |
| 39 | 12 | 2 | 0 | 34 | 0 |
| 40 | 12 | 18 | 0 | 35 | 0 |
| 41 | 13 | 1 | 0 | 24 | 0 |
| 42 | 13 | 2 | 0 | 33 | 0 |
| 43 | 13 | 2 | 0 | 39 | 0 |

ARRAY LENGTH TABLE

THIS TABLE CONTAINS DIMENSIONS FOR ALL ARRAYS THAT APPEAR IN A MODULE. EACH ENTRY IN THIS TABLE IS POINTED TO BY THE ALTAR POINTER OF THE ARRAY NAME IN THE SYMBOL TABLE.

| INDEX | DIM. 1 | DIM. 2 | DIM. 3 |
|---|---|---|---|
| 1 | 5 | 1 | 1 |
| 2 | 20 | 1 | 1 |
| 3 | 10 | 30 | 1 |
| 4 | 10 | 1 | 1 |
| 5 | 2 | 3 | 1 |

TRANSITION PAIRS TABLE

| | |
|---|---|
| 50000 | 1 |
| 8 | 6 |
| 8 | 9 |
| 14 | 20000 |
| 15 | 30000 |

NODE TABLE

| STATEMT NO. | STATEMT TYPE | USETAB PTR. | SUCCESS. PTR. | SUCCESS. NO. | PRED. PTR. | PRED. NO. |
|---|---|---|---|---|---|---|
| 1 | 30 | 2 | 2 | 1 | 2 | 1 |
| 2 | 36 | 5 | 3 | 1 | 3 | 1 |
| 3 | 36 | 11 | 4 | 1 | 4 | 1 |
| 4 | 36 | 14 | 5 | 1 | 5 | 1 |
| 5 | 28 | 17 | 6 | 1 | 6 | 1 |
| 6 | 53 | 19 | 7 | 1 | 7 | 2 |
| 7 | 52 | 21 | 8 | 1 | 9 | 1 |
| 8 | 52 | 23 | 9 | 2 | 10 | 1 |
| 9 | 52 | 28 | 11 | 1 | 11 | 1 |
| 10 | 52 | 31 | 12 | 1 | 12 | 1 |
| 11 | 52 | 33 | 13 | 1 | 13 | 1 |
| 12 | 52 | 38 | 14 | 1 | 14 | 1 |
| 13 | 52 | 41 | 15 | 1 | 15 | 1 |
| 14 | 51 | 0 | 16 | 1 | 16 | 1 |
| 15 | 14 | 0 | 17 | 1 | 0 | 0 |

| INDEX | SUCCESSOR TABLE | PREDECESSOR TABLE |
|---|---|---|
| 1 | 17 | 16 |
| 2 | 2 | 50000 |
| 3 | 3 | 1 |
| 4 | 4 | 2 |
| 5 | 5 | 3 |
| 6 | 6 | 4 |
| 7 | 7 | 8 |
| 8 | 8 | 5 |
| 9 | 6 | 6 |
| 10 | 9 | 7 |
| 11 | 10 | 8 |
| 12 | 11 | 9 |
| 13 | 12 | 10 |
| 14 | 13 | 11 |
| 15 | 14 | 12 |
| 16 | 20000 | 13 |
| 17 | 30000 | 0 |

DO LOOP TABLE

| ST. NO. | LABEL PTR. | INDEX PTR. | VP-CODE | INIT. VALUE | VP-CODE | TERM. VALUE | VP-CODE | INCREMENT |
|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 1729 | 772 | 0 | 1 | 0 | 10 | 0 | 1 |

STORAGE ALLOCATION TABLE

THIS TABLE CONTAINS LISTS OF ALL COMMON VARIABLES IN ALL MODULES AND THEIR ASSOCIATED WORD LENGTHS. ONE ENTRY IS MADE EACH TIME A COMMON VAR. IS DECLARED IN A COMMON BLOCK.

CNTAB PTR.--POINTS TO NAME OF COMMON BLOCK IN COMMON NAME TABLE
SYMTAB PTR.--POINTS TO VARIABLE NAME IN SYMBOL TABLE
STARTING LOCATION--INDICATES STORAGE STARTING LOCATION RELATIVE TO CURRENT COMMON BLOCK
NO. OF WORDS--TOTAL DIMENSION OF VARIABLE

| INDEX | CNTAB PTR. | SYMTAB PTR. | START LOC. | NO. OF WDS. |
|---|---|---|---|---|
| 1 | 0 | 10 | 0 | 0 |
| 2 | 6 | 1270 | 1 | 1 |
| 3 | 6 | 770 | 2 | 1 |
| 4 | 6 | 506 | 3 | 1 |
| 5 | 6 | 439 | 4 | 1 |
| 6 | 6 | 1475 | 5 | 1 |
| 7 | 4 | 575 | 1 | 20 |
| 8 | 4 | 20 | 21 | 1 |
| 9 | 7 | 928 | 1 | 300 |
| 10 | 7 | 1701 | 301 | 1 |

LOCAL TABLES GENERATED FOR MODULE 303

SYMBOL TABLE

THIS TABLE CONTAINS ALL NAMES AND LABELS THAT APPEAR IN EACH MODULE.
NAMES ARE HASH-CODED AND APPEAR WITH FOLLOWING CODES AND POINTERS

| TYPE CODES | CLASS CODES |
|---|---|
| 0--INTEGER | 0--PROGRAM NAME |
| 1--FLOATING POINT | 1--SUBROUTINE NAME |
| 2--DOUBLE PRECISION | 2--STATEMENT FUNCTION NAME |
| 3--COMPLEX | 3--ARRAY NAME |
| 4--LOGICAL | 4--FUNCTION NAME |
| 5--NEUTRAL | 5--LABEL |
| | 6--VARIABLE |
| | 7--COMMON BLOCK NAME |

USETAB TOP POINTER--POINTS TO FIRST ENTRY OF NAME IN USAGE TABLE
USETAB BOTTOM POINTER--POINTS TO FIRST ENTRY OF NAME IN USAGE TABLE
STALTH PTR.--POINTS TO LOCATION OF NAME IN STORAGE ALLOCATION TABLE IF NAME IS A COMMON VARIABLE.
OR TO ENTRY IN PARTAB IF SUB. OR FUNCTION NAME
ALTAB PTR.--POINTS TO LOCATION OF NAME IN ARRAY LENGTH TABLE IF NAME IS AN ARRAY.

| INDEX | NAME | TYPE | CLASS | USETAB TOP PTR. | USETAB BOT. PTR. | STALTH PTR. | ALTAB PTR. |
|---|---|---|---|---|---|---|---|
| 20 | VAR1 | 1 | 6 | 13 | 31 | 8 | 0 |
| 21 | VAR2 | 1 | 6 | 14 | 14 | 9 | 0 |
| 93 | BLOCK3 | 5 | 7 | 15 | 15 | 0 | 0 |
| 155 | I5 | 5 | 10018 | 50 | 53 | 0 | 0 |
| 159 | X2 | 1 | 6 | 4 | 42 | 0 | 0 |
| 262 | F | 0 | 6 | 51 | 55 | 0 | 0 |
| 288 | FUNCT1 | 1 | 4 | 2 | 60 | 31 | 0 |
| 290 | Y | 1 | 6 | 20 | 56 | 0 | 0 |
| 438 | DELTA | 1 | 6 | 9 | 30 | 5 | 0 |
| 456 | I0 | 5 | 10020 | 34 | 59 | 0 | 0 |
| 506 | GAMMA | 1 | 6 | 8 | 23 | 4 | 0 |
| 533 | MATRX | 0 | 3 | 16 | 16 | 11 | 3 |
| 575 | IARRAY | 0 | 3 | 12 | 39 | 7 | 2 |
| 709 | I0 | 5 | 10014 | 36 | 44 | 0 | 0 |
| 770 | BETA | 1 | 6 | 7 | 45 | 3 | 0 |
| 772 | I | 0 | 6 | 37 | 40 | 0 | 0 |
| 870 | EPSILON | 1 | 6 | 10 | 18 | 6 | 0 |
| 928 | MAT:IX | 0 | 2 | 54 | 54 | 35 | 0 |
| 964 | I0 | 5 | 10012 | 25 | 35 | 0 | 0 |
| 1219 | I0 | 5 | 10009 | 19 | 26 | 0 | 0 |
| 1270 | ALPHA | 1 | 6 | 6 | 29 | 2 | 0 |
| 1360 | TEMP | 1 | 6 | 38 | 47 | 0 | 0 |
| 1394 | BLOCK1 | 5 | 7 | 5 | 5 | 0 | 0 |
| 1411 | ITEMP | 1 | 6 | 45 | 61 | 0 | 0 |
| 1643 | BLOCK2 | 5 | 7 | 11 | 11 | 0 | 0 |
| 1701 | INDEX | 0 | 6 | 17 | 57 | 11 | 0 |
| 1709 | X1 | 1 | 6 | 3 | 41 | 0 | 0 |

## USAGE TABLE

THIS TABLE CONTAINS ONE ENTRY FOR EACH APPEARANCE OF A NAME IN A MODULE. EACH ENTRY CONTAINS THE STATEMENT NO. IN WHICH THE NAME APPEARS IN THE SOURCE LISTING AND A USE CODE INDICATING HOW THE NAME IS USED IN THAT STATEMENT. EACH ENTRY IS LINKED TO THE PREVIOUS AND NEXT ENTRIES OF THE SAME NAME BY BACKWARD AND FORWARD POINTERS. THE FIRST ENTRY OF A NAME CONTAINS A POINTER TO THE NAME IN THE SYMBOL TABLE.

BACK POINTER--POINTS TO PREVIOUS ENTRY OF NAME IN USAGE TABLE OR TO NAME IN SYMBOL TABLE
FORWARD POINTER--POINTS TO NEXT ENTRY OF NAME IN USAGE TABLE. FOR LAST ENTRY, FORWARD POINTER IS 0.
BP-CODE--1 WHEN BACK PTR. POINTS TO SYMBOL TABLE
0 WHEN BACK POINTER POINTS TO USAGE TABLE

| INDEX | STATEMT NO. | USE CODE | BP-CODE | BACK PTR. | FORWARD PTR. |
|---|---|---|---|---|---|
| 1 | 0 | 61 | 0 | 0 | 0 |
| 2 | 1 | 0 | 1 | 288 | 60 |
| 3 | 1 | 16 | 1 | 1709 | 32 |
| 4 | 1 | 16 | 1 | 159 | 33 |
| 5 | 2 | 0 | 1 | 1394 | 0 |
| 6 | 2 | 11 | 1 | 1279 | 21 |
| 7 | 2 | 11 | 1 | 770 | 22 |
| 8 | 2 | 11 | 1 | 504 | 23 |
| 9 | 2 | 11 | 1 | 439 | 30 |
| 10 | 2 | 11 | 1 | 870 | 19 |
| 11 | 3 | 0 | 1 | 1643 | 0 |
| 12 | 3 | 11 | 1 | 575 | 39 |
| 13 | 3 | 11 | 1 | 20 | 24 |
| 14 | 3 | 11 | 1 | 21 | 0 |
| 15 | 4 | 0 | 1 | 99 | 0 |
| 16 | 4 | 11 | 1 | 533 | 0 |
| 17 | 4 | 11 | 1 | 1701 | 49 |
| 18 | 5 | 20 | 0 | 10 | 0 |
| 19 | 6 | 10 | 1 | 1219 | 26 |
| 20 | 7 | 1 | 1 | 290 | 27 |
| 21 | 7 | 2 | 0 | 6 | 28 |
| 22 | 7 | 2 | 0 | 7 | 29 |
| 23 | 7 | 2 | 0 | 8 | 0 |
| 24 | 7 | 2 | 0 | 13 | 31 |
| 25 | 8 | 10 | 1 | 964 | 35 |
| 26 | 9 | 9 | 0 | 19 | 0 |
| 27 | 9 | 1 | 0 | 20 | 43 |
| 28 | 9 | 2 | 0 | 21 | 0 |
| 29 | 9 | 2 | 0 | 22 | 44 |
| 30 | 9 | 2 | 0 | 9 | 0 |
| 31 | 9 | 2 | 0 | 24 | 0 |
| 32 | 10 | 20 | 0 | 3 | 41 |
| 33 | 10 | 20 | 0 | 4 | 42 |
| 34 | 11 | 10 | 1 | 456 | 49 |
| 35 | 12 | 9 | 0 | 25 | 0 |
| 36 | 12 | 25 | 1 | 709 | 44 |
| 37 | 12 | 5 | 1 | 772 | 45 |
| 38 | 13 | 1 | 1 | 1360 | 47 |
| 39 | 13 | 2 | 0 | 12 | 0 |
| 40 | 13 | 10575 | 0 | 37 | 0 |
| 41 | 13 | 2 | 0 | 32 | 0 |
| 42 | 13 | 2 | 0 | 33 | 0 |
| 43 | 13 | 2 | 0 | 27 | 54 |
| 44 | 14 | 9 | 0 | 34 | 0 |
| 45 | 14. | 1 | 1 | 1411 | 58 |
| 46 | 14 | 2 | 0 | 29 | 0 |
| 47 | 14 | 2 | 0 | 30 | 0 |
| 48 | 15 | 20 | 0 | 17 | 52 |
| 49 | 16 | 10 | 0 | 34 | 50 |
| 50 | 17 | 25 | 1 | 155 | 53 |
| 51 | 17 | 5 | 1 | 262 | 55 |
| 52 | 17 | 7 | 0 | 48 | 57 |
| 53 | 18 | 9 | 0 | 50 | 0 |
| 54 | 18 | 0 | 1 | 929 | 0 |
| 55 | 18 | 16 | 0 | 51 | 0 |
| 56 | 18 | 16 | 0 | 47 | 0 |
| 57 | 19 | 1 | 0 | 52 | 0 |
| 58 | 19 | 2 | 0 | 45 | 61 |
| 59 | 20 | 9 | 0 | 49 | 0 |
| 60 | 20 | 1 | 0 | 2 | 0 |
| 61 | 20 | 2 | 0 | 58 | 0 |

ARRAY LENGTH TABLE

THIS TABLE CONTAINS DIMENSIONS FOR ALL ARRAYS THAT APPEAR IN A MODULE. EACH ENTRY IN THIS TABLE IS POINTED TO BY THE ALTAR POINTER OF THE ARRAY NAME IN THE SYMBOL TABLE.

| INDEX | DIM. 1 | DIM. 2 | DIM. 3 |
|---|---|---|---|
| 1 | 3 | 1 | 1 |
| 2 | 20 | 1 | 1 |
| 3 | 10 | 30 | 1 |

TRANSITION PAIRS TABLE

| | |
|---|---|
| 50000 | 1 |
| 5 | 6 |
| 5 | 7 |
| 6 | 9 |
| 10 | 11 |
| 8 | 12 |
| 10 | 12 |
| 14 | 12 |
| 14 | 15 |
| 15 | 16 |
| 15 | 17 |
| 18 | 17 |
| 18 | 19 |
| 11 | 20 |
| 16 | 20 |
| 21 | 20000 |
| 22 | 30000 |

NODE TABLE

| STATEMT NO. | STATEMT TYPE | USETAB PTR. | SUCCESS. PTR. | SUCCESS. NO. | PRED. PTR. | PRED. NO. |
|---|---|---|---|---|---|---|
| 1 | 27 | 2 | 2 | 1 | 2 | 1 |
| 2 | 36 | 5 | 3 | 1 | 3 | 1 |
| 3 | 36 | 11 | 4 | 1 | 4 | 1 |
| 4 | 36 | 15 | 5 | 1 | 5 | 1 |
| 5 | 10 | 18 | 6 | 2 | 6 | 1 |
| 6 | 45 | 19 | 8 | 1 | 7 | 1 |
| 7 | 52 | 20 | 9 | 1 | 8 | 1 |
| 8 | 45 | 25 | 10 | 1 | 9 | 1 |
| 9 | 52 | 26 | 11 | 1 | 10 | 1 |
| 10 | 10 | 32 | 12 | 2 | 11 | 1 |
| 11 | 45 | 34 | 14 | 1 | 12 | 1 |
| 12 | 53 | 35 | 15 | 1 | 13 | 3 |
| 13 | 52 | 38 | 16 | 1 | 14 | 1 |
| 14 | 52 | 44 | 17 | 2 | 17 | 1 |
| 15 | 10 | 48 | 19 | 2 | 18 | 1 |
| 16 | 45 | 49 | 21 | 1 | 19 | 1 |
| 17 | 53 | 50 | 22 | 1 | 20 | 2 |
| 18 | 52 | 53 | 23 | 2 | 22 | 1 |
| 19 | 52 | 57 | 25 | 1 | 23 | 1 |
| 20 | 52 | 59 | 26 | 1 | 24 | 3 |
| 21 | 51 | 0 | 27 | 1 | 27 | 1 |
| 22 | 14 | 0 | 28 | 1 | 0 | 0 |

| INDEX | SUCCESSOR TABLE | PREDECESSOR TABLE |
|---|---|---|
| 1 | 28 | 27 |
| 2 | 2 | 50000 |
| 3 | 3 | 1 |
| 4 | 4 | 2 |
| 5 | 5 | 3 |
| 6 | 6 | 4 |
| 7 | 7 | 5 |
| 8 | 9 | 5 |
| 9 | 8 | 7 |
| 10 | 12 | 6 |
| 11 | 10 | 9 |
| 12 | 11 | 10 |
| 13 | 12 | 8 |
| 14 | 20 | 10 |
| 15 | 13 | 14 |
| 16 | 14 | 12 |
| 17 | 12 | 13 |
| 18 | 15 | 14 |
| 19 | 16 | 15 |
| 20 | 17 | 15 |
| 21 | 20 | 18 |
| 22 | 18 | 17 |
| 23 | 17 | 18 |
| 24 | 19 | 11 |
| 25 | 20 | 16 |
| 26 | 21 | 19 |
| 27 | 20000 | 20 |
| 28 | 30000 | 0 |

DO LOOP TABLE

| ST. NO. | LABEL PTR. | INDEX PTR. | VP-CODE | INIT. VALUE | VP-CODE | TERM. VALUE | VP-CODE | INCREMENT |
|---|---|---|---|---|---|---|---|---|
| 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 12 | 709 | 772 | 0 | 1 | 0 | 20 | 0 | 1 |
| 17 | 155 | 262 | 0 | 1 | 1 | 1701 | 0 | 1 |

STORAGE ALLOCATION TABLE

THIS TABLE CONTAINS LISTS OF ALL COMMON VARIABLES IN ALL MODULES AND THEIR ASSOCIATED WORD LENGTHS. ONE ENTRY IS MADE EACH TIME A COMMON VAR. IS DECLARED IN A COMMON BLOCK.

CNTAB PTR.--POINTS TO NAME OF COMMON BLOCK IN COMMON NAME TABLE
SYMTAB PTR.--POINTS TO VARIABLE NAME IN SYMBOL TABLE
STARTING LOCATION--INDICATES STORAGE STARTING LOCATION RELATIVE TO CURRENT COMMON BLOCK
NO. OF WORDS--TOTAL DIMENSION OF VARIABLE

| INDEX | CNTAB PTR. | SYMTAB PTR. | START LOC. | NO. OF WDS. |
|---|---|---|---|---|
| 1 | 0 | 11 | 0 | 0 |
| 2 | 0 | 1270 | 1 | 1 |
| 3 | 0 | 770 | 2 | 1 |
| 4 | 0 | 506 | 3 | 1 |
| 5 | 0 | 439 | 4 | 1 |
| 6 | 0 | 870 | 5 | 1 |
| 7 | 0 | 575 | 1 | 20 |
| 8 | 0 | 20 | 21 | 1 |
| 9 | 0 | 21 | 22 | 1 |
| 10 | 10 | 533 | 1 | 300 |
| 11 | 10 | 1701 | 301 | 1 |

GLOBAL TABLES

DIRECTORY

THIS TABLE LISTS NAMES OF ALL MODULES IN SYSTEM ALONG WITH AN ASSIGNED MODULE NO. AND CODE.

| INDEX | MODULE NAME | FILE | NUMBER |
|---|---|---|---|
| 48 | TEST1 | 1 | 201 |
| 85 | SUB2 | 2 | 202 |
| 95 | FUNCT1 | 3 | 303 |

COMMON NAME TABLE

THIS TABLE CONTAINS NAMES OF ALL COMMON BLOCKS IN THE SYSTEM. ONE ENTRY IS MADE FOR EACH DECLARATION OF A COMMON BLOCK. POINTERS TO THE TOP AND BOTTOM OF THE LIST OF VARIABLES IN THE COMMON BLOCK ARE STORED WITH EACH ENTRY.

STALTR TOP PTR.--POINTER TO FIRST VARIABLE OF COMMON BLOCK IN STORAGE ALLOCATION TABLE
STALTR BOTTOM PTR.--POINTER TO LAST VARIABLE OF COMMON BLOCK IN STORAGE ALLOC. TABLE
MODULE NO.--NO. OF MODULE IN WHICH COMMON BLOCK DECLARATION APPEARS.

| INDEX | BLOCK NAME | STALTR TOP PTR. | STALTR BOT. PTR. | MODULE NO. |
|---|---|---|---|---|
| 1 | | 0 | 0 | 0 |
| 2 | BLOCK1 | 2 | 6 | 201 |
| 3 | BLOCK2 | 7 | 9 | 201 |
| 4 | BLOCK3 | 10 | 11 | 201 |
| 5 | BLOCK1 | 2 | 6 | 202 |
| 6 | BLOCK2 | 7 | 8 | 202 |
| 7 | BLOCK3 | 9 | 10 | 202 |
| 8 | BLOCK1 | 2 | 6 | 303 |
| 9 | BLOCK2 | 7 | 9 | 303 |
| 10 | BLOCK3 | 10 | 11 | 303 |

PARAMETER TABLE

THIS TABLE CONTAINS ENTRIES FOR ALL PARAMETERS IN SYSTEM. THE CODE IN EACH WORD DISTINGUISHES THE TYPE OF POINTER STORED IN THAT WORD. THE MOD. NO. IS THE MODULE IN WHICH THE PARAMETER APPEARS

CODES

0--PTR. TO DUMMY PARAMETER IN SYMTAB
1--PTR TO ACTUAL PARAMETER IN SYMTAB
2--PTR. TO PARAMETER LIST FOR NEXT CALL OF SAME ROUTINE
3--PTR. TO SUB. OR FUNCTION NAME IN SYMTAB
4--THIS WORD MARKS END OF CURRENT PARAMETER LIST
5--THIS IS A CONSTANT PARAMETER. POINTER FIELD CONTAINS CONSTANT TYPE

| INDEX | CODE | MODULE NO. | ARGUMT NO. | POINTER |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 38 |
| 2 | 3 | 201 | 1 | 288 |
| 3 | 1 | 201 | 1 | 506 |
| 4 | 1 | 201 | 2 | 439 |
| 5 | 4 | 0 | 0 | 0 |
| 6 | 3 | 201 | 1 | 1656 |
| 7 | 1 | 201 | 1 | 770 |
| 8 | 1 | 201 | 2 | 1389 |
| 9 | 2 | 0 | 0 | 10 |
| 10 | 3 | 201 | 1 | 1656 |
| 11 | 1 | 201 | 1 | 1270 |
| 12 | 1 | 201 | 2 | 1389 |
| 13 | 1 | 201 | 3 | 772 |
| 14 | 2 | 0 | 0 | 15 |
| 15 | 3 | 201 | 1 | 1656 |
| 16 | 1 | 201 | 1 | 1270 |
| 17 | 1 | 201 | 2 | 1134 |
| 18 | 4 | 0 | 0 | 0 |
| 19 | 3 | 202 | 1 | 1656 |
| 20 | 0 | 202 | 1 | 67 |
| 21 | 0 | 202 | 2 | 320 |
| 22 | 4 | 0 | 0 | 0 |
| 23 | 3 | 202 | 1 | 288 |
| 24 | 1 | 202 | 1 | 545 |
| 25 | 1 | 202 | 2 | 290 |
| 26 | 2 | 0 | 0 | 27 |
| 27 | 3 | 202 | 1 | 288 |
| 28 | 1 | 202 | 1 | 545 |
| 29 | 5 | 202 | 2 | 0 |
| 30 | 4 | 0 | 0 | 0 |
| 31 | 3 | 303 | 1 | 288 |
| 32 | 0 | 303 | 1 | 1709 |
| 33 | 0 | 303 | 2 | 159 |
| 34 | 4 | 0 | 0 | 0 |
| 35 | 3 | 303 | 0 | 928 |
| 36 | 0 | 303 | 0 | 262 |
| 37 | 4 | 0 | 0 | 0 |
| 38 | 0 | 303 | 0 | 290 |
| 39 | 0 | 0 | 0 | 0 |

# REFERENCES

[1] Brown, J. R. and Hoffman, P. H. "Evaluating the Effectiveness of Software Verification--Practical Experience with an Automated Tool." Proceedings of the AFIPS Fall Joint Computer Conference 1972, Vol. 41, Part I, pp. 181-189.

[2] Elspas, Bernard; Levitt, Karl; Waldinger, Richard; and Waksman, Abraham. "An Assessment of Techniques on Proving Program Correctness." Computing Surveys, Vol. 4. No. 2, June, 1972, pp. 97-147.

[3] Ferguson, H. Earl and Berner, Elizabeth. "Debugging Systems at the Source Language Level." Communications of the ACM, Vol. 6. No. 8, August, 1963, p. 430.

[4] Grishman, Ralph. "The Debugging System AIDS." Proceedings of the AFIPS Spring Joint Computer Conference 1970, Vol. 36, pp. 59-64.

[5] Haller, Ann Peek; Lasseter, Gene; Meeker, R. E., Jr.; Turner, J. "Final Report--FORTRAN Automatic Code Evaluation System." Internal report, Information Research Associates, Austin, Texas, 1973.

[6] Ingalls, Daniel H. H. "FETE: A FORTRAN Execution Time Estimator." Report to National Technical Information Service from Computer Science Department at Stanford University, February, 1971.

[7] Krause, K. W.; Smith, R. W.; and Goodwin, M. A. "Optimal Software Test Planning Through Automated Network Analysis." Report of the 1973 IEEE Symposium on Computer Software Reliability, pp. 18-22.

[8] Meeker, R. E., Jr. "A Study of Software Reliability and Evaluation." Master's Thesis. University of Texas, August 1972.

[9] Ramamoorthy, C. V.; Meeker, R. E., Jr.; and Turner,J. "Design and Construction of an Automated Software Evaluation System." Report of the 1973 IEEE Symposium on Computer Software Reliability, pp. 28-37.

[10] Russel, E. C. and Estrin, G. "Measurement Based Automatic Analysis of FORTRAN Programs." Proceedings of the AFIPS Spring Joint Computer Conference, 1969, Vol. 34, pp. 723-732.

[11] Stucki, L. G. "A Prototype Automatic Program Testing Tool." Proceedings of the AFIPS Fall Joint Computer Conference 1972, Vol. 41, Part II, pp. 829-833.

[12] Stucki, L. G. "Automatic Generation of Self-Metric Software." Report of the 1973 IEEE Symposium on Computer Software Reliability, pp. 94-100.

[13] Taylor, Richard C. "Source Language Debugging." Computers and Automation, Vol. 20. No. 2, February, 1971, pp. 19-22.

# VITA

Julia Ann Peek Haller was born in Waco, Texas, on November 3, 1951, the daughter of JoAnn Torrence Peek and James Robbins Peek. She received a diploma from Richardson High School, Richardson, Texas in 1969 and received a Bachelor of Arts Degree with a major in mathmatics from the University of Texas at Austin in 1972. While working toward a Master of Arts degree in Computer Science at the University of Texas, she was employed as a programmer for Information Research Associates. She also held a teaching assistantship at the University of Texas and taught an introductory computer science course.

Permanent address: 1324 Chickasaw
Richardson, Texas

This thesis was typed by Ann Haller.