RUN-TIME DIAGNOSTICS IN PROGRAMMING LANGUAGES WITH DATA-STRUCTURING FACILITIES

by

N. H. WHITE

A thesis presented in support of an application for the Degreee of Doctor of Philosophy in the University of Keele.

March 1980

Preface

The author graduated with Upper Second Class Honours in Physics at Oxford University in 1975. During the years 1975-1978 he has been engaged full-time in the research described in this thesis. Since 1978 he has been engaged as the Computing Assistant of the Computer Science Department at Keele University.

Acknowledgements

The author wishes to thank Dr K H Bennett for his supervision and guidance throughout this project.

He is also indebted to many colleagues in the Department of Computer Science for discussion and criticism. In particular, thanks are due to Professor C M Reeves

The author is grateful for the computing services provided by the Computer Centre at Keele and thanks are also due to the Science Research Council for financial support.

Abstract

Modern high level programming languages have been designed with the intention of providing the means of expressing the solution to a problem in the most natural way possible. This thesis is concerned with the correctness of such solutions.

The reliability of programs is discussed and techniques for increasing the likelihood of producing a correct program are examined. In particular, the use of certain high level languages affording program control structures and data types that allow an easy and natural expression of a real problem is seen to be of paramount importance. It is argued that using such programming languages is severely hampered if, in the event of a program error, diagnostic information is not available in terms of the program structure and data.

This thesis is concerned with the provision of run-time diagnostic facilities. In particular, the provision of such diagnostics for the user of data structures is seen as a currently neglected area.

The implementation of the programming language Pascal is described. Within this implementation, a unique interactive diagnostics system is built to allow the programmer complete diagnostic capabilities expressed in terms of the Pascal language. The main innovation provided is the ability to display the data structures built within a program in a manner in which the programmer views them. The reactions to this system are described and the extent to which it has achieved its aims discussed.

CONTENTS

Chapter 4 Implementation of Pascal

Chapter 6 Implementation of the Diagnostics System

CHAPTER ONE

INTRODUCTION

1.0 Introduction

The first instance of a stored program being executed is claimed to be one for calculating highest common factors and ran on the Manchester Mark 1 computer in 1948 {LAV175} The storage system used was William's tubes which retain information on continually refreshed cathode ray tubes. The computer had a monitor William's tube which gave a direct visual display of the bit pattern of selected areas of memory. More by accident than design an extremely useful facility arose from this tube. The operation of fetching an instruction word resulted in a short lived brightening of the display of that word allowing the user to see the program flow. It is interesting to note that these two facilities of memory interrogation and program flow trace form the basis of modern day diagnostic facilities in both high and low level language programming.

Some form of examining memory locations and instruction flow is often provided by the hardware of modern day computers. The greatly increased speed of present computers and the fact that they will often schedule several programs by interleaving their execution makes this facility much less useful. Such diagnostic facilities are commonly provided by software which can be tailor made to suit the requirements of the user.

1.1 Erroneous programs

Martin {MART70} relates several instances where a programming error or oversight has led to quite spectacular consequences. One program used by a magazine distributer to print address labels printed the same address on each label. The result was several lorryloads of that magazine being delivered to one subscriber. As Martin points out, 'The unsupervised computer system lacks the 'sense of the absurd' possessed by even the humblest clerk'. A common area of such program error is not catering for all possible eventualities. A particular computer system used by an American city authority only catered for peoples' ages being two digits. A woman aged 107 was discovered by the system as not attending school and as her age appeared as the digits 07 a truant notice was issued. Much more serious consequences can arise than this. The Apollo 13 moon landing mission failed and the astronauts' lives were seriously endangered. The fault was placed on a component failure. At the colloquium of the Inter University Computing Committee (IUCC) in 1978, Texas Instruments declared that the fault was actually in the software. In such space projects, where errors can endanger life and where large monetary investments are involved, great care ought to be taken. Another such project, a Venus space probe, is reputed to have failed due to a particularly error prone construct in FORTRAN. The kind of fault occurred as the result of a typographical error in a Fortran DO loop. An example of such a loop is:

```
       DO  10  I  =  1,15
        .
        .
        .
        .
        .
  10:   CONTINUE
```

The statements between the DO statement and the labelled continue statement are obeyed 15 times with the integral variable I taking on the values one through fifteen. The comma was mispunched as a full stop resulting in:

```
       DO  10  I  =  1.15
        .
        .
        .
        .
        .
  10 : CONTINUE
```

This is taken to be an assignment to the <u>real</u> variable DO10I of the value 1.15. Fortran does not require declaration of scalar variables and so the incorrect statement went undetected with I not having the expected value and no repetition.

There exist many more examples where software errors have cost a great deal of money or provided some embarrasment to individuals. It is clearly important to create correctly working programs particularly if their failure could have more than a trivial consequence. Flight controls for aircraft, data base management systems and industrial chemical plant control are three examples of many such areas of concern. Examples of software errors show both the catastrophic consequences that are possible and the continuing need for work to be done to minimise their occurrences.

1.2 Overview

The principal development reported in this thesis is the design and production of sophisticated run-time diagnostic facilities for programmers. In particular, the provision of convenient displays of the dynamic data structures that are a feature of modern high level languages is included. The high level language used for this work was Pascal. This choice was largely influenced by the needs of a substantial group of local users within the Computer Science Department, which teaches Pascal as the principal language.

Two Pascal implementations were made; the first on a Digico Micro 16E minicomputer was transferred to a GEC 4080 system. The reasons for making two implementations are historical; at the start of this project, the recent acquisition of the GEC 4080 had not been envisaged. The experience gained allows comments to be made on the portability of such compiler systems.

The need for powerful diagnostic facilities is argued. Techniques for proving the correctness of programs have shown the importance of certain programming styles, but they have not removed the programmer's need to test programs. The aim has been to allow the programmer to test programs in the most convenient manner possible. A brief account of the reaction to this system is included.

CHAPTER TWO

PROBLEMS OF PROGRAM RELIABILITY

2.0 Introduction

Programming can be described as a process of translation of an abstract problem into an algorithm which a (possibly abstract) machine can execute. The program is then an equivalent expression of the abstract problem in a form which is 'machine readable'. A machine readable form is one which is expressed in a programming language which is available on the (abstract) machine being used.

A programmer may often have to produce the abstract problem from a given set of specifications and variants. In large software companies this process may be performed by an analyst.

As an example, the set of specifications may be:

> 'Find all positive integral powers of 2 less than a given number'.

This specification could lead to the following abstract solution:

> 'Starting at 1, continually multiply by 2 until the result is that given number or more, printing out each value'.

The final machine readable solution could then be expressed (in Pascal) as:

```
read (m);          {The given number}
n: = 1;
while n< m         do
                    begin
                    writeln (n);      print the result
                    n: = n*2          calculate the next power
                    end;
```

This program text provides the results demanded by the specifications calculated in the manner prescribed by the abstract solution. It should be pointed out at this stage that although the above program appears to be correct in that it will produce successive powers of two less than a given value, it can fail.

If the machine being used allowed a range of integers from -32768 to +32767 (16 bit integer representation) and the value read was 20,000 then after printing the final result of 16384 the multiplication of this result by two would be performed. The outcome of this multiplication would depend on the machine and the implementation of the particular programming language. Either an error would be flagged indicating that arithmetic overflow had occurred in which case the program would fail, or the error would be disregarded and the result -32768 assigned to n (being the twos complement interpretation of the integer +32768). This would cause the program to continue but printing out unwanted 'results'. (In fact if arithmetic overflow were not detected, this program would not terminate).

Clearly the programmer's aim should be that the program acts as the original specifications demand. If the program behaves exactly as the specifications require then that program is said to be correct. Another way of expressing this is to say that a program is correct if it gives the right results for all possible sets of valid data in a finite time and rejects invalid data {COLE 78}.

About half the time and cost of current large programming projects is spent testing the program {LOND 75} . Despite this, much software in existence is not reliable. Many examples of commercial software errors are reported in the press. Such incidents as incorrect bills being sent to customers by a firm's computer system are common.

Given then, that after writing a program it may not be correct, and in practice often is not, there is a third stage involved in programming - attempting to convert a possibly incorrect program into a correct one. This stage is that of testing the behaviour of the program. Testing involves executing the program with selected sets of input and inductively reasoning that the program is then behaving correctly. Proving a program correct is the formulation of a mathematical proof that the program will behave correctly and does not necessarily require the program to run.

During the process of testing or proving, the program could be found to be incorrect. If a program is incorrect then the error(s) (commonly referred to as bug(s)) are to be located and removed. This process of removing bugs is referred to as debugging. Brady {BRAD77} argues that there is no demarcation between testing and proving but that they are 'just opposite ends of a continuum of supporting evidence for reliability'. He also states that it is difficult to be sure that a given proof is correct. Leavenworth {LEAV70} found an error in a program which Naur had proved correct {NAUR69} . Dijkstra has said of testing that it only shows the presence of bugs and not their absence. This statement can be levelled against proving if there is uncertainty about the proof. Per Brinch Hansen {HANS77} argues that it is worthwhile to show the presence of bugs and remove them one at a time. The time and effort involved in correcting a program once written can be kept minimal if preventative steps are taken when writing the program.

The kind of preventative steps possible and the means by which they are readily available are discussed later in this chapter. The discussion now centres on details of the tools available to the programmer to assist the production of reliable programs: testing and proving which will give increased confidence that a program is correct; programming style and self discipline which will increase the chance of a program being either correct or more easily corrected; and finally the role of debugging which is a term covering the process of turning a possibly incorrect or unreliable program into a correct, reliable one.

2.1 Program proving and Testing

A program proof is taken to be a formal mathematical argument about a program's behaviour. Assuming that the proof is itself correct then the program will behave correctly for all input that conforms to the initial conditions specified in the proof. If it is both possible and practical to prove a program correct readily then such an approach would clearly be desirable.

Testing a program involves presenting sample input to the program and checking that the resulting behaviour is correct for that. input. Having been satisfied that the program behaves correctly for several representative samples, a degree of confidence is gained that the program is correct. The only way to be certain that a program is correct is to test it for all possible sets of input. This exhaustive testing would not normally occur for one of three reasons.

Firstly, it is likely that the length of time required to test a program exhaustively is prohibitively high. For example, a simple program using three input variables each of which may take one of 4×10^9 values would require 6.4×10^{28} tests. If the program executed for a relatively short time of say ten milliseconds this exhaustive test would take slightly longer than $2_{10}19$ years to perform. Dijkstra {DAHL72} points out that an exhaustive test of a particular computer's multiplier would take 10,000 years.

Secondly, if it is fairly straightforward and quick to test a program in this manner then that program is likely to be so trivial that it would never have been written.

Lastly, assuming the program is not so trivial and that it was exhaustively tested, then its existence is no longer necessary as it can be replaced by the table of results so produced.

2.11 Program proving

Taking an operational viewpoint, programs are written in languages executable by, possibly abstract, machines. If a section of a program adheres to the rules of a particular language then some form of action as defined by the semantics of that language will occur. It is assumed that a computer will perform exactly the functions required by a particular language, that is to say it is to be expected that for all defined constructs of a given language, the computer will perform those operations which the language definition demands and no more or less.

In making this assumption it is noted that the discussion of correct programs can equally be applied to the language implementation and indeed the underlying machine structure. Even with this assurance it is still the case that programmers require some alternative language to express the function of a program. This is so for two reasons.

Firstly, as Brady {BRAD77} points out, all languages allow some form of comment facility and comments are used by most programmers to describe sections of the program to readers other than the computer. Some languages positively encourage documentation aimed at humans being embedded in the program. Much of the mandatory proforma of Cobol programs are one such instance. Algol 60 encourages labelling the end of a block or procedure by allowing a 'free comment' after each end. (It is, however, ironical that this facility is one of the greatest sources of error when a semicolon is inadvertably forgotten after an end and the following statement ignored by the compiler as a comment). The common belief, expressed for instance by Yourdon {YOURD75} , that program quality is increased by a large number of (meaningfull) comments is further evidence that programmers need an alternative means of expressing the intended functions of sections of a program to that of the programming language itself.

Secondly, it is common for a programmer to believe that the language constructs used will perform the intended tasks only to find later that this faith was misguided. When a comment occurs in a program, it is usually intended to describe that section of the program, the functions of particular variables used, the method used and the state of the program at that stage. It is this last function of a comment that is most relevant to this discussion. At points within the program text the state of the program can be specified or 'asserted'. In 1946 J Neumann and H Goldstone introduced flow charts and the original flow charts they used included 'assertion boxes' {NEUM63} . In the mid 1960's it was suggested that programs can be provably correct by expressing matters of fact concerning the program in a formalised mathematical notation. Quite independently, P Naur in 1966 and R Floyd in 1967 used this method for proving correctness. Naur {NAUR66} called these specifications of the state of the program 'snapshots' and Floyd {FLOY67} called them 'tags' or 'assertions'. McCarthy {MCCA63} did similar work in proving the properties of certain evaluations of recursive functions.

Floyd used a flowchart language and expressed the assertions in first order predicate logic. The flowcharts were constructed from five basic units:

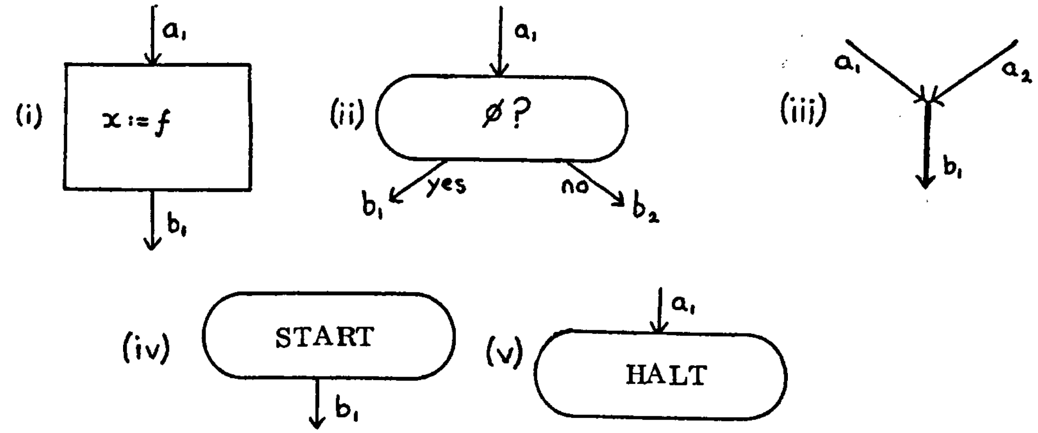


Fig 2.1

At each edge (connecting arrow) of the flow chart, an associated assertion is made. For each edge e the assertion is I(e). For example, in (i) the assignment command is executed and immediately prior to the execution, $I(a_1)$ is true. After execution of the assignment $I(b_1)$ will be true where:

$$I(a_1) \wedge x = f \vdash I(b_1)$$

where P ⊢ Q means 'from P one can deduce Q'. In (ii) the assertions will be obtained from:

$$I(a_1) \wedge \varnothing \vdash I(b_1) \qquad I(a_1) \wedge \overline{\varnothing} \vdash I(b_2)$$

and in (iii)

$$I(a_1) \vee I(a_2) \vdash I(b_1).$$

To prove a program expressed in this flowchart notation, it is shown that whenever a command is reached by a particular edge and that the associated assertion for the edge is true and that the command will be left (if at all) by any exit edge with its associated assertion true. Then by induction over the number of commands executed it is shown that if a program is entered (by a unit of type (iv)) with the associated assertion true, it will be left (if at all) with the associated assertion for the exit edge true (exit is through a unit of type (v)).

The proviso 'if at all' is included because this method of proof does not prove that the program will terminate or that any substructure of the flowchart will ever be left. Separately, a proof that a program or substructure will terminate is given. This termination proof is based on showing that particular variables have their values bounded and as such cannot continually increase or decrease. The proof of termination is required because of the definition of a correct program given earlier including the phrase 'in a finite time'.

This approach, involving a flowchart language, is that of stating in rigorous mathematical terms the state of the program in terms of its variables. These statements of fact are in addition to the program itself. Hoare {HOAR71} argues that programs should be built and proven at the same time. Hoare's argument includes the opinion that the proof and program can be partially merged by including assertions in the program text. The basis for this is twofold. Firstly, the programmer is encouraged to state explicitly the assertions that are otherwise assumed and so perhaps is more likely to question their truth; and secondly, any deviation from the assertions at any point can be detected and reported, at run-time, at the instance of its occurence. Igarishi, London and Luckman {IGAR73} have introduced an assert command into Pascal.

More recent work in program proving, especially by Hoare and Dijkstra {DAHL72} {HOAR71} , has been in the area of analysing what styles of program structure are more readily amenable to proof. Their approach includes isolating constructs which when used make a proof more difficult to formulate and those which when used make a proof more straightforward. An example of one such construct which makes proof more straightforward is the while statement:

$$\underline{\text{while}}\ B\ \underline{\text{do}}\ S$$

B is a condition yielding true or false and S is one or more sequential statement(s). The notation used in proving such constructs is that of:

$$P\ \{S\}\ Q$$

where P and Q are assertions and S is a statement or several sequential statements, and means that if P is true prior to execution of S then Q will be true after the execution of S. The theorem stated for the whole construct is as follows. If it can be shown that some assertion P is true after one iteration of S providing that it is true prior to that iteration, then it will be true after any number of iterations, including none. P is then invariant. When the while construct terminates, B will be false (unnatural exits from the loop not being permitted). Immediately prior to the execution of S, B will be true. The resulting notation for the while construct is then:

$$P\ \{S_1\}\ P$$

leading to:

$$P\ \{\text{while } B \text{ do } S_2\}\ P \wedge \overline{B}$$

where:

$$P \wedge B\ \{S_2\}\ P$$

Dijkstra {DAHL72} argues that the structure of a program text should reflect the structure of the computation. He suggests 6 basic units for building programs:

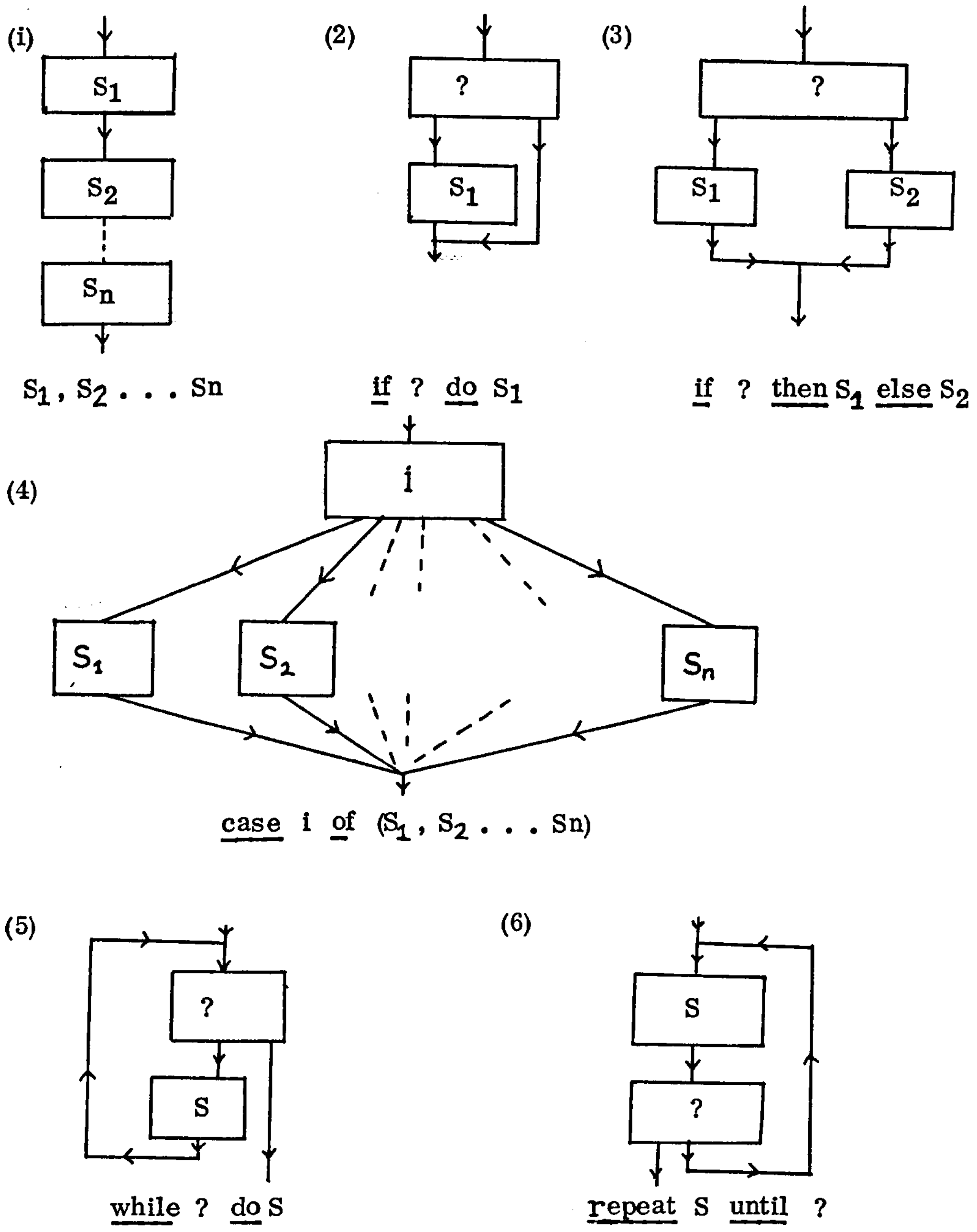


Fig 2.2

These all represent a decomposition of the program into basic units. The first is called concatenation, (2) (3) and (4) selection and the last two repetition. The important property of each unit is that it has only one entry and one exit. This tends to simplify the proof of a program when written using only these units. An example of a proof using constructs of type(1) (2) and (5)now follows.

2.12 Example of a proof

The following algorithm expressed in Pascal is designed to find the largest value contained in a set of numbers. The following declarations are assumed:

```
const n  =  . . . ;          { a positive value      }

var  a: array    [1..n] of integer;
       i, max: integer
```

The algorithm is expressed as:

```
1)    max: =  a [ 1]  ;  i: = 2;
2)    while  i  <=  n  do begin
3)    if  a  [ i ]  >  max then max: = a [ i ]
4)    i: =  i + 1      end;
5)
```

Before commencement of the algorithm we assert that $n > o$. After execution of (1) we can assert that 'max' is equal to a [1] i.e 'max = a_1'. We can also assert that 'i = 2'. The first assertion can be alternatively expressed as:

$$\forall_{1}^{i-1} x \;\Big|\; max \geqslant a_x \wedge n > o \wedge i = 2$$

meaning for all values of x between 1 and i - 1, max is greater than or equal to a [x] as well as n being greater than zero. This is clearly so as i - 1 has the value 1 and so there is only one value of x in this range, the value 1, and a [1] is equal to max.

On reaching (2), if i (which has the value 2) is greater than n, then the algorithm terminates at (5) with the truth of:

$$\forall_{1}^{i-1} x \,\Big|\, \max \geqslant a_x \;\wedge\; i > n \wedge i = 2 \wedge n > o \qquad \text{(P)}$$

The truth of $i > n \wedge i = 2$ yields $n < 2$. This coupled with $n > o$ yields the truth that $n=1 = i - 1$ and so (P) simplifies to:

$$\forall_{1}^{n} x \,\Big|\, \max \geqslant a_x \wedge n > o \qquad \text{(P')}$$

If, however, on first reaching (2) i is not greater than n, the loop is entered at (3)

At (3), on this first execution, the assertion

$$\forall_{1}^{i-1} x \,\Big|\, \max \geqslant ax \wedge i \leqslant n \wedge n > o \qquad \text{(Q)}$$

is true.

After obeying (3), max will either retain its value (if this is greater than or equal to a [i]) or will equal a [i] (if this value was larger than max). This then asserts the truth of:

$$\max \geqslant a\,[\,i\,]$$

The formula $\forall_{1}^{i-1} x \,\Big|\, \max \geqslant ax \wedge \max \geqslant a_i$ is then true and simplified to:

$$\forall_1^i \; x \mid \max \geqslant ax \wedge n > o$$

At (4), i has not changed value and so the assertion $i \leqslant n$ will still hold and so we can assert

$$\forall_1^i \; x \mid \max \geqslant ax \wedge i \leqslant n \wedge n > o$$

After obeying (4) i is increased by 1. This is reflected in the assertion as:

$$\forall_1^{i-1} \; x \mid \max \geqslant ax \wedge i-1 \leqslant n \wedge n > o \qquad \text{(R)}$$

This can be compared to (Q) above and it is certain that if (Q) is true then so is (R). In other words, the truth of (R) is not affected by the execution of statements (3) and (4). (R) is then the 'invariant' of the loop. Dijkstra {DAHL72} states the theory on page 14 that if a relation holds on entry of a reptition and one execution of the repeated statement does not destroy the truth of that relation then the relation will hold on the loop's termination. The relation (R) will then hold if and when the loop terminates. For the loop to terminate, the controlling relation must be false, that is to say

$$\neg \; i \leqslant n$$
$$\text{or} \;\; i > n \qquad \text{(S)}$$

Then at (5) we have both (R) and (S):

$$\forall_1^{i-1} x \mid \max \geqslant ax \wedge i-1 \leqslant n \wedge i > n \wedge n > o$$

Introducing k as an identity equal to i - 1 we have k=i - 1. As $i > n$, this can be expressed as $k + 1 > n$, or $k \geqslant n$. The final assertion is then:

$$\forall_1^k x \mid \max \geqslant ax \wedge k \geqslant n \wedge k \leqslant n \wedge n > o$$

The relation $k \geqslant n \wedge k \leqslant n$ yields the truth of $k = n$ and the final assertion becomes:

$$\forall_1^n x \mid \max \geqslant ax \wedge n > o \qquad \text{(T)}$$

This is identical to (P') and is the condition that must hold if the algorithm is correct. It should be noted that this proof has not included any reference to the finite range of values of integers representable by a program's variables and also does not prove that this program will terminate.

2.13 Conclusions

The above example shows how lengthy a proof can be for even a small program. Generally, the effort involved in proving a program correct will be far greater than that involved in writing the program. Some effort has been made to automate the process of proving a program correct. Igarishi,London and Luckham {IGAR73} devised a system which,given assertions corresponding to each line of control in a program will simplify all path conditions where possible. This then leaves only a few path conditions for the user to prove. An interactive system for formulating proofs and defining theorems is described by Gordon {GORD77a} , {GORD77b}. The system LCF (Logic for Computable Functions) helps generating proofs in an interactive environment.

Proving programs correct may seem to be the ideal solution to the problems of program reliability. It is the large effort involved in proving a program correct and the expertise required to formulate the proof that inhibits more widespread use of this technique.

Consider a large program such as a commercial applications package or an operating system. It will almost certainly be impractical to exhaustively test such programs, but equally it could be considered impractical to prove their correctness if this process was lengthy. Preventative techniques would be employed in the production of such programs. The program would be tested and errors arising investigated. At some stage in time the program would be expected to fail rarely. Proving such a system correct may well take considerably more effort than the original production and testing. It is the consideration of expense of effort which has a large role to play in the decision as to whether or not a large project will be justified or not.

A similar decision will often be made when a programmer is considering a small program which may not be run very often or which has a short life. In this situation a combination of error preventative techniques, diagnostic facilities and some proportion of 'informal proving' is often used whereby the programmer is convinced beyond 'reasonable doubt'.

A particular environment where program proving is at present inappropriate is that of students learning a programming language for the first time. Firstly it is extremely difficult both to learn a language and learn how to formulate proofs simultaneously. Secondly it is unlikely that a correctly formulated proof will be given by a beginner. An incorrect program proven correct by an incorrect proof is a potentially dangerous situation. There are currently no known examples of students being taught to prove programs correct while they are learning a programming language for the first time.

In conclusion, program proving is clearly a desirable ideal but cannot be regarded as the programmer's sole tool for writing reliable programs. The concept of a neat, formalised mathematical proof is appealing but is itself dependant on the assumption that mathematics is absolutely correct - or indeed suitable for that particular environment of the program. As mentioned above, Leavenworth {LEAV70} , by testing one of Naur's programs which Naur {NAUR69} had 'proven' correct, showed that program proving and program testing are similar. Proving is a method of testing a program but without running it on a machine. The programmer interprets the program by hand making assertions about its behaviour in a predicate logic. In testing, the programmer runs the program on a machine making assumptions about its reactions to certain inputs. As has been shown, neither method can be considered ideal at the present time. Testing is not ideal because it does not cater for all eventualities and proving is not ideal because it relies on a correct programmer. However, program proving must play a major role in attempts to write reliable software. Advances made have much reduced the effort needed to prove correctness and by automating proof checking, have lessened the reliance placed on the programmers expertise in logic.

It is expected that doubt should be cast on a program declared to be correct by a programmer who has tested it thoroughly. Similarly doubt must be cast on a program which has been proven correct. It is perhaps even more critical to doubt a program proof. Because of the great reliance placed on mathematical proofs it can be easy to accept a program proof as testimony to a program's correctness. If the proof is wrong then this misplaced reliance is dangerous. Even considering such disadvantages of program proving, its role in programming is clearly justified if only because its existence as a limit will tend to pull the average programming practice towards the correctness that it demands.

2.2 Structured Programming

The previous discussion on program proving suggested that the style of writing a program has a large influence on the effort required to prove its correctness. Dijkstra {DAHL72} argues that programs are more readily proven correct if there is a close relationbetween the progress through the computations of a program and the progress through the program text. In discussing structured programming Dijkstra states that one of the aims is to make well structured programs so that 'the intellectual effort (measured in some loose sense) needed to understand them is proportional to the program length (measured in some equally loose sense)'.

The term structured programming has been used to describe many programming practices such as modular programming, stepwise refinement and goto-less programming. In many ways structured programming covers all these areas.

The common belief that structured programming must mean goto-less programming is incorrect but not without foundation. An unrestricted use of goto can easily lead to unstructured programs. In a large program, the presence of a label can mean that elsewhere in the program any number of gotos could transfer control to that label. In considering whether or not a section of program that contains a label is correct assumptions are made. To make a valid assumption means that consideration of all the sections of the program that contain a jump to that label must be made. This is clearly more error prone than it would be if no label existed. It is this uncertainty of transfers of control that make the goto statement error prone. It has been said by Barron that if a goto statement is justified then so is a comefrom statement listing all labels which (otherwise) would be goto statements.

A program that is well structured implies that the whole program can be understood by looking at small sections at a time and in considering each section little or no cross-reference searching through textually distant sections of the program is necessary. This has several consequences. Firstly if a section is textually too large to consider as a whole then it should be split into distinct sub-sections which are individually small enough to consider. Secondly any use of a goto statement should be very restricted so that it is both immediately clear where the control is passing to and that this jump distance is small and in the same section. Thirdly this restriction on localisation of the use of goto necessitates (in a larger than one or two section program) the use of procedure calls. Finally as each procedure call would need to be self evident in its action in order that the first consequence is not violated this implies that procedures should perform well defined basic tasks that are fundamentally required in defining the program's function.

In a large program many procedures would be needed and they would essentially be used as basic statements thus creating a new language on top of the programming language being used.

The resulting program structure can be represented as a tree structure:

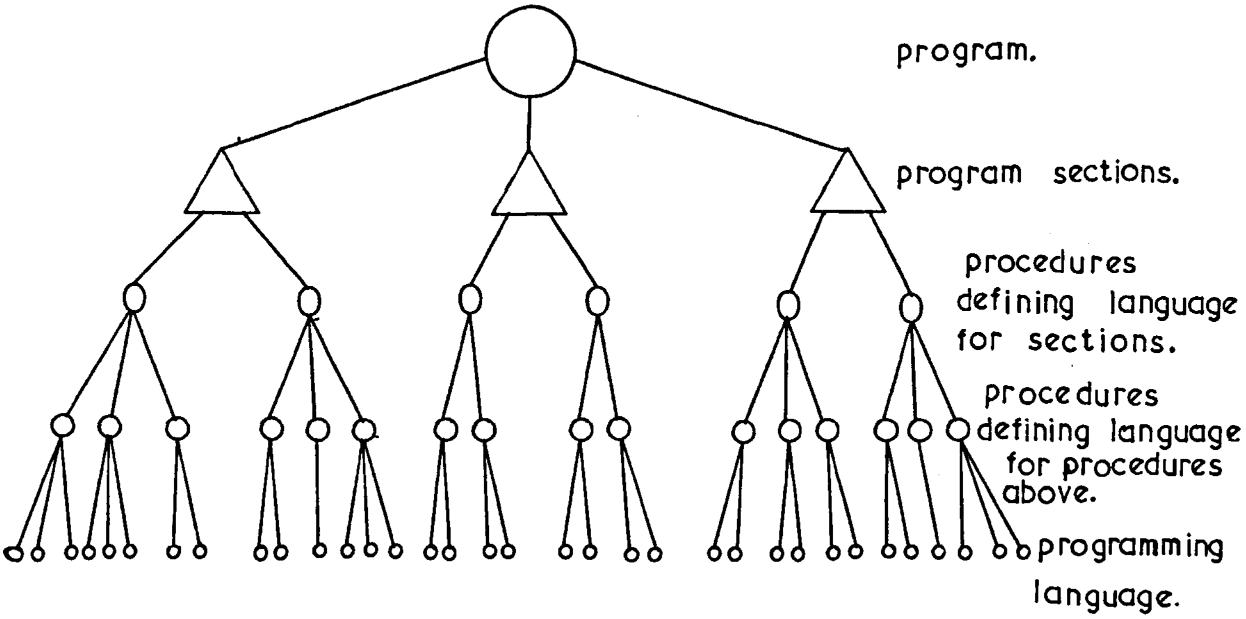

program.
program sections.
procedures defining language for sections.
procedures defining language for procedures above.
programming language.

Fig 2.3

This diagram depicts several levels. At the bottom are the programming language statements which define the bottom-most procedures. These basic procedures are used in defining more advanced procedures which in turn are used in eventually defining the program's function. How deep the tree is will depend upon the size of the program.

The notion of stepwise refinement {WIRT71a} is apparent in this tree structure. In stepwise refinement one starts by defining the program's function as a short statement. This corresponds to the root of the tree. This statement is then refined into two or three statements which themselves represent subgoals of the initial statement. This refinement continues down the tree until the statements are simple enough to be expressed in an existing programming language.

The structured approach to programming has been shown earlier to result in programs which are easier to prove correct and this reason is Dijikstra's recommendation of structured programming {DAHL72} . If a program is to be structured in a way that minimises the task of proving it correct, by definition, this increases the ease of asserting conditions on the program's behaviour. It follows from this statement that each section of the program, being expressed in a way facilitating a correctness proof, is in the best form for the programmer to understand exactly the resultant behaviour and to be convinced in an informal manner that this section of the program is correct. By restricting program flow to the small set of basic building blocks described in the previous section the programmer, being well aquainted with such control structures, can call upon the well established experience of their behaviour. This knowledge enables ready understanding of the program by others and increases the likelihood of the programmer being satisfied of its correctness.

2.21 Modular programming

The term modular programming is used to describe the division of a program task into several sub-tasks. Each sub-task should be an isolated function of the program that is self contained to the extent that it can be written outside the context of the complete program. This technique has two advantages. Firstly it is easier to comprehend several self contained programs individually than one large program performing each task with no demarcation. Each identifiable sub-task can be understood as a complete unit without reference to other units. Secondly, modular programming is advantageous should a program be written by more than one person. Each member of such a team could be assigned one or more complete units which could be programmed independently of the other units. The programmer would work to a specified interface between units but need only know what other units do and not the details of how they do it.

At the start of this section, the statement made by Dijksta was quoted. The statement suggests that the effort required to understand a program is proportional to its length. If the complete function of a program is expressed as a whole with no attempt to divide it wherever individual tasks can be isolated then it seems reasonable that the effort required to understand such a program would not be proportional to its length. The effort required is likely to increase in an exponential manner for such large programs. There is a limit to the amount of program text that can be retained in the human mind at a given time. Understanding a large program is then a process of understanding individual functions of that program separately. If these functions are not logically and textually isolated this process is clearly hampered. It would seem reasonable that the effort required to understand one small program is not affected by how many similar sized programs are to be understood next or have just been understood. If the process of understanding a large program is that of understanding each small section individually, separately and sequentially then Dijkstra's statement is satisfied. The effort required to understand large programs written in a modular fashion is then completely dependant on the programming style used within each unit.

The units described can be regarded as the nodes of the tree on figure 2.3 down to a certain depth. The depth of the lowest node which would be considered a complete unit depends on the size of the program. The size of a unit cannot be fixed but is bounded by two considerations. The first consideration is that as described above; each unit should be small enough to be understood readily as a whole. If a unit is considered too large then functions within it are isolated and treated separately. This process adds an extra level into the tree. The second consideration is that as more levels are introduced into the tree there comes a point where the tree is too deep for easy understanding of the higher level units. In understanding a unit it is necessary already to understand the functions of the units below on the tree. As the depth of the tree grows the rapidly increasing number of units comprising the program may become too large. Understanding a unit is easier if the need for that unit is realised. To understand sequentially many units with little realisation of their intended purpose could be more difficult than understanding fewer larger units. This is particularly true where it is difficult to isolate sub-tasks performed within a large unit.

An aspect of modular programming equally important as splitting the program statements into manageable sizes is that of the data used by these units. It would be possible to declare every data item used by the program globally. That is, all data items would be the property of the top tree node. Apart from the obvious problems that could arise from recursively called procedures this approach is not desirable. For the same reasoning that has been used to keep program units to a manageable size, the amount of data declared within a unit should be kept to a small size. Data declared within a unit should be only that required either directly by that unit or by more than one unit at the next level down the tree. The data declarations are then spread down the tree of figure 2.3 to where they are required. This refinement is especially necessary if several people are writing units of a program. Each unit can operate on its own declared data without regard to possible corruption of that data by other units. Even when one person is writing the program, it is helpful if when writing a unit, regard does not have to be paid to other units' access to common data. This aspect of localising data to the units that act upon it is discussed in the next section in relation to classes and modules.

2.22 Stepwise refinement

The notion of stepwise refinement was introduced by Wirth {WIRT71a} The essence of stepwise refinement can be described using figure 2.3. The tree is created from the top downwards. First of all, the function of the program is described. This is then refined by splitting this function description into several more detailed descriptions, thus creating the nodes one level down. This process is repeated on each node until the detailed description is directly represented in a programming language. In parallel with this decomposition of the initial specification Wirth includes the refinement of the data used. As tasks are decomposed, the data structures may need refining to suit the more detailed descriptions.

This process does not imply that initial refinements are binding on the final program. Earlier descriptions may be revoked at any refinement stage and restated. This backing up the tree can continue to the top if necessary. The notation used for each refined description depends on the problem being solved. While the target computer language is the final result, Wirth argues that the notation should be one that is 'natural to the problem' for as much of the refinement process as possible. The direction that this notation proceeds is determined by the target language and therefore stepwise refinement is more suitable if this target language allows a natural expression of the problem and its data.

Stepwise refinement inevitably leads to a modular program as each refinement produces well defined units. The program's data is refined and declared around the relevant units as the process proceeds. Wirth does, however, point out that decisions about data representation may have to be delayed as the target code that will result may not be readily forseeable. The detailed instructions to manipulate the data will dictate a natural representation of the data structures required.

2.23 Summary

The principal aim of structured programming is to produce programs with a high probability of being correct. The main rule of this technique is that programs should be written in units which are both short enough to understand easily and written in a style that reflects the problem being solved in a natural form. The data representation chosen should ideally reflect the abstract objects the algorithm is intended to manipulate. For complex problems the ability to structure the data types used is a clear advantage. For this reason it is easier and more natural to use structured programming techniques in languages such as Algol or Pascal than Fortran or Basic, the latter languages offering little scope for the programmer to design a data representation. At the beginning of this section, the consequences of programming in units of manageable size was discussed. The program inevitably takes on a tree structure as illustrated in figure 2.3. The advantages of such modular programming and a technique for ensuring its use were related. Stepwise refinement is a top-down programming technique. In building the tree in figure 2.3 no node is created before the node it is connected to above. Strict bottom-up programming whereby the lowest levels are built first implies building basic functions on a language then using these functions as a new higher level language to create the functions one level higher and so on until the program is complete. To decide what basic functions are required would require a knowledge of the functions above and in practice a judicious mixture of these two methods is often used. It would be rare to build a program by strictly adhering to a policy of completing one level of the tree at a time and neither bottom-up nor top-down techniques demand this. They do demand that the tree is built strictly by following the connections between the nodes without skipping over a level. In summarising structured programming four points are detailed. Firstly programming takes place in small steps. Concentration on a small isolated function of a program should ease the task of building the program and give the opportunity of proving each function correct formally or informally convincing oneself of correctness at each stage. Secondly when a change to the program is required, it should be isolated to one or more basic units which can be altered with minimal alteration of the program as a whole. This can be most important in commercial programming where program specification can easily alter. Thirdly the program should be more easy to understand as each basic function can be self documenting and concentration on small areas of the program at a time is possible. Finally the opportunity of reflecting the structure of the data to the structure of the program is provided. The various manipulations required on the data structures of the program are reflected in the design of those structures and a natural correllation can exist between the data and the statements {WIRT76}.

2.3 Syntactic constraints

A syntactic constraint is a method whereby a language demands the explicit declaration of an object or operation which could otherwise be deduced. The constraints imposed by the syntax definition of a programming language can play a large part in creating correct programs. Some common sources of error can be recognised and programming languages defined such that syntactic constraints eliminate such potential sources of error when possible. A common method of imposing such constraints in programming languages is often termed redundancy. That is, a programmer is required explicitly to declare some attributes of a piece of coding or of a variable which could otherwise be inferred by the compiler. Languages such as Algol and Pascal require that all variables be introduced by a declaration. Fortran does not require a declaration; instead a variable is introduced by its use in the program. The type of the variable is dictated by the first letter of the variable name: if the first letter is one of I, J, K, L, M, N than that variable is an integer otherwise it is a real. In using Fortran, a misspelling of a variable name will result in two different variables being allocated by the compiler - and a potentially difficult error to find.

2.31 General

Another instance of redundancy is that of associating a variable name with a type. In Pascal, for example, a declaration might be:

```
var          i : integer;
             b : Boolean;
             x : real;
        smalli : 0..99;
```

In this case, i, b, x, smalli are variables each having a particular type which defines the set of values these variables may have. An assignment such as:

```
i : = b
```

would not be allowed by Pascal syntax rules. b is a Boolean variable and i an integer. It is not immediately obvious what the programmer would mean by this assignment. The variable smalli is a subrange of the integer type and may only have the values ranging from 0 to 99. Whereas any integer or subrange of an integer may be assigned to smalli without causing a syntactic error, at run time a value out of the range 0 to 99 would cause an error. In PL/1 the assignment i : = b would be acceptable and the values true and false yield the integers 1 and 0 respectively. In Pascal the same effect is achieved by:

```
i : = ord (b)
```

where ord is a function which maps other types onto the integer range of values (if possible).

The need for a programmer explicitly to state that a type conversion is required can be argued on the grounds that by forcing the programmer to do so and not allowing the compiler to assume conversion is less likely to result in an error. An example {BARR77} of this is the statement

$$a < b < c$$

where a, b, c are integers. As a mathematical expression, this would mean that b is within the range of values a to c. In PL/1 this would be parsed as follows:

$a < b$ would produce a result true or false.
This would then form

true<c or false< c

true and false would be automatically converted to 1 or 0 and compared with c to yield a Boolean result.

This may be the intention of the programmer, but by forbidding implicit type conversion a possible error is averted.

As another example, consider the following assignment where i, j are integers and x is real:

```
i: = j + x
```

This will usually be interpreted as follows:

1) j' is formed as the real equivalent of the integer j.

2) the sum j' + x is calculated yielding a real result r.

3) the integer equivalent of r, r' is formed (if possible)

4) r' is assigned to i.

In addition the variable names are (usually) dereferenced automatically to yield their values.

In Algol 60, step 3 will result in the rounding up of the real value so that for instance:

```
j: = 1;
i: = j/2;
```

will result in i = j being true. This implicit type conversion can clearly lead to unexpected results because the rules for conversion vary between languages and are not explicit in the program text.

In Pascal, an assignment of the form

```
        i: = <expr>
```

where <expr> results in a real value and i is an integer is not allowed. Instead the forms:

```
        i: = trunc ( < expr > )
     or i: = round ( < expr > )
```

must be used where the conversion is explicitly stated as being by truncation or rounding respectively.

A particular instance where this syntactic constraint can detect a possible error is the case of:

```
        i: = j/k
```

where i, j, k are integers. The result of j/k will generally not be integral and in Algol 60 results in a rounded result. Because Pascal does not implicitly convert real to integer this assignment would be rejected as / is defined to produce a real result.

Perhaps the most error prone numerical operation is that of exponentiation:

In Algol 60, the type of the result yielded by

```
        a ↑ b
```

depends not only on the types of a and b but also their values. The complex rules for the value and type of the result are given in the Algol 60 report {NAUR62} as follows:

1) a of type integer or real
b of type integer

b > o the type is the same type as a.
b = o the type is the same type as a and the value is 1 (1.0) unless a is zero in which case the value is not defined.
b < o The type is real and the value undefined if a = o

2) a of type integer or real
b of type real

a > o the type is real
a = o the type is real and the value is 0.0 unless b ⩽ o when the value is not defined.
a < o the type and value are not defined.

For example 2 ↑ i would yield an integer with value 4 if i was +2 and a real with value 0.25 if i was -2.

These complex rules are not so much a comment on Algol 60 as a comment on the exponentiation operator. Because of this dependance on the values, the value (if defined at all) and type of a ↑ b can lead to unexpected results. It is mainly because of this uncertainty (and difficulty of implementation) that Pascal imposes the ultimate syntactic constraint of not including an exponentiation operator. In this way, programmers must implement exponentiation themselves and include explicit actions to be taken depending on the result of the operation (if any).

2.32 Classes and Modules

A very important syntactic structure is that of then 'class' of Simula {BIRT73} and the 'module' of Modula {WIRT77} . Both constructs represent a very localised section of self contained program which contains a set of data items and all procedures necessary to manipulate this data. In this way, specific functions required by the main program can be built and tested in isolation from each other. An attempt to manipulate a class's data other than by using one of its procedures will result in a compilation error. The idea behind this design is to localise an error caused by incorrect manipulation of data items. As an example, consider the operation of a stack. Four procedures can be given:

1) Pop an item off the stack
2) Push an item onto the stack
3) Return the value of the item on top of the stack
4) Test for empty stack

This may be programmed in Simula as follows:

```
class  stack (size); integer size;
begin
    integer SP; integer array  st k (1:size);

    procedure    pop;
    if SP > 0 then SP: = SP-1 else
                            error ('stack underflow');
    procedure  push (i); integer i;
    begin
    SP: = SP + 1;
    if SP > size  then  error ('stack overflow')
                  else  stk (SP): = i;
    end;
```

```
integer procedure  TOS;
     if  SP = 0  then  error ('empty stack accessed')
                   else  TOS: = stk (SP);
Boolean procedure  empty;  empty: = SP = 0;
     SP: = 0      {initialisation - obeyed once when a variable
                   of this class is declared}
end * * * stack * * *
```

A declaration of the form:

```
ref (stack) s1 (4);
```

would create an object of class stack. This object would contain the stack itself, its size (4) and a stack pointer (SP). The statement(s) following the procedure definitions in the class would be automatically obeyed, initialising the stack as being empty (SP: = 0). The operations then available would be:

s1. pop	remove the top element
s1. push (n);	push the value of n onto the stack
s1. TOS;	the value of the top element of the stack
s1. empty;	indicate whether the stack is empty.

There is no syntactically correct means of accessing the stack's elements other than by these four procedures. Should an error occur, then by examining the nature of the error the section of program responsible is readily isolated. For example, a programmer manipulating a stack directly in several sections of a program may attempt to remove several items from the stack without correctly altering the value of the stack printer. Similarly confusion may arise as to whether the stack pointer is pointing to the next free location on the stack or the top location of it. This localisation of data manipulation has three intended purposes. Firstly, should an error occur it can be localised more easily; secondly all elementary operations on the data items can be tested in isolation and finally new class's can be introduced without the fear that they will interfere with existing ones and result in indirect errors.

2.33 Summary

If a language design incorporates certain syntactic constraints, the chance of a program containing errors can be decreased. Clearly there exists a balance between constraining the syntax to eliminate errors and providing a language that is usable. The particular case where syntactic constraints designed to minimise the chance of a program error are of greatest importance is in the area of concurrent programming. Per Brinch Hansen {HANS77} makes it quite clear that ideally all errors should be trapped at compile time because when two or more processes are corunning it may be impossible to repeat a particular event or error. Brinch Hansen designed Concurrent Pascal to have a syntax containing such constraints designed to eliminate program errors as discussed above. At the time of writing, Niklaus Wirth is working on a new language for programming operating systems. The language is a combination of Pascal and Modula {WIRT71} , {WIRT77} and is being designed with these points in mind.

2.4 Diagnostics

Techniques have been described that increase the chance of a program being correct. It can be argued that if structured programming and program proving techniques are used then the program is guaranteed to be correct. Even so, it is still true that incorrect programs exist. This is so for three reasons:

1) Such techniques as proving were not used.
2) The techniques were not used correctly.
3) Beginners may not be sufficiently capable of using such techniques.

Some justification for the first reason has already been given and the second reason is a particular case of 'Murphy's law' that is if something can go wrong it will. For whatever reason, a programmer will often find that the program written is not correct. When this happens, the programmer will embark on what is commonly called the debugging phase, that is, finding out where the errors (bugs) exist and eliminating them. To locate the errors, the programmer will make use of any evidence of error available. Most usefully, the computer system will provide diagnostics at the instance of an error. A diagnostic can be said to be a computer given clue to the incorrectness of a program. The use of such diagnostics has varied from system to system and from error to error. If a program runs and stops with the message:

```
Failed
```

then this is diagnostic. It informs the programmer that the program has not run correctly (unless, of course, failure was intended). If the program stops running and a message states:

```
Failed - division by zero.
```

then this is a better diagnostic because it gives the reason for failure and so indicates that the section of program including division enclose the error. Following on, a better diagnostic would be:

```
Failed - attempted division by zero in line 36.
```

because this informs the programmer the line in which the error was detected. This may not necessarily be the line which contains the bug and perhaps the best diagnostic would be of the form:

```
Failed - attempted division by zero at line 36. Variable
        b was assigned the value zero at line 25.
```

The usefulness of a diagnostic can be determined by the effort involved to locate the error and subsequently correct it.

Diagnostics are especially useful in the case of a very error prone programmer. The most common case of such a programmer is the beginner learning how to program or learning a new programming language. Such beginners cannot generally be assumed capable of proving programs when they do not know the language itself properly. They will need good diagnostic information as errors are very likely to occur and the cause of these errors will probably not be understood. Not only is it a help to the beginner to learn the language if good diagnostics exist, but also, if such diagnostics are absent then this will be an impediment in such a learning process.

High quality diagnostics are not just necessary for the beginner. The experienced user can often save time if at the instance of an error, good quality diagnostics are available. Many systems provide a facility for a post mortem dump to be provided as a 'last resort diagnostic'. The programmer is expected to interpret this vast array of digits and (usually with the aid of a manual) locate the section of program that is incorrect. One or two manufacturers are currently marketing calculators which will perform arithmetic in either decimal, octal or hexadecimal notation and convert numbers between these radices. Presumably these are aimed at people who have to interpret such post mortem dumps. Surely the computer system which converted the programmer's source program and data items into the post mortem dump form is much more suited to interpreting it than the programmer. This point is expanded in the following chapter but serves the purpose of justifying the need for good quality diagnostics to be provided for all users of a system including the experienced ones.

Diagnostics can be placed intwo main categories: compile-time diagnostics and run-time diagnostics. Compile-time diagnostics are given when a program does not conform to the syntactic rules of the language, that is it is either grammatically incorrect (context free) or is not consistent with some other section of the program (context sensitive). The first type of error could be, for example, a typographical error in formulating a particular language construct and the second type could be the attempted use of a variable that has not been declared.

2.41 Compile-time diagnostics

In general compile-time diagnostics of a high quality are capable of being implemented. The compiler, on detecting an error,can hold sufficient information to diagnose the fault extremely well. The compiler after all should be well aquainted with the language it compiles. Sometimes the presence of an error may sufficiently upset the compiler that further errors may be wrongly diagnosed or not diagnosed at all and some correct statements may be faulted. The degree to which this occurs depends upon how well the compiler is able to recover. Two examples of compilers which provide means for recovery on detecting an error are the Pascal P compiler {WIRT71c} and the PL/1 checkout compiler {CONW73} .

The Pascal compiler takes advantage of the fact that Pascal's syntax can be presented as a finite set of pseudo-finite-state recognisers and the syntax analysis can follow the method described by Conway {CONW63} , that is a separable transition diagram technique. Pascal was then designed so that parsing is possible with the constraint that only one symbol look ahead is necessary. In this way, the syntax diagram can be represented in the compiler as a corresponding set of procedures, each parsing a subgoal and using top-down parsing techniques. Each procedure is provided with two sets of symbols of the language. These symbols are the reserved words of the language which introduce particular syntactic constructs. The first set is comprised of the symbols which may permissably follow the construct being parsed. In this way, having parsed a construct, if the next symbol is not contained in this set then an error occurs. In the event of an error, the text is skipped over until a symbol in this set is encountered. In order not to skip over important sections of the program a second set of symbols is provided. This set consists of those symbols which may introduce a construct which in the current context should not be skipped over. As an example, in parsing an assignment statement such as:

```
x: = a + b
```

the possible follow symbols would be

```
; end
```

and those symbols which must not be skipped over are those which may legally start a statement such as case with if <identifier> begin while repeat and for.

The parsers proceed using the following type of statement:

```
if symbol = next legal symbol then get next symbol
                              else error
```

In this way, omission of certain key symbols will be detected but the compiler effectively inserts them. This error recovery is fully described by Wirth {WIRT71c} ,{WIRT76} .

The PL/1 checkout compiler uses a different approach for error recovery. It has four basic actions which it may take in the event of an error:

1) Delete the next symbol
2) Insert a synthetic symbol
3) Replace the next symbol with a synthetic symbol
4) Delete the previous symbol

For example, confronted with two consecutive operands, it could either delete one of the two operands or insert a binary operator between them. The tactic applied is heavily dependent on the context. As a last resort the current statement is replaced by a nil statement, which will identify itself at run time, and the source is skipped over to the next reserved word.

These two different approaches are employed with different aims. The Pascal compiler attempts to detect each error, report it and recover without giving spurious extra errors. The PL/1 compiler aims to produce a compiled form of the program that will run no matter how many errors are present. This approach is justified for the environment where very few attempts may be made on any one day to run a program and so time may be saved if the compiler corrects compile-time errors and does so correctly.

In general, both compilers can recover from errors remarkably well; in particular the Pascal compiler would be expected to because one of the design aims of the language was to permit easy error recovery by a compiler.

2.42 Run-time diagnostics

Run-time errors are not as straightforward to diagnose. The compiler detects and diagnoses compile-time errors well because the cause of compilation error comes from a finite set of possible causes and the compiler by definition contains all the language rules. A run-time system can detect an error symptom but not necessarily its cause. It cannot automatically recover from an error, as is sometimes possible during compilation, unless it has a knowledge of the program's intended function.

This in itself would mean that a correctly working program already existed and therefore is somewhat paradoxical. To provide good run-time diagnostics then, the system would need to detect errors and provide whatever extra information was required by the programmer to find the cause of the error. Recovery from a run-time error, while not normally expected, could be possible.

2.43 Conclusions

Run-time diagnostics have a distinct place in any process of producing correct programs. They provide diagnostic information during the testing phase or debugging phase. Earlier discussion pointed out the parallel between testing and proving. A run-time system providing good diagnostics has as its parallel an interactive proof checker such as that of Igarishi {IGAR73} or the Edinburgh LCF {GORD77a} .There is a danger, however, in drawing this parallel too closely because proving programs can easily be considered much more of a preventative technique (which, ideally, it is). Concepts of structured programming and syntactic constraint are preventative techniques used in writing correct programs. Debugging is a cure. A loose analogy with these techniques can be made with medicine. Human diseases can usually be diagnosed and often cured. Similarly preventative medicine serves to lessen the chances of such diseases. People will often arrange their lifestyle so as not deliberately to contract known diseases. In finding cures and new diseases, as a result of diagnostic information, medicine will add to its preventative procedures as a result of this new knowledge. This analogy can now be drawn back to programming. Experience of the difficulties of debugging unstructured programs reinforces the techniques of structured programming. Experience of common causes of error gained in debugging leads to preventative techniques being employed in languages to help eliminate these sources of error. Classes and Modules are two similar examples.

This feedback is of great importance to programming and by itself justifies sufficient attention being paid to debugging and diagnostics. This thesis is concerned with high quality diagnostics at run-time and in particular, the emphasis is placed on diagnostics concerned with data structures, an area that is emphasised in the following chapters.

CHAPTER THREE

RUN-TIME DIAGNOSTICS

3.0 Introduction

This chapter describes the provision of run-time diagnostics. The need for run-time diagnostics systems is argued for in terms of their usefulness, and a case is presented suggesting that they fulfill as necessary a function as a compiler. A set of currently existing diagnostic systems is described. Among this set are examples of good diagnostic systems and some extremely bad systems. The facilities that ought to be provided in a good diagnostics package are then identified. Of particular interest is the provision of diagnostics concerned with data structures and pointers which is seen to be an area where little progress has been made so far. The filling of this gap is the prime objective of the work here described and a diagnostics package is described which meets such a need.

3.1 Problem orientation

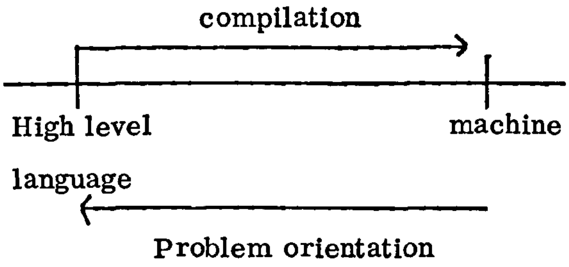
Introduction

The purpose of this section is to show that run-time diagnostics are required by a programmer as a direct consequence of the availability of high level languages. The term 'problem orientation' is introduced as a means of illustrating this need for diagnostics. The process of programming involves formulating and expressing the application of certain activities on specified data. A machine can be programmed in terms of a given repetoire of tasks - the elements of an available programming language. More powerful tasks can be defined in terms of those functions that exist on the machine thus adding to the available set of functions. As an example, the task of moving the contents of a given number of storage locations from one address to another may not be an available hardware function. This task can be performed using a number of available instructions in a subroutine which can be made available as a general utility. In such a way, many functions that are more powerful than the machine's instruction set are available. In effect, a language is created on top of the machine's language - the instruction set. This process is repeatable such that another layer of functions can be created using this new set of instructions. As such a language develops, the functions used to express algorithms become increasingly distant from the machine's instruction set. The expression of an algorithm is, usually, more laborious if it takes place using the lower level functions. The set of functions available - the language - can be given a value relating to the relative ease in which a particular algorithm can be expressed. In addition, the higher level data manipulation functions that can be available may aid the task of expressing a program's data structures. It is this loosely defined value of the relative ease w ith which a particular program may be expressed that is called the problem orientation.

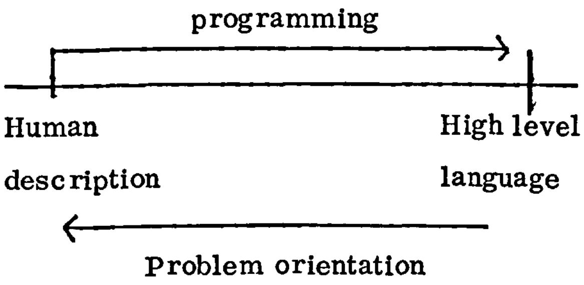
3.11 The degree of Problem Orientation

The term problem orientation can be applied to available programming languages. In doing so we can say that for a given task, language A has a higher problem orientation than language B if (assuming the programmer is equally conversant with both languages) the task is more readily expressed in language A. This does not necessarily imply that language A is a better language. A different task may be more readily expressed in language B. It is therefore the case that any ordering of languages by their problem orientation can only apply for a particular t ask or set of tasks.

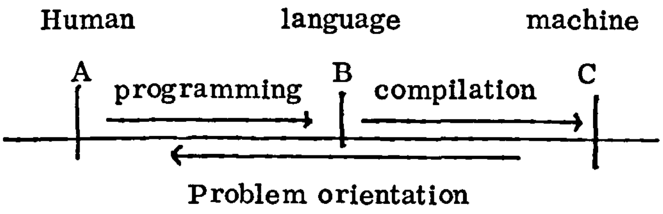
The phrase 'high level language' is often used to describe a programming language that has to be translated into a 'low level language', before it is executed on a machine. The translation is performed by a compiler which embodies the definition of the high level language in terms of the low level language of the machine's instruction set. For most high level languages and most tasks, the high level language has a higher problem orientation than the machine language and the translation can be represented graphically on a line of decreasing problem orientation.



The higher problem orientation of a high level language is the main reason for its existence. Other reasons include standardisation and portability. Before writing a program in a high level language there is a translation from the description of a task to another description of it in terms of the programming language. This translation is what we mean by the term programming. It too can be represented in a similar manner to compilation.

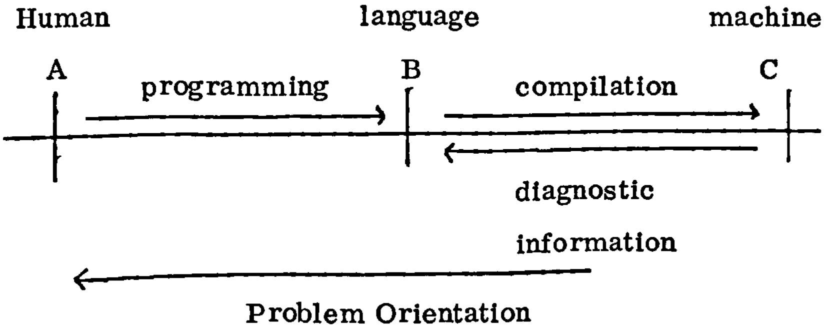


The complete process from human specification to machine language specification is then:



The effort required by the programmer is to accomplish the transition along the path A to B. With no high level language available the effort is increased to the path A to C. It is clear that the programmer's task is eased if the 'distance' A to

B is small. This is accomplished by increasing the distance B to C - by producing programming languages which have facilities for representing as closely as possible the actions and objects of the tasks to be solved. The disadvantage of such a system, as so far explained, occurs in the event of a program error. The high level language constructs are in general unlikely to occur at the machine level and once compiled, the specification of the program in the high level language is no longer used. Should an error occur during the execution of the program or should the programmer wish to locate errors by tracing the program's flow then such diagnostic facilities are available in the first instance in terms of the machine language only. The programmer uses the high level language in order to avoid the mechanics of translating a program into a machine code. Providing diagnostic information in machine level terms is not particularly helpful to a high level language user. If the programmer had used the basic machine's code then diagnostic information at that level would be useful. If this is the only diagnostic information available then the high level language programmer is actually at a disadvantage. The compiler hides the details of representing language constructs in a machine code. Given that such details of representation are hidden from the programmer they should remain hidden. For example, an error report that an illegal operation was attempted at a certain machine store location should not be reported as that. A process working in the opposite direction to compilation should map the machine language constructs back onto the high level language constructs. An error should be reported in terms of those same constructs used to write the program. The complete system can be represented as below:



The process of providing diagnostic information along the path C to B is performed by a diagnostic program which should be as essential as the compiler itself if the illusion that the distance B to C does not exist is to be fairly presented to the programmer. It could be argued that a diagnostics program could report information at a level between A and B. The programmer may have developed a set of procedures which perform certain basic tasks and in effect constitute a language of higher problem orientation than the point B. For example, an Algol 60 programmer may develop a set of routines to perform elements of list processing representing list atoms as two consecutive elements of an array. It is argued that the diagnostics package should not be expected to have the facilities to be informed of such user defined procedures. The only way of informing the diagnostics package of their function would be by a process similar to the programmer writing these procedures in a recognised language. As that process may have been in error, the diagnostics package may similarly be given erroneous information.

3.12 Conclusions

In this discussion the concept of problem orientation and its use as a graphical representation of compiling and programming has been introduced to show the necessity for diagnostics packages of a high quality. Ideally, such a package should perform a task in the opposite direction to the compiler and to the same extent. The remainder of this chapter describes a sample of currently available diagnostic systems. Features that are absent are discussed with a view to include them in a diagnostics system.

3.2 Existing diagnostic systems

Introduction

Run-time diagnostic facilities are provided to help the programmer to test a program and locate errors. The variation in the facilities provided and the way they are presented is large. This section describes several existing diagnostic systems. Whereas the sample described is by no means a complete list of all such systems it gives an indication of the variations that exist.

3.21 PL/C execution supervisor

The PL/C system comprises a 'checkout compiler' and an 'execution supervisor' {CONW73} . The language processed is PL/1. The system is orientated to a batch environment and its main philosophy is to continue at all cost. The checkout compiler is able to take educated guesses at the cause of a compilation error and, on the basis of this guess, effect a repair. The execution supervisor detects run-time errors, reports them in source language terms, makes a repair and continues the execution of the program. There are some error conditions which are considered fatal to the program run but these are few. The principal reason given for this continuation after error conditions is that in a batch environment runs of a program are infrequent and if a program's life can be extended then more useful diagnostic information may be made available and the repair may have been successful. The argument levelled at this approach is that it tolerates bad practices. If a repair is successful then the programmer may not correct the error or if the programmer is experienced in the repairs that will be provided, erroneous short cuts could be taken. It is tempting to draw an analogy with optimising compilers of which one opinion is that the programmer, knowing the compiler will optimise the program, will take no steps to produce efficient constructs. The analogy is, however, deceptive. It is true that the argument levelled at optimising compilers is much the same as that argued in the case of PL/C but optimising compilers can make many optimisations that the language does not permit the programmer to do - this is not true of the PL/C execution supervisor. Programming is a discipline and short cuts are generally to be avoided on the grounds that they may be error prone or they may adversely affect portability. It is difficult to believe that a program such as the PL/C execution supervisor will make an intelligent repair in the case of a run-time error.

For it to continue sensibly after, for example, an array subscript being out of legal range would require the execution supervisor tobe intimately aware of the program's purpose. If the supervisor were capable of this then either the program, the programmer or both are made redundant.

The diagnostics provided by the PL/C execution supervisor are good. When a program fails, either because a repair of a run-time error is not possible or the user has specified that no further repairs are required, a detailed dump is given. This dump lists all scalar and array variables giving their source program names and current values. A program trace of the last eighteen changes in flow control is provided and a count given of the number of times each procedure or label was encountered. By request prior to the program run, a trace of the program's control flow is provided.

3.22 DITRAN Diagnostic Fortran

DITRAN provides run-time diagnostics for Fortran programs. It was implemented on a CDC 1604 in 1965 {MOUL67} . At run-time, all Fortran variables are accessed indirectly through individual control blocks. Each control block contains eight components. These components include the variable's source name and run-time address. By manipulating these eight components, DITRAN is able to detect many error conditions. A variable can be flagged as undefined or not yet initialised. One component detects whether the variable is an active parameter of a DO loop so that assignments to such variables can be arrested.

When an error is detected, the message produced gives the identifier name and the position in the program where the error occurred. The location is given relative to the most recent statement label. The error messages are produced by a separate utility program which handles three hundred possible messages. Each message contains special characters which indicate positions for substitution of components such as the identifier name or the statement number.

DITRAN attempts to detect errors at compilation time whenever this is possible. Some abuses of a variable controlling a DO loop are detected at compilation time as are the uses of FORMAT statements - these are frequently not checked until run-time by most Fortran systems.

DITRAN was designed to meet the needs of student users learning to program. It appears to have succeeded in identifying areas where errors are frequent in that environment and produces good error messages. Information is collected concerning the kinds of error detected. This information is then available for further enhancement of the diagnostics.

3.23 ALGDDT Dec 10 Algol dynamic debugging system

ALGDDT is an interactive run-time package for Algol 60 implemented on the Dec system 10 {DECa} . The system allows inspection of and alteration to the values of scalar variables including array elements. Break points may be set at any statement in the program. On encountering a break point or an error, the diagnostics package is invoked. Additionally the user may interrupt the program causing entry to the package. Associated with a break point is an optional command list which can contain any commands acceptable to the diagnostics system. A common use of this facility is to type out the values of certain variables and then continue from the breakpoint automatically each time the break point is reached. A dump of all extant variables can be printed. ALGDDT provides two facilities for inspecting program flow. A trace facility will print out the most recent history of program flow in terms of procedures called and labels passed through. A profile of the program is available which lists the number of times each procedure and each label was encountered.

All variables are referred to by their source program identifier name and the syntax of commands to the diagnostics package has been designed to resemble Algol 60. The resemblance to the syntax of Algol 60 is tenuous and rather cosmetic in appearance. The facilities provided are good. The system is available to the interactive user only and presents a powerful program development and testing tool.

3.24 Glasgow Pascal diagnostic system

The Glasgow diagnostic system is a post-mortem program designed for use with Pascal programs on the ICL 1900 series {WATT77} .When a program terminates, normally or due to failure, preselected information is provided by this post-mortem program. The facilities available are a post-mortem dump, a profile, a retrospective trace and a forward trace.

The post-mortem dump is provided if the program failed. It contains a list of all extant variables and their values. The format of the variables' values is that of Pascal; for example a Pascal set is listed in the same way it would appear in a Pascal program. Arrays are partially printed, the first six and last one element of each dimension appearing. Records are expanded such that each field is displayed. Pointers are displayed as either 'nil' or the machine location they refer to. This allows the user to determine equality of two pointers but no display of the objects such pointers refer to is available.

The profile is a listing of the source program with the addition of a frequency of execution of each statement.

The retrospective trace is a list of source statements obeyed immediately prior to termination and in the order of their execution. The number of statements reported is by default fifty.

The forward trace is a history of program flow from the start of execution. For long running programs this list would be too large and two ways of controlling the trace are available. The programmer can select which sections of the program are to be traced, and the diagnostics package suspends tracing individual statements after they have been traced a given number of times - two by default.

This system is designed for batch operation. The facilities required are specified before the program run and the user, because of the batch environment, cannot request incremental diagnostics during the program run as the behaviour of the program advances. As a batch system its facilities are very good. Display of extant variables in source language terms and detailed history of the program flow gives great help to the programmer. The major failing is that only variables that are declared in the program can be listed. No structures on the heap, created dynamically, are displayed.

3.25 Algol 68R

The Algol 68R compiler is available on ICL 1900 series and was written by the Royal Radar Establishment {WOOD72} . This system is one of the worst with respect to run-time diagnostics. The only diagnostic information provided by the Algol 68R system is the current line of program input and output with an indication of how much of that line has been processed. All other messages are produced by the operating system George.

Several programs containing errors were submitted to this system. The errors given were accessing undefined scalars, running out of store and accessing a data structure via a pointer 'dangling' after that structure had been deleted.

In accessing scalar variables which have been given no value, an integer was printed, with no error being detected, as -6815692 and when accessing a real value the program halted with the message 'overflow set'.

Two methods of exceeding store limits were tried, infinite recursion and infinite data creation. Both result in the same message being printed:

```
Illegal at instruction 313: 10 3 0(2)
```

with slight variations in the final numbers.

The dangling pointer access was not faulted and the 'object' being pointed at printed out. It is clear from these examples that locating an error in this system can be very difficult. The user has to simulate diagnostics by hand and include statements at strategic positions in the program to print out various aspects of the program's state. In reference to the diagrams of problem orientation given in the previous section this system makes no attempt to relate errors detected at the machine level back to the high level language.

3.26 Atlas Autocode and Manchester in-core compilers

Atlas Autocode {BROO66} is an Algol-like language which existed on the Manchester Atlas computer. This system contained the first known instance of a diagnostic package which refers to source program identifier names {CLAR67}. In the event of a run-time error, the program was halted and a message giving the line number where the error was detected and the fault discovered. Following this was a list of all extant variable names and values at each level of procedure resting. The only scalar types in Atlas Autocode were real and integer. Arrays were not displayed by this package.

A similar system operates for the Manchester in-core compilers. The term in-core relates to the fact that they compile programs directly into core rather than creating an object code file for subsequent execution. In both the Algol 60 {MANC76} and Fortran {MANC75} systems all extant scalars are printed when an error is detected. Arrays are not printed. The local variables of all nested procedures active at the time of error are displayed along with the global variables of the main program. In the case of Algol 60, recursive calls of a procedure will be unstacked giving the variable values at each level of recursion. This display is terminated if it spans over two hundred lines. Both compilers give a message explaining the error and stating the source line number where it was detected.

3.27 ICL Cobol

The ICL Cobol compiler {ICL76} is the most used compiler on ICL commercial installations. Despite the large use of this system the run-time diagnostic facilities are almost non existant. For example, should arithmetic overflow arise a cautionary message is printed and the program continues. The manual states that, in this case, 'the result will be indeterminate'. The programmer does, however, have the opportunity to include routines in a program which will be obeyed when such an error occurs. Cobol array subscripts are not checked. The manual states that when a subscript is out of range 'the program can behave unpredictably. The programmer is advised to test the value of subscripts before using them'. In order to monitor the flow of a program, the manual recommends insertion of write statements at certain places. The manual continues by describing how such tracing of the program flow can be made more sophisticated by the programmer inserting conditional write statements. It is strange that a system so heavily used as the ICL Cobol compiler offers so little support in debugging or developing a program. The Cobol programmer invariably still relies on machine store dumps and a manual describing how to interpret such listings. The purely mechanical processes the Cobol programmer has to perform in order to cater for each program error eventuality could be readily provided by the machine - in most cases at very little cost.

3.28 GEC 4080 Fortran and Algol 60

The GEC 4000 series computers contain Fortran and Algol 60 compilers with very similar run-time diagnostic systems. In the event of an error at run-time, a message describing the kind of error is given. The location of the error is provided as a machine store location. The user has to go through a two stage process in order to map this storage location onto a source program statement. The compiled program is linked with the system's standard routines prior to execution. Examination of the link program's listing is necessary to find the procedure or program containing the given store location. The value of the store location must then be altered by subtracting the store location of the beginning of the program. This then gives the location where the error was detected relative to the start of the program. The programmer must then refer to a compilation listing of the program which contains a list of the first store location each line has been compiled to. Matching the value found from the link list then identifies the offending program source line. Neither the compilation list giving machine store locations nor the link list giving the start of the program are provided unless specifically requested. If they are not present, the program must be recompiled in order to locate an error. These systems require the programmer to delve into the details of how a program is mapped onto the machine. It should not be necessary for a programmer to do this. The purely mechanical and tedious task of mapping a given store location onto a program statement is well within the capability of the machine. The machine's prime use is to perform the mechanical and tedious tasks.

3.29 Conclusions

The above descriptions of some available diagnostic systems illustrates the wide variation of facilities provided to the programmer at run-time. Some systems providing very good diagnostics exist but it is unlikely that the average computer user is lucky enough to be using them. Like compilers, diagnostic packages are generally written for a particular language on a particular machine. For diagnostics to be universally available, a separate package is necessary for each different compiler. It is not yet generally accepted that a compiling system should include a good diagnostics package. Manufacturers proclaim that their machines can provide certain languages but it is rare to see diagnostics packages feature in such advertisements. At the present moment there appears to be little market demand to encourage manufacturers to provide diagnostics packages as a matter of course. It has already been argued that a diagnostics package of similar power to a compiler is an essential part of any compiling system. It is rare to find such capable diagnostic systems and unfortunately too common to see their total absence from some installations. This section has examined some of the facilities that can be provided by run-time diagnostics packages. The final section of this chapter analyses these facilities in order to design a diagnostic package which meets the requirements argued for.

3.3 The design strategy of a diagnostics package

3.31 Interactive and batch programming

Among the diagnostic systems described are packages designed primarily for use in a batch environment, such as the Glasgow Pascal system, and others designed for interactive use such as the Dec system 10 Algol package. These two environments provide quite different facilities. The interactive environment permits a dialogue between the user and the machine. With a batch system the man machine communication is usually an initial monologue. A batch system is useful where programs run for a long time with no need for interaction; the tasks to be performed are well defined in advance. The interactive system is more useful where the data given to the program can be decided upon after analysis of earlier results, the progress of the program can be continually adapted in the knowledge of its behaviour so far. The purpose of a diagnostics system is to obtain information about the behaviour of a program and locate sections of that program that need alteration. This process is enhanced if a continual monitoring of a program's execution is possible. During this monitoring, the state of the program can be inspected selectively and errors located by a process of elimination. For the purpose of diagnosing errors and testing programs, an interactive environment is then more suitable than a batch system . This view is supported by Bate {BATE74} who describes an interactive test bed for the Culham laboratory system development language, arguing that the most useful diagnostic facilities cannot be attained in a batch system. A batch system would provide information in the form of a trace analysis and full dump at several selected points in a program but generally the volume of such information ensuing would be prohibitively large and much time would be required to interpret it.

3.32 Facilities offered by diagnostic packages

Many facilities currently existing in diagnostics packages have been described. The main facilities provided are now discussed with the aim of identifying their usefulness in particular situations.

3.321 Program dump

The dump facility is usually associated with a batch system. A program running in a batch environment fails and a partial or complete list of that program's extant data is produced. This list may contain all such data or limited data types. The UMRCC in-core compilers as previously described, do not produce listings of arrays but just scalar variables. The reason for this is that listing all arrays in some programs would produce a prohibitively large amount of data. Other systems, for example the PL/C execution supervisor, will list all data including large arrays. A measure of the usefulness of a diagnostic facility is how readily the programmer can locate errors when using it. If the dump is very large it will take time to assimilate this information and locate that data which is relevant. Systems such as ICL Cobol also produce dumps. However these dumps are of the machine's storage locations and do not directly refer to the program's source identifiers. These dumps (often given in octal or hexadecimal format) have to be processed by the programmer who must map the program's data structures onto the dump before proceeding with diagnosing the fault. Clearly a dump is of limited use.

It may be the best diagnostic available for small batch programs but its usefulness is reduced when the dump is swamped by irrelevant information as is likely with a large program.

3.322 Program trace

Arguably the best examples of trace diagnostics are those given by the Glasgow Pascal system. This provides a trace of the specified number of statements obeyed immediately before program termination and a trace of the program flow of control from the start. This latter trace is considerably edited by avoiding repetition. The trace is used so that the programmer can check that program flow was as expected. If not, then an indication is given of where deviation occurred. The full trace of a given number of statements executed prior to failure is provided on the assumption that the event causing the error detected is likely to have been recent. This assumption is not generally valid. The error detected may be distant from the program statement that is in error. For example parameters such as array indices may be calculated in some initialisation phase for later (incorrect) use. As is the case with a program dump, a trace as described may be the only effective diagnostic available in a batch system which gives information concerning program flow events. In an interactive environment, such post-mortem diagnostics can be improved upon by a selectable monitoring of the program being produced as the program proceeds. It should be noted that the overheads involved in providing a trace can be large as a record must be kept, often on disc files, each time a statement is executed. As with a program dump, most of the information given is probably not required. When the run-time diagnostic aids were added to Atlas Autocode {CLAR67} it was found that the trace facility was not used much by programmers when the values of extant variables are given. If a program is well structured, much can be deduced about program flow if the values of variables involved in the control of program constructs are known.

3.323 Profile

A profile of a program is a program listing with a count attached to each source line or statement giving the number of times that line or statement has been obeyed. The provision of such a facility has two purposes. Firstly, it indicates those sections of a program which have been executed a large number of times and those executed a small number of times. This may indicate that, for example, a loop construct has not been executed the expected number of times, or that the incorrect evaluation of a particular expression has resulted in a section of the program being skipped. The second use of a profile is to examine the program in order to locate frequently obeyed sections. These critical regions can then be looked at with a view to increasing their efficiency. In this way a profile is an invaluable tool for tuning a correct program by locating potential sources of inefficiency. Inefficiency in itself does not necessarily affect program correctness but can be considered an error in that curing inefficient sections of a program produces a more desirable product. Producing a profile involves an overhead of time and space. This overhead is not as large as that for producing a trace, for example, but requires maintaining a vector of counts, one for each program statement.

3.324 Interrogation of data

The facility for selective interrogation of data is commonly provided by interactive diagnostic packages. The user specifies the name of a data item and the system prints its value. This compares with the batch facility of a dump in which all values are printed because no selection by the user is possible at the time of failure. The types of variable that can be printed are often restricted. Some interactive diagnostic systems may only permit the printing of scalar variables even thought other structural types are available. For example, the Dec 10 Cobol system is very similar to the Dec 10 Algol 60 system already described. However, Cobol records cannot be printed by the diagnostics package. The Dec Algol 60 diagnostics, however, does provide the mechanism for printing all types of Algol 60 variables. Clearly restrictions are undesirable and such systems do not meet the criteria, outlined in the previous section, of mapping the compiled form of the program back onto the full source language. The Dec 10 Algol 60 diagnostics system, as well as permitting the inspection of data values, allows the user to alter these values. In this way, at a particular point in the program, a value that is not as expected can be altered using the diagnostics system and the program continued.

3.325 Control and monitoring of program flow

With an interactive diagnostics system, a means of interrupting the program flow is desirable. The user can then specify a condition for interruption of the program execution in order to call upon the diagnostics package. A common method of implementing this facility is for the programmer to specify a particular program statement. When this statement is reached, the diagnostics package is entered and the user can then interrogate the state of the program. The program may then be resumed at the point it was suspended. This facility, often called a break point, permits the user to freeze the program execution at several points in its progress and thereby monitor the actions performed. Some diagnostics packages permit the resumption of program control at a different location to that of the break point - in effect simulating a goto statement at any position in the program. This facility can be used if it is found that program control has deviated from that expected. The ability to interrupt program flow is necessary in order to test the program. The diagnostics package will be called upon in the event of an error but inspection of the state of the program on earlier occasions aids testing the correctness of a program. This facility can be regarded as similar to the assert command described in the previous chapter where at certain points in the program the various data items can be checked to verify they contain the correct values.

3.33 Conclusions and objectives

Having discussed the need for run-time diagnostic facilities and described currently available systems, the objective is to design a diagnostics package to meet the criteria specified. The objective is formulated by discussing facilities that are not generally available and those currently in existence that are considered useful.

3.331 Alteration of the program

Most diagnostic facilities described are passive - the user monitors the program at a distance. Two facilities described are active, the alteration of data values and program control. Both of these are equivalent to a temporary alteration of the program - the first by emulating an assignment statement and the second emulating a goto statement. These techniques should not be regarded as diagnostics but program repair. It can, of course, be argued that they are of a diagnostic nature because they can be used as a 'try it and see' technique. It is argued that not only do they not properly fit into the role of diagnostics but also they create an environment which is itself error prone. The temporary alteration of a program, by patching in extra statements which are invisible, can lead to confusion if used frequently. The purpose of a diagnostics system is to map the compiled program back onto the source language. Altering the meaning of a program is outside this objective. A program can be more properly altered at its source level and recompiled. It would be better if this process of editing the program and recompiling can be made as easy as possible but diagnostic packages permitting alteration of data and program flow do not do this - they allow the temporary insertion of an assignment or goto statement. It is likely that such a simple insertion is not the best repair of an error and allowing it would tend to persuade the programmer to think in terms of those simple repairs when finally correcting the program. The ready availability of these repairs could be used to fix many errors. Having done this, the programmer is likely to insert such repairs in the program with the probable result of destroying any structure or clarity that previously existed. For these reasons such facilities are not considered to belong in a diagnostics package.

3.332 Currently lacking diagnostics

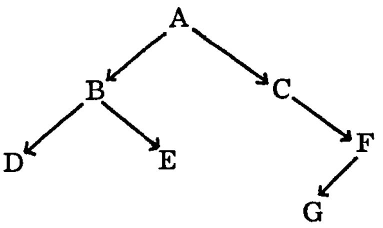
Diagnostic facilities afforded by systems such as the Dec 10 Algol 60 system and the Glasgow Pascal package are probably complete for languages like Algol 60. The values of all variables can be inspected and flow monitored. The area that has not been catered for is the suitable inspection of dynamically created variables. Where languages contain data objects that are pointers to other objects there are no suitable diagnostics available. The two problems that occur are:how is the value of a pointer described and how are the interconnecting structures that can be created, linked by pointers, displayed?

The value of a pointer.

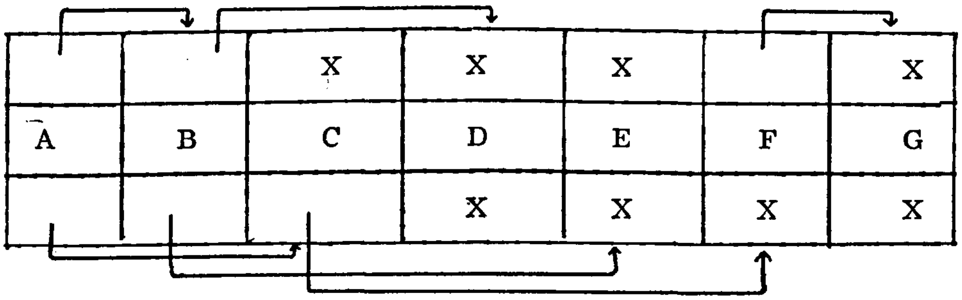
In practice, a pointer is implemented as having as its value the machine store location where the object being pointed at is stored. If the user is to be free of such implementation details then this value is not very useful. The main problem that exists is that objects referred to by pointers have typically been created by invoking some dynamic store allocation mechanism. Such dynamically created objects do not appear in the declaration section of the program and therefore have no name the user can refer to them by. It is suggested that as such objects are created they are given unique names. These names are then available to the user of the diagnostic package. Additionally these names are the values of the program's pointer variables. The names chosen must be unique to avoid confusion. The actual store location address could be used but apart from being unnatural it could lead to abuse by the programmer referring to a location that is not a valid address if stringent checks were not made. It is suggested that the names generated are simply ascending integers. Most languages forbid the use of integers as identifiers and therefore no confusion with already declared identifiers would arise. The generation of alphabetic names could result in such confusion and additionally would inevitably produce names which are unpronounceable and hence difficult to remember. The assignment of consecutive integers as names to objects as they are created has the bonus that they indicate the chronological ordering of such objects which may be useful information.

Display of interconnected structures.

Programming languages that provide pointers permit an infinite variety of data structures to be created. Typically such a structure is composed of objects which contain primitive data fields and one or more pointers to other such objects. The content of these objects is of interest as is the manner in which they are linked. Consider a binary tree structure. The contents of the individual nodes is only part of that tree; the shape of the tree as envisaged by the programmer diagramatically is also of interest. Consider the binary tree shown:



This may be represented in store as



		X	X	X		X
A	B	C	D	E	F	G
			X	X	X	X

It is more meaningful to the programmer to be provided with a display, as in the first diagram, constructed from the store layout of the second. Displaying such a structure, particularly more complex structures, presents certain difficulties. A graphics tube would be the most appropriate medium but should not be considered as a generally available device yet. To display a structure on a line oriented device, such as a visual display unit or teletype, requires formulating a display without drawing interconnecting lines but not losing definition of these connections. Consider a data structure composed of objects each containing n pointers. Each object is represented by m characters on the same line. If the maximum depth of this structure is known to be d levels then at the bottom level the maximum width of display to contain this structure is given by:

$$w = mn^{d-1}$$

By using this width, the structure can be arranged with its root occupying a position midway, at position w/2. This width w is then split into n equal parts. Each of the n pointers of the root are allocated a width w/n and displayed midway within this width. This algorithm is repeated at each level down the tree. Consider the tree structure above. For this object we have:

$$n = 2$$

$$m = 1 \quad \text{say}$$

$$d = 4$$

$$w = mn^{d-1} = 8$$

The width of eight is then allocated as shown:

1	2	3	4	5	6	7	8
			A				
	B				C		
D		E				F	
						G	

No confusion arises as to the intended connections between the nodes as each position is unique. It is noticed however that the symmetry of the tree has been distorted. This arises because when n, the number of pointers, is even, they cannot be symmetrically positioned underneath the root. To overcome this problem, the algorithm is modified so that:

```
n: = number of pointers;
if n is even then n: = n + 1
```

For the above tree, the parameters become:

$$n = 3$$
$$m = 1$$
$$d = 4$$
$$w = mn^{d-1} = 27$$

and the resulting diagram is:

													A													
				B																		C				
	D						E																		F	
																								G		

The extra pointer introduced to ensure that n is an odd number is positioned midway between the actual pointers of the object composing the structure and is taken always to have a nil value - pointing to nothing.

This algorithm for displaying a structure can be generalised to include objects of different types linked together. Thus the display can consist of different objects each containing differing numbers of pointers.

Mapping the display

The above algorithm can produce a matrix display of any data structure no matter how large. The question of accommodating a cycle in a structure where at one level a pointer points back up to a higher level is dealt with later. The problem then is to map this display onto a physical unit such as VDU. The matrix produced is of dimensions d x w. If the VDU screen has dimensions h x b then the display is easily printed if:

$$h >= d \quad \underline{\text{and}} \quad b >= w$$

If b is sufficiently larger than w then the number of characters used to describe each object m can be increased in order to provide more information about each object. The main problem arises when the above relationship is false. In this case, two tasks are initially undertaken. Firstly the display is examined to find the maximum width. This will usually be less than w - the actual width will only be w in the case where all leftmost and rightmost pointers are pointing at an existing object. This actual width w' is then used. The second task that can be performed is to reduce m - the number of characters used to describe each object. Having performed both these tasks, the display may still be too large. In this case, two techniques exist to print the structure. These are referred to as 'squashing' and 'windowing'.

Squashing

As the name implies, this process consists of printing a form of the display in a smaller area than that of the display. If the display matrix is regarded as being somehow elastic, it is pushed in until it fits the matrix of a VDU screen. For the display of dimensions d x w and the VDU screen of dimensions h x b this squashing is achieved as follows:

> Consider each cell of the matrix h x b with coordinates x, y. Into this cell must be placed s cells from the matrix d x w. In order to ensure that s is integral, the scaling used to squash the display is integral. In general, to map a vector of length p onto one of the smaller length q, the ratio employed is:
>
> $$(p + q - 1) \div q$$
>
> Then an integral number of cells from p is placed in each cell of q. This process will only result in utilising all of q if q is a multiple of p but guarantees that at least half of q is used. Using the above formula separately for the depth and width of the display produces two scaling factors. The number of display cells mapped onto each cell of the VDU is then the product of these two scaling factors.

Clearly this technique of squashing the display loses information. In order to minimise this information loss, the character printed on the VDU screen for each screen cell should indicate the number of cells that were present in the block of display cells squashed into it. This technique does allow the user to see an overall shape of a structure.

Windowing

Windowing consists of the user selecting a sub-matrix of the display that can be mapped directly onto the VDU screen. Having viewed the entire data structure as squashed onto the screen, the user can then move such a window about this display in order to examine the details in each part of the structure.

Cycles in data structures

The possibility of a cycle appearing in a structure creates the problem of representing this eventuality on a two dimensional display. Ideally where an object points to another object at a higher level this is displayed in three dimensions by folding back the two dimensional display onto itself. This would not be possible to display on a device such as a VDU screen without ambiguities arising from the lack of definition. A VDU screen is typically a matrix of 22 x 80 characters and leaves little scope for such three dimensional projections. Instead it is suggested that where a cycle is found, the object pointed at is displayed, but printed such that it is clear a cycle is present and where this cycle leads back to. This involves the repetition of display of one object for each cycle.

3.333 Summary

The discussion of various diagnostic facilities has led to a description of a diagnostics package. While many facilities to be included in such a package already exist in other such systems, the inclusion of structure display is not found elsewhere. The ability to interrogate the values of objects created dynamically during the execution of a programme is included and facilitated by assigning unique names (integers) which serve to identify such objects and indicate their age. In order to build such a diagnostics package, a language and compiler must be chosen. While it is hoped that the ideas inherent in such a package are not restricted to one compiler or one language some implementation details clearly will be. It was decided to use the language Pascal {WIRT75} . This language was chosen for several reasons. Pascal allows the formulation of programs in a natural and structured manner. It has already been argued that programming in a well structured form is likely to produce more readable programs that can be verified more readily. It would be wrong to abandon this ideal for no reason. Pascal permits dynamic creation of objects and allows data structure linking by pointers.

This is essential to test the routines for display of such structures. Finally, Pascal is chosen as the language to be used locally for teaching and research and so a large number of users are then available to test the success or failure of this diagnostics package.

The diagnostics package is to be an interactive system providing the following facilities as discussed above.

1) Display of the shape of linked data structures.
2) Interrogation of the values of dynamically created objects.
3) Interrogation of all Pascal data types in terms of the source language.
4) A facility to set a break point in a program to enter the diagnostics package at that point.
5) A program profile obtainable at any selected point during the program run.

It is believed that these basic facilities, probably enhanced as experience of their use is gained, will provide the Pascal programmer with a powerful tool for testing and debugging programs. At any selected point in the program flow all extant data items may be examined and a profile of the program at each point will aid verification of the program's state. It is thought that the expensive overhead of program tracing can be avoided by judicious use of break point facilities.

In the next chapter, implementation of Pascal is described. While implementing Pascal is seen as building the apparatus to test a diagnostics package, it in itself provides opportunity to discuss compiler implementation and portability. Finally the implementation of this diagnostics system is described and the extent to which it has achieved its aims is discussed.

CHAPTER FOUR

IMPLEMENTATION OF PASCAL

4.0 Introduction

This chapter describes the implementation of the Pascal P4 Pcode compiler. Problems of implementation of this compiler in general are discussed and some difficulties relating to the individual machines used highlighted.

The implementation of Pascal was required to have available an interactive high level system containing run-time diagnostics. Pascal was also required locally for undergraduate computer science teaching and so the system had to be efficient to cope with this large load.

It should be explained why two implementations were necessary. When the first compiler was started, the only interactively available machines accessible to Keele's Computer Science Department were the Department's own Digico Micro 16E {DIGIb} and the central service department's Elliott 4130. The 4130 was due for replacement and so its future was short. It was not known either when it would be replaced or what its replacement would be. The Digico was chosen as the only viable prospect and Pascal was implemented. This implementation is used for undergraduate teaching but can only support one user at a time. By the time the central service 4130 was replaced by a GEC 4080 {GECa} it was clear that full implementation of Pascal with run-time diagnostics would not only be less tedious on the 4080, due to the technical limitations of the Digico, but also more widely used and useable. The technical limitations on the Digico are lack of system software support and restrictions (imposed by the hardware) on addressable memory.

Pascal was then implemented on the GEC 4080. This exercise was quite short compared to the first implementation in that it was only a matter of a few weeks before a Pascal program was running on the 4080 albeit with an extremely crude system. The system was then refined to eliminate some of the defects of the P4 Pascal and provide a usable system. This process was by far the most time consuming but essential if the system is to be used by many undergraduate students. Some enhancements were made either because they were required by the run-time diagnostics system (which is a Pascal program) or because they increased the uses that could be made of the language in a University environment.

Considering the system is interpretive it has proved acceptable to use in a teaching class of 16 students. The system has been in service for several months and shown itself to be extremely reliable.

The two implementations described here have provided an insight into the problems encountered in implementing a language on less than ideal architecture and the general problems of portability. The question of portability is dealt with separately in the next chapter.

4.1 The P4 compiler

The Pascal P4 compiler is the fourth version of a compiler known as the Pcode compiler'. This compiler is written in Pascal. The original authors are Urs Ammann, Kesav Nori and Christian Jacobi from the Institut Fuer Informatik Zurich. {WIRT71c} , {JACO76} .

The P4 compiler was completed in 1976 and is used in many currently existing Pascal systems. The fact that this compiler is so widely used is born out by correspondence in Pascal News {MICK} , where the vast majority of new implementations announced are based on the P4 compiler and the 'implementation section' concentrates on discussion about methods of implementing this compiler and improvements to it.

4.11 General

The attraction of the P4 compiler to a potential implementer of Pascal is twofold. Firstly it is recognised as being the most expedient way of implementing Pascal on a machine; and secondly its huge popularity means that there exists much experience of implementation. A large group of people are constantly producing corrections to it as improvements are made and errors within it detected.

The compiler is a one pass system producing an object file in assembler-like mnenomics. The object code is called Pcode which is a stack oriented language of relatively high level and complexity when compared to commonly existing machine codes. Until quite recently no machine existed which executed Pcode directly and as a result, the resulting Pcode is usually interpreted by a program written for the host machine.

In order to implement the P4 compiler, two basic paths exist. The first and most commonly used is to write a Pcode interpreter for the host machine to simulate a Pcode machine. The second is to alter the compiler to produce object code of the potential host machine.

The second method is by far the lengthier and more difficult. It is recommended in the implementation notes {JACO76} where the expected Pascal programs to compile are relatively large (greater than 500 lines). For large programs, interpreting the compiler leads to high compilation times which may be unacceptable. The effort required to alter the code generation of the P4 compiler to suit a particular machine is much greater than that required to build a Pcode interpreter. Two major problems exist in this approach. Firstly the compiler produces code in a reverse Polish manner suited to a machine with good stack operations and so a substantial alteration would have to be made to the compiler to produce infix code. Alternatively, a second pass could be made on the code to convert it to infix using techniques described by Rohl {ROHL75} . This second pass would however increase the effective compilation time and ambiguities concerned with whether objects are loaded onto the stack as part of an arithmetic expression evaluation or to create the parameter section of a stack frame, as described later, would need to be resolved. The second problem in converting the compiler is that it is based in a quite sophisticated object code. Pcode contains several complex instructions - all set functions such as set intersection are one Pcode instruction - leaving much less work for the compiler.

The first implementation method was then chosen as the easier and most expedient course. The result is quite acceptable for student programs which are the main users the system is intended for.

As mentioned above, the P4 compiler is written in Pascal itself and as such cannot be executed without a working Pascal compiler being in existence. If we assume such a working compiler does not exist on the intended host then the compiler must be executed, initially, on some other machine which does have a working Pascal compiler. These are referred to as the 'cross compiler' and the 'cross machine'. In the case of the first implementation (on the Digico Micro 16E) the P4 compiler was first executed on the CDC 7600 at the University of Manchester Regional Computer Centre (UMRCC) which is connected to the North West Universities' network.

4.12 General Implementation

The implementation proceeds as follows. Firstly, a second form of the P4 compiler is created to be compiled by the cross compiler. The reasons for this are twofold. Firstly the P4 compiler accepts a slightly different form of Pascal to that described in the Pascal report {WIRT75} . These differences are detailed in appendix 1. In particular, the P4 compiler recognises different standard procedures for heap disposal and these will not be accepted by the cross compiler. The second reason for needing a modified P4 compiler is the different character sets on the cross compilermachine and the host machine. These differences are explained in more detail in the next chapter concerning the portability of this compiler.

This modified P4 compiler is then compiled by the cross compiler. The resulting running program then acts on the original P4 compiler to produce the Pcode instructions corresponding to this compiler.

The final stage is the transfer of the Pcode instructions to the host computer and their subsequent interpretation. This process can be described diagrammatically.

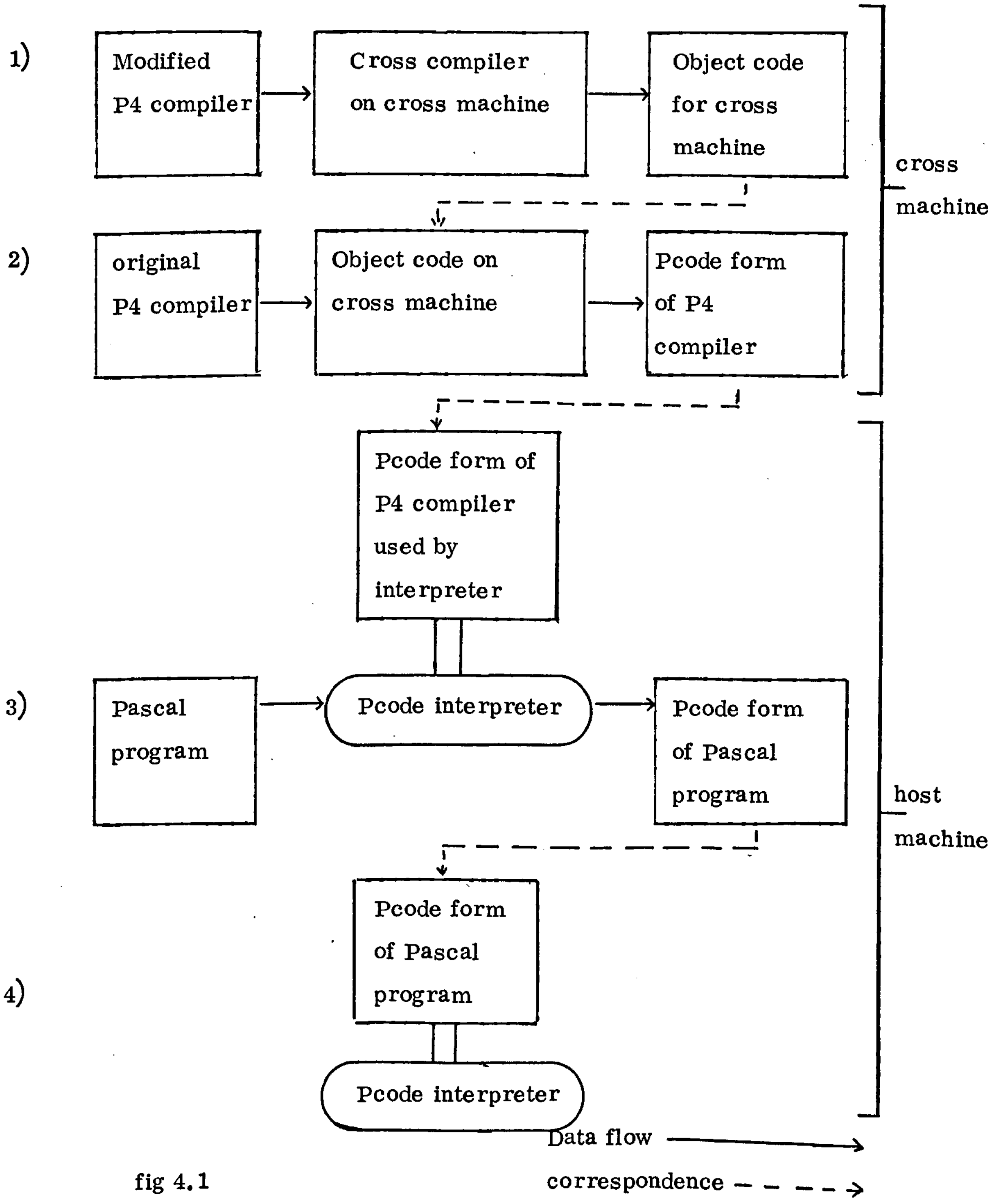

1)
Modified P4 compiler
Cross compiler on cross machine
Object code for cross machine
2)
original P4 compiler
Object code on cross machine
Pcode form of P4 compiler
cross machine
Pcode form of P4 compiler used by interpreter
3)
Pascal program
Pcode interpreter
Pcode form of Pascal program
host machine
4)
Pcode form of Pascal program
Pcode interpreter
Data flow
correspondence

fig 4.1

Stage 1 is that of the modified P4 compiler being compiled by the cross compiler to produce an object code form of the P4 compiler which can be executed on the cross machine.

Stage 2 is the execution of the modified P4 compiler compiling the original P4 compiler and producing the Pcode instructions corresponding to it.

Stage 3 is the use of this compiler on the host machine. When the Pcode instructions generated by stage 2 are transferred to the host machine, the Pcode interpreter interprets them forming a running Pascal compiler. Pascal programs are compiled into Pcode.

Stage 4 shows the subsequent execution of the Pcode instructions, produced in stage 3, by the same Pcode interpreter.

As stated above, in the first implementation the cross machine was a CDC 7600 at UMRCC, being the only machine readily accessible to Keele University at that time (December 1977) with a Pascal compiler. The host machine was a Digico Micro 16E minicomputer. The fact that the two machines are not immediately physically linked meant that the above description of this process is a simplification of the actual task but serves to show the general mechanics of it.

The P4 compiler is well suited to different machines' addressing structures and each of the standard types in Pascal is described by the definition of two constants. The first is the size (in whatever basic address units the interpreter is written to deal with) that a variable of that type will occupy. The second is the alignment (in the same basic address units) that the compiler will use to allocate storage addresses. In general, the compiler handles such alignments sensibly; packing variables together is allowed but several bugs in the compiler were discovered when implemented on the GEC 4080 due to oversights by the compiler's authors when dealing with certain combinations of declaration of standard types which occupy a smaller amount of space.

4.13 Operation of the P4 compiler

4.131 General

Pascal is parsed as an LL(1) grammar. That is, the compiler has a top down parser and requires one symbol look ahead. The compiler requires just one pass of the source program and produces Pcode with mnemonic instructions and labels. The compiler is approximately 4,500 lines of Pascal which compiles into almost 17,000 Pcode instructions.

4.132 Lexical analysis

The compiler contains a procedure named insymbol which reads and interprets one or more characters from the source program to produce the next Pascal symbol. Insymbol, when activated, will update the values of eight global variables. These variables are the only interface between the lexical and syntactic analysis. They are:

1 SY This variable describes the symbol. Its possible values dictate whether the symbol is an identifier, a constant, an operator, or a Pascal reserved word. If the symbol is an operator or reserved word, SY will determine which word, or the group of operators the symbol belongs to.

2 OP If the symbol is an operator, OP will determine which operator.

3 VAL If the symbol is a constant, VAL is the value of the constant.

4 LGTH If the symbol is a string constant, this variable gives the length of the string.

5 ID If the symbol is an identifier this string variable is that identifier's name.

6 KK This is the number of characters (in ID) which constitute the identifier.

7 CH This is the last character processed.

8 EOL A Boolean variable, true if the scanner has processed the last character of a line.

As can be seen, for a particular symbol, not all eight variables will be meaningful.

Internal to insymbol are two procedures nextch and options. Nextch's function is to read the next character and assign it to CH, set EOL if this character is the last on a line, print the character on the listing device and call a procedure to report compilation errors if EOL was true when nextch was called. Options is called to handle compile options which are embedded in comments.

Whereas character and integer constants are represented as chars or integers, insymbol represents real constants as a string of characters. In this way the compiler does not use any variable of type real. This feature is presumably to cater for efficient implementation of the Pascal compiler on machines with no floating point hardware as well as allowing a Pascal system on such a machine to operate initially without catering for floating point instructions.

4.133 The name table

The compiler represents identifiers and types using two record types as shown:

```
type  CTP        = ↑ identifier;   STP = ↑ structure;
      identifier = record
                   name: array  [1 . 8] of char
                   llink, rlink, next: CTP;
                   idtype: STP;
                   case  klass: idclass of
                   konst: (values: Valu);
                   vars  : (vkind: (actual, formal);
                            vlev, vaddr: integer);
                   field: (fldaddr: integer);
```

```
                    proc, func : (case pfdeckind:declkind of
                                  standard : (key : integer);
                                  declared : (pflev, pfname : integer:
                                               case pfkind : (actual, formal) of
                                               actual : (forwdecl, extern:
                                                          Boolean)))
          end;

          structure =   record
                        size : integer
                        case form : structform of
                        scalar : (case scalkind : (standard, declared) of
                                           declared : (fconst : CTP);
                        subrange : (rangetype : STP; min, max : valu);
                        pointer  : (eltype : STP);
                        power : (elset : STP);
                        arrays  : (aeltype, inxtype : STP);
                        records : (fstfld : CTP; recvar : STP);
                        files : (filtype : STP);
                        tagfld : (tagfieldp : CTP; fstvar : STP)
                        variant : (nxtvar; subvar : STP; varval : valu)
                        end;
```

The type 'valu' is a record giving the value and type of a constant.

For example, consider the following Pascal declaration:

```
          var a : array [1..10] of char;
```

This would produce the following internal representation:

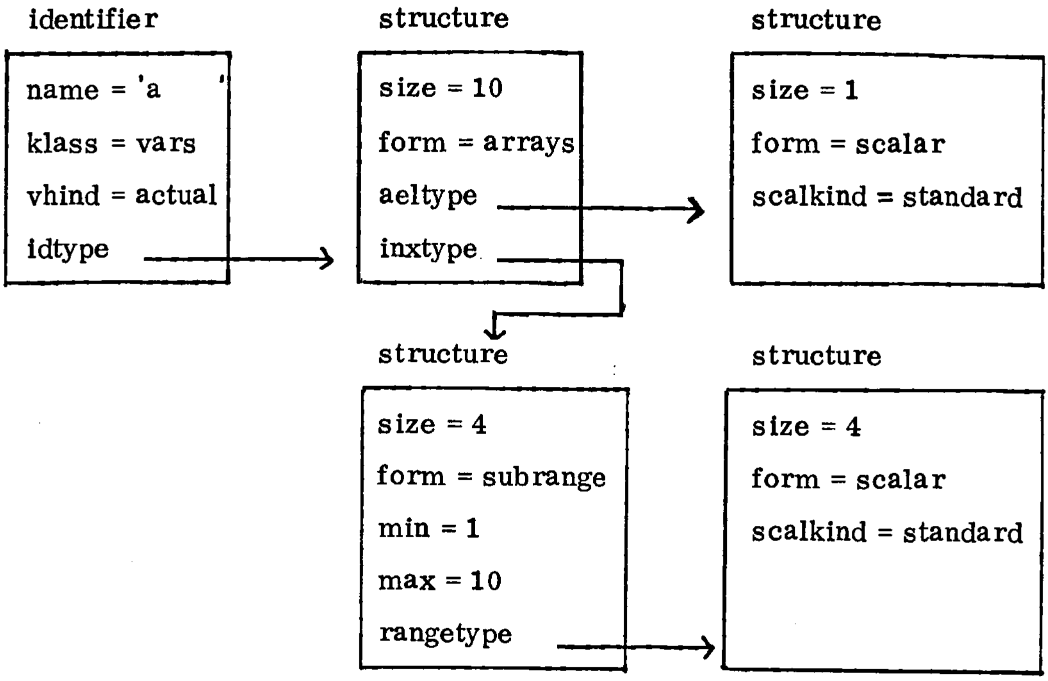


fig 4.2

The two'structures' on the extreme right are the predeclared objects defining the types char and integer.

In the definition of the type 'identifier', the field 'next' is used in three instances. Firstly if the 'identifier' is a procedure or function, the 'next' field points to the 'identifier' of the first parameter: the parameters are linked by their 'next' fields in order. Secondly within a record declaration, the 'identifiers' for each field are linked in order of declaration (for parsing a call of the standard procedure new which contains definitions of the variants required). Thirdly, the 'next' field is used when parsing a multiple declaration such as:

```
var a, b, c : integer;
```

Because the compiler has only one symbol look ahead, the objects of type ' 'identifier' for a b and c will all have been created before their type is known to be integer. These three 'identifier' records will be linked by the 'next' field. This chain is then followed filling in the 'idtype' field as integer.

The name table is built on an array of pointers to identifier's. This display has one entry for each level of declaration and the entry points to the 'identifier' corresponding to the first declared at that level. This 'identifier' is the root of a binary tree which contains all the 'identifier's declared at that level. The tree is linked by the two fields llink and rlink. When searching for an identifier, the display is scanned from the current level down to the zero'th level to obtain the most recently declared 'identifier' of that name. At each level, the tree, ordered alphabetically, is searched until the required identifier is found. The zero'th level of the display contains the 'identifier's for all predeclared Pascal variables, constants, types, procedures and functions.

Identifiers which are fields of a record are only accessible within the context of that record and therefore cannot be placed in the display at the same level. When parsing a record declaration, the level of the display is incremented and so all the fields form, their own ordered binary tree in isolation. The root of this tree is then assigned to the field 'fstfld' of the 'identifier' corresponding to the record. This approach eases the parsing of a Pascal with statement. For the duration of the with statement, the level of the display is incremented and the fields' tree hooked onto it from the 'fstfld' of the named record. Thus the fields of the record will automatically be found in the display before any other identifier which may have the same name.

4.134 Expression evaluation and code generation

While compiling an expression, the compiler uses a global variable named 'gattr' to hold the attributes of the expression as it is parsed and code generated. This variable is of type attr defined as:

```
type    attrkind  =  (cst, varbl, expr);
        vaccess   =  (drct, indrct, inxd);

        attr      =  record
                     typtr : STP
                     case kind : attrkind of
                        cst:  (cval : valu);
                      varbl: (case  access : vaccess of
                               drct : (vlevel, dplmt : integer);
                               indrct : (idplmt : integer))
                      end;
```

Where valu and STP are as defined above.

Attrkind gives the kind of expression as parsed so far. The three possibilities are a constant, a variable or an expression. In the latter case code has been generated so that the value of the expression is on top of the Pcode stack. Vaccess gives the method of access if the expression as parsed so far is a variable. The access is either direct in which case the level and address within that level of the variable are held as vlevel and dplmt, or indirect in which case the address of the object is on top of the stack and is to be offset by idplmt. The access inxd is not used. Its existence is historical and an earlier version (either P1, P2 or P3) of the compiler used it for array indexing.

Consider the statement:

```
b : = a [ 3 ] .p;
```

The compiler will parse this as follows:

Current symbol	state of gattr	code generated
b	kind : = varbl access : = drct vlevel, dplmt set to level and address of b. idtype set from b.	
: =	save gattr in variable lattr	load address given by vlevel, dplmt
a	kind : = varbl access : = drct vlevel, dplmt set to level and address of a. idtype set from a.	
[	access : = indrct idplmt : = 0	load address given by vlevel, dplmt
3	kind : = cst cval : = 3 kind : = expr	load constant integer 3
]	kind : varbl	multiply top of stack by size of element of a, add result to address below top of stack.
.		
p	idplmt : = idplmt + field address of p	

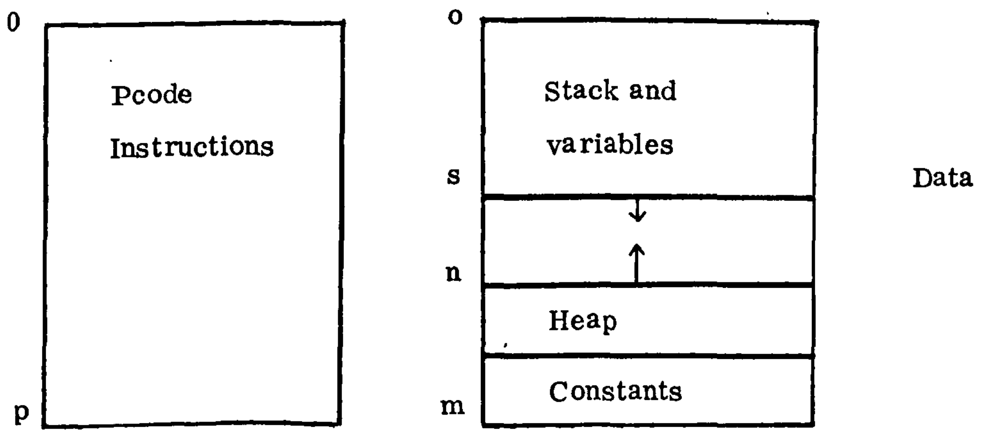
;	kind : = expr	load value whose address is that on top of stack + idplmt, store in address given beneath the top of stack.

The field typtr is altered accordingly at each stage and used to check the legality of the statement.

4.2 The Pcode machine

4.21 General

The P4 compiler produces object code known as Pcode. There are 61 instructions and 23 standard procedure calls. This instruction set was modified to accommodate 128 possible instructions when the second implementation was written for the GEC 4080. Both instruction sets are listed in appendix 2. Many of the original 61 instructions contained a field indicating the type and therefore size of the object to be manipulated. This field was eliminated and each instruction that required it expanded into a separate instruction for each object size. The reason for this decision was purely that of speed. Rather than introduce a test on the type field, which would have occurred frequently as these instructions are the most common, it is much faster to branch immediately to a tailor made routine in the interpreter for each type. The Pcode machine implemented has the following storage layout:



p	number of Pcode instructions accommodated
s	current size of the stack
m	data store size
n	marks the heap size

Fig 4.3

The heap is the data area used for dynamically created variables - created by the Pascal procedure 'new'. The heap and stack grow to meet each other. The Pcode machine has five registers:

PC	Pcode location counter
SP	Stack pointer
NP	New (heap) pointer
MP	Mark pointer
EP	Extreme pointer

The two store areas of code and data respectively are regarded as two distinct vectors with indices ranging from zero to the implementation fixed maximum size. PC is an index into the code vector and the other four registers are indices into the data vector. The functions of these five registers are as follows:

PC

The program counter is a pointer into the code vector indexing the Pcode instruction that the interpreter will execute next.

SP

The stack pointer is a pointer into the data vector marking the topmost object on the stack.

NP

The new pointer indicates the next free location on the data vector for use when the Pascal procedure 'new' is called. The pointer is decremented by the size of the object created.

MP

The mark pointer is an index to the base of the 'stack frame' in the data vector corresponding to the level of the procedure being executed. It is used as the base address for accessing local variables.

EP

The extreme pointer is the largest value that the stack pointer can possibly have during the current procedure. It is altered on entry to and exit from a Pascal procedure or function. The reason for the existence of this register is for testing for store overflow. Without such a register, on every instruction that required a growth of the stack, the stack pointer would have to be compared with the new pointer (NP) in order to detect the stack and heap meeting. By setting up the extreme pointer on entry to a procedure it need only be compared with the new pointer once for each procedure entry. Similarly, whenever a call on the heap is made and the new pointer decremented it is compared with the extreme pointer to check for the possibility of store overflow.

Stack frames

Whenever a procedure or function call is executed, a stack frame is added to the top of the stack and the stack pointer updated accordingly. The compiler implements the initial program entry as a procedure call. The form of the stack frame is as follows:

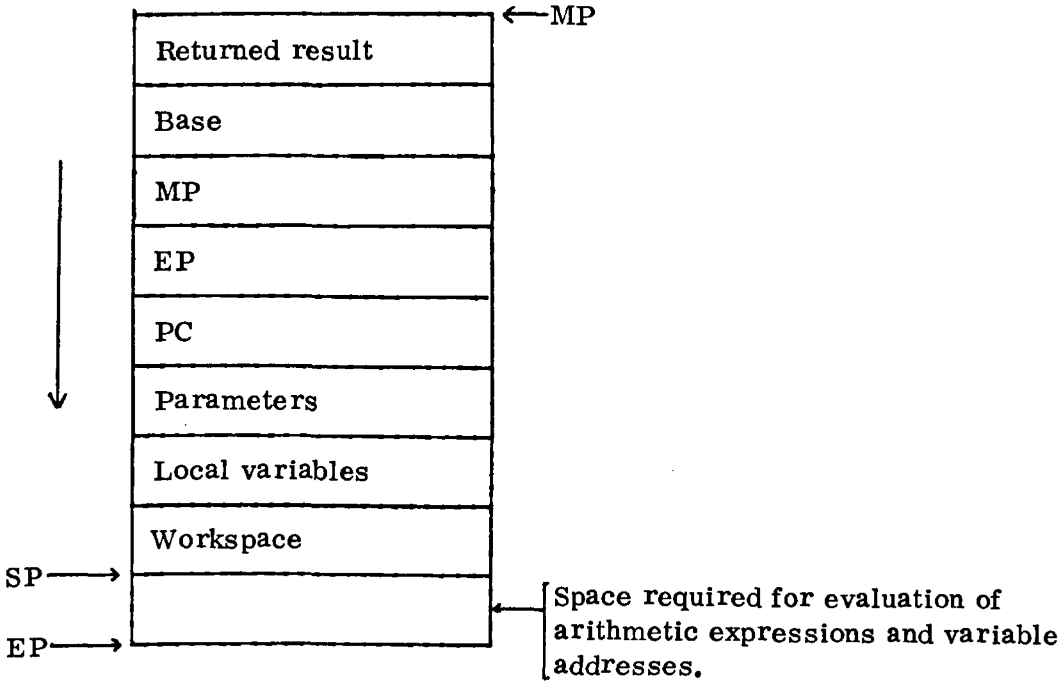

MP
Returned result
Base
MP
EP
PC
Parameters
Local variables
Workspace
SP
EP
Space required for evaluation of arithmetic expressions and variable addresses.

Fig 4.4

The first entry is relevant to function calls only but is always present to avoid treating function and procedure calls differently. The size of this section is that of the largest type of value a function can return; this is usually the size of a Pascal set (in the GEC implementation this is 16 bytes long). When the function is assigned a value it is placed in this section. On exit from the procedure or function, the stack pointer is adjusted so that the required object is left on top of the stack. In the case of return from a procedure, the required size is zero. Thus the net effect of a function call is to load a variable of that function type onto the stack.

The next section of the stack frame is the 'base'. This forms the 'static link' of procedure/function calls. The value of base is always that of the start of the stack frame of the procedure/function one nested level down as declared in the Pascal program.

Following the base the three registers MP, EP and PC are saved for restoration on exit from the procedure or function. The mark pointer always points to the base of the current stack frame and so the saved values of the mark pointer form the 'dynamic link' linking all stack frames currently on the stack.

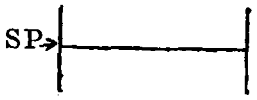
The next section is for storing any parameters in the procedure/function call and its size is dependant on the number of parameters. Following this is the space required for all variables declared local to the procedure/function.

The final workspace is only present when the procedure or function contains one or more with statements and the record(s) used on the statement(s) were passed as parameters by reference. When this reference to a record is passed, its base address is stored in a part of this workspace for later use within the with statement.

Example of procedure/function call

To illustrate the use of the stack frame, the following example shows the mechanism of procedure call and return on the Pcode machine.

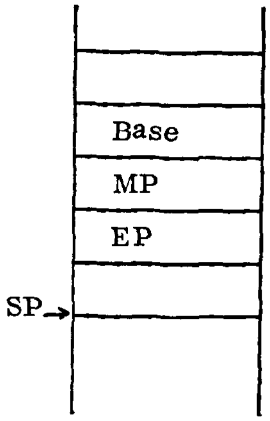
Pcode machine actions | State of stack | Comments



before the call

Mark stack

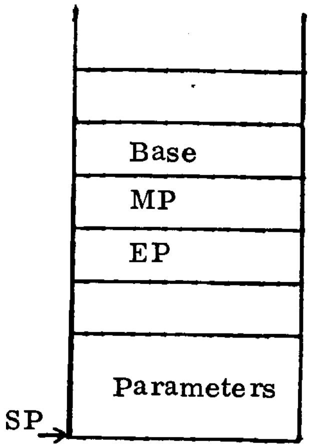
Calculate the base of the stack frame of the procedure/function one nested level down from the procedure being called. Reserve space for possible function result. Save the calculated base and the registers MP, EP on the stack. Adjust the stack pointer to leave one location free for subsequent storage of the return program counter PC



Pcode machine actions   State of stack   Comments

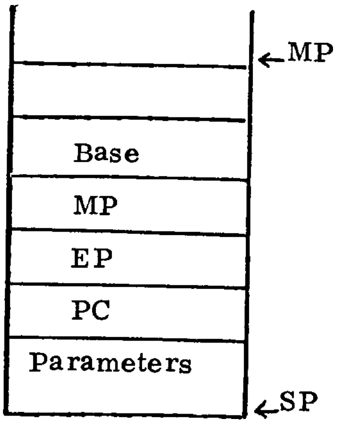
Load parameters

Load on top of stack all parameters required by the call



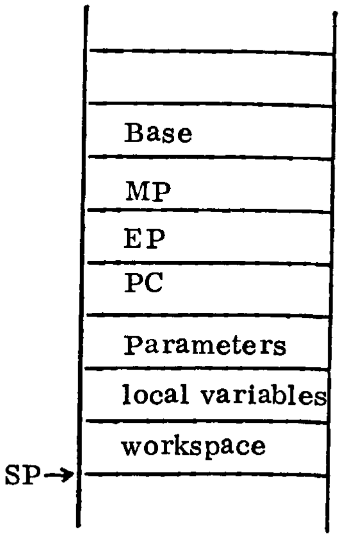
Call procedure

Save the program counter in the location reserved. This location position is calculated by subtracting the size occupied by the parameters from the SP. Adjust MP to point to the base of the frame and alter PC to the start of the procedure



Entry 1

Reserve space on the stack for local variables and any work space



This is always the first action on procedure entry

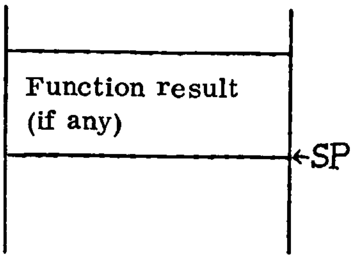
Entry 2

Add on to SP the amount of stack required by this procedure as calculated by the compiler. Store this value in EP and check it against NP for store overflow.

This is always the second action on procedure entry.

Procedure / function exit

Assign the value of MP adjusted by the size of the function result to SP. Restore EP, MP and PC from the stack frame

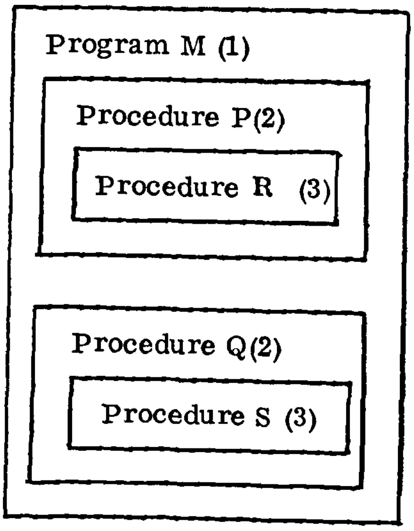


For a procedure return, SP will point to below the stack frame

fig 4.5

4.22 The use of MP and Base as stored on the stack frame

The Pascal language allows the nesting of procedures and functions in a program. At each depth of nesting, the level of declaration of variables increases by one. Variables declared in the main program (global variables) are said to be at level 1. Predeclared variables are at level O. Variables declared in some procedure P are said to have been declared at level 2 and variables declared in some procedure Q which is itself declared within procedure P are said to have been declared at level 3 and so on. At any point in a program, the current level can be determined. The scope rules of Pascal, as in most other block oriented languages such as Algol 60, permit reference to any identifiers declared at that level and at the lower levels surrounding the current level. For example, consider the following program layout:



The numbers are the lexical level of declaration of any identifiers within that procedure.

fig 4. 6

Within procedure R, all identifiers declared within R, P and M are in scope. Identifiers within Q are not because although declared at a lower level of 2, procedure Q does not surround procedure R. The purpose of the base entry in the stack frame is to link the stack frames of all procedures whose variables are currently in scope (or accessible). The state of the Pcode stack while procedure R is running would be:

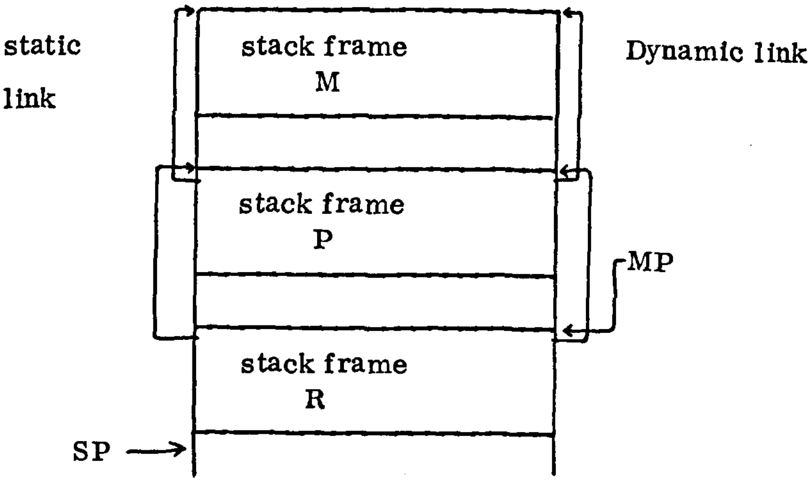


fig 4. 7

This link is the static link, ie. static in the sense that occurrences of stack frames are linked by virtue of the level as declared in the program text. The dynamic link is given by the saved values of the mark pointer in the stack frame. This link is used for procedural exit and points to the stack frame of the procedural occurrence which called the current procedure.

For example, suppose that within procedure R a call of procedure Q was made which in turn called procedure S. The stack frames would then appear as follows:

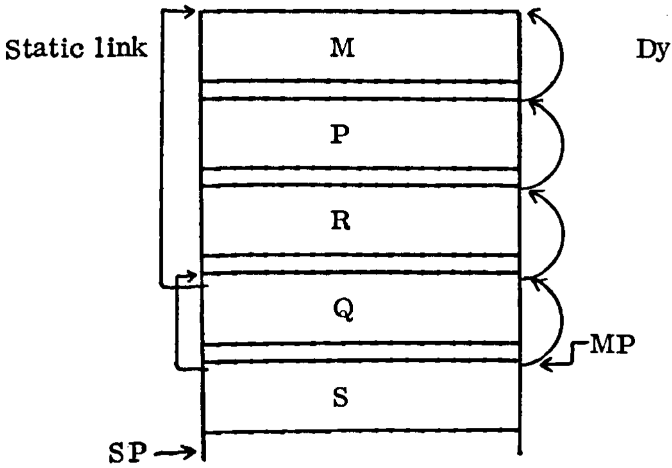


fig 4. 8

If procedure S then called itself recursively to a depth of three, the stack frames would then be:

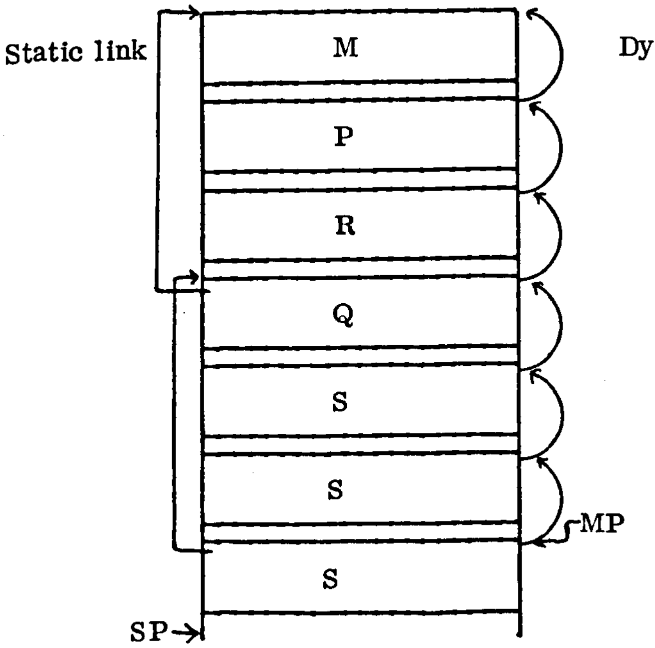


fig 4. 9

The base entry always points back to the surrounding procedure one level below. To access a variable, the address is specified by two components: the level of declaration and the address within the stack frame of its procedure. The Pcode machine then follows back the base entries as many times as the difference between the current level and the level of declaration of the required object. The resulting address is then the base of the stack frame containing the required variable. Added to the address of the variable within the stack frame locates the required object.

This method of access is frequently used by the Pcode machine and for deeply nested procedures accessing variables declared several levels below can be inefficient. It is for this reason that the base vector is stored in fast registers if such registers are available.

4.23 An existing Pcode machine

Most implementations of the P4 Pascal compiler consist of a Pcode interpreter. More ambitious implementations modify the compiler to produce object code for the target machine.This approach can be very time consuming as mentioned earlier.

One of the most successful and widely known implementations of P4 Pascal is that of UCSD Pascal {BOWL78} .The success is largely due to the fact that the same system has been implemented on machines ranging from mainframes to microprocessors. Perhaps the most interesting implementation of UCSD Pascal is that on the Pascal Microengine {MICR78} . The Pascal microengine has as its instruction set the Pcode instruction set used to implement UCSD Pascal. The microengine comes in its most basic form as five chips. These are a control processor, an arithmetic unit and three micro coded processors which implement the Pcode. In that respect, the microengine does actually interpret Pcode but the micro instructions required for this interpretation are executed at great speed. The speed of the microengine is claimed to be five times faster than the UCSD Pascal implementation on the PDP11.

It is encouraging that rather than altering the object code produced to fit a machine, a machine should be designed to fit the object code. If this trend continues then it shows that new ideas in programming languages can have an effect on subsequent ideas of machine architecture.

4.3 Transfer of Pascal system from the cross machine to the host

The process of implementing the Pcode compiler on a machine has been described above. Where the host machine does not already contain a Pascal system, a second machine, referred to as the cross machine, is required. The first implementation as described required the compilation of the P4 compiler on a CDC 7600 at UMRCC and the transfer of the resulting Pcode form of the compiler to the Digico Micro 16E. The compiler was created and compiled on this machine and then transferred to the GEC 4080. This section deals with these two transfers and highlights some of the obstacles encountered in what at first sight may appear a simple task.

At the time of the implementation of Pascal on the Digico minicomputer, Keele University computer centre had access to the North West Universities' network. This network had the following facilities: a remote job entry station at Keele to run batch jobs on either the ICL 1906A/CDC 7600 complex at Manchester or the ICL 1906S at Liverpool. As described previously, the cross compilation had to be performed at Manchester as this was the only accessible machine that supported Pascal. The only other connection that Keele had with this North West network was four terminals which could be connected independently to either Liverpool, Lancaster or Salford. Keele's own mainframe machine was not connected to the network in any fashion.

The Digico Micro 16E minicomputer has the following peripherals:

A single fixed/exchangeable 11 M Byte cartridge disc.

Two floppy disc drives.

A 'pullman' hand operated tape reader.

A printer.

A console VDU.

Two general communications interfaces.

There existed two options to transfer information from Manchester to the Digico. The first was to punch out the files required on paper tape and process them using the Digico's pullman reader. The second was to utilise one of the Digico's communications lines to link into the North West Network (somehow).

The first option would have resulted in approximately a mile of paper tape. This would then have to be pulled through the Digico's pullman tape reader by hand in sections and a suitable program written to transfer this information to the discs. The pullman reader cannot be used at the same time as the disc drives because being completely manually driven, it interrupts the processor in a manner that fatally interferes with any interrupts occurring from the normal sequence of events, including the disc drive. The tape reader was designed to load simple programs prior to any other peripheral activity. This method of transfer was discarded as being extremely slow and error prone.

The second option of utilising one of the Digico's communications lines was then investigated. The communications line as present on Keele's Digico is designed for connection via a modem {DIGIa} . The only convenient link into the North West Universities Network is a network terminal which can access the 1906S at Liverpool. Connection of the communications line to the network control computer at Keele was not allowed but this difficulty was circumvented by making use of the printer output socket on the visual display units connected to the network {NEWB} . The printer output socket can be utilised under software control. Upon the VDU's receipt of a certain control character (ASCII 15) the device connected to the printer socket is connected through to the computer attached to the VDU. This connection is broken on subsequent receipt of another control character (ASCII 16). By connecting the Digico's communications line to this printer socket, the Digico minicomputer can, in principle, communicate with any computer linked to the North West Network.

The problem of file transfer was then twofold. Firstly, the Digico communications line had to be connected to a simple printer socket in such a way that electrically, the printer socket appeared as a modem connection. Secondly, having achieved connection to a computer in the network a file had to be sent along the communications line to the Digico's disc drive.

The first problem was simply solved by a soldering iron and the second by a file transfer program written for the Digico. The connections necessary for this link are described in appendix 4. The transfer program, named LINK79, communicates with the North West Network via the control computer at Keele - an ICL 7905. The program's operation is as follows {WHIT79a} .

The communications line is continually scanned for information being sent. Any characters received are buffered for display on the Digico's console VDU. Whenever this buffer is empty, the Digico's VDU is scanned and any character typed sent down the communications line to the connected computer in the network. In this manner, the console VDU on the Digico appears as a normal VDU connected into the network. The algorithm as described gives priority to received characters; there are two reasons for this. Firstly priority is given to receipt in order to ensure no loss of data and secondly the Digico VDU is set up to be capable of receiving characters at four times the speed the network computer can send them.

The communications program LINK79 was then written to recognise a protocol for flagging the start and finish of a file transfer. The protocol used is that of a special character at the start of a new line. For starting a transfer the character is \ (backslash) and for terminating ? (question mark). On receipt of the start line the communications program copies all following lines of information received to a disc file, as well as displaying them on the VDU, until the finish line is received.

To transfer a file, the file is edited on the remote machine such that there is a line consisting of a backslash at the beginning and a line consisting of a question mark at the end. Via the communications program, the particular command to type this file out on the VDU is issued and the transfer is thereby initiated. This method of transferring files was subsequently used by the Computer Centre at Keele when the new machine was installed and, surprisingly, remains the most direct connection into the North West Network for this purpose. No corruption of data has occurred on transmitting in this way.

The program described can only connect to machines accessible to a local network terminal. These machines did not include Manchester's which held the required files. A facility did, however, exist whereby files can be transmitted from Manchester to Liverpool via Keele's remote job entry computer which could access both these sites. The eventual journey taken of the Pascal files is shown diagrammatically:

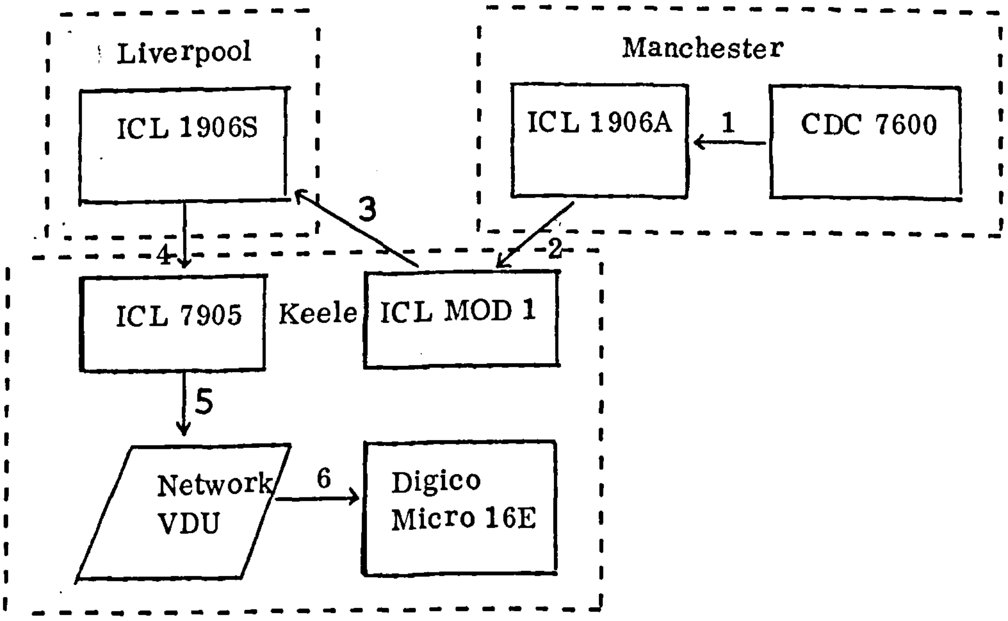


fig 4.10

For the implementation of Pascal on the GEC 4080, the cross compilation was performed on the Digico micro 16E. The file transfer from the Digico to the GEC 4080 was performed using much the same technique as described above. The communication line on the Digico was connected to the printer socket of a VDU attached to the 4080.

The communications program LINK79 was substantially modified for the GEC 4080. The modifications required were designed to permit transfers from the Digico and had to account for the different protocol the GEC machine uses. To transfer a nominated file from the Digico to the GEC 4080, the process is as follows. The modified communications program (named LINK48 {WHIT79b}) enables the Digico's console VDU to act exactly as though directly connected to the GEC 4080. On pressing the escape key, the program sends the file a line at a time. In this way, a command is given to the GEC 4080 to accept a file as though it were to be typed in. Two important considerations had to be made. The first was that in order to ensure that the GEC 4080 was 'keeping up' with the rate of information transfer, the program has to await a prompt sent by the 4080 before sending each line. The second consideration was that there is a delay of a few milliseconds between the 4080 sending the prompt and being ready to accept the first character of the line being sent. The delay is usually about ten milliseconds but can be as much as fifty milliseconds. It was considered that the solution consisting of waiting for the worst possible case was extremely inefficient when transferring almost fifty thousand lines of information. The solution adopted was to await the prompt and then repeatedly send a character until the 4080 echoed this character back. The rubout character (ASCII 127) is used as this character. Although echoed back, it is ignored on inputting a file on the 4080. As a check that the information transferred was correct, the communications program checks that every character sent is echoed back by the GEC 4080 correctly. The algorithm is then as follows:

```
while not end of file do
    begin
    await prompt;

    ,repeat
            send rubout character
    until  rubout character received;

    repeat
            send next character
            await receipt of character other than rubout;
            check character received = character sent;
    until  end of line character sent
    end
```

The loop which sends the characters and checks the echoed character ignores echoed rubout characters. This is because several rubouts may have been sent in the previous repeat loop which are still being echoed back. In defining, 'next character' in the line 'send next character' any rubout characters in the file are ignored.

In total the amount of information transferred was as follows:

Transfer route	Number of characters	Time for transfer
Liverpool → Digico	840,000	7.8 hours
Digico → GEC 4080	882,000	8.2 hours

The information transferred was two files in each case, these files being the Pascal source and Pcode form of the P4 compiler. The compiler transferred to the GEC 4080 was slightly larger than that originally stored on the Digico Micro 16E due to several modifications made to it during the time between the two transfers (about 8 months).

The two communications programs described LINK79 and LINK48 have been used several times subsequently and exist as the most expedient method of transfer of large files from the North West Network to Keele's main service machine, which at the time of writing is not due to be connected to the network for at least a year.

The programs were written in Digico machine code as this is the only language available on the Micro 16E that permits adequate control of the communication link. They each contain approximately 1500 instructions and the task of implementing then spread over a period of about six months.

4.4 Implementation of Pascal on the Digico micro 16E

The Digico 16E is a minicomputer with a small instruction set. The machine at Keele has no floating point hardware and no integer divide or multiply hardware instructions.

4.41 The Digico micro 16E at Keele

The facilities afforded by Keele's Digico machine are:

Micro 16E CPU
128 K Bytes RAM store
2 communications channels(GCIC)
1 console VDU
1 Diablo printer
1 disc drive
2 floppy disc drives
1 Pullman hand operated paper tape reader

The Pcode form of the compiler occupies approximately 64 K Bytes of store and requires about 20 K Bytes of data area to compile small programs that students may typically write (100/200 lines). The operating system for the Digico occupies just over 42 K Bytes. It was clear that little space is left available in the 128 K Byte memory if all the system was to be resident. The final layout is as follows:

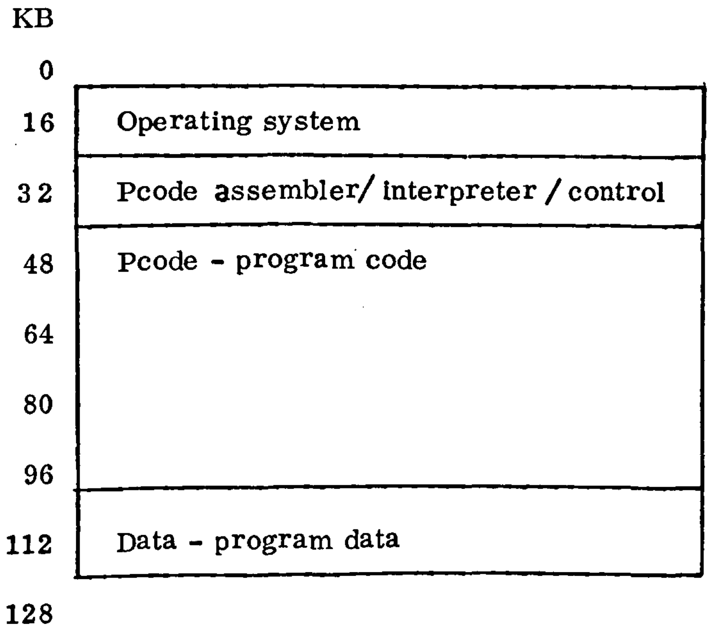


fig 4.11

The size of the store area reserved for Pcode was chosen to accommodate the compiler. The controlling program written in Digico's assembly language {DIGIc} occupies 7400 two byte words including space for local data and I/ O buffers. The controlling program has three functions:

1 User interface, with simple job control language
2 Assembly of Pcode instructions
3 Interpretation of Pcode instructions

User interface

The Digico operating system has a very basic job control language limited to loading and running machine code programs. This is not suitable for specifying compilations and executions and so a small command language was created for this purpose. The commands provided are:

C	program name	Compile Pascal program
G	optional file name	Assemble (Get) Pcode file.
H		Help message typed out
L	machine code program file	Load assembled Pcode program or machine code program.
R	file parameters	Run Pascal Pcode program or machine code program.
S	file name	Save assembled Pcode program
X		Exit from Pascal system.

In addition there are some extra commands, for example combining the function of compilation and assembly. The system does not require the user to distinguish between Pascal programs assembled and saved in a binary image and machine code programs which have been assembled by the Digico assembler. The main reason for including this facility is so that the Pascal system does not need to be left in order to input or edit files - the file editor which exists as a machine code program can be called upon just as though it were a saved Pascal program. The system has proved itself extremely easy to use by first-time users.

All allocations of disc files to Pascal programs including the compiler are performed by this user interface.

4.42 Assembler

The second function of the Digico Pascal system is that of assembling the Pcode produced by the compiler into the storage areas in a form suitable for the interpreter. The Pcode generated by the modified compiler exists as a readable text file. The main functions of the assembler in operating on this text file are:

1 Encoding and packing the relevant fields of a Pcode instruction.
2 Filling in forward jump references.
3 Allocating space and packing constants into the data area.

The constant table contains 4 kinds of entry. These are objects too large to fit in the Q field (see below) of an assembled Pcode instruction. The instruction is altered to load them into the stack indirectly when the constant is allocated space on the table. The constant table entries are:

Sets occupying 16 bytes
Reals occupying 6 bytes
Boundary pairs occupying 4 bytes
Literal strings occupying 2 bytes per character.

Boundary pairs are the values of the upper and lower limits of subranges used for compiler inserted range checking. All types are allocated at least two bytes of storage as Digico words are two bytes long and word addressing is much less restrictive than byte addressing for this implementation even for the string handling operations.

The form of a Pcode instruction as stored by this assembler is as follows. Two consecutive words are used:

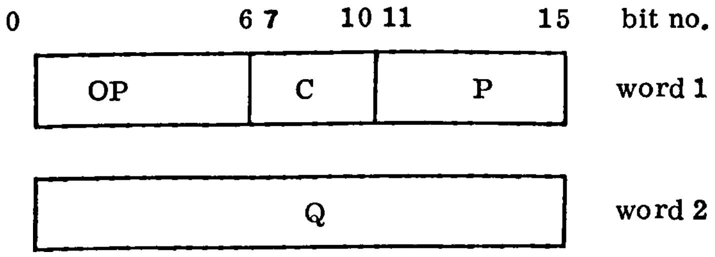


fig 4.12

The fields are as described previously in the description of the Pcode machine. The C field is used to distinguish the type and therefore size of the object the operation is performed on. The possible values being:

C	Type	Size in words
0	Integer	1
1	Char	1
2	Boolean	1
3	Pointer	1
4	Real	3
5	Set	8
6	Array	Variable - given by Q field
7	Imaginary null object	0
8	Nil pointer	1
9	Constant set	8

The value 7 of an imaginary object of zero size is used for procedure/function return where the size of the result of a function is required to set the stack pointer. For procedures the size of the result is zero.

4.43 Interpreter

4.431 General

The complete assembler/interpreter/control program for the Digico Pascal system is 8Kb in size. The interpreter section occupies approximately half this area of store. The interpreter caters for 63 Pcode Instructions; each is interpreted by its own subroutine. The subroutines vary in size from six instructions to about sixty. The average time taken for execution of one Digico M16E instruction is 2.5μs. The resultant execution speed then varies between 15μs and 150μs for each Pcode instruction.

Added to this is the speed of execution of the interpreter's main loop. The main loop decodes each instruction and branches to the subroutine corresponding to the Pcode operator found. This loop takes 49μs to complete. This speed is acceptable in that a compilation of a program proceeds at a rate of approximately six lines a second including delays for I/O transfers.

4.432 Addressing methods

The Digico Micro 16E has a poor set of addressing instructions. There are no index registers. The ordinary store accessing instructions between a given store location and the accumulator allow twelve bits for the address. This allows an addressing region of 4096 words. Similarly the simple jump instruction and all conditional jump instructions allow only twelve bits for the address. With these instructions, the high order four bits are taken to be the same as those of the address of the instruction being obeyed. The result of this is that the 64K words of store are effectively split into sixteen 4K 'stacks'. Without cumbersome portioning of a program and restricting addressing of data to the use of the three indirect data access instructions a program's code and data must wholly reside within one of these 4K stacks.

The interpreter requires much more storage space than this restriction permits and as a result, all access to the Pcode and data areas has to be made using the three indirect addressing instructions. These are:

GTI	n	load accumulator with word whose address is in n
STI	n	store accumulator in word whose address is in n
ADI	n	add to accumulator the word whose address is in n

The sequence for the Pcode instruction STO (store the top of stack in the address given beneath the top of stack) is interpreted as:

```
GTI   SP       load top of stack
DEC   SP       decrement stack pointer
STO   TEMP1    save value
GTI   SP       load top of stack (address)
DEC   SP       decrement stack pointer
STO   TEMP2    save address
GET   TEMP1    load value
STI   TEMP2    store at address
```

If an index register existed that could be used to hold the stack pointer the above code, which is typical of many of the interpretive routines, would be greatly improved.

The interpreter's main loop which decodes Pcode instructions requires three combinations of logical shifting of a word and masking selected portions using a logical 'and' instruction to recover the four fields. The implementation on the GEC 4080 is capable of recovering the required fields without any masking or shifting. The addressing allowed gives access to any byte or halfword within a 4080 word and makes the process of recovering the fields of an instruction trivial and fast.

4.433 Arithmetic operations

The most notable restriction imposed on the evaluation of arithmetic expressions is that the only hardware arithmetic instructions on the Digico are integer addition and subtraction. Division and multiplication are handled by two software routines. This makes Pascal array accesses particularly slow as the product of two values is required in calculating the address, and the size of each element of an array is not necessarily a multiple of twowhich would allow simple multiplication by shifts.

No floating point hardware is available on the Digico at Keele. A set of routines to interpret three word floating point values was used. This package also includes the routines for the standard procedures and functions in Pascal which operate on real values. This floating point package is approximately 2,300 words long. The package plus the interpreter are too large to coreside in one 4K stack. Instead, the floating point package resides in the first 4K stack allocated to Pcode code when a user's program is being executed. When the compiler is being interpreted, the floating point package is written over by the first part of the compiler Pcode. The compiler does not use any floating point instructions. The package used is an adaption and enhancement of a floating point package written at Keele {SING78} This package required additional routines to optimise transput of floating point values and the code to implement the standard routines sin, cos, sqrt, atn, log and exp was added.

The most difficult Pcode instructions to implement were the operations on sets. To restrict the set size to one word would result in sets containing at most 16 elements. Eight words are used to hold a set value. Implementation of operations such as set inclusion when the only logical operations available are 'and' and 'not' is extremely tedious.

4.434 Conclusions

In the above section, the difficulty of implementing a language such as Pascal on a small machine with a very limited instruction set has been pointed out. The interpreter for the Digico Micro 16E was developed over a period of twelve months although a crude system was working without implementing sets or reals after two months. The fact that Pascal can be implemented on such a basic machine in such a relatively short time by one person is itself remarkable. The Pascal system, although useable by one person only at a time, has proved itself reliable and easy to use since its completion in 1978.

4.5 Implementation on the GEC 4080

4.51 General

The GEC 4080 machine at Keele has the following technical description:

1 4082 processor

512 K bytes core store

4 M bytes drum storage

2 disc drives - 70 Mb each

3 magnetic tape drives

1 paper tape station

Graphical station - graphics tube, plotter and digitiser

The instruction execution time varies from 1.3 micro seconds for a half word (2 bytes) integer addition to 60.8 microseconds for a double word floating point division.

The GEC 4080 contains eight registers. These are as follows:

A	The main accumulator - 32 bits
B	The main floating point accumulator - 32 bits
X	Index register - 16 bits
S	Sequence register - 16 bits
L	Local workspace pointer - 16 bits
Y,Z	Base registers - 16 bits
C	Conditions register - 8 bits

In addition, as all instructions address via a register, an imaginary zero register exists containing zero. A 4080 program (or 'process') typically contains several code chapters. Each chapter can be regarded as a self contained piece of code. The main need for several code chapters is that direct access of local data is restricted to 128 objects for each chapter. Each chapter will contain a local workspace area of which only the first 128 objects may be directly addressed. The L register always points to the base of the local workspace for the current chapter. In addition, a global workspace area accessible to all code chapters exists starting at storage location zero. This may contain 256 objects of directly addressable data.

The usual method of accessing the rest of the data storage is either to consider it as a set of vectors indexed by the X register or to access it relative to bases which are loaded into the Y or Z register.

In referring to the local workspace, the word 'object' was used to mean any of the standard types available to 4080 instructions. These are:

bytes	8 bits	aligned at 1 byte intervals
halfwords	16 bits	aligned at 2 byte intervals
fullwords	32 bits	aligned at 4 byte intervals
reals	32 bits	aligned at 4 byte intervals
doubles	64 bits	aligned at 8 byte intervals

Doubles are double precision reals (floating point numbers).

The displacement/address quoted for accessing an object is in units of that object. For example:

Load byte 10	will load byte number 10 into the accumulator
Load halfword 10	will load bytes 20 and 21 into the accumulator
Load fullword 10	will load bytes 40, 41, 42 and 43 into the accumulator

The base address used (usually the contents of the L register) is always a byte address. A program on the 4080 at Keele may have, in theory, access to four megabytes of store. At any one time, only sixty four kilobytes of store is accessible.

The 4080 contains a hard wired nucleus as the core of its operating system. This nucleus performs all address checking and user initiated store requests. The store accessible at any time is defined by the nucleus and appears as follows:

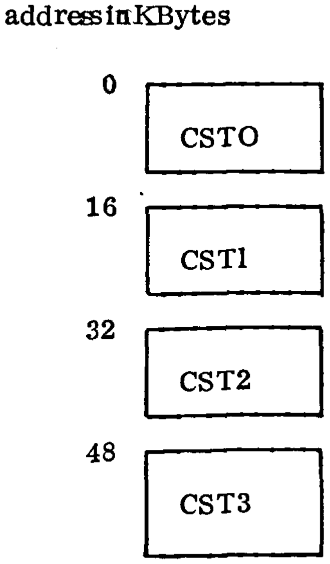


fig 4.13

The program can access four segments at any one time. A segment is a contiguous piece of store of maximum size 16 kilobytes. A program can be composed of up to approximately fifty segments, any of which can be nominated to be one of the four currently accessible. This storage manipulation is governed by two tables, the PAST and the CST tables. These are altered and set up by the nucleus and accessed automatically by the hardware. The PAST (Program Accessible Segment Table) has an entry for each segment owned by the program. The CST (Current Segment Table) contains the PAST numbers of the four segments currently accessible. Segments are also the units of transfer to the drum when overlaying.

Two of the four CST segments have special uses. CSTO always contains the global and local workspaces for the program and so must always be present. CST3 contains, usually, the code for the program. If the code is greater than 16KB or has deliberately been placed in more than one segment then the operating system will automatically overlay into CST3.

CST1 and CST2 are overlayed to contain any owned segment under program control. It should be noted that overlaying does not involve the copying of a segment but just nominating a segment to be the one currently known as CST1 or CST2. The nucleus then alters the relevant entry in the CST table.

4.52 The Pcode Interpreter

4.521 General - store access

The Pcode interpreter uses the four current segments in the following manner:

CST0	Workspace for the interpreter
CST1	Pcode area of program
CST2	Store area of program
CST3	Code of the interpreter

Should a Pascal program,when compiled, require more than 16 Kbytes of store to contain the Pcode or more than 16Kbytes of a data store, or both, then extra segments are allocated and overlayed into CST1 and CST2 respectively.

When such overlaying is necessary, an explicit check is required for each Pcode machine store address in order to ascertain whether the segment containing the required item is the one currently accessible. Unfortunately no use can be made of hardware interrupts to detect access to a non resident segment. The reasons for this are that accessing an address outside the segment in CST1 would be taken as an access to CST0 or CST2. Additionally, should a store access be regarded as illegal by the hardware, then the GEC 4080 operating system does not allow continuation of the program under any circumstances.

Checking every store access is a large overhead and has been found to almost double the execution time of a program. For this reason, the Pcode interpreter can operate in two store access modes. In the first, the maximum store allocation is 16 Kbytes; this requires no checking of store accesses for the required segment as it is permanently resident as CST2. The second allows 64 Kbytes of store and overlays four segments into CST2.

The P4 compiler allocates all addresses as byte addresses which are interpreted as follows:

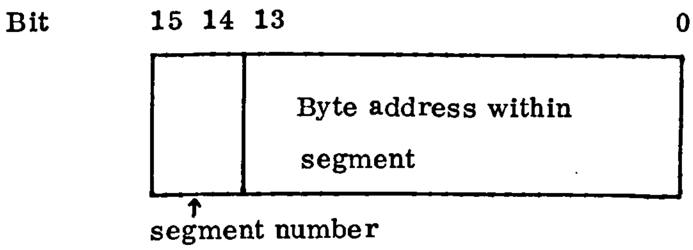


fig 4.14

The required segment is found by adding the segment number to the PAST table number of the first store segment. This is then overlayed into CST2 (if not already there) and the least significant 14 bits are used to index CST2.

Pcode instructions always occupy four bytes (one full word) each. CST1 is then accessed as a vector of 4096 Pcode instructions. Should a Pascal program be sufficiently large to require more than 4096 Pcode instructions then up to four segments are overlayed onto CST1. The way this is accomplished is as follows:

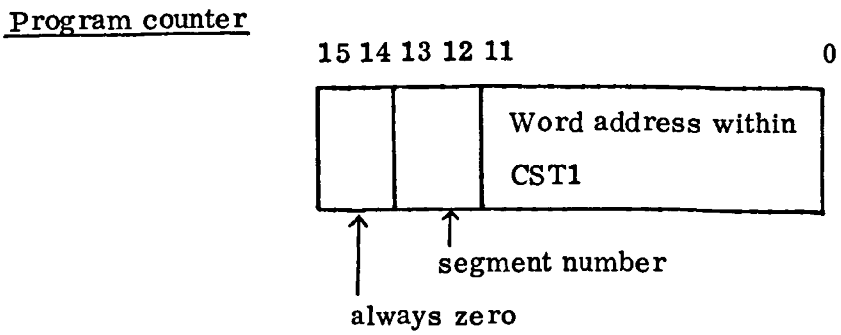


fig 4.15

Whereas accesses to the program's data store are in general random and checks have to be made on every store access, if the program requires more than one store segment, access to the code follows a much more well behaved pattern. Unless the current Pcode instruction is a branch instruction, the next access is always for the next word in CST1. A check is made on the program counter, to establish whether the required segment is resident in CST1, whenever a jump instruction has been operative. There is one further possibility of the program counter leaving the current segment, when the current instruction is the last one in a segment and the program is about to 'march over' a segment boundary. Rather than check whether the current instruction is the last of a segment, which would be as costly as checking the required segment number of the program counter at each instruction, a dummy instruction is inserted as the last one in each code segment. This instruction is simply an unconditional branch to the following Pcode location. As this is a branch instruction the program counter will then automatically be checked against the current Pcode segment by the interpreter.

When a program does not require more than one Pcode segment, these checks on the program counter are inhibited.

4.522 Pcode instruction format

A Pcode instruction is coded on the GEC 4080 implementation as shown:

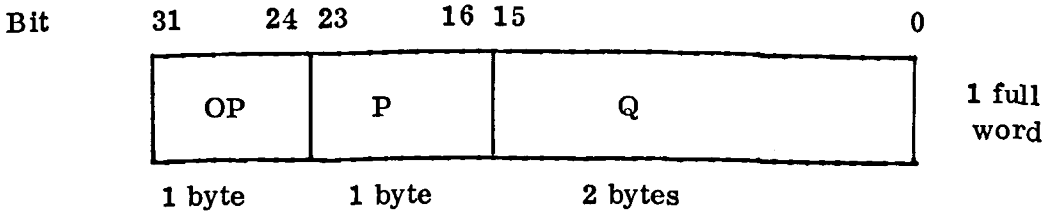


fig 4.16

Unlike the Digico implementation, there is no C field determining the size of the object the instruction is to operate upon. Instead, the C field, when present, is merged into the OP field. Of the 61 original Pcode instructions, 18 require a C field. The number of different sizes of object operated upon by the interpreter is three - 1 byte, 4 bytes and 16 bytes. The number of extra instructions ncessary to eliminate the C field is then 36. The alteration of the OP field is performed, after compilation, by the Pcode assembler described below. In addition, certain operations, such as comparison of two real values have to be handled by a different instruction than comparison of two integral values - even though both these types occupy four bytes.

4.523 Operation of the 4080 Pcode interpreter

As mentioned above, a GEC 4080 program (called a process by GEC) is usually composed of 'chapters'. A chapter is either a data chapter or a program (code) chapter. Several code chapters are required for large programs.

The Pcode interpreter consists of five chapters (four code and one data) and fourteen system chapters (thirteen code and one data). The system chapters handle the basic input/output and the standard mathematical routines - sin, cos, atan, log and exp. The code chapter being obeyed is always resident in the current segment CST3 referred to above, and all data chapters are resident in CST0.

The five Pcode interpreter chapters are:

1 INTPRTGD. The Global data chapter.
This contains all directly accessible data required by the four code chapters.

2 INTERPRETER. The main code chapter.
This chapter contains the code to read in the program, interpret Pcode instructions and handle all storage allocation and error detection.

3 CSPCHAP. Standard routines chapter.

This chapter contains the code to implement the Pcode standard routines. In most cases, either a systems routine is called when the standard routine is one of the standard mathematical routines, or the relevant routine in the chapter PASCALIO when the standard routine is dealing with input or output. An example of the sort of task handled by this chapter is transput of the textual form of printed integers or reals. More primitive routines for reading and writing characters in PASCALIO would be called in that process.

4 PASCAL IO. Pascal input and output routines.

This chapter handles all the basic transput interfaces with the standard system routines. All basic I/O initiated by the Pascal program such as reading and writing one file component is handled by this chapter.

5 MERR. Mathematical error.

This chapter is required by the system mathematical routines sqrt, cos, sin, atan, log and exp. It is called in the event of an error such as asking for the logarithm of a negative number. In this event, this chapter calls the error handling routine in the main chapter INTERPRETER.

The interpreter consists of just over three thousand lines of Babbage assembly code which assembles into four and a half thousand GEC 4080 machine instructions.

4.53 The Pcode assembler for the GEC 4080 Pascal implementation

The three functions of control, assembly and interpretation required to implement this Pcode Pascal system are dealt with separately, unlike the Digico implementation which handles all three functions in one program. The facilities exist in the 4080 operating system to call programs under the control of a useable job Control Language. These facilities to call programs and associate them with particular I/O devices do not exist on the Digico. The assembly stage can be considered a second pass of the compilation in that it is required once after each compilation before a program can be executed.

The functions of the assembler for the 4080 implementation are:

1 Encoding the textual form of Pcode into four byte instructions.

2 Allocating space in the constant table and filling in such references to the table.

3 Filling in of all mnemonic addressing such as control transfer.

4 Generation of the extra instructions, where necessary, to eliminate the C field.

5 Generation of the extra jump instruction at the end of each segment as described above.

6 Generating information for the interpreter as to the size of store required for execution.

The assembler produces a file containing the assembled Pcode. The file is a 'core image' so that the interpreter need only read the file directly into the Pcode code and the data areas prior to execution.

The assembler, written in Babbage assembly language {GECb} , is just over 600 lines long and occupies 14K bytes of store. As an indication of the speed of the assembler, it requires IO CPU seconds to assemble the textual Pcode form of the compiler. The assembler produces 16733 Pcode instructions and 2176 bytes of constants. This is a speed of 1673 Pcode instructions a second.

4.6 Enhancements/Improvements made to GEC 4080 Implementation

4.61 General

Several improvements were made to both the P4 compiler and the Pcode interpreter. These changes were made for two reasons. Firstly, P4 Pascal differs, in several respects, from standard Pascal as described by the Pascal report { WIRT75 } and the British draft standard {ADDY78} . Secondly some alterations were necessary in order that the compiler was more suited to an interactive system used for teaching and research. Where new features are added, the compiler is able to warn the user, if requested, whenever a non standard feature is used. Alterations made to enable the run-time diagnostics to be added are described in a later chapter.

4.62 Removal of deficiencies

It was decided that, as far as possible, the Pascal compiler should adhere to the specification of the draft standard mentioned above. The main exception to this approach concerns procedural and functional parameters as described on page 20 of the report. These parameters are one of the main points of contention hindering an agreement of an international standard and so have not been included pending any outcome. The following differences were removed. Appendix I contains a list of the differences between P4 Pascal and the draft standard.

a) Files

P4 Pascal does not cater for the declaration of any files, and will only recognise four predefined 'text' files 'input', 'output', 'prr' and 'prd'. The compiler and interpreter were changed so that only input and output are predefined as type 'text'. and that declarations of files of any type is permissible. On entering the main program an automatic call of rest (input) and/or rewrite (output) is made if either file is declared in the program heading. The standard type 'text' is included in the compiler.

b) Program heading

P4 Pascal does not require the presence of a program heading. This was changed according to the report. Any file variable that appears in the program heading must be declared as a file type globally in the program. Files in the program heading are capable of being connected to actual devices by the job control statement initiating the execution of the resultant program. The program name in the heading is retained and printed at the start of execution for identification purposes. Unless otherwise connected to specific devices as described later, files declared that do not appear in the programme heading are connected to temporary disc files. The file names in the program heading are used by the compiler to produce a 'proforma' which is used by the GEC 4080 operating system to identify the file names used with physical devices.

c) Standard procedures

The standard procedure 'page' and function 'round' was added to the P4 compiler. The second optional field width parameter for writing real values was allowed.

d) Sets

P4 Pascal does not allow the form:

< expr> . .< expr >

as a set constructor. The only valid set constructors are expressions separated by commas. The compiler was altered so that the above subrange notation is allowed. The Pcode instruction SGS which generates a singleton set was then altered to require two parameters on the stack - the upper andlower bound of a subrange. Its original function is then the special case of the upper and lower bound being the same. Where both the expressions in the subrange set constructor are constants, the compiler will generate the required set itself.

e) General

The predeclared constant 'maxint' is included as the maximum integer capable of being represented by the GEC 4080 (214748 3647). P4 Pascal imposes a maximum length of literal strings of sixteen characters. This has been changed to eighty characters. As a literal string may not extend over more than one program line, eighty characters seemed a more reasonable upper limit. The draft report allows upper and lower case alphabetic characters to be used on identifiers. The draft also specifies that for the purposes of denoting two identifiers as the same, upper and lower case counterparts are considered equal. This was implemented on the compiler along with the use of curly brackets { } to delimit contents.

4.63 Enhancements

Several enhancements were considered desirable in order that the Pascal system is suited to a teaching environment. P4 Pascal produces a listing of the program which gives line numbers and the Pascal address/storage address allocated to the first statement/declared variable on that line. When an error occurs at run-time, the interpreter can then inform the user of the Pcode location where the error can be found. This location is then matched with the Pcode locations on the listing to find the source statement in question. The compiler was altered so that (unless specifically requested) no Pcode locations are listed, just line numbers. The compiler then inserts a new Pcode instruction at the beginning of each statement which starts on a new line identifying the line number. The interpreter then quotes the line number of the erroneous line.

The P4 compiler indicates compilation errors when detected and prints a number corresponding to the error as given in the table in the Pascal manual {WIRT75} The two objections to this are that the error number has to be looked up in a list manually in order to find out what the error is, and that in a large program it may be difficult to look through each page of the listing to spot any errors. This was altered so that no error number is printed; instead the compiler prints the full error message. The table of error messages is held on a disc file which the compiler accesses randomly. At the end of the listing, the compiler will print a summary. This summary consists of the line number and error message for each error detected. This summary is optional as it is only really useful for large programs. When a small program is being compiled, and the listing is being printed on a VDU screen, a summary would not usually be wanted, as it would cause program text to be scrolled off the top of the screen.

For efficiency reasons, the compiler was made to stop producing code when it found a compilation error. The user will receive as the 'compiled' program a program, which, if execution is attempted, prints a suitable message and stops.

4.64 New features

Some non-standard features were added to Pascal either to make use of facilities available on the GEC 4080 or because they were considered a necessary facility in a language to be used for teaching certain Computer Science courses at Keele. In each case, the compiler is able to warn the user that non-standard features have been used in the program. These features are:

a) Random access files

The reserved word 'random' was introduced. This word may precede the word file in a file declaration. In this case, the file is opened as a random access file. The operations available on a randomfile in addition to reset/rewrite are:

getrandom (<file name> , <integer expression>)
putrandom (<file name> , <integer expression>)

The integer expression denotes a key which must be larger than zero. This key is found before the get/put is obeyed. The standard function eof will be true following an attempted getrandom for a non-existing key.

update (<file name>)

This procedure is similar to reset/rewrite but allows both getrandom and putrandom to be applied to the file. 'rewrite' applied to a 'random' file will create a random access file ready for initialising.

The main reason for including this facility was so that the compiler can access a random access file of the compile-time error messages. The file used by the compiler is declared as:

```
var   errfil : random file of array [1..80] of char;
```

b) Dynamic connection of files to physical devices

The procedures reset, rewrite and update (as described above) were redefined to cater for an optional second parameter. This parameter, if present, must be a literal string or an array of characters. In performing the reset/rewrite/update, the file variable will be connected to the physical device specified in the string. The device is specified in the same manner as expected in the job control language of the GEC 4080 operating system.

c) Pseudo random numbers

The predeclared procedure 'randomise' and predeclared function 'rnd' was added.

Rnd is a real function that returns a pseudo random number on the range:

$$o \leqslant x < 1$$

The same series of numbers produced will always occur unless the procedure randomise is called. This procedure sets the seed of the pseudo random sequence to a value dependant on the reverse order of the bit sequence constituting the time of day in milliseconds.

d) Halt

The predeclared procedure 'halt' causes termination of the program. A remaining deficiency of this Pascal system is that a goto statement may not lead out of a procedure. A common use of such an abnormal exit from a procedure is to jump to the end of the program. The procedure halt overcomes the difficulty and is arguably a neater method of providing the facility.

e) Environmental enquiries

The predeclared integer function 'environ' was introduced. This function requires one integer parameter and the value returned depends on the value of the parameter as follows:

Parameter value	Value returned
* 0	0
* 1	1
2	Time of day in seconds
+ 3	Time of day as the integer HHMMSS
+ 4	Date as the integer DDMMYY
5	Amount of storage space left
6	Maximum amount of storage space used
7	Number of milliseconds since the program started
* 8	8
* 9	0
* 10	0

*These calls of environ are only meaningful to the compiler or the run-time diagnostic system as described later. They perform special communication with the interpreter or operating system. For example, environ (0) is called by the compiler to inform the operating system that compilation errors were detected.

+For example for ten minutes and twenty seconds past ten o'clock am. on the twenty eighth of September nineteen seventy nine, environ (3) and environ (4) would return the decimal integers 101020 and 280979 respectively.

4.65 Compiler options

Compile time options are specified, as in many Pascal implementations, within a comment. If the first character of a comment is a hash sign # , then following should be a series of a letter followed by a plus or minus sign separated by commas. For example:

```
(* # A+, D-, H-*)
```

Would cause option A to be set and options D and H to be unset. Any deviation from the defined sequence results in the rest of the comment being ignored as usual.

The options already provided by the P4 compiler are:

Option	Action	Default value
T	Print compiler tables for each procedure	off
L	Produce compilation listing	on
D	Produce run-time range checks for subranges/array bounds	on
C	Produce Pcode	on

Additional options were incorporated as follows:

Option	Action	Default value
A	Produce listing of Pcode addresses as original P4 compiler	off
H	Produce heading incorporating program name when program is executed	on
S	Produce error summary at end of listing	off
W	Produce warnings whenever non standard features are used and at the end of the listing if any such features were used.	off
E	Terminate compilation on first error	off

4.66 Removal of errors

Several errors exist in the P4 compiler. A list of known errors is published in Pascal News {MICK} and these have been corrected. These errors are mainly concerning the new features that were added to the P3 Pascal compiler to produce the P4 compiler. The new features concern alignment of data types occupying different storage sizes.

4.67 Deficiencies remaining

Not all the differences between the P4 Pascal implemented and the draft standard have been removed. Those remaining are:

1 Objects on the program heading parameter list may only be files.

2 procedure and function formal parameters are not allowed.

3 nil is a predeclared constant, not a reserved word.

4 goto s may not lead out of a procedure or function block.

5 The standard procedure 'dispose' is replaced by 'mark' and 'release'.

6 Read/write may only be applied to actual text files .

4.7 Experience of use of the Pascal system on the GEC 4080

This Pascal system has been used for teaching purposes for the Computer Science Department's principal and subsidiary courses at Keele University for the teaching session 1979/80. The system has performed reliably and no errors within it have made themselves apparent. The interpreter, assembler and the Pcode form of the compiler are all reentrant and one copy is thus shared between all simultaneous users.

The system can degrade considerably under extremely heavy use when sixteen or more people are attempting a Pascal compilation. This problem is thought to be avoidable when the compiler is truly compiled rather than interpreted, at some later date, or if the 4080 is upgraded to its maximum store size.

CHAPTER FIVE

PORTABILITY

5.0 Introduction

The experience of implementing Pascal on two very different machines raises many topics associated in a more global sense with the field of software portability. In general, two machines which contain an implementation of a particular programming language will use different methods for obeying certain language constructs. Quite often, an implementation will not cater for all language features defined and will allow some additional language features which are not defined as any form of standard. The result of this is that the transfer of a piece of software from one machine to another will involve two stages. Firstly the program itself has to be moved from the storage medium of the donor computer to the storage medium of the host. Secondly the program will be modified in order to be acceptable to the host and run in this new environment. The combined ease of these two tasks comprises the portability of a piece of software.

5.1 Objectives

This chapter considers portability from three viewpoints. Firstly, the more common problems that may arise when transferring any piece of software are discussed. Secondly the problems encountered in implementing the P4 portable compiler are discussed. Reference is made to particular problems which arose and a comparison of the two implementations made. Finally the strategy employed to attempt to minimise portability problems is explored. Such a strategy, inevitably,will be comprised of optimum routes to be taken, and suggestions for changes and improvements to both software and hardware to ease the problems involved, are made.

5.2 General Portability Problems

There exist many potential obstacles to the transfer of a piece of software from one machine to another. Perhaps the most common are those of language differences, character code differences, the differences in machine architecture and the media available for the physical transfer. Other considerations include the interface with the host machines operating system and the environment within which the software will be used. Aspects of some of these problems are now discussed; examples of their effects on the implementations of Pascal described are dealt with later in this chapter.

5.21 Language differences

Given any high level language, it is likely that implementations on different machines will not treat particular programs in the same manner. Wichman WICH76 and Knuth KNUT67 list incompatibilities of different implementations of Algol 60. Several different representations of the Algol 60 symbols exist, for example:

'BEGIN', "BEGIN", BEGIN

Less obvious differences include Wichmann's example of side effects in the expression:

s + f (s)

where s is an integer and f an integer function, can result in different values depending on whether s is evaluated before f (which has the side effect of altering the value of s). The KDF9 Algol system, for instance, will evaluate the expression strictly from left to right while the ICL 1900 system evaluates functions first. Most implementations of languages will allow non-standard features of that language such as 'long reals' in the DEC system 10 DECa .

Pascal was designed with great emphasis on its subsequent portability. A detailed definition of the representation of symbols is given in the Pascal report. Although Pascal does not define the order of evaluation of an expression (such as the example above) the report explicitly states that the programmer should not assume which term is evaluated first but cater for both possibilities. Pascal also gives some consideration to another common problem existing between different implementations, that of the number of significant characters of an identifier. Pascal allows identifiers of any length but the report stipulates that only the first eight will be considered significant. In Algol 60, for example, no limit is defined on the number of significant characters of an identifier but most implementations will impose one. The unfortunate outcome is that some Algol 60 compilers will consider the first eight characters (for instance) while others consider only the first six characters. In this case, a program containing the declaration:

```
integer  number 1, number 2;
```

will not compile if only the first six characters are significant. A more dangerous structure is of the form:

```
begin  integer  number 1
          .
          .
          .
     begin  integer number 2;
     number 2 : = 2 * number 1
          .
          .
     end
          .
          .
          .
end
```

This program would compile on all such Algol 60 compilers but give different results. There is in fact an instance in the Pascal P4 compiler where the same identifier is spelt quite differently (after the eighth character) in two different parts of the compiler.

5.22 Character code differences

There exist several different sets of character codes. The character coding varies from one manufacturer of a computer to another. The DEC system 10 {DECb} has two character sets on the same machine; these are ASCII and SIXBIT. ASCII (American Standard Code for Information Interchange) is a seven bit coding including 128 characters, while SIXBIT is a six bit coding comprising the ASCII character codes 32 to 96. The CDC range of machines has three character sets each containing 63 or 64 characters. These are the 'ASCII character set with CDC's ordering'. The scientific character exists in two forms, the 64 character form and the 63 character form which is the same as the 64 character set except that the code for ':' is different and the character '%' does not exist.

In order to transfer a program from one machine to another which has a different character set, a processer will be necessary to convert from one character set to the other. This operation is a simple table lookup if the two character sets contain the same characters or more correctly if the donor's set is contained in the host's set. If this is not the case, then alternative representations for characters that are not representable must be used. Unfortunately, machines with a limited character set will often contain compilers which, because of 'missing characters', implement a non-standard version of the language concerned, hence making transfers of programs doubly difficult.

Another common difference between many machines is the standard representation of the end of a line. Some will use the character pair 'carriage return and line feed' others one 'newline' character which is typically 'carriage return' or 'escape'. The GEC 4080 does not represent the end of a line by a character but prefixes each line with its length. These differences lead to Pascal not defining an end of line character but including a Boolean test for the end of a line.

5.23 Machine architecture differences

Arguably the greatest obstacle to the portability of a program is the wide variation in machine architecture. If the program is a 'Portable Compiler' then a new code generator or interpreter will be required. The host machine may have insufficient storage for an efficient implementation or may lack certain machine orders which result in a very slow operation of certain high level functions such as Pascal set manipulations or recursive function calls. In this case, if transfer of a program necessitates transfer of an efficiently executable program certain optimisations may be necessary to overcome local peculiarities. Some of the most common machine architecture differences are now discussed.

5.231 Word size/accuracy

The term word size is used here to mean the size in bits of the basic storage unit of a machine - this storage unit being treated as an indivisible data item by some of the machine's instructions. Some machines are capable of handling more than one size of storage unit. The GEC 4080, for example, can process storage units of four different sizes {GECa} . In this case the word size is that unit of storage which is normally used to hold integral values and upon which the arithmetic operations of the machine can apply. This word size is as small as eight bits for many microprocessors and as large as sixty bits, for instance, on the CDC 7600. The word size determines the range of values that variables may take. With a word size of eight bits, the integers -256 to +255 may be represented whereas with sixty bits, a range of approximately $\pm 10^{18}$ is allowed. This large difference can have great implications if a piece of software relies on being able to represent a particular range of integral values. If a machine's word size is too small for this purpose then two or more successive words will have to be considered as one unit and all operations on this larger 'word' will have to be implemented by software. This will result in the program executing at a speed which is reduced greatly and thus may be unacceptable if the program requires a certain speed such as in a 'real time' application. The word size of a machine will also dictate the range of floating point representations. Because floating point representations are a finite set of values they are only an approximation to the infinite number of values within any range. The number of different values representable within a particular range is also related to the machine's word size. The word size then not only limits the range of floating point values but also their precision. Ford {FORD76} discusses the portability of NAG library routines and quotes a particular instance of a mathematical technique of evaluating the incomplete gamma function. This method has a precision of 10^{-8} which is absolute. The algorithm cannot be used to aquire a greater precision which may be required on a machine with a large representation of floating point values and as such is not portable. Lack of a particular precision can also result in some mathematical techniques being inefficient or not applicable.

5.232 Store size

The amount of memory a program may be allocated is a function of the physical amount of memory the machine possesses, the size of any operating system that is present and the number of simultaneous users of the machine. This available memory space is again very different on different machines. A program may require more memory than is available in which case some form of virtual memory implemented on backing storage would be necessary. Depending upon the nature of access to this virtual memory a system of either paging or overlaying would need to be used. Assuming sufficient backing storage is available, any size program can be implemented on a small machine but may have the effect of a large reduction in operating speed - access to backing store being typically several orders of magnitude slower than access to main memory.

5.233 Registers

The number of registers on different machines varies. Every machine possesses a sequence register which contains the address of the machine instruction to be obeyed and most contain at least one register called the accumulator. In addition, some machines possess additional registers commonly used to index the store. The Dec 10, for instance, contains fourteen index registers which may alternatively be used for accumulating arithmetic results or as stack pointers. The Digico Micro 16E, and many other machines, have no index register and an indexed address then must be calculated, stored and an indirect instruction obeyed. Lack of several index registers does not usually preclude the implementation of a program but inevitably necessitates a more long winded approach. This feature of machine architecture is, again, one affecting the subsequent efficiency of a program rather than one that seriously hampers the transfer of software.

5.234 Machine instructions

A machine will have amongst its repertoire of machine instructions functions for testing and altering storage locations, performing some arithmetic manipulations and performing input and output. In addition, each machine will usually be capable of many more complex functions. The number of machine code instructions available is typically of the order of a hundred. The variation can be large, for example the Digico Micro 16E has approximately sixty machine orders whilst the Dec 10 has a possible repertoire of 512 machine orders. The Dec 10 has a large repertoire, but many instructions exist for completeness rather than being useful. The main areas of common concern when considering useful machine instructions are the availability of floating point instructions, stack manipulation and the ease of manipulating small data items such as characters and bits. Lack of such facilities will inevitably result in small routines to interpret absent instructions with the corresponding decrease in speed. Confining the machine code to a small number of basic instructions, in addition to resulting in a decrease in speed will also cause an increase in the program size. This fact, combined with the case that machines with very basic machine instruction repertoires tend to also have a small store size will entail a large effort in implementing a program on a small machine.

5.24 Media available for the transfer of software

Currently available media for data transfer from one computer to another are:

Direct link
Portable discs
Magnetic tape
Punched cards
Paper tape

The problems realised by different character code conventions between the two machines have been dealt with earlier. Unless the two machines are of the same manufacture or have been linked together, the last two options will often be the only method of transfer. Whilst in principle paper media are fairly straightforward, they can become impractical for large data transfer. The P4 compiler and its Pcode form for example would occupy almost one and a half miles of paper tape or a stack of cards nearly fourteen feet high. This approach can be cumbersome and wasteful particularly as several transfers may be needed if modifications are necessary which can only be made on the donor machine.

The first method, the direct link, is the simplest and often fastest method of transfer. Very few computers of a different origin are connected together. The difficulty in inter-machine communication is often that of formulating a common protocol. For transfer of files, the protocol needs to be able to communicate when a file is to be transferred and what is to be done with it. A common approach to this is for a machine to emulate a known device such as a card reader or paper tape reader.

The second two methods, magnetic tape and disc, involve conventions of blocks or groups of information. These media are always formatted in units referred to as blocks or sectors. The size of these blocks is rarely the same on two machines of a different make or series. There has been some work on producing industry standards for magnetic tape but these standards are far from generally recognised.

The problems involved in physically moving data from one machine to another can be seen as non-trivial. Griswold {GRIS76} points out that this process can often take longer and involve more work than installing the program that has been transferred. There is clearly some scope for development in this area; a commonly agreed and extremely simple protocol to send a specified number of characters along a direct computer link would be extremely useful and in theory be fairly easy to implement. There exist several networks of computers but each has its own complex protocol. It would be reassuring if a simple transfer program were seen to be as common and expected as a systems program such as a Fortran compiler.

5.3 Some other portable compilers

5.31 UCSD Pascal

The UCSD (University of California at San Diego) Pascal project {BOWL78} had as its aim to 'provide minimal-cost computing facilities for introductory computer science classes'. The project is discussedunder a heading of portability because it has been implemented on many microprocessors, in each case providing identical systems to the user.

5.311 UCSD Pascal compiler/language

UCSD Pascal is based on the Pcode compiler P2. This has been modified to eliminate some deficiencies and enhanced in many respects. In all implementations but one Pcode is produced and then interpreted. The Pascal microengine {MICR78} has Pcode as its instruction set and thus has a compiled UCSD Pascal system. The differences remaining between UCSD Pascal and standard Pascal are listed in appendix 3.

5.312 UCSD Pascal environment.

UCSD Pascal is a complete programming environment containing its own file handling package and editor. The filer has the ability to perform functions of copying information between peripherals, listing directories and performing several housekeeping tasks. The editor is a screen oriented utility with the ability to move a cursor to any position of a file and perform editing functions at the cursor position. By including such an environment, use of UCSD Pascal on different microprocessors is easily learned. The filer, editor and compiler are all Pascal programs and all that is required to accommodate UCSD Pascal is a Pcode interpreter.

5.313 Portability of UCSD Pascal

UCSD Pascal is available on many microprocessors. The project has taken great lengths to ensure that compatability between the different implementations exists and by charging $20,000 for any request for source code of the system has lessened the chance of incompatable versions being created. This approach, while being somewhat dictatorial, has ensured a level of portability of Pascal programs across many machines not often seen elsewhere. The existence of a microprocessor designed purely for the UCSD Pascal system { MICR78 } is an indication of its widespread popularity and success.

5.32 The Belfast Pascal compiler

The Belfast Pascal compiler is a modified version of the original Zurich compiler { WIRT71c } . This original compiler which produces code for the CDC 6000 series was altered to produce ICL 1900 code { WELS72 } . The main differences between these two machines are that the 1900 has a much smaller word size and fewer arithmetic registers. The modifications required were not insubstantial and required six man-months effort. The compiler had to be compiled at Zurich and economic constraints restricted the Belfast team to one short trip to Zurich. In order to test the code produced at Zurich, a 1900 code interpreter was written in Pascal. This interpreter had to cater for the subset of thirty 1900 instructions which the compiler generates, and fourteen operating system routines. The compiler was tested at Zurich and a compiler version in 1900 code produced in only eight days. One intermittent error remained in the compiler which was eliminated after their return to Belfast.

Transport of a compiler from one machine to another is a large task. The P4 compiler is usually transferred by writing a Pcode interpreter for the host machine. In the case of the Belfast compiler, the code production sections of the compiler were altered. The 1900 machine architecture is similar to the CDC 6000 to the extent that a complete overhaul was not required. The Belfast team point out that the transport of the compiler 'proceeded with an ease and speed uncharacteristic of such operations' and attribute this to four factors. Firstly the Pascal language permits the compiler to be written in a clear and structured manner reducing the likelihood of errors when altering it. Secondly the system at Zurich provided fast interactive job turn around. Thirdly a carefully selected set of Pascal test programs were used to test the compiler while at Zurich. Finally the use of a host machine emulator on the donor machine allowed testing of the compiler without the need to transfer the code to the host machine until the compiler is complete.

5.33 MUSS portable compiling systems

MUSS is a system of compilers and an operating system portable over a range of machines. The heart of the compilers is the language CTL (compiler target language) {CAPO71} . CTL is a high level language and once a compiler for it is written for a machine, the different language compilers are created by adding prewritten front ends to this CTL compiler. In this manner, once the CTL compiler is implemented, new languages should be available by a minimal effort to write the compiler's front end. This process enables new languages on an existing machine to be more easily provided but transporting the CTL compiler requires the effort of writing a compiler for a language such as Algol 60. A lower level interface is provided in the form of TML (target machine language) {BARR79} . Unlike Pcode TML is designed to be receptive to a variety of languages. TML distinguishes between arithmetic on integers, for example, and address calculations such as array indexing. In this way, any existing hardware functions which are designed for array indexing or record field accessing may be exploited. TML is the MU5 target machine model {BARR79} and portability of the MUSS compilers is good over the range of MU5 type machines such as the ICL 2900. As is noted by Barrington, TML can be implemented on other machines such as the CDC 7600 but because of the architectural machine differences, the resulting efficiency would not be ideal.

5.4 Comparison of the two implementations of Pascal

This section compares the implementations of Pascal on the Digico Micro 16E and the GEC 4080. Problems met in both implementations are detailed in the following section, whilst this section concentrates on the differences as seen by the user and on the different approaches adapted for the two systems. Both systems involve the P4 compiler adapted to the local requirements but despite this large area of common software the implementation methods required and the resulting system so viewed by a user are significantly different.

5.41 The user interface

The user interface to a language implementation will largely be dictated by the operating system running on the computer. A custom built environment will often be possible for use within an existing operating system - many computers implement BASIC with its own defined environment. Pascal has no environment defined and it is assumed that Pascal should be used in a similar manner to that of other compilers present on a computer such as Fortran and Algol 60. To call a compiler and subsequently cause execution of the compiled program will necessitate using certain commands particular to the operating system existing on the machine. The GEC 4080 has an operating system and a standard method for users to compile and execute programs and it then seemed natural to present Pascal in this conventional manner. The Digico Micro 16E does not have the capability of allowing multi-access to Pascal both because of storage restrictions and because scheduling of Pascal can not be achieved by the operating system. There is no standard method of operating compilers on the Digico because there is only one compiler (BASIC) provided and this has its own environment. The Digico operating system does not provide an easy method of running programs in certain sequences and it was therefore decided, because of these considerations, to provide a Pascal environment. The user interface for the Digico is then that of a Pascal machine whereas the GEC 4080 implementation provides a GEC 4080 machine with Pascal capabilities. These two approaches raise the question of what a computer user requires - a particular language or a particular computer. Someone who is a devoted Pascal user on several machines would probably prefer each machine providing Pascal in a similar environment. The provision of a language with its own environment as standard as the language itself is the approach that tends to be adopted with microprocessors. Many 'personal computer' systems provide BASIC with its environment and Pascal with the UCSD environment as discussed in the previous section. Microprocessors, unlike mainframes, do not currently offer a host of compilers. Should they do so then the language environment approach may disappear. At the present time, popularity of microprocessors tends to place more emphasis on the software than the hardware and provision of a Pascal or Basic machine becomes the natural solution.

5.42 Storage allocation

The storage manipulation and requirement of the Pcode machine has been described in the preceding chapter. The manner of implementing this requirement on the two machines is quite different. The Pcode machine requires two memory areas, these being for holding the Pcode instructions and the program's data/workspace. The Digico allows absolute addressing of machine locations only and a rather static storage allocation approach is taken. The GEC 4080 is a virtual address machine. The two Pcode machine store areas are mapped into two of the 4080 current segment areas, as described in the preceding chapter, and by nominating segments to these areas gives the effect of a two dimensional store. The size of the two areas is independent of each other and so allows much greater flexibility than the Digico. The Digico has no provision for dynamic allocation of store to a program. This is not a problem, however, since in a single user system it is quite acceptable to allocate all the remaining store to the Pcode machine's requirements permanently. The GEC 4080 is a multi user system and it is therefore wise to only allocate as much storage area as is required at any one moment. As the stack or heap grows, more store can be allocated in certain sized amounts. The unit of store expansion ideally should be the amount of store typically used by small programs. In this way small programs would not be allocated a wasteful excess of store. The unit of store expansion should also not be so small that programs requiring steadily growing amounts of store have the overhead of frequent store requests for expansion. Experience has shown that a unit of expansion of 2K bytes would be acceptable. This however, poses a problem when considering the expansion of the heap. The heap, as discussed in the preceding chapter, grows from the top of allocated store downwards. When allocating a segment of store of 2K bytes, the required layout is as shown:

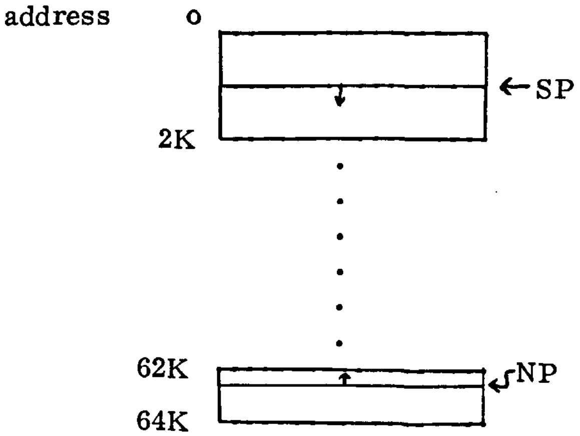


fig 5.1

However, a partial segment - that is, one allocated to be less than 16K bytes - is recognised to be at the low address end of a segment. No partial segment may be allocated starting part way along the address region it occupies. The result is that the heap must grow in units of 16K bytes. It is also unfortunate that in a modern machine such as the GEC 4080, access to a medium or large store area requires the user to explicitly overlay segments as described in the previous chapter.

5.43 Operating system interface

This section is concerned with the implementation of a suitable interface between Pascal and the operating system for the purposes of performing input and output. This particular aspect of implementation is greatly hindered if the operating system does not provide general purpose routines for dealing with different peripheral devices. The Digico provides separate 'executive calls' for each peripheral device. Each device requires a different amount of information at each call. The main details of these requirements are as follows.

5. 431 Digico VDU

This device operates on a one character buffer. There is a facility in the operating system to buffer a line of information but the editing facilities provided by it are very basic. The Pascal system builds up input from the VDU until a carriage return is received. The building up of the line includes facilities for deletion of characters typed in, deletion of the whole line, verification of the line typed so far, tabulation and detection of an 'end of file' character.

5.432 Digico discs

To access a disc file, the Digico operating system has a file handling package called DOS {DIGIb} . This routine requires the provision of a twenty five word accounting and description buffer and a four hundred and forty eight word data buffer. The Pascal interpreter packs and unpacks characters from the data buffer checking for the end of file in the case of reading a character. The file handling package includes a routine for performing disc I/O on a character basis but copies over the twenty five word description on each such call. This results in a simple file copy program taking five times longer using the supplied routine than manipulating the disc file buffer within the interpreter.

5.433 Digico printer

The Digico installation at Keele operates a Diablo 'intelligent terminal' { DIAB76 } primarily as a printing device. The terminal is connected to the Digico Micro 16E via a general communications channel {DIGIa} . There exists two protocol systems in this link - that of the channel and that of the Diablo terminal. Both protocols have to be adhered to by the program using the Diablo. The operating system requires detailed information each time the channel is accessed but leaves such tasks as checking the state of the channel to the user program.

5.434 GEC operating system interface

The GEC operating system provides a set of data management routines as an interface between the user program and the peripheral devices connected to the GEC 4080. The use of these routines is independent of the actual device being used. The basic routines are as follows:

a) Connect

Associate a channel number (in the range 1 to 12) with a specified device. The specification of a device is a. character string in the same format as used in the job control language.

b) Open

Open the connected device - or reopen this device - for reading or writing

c) Close

Close the connected device - used when all data has been read or written.

d) Get

Read one record/line of data into the specified buffer. The number of characters read in to the buffer is provided on return from this routine.

e) Put

Write one record/line of a given length from the given buffer.

5.435 Summary

It is clear from the above descriptions that the two operating systems of the Digico Micro 16E and the GEC 4080 provide a very different interface between a program and peripheral devices. The Digico does little other than simple data transmission. The program must be aware of what actual device is being used and provide memory areas whose use and size is completely device dependent. The GEC 4080 provides a much more flexible approach. The user does not need to be aware of what actual device is being used. The memory space provided and the routines called being the same whether the program is writing a disc file, printing the file on a line printer or punching a file on paper tape. The flexibility afforded by this approach has two effects. Firstly the programmer is not impelled individually to cater for a range of peripheral devices. Secondly, every systems program using these routines can automatically handle the full range of devices available thus ensuring I/O compatability between existing and future programs. The Digico approach requires each site to produce its own set of routines to drive peripheral devices. This results in a great deal of unnecessary duplication of effort and a likelihood of data being incompatible between different Digico machines.

5.44 Interpretation of Pcode instructions

The P4 Pascal compiler is intended to be a portable compiler. With this aim, the code generated cannot make sweeping assumptions concerning the actual machine instructions available. A good example of this concerns the Pcode instructions to manipulate sets. Some of these are:

INN	Test if a given value is present in the set.
SGS	Generate a singleton set
UNI	Generate the union of two sets.
INT	Generate the intersection of two sets.

In both implementations a set is represented as a group of consecutive words, each bit representing the presence of a set member. One word will not suffice as in both implementations this would require a restriction on the size of sets that would not permit execution of the compiler. A member of a set is then represented by two parameters: the relative word within the set and the relative bit within that word. The above four instructions are then interpreted as follows:

m	is the relative word of the set for this value.
n	is the relative bit in word m for this value.
s	is the number of words comprising a set.

INN	Test bit n of word m.
SGS	Set all s words to zero. Set bit n of word m
UNI	Perform a logical 'or' on each s pairs of words
INT	Perform a logical 'and' on each s pairs of words.

The Digico Micro 16E can only reference a particular bit by logical shifts to a testable position - the sign bit. The Digico does not have a logical 'or' construction. The GEC 4080 has instructions for testing or setting any bit within a word and has both 'and' and 'or' instructions. Hence the Digico is fairly slow in interpreting set instructions while the GEC 4080 is relatively fast to perform such instructions. As an example the following operations were tested on each implementation:

(a) ch in ['0' . . '9']

(b) (ch <= '9') and (ch >= '0')

These two Boolean constructs are common methods of testing whether a character is one of the numeric characters. The tests were placed in a Pascal for statement and the program execution time noted. The time taken for the same program without the Boolean expression was then subtracted and the following times represent the number of seconds to evaluate the two expressions 20,000 times.

machine	expr	time (secs)	relative time
Digico	(a)	11	1
Digico	(b)	7	0.64
GEC 4080	(a)	1	1
GEC 4080	(b)	2	2

It is clear then, that the quicker way to test whether a character is a numeric character depends on the machine being used.

A second test was performed to give an indication of the time taken to perform floating point instructions. The Digico implementation requires a software floating point package while the GEC 4080 contains floating point hardware. The two expressions evaluated were:

(c)	I + I - I * I DIV I	I is <u>integer</u>
(d)	X + X - X* X / X	X is <u>real</u>

The repetition was again 20,000 and the results obtained were:

machine	expr	time	relative time
Digico	(c)	16	1
Digico	(d)	154	9. 63
GEC 4080	(c)	9	1
GEC 4080	(d)	9	1

The point is again made that the speed of certain constructs can vary widely on different implementations. A particular program written in Pascal may be portable between two implementations but its relative efficiency to a similar program will not necessarily be the same.

5. 5 Particular portability problems encountered

Some of the problems which may arise when transferring a program have been discussed. The Pascal P4 compiler has been transferred twice as described: firstly from a CDC 7600 at UMRCC to the Digico Micro 16E at Keele; and secondly from the Digico to the central service machine at Keele, the GEC 4080. Accompanying both implementations were several events which hindered the transfer. The more notable problems encountered are now described as an illustration of the more general obstacles to portability already discussed. These illustrations range from incompatabilities between the machines to problems stemming from the language itself.

5.51 Character code differences

The Pascal P4 compiler is claimed to be independent of character set differences { JACO76 } . This claim is supported by the fact that the Pcode produced contains character constants represented as readable characters rather than integer codes. In the process of transporting the compiler, these characters are produced within the Pcode form of the compiler on the donor machine. The P4 compiler does, however, use the internal code of characters when producing a jump table for a Pascal case statement. This problem was encountered with regard to the scanner section of the compiler. This section contains a case statement for actions to take depending upon the character read. The code produced for a case statement is as follows. Consider the statement:

```
case  ch  of             (* ch is of type char *)
          'A' : S1;
          'B' : S2;
          'Z' : S3
          end;
```

The compiler will first calculate the range of the case variants. In this case it is 'A' to 'Z'. The compiler then produces a jump table of this size. In this case it might appear as:

```
 1)  JUMP   L 9      'A'
 2)  JUMP   L10      'B'
 3)  ERROR           'C'
 4)  ERROR           'D'
 .
 .
 .
26)  JUMP   L11      'Z'
```

The code produced to operate this table consists of loading the ordinal value of the character, subtracting the ordinal value of the character, specified in the case statement, of lowest ordinal value and then using this value to perform an indexed jump into the jump table. For the CDC 7600 the code for the character 'A' is 1 and the code produced for the above statement is then:

```
load ordinal value of character
subtract 1
Index jump in jump table
```

When this code is then obeyed on the Digico it will fail. The Digico represents the character 'A' by the code 65 and as a result the wrong entry in the jump table will be selected (if it exists). This has the effect that any Pascal program (including the compiler) cannot be compiled on the donor machine and executed correctly on the host machine if it contains a case statement, the case variable is of type char and the two machines have different character sets. The P4 compiler contains such a case statement and three options are available to overcome this problem:

1 Amend the Pcode produced to subtract the correct number
2 Amend the compiler so that it does not use a case variable of the type char.
3 Amend the compiler on the donor machine to be aware of the character set when dealing with case statements.

None of these solutions is entirely satisfactory. The first solution is rather untidy. The Pcode has to be examined to locate the necessary instruction and this then amended. This is something which has to be done each time the compiler is compiled on the donor machine and this process may have to be repeated several times before a satisfactory version of the compiler can run on the host machine. The second solution makes the compiler completely independent of character set differences but not other programs. When building the interpreter it is wise to test it with several sample portions of particular Pcode instructions. This is achieved by compiling small test programs, each utilising a particular language feature, on the donor machine and attempting to interpret the resultant Pcode on the host machine. The second option rules out the ability to test such a case statement. The third solution is the option chosen in this particular case. It is not wholly satisfactory in that it has to be repeated to transport the compiler to another machine with yet another different character set. The P4 compiler running on the donor machine is then altered to amend the code produced for case statements. This compiler is referred to as the cross compiler in the previous chapter. The compiler which will run on the host machine does not require amending.

5.52 Language differences

The language compiled by the P4 compiler is slightly different to standard Pascal as detailed in appendix 1. The particular difference noted here is that P4 Pascal does not recognise the standard procedure dispose. Instead it uses two procedures mark and release which remove items from the heap in a stack-like fashion. The P4 compiler uses the procedures mark and release to release the name tables associated with the local variables of a procedure when it has finished compiling that procedure. When the P4 compiler is compiled on the donor machine under a standard Pascal compiler, it will fail because it has used the unknown procedures mark and release. This can cause the implementer some effort in rewriting this section to 'dispose' of each name table entry individually. Fortunately in this particular instance the CDC 7600 Pascal compiler does recognise release as a standard procedure. It does not recognise the procedure mark but as mark is equivalent to a call of 'new' with the parameter being a pointer to a pointer this is not a large problem. The cross compiler is then altered to call 'new' instead of 'mark'.

5.53 Standard types - size and alignment

Several problems arose during the course of implementing Pascal on the two machines due to different Pascal object types occupying different amounts of store. These problems are mainly concerned with the alignment in store of different sized objects and the manipulation of the stack which grows on fixed units.

5.531 Alignment with records

Consider the record type:

```
record
b : Boolean
i : integer
end
```

In the GEC 4080 implementation, Booleans and chars occupy successive bytes while integers occupy four bytes aligned on a four byte multiple. The above record would then occupy the following layout in store:

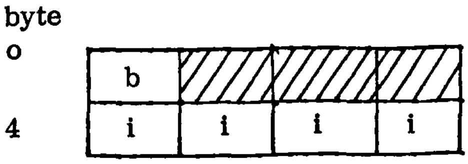


fig 5.2

The shaded area is unused and inaccessible to the Pascal program. Without initialisation of all store when this object is created, either on the stack as a local variable or on the heap by a call of new, the shaded area will contain undefined information. This can result in two records which contain identical Pascal fields being compared and not found equal. The Pcode instruction for comparing two records is given the addresses of two records and their size and so the undefined sections will also be compared. The most practical solution to this problem is to initialise store when allocated - in the GEC 4080 implementation, store is initialised to zero.

5.532 Stack manipulation

The stack, as implemented on the GEC 4080, grows in units of four bytes. Problems can arise when an object is loaded on the stack which is not four bytes in size. In the case of characters and Boolean variables their size is one byte. The most natural way of accommodating one byte objects on the stack would be to place them in the least significant byte of a stack word. This is certainly the easiest way for the interpreter. The stack is, however, also used for the passing of parameters to procedures and functions and for the return of function values. In this manner, objects loaded onto the stack may be subsequently accessed as a normal variable within the local data of a procedure and vice versa. The compiler demands the alignment of one byte objects to occupy the most significant part of a word where possible and this is then the position in which characters and Booleans must be loaded onto the stack.

The second problem which arises with manipulation of the stack concerns sets. Sets in both implementations occupy more than one stack element and therefore require multiple stack operations. This in itself is a matter of occasional inconvenience within the interpreter but does in fact have a serious consequence with regard to the c alculation of stack space required which is performed by the compiler. On generating each instruction that alters the size of the stack, the compiler calculates the maximum size the stack could possibly grow to within each procedure. This value is later used to update the EP Pcode register as described in the preceding chapter. This EP register is used to determine whether the stack could grow beyond the maximum storage space available. The compiler assumes that the stack grows in fixed sized elements. An instruction to load a <u>set</u> onto the stack would however increase the stack size by a greater amount than an instruction to load an <u>integer</u>.

The compiler could be altered to take account of the known size of the object(s) being added or removed from the stack by each instruction. This would, however, add a relatively costly piece of coding into an already critical section of the compiler reducing compilation speed by a noticeable amount. Alternatively the compiler could assume the worst case that all objects loaded onto the stack are sets. This could clearly be wasteful and cause a program to 'run out of store' prematurely. The size that the stack is assumed to grow by could be taken as some calculated average but this approach is error prone if a lot of set manipulations are performed in a program. The final alternative is to only place the address of a set on the stack. This suffers from the disadvantage that temporary storage may need to be allocated in interpreting a set expression - an unfortunate step to take for a stack oriented machine. The solution adapted in both implementations is to assume the worst case - that the sets are always being loaded onto the stack when evaluating an expression. This does not, in practice, lead to a greatly exaggerated prediction of the stack growth as the amount of store used for the purpose of evaluating expressions is usually small compared to that used for local variables and the exaggerated increase on possible stack size requirement for a procedure is relatively small.

5.54 Input/output

The problem that caused most concern with regard to input and output is that of an interactive device such as a VDU being used as a Pascal file. Pascal defines that after a call of the standard procedure reset, the first file component is available to the program. For instance, if the standard file input is connected to the VDU, then at the start of a program, the file component input↑ is the first character typed as input to the program. This requires the user to type the first character, or more usually the first line, of input before the program commences. This is neither natural nor desirable. Two solutions to this problem have arisen in known implementations. UCSD Pascal has redefined the standard file input to be of a new standard type 'interactive'. This type is the same as the type 'text' except no character 'look ahead' facility is provided. With a text file, the operation of the call read (ch) where ch is of type <u>char</u> is equivalent to the sequence:

```
ch:=input↑  ; get (input)
```

The UCSD type interactive defines the sequence to be:

```
get (input): ch : = input↑
```

This suffers from two disadvantages. Firstly no look-ahead facility is provided which can hinder certain programming techniques. Secondly this approach severely hampers portability of programs between standard Pascal and UCSD Pascal installations.

The second solution is the one adopted in the two implementations. The procedure reset, when applied to files of type text does not cause any input to be taken from that file. Instead, the file buffer contains a space and the function eoln delivers the value true. This is the same as saying that on calling reset, the file pointer is at the end of an imaginary line immediately preceding the first line of the file. This approach does not seriously hamper portability as a call of either 'readln' or 'get' applied to the file immediately after a call of reset will position the file buffer at the first character of the file as defined by the Pascal report. This approach has been supported by Dijikstra {DIJK79} as the best method of overcoming the problem of interactive Pascal files.

<u>5.55 Storage constraints for compiler bootstrap</u>

Having implemented Pascal on a computer, the problem of maintenance of the compiler arises. The original Pcode form of the compiler is produced on a different machine. If any change is to be made to the compiler, for example to tune it to its environment, it is desirable if the process of recompiling the compiler can be performed on the host machine thus removing problems of transporting the Pcode. The compiler is a large Pascal program and in order to compile it a large amount of store is required. This store consists of approximately 65K bytes to hold the Pcode of the running compiler and 48K bytes of data storage to compile the compiler. Storage space on the machine is also required for the interpreter and operating system. This large store requirement is not available on the Digico Micro 16E which has 128K bytes of memory. In order to bootstrap compile the Pascal compiler it is therefore necessary to use other storage media. The compiler can be overlayed or paged in and out of backing store. The Digico interpreter was adapted to page the Pcode form of the compiler between main memory and disc such that only half of the Pcode form of the compiler is present in main memory at any time. This has the effect of reducing the compilation speed considerably. The compiler takes approximately two hours to be compiled using this method.

5.56 Word size limitations

One particular problem arising during the transfer of the Pascal compiler from the Digico Micro 16E to the GEC 4080 is due to the smaller word size of the Digico. Integers are held in 16 bit words on the Digico and in 32 bit words on the GEC 4080. The Pascal predeclared constant maxint, being the maximum legal integral value, is then 32767 and 2147483647 respectively. The compiler destined for the GEC 4080 must then be capable of allowing integral constants of absolute value less than or equal to 2147483647. This value is conveyed to the compiler in the constant declarations specifying the machine dependant parameters of the host computer. The P4 compiler destined for the 4080 then has to be compiled on the Digico which will reject this specification of maxint as too large for it to accommodate. This is a portability problem hindering any move of the P4 compiler from a 'small' to a 'large' machine. The problem is solved by specifying the constant maxint as being any number small enough for the Digico compiler to accept. The resulting Pcode is then edited to alter the single occurrence of this value to the correct value.

5.6 Conclusions on portability

Several points have been made relating to the portability of software in general and in particular, the portability of the P4 compiler. The meaning of the term portability and its application to the P4 Pascal has been discussed. This section attempts to expand that discussion within the framework of the two implementations of this compiler described.

5.61 Non standard Pascal features

The British standard draft report on Pascal {ADDY78} recognises that implementations of Pascal will often contain non-standard features of and extensions to the language. The language Pascal is designed to be capable of efficient implementation on as many machines as possible. This design feature restricts the features of Pascal to those readily available on most machines. Some features which are commonly added to Pascal are:

1) A facility to pass a simple message to the operating system, usually to indicate the success of the program.
2) Environmental enquiries of, for example, the time and date.
3) Pseudo random number generation.

Pascal does not expressly forbid such extensions. The proliferation of particularly these three extensions is large as is the number of different ways in which they are presented. In recognition of such common extensions it could have been helpful for implementers of Pascal to have been given guidelines to their implementation. Such guidelines could be presented by defining Pascal under two headings. The first heading could be the standard language which should be complete in each implementation; the second would be a set of extensions to the language which are strictly to be considered non standard and their inclusion in an implementation optional. For example, under this heading there could be a standard function called 'time' which is defined to give as a result the time of day in seconds. Some implementations may find the presentation of such a function in a different manner more convenient. In this case the implementer is free to do so but should not use the name 'time'. These guidelines for extensions would greatly ease the portability of Pascal. It is a fact of life that programs will use extensions to languages. If portability problems due to such use can be lessened by certain conventions then this must be a worthwhile exercise.

5.62. Portability and efficiency.

The usual method of implementing P4 Pascal is to construct a Pcode assembler and interpreter. The efficiency of such a system will depend upon the effort in constructing the interpreter. A Pascal system can be made available quite quickly by constructing an interpreter which contains routines to execute each Pcode instruction written with the main motivation being speed of constructing the interpreter rather than the interpreter's subsequent running speed. In other words, it is not just the question of how portable P4 Pascal is but also how portable an efficient P4 Pascal system is. In the instance of the GEC 4080 implementation, it could have been declared that P4 Pascal was available after little more than a month of work. The system in that embryonic stage, whilst able to compile and run Pascal programs, was very difficult to use and very slow. The system then had a very primitive user interface, it was tediously difficult to access disc files, using more than a certain amount of store resulted in halving the program's execution speed and the compiler was not, at that stage, shareable between several simultaneous users. Experience with both implementations suggests that the core of the system - a primitive interpreter - is a relatively small part of the effort required to produce a useable system. The majority of the time is spent on tuning this core, and creating the periphery connections to fit the system neatly into its environment. This stage involves the order of six times the effort required to build the basic interpreter. When writing a compiler from scratch this second stage would normally require less effort than writing the core of the compiler. This reversal in these two relative efforts is evidence of the success of the P4 Pascal compiler.

Another aspect of efficiency when considering portable software is that of catering for several possible eventualities. The P4 compiler is parameterised to cater for different store allocations and alignments to variables when assigning addresses. In this case, a procedure is obeyed when a variable is to be allocated an address which aligns the address of the next free location according to the alignment required by that variable. In some implementations each type of variable will have the same alignment requirement and this section of the compiler is an unnecessary overhead. In this case it may be worth altering the compiler to suit one particular machine. To do this wherever possible would take a large degree of effort and this effect could be achieved automatically by some form of conditional compilation mechanism.

5.63 Portability of Pcode

The Pcode Pascal P4 compiler produces Pcode machine instructions which are usually interpreted by a host machine. The portability of this compiler then largely depends upon the portability of Pcode. The success of a machine code designed for a particular language and its implementation depends on three factors.

Firstly the machine code must be designed to minimise the mapping of Pascal constructs onto it. This factor is responsible for Pcode being based on a stack machine. A stack machine enables a natural implementation of recursive procedure calls and expression evaluation where the syntactic definition of a Pascal expression is recursive. Pcode contains instructions whose method of addressing variables has two components, a base and an offset. This feature enables ready access to variables declared in outer procedures that are neither local to the currently existing procedure nor global to the program.

Secondly Pcode must be capable of efficient interpretation in order that interpretive implementations afford an acceptable execution speed. For this to be true, Pcode instructions should be capable of ready interpretation without regard to the Pcode instructions either about to be interpreted or which have been interpreted. In other words, a Pcode instruction should contain all the information required for its interpretation without considering the context of its use. Thirdly Pcode should be designed so that few assumptions are made about the architecture of potential host machines.

The second and third factors are in a sense contradictory. The more high level the target machine code is, the easier the task of the compiler as more effort is moved from the compiler to the interpreter. At the same time, the more low level the target machine code, the more likely the interpreted execution of a program is slower. For example consider the set operations as discussed in section 5.44 of this chapter. The operation of testing for set memberships has as operands a set and an integral value. This is a high level instruction requiring a greater than average effort of implementation whilst being more difficult to implement than most other Pcode instructions it was a wise decision to include this instruction at this level. The interpretation of this instruction is usually that of testing whether a particular bit is present in the bit pattern comprising the set. If the design of Pcode were to break this task down into several lower level tasks it would have to assume a generally available method of testingthe presence of a particular bit. As described previously, the Digico Micro 16E implementation tests the presence of a bit by performing a logical shift on the relevant word of the set such that the selected bit occupies the sign position of a word. This is then tested by examining the sign of the result. This is the only method of performing this test on many computers. The GEC 4080 implementation makes use of this machines ability to test the presence of any selected bit of a word and this is faster than shifting the bit into the sign position. In this respect the set membership instruction is high level compared to the Digico instruction set but equal to the power of a particular GEC 4080 instruction allowing the most efficient method of interpretation in both implementations.

The balance between these two factors of allowing ready interpretation of Pcode instructions without enforcing a potentially inefficient result is quite critical. The two implementations described here have been based on a simple machine (the Digico) and a relatively more powerful machine in the form of the GEC 4080. Implementing the same compiler on two very different machines has shown that this balance has been chosen well. Pcode interpretation is fairly complex on the Digico whilst on the GEC 4080 the opportunity to utilise many of this machines more powerful instructions was welcome. The discussion shows that portability considerations do not just include the question of how fast can P4 Pascal be implemented on a machine but also that of how efficient the resulting code will execute.

5.64 Portable environments

In section 5.41 the environment surrounding Pascal implementations was mentioned. The lack of an adequate user interface on the Digico Micro 16E provoked the creation of a user environment in which Pascal is used. Generally, all compilers on a particular machine are invoked in a similar manner. Programs are stored and edited using the same system for each language. Some languages such as Basic, have an environment defined as part of the language and in most instances the user sees little difference in the way Basic is used on different machines. Such an environment is usually created for languages which are primarily intended for interactive use such as Basic, POP2, and UCSD Pascal as previously described. The interactive use of a language involves more than presenting a compiler with a program as in a batch system. The user develops and edits a program on-line and the commands used for these purposes can become as much a part of the language as the program statements themselves. In this case, any facilities offered by the operating system which are not included in the defined environment for a language such as Basic can be considered as extensions to the language. The portability of such an environment is subject to the same limitations as the portability of program constructs. The filing and editing commands must be such that they can be implemented on most machines if the language is to be widely used. Defining an environment for a language has some advantages for the user. If a computer user generally uses one language interactively, then portability of that language, as seen by the user, is greatly enhanced if the environment remains the same. The problems of learning a new job control language are eliminated. The main disadvantage to this definition of an environment is that production programs created under it are usually only capable of execution by entering the particular environment. This may inhibit the use of such programs under some operating systems which assume production programs run under the operating system's environment. The environment also inhibits portability between languages. The user of several languages will have to be aware of several environments and this is undesirable. The inevitable conclusion to this discussion is to enjoy the benefits of both systems by defining one environment for all languages. Whilst some work is in progress to define portable operating systems {FRAN77} , this has shown few signs of achieving general acceptance by computer manufacturers.

5.65 Increased portability

In this chapter the meaning of the term portability has been discussed and attempts made to identify the factors which contribute or otherwise to it. Some particular recommendations have emerged which can have the effect of increasing portability. In concluding this chapter these recommendations are highlighted and a definition of the term portability given.

Throughout this discussion, portability has been used to give some measure of the effort involved in moving a piece of software from one machine to another (or one language to another) and the degree to which the host and donor implementations behave in the same way. It is not sufficient to move a compiler from one machine to another in the easiest possible manner if the result is severely different from the original or considerably less efficient than it could be. Portability of Pascal does not just mean making the language available but making an efficient and useable implementation of the language available. For this reason, a potential implementer must be prepared to put in some effort to produce such a useable and efficient system on the recipient machine.

Certain language features as described in section 5.21 can arise in programs which are written in the same language as defined but which behave differently on different implementations of that same language. The order of evaluation of expressions is one quoted example. Programs will be less portable where these ambiguities arise and any definition of a programming language must resolve such eventualities. Language definitions ought to pre-empt the inevitable enhancements that will arise and whilst not necessarily including those enhancements which are not generally available give conventions for their supplementation. This would result in a small increase in portability where implementations adhere to these conventions.

The portability of a language can be threatened by the inclusion in that language of characters not generally available. There are a limited number of characters which all computers recognise (those represented by ASCII code 32 to 96) and a language that requires distinction between upper and lower case letters for instance will give rise to programs that are extremely difficult to move to a machine with a limited character set. APL is a language that requires a very large character set and any implementation often requires new hardware to cope with it. This is an extreme example of a language that severely injures its own portability by this requirement.

The problems involved in the physical transfer of a piece of software were detailed in section 5.24. The acceptance by computer manufacturers of the difficulties involved in this process is paramount. It would be a very positive step for computers to be produced with a very simple communications channel for this purpose. The channel could appear as simple as a paper tape station where characters are sent and received individually. It is suggested that no sophisticated protocol should be implied for such a device, leaving the actual transfer of information at a very low level - a level that is easily recognised by other computers.

In section 5.62 the possibility of some form of conditional compilation was suggested in order that the P4 compiler can be tailor made for particular installation parameters. This has been studied by the group that wrote this compiler {WIRT74} but no progress in this direction is apparent. The conditional compilation can be performed by a special program which produces a particular version of the compiler given certain parameters. An alternative is suggested by Bhaskar in Pascal News number 15 whereby all parameters specified as constants can be detected by an optimising compiler. For example:

```
const versionl = false;
  .
  .
  .
  .
if versionl. then Sl;
```

The above statement could be ignored by an optimising compiler. This solution is fine if such a compiler is available - the P4 compiler does not optimise code in this manner. A solution suggested above of having a program to custom build a compiler from a general compiler appears much easier to construct and would achieve the same result. This would not be conditional compilation as the compiler would be presented with a complete program.

Finally, this discussion concludes with the term portability. This term cannot be applied as an absolute but is a measure of effort required. There is no program that is 'portable' to the extent that at a flick of a switch a program appears on another machine working in exactly the same way and as fast as it does on the donor machine. In the same way there is no program that in principle cannot be moved to another machine - unless it makes essential use of particular hardware devices. The term that is of interest is how portable a program is, how difficult it is to transfer to another environment. Any new feature of computing, whether hardware or software should consider portability as a parameter to be maximised particularly if the effort involved in maximising portability is less than the decreased effort required by all subsequent implementations as a result. Jacobi {JACO76} in reference to the portability of P4 Pascal states 'The work required to move an especially machine and system dependent piece of software - like a compiler - is not negligible. However, if the work required is approximately an order of magnitude less than that for writing the whole software from 'scratch', we would consider the method as viable for the purposes of portability'

CHAPTER SIX

IMPLEMENTATION OF THE DIAGNOSTICS SYSTEM

6.0 Introduction

This chapter describes the design and implementation of a novel run-time diagnostic system. The system is designed for the Pascal implementation on the GEC 4080 computer. The system comprises three parts; these are alterations to the P4 Pascal compiler and the Pcode interpreter and a diagnostics program. The latter is a Pascal program which may be invoked by the Pcode interpreter under user control. The features provided by this diagnostic system were outlined in chapter three. The system was written with the aim of keeping inherent overheads incurred by its use to a minimum. As little as possible time overhead is involved when the user's program is running. The user is provided with a two state environment. Either that user's program is running or the diagnostics program has been invoked. The diagnostics program provides the user with an interactive environment in which the program being diagnosed can be fully inspected in source language terms. The two facilities that are of major interest as a contribution to the area are those of displaying the shape of linked structures diagrammatically and providing names for dynamically created objects on the heap. These two facilities are considered to bridge the current gap between available diagnostic systems and modern high level programming languages. This chapter describes the implementation details of this diagnostic system and includes several illustrations of its use.

6.1 Method of operation

When using the diagnostics system, there exists two states apparent to the user. Either the user's program is running or the diagnostics program is being used. When the user program is running, the interpreter performs basic monitoring tasks which determine whether to invoke the diagnostics program. The system can be viewed diagrammatically as shown:

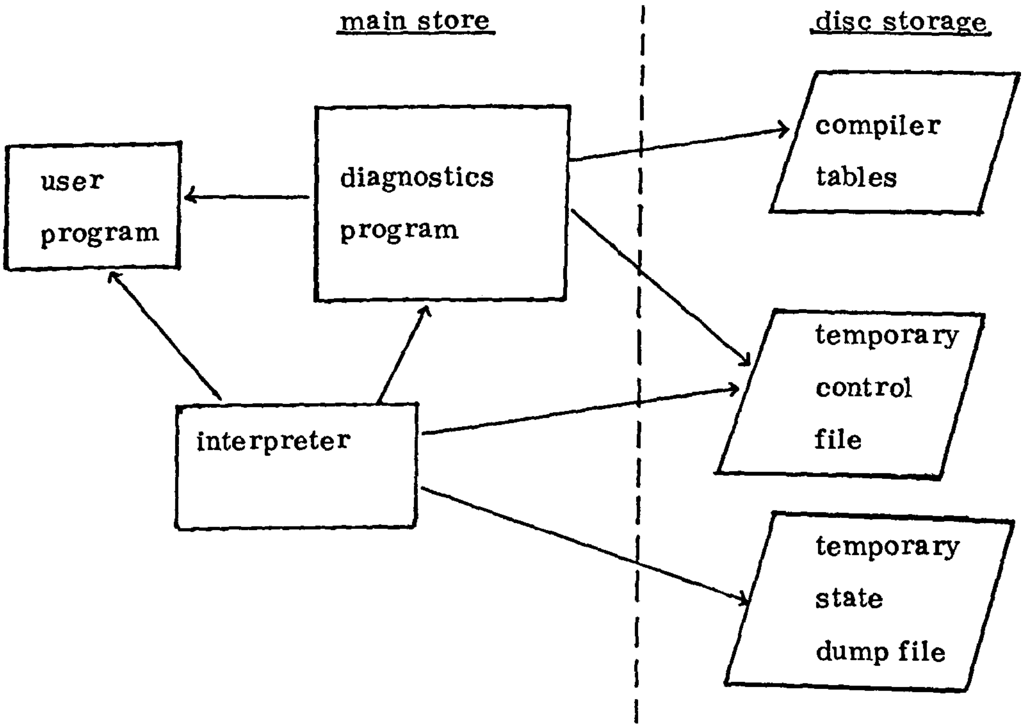


fig 6.1

The direction of the arrows shows the access each unit has - for example the interpreter controls the user and diagnostics programs and two disc files. The interpreter has two modes of operating. It is either interpreting the user program or the diagnostics program. The temporary control file communicates to the diagnostics program the state of the user program. The information placed in this file by the interpreter consists of the reason for halting the user program, the size of the user program's stack and heap and the location at which it was halted. The temporary state dump file is a core image of the volatile workspace of the interpreter. This workspace comprises those interpreter variables which are overwritten by invoking a second Pcode program - the diagnostics program. This workspace is 100 bytes long. The workspace can be dumped or restored by one simple I/O operation. The volatile data pertaining to the diagnostics program is not dumped when the user program is re-entered. The diagnostics program is always invoked from its initial state; knowledge of its previous state is irrelevant to the working of this system. When the diagnostics program is invoked, it first reads in the control file created by the interpreter. Depending upon where the user program was suspended, the diagnostics program selects the compiler tables relating to the user variables extant and assimilates them. The diagnostics functions performed by the interpreter are kept minimal so that when the user program is running, few overheads exist due to the use of the diagnostics system. As will be described in more detail below, the extra functions performed by the interpreter are detection of the break point, a user interrupt and program error and the updating of the statement counts for production of a profile by the diagnostics program. Unless otherwise requested, the compiler produces a special Pcode instruction at the start of each Pascal statement which starts on a different line from the previous statement. On encountering this instruction, the interpreter checks for user interruption or the existence of a break point and then increments the line count for later production of the program profile. This special Pcode instruction also contains the source line number of the program line corresponding to the Pcode instructions following. Detailed explanation of the mechanics of these diagnostic functions are given below.

6.2 Changes to the compiler

There were two main alterations to the compiler in order toprovide the information required by the diagnostics system. These were the production and output of the compiler's name tables so that source identifiers can be interpreted by the diagnostics program, and the insertion of code at the beginning of each source line that contains Pascal statements. This code is used to implement the break point facility and profile. In addition to these two changes, several minor alterations to the compiler were necessary. The diagnostics program is written in Pascal and extra predeclared routines are necessary in the compiler to perform actions such as interrogation of the contents of the user program store area by the diagnostics program. The diagnostics program must also have a mechanism for signalling to the interpreter that the user program can be resumed. The alterations made are detailed below.

6.21 Compiler table production

The compiler tables are discussed in chapter four. The space occupied by that part of the name table which describes objects local to each procedure or function is reclaimed upon completion of the compilation of that procedure. Immediately prior to this reclamation, the section of the name table no longer required by the compiler is written to a disc file. By selective reading of this file, the tables can be reconstructed to correspond to any required lexical position of the user's program. The compiler's name table is composed of records of type 'identifier' describing each identifier and objects of type 'structure' containing type descriptions. These two structures were described in chapter 4. In addition, a new object named blockrec is defined. This record is placed in the file to identify the procedure to which the following objects belong. The record contains the level of the procedure and the first and last Pcode locations occupied by that procedure's code. The file produced is declared as:

```
tablefile : file of record
                ptrval : integer;
                case rectype : filrectype of
                    XCTP : (XCTPA : identifier);
                    XSTP : (XSTPA : structure);
                    XBLK : (XBLKA: blockrec)
                end
```

The records of the file can then describe identifiers, types or blocks. The integer ptrval is the ordinal value of the pointer that pointed at either the identifier or structure type record while in store during compilation. This value is used by the diagnostics program to regenerate the tables as described later. To write all identifier types to the file, the name display binary tree for the required level is scanned in preorder fashion listing all identifier types. For each, at least one structure type is associated. Several identifiers will refer to the same structure type; for example all identifiers of type integer will point to one structure describing an integer. In order to avoid duplication, an extra field is introduced into structure types. This is a Boolean field which indicates whether this structure type has already been written. In order to indicate the resulting order of the table file, consider the following program structure:

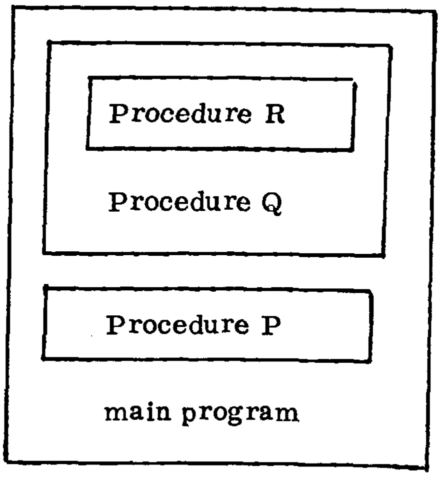


fig. 6.2

The file will be created as:

Block record R - level 3
identifiers/structures of R
Block record Q - level 2
identifiers/structures of Q
Block record P - level 2
identifiers/structures of P
Block record main program - level 1
identifiers/structures global to program

Block record predeclared - level 0

identifiers/structures of predeclared items

fig 6.3

6.211 Variant records

One alteration to the definition of the objects of type structure was necessary. This concerns the way the compiler treats variant records. Consider a record declaration:

```
record
a, b : integer;
case d : integer of
     o : (x, y : real);
     1 : (w, v : char)
end
```

The compiler does not associate the existence of the fields x and y with the value o of d. This correspondence is required in order that the diagnostics program can reconstruct the type definition. The type structure was altered to include a field connecting variant values with the associated fields.

6.22 Production of line code

The compiler produces two Pcode instructions marking the start of each source program line containing compiled statements. The first instruction contains the source line number and the second is zero to accommodate the construction of the program profile. The interpreter recognises the existence of the profile count and increments it on encountering the line instruction; it then skips over it. At the start of each statement the compiler produces these instructions unless the previous statement was on the same line. In this way, a line instruction does not occur within a basic Pascal statement. These line instructions mark possible break points set by the diagnostics package which clearly should not occur within a statement to avoid confusion in use.

6.23 Control of compiler operations

These extra functions of the compiler are performed by default. Should the programmer not wish to use the diagnostics system these functions can be inhibited. They are controlled by two compile-time options. Compile-time options were described in chapter 4. Two existing options were adapted. The D option existed to control the production of code to test the violation of subrange limits and the T option existed to control the original production of readable compiler tables. The P4 compiler contained routines to include a textual dump of the compiler tables in order to test the compilers operation. These routines were adapted to produce the name table file as described. The D option was used to additionally control the production of line codes. The T option controls the creation of a name table file. To inhibit the production of this file slightly decreases compilation time and reduces the size of the final object code file which after assembly of the Pcode contains the name table file. The absence of the name table file results in the interpreter not invoking the diagnostics package. If the line code is present, the interpreter will specify the line number where an error is detected. By default, both the T and D options are set upon initialisation of the compiler.

6.24 Additional predeclared procedures

The diagnostics package is written in Pascal. This means that all its functions must be expressed in Pascal; there is no facility to include, for example, machine code routines. The package has to perform four functions which are particular to its operation:

1 Examine the contents of the user program store

2 Examine the user program code to produce a profile

3 Alter line instructions in the user code to implement breakpoints

4 Instruct the interpreter to resume execution of the user program.

To permit this communication use was made of the function environ (described in chapter 4) and two procedures and two functions were predeclared. This use of environ is described in the next section accomplishing the fourth function listed above. The additional predeclared items are:

```
procedure pokec (a, v : integer);
This places the integer v in Pcode code location a
function peekc (a : integer) : integer:
Returns the Pcode instruction at location a

procedure pokes (a, v : integer);
As pokec but a refers to a Pcode store location.
function peeks (a : integer) : integer;
Returns the value of the Pcode store location a as an integer
```

For the reasons given in chapter 3, pokes is not used to alter variables' values and is included for completeness only. The compiler generates previously unused Pcode instructions on encountering these procedures/functions.

6.3 Changes to the interpreter

Several alterations were necessary for the interpreter to provide the environment required by the diagnostics program. These included being able to accommodate two Pcode programs - the user's program and the diagnostics - and switch between them. This change was the largest. The interpreter also had to implement the new instructions generated by the compiler for marking source lines and interrogating the user program storage area. These alterations are now described.

6.31 Two state operation

At any instant, the interpreter is obeying either Pcode instructions belonging to the user program or those making up the diagnostic program. The Pcode store area is also partitioned into two sections one for each program. At the basic level, this is achieved by setting three data variables:

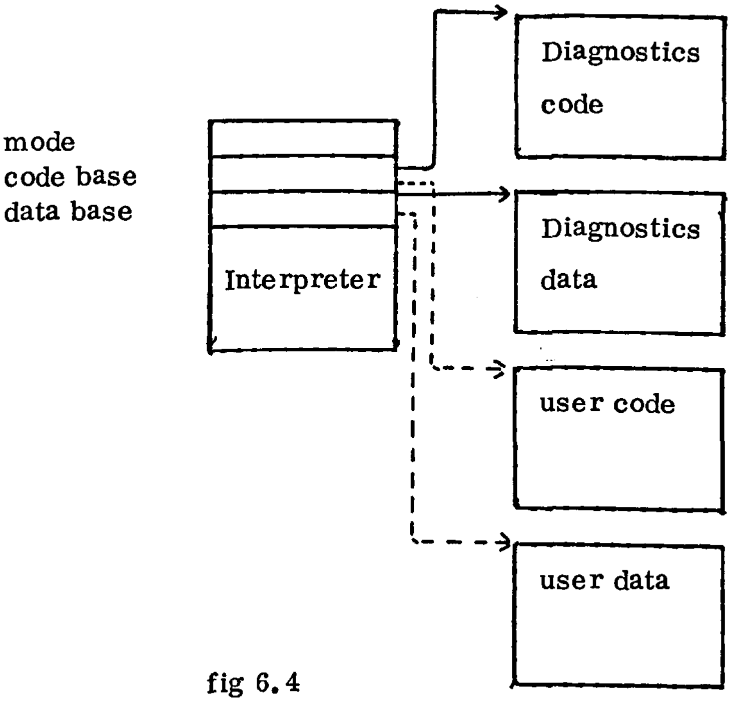


fig 6.4

The mode variable is a flag set if the diagnostics program is being obeyed. The code and data base variables signify the start of those respective areas. In practice these variables name the GEC 4080 segments which are overlayed into the current segments one and two as described in chapter 4. Several other tasks are performed in order to accomplish the switch between these two programs. The twelve I/O streams available to a GEC 4080 program are divided between the two programs.

The diagnostics program requires three channels for interactive input and output and for reading disc files such as the name table file. This allocation of physical channels is also performed by altering two variables specifying the first and last stream available. In this way, the I/O sections of the interpreter function with little alteration. As described in section 6.1, two temporary disc files are used to enable a switch between the two programs - one for communication of the state of the user program to the diagnostic program and the second to hold the state of the interpreter workspace prior to the suspension of the user program. The operation of the swap between the two programs is as follows:

Switching from the user program to the diagnostics program.

1) Create the control file containing information on where and why the user program has been suspended.

2) Write the volatile workspace to a disc file.

3) Set the store/data base variables to the diagnostics areas. Set the mode flag. Set the I/O channels available.

4) Set the Pcode registers to their initial state and allocate data storage for the diagnostics program.

5) Continue interpretation of Pcode instructions.

Switching from the diagnostic program to the user program.

1) Release the diagnostics program data area.

2) Restore the volatile workspace from disc file. This includes the restoration of the Pcode registers.

3) Restore code/data bases. Unset the mode flag. Reset the available I/O channels.

4) Continue interpretation of Pcode.

There are four ways of involving a swap to the diagnostics package:

1) By specifying entry to the diagnostics prior to the program starting.

2) Upon encountering a user set break point.

3) When manually interrupted by the user.

4) In the event of detecting a run-time error.

The diagnostics program causes a swap back to the user program by calling the environ function, described in chapter 4, with a parameter of 8. Such a call with this parameter is ignored by the interpreter if the mode flag is unset.

The user suffers from no overheads arising from the presence of the diagnostics code and data areas when the user's program is running. This is because the GEC 4080 is a virtual segment machine and no penalty arises from segments' existence except when accessed. The diagnostics data area is only present when needed and the code area, being unaltered by execution, is shared by all users just as the compiler Pcode is.

The diagnostics program is always invoked from its start. All information concerning the state of the user program is deduced from the control file created by the interpreter.

6.32 Monitoring actions of the interpreter

The second major change to the interpreter is the inclusion of some basic monitoring facilities. These facilities fall into two categories. The first is the maintenance of a count of executions of each source line in order that the diagnostics program can produce a profile. The second is to detect conditions for calling the diagnostics program. Both these functions are performed when the interpreter meets the line instruction described earlier. The word following each line instruction is reserved for holding the profile count. This is incremented each time the line instruction is obeyed. The line instruction is alterable by the diagnostics program to provide a break point. Additionally, there exists a facility whereby the user may manually interupt the program. This is provided by the GEC 4080 operating system, in the form of sending a message. The user types:

```
? A SEND USER
```

Optionally, a line of text may be appended. The interpreter can check whether a message has been sent. The above command is started by the sequence 'question-mark A' which is regarded as an attention symbol. The line code has the following format:

OP	P	Q	
62	n	line number	line instruction
O			profile count

The three fields OP, P and Q are as described in chapter 4. The value of the P field n is used to flag a break point. The Q field contains the source line number. The interpreter action for the line instruction is as follows:

```
    increment the profile count;
    increment the program counter;   {to skip over the count}
    if    n ≠ 0  then
                 begin
    set  P field = n - 1;
    if   P field = o then  call diagnostics
    end;
if message received then call diagnostics;
```

The diagnostics package then can set the P field of an instruction to one and a break point is enabled. By setting the P field to a value greater than one the option is provided to break after a specified number of passes through a break point. It would be possible for a user interrupt to be neglected. Consider the following source line:

```
x : = x + 1; while a < x do a : = a + b;
```

The line code is only produced prior to the assignment to x. The while loop will not contain a line instruction and then indefinite repetition would not be interrupted. To avoid this, a check for receipt of a message is made upon meeting any of the three Pcode jump instructions. These instructions do not ocarwithin a basic statement but at the end of a statement and so immediate entry to the diagnostics program can be made.

6.33 Privilege of the diagnostics program

In certain circumstances the interpreter will recognise the execution of the diagnostics program as a special case. For example, on detection of a run-time error this would be considered serious if the diagnostics program was running. In particular, one error condition is ignored if the diagnostic program is running. This is the condition of attempted reading of a non-existant file. The diagnostics program asks the user to type in the name of the source program file as described in the next section. If this is incorrectly typed then the error of trying to read a non-existant file is suppressed. A flag is set to indicate this event. This flag may be sampled by the diagnostics package by calling the environ function with a parameter of 9. The other operation considered privileged is the invocation of the procedures and functions peeks, peekc, pokes and pokec described earlier.

6.34 Generation of names for heap objects

The interpreter generates an integer for each object on the heap - created by the standard procedure new. The integer is used to identify an otherwise unnamed object within the diagnostics program. When space is allocated on the heap, one extra word is provided preceeding the object created. This word is given a value that is unique to the object. The interpreter allocates consecutive integers to these objects starting at one. The use of this value is described in more detail in the following section.

6.4 Implementation details of the diagnostics program

The facilities to be provided by the diagnostics system were defined in chapter 3. This section describes the implementation details of the diagnostics program providing these functions. The diagnostics program is initially provided with three sources of data. Firstly the interpreter creates a control file containing information concerning the state of the user program and why it was suspended. Secondly the compiler table file contains a representation of the name table for each procedure in the program. Finally the user program's code and data areas can be examined. Diagrammatically, the system initially appears as shown:

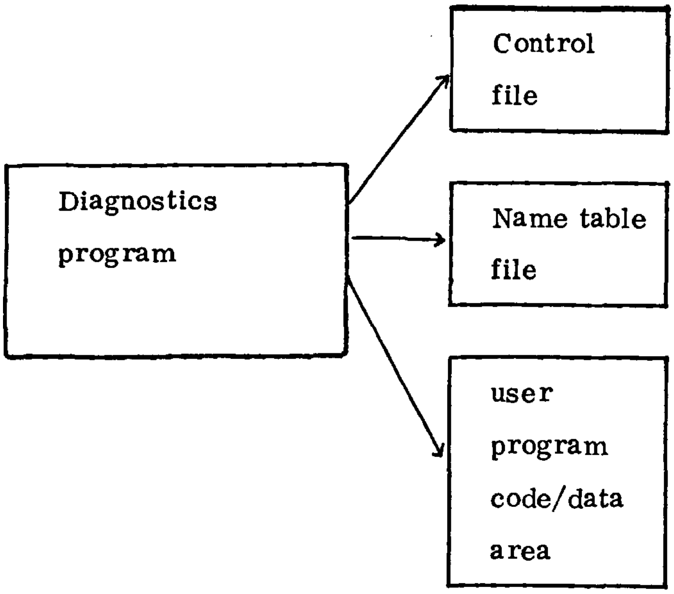


fig 6.5

Upon entry to the diagnostics program, the control file is first read. Using the information in the control file, the relevant sections of the name table file are read and the compiler name table reconstructed to the state it was in when compiling the statement just executed by the user program. After this process, these two files are no longer needed in this invocation of the diagnostics program. The other file that can be required is the source program. Then, when initialised and ready to accept commands from the user, the diagnostics system appears as follows:

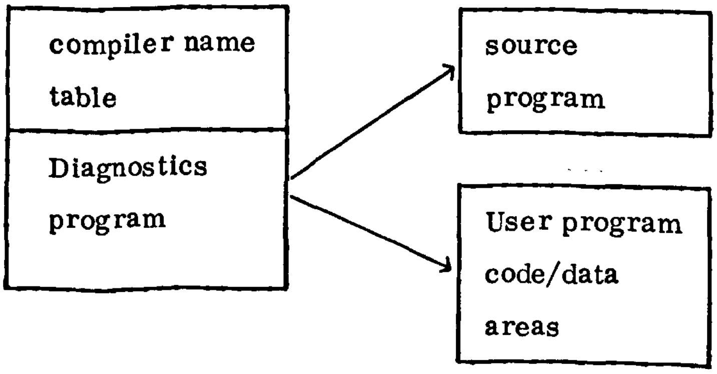


fig 6.6

The details concerning implementation of the particular features described in chapter 3 now follow.

6.41 Reconstruction of the name table

The structure of the name table file was described earlier in this chapter by figures 6.2 and 6.3. The diagnostics program must select the sections of this file that are relevant to the position within the program when its execution was suspended. This allows access to all variables that are in scope at that location. The name table file is constructed from objects of type identifier and structure. The task of recreating the name table is that of linking the pointers within each object. Each object contains pointers whose ordinal values are the Pcode store locations of the position of the pointed at object when the compiler was running. In addition, each object in the file is marked with a field specifying the location in Pcode store it occupied. This value can be used to cross reference the location each object will occupy within the diagnostics store area with that position it occupied within the compiler's store area. For example, if an object of type identifier resided in store at location 1000 when the compiler was running, this value 1000 is included in the name table file. Pointers in other such objects which pointed at this object will exist within the file with ordinal value 1000. When the object is recreated in the diagnostics program store area, all pointers with ordinal value 1000 are then allowed to point at it.

Demanding that these two positions within the compiler and the diagnostics program should be the same is not only difficult to implement but,as the compiler contains many other objects on its heap and due to it possessing a larger constants area thus starting its heap in a lower position of store, would waste considerable heap space in the diagnostics program. The algorithm for reconstruction of a linked structure such as this name table is now given.

The name table file consists of several blocks, each headed by a block record as shown in figure 6.3. There is one block for each procedure including the main program and the predeclared items considered to be in an outer block surrounding the main program at lexical level zero. The file is scanned until the procedure most local to the current program location is found. This block is processed. Following this each block in turn is examined. If its lexical level is less than that of the block just processed then it is itself processed otherwise it is skipped over. This process is continued until the block of lexical level zero had been assimilated. As can be seen from figure 6.3 and 6.2 this algorithm ensures that all name table objects that existed in the compiler when compiling the statement where execution was suspended are now processed by the diagnostics program. The manner in which these objects are processed is described now. As each object is read, the field tagged to it identifies the location it occupied in the compiler's store area. This index is used in a cross reference array defined as shown:

```
var table : array [o..tablemax] of record
                                     used : Boolean;
                                     case what : tabletype of
                                          isSTP : (STPP : STP);
                                          isCTP : (CTPP : CTP)
                                     end;
```

The location occupied is processed before being used to index this array to maximise use of the array:

index := (firstlocation - location) DIV minsize

Location is the location occupied, firstlocation is the start of the heap in the compiler and minsize is the minimum of the sizes of the objects under consideration, in this case 'identifier s' and 'structures'. This reduction of the index used maximises the use of the cross reference table. When an identifier or structure is read, the index is calculated and the appropriate element of table selected. With this element, the flag'used'is set, the field 'what' assigned the appropriate value depending upon whether the object is an identifier type or a structure type and the relevant field STPP or CTPP assigned to point at that identifier or structure now in the diagnostics program store. When all such objects have been created, the table array is scanned sequentially. For each element of the table that is flagged as used, the identifier type or structure type pointed at by either CTPP or STPP respectively is examined. The fields within the identifier/structure which are pointers to other identifiers/structures are then filled in by applying their existing ordinal values, their locations in the compiler's store, in the above index calculation formula and selecting that element from the table array.

This method then reconstructs the linked name table structure to its shape when extant in the compiler. It is a way in which any linked structure may be recreated from a backing store dump albeit rather long winded. The table array used is 2000 elements long. As its existence is temporary it is allowed to occupy all available store maximising the number of objects that can be catered for.

It could be argued that by not requiring invocation of the diagnostics afresh but retaining such tables between invocations time would be saved. It should be pointed out, however, that only lexical levels zero and one can be guaranteed to exist at all locations and replacing higher level name tables between invocations would require a large overhead of reconstructing the retained portion of the table array or retaining the large space occupied by it. In practice, this process of constructing the name table is quick and presents little inconvenience to the user.

6.42 Setting a break point

A break point is set by the diagnostics program by altering the relevant line code instruction. As described earlier, if the Pfield of this instruction is non-zero it constitutes a break point. In this case, control will pass to the diagnostics program on the Pth occasion this instruction is encountered.

When the diagnostics package is entered, the user program's Pcode is scanned to find the first and last line instruction and the corresponding first and last line numbers. When the user specifies a line number where a break is to be made, these limits are used to check the validity of the line number and to calculate an initial estimate of the position of the corresponding line code. The user may specify how many passes should be made before a break is made, this number plus one is then placed in the line code P field. If no line code exists for a given line, because that line does not contain a complete statement, then the first line code following is used - this is usually the user's intention. If the line specified is out of range of the program then either the first or last valid line is used depending upon whether the line specified was too small or too large.

Only one break point may be in existence at a time. This restriction allows simpler management of the system and has remained only because in use there has been no demand for more than one break point in an interactive system.

6.43 Interrogating the source program

The user requires the facility to list selected portions of the source program to relate the position of an error, decide upon location of break points and to be reminded of the data declarations used. This is provided by listing a specified group of lines of the program which is available to the diagnostics program. There is no completely secure way of the diagnostics program knowing the name of a source program file. This could be embedded in the Pcode but the user may rename the source file after compilation. When the source program is required by the diagnostics program for the first time, the user is asked its name. Future requests for listing portions of the file will refer to this name. The user may respecify the source file should it have been entered incorrectly. This facility allows the selective listing of any file which may be useful if the program in question operates upon data files. As mentioned in section 6.33, the diagnostics program can check for files' existence and act accordingly.

6.44 Producing a program profile

The profile of a program run is maintained by the interpreter. For each source program line that contains a Pascal statement, a count is available of the number of times that statement has been executed. The profile is presented as a table of source line numbers and number of times executed. The user may specify a group of consecutive lines for which a profile is required. In order to display the profile on a VDU screen, the diagnostics program restricts the total number of lines displayed to twenty. This ensures that all information given appears on the VDU screen. When the number of lines requested is greater than twenty, the counts for each line are grouped into equal divisions. For example, if m source lines are to be profiled, the diagnostics program will display n lines where:

$$\text{scalefactor} = (m - 1) \text{ DIV } 20 + 1$$
$$n = m \text{ DIV scalefactor}$$

Each line displayed will include the count for scalefactor consecutive source program lines. In addition to this number, the option of displaying a histogram representation of the counts or of displaying the source line next to the count is provided.

6.45 Interrogation of data items

The interrogation of data items is provided in two ways. The name table that has been constructed may be interrogated to verify the type definition belonging to an identifier and variables' values can be displayed in a format dictated by the variables' type.

6.451 Direct interrogation of name table

Directly interrogating the name table, without reference to the user program's store, provides information on the meaning or type of program identifiers. For any identifier that exists in the name table, the diagnostics program can list all attributes attached to it. If the identifier is a variable, then its type description is listed as it would appear in the program text. This feature allows the user to verify such details as array subscript ranges and the format of record descriptions. All such descriptions are given in Pascal language syntax.

6.452 Interrogation of variables' values

For each variable identifier, the name table contains the associated store address. Knowledge of the store address and the variable's type permits its value to be presented. The value is always given in a format consistent with Pascal syntax. Where an identifier's type is defined as an enumerated list of constants then these constant names are used. Where an identifier is an array or record, then the complete array or record can be listed. Alternatively, the user may specify an index of an array or a record's field. This specification is permitted with the adherence to the Pascal syntax. The diagnostics program will accept the Pascal syntax definition of a variable {WIRT75} with the single restriction that array subscripts may be constants only. Then, assuming appropriate definition of the identifiers used, the following would be acceptable variables that can be listed:

```
x
a [ 3]
p ↑ . field1
b [ 'A', true, 10]  . C ↑ [ 9 ]
```

Listing arrays

To list a vector, the value of each element is printed out. For arrays of more than one dimension, each dimension is printed in turn. The Pascal definition of a two dimensional array, for example, is an array of arrays. Thus each element of such an array is a vector and is listed as such. There exists an option to the user to print out an array completely. Normally, arrays are not listed as large arrays would result in producing much, probably unwanted, information. For instance, if a record is listed and one of the fields is a large array, the listing of all the other fields would be swamped by the array listing. The user is provided with two commands for listing variable values. One does not list arrays, the other does.

Listing records

Where a record variable is specified, each field of the record is listed. As with the listing of array elements, this process is accomplished by calling the procedure producing the listing recursively for each field. If fields are themselves records, then these records are expanded. Where a record contains a variant part, the tagfield is examined and the appropriate variant selected. The user does however have the option of explicitly requesting the listing of fields belonging to another variant which is useful if the tagfield had been set correctly.

Listing of pointers

Pascal pointers point at objects in the heap - created by the standard procedure new. When this occurs, the interpreter reserves one extra word for each object created by new and inserts an integer in this word. The integer is incremented between calls of new such that each object on the heap has a unique value associated. The value of a pointer is given as the integer associated with the object preceded by a hash symbol ' # '. The two alternative values that can be given are nil and undefined. The latter is given if the pointer has either never been assigned a value or the space occupied by the object it pointed at has been reclaimed. The values given to pointers in the format # < integer > may be used as legal identifiers of the objects pointed at. In this way all data variables including those created dynamically on the heap may be identified by name and listed. This facility is unique to this diagnostic program. The association of these hash identifiers with the objects referred to is described in the next section. Examples of using this facility are given later.

The listing of variables is performed by a procedure within the diagnostics program. The format of this procedure is:

```
procedure  writevar  (address : integer; idtype : STP);
begin
      case    typekind of
      scalar, subrange : begin
                         write value at address in appropriate format.
                         If subrange, print warning message if this
                         value is out of range
                         end;
      pointer          : if nil then write ('nil') else if undefined then
                         write ('undefined') else write (' # ') followed
                         by associated integer.
      power            : begin
                         write (' [  '); write each set element present;
                         write ('  ] ')
                         end;
      arrays           : begin
                         writeln ('('); call writevar for each element;
                         write (')')
                         end;
      variant, records : begin increment indentation on each write;
                         if variant then write ('(') else write ('Record');
                         for each field up to but not including any tagfield
                         write the field name then call writevar for this
                         field.
                         if tagfield present then
                            begin
                            write tagfield and value. Select appropriate
                            variant and call writevar if this variant is legal.
                            end;
                         if variant then write (')') else write ('end')
                         end
      end
 end;  {writevar}
```

Interpretation of variables

The diagnostics program permits the selection of record fields and array elements when specifying a variable. The object pointed at by a pointer is signified by appending an up-arrow ↑ to the pointer specification. This allows specification of any data item belonging to the user program. The routines to interpret this specification are adapted from the routines used in the compiler for analysis of a variable. At those positions in this analysis where the compiler produces Pcode instructions to index the appropriate field or array element, the diagnostics program performs this action directly - simulating the Pcode machine by a small expression stack. During this analysis, syntax errors may be detected. The message printed informs the user of such an error and gives the location at which the error was found. Three semantic errors can occur which are detected and reported. Firstly a subrange variable may be out of range. The out of range value is printed and a warning message issued. If this subrange violation occurs when specifying an array index, an error message is given. Secondly, there may be no record variant catering for a particular value of a tagfield. This may have been intentional and a warning message is printed. Finally, the user may attempt to access the non-existant object pointed at by a pointer with an undefined value or with the value nil. Again, the appropriate error message is given.

Representation of types

The diagnostics program has access to the user program's store in the form of a vector of integers. Having located a variable, the integral representation of it will in general have to be manipulated if that variable is not an integer. The diagnostics program has to cater for the representation details of scalar values and sets. This section of the program is clearly machine dependent. Characters are stored four to a word and sets occupy four words. Use is made of the record variant facility in order to perform this transfer of representation. Consider the following record:

```
record
case typerequired of

int : (il, i2, i3, i4: integer);
rl  : (r : real);
ch  : (chl, ch2, ch3, ch4 : char);
B1  : (B1, B2, B3, B4 : Boolean);
st  : (s : set of 0..127)
end;
```

By placing an integer in il, the required character, Boolean or real variable can be selected as they are known to occupy the same store space within the record. By placing four integers in il to i4 the representation of a set as S is provided. It should be noted that this is an abuse of the record variant facility and some implementations may reject it at run-time. By performing this conversion, standard Pascal facilities can be used to print the values of such variables.

6.46 Displaying structures

In chapter three, the facility to display the linking of data structures graphically was described. This facility is considered an essential tool to programmers using data - structuring algorithms and yet does not exist in any other diagnostic systems. The process of displaying a structure is as follows. The user selects a record which is considered the root of the structure. Using this record, all pointers are followed building a two dimensional image of the relative positions each record occupies. The relative location of each record is defined by the ordering of declaration of each pointer in the declaration of that record type. Thus a position is reserved for each pointer of each record. These positions are taken symmetrically beneath each record with the earliest declared pointer taking the leftmost location and the last pointer declared in the record declaration holding the rightmost location. As mentioned in chapter three, the structure is first scanned to find the width of array necessary to hold it. Consider the following structure:

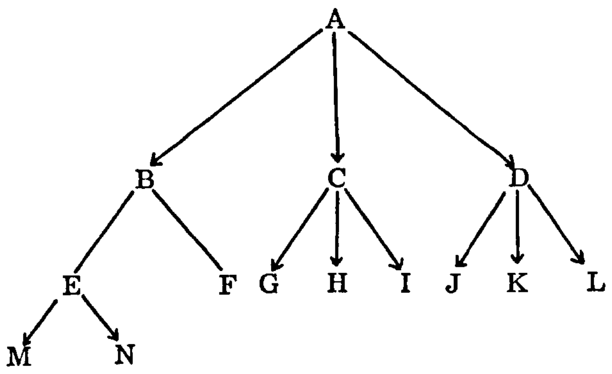


fig 6.7

Objects A, C and D have three pointers, objects B and E have two pointers and objects M N F G H I J K and L have their pointers set with value nil. The number of these nil pointers is not relevant to the display. The width of the display required is calculated as follows. Each object requires at least one unit of width. But if any of its pointers are not nil, it requires for each pointer the maximum width required of each object pointed at multiplied by the total number of pointers. For the purpose of displaying a symmetric structure, where the number of pointers contained in an object is even, it is taken to be one greater. This extra pointer is taken to be nil and displayed in the centre of the existing pointers. Then the width required by the structure is that required by A. Objects M N F G H I J K and L only require one unit each as all have no non-nil pointers. Object C requires the maximum of the widths required by objects G H and I multiplied by the number of pointers C contains. This is then three units. Similarly D requires three units. Object E contains two pointers. For the purpose of display, this is taken to be three and the width required by E is then three. E's width is the maximum width of the objects B points to. B contains two pointers and the width required is then (2+1) x 3 = 9. B requires the maximum width of the three objects pointed atby A's pointers. A contains three pointers and therefore requires a width of 27. A is then placed in the middle of this width at unit position fourteen. B C and D are allocated 9 units each. Each of these 9 units is split equally between the objects below producing the following layout:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
													A													
				B									C									D				
	E						F			G			H			I			J			K			L	
M		N																								

fig 6.8

This display is symmetric about A and as such is different to figure 6.7. but does not require the drawing of lines to show connections as these are deduceable from objects' relative positions. The implementation details of manipulating this matrix are now described.

6.461 Representation of the display array

If a display contains objects with n pointers and is d levels deep, the potential size of array required is:

$$d \times n^{d-1}$$

As d increases, this array will become extremely large and it may be difficult to represent it using simple techniques without running into difficulties of storage space availability. The array required is not rectangular, at each depth only n^{d-1} units width are required. The actual array that would suffice for n=3 and d=4 is:

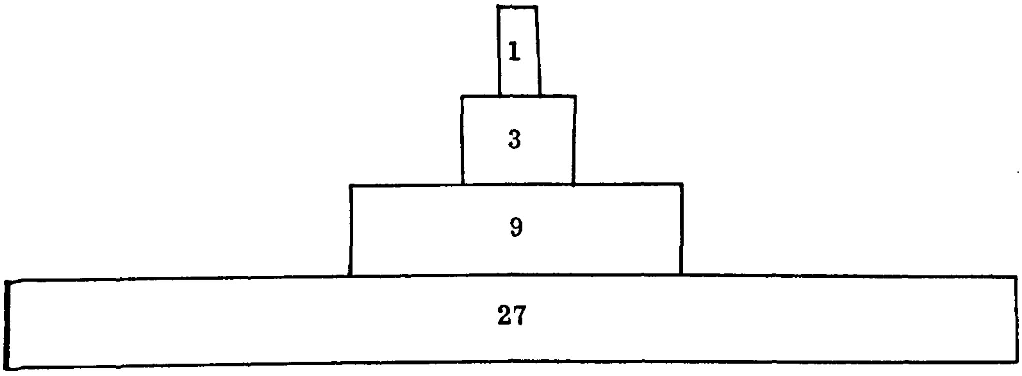


fig 6.9

That is, a size of $\sum_{r=1}^{d} n^r$ instead of $d \times n^{d-1}$ in this case, the size required is only 40 units rather than 108. In addition, not all these units hold an object; figure 6.8 holds only 14 objects. The array is then very sparse. The representation used requires storage space only for objects present. It consists of a vector representing the rows of the matrix. Each row is a linked list of the objects present in that row. The column within which the object lies is defined within the list elements. The array is then represented by the following data types:

```
type
        parec   = ↑arec

        arec    = record
                    ind, value : integer
                    next : parec
                    end;

        airec   = record
                    lowind, highind : integer;
                    first, last : parec
                    end;

var     aindex : array [1.. maxdepth] of airec;
```

The vector aindex contains a record for each row. The first object in that row is pointed at by 'first'. The record also holds a pointer to the last object in the row to assist insertion at the end of the list. The highest and lowest bounds of the row are also held to ease calculation of the array's width and to aid the decision of where to insert a new object. The objects are represented by type arec which holds the column number in 'ind', a pointer to the next object in 'next' and the integer value name given to the structure objects in 'value'. The display array holding the structure shown in fig 6.8, for example, appears as follows:

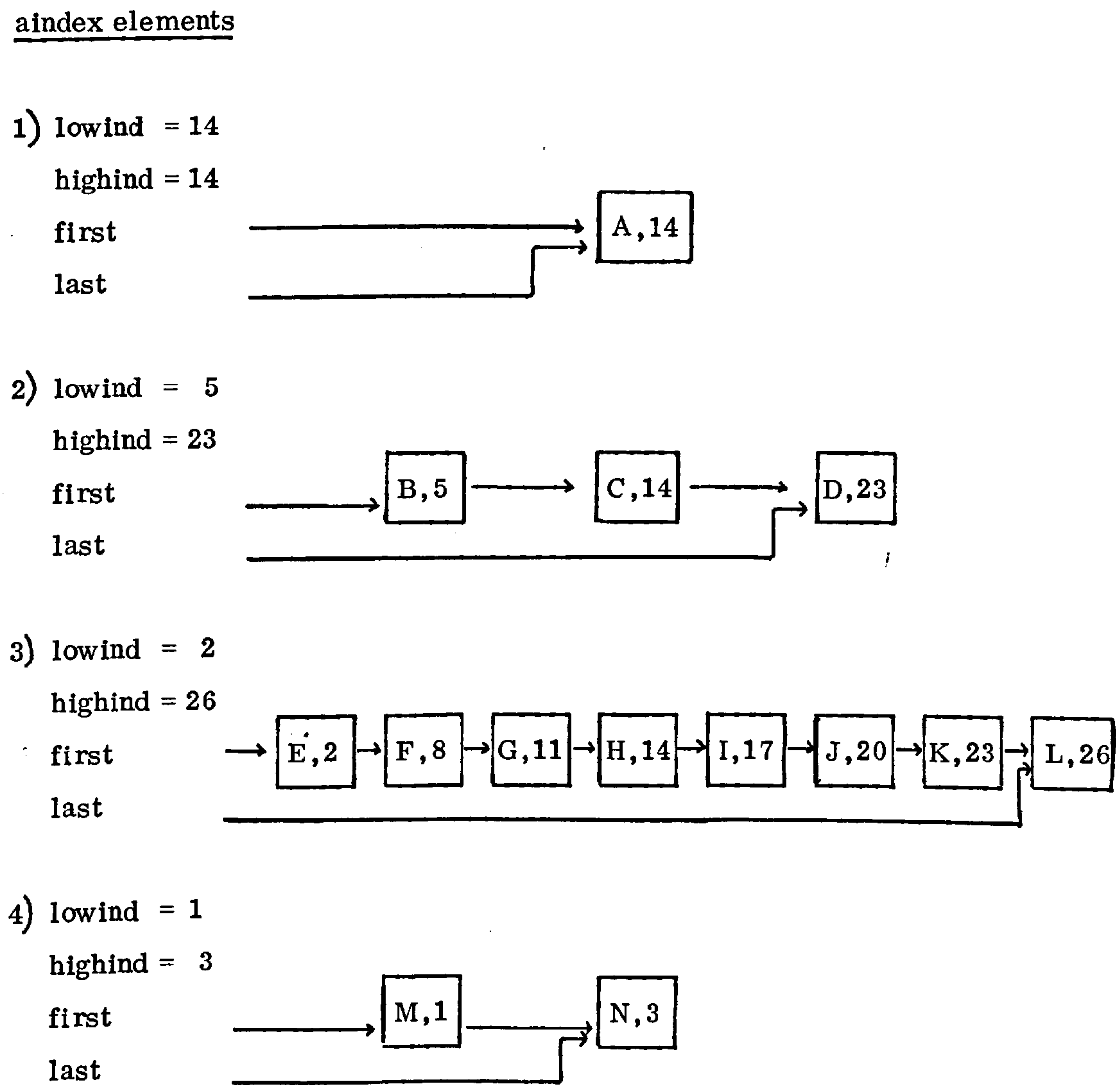


fig 6.10

This array is manipulated by two procedures which emulate a rectangular array. The space occupied by the objects of type arec is reclaimed when a new structure is selected for display. The maximum depth that can be catered for is currently set at a hundred. Assuming a maximum core size availability of 50K words, a structure of objects containing three pointers could consume this amount of store at a depth of eight levels. Unless the user's structure consists of objects of size less than the type arec and uses all store available the diagnostics program will be able to hold a representation of its display. The prime reason for allowing such a large depth is in the special case of list processing where often the number of pointers is one and the array size necessary can then be one column.

6.462 Detection of cycles

A cycle may appear in a structure where an object points at another object that has already been placed in the array. This is detected, and the object included in the array for the second time but with two differences. Firstly it is marked as being a repetition so that the user is informed of it when displayed; and secondly all its pointers are treated as though they were nil so that the cycle is stopped. The detection is enabled by noting the integer name of each object. A Pascal set of integers is used to detect whether that integer is already present. In practice, an array of sets is used as in this implementation the maximum size of a set is 128 members.

6.463 Undefined pointer values

Where a pointer has an undefined value, this is treated as pointing to a special object which has no pointers within it. In this way an object (displayed as a question mark) is displayed to inform the user of this error.

6.464 Display of the array

Having produced the representation of a user's structure, this matrix is displayed. It is assumed that the medium for display is a VDU screen. This is regarded as a matrix of characters 20 x 80 in dimension. In chapter three, the techniques for mapping the display array onto the VDU screen were described. There exist three possible cases. Firstly the array size is the same size or smaller than the VDU screen. In this case the correspondence is one to one and a character is printed to represent each object. Where a cycle exists, the character printed is an up-arrow; if an undefined pointer is present, a question mark is printed otherwise the digit 1 is printed, meaning one object. The second case is when the display array is larger than the VDU screen in either dimension. The array is then squashed to fit the screen. To squash the array, each character on the screen is taken to represent a group of objects in the array. The character printed represents the number of objects that are present in that group. The characters used are 1 to 9 and A to Z. Thus if an A is displayed, this would mean that 10 objects are present in that group. Should any of those objects represent an undefined pointer, then a question mark is printed instead unless any other object in that group represents a cycle in the structure when an up-arrow is displayed. Finally the third case exists where the display array is significantly smaller than the VDU screen. When this is true, more than one character is printed for each display array element. The characters printed are the objects' associated names in the form of # < integer > . The display array has to be smaller in width by a factor of

$$\text{trunc}\ (\log_{10} n) + 2$$

where n is the maximum integer in order to qualify for display in this manner. Where an object represents a cycle then an up-arrow is printed instead of the hash sign. By finding the corresponding hash integer in the display, the cycle is then found. Undefined pointer objects are displayed as question marks.

Windowing

Where the display array is large, the user will be given a representation of the overall shape of the structure. Having seen this, an area may be selected for more detailed examination. The user can specify a window of the display by four integers denoting the co-ordinates of the window's edges. This section of the display is then treated by the same process that printed the complete structure and more detail is then provided concerning the selected area. For example, a structure might be displayed as follows:

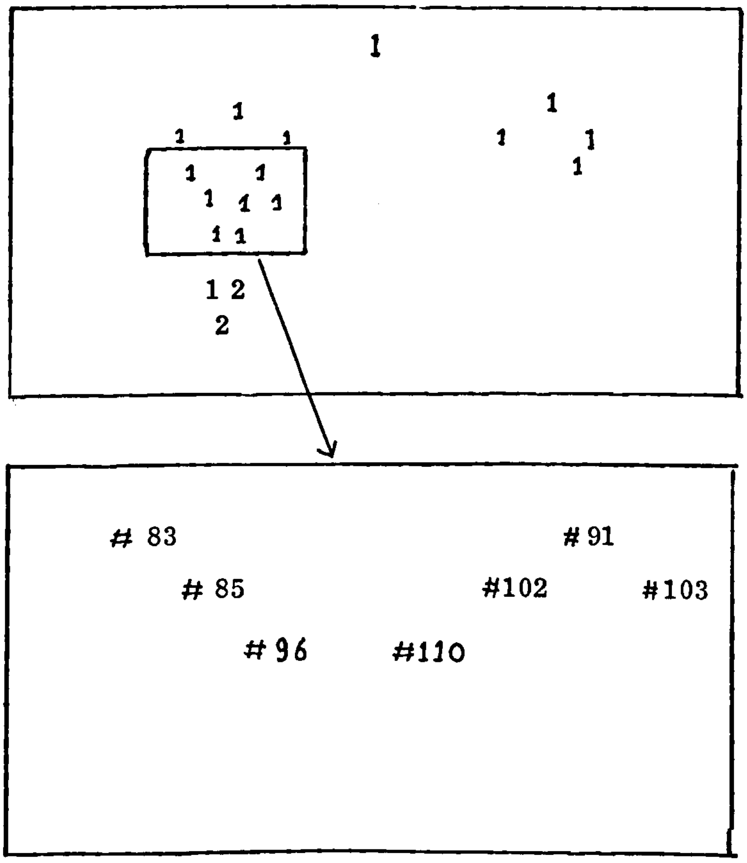


fig 6.11

The first display has been squashed and the shown window is selected. This window is then expanded and the names of the objects listed. The window can be moved over any part of the structure and can be defined as any size as it is moved. The effect produced is similar to that of blowing up part of a photograph in order to examine the finer detail except that resolution is not impaired by doing so. Examples of the use of this facility are given later in this chapter.

6.465 Hash integer implementation

The allocation of names as consecutive integers to objects created on the heap is performed by the interpreter. These names can be used within the diagnostics program in the same way as declared identifiers. In figure 6.11 the hash numbers may be referred to and the diagnostics program will then print out the values of the associated object's fields. As the integers are consecutive, they additionally inform the user of the objects' ages. The integer associated with each object on the heap is stored in the preceding word. The value of the pointer referring to this object is used to locate this identification. This access is purely in one direction. Referring to the identifying integer requires the construction of a cross reference table. Such a table could be maintained by the interpreter but this approach is rejected for two reasons. Maintaining a table is costly in space and time. A large table may have to be kept on backing store which would increase the time overhead considerably. The second reason is the complications that would arise when heap space is reclaimed. The table would have to be altered to account for this. The approach used is for the diagnostics program to maintain a cross reference table by inserting the locations of hash numbered objects into the table whenever a hash number is displayed to the user. Then whenever hash numbers are displayed in the production of a structure diagram such as in figure 6.11 or by the listing of a pointer, they are placed in this table. This approach seems reasonable on the grounds that the user cannot know of hash numbers existing until they are listed by the diagnostics program. The table kept holds the fifty most recently displayed unique hash numbers.

6.466 Variant records

Pascal records may contain variant sections where, depending on the value of a tagfield, alternative record fields can exist. For example:

```
     record
           i : integer;
     case tag : Boolean of
         true : (a, b : integer);
        false : (x, y : real)
     end
```

The record above consists of either the fields named i, tag, a, b, or the fields i, tag, x, y. These two variants' existence depends on the value of the field tag. The tagfield is alterable at run-time and so such records can be considered to have a dynamic type. When such a record is printed out by the diagnostics program, the tagfield value is used to select the appropriate variant fields to be printed. The variant fields of a record can pose a problem when displaying a structure if they can contain a pointer. The number of pointers associated with each record may then vary. In this situation it is difficult to decide how these variant pointers should be incorporated into a display. There are two possible approaches. The first is to include variant pointers where they exist. This is the same as regarding each variant as a completely separate type definition. The second approach is to ignore the existence of variant records. This is essentially implying that variant fields do not alter the type of the object. Both approaches can be justified and depend on the particular application for which the record is used. In the area of list processing, a list is commonly represented by a sequence of records each containing a head and a tail. The tail is always a pointer to another list while the head is either a pointer to another list or an item of data. In this application the two variants are regarded as different types. Another application where a variant may contain a pointer could be personnel records which under one variant a class of people have extra information about them recorded. If this information is both large and common to several people it can be considered better to keep one copy of it and include a pointer to it in the relevant personnel records. In this application, the pointer is used much as a key and a value such as an array index could have been used with similar effect. In this case the user probably does not consider such a pointer as contributing meaningful information about the overall shape of the data structure of personnel records. The structure emanating from these variant pointers is a substructure and including it in a display would be confusing; the programmer would view such a structure as part of a record rather than loosely connected to it.

The diagnostics program does not consider pointers which exist in variant fields for the purpose of displaying a structure. There are two reasons behind this decision. The first reason is simply the effort involved in detecting any abuse of the Pascal record variant. The variant tagfield may have been incorrectly set by the programmer and a value which can be regarded as a pointer used in the guise of an integer for example. Such an error, unless detected, would lead to dramatic consequences when manifested in the display. The compiler would have to be altered considerably in order to produce such run-time checks as would be necessary. Such an alteration is, of course, possible and this suggestion is made in the next chapter. In implementing a system such as this diagnostics package, several decisions have to be made in order to allow its construction in a finite time. The second reason behind this decision is that the user may select any such variant pointers explicitly thus obtaining their individual displays. This leaves the onus on the programmer to decide which substructures should be displayed but does give the option to display them albeit indirectly. Given that the programmer may select these structures for individual display it was considered acceptable to take this approach. Should such a variant be selected erroneously then the resulting erroneous display is localised to that selection and does not interfere with any display containing this erroneous record. This approach is not regarded as completely satisfactory. The further enhancement of the diagnostics package is discussed in the next chapter after describing users reaction to it.

6.47 Summary

This section has described the main implementation details of a Pascal program which provides run-time diagnostic facilities. The aim has been to allow the user full diagnostic information at the level of Pascal syntax without needing to know any details of implementation. The major contribution this program makes concerns the area of data structuring where the user may interrogate the values and shapes of dynamically created structures. This display of structures is performed by creating a two dimensional graphical representation of the selected structure. This representation can then be displayed on a VDU screen and the option is provided to examine the finer details of selected parts. The next section presents the interface existing between these diagnostics and the user, following this are several examples of the use of this diagnostics system.

6.5 The user interface

The main facilities provided by the diagnostics system have been described. This section relates the interface provided to the user - the environment in which the diagnostics program is used. The user communicates with the diagnostics program by a simple command language consisting of seventeen commands. These commands are now described.

6.51 The command language

The seventeen commands provided fall into seven categories. These categories and the associated commands are listed below and the diagnostics described in more detail.

Category		Commands
1	Control	BREAKAT GO END
2	Interrogation of data	EXPLAIN WRITE WRITEALL FORMAT
3	Environment enquiry	BREAKS STATUS
4	Program profile	PROFILE PROFILES

5	Source program listing	TYPE SOURCE
6	Structure display	CURRENT DISPLAY RESTORE
7	Help to use commands	HELP

6.52 Control

The three commands within this category describe the passage of control between the user's program and the diagnostics program. The command Breakat is followed by a source line number and optionally a pass through count. The break point is not implemented until control is passed back to the program by the command Go. Then, the relevant line code is altered. Only one break point is implemented. If a second call of Breakat is used, this overides the first and when the command Go is invoked, the most recent break point request is implemented. The commands Breakat and Go always print a message stating where the break point is set. Setting a break point at line zero is equivalent to there being no break point. The command End causes termination of the diagnostics and user programs and a return to the operating system command process. The command Go will not be allowed if the reason for entry to the diagnostics was a run-time error.

6.53 Data interrogation

The two means of interrogating data variables were described in the previous section. The command Explain requires one program identifier as a parameter. This identifier is sought in the name table and its attributes listed. The command Write is followed by the specification of a program variable in Pascal syntax format. The value, or values in the case of a structured variable, is then printed in a format consistent with its type. If Writeall is used, then additionally any arrays encountered are fully expanded printing the value of each element. The command format expects one or two integer parameters. These values are used as field width specifiers for subsequent listing of variables of type real. These commands will accept the hash number names given to objects on the heap.

6.54 Environment enquiry

The two commands in this category produce information on the state of the user program. The Breaks command informs the user of the location of the break point currently set and the pass through count if this is non zero. The Status command also produces this information on the break point. In addition, this command lists the line number of the source program where entry to the diagnostics program was made, the reason for entry to the diagnostics and in the case of entry due to a run-time error the error that was detected. The Status command also lists the amount of storage currently used by the user program and the amount it has remaining.

6.55 Program profile

The two commands Profile and Profiles produce a list of program line numbers and their execution counts.Following the execution count is a histogram display of the count if the command used was Profile or the source line if the command used was Profiles. Both commands may be optionally followed by a range of line numbers in the form n-m. This limits the profile to this range. If a range is not specified then it is to be taken as the range of source program lines that contain Pascal statements. If the number of lines in the profile is too large to display all of them on the VDU screen then they are grouped into equal sections so that no more than a VDU screen size is produced. When this occurs, the user can then specify the line range of interest in order to obtain detail of each individual line.

6.56 Source program listing

The command Type causes a listing of the user source program. The filename containing the source program is communicated to the diagnostics program by the command Source. If the command Source has not been used prior to the command Type (or Profiles) then the user is prompted to provide this name. This name provided remains in force until the command Source is used specifying a different file name. By allowing this generality the user may list any file such as program data files. The command Type may be optionally followed by a line number range in the same format used for the profile commands allowing listing of a section of the program.

6.57 Structure display

The command Current is followed by the specification of an object that is a Pascal record. This object is then considered the root of the structure display and additionally may be referred to by the name #C - the current object. The display matrix is built and displayed. The command Display causes display of the matrix within the current window setting. This command may be followed by up to four signed integers which move the window to the position given by these integers relative to the current window. The four integers specify the first and last column of the display and the first and last row of the display. They are in units of a tenth of the dimensions of the current window. Thus, to move the window to the right immediately adjacent to its current position, the command used would be:

```
DISPLAY 10 19
```

The first and last row positions are unchanged as they are not specified. To move the window to the left, adjacent to its current position but covering half as many columns, the command would be:

```
DISPLAY - 5 - 1
```

To display the area in the middle of the current window of half the width and height of the current window the command would be:

```
DISPLAY 3 7 3 7
```

Then the four integers are expressed in a co-ordinate frame that has the top left hand corner of the current window as its origin and the current window width and height are always 10 units in length. The window can be expanded, moved or contracted by suitable choice of parameters to the Display command. The diagnostics program does not allow moving the window outside the display matrix. Should this be attempted then the window is automatically truncated at the display matrix edge. Moving the window completely out of the display results in the window being restored to encompass the complete display. The command Restore will also cause restoration of display of the whole matrix.

6.58 Help to use the commands

Whenever the diagnostics program is entered, the user is reminded of the existence of the command Help. This command produces a list of all available commands with a brief explanation of their use. More detailed information about each command is given if the command Help is followed by that command name. The user has all the information necessary to use the diagnostics program on-line. This is considered much more desirable than expecting the user to have the relevant paper documentation at hand and provides a very user friendly environment in which to work.

6.59 Summary

The diagnostics program environment has been described. The user may enter this environment, find out what commands are available and what they do using the help facility, then list program source, variables, types and data structure shapes. The user may specify initial entry to the diagnostics program prior to the start of the user program's execution by including the parameter DIAGS in the operating system command invoking the program run. In this way the break point may be set at any given location. The commands provided are thought to provide a natural way for the user to test programs and diagnose errors. Examples of using this diagnostics system now follow.

6.6 Examples of using the diagnostics system

This section includes fifteen examples of the use of the diagnostics system. Each example is an extract of a dialogue. These examples are designed to show how the facilities described are presented to the user. Within the examples, lines which start with the symbol > were typed by the user. All other lines were produced by the diagnostics system. These examples were printed directly from sessions using the diagnostics system on the GEC 4080.

The first example shows the user, having specified initial entry to the diagnostics, setting a break point at the program end and then causing the program to start. The program returns control to the diagnostics system after writing the line 'DONE'. The program used in this example is listed in example nine.

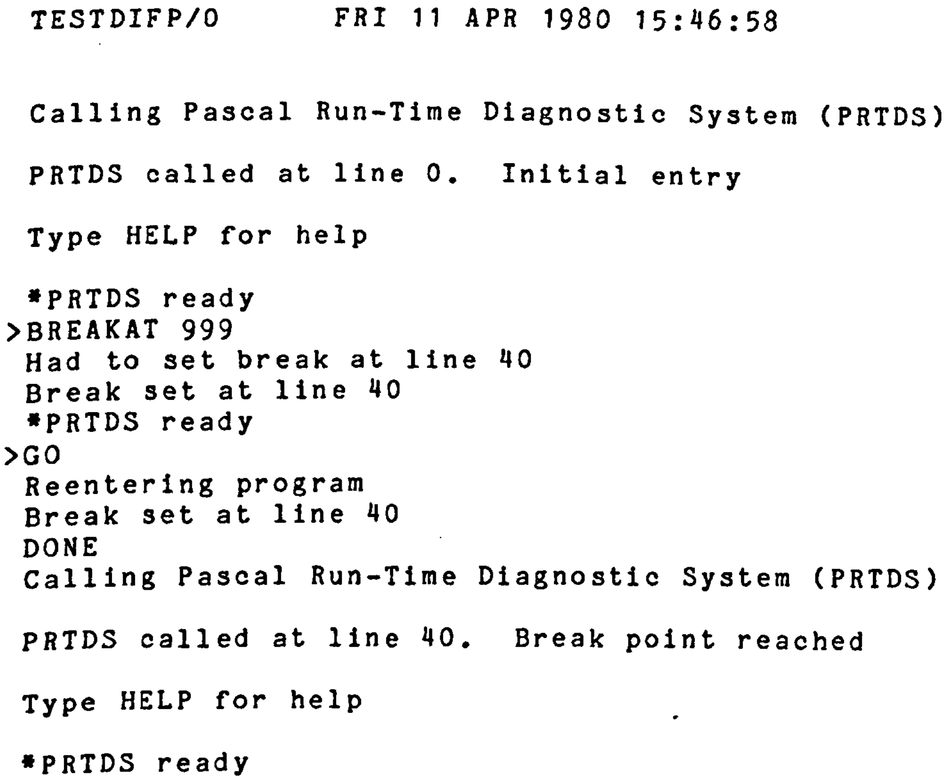
Example two shows the program used to produce examples three, four and five. It has been listed using the diagnostics system Type command. This program was written containing a variety of Pascal data types. Example three contains several instances of the Explain command which lists the attributes of program identifiers. The fourth example illustrates the Write command. The program was suspended by a break point set at the program end. Example five is a continuation of example four. This shows the Writeall command used to list arrays and a display of the simple structure created within the program. The structure contains one undefined pointer and a cycle. The cycle is seen at the bottom right of the display which points back to the structure's root. The undefined pointer is shown as two question marks.

Example six shows a listing of a program performing a sort of eleven integers. The program constructs an ordered binary tree which is traversed in an inorder fashion. Examples seven and eight were produced while this program was suspended by a break point set at its end. Example seven shows the selection of the ordered binary tree for display using the Current command. The display has been squashed and at one point two nodes of the tree are represented as one cell of the display. The display is then windowed by using the Display command. The window first selected specifies columns three to four of the original display. This window is then further reduced to its own columns four and five and rows three onwards. The resulting display is no longer squashed but is still too large for details of the nodes to be revealed. Example eight shows further reduction in the window resulting in the identification of four nodes of the tree. One of these nodes is then examined using the Write command.

Example nine shows a program that creates a structure containing nodes of two different types. One contains two pointers, the other contains three pointers. Example ten shows the display of this structure which is small enough to be shown in detail. The types of the component nodes are given by the diagnostics system in response to the Explain command.

The facility provided by the Help command is shown in examples eleven, twelve and thirteen. The Help command can be used to obtain information concerning all the commands and facilities of the diagnostics system.

The final two examples show the program profile created. The program used is shown as example fourteen. This program prints the moves required to solve the tower of Hanoi problem where a tower of disks is moved, one disk at a time, from one peg to another using an intermediate peg. The profiles are shown in example fifteen after completion of the program. The program, in this example, provided the solution for a tower of five disks.



```
 TESTDIFP/0      FRI 11 APR 1980 15:46:58


 Calling Pascal Run-Time Diagnostic System (PRTDS)

 PRTDS called at line 0.  Initial entry

 Type HELP for help

 *PRTDS ready
>BREAKAT 999
 Had to set break at line 40
 Break set at line 40
 *PRTDS ready
>GO
 Reentering program
 Break set at line 40
 DONE
 Calling Pascal Run-Time Diagnostic System (PRTDS)

 PRTDS called at line 40.  Break point reached

 Type HELP for help

 *PRTDS ready
```

Example 1

```
>TYPE
   1: PROGRAM TESTPRTDS(OUTPUT);
   2: TYPE
   3:  PREC=^REC;
   4:  REC = RECORD DATA:INTEGER; L,R:PREC END;
   5:  VREC = RECORD
   6:        A,B:INTEGER;
   7:        CASE X:BOOLEAN OF
   8:        TRUE:(Y,Z:REAL);
   9:        FALSE:( C:CHAR;
  10:                CASE BB:BOOLEAN OF
  11:                TRUE:(D:REAL);
  12:                FALSE:(CASE I:INTEGER OF 1:(J:INTEGER)))
  13:        END;
  14:
  15: DAYS = (MON,TUE,WED,THU,FRI,SAT,SUN);
  16:
  17: VAR VR:VREC;
  18:  P,Q,R,T:PREC;
  19: RA:ARRAY[5..20] OF REAL;
  20: IA:ARRAY[1..5,1..5] OF INTEGER;
  21: I,J:INTEGER;
  22: SD:SET OF DAYS;
  23: SC:SET OF CHAR;
  24:
  25: PROCEDURE DOREC(VAR P:PREC; LP,RP:PREC; D:INTEGER);
  26: BEGIN
  27: NEW(P);
  28: WITH P^ DO
  29:  BEGIN
  30:  DATA:=D; L:=LP; R:=RP
  31:  END
  32: END (*DOREC*);
  33:
  34: BEGIN
  35: FOR I:=5 TO 20 DO RA[I]:=1/I;
  36: FOR I:=1 TO 5 DO FOR J:=1 TO 5 DO IA[I,J]:=ROUND(I/J*10);
  37:
  38: DOREC(P,NIL,NIL,4);
  39: DOREC(R,NIL,NIL,5);
  40: DOREC(T,P,R,3);
  41: DOREC(P,NIL,T,1);
  42: NEW(Q);
  43: WITH Q^ DO BEGIN DATA:=2; R:=NIL END;
  44: R^.R:=P; P^.L:=Q;
  45:
  46: WITH VR DO BEGIN
  47:  A:=1; B:=2; X:=FALSE; C:='A'; BB:=FALSE; I:=1; J:=99 END;
  48:
  49: SD:=[TUE..THU,SUN];
  50: SC:=['A'..'H','!',' ','x'..'z','(',')'];
  51: END.
```

Example 2

```
>EXPLAIN VREC
 VREC     Type. Type definition is:-
  RECORD
  A : INTEGER
  B : INTEGER
  CASE X : BOOLEAN OF
  FALSE:(
       C : CHAR
       CASE BB : BOOLEAN OF
       FALSE:(
            CASE I : INTEGER OF
            1:(
                 J : INTEGER
                 )
            )
       TRUE:(
            D : REAL
            )
       )
  TRUE:(
       Y : REAL
       Z : REAL
       )
  END

 *PRTDS ready
>EXPLAIN REC
 REC      Type. Type definition is:-
  RECORD
  DATA : INTEGER
  L : PREC
  R : PREC
  END

 *PRTDS ready
>EXPLAIN DAYS
 DAYS     Type. Type definition is:-
                     (MON,TUE,WED,THU,FRI,SAT,SUN)
 *PRTDS ready
>EXPLAIN RA
 RA       Variable Type is:- ARRAY [5..20] OF REAL
 *PRTDS ready
>EXPLAIN IA
 IA       Variable Type is:-
                     ARRAY [1..5] OF ARRAY [1..5] OF INTEGER
```

Example 3

```
>WRITE VR
 Value =
 RECORD
        A           = 1
        B           = 2
        X           = FALSE     (CASE TAG FIELD)
        (
               C           = 'A'
               BB          = FALSE     (CASE TAG FIELD)
               (
                      I           = 1     (CASE TAG FIELD)
                      (
                             J           = 99
                      )
               )
        )
 END

 *PRTDS ready

>WRITE SC
 Value =
 [' ','!','(',')','A','B','C','D','E','F','G','H','x','y','z']
 *PRTDS ready

>WRITE SD
 Value = [TUE,WED,THU,SUN]
 *PRTDS ready

>WRITE P
 Value = pointer = #4
 *PRTDS ready

>WRITE #4
 Value =
 RECORD
        DATA        = 1
        L           = pointer = #5
        R           = pointer = #3
 END

 *PRTDS ready

>WRITE #4.L^
 Value =
 RECORD
        DATA        = 2
        L           = pointer = UNDEFINED
        R           = pointer = NIL
 END
```

Example 4

```
>WRITEALL IA
 Value =
 (
 (10,5,3,3,2),
 (20,10,7,5,4),
 (30,15,10,8,6),
 (40,20,13,10,8),
 (50,25,17,13,10))
 *PRTDS ready

>WRITEALL RA
 Value =
 (0.200,0.167,0.143,0.125,0.111,0.100,0.091,0.083,0.077,0.071,
  0.067,0.063,0.059,0.056,0.053,0.050)
 *PRTDS ready

>CURRENT P^
         0    1    2    3    4    5    6    7    8    9
  0                         #4                             0
  3       #5                                  #3           3
  6 ??                                  #1          #2     6
  9                                                   ^4   9
 Full display of structure
```

Example 5

```
>TYPE
   1:   PROGRAM TREESORT(OUTPUT);
   2:   TYPE NODE = RECORD DATA:INTEGER;
   3:                   LEFT,RIGHT:^NODE
   4:           END;
   5:        NODEPTR = ^NODE;
   6:   VAR
   7:   I:0..10;
   8:   TREE:NODEPTR;
   9:   NUMBER:INTEGER;
  10:   DATA:ARRAY[0..10] OF INTEGER;
  11:
  12:   PROCEDURE GROW(NUMBER:INTEGER; VAR TREE:NODEPTR);
  13:   BEGIN
  14:   IF TREE=NIL
  15:   THEN BEGIN
  16:   NEW(TREE);
  17:   WITH TREE^ DO
  18:   BEGIN
  19:           DATA:=NUMBER;
  20:           LEFT:=NIL;
  21:           RIGHT:=NIL;
  22:   END END
  23:   ELSE IF NUMBER < TREE^.DATA
  24:   THEN GROW(NUMBER,TREE^.LEFT)
  25:   ELSE GROW(NUMBER,TREE^.RIGHT)
  26:   END; (* GROW *)
  27:
  28:   PROCEDURE STRIP(VAR TREE:NODEPTR);
  29:   BEGIN
  30:   IF TREE<>NIL THEN BEGIN
  31:   STRIP(TREE^.LEFT);
  32:   WRITELN(TREE^.DATA);
  33:   STRIP(TREE^.RIGHT);
  34:   END
  35:   END; (* STRIP *)
  36:   BEGIN
  37:   TREE:=NIL;
  38:   DATA[0]:=-1;  DATA[1]:=2;  DATA[2]:=-3;
  39:   DATA[3]:=999;  DATA[4]:=4;  DATA[5]:=0;
  40:   DATA[6]:=27;  DATA[7]:=-1;  DATA[8]:=92;
  41:   DATA[9]:=33;  DATA[10]:=-4;
  42:
  43:   FOR I:=0 TO 10 DO
  44:   BEGIN
  45:   WRITELN(DATA[I]);
  46:   GROW(DATA[I],TREE)
  47:   END;
  48:   WRITELN;
  49:   STRIP(TREE)
  50:   END.
```

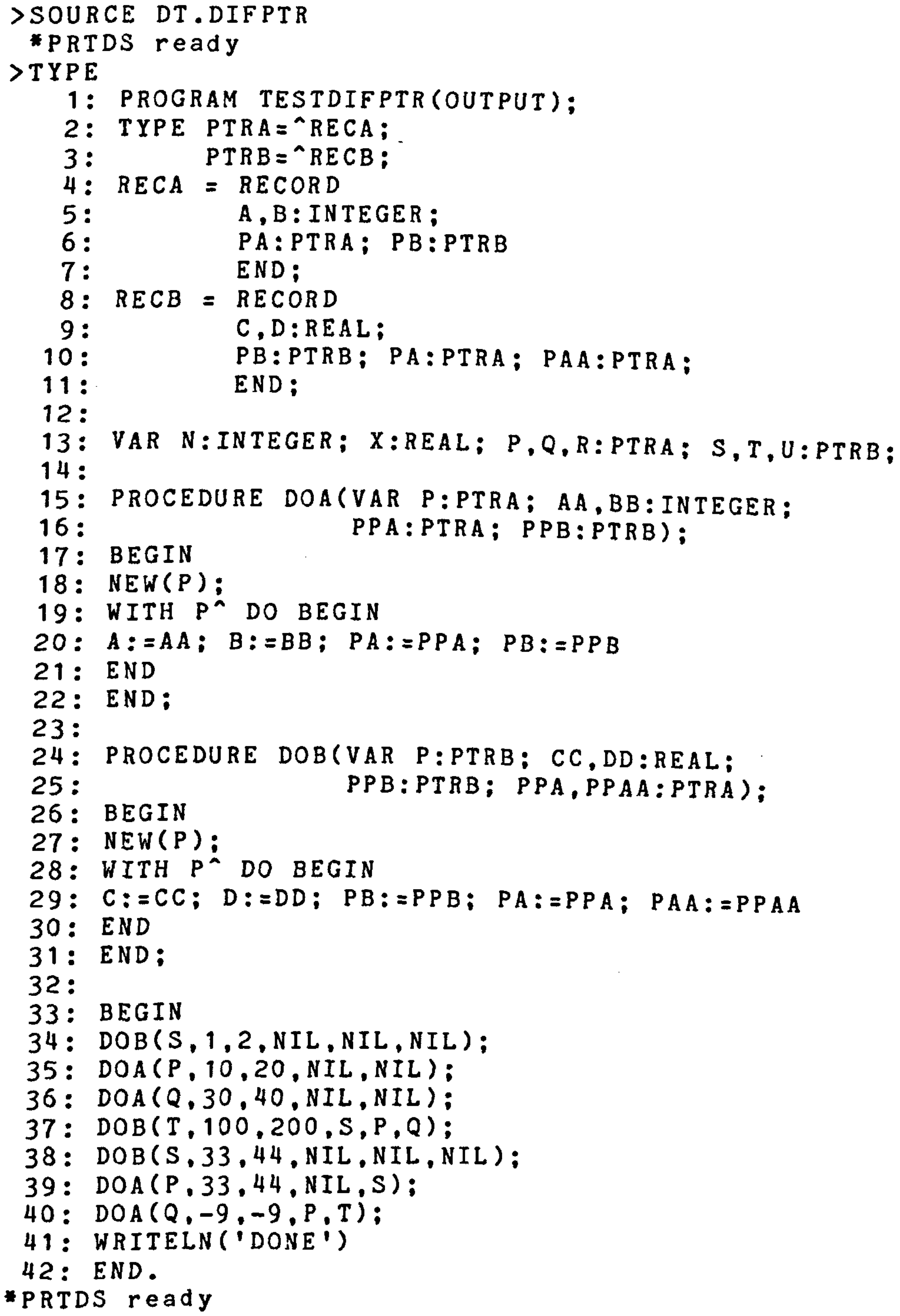
Example 6

Example 7

```
>CURRENT TREE^
   000000011111112222222333333344444445555555666666677777777888888899999999
 0  1                                                                       0
 1                                                  1                       1
 2                                  1                               1       2
 3                             1         1                                1 3
 4                              1          1                                4
 5                               1         11                               5
 6                               2          1                               6
 7                               1                                          7
 Structure squashed by a factor of 15 by width, and 1 by height. (expan 0)
 Full display of structure
 *PRTDS ready

>DISPLAY 3 4
    00000000111111112222222233333333444444445555555566666666777777778888888
 0                                                                          0
 1                                                                          1
 2                                                          1               2
 3                               1                                          3
 4                                        1                                 4
 5                                           1                              5
 6                                          1 1                             6
 7                                          1                               7
 Structure squashed by a factor of 3 by width, and 1 by height. (expan 0)
 Window of full structure (on a scale of 1..100) is :-
 By column 31..51            By row 1..100
 *PRTDS ready

>DISPLAY 4 5 3
    0000011111222223333344444555556666677777888
 0                                              0
 1                           1                  1
 2                                    1         2
 3                                 1     1      3
 4                                  1           4
 Structure squashed by a factor of 1 by width, and 1 by height. (expan 1)
 Window of full structure (on a scale of 1..100) is :-
 By column 39..43            By row 43..100
```

Example 8

```
>DISPLAY 5
    0001112223334445556667778
 0                            0
 1   1                        1
 2            1               2
 3         1     1            3
 4          1                 4
 Structure squashed by a factor of 1 by width, and 1 by height. (expan 2)
 Window of full structure (on a scale of 1..100) is :-
 By column 42..44            By row 43..100
 *PRTDS ready

>DISPLAY 2 4
          0      1      2      3      4      5      6      7      8      9
  0                                                                           0
  2                                                                           2
  4                                     #10                                   4
  6            #11                                          #13               6
  9                    #16                                                    9
 Window of full structure (on a scale of 1..100) is :-
 By column 42..43            By row 43..100
 *PRTDS ready

>WRITE #10
 Value =
 RECORD
     DATA     = 'A'
     LEFT     = pointer = #11
     RIGHT    = pointer = #13
 END
```

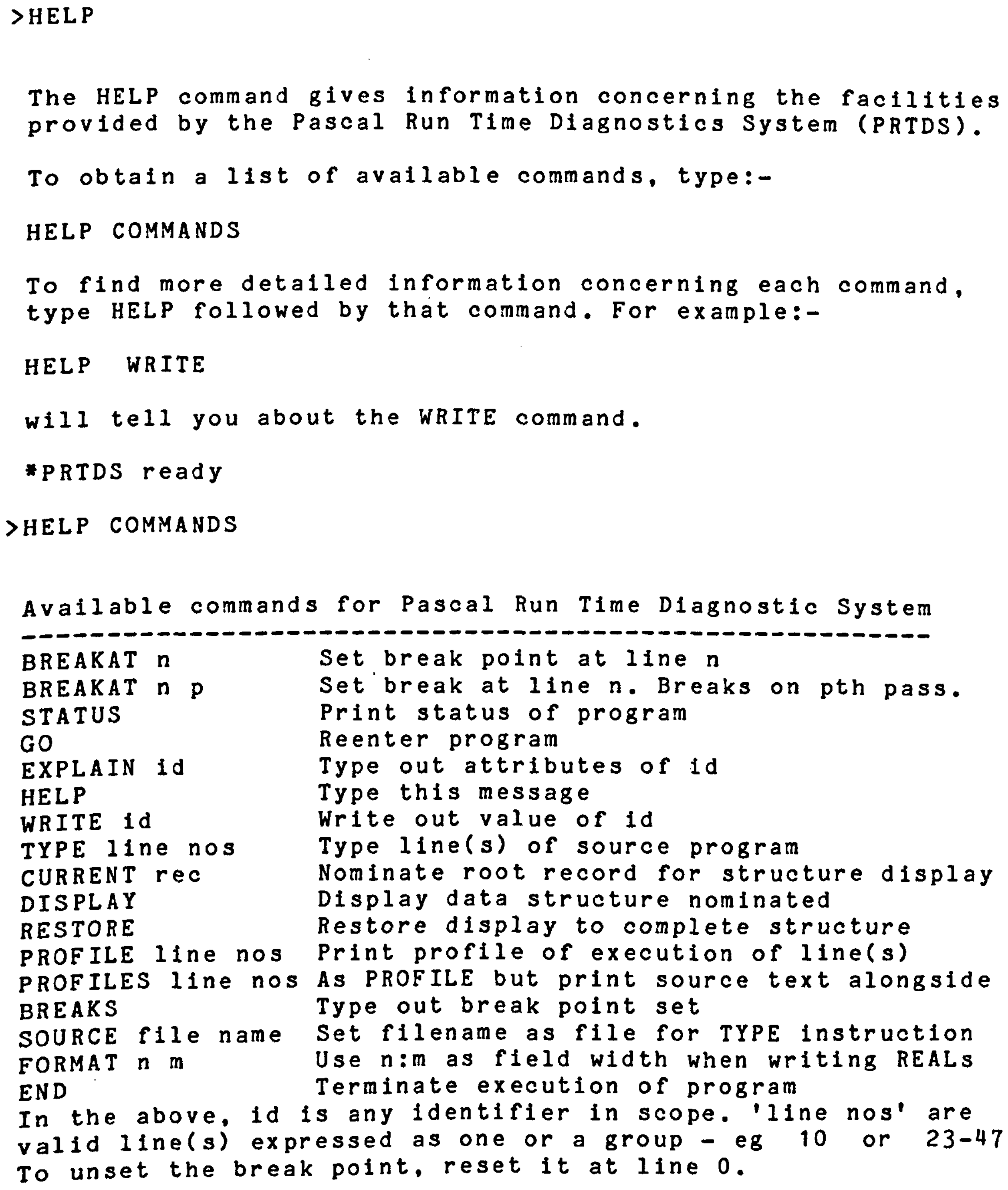


```
>SOURCE DT.DIFPTR
 *PRTDS ready
>TYPE
   1: PROGRAM TESTDIFPTR(OUTPUT);
   2: TYPE PTRA=^RECA;
   3:      PTRB=^RECB;
   4: RECA = RECORD
   5:        A,B:INTEGER;
   6:        PA:PTRA; PB:PTRB
   7:        END;
   8: RECB = RECORD
   9:        C,D:REAL;
  10:        PB:PTRB; PA:PTRA; PAA:PTRA;
  11:        END;
  12:
  13: VAR N:INTEGER; X:REAL; P,Q,R:PTRA; S,T,U:PTRB;
  14:
  15: PROCEDURE DOA(VAR P:PTRA; AA,BB:INTEGER;
  16:               PPA:PTRA; PPB:PTRB);
  17: BEGIN
  18: NEW(P);
  19: WITH P^ DO BEGIN
  20: A:=AA; B:=BB; PA:=PPA; PB:=PPB
  21: END
  22: END;
  23:
  24: PROCEDURE DOB(VAR P:PTRB; CC,DD:REAL;
  25:               PPB:PTRB; PPA,PPAA:PTRA);
  26: BEGIN
  27: NEW(P);
  28: WITH P^ DO BEGIN
  29: C:=CC; D:=DD; PB:=PPB; PA:=PPA; PAA:=PPAA
  30: END
  31: END;
  32:
  33: BEGIN
  34: DOB(S,1,2,NIL,NIL,NIL);
  35: DOA(P,10,20,NIL,NIL);
  36: DOA(Q,30,40,NIL,NIL);
  37: DOB(T,100,200,S,P,Q);
  38: DOB(S,33,44,NIL,NIL,NIL);
  39: DOA(P,33,44,NIL,S);
  40: DOA(Q,-9,-9,P,T);
  41: WRITELN('DONE')
  42: END.
*PRTDS ready
```

Example 9

Example 10

```
>CURRENT Q^
          0      1      2      3      4      5      6      7      8      9
  0                                #7                                         0
  4     #6                                                    #4              4
  9              #5                                  #1       #2       #3     9
 Full display of structure
 *PRTDS ready

>EXPLAIN PTRA
 PTRA     Type. Type definition is:- ^RECA
 *PRTDS ready

>EXPLAIN RECA
 RECA     Type. Type definition is:-
  RECORD
  A : INTEGER
  B : INTEGER
  PA : PTRA
  PB : PTRB
  END

 *PRTDS ready

>EXPLAIN PTRB
 PTRB     Type. Type definition is:- ^RECB
 *PRTDS ready

>EXPLAIN RECB
 RECB     Type. Type definition is:-
  RECORD
  C : REAL
  D : REAL
  PB : PTRB
  PA : PTRA
  PAA : PTRA
  END
```

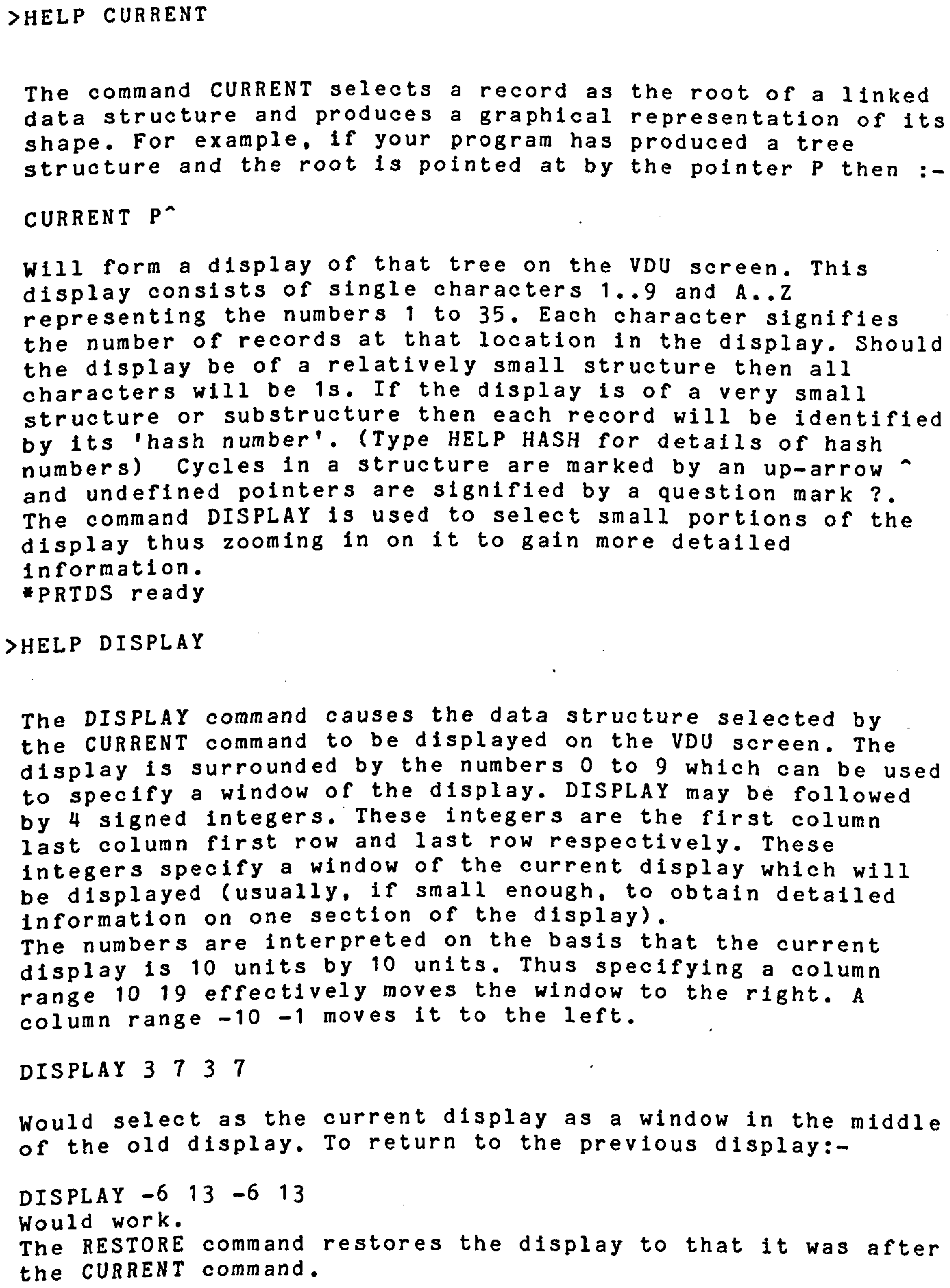


```
>HELP


 The HELP command gives information concerning the facilities
 provided by the Pascal Run Time Diagnostics System (PRTDS).

 To obtain a list of available commands, type:-

 HELP COMMANDS

 To find more detailed information concerning each command,
 type HELP followed by that command. For example:-

 HELP  WRITE

 will tell you about the WRITE command.

 *PRTDS ready

>HELP COMMANDS


 Available commands for Pascal Run Time Diagnostic System
 ----------------------------------------------------------
 BREAKAT n         Set break point at line n
 BREAKAT n p       Set break at line n. Breaks on pth pass.
 STATUS            Print status of program
 GO                Reenter program
 EXPLAIN id        Type out attributes of id
 HELP              Type this message
 WRITE id          Write out value of id
 TYPE line nos     Type line(s) of source program
 CURRENT rec       Nominate root record for structure display
 DISPLAY           Display data structure nominated
 RESTORE           Restore display to complete structure
 PROFILE line nos  Print profile of execution of line(s)
 PROFILES line nos As PROFILE but print source text alongside
 BREAKS            Type out break point set
 SOURCE file name  Set filename as file for TYPE instruction
 FORMAT n m        Use n:m as field width when writing REALs
 END               Terminate execution of program
 In the above, id is any identifier in scope. 'line nos' are
 valid line(s) expressed as one or a group - eg  10  or  23-47
 To unset the break point, reset it at line 0.
```

Example 11

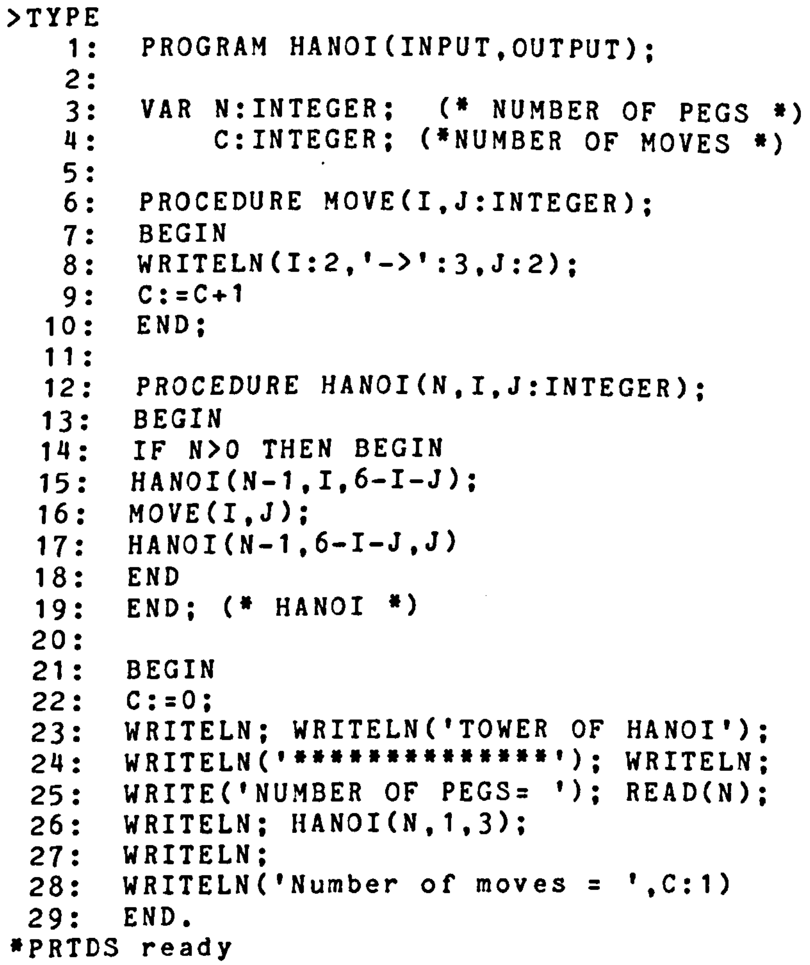
```
>HELP WRITE


 The command WRITE expects a program variable following it.
 The program variable may be expressed in any legal Pascal
 syntax but if array subscripts are provided they must be
 constants. If the variable is of type array, then no listing
 of it is given in case the array is very large. Instead the
 command WRITEALL should be used. WRITEALL is identical to
 WRITE except if the variable is an array or a record
 containing an array all elements of that array will be listed.
 A variable created by the Pascal procedure NEW can be listed
 by giving its 'hash number'. (Type HELP HASH for details about
 hash numbers)

 For example:-

 WRITE A[4,6]
 WRITE G
 WRITE FRED^.LEFT^.FIRSTFIELD[10]
 WRITE  #45

 Are legal commands assuming the given variables are of the
 appropriate type.
 *PRTDS ready

>HELP HASH


 Variables created by the Pascal procedure NEW do not have
 identifying names in the text of the Pascal program. In order
 for you to refer to them each such variable is assigned a
 'hash number' upon creation. This number is unique to each
 variable and is specified as a hash sign # followed by the
 assigned integer.  When you request the value of a pointer
 with the WRITE command the hash number of the object pointed
 at will be given (if it exists). The display of a data
 structure given by the commands CURRENT and DISPLAY can also
 contain hash numbers.
 The hash number may be used as any other identifier with the
 commands WRITE WRITEALL EXPLAIN and CURRENT.
 The record selectecd by the CURRENT command can additionally
 be refered to as  #C.

 The hash numbers are assigned in order starting at 1. Thus
 the relative age of such variables is known by comparing
 their hash numbers.
```

Example 12



```
>HELP CURRENT


 The command CURRENT selects a record as the root of a linked
 data structure and produces a graphical representation of its
 shape. For example, if your program has produced a tree
 structure and the root is pointed at by the pointer P then :-

 CURRENT P^

 Will form a display of that tree on the VDU screen. This
 display consists of single characters 1..9 and A..Z
 representing the numbers 1 to 35. Each character signifies
 the number of records at that location in the display. Should
 the display be of a relatively small structure then all
 characters will be 1s. If the display is of a very small
 structure or substructure then each record will be identified
 by its 'hash number'. (Type HELP HASH for details of hash
 numbers)  Cycles in a structure are marked by an up-arrow ^
 and undefined pointers are signified by a question mark ?.
 The command DISPLAY is used to select small portions of the
 display thus zooming in on it to gain more detailed
 information.
 *PRTDS ready

>HELP DISPLAY


 The DISPLAY command causes the data structure selected by
 the CURRENT command to be displayed on the VDU screen. The
 display is surrounded by the numbers 0 to 9 which can be used
 to specify a window of the display. DISPLAY may be followed
 by 4 signed integers. These integers are the first column
 last column first row and last row respectively. These
 integers specify a window of the current display which will
 be displayed (usually, if small enough, to obtain detailed
 information on one section of the display).
 The numbers are interpreted on the basis that the current
 display is 10 units by 10 units. Thus specifying a column
 range 10 19 effectively moves the window to the right. A
 column range -10 -1 moves it to the left.

 DISPLAY 3 7 3 7

 Would select as the current display as a window in the middle
 of the old display. To return to the previous display:-

 DISPLAY -6 13 -6 13
 Would work.
 The RESTORE command restores the display to that it was after
 the CURRENT command.
```

Example 13



```
>TYPE
   1:  PROGRAM HANOI(INPUT,OUTPUT);
   2:
   3:  VAR N:INTEGER;  (* NUMBER OF PEGS *)
   4:      C:INTEGER; (*NUMBER OF MOVES *)
   5:
   6:  PROCEDURE MOVE(I,J:INTEGER);
   7:  BEGIN
   8:  WRITELN(I:2,'->':3,J:2);
   9:  C:=C+1
  10:  END;
  11:
  12:  PROCEDURE HANOI(N,I,J:INTEGER);
  13:  BEGIN
  14:  IF N>0 THEN BEGIN
  15:  HANOI(N-1,I,6-I-J);
  16:  MOVE(I,J);
  17:  HANOI(N-1,6-I-J,J)
  18:  END
  19:  END; (* HANOI *)
  20:
  21:  BEGIN
  22:  C:=0;
  23:  WRITELN; WRITELN('TOWER OF HANOI');
  24:  WRITELN('**************'); WRITELN;
  25:  WRITE('NUMBER OF PEGS= '); READ(N);
  26:  WRITELN; HANOI(N,1,3);
  27:  WRITELN;
  28:  WRITELN('Number of moves = ',C:1)
  29:  END.
*PRTDS ready
```

Example 14

```
>PROFILE
 Line. Passes

   8:   62 *********************************
  10:   31 ****************
  12:    0
  14:   94 **************************************************
  16:   62 **********************************
  18:   94 **************************************************
  20:    0
  22:    2 *
  24:    2 *
  26:    2 *
  28:    2 *
 *PRTDS ready

>PROFILES 8-18
 Line. Passes

   8:   31  WRITELN(I:2,'->':3,J:2);
   9:   31  C:=C+1
  10:   31  END;
  11:    0
  12:    0  PROCEDURE HANOI(N,I,J:INTEGER);
  13:    0  BEGIN
  14:   63  IF N>0 THEN BEGIN
  15:   31  HANOI(N-1,I,6-I-J);
  16:   31  MOVE(I,J);
  17:   31  HANOI(N-1,6-I-J,J)
  18:   31  END
 *PRTDS ready

>PROFILE 8-18
 Line. Passes

   8:   31 *************************
   9:   31 *************************
  10:   31 *************************
  11:    0
  12:    0
  13:    0
  14:   63 **************************************************
  15:   31 *************************
  16:   31 *************************
  17:   31 *************************
  18:   31 *************************
 *PRTDS ready
```

Example 15

6.7 Summary

This chapter has described the implementation of a diagnostics system which was introduced in outline in chapter three. The system consists of three parts. First of all the compiler has been adapted so that it generates a representation of its name tables and includes certain codes in the Pcode object file for each source line containing Pascal statements. Secondly the interpreter performs monitoring tasks which comprise error detection, checking for break points and user interrupts all of which cause transfer of control to the diagnostics program and, whenever an object is allocated space on the heap the interpreter gives that object a unique name. The third part of the diagnostics system performs the majority of the work involved. This is the diagnostics program which when invoked acts as an interactive post-mortem program. This program may be invoked at the beginning of the first Pascal statement of any source line in the users program. The diagnostics program can then cause continuation of the user's program. The diagnostics program is written in Pascal. It is almost 2,500 lines long and compiles into just over 7,500 Pcode instructions. This diagnostics system has been used by computer science students at Keele University in both basic first year programming courses and final year projects. The final chapter contains a discussion of the user comment on this system, an appraisal of the diagnostics system, and an assessment of the work described in this thesis leading up to the implementation.

CHAPTER SEVEN

CONCLUSIONS

7.0 Introduction

This final chapter has three objectives. Firstly a review and summary of the thesis is provided. Secondly an appraisal of the diagnostics system is given in light of user reaction to it. Thirdly suggestions are made for improvement of the diagnostics and some implications that such improvements may have on programming language design and hardware features are discussed.

7.1 Review

The initial chapters of this thesis have been concerned with the problems involved with producing reliable software. It is still true that the majority of effort in commercial programming is due to maintenance of software. The United States defence department attributes 80% of its software costs to maintenance resulting in an annual maintenance cost of three billion dollars. The importance of producing reliable software has been stressed and the commonly used programming techniques and tools have been described. Such techniques include the structured use of high level languages which allow a more natural expression of a program than machine code. The ease of expressing programs' algorithms and data in high level languages has been discussed using the idea of a problem orientation attributed to such languages.

During this discussion, the problems that can arise when a program is not behaving correctly were stressed. In certain existing systems, the programmer is forced to be aquainted with the details of implementation of the high level language in order to trace an error diagnosed in terms of primitive machine functions. The argument develops to the conclusion that such implementation details are the concern of the system and not the programmer. Such a principle requires the system to communicate information concerning erroneous program behaviour in terms of the high level language. The mechanism provided to perform this task is seen as a diagnostics system which is able to map the compiled program back onto the source language. It is argued that this diagnostics system is of equal importance to the compiler. The diagnostics system is not necessarily invoked just to report errors but can be a vehicle allowing the programmer to monitor the program's behaviour.

Currently existing diagnostic systems were reviewed and useful facilities they provide described. This review enables those facilities which have been considered important to be isolated, of particular concern was the fact that existing diagnostics systems do not cater for all the facilities available in modern high level languages. Most notably, diagnostics systems are lacking in the area of data structures. The outline of a diagnostics system is described. The prime motivation is that the diagnostics system should cater for all the features of the high level language. There are two innovations in this system. The first is the provision of a mechanism enabling the programmer to interrogate the values of dynamically created data objects. The second is a mechanism for displaying the linking of such objects to form a data structure. A programmer views such data structures graphically as objects connected together in a certain manner. This structure is usually considered to be two dimensional but is implemented in a one dimensional fashion on the machine. The diagnostics system is capable of creating a two dimensional graphic display from its representation in store.

The language chosen for this diagnostics system was Pascal. Pascal was not available locally for interactive use and an implementation of the language was then necessary. For historic reasons two implementations of Pascal were undertaken. The first, on a Digico Micro 16E was later transferred to the GEC 4080 which is Keele University's main service machine. The P4 Pcode based Pascal compiler was used in both cases. The implementation details show the different approaches necessary for a relatively primitive machine such as the Digico and the more powerful GEC 4080. The portability problems encountered are described within the general framework of software portability. It is pointed out that portability is a term used to describe the effort involved in moving a piece of software but such a transfer should consider the end product's efficiency. In this way, portability covers both the physical implementation and its subsequent adjustment to operate efficiently in a new environment. The discussion on portability concludes with some specific recommendations for both software and hardware developments. Following this discussion is the design and implementation of the diagnostics system. The performance of the system and the reactions to it are now described.

7.2 User reaction to the diagnostics system

The diagnostics system on the GEC 4080 at Keele University has been used by staff and students for five months. The majority of these users belong to the Computer Science department as few other University departments currently use Pascal. The users' abilities vary from the absolute beginner to experienced programmers constructing large programs. Throughout this range of experience, the general comment is that the system is extremely useful and learning how to use its facilities takes very little time. The students learning how to program have not had much need to use the facilities for displaying structures and their comments are concerned with the more basic functions of the diagnostics system. Comments on the ability to display data structures have appeared from staff users and some final year students who have been concerned with their own particular projects. There has been little opportunity for comment from second year students as, due to syllabus revisions, they do not yet use Pascal for their programming exercises.

Comments on non-display facilities

All comments received concerning these facilities provided by the diagnostics system were good. Users were particularly pleased with the help facility which allows use of the diagnostic facilities without reference to written documentation. One person, whilst complimenting the facilities provided, did object to the length of the commands. The most commonly used command is WRITE.For listing many variables it may become tedious typing in the word WRITE each time and it could be more useful if the command WRITE is made to accept several objects which would then be listed in turn. Another possibility is to allow abbreviation of commands. This would be particularly useful for commands such as WRITEALL which is eight characters long. Generally, users were impressed by the facility to type out the value of variables expressed in Pascal terms and the ability to list arrays, sets and records was much appreciated.

Comments on the display and pointers

As mentioned previously, some large programs, written principally by staff and final year students, were tested using the display facilities. The comments received were favourable and the ability to see the overall shape of a data structure was helpful in many cases. Two general criticisms emerged concerning the display. The first was that due to the format of display presented it can take the user some time to appreciate the implied interconnections between objects. The display is in a format such that when not squashed, the position of each object uniquely defines which other objects are connected. To do this requires a symmetric display of what may not necessarily be a symmetric structure. Once the format is appreciated, users can readily interpret the display provided but there exists a small learning process.

For some data structures, the display provided can appear alien to the user's preconception of the structure. For instance, a list of lists would be displayed in the usual way a binary tree is conceived, that is each object having two objects displayed symmetrically below it. The user would probably conceive of a list of lists as a horizontal list representing the top level with vertical lists hanging from it. In this particular case the required effect can be achieved by tilting either the VDU or the user through 45^0 but other examples could be found where a different format of display would be advantageous.

It is not immediately clear how the diagnostics system would be informed of the required format. The best description would probably be communicated by an interactive question and answer system. Ideally the directional information would be conveyed by some suitable device such as a light pen on a graphics tube. The possibilities of using other media are discussed in the next section. With some structure displays, it may be more expedient for the user to draw the display on paper using the diagnostics system to write out the contents of each object field as it is traversed but it is hoped that this should not be necessary.

The ability to refer to heap objects and list them was well received. The value of a pointer is displayed as the name of the heap object it points to. This facility was described by one user as the best feature of the system.

Overall, the user reaction is extremely favourable. While being impressed by the display facility, useful comment on improving it was received and such improvement is described later. In conclusion, one particular comment was received which is representative of several users' views. This was as follows:

> 'Having obtained a fairly reasonable grasp of how the system operated, I found that whatever the fault the limiting factor on finding it was the time it took to type in the commands'.

This user was developing a program that produces bridge bidding sequences using generalised decision rules linked in lists.

7.3 Areas for further work

The work described here concerns a diagnostics system built with certain objectives in mind. As is the case with many projects, this one is open ended and further developments may be made in the area of diagnostics and particularly in displaying data structures. Suggestions for further work are described and the implications of such work discussed.

7.31 Graphics display

The display of data structures has been provided for output on a line oriented device such as a VDU. This provides such a display to common users of the system but limits the format of this display and the amount of information that can be provided. The ideal device for the display would be a graphics tube but expense inhibits general availability of such devices. There now exists several small computers based on microprocessor technology which are relatively inexpensive and have graphics capabilities displayed on domestic television sets. The Apple II microcomputer is such a device and supports UCSD Pascal {BOWL78} . The Apple II has a high resolution graphics facility consisting of a dot matrix 280 dots wide and 192 dots high. Each dot can be one of six different basic colours. Included is a light pen which can be pointed at any position on the screen. This position is available to the running program in terms of its two co-ordinates.

Such a display screen can be used to produce pictures of objects connected by lines. The different colours can be used to indicate the density of objects, if the display is squashed, or types of objects if they are fully displayed. The light pen could be used to select a window on the display. These capabilities would produce a visually better display which contains more information and is easier to use. This sort of display is defined by a dedicated memory location. A picture of the whole display could then be created in memory using an imaginary large screen and the window of the screen passed over this area of memory by relatively fast bulk memory moving. This would result in the user being able to move a window over the display quite quickly as the form of the display does not have to be continually recalculated. Such Apple computers have recently been aquired locally and a project is under consideration to implement the diagnostics system on them. For users of the existing mainframe computer, the Apple could be used principally as a terminal but could run the diagnostics program as required. The linking of Apples to the GEC 4080 is still at an early stage. The diagnostics package may additionally be implemented within the UCSD Pascal available on the Apples. This is currently prevented by difficulty in obtaining the source code of that Pascal system.

7.32 Concurrent and distributed computing.

The diagnostics system described is designed for individual sequential processes running on one machine only. The areas of concurrent programming and distributed computing pose new problems where diagnostics are concerned. Concurrent programming involves more than one program running autonomously but communicating with each other. Each program may be running on a separate processor or it may be scheduled with each program running for a small period at a time. Where separate processors are used, some form of physical intercommunication is provided such as a distributed computer network. Brinch Hansen {HANS77} discusses errors which are particularly associated with concurrent processes. He points out that certain events may be impossible to repeat as the exact state of all other processes is not known and these states collectively form the environment of each program. Precautions are made in concurrent programming languages such as Concurrent Pascal {HANS77} which restrict data accesses to well defined regions by syntactic constraint. In this way many potential errors are avoided. The programmer is still faced with the possibility of errors and the desire to monitor the programs' states.

It is not immediately clear how a diagnostics system would be used in such an environment. If a program is deliberately frozen at a certain event so that the programmer may examine its state then questions such as whether all other programs should be frozen at that point are raised. It may not be desirable to halt all other programs and it may even be impossible to do so. In a distributed system, the continued execution of a program may depend on the execution of another. If more than one program is to be stopped then this must happen at a meaningful point in their execution which need not be at the same instant for every program. A diagnostics system can be envisaged which could be used to interrogate each program in turn but suffers from two serious drawbacks. The first is that described of deciding which processes to suspend. The second is a problem of communications where several distributed processors are involved. The diagnostics system could exist on each processor but each such system would probably need to communicate with each other and the user may wish for a form of central control. The system could exist on one, possibly dedicated, processor but questions concerning communication with other processors arise and a break of machine communication would alienate the diagnostics system from that machine.

The problems that arise in the area of diagnostics and concurrent programming are many and most are not obviously solved. Despite these obstacles a good case for providing diagnostic facilities to concurrent processes can be made especially as run-time errors are difficult to reproduce or locate. An error in one process can indirectly affect other processes and rapidly spread through the system. It would be extremely helpful if it were possible to devise a diagnostics system that could aid the concurrent programmer.

7.33 Options on structure display

The display of a data structure as described in this diagnostic system, consists of following all pointers with a defined value from a root record down to the bottom of the structure. As described in the previous chapter, pointers in variant parts of records are not followed. There were two reasons given for not following these variant pointers. The first concerns whether or not the user would consider these pointers as linking a substructure rather than making a meaningful contribution to the overall display. This point is further elaborated below. The second reason was due to the error prone nature of abusing record variants. A particular tagfield value may indicate the existence of a pointer but the associated variant may not have been the most recently accessed field and an erroneous pointer value would be used to provide the display. This problem is discussed in the next section where it is argued that it is a problem for the programming language rather than a diagnostics system.

Whether or not the pointers that may exist in a record variant constitute substructures depends on the application. Similarly, any pointer within a record could be considered by the programmer as introducing a substructure. A structure may be created comprising several objects listed in an ordered manner and, for reasons of economy of space, certain attributes of each object may be implemented by pointers to common data objects. In this respect, such pointers are regarded as data values rather than structural building blocks. This distinction can only be made by the programmer. If the hazardous possibilities incurred by variant pointers can be avoided, the display of data structures can be made more useful if the programmer can indicate which pointers within a record are to be used in creating the display. This can be implemented by naming certain pointers or selecting a set of record types which are to comprise the display.

7.4 Conclusions

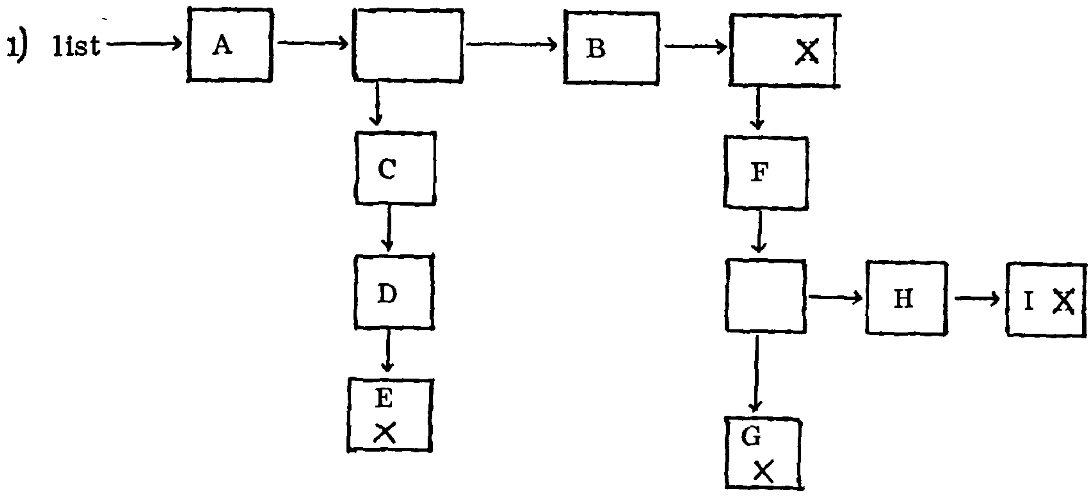
This final section is concerned with summarizing the main features of the work described in this thesis. The portability of the diagnostics system is discussed and some implications on programming language design described.

7.41 Portability of diagnostic systems

The fundamental aim of a diagnostics system is seen in terms of the problem orientation lines introduced in chapter three. The main task is to map the machine implementation of a program back on to the source program. In this respect, the diagnostics system is performing functions of a similar order of complexity to the tasks undertaken by the compiler but working in the opposite direction. The diagnostics program is then seen to be as portable as a compiler. Clearly sections of a compiler and a diagnostics program are machine dependent. These sections can at best be kept small, isolated textually in the program and, ideally, parameterised. The machine dependent sections of the diagnostics program are isolated and the complete system of the P4 Pascal compiler and diagnostics may be transferred to another machine with much the same ease as moving the compiler alone. If the diagnostics system is to be moved to another Pascal system which is not based on Pcode then its portability is less. The diagnostics program would need to be adapted in order to cater for a different compiler interface. This interface comprises the identifiers' name table and the storage layout of data variables. In addition, the alterations made to the P4 compiler as described in the previous chapter would have to be included in this new compiler. Of greater interest is the possibility of transferring the facilities provided by the diagnostics system to another language. Diagnostic systems have been described for other languages. There are sections of this diagnostics system that could be applicable to other languages. The techniques employed for naming heap objects and creating a display of linked records are not confined to Pascal. The facilities provided and their implementation are seen as diagnostic techniques in general in the same way that many compiling techniques exist without being contained in one language. This diagnostics system is aimed at high level languages which provide data-structuring facilities and dynamic creation of data items. The diagnostic facilities provided are applicable to such languages; the interface to different languages concerns the details of implementing dynamic data objects and pointer representation. The diagnostics system has shown that it is possible and desirable to provide such information to the high level language user. In isolating an area that is ignored by previous diagnostic systems and filling the gap found it is hoped that future systems will do the same.

7.42 Structures using pointers

The display of data structures as described assumes a linking of objects by pointers. Pointers are used in the creation and manipulation of such structures in languages such as Pascal and Algol 68. The linked representation of structures is not, however, the only method. In list processing languages such as LISP, although the list elements are linked by pointers, this fact does not need to be of concern to the user. Two possible representations of the same list structure are:



2) list = (A, (C, D, E), B (F, (H, I), G))

fig 7.1

Both the above representations refer to the same list. The first indicates a linking of twelve elements. The second shows one structure with several substructures denoted by nested paranthesis. A third representation of this list which shows the hierarchy of the substructures could be:

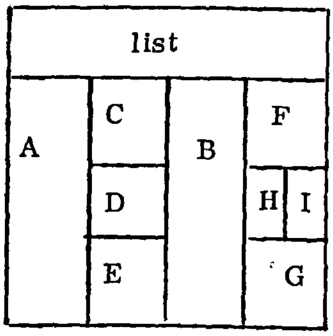


fig 7.2

Which representation is most meaningful to the programmer will depend on the particular application.

A second area where pointers are not directly required in representation is a related data base. Consider the following data base and its relational table:

Employee relation

Key	Name	Age	Dept
Thomas			
Smith			
Jones			
Morgan			
Brown			
Lloyd			

'Works for' relation

Thomas	Smith
Thomas	Jones
Jones	Morgan
Jones	Brown
Jones	Lloyd

fig 7.3

The employee relation contains entries consisting of a key (in this case a name) and some data relating to that key. The 'works for' relation shows the interconnections between the data objects. The first entry in this table shows a connection between Thomas and Smith. Later in the table, Jones is shown to be connected to Morgan, Brown and Lloyd. An alternative representation of this data base is:

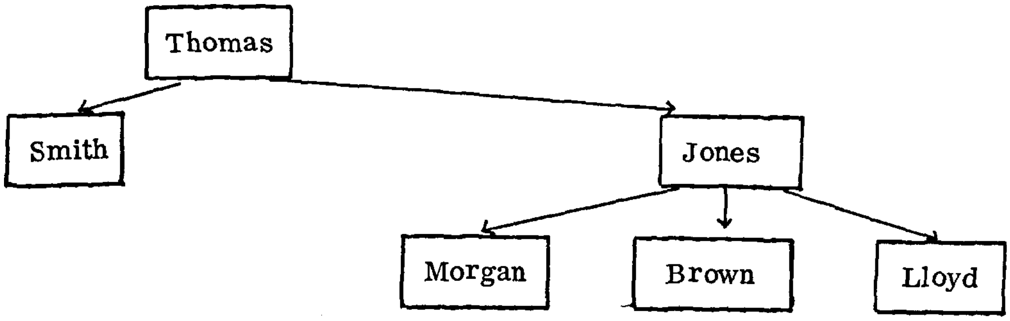


fig 7.4

Within the relations, the keys used are effectively pointers or more precisely each element of the works for relation indicates the presence of a pointer. The main reason for this method of representing the above structure is that an alteration of the data base is simple. Pointers can be added or removed without altering the employee relation but by amending the works for relation. Using names as keys instead of absolute addresses allows alteration of the ordering of the data base without altering its structure. This representation is used for efficiency and access path independent representation in manipulating a large data base.

The list representation that does not refer to pointers is not easily applied to structures which are implicitly recursive or involve cycles. The relational data base can easily accommodate cycles as each relation represents a pointer. In his book on algorithms and data structures, Wirth {WIRT76} points out that the use of pointers can be hazardous and shows that assignment to a pointer is parallel in data structures to a goto statement in algorithms. Hoare also points out the error prone nature of pointers {HOAR75} . Wirth justifies the use of pointers in two ways. To otherwise create a recursively defined data structure requires knowledge of the complete structure beforehand. By using pointers, a structure can be developed incrementally. Secondly using pointers allows sharing of common storage. Although in accessing common data, it does not matter to the programmer whether that common data is duplicated within the structure, this difference is important when such data is updated. Wirth gives a good example of a family tree, which is usually considered a regular structure, that is extremely irregular and representation without pointers would be difficult. It involves the actual case of a man in Zurich in 1922 who through several incidences of marriage becomes his own grandfather and whose brother is also his grandson. The full story is given in Appendix 5.

The decision on representation of a structure by a display involves a knowledge of the application. The display provided by the diagnostics system described uses pointers and is general purpose. The reason for this is that Pascal uses pointers enabling general purpose data structuring facilities. Where the application is specific such as list processing a different representation may be applicable. This special case applied to lists is supported by languages designed specifically for list processing. As indicated earlier, the display provided could be made more useful if the user can specify the format to be used and media such as a graphics tube are available. Where an application is considered unique enough to warrant special languages such as list processing, this uniqueness should be included in any diagnostic display of the resulting lists. This uniqueness is arguably confined to such languages.

7.43 Implications for program language design

Two main problems in the language processed by the P4 Pascal compiler arose during the implementation of the diagnostics system. The first is the insecurity of the record variant and the second is the detection of undefined pointer values. The problem with record variants lies in altering a tagfield value and accessing the variant fields selected. Consider the following example:

```
var rec : record
          case which : Boolean of
          true : (i : integer),
          false : (p : ↑integer)
          end:
.
.
.
.
rec. i : =10;
rec. which : = false;
.
.
.
```

Following the alteration of the tagfield 'which' the assumed extant variant is the pointer p which in this implementation would point to an 'integer' at store location 10. The Pascal report states that in this case, the programmer must realise that the value of the pointer p is to be considered illegal. This leaves the onus on the programmer to expect undefined values in such circumstances. The unfortunate consequence is that the diagnostics system, when tracing the above record for construction of a display, would assume the pointer p exists and follow it through. If the value found in p is coincidently within the current heap range it will not be considered illegal and a seriously misleading display would ensue.

This problem does not occur in either Algol 68 or ADA which have a similar facility. In Algol 68 {WIJN76} the associated tagfield is hidden from the programmer. Whenever such a variant is used, the action to be taken for each possible variant must be specified. In ADA {ICHB79} the syntax of a record variant is very similar to that in Pascal. The tagfield variable, however, cannot be assigned to as an individual object. The tagfield value can only be changed when the record is defined or if the entire record is assigned to. To remove this insecurity in Pascal would require a large overhead at run-time. Whenever a tagfield is altered,the variant fields exposed would have to be marked as undefined.

The existence of variables with an undefined or illegal value poses the second problem referred to above, the detection of such values. The Pascal report and the draft British standard for Pascal both refer to undefined values for data objects in certain circumstances. These circumstances include objects' initial value, variant record fields exposed by a change in the tagfield value and scalar items used as the control variable of a for statement after completion of that loop. The reports do not state that the use of an undefined value, if coincidentally valid, is to be regarded as an error. The reason for this is that detection of such an error requires significant run-time checks.

When a variable becomes undefined it would have to be given a unique value which if used would be detected as an error. If this were done, two difficulties arise. Firstly, it may be impossible to find one value which can be used for each standard data type representation. Secondly the position and size of exposed variant fields would have to be known at each instance of a tag-field changing value. Whilst neither of these problems is insurmountable the overheads involved to set each variable to the unique undefined value that is assigned to each scalar type would be large.

There still exists the need for the diagnostics system to detect pointers which are officially undefined. The initial value of all data items in this implementation is always set to zero (for the reasons given in chapter four concerning comparisons of records which contain inaccessible fields due to alignment conditions). The value zero is never a legal value for a pointer in this system as the stack is formed from store location zero. The diagnostics system can then detect pointers that have not been assigned a value. This leaves two possible occasions where a pointer could have a value that strictly is undefined but in practice could point to a valid area in the heap. The first occasion is when a tagfield value is changed. The second occasion is commonly referred to as the dangling pointer. Consider a pointer p as shown:

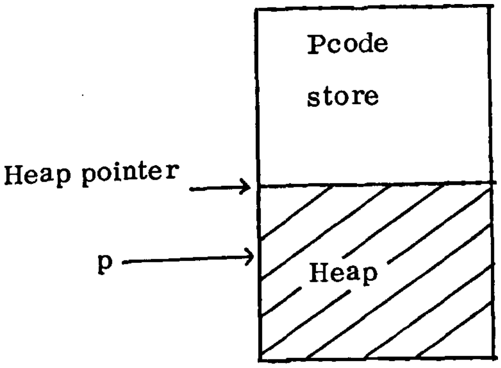


fig 7.5

If space is reclaimed on the heap, by use of the P4 Pascal procedure release, the situation could be as follows:

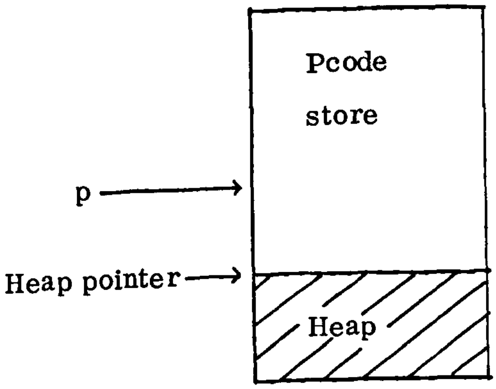


fig 7.6

This pointer would be detected as having an illegal value as it does not point into the current heap area. Should new objects be created on the heap, the pointer p could then point into the heap area but not refer to any meaningful object. This pointer would then cause an erroneous display to be created by the diagnostics system. This error can only be avoided in the P4 Pascal system by inspecting every extant pointer prior to allocating space on the heap. The location of every pointer would be very costly unless each Pcode store location is flagged with its type.

Both these problems concerning undefined pointers require prohibitively expensive detection mechanisms. Neither mechanism has been implemented on this system on the grounds that where such large overheads are involved it is better to consider the language design that causes them. To eliminate the problem associated with variant record fields, the approach described for ADA seems appropriate. By forbidding the alteration of a tagfield except where the whole record is assigned to removes the problem with little inconvenience to the programmer. The dangling pointer problem stems from the method used for reclaiming heap storage. The standard Pascal procedure for this reclamation is dispose. This has the effect of releasing the store object pointed at by a named pointer. The pointer so named can then easily be given a suitable value but other pointers may refer to the released data object. Algol 68 does not contain a procedure whereby the programmer can specify store reclamation. The run-time environment reclaims store by garbage collection; when store is exhausted, all pointers are examined in order to locate areas of store no longer referred to. While this process is extremely costly, it is only invoked when all store has been used. There appears no cheap solution to this problem. The process of automatic garbage collection would seem to be the only error free solution.

The overhead involved in each of these techniques is that of identifying which data objects are pointers. This overhead can be overcome by suitable hardware. The overhead of checking array subscripts, for example, has been considerably lessened on many modern machines (eg MU5) by hardware implemented checks. Some machines now include tag bits on each machine word identifying its type. The Burroughs B5700 and B6700 {ORGA73} were amongst the first to adopt this idea. The pointers can then be readily identified and an extension to this would be for the hardware to alter all pointers which become undefined as storage allocation changes to a particular value.

7.44 Summary

The work described has been concerned with reliable software and in particular those tools available to a programmer aiding the testing of programs and helping locate errors. The area where such tools are needed was found to be that of data structures and the associated facilities of dynamically creating data objects during a program run. The provision of a diagnostics system is argued to be as essential as the compiler itself if the user is to be shielded from details of language implementation. Two implementation details are seen as particularly interesting. These are the mechanism for creating data objects and the representation of linked structures in a manner in which the programmer may recognise. The dynamically created data objects are allocated unique names for subsequent identification. The diagnostics system is most useful as an interactive aid because the user may then selectively examine data objects as experience of the program behaviour develops. There existing no suitable interactively available language locally; Pascal was implemented first of all on a Digico Micro 16E computer and then on the GEC 4080 central service computer. The latter is used for teaching and research by a large population in excess of a hundred. The diagnostics system was then incorporated into this implementation and received much use. The user reaction is favourable and the facilities to display structures and give the values of all extant data items in source language items is well appreciated. The provision of these facilities is seen as incorporating those functions which are of most use to the programmer whether a beginner or an experienced user. There is scope for improvement of the diagnostics system and, particularly since the arrival of cheap personal computers with graphics facilities, suggestions for enhancements are made. Such enhancements should be limited so that useful features are not lost amongst a host of little used extras. Simplicity of a tool such as the diagnostics system described is a good measure of its usefulness. It is believed that a balance between this simplicity and the provision of facilities complementing the full power of a language such as Pascal has been achieved.

Appendix 1

Differences between Pascal P4 and Standard Pascal

The differences between the language processed by the Pascal P4 compiler {JACO74} and the standard Pascal as defined in the user manual and report {WIRT75} are as follows:

1 Procedure and function formal parameters are not allowed

2 Files are not implement except for four predeclared files, input, output, prd, prr, of type file of char.

3 goto s may not lead out of a procedure or function body.

4 The subrange form of set constants is not allowed.

5 nil is not a reserved word but predeclared.

6 'text' and 'maxint' are not predeclared.

7 The standard function 'dispose' is replaced with 'mark' and 'release'.

8 Standard procedures 'round' 'page' 'pack' and 'unpack' are not implemented.

9 Output of Boolean quantities is not supported.

10 Only one field width specifier may be used on a formatted write of real quantities.

11 The reserved word packed is ignored.

Appendix 2

Pcode instruction sets

A. Original P4 Pcode instruction set

Operation codes

0	ABI	integer abs
1	ABR	real abs
2	ADI	integer add
3	ADR	real add
4	AND	logical and
5	DIF	set difference
6	DVI	integer divide
7	DUR	real divide
8	EOF	test end of file
9	FLO	float integer below top of stack
10	FLT	float integer top of stack
11	INN	test set membership
12	INT	set intersection
13	IOR	logical or
14	MOD	integer modulus
15	MPI	integer multiply
16	MPR	real multiply
17	NGI	integer negate
18	NGR	real negate
19	NOT	logical not
20	ODD	test whether integer is odd
21	SBI	integer subtract
22	SBR	real subtract
23	SGS	generate singleton set
24	SQI	integer square
25	SQR	real square
26	STO	store indirect
27	TRC	function trunc
28	UNI	set union
29	STP	stop execution
30	CSP	call standard procedure/function
31	DEC	decrement
32	ENT	reserve local variable space
33	FJP	jump if top of stack false
34	INC	increment
35	IND	index array
36	IXA	calculate indexed address
37	LAO	load address
38	LCA	load constant address
39	LDO	load
40	MOV	move region of store
41	MST	mark stack
42	RET	return from procedure
43	SRO	store
44	XJP	indexed jump
45	CHK	check subrange bounds
46	CUP	call user procedure
47	EQU	test equality
48	GEQ	test greater or equal
49	GRT	test greater
50	LDA	load address at given lexical level
51	LDC	load constant
52	LEQ	test less or equal

Appendix 2 (cont)

53	LES	test less
54	LOD	load at given lexical level
55	NEQ	test inequality
56	STR	store at given lexical level
57	UJP	jump
58	ORD	function ord
59	CHR	function chr
60	UJC	error in case statement

B Pcode Instruction set for GEC 4080 implementation

Operation codes

0	ABI	integer abs
1	ABR	real abs
2	ADI	integer add
3	ADR	real add
4	AND	logical and
5	DIF	set difference
6	DVI	integer divide
7	DVR	real divide
8	EOF	test end of file
9	FLO	float integer below top of stack
10	FLT	float integer on top of stack
11	INN	test set membership
12	INT	set intersection
13	IOR	logical or
14	MOD	integer modulus
15	MPI	integer multiply
16	MPR	real multiply
17	NGT	negate integer

18	NGR		negate real
19	NOT		logical not
20	ODD		test whether integer is odd
21	SBI		integer subtract
22	SBR		real subtract
23	SGS		generate set with given membership range
24	SQI		integer square
25	SQR		real square
26	STO		store indirect
27	TRC		function trunc
28	UNI		set union
29	STP		stop execution
30	CSP		call standard procedure/ function
31	DEC	1	decrement scalar
32	ENT	1	reserve variable space (procedure entry)

Appendix 2 (cont)

33	FJP		jump if top of stack false
34	INC	1	increment scalar
35	IND	1	index scalar array
36	IXA		calculate index address
37	LAO		load address
38	LCA		load constant address
39	LDO	1	load scalar
40	MOV		move region of store
41	MST		mark stack
42	RET		return from procedure
43	SRO	1	store scalar
44	XJP		indexed jump
45	CHK		check subrange bounds
46	CUP		call user procedure
47	EQU	1	test scalar equality
48	GEQ	1	test scalar greater or equal
49	GRT	1	test scalar greater
50	LDA		load address at given lexical level
51	LDC	1	load constant scalar
52	LEQ	1	test scalar less or equal
53	LES	1	test scalar less
54	LOD	1	load scalar at given lexical level
55	NEQ	1	test scalar inequality
56	STR	1	store scalar at given lexical level
57	UJP		jump
58	ORD		function ord
59	CHR		function chr
60	UJC		error in case statement
61	LCI	1	load scalar indirect
62	LIN		line instruction
63			
64	ENT	2	reserve stack space (procedure entry)
65			
66			
67			
68			
69			
70			
71			
72			
73			
74			
75			
76			
77	CHKA		check address
78			
79	EQU	M	test array/record equality
80	GEQ	M	test array greater or equal
81	GRT	M	test array greater
82			
83			
84	LEQ	M	test array less or equal
85	LES	M	test array less
86			

Appendix 2 (cont)

87	NEQ	M	test array/record inequality
88			
89			
90	STO	S	store set indirect
91	GEQ	R	test real greater or equal
92	GRT	R	test real greater
93			
94			
95	LEQ	R	test real less or equal
96	LES	R	test real less
97	STO	C	store byte direct
98			
99	IND	S	index set array
100			
101			
102	DEC	C	decrement byte
103	LDO	S	load set
104			
105	INC	C	increment byte
106	IND	C	index byte array
107	SRO	S	store set
108			
109			
110	LDO	C	load byte
111	EQU	S	test set equality
112	GEQ	S	test set inclusion
113			
114	SRO	C	store byte
115	LDC	S	load constant set
116	LEQ	S	test set included
117			
118	LOD	S	load set at given lexical level
119	NEQ	S	test set inequality
120	STR	S	store set at given lexical level
121			
122	LDC	C	load constant byte
123			
124	LDC	N	load nil address
125	LOD	C	load byte at given lexical level
126	NOP		no operation
127	STR	C	Store byte at given lexical level

Standard procedures/functions

1P	GET	input
2P	PUT	output
3P	RDI	read integer
4P	RDR	read real
5P	RDC	read character
6P	WRI	write integer
7P	WRB	write Boolean
8P	WRR	write real - 1 format parameter
9P	WRC	write character
10P	WRS	write string

Appendix 2 (cont)

11P	PAK	pack (not implemented)
12P	NEW	claim heap space
13P	RST	restore heap space
14P	ELN	test for end of line
15P	SIN	)
16P	COS	)
17P	EXP	)standard
18P	SQT	)mathematical
19P	LOG	)procedures
20P	ATN	)
21P	RLN	read line
22P	WLN	write line
23P	SAV	save value of heap pointer
24P	RES	reset file
25P	REW	rewrite file
26P	HLT	halt
27P	ENV	function environ
28P	RND	generate random number
29P	RAN	randomise random number seed
30P	PAG	write page
31P	OPN	open file
32P	CLS	close file
33P	ROU	function round
34P	GRM	random access file input
35P	PRM	random access file output
36P	WRF	write real - 2 format parameters
37P	RSP	restart program
38P	UPD	update random access file

Appendix 3 Differences between UCSD Pascal and standard Pascal.

1 The standard function arctan is called atan

2 If no match is found in a case statement, the case statement is ignored in UCSD Pascal.

3 The standard procedure dispose is replaced by mark and release.

4 The standard files input and output are predeclared as type 'interactive'. Files of this type have no character look ahead facility, but are otherwise similar to text files.

5 Random access files are allowed.

6 Read/write may only apply to files of type text or interactive.

7 Goto may not lead out of a procedure or function, but a procedure called exit is provided which can cause control to leave any named procedure.

8 Any parameter list specified in the program heading is ignored by UCSD Pascal.

9 The procedures reset/rewrite may contain a string parameter to specify an actual I/O device.

10 Many string handling facilities are provided, and 'long' integers can be processed.

11 Procedures/functions may not be formal parameters.

Appendix 4 V24 Newbury VDU modem connections.

The printer socket of the Newbury VDU is a standard V24 interface. This requires three connections - transmit, receive and a common earth. A modem connection requires, in addition, four more connections (some modems have five extra connections, the fifth being used to select the transmission speed). These four are used to detect whether the machine the modem is connected to is ready to transmit or receive data as follows:

1) RQTS Request to send a signal to the modem.
This signal should be present to ensure the modem becomes enabled to receive data.

2) RFS Ready for sending.
This signal is present if the modem is ready to transmit data.

3) DSR Data set ready.
This signal is present when the modem is ready to receive data.

4) DCP Data carrier present.
This signal signifies that the modem has a carrier signal. The modem connection is completely disabled if this is not present.

The first signal is always present from the Digico line which expects the other three to be present before it is operative. To enable the Digico line, in addition to the three data connections to the printer socket, the above four connections are soldered together.

Appendix 5 Family Tree

This story appeared in a Zurich newspaper in July 1922 and is quoted by Wirth in his book 'Algorithms & Data Structures = Programs':

> 'I married a widow who had a grown-up daughter. My father, who visited us quite often, fell in love with my step-daughter and married her. Hence, my father became my son-in-law, and my step-daughter became my mother. Some months later my wife gave birth to a son, who became the brother-in-law of my father as well as my uncle. The wife of my father, that is my step-daughter, also had a son. Thereby, I got a brother and at the same time a grandson. My wife is my grandmother, since she is my mother's mother. Hence I am my wife's husband and at the same time her step-grandson; in other words I am my own grandfather'.

ACM77 ACM 1977. Proceedings of the 6th Symposium on Operating System Principles.
Operating Systems Review. Vol II No. 5 Nov. 1977

ADDY78 Addyman AM et al 1978. 'Working Draft/3 on Pascal'
British Standards Institute.

APPL79 Apple Computer Inc. 1979. 'Apple II Reference Manual'.

BARR77 Barron D 1977. 'An Introduction to the Study of Programming Languages'. Cambridge University Press.

BARR79 Barringer H, Capon P C and Phillips R 1979. 'The Portable Compiling Systems of MUSS'. Software - Practice and Experience Vol. 9 No. 8 (645-656).

BATE74 Bate DG 1974. 'Design and Implementation of an Interactive Test Bed'. Software - Practice and Experience Vol. 4 No. 1 (91-109).

BIRT73 Birtwhistle GM, Dahl Mhyrbaug and Nygaard 1973.
'Simula Begin' Auerbach.

BOWL78 Bowles K 1978. 'The UCSD Pascal Project'. Educom Bulletin Vol 13.

BRAD77 Brady JM 1977, 'The Theory of Computer Science.' Chapman and Hall.

BROO66 Brooker RA, Rohl JS and Clark SR 1966. 'The Main Features of Atlas Autocode.' Computer Journal Vol 8 No. 4 (303-311).

CAPO71 Capon PC, Morris D, Rohl JS and Wilson R 1971. 'The MU5 Compiler Target language and Autocode' Computer Journal Vol 15 No 2 (109-124).

CLAR67 Clark SR 1967 'Compiling Techniques' PhD Thesis
Manchester University.

COLE78 Coleman D 1978 'A Structured Programming Approach to Data'.
MacMillan.

CONW63 Conway M 1963 'Design of a separable transition diagram compiler' Communications of the ACM Vol 6 No. 7 (396-408).

CONW73 Conway RW and Wilcox TR 1973. 'Design and Implementation of a Diagnostic Compiler for PL/1.' Communications of the ACM Vol 16 No. 3 (169-179).

DAHL72 Dahl O, Dijkstra EW and Hoare CAR 1972. 'Structured Programming' Academic Press.

DECA 'DEC System 10 Algol Manual' DEC Computers Ltd

DECB 'DEC System 10 User Manual' DEC Computers Ltd.

DIAB76 Diablo 1976. 'Hyterm Communication Terminal User Manual' Diablo Systems Incorporated.

DIGIA 'Digico. Generalised Communication Interface card - GCIC' Digico Ltd

DIGIB 'Digico. Micro 16E User Manual.' Digico Ltd.

DIGIC 'Digico. Assembler users Handbook.' Digico Ltd.

DIJK68 Dijkstra EW 1968. 'The Goto Considered Harmful' Communications of the ACM Vol. 11 (147-148).

DIJK79 Dijkstra EW 1979. 'A Discipline for the Programming of Interactive I/O in Pascal'. ACM Sigplan Notices Vol 14 No. 12 (59-61).

FLOY67 Floyd R. 1967. 'Assigning Meanings to Programs'. Proceedings of a Symposium in Applied Mathematics Vol. 19 (ed. Schwarz) American Mathematical Society (19-32).

FORD76 Ford B 1976. 'The Evolving Nag Approach to Software Portability.' P. Brown. Cambridge University Press

FRAN77 Frank GH 1977. 'A Portable Operating System.' Software Portability. P Brown. Cambridge University Press

GECA 'GEC 4000 Series.' GEC Computers Ltd.

GECB 'GEC Babbage User Manual.' GEC Computers Ltd.

GORD77A Gordon M, Milner and Wadsworth 1977. 'Edinburgh LCF'. Department of Computer Science. Edinburgh University.

GORD77B Gordon M, Milner, Wadsworth, Morris and Newey. 1977. 'A Metalanguage for Interactive Proof in LCF' Department of Computer Science. Edinburgh University.

GRIS76 Griswold RE 1976. 'Engineering for Portability.' Software Portability. P.Brown. Cambridge University Press

HANS77 Hansen PB 1977. 'The Architecture of Concurrent Programs'. Prentice Hall.

HOAR71 Hoare CAR 1971 'Proof of a Program Find.' Communications of the ACM Vol 14, No. 1 (39-45)

HOAR75 Hoare CAR 1975. 'Data Reliability' ACM Sigplan Notices Vol. 10 (528-533)

ICHB79 Ichbiah JD et al 1979. 'Rationale for the Design of the Ada Programming Language.' ACM Sigplan Notices. Vol. 14 No. 6 Part B.

ICL76 ICL 1976 'Cobol 1900 Series.' ICL Computers Ltd

IGAR73 Igarishi London and Luckman 1973 'Automatic Verification of Programs. ' Stanford Report CS365.

JACK75 Jackson MA 1975 'Principles of Program Design. ' Academic Press.

JACO76 Jacobi C. Nori, Amman, Jansen and Nageli. 1976 'Pascal (P) Compiler Implementation Notes. ' ETH Zurich.

JENS75 Jensen K and Wirth N. 1975. 'Pascal User Manual and Report. ' Prentice Hall.

KNUT67 Knuth DE 1967. 'The Remaining Trouble Spots in Algol 60. ' Communications of the ACM Vol.10 No.10 (611-618).

LAVI75 Lavington SH 1975. 'A History of Manchester Computers. ' National Computing Centre.

LEAV70 Leavenworth B 1970 'Review of a Paper by P Naur. ' Computer Reviews Vol.11 No.7 (19,420 pp 396-397).

LOND75 London RL 1975. A View of Program Verification. ' Proceedings of the International Conference on Reliable Software. Los Angeles (534-545).

MANC75 Manchester University Regional Computer Centre 1975. 'Using the UMRCC Fortran In-core Compiler. '

MANC76 Manchester University Regional Computer Centre 1976. 'Using the UMRCC Algol 60 In-core Compiler. '

MART70 Martin J 1970 'The Computerised Society. ' Prentice Hall.

MCCA63 McCarthy J 1963. 'A Basis for a Mathematical Theory of Computation'. Computer Programing and Formal Methods (ed. Braffort and Hirschberg), North Holland (33-69).

MICK Mickel A (Editor) 'Pascal News' Pascal Users' Group Newsletter.

MICR78 Microengine Company 1978. 'The WD900 Pascal Microengine'.

MOUL67 Moulton PG and Muller ME 1967. 'Ditran - A Compiler Emphasising Diagnostics. ' Communications of the ACM Vol.10 No.1 (45-52).

NAUR62 Naur P 1962 (Editor) 'Revised Report on Algol 60'. International Federation for Information Processing.

NAUR66 Naur P 1966. 'Proof of Algorithms by General Snapshots'. Bit 6 (310).

NAUR69 Naur P 1969 'Programming by Action Clusters'. Bit 9 (250-258).

NEUM63 Neuman JV 1963 'Collected Works 5' MacMillan.

NEWB Newbury Laboratories Ltd. 'Operator Instruction Manual - Newbury Visual Display Terminal. '

ORGA73 Organick ET 1973. 'Computer System Organisation: the B5700/B6700 Series. ' Academic Press.

RAND78 Randell B, Lee P, Treleaven P 1978. 'Reliability Issues in Compiling System Design.' Computer Surveys.

ROHL75 Rohl JS 1975 'An Introduction to Compiler Writing.' MacDonald.

SING78 Singleton P 1978. 'A Floating Point Package for the Digico Micro 16E.' Computer Science Department, Keele University.

WATT77 Watt DA and Findley W 1977. 'A Pascal Diagnostic System' Proceedings of the Third Computer Studies Symposium. Southampton.

WELS72 Welsh J and Quinn C 1972. 'A Pascal Compiler for ICL 1900 Series.' Software - Practice and Experience. Vol 2 No. 1 (73-77)

WHIT79A White N 1979. 'Digico to 7905 Link.' Computer Science Department, Keele University.

WHIT79B White N 1979. 'Digico to 4082 Link.' Computer Science Department, Keele University.

WICH76 Wichmann B 1976. 'Use of Algol 60.' Software Portability. P. Brown, Cambridge University Press.

WIJN76 Wijngaarden A et al 1976. 'Revised Report on the Algorithmic Language Algol 68.' Springer Verlag.

WIRT71A Wirth N 1971. 'Program Development by Stepwise Refinement.' Communications of the ACM Vol 14 No. 4 (221-227).

WIRT71B Wirth N 1971. 'The Programming Language Pascal.' Acta Informatica Vol 1 (35-63).

WIRT71C Wirth N 1971. 'Design of a Pascal Compiler.' Software - Practice and Experience Vol 1 (309-333).

WIRT74 Wirth N 1974. 'Pascal and New Pascal.' Pascal Newsletter No 2

WIRT75 Wirth N and Jensen K 1975. 'Pascal User Manual and Report.' Prentice Hall.

WIRT76 Wirth N 1976. 'Algorithms + Data Structures = Programs.' Prentice Hall.

WIRT77 Wirth N 1977. 'Modula'.Software - Practice and Experience Vol 7 No. 1 (3-35).

WOOD72 Woodward PM and Bond SG 1972. 'Algol 68R Users' Guide.' Ministry of Defence

YOUR75 Yourdon E 1975. 'Techniques of Program Structure and Design.' Prentice Hall.