THE PROVISION OF DEBUGGING FACILITIES FOR HIGH LEVEL LANGUAGES

by

Lindsay J. Groves

A thesis presented in partial fulfilment

of the requirement for the degree of

Master of Science in Computer Science

at

Massey University

December 1974.

ABSTRACT

In this thesis computer programming and programming systems are studied, with a view to reducing the time required to debug programs written in high level programming languages. A general approach to the detection, location and correction of bugs is developed; and the provision of debugging facilities for high level languages is discussed with respect to three major issues :

- What information does the programmer need to help him debug his programs ?
- How should this information be presented to him ?
- How can this information be made available ?

Firstly, the programming process is analysed, and the various types of program errors are identified. Ways in which these errors can be avoided, and treatment of those errors which can be detected by the computer are discussed. Special attention is then paid to those errors known as 'bugs', which cannot normally be detected by the computer.

A variety of debugging tools, for use in both batch-processing and interactive environments, is discussed, with emphasis on the information they provide and how they can be controlled. Techniques for using these tools are developed.

Examination of existing debugging facilities shows that four different methods of implementation are commonly used. These methods are described, and examples given of how they have been used in existing systems. Advantages and disadvantages of each method are discussed, and some improvements suggested for each.

Finally, the application of one of these methods in providing debugging facilities for a particular language is described in detail. The design of the debugging facilities and their implementation are developed in a way which could easily be adapted to other programming languages.

ACKNOWLEDGEMENTS

In presenting this thesis I would like to take this oppurtunity to express my thanks to the following people :

First and foremost to my supervisor, Lloyd Thomas, whose continued guidance, criticism and encouragement at all stages of the work has been most helpful.

To the various members of the Computer Science Department and Computer Unit at Massey University who, through numerous and assorted discussions, assisted in the gathering of information and formulation of the ideas contained in this thesis; especially to Nola Simpson, for her assistance with details regarding the design and implementation of MUSSEL.

To the Applied Mathematics Division, D.S.I.R., for their patience and assistance during the latter stages of the preparation of this thesis; especially to Maria Newman, for her helful comments during the preparation of the draft copy.

Finally, to Eloise Doherty for her fine work at the typewriter.

Massey University
December 1974

Lindsay J. Groves

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 0

INTRODUCTION

A useful and instructive paradigm for computer programming is to consider it as a game, played by a programmer and a computer. In this game the programmer's goal is to make the computer perform a specified task, while the computer's goal is to avoid doing it. In each round the programmer and the computer have one turn each, and the game finishes when the programmer succeeds in his goal. The programmer, in his turn, presents the computer with a schedule of commands, called a program, which is intended to instruct the computer to perform the required task. For its turn, the computer takes the program and, unless it can find something wrong with the program, proceeds to obey the program commands; this is called execution of the program. If the computer succeeds in finding a fault in the program, or if it executes the program without performing the required task, then the programmer is usually said to have a bug in his program, and the game continues. Between rounds the programmer examines his program and, using any information the computer has provided about it, tries to locate and remove the bugs; this is called debugging the program.

The programmer does not usually do very well at this game and games often last for many rounds. This is partly because of the programmer's limited ability to express himself in a consistent logical fashion; and partly because the rules of the game are rather unfair on him. Progressively the rules have been changed to improve the programmer's performance and reduce the number of rounds in each game. Most notably, he may now present his program in a language which allows him to express himself more easily, rather than one which is easy for the computer to understand. Such languages are commonly called high level programming languages; and the computer understands programs written in these languages by using other programs, called translators or compilers, to translate them into its own language.

The rules of the game, however, still allow the computer to be elusive in describing any errors it finds, and to withhold information about an attempted execution of the program (see for example (Barr 71)). Though it has long been held by some that computers should not be allowed to use these tactics (see for example (Halp 65)), little has been done about it. In this thesis, programming and programming systems are examined with a view to improving the programmer's performance.

The kinds of errors which occur are analysed and some suggestions made on how the programmer can improve his play (Chapter 1). Rules are discussed which make the game fairer on the programmer, by dictating what information the computer must divulge, and giving the programmer control over what information he receives. These rules are called debugging tools, and ways in which the programmer can use them, called debugging techniques, are also discussed (Chapter 2). A collection of debugging tools is called a debugging facility or debugging system, and four different methods of implementing debugging facilities (i.e. ways of enforcing these rules) are discussed. A number of programming systems which have used each of these methods are discussed (Chapter 3), and details of the application of one of these methods in providing debugging facilities for a particular language are presented (Chapter 4). The language used is called MUSSEL, for Massey University Structured Student Language (Gord 72), and details of this language are given in the text and in the appendices.

Chapter 1

PROGRAM PATHOLOGY

1.0 Introduction to Debugging

> "Programmers call their errors 'bugs' to preserve their sanity; that number of 'mistakes' would not be psychologically acceptable"
>
> (Hopkins, NATO Conf. 1968)

The set of all errors which can occur in computer programs written in high-level languages is uncountable. In this chapter these errors are discussed, with respect to cause, prevention, detection, diagnosis and correction. Suggestions are made concerning language design, compiler design, programmer education and programming itself. Attention to these points can result in programs containing fewer bugs, and in those bugs being easier to find and correct.

1.1 Classification of Errors

When a program is known to be incorrect it is for one of the following reasons:

- it violates the rules governing formation of programs in the language, i.e. a syntax error is detected.
- it violates the rules governing the use of symbols in the language, i.e. a semantic error is detected.
- it violates the limitations of the hardware or operating system, i.e. a system error is detected.
- it executes but does not produce the correct output, i.e. an output-scrutinization-time error is detected.

A run of a program consists of three stages: compilation, execution and output scrutinization. Particular types of errors, from the four types listed above, are detected at each stage. In the first two stages errors are detected by the computer, and in the last stage they are detected by the programmer.

1.1.1 Compilation

Compilation errors, or compile-time errors, are errors which can be detected in the static text of the program. These include all syntax errors, some semantic errors and some system errors.

The semantic errors detected at compile-time are those involving the attributes of symbols which are known at compile-time. Many of these are connected with declarations: e.g. using an identifier which has not been declared, declaring an identifier more than once, or using a variable in a context which is not compatible with the declared attributes. These will be called static semantic errors.

The system errors detected at compile-time are errors in which the static form of the program violates the system limitations, e.g. static storage required for program and data areas exceeds the amount of core available, or reference is made to a non-existent I/O device. These will be called static system errors.

If the compiler has a separate loading phase, in which the loader combines program modules to build the core load, then most static system errors will normally be detected by the loader. In 'load-and-go' compilers, e.g. WATFOR (Shan 67), DITRAN (Moul 67), CORC (Conw 63) and PL/C (Conw 73), the program is compiled directly into core and there is no separate loading phase. Since loading is not necessarily a distinct phase, no distinction is made between errors detected during loading and other compile-time errors.

1.1.2 Execution

Execution errors, or run-time errors, are errors which can be detected in the dynamic execution of the program. These consist of semantic errors and system errors which are only apparent when the program is executed. These will be called dynamic semantic errors and dynamic system errors respectively.

Dynamic semantic errors are usually concerned with the values of variables or expressions, e.g. referencing a variable which has not been defined; using an array index which is outside the declared bounds; dividing by zero, or making some other arithmetic error, such as causing overflow or underflow. The attributes of identifiers and expressions can mostly be determined at compile-time. The most notable exception is checking the number and types of actual parameters passed in procedure calls. Whether parameters can be checked at compile-time depends on the language. In FORTRAN no checking of parameters is possible since subroutine names are merely treated as external symbols. In ALGOL 60 (Naur 63) actual parameters can be checked except where the procedure name is a formal parameter. In ALGOL 68 (Lind 71) procedure declarations include the number and type of parameters of formal procedures, allowing all parameter checking to be done at compile-time (Lang 73). In Burroughs B6700 ALGOL parameters of formal procedures may optionally be declared. If declared they are checked at compile-time. Requiring that formal parameters be completely specified, including the parameters of formal procedures, not only allows errors in procedure calls to be detected at compile-time, rather than at run-time, but also allows procedure calls to be implemented more efficiently, since there is no need to check the number and types of parameters at run-time.

Dynamic system errors are usually concerned with values outside the representable range (i.e. overflow and underflow), dynamic storage requirements exceeding the available core, or the exceeding of arbitrary limitations imposed by the operating system or user, e.g. size of execution stack, execution time or number of pages of output allowed.

A significant feature of run-time errors is that they are very often data-dependent. A program might work perfectly well on one set of data, and fail with another. In most cases this is due to faulty logic, since the logic does not cater for some unusual combination of data which may arise. In other cases the new data may involve larger values, more dynamic storage demands or more execution time, and thus cause the violation of some system limitation. For example, a program using recursive procedure calls may overflow the processor stack for some data, but not for others, depending on the depth of recursion reached.

1.1.3 Output Scrutinization

The fact that a program is compiled and executed without any errors being automatically detected is, of course, no guarantee that it is correct. Errors remaining at this stage are detected by the programmer on inspection of the output. In some cases the presence of an error will be immediately obvious from the output, in other cases the output may look quite reasonable. When this happens the error can only be discovered on close examination of the results.

As with run-time errors, errors discovered from output scrutinization can be data-dependent, and can often go undetected until a program has been in production use for some time. For this reason program testing is an extremely important stage in the development of a program. In any non-trivial program exhaustive testing is not possible, yet test data must constitute a sufficiently wide sample that all sections of the program are tested adequately. Systematic methods of program testing have been developed, for further discussion the reader is referred to a collection of papers covering various aspects of program testing, edited by W. C. Hetzel (Hetz 73).

1.2 Treatment of Compilation Errors

The treatment of syntactically incorrect programs has been largely overlooked in the implementation of most compilers and completely ignored by most authors writing on the subject.

There is a tacit assumption that the programs being analysed are correct, with no provision being made for adequate diagnosis of incorrect programs, largely because such provision detracts from the elegance and efficiency of the parsing algorithm.

The detection of syntax errors, in most cases, is quite adequate, though diagnosis of them usually leaves a lot to be desired. The compiler should be able to pinpoint the exact symbol at which no valid parse could be continued. This is a failing of most top-down recognisers, since an error may not be apparent until several more symbols have been parsed. It is then difficult to go back and find out how much of the statement was correct.

Having located the error, a meaningful diagnostic message should be given. This means an actual descriptive message, not a cryptic error code which must be looked up in the appropriate manual. The message should also be reasonably specific. Examples of vague and ambiguous error messages can be found in most FORTRAN manuals, e.g. 'Unidentifiable statement', or 'Syntax error in FORMAT statement'. On the other hand there is a danger in making the error message more specific than is warranted. For example, the Burroughs Extended Algol compiler often issues messages such as 'right parenthesis expected'. Seeing this error message leads many programmers to believe that they should insert a right parenthesis at the point indicated. Unfortunately this is usually not true. The lesson here is that diagnostic messages should encourage the programmer to examine the statements he has written and think about what he intended them to be rather than attempting to determine what correction should be made.

Diagnosis of static semantic errors is more difficult than diagnosis of syntax errors. Many such errors are not detected until quite late in the compilation phase (e.g. reference to a non-existent label, where labels are not declared explicitly). By this stage most information about the source program has been lost and it is not possible for the diagnostic to indicate in which statement the error occurred, or the name of the symbol involved.

In order to give good diagnostics it is necessary to retain as much information about the source program as might be required. In particular it should be possible to quote the symbolic name associated with any program entity, and to determine from which source statement any section of object code was compiled. One way of doing this is to insert a dummy instruction at the beginning of the object code for each source statement. Another method is to create a 'statement map' which points to the beginning of the object code for each source statement.

In many cases the type of error described above could be detected much earlier if suitable modifications were made to the syntax of the language. For example, reference to a non-existent label is detected immediately if the language requires that all labels be declared before they are used, as in Burroughs B6700 ALGOL. In a language allowing calls to separately compiled subprograms, such as in FORTRAN, calls to unknown subprograms are detected immediately if such external identifiers are declared at the beginning of the program.

The next problem is: what does the compiler do after it detects an error? In an incremental compiler (Katz 69, Pecc 68, Rish 70, Ryan 66), after the error message has been typed the programmer can retype the line and the compiler can begin to parse it again, without affecting the compilation of the rest of the program at all. In a non-incremental compiler it is necessary to have an error-recovery strategy, built into the parsing algorithm and semantic routines, allowing the parse to recover after an error has been detected. The aims of the error-recovery scheme are firstly, to allow analysis of the program to continue, so that as many errors as possible are detected in each run, and secondly to prevent the production of large numbers of consequential error-messages (i.e. messages describing conditions which are the results of earlier errors) in statements which do not themselves contain any errors. For example, if the declaration of an identifier is incorrect or omitted, subsequent use of the identifier normally leads to a consequential error-message indicating the

use of an undeclared identifier. Omission of a semi-colon (where it is used as a statement delimiter), or a closing bracket (e.g. string quote, parenthesis or end) usually leads to an assortment of consequential error-messages. In this case it is usually quite difficult to determine where the omitted character should have been, and recovery is difficult.

Recovery from a syntax error typically consists of the addition, alteration or deletion of one or more symbols so that a valid construction can be parsed (Grie 71). The simplest approach, which is frequently used, is to search for the beginning of the next statement (indicated by a semi-colon, or end-of-line), and continue scanning from there. Using this approach, if a statement contains more than one error, only one is detected. Also, consequential error-messages may be produced later, as a result of code being ignored when scanning is resumed at the beginning of the next statement (Smit 67).

A more satisfactory scheme, which is used in PL/C (Conw 73), is to search for the next statement keyword, and continue scanning from there. This is greatly facilitated by the fact that, in PL/C, statement keywords (BEGIN, END, CALL, GOTO, etc.) are reserved. This approach is more successful than searching for the beginning of the next statement, since the next keyword is likely to occur before the beginning of the next statement, and because omission of semicolons is one of the most frequent syntax errors in PL/I programs. This approach could not easily be used in a language allowing keywords to be used as identifiers. Special treatment is required for any construction which does not begin with a keyword; prefixes and assignment statements in the case of PL/I.

A syntax-checking method for ALGOL 60 which always continues scanning at the symbol immediately following an error is described in (Liet 64). This syntax-checker uses an extended form of table-directed parser, and does no translating. Translation is performed later, if the syntax is correct.

Recovery from static semantic errors presents rather different problems. These errors typically involve incorrect or omitted declarations, or misuse of identifiers. In most cases an operation on the symbol-table can be used to effect recovery. For example, if a reference to an undeclared variable name is detected, then that name is placed in the symbol-table, along with any attributes that can be deduced from the context of its use, and an indication that the entry was made by an error-recovery routine. Then any subsequent use of the identifier in a compatible context will not result in an error-message, since the identifier will already be in the symbol-table.

Some compilers, after detection of an error, attempt not only to continue the pause, but to correct the error and continue translating. Using automatic error-correction, as this is called, executable object code is produced for every program, and execution is nearly always attempted. The argument supporting automatic error-correction (Free 64, Hedr 70, Iron 63) is that by executing every program, even though compilation errors may have been detected, the programmer received maximum information from every run, and thus takes fewer runs to debug his program. In CORC (Free 64) this philosophy is carried to the point where a statement which cannot be corrected is ignored, and replaced by a statement which prints a message indicating that at that point a statement has been omitted.

While the execution of a 'corrected' program may lead to the discovery of further bugs, it is also destined, by the garbage-in-garbage-out principle, to lead to the production of many incorrect and misleading results. This being the case, the effort required to differentiate between output, including error messages, which is genuinely relevant to the program and that which is the result of unsuccessful error-correction will normally prevent the programmer from obtaining any significant information. In the case of a program using disk or tape files, it may be particularly dangerous to execute a program which has been 'corrected'. It is usual to allow the programmer to set an option specifying the number of errors which will be tolerated. By setting this to zero any such risks can be avoided.

1.3 Treatment of Run-time Errors

The treatment of run-time errors is, in most cases, far worse than treatment of compile-time errors. In many cases errors are simply not detected, with unpredictable consequences (e.g. array bound errors, or modification of loop control variable, in many FORTRAN compilers), and when errors are detected the diagnosis of them is often completely unhelpful. (E.g. the case described by Barron (Barr 71), in which the manual described seven different error conditions which could have led to the error message given, none of which was appropriate).

In order to detect run-time errors the compiler must generate extra code, or use run-time routines, to test for illegal conditions. In addition, run-time routines are used to keep track of the executing program and handle the diagnosis of errors which are detected. This collection of routines, together with the compiler itself, is known as the language subsystem. The language subsystem usually also contains library programs, and handles all input and output.

In its most complete form, the language subsystem at run-time is an interpreter, which has complete control of the executing program. In this case all checking of illegal conditions is done by the subsystem, and no extra code need be generated by the compiler.

Whatever the form of the run-time subsystem, tests should be made for all possible infringements of the semantic rules of the language. The earlier an error is detected, the more accurately it can be diagnosed. In addition to reporting illegal conditions, the subsystem should also issue warnings of suspicious conditions. Many I/O conditions fall into this category, e.g. printing an integer variable with a real specification. PUFFT (Rose 65) issues warnings onmany non-fatal conditions, each of which has a severity code. The programmer should be able to suppress these messages if he wishes.

Most dynamic system errors cannot easily be detected by the language subsystem. Instead they are detected by the operating system, or by hardware. When a dynamic system error is detected the operating system, rather than diagnosing the error itself, should pass control to the language subsystem which can then diagnose the error in terms of the source program and the language used. In DITRAN (Moul 67), a scheme was devised whereby error messages issued by the operating system were intercepted by DITRAN and replaced by meaningful messages. When a system error is detected on the Burroughs B6700, the MCP (Master Control Program) passes control to the language subsystem to diagnose. The diagnostics given indicate the type of error and the sequence number of the line being executed at the time, but not the name or value of the symbol involved. For example, an array bound error can be detected by automatic hardware array bound checking, but the error message given does not indicate either the name of the array, or the subscript value used in the illegal reference.

It is particularly important that meaningful error messages, and not cryptic error codes, be given for run-time errors. The programmer has the difficult task of determining what the logical error is, in the program text, which led to the semantic or system error in the execution of the program, and needs a good description of the error, in terms of his program.

In DITRAN, error messages are stored as macros which are called with the appropriate symbolic information as parameters. This allows the error messages to be separate from the compiler, so that they can be modified easily. DITRAN also keeps statistics of how often each error message is issued. These statistics and the method used to store error messages allow the diagnostics to be continually upgraded. If a particular message is issued very frequently it can be examined, and either replaced by a more detailed message, or subdivided into two or more more specific messages. In some cases the

examination might reveal that a spurious error message is being given, or it might reveal an area of weakness in the training of programmers. When error codes are given, the implementor is bound to the set of error messages published in the manual and no upgrading can be done. This is especially bad in the case of a new language, where the implementor has no prior knowledge of the types of errors which programmers will make.

All diagnostic messages should be given in terms of the source program, and should include the line number of the source statement in which the error was detected, the name of the symbol and any values involved, e.g. if an array bound error is detected the name of the array and the value of the offending subscript should be quoted in the error message. It is desirable that the text of the source line also be given, since this reduces confusion in locating whereabouts the error was detected.

In order to provide this information the language sub-system must be able to determine what source statement corresponds to a given section of object code. This is done by either inserting markers in the object code, or creating a statement map during compilation. If source text is to be quoted with error messages, then the source program must be saved during execution. The subsystem must also be able to determine the name of a location, whether it is a storage location or a labelled program location. In order to do this the symbol-table created by the compiler must be available at run-time.

Once an error has been detected, and adequately diagnosed, the next question is: what should be done about it? In an on-line environment control is merely passed to the programmer at his terminal where he can decide what should be done next. In an off-line environment the sub-system must decide whether to terminate execution or whether to attempt to recover from the error and continue execution. In CORC (Free 64) every attempt is made to continue execution. When an error is detected a message is given which describes the error and the

recovery procedure.
e.g. IN STATEMENT _____, ZERO TO NEGATIVE POWER, ASSUME 1.
A limit can be set on the number of errors which will be tolerated before execution is terminated.

It is often possible for the execution to recover after the detection of an error if some assumption is made about the values involved. For example: a value which underflows could be assumed to be zero; the result of a division by zero could be assumed to be the largest number representable; an out-of-bounds subscript could be given the value of the nearest bound. Whether recovery should be attempted or not depends on the extent to which the computation will be effected. For example, an underflowed value assumed to be zero, or an output error, will often have little or no effect on the computation. Assuming the nearest bound in the case of an array bound error will obviously lead to meaningless values and, most likely, a string of misleading consequential error messages.

Any error recovery action which is to be taken should be included in the semantic rules of the language, so that the programmer knows exactly what will happen in any circumstance. An example of this kind of semantic rule is found in section 4.3.5 of the Revised ALGOL Report (Naur 63) which states:

> "A go to statement is equivalent to a dummy statement if the designational expression is a switch designator whose value is undefined."

The semantics of the language should always specify what will be done i.e. whether a condition is an error, or what effect it will have. Under no circumstance should the result of a condition be undefined. An example of such a poor semantic rule is found in section 4.6.6 of the Revised ALGOL Report:

> "The effect of a go to statement, outside a for statement which refers to a label within the for statement, is undefined".

Rules such as this lead to inconsistencies between different implementations of the same language, and sometimes induce programmers to use "tricks" which reduce the portability and comprehensibility of their programs. This is especially true of the many FORTRAN compilers which allow such illegal practices as: negative increments in DO statements; changing the parameters of DO statements within the loop; changing the values of a constants passed as subroutine parameters.

If recovery is not to be attempted then control is passed to a post-mortem routine which gives whatever debugging information is available. In many cases it is far more useful to the programmer if execution is terminated and a meaningful diagnostic given, along with a Post Mortem dump or execution history (these will be discussed in Chapter 2), than if execution is allowed to continue. This allows him to attend to the initial error, rather than trying to determine which of subsequent error messages are consequential, and which relate to new errors.

A type of continuation of execution that would be useful is to allow the programmer to specify a label where execution can continue, following the issuing of an error message and dump. This would allow later sections, which are not dependent on actual values for their success, to be checked despite the occurence of an error. The printout routine, for example, can often be checked despite the presence of incorrect values.

1.4 The Nature of Bugs

A program which is known to be in error is said to contain a bug. This section deals with the most troublesome form of bug: the logic error. A logic error does not violate the syntax rules, but results in the program not performing the required task. This is independent of whether or not a run-time error is detected, and of whether the error arose from a slip-of-the-pen, carelessness, or misunderstanding of the problem, the language or the implementation.

A logic error causes the computation to enter an erroneous state, either by causing the flow of control within the program to be incorrect, or by causing incorrect values to be computed. These two types of error are known as control errors and computation errors respectively.

Control errors can be quite troublesome as it is usually not clear what has happened. The type of control error which arises is very much dependent on the language constructions used to control flow in the program. Control statements relying on branching to labels (e.g. the go to and the FORTRAN arithmetic IF statement) are particularly dangerous. Errors in such statements are easy to make, and hard to detect, due to the conceptual gap between the static text and the dynamic execution of programs using them (Dijk 68a). Another common habitat of control errors is in the boundary conditions of an iterative or recursive process. Such errors often lead to loops not being traversed the correct number of times, and possibly continuing indefinitely.

Structured programming (Dahl 72, Dora 72, Wirt 71) is particularly useful in preventing and detecting control errors. By writing the program in a way which reflects the structure of the process the programmer is less likely to make control errors, and those which he does make are more easily detected.

Computation errors cannot usually be removed from a program until after the control errors have been removed, since it is difficult to determine which incorrect values are due to computation errors and which are the consequence of control errors. Once the control structure of the program is correct the calculation of values can be checked and corrected fairly easily.

Computation errors result from incorrect manipulation or definition of data, which, in turn, can be caused by common slip-of-the-pen errors, such as + instead of -, or incorrect variable names, or by misconceptions in the formulation of the algorithm (including the use of an unstable numerical process) or the representation of the data.

Once the computation first enters an erroneous state, it is transformed from one erroneous state to another by successive statements, until it terminates, either by normal or abnormal means. The final condition (i.e. whetever output is produced, plus an error message in the case of abnormal termination), is thus only a consequence of the bug itself and does not normally indicate immediately what the logical error is that will eventually have to be corrected. Some information about the bug may be drawn from the final condition, indicating whether it is more likely to be a control error or a computation error, and where it is likely to be found.

The following examples show the kind of error likely to have led to a given final condition. In most cases the conclusion is based on the features of the output produced by the section or module of the program which was being executed immediately prior to termination. The conclusion is generally independent of whether termination was normal or abnormal. These examples are based on observations made by the author while marking student programs at Massey University in 1973. The language used was FORTRAN, but the principles apply to other high-level languages.

1. Output is correct up to point of termination, or correct output plus some other spurious output is produced.

The most likely bug in this case, is an error in the terminating condition of a loop. The production of correct results indicates that both control and computation are correct up to that point. If a run-time error is detected it will quite likely be an error such as overflow, subscript too large, or reference to an uninitialized array element.

2. Output is complete, in that the correct number of values are printed, but some or all of the values are incorrect.

Global control is probably correct for the output to be complete. The error is likely to be a computation error, or an error in local control used for conditional computation.

This often happens when assignments are made to a variable in more than one place in the program.

3. No output, or only headings.

This is almost certainly a control error, probably in the initial conditions of a loop, though it may arise from variables not being initialized. If a run-time error is detected it is likely to be division by zero, undefined variable or subscript too small.

4. Output shows no signs of completeness or correctness.

This is quite likely a combination of both control and computation errors. When this happens the programmer should rethink the specifications of the program and review his algorithm and its implementation before even attempting to do anything to the program.

A further variety of bug is that concerned with I/O. Output errors are usually fairly easy to locate and correct. They may involve the wrong variables being printed, or values not being presented as required. In some cases a modification to the algorithm may be required, such as moving an output statement to a different position in a loop.

Input errors involve incorrect data being read, either because the data itself is incorrect, or because it is not read properly. This kind of bug is often very hard to detect, and is thus potentially very dangerous. It is always good practice to include tests for unexpected data values in any program so that this sort of bug can be trapped quickly.

1.5 Debugging

Debugging is the process of removing bugs from computer programs. This process is usually performed in a disorganised manner with very little assistance being provided by the machines, and often accounts for more of the programmer's time than all other activities combined (Gain 69). Debugging is an important stage in the broad activity of programming, and

should be central to any course in programming. Such a course would result in the programmer being aware of the pit-falls in writing his programs, and able to debug them quickly and easily without first going through the 'school of hard knocks' where most of us learnt to debug programs. The closest approximation to a text for such a course known to the author is the book 'Structured Programming using PL/C: an Abecedarian' (Wein 73).

For the purposes of this discussion the programming process is defined as consisting of the following steps:

Problem formulation: the programmer develops his notions or is given specifications of what the program is supposed to do.

Construction of algorithm: description of the data and the manipulations upon it required to solve the problem, usually in terms of some form of flow diagram.

Implementation of algorithm: the algorithm is expressed in terms of the programming language being used.

Test run of program: the program is run with test data, and output scrutinized.

Removal of bugs: the programmer locates and corrects bugs uncovered in the previous steps.

Errors can be introduced at each of these steps. With an error introduced in any of the first three steps it is usually necessary to reconsider the program at that level, comparing the actual behaviour of the program with intended behaviour. When the error is located, the step in which it occurred and subsequent steps are revised. It should be pointed out here that it is very important that sufficient time be spent in the first two steps getting to understand the program. This reduces the chance of errors occurring in these steps, and makes it easier for the programmer to locate and correct any bugs which occur, because he understands what the program is supposed to do and how it is supposed to do it. It is clear that documentation is also very important in this

process, as it allows the programmer to quickly recall what each section of the program is supposed to do, and how.

Errors in the implementation step are less likely to occur, and easier to locate, if the algorithm is expressed in a form which closely represents the constructions of the language, e.g. the structure diagrams used in writing algorithms for MUSSEL programs (Dora 72).

Correcting an error always creates an opportunity for making another error. One danger is that of not considering all ramifications of a proposed correction, and making a correction which is locally right, but globally wrong. A prime example is the use of procedures with side-effects, since one small modification can lead to many other, unforseen consequences. Several other errors of this type often arise through inconsistent use of identifiers, for example : redefining a variable whose value is required later; multiple declarations; duplicate labels; or misspelling of variable names. This type of error can be avoided if the compiler gives a cross reference table, which is a table of all identifiers used in the program, indicating for each the line number of each appearance in the program and the manner in which it is used. Cross reference tables can also be used in locating the above types of errors. For example, Weinberg (Wein 71) quotes a case where a programmer had two variables called "SYSTSTS" and "SYSSTSTS" and, after the program had been in use for some time discovered a statement where one name had been substituted for the other. While the use of two such similar names is very poor programming practice, this error could have easily been found from the examination of a cross reference table.

The programming process, as described above, contains some implicit loops. Locating a bug might require rerunning of the program to obtain more information about what happened in the previous run, as well as the regression to an earlier step described above. When the program runs satisfactorily with one set of test data it is run with the next set. If a

bug is detected, then after it has been corrected, testing begins again so that the full effect of the correction can be seen.

It can be seen, from the above description, that testing and debugging are parallel processes which continue to interact until the program works satisfactorily for all the test data. Debugging, however, should not be an activity entered into only when there is evidence of program failure. The first stage in debugging a program consists of checking the algorithm and then the program before the program is ever run. Thereafter each run should serve either to locate bugs, or to increase or decrease confidence in various sections of the program.

A wide range of techniques are available for checking an algorithm or program before running the program. The easiest and probably most often used is mental analysis of the process described. The manipulation performed is mentally compared with the programmer's notion of what it should do. The failings of this approach are obvious. It is difficult for the human mind to foresee all possible eventualities. Also it is easy for the human mind to skip the details in a section of logic and assume that it does something which it does not do. This is especially easy when the programmer is checking his own program, as he usually believes that it is correct.

A more systematic approach to checking an algorithm or program, is to perform a "desk-check". This consists of demonstrating, by a process of pencil-and-paper simulation, that the sequence of operations described is correct for a particular set of data. This method is quite laborious, and often impractical for a program of any appreciable length. If the program is written in a structured or modular fashion it should be possible to desk-check each section separately and then desk-check the program, assuming the results of each section. Desk-checking is also subject to the failing of intuitive interpretation of the process described by the algorithm or program, although not to such a degree as is

mental analysis. The major drawback in desk-checking is that it only demonstrates the correctness of the algorithm or program with respect to a particular set of data, or as many sets as the programmer has the patience to perform desk-checks for. Often it is possible to find most bugs from desk-checking, but it is not an adequate basis on which to predict the performance of the program on an arbitrary set of data.

The Utopia of algorithm or program checking is the formal proof of correctness. The usual approach taken is Floyd's method (Floy 67) of making assertions about the state of the computation following each statement or flow-chart box. A proof that each assertion is always true then constitutes the required proof of correctness, provided that the assertions together form a sufficient condition for correctness. The present state of the art makes proving the correctness of an algorithm or program a complicated exercise in formal logic, which is quite beyond the patience and mathematical ability of most competent programmers. Besides this, it is just as easy to write down an incorrect proof as it is to write down an incorrect program.

A more practical approach to proving correctness would be to write a short proof of each section of the algorithm as it is written. This would involve, for example, giving a proof that a loop will in fact terminate. This type of proof is quite informal, and is developed alongside the algorithm. It is not intended to prove conclusively that the algorithm is correct, but rather as a tool to help the programmer think over each section of the algorithm as he writes it, thus reducing the occurrence of bugs. By combining logical sections in a structured fashion he can then prove, in the same manner, that the total effect is correct. If the approach of successive refinement (Wirt 71) or top-down programming (Mill 71) is used, the overall structure is verified first, and the component sections later.

No matter how much care is taken in proving the correctness of an algorithm or program, bugs can still arise due to

various forms of human error e.g. misreading or mispunching program statements. These errors, however, are not as difficult to find and correct as the logic errors which a proof of correctness attempts to avoid. A discussion of the techniques used in proofs of correctness is found in (Elsp 72), and an extensive bibliography on the subject is found in (Lond 70b).

Once a program has been run, and shown to be in error, then the process of specifically looking for bugs begins. Finding the location and nature of a bug requires an act of intelligence on the part of the programmer (Gain 69), usually consisting of some sort of searching process in which some possibilities are ruled out, and others shown to be more likely. The precise search procedure used will depend on a number of factors, all of which affect the bugs which are likely to be found, for example, the language being used, the nature of the problem, and the programmer's personality. Each programmer has his own set of bugs which he is likely to make. If he is aware of what these are he can specifically attempt to avoid them. When a bug is shown to exist he should check over this set of likely bugs before proceeding further.

In abstract terms the process of hunting out a bug can be considered as "examining the space whose elements are the possible mistakes he might have made, whose consequences could be the errors he has noticed, and from these selecting one, hopefully in a reasonable period of time" (Gain 69). This examination consists of analysing the statements of the program and comparing their interpretation, with respect to some data, with prior knowledge of what the program is intended to do and how it is meant to do it. This analysis might be carried out in terms of abstract data or some specific data, preferably the actual data used by the program.

In the course of this analysis the programmer will develop some notion as to the probability that each section of the program is correct. This confidence rating, initially, will be low for newly written or recently modified code, and high for sections of code which have been previously tested,

or for which proofs have been written. As he compares the effect of each section of code with his knowledge of what it is intended to do, he will variously increase and decrease his confidence ratings for various sections. When a particular section is shown not to behave as expected it is given a zero confidence rating, and the components of that section are then analysed.

Successful application of this approach depends on the program being written in a well structured fashion, since the sections then can, initially be large chunks of logic, and the interaction between them can easily be verified. In a poorly structured program the sections which can be dealt with are much smaller, and the control mechanism linking them is far more complex. Thus far more detail must be considered than is necessary with the well structured program.

This model of debugging is based on the model of a program described by Dijkstra (Dijk 72), in which each level of refinement of the program is a simulation of a machine which provides the primitive operations for the execution of the program as defined at the next higher level. Using a tree structured model of the program, the programmer, at any stage, only needs to consider the subtree which comprises the section of program he is analysing, and can trace a path through incorrect nodes until he finds the bug. If the error involves a misconception about the problem or its method of solution it is easy to locate the level of refinement at which this occurred, and only the subtree below the affected node need be modified.

Each section of the program is analysed with respect to three questions:

- What is it supposed to do, in relation to the rest of the program?
- How is it meant to do it?
- Does this code do that?

The first two questions emphasise the importance of documentation. There is no use trying to find a bug in a program without a clear idea of what is supposed to be

happening. The third question is considered with the help of output from the program and whatever debugging tools are available. This usually consists of some sort of trace and possibly some form of dump. The traces generally available are unstructured, and it is necessary to work through every step of the program in order to locate the point at which the computation first entered an erroneous state. This can be done either by working through correct states towards the bug (this is called forwards debugging), or by starting at a point where the computation is already in error and working backwards to the bug (this is called backwards debugging). It is often quicker to take the backwards debugging approach since the program is likely to have spent less time in erroneous states than correct states.

In the structural model of debugging discussed above it would be preferable to have debugging tools which will give the net change in state due to the execution of a compound statement with its inner mechanisms being transparent, just as a normal source level trace in fact gives the net change in state due to a number of machine instructions. This can be thought of as an ultra-high level trace.

The model discussed above is an idealised view of debugging, rather than a synopsis of common practice. Since effective use of debugging facilities is an important consideration this model will be used as a basis for subsequent discussion of debugging tools and techniques. This is done in the hope that such an approach will, with the common adoption of the structured approach to programming, become a standard part of programmer education.

Chapter 2

DEBUGGING TOOLS AND TECHNIQUES

2.0 Introduction

In this chapter various types of debugging tools are discussed, with respect to the information or assistance they provide for the programmer, the techniques employed in using them, and the commands used to invoke them. This discussion is oriented towards the programmer, rather than the implementor, and thus is independent of the method of implementation. The object of the chapter is to develop debugging tools which satisfy the following two requirements: firstly, they should provide useful information about the program, expressed in terms of the source program; secondly, they should be easily invoked using commands which are a natural extension of the source language. If a more complicated command language is used to control the debugging facility, or if the command language differs significantly from the source language in its conventions, then the programmer needs to make a concerted effort to learn how to use the facility, and there is a high likelihood of his making an error in a debugging command. This tends to discourage programmers from using the debugging facility, as Grishman found from the usage of his debugging system AIDS (Gris 71).

No attempt is made to furnish an all encompassing set of debugging tools, since there will always be a need to develop new debugging tools as new programming techniques and new programming languages are developed. Section 2.9 deals with ways in which a debugging facility can provide assistance for programmers who need to build their own debugging tools. Some of the tools and techniques discussed here are relevant to any programming environment, while others are applicable only in an on-line or interactive environment. Unless specifically stated, no assumption is made about the environment.

2.1 Execution Counts

The number of times a source statement has been executed, which is called its execution count, provides extremely useful debugging information, which can be used in a number of ways.

The collection of all the execution counts for a program at the end of execution, along with some estimate of the time spent executing each statement, is called a program profile (Knut 71). The execution counts given in a program profile can help the programmer in locating a number of control errors, such as a statement which is never executed, and gives a quick and easy way of detecting an infinite loop. Program profiles show clearly what parts of the program take the longest to execute, and are thus very useful in source level optimization (Inga 72).

When an error is detected by the machine, the execution count of the statement being executed at the time, given as part of the diagnostic, gives additional information about the error. For example, if an error is detected on the first execution of the statement, it is more likely that that statement is at fault than if it had been executed successfully a number of times before the error was detected. Similarly, if an error is detected on the first or last iteration of an iterative loop, it is likely that the control structure of the loop, or its initial values, are at fault, whereas on any intermediate iteration the computations performed in the loop are more likely to be at fault. Execution counts can, similarly, be given as part of debugging messages. In this case the execution count helps the programmer to relate the static text of the program to the dynamic sequence of statements which are executed.

In MUSSEL every diagnostic or debugging message given during execution includes the line number and the execution count of the statement being executed when the message was given. Such messages have the form:

*** Line number (Execution count) Message**

Examples :

```
*   56(  15) ERROR : SUBSCRIPT TOO SMALL, X(5,-98)
*  127(   3) CASE INDEX IS  21
*  127(   6) ERROR : CASE INDEX TOO LARGE, INDEX IS 28
*   72(  48) IF CONDITION IS FALSE
*  673(   1) ERROR : DIVISION BY ZERO
```

A further use of execution counts is in controlling other debugging tools. Very flexible control of debugging facilities can be obtained if the execution count of a statement is accessible to the program, as an attribute of the statement. For example, in MUSSEL the value of the reserved identifier EXCOUNT is defined to be the execution count of the statement in which it appears. Examples of this use are found later in this chapter.

Execution counts of labelled statements, at termination, are given on request in PL/C (Wein 73) and Burroughs B6700 FORTRAN. These execution counts give some of the information given by a program profile, but not all. In particular, they give no information about the behaviour of control structures which do not rely on the use of labels. A full program profile is given on request in ALGOL W (Satt 72).

2.2 Traces

A trace is a sequential record of the changes of state of a computation. Traces are used in analysing the actual behaviour of a program, either to locate errors or to check the validity of the traced program. In either case the programmer's confidence in the traced sections of code will increase or decrease, depending on how the trace compares with the expected behaviour of the program.

Different traces can be used, each recording particular kinds of changes of state. The programmer selects the traces he wishes to use on the basis of the kind of error he expects

to find. The types of trace offered and the flexibility with which traces can be controlled are a matter for the implementor to decide in relation to usefulness to the programmers who will use them, and the cost of implementation. The following set of traces is adequate for most situations.

Store Trace: a record of the values assigned to specified data items.

Fetch Trace: a record of the values fetched from specified data items.

Flow Trace: a record of the deviations from sequential flow of control through the program.

Parameter Trace: a record of the parameters passed in procedure calls.

Source Trace: a record of the source statements as they are executed.

Trace output should be presented in such a way that:

- trace output is clearly distinguishable from programmed output.
- output from different types of trace are distinguished clearly.
- the line number of the source statement corresponding to each line of trace is given.
- each data item whose value is given is identified.
- each value given is presented in a format appropriate to its usage in the program.

This last requirement is, to some extent, dependent on the data types which are available in the language. For example, in FORTRAN there is no way of knowing when the value of an integer variable is to be interpreted as a character, so programmers need to be familiar with the internal representation of characters if they wish to use standard traces with programs which perform character manipulations. Such a program could only be traced according to the above requirements if the language offered a special data type for characters, such as the type char provided in ALGOL 68 (Lind 71).

If available, the execution count of the statement should be included in each trace message.

Example :

If the statement :

```
A[i,j] := A[i,j]*D
```

whose line number is 12, is executed for the third time with store and fetch traces selected for A and D, then trace output might be as follows :

```
*   12(   3) FETCH  A[7,10] =  52.6
*   12(   3) FETCH  D =  3.0
*   12(   3) STORE  A[7,10] =  157.8
```

Example :

When an ALGOL-type if statement is executed, and flow trace has been selected, a message indicating the value of the condition is printed, eg :

```
*   16(  19) IF CONDITION IS TRUE
```

Example :

If the ALGOL procedure whose heading is :

```
procedure PRANCE(x,y); value x; integer x,y;
```

is called, at line number 9, by the statement :

```
PRANCE(2*a+1,b)
```

and flow and parameter traces have been requested, then the trace output might be as follows :

```
*   9(   1) ENTER PRANCE
*   9(   1) PARAMETER  x =  5, VALUE
*   9(   1) PARAMETER  y =  12, NAME
```

The extent to which the programmer can control a trace facility is very important in determining how useful the trace will be. The programmer should be able to specify for which variables a store or fetch trace is required, or for which procedures a parameter trace is required. He should also be able to turn tracing on and off dynamically during execution. That is, the programmer should be able to

specify what is to be traced, the range over which it is to be traced, and the condition under which tracing is to be given within that range.

To give this kind of flexible control, statements must be available which turn different traces on and off. The following examples demonstrate the TRACE and UNTRACE statements used to control tracing in MUSSEL. The syntax of these statements is found in Appendix A.

Examples :

```
TRACE STORES
IF EXCOUNT .GT. 3 THEN UNTRACE SOURCE
TRACE PARAMETERS PROC1,PROC2,PROC3
IF A .GT. 100 THEN TRACE FLOW
```

A simple but effective trace for initial program checking can be given by tracing all statements the first few times they are executed. That is, all traces are deemed to be active for any statement whose execution count does not exceed a certain value. Such a trace can be used either to locate errors, or to informally verify the logic if no error is observed. This verification is based on the observation that most forms of programmed iteration and recursion are the analogues of mathematical induction, and that it is thus sufficient to analyse the first few steps and one later step in order to verify the process. Such a frequency controlled trace is implemented in ALGOL W (Satt 72, Site 71). The programmer can specify, by means of the debug control card, the number of times each statement is to be traced, with a default value of three.

An entirely different method of controlling traces is used in SNOBOL 4 (Gris 68). A reserved identifier, &TRACE, is set to the maximum number of trace records which are to be produced. Tracing is terminated when &TRACE becomes zero, either as it is decremented due to tracing, or by being explicitly assigned the value zero. Tracing can similarly be prolonged or restarted by assigning a new positive value to &TRACE. The different types of trace, including user-written trace functions, can be enabled and disabled independ-

ently, but all are subject to the value of &TRACE for their activity.

It is normal practice for trace output to be written on the principal output device being used by the program, so that trace output and programmed output are mixed. In some systems it is possible to specify a separate file for trace output, which is listed later. This has the advantage that, in the event of the program working correctly, the output is not mixed up with trace output. A trace file, such as this, should include programmed output as well as trace output, so that trace output can be related to execution. A further advantage of writing trace output on a separate file is that trace output can be listed conditionally upon the behaviour of the program. In an on-line environment the programmer can decide, after termination of execution, whether he wants the trace file to be listed. In a batch environment a control card option might specify that the trace file is to be listed only in the event of abnormal termination. In Burroughs B6700 FORTRAN the programmer can specify a previously written trace file with which trace information is to be compared. A dump is given in the event of a mismatch.

The notion of a trace file which is separate from the program's output file can be extended to the notion of a history file, which is a file into which a history of all the changes of state of the computation are written. Information in the history file can then be used to give various forms of traces. In an interactive environment the programmer can request traces to be extracted from the history file, on the basis of the present execution, rather than the previous execution. Since traces are taken from a static file, rather than from an executing program, the programmer can use a fairly general trace to locate approximately where an erroneous state was entered, and then use a more detailed trace to locate the error precisely. Traces can be given either forwards or backwards and it is always possible to go back and take a closer look at what happened at any point in the execution. The principles involved in the use of a history file are found in (Balz 69), which describes the debugging system EXDAMS, in

which all debugging tools are based on the use of a history file.

Given a debugging facility with a range of different traces, the programmer must use them wisely in order to reap the maximum benefit from each run. On an initial run a frequency controlled trace is most helpful for both verifying seemingly correct sections of program, and for finding the approximate location of any error which is observed. On subsequent debugging runs he will select traces according to the type of error he expects to find, and whereabouts in the program he expects to find it. In general, store and fetch traces are used in locating computation errors, whereas flow traces are used in locating control errors. Parameter traces are particularly useful in locating errors in recursive processes, be they control or computation errors. A source trace does not add any information beyond that provided by a flow trace, but simplifies the task of analysing trace output, since it reduces the need to refer to the program listing as well as the trace output. Usage of source trace depends on the type of output device being used. With a slow device, such as a teletype, the extra typing slows execution significantly and could be considered to be an unnecessary burden. With a display terminal, on the other hand, there is very little overhead in displaying extra information and a source trace could be used quite freely.

2.3 Dumps

A dump is a record of all or part of the state of a computation at a given point. Different usages give rise to the following types of dumps:

Snap Dump: a record of the current values of all currently valid variables at a point during execution, except that arrays may not be given in full.

Partial Dump: a record of the current values of specified variables at a point during execution.

Post Mortem Dump: a record of the final values of all variables used in the program, given after termination of execution. It may include other

information, such as the names of any procedures which have been entered, but not exited.

These forms of dump have developed from the traditional core dump, which is a dump of all core locations used by the program. The core dump is essentially a machine language level debugging tool, and is given in a system oriented format, usually either in octal or hexadecimal code. Although it has been widely used by high level language programmers, the core dump is quite unsuitable for their needs. In order to learn anything about his program from a core dump, a programmer needs to know the core addresses of his variables, which may be available as a compiler option. He then needs to convert the system oriented data into values which are meaningful in terms of his source program. When used for debugging high level language programs, this process requires far more effort and specialised knowledge on the part of the programmer than is either necessary or desirable, and can be eliminated by the provision of more suitable debugging tools.

Dumps to be used with high level languages should present each data item in a format appropriate to its usage in the program, and annotate it with its symbolic name. As with tracing this requirement may be restricted by the data types available in the language. If a variable has not been defined prior to its being dumped this should be indicated explicitly, rather than garbage being printed.

The dumping of arrays and structures requires special consideration, since a complete dump of an array or structure will often lead to the production of large amounts of output, most of which is not of interest. For this reason array or structure names should not be specified in a partial dump, although individual elements might be. Sets of elements to be dumped could be specified using a notation similar to the implied DO-loop in FORTRAN, or the ALGOL 60 for list. In most cases the set specified would be a contiguous range of elements. In MUSSEL an array range can be specified in a dump statement, for example:

```
DUMP  SIZE(1 TO N),SHAPE(I1 TO I2,J1 TO J2)
```

For snap dumps and Post Mortem dumps some convention must be adopted to determine which values are dumped. One possibility is to dump only non-zero values, since these will usually be of most significance. Another possibility is that only a block of values at the beginning of the array or structure be dumped. For example in ALGOL W (Satt 72, Site 71) only the first seven elements and the last element of an array are given in a Post Mortem dump. A control card parameter could be used to allow the programmer to specify how many elements should be dumped from each array or structure. Further provision for special control of dumping, especially in the case of structures, could be given, but only at the expense of simplicity in the debugging control language. Complete generality would be available in an extensible debugging environment, since the programmer would be able to write his own dump routines where the standard ones were not suitable.

The most common form of dump, the Post Mortem dump, is used to indicate the state of the computation at the point where execution was terminated, whether termination was normal or abnormal. The values given in a Post Mortem dump allow the programmer to determine which variables have been affected by an error and which have not. This information then gives a good guide as to whereabouts in the program the logical error will be found. For a Post Mortem dump to be of most use, it should be given as soon as possible after the computation first enters an erroneous state, since further execution might destroy the evidence which is needed to locate the bug. It is thus important that errors which can be detected by the machine be detected as early as possible.

Provision of a Post Mortem dump greatly increases the amount of information obtained from a run. When a run-time error is detected, a Post Mortem dump can often lead directly to the bug, thus saving an extra run to find out what happened, otherwise it will give a guide as to what debugging options should be selected in the next run. If no error is detected, the final values given can serve to verify the logic, allowing the programmer to check the final values of variables not included in the output. A Post Mortem dump following abnormal termination is standard practice in CORC (Conw 63) and PL/C (Conw 73), and is the default option in ALGOL W (Site 71).

This procedure should always be used during the development of a program, unless an on-line system allowing examination of data after termination is being used. The programmer should be able to specify that a Post Mortem dump be given irrespective of the manner of termination, to give assistance in finding bugs which do not lead to abnormal termination.

A snap dump is normally used to check the overall effect of a block of program logic. By comparing the dumped values with an earlier set of values (either the initial values, the input, or values from an earlier dump), the programmer can determine whether the proper transformation has been performed. A snap dump is normally only used in a fairly large program, between logical distinct sections of code. The dump is produced in response to a program command at that point, which may include an optional condition. For example:

```
DUMP IF X .GT. MAXX
```

A partial dump is used for checking the net effect of a smaller logical construction, such as the body of a loop, procedure or compound statement. Only those variables which are altered in the construction being checked are specified to be dumped. Partial dumps can be used to give an ultra-high-level trace as described in Chapter 1. In a system such as EXDAMS (Balz 69), where traces and dumps are given by retrieving data from a history table, the programmer could use partial dumps to obtain a trace of his program in terms of its primitive components (i.e. the first level of refinement in terms of Dijkstra's model of programming). Having determined that a particular primitive harbours a bug, the programmer can then use further partial dumps, or an ordinary trace, to further examine the logic of that primitive (i.e. trace that section at the next level of refinement).

2.4 ASSERT Statements

An ASSERT statement is a statement allowing the programmer to make assertions about the invariant conditions of his program, and have them checked during execution. If the

program is correct then the invariant condition specified in each ASSERT statement must be true each time that ASSERT statement is executed. If any of the invariant conditions specified is found to be false, then the program contains an error (unless the invariant condition was incorrectly stated), and execution is terminated. A diagnostic message is given, indicating failure in an ASSERT statement, and a Port Mortem dump given. In an on-line environment, control would be given to the programmer's terminal for further commands to be typed. The ASSERT statement allows logical errors to be trapped at an early stage, instead of allowing erroneous execution to continue until a run-time error is detected or the program finishes with incorrect output. This allows all errors to be treated iniformly, and gives maximum benefit from the Post Mortem dump facility, as the state represented in the dump is near to the first erroneous state of the computation.

If strong enough conditions are specified for every statement of the source program, then all the conditions will be true if and only if the program is correct (cf - the use of assertions in proving the correctness of programs (Floy 67, Naur 66)). Due to the difficulty inherent in formulating such conditions, and the specialised language required to state them, this equivalence between truth of assertions and program correctness is not likely to be achieved. Rather, by asserting weaker conditions, a one way implication can be established, such that the program contains a bug if any assertion is found to be false, but is not necessarily correct because no assertion is false. Thus, the likelihood of the program containing a bug though all the assertions are true is inversely related to the strength of the asserted condition.

By asserting even fairly weak conditions, a programmer can build tests for many illegal conditions into his program. For example, an ASSERT statement placed at the beginning of the body of an iterative loop could be used to check the number of times the loop is traversed, by asserting that the execution count of the ASSERT statement will never exceed a specified bound. In a strong condition the bound would be

the exact number of iterations to be performed. Detailed analysis might lead to an expression, say, $mn - 2(m+n) = 1$, where m and n are parameters of the loop. The following ASSERT statement would then be used:

```
ASSERT EXCOUNT .LE. M*N-2*(M+N)+1
```

This ASSERT statement would lead to the detection of any error in the control structure of the loop which caused too many iterations to be performed. A programmer who did not know the expression for the exact number of iterations might formulate a weak condition by using some upper bound on the number of iterations which is not necessarily the least upper bound. Making an approximation, the bound obtained might be $\max(m,n)^2$, in which case the following ASSERT statement would be used:

```
ASSERT EXCOUNT .LE. MAX(M,N)**2
```

This ASSERT statement would lead to the detection of any error in the loop control which caused the number of iterations to exceed the specified bound. In particular, an infinite loop would be detected. Errors which lead to a few extra iterations being performed might not be detected by this ASSERT statement, though they would be detected if the strong condition given above was used. Further difficulties arise in formulating the invariant condition if the loop is entered several times, since the execution count of the ASSERT statement is the sum of all the previous iterations.

The ASSERT statement, as implemented in ALGOL W (Satt 72, Site 72), has the form:

assert <logical condition>

and is semantically equivalent to:

if <logical condition> then end execution

where "end execution" denotes a procedure which terminates the execution of an ALGOL W program and initiates a Post Mortem dump. A compiler option can be set indicating that ASSERT statements are to be treated as null statements, allowing them to remain as part of the documentation once the program is working correctly, and to be easily reactivated if required during later modifications.

2.5 Logical Breakpoints and Traps

A trap is a debugging tool which allows execution to be interrupted whenever a specified event occurs. Events which may be trapped include:

- execution of a specified statement
- store to a specified variable
- fetch from a specified variable
- transfer to a specified label
- call to a specified procedure or function

The first of these is a special case, and will be called a logical breakpoint. (This form is often called a breakpoint. The "logical" is added here to distinguish it from the physical breakpoint discussed in Chapter 3, though it will be omitted in this section). Breakpoints are more commonly available than the other forms of traps, and are most often used in interactive programming systems to give control to the programmer's terminal at specified points in the execution. Once given control the programmer can enter commands in the command language, which may allow him to: inspect or modify data; set or delete breakpoints; modify and recompile the program; incrementally modify the program; resume execution at the next statement or any labelled statement; enter statements to be executed immediately; or terminate the session. The programmer can thus make decisions as to what debugging information he requires as execution proceeds.

In some systems a set of statements can be associated with each breakpoint. These statements are executed whenever the breakpoint is reached. For example, the DEBUG facility in IBM System /360 FORTRAN allows groups of statements to be associated with labelled statements in the program in this manner.

The notion of a trap is merely an extension of the notion of a breakpoint. Most forms of debugging tools can be built using traps. For example, the various types of trap correspond to different types of trace. Traces can thus be built using traps, merely by associating a print statement with each type of trap. Traps can be used to provide extensible debugging

facilities, in which programmers can build their own debugging tools, as in PEBUG (Blai 71).

Two examples of debugging tools based on the use of traps are the MONITOR declaration in Burroughs ALGOL, and the ON statement in Burroughs ALGOL and PL/I. When a store is made to a variable specified in a MONITOR declaration, execution is trapped and the name and value of the variable are either written on a specified file, or passed as parameters to a specified procedure. The ON statement allows the programmer to specify what action is to be taken when any of a given range of conditions arises. In PL/I, in addition to the conditions provided, the programmer can declare his own conditions which are raised by the use of SIGNAL statements.

2.6 Statement-by-statement Execution

The major advantage of debugging programs in an interactive environment is that intermediate results can be examined as execution progresses. For full benefit to be obtained, the programmer should be able to step through the execution, and examine the effect of each step.

To the low level language programmer the unit of execution is the machine instruction. Most small machines have a single instruction mode of execution as a hardware feature, which is very useful for debugging programs written in low level languages. Like the core dump, this facility has frequently been used by high level language programmers, in lieu of any more suitable facilities.

To the high level language programmer the unit of execution is the source statement. When the definition of a statement is recursive, as in ALGOL 60 (Naur 63), the execution of a statement is further broken up into steps, such that the end of each step is the beginning of the "next" statement, which in turn may be part of the present statement. For example, in the *for* statement:

```
for i:= 1 step 1 until n do
begin
     if A[i] > m then k := k + 1;
     SUMA := SUMA + A[i];
end
```

one iteration might consist of the following steps:

1 - Assign the next value to i and test against the value of n.
2 - Evaluate and test the if condition.
3 - Increment k.
4 - Add A[i] to SUMA.

If the if condition was false, the third step above would be omitted and the fourth step would become the third. If, in the first step, the loop is found to be exhausted, the second step would begin execution of the statement following the for statement.

Statement-by-statement mode can be entered at any time that the terminal has control, using a command such as:

STEP

When in statement-by-statement mode, depression of a designated button causes the next step to be executed, after which control is again returned to the terminal. The programmer may then examine or modify program variables, using the normal command language, and either continue execution in statement-by-statement mode, resume normal execution mode, or terminate execution.

This type of debugging is especially suited to use with a display terminal. By dumping a selected set of values each time control is passed to the terminal, a display of those variables currently involved in the computation can be maintained, and all calculations can be checked immediately.

2.7 Reversible Execution

One of the greatest limitations in debugging a program is that information which may be required for debugging is only available once. If the information is not recorded when it is available, it is normally necessary to re-execute

the program in order to obtain it. In an interactive environment it would be especially useful for the programmer to be able to backtrack through the execution, either from a point where an error was detected or from any other point at which he has control at his terminal. This would allow the programmer to go back and look at the events leading to an observed state, requesting traces and dumps as necessary. Execution can thus be analysed from the point where an erroneous state is seen to exist, backwards to the point where the logical error first caused the program to enter an erroneous state. This backwards debugging offers the programmer a different approach to debugging, which may suit his needs better than the conventional approach.

The execution of a computer program is essentially a non-reversible process, due to the destruction of information about earlier states by erasure or overwriting of data, and entry into portions of the program addressed by several different transfer instructions (Benn 73). Any scheme allowing reversal of execution must therefore use a history containing information about earlier states.

A simulated reversible execution is offered in EXDAMS (Balz 69), allowing execution to be traced either forwards or backwards from any point. A history file is created as the program is executed and debugging routines merely retrieve information from the history file, format it and present it to the programmer. All debugging takes place after execution has terminated, so there is no interaction with the executing program.

A history table can also be used to restore the execution to an earlier state, allowing the programmer to go back to an earlier point in the execution, change some values and re-execute from there with the altered data. This is done in an extension of PL/C using the RETRACE statement (Zelk 73). This statement specifies that execution be retraced either to a given label, or a certain number of statements, or until a given condition is satisfied. A PL/C statement is then executed and normal execution is resumed. Such a statement can be used in conjunction with an ON statement to reverse

execution and re-execute a section of the program with tracing. In an interactive environment the RETRACE statement could be typed at the console for immediate execution at any point at which the terminal was in control. Used in conjunction with an incremental compiler, the RETRACE statement would allow the programmer to go back to an earlier point in the execution and modify his program before continuing execution.

2.8 Incremental Compilation and Execution

In an interactive environment the programmer's ability to manipulate and modify his program, and thus debug it quickly, is greatly increased by the use of an incremental compiler. An incremental compiler is a language processor which allows compilation, execution and modification of a program to proceed on a statement-by-statement basis. Furthermore these activities can be freely mixed. Features of an incremental compiler include:

- Each statement is analysed as it is typed, allowing syntax errors to be detected immediately, rather than after the whole program has been entered. Each statement is thus ready for execution as soon as it has been typed.
- A statement can either be added to the program or executed immediately, or both. This allows the programmer to do calculations, try out formulae or modify data, independently of the program.
- The program and its data can be modified at any point without disturbing the program environment.
- A program can be executed as it is being typed. An incomplete program can be executed, so long as it does not refer to any data or label which is not known, thus statements or groups of statements can be executed as they are typed.
- Execution can be started or stopped at any point, allowing modular testing of a program, without the programmer

having to build each module into a separate job as is otherwise necessary.

- The programmer can exploit the flexible control of execution to obtain debugging information as required.

Incremental compilers are highly suited to simple languages like BASIC (Keme 67), where statements have a simple structure, allowing them to be compiled separately and the code added to a linked list as described in (Rish 70). The flow of control between statements is then handled interpretively. In structured languages, object code needs to be arranged in more complex data structures which reflect the structure of the program. Adaption of incremental compilation to languages such as ALGOL 60 and PL/I is discussed in (Lock 65, Pecc 68, Ryan 66). For more general discussion of incremental compilers see (Evan 66, Katz 69).

2.9 Extendable Debugging Facilities

In the preceding sections, emphasis has been placed on the provision of useful debugging tools which can be controlled with reasonable flexibility using simple commands. These tools are not intended to cater for every situation, though they would be adequte in most instances. Most cases where these tools are not adequate are so specialised that the programmer can only obtain the flexibility he requires by writing his own debugging routines. This task can be greatly facilitated by the use of an extendable debugging facility. That is, a debugging facility which allows user-written debugging routines to be given a special status, similar to that of system-provided debugging routines. It is this special status that distinguishes between debugging routines written using an extendable debugging facility and the type of debugging statements a programmer adds to his program when no other debugging tools are available.

Since extendability is a tool for use in special circumstances, a programmer should be able to use standard debugging tools where they are suitable, and build his own debugging tools only if it is necessary. Three approaches to the provision of extendable debugging facilities are discussed here.

The first approach to the provision of extendable debugging facilities is to allow the programmer to write his debugging routines as part of the program, using the source language or a subset of it, and to remove them from the program by some simple action.

This type of extendability is provided in IBM System /360 FORTRAN, by allowing groups of statements to be associated with labelled statements in the program. When control is passed to a labelled statement with which a group of statements have been so associated, that group of statements is executed prior to the execution of the labelled statement. All such groups of statements referring to labels in one program or subprogram are grouped together in a debugging packet at the end of that program or subprogram. The statements which may be used in the debugging packet consist of a subset of FORTRAN, plus statements which control the standard debugging facilities. These standard debugging facilities include flow trace (i.e. trace of labelled statements executed), store trace, subroutine trace, and partial dump. The programmer can either use the standard debugging facilities by themselves, or he can use them in conjunction with ordinary FORTRAN statements (with a few restrictions) to build his own debugging tools. When no longer required, the debugging statements are eliminated by merely removing the debugging packet from the program deck. The following example demonstrates the use of the /360 FORTRAN debugging facility. Values of I, A(I,I) and D(I) are dumped every time an assignment is made to a diagonal element of A. Flow of control through the loop is traced for the first fifteen iterations, that is, while the execution count of statement number 10 (i.e. the value of XC10) is less than fifteen. The values of A are dumped just before the loop is exited.

Example :

```
          .
          .
          .
     4 DO 10 I=1,N
       DO 10 J=1,M
          .
          .
     6 A(I,J) = A(I,J)*D(J)
          .
          .
    10 CONTINUE
          .
          .
          .
       DEBUG TRACE
       AT 4
            IF(XC10 .GT. 15) GO TO 100
            TRACE
  100       CONTINUE
            DATA XC10/0/
       AT 6
            IF(I .NE. J) GO TO 101
            AI = A(I,I)
            DI = D(I)
            DISPLAY I,AI,DI
  101       CONTINUE
       AT 10
            XC10 = XC10 + 1
            IF(XC10 .LT. 15) GO TO 102
            TRACE OFF
  102       IF(I .LT. N .OR. J .LT. M) GO TO 103
            DISPLAY A
  103       CONTINUE
       END
```

An alternative method of providing this kind of extendability is to allow any statement to be designated as a pseudo-comment, that is, a statement which may be treated either as an ordinary statement or as a comment (or dummy statement) according to a compiler option. This would allow the programmer to include debugging routines in his program, using standard debug commands and source statements, and to delete them merely by changing a compiler option. For example, consider an ALGOL dialect in which any statement with the label C* is treated as a pseudo-comment. When the compiler option is set so as to omit debugging statements, any statement having the label C* is treated as a dummy statement, and any label preceding this label becomes a label on the dummy statement. The program of the previous example might then be written as :

Example :

```
begin
C* :  XC10 := 0 ;
      .
      .
      .
L4:C*:if XC10 < 15 then TRACE ;
      for i := 1 step 1 until n do
      for j := 1 step 1 until m do
      begin
            .
            .
     L6:C*:if i = j then DISPLAY i,A[i,i],D[i] ;
           A[i,j] := A[i,j]*D[i] ;
            .
            .
     C*:L10:XC10 := XC10 + 1 ;
     C* : if XC10 > 14 then TRACE OFF ;
     C* : if i = n ∧ j = m then DISPLAY A ;
       end ;
       .
       .
       .
end ;
```

The second approach to the provision of extendable debugging facilities is to allow the programmer to write his own debugging procedures and treat them like standard debugging procedures.

A simple example of this type of extendability is found in Whetstone ALGOL (Rand 64). Debugging procedures are given identifiers beginning with TEST. A compiler option allows automatic deletion of all procedure declarations and procedure statements concerning procedure identifiers starting with the letters TEST.

In SNOBOL 4 (Gris 68) the programmer can write his own trace procedures and control them in the same manner as

system-provided traces. Three keywords and three functions are provided to assist programmers in writing trace procedures. The keywords &STNO, &LASTNO, and &RTNTYPE respectively give: the statement number of the statement currently being executed; the statement number of the last statement executed; and the type of return made by the last function to return (success, fail or return from programmer-defined function). The functions ARG (function,n), LOCAL(function,n), and FIELD(data type,n) respectively return: the name of the nth argument of a programmer-defined function; the name of the nth local variable of a defined function; and the name of the nth field of a programmer-defined data type. As described in section 2.2, all traces in SNOBOL 4, including programmer-defined traces, are controlled by the keyword &TRACE.

The third approach to the provision of extendable debugging facilities is the extendable debugging system, as exemplified by EXDAMS (Balz 69) and PEBUG (Blai 71), Although neither of these authors describes the manner in which the extendability feature of his debugging system is used, the general approach taken is as follows. The basis of the system is a set of primitive routines which make certain debugging information available. These primitive routines are then used as building blocks for extending the debugging system. The debugging system provides some debugging routines, and is organised in such a way that new ones can easily be added. A programmer can, thus, use existing debugging tools if they are suitable, or build new ones and add them to the system. Since the system itself is extended, any new debugging tools that are added to the debugging system by one programmer will be available to any other programmers using the same copy of the system. Extendability is thus at the installation level, rather than at the individual programmer level.

Chapter 3

IMPLEMENTATION OF DEBUGGING FACILITIES FOR HIGH LEVEL LANGUAGES

3.0 Introduction

Implementation of debugging facilities for high level languages presents two major problems which must be solved. Firstly, how to give control to the debugging routine at the required time; and secondly, how to give the debugging routine access to the necessary symbolic and structural information about the source program. In this chapter, four basic approaches to the implementation of debugging facilities, involving various solutions to these two problems, are discussed. Examples of the use of each of these approaches are given, and some suggestions are made regarding ways in which each method could be extended to allow better debugging facilities to be provided.

The choice of which approach to use in a particular implementation depends on the aims and limitations of the project. For example, it may be required that as little modification as possible be made to existing software. This would immediately rule out the approaches discussed in sections 3.3 and 3.4. Other criteria might be: easy elimination of debugging overhead; quick turnaround for debugging and testing runs; or whether the facility is to be used in a batch or interactive environment, or both.

For the purposes of this discussion the term compiler is restricted to mean a program which translates a source program into an equivalent machine language program which can be executed directly by hardware, as opposed to an interpreter, which is a program which accepts a source program and executes it itself (Grie 71).

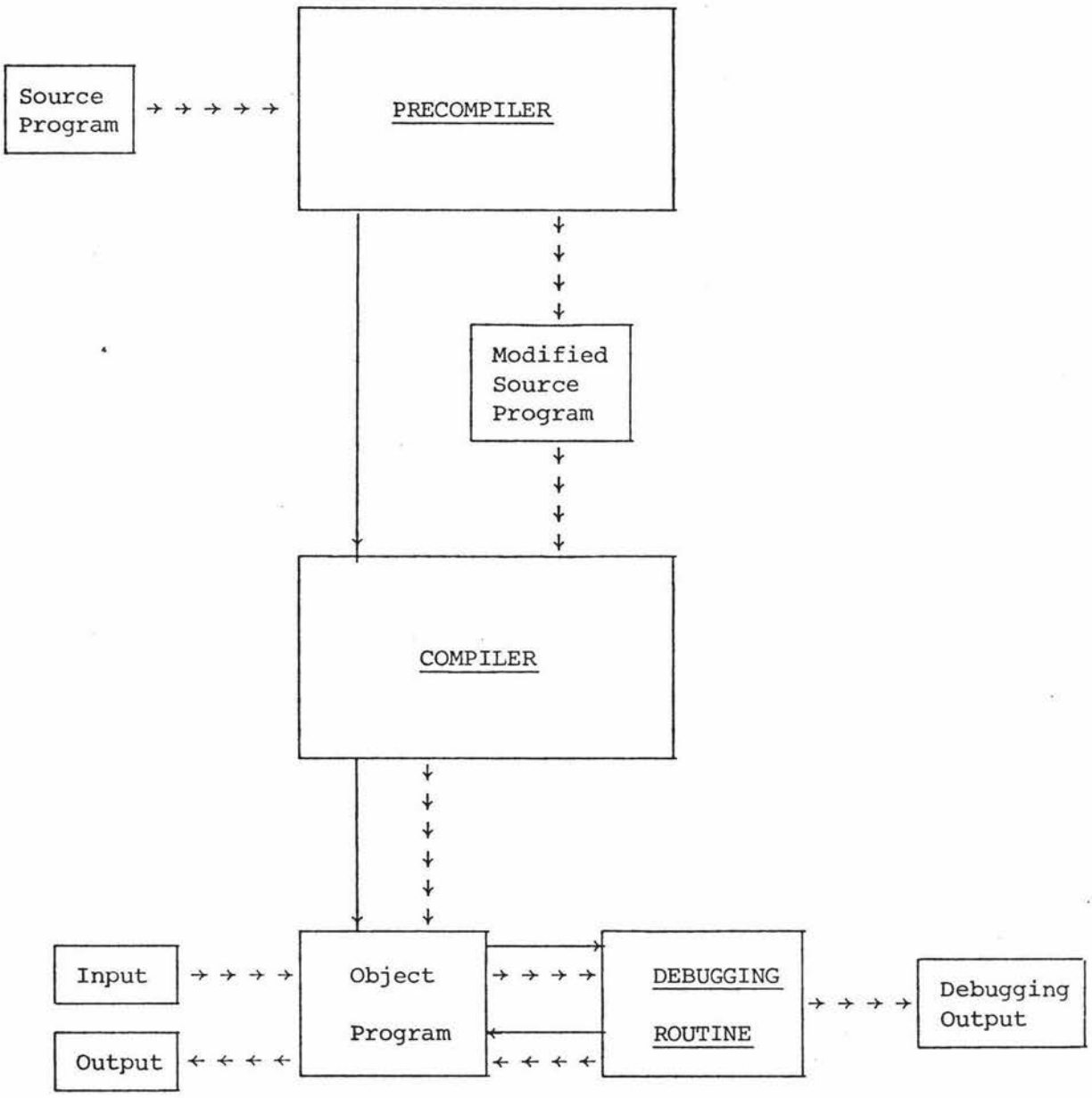
3.1 Implementation of Debugging Facilities using a Compiler

A precompiler is a program which performs some editing function on a source program, prior to its being analysed by a compiler. A debugging system implemented using a precompiler typically consists of a precompiler, and a set of debugging routines which are present during execution (See Fig. 3.1). These routines can either be distinct subroutines or different entries into the same subroutine. The former case involves more work for the system in loading the subroutines into core, but makes it easier to modify or expand the debugging system. The precompiler is used to modify the source program by inserting calls to the debugging routines. Traceable events are located from analysis of the source program. Calls to the debugging routines are inserted where traceable events are found and where dumps are requested.

Names and values are passed to the debugging routines as parameters in the procedure (or subroutine) calls inserted by the precompiler. A name is passed as a character string, and a value is passed by inserting the variable as a value parameter. The precompiler only deals with source symbols and there is never any need to refer to the symbol-table to find addresses or types of variables, as this is handled by the compiler.

Two debugging systems for FORTRAN which have been implemented using precompilers are BUGTRAN (Ferg 63) and DEBUG (Tayl 71). Both offer a fairly standard range of debugging tools: store trace, flow trace, partial dump and the ability to turn tracing on and off dynamically during execution. In addition to this BUGTRAN offers a trace of subroutine entries and exits, an option to print comments from the source program as documentation of output, and a conditional termination of execution. DEBUG also offers a trace of values fetched in assignment statements. The fetch trace is combined with the store trace, so that either stores only, or stores and fetches, can be traced.

A common design goal of these two systems was ease of eliminating debugging commands. Two quite different solutions to this probelm were used.



Note : Solid arrows indicate flow of control, while broken arrows indicate flow of information.

Fig 3.1 Implementation of Debugging Facilities using a Precompiler

In BUGTRAN the debug commands are grouped together in a debug packet at the beginning of the source deck. A BUGTRAN command refers to the relevant section of source code via the name of the subprogram in which it occurs, and statement numbers within the program. To eliminate debugging it is merely necessary to remove the debugging packet. Thus BUGTRAN commands can be added or removed without the source program being altered, except for the placing of statement numbers to be referenced by BUGTRAN commands. Selectivity is gained by including a conditional element in the BUGTRAN command, which references variables in the program, and renders the BUGTRAN command active or inactive according to the value of an arithmetic expression. The BUGTRAN command is inactive if the expression has the value zero.

Examples :

```
1 IN MAIN (FROM 10 TO 63), IF(((K-1)/10)*10),CHECK
  VARIABLES,K,A,B,DONKEY,ZAMBAM
```

Assignments to K, A, B, DONKEY and ZAMBAM are traced between statements 10 and 63 in the subprogram MAIN, while the value of K is between 1 and 10 inclusive.

```
3 IN MYSUB, CHECK FLOW
```

Flow of control is traced for the whole of the subprogram MYSUB.

```
8 IN YORSUB (83), IF(FLAG-1), TERMINATE
```

Execution is terminated if FLAG has the value one when statement number 83 in the subprogram YORSUB is reached.

The form of expression allowed in the condition part is somewhat restrictive. Inequality rather than equality is a more natural condition for controlling debugging. In this version of BUGTRAN it would often be necessary to create new variables in the program to control debugging, such as FLAG in the last example above. Allowing logical expressions, as in FORTRAN IV, to be used would allow more flexible control of the debugging facilities.

In DEBUG the debug commands are placed in the source program at the point required, and have the status of pseudo-comments. In this case a pseudo-comment is a card beginning

with C$, which is an ordinary comment if encountered by a FORTRAN compiler, but which is taken as a DEBUG command by the DEBUG compiler. Thus debugging is automatically eliminated by compiling the program without the precompiler, and the DEBUG commands can remain in the program as part of the documentation, and can be easily reinstated if needed again during later modification of the program.

Examples:

```
C$ DEBUG(NEW) HOW,WHY,WHEN,WHERE
```

Subsequent assignments to the listed variables are traced.

```
C$ DEBUG(ALL) ACE,KING,QUEEN,JACK
```

Subsequent assignments to and fetches from the listed variables are traced.

```
C$ DEBUG(OFF)
```

Tracing of stores and fetches is suspended.

```
C$ TRACE ON
```

Turns on flow trace.

```
C$ DEBUG A,AMIN,AMAX,ANEXT
```

Gives a partial dump consisting of the listed variables.

Selectivity is obtained by using DEBUG commands in conjunction with FORTRAN IF statements, for example the following statements would cause a trace of all assignments to be turned on when K has the value one , and to turn tracing off when the value of K becomes greater than twenty.

Example :

```
      IF(K - 1)2,1,2
    1 CONTINUE
C$ DEBUG(NEW)
      GO TO 4
    2 IF(K - 20)4,4,3
    3 CONTINUE
C$ DEBUG(OFF)
    4 CONTINUE
```

This method of gaining selectivity has two major disadvantages. Firstly, the statements introduced to control the debugging facilities cannot be eliminated as easily as the DEBUG commands can be, and secondly, lack of provision for labels on DEBUG commands necessitates the use of CONTINUE statements to jump to or around DEBUG commands. The DEBUG command language could be greatly improved by allowing any FORTRAN statement to be designated as a pseudo-comment, and allowing existing DEBUG commands to be labelled. The above example could then be written as:

Example:

```
C$     IF(K - 1)2,1,2
C$   1 DEBUG(NEW)
C$   2 IF(K - 20)4,4,3
C$   3 DEBUG(OFF)
C$   4 CONTINUE
```

As well as allowing all statements concerned with debugging to be eliminated in the same manner, this modification also clearly distinguishes such statements from the rest of the program on the program listing.

The debugging system EXDAMS (Balz 69) also uses a precompiler. In this case the precompiler inserts procedure calls to pass information to debugging routines which create a history file. This is used after execution has terminated to provide information for traces and dumps. The precompiler also builds a model of the program's structure, which is used in interpreting the information in the history file. Since the history file always encompasses the whole program there is no need for special commands to direct the operation of the precompiler, as there was in the previous systems.

A major disadvantage in using a precompiler to implement debugging facilities is that there is no simple way of gaining control when a run-time error is detected. In order to be able to give a Post Mortem dump following a fatal error, it would be necessary to alter the operating system so that control was passed to the debugging system when an error was detected. Such an alteration is often not desirable, since the main

justification for using a precompiler is that it avoids the need to modify existing software. The conditional termination offered by BUGTRAN allows errors to be trapped in the same way as ASSERT statements, but could not normally be relied upon to trap all errors.

In BUGTRAN and DEBUG it is often necessary for the precompiler to create new statements as well as new variables in order to pass the necessary information to the debugging routines. For example, in order to give flow trace when the following statement is executed:

```
68 IF(B**2 - 4.0*A*C)1,2,3
```

the statement would be replaced with three statements, suchas:

```
68 QQ = B**2 - 4.0*A*C
   CALL FLOW(1,QQ)
   IF(QQ)1,2,3)
```

where FLOW is the flow trace routine, and its first parameter indicates the type of control statement executed.

In many cases it is neater to use function calls to pass information, rather than subroutine calls. This is done in EXDAMS to record the value of IF-condition, but could be put to more general use. For example ifstore and fetch trace are required for the statement:

```
D := A[I]*R + 1
```

function calls might be inserted as follows

```
D := store('D', fetch('A[I]',A[sub('I',I)])*fetch('R',R)+ 1)
```

where "store", "fetch" and "sub" are functions, each of which returns the value of its second parameter, and writes a trace record as a side-effect.

The use of a precompiler offers a good technique for implementing debugging facilities where it is not practical to modify existing software. The technique is quite flexible and is inherently source-level orientated. The major objection to the use of a precompiler is the resulting increase in compilation time. Since the precompiler must perform a nearly complete syntactic analysis of the program, as well as introducing extra source for the compiler to analyse and translate, the total time for the precompile and compile phase is likely to be at least 20 to 40 percent longer than an ordinary compilation. The actual increase would depend

on how comprehensive the debugging options were, and whether the precompiler was written in a high level or low level language. Execution time is also increased significantly, due to the frequent execution of subroutine or procedure calls. The increased execution time is quite acceptable in the interests of debugging, but the increased compilation time is not consistent with the aim of providing quick turnaround for debugging runs.

Probably the greatest value of a precompiler is as an experimental tool. New debugging tools or other language features can be implemented using a precompiler and user reaction measured before the new tools are added to the compiler.

3.2 Implementation of Debugging Facilities using Physical Breakpoints

A physical breakpoint is an interruption in the execution of a program which allows the execution of an independently specified routine. The use of physical breakpoints allows branches to debugging routines to be inserted and deleted during execution (See Fig. 3.2). In the remainder of this section physical breakpoints will be referred to as breakpoints.

A breakpoint is effected using an instruction which saves its own storage address (i.e. the instruction counter) in a specified location and branches to a breakpoint handling routine. This branch may be made either directly, in which case the breakpoint looks exactly like a parameterless subroutine call, or indirectly via a fixed core location, using a special breakpoint instruction, such as UUO on the PDP-10, TRAP on the PDP-11, or SVC on the IBM System/360. A breakpoint is different from a parameterless subroutine call, however, in that the branch instruction, rather than being generated during compilation, is inserted into the object program by the debugging routine, displacing the machine instruction which previously occupied that location. Instructions so displaced are saved in a table, so that they may be executed when necessary, and restored when the breakpoint is deleted. In a batch environment, breakpoints can be set by the loader, as is done for FORTRAN programs using the

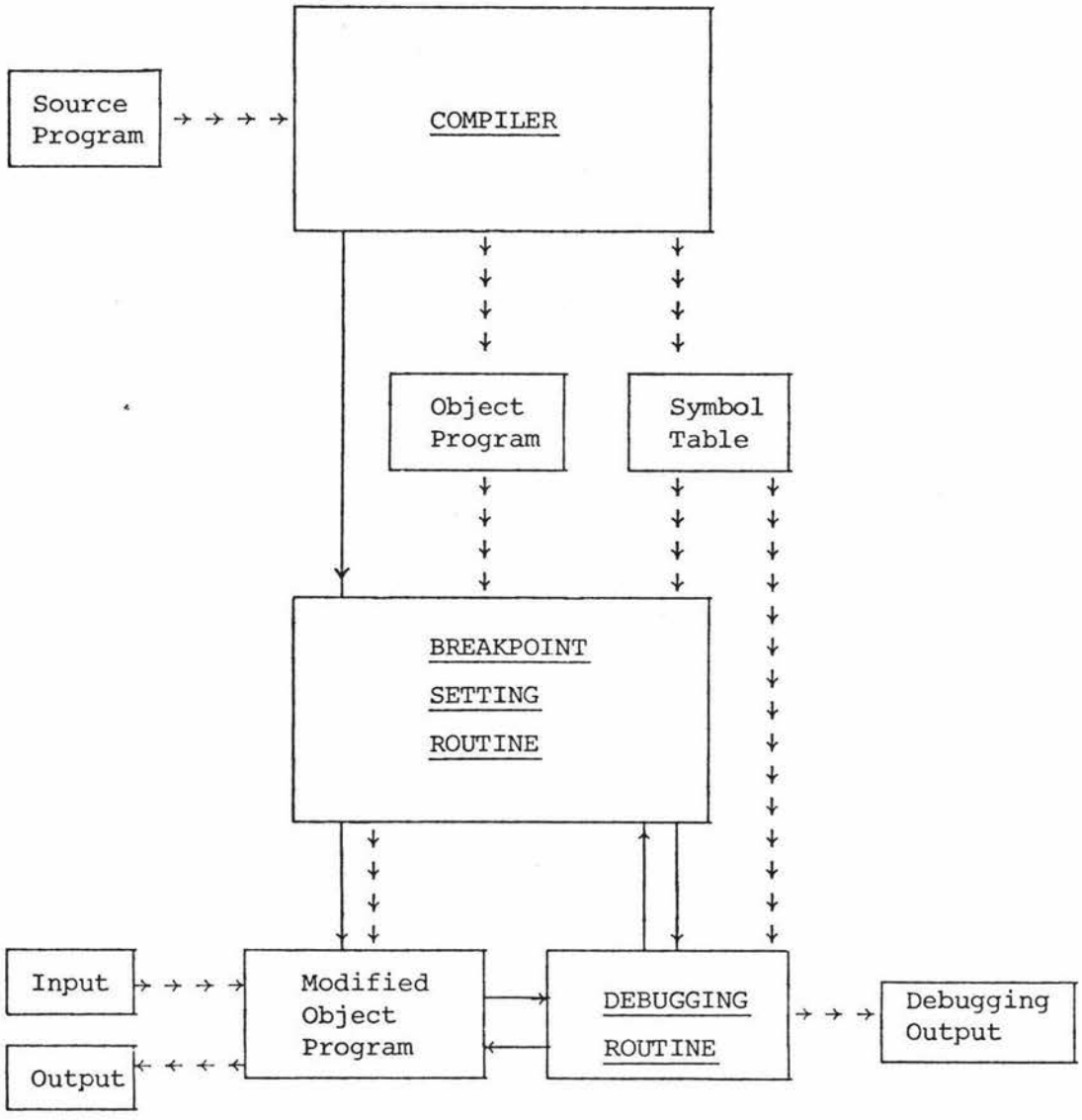


Fig 3.2 Implementation of Debugging Facilities using Physical Breakpoints

IBM 7090 IBSYS Operating System. In an interactive environment breakpoints are normally set and deleted by the debugging control program, or the time-sharing operating system, in response to commands entered at the programmer's terminal.

Each breakpoint is uniquely identified by a field in the branch instruction, enabling the breakpoint handling routine to determine which breakpoint was executed. This allows an action to be associated with each breakpoint, which is initiated by the breakpoint handling routine whenever the breakpoint is executed. This action may be a call on a standard debugging routine, a request for a command to be typed at the terminal, or a previously defined set of commands to be executed. After this action has been performed, providing it does not alter flow of control, the displaced instruction is executed. Provided that this instruction is not a branch or halt instruction, control is then passed to the instruction following the breakpoint.

Two different methods can be used to execute the displaced instruction. The first method is to execute the displaced instruction remotely, either by branching into the table of displaced instructions and branching out again, or by using an instruction, such as EX on the IBM System/360 EXEC on the IBM 7090, or OBEY on the ICL 1900, which causes an instruction at a specified address to be executed, without branching to that address. Problems can arise with this method if the displaced instruction is subject to address modification or if it is an instruction whose effect is dependent on its storage address (e.g. the short branch instruction on the IBM 1130, which address field is a displacement rather than a core address). Most compilers do not use these types of instruction in the object code they produce, so few problems should be encountered, so long as the machine language program being run is the output of a compiler.

The second method is to execute the displaced instruction interpretively. This method overcomes any problems which may arise using remote execution, but also requires

that a simulator for the machine language be written. In this case the simulator can also be used to execute the whole program (See Section 3.4), allowing the programmer to choose between normal execution with breakpoints and simulated execution, as in Blair's debugging system, PEBUG (Blai 71).

The need to execute displaced instructions can be avoided entirely at the cost of fairly small modification to the compiler. In a debugging system for a high level language a breakpoint is always placed at the beginning of the object code for the source statement at which the interruption is required. If the compiler generates a dummy instruction at the beginning of the object code for each source statement, then this instruction can be displaced to permit a breakpoint to be set, and does not have to be executed. This solution is not generally used, however, since most people who have implemented debugging systems using breakpoints have specifically set out to avoid modifying existing software.

The breakpoint handling routine deals only with a machine language program, and is to this extent independent of the language in which the program was written and the compiler which was used. In order that commands given as input to the debugging system and information printed by it can be expressed in terms of the source program, some way of relating source symbols and statements to machine addresses is needed. Most debugging systems using breakpoints rely on the compiler being modified so as to make the compilation symbol-table available to the debugging system (Ashb 73, Blai 71, VerS 64, Wilk 61). The aim of avoiding modification of existing software, thus, is not fully realized, although this is a minor modification to make. In his debugging system, AIDS (Gris 70, Gris 71), Grishman avoided making even this change by using a listing reader, which reads the compiler listing, analyses it, and constructs a symbol-table. Presumably, the listing reader (of which no details are given), mimics the compiler's storage allocation and code generation functions to determine the locations used by program statements

and data. To write such a program would require detailed knowledge of the compiler, and to execute it would take nearly as long as a compilation. The cost incurred seems to be somewhat greater than that of the simple modification to the compiler required to make its symbol-table available.

Given a symbol-table, the debugging system can accept commands and output messages in terms of the source program, although reference to source statements is restricted to labelled statements. To allow reference to unlabelled statements the debugging system needs to be able to locate the beginning of the object code for each source statement. In order to do this the compiler must be modified to either insert a dummy instruction at the beginning of the object code for each statement, or create a storage map with a pointer to the beginning of the object code for each statement.

In most debugging systems which have been based on the use of breakpoints, the major facility provided is that of setting and deleting breakpoints. The programmer can then specify actions to be taken at each breakpoint in order to obtain a store trace for a particular variable, the programmer must set a breakpoint immediately after each statement which assigns a value to that variable, and specify that, when any of these breaks is reached, the name and value of the variable should be printed. Similarly, a flow trace is obtained by setting a breakpoint at each statement which can be branched to, either explicitly or implicitly, and specifying that some identification of that statement be printed whenever the breakpoint is executed. Such a facility allows very flexible debugging, but requires far more effort on the part of the programmer than is either necessary or desirable.

A debugging system implemented using breakpoints should use the breakpoint facility to provide a range of debugging tools, rather than offering only the breakpoint facility; in particular, traces should be provided automatically. The easiest way of doing this is to allow the programmer to request a variety of traps, rather than only logical breakpoints. For example, in MANTIS (Ashb 73) the programmer can set traps on: execution of a labelled statement; stores

to a specified variable; calls to or returns from a specified subroutine. The debugging system uses a table of occurences of stores and subroutine calls and returns to place breakpoints at the necessary locations. This table is created by the debugging system by examining the object program, but could be supplied by the compiler. Unlike the breakpoints set explicitly by the programmer, these breakpoints need not necessarily be placed at the beginning of the object code for the corresponding source statement. A group of MANTIS commands is associated with each trap, rather than with each breakpoint. The different forms of traps can be used to give dumps, store traces, and subroutine traces, respectively.

MANTIS also offers a flow trace, which may be printed out as execution proceeds (using the command TRACE OUTPUT), or recorded in a trace table (using the command TRACE) which may be listed at any time (using the command HISTORY). Deviations from sequential flow of control are trapped by placing breakpoints at program branches, which are located by code inspection. MANTIS has to distinguish between program branches and branches used to control DO statements; this task becomes considerably more difficult in a language with more complicated control structures, such as ALGOL 60 or PL/I. A table of program branches could be provided by the compiler, at the cost of a minor modification.

Some other debugging tools can be implemented in a similar manner, provided the necessary information is supplied by the compiler. Statement-by-statement execution and execution counts can be provided, by placing a breakpoint at the beginning of code for each source statement, provided the compiler produces a statement map or places a dummy instruction at the beginning of the object code for each source statement. A history table can be kept, provided the debugging system can distinguish between program variables and temporary locations assigned by the compiler, and between program branches and other branches inserted by the compiler (e.g. loop control, jumping over data areas).

Implementation of a Post Mortem dump creates another problem, since the debugging system needs to gain control

when execution is aborted in order to give the dump. This can only be done by modifying the operating system so that control is passed to the debugging system whenever a job being run with the debugging system is terminated. This problem does not arise if the debugging system is embedded in the operating system, as suggested by Gaines (Gain 69). One debugging system, however, cannot be expected to cater for every high level language which might be used under the supervision of the operating system, and those debugging facilities which have been implemented in this way are invariably more suitable for use with low level languages.

Further problems arise in the handling of breakpoints when a large number of breakpoints are being set, as would be the case if such tools as execution counts, traces, history tables or statement-by-statement execution were offered. More than one command might be required to set breakpoints at the same location. For example, a dump command sets a breakpoint at the location corresponding to the beginning of code for the source statement at which the dump is required. A breakpoint may, however, already have been set at this location to give, say, the execution count of the statement. In order to avoid displacing breakpoints to set other breakpoints, the debugging system would need to check to see if the location already contained a breakpoint. Instead of a single action (which may be a group of commands), each breakpoint would then have a list of actions associated with it. Once a breakpoint had been set, actions could be added to or deleted from the list. The breakpoint itself would be deleted only when the last action in its list was deleted.

In summary, the use of breakpoints allows implementation of interactive debugging facilities, with only minor modifications being made to existing software; the modifications required depending on the type of debugging facility which is to be provided. The more sophisticated the debugging facility is, the greater the amount of information which has to be supplied by the compiler. The advantages of this method of implementation are: no alteration need be made

to the source program for debugging purposes; no extra object code is generated; it is not necessary to recompile or reload the program in order to change the debugging options; and debugging overhead can be removed at any time during execution.

This method of implementation, however, has a number of disadvantages which tend to outweigh the advantages. Firstly, the method is far better suited to communication at machine language level, rather than at source level, and the range of debugging tools which can be implemented is rather limited. The work done on MANTIS, however, shows that the use of breakpoints could be developed further to provide far better facilities for debugging in high level languages than has previously been done using breakpoints. Secondly, a number of problems arise when portions of the program are overlaid, as is the case in a time-sharing system, since breakpoints can only be set in the portion which is in core (Blai 71). In many larger systems, in particular those using paged memory, it is not possible to alter the contents of program words. In such systems, breakpoints simply cannot be used.

3.3 Implementation of Debugging Facilities using a Compiler

A common method of implementing debugging facilities for a high level language is to incorporate them in the compiler for the source language. The debugging facilities can then be controlled by commands included as source statements in the program, which may be given the status of pseudo-comments, or by compiler options, or by a combination of the two. During the scanning and analysis phases the compiler gathers information about the program which is needed for debugging. This information is used to provide tables needed during execution, such as a symbol table, and to guide the code generation phase in the generation of extra object code to give the required debugging options. This extra code normally consists of calls to run-time routines, though some debugging tools, for example execution counts, could be implemented using in-line code (See Fig. 3.3).

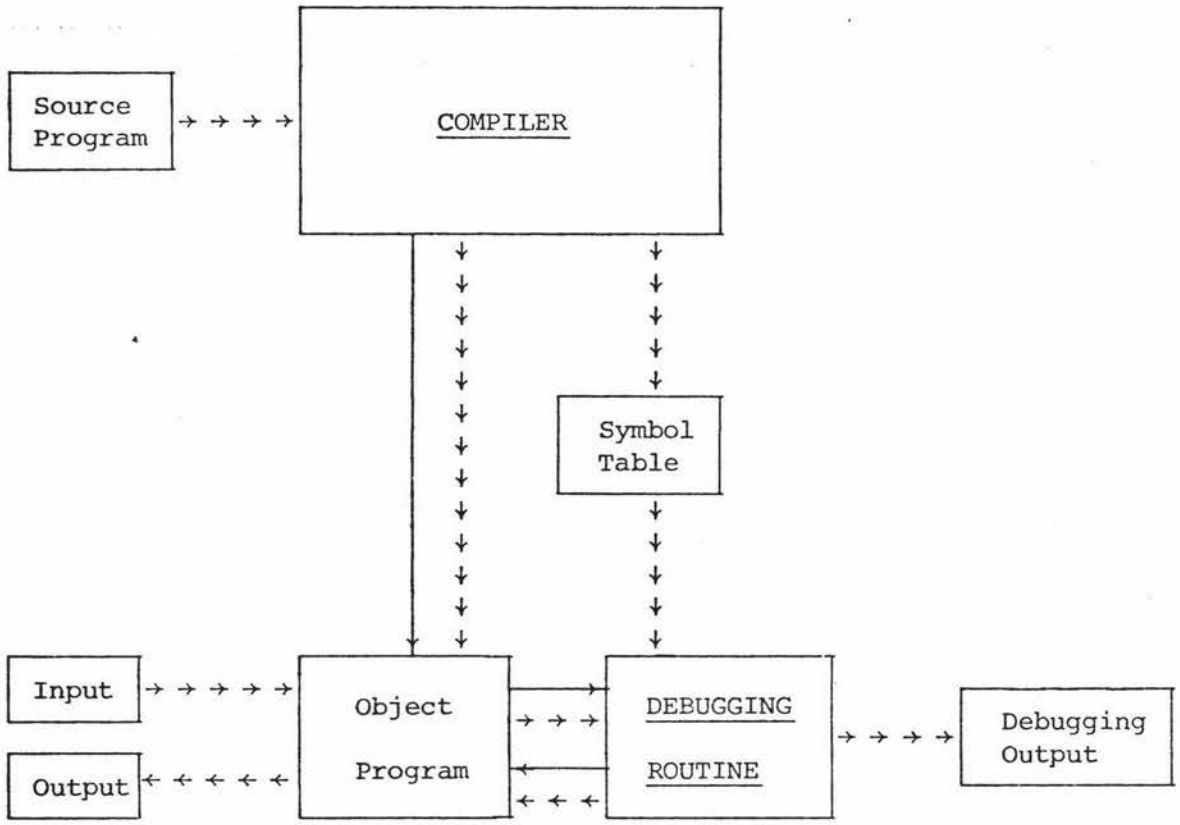


Fig 3.3 Implementation of Debugging Facilities using a Compiler

As with a precompiler, control is passed to the debugging routines using calls inserted in the program on the basis of a scan of the source program. Access to symbolic and structural information about the source program, however, is more difficult. Since the compiler is expected to provide diagnostic facilities at run-time as well as debugging facilities, the method used with the precompiler of passing symbolic information as parameters to the debugging routines is not adequate. Furthermore the object code produced by a compiler normally carries very little information about the structure of the source program. Some structural information could be retained in the object program; for example, dummy instructions could be used to mark the beginning of the object code corresponding to each source statement. Other symbolic and structural information must be passed on to the debugging routines in the form of tables, such as the symbol-table. In Multics PL/I (Wolm 72) the compiler also provides a statement map giving information about the location of the object segment for each source statement. Other tables required would depend on the source language itself and the type of checking and debugging tools being implemented.

The range of debugging tools which can be implemented in this manner is much the same as can be implemented using a precompiler, and without the overhead of the extra analysis of the source program. Again, in order to provide a Post Mortem dump facility it is necessary to modify the operating system so that control is passed to the language subsystem whenever a run-time system error is detected.

The diagnostic and debugging capabilities of a compiler can be greatly increased by using run-time routines to monitor certain phases of the execution, rather than just to implement trace, dumps and other debugging tools. For example, in PL/C (Conw 73) the execution supervisor handles GO TO statements, string manipulation, I/O, dynamic storage allocation and stack manipulation during block entry and exit. These supervisor functions are requested through calls to run-time routines which are compiled into the object code. Other phases of the execution which require run-time checking

for example, references to structured data or control of iterative loops, could be handled by run-time routines, rather than the compiler duplicating the checking code every time it is needed. When extensive use is made of run-time routines, execution is largely interpretive and the debugging facilities which can be offered are similar to those which can be provided using an interpreter (See section 3.4).

For good debugging facilities to be provided using the compiler it is generally necessary that the debugging facilities be part of the initial design of the compiler, or at least that the compiler be designed with due consideration being given to the later addition of debugging facilities. Building debugging facilities into an already existing compiler which has not been so designed is a much more difficult task.

3.4 Implementation of Debugging Facilities using an Interpreter

An interpretive system consists of a translator and an interpreter. The translator accepts the source program and translates it into a form suitable for interpretation. This intermediate form is then executed by the interpreter. The significant difference between an interpretive system and a compiler is in the control which the language subsystem has over the execution of the program. Using a compiler, the language subsystem only gains control when the object program branches to the execution supervisor. In an interpretive system, however, the interpreter has control of execution at all times, since all instructions executed directly by hardware are part of the interpreter (See Fig. 3.4). This is ideal for providing diagnostic and debugging facilities as it allows run-time semantic errors to be detected as early as possible and diagnosed accurately. Run-time system errors could also be detected by the interpreter, but only at the cost of a lot of extra checking. It is far more efficient for the interpreter to be designated as a privileged user, and for the operating system to pass control to the interpreter whenever an error is detected, either by

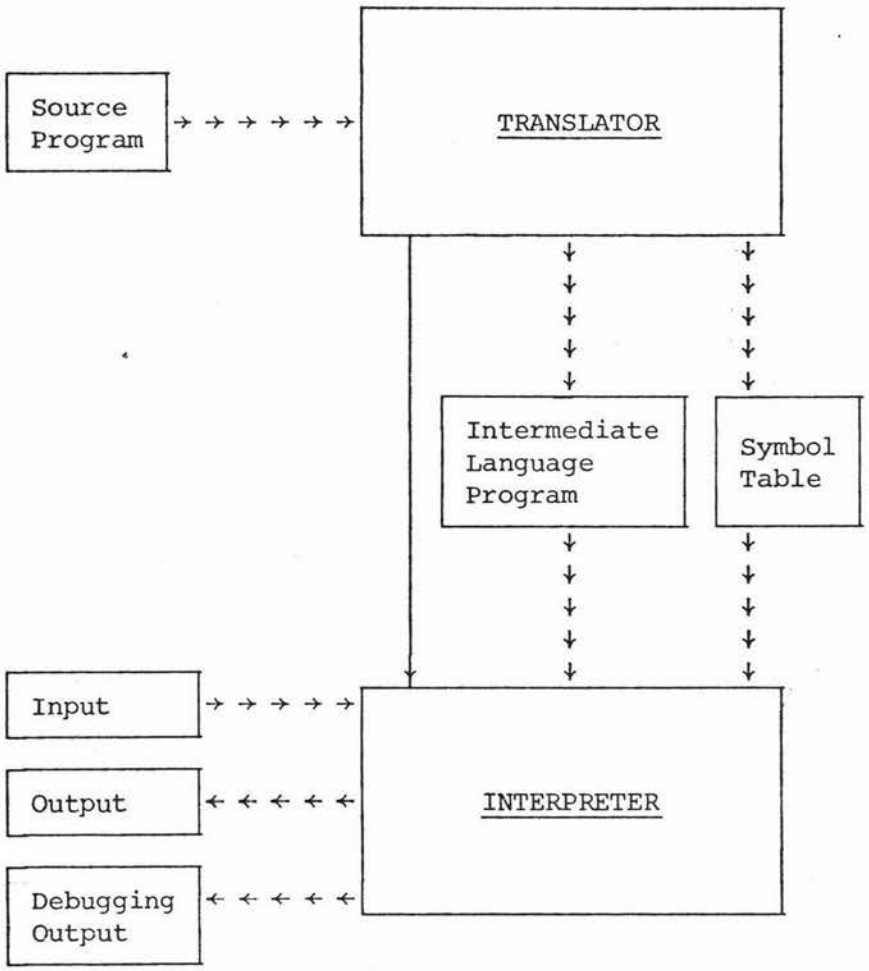


Fig 3.4 Implementation of Debugging Facilities using an Interpreter

the operating system or by the hardware, in the execution of an interpreter routine.

Three kinds of interpreters are defined, according to the amount of work done by the translator, and the resulting form of the input to the interpreter. This distinction is based on the model of the three-pass compiler (lexical scanning, syntactic and semantic analysis, and code generation).

(i) Source Interpreter

The translation phase is either non-existent or, more commonly, consists only of a lexical scan, during which the source program is translated into a stream of tokens. During this scan the translator may also perform some syntactic analysis, in particular checking the formation of identifiers, constants and delimiters, and build a symbol-table which is passed on to the interpreter. The interpreter completes the syntactic and semantic analysis of each statement as it is executed. The input to the interpreter is identical to the source program, except for the replacement of identifiers and constants with pointers to the symbol-table, and the substitution of internal codes for delimiters. The interpreter thus has access to all the information about the source program, allowing excellent diagnostic and debugging facilities to be provided. Execution using a source interpreter, however, is very slow, since the analysis of every statement is repeated each time it is executed.

(ii) Intermediate Language Interpreter

The translator performs a lexical scan, syntactic analysis and most of the semantic analysis, and translates the program into an intermediate language which is similar to the output of the second pass of a three pass compiler. The interpreter then executes this intermediate language, effectively replacing the code generation phase of the compiler. An intermediate language interpreter gives quick turnaround on non-production jobs, which often contain errors, as translation is fast and all static errors can be detected during translation. The diagnostic and debugging capabilities of an intermediate

language interpreter during execution are very good, since little information about the source program is lost during translation. These capabilities can be enhanced if the intermediate language is designed to reflect the structure of the source program precisely. In particular, this would allow the interpreter to determine, at any time, which source statement is being executed and what type of statement it is. The principles involved in writing an intermediate language interpreter are discussed in (Glas 69), and a particular implementation is described in detail in (Rand 64). This kind of interpreter has been used to provide diagnostic and debugging facilities in QUICKTRAN (Dunn 64, Kell 64), SPLINTER (Glas 68) and the PL/I Checkout Compiler (McHa 71, Cuff 72, Mark 73).

(iii) Simulator (Machine Language Interpreter)

The translator is a normal compiler, producing machine code, and the interpreter is a hardware simulator with additional diagnostic and debugging facilities. The advantage of using a simulator is that once a program has been debugged it can then be run efficiently, on hardware, without any modification or use of a different compiler. Indeed, parts of the program which have been debugged can be run on hardware, while other parts are executed by the simulator. This facility, called partial interpretation, has been implemented in ALGOL W (Satt 72) and in PEBUG (Blai 71). The use of a simulator, however, has two disadvantages when compared with an intermediate language interpreter. Firstly, translation is slower (much slower in the case of an optimizing compiler), thus giving slower turn-around for debugging runs. Secondly, the machine code which is interpreted carries very little information about the structure of the source program, making good source level debugging facilities difficult to implement. The information required could be provided by the compiler in the form of tables, but this is messy compared with the interpretation of an intermediate language which itself reflects the structure of the source program.

The use of an interpretive system offers excellent opportunities for provision of good source level debugging facilities. The overhead of interpretive execution during debugging, rather than being a disadvantage, is a worthwhile investment, since the number of runs and the amount of programmer time required to debug a program can be substantially reduced. Interpretive execution is also quite acceptable for short 'one-off' jobs which, with an optimizing compiler, typically spend an order of magnitude more time being compiled than they do executing. For most production running, however, interpretive execution is not acceptable, and a scheme is needed which allows interpretive execution during debugging, and efficient hardware execution for production running. Two such schemes have been described in the literature. One is to use a simulator, as described above for ALGOL W and PEBUG; the other is to use one compiler for debugging and a different one for production jobs.

The first reference to the use of two different compilers was made by Duncan (Dunc 62), who described the proposed 'Whetstone' and 'Kidsgrove' implementations of ALGOL 60 for the English Electric KDF9. The two compilers were to support an identical source language; the Whetstone version aiming at fast translation with no attempt to produce efficient code (being interpretive), while the Kidsgrove version aimed at producing efficient object code at the expense of slower translation. This idea has been applied more recently by IBM to its implementation of PL/I for the IBM System/360. The PL/I F compiler has been superseded by a pair of compilers; the PL/I Checkout Compiler and the PL/I Optimizing Compiler. The Checkout Compiler provides fast translation and a range of debugging tools using interpretive execution. The Optimizing Compiler is used to produce efficient object code for a program which is used repeatedly. Special commands used as directives to the Checkout Compiler are ignored by the Optimizing Compiler, and vice versa.

The major difficulty with the use of two separate compilers for the same language is in implementing and maintaining the two, so that they both support exactly the same source language. An alternative scheme is now presented which avoids this difficulty. This scheme is based on the observation that the translator used with an intermediate language interpreter performs the same tasks as the first two passes of a three-pass compiler, namely lexical scanning and syntactic and semantic analysis. The scheme, then is that a single translator is used to analyse the program and translate it into the intermediate language, which is then processed either by an interpreter or by an optimizing code generator, according to a compiler option (see Fig 3.5). Since the same translator is always used there is no difficulty with source language compatibility, but care is required to ensure that the treatment of the intermediate language is compatible. This is a comparatively easy task, involving the comparison of a pair of processors for each intermediate language operator; one which interprets the operator and one which generates object code for it. It then remains only to ensure that optimization does not corrupt the logic of the program.

The analysis phases of the PL/I Checkout Compiler and Optimizing Compiler are somewhat different, due to the differing needs of the latter phases of the two compilers; the major difference being that the intermediate language used by the interpreter carries extra symbolic information required for the provision of diagnostic and debugging facilities. When only one translator is used, as in the above scheme, the translator is oriented towards the diagnostic and debugging needs of the interpreter. That is, it is fast, so as to give quick turnaround for debugging and test runs, and retains as much symbolic and structural information as possible. If the code generator is used, this information can be used to guide optimization, instead of tables built during translation, and is otherwise ignored.

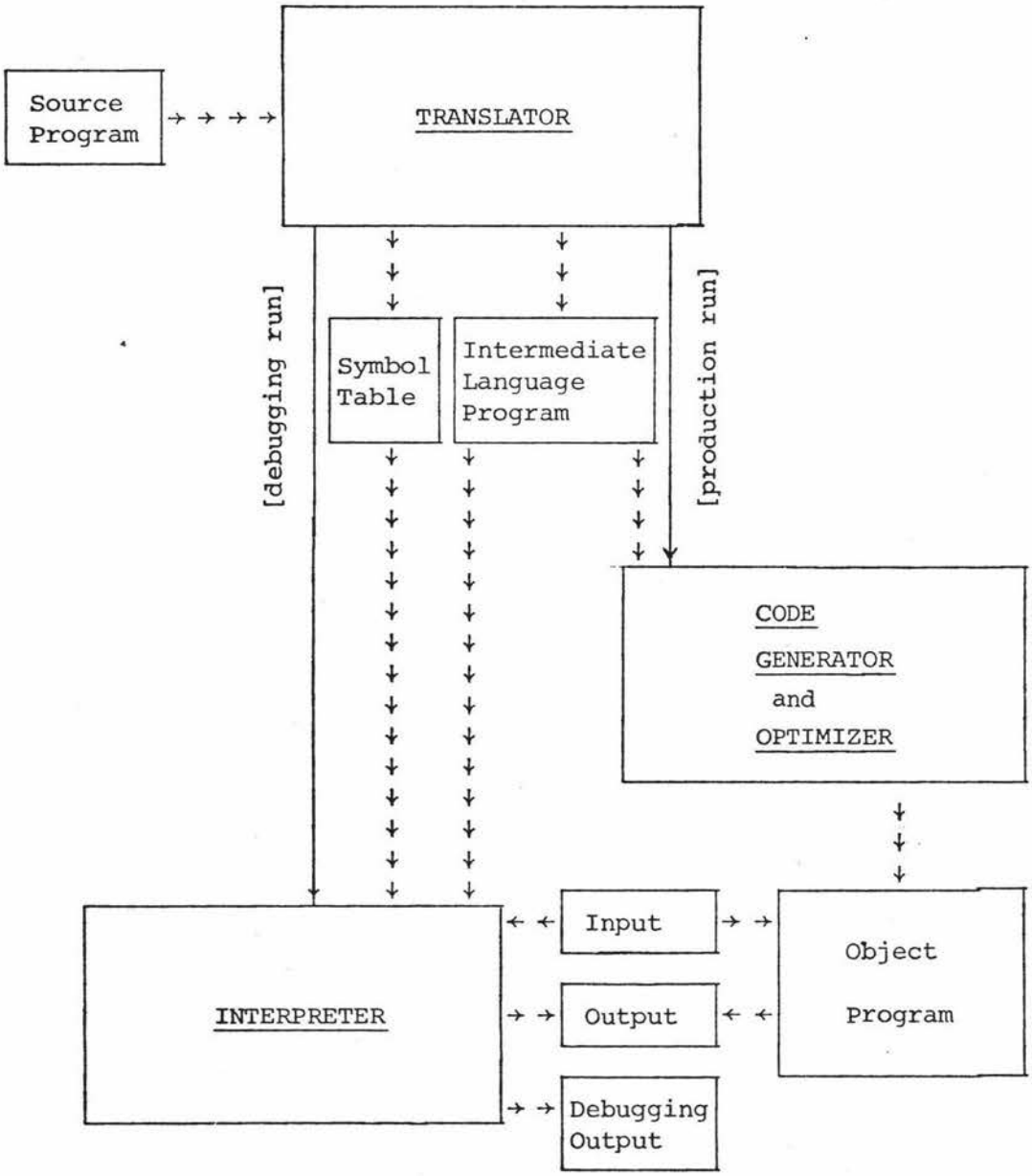


Fig 3.5 Logical Structure of Compiler/Interpreter System

This scheme could be extended to allow some sections of the program to be executed interpretively, while other sections are compiled to machine code and executed by hardware. This would only be a significant advantage when very large programs were being debugged. Some techniques for combining interpretive code with machine code are discussed in (Daki 73) and (Daws 73).

Specific details of the implementation of debugging facilities using an interpreter are discussed in the next chapter.

Chapter 4

PROVISION OF DEBUGGING FACILITIES FOR MUSSEL

4.0 Introduction

In this chapter the provision of debugging facilities for a particular language is discussed in detail. The language, which is called MUSSEL, is described briefly and suitable debugging facilities designed. A general description of the implementation of MUSSEL is then given, followed by a detailed description of the implementation of the debugging facilities. The implementation has been designed so as to be easily adaptable to use in an interactive environment.

Although a specific language is used for this discussion, the main interest is in the approach taken in designing the implementation. Modified appropriately, this approach could be used in implementing debugging facilities for any high level language which can be implemented using a run-time stack.

4.1 Overview of MUSSEL

MUSSEL (Massey University Structured Student Language) was designed to allow the expression of structured programs in a clear and simple manner. The language has only a few arbitrary syntactic restrictions, which were imposed to preserve its simple structure, so that it would be suitable as a teaching language. MUSSEL was designed and implemented on an IBM 1620 II at Massey University by Nola Simpson (Simp 73) and Peter Gibbons (Gibb 72). The syntax of MUSSEL is given in Appendix A.

The basic character set used in MUSSEL consists of: the 26 upper case letters, A through Z; the digits 0 through 9; and the special characters + - * / ! : . , ( ) $ and blank. MUSSEL has three data types:

Numbers of which integers are a subset;

Booleans taking the values TRUE or FALSE, which may be abbreviated to T or F;

Strings consisting of sequences of characters from the basic character set. String constants are enclosed between exclamation marks, and an exclamation mark is represented within a string as two consecutive exclamation marks.

Data values are stored either as simple variables or in arrays. The data type associated with each data item is determined dynamically, and thus may change any number of times during the execution of the program. In particular, this means that the elements of an array do not have to be all of the same type.

The assignment statement in MUSSEL has the form:

SET <list part> TO <expression>

where <list part> is a list of variable names separated by commas.

Examples :

```
SET A,B,C TO 0
SET K TO K+1
SET DATE TO DAY .CAT. !/! .CAT. MONTH
SET DATE TO DATE .CAT. !/! YEAR
SET M(I,J) TO M(I,J) + 1.5*Q(J) - 4.76
```

Program structuring in MUSSEL is achieved using DO groups (which are equivalent to compound statements in ALGOL 60) and procedures. A DO group is an optionally labelled group of statements enclosed between the statement parentheses DO and END. MUSSEL has no equivalent of the ALGOL block; all variables in the main program (called Main variables) are reserved at the beginning of the program, and are global to the program and its procedures. This avoids many of the difficulties arising from the scope rules in languages such as ALGOL 60. Main arrays have static bounds.

Procedures are declared at the begining of the program, immediately after the reservation of main variables. Local variables are reserved at the begining of the procedure, and local arrays may have dynamic bounds. All parameters are 'call-by-reference'. Procedure declarations may not be nested, enforcing a simple, two-level structure. Procedures may, however, be mutually recursive; that is, any procedure may call any other procedure including itself.

Choice constructions used in MUSSEL are: the IF-group, Conditional-group, CASE-group, and CHOICE-group.

Examples :

(1)
```
IF X .GT. Y .OR. Y .LT. Z THEN SET F TO 1
```

(2)
```
IF M .EQ. N
THEN DO
          SET MEAN TO SUMX/N
          SET VAR TO SUMX2/N - MEAN**2
     END
```

(3)
```
IF I .EQ. 0
THEN DO
          SET S TO N+K
          SET P TO P+S
     END
ELSE DO
          SET S TO N-K
          SET P TO P-S
     END
END
```

(4)
```
DO CASE OPCODE IN (22,26) OF
     SET OP TO !SUBTRACT!
     SET OP TO !MULTIPLY!
     SET OP TO !COMPARE!
     SET OP TO !TRANSMIT DIGIT!
     SET OP TO !TRANSMIT FIELD!
END
```

(5)
```
DO CHOICE OF
     IF TEMP .LT. 0 THEN SET S TO !FREEZING!!!
     IF TEMP .LT. 10 THEN SET S TO !COLD!
     IF TEMP .LT. 15 THEN SET S TO !COOL!
     IF TEMP .LT. 25 THEN SET S TO !MILD!
     IF TEMP .LT. 30 THEN SET S TO !WARM!
     IF TEMP .LT. 50 THEN SET S TO !HOT!
     ELSE SET TEMP TO :BOILING!!!
END
```

Iteration is achieved in MUSSEL using REPEAT groups, which have the form:

```
DO REPEAT (control)
   .....statements.....
END
```

The control part may be selected from a variety of options, which correspond essentially to the PERFORM verb options in COBOL.

In order to emphasise the structure of algorithms, MUSSEL has no GO TO statement. Instead program flow is controlled using the various control structures described above, with the addition of the EXIT statement, which specifies an exit from the innermost loop. An outer loop can be exited from by specifying its label.

Example :

```
HUNT : DO REPEAT FOR I FROM 1 TO I1
           DO REPEAT FOR J FROM 1 TO J1
                IF A(I,J) .EQ. MATCH
                THEN EXIT FROM HUNT
           END
       END
```

Simple input-output is provided using unformatted stream input and standard format output. More sophisticated output is provided using picture specifications, similar to those of COBOL and PL/I, and the printer control statements: NEWPAGE, NEWLINE(n), SPACE(n), AND TAB(n).

Examples :

```
READ X,Y,Z

PRINT A,B,C

PRINT NEWPAGE,TAB(30),!TITLE!,NEWLINE,TAB(30),!*****!

PRINT NEWLINE(2),TAB(28),!X = !,Z(PIC=***.**)
```

4.2 Debugging Facilities in MUSSEL

Debugging tools were selected for inclusion in the MUSSEL debugging facility on the basis of their usefulness (as discussed in Chapter 2), with consideration to the features of the MUSSEL language and the fact that MUSSEL is intended as a teaching language. Since MUSSEL is currently implemented as a batch processing system, none of the 'interactive only' debugging tools discussed in Chapter 2 could be included. The debugging facilities, which include execution counts, various forms of traces and dumps, and an ASSERT statement, are controlled using a combination of control cards and source statements. The debugging commands used in MUSSEL allow simple invocation of basic debugging options as well as flexible control of debugging for the more sophisticated programmer, and have been designed to be a natural extension of the MUSSEL language.

Execution counts are kept for all programs, and can be used within the program to control debugging. The reserved identifier EXCOUNT gives the value of the execution count of the statement in which it appears. Execution counts are used to give a program profile at the end of execution; and the execution count and line number of the relevant line of source code are given with every diagnostic or debugging message printed during execution. For example, the message:

```
12( 3) ERROR : DIVISION BY ZERO
```

indicates that the described condition was detected while the statement whose line number is 12 (i.e. the first number shown) was being executed for the third time (given by the number in brackets).

Traces are enabled and disabled by the TRACE and UNTRACE statements, respectively. The syntax of these statements is as follows:

```
<trace> ::= TRACE|TRACE <trace option list>

<untrace> ::= UNTRACE|UNTRACE <trace option list>

<trace option list> ::= <trace option>|<trace option list>,
                        <trace option>
```

```
<trace option> ::= STORES|STORES(<var list>)|
                   VALUES|VALUES(<val list>)|
                   TYPES|TYPES(<var list>)|
                   FLOW|LOOPS|SOURCE|
                   PROCEDURES|PROCEDURES(<proc list>)

<var list> ::= <var list elt>|<var list>,<var list elt>

<val list> ::= <val list elt>|<val list>,<val list elt>

<var list elt> ::= <simple variable>|<array id>

<val list elt> ::= <var list elt>|<function id>

<proc list> ::= <procedure id>|<proc list>,<procedure id>
```

Examples :

```
TRACE VALUES(A,B,C)

TRACE FLOW,STORES(X,Y),PROCEDURES(PROC1,PROC2,PROC3)

UNTRACE TYPES

IF EXCOUNT .GT. 1 THEN UNTRACE SOURCE

TRACE
TRACE STORES(THIS,THAT),FLOW,LOOPS,PROCEDURES
```

The trace options are as follows:

STORES: store trace, as described in Chapter 2.

VALUES: a combination of the store and fetch traces, as described in Chapter 2.

TYPES: trace of changes in the types of variables listed, or all variables if no list is given. MUSSEL's dynamic type feature saves many of the syntax errors which result from the typed declarations used in languages such as ALGOL, but can lead to very serious logical errors if misused. This trace is a tool enabling the programmer to keep track of intentional type changes and to locate any unintentional type changes. While

this trace is enabled, the new type and value is printed for each variable whose type is changed due to the execution of an assignment or READ statement. This includes the initialization of a variable, when its type is changed from 'undefined' to some other type.

FLOW: trace of flow of control through the program. Trace output includes: entries to and exits from DO groups and procedures: the option taken in each IF, CASE or CHOICE statement; and the value of the condition in each ASSERT statement. Output given on exit from a DO REPEAT group includes the number of iterations which were performed.

LOOP: trace of the control of iterative loops. Entries to and exits from DO REPEAT groups are traced as for flow trace. At the beginning of each iteration the iteration count is printed, accompanied by the value of the loop control variable, in the case of a REPEAT SET or REPEAT FOR loop, or the value of the loop condition, in the case of a REPEAT WHILE or REPEAT UNTIL loop.

PROCEDURES: trace of calls to procedures named in the list, or all procedures if no list is given. Entries to and exits from procedures are traced as for flow trace. In addition, for each parameter passed, the name of the formal parameter and the value of the actual parameter are printed. There is no need to identify the calling style, since all parameters are call-by-reference.

SOURCE: source trace, as described in Chapter 2.

If no trace option list is used in a TRACE or UNTRACE statement, then all traces are turned on or off, respectively. Thus the following two statements are equivalent:

```
TRACE
TRACE VALUES,TYPES,FLOW,LOOPS,PROCEDURES,SOURCE
```

TRACE and UNTRACE statements are effective only if a $DEBUG control card is used, and are otherwise treated as comments. Tracing can thus be deleted merely by removing this control card.

A simple frequency controlled trace is provided which allows all statements to be traced only a specified number of times. This facility is invoked using a $TRACE ALL control card. If no number is specified, a default value of three is assumed.

A history of the execution leading up to the point of termination can be requested, either conditionally upon the manner of termination or unconditionally, using a $HISTORY control card. This control card has two parameters. The first parameter is empty if the history is to be given only in the event of abnormal termination and is specified as ALWAYS if the history is to be given irrespectively of the manner of termination. The second parameter is a number specifying the number of lines of history which are to be printed. The information contained in the history is the same as would have been printed if all traces were active at that time.

Snap dumps and partial dumps are requested using the DUMP statement, which has the following syntax :

<dump> ::= DUMP|DUMP <dump list>

If no dump list is used then a snap dump is given, including all the variables which are currently accessible. Since procedure declarations in MUSSEL can not be nested, only local variables of the current procedure and main variables are accessible during the execution of a procedure. For arrays in the current environment, only the first five and the last elements from the first five and the last rows are printed.

If a list is given then a partial dump, including those items specified in the list, is given. Items in the list may be simple variables, array elements or array ranges, and conditions may be attached to single items or groups of items.

The dump list has the following syntax :

```
<dump list> ::= <dump list elt>|<dump list>,<dump list elt>
<dump list elt> ::= <dump list item>|
          IF <cond exp> THEN <dump list item>
<dump list item> ::= <simple variable>|<array range>|
          (<dump list>)
<array range> ::= <array id>(<sub range>)|
          <array id>(<sub range>,<sub range>)
<sub range> ::= <val exp>|<val exp> TO <val exp>
```

Examples :

```
DUMP N,IF I.EQ.0 THEN (A,B,IF J.EQ.0 THEN X,C),K
DUMP ARRA(K+4,I1 TO I2)
```

Like the TRACE and UNTRACE statements, the DUMP statement is effective only if a $DEBUG control card is used.

A Post Mortem dump is given for all programs which terminate abnormally, and can be requested irrespective of manner of termination using the control card:

```
$PM DUMP
```

The PM dump includes the data for each procedure which has been entered but not exited at the point of termination, starting from the current procedure and working back to the main program. If the program terminates normally, the final values of local variables of procedures are lost, since they have been wiped from the stack. Hence in this case the Post Mortem dump contains only the main variables and arrays, which are preserved as they are not stored in the stack. As in the snap dump, only the first five and the last elements of the first five and the last rows are printed.

The ASSERT statement in MUSSEL has the same form as described in Chapter 2, and is effective only if a $DEBUG control card is used. When the condition in an ASSERT statement is found to be false, an error message is printed and execution is terminated. Since this is an abnormal termination, a Post Mortem dump will then be given.

4.3 Implementation of MUSSEL

MUSSEL has been implemented as an interpretive system using an intermediate language interpreter. This method of

implementation was chosen in order to minimize compilation time and maximize diagnostic and debugging capabilities at run-time; both of these goals were considered essential in the implementation of a teaching language. Though the present implementation is as a batch system, it is hoped that MUSSEL will eventually be implemented as an interactive system; the debugging facilities described in this chapter have been designed so that they could easily be adapted to provide interactive debugging facilities. Since MUSSEL is intended as a teaching language it is not likely to be used for running long production jobs, so there is no need to provide an efficient mode of execution and the interpreter is used for all jobs.

The MUSSEL system consists of two main parts; the Translator and the Interpreter. Each of these is divided into three phases as described below. The general structure of the MUSSEL system is shown in Fig 4.1.

The first phase of the Translator is the main control routine for the batch system. This phase reads the source program from cards, stores it on disk, and prints a job heading which includes the programmer's name, the date, and a list of the control cards used. If a $CARD LIST control card is used, a listing of the program as read from cards, with line numbers added, is given.

The second phase is the main body of the Translator, and performs: lexical analysis; syntactic analysis, using a top-down, syntax-directed recognizer; and various semantic actions, including generation of the Intermediate Language program. This phase also builds a template (Grie 71) of the storage required by the main program and by each procedure. The procedure templates are used at run-time for accessing the symbol-table entries of local variables.

The third phase of the Translator is an interface module. Its first task is to print the compilation cross-reference table, which gives the line number of every occurence of each identifier used in the source program. If any compilation errors were detected during the second phase, this phase gives

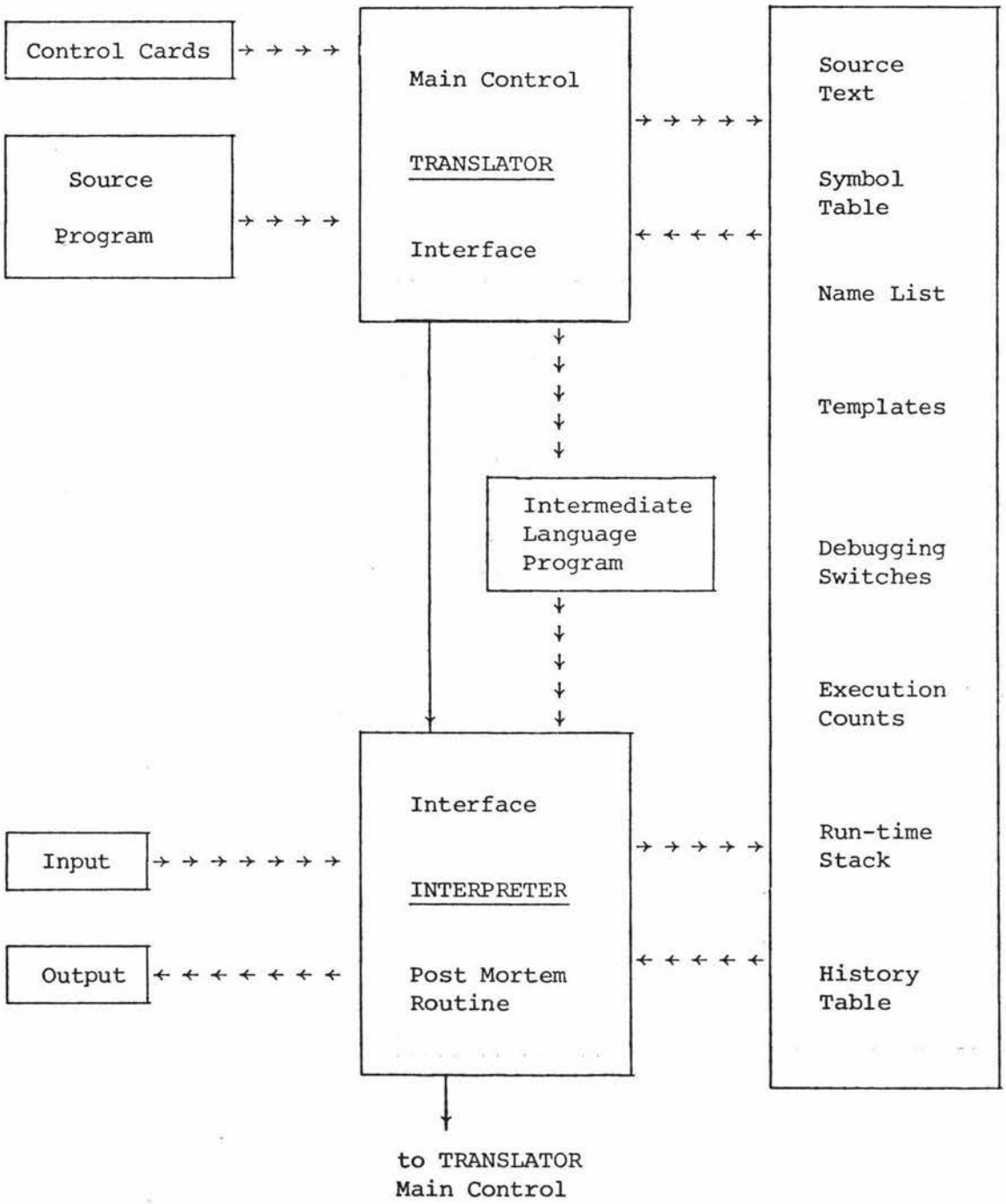


Fig 4.1 General Structure of the MUSSEL System

a listing of the program which is correctly indented with line numbers added, and has the appropriate error message(s) following each line in which an error was detected; control is then passed to the main control routine (the first phase of the Translator) which begins processing the next job in the batch. If no compilation errors were detected, the interface module links to the Interpreter, passing the symbol-table, the Intermediate Language program, and the procedure templates. If a $CODE control card is used, a listing of the Intermediate Language program is given.

The first phase of the Interpreter is an interface routine, which merely initializes the data areas used by the Interpreter, and prints a message indicating the beginning of execution.

The second phase, which is the main body of the Interpreter, handles the execution of the Intermediate Language program, using a run-time stack for evaluation of expressions and allocation of dynamic storage. This phase consists of: the Interpreter Control Routine; a set of interpretive routines corresponding to the Intermediate Language operators, which are called by the Interpreter Control Routine; and a few service routines used by the interpretive routines. These service routines include the Debug Output Routine (DOR), which is used to output information including debugging messages, run-time error messages, and to build the history table, if required. The symbol-table is kept during execution, and is used to access the symbolic names of variables, procedures and labels.

The third phase of the Interpreter is the Post Mortem Routine (PMR), to which control is passed when execution of the Intermediate Language program is terminated, whether normally or abnormally. The Post Mortem Routine firstly gives a listing of the program which is correctly indented and has line numbers and execution counts added. Then, according to the control cards used and the manner of termination, a Post Mortem dump and a history may be printed. The Post Mortem Routine finally links to the main control routine (the first

phase of the Translator) to commence processing the next job of the batch.

The Intermediate Language used in MUSSEL is essentially a 'Reverse Polish' notation, and is similar to, though much simpler than, that used by Randell and Russell (Rand 64). A special operator (LN) is inserted at the beginning of the object code (Intermediate Language) for each line of source code, allowing reference back to the source code corresponding to any section of object code. A number of modifications to the Intermediate Language are required in order to implement the debugging facilities described in section 4.2. These include: addition of new operators to control debugging; addition of new operators which provide structural information about the source program; and alteration of the interpreter's treatment of a number of operators. These modifications are discussed in section 4.4, and the modified form of the Intermediate Language is described in detail in Appendix B.

An important feature of the implementation is the run-time symbol table, which contains entries for all symbols used in the source program. (In the initial implementation only main variables, main array names and procedure names were included in the run-time symbol table, while labels, formal parameters, local variables and local array names were omitted.) Each symbol table entry consists of: a pointer to the name of the symbol, which is stored in the name area; a type field, indicating the kind of entry; some trace switches; and a value part, which takes on various different forms according to the kind of entry. The type field of a main variable is initially set to 'undefined' and is set to 'number', 'Boolean' or 'string', as appropriate, whenever an assignment is made to that variable. The value of the variable is stored in the value part of the symbol table entry unless the value is a string, in which case it is stored in the string area and the appropriate string descriptor is stored in the value part. The forms of the value part used for other kinds of symbol table entries will be discussed later, where they are relevant to the implementation of the debugging facilities.

The trace switches in the symbol table are used to control the symbol-oriented traces (i.e. Store, Value, Type and Procedure Traces), while global trace switches are used to control the non-symbol-oriented traces (i.e. Flow, Loop and Source Traces). Eleven new operators are used to turn these trace switches on and off. These operators fall into three groups :

(1) Operators which turn on or off a symbol-oriented trace for a single symbol :

TSS (on/off, S.T. addr)	Trace Single Stores
TSV (on/off, S.T. addr)	Trace Single Values
TST (on/off, S.T. addr)	Trace Single Types
TSP (on/off, S.T. addr)	Trace Single Procedures

One of these operators is generated by the Translator for each item in a trace list for a symbol-oriented trace option in a TRACE or UNTRACE statement.

(2) Operators which turn on or off a symbol-oriented trace for all symbols of the appropriate type :

TGS (on/off)	Trace General Stores
TGV (on/off)	Trace General Values
TGT (on/off)	Trace General Types
TGP (on/off)	Trace General Procedures

One of these operators is generated for each symbol-oriented trace option with no trace list in a TRACE or UNTRACE statement.

(3) Operators which turn on or off a non-symbol-oriented trace :

TGF (on/off)	Trace General Flow
TGL (on/off)	Trace General Loops
TGX (on/off)	Trace General Source

One of these operators is generated for each non-symbol-oriented trace option in a TRACE or UNTRACE statement.

4.4 Implementation of Debugging Facilities in MUSSEL

4.4.1 Execution Counts

During translation, the source program is divided into basic blocks, where a basic block is defined to be :

> "a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed)" (Alle 70).

The beginning of the object code for each basic block is indicated by a BBB (Begin Basic Block) operator, and the interpreter keeps an execution count for each basic block. The number of basic blocks in the program is supplied by the Translator; and the interface routine at the beginning of the Interpreter allocates storage for the basic blocks and initializes them all to zero.

The BBB operator has one parameter, which is the cardinality of the BBB operator in the program. Each time a BBB operator is executed, the Interpreter uses this

parameter to access the corresponding execution count, which it then increments by one. The new value is then stored in the reserved variable EXCOUNT, so that the execution count of the current statement is always available to the interpreter, for diagnostic and debugging purposes, and to the program itself. The Translator ensures that the program does not attempt to redefine the value of EXCOUNT.

In order to avoid the possibility of an execution count becoming too large and causing an overflow condition to occur, an upper limit of 9,999 is placed on the values of execution counts which are stored. Whenever the value of an execution count reaches 10,000 it is reset to zero and a flag set. When the execution count is subsequently printed, this flag causes an asterisk to be printed preceding the value of the execution count, indicating that the printed value is the actual execution count modulo 10,000.

Example:

The following message might have been printed in the 11056th or 21056th etc. execution of line number 74:

```
74(*1056) FLOW : IF CONDITION IS TRUE
```

4.4.2 Store Trace

A store operation may be initiated by a SET statement or by a READ statement. In the case of a SET statement, the translator generates object code to push the address of the data item and the value to be stored into the stack. This is followed by a ST (Store) operator, or a STA (Store Also) operator in the case of multiple assignments. In the case of a READ statement, the translator generates object code to push the address of the data item into the stack, followed by a RD (Read) operator, which takes the next data value from the card input buffer and pushes it into the stack.

The ST, STA and RD interpretive routines all call a common routine which performs the actual store operation. This routine first of all calculates the address of the symbol table entry corresponding to the data item to which the store is being made, and tests its store trace switch. If this switch is on, or if the TRACE ALL or the HISTORY facility is active, the symbol table address and the value to be stored are passed to the Debug Output Routine, which outputs the appropriate store trace message.

The Debug Output Routine uses the symbol table address to retrieve the symbolic name of the data item, a pointer to which is stored in the symbol table entry. In calculating the symbol table address, four kinds of data item need to be considered: main variables, main array elements, local variables, and local array elements. In the case of a main variable, the address in the second-to-top stack location is the required symbol table address, since main variables are stored in the symbol table.

If the data item is an array element, then to identify it uniquely the Debug Output Routine needs the symbol table address of the array word, which contains a pointer to the symbolic name of the array, and the values of the subscripts. In calculating the address of an array element, the translator generates object code which pushes the address of the array word and the values of the subscripts into the stack. This is followed by an I DA (Index Address) operator, which uses these values to calculate the address of the array element, and pushes this address into the stack. In the initial implementation of MUSSEL, as is usual when using a run-time stack, the array word address and the values of the subscripts are wiped from the stack as soon as they have been used to calculate the address of the required element. In order that stores to array elements can be traced adequately, the interpretation process has to be modified so that these values remain in the stack, and are only wiped after the store operation has been performed. This also allows array elements

to be identified correctly in the event of a run-time error being detected. In the case of a main array, the array word address which is in the stack is the required symbol table address (See Fig 4.2).

If the data item is a local variable, or a local array element, then accessing its symbol table entry is more difficult; the initial implementation of MUSSEL did not include local variables or local array names in the run-time symbol table, and no means of accessing their symbolic names was provided. In order to allow source level traces to be given, the run-time symbol table must be extended to include an entry for each local variable and local array, which contains: a pointer to the symbolic name; a type field indicating 'local variable' or 'local array word'; the symbol table address of the Procedure Address Word (PAW); and the relative address of the variable or array name (this corresponds to the address couple normally used, however the block level, which comprises the other half of the address couple, is not necessary because of MUSSEL's simple two-level structure). This symbol table entry is accessed using the procedure template built during translation.

When a store is made to a local variable, the value to be stored is found at the top of the stack as usual, while the second-to-top stack unit contains the address of the local variable relative to the Return Address Word (RAW) in the stack. The RAW contains a pointer to the base of the template for that procedure, which in turn contains a pointer to the PAW for the procedure in the symbol table. The other entries in the template are pointers to the symbol table entries for the corresponding variables or array names in the procedure. Thus, the relative address of the local variable, which is in the second-to-top stack location is used as an Index into the template to retrieve the required symbol table address.

If the data item to be stored into is a local array element, then the data in the stack just before the store operation is performed is the same as for a main array element,

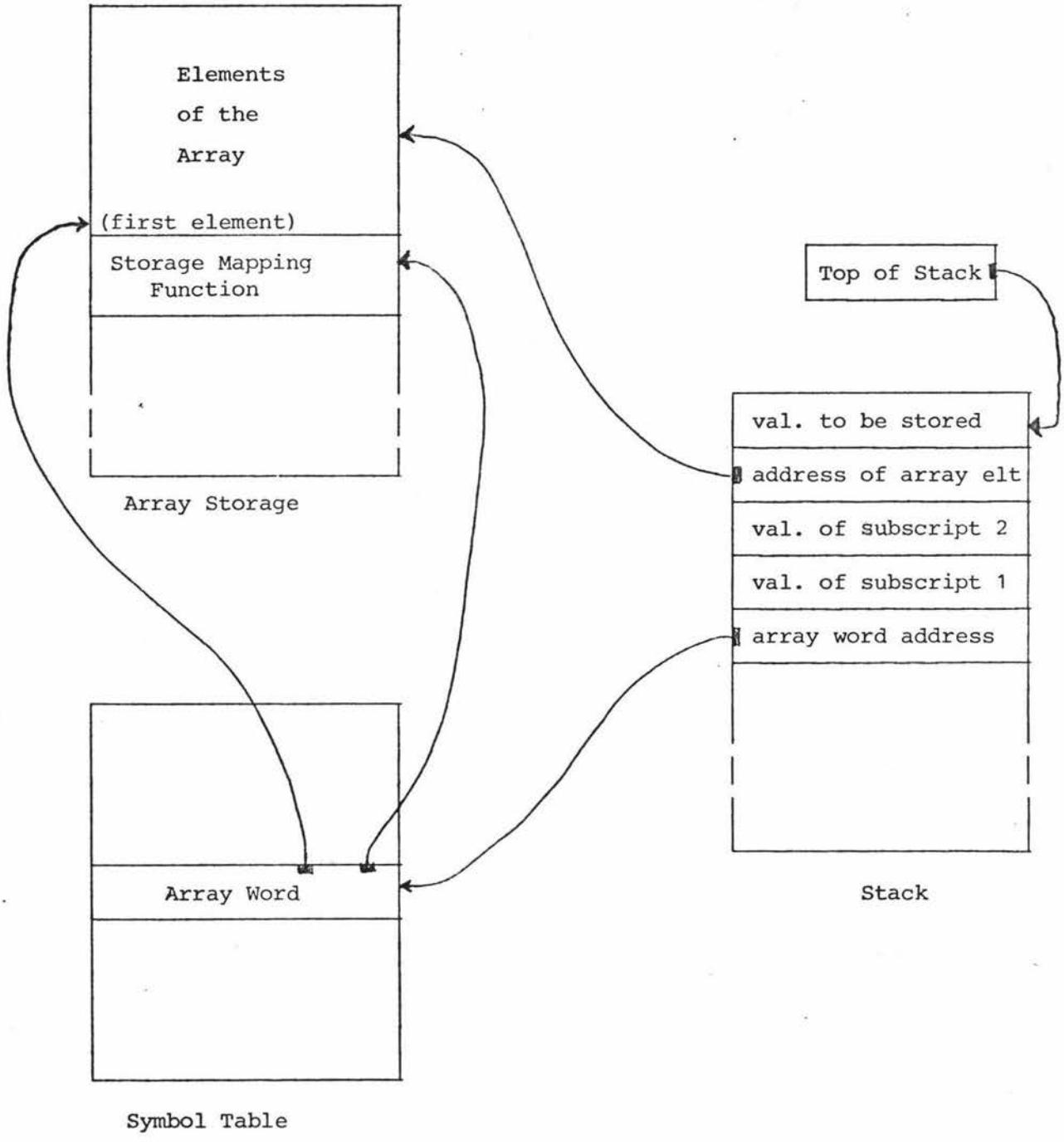


Fig 4.2 Accessing Symbol Table Entry for Main Array

except that the array word address is a relative stack address rather than a symbol table address. This relative address is used to retrieve the symbol table address from the procedure template, just as for a local variable (See Fig 4.3).

Example :

(line no.) Source code :

```
                .
                .
     7          TRACE STORES
     8          SET I,J,K TO 1
     9          SET A(I) TO J
    10          SET A(I+1) TO A(I)+1
                .
                .
```

Trace output :

```
 8(    1) STORE : I = 1
 8(    1) STORE : J = 1
 8(    1) STORE : K = 1
 9(    1) STORE : A(1) = 1
10(    1) STORE : A(2) = 2
```

4.4.3 Value Trace

The value trace consists of a store trace, as described in section 4.4.2, and a fetch trace. When a variable is referenced in an expression, the translator generates object code to push its address into the stack. In the case of an array element the address of the array word and the values of the subscript expressions are pushed into the stack. This is followed by a TR(Take Result), TLR (Take Local Result), or INDR (Index Result) operator, according to whether the data item is a main variable, a local variable, or an array element. This operator is responsible for fetching the required value and placing it at the top of the stack, after wiping the data used from the stack.

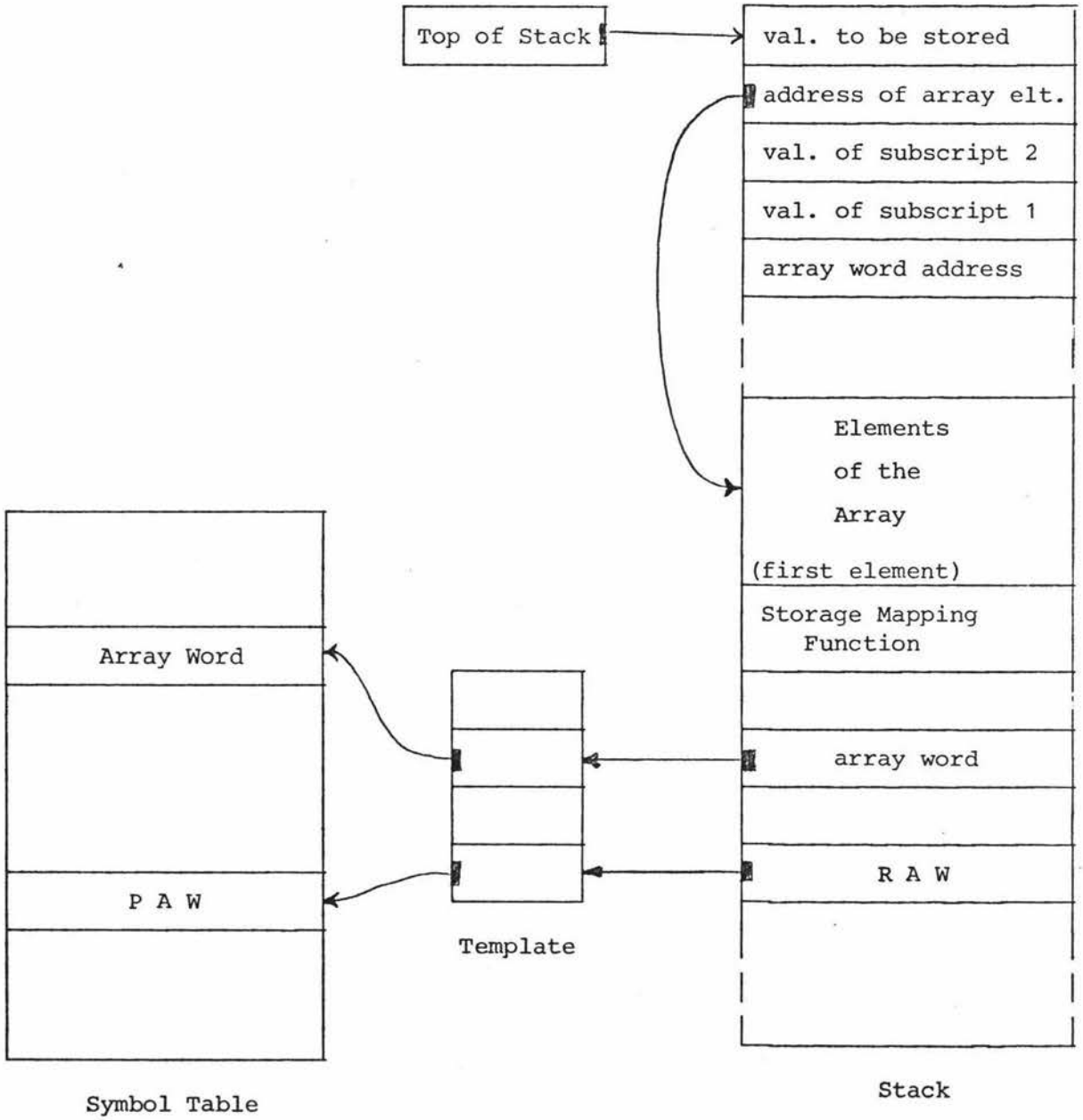


Fig 4.3 Accessing Symbol Table Entry for Local Array

The TR, TLR and INDR interpretive routines all call a common routine which performs the actual fetch operation. This routine first of all calculates the address of the symbol table entry corresponding to the data item whose value is to be fetched, and tests its fetch trace switch. If this switch is on, or if the TRACE ALL or HISTORY facility is active, the symbol table address and the fetched value are passed to the Debug Output Routine, which outputs the appropriate fetch trace message.

The procedure for calculating the symbol table address for the fetch trace is similar to that used for the store trace. If the data item is a main variable, then the address at the top of the stack when the TR operator is executed is the required symbol table address. If it is a local variable, the top stack unit contains the relative stack address of the variable, which is used to obtain the required symbol table address from the procedure template. In the case of an array element, the subscript values and the array word address are retrieved from the stack as in the store trace.

The value trace also includes a trace of values returned by functions. In translating a function call the translator generates a RAF (Return Address Function) operator, followed by code to stack the values of the parameters, and a CP (Call Procedure) operator. The RAF operator reserves a stack location, with type field set to 'undefined', for the value of the function. The value is later stored in this location by a VI (Value Is) operator in the body of the function. The VI interpretive routine tests the fetch trace switch for the function, and if this switch is on, or if the TRACE ALL or HISTORY facility is active, the symbol table address of the Procedure Address Word and the function value are passed to the Debug Output Routine. The symbol table address of the Procedure Address Word is obtained from the base entry of the procedure template, a pointer to which is found in the RAW. (See Fig 4.4).

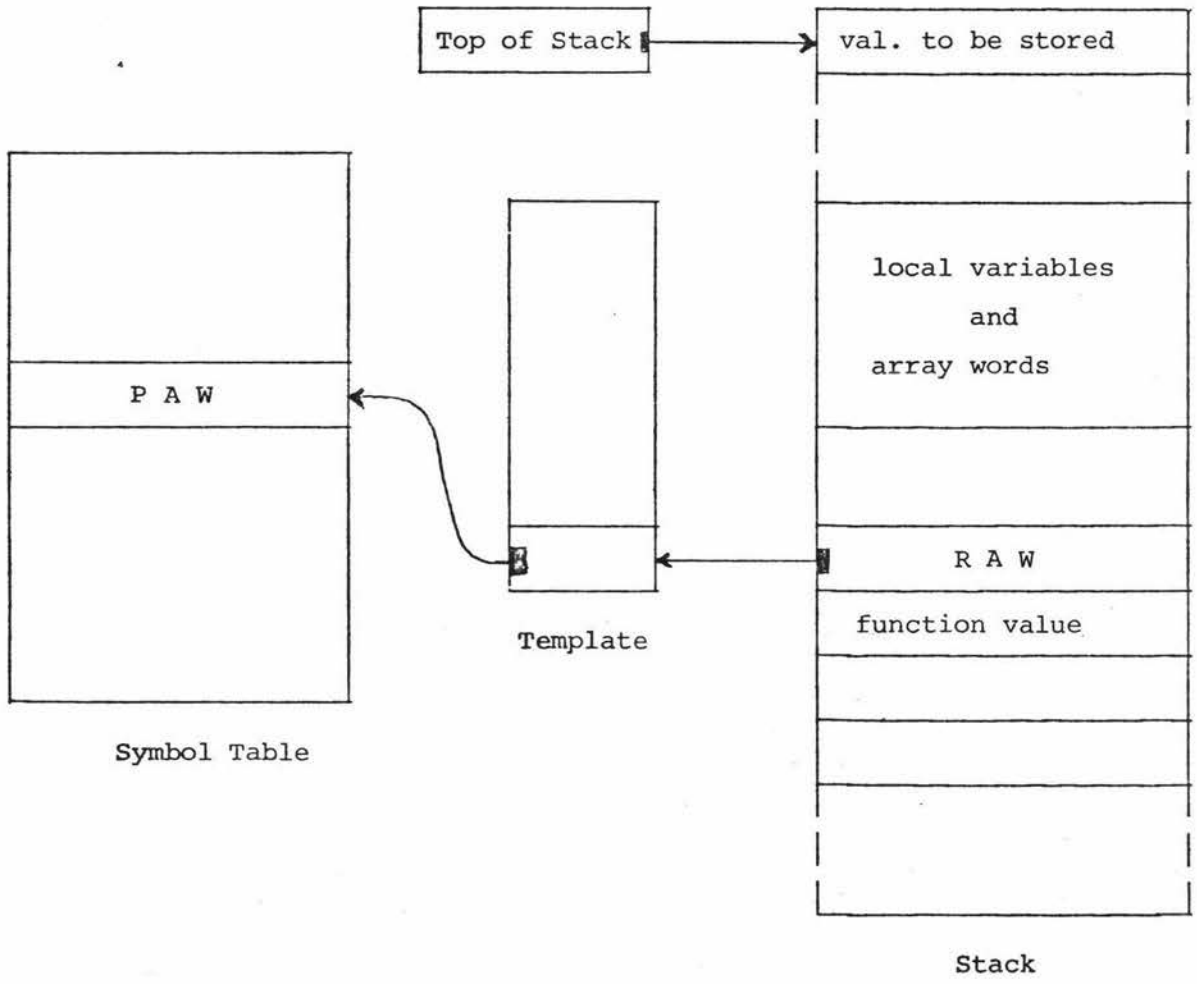


Fig 4.4 Accessing Symbol Table Entry for Function

Example :

Source code :

```
(line no.)
                .
                .
                .
    7           TRACE VALUES (I,J,A)
    8           SET I,J,K TO 1
    9           SET A(I) TO J
   10           SET A(I+1) TO A(I)+1
                .
                .
                .
```

Trace output :

```
 8(    1) STORE : I = 1
 8(    1) STORE : J = 1
 9(    1) FETCH : I = 1
 9(    1) FETCH : J = 1
 9(    1) STORE : A(1) = 1
10(    1) FETCH : I = 1
10(    1) FETCH : I = 1
10(    1) FETCH : A(1) = 1
10(    1) STORE : A(2) = 2
```

4.4.4 Type Trace

Whenever a store operation is performed, just after it has tested the store trace switch, the store routine tests the type trace switch in the symbol table entry corresponding to the data item to which the store is being made. If this switch is on, or if the TRACE ALL or HISTORY facility is active, the type of the value to be stored, which is at the top of the stack, is compared with the type of the value already in the location whose address is in the second-to-top stack location. If these two types are different they are passed, along with the corresponding symbol table address, to the Debug Output Routine, which outputs the appropriate type trace message.

Since the type field of every data location is initially set to 'undefined', the initialization of every data item is recorded by the type trace.

Example :

Source code :

```
(line no.)

                .
                .
3               TRACE TYPES
4               SET X TO !CHAIN!
5               SET X TO FALSE
6               SET X TO 10
                .
                .
```

Trace output :

```
3(    1) TYPE : UNDEF TO STRING, X = !CHAIN!
4(    1) TYPE : STRING TO BOOLEAN, X = FALSE
5(    1) TYPE : BOOLEAN TO NUMBER, X = 10
```

4.4.5 Flow Trace

In order to implement the flow trace, the Interpreter must be able to recognise the various types of program structures from the Intermediate Language code being executed. Some modifications to the Intermediate Language are required in order to achieve this.

1. Procedures

Entry to or exit from a procedure is characterized by the execution of a CP (Call Procedure) or RE (Return) operator, respectively. When either of these operators is executed, the symbol table address of the PAW is retrieved from the base entry of the procedure template, and the procedure trace switch in the PAW is tested. If this switch is on, or if the flow trace switch is on, or if the TRACE ALL or HISTORY facility is active, the symbol table address of the PAW is passed to the Debug Output Routine, which outputs the appropriate flow trace message.

Example :

Source code :

```
(line no.)
                .
                .
     3          DEFINE BUMP ON A,I AS
     4          DO
     5               SET A TO A+I
     6          END
                .
                .
                .
    20          TRACE FLOW
    21          EXECUTE BUMP(A,4)
                .
                .
```

Trace output :

```
 20(    1) FLOW : ENTER PROCEDURE BUMP
  5(    1) FLOW : EXIT PROCEDURE BUMP
```

2. IF groups and Conditional groups

Entry to an IF group or conditional group is characterised by the execution of an IFJ (If False Jump) operator, which is used to test the value of the IF condition. If the flow trace switch is on, or if the TRACE ALL or HISTORY facility is active when an IFJ operator is executed, the value of the IF condition is passed to the Debug Output Routine which outputs the appropriate flow trace message.

Example :

Source code :

```
(line no.)
                .
                .
    12          SET X TO 2
    13          TRACE FLOW
    14          IF X .EQ. 1 THEN SET X TO X+1
    15          IF X .GT. 1
    16          THEN DO
    17                    SET X TO X/2
    18                    PRINT X
    19              END
                .
                .
```

Trace output :

```
 14(    1) FLOW : IF CONDITION IS FALSE
 15(    1) FLOW : IF CONDITION IS TRUE
```

3. CHOICE groups

A CHOICE group is equivalent to a series of IF groups, and as such does not need any special Intermediate Language operator to control it. The following example demonstrates a CHOICE group and an equivalent series of IF groups; the two forms would be translated identically by the initial version of MUSSEL, with the exception that the label used in the second version would not appear in the symbol table for the first version.

Example : (i) CHOICE group

```
(line no.)
                .
                .
    15          DO CHOICE OF
    16               IF RN .EQ. !BROTHER! THEN SET GENGAP TO 0
    17               IF RN .EQ. !FATHER! THEN SET GENGAP TO 1
    18               IF RN .EQ. !SON! THEN SET GENGAP TO 1
    19               IF RN .EQ. !GRANDDAD! THEN SET GENGAP TO 2
    20               IF RN .EQ. !GRANDSON! THEN SET GENGAP TO 2
    21               ELSE SET GENGAP TO 3
    22          END
                .
                .
                .
```

(ii) Series of IF groups

```
(line no.)
                .
                .
                .
    15 GAPTEST:DO
    16              IF RN .EQ. !BROTHER!
    17              THEN DO
    18                        SET GENGAP TO 0
    19                        EXIT FROM GAPTEST
    20                   END
    21              IF RN .EQ. !FATHER!
    22              THEN DO
    23                        SET GENGAP TO 1
    24                        EXIT FROM GAPTEST
    25                   END
    26              IF RN .EQ. !SON!
    27              THEN DO
    28                        SET GENGAP TO 1
    29                        EXIT FROM GAPTEST
    30                   END
    31              IF RN .EQ. !GRANDDAD!
    32              THEN DO
    33                        SET GENGAP TO 2
    34                        EXIT FROM GAPTEST
    35                   END
    36              IF RN .EQ. !GRANDSON!
    37              THEN DO
    38                        SET GENGAP TO 2
    39                        EXIT FROM GAPTEST
    40                   END
    41              SET GENGAP TO 3
    42              EXIT FROM GAPTEST
    43         END GAPTEST
                .
                .
                .
```

In order to allow the interpreter to recognise the beginning of a CHOICE group, a new operator, CH (Choice), is introduced which is inserted at the beginning of the object code for each CHOICE group. The only function of the CH operator is to see if flow trace is required. If the flow trace switch is on, or if the TRACE ALL or HISTORY facility isactive, a call is made to the Debug Output Routine, which outputs the appropriate flow trace message.

Flow within the CHOICE group is indicated by the result of each IF condition tested.

Example : Flow trace of the execution of above CHOICE group, with RN set to !FATHER!

```
15(   1) FLOW : ENTER CHOICE GROUP
16(   1) FLOW : IF CONDITION IS FALSE
17(   1) FLOW : IF CONDITION IS TRUE
```

4. CASE groups

Entry to a CASE group is characterized by the execution of a CS (Case) or CSWB (Case Without Bounds) operator, depending on whether bounds for the case index are specified. When either of these operators is executed the flow trace switch is examined, and if it is on, or if the TRACE ALL or HISTORY facility is active, the value of the case expression and the case index are passed to the Debug Output Routine, which prints the appropriate flow trace message.

Example :

Source code :

```
(line no.)
                .
                .
   12     DO CASE OPCODE IN (34,39) OF
   13          SET OP TO !CONTROL!
   14          SET OP TO !DUMP NUMERICALLY!
   15          SET OP TO !READ NUMERICALLY!
   16          SET OP TO !READ ALPHAMERICALLY!
   17          SET OP TO !WRITE NUMERICALLY!
   18          SET OP TO !WRITE ALPHAMERICALLY!
   19     END
                .
                .
```

Trace output :

```
12(   1) FLOW : CASE EXPRESSION IS 36, INDEX IS 3
```

5. REPEAT groups

In the initial implementation of MUSSEL, execution of an RF (Repeat For), RFWT (Repeat For Without Test), RS (Repeat Set), or RT (Repeat Times) operator characterizes the corresponding form of REPEAT group, while no such special control operator

is used for the repeat-while, repeat-until or repeat-without-control forms. For a repeat-while or repeat-until loop the translator generates object code to evaluate the loop condition and, in the case of a repeat-until loop, negates it. This is followed by an IFJ operator which causes a jump out of the loop if the loop condition is satisfied. Use of the IFJ operator in controlling repeat-while and repeat-until loops means that the interpreter can not recognise either of these forms of loops, and also that the identification of an IF group or conditional group by the execution of an IFJ operator is invalid. Both of these problems are solved by the introduction of the following two new operators:

RW (I.L. addr, line no.)	Repeat While
RU (I.L. addr, line no.)	Repeat Until

The first parameter in each case is the Intermediate Language address to which a branch is to be made if the value at the top of the stack is TRUE, in the case of a RW operator, or FALSE, in the case of a RU operator. These operators are used in place of the IFJ operator used by the initial implementation in translating repeat-while and repeat-until loops, except that it is not necessary to negate the value of the loop condition in a repeat-until loop before it is tested.

The only remaining form of REPEAT group with no characteristic control operator is the repeat-without-control form. In order to allow the interpreter to recognise all forms of REPEAT groups, a further new operator is introduced:

RWC (line no.)	Repeat Without Control

This operator, which is placed at the beginning of the object code for a repeat-without-control loop, has no special control function, but merely serves to identify the loop for tracing purposes.

On entry to a REPEAT group the top stack location is reserved to be used as the Loop Control Word, and the necessary loop control data is pushed into the stack. The Loop Control Word consists of the following three fields: a type field,

which is initially set to 'number'; a label pointer field which contains a pointer to the symbol table entry for the label of the group if there is one, and zero otherwise; and a field, initially set to zero, which is used to keep the repetition count for the loop. The first time the loop control operator (RF, RFWC, RT, RS, RW, RU or RWC) is executed it changes the type of the Loop Control Word to 'loop conrol word', indicating that the loop has been entered, and tests the flow trace switch. If this switch is on, or if the loop trace switch is on, or if the TRACE ALL or HISTORY facility is active, a call is made to the Debug Output Routine, which outputs a flow trace message indicating entry to the relevant form of repeat group. If the label pointer in the Loop Control Word is non-zero, the name of the label is retrieved and is included in the flow trace message. The loop control operator increments the iteration count by one at the begining of each iteration.

On exit from a REPEAT group a BA (Branch Address) operator is used to unstack the loop control data and branch to the Intermediate Language location following the end of the loop. The BA operator has a new parameter, which is a label pointer. In the case of a single-level exit (ie satisfying the loop termination condition, or execution of an EXIT statement) this parameter has the value zero; and in the case of a multi-level exit (ie execution of an EXIT FROM <label> statement) it contains a pointer to the symbol table entry of the label specified.

When a BA operator is executed, if the parameter is zero the BA interpretive routine merely unstacks the loop control data for the innermost loop. If the parameter is non-zero, the BA interpretive routine procedes to unstack sets of loop control data, starting with that of the innermost loop and working outwards, until it finds a Loop Control Word whose label pointer field matches the value of the label pointer parameter. Finally, the loop control data associated with that Loop Control Word is unstacked. Each time a Loop Control Word is encountered during this unstacking process, the BA interpretive routine tests the flow trace switch and the

loop trace switch. If either of these switches is on, or if the TRACE ALL or HISTORY facility is active, a call is made to the Debug Output Routine, which outputs a flow trace message indicating exit from a loop and the number of iterations which were performed. Again, if the label pointer in the Loop Control Word is non-zero, the name of the label is retrieved and is included in the trace message. (Note that the label pointer field stored in the Loop Control Word is used to control multi-level exits, which were omitted from the initial implementation, as well as for tracing labelled REPEAT groups.)

Example :

Source code :

```
          .
          .
46      TRACE FLOW
47      HUNT : DO REPEAT FOR I FROM I1 TO I2
48                   DO REPEAT FOR J FROM J1 TO J2
49                        UNTRACE FLOW
50                        IF A(I,J) .EQ. MATCH
51                        THEN DO
52                                  TRACE FLOW
53                                  EXIT FROM HUNT
54                             END
55                        TRACE FLOW
56                   END
57              END HUNT
          .
          .
```

Trace output :

```
47(   1) FLOW : ENTER REPEAT FOR GROUP, HUNT
48(   1) FLOW : ENTER REPEAT FOR GROUP
55(   1) FLOW : EXIT REPEAT GROUP AFTER 20 ITERATIONS
48(   2) FLOW : ENTER REPEAT FOR GROUP
55(   2) FLOW : EXIT REPEAT GROUP AFTER 20 ITERATIONS
48(   3) FLOW : ENTER REPEAT FOR GROUP
55(   3) FLOW : EXIT REPEAT GROUP AFTER 20 ITERATIONS
48(   4) FLOW : ENTER REPEAT FOR GROUP
53(   1) FLOW : EXIT REPEAT GROUP AFTER 9 ITERATIONS
53(   1) FLOW : EXIT REPEAT GROUP AFTER 4 ITERATIONS, HUNT
```

4.4.6 Loop Trace

The mechanism used in implementing the loop trace has already been introduced in the section on the flow trace. Whenever a loop control operator is executed, the repetition count for the loop is incremented by one, and the loop trace switch is examined. If this switch is on, or if the TRACE ALL or HISTORY facility is active, a call is made to the Debug Output Routine, which outputs the appropriate loop trace message. This message includes: the new value of the repetition count; the new value of the loop control variable, in the case of a repeat-for, repeat-for-without-test or repeat-set loop; and the value of the loop condition in the case of a repeat-while, repeat-until loop.

Example :

Source code :

```
          .
          .
46    TRACE LOOPS
47    HUNT : DO REPEAT FOR I FROM I1 TO I2
48                 DO REPEAT FOR J FROM J1 TO J2
49                      IF A(I,J) .EQ. MATCH
50                      THEN EXIT FROM HUNT
51                 END
52            END HUNT
          .
          .
```

Trace output :

```
47(   1) FLOW : ENTER REPEAT FOR GROUP, HUNT
47(   1) LOOP :  1, I = 8
48(   1) FLOW : ENTER REPEAT FOR GROUP
48(   1) LOOP :  1, J = 6
48(   1) LOOP :  2, J = 7
48(   1) LOOP :  3, J = 8
48(   1) LOOP :  4, J = 9
49(   4) FLOW : EXIT REPEAT GROUP AFTER 4 ITERATIONS
47(   1) LOOP :  2, I = 9
48(   2) FLOW : ENTER REPEAT FOR GROUP
48(   2) LOOP :  1, J = 6
48(   2) LOOP :  2, J = 7
48(   2) LOOP :  3, J = 8
50(   1) FLOW : EXIT REPEAT GROUP AFTER 3 ITERATIONS
50(   1) FLOW : EXIT REPEAT GROUP AFTER 2 ITERATIONS, HUNT
```

4.4.7 Procedure Trace

When a CP operator is executed, characterising a procedure call, the procedure trace switch in the PAW, (whose symbol table address is a parameter to the CP operator) is tested. If this switch is on, or if the TRACE ALL or HISTORY facility is active a call is made to the Debug Output Routine which outputs the appropriate procedure trace message, indicating the name of the formal parameter and the value of the actual parameter, for each parameter which is not an array. Array parameters are not included in the Procedure trace, since this would lead to the generation of large amounts of unhelpful trace output.

The names of the formal parameters are retrieved by using the procedure template to access their symbol table entries. The first template entries (after the base entry) correspond to the formal parameters, and the number of formal parameters is found in the PAW. The values of the actual parameters are retrieved from the stack.

Example :

Source code :

```
 1  DO
 2       RESERVE A(10),STRIP,TAIL
 3       DEFINE MIRRA ON I,J,S,F1 AS
 4       DO
 5            RESERVE F2
 6            DO IF I .NE. 0
 7            THEN DO
 8                      SET F2 TO A(J+1)
 9                      SET S TO S .CAT. F2
10                      EXECUTE MIRRA(I-1,J+1,S,F2)
11                 END
12                 ELSE SET S TO S .CAT. !*!
13                 SET S TO S .CAT. F1
14       END
15       TRACE PROCEDURES
16       SET STRIP,TAIL TO !!
17       SET A(1) TO !A!
18       SET A(2) TO !B!
19       SET A(3) TO !C!
20       SET A(4) TO !D!
21       EXECUTE MIRRA(4,0,STRIP,TAIL)
          .
          .
```

Trace output :

```
21(   1) FLOW : ENTER PROCEDURE MIRRA
21(   1) PARAMETERS : I = 4, J = 0, S = !!, F1 = !!
10(   1) FLOW : ENTER PROCEDURE MIRRA
10(   1) PARAMETERS : I = 3, J = 1, S = !A!, F1 = !A!
10(   2) FLOW : ENTER PROCEDURE MIRRA
10(   2) PARAMETERS : I = 2, J = 2, S = !AB!, F1 = !B!
10(   3) FLOW : ENTER PROCEDURE MIRRA
10(   3) PARAMETERS : I = 1, J = 3, S = !ABC!, F1 = !C!
10(   4) FLOW : ENTER PROCEDURE MIRRA
10(   4) PARAMETERS : I = 0, J = 4, S = !ABCD!, F1 = !D!
14(   1) FLOW : EXIT PROCEDURE MIRRA
14(   2) FLOW : EXIT PROCEDURE MIRRA
14(   3) FLOW : EXIT PROCEDURE MIRRA
14(   4) FLOW : EXIT PROCEDURE MIRRA
14(   5) FLOW : EXIT PROCEDURE MIRRA
```

4.4.8 Source Trace

Whenever a LN (Line Number) operator is executed, indicating the beginning of the object code for a new line of source text, the source trace switch is examined. If this switch is on, or if the TRACE ALL or HISTORY facility is active, the line number (which is the parameter of the LN operator) is passed to the Debug Output Routine which retrieves the corresponding line of source text and outputs it.

4.4.9 Trace All

When a $TRACE ALL control option is specified, the translator sets the Trace All switch and stores the Trace All Limit, which is the number specified on the control card, with a default value of three. Then, whenever a traceable event occurs, the Trace All facility is considered to be active if the execution count of the statement being executed (given by the value of EXCOUNT) is less than or equal to the value of the Trace All Limit.

A traceable event is any of the events described in sections 4.4.2 to 4.4.8 which result in trace output.

4.4.10 History

If a $HISTORY control option is specified, the translator sets the History switch and stores the value of History Size, which is the number specified on the control card. If the option ALWAYS is specified on the $HISTORY card, the Print History switch is also set. History Size indicates how much space the interpreter should reserve for the history file, which is kept as a circular table on disk. Whenever a traceable event occurs and the History switch is on, the Debug Output Routine writes the appropriate trace message in the next available record in the history table, and begins again from the top of the table once the table is full.

After execution has terminated the Post Mortem routine lists the contents of the history table if the Print History switch is on, or if termination was abnormal. In printing the History, records from the table are listed from the record following the most recent entry to the bottom of the table, then from the top of the table to the most recent entry.

4.4.11 Partial Dump

Three new Intermediate Language operators are used to implement the Partial Dump:

DSV	Dump Simple Variable
DAR	Dump Array Range
TI (on/off)	Trace Inhibit

For each simple variable in a dump list, the translator generates a TA or TLA operator which places the address of the variable at the top of the stack. This is followed by a DSV operator which passes the symbol table address and the type and value of the variable to the Debug Output Routine. The symbol table address, which is used to retrieve the symbolic name of the variable, is the address at the top of the stack in the case of a main variable, and is retrieved from the procedure template in the case of a local variable.

For each array range in a dump list, the translator generates object code to push the address of the array-word and the values of the subscript bound expressions into the stack. This is followed by a DAR operator which retrieves these values and steps through the specified range, passing each value in turn to the Debug Output Routine to be printed. A single array element is compiled as an array range with equal bounds.

The TI operator is used to turn the Trace Inhibit switch on and off at the beginning and end, respectively, of the code for a partial dump. This switch is set during the execution of a partial dump in order to inhibit any tracing which might occur, in particular: fetches, function calls, and tests in conditional dump lists.

An example of a partial dump request and its output is given in Fig 4.5.

4.4.12 Snap Dump

A DUMP statement with no dump list is translated as a single Intermediate Language operator, namely:

DUMP	Snap Dump

Source code :

line no.

```
  15          DUMP S,INRANGE, IF INRANGE THEN( I1,I2,A(I1 TO I2)),C
```


Output :

```
15(    1) DUMP : S = !ACKERMANNS FUNCTION!    INRANGE = TRUE    I1 = 3    I2 = 19
15(    1) DUMP : A(3) = 61    A(4) = 125    A(5) = 253    A(6) = 509    A(7) = 1021
15(    1) DUMP : A(8) = 2045    A(9) = 4093    A(10) = 8189    A(11) = 16381    A(12) = 32765
15(    1) DUMP : A(13) = 65533    A(14) 131069    A(15) = 262141    A(16) = 524285
15(    1) DUMP : A(17) = 1048513    A(18) = 2097149    A(19) = 4194301    C = UNDEF
```

Fig 4.5 Sample Output from Partial Dump

The Snap Dump includes all data in the current environment, that is the current procedure (if the DUMP statement appears in a procedure) and the main program, except that in the case of an array, only the first five and the last elements from the first five and the last rows are printed. The SD operator passes the symbol table address and the type and value of each data item to the Debug Output Routine, to be printed. The templates for the current procedure and main program are used to locate the symbol table entries and the storage locations of the data to be dumped.

4.4.13 Post Mortem Dump

Two switches are used to control the Post Mortem Dump facility: the PM Dump switch, which is set by the translator if a $PM Dump control option is selected; and the Termination Mode switch, which is set to 'abnormal' prior to the commencement of execution and is set to 'normal' when an END operator is executed. After printing the program listing, with execution counts added, the Post Mortem Routine examines these two switches and if the PM Dump switch is on, or if the Termination Mode switch is set to 'abnormal', a Post Mortem Dump is given.

Data to be printed in a Post Mortem Dump is retrieved in the same way as for a Snap Dump, except that local data for all active procedures is printed. The link data in the Return Address Word for each procedure is used to access local data for the procedure which called it, via its Return Address Word.

4.4.14 ASSERT Statement

A new Intermediate Language operator is used to implement the ASSERT statement:

IFT	If False Terminate

In translating an ASSERT statement, the translator generates object code to evaluate the asserted condition, followed by an IFT operator. The IFT operator tests the value at the top of the stack and if it is false an error

message is printed, indicating failure in an ASSERT statement, and execution is terminated. If the value at the top of the stack is true the Flow Trace switch is tested, and if it is on, or if the TRACE ALL or HISTORY facility is active, a call is made to the Debug Output Routine, which prints the appropriate flow trace message.

Further examples of MUSSEL programs and their debugging output, including dumps, HISTORY, and TRACE ALL, are given in Appendix C.

Chapter 5

SUMMARY AND CONCLUSIONS

In the Introduction to this thesis, computer programming was likened to a game in which the programmer performed rather poorly. This was done in order to informally define several terms and to identify some of the problems facing the programmer. The rest of the thesis has been concerned with the solution of these problems.

Chapter 1 introduces the subject of Program Pathology by investigating the various causes of program malfunction. The different types of program errors are discussed, and methods for detecting and diagnosing errors during compilation and execution are described. In attempting to develop a general approach to debugging it is found that the most difficult bugs to locate are those concerned with flow of control, and thus with program structure. Dijkstra's structural model of programming is extended to give a model for debugging in terms of program structure.

In Chapter 2 a wide range of debugging tools is discussed from the point of view of how they can help the programmer to debug his programs. It is noted that no debugging facilities can cure poor programming, and that much of the value of a debugging tool depends on the manner in which it is used. The notion of language-independent debugging facilities, which has been pursued by several authors, is found to lead to machine-dependent, low-level debugging facilities which are of little or no use to the high-level language programmer. The design of debugging facilities for a high-level language, therefore, must be language-dependent, though the design approach should be language-independent.

Implementation of debugging facilities for high-level languages presents two major problems : how to give control to the debugging routine at the required times; and, how to give the debugging routine access to the necessary symbolic and structural information about the source program. Four basic methods of solving these problems are discussed with

reference to systems which have used them. Each method has both advantages and disadvantages. In particular, an interpreter offers excellent diagnostic and debugging capabilities, as well as quick turnaround for debugging runs, but execution speed is too slow to be used for production runs. A scheme is presented whereby a single programming system can offer the advantages of interpretive execution for debugging runs, as well as efficient execution for production runs.

The provision of debugging facilities for a particular language, namely MUSSEL, is described in detail in Chapter 4. The design of the debugging facilities and their implementation is developed in a manner that is machine-independent, and which could be used in implementing debugging facilities for other high-level languages. Though intended for use in a batch processing environment, these debugging facilities are designed to be easily adaptable to use in an interactive environment.

Certain features of MUSSEL make the task of providing debugging facilities easier than it would be for most other high level languages; in particular, the absence of the GO TO statement, and the simple two-level structure imposed by disallowing nested procedure declarations and blocks. That these language features greatly simplify the tracing of control flow and data flow, respectively, suggests that they should also make programs written in the language easier to understand. It appears then that these are good language features, since they also have the property that they do not seriously restrict the class of algorithms which can be expressed in the language.

Appendix A

The Syntax of MUSSEL in BNF

```
<program> ::= <group begin><eol><reserve><eol>
              <procedure defn part><eol><statements><eol>
              <group end>

<group begin> ::= DO|<label> : DO

<group end> ::= END|END <label>

<reserve> ::= RESERVE <loclist>|RES <loclist>

<loclist> ::= <loclist elt>|<loclist>,<loclist elt>

<loclist elt> ::= <simple variable>|<array spec>|
              <multiple array spec>

<simple variable> ::= <identifier>

<array spec> ::= <array id>(<bound pair list>)

<bound pair list> ::= <bound pair>|<bound pair>,<bound pair>

<bound pair> ::= <valexp> : <valexp>

<array id> ::= <identifier>

<multiple array spec> ::= (<array segment>)

<array segment> ::= <array list>,<array spec>

<array list> ::= <array id>|<array list>,<array id>

<procedure defn part> ::= <procedure defns>|<empty>

<procedure defns> ::= <procedure defn>|<procedure defns><eol>
              <procedure defn>

<procedure defn> ::= <procedure head><eol><reserve part>
              <statements><eol><group end>

<reserve part> ::= <reserve><eol>|<empty>

<procedure head> ::= DEFINE <procedure id> AS|
              DEFINE <procedure id> ON <parameter list> AS

<procedure id> ::= <identifier>

<parameter list> ::= <parameter id>|
              <parameter list>,<parameter id>
```

```
<parameter id> ::= <identifier>

<statements> ::= <statement>|<statements><eol><statement>

<statement> ::= <simple statement>|<group statement>

<simple statement> ::= <assign>|<procedure execute>|
          <function value assign>|<exit>|<assert>|<trace>|
          <untrace>|<dump>|<read>|<print>

<group statement> ::= <simple group>|<if group>|
          <conditional group>|<choice group>|
          <case group>|<repeat group>

<simple group> ::= <group begin><eol><statements><eol>
          <group end>

<if group> ::= IF <condexp> THEN <simple statement>|
          IF <condexp><eol> THEN <statement>

<conditional group> ::= <group begin> IF <condexp><eol>
          THEN <statement><group end>|<group begin> IF
          <condexp><eol> THEN <statement><eol> ELSE
          <statement><eol><group end>

<choice group> ::=<group begin> CHOICE OF <eol><group of ifs>
          <eol><choice end>

<group of ifs> ::= <if group>|<group of ifs><eol><if group>

<choice end> ::= <group end>|ELSE <statement><eol><group end>

<case group> ::= <case head><eol><statements><eol><group end>

<case head> ::= <group begin> CASE <valexp> OF|
          <group begin>CASE <valexp> IN (<integer>,
          <integer>) OF

<repeat group> ::= <group begin> REPEAT <eol><statements>
          <eol><group end>|<group begin> REPEAT <control>
          <eol><statements><eol><group end>

<control> ::= UNTIL <condexp>|WHILE <condexp>|<valexp> TIMES|
          FOR <variable> FROM <valexp><for end>|
          FOR <variable> SET TO <exp list>

<for end> ::= TO <valexp> BY <valexp>|BY <valexp>|
          TO <valexp>|<empty>

<assign> ::= SET <list part> TO <expression>
```

```
<list part> ::= <variable>|<list part>,<variable>

<variable> ::= <simple variable>|<array elt>|<array id>

<array elt> ::= <array id>(<valexp>)|
          <array id>(<valexp>,<valexp>)

<procedure execute> ::= EXECUTE <procedure id>|
          EXECUTE <procedure id>(<exp list>)

<exp list> ::= <expression>|<exp list>,<expression>

<function value assign> ::= VALUE IS <expression>

<exit> ::= EXIT|EXIT FROM <label>

<assert> ::= ASSERT <cond exp>

<trace> ::= TRACE|TRACE <trace option list>

<untrace> ::= UNTRACE|UNTRACE <trace option list>

<trace option list> ::= <trace option>|
          <trace option list>,<trace option>

<trace option> ::= STORES|STORES(<var list>)|
          VALUES|VALUES(<val list>)|
          TYPES|TYPES(<var list>)|
          FLOW|LOOPS|SOURCE|
          PROCEDURES|PROCEDURES(<proc list>)

<proc list> ::= <procedure id>|<proc list>,<procedure id>

<var list> ::= <var list elt>|<var list>,<var list elt>

<val list> ::= <val list elt>|<val list>,<val list elt>

<var list elt> ::= <simple variable>|<array id>

<val list elt> ::= <var list elt>|<function id>

<dump> ::= DUMP|DUMP <dump list>

<dump list> ::= <dump list elt>|<dump list>,<dump list elt>

<dump list elt> ::= <dump list item>|
          IF <cond exp> THEN <dump list item>

<dump list item> ::= <simple variable>|<array range>|
          (<dump list>)

<array range> ::= <array id>(<sub range>)|
          <array id>(<sub range>,<sub range>)

<sub range> ::= <valexp>|<valexp> TO <valexp>
```

```
<read> ::= READ <list part>

<print> ::= PRINT <print list>

<print list> ::= <print elt>|<print list>,<print elt>

<print elt> ::= <expression><picture control>|
          (<expression>,<print elt>)|<print control>

<picture control> ::= (PIC = <picture>)|<empty>

<picture> ::= <alpha picture>|<numeric picture>

<alpha picture> ::= <first alpha>|<alpha picture><first alpha>|
          <alpha picture>,<first alpha>|
          <alpha picture>(<integer>)

<first alpha> ::= *|B

<numeric picture> ::= <sign><integer part><decimal part>
          <exponent part><sign>

<sign> ::= -|S|<empty>

<integer part> ::= <firstnum>|<integer part><firstnum>|
          <integer part>(<integer>)|<empty>

<decimal part> ::= .<integer part>|<empty>

<exponent part> ::= E<sign>**|E<sign>99|<empty>

<firstnum> ::= *|9|B

<print control> ::= NEWPAGE|NEWLINE|NEWLINE(<valexp>)|
          SPACE(<valexp>)|TAB(<valexp>)

<expression> ::= <valexp>|<condexp>|<stringexp>

<valexp> ::= <term>|<adding op><term>|<valexp><adding op><term>

<term> ::= <factor>|<term><mult op><factor>

<factor> ::= <primary>|<factor>**<primary>

<primary> ::= <unsigned number>|<variable>|
          <function call>|(<valexp>)

<adding op> ::= +|-

<mult op> ::= *|/|./.

<condexp> ::= <logical term>|<condexp> .OR. <logical term>
```

```
<logical term> ::= <logical factor>|
          <logical term> .AND. <logical factor>

<logical factor> ::= <logical primary> .NOT. <logical primary>

<logical primary> ::= <logical value>|<variable>|
          <function call>|<relation>|(<condexp>)

<relation> ::= <valexp><relational op><valexp>|
          <stringexp><relational op><stringexp>

<stringexp>  ::= <string term>|<stringexp> .CAT. <string term>

<string term> ::= <string primary>|
          <string primary> .UPTO. <string primary>|
          <string primary> .AFTER. <string primary>

<string primary> ::= <variable>|<string>|
          <function call>|(<stringexp>)

<function call> ::= <function id>|<function id>(<exp list>)

<function id> ::= <identifier>|<library function>

<library function> ::= LOG|EXP|ABS|SIN|COS|SQRT|STRING|
          NUMBER|INTEGER|CONDITION|LENGTH|POSITION|
          SUBSTRING|COUNT|MAXIMUM|MINIMUM

<label> ::= <identifier>

<identifier> ::= <letter>|<identifier><letter>|
          <identifier><digit>

<number> ::= <unsigned number>|<adding op><unsigned number>

<unsigned number> ::= <decimal number><exponent>|
          <decimal number>

<exponent> ::= E <integer>

<decimal number> ::= <unsigned number>|<decimal fraction>|
          <unsigned number><decimal fraction>

<decimal fraction> ::= .<unsigned integer>

<integer> ::= <unsigned integer>|<adding op><unsigned integer>

<unsigned integer> ::= <digit>|<unsigned integer><digit>

<logical value> ::= TRUE|FALSE|T|F

<string> ::= !<proper string>!
```

<proper string> ::= <string character>|
<proper string><string character>

<string character> ::= <letter>|<digit>|<special character>|
<quote>|<empty>

<letter>::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|
U|V|W|X|Y|Z

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<special character> ::= :|.|*|+|-|/|$|@| |(|)

<quote> ::= !!

<empty> ::=

<eol> ::= End of line, not represented externally.

Appendix B

Explanation of the Intermediate Language of MUSSEL

General

The beginning of object code for each source line is indicated by an LN op (ie operator), and the beginning of object code for each basic block is indicated by a BBB op. Where the beginning of a line coincides with the beginning of a basic block, the LN op precedes the BBB op. The end of the object code is indicated by an END op.

LN(line no.) — Line Number

Parameter is the line number of the corresponding source line. Function of op is to load this value into the line number register (of the interpretive pseudo-machine). This allows the interpreter to keep track of the current position in the source program. The line number is used by the Debug Output Routine (DOR) to retrieve lines of source for the source trace.

BBB(block no.) — Begin Basic Block

Parameter is the number of the corresponding basic block. Function of op is to increment the execution count for that basic block and load its new value into the execution count register. This value is printed along with the line number with all diagnostic and debugging messages, and is accessible to the user program via the reserved identifier EXCOUNT.

END — End of Object Program

Op sets the Termination Mode switch (TM) to 'normal' and passes control to the Post Mortem Routine (PMR).

Expressions

Expressions are translated into Reverse Polish form using

the following five sets of ops.

(1) Constants

The following ops are used to place constants at the top of the stack.

TN(number) — Take Number

Op places the number which is its parameter at the top of the stack, along with a 'number' type field.

TN0 — Take Number Zero

TN1 — Take Number One

These ops place the values 0 and 1, respectively, at the top of the stack, along with a 'number' type field.

TB(Boolean) — Take Boolean

Op places the Boolean value which is its paprameter at the top of the stack, along with a 'Boolean' type field.

TS(string) — Take String

Op places the (variable length) string which is its parameter in the string area, and places its address at the top of the stack, along with a 'string address' type field.

TNS — Take Null String

Op places a null string at the top of the stack, along with a 'null string' type field.

(2) Main Variables

TR(ST addr) — Take Result

The value and type of the main variable, whose symbol-table (ST) address is the parameter, are placed at the top of the stack. If fetch trace is

required the ST address and the stacked value and type are passed to the DOR.

TA(ST addr) Take Address

Parameter is the ST address of a main variable or array-word. This address is placed at the top of the stack, along with a 'symbol-table address' type field.

(3) Local Variable

TLR(local addr) Take Local Result

The value and type of the local variable, whose relative stack address is the parameter, are placed at the top of the stack. In the case of a formal parameter, the stack unit addressed may itself be an address. In this case op chains through such addresses until the value is found. If fetch trace is required the ST address for the local variable is retrieved from the procedure template and passed to the DOR, along with the stacked value and type.

TLA(local addr) Take Local Address

Parameter is the relative stack address of a local variable or array-word. This address is placed at the top of the stack, along with a 'stack address' type field.

(4) Subscripted Variables

When the value or address of an array element is required, the translator compiles code to stack the address of the array word (using a TA or TLA op), and the value of each subscript expression. This is followed by an INDR or INDA op, which calculates the address of the array element using the stacked values and the storage mapping function for the array, and places either the value at that address, or the address itself, at the top of the stack.

INDR — Index Result

Op uses stacked array-word address and subscript values, wipes them from the stack, and places the value and type of the addressed array element at the top of the stack. If fetch trace is required the ST address of the array-word is retrieved, from the stack in the case of a main array and from the procedure template in the case of a local array. This ST address is passed to the DOR, along with the values of the subscripts, and the value and type of the addressed array element.

INDA — Index Address

Op uses the stacked array-word address and subscript values, and places the address of the array element at the top of the stack, along with an 'array address' type field. The array-word address and subscript values remain in the stack so that the array element can be identified, if required in a diagnostic message or store trace message.

(5) Arithmetic, Logical and String Operations

OP(op-code) — Operator

Parameter specifies one of the following operators : .OR.,.AND.,.NOT.,.GT.,.GE.,.LT.,.LE.,.EQ.,.NE.,+,-, *,/,./.,.NEG.,**,.CAT.,.UPTO.,.AFTER.. In the case of a unary op, the type of the top stack unit is checked, and if it is correct the appropriate operator is applied. The result replaces the top stack unit. In the case of a binary op, the types of the top two stack units are checked, and if they are correct the appropriate operator is applied. The top two stack units are wiped, and the result placed at the top of the stack, along with the appropriate type field.
If an error is detected in the types or values of the operand(s) or the result, the value(s) and

type(s) of the operand(s), and the operation specified, are passed to the DOR.

Assignments

To make an assignment, the translator compiles code which places the address to be stored into in the second-to-top stack unit, and the value and type to be stored in the top stack unit. In the case of assigning to an array element, the array-word address and the subscript expression values are also placed in the stack. The assignment is performed by an ST or STA op. If store trace is required the necessary ST address is obtained as follows: for a main variable the ST address is in the stack; for a main array element, the ST address of the array-word is in the stack; for a local variable or array element, the ST address is retrieved from the procedure template.

ST — Store

After performing the store, op wipes the value and address from the stack. In the case of a store to an array element, the subscript value and the array-word address are also wiped from the stack.

STA — Store Also

Used in translating multiple assignments. After performing the store, stack units used are wiped from the stack as for the ST op, except that the stored value and type are left at the top of the stack.

Conditional Groups, IF Groups

In translating conditional groups and IF groups, the translator compiles code to evaluate the conditional expression and leave its value at the top of the stack. This is followed by an IFJ op, which in turn is followed by the code for the THEN clause. Where appropriate, this is followed by a UJ op and the code for the ELSE clause.

IFJ(IL addr) — If False Jump

If flow trace is required the value at the top of the stack is passed to the DOR. Then, if the value is false, a branch is made to the Intermediate Language (IL) address specified by the parameter. The top stack unit is then wiped.

UJ(IL addr) — Unconditional Jump

A branch is made to the IL address specified by the parameter.

ASSERT Statement

In translating an ASSERT statement, the translator compiles code to evaluate the asserted condition and leave its value at the top of the stack. This is followed by an IFT op.

IFT — If False Terminate

If flow trace is required the value at the top of the stack is passed to the DOR. Then, if the value is false, the IFT op sets the Termination Mode switch (TM) to 'abnormal' and passes control to the Post Mortem Routine (PMR); otherwise the IFT op merely unstacks the top stack unit.

CHOICE Group

The CHOICE group is implemented as a series of IF groups. These are preceded by two UJ ops and a CH op. The first UJ op merely jumps over the second one, which is used to exit from the CHOICE group. The THEN clause of each IF group is followed by a UJ op which jumps to this UJ op.

CH — Choice

Op is only used to indicate entry to a CHOICE group. If flow trace is required a call is made to the DOR. The option selected is shown from the trace of IFJ ops.

CASE Group

Translator compiles code to evaluate the case expression and, where appropriate, the values of the case bounds and leave these values in the stack. This is followed by the code for the case statements, preceded by a UJ op to jump over it. A CS or CSWB op is then compiled, followed by a list of UJ ops, one for each case statement.

CS(no. of case stmts) Case

Op uses values of case expression and case bounds to calculate the case index, and check that it is in the correct range. If flow trace is required the value of the case expression and the case index are passed to the DOR. The case index is then used to select the appropriate UJ op and branch to the corresponding case statement.

CSWB(no. of case stmts) Case Without Bounds

As for CS op, except that a lower bound of 1 is assumed. If flow trace is required the value of the case expression is passed to the DOR. The case index is not needed as it is identical to the case expression.

Repeat Groups

All repeat groups are compiled in a similar form consisting of the following four sections: a loop prologue, which is executed on entry to the loop and contains the mechanism for loop exit; loop control, which is executed at the begining of each iteration; the body of the loop; and finally, a UJ op which is used to branch to the loop control section at the end of each iteration.

The loop prologue section consists of: a UJ op, a BA op, a TN0 or TN op, and code which pushes the required loop control units into the stack. The UJ merely jumps over the BA op, which in turn is used to exit from the loop. The TN0

or TN op is used to reserve a stack location for the Loop Control Word. This is a stack unit consisting of three fields: a type field, set initially to 'number'; a label pointer field, which contains a pointer to the ST entry for the label of the repeat group if there is one, and zero otherwise; and a repetition count field, which is initially set to zero. Thus if the repeat group has no label, a TN0 op is used, otherwise, a TN op is used whose parameter will set the label field appropriately.

In most cases the loop control section consists of a single loop control op (RWC, RT, RF, RFWT, or RS), however, in the case of a repeat-while or repeat-until loop, the loop control op (RW or RU) is preceded by code which evaluates the loop condition and stores its value at the top of the stack. On entry to the loop, the loop control op checks the types of the loop control units (if there are any), and changes the type of the Loop Control Word from 'number' to 'loop control word'. This type serves to indicate, on later iterations, that the loop has already been entered, and is used by the BA op in unstacking the loop control data when the loop is exited. If flow trace is required the loop control op then calls the DOR, which issues a message indicating entry to the appropriate form of repeat group.

Next (or first, on subsequent iterations), the loop control op sets the line number register to the line number of the DO REPEAT statement, which is given by the line no. parameter of the loop control op. The loop control op then performs whatever loop control functions are required, which may (except for a RWC op or RFWT op) cause termination of the loop. Then, if loop trace is required the loop control op calls the DOR, passing the repetition count and, where appropriate, the value of the conrolled variable or the value of the loop condition. Finally the loop control op increments the repetition count in preparation for the next iteration. The specific loop control functions performed are given below.

BA(IL addr, label ptr) Branch Address

If the label pointer parameter is zero, op merely unstacks the top-most set of loop control data (ie loop control units and Loop Control Word, as stacked by the loop prologue section). If this parameter is non-zero, op proceeds to unstack successive sets of loop control data until it finds a Loop Control Word whose label pointer field matches the value of the label pointer parameter. Finally, the loop control data associated with that Loop Control Word is unstacked. If flow trace is required, then each time a Loop Control Word is encountered during this unstacking process, a call is made to the DOR which issues a message indicating exit from a repeat group. This message includes the number of iterations performed and the label of the repeat group if there is one.

Loop Control Ops

RWC(line no.) Repeat Without Control

No special function. No control units used.

RT(BA addr, line no.) Repeat Times

Control unit is the value of the times expression. Its type and value are checked on entry to ensure it is an integer. On each iteration, if the value of the control unit is positive, op decrements that value by one. Otherwise, a branch is made to the BA op for the loop, whose address is given by the first parameter.

RF(BA addr, line no.) Repeat For

Control units are: the address of the controlled variable, the starting value, the test value, and the increment value (assumed to be 1 if omitted). On entry op checks the types of the four control

units, and initializes the controlled variable.
On subsequent iterations the stacked values are used to increment and test the controlled variable.
If, after being incremented, the value of the controlled variable is greater than the test value, a branch is made to the BA op for the loop, whose address is given by the first parameter.
Note: a repeat-for group is always executed at least once.

RFWT(line no.) Repeat For Without Test

Similar to the RF op, except that there is no test value in the stack, and no branch out address is required.

RS(no. set elts, BA addr, line no.) Repeat Set

Control units are: the address of the controlled variable, and values of the set list elements.
On each iteration, while the iteration count, after incrementing, is not greater than the first parameter, the value of the iteration count is used to index the list of set list values and assign the appropriate value to the controlled variable.

RW(BA addr, line no.) Repeat While

Op tests the value of the loop condition, which has been stored at the top of the stack. If this value is false a branch is made to the BA op for the loop, whose address is given by the first parameter.

RU(BA addr, line no.) Repeat Until

As for RW op, but branch to BA op if value at top of stack is true, rather than when it is false.

Procedure and Function Declarations

The following ops are used in translating procedure and function declarations.

PE(no. FPs, no. LIs) Procedure Entry

Inserted at the beginning of object code corresponding to procedure or function declaration. Parameters are number of formal parameters, and number of local identifiers, respectively. These are used to check the number of actual parameters passed, and to reserve space in the stack for local storage.

MSF(local addr, no. AIs, dim) Make Storage Function

Used in the case of local arrays. Parameters are: local address of last array word; number of array identifiers in the segment; and the dimension of the arrays. Each array segment is translated into code to evaluate the subscript bound expressions, followed by an MSF op. Op uses stacked subscript bound expression values to set up a storage mapping function, which replaces the used values on the stack. Space for the array elements is then reserved in the stack.

VI Value Is

Used to store the value of a function. The value and type at the top of the stack are stored in the stack location reserved immediately below the Return Address Word for the function. If fetch trace is required the ST address for the function is obtained from the Return Address Word in the stack. Finally, the top stack unit is wiped.

RE — Return

Used to return from a procedure or function. If flow trace is required the ST address of the Procedure Address Word is obtained from the Return Address Word in the stack. Op wipes all stack units above and including the Return Address Word, and makes a branch to the specified return address.

Procedure and Function Calls

In translating a function or procedure call, the translator compiles an RAF or RAP op, which reserves space in the stack for the Return Address Word, followed by object code which places the addresses or values of the actual parameters on the stack. This is followed by a CP op which enters the function or procedure.

RAP — Return Address Procedure

Reserves space in the stack for procedure return address.

RAF — Return Address Function

Reserves space in the stack for function return address.

CP(ST addr, no. APs) — Call Procedure

Op places return address in the stack unit reserved for it by the RAP or RAF op, and stores the number of actual parameters, given by the second parameter, to be checked against the number of formal parameters by the PE op. If procedure trace is required op passes the ST address of each formal parameter which is not an array name, and the values of the corresponding actual parameters, to the DOR, along with the ST address of the Procedure Address Word (PAW). Op then branches to the beginning of object code for the procedure, the address of which is found in the PAW, whose address is the first parameter.

Input

MUSSEL has a simple unformatted stream input; data being separated on the data cards by blanks or commas. Strings have the additional delimiter '!'. Arrays may be read in row-wise by specifying only the array name. For each element in the read list, the translator compiles code to place the corresponding address at the top of the stack, followed by an RD op.

RD — Read

If the address at the top of the stack is the address of a variable or array element, the next value from the card input buffer is stored at that address along with its type. If store trace is required the trace message is produced as for an ST op.
If the address at the top of the stack is the address of an array-word, data is read from the card input buffer and stored in the array, row-wise, until it is filled.

Output

MUSSEL provides both a simple fixed-format output, and a more flexible picture controlled output. The two forms can be mixed in the same PRINT statement. For each print list element without a PIC specification, code is compiled which places the value the the expression at the top of the stack, followed by a PR op. For picture controlled output, code is compiled to stack the values of all items controlled by the PIC specification, followed by a PRC op.

PR — Print

The value at the top of the stack is printed in the appropriate format.

PRC(no. list elts, PIC spec.) Print Control

First parameter gives the number of values which have been placed in the stack. These values are printed according to the PIC specification, which is given by the second parameter.

TRACE and UNTRACE Statements

Eleven new ops are used in translating TRACE and UNTRACE statement, each of which turns on or off one or more trace switches. In each case the first parameter of the op is 'on' for a TRACE statement, and 'off' for an UNTRACE statement.

The following four ops are used for trace options in which a trace-list is used. The second parameter in each case is the symbol table address of the corresponding list element.

TSS(on/off, ST addr) Trace Single Stores

Op turns on or off the Store Trace switch in the addressed ST entry.

TSV(on/off, ST addr) Trace Single Values

Op turns on or off both the Store Trace and Fetch Trace switches in the addressed ST entry, or the Fetch Trace switch only in the case of a PAW for a function.

TST(on/off, ST addr) Trace Single Types

Op turns on or off the Type Trace switch in the addressed ST entry.

TSP(on/off, ST entry) Trace Single Procedures

Op turns on or off the Procedure Trace switch in the addressed symbol table entry.

The following seven ops are used for trace options in which no trace-list is used.

TGS(on/off) Trace General Stores

Op turns on or off the Store Trace switch in every ST entry corresponding to a variable or array name.

TGV(on/off) Trace General Values

Op turns on or off both the Store Trace switch and the Fetch Trace switch in every ST entry corresponding to a variable or array name, and the Fetch Trace switch in every ST entry which is the PAW for a function.

TGT(on/off) Trace General Types

Op turns on or off the Type Trace switch in every ST entry corresponding to a variable or array name.

TGP(on/off) Trace General Procedures

Op turns on or off the Procedure Trace switch in every ST entry which is a Procedure Address Word.

TGF(on/off) Trace General Flow

Op turns on or off the global Flow Trace switch.

TGL(on/off) Trace General Loops

Op turns on or off the global Loop Trace switch.

TGX(on/off) Trace General Source

Op turns on or off the global Source Trace switch.

If no trace option is given, then one of each of the last six ops is compiled.

DUMP Statement

Four new ops are used in translating DUMP statements. DUMP is used for snap dumps, while DSV, DAR and TI are used for partial dumps.

A DUMP statement with no dump-list is a request for a snap dump, and is compiled as a single DUMP op.

If a dump-list is given, then for each simple variable in the dump-list a TA or TLA op is compiled to place the address of the variable at the top of the stack, followed by a DSV op. For each array range in the dump-list, code is compiled to stack the address of the array-word and the values of the subscript bound expressions, followed by a DAR op. A single array element is treated as an array range with equal bounds. For each conditional dump element the translator compiles code to evaluate the conditional expression, followed by an IFJ op which tests the value of the conditional expression and jumps over the code for the THEN clause if it is false.

The TI op is used to turn the Trace Inhibit switch on and off at the beginning and end, respectively, of the code for a partial dump. This switch is set during the execution of a partial dump in order to inhibit any tracing which might occur from the use of fetches, function calls, and tests in conditional dump lists.

DUMP — Snap Dump

Op uses the templates of the main program and procedures to give a dump of all variables in the main program and all procedures which have been entered but not exited.

DSV — Dump Simple Variable

The value and type of the variable whose address is at the top of the stack are passed to the DOR along with the ST address of the variable. The ST address is obtained from the procedure template in the case of a local variable, otherwise it is the stacked address.

DAR — Dump Array Range

Op retrieves the subscript bound values and the array-word address from the stack. If the array-word address is a 'stack address', the ST address for the array is obtained from the procedure template, otherwise it is the stacked array-word address. Op then steps through the specified range of elements, and for each element passes the ST address, the subscript values and the value and type of the array element to the DOR. Where ranges are specified on both dimensions of a two-dimensional array, elements are dumped in row-wise order.

TI(on/off) — Trace Inhibit

Op turns on or off the Trace Inhibit switch, according to the value of its parameter. Tracing will only occur while this switch is turned off.

Appendix C

Examples of MUSSEL Programs

with

Debugging Output

This Appendix contains two versions of a program for Hoare's QUICKSORT algorithm, written in MUSSEL and demonstrating the use of the MUSSEL debugging facilities. The first version contains an error which was (unintentionally) introduced in translating the published algorithm into MUSSEL ; this error has been corrected in the second version. The output shown is that which would be produced by an implementation of the MUSSEL system as described in Chapter 4.

```
CONTROL CARDS :
$MUSSEL   GROVES, L.J.
$DEBUG
$TRACE ALL 1
$HISTORY  30


COMPILATION :
    NO ERRORS DETECTED
    76 STATEMENTS PROCESSED
```

CROSS-REFERENCE TABLE (PAGE 1)

NAME	TYPE	OCCURENCES
DATA	MAIN ARRAY	7,72,73,74,75
EXCHANGE	PROCEDURE	11,18,38,45,50
A	PARAMETER	11,14,14,15,16,16,17
I	PARAMETER	11,14,14,15,16
J	PARAMETER	11,14,14,16,17
TEMP	LOCAL VARIABLE	13,15,17
PARTITION	PROCEDURE	22,54,63
A	PARAMETER	22,26,30,34,38,45,50
M	PARAMETER	22,25,27,36
N	PARAMETER	22,25,28,32
I	PARAMETER	22,27,29,30,31,31,32,38,39,39,43,45,46,46
J	PARAMETER	22,28,29,34,35,35,36,38,40,40,48,50,51,51
F	LOCAL VARIABLE	24,25,26,43,45,48,50
X	LOCAL VARIABLE	24,26,30,34
PART	LABEL	29,41
POST	LABEL	42,53
QUICKSORT	PROCEDURE	58,65,66,68,74
A	PARAMETER	58,63,65,66
M	PARAMETER	58,61,63,64,65
N	PARAMETER	58,61,63,64,66
I	LOCAL VARIABLE	60,63,64,64,66
J	LOCAL VARIABLE	60,63,64,64,65

EXECUTION :

```
UNSORTED DATA
   64          7          12          6          18          25          12          97          37          43
   74(     1) FLOW : ENTER PROCEDURE QUICKSORT
   74(     1) PARAMETERS : M = 1, N = 10
   61(     1) FETCH : M = 1
   61(     1) FETCH : N = 10
   61(     1) FLOW : IF CONDITION IS TRUE
   63(     1) FLOW : ENTER PROCEDURE PARTITION
   63(     1) PARAMETERS : M = 1, N = 10, I = UNDEF, J = UNDEF
   25(     1) FETCH : M = 1
   25(     1) FETCH : N = 10
   25(     1) STORE : F = 5
   25(     1) TYPE : UNDEF TO NUMBER, F = 5
   26(     1) FETCH : F = 5
   26(     1) FETCH : A(5) = 18
   26(     1) STORE : X = 18
   26(     1) TYPE : UNDEF TO NUMBER, X = 18
   27(     1) FETCH : M = 1
   27(     1) STORE : I = 1
   27(     1) TYPE : UNDEF TO NUMBER, I = 1
   28(     1) FETCH : N = 10
   28(     1) STORE : J = 10
   28(     1) TYPE : UNDEF TO NUMBER, J = 10
   29(     1) FLOW : ENTER REPEAT WHILE GROUP, PART
   29(     1) FETCH : I = 1
   29(     1) FETCH : J = 10
   29(     1) LOOP :  1, WHILE CONDITION IS TRUE
   30(     1) FLOW : ENTER REPEAT WHILE GROUP
   30(     1) FETCH : X = 18
   30(     1) FETCH : A(1) = 64
   30(     1) LOOP :  1, WHILE CONDITION IS FALSE
   30(     1) FLOW : EXIT REPEAT WHILE GROUP AFTER 0 ITERATIONS
   34(     1) FLOW : ENTER REPEAT WHILE GROUP
   34(     1) FETCH : X = 18
   34(     1) FETCH : J = 10
```

```
34(     1) FETCH : A(10) = 43
34(     1) LOOP :  1, WHILE CONDITION IS TRUE
35(     1) FETCH : J = 10
35(     1) STORE : J = 9
36(     1) FETCH : J = 9
36(     1) FETCH : M = 1
36(     1) FLOW : IF CONDITION IS FALSE
34(     1) FETCH : X = 18
34(     1) FETCH : J = 9
34(     1) FETCH : A(9) = 37
34(     1) LOOP :  2, WHILE CONDITION IS TRUE
34(     1) FETCH : X = 18
34(     1) FETCH : J = 8
34(     1) FETCH : A(8) = 97
34(     1) LOOP :  3, WHILE CONDITION IS TRUE
34(     1) FETCH : X = 18
34(     1) FETCH : J = 7
34(     1) FETCH : A(7) = 12
34(     1) LOOP :  4, WHILE CONDITION IS FALSE
34(     1) FLOW : EXIT REPEAT WHILE GROUP AFTER 3 ITERATIONS
38(     1) FLOW : ENTER PROCEDURE EXCHANGE
38(     1) PARAMETERS : I = 1, J = M
14(     1) FETCH : I = 1
14(     1) FETCH : J = 7
14(     1) FETCH : I = 1
14(     1) FETCH : A(1) = 64
14(     1) FETCH : J = 7
14(     1) FETCH : A(7) = 12
14(     1) FLOW : ASSERT CONDITION IS TRUE
15(     1) FETCH : I = 1
15(     1) FETCH : A(1) = 64
15(     1) STORE : TEMP = 64
15(     1) TYPE : UNDEF TO NUMBER, TEMP = 64
16(     1) FETCH : I = 1
16(     1) FETCH : J = 7
16(     1) FETCH : A(7) = 12
```

```
16(     1) STORE : A(1) = 12
17(     1) FETCH : J = 7
17(     1) FETCH : TEMP = 64
17(     1) STORE : A(7) = 64
18(     1) FLOW : EXIT PROCEDURE EXCHANGE
39(     1) FETCH : I = 1
39(     1) STORE : I = 2
40(     1) FETCH : J = 7
40(     1) STORE : J = 6
29(     1) FETCH : I = 2
29(     1) FETCH : J = 6
29(     1) LOOP :  2, WHILE CONDITION IS FALSE
31(     1) FETCH : I = 2
31(     1) STORE : I = 3
32(     1) FETCH : I = 3
32(     1) FETCH : N = 10
32(     1) FLOW : IF CONDITION IS FALSE
14(     2) ERROR : ASSERT CONDITION IS FALSE
```

PROGRAM LISTING (PAGE 1)

```
  LINE(COUNT)                    STATEMENT

     1          *
     2          *  HOARES QUICKSORT ALGORITH
     3          *  CACM 4,7 (JULY 1961), P. 321
     4          *  FIRST ATTEMPT
     5          *
     6(    1)   DO
     7(    1)        RESERVE DATA(1:10)
     8          *
     9          *  EXCHANGE - SWAP VALUES OF A(I) AND A(J), CHECK ORDERING IS PRESERVED.
    10          *
    11(    2)        DEFINE EXCHANGE ON A,I,J AS
    12(    2)        DO
    13(    2)             RESERVE TEMP
    14(    2)             ASSERT (I.LT.J).AND.(A(I).GT.A(J))
    15(    1)             SET TEMP TO A(I)
    16(    1)             SET A(J) TO A(I)
    17(    1)             SET A(J) TO TEMP
    18(    1)        END EXCHANGE
    19          *
    20          *  PARTITION - USED BY QUICKSORT TO PARTITION ARRAY SEGMENT
    21          *
    22(    1)        DEFINE PARTITION ON A,M,N,I,J AS
    23(    1)        DO
    24(    1)             RESERVE F,X
    25(    1)             SET F TO (M+N)./.2
    26(    1)             SET X TO A(F)
    27(    1)             SET I TO M
    28(    1)             SET J TO N
    29(    1)        PART : DO REPEAT WHILE I .LT. J
    30(    2)                  DO REPEAT WHILE X .GE. A(I)
```

PROGRAM LISTING (PAGE 2)

```
 LINE(COUNT)                    STATEMENT

   31(     4)                        SET I TO I+1
   32(     0)                        IF I .EQ. N THEN EXIT
   33(     2)                   END
   34(     2)                   DO REPEAT WHILE X .LE. A(J)
   35(     5)                        SET J TO J-1
   36(     0)                        IF J .EQ. M THEN EXIT
   37(     2)                   END
   38(     2)                   EXECUTE EXCHANGE(A,I,J)
   39(     1)                   SET I TO I+1
   40(     1)                   SET J TO J-1
   41(     0)              END PART
   42(     0)     POST : DO CHOICE OF
   43(     0)                   IF I .LT. F
   44(     0)                   THEN DO
   45(     0)                             EXECUTE EXCHANGE(A,I,F)
   46(     0)                             SET I TO I+1
   47(     0)                        END
   48(     0)                   IF F .LT. J
   49(     0)                   THEN DO
   50(     0)                             EXECUTE EXCHANGE(A,F,J)
   51(     0)                             SET J TO J-1
   52(     0)                        END
   53(     0)              END POST
   54(     0)        END PARTITION
   55          *
   56          *  QUICKSORT - SORTS THE ARRAY SEGMENT A(M) TO A(N).
   57          *
   58(     1)        DEFINE QUICKSORT ON A,M,N AS
   59(     1)        DO
   60(     1)             RESERVE I,J
```

PROGRAM LISTING (PAGE 3)

```
  LINE(COUNT)                    STATEMENT
    61(     1)              IF M .LT. N
    62(     1)              THEN DO
    63(     1)                        EXECUTE PARTITION(A,M,N,I,J)
    64(     0)                        ASSERT (M.LE.J).AND.(J.LT.I).AND.(I.LE.N)
    65(     0)                        EXECUTE QUICKSORT(A,M,J)
    66(     0)                        EXECUTE QUICKSORT(A,I,N)
    67(     0)                    END
    68(     0)          END QUICKSORT
    69          *
    70          *  MAIN PROGRAM - SORT 10 NUMBERS USING QUICKSORT.
    71          *
    72(     1)          READ DATA
    73(     1)          PRINT !UNSORTED DATA!,NEWLINE,DATA(PIC=*****)
    74(     1)          EXECUTE QUICKSORT(DATA,1,10)
    75(     0)          PRINT NEWLINE(2),!SORTED DATA!,NEWLINE,DATA(PIC=*****)
    76(     0)  END
```

POST MORTEM DUMP (PAGE 1)

EXCHANGE

I = 6 J = 4

PARTITION

M = 1 N = 10 I = 6 J = 4 F = 5 X = 18

QUICKSORT

M = 1 N = 10 I = 6 J = 4

MAIN PROGRAM

DATA(1) = 12 DATA(2) = 7 DATA(3) = 12 DATA(4) = 6 DATA(10) = 43

HISTORY (PAGE 1)

```
34(    2) FETCH : J = 6
34(    2) FETCH : A(6) = 25
34(    2) LOOP :  1, WHILE CONDITION IS TRUE
35(    4) FETCH : J = 6
35(    4) STORE : J = 5
36(    4) FETCH : J = 5
36(    4) FETCH : M = 1
36(    4) FLOW : IF CONDITION IS FALSE
34(    2) FETCH : X = 18
34(    2) FETCH : J = 5
34(    2) FETCH : A(5) = 18
34(    2) LOOP :  2, WHILE CONDITION IS TRUE
35(    5) FETCH : J = 5
35(    5) STORE : J = 4
36(    5) FETCH : J = 4
36(    5) FETCH : M = 1
36(    5) FLOW : IF CONDITION IS FALSE
34(    2) FETCH : X = 18
34(    2) FETCH : J = 4
34(    2) LOOP :  3, WHILE CONDITION IS FALSE
34(    2) FLOW : EXIT REPEAT WHILE GROUP AFTER 2 ITERATIONS
38(    2) FLOW : ENTER PROCEDURE EXCHANGE
38(    2) PARAMETERS : I = 6, J = 4
14(    2) FETCH : J = 4
14(    2) FETCH ! I = 6
14(    2) FETCH : I = 6
14(    2) FETCH : A(6) = 25
14(    2) FETCH : J = 4
14(    2) FETCH : A(4) = 6
14(    2) ERROR : ASSERT CONDITION IS FALSE
```

CONTROL CARDS :
$MUSSEL GROVES, L.J.
$DEBUG

COMPILATION :
    NO ERRORS DETECTED
    80 STATEMENTS PROCESSED

CROSS-REFERENCE TABLE (PAGE 1)

NAME	TYPE	OCCURENCES
DATA	MAIN ARRAY	7,76,77,78,79
EXCHANGE	PROCEDURE	11,19,40,47,52
A	PARAMETER	11,14,14,15,16,16,17,18
I	PARAMETER	11,14,14,15,16
J	PARAMETER	11,14,14,16,17
TEMP	LOCAL VARIABLE	13,15,17
PARTITION	PROCEDURE	23,56,65
A	PARAMETER	23,27,31,35,40,47,52
M	PARAMETER	23,26,28,37
N	PARAMETER	23,26,29,33
I	PARAMETER	23,28,31,32,32,33,39,40,41,41,45,47,48,48
J	PARAMETER	23,29,35,36,36,37,39,40,42,42,50,52,53,53
F	LOCAL VARIABLE	25,26,27,45,47,50,52
X	LOCAL VARIABLE	25,27,31,35
PART	LABEL	30,43
POST	LABEL	44,55
QUICKSORT	PROCEDURE	60,68,69,71,75,78
A	PARAMETER	60,65,68,69
M	PARAMETER	60,63,65,67,68
N	PARAMETER	60,63,65,67,69
I	LOCAL VARIABLE	62,65,66,67,67,69
J	LOCAL VARIABLE	62,65,66,67,67,68

EXECUTION :

```
UNSORTED DATA
   64          7          12          6          18          25          12          97          37          43
   78(     1) FLOW : ENTER PROCEDURE QUICKSORT
   78(     1) PARAMETERS : M = 1, N = 10
   18(     1) DUMP : A(1) = 12     A(2) = 7     A(3) = 12     A(4) = 6     A(5) = 18     A(6) = 25
   18(     1) DUMP : A(7) = 64     A(8) = 97     A(9) = 37     A(10) = 43
   66(     1) DUMP : J = 4    I =  6
   68(     1) FLOW : ENTER PROCEDURE QUICKSORT
   68(     1) PARAMETERS : M = 1, N = 4
   18(     2) DUMP : A(1) = 6     A(2) = 7     A(3) = 12     A(4) = 12     A(5) = 18     A(6) = 25
   18(     2) DUMP : A(7) = 64     A(8) = 97     A(9) = 37     A(10) = 43
   66(     2) DUMP : J = 1    I =  3
   68(     2) FLOW : ENTER PROCEDURE QUICKSORT
   68(     2) PARAMETERS : M = 1, N = 1
   71(     1) FLOW : EXIT PROCEDURE QUICKSORT
   69(     1) FLOW : ENTER PROCEDURE QUICKSORT
   69(     1) PARAMETERS : M = 3, N = 4
   66(     3) DUMP : J = 3    I =  4
   68(     3) FLOW : ENTER PROCEDURE QUICKSORT
   68(     3) PARAMETERS : M = 3, N = 3
   71(     2) FLOW : EXIT PROCEDURE QUICKSORT
   69(     2) FLOW : ENTER PROCEDURE QUICKSORT
   69(     2) PARAMETERS : M = 4, N = 4
   71(     3) FLOW : EXIT PROCEDURE QUICKSORT
   71(     4) FLOW : EXIT PROCEDURE QUICKSORT
   71(     5) FLOW : EXIT PROCEDURE QUICKSORT
   69(     3) FLOW : ENTER PROCEDURE QUICKSORT
   69(     3) PARAMETERS : M = 6, N = 10
   18(     3) DUMP : A(1) = 6     A(2) = 7     A(3) = 12     A(4) = 12     A(5) = 18     A(6) = 25
   18(     3) DUMP : A(7) = 64     A(8) = 43     A(9) = 37     A(10) = 97
   66(     4) DUMP : J = 9    I =  10
   68(     4) FLOW : ENTER PROCEDURE QUICKSORT
   68(     4) PARAMETERS : M = 6, N = 9
   18(     4) DUMP : A(1) = 6     A(2) = 7     A(3) = 12     A(4) = 12     A(5) = 18     A(6) = 25
   18(     4) DUMP : A(7) = 37     A(8) = 43     A(9) = 64     A(10) = 97
   66(     5) DUMP : J = 8    I =  9
```

```
68(     5) FLOW : ENTER PROCEDURE QUICKSORT
68(     5) PARAMETERS : M = 6, N = 8
66(     6) DUMP : J = 6   I =  8
68(     6) FLOW : ENTER PROCEDURE QUICKSORT
68(     6) PARAMETERS : M = 6, N = 6
71(     6) FLOW : EXIT PROCEDURE QUICKSORT
69(     5) FLOW : ENTER PROCEDURE QUICKSORT
69(     5) PARAMETERS : M = 8, N = 8
71(     7) FLOW : EXIT PROCEDURE QUICKSORT
71(     8) FLOW : EXIT PROCEDURE QUICKSORT
69(     6) FLOW : ENTER PROCEDURE QUICKSORT
69(     6) PARAMETERS : M = 9, N = 9
71(     9) FLOW : EXIT PROCEDURE QUICKSORT
71(    10) FLOW : EXIT PROCEDURE QUICKSORT
69(     7) FLOW : ENTER PROCEDURE QUICKSORT
69(     7) PARAMETERS : M = 10, N = 10
71(    11) FLOW : EXIT PROCEDURE QUICKSORT
71(    12) FLOW : EXIT PROCEDURE QUICKSORT
71(    13) FLOW : EXIT PROCEDURE QUICKSORT

SORTED DATA
    6          7         12         12         18         25         37         43         64         97
```

PROGRAM LISTING (PAGE 1)

```
  LINE(COUNT)                    STATEMENT

     1          *
     2          *  HOARES QUICKSORT ALGORITH
     3          *  CACM 4,7 (JULY 1961), P. 321
     4          *  SECOND ATTEMPT
     5          *
     6(     1)  DO
     7(     1)       RESERVE DATA(1:10)
     8          *
     9          *  EXCHANGE - SWAP VALUES OF A(I) AND A(J), CHECK ORDERING IS PRESERVED.
    10          *
    11(     4)       DEFINE EXCHANGE ON A,I,J AS
    12(     4)       DO
    13(     4)            RESERVE TEMP
    14(     4)            ASSERT (I.LT.J).AND.(A(I).GT.A(J))
    15(     4)            SET TEMP TO A(I)
    16(     4)            SET A(J) TO A(I)
    17(     4)            SET A(J) TO TEMP
    18(     4)            DUMP A(1 TO 10)
    19(     4)       END EXCHANGE
    20          *
    21          *  PARTITION - USED BY QUICKSORT TO PARTITION ARRAY SEGMENT
    22          *
    23(     6)       DEFINE PARTITION ON A,M,N,I,J AS
    24(     6)       DO
    25(     6)            RESERVE F,X
    26(     6)            SET F TO (M+N)./.2
    27(     6)            SET X TO A(F)
    28(     6)            SET I TO M
    29(     6)            SET J TO N
    30(     6)       PART : DO REPEAT
```

PROGRAM LISTING (PAGE 2)

```
LINE(COUNT)                    STATEMENT

    31(     8)                 DO REPEAT WHILE X .GE. A(I)
    32(    15)                      SET I TO I+1
    33(     4)                      IF I .EQ. N THEN EXIT
    34(     8)                 END
    35(     8)                 DO REPEAT WHILE X .LE. A(J)
    36(    10)                      SET J TO J-1
    37(     3)                      IF J .EQ. M THEN EXIT
    38(     8)                 END
    39(     6)                 IF I .GE. J THEN EXIT
    40(     2)                 EXECUTE EXCHANGE(A,I,J)
    41(     2)                 SET I TO I+1
    42(     2)                 SET J TO J-1
    43(     6)            END PART
    44(     6)       POST : DO CHOICE OF
    45(     6)                 IF I .LT. F
    46(     0)                 THEN DO
    47(     0)                           EXECUTE EXCHANGE(A,I,F)
    48(     0)                           SET I TO I+1
    49(     0)                      END
    50(     6)                 IF F .LT. J
    51(     2)                 THEN DO
    52(     2)                           EXECUTE EXCHANGE(A,F,J)
    53(     2)                           SET J TO J-1
    54(     2)                      END
    55(     6)            END POST
    56(     6)         END PARTITION
    57          *
    58          *  QUICKSORT - SORTS THE ARRAY SEGMENT A(M) TO A(N).
    59          *
    60(    13)          DEFINE QUICKSORT ON A,M,N AS
```

PROGRAM LISTING (PAGE 3)

```
  LINE(COUNT)                    STATEMENT

    61(    13)        DO
    62(    13)             RESERVE I,J
    63(    13)             IF M .LT. N
    64(     6)             THEN DO
    65(     6)                       EXECUTE PARTITION(A,M,N,I,J)
    66(     6)                       DUMP J,I
    67(     6)                       ASSERT (M.LE.J).AND.(J.LT.I).AND.(I.LE.N)
    68(     6)                       EXECUTE QUICKSORT(A,M,J)
    69(     6)                       EXECUTE QUICKSORT(A,I,N)
    70(     6)                  END
    71(    13)        END QUICKSORT
    72           *
    73           *  MAIN PROGRAM - SORT 10 NUMBERS USING QUICKSORT.
    74           *
    75(     1)        TRACE PROCEDURES(QUICKSORT)
    76(     1)        READ DATA
    77(     1)        PRINT !UNSORTED DATA!,NEWLINE,DATA(PIC=*****)
    78(     1)        EXECUTE QUICKSORT(DATA,1,10)
    79(     1)        PRINT NEWLINE(2),!SORTED DATA!,NEWLINE,DATA(PIC=*****)
    80(     1)  END
```

References and Bibliography

Alle 70 Allen, F.E.
Control Flow Analysis
SIGPLAN Notices 5,7 (July 1970), p. 1-19.

Ashb 73 Ashby, G., Salmonson, L. & Heilman, R.
Design of an Interactive Debugger for FORTRAN : MANTIS
Software - Practice and Experience 3,1 (1973), p. 65-74.

Bake 72 Baker, F.T.
System Quality Through Structured Programming
Proc. AFIPS 1972 FJCC, Vol. 41, Pt. I, p. 339-343.

Balz 69 Balzer, R.M.
EXDAMS - EXtendable Debugging and Monitoring System
Proc. AFIPS 1969 SJCC, Vol. 34, p. 567-580.

Barr 69 Barron, D.W.
A Note on Program Debugging in an On-Line Environment
The Computer Journal 12,2 (Feb 1969), p. 104.

Barr 71 Barron, D.W.
Programming in Wonderland
The Computer Bulletin 15,4 (April 1971), p. 153.

Bate 67 Bates, F. & Douglas, M.L.
Programming Language / One
Prentice-Hall, Englewood Cliffs, N.J., 1967.

Baye 67 Bayer, R. et al
The ALCOR Illinois 7090/7094 Post Mortem Dump
C.A.C.M. 10,12 (Dec 1967), p. 804-808.

Benn 73 Bennett, C.H.
Logical Reversibility of Computation
IBM Journal of Research and Development 17,6
(Nov 1973), p. 525-532.

Blai 71 Blair, J.
Extendable Non-Interactive Debugging
Debugging Techniques in Large Systems, Rustin, R. (Ed),
Prentice-Hall, Englewood Cliffs, N.J., 1971, p. 93-115.

Bull 72 Bull, G.M.
Dynamic Debugging in BASIC
The Computer Journal 15,1 (Feb 1972), 21-24.

Conw 63 Conway, R.W. & Maxwell, W.L.
CORC - The Cornell Computing Language
C.A.C.M. 6,6 (June 1963), p. 317-321.

Conw 73 Conway, R.W. & Wilcox, T.R.
Design and Implementation of a Diagnostic Compiler for PL/1
C.A.C.M. 16,3 (March 1973), p. 169-179.

Cuff 72 Cuff, R.N.
A Conversational Compiler for full PL/1
The Computer Journal 15,2 (May 1972), p. 99-104.

Dahl 72 Dahl, O.-J., Dijkstra, E.W. & Hoare, C.A.R.
Structured Programming
Academic Press, London, 1972.

Daki 73 Dakin, R.J. & Poole, P.C.
A Mixed Code Approach
The Computer Journal 16,3 (Aug 1973), p. 219-222.

Daws 73 Dawson, J.L.
Combining Interpretive Code with Machine Code
The Computer Journal 16,3 (Aug 1973), p. 216-219.

Dijk 68a Dijkstra, E.W.
Go To Statement Considered Harmful
C.A.C.M. 11,3 (March 1968), p. 147, Letter to the Editor.

Dijk 68b Dijkstra, E.W.
A Constructive Approach to the Problem of Program Correctness
BIT 8 (1968), p. 174-186.

Dijk 72 Dijkstra, E.W.
Notes on Structured Programming
Dahl, O.-J., Dijkstra, E.W. & Hoare, C.A.R., Structured Programming, Academic Press, London, 1972. p. 1-82.

Dora 72 Doran, R.W. & Tate, G.
Approach to Structured Programming
Massey University Computer Unit, 1972.

Dunc 62 Duncan, F.G.
Implementation of ALGOL 60 for the English Electric KDF9
The Computer Journal 5,2 (1962), p. 130-132.

Dunn 64 Dunn, J.M. & Morrissey, J.H.
Remote Computing : An Experimental System
Part 1 : External Specifications
Proc. AFIPS 1964 SJCC, Vol. 25, p. 413-423.

Earl 72 Earley, J. & Caizergues, P.
A Method for Incrementally Compiling Languages with Nested Statement Structure
C.A.C.M. 15,12 (Dec 1972), p. 1040-1044.

Elsp 71 Elspas, B., Green, M. & Levitt, C.
Software Reliability
Computer 4,1 (1971), p. 21-27.

Elsp 72 Elspas, B. et al
An Assessment of Techniques for Proving Program Correctness
Computing Surveys 4,2 (June 1972), p. 97-147.

Evan 66 Evans, T.G. & Darley, D.L.
On-Line Debugging Techniques : A Survey
Proc. AFIPS 1966 FJCC, Vol 29, p. 37-50.

Ferg 63 Ferguson, H.E. & Berner, E.
Debugging Systems at the Source Language Level
C.A.C.M. 6,8 (Aug 1963), p. 430-432.

Floy 67 Floyd, R.W.
Assigning Meanings to Programs
Proc. Symp. Appl. Math. 19 (1967), p. 19-32.

Fong 73 Fong, E.N.
Improving Compiler Diagnostics
Datamation 19,4 (April 1973), p. 84-86.

Free 64 Freeman, D.N.
Error Correction in CORC
Proc. AFIPS 1964 FJCC, Vol. 26, Pt. I, p. 15-34.

Gain 69 Gaines, R.S.
The Debugging of Computer Programs
Ph.D. Thesis, Princeton University, 1969.

Gibb 72 Gibbons, P.B.
The Design and Implementation of a Structured Programming Language with few Arbitrary Syntactic Restrictions - The Interpretive Phase
M.Sc. Thesis, Massey University, August 1972.

Glas 68 Glass, R.L.
SPLINTER - A PL/1 Interpreter Emphasising Debugging Capabilities
The Computer Bulletin 12,5 (Sept 1968), p. 180-185.

Glas 69 Glass, R.L.
An Elementary Discussion of Compiler/Interpreter Writing
Computing Surveys 1,1 (March 1969), p. 55-77.

Gord 72 Gordon, N.M. & Gibbons, P.B.
The Design and Implementation of a Structured Teaching Language
Massey University Computer Unit, Publication No. 8, 1972.

Grie 71 Gries, D.
Compiler Construction for Digital Computers
Wiley, New York, 1971.

Gris 68 Griswold, R.F., Poage, J.F. & Polonsky, I.P.
The SNOBOL 4 Programming Language
Prentice-Hall, Englewood Cliffs, N.J., 1968.

Gris 70 Grishman, R.
The Debugging System AIDS
Proc. AFIPS 1970 SJCC, Vol. 36, p. 59-64.

Gris 71 Grishman, R.
Criteria for a Debugging Language
Debugging Techniques in Large Systems, Rustin, R. (Ed), Prentice-Hall, Englewood Cliffs, N.J., 1971, p. 57-75.

Hahn 72 Hahn, K.W. & Athey, J.G.
Diagnostic Messages
Software - Practice and Experience 2,4 (1972), p. 347-352.

Halp 65 Halpern, M.
Computer Programming - The Debugging Epoch Opens
Computers and Automation 14,11 (Nov 1965), p. 28-31.

Hedr 70 Hedrick, G.E.
Classification of Programming Errors for Automatic Error Correction
Report IS-2352, Iowa State University, July 1970.

Hetz 73 Hetzel, W.C. (Ed)
Program Test Methods
Prentice-Hall, Englewood Cliffs, N.J., 1973.

Inga 72 Ingalls, D.
The Execution Time Profile as a Programming Tool
Design and Optimisation of Compilers, Rustin, R. (Ed), Prentice-Hall, Englewood Cliffs, N.J., 1972, p. 107-128.

Iron 63 Irons, E.T.
An Error-Correcting Parse Algorithm
C.A.C.M. 6,11 (Nov 1963), p. 669-673.

Jame 71 James, L.R.
A Syntax Directed Error Recovery Method
M.Sc. Thesis, University of Toronto, 1971.

Katz 69 Katzan H. (Jr)
Batch, Conversational and Incremental Compilation
Proc. AFIPS 1969 SJCC, Vol. 34, p. 47-56.

Kell 64 Keller, J.M., Strum, E.C. & Yang, G.H.
Remote Computing : An Experimental System
Part 2 : Internal Design
Proc. AFIPS 1964 SJCC, Vol. 25, p. 425-443.

Keme 67 Kemeny, J.G. & Kurtz, T.E.
BASIC Programming
Wiley, New York, 1967.

Knut 71 Knuth, D.E.
An Empirical Study of FORTRAN Programs
Software - Practice and Experience 1,2 (1971), p. 105-133.

Lang 73 Langmaack, H.
On Correct Procedure Parameter Transmission in Higher Programming Languages
Acta Informatika 2,2 (1973), p. 110-142.

Liet 64 Lietzke, M.P.
A Method of Syntax Checking ALGOL 60
C.A.C.M. 7,8 (Aug 1964), p. 475-478.

Lind 71 Lindsey, C.H. & Van Der Heulen, S.G.
Informal Introduction to ALGOL 68
North-Holland, Amsterdam, 1971.

Lock 65 Lock, K.
Structuring Programs for Multi-Program Time-Sharing On-Line Applications
Proc. AFIPS 1965 FJCC, Vol. 27, Pt. I, p. 457-472.

Lond 70a London, R.L.
Proving Programs Correct - Some Techniques and Examples
BIT 10,2 (1970), p. 168-182.

Lond 70b London, R.L.
Bibliography on Proving the Correctness of Programs
Machine Intelligence 5, 1970, p. 569-580.

Mark 73 Marks, B.L.
Design of a Checkout Compiler
IBM System Journal 12,3 (1973), p. 315-327.

McHa 71 McHaffie, R.
PL/1 Optimising and Checkout Compilers
Data Processing 13,1 (1971), p. 22-26.

Mill 71 Mills, H.D.
Top Down Programming in Large Sytems
Debugging Techniques in Large Systems, Rustin, R. (Ed), Prentice-Hall, Englewood Cliffs, N.J., 1971, p.41-55.

Moul 67 Moulton, P.G. & Muller, M.E.
DITRAN - A Compiler Emphasising Diagnostics
C.A.C.M. 10,1 (Jan 1967), p. 45-52.

Naur 63 Naur, P. (Ed)
Revised Report on the Algorithmic Language ALGOL 60
C.A.C.M. 6,1 (Jan 1963), p. 1-17.

Naur 66 Naur, P.
Proof of Algorithms by General Snapshot
BIT 6,4 (1966), p. 310-316.

Naur 69 Naur, P.
Programming by Action Clusters
BIT 9,3 (1969), p. 250-258.

Pecc 68 Peccoud, M., Griffiths, M. & Peltier, M.
Incremental Interactive Compilation
Proc. IFIP Congress 1968, Vol. 1, p. 384-387.

Peck 71 Peck, J.E.L. (Ed)
ALGOL 68 Implementation
North-Holland, Amsterdam, 1971.

Pull 69 Pullam, J.M.
An Object-Time Diagnostic Facility for High Level Languages
M.Sc. Thesis, University of Toronto, 1969.

Rain 73 Rain, M.
Two Unusual Methods for Debugging System Software
Software - Practice and Experience 3,1 (1973), p. 61-63.

Rand 64 Randell, B. & Russell, L.J.
ALGOL 60 Implementation
Academic Press, London, 1964.

Rish 70 Rishel, W.J.
Incremental Compilers
Datamation 16,1 (Jan 1970), p. 129-137.

Rose 65 Rosen, S., Spurgeon, R.A. & Donnelly, J.K.
PUFFT - The Purdue University Fast FORTRAN Translator
C.A.C.M. 8,11 (Nov 1965), 661-666.

Rust 71 Rustin, R, (Ed)
Debugging Techniques in Large Systems
Prentice-Hall, Englewood Cliffs, N.J., 1971.

Ryan 66 Ryan, J.L. et al
A Conversational System for Incremental Compilation and Execution in a Time-Sharing Environment
Proc. AFIPS 1966 FJCC, Vol. 29, p. 1-21.

Satt 72 Satterthwaite, E.
Debugging Tools for High Level Languages
Software - Practice and Experience 2,3 (1971), p. 197-217.

Schw 71 Schwartz, J.T.
An Overview of Bugs
Debugging Techniques in Large Systems, Rustin, R. (Ed), Prentice-Hall, Englewood Cliffs, N.J., 1971, p. 1-16.

Shan 67 Shantz, P.W. et al
WATFOR - The University of Waterloo FORTRAN IV Compiler
C.A.C.M. 10,1 (Jan 1967), p. 41-44.

Simp 73 Simpson, N.M.
The Design and Implementation of a Structured Programming Language with few Arbitrary Syntactic Restrictions - The Compilation Phase
M.Sc. Thesis, Massey University, June 1973.

Site 71 Sites, R.L.
ALGOL W Reference Manual
Report CS-71-230, Stanford Computer Science Dept., Stanford University, 1971.

Smit 67 Smith, L.B.
Part Two : A Survey of Most Frequent Syntax and Execution-Time Errors
Stanford Computation Centre, Feb. 1967.

Tane 73 Tanenbaum, A.S. & Benson, W.H.
The People's Time Sharing System
Software - Practice and Experience 3,2 (1973), p. 109-119.

Tayl 71 Taylor, R.C.
Source Language Debugging
Computers and Automation 20,2 (Feb 1971), p. 19-22.

VanH 68 Van Horn, E.C.
Three Criteria for Designing Computing Systems to Facilitate Debugging
C.A.C.M. 11,5 (May 1968), p. 360-365.

VerS 64 Ver Steeg, R.L.
TALK - a High-Level Source Language Debugging Technique with Real-Time Data Extraction
C.A.C.M. 7,7 (July 1964), p. 418-419.

Wein 63 Weinberg, G.M. & Gresser, G.L.
An Experiment in Automatic Program Verification
C.A.C.M. 6,10 (Oct 1963), p. 610-613.

Wein 71 Weinberg, G.M.
The Psychology of Computer Programming
Van Nostrand Reinhold, N.Y., 1971.

Wein 73 Weinberg, G.M., Yasukawa, N.F. & Marcus, R.
Structured Programming Using PL/C : An Abecedarian
Wiley, New York, 1973.

Wich 73 Wichmann, B.A.
ALGOL 60 Compilation and Assessment
Academic Press, London, 1973.

Wilk 61 Wilkerson, M.
JOVIAL Checker : An Automatic Checkout System for Higher Level Language Programs
Proc. AFIPS 1961 WJCC, Vol. 19, p. 397-404.

Wink 66 Winkler, W.S.
Computer Program Virtually Eliminates Machine Errors
Journal of Irreproducible Results 15,2 (Dec 1966), p. 39.

Wirt 71 Wirth, N.
Program Development by Stepwise Refinement
C.A.C.M. 14,4 (April 1971), p. 221-227.

Wolm 72 Wolman, B.L.
Debugging PL/1 Programs in the MULTICS Environment
Proc. AFIPS 1972 FJCC, Vol. 41, Pt. I, p. 507-514.

Youn 70 Youngs, E.A.
Error-Proneness in Programming
Ph.D. Thesis, University of North Carolina at Chapel Hill, 1970.

Zelk 73 Zelkowitz, M.V.
Reversible Execution
C.A.C.M. 16,9 (Sept 1973), p. 566.

Manuals

Burroughs B6700/B7700
Extended Algol Information Manual
Burroughs Corporation, Form No. 5000128, June 1972.

Burroughs B6700/B7700
FORTRAN Reference Manual
Burroughs Corporation, Form No. 5000458, July 1972.

Burroughs B6700
Master Control Program Information Manual
Burroughs Corporation, Form NO. 5000086, November 1970.

IBM System/360 Operating System
FORTRAN IV Language
IBM Corporation, Form C28-6515.

IBM System/360 Operating System
PL/1 Language Specifications
IBM Corporation, Form C28-6571-4.

IBM 7090/7094 IBSYS Operating System, Version 13
IBJOB Processor Debugging Package
IBM Corporation, Form C28-6393.