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.