TOWARDS A COMPUTER UNDERSTANDING OF PROGRAM DESIGN

VOLUMES 1 AND 2

VOLUME 1

Philip Ainsley Fox

Submitted for the Degree of

Doctor of Philosophy

The University of Aston in Birmingham

November 1984

The University of Aston in Birmingham

Towards a Computer Understanding of Program Design

Philip Ainsley Fox

Submitted for the Degree of
Doctor of Philosophy

1984

Summary

Program design is one of the many processes involved in program development and is considered to be essential to the development of structured programs. Consequently this research has been concerned with the analysis of program design since it is considered to be of equal importance to other areas of Artificial Intelligence (AI) research, which analyse the program code. Because a rigorous program design results in a program containing few errors, a system capable of analysing program designs should assist these other related areas of AI.

This research has developed the Framework for Analysing Program Designs (or FAPD) in order to analyse the kinds of program design produced by programmers using the principles of structured programming. The process of analysis is viewed as comprising four distinct phases, which are referred to as pre-semantic analysis, semantic analysis, generation of comments and code generation. The results of analysis take the form of a coded version of the program design together with any comments about the code. Analysis is based on a set of structures which have been developed in order to represent phrases and statements often used in a program design. Attached to each structure is a procedure, referred to as a class instance, which translates its structure into a particular programming language.

FAPD has been implemented and tested within a system called DACE (which is a Design Analysing and Commenting Environment). FAPD is discussed within the context of the system and the results from testing it are discussed in detail. The conclusions are drawn that FAPD represents a viable approach to the computer analysis of program designs, the system has some influence on those who use it and that class instances are a useful acquisition to the set of tools currently available to researchers in AI.

Keywords : design, program, structure, class instance.

Acknowledgements

I would like to thank my supervisor, Dr. A.J. Harget for his many helpful comments and encouragement, and Mr. I.H. Chisholm for his assistance in all aspects of this research. I would particularly like to thank all my colleagues at Cadbury-Schweppes plc who made the final stages of this research possible. Finally, I would like to thank my parents and Mr. and Mrs. M.R. Thompson for their assistance in getting this thesis into its printed form.

The work of this thesis has been supported financially by the Science Research Council.

CONTENTS

Volume 1

CONTENTS

Volume 2

DIAGRAMS

1. INTRODUCTION

1.1 Aims and Objectives

The motivation behind this work was derived from studying the topic of program understanding which is an area of research in Artificial Intelligence (AI). The objective of program understanding is to determine whether or not a program performs as intended, by matching a program's actual performance with a specification of what it is intended to achieve. Any discrepancies between the two will indicate the departure of the program from its specification and then an attempt can be made either to correct the program or to provide some useful debugging information.

Program design is one of the many processes involved in program development of which coding is the final part. The importance of program design is well established and is considered to be essential to the development of structured programs. In our opinion, research concerned with the analysis of program design should be of equal importance to that given to the related area of program understanding. This is not the case at the present time. Thus in an attempt to rectify this situation, a system for analysing program designs was investigated. It is hoped that such a system could be used to impress upon a programmer the importance of the design process and the level of detail required in a program design.

In the remaining sections of this chapter we define the term "program design" and then consider how, in general terms, a program design may be analysed. Sections 1.2 and 1.3 are concerned with the principles of program design. Section 1.4 concludes the Introduction by discussing how the results from analysing a program design can be suitably represented.

1.2 Program Design

To-day we live in a society which places considerable reliance on the computer. Recent progress in the area of hardware technology, together with ever-reducing costs, have led to computers being used in a larger number of applications. Consequently software has increased in complexity with a concomitant increase in the need for software clarity, modifiability and efficiency. These requirements can only be achieved if programmers adopt a disciplined approach to the process of program development.

Early attempts at imposing discipline led to the development of the principles of structured programming [Dijkstra 1968, Wirth 1971]. These principles propose that a program should be successively refined into a series of sub-problems, each of which needs to be solved in order to solve the original problem. This has the benefit that each sub-problem produced is easier to solve than the original. Furthermore, each sub-problem can be considered separately and decomposed further until as Wirth [Wirth 1971] states:

> "this successive decomposition or refinement
> of specifications terminates when all
> instructions are expressed in terms of an
> underlying computer or programming language ..."

A solution to the original problem, namely a program design, can be expressed using suitable combinations of:

a) a sequence of actions;

b) a selection of actions according to the results of some condition; and

c) a repetition of actions,

where an action is defined to be either a single instruction, such as the addition of two numbers, or an instruction which is itself comprised of a set of simpler actions. The latter is often referred to as a compound statement. Consequently at each stage of the decomposition the programmer must decide how his solution can be expressed using a combination of the three programming options described above.

Let us consider how this method might be used in order to design an ALGOL 68C program for the following problem specification:

"A company has a number of weekly paid employees who receive their wages in cash. The company operates a piecework scheme which means the wage bill can vary considerably from week to week. The number of employees together with their individual earnings (in pence) are recorded weekly in a data file. Calculate the number of £5 and £1 notes, together with the number of 50p, 10p, 5p, 2p and 1p coins the cashier will need in any given week to pay out the wages"

A solution to this problem is shown in diagrams 1 to 4 inclusive. The first stage in the solution is to decide how the problem can best be solved using a combination of the three options outlined above.

Typically a programmer can use the target language, chosen here to be ALGOL 68C, to express the solution to those sub-problems which are easily solved. Less tractable sub-problems can be left until a later stage in the design process. A typical first attempt at the program design is shown in diagram 1. This illustrates that in terms of the programming options given earlier (see section 1.2) the initial design is described in terms of a single or direct action, the read statement in line 2 followed by "n" repetitions of the single activity in line 4 and a second direct action, the print statement in line 5.

The solution in diagram 1 is now defined in terms of the two sub-problems in lines 4 and 5, namely the processing of an employee's data and the printing of the results. The programmer can now concentrate attention on the first of these two sub-problems. Since the process for analysing an employee's data involves several calculations, a compound statement is chosen. The result is shown in diagram 2 which illustrates how the processing of an employee's data has been broken down into the eight sub-problems shown in lines 4 to 11 inclusive. Collectively these form a compound statement delimited by the ALGOL 68C reserved words DO and OD. The solution is now defined in terms of these eight sub-problems together with the sub-problem in line 13 which still remains to be considered.

Each of the steps contained within the loopbody may be considered in turn and diagram 3 illustrates how the first two steps may be made more explicit. At this stage the solution has been reduced from nine to seven sub-problems, (shown in lines 10 to 15 and line 17 of

```
begin   int  n;
        read (n);
        for  i  to  n
           do  process data for employee  od;
        output the number of coins and the
                     number of notes needed
end
```

Diagram 1

The First Stage of a Program Design

```
begin   int  n;
        read (n);
        for  i  to  n
           do  input the value of wage ;
               calculate the number of fivepounds
                              needed so far ;
               calculate the number of poundnotes
                              needed so far ;
               calculate the number of fiftypences
                              needed so far ;
               calculate the number of tenpences
                              needed so far ;
               calculate the number of fivepences
                              needed so far ;
               calculate the number of twopences
                              needed so far ;
               calculate the number of onepences
                              needed so far ;
           od ;
        output the number of coins and the
                     number of notes needed
end
```

Diagram 2

The Second Stage of a Program Design

```
begin   int  n, wage, fivepounds ;
        read (n) ;
        fivepounds := 0 ;
        for i  to  n
           do  read (wage) ;
               while  wage  >= 500
               do     fivepounds := fivepounds + 1 ;
                      wage := wage - 500
               od ;
               calculate the number of poundnotes
                                   needed so far ;
               calculate the number of fiftypences
                                   needed so far ;
               calculate the number of tenpences
                                   needed so far ;
               calculate the number of fivepences
                                   needed so far ;
               calculate the number of twopences
                                   needed so far ;
               calculate the number of onepences
                                   needed so far
           od ;
        output the number of coins and the
                    number of notes needed
end
```

Diagram 3

The Third Stage of a Program Design

diagram 3), and a moment's thought at this stage shows that each of the remaining calculations in the loopbody will involve similar design decisions to those taken for the first calculation. Hence because similar processing is required the programmer may decide to implement each calculation in the form of a procedure. The final program would then be similar to that shown in diagram 4.

By using the principles of structured programming, a concise and efficient implementation has been achieved without any subsequent loss of clarity. The decomposition has not followed any practical guidelines and each decision has been based largely on a knowledge of the use of certain programming constructs and schema to achieve a desired result. Recent work in the area of structured programming has been directed towards imposing some criteria on which to base this decision-making process. Current programming methodologies such as those of Jackson [Jackson 1975] and Warnier [Warnier 1974] propose structuring programs on the basis of the logical structure of the data, whereas Constantine [Yourdon and Constantine 1975] and Myers [Myers 1975] propose programs should be structured according to the functional decomposition of the problem.

An analysis conducted at the University of Aston amongst 85 students attempted to guage programmer's behaviour and attitudes to the design stage of program development. Each student was asked to complete a questionnaire and this together with the results obtained are given in Appendix F. The students represented a considerable variation in programming experience and knowledge, from novice programmers to those with several

```
begin   int  n, wage, fivepounds, poundnotes,
             fiftypences, tenpences, fivepences,
             twopences, onepences ;
        proc  denominations = (ref int numberof,
                               int value) void :
              begin  while  wage  >=  value
                     do     wage   :=  wage - value ;
                            numberof   :=  numberof
                                            + 1

                     od
              end ;
        read (n) ;
        fivepounds  :=  poundnotes  :=  fiftypences
        :=  tenpences  :=  fivepences  :=  twopences
        :=  onepences  :=  0 ;
        for  i  to  n
           do  read (wage) ;
               denominations (fivepounds, 500) ;
               denominations (poundnotes, 100) ;
               denominations (fiftypences, 50) ;
               denominations (tenpences, 10) ;
               denominations (fivepences, 5) ;
               denominations (twopences, 2) ;
               denominations (onepences, 1)
           od ;
        print (fivepounds, "fivepound notes are
                                required", newline,
              poundnotes, "onepound notes are
                                required", newline,
              fiftypences, "fiftypence coins are
                                 required", newline,
              tenpences, "tenpence coins are
                                 required", newline,
              fivepences, "fivepence coins are
                                 required", newline,
              twopences, "twopence coins are
                                 required", newline,
              onepences, "onepence coins are
                                 required", newline)
end
```

Diagram 4

The Coded Version of a Program Design

years programming experience. The novice programmers, that is those currently learning programming, formed the dominant group (62 students). The main conclusions drawn from an analysis of the questionnaires are:

a) 42 of the 62 novices do not write out a program design <u>every time</u> a program is developed;

b) 37 out of 61 students stated that for problems considered to be simple, program designs were not developed;

c) 53 students thought the time spent teaching them program design was adequate but 55 felt they would benefit from extra tuition. Furthermore 72 said they would take advantage of a system capable of analysing program designs;

d) 39 students found the program design stage more difficult than coding. Only 18 students thought coding was the more difficult and the remainder felt they were both equally difficult.

This latter result indicates that students find the formulation of program designs difficult and that they would benefit from any support that could be given to them during this stage. Such support would be important because a rigorous program design facilitates program development. Hence a system such as that proposed should prove beneficial because deficient program designs will be highlighted.

This thesis proposes a framework for analysing examples of program design which is referred to hereafter as the Framework for Analysing Program Designs (or FAPD). Since none of the criteria for decomposition, which are outlined above, have been universally accepted, examples of program design are often of widely differing forms. Because of this and because of time constraints, it has not been possible to investigate methods for analysing all of the different approaches to program design. Consequently before proposing a method, a decision is needed concerning the kind of program design which should be studied. The choice of this form is the subject of the following section.

1.3 Scope of Program Design for this Project

It was decided that attention should be concentrated on analysing program designs which have been written using an informal method similar to that used in section 1.2 . It was also decided that FAPD should aim to analyse program designs which use only a limited set of basic programming constructs. The reason for this is that because of time constraints it has not been possible to analyse program designs whose solution requires the use of a wide range of programming constructs. Consequently it was decided to concentrate on those designs which can be coded using suitable combinations of assignment, read and print statements, loops and conditionals and to omit more advanced programming concepts such as procedures. The implications of this omission are discussed in the final chapter.

In order to define more clearly the kinds of program design which this project should concentrate on, let us now consider how these basic programming constructs can be introduced to students who do not have prior knowledge of computing. At the University of Aston, first-year computer science students initially learn that programming consists of two related activities. The first of these involves understanding a problem and formulating a program design to solve the given problem. The second involves converting the design into a particular programming language. Students are taught to formulate a design in a manner suitable for conversion into a target language and consequently they are introduced to structures for denoting repetition and choice. These structures are identified as having the same format as those used in the target language. If ALGOL 68C was the programming language, then the structures would be identified as WHILE - DO - OD for repetition and IF - THEN - [ELSE -] FI for choice, where [ELSE -] represents an optional item. At each stage of the design process, the decisions available to the novice may be summarised as:

a) a sequence of actions

b) a selection of actions which is achieved using a conditional structure of the same format as that used in the target language; and

c) a repetition of actions which is achieved using a loop structure of the same format as that used in the target language,

where an action could be either a single instruction or a compound statement. Examples of single instructions are the arithmetic expression, the read, print or assignment statement.

After being taught how to formulate a design the student is then taught the coding details of ALGOL 68C such as the exact forms of the assignment, print and read statements together with other syntactic details such as the declaration of variables and the placement of semi-colons. With time and experience the student also becomes familiar with other constructs such as the CASE clause for denoting a special form of selection, the FOR loop as an alternative to the WHILE construct and data structures such as the array.

The program design in diagram 3 has been generated in order to illustrate how an experienced programmer might tackle the problem. Similarly the design in diagram 5 has been generated in order to illustrate the kind of program design which FAPD, described later in this thesis, can analyse. The latter diagram contains statements such as:

```
        initialise fivepounds  to  0        (1.1)
```

whereas the design in diagram 3 has specified the same instruction in terms of the target language, viz:

```
        fivepounds     := 0                 (1.2)
```

Statement (1.1) can be used instead of (1.2) when the programmer is inexperienced in using the syntactic features of the target language. Once the program design has been written in sufficient detail then the programmer need only concentrate on the coding details.

If we compare diagrams 3 and 4, the differences between the two can be described in terms of the decomposition. It has been determined that each of the calculations enclosed in the loopbody requires a loop structure and so the procedure facility of ALGOL 68C has

```
read the first number into n
initialise fivepounds to 0
initialise i to 1
while  i is less than or equal to n
do     read the next number into wage
       while  wage is greater than or equal to 500
       do     increment the value of fivepounds
                         by 1
              decrease the value of wage by 500
       od
       calculate the number of poundnotes
                 needed so far
       calculate the number of fiftypences
                 needed so far
       calculate the number of tenpences
                 needed so far
       calculate the number of fivepences
                 needed so far
       calculate the number of twopences
                 needed so far
       calculate the number of onepences
                 needed fo far
       increment i
od
output the number of coins and the number
                 of notes needed
```

Diagram 5

An Alternative Program Design

been used to collectively describe these calculations. Consequently this decomposition has resulted in a somewhat simple and efficient solution. However, if the programmer has no comprehension of advanced programming concepts such as a procedure, the stage following that shown in diagram 3 might merely show each of the remaining calculations decomposed into the appropriate loop structure. If the programmer has learnt the coding details of the target language then the design is now converted into code, otherwise the solution has been expressed as explicitly as his limited knowledge of programming has allowed.

This section is concluded by stating that the term "program design" is used throughout the remainder of this thesis to mean designing programs using the principles of structured programming in the manner already described. Also for the reason outlined at the beginning of this section, the Framework for Analysing Program Designs is aimed at analysing examples such as that shown in diagram 5 which can be coded using a limited set of target language constructs. By accepting designs similar to that shown in diagram 5 FAPD should be of benefit to programmers of varying experience. It is interesting to note from the questionnaire that 67 out of 84 students thought that the program design stage was necessary for all programmers whatever their experience. Nevertheless it is expected that novice programmers will derive the greatest benefit.

1.4 Analysing a Program Design

The concluding remarks of the previous section defined the term "program design" to be the process of designing a program according to the principles of structured programming. Having defined this term and shown how a program design can be produced according to these principles (see diagram 5) we must now consider how program designs of this type can be analysed. This project has taken the view that analysing a program design is a process of translating a design into an alternative format which can then be manipulated more easily than the original design. This format does not contain any of the ambiguities or inferences which may have existed in the original design because they will have been removed during the translation process. If any comments are generated during translation then the programmer can use them as a basis for revising the solution before finally submitting a coded version of the design to a computer for compilation and execution.

In terms of this project, a series of assertions has been chosen as the format into which a program design is translated. These assertions represent a coded version of the design and can then be used to produce a program together with any comments about its content. There are several reasons why this representation has been chosen.

Firstly, it provides a convenient format for showing a user if the process of designing the program is complete. If the process is <u>not</u> complete then the results show those statements in the design which have not been analysed and which require further refinement. Statements that have been analysed successfully are now expressed in terms of the target programming language and therefore need no further refinement. The design process is complete when all statements have been successfully analysed. The programmer then knows the design process is complete and any final modifications can be made before running the program on the computer. It is interesting to note that 36 out of 84 students who completed the questionnaire on program design usually wrote out a <u>single</u> program design before converting it into a programming language. This indicates that a process of stepwise refinement has not been followed and consequently the resulting program design could lack structure and detail. In this case, highlighting those statements which should be refined further will encourage students to spend more time on designing programs.

Secondly, this representation could prove particularly useful for novice programmers. Typically, a novice might have been taught the principles of program design prior to learning the coding details of the particular target language. FAPD could then be used within a system which takes the role of an experienced programmer who can show a novice how his design could be implemented. Any anomalies such as using variables without first initialising them, together with information on how statements in the design have been converted into code could be noted and commented upon.

A third reason for choosing this definition of analysing a program design is that FAPD could be used to act as a front-end to an existing system of program understanding. If FAPD is capable of producing a coded version of the design, the code could then be tested:

a) for syntactic correctness by using an existing compiler for the target programming language; and
b) by using some of the existing theories of program understanding.

Program understanding attempts to match a program's actual performance against its specification. Any discrepancies between the two show that the program, and hence the design from which it has been derived, is in error.

This section concludes the Introduction to the topic of program design analysis. Chapter 2 provides a discussion of some related AI work before the discussion returns to the Framework for Analysing Program Designs in Chapter 3. FAPD is described with reference to a system which is capable of analysing and commenting upon some simple program designs. Chapter 4 discusses details of the system's implementation and Chapters 5 and 6 analyse some of the results obtained from using the system. Chapter 7 concludes the thesis with an evaluation of this research together with some suggestions for further work.

2. RELATED AREAS OF ARTIFICIAL INTELLIGENCE

2.1 Program Verification

A method for analysing programs to determine whether or not they perform as intended has been a goal of computer science for many years. The initial work in this area came to be known as program verification. Program verification uses mathematical logic as the basis for analysis and attempts to prove the correctness of a program in a similar manner to the way a mathematical theorem is proved. The deficiencies of this area will now be discussed in order to illustrate the reasons behind the development of program understanding as an AI topic. Some of the approaches to program understanding are then discussed in Section 2.2.

A prerequisite of proving a program using this method is a specification of what the program is intended to achieve. This specification is represented by a series of assertions which describe the intended values of the program's output variables in terms of the program's input variables. Any restrictions on the program's inputs must also be represented in a similar manner. Because of its similarity to mathematical theorem proving, a theory of program verification often represents these assertions in a form based on first-order predicate logic.

In order to analyse a program other assertions must also be made to describe the values of variables at various points in the program. To determine whether a program performs as intended entails proving the truth of these assertions together with those describing the intended output values. Successive assertions are proved true by showing that a previous assertion together with the intervening code, imply the truth of the current assertions. If all assertions are proved true then the program has been successfully matched against its specification. The disadvantage of program verification is that it only proves whether or not a program performs as intended. It does not attempt to diagnose the cause of an error. This limitation has led to the growth of a related area of research which throughout this thesis is referred to as program understanding.

2.2 Program Understanding

The topic of program understanding will be described in terms of those research workers considered to have made major contributions to the topic.

2.2.1 Katz and Manna

As we stated in the previous section many systems which attempt to verify a program are inadequate since they do not diagnose the cause of errors in incorrect programs. However a further disadvantage is that the system user must provide not only those assertions describing the program's output values, but also the intermediate inductive assertions. Katz and Manna [Katz and Manna 1976] have suggested a unified solution to these problems and have proposed that the analysis of a program should be based on what is actually occurring in the program rather than some theoretical specification. Whenever a system of program verification fails to prove a program it is unclear whether the code is bugged or the system is unable to produce a correct proof. Hence Katz and Manna have suggested that program analysis should be based on, what they call, invariant assertions. These are used to express the actual relationships among the variables of the program and are derived directly from the program text rather than from a separate definition given by the programmer. Consequently these invariants are independent of the program's output specification and can be used either to verify that the program performs as intended or that it is bugged. In the latter case, the same invariant assertions can then be used to locate the errors and modify the program.

To eliminate erroneous code two approaches have been advocated. The first has been termed a conservative approach and means that the program must be proved incorrect before it can be modified. The second approach which is more radical modifies the program regardless of its state of correctness. This means a correct program is often modified and its efficiency may be reduced as a result. However this approach is of merit since modification guarantees a proof of correctness.

Whichever approach is chosen, the basic technique of debugging is the same. This technique modifies a program systematically by using the invariants together with information about how they were generated. This information is stored in the form of an invariant table which contains everything used to establish each variant such as the rule applied and precisely how the program statements and/or other variants were used in its derivation. Debugging proceeds by walking through this invariant table, proposing and testing new variants which have been generated as candidates that could lead to the program being proved correct.

Although the discussion above is based on a set of proposals which have not been implemented, this work is of significance since it demonstrates the inadequacies of program verification and has put forward some proposals for overcoming them. Many of the other theories, outlined in this section, stress the importance of building a rich description of how the program can be analysed. This description often performs a similar function to the invariant table discussed above and is used in a similar way to aid the debugging process.

2.2.2 Goldstein

Goldstein [Goldstein 1975] discusses a system called MYCROFT for debugging simple LOGO programs. The input to MYCROFT is a bugged LOGO program together with a model which uses pre-defined geometric predicates to describe the intended outcome of that program. MYCROFT analyses the program and builds a description of the picture actually drawn and a plan explaining the relationship between the program and model. This plan allows MYCROFT to bind sub-pictures to model parts and to produce a list of violated model statements. The debugger then attempts to repair each violation in the list in order to produce an edited program which satisfies the model.

The first operation that MYCROFT undertakes is to document how the program performs. This documentation is organised as sets of assertions in a database bound together with sequences representing what happened and why. There are three kinds of documentation which may be summarised as:

a) process annotation which records the effects of executing each program statement. This annotation is generated by imperative semantics associated with each LOGO primitive;
b) planning advice which tries to find clues on how the program can be segmented. In this respect MYCROFT views a program as comprising main steps (which are represented by the code required to achieve a particular goal) and prepatory steps (which are the interfaces between main steps);
c) debugging advice which describes suspicious code within the program such as sequences of contiguous uses of the same primitive.

The second operation within MYCROFT is to find the plan. The plan finder assumes a linear structure to the user's plan and attempts to match model parts with modular main steps and relations between model parts with prepatory steps. The result of this matching operation is a list of violated model predicates.

The final operation is a debugging operation and involves correcting these violations. To achieve this the debugger uses two types of procedural knowledge. The first of these is a collection of general debugging strategies which use a linear attack as they try to repair a program. The first step in debugging is to fix each main step independently. Following this the main steps are treated as inviolate and the relations between model parts are fixed by debugging prepatory steps. MYCROFT will also use comments generated by the plan finder to suggest the location of repairs and it will compare alternative debugging strategies in an attempt to choose those which will cause minimal change to the user's code. The second type of procedural knowledge used by the debugger is concerned with giving directions for fixing particular geometric and logical predicates.

Goldstein's work is of significance for showing how the concept of linearity together with rich program descriptions facilitate understanding and debugging. However the two main criticisms of his theory are:

a) the subset of LOGO used is too restrictive; and
b) the model used to specify the intended effect of a program is very detailed and often more complex than the program it describes.

The Framework for Analysing Program Designs is similar to Goldstein's work since they both represent some of the results of analysis in the form of assertions stored in the database. Goldstein's work is also of relevance to the author's since MYCROFT does not use the model of intended outcome in order to document how a program performs. This illustrates that some useful information about a program can be derived without necessarily knowing what that program is intended to achieve.

2.2.3 Ruth

Ruth [Ruth 1976] was concerned with various implementations of a known algorithm. His theory of intelligent program analysis is based on a knowledge of what must be accomplished and how code is used to express intentions. This theory has been implemented in a system, written in the AI programming language CONNIVER, which analyses a program by using a description of the task the program is to accomplish (c.f. Goldstein's model of intended outcome), which the user provides, together with a built-in body of knowledge of how intentions can be realised in code. The system's knowledge is in the form of programming experts which know how actions can be coded and organised and what the common sources of errors in program writing are.

The user provided description of the program task must be pre-defined using constructs and mechanisms (i.e. loops and conditionals) in a form which the analyser can recognise. The analyser knows how these constructs and mechanisms can be re-arranged and reorganised to produce equivalent variations and how they can be coded. The user can then type in a program, which must be written in a simple LISP-like language, for analysis. If the program is correct but the system cannot match it against the pre-defined description, it will be either misunderstood or not understood at all.

The pre-defined description and the program both comprise a list of actions and analysis is concerned with matching the two lists. This analysis is undertaken by an action list matcher(ALM) which will continue operating until there is a failure or the list of actions in the pre-defined description has been exhausted. For an action in the description to be matched with an action in the program they must be equivalent not only in terms of their values but also in terms of the constructs they use. To do this the system has an expert for each action that can be used in the predefined description. An expert checks whether the current action that the ALM is trying to match is present and properly implemented at the current point in the code. If it is not, then an error is reported. Errors are classified as either recoverable or non-recoverable. The analyser has specific knowledge of a few common programming errors which it can recognise and fix. These are termed recoverable errors because they can be fixed without substantial change to the observed code. Generally speaking, non-recoverable errors are those where something vital is missing or something unwanted is present.

Although Ruth's work is impressive, an important drawback is that analysis concentrates on a description of the values of the variables. Later research [Lukey 1980] has shown that other types of description can provide useful aids to understanding. However Ruth's work is of relevance because it shows how recognition of various schema can contribute to program understanding.

The framework described in this thesis proposes that the translation of a program design statement into a target language can be achieved using a procedure called a class instance. In this respect class instances are similar to Ruth's experts except that an expert is called on the basis of the actions contained in the predefined description of a program task, whereas a class instance is called on the basis of what appears in a program design. It should also be noted how they are used for different purposes. An expert is used to determine whether or not an action has an equivalent form in the program, whereas a class instance is used to create a coded version of a statement or phrase.

2.2.4 Lukey

Lukey [Lukey 1980] has developed a system, called PUDSY, which can understand and debug some simple PASCAL (sub-) programs. He distinguishes between two types of debugging. The first is based on recognising general constraints on correct and rational programs. An error typical of this kind is a loop which will never terminate. The second type is based on a comparison of a program's intended and actual operation. The input to PUDSY is a PASCAL program together with a formal specification of its intended outcome. The system will then build up a description of how the program actually operates and matches this against its specification. Any discrepancy between the two indicates the program is bugged. The code is then edited by identifying and generating a specification for the piece of code responsible.

Lukey emphasises how the success of his debugging strategy depends to a large extent on the availability of a rich program description. In this respect the process of understanding a program involves:

a) segmenting a program;

b) describing its flow of information;

c) describing the values of variables; and

d) recognising debugging clues.

The first step in this process is to segment the program into distinct units, which Lukey calls chunks. Once this has been achieved PUDSY will then specify how these chunks communicate with each other. This involves identifying those variables whose values have been used in, but which were determined prior to, the current chunk. These are known as a chunk's inputs. Similarly, a chunk's outputs are those variables whose values are used by subsequent chunks or which are returned to the main body of a program as either the value of the subprogram or the value of a parameter. The second type of program description is based on the analysis of the inputs and outputs and is a high-level description of how information flows from one chunk to another.

The segmentation of a program together with the description of information flow provides a framework for the third type of program description which describes the values of a program's variables. Each chunk may now be described by making assertions about its output variables. These assertions describe the values held by the output variables, in terms of the input variables, when control leaves the chunk. To do this two methods are used. The first method involves the recognition of a particular series of statements followed by their description. The second method uses a technique of symbolic evaluation in order to derive the necessary assertions.

The fourth type of program description involves a recognition of debugging clues. For instance, the way in which a variable is intended to be used in a program could possibly be determined from its name. For example PUDSY makes a note of a variable named COUNT if it is not used to count anything. By comparing a program's specification with its description, a list of mismatches can also be produced and by tracing a path back through the assertions which it has produced, PUDSY identifies the code source of a mismatch. Once this has been done a series of edits are proposed and tested and the most successful of these is chosen. Finally the consequences of an edit are tested to ensure that it has removed the bug.

Lukey's work is impressive because he has demonstrated that to understand a program, other types of description, in addition to the values of variables are useful. He has also shown the importance of these different types interacting. However, he does point out that to a large extent this method of description is also inadequate since it does not make use of some potentially useful sources of information such as, for example, input and output pairs, information derived from execution errors or traces of a program's execution.

2.2.5 Rich and Shrobe

Rich and Shrobe [Rich and Shrobe 1978] have developed a system which plays the role of a programmer's apprentice for expert programmers who are writing LISP programs to manipulate hash tables. These programs are described by the system in terms of the hash tables on which they operate, the input and output specifications of the segments which comprise the program and the hierarchical representation of the program's internal structure. The latter of these descriptions is referred to as the plan.

The first type of description is concerned with hash tables which in effect form the data for a program and which the user must describe in terms of the abstract definition known to the system. The second form of description is represented by the input and output specifications of the program's segments and is supplied by the programmer. In terms of code, a program segment could be, for instance, a function definition, the body of a conditional or several lines of open code. A segment is described by a series of specifications which contain information about the data flowing into and out of the segment. These specifications are a formal statement of the conditions acting upon or the relationships between, values of the data at the time the segment is entered. A segment's output values are also described in a similar manner.

One of the most interesting aspects of this work is the third form of program description, known as the plan. Rich and Shrobe have devised a method of representing plans which allows them to be used not only for describing a user's program but also for describing the system's programming knowledge. The programmer and apprentice first work at this plan level and interact in order to develop an abstract representation of the program's intended structure. To do this the apprentice must know some of the basic techniques for manipulating hash tables such as deleting elements from a linked list. The apprentice can now compare the segment specifications with the plan and the user can modify it if any errors are found. When the segment specifications are found to be consistent, the user can type in the code and the apprentice ensures that it conforms to the predefined plan.

In order to describe the structure of a program the apprentice uses two kinds of plan. The first is called a surface plan and describes the flow of control and of data between various parts of the program. The second is referred to as the deep plan which shows how a program operates and whereas the surface plan is explicitly stated in a program, the deep plan is not. In order to understand a program the apprentice makes use of the code, the surface plan and the deep plan. To establish whether or not the code fits the plan, the apprentice first uses the program to derive the surface plan and then compares it with the deep plan by using its general programming knowledge. A deep plan is expressed in terms of purpose links which describe the logical structure of a program. Consequently if a programmer attempts to modify a program, it is the purpose links that denote which of the other segments will be affected and in what ways. These links are also used when a surface plan segment is matched against a deep plan segment. The apprentice declares the two forms of plan are equal, only if the data and control flow links surrounding the surface plan segment are consistent with the data flow and purpose links surrounding the deep plan segment.

In conclusion we can say that the work of Rich and Shrobe has made a significant contribution to automatic program understanding. Their work is significant not only because certain aspects of it have been implemented in a system, whereas some other studies have not, but also because their notion of a deep plan shows how they have confronted the problem of finding a suitable representation for programming knowledge. However a disadvantage of their proposals is that the user must still supply some of the information required for analysis. The user's task would be simplified if information about the deep plan or about the input and output values of the program segments, at the level of detail required by the present system, did not have to be supplied.

2.2.6 Waters

The work of Waters [Waters 1976, Waters 1978, Waters 1979, Waters 1982] has close links with that described in the previous section. Rich and Shrobe have laid out the initial design for a programmer's apprentice and have developed the concept of a plan for representing programming knowledge. Waters [Waters 1976] has designed a limited system aimed at the area of mathematical FORTRAN programs and has extended the notion of a plan by proposing how it could be segmented. Recently Waters [Waters 1982] has also produced an initial implementation of this programmer's apprentice. The following discussion will concentrate only on this implementation because, in terms of this study, it is the most relevant aspect of his work.

The programmer's apprentice (PA) which has been implemented is comprised of five parts. These are an analyser which constructs the plans relating to a program, a coder which converts a plan into a program, a drawer which converts a plan into a graphical representation, a library of plans and a special plan editor which allows the plan rather than the program to be edited. From this we can see that the concept of a plan is central to this implementation. Indeed, one of the most significant aspects of this work has been the use of plans to represent not only programs but also programming knowledge.

The implementation of the PA is in the form of an editor which allows a user to build up a program and then edit its plan. To build a program the user types in commands requesting the PA to undertake operations such as the definition of a procedure for which the user has provided an appropriate name. The procedure body can then be filled in by using phrases such as "successive refinement" to indicate that the result is calculated using a loop construct. In terms of the PA's components described above, the plan relating to this phrase is stored in the library and the coder knows how this plan can be represented in a programming language.

Having built up a program by using library plans together with actual pieces of code, the user can then edit his program by modifying its structure. To do this a programmer must use the system provided vocabulary to refer to plans and parts of a plan. Once a plan is modified the coder can then be called to translate the plan into code. However a user may sometimes use the normal text editor instead of the plan editor. In this situation the PA is used in the opposite sense and the analyser is called to determine the form of the resulting plan.

So far the discussion has concentrated on what the PA is capable of analysing, however as Waters points out, there are three areas of which it has no comprehension. Firstly, the PA uses no description to aid analysis, unlike the apprentice of Rich and Shrobe which uses a description of a hash table to aid analysis. Secondly it does not have any knowledge of the program specification and thirdly it is not capable of recognising that library plans may be inter-related.

There are two aspects of the PA which are relevant to the research described in this thesis. Firstly, both areas of research are concerned with analysing statements in terms of a programming language. However, there is considerable difference in the way this knowledge is used. Waters is concerned with creating and modifying an abstract specification of a program whereas our study is concerned with producing a coded version of the design. In this respect, the work of Waters is more ambitious since it directly attacks how programming knowledge can be represented, independently of the target language. Also this research is concerned with automatically detecting any anomalies in a program, whereas the PA leaves this task to the programmer. Secondly, when editing a plan, the PA allows the programmer to use the pronoun "it" to refer to the object which is the current focus of the system's attention. This research uses a similar approach to deal with any pronominal references found in a program design.

The work of Waters, together with that of Rich and Shrobe represents some of the most significant research in this area at the present time. Their objective of finding a suitable representation for programs and programming knowledge and the implementation of that representation dictates that the project is long-term. Nevertheless their work provides some justification for believing that future systems will be capable of providing some sinificant programming support.

2.2.7 Others

Let us now conclude the discussion of program understanding by referring to the research undertaken by Smith and Hewitt [Smith and Hewitt 1974], Miller [Miller 1978 , Ramsay [Ramsay 1980] and Eisenstadt and Laubsch [Eisenstadt and Laubsch 1980]. These are discussed in chronological order.

Smith and Hewitt have put forward proposals for a programmer's apprentice which are designed to work within the area of Hewitt's ACTORS formalism [Hewitt, Bishop and Steiger 1973]. Their aim is to develop an apprentice which can assist the programmer in tasks such as formulating and maintaining the consistency of specifications and ensuring that the modules which comprise a program perform as intended. It is also envisaged that the apprentice will be able to answer questions about the relationships between modules.

In order to verify a program, Smith and Hewitt have developed a technique which they have called meta-evaluation. This technique is based on the process which a programmer goes through when he symbolically executes his program to see if it works. These proposals have the disadvantage that analysis of a program depends, to a large extent, on the specifications provided by the user, and analysis does not produce any detailed descriptions of its own.

Miller's work is worth mentioning since it aims to understand both the planning and debugging processes. This work is discussed by Miller within the context of a system, called SPADE-0, which interacts with programmers who are planning and debugging programs written in the LOGO programming language. SPADE-0 leads a programmer through a hierarchical planning process by providing a vocabulary of concepts for describing plans, bugs and debugging techniques. The system represents the planning process in terms of a tree-like structure. The system user is shown this structure so that he can identify the alternative paths that can be followed in order to produce a program, SPADE-0 will then lead its user through these paths by choosing the next likely goal. As the tree is traversed, the system leaves messages on the various paths which may be used later on to guide the debugging process.

Some of the bugs which Miller has identified are based on his adopted theory of planning. Thus, a pragmatic bug is defined as an incorrect choice of path in the planning tree. Conversely, a semantic bug is where the picture produced by the LOGO program is not the intended one. Most of the other research studies into program understanding have derived the information necessary for debugging from the program. Consequently Miller's work is of interest since some of the information used by SPADE-0 is derived from another source, namely its record of those decisions taken by the programmer during the planning process.

Eisenstadt and Laubsch have discussed their work on a debugging assistant. The assistant is intended to help students who are using the programming language SOLO, which has primitives similar to MICRO-PLANNER, for operating on assertions in a data base. Students use the assistant when problems arise which need to be solved. The assistant is comprised of four modules which are referred to as the intent-specifier, the instantiator, the coder and the translator.

The intent-specifier is used to determine what the code is supposed to achieve. To produce these intentions the intent-specifier uses a plan library which is comprised of high and low level plans. Low level plans denote how a general operation, such as an assignment, can be achieved whereas a higher level plan is used to denote operations which are relevant to the particular problem the student is working on. The intentions produced by the intent-specifier are then used by the instantiator in order to propose several possible plans for execution. How these plans can be implemented in SOLO is known to the coder. However the results of its analysis are not in the actual form required by the SOLO syntax, but instead they are expressed in a conceptual form suitable for execution by a SOLO virtual machine. The fourth module is the translator which takes the student's code and translates it into a form which can be compared directly with the abstract plan. This comparison is based on symbolic evaluation and shows why a piece of code has failed. If the assistant has a model of what the code is intended to achieve then the student can be shown examples of a correct implementation.

The work of Eisenstadt and Laubsch is of interest since it is concerned with using both domain independent program understanders as used by Rich and Shrobe and Lukey (see sections 2.2.5 and 2.2.4) as well as expert debuggers such as those used by Ruth (see section 2.2.3). A third aspect which has also been emphasised is that the assistant should provide a friendly user interface. Since the research described in this thesis has also been implemented in an interactive system, this third aspect is also a goal in developing the Framework for Analysing Program Designs.

Finally, let us consider the work of Ramsay, who has developed a system, called SH4, which matches a LISP program against an English description of what it is supposed to achieve. SH4 has two sets of data to analyse - the description and the program. Each is analysed and the two sets of records which are produced can then be matched against each other to see if the program performs as expected. The system then makes a copy of the program and uses the English description to insert comments into that program. A set of flow charts representing the procedures which have been described, together with fragments of code showing how these procedures have been implemented, is also produced.

Using a piece of English text to describe the program means that after producing the two sets of records neither the normal techniques of program verification, nor symbolic evaluation can be used, since it is unclear which parts of the program fit which specification. Ramsay has tackled this problem by using hypothesisers to suggest links between the program and its text. Once these links have been established symbolic evaluation can be used to verify the program.

Many of the existing theories of program understanding rely on proving assertions at various points throughout the program. However, these theories require that the intended outcome of a program should be specified in a formal manner. Consequently this specification is awkward to define and error-prone. Ramsay's work is important since it has shown how a less formal program specification can be used which, from a system user's point of view, is the preferred approach. However as

Ramsay admits, the descriptions on which SH4 operates are too detailed for the system to be a practical tool at present.

2.3 Automatic Programming

The area of automatic programming is concerned with developing a system which can generate a program from a formal specification of what the program is intended to achieve. Within this area, the work of Sussman [Sussman 1975] and his system HACKER, have received a great deal of attention in the AI literature. HACKER inhabits the same world as Winograd's SHRDLU [Winograd 1972] and writes programs containing instructions which undertake primitive operations such as picking up a block.

If the first program, which has been produced to solve a given problem, is not totally correct then an iterative procedure, aimed at locating and eliminating all bugs, is entered. Whenever a bug is found, HACKER tries to classify it, so that a similar error can be avoided in the future. For instance, if HACKER is asked to pick up a block which is currently supporting another, it is not able to determine that the uppermost block must be removed before the lower one can be accessed and as a result the program which it produces to undertake this operation will be bugged. However this error is then analysed in general terms so that in the future, a similar or identical situation will not lead to the same error being committed again.

The debugging process is based on a detailed purposive commentary which HACKER uses to denote the intended outcome of each section of code. The first time a program is run, this commentary is used to check if the code performs as specified and any bugs are classified according to the five categories of error defined for HACKER. The work of Sussman is significant, not only for its contribution to the area of automatic programming, but also because it is relevant to AI theories of program understanding and debugging and skill-learning.

In recent years an automatic programming system, called PSI [C. Green 1976, C. Green 1977] has been developed by a research team at Stanford University. Green and Barstow [Green and Barstow 1978] who are part of this team, have emphasised how their work is concerned primarily with the organisation and structure of programming knowledge which can be used by a computer to write programs. The PSI system contains knowledge in the form of approximately 400 rules and is comprised of two phases: an acquisition phase and a synthesis phase. The acquisition phase is concerned with finding out, from the user, what the program is intended to achieve and building a high level model of this intention. The synthesis phase uses a coder, written by Barstow, and an effeciency expert written by Kant [Kant 1977] which combine in order to produce an efficient program. The rules which the coder uses are sufficiently general for them to be used in various domains such as symbolic programming, sorting, graph theory and simple number theory. The coder writes programs in LISP and although some of the rules are specific to this language, approximately three-quarters of them are independent of any programming language.

Manna and Waldinger [Manna and Waldinger 1975] have claimed that an automatic program synthesis system must combine reasoning and programming ability with a good deal of knowledge about the subject matter of the program. This approach towards program synthesis is the method on which HACKER and PSI are based. Despite the claims of Manna and Waldinger, Bauer [Bauer 1979] has attempted to show that some useful analysis can still be undertaken without knowing the subject matter of the program. Bauer has developed a program which can synthesise procedures for computations such as, for example, multiplying two numbers using repeated addition or sorting the values held in an array. Since it does not use a problem specification its analysis is based upon a knowledge of variables and parameters and their general use.

Finally let us briefly consider the work of Koffman and Blount [Koffman and Blount 1975] who have embodied a method of automatic programming within a teaching system. This system teaches machine language programming and represents all problems given to a user in terms of an AND/OR goal tree. This tree represents a complex problem in terms of three sub-problems which are referred to as the input, processing and output phases. The system represents each sub-problem as a sequence of primitive tasks for which it can generate alternative forms of machine code. This means that either a user can be supplied with the code for the simpler sub-problems so that attention can be diverted to more difficult areas or, each of a user's statements can be checked against those produced by the system.

Unfortunately the power of the system is limited since it can produce code only for the primitive tasks and although there may be more ways of solving a problem the user must follow a similar solution to that defined by the system.

In terms of the research discussed in this section the work on the PSI system would seem to hold the best prospects for the future. It is interesting to note how long term projects such as this and the programmer's apprentice of Waters (see section 2.2.6) are both concerned with finding a suitable representation for programming knowledge. Since PSI is an automatic programming system and the programmer's apprentice is concerned with program understanding and debugging it would seem that research into the representation of programming knowledge could benefit those areas of AI which are concerned with understanding different aspects of the programming process.

2.4 Intelligent Teaching Systems

Because the Framework for Analysing Program Designs has been incorporated within an interactive system, a discussion of how AI techniques can be applied to the area of computer assisted instruction (CAI) is relevant to this chapter. Embedded within an intelligent teaching system is a coach which may perform all, or a subset of the following:

a) checking a student's answer;

b) generating meaningful error messages;

c) providing "hints" on how to solve a problem when the student requests help;

d) providing a model solution to a problem; and/or

e) updating a model of a student's knowledge.

At the present time systems have been developed which incorporate coaching to teach basic mathematical skills [Burton and Brown 1979], basic reasoning techniques [Goldstein 1979], electronic trouble shooting [Brown, Burton and Bell 1975, Brown, Burton and de Kleer 1982] the solution of quadratic equations [O'Shea 1978] and medical diagnosis [Clancey 1979]. There are four principal features of these systems.

Firstly they have an expert embedded within the system which can solve problems in the given domain. As a result there is no need to store a data base of model solutions. Secondly, all problems given to the student, together with the answers to these problems and the student's state of knowledge are all defined in terms of a fundamental set of skills. Hence if the expert is asked to solve the same problem as the student, then the two answers can be analysed in terms of the same skills. A comparison of the two will now show which skills the expert used and the student did not. Such an analysis highlights those techniques in which the student is deficient and since the problems are also defined in terms of the same skills, the next problem can be chosen in order to give practice in the areas of weakness. Goldstein [Goldstein 1979] not only analyses his subject area into an underlying set of skills but he also sees each skill going through five phases of development and refinement as the student becomes more competent.

A third feature of an intelligent teaching system is that answers are not assessed for correctness but are analysed in terms of whether the appropriate skills have been used. Consequently error messages can emphasise the techniques which an expert would have used in the same situation. The fourth feature is that any hints given to the student can be based on an expert's approach to solving the problem. This highlights one of the main disadvantages of the expert known as SOPHIE [Brown, Burton and Bell 1975] which has been called a "black box" because it does not solve a problem in the same way a student is expected to. Because the underlying mechanisms which the expert used were not passed on to the student, subsequent versions of SOPHIE [Brown, Burton and de Kleer 1982] have aimed to use inference techniques similar to those used by students.

In terms of the research described in this thesis, one objective has been to develop a system which possesses the first of these four features, that is a system which displays expertise in the subject area of analysing and commenting upon a program design. The form of analysis which FAPD undertakes allows the generation of some meaningful error messages and in this respect FAPD does not mark any program designs as merely right or wrong but instead undertakes a deeper analysis. However, FAPD does not analyse a program design in terms of a fundamental set of skills.

Barr et al [Barr, Beard and Atkinson 1976] have developed a system for teaching introductory programming techniques in BASIC. The curriculum used by their system has one hundred different programming problems which are defined in terms of skills such as printing a literal string or using a counter variable in a loop. The student is also modelled in terms of the skills he has acquired and consequently a problem can be selected on the basis of how that student has performed on earlier problems. Once the program has been written, data can then be used to test the program. The program is also checked for the BASIC statements that should have been used. For example a problem might have been chosen to teach the FOR statement, and so the checker analyses the answer to determine if this has been included. If it has not then a suitable error message is printed. In common with the attributes of an intelligent coach described earlier in this section, this system can also provide the student with useful hints on how to solve a problem.

Generally speaking, defining the programming process in terms of a fundamental set of skills is a research topic which is growing in importance. Consequently the work of experimental psychologists such as Green [T.R.Green 1977] who has investigated techniques for measuring how well a program has been understood, could be used in intelligent CAI systems which teach programming. The growing interest in applying AI techniques to CAI systems together with experimental work such as that just described indicate that research into these systems could increase significantly in the future.

2.5 Computational Linguistics

There has been a considerable amount of research into computational linguistics. Although numerous natural language question answering systems have been developed (for a review of the entire field see Bruce [Bruce 1975]), this discussion will concentrate on the work of Burton [Burton 1976] since it is considered most relevant to the problems with which we are concerned.

Burton discusses a paradigm for constructing efficient, friendly man-machine interface systems using subsets of natural language in limited domains. The primary purpose of his work was to develop a set of techniques for embedding semantic and pragmatic information into a natural language interface module. The techniques were implemented in the "intelligent" CAI system SOPHIE [Brown et al 1975, Brown et al 1982], which is a reactive learning environment concerned with electronic troubleshooting. In a typical troubleshooting session the student is confronted with an electronic circuit containing a fault. The student can then interrogate SOPHIE in an effort to locate the fault.

The natural language subset which SOPHIE accepts is described by a "semantic grammar". A semantic grammar is so-called because it specifies relationships in both semantic/conceptual and syntactic terms. It has two advantages over syntactic grammars. Firstly, semantic constraints can be used to make predictions during the parsing process which reduces both the number of alternatives which must be checked and the amount of syntactic (grammatical) ambiguity. It also allows the parser to skip words at controlled places in the input and ellipsed or deleted phrases to be recognised. Secondly, a semantic grammar can be used to characterise those sentences which the system should try to handle.

Because the grammar is based on conceptual entities, semantic interpretation can proceed in parallel with parsing. Each rule in the grammar characterises all of the ways of expressing a concept or relationship in terms of other constituent concepts. Thus the rule for <MEASUREMENT> is:

<MEASUREMENT> := <MEASURABLE/QUANTITY><PREP> <PART>

which defines all the ways a student can express a measurable quantity. Rules of this type allow similar concepts to be generalised and so voltage, current, resistance and power for example would each be termed a <MEASURABLE/QUANTITY>. This is similar to the method adopted in this research whereby words such as ASSIGN, CALCULATE, DECREASE and FIND are all defined as <assignment command word>'s. They all have the same definition because their occurrence in a design statement indicates the statement can be implemented as an assignment statement.

One use of a semantic grammar is to predict possible alternatives that must be checked. The <MEASUREMENT> rule for example, can be used in conjunction with the phrase "the voltage at it" to restrict the possible interpretations of "it" to locations such as nodes and terminals. A second use of the semantic grammar is to recognise simple deletions. When the grammar finds the phrase "the collector" it uses the fact that the concept of a TERMINAL has constituent concepts of TERMINAL-TYPE and a PART to deduce that a PART has been deleted. Because the dependencies between the constituent parts determine that the deleted PART must be a transistor, the meaning of the phrase is then "the collector of some transistor". Which transistor is determined when the meaning is evaluated in the present dialogue context. Thirdly, the semantic grammar can be used to overcome the problem of ellipsis. In the following example:

What is the voltage at node 5 ? (2.1)

At node 1 ? (2.2)

At node 2 ? (2.3)

What about between nodes 7 and 8 ? (2.4)

(2.2), (2.3) and (2.4) are elliptic utterances because they do not express complete thoughts but only give differences between the intended thought and (2.1). The appropriate grammar rule can be used with these examples to identify which concept is possible given the current context.

Once the parser has determined the existence and class of a pronoun/deleted object, the context mechanism is invoked. This mechanism uses the meaning of the student's previous statements and the response calculated by the system to determine the proper referent. The context mechanism also knows how each procedural specialist appearing in the parse uses its arguments. For example, the specialist MEASURE's first argument must be a quantity and the second argument a part, junction, section, terminal or node. Thus when the context mechanism looks for a referent which can be either a PART or a JUNCTION it will look at the second argument only of MEASURE.

The problem of ellipsis is concerned with finding a previously mentioned use for a currently specified object. In the example:

What is the base current of Q4 ? (2.5)

In Q5 ? (2.6)

the given object is "Q5" and the earlier function is "base current". Since Q5 is recognised by the non-terminal <TRANSISTOR/SPEC>, the context mechanism searches for a specialist in a previous parse which accepted the given class as an argument. When one is found, the new phrase is substituted into the proper argument position and the substituted meaning is used as the meaning of the ellipsis. This research has also been concerned with how the context of a statement or phrase can help to determine its meaning. Section 5.3 discusses how the word RESULT for example, cannot be analysed in isolation but must be considered in its wider context.

Burton's work is important because it shows how a semantic grammar provides a paradigm for organising knowledge required for understanding. If a system does not encompass a useable subset of the language a student must expend problem solving energies discovering how to formulate questions. A semantic grammar helps to overcome this problem by providing insights into a useful class of dialogue constructs. Burton has also shown that it can permit efficient handling of pronomalisations and ellipsis. However the work does have limitations. Firstly, the context mechanism works well in the given domain but does not solve all the problems of reference since Charniak [Charniak 1972] has shown how much real world knowledge is sometimes required. The major limitation of the current technique is its inability to return more than one possible referent. At present it considers each in turn until it finds one satisfactory. Secondly, as Burton admits, the primary goal was to develop a useful system and as such the research does not advance our theoretical understanding of natural language.

2.6 Programming Languages for Novice Programmers

Kreitsberg and Swanson [Kreitsberg and Swanson 1974] describe the "computer shock" which novice programmers may encounter when faced with the problem of planning an algorithm. Novices have problems in understanding what a program can do for them and its relation to the problem which they are trying to solve. Miller [Miller 1975] found that when specifying a plan to a human being the specification was "qualificational" rather than "conditional". Thus to a human being we might say "PUT RED THINGS IN BOX 1" whereas a computer program must specify "IF THING IS RED THEN PUT IN BOX 1". In this respect a programming language such as PROLOG [Pereira, Pereira and Warren 1979] might have advantages for novices. This is because PROLOG specifies plans in terms of goals rather than in terms of an algorithm.

Novices find specifying the flow of control very difficult [du Boulay and O'Shea 1980]. Because this is central to programming in algorithmic languages it may be beneficial to implement programming languages so that certain hidden actions are accompanied by external changes. Flow of control within the BIP system [Barr et al 1976] is made visible by showing pointers which move around the program text as it is executed. Similarly Mayer [Mayer 1979] represents the workings of a BASIC machine in terms of a small set of "transactions" where a transaction consists of an "operation", an "object" and a "location". The transactions explain the sequence of events while a BASIC program is running and are simple enough to be understood by a novice.

Du Boulay and O'Shea also describe three languages designed specifically for novices. The first is SOLO [Eisenstadt 1978] . which is a language for manipulating a relational database. User defined procedures can invoke primitives which add, remove, print etc database structures. Because the students had no prior knowledge of computing, the software enviroment had to be non-threatening [Eisenstadt 1983]. To achieve this SOLO was designed so that students could quickly use it to undertake powerful operations. Hence, although the language has only ten primitives, these are sufficiently powerful for beginners to do interesting projects. The English meanings of primitive names such as NOTE, FORGET and DESCRIBE correspond closely to the actual jobs they perform within the SOLO virtual machine. This is similar to the way in which words such as GET, OUTPUT and INCREMENT are used to specify actions within a program design (see Chapter 3). Functional simplicity was achieved in SOLO by restricting the scope of the database searching mechanism and by delaying the introduction of certain language features until the novice had progressed to a given point. Syntactic simplicity was increased by arranging that whenever a student typed the IF part of a conditional, the system would issue prompts for both the THEN and the ELSE part. Sime et al [Sime, Arblaster and Green 1977] has shown that this is a successful method of reducing errors in conditionals. The visibility of the language is enhanced by presenting database items at the terminal in a form that both suggests the meaning of the item and is in agreement with the teaching material.

The second language is a microprocessor based assembly language. The system (based on the Intel 8049) provides only ten instructions: LOAD, STORE, ADD, DECREMENT, JUMP, JUMP IF ZERO, INPUT, OUTPUT, CALL and EXCLUSIVE OR. The system also contains a number of predefined subroutines that can be called by the user's program and whose instructions can be examined although the code for the interpreter itself is inaccessible. These subroutines illustrate the idea of program modularity. The functional simplicity of the notional machine is achieved at the expense of having a complicated program interpreting the user's key presses. The facility to examine the code of the subroutines is one step towards language visibility although it is accepted that visibility could be improved. Despite these restrictions, the work is important since it allows the user to be introduced to a wide range of computing ideas including planning, coding, running and debugging programs and flow of control.

The third programming language developed for novices is ELOGO. This is a procedural, interactive language with facilities for drawing using a turtle and for symbol manipulation using integers, words and lists as data types [McArthur 1974]. A user's initial introduction to programming is via a buttonbox and a turtle, where each button represents an instruction. Thus labels on a button correspond to what the novice must type when a teletype is used. This simple notional machine implied by the button box and the turtle provides a foundation to build the user's understanding of the complete ELOGO system implemented on the mainframe. The main task for the system's users is the interactive definition, testing and debugging of procedures. A novice decomposes a complex task into simpler sub-tasks which may also need further decomposition. Because the basic programming unit is the procedure, the notional machine is functionally simple. In an effort to increase language visibility hidden actions such as storing a procedure are concluded with a written comment from the system.

Languages such as those described above are important for two reasons:

a) they allow the novice to start writing and running programs very quickly, which helps to sustain interest; and

b) they embody facilities for making certain of the actions of the notional machine open to view.

In terms of the research described in this thesis, the first of these reasons was a primary consideration in choosing an appropriate program design language.

3. THE FRAMEWORK FOR ANALYSING PROGRAM DESIGNS

3.1 General Points

This chapter details a method of analysing a program design referred to as the Framework for Analysing Program Designs (or FAPD). FAPD views analysis as the translation of a program design into a series of assertions which represent how the design could be implemented in a particular programming language. These assertions are then used to produce a coded version of the design together with any comments concerning its implementation. Broadly speaking, the process of analysis is viewed as comprising four distinct phases:

a) pre-semantic analysis which converts a program design into a form acceptable to the semantic analyser;

b) semantic analysis which analyses statements often found within a program design in terms of the particular programming language in which the design will be implemented;

c) generation of comments which uses the assertions produced through semantic analysis to derive the implications of implementing the design in code. At this stage these comments are also represented by a series of assertions; and

d) code generation which uses the results of the previous two phases to produce a program in the target language together with any comments concerning its implementation in this form.

FAPD is directed towards analysing and commenting upon the kind of program design produced using a methodology similar to the one taught to first year computer science students at the University of Aston (see section 1.3). At the time research commenced the primary programming language taught to these students was ALGOL 68. Consequently this language was chosen as the target language for this study. The system described in this thesis analyses a program design by converting it into an ALGOL 68C program together with any comments considered pertinent. For this reason examples of coded statements used in the remainder of this chapter will be written in ALGOL 68C.

In this respect we can say that the system is an implementation of FAPD. Since first year students are now taught the same method of program design but use PASCAL as the target language, FAPD could also be implemented within a system which analyses a program design in terms of the target language PASCAL. In general FAPD is limited more by the format of the program design than by the choice of programming language. This chapter discusses FAPD within the context of the system and attention will be drawn to those aspects which are dependent on the choice of implementation.

3.2 Pre-Semantic Analysis

3.2.1. Introduction

The first phase of the analysis process has been termed pre-semantic analysis. This phase is responsible for converting the design into a form acceptable to the semantic analyser. It consists of two processes, the first of which undertakes lexical and syntax analysis in order to determine whether or not the design conforms to a pre-defined syntax (referred to as the "grammar of a program design"). Successful analysis means the design can be (partially) analysed, but failure means it contains programming language constructs and/or design statements which are outside the scope of FAPD. The second process amends the syntax tree which has been produced by the first process. This amendment involves eliminating any insignificant words and converting the syntax tree into a series of structures. The result of this second process is referred to as an "amended syntax tree" and represents the data on which the semantic analyser operates.

Throughout this section examples of syntax trees and amended syntax trees have been illustrated in a format more helpful to the discussion than the actual format produced by the system. This latter format is often a LISP list structure, examples of which are contained in Appendix A. It should also be noted that all design statements used in this chapter are shown using upper-case characters. This is because the system described in Chapters 4, 5 and 6 requires a program design to be inputted using this format.

3.2.2 Lexical and Syntax Analysis

3.2.2.1 Function of Syntax Analysis

Any system which understands natural language must be limited by the number of words contained in the vocabulary of that system. Similarly FAPD can only analyse those examples which use the set of programming language constructs which have been considered for inclusion. Thus FAPD is limited both by the size of its vocabulary and by the number of target language constructs that have been considered. However, there is an additional difficulty in analysing a program design. This arises because of the unlimited number of variable names that can be used and which must be recognised by the system if a complete analysis is to be accomplished. Because of this a method of keyword analysis such as that used by ELIZA [Weizenbaum 1966, Weizenbaum 1967] is inappropriate to this research. Simply noting variable names when they are declared is also not possible in this case, since program designs do not generally contain variable declarations.

The recognition of variable names could be simplified by defining a list of names which a programmer must use. However, this approach is rejected as too restrictive. Since meaningful variable names are an important feature of quality software, a programmer should not be constrained to a list of variable names which may prove inappropriate for a particular application. The method adopted in this study is to define a syntax to which design statements must adhere. This syntax defines where identifiers are allowed and in so doing it gives FAPD a criterion for determining which of the unrecognised words are possible variable names. Although this approach obviously imposes some limitations on the variety of statements which can be accepted, it is hoped these limitations are not too restrictive.

3.2.2.2 Scope of the Syntax

Although this research has concentrated on analysing program designs similar to that shown in diagram 5 (see section 1.2), considerable variations in program design may exist in practice. Whereas diagram 5 contains design statements such as:

```
        INITIALISE FIVEPOUNDS TO 0          (3.1) and
        READ THE NEXT NUMBER INTO WAGE      (3.2)
```

the author has noticed examples where other variations such as:

```
        FIVEPOUNDS ← 0                            (3.3)
        READ THE NEXT NUMBER (AND CALL IT WAGE)   (3.4)
```

are used. The observed form of a program design is often a combination of personal trait and teaching method. Also the distinction between design statements and code is often less marked when the programmer has experience of a particular programming language. In this respect a design statement such as:

```
        WHILE I IS LESS THAN N                     (3.5)
        DO   one or more design statements   OD    (3.6)
```

may sometimes be written as:

```
        WHILE   I < N                              (3.7)
        DO   one or more design statements   OD    (3.8)
```

The variety of statements which FAPD aims to encompass should not be unduly restricted. However in order to undertake syntax analysis it is necessary to define the kinds of statements that should be included. Consequently it was decided to concentrate on defining the syntax of statements such as (3.1), (3.2) and (3.5) rather than (3.3), (3.4) or (3.7). By doing so it can be stated clearly that a program design should not contain operators such as "$\leftarrow$" or symbols such as parentheses. Since target language constructs such as "$<$" are also prohibited, this means that the only features of a programming language a programmer need know are those used to denote selection and repetition of actions. In a system which uses ALGOL 68C as the target language, selection and repetition are denoted by the constructs IF-THEN-ELSE-FI and WHILE-DO-OD respectively.

The syntactic format of a program design, which FAPD is capable of analysing is expressed formally as a meta-language (see Appendix B). Program designs not adhering to this format are rejected before they are passed to the semantic analysis routines. Hence the syntax adopted imposes one of the main limitations on the scope of FAPD.

3.2.2.3 Definition of Recognised Words

Many of the statements within the type of program design under consideration are in an imperative form. Statements (3.1) and (3.2) are typical of this form since they use the verbs INITIALISE and READ in an imperative context. Because a sentence which uses INITIALISE in this manner will normally be coded into an assignment statement, INITIALISE has been defined within FAPD as an "assignment command word". Other imperatives which are defined as assignment command words include ASSIGN, INCREMENT and SET. Similarly, imperatives may be defined as arithmetic command words, read command words and print command words since they indicate that the design statements in which they are used are normally coded as arithmetic expressions and read and print statements respectively.

Any word that is to be recognised must be entered into a dictionary. Each entry is of the following form:

[<word> <list of one or more definitions >] (3.9)

where a word unless it is one of the reserved words such as IF, WHILE or DO for example, must be described in terms of one or more of the nineteen different definitions on which the syntax is based. Hence, the imperative form of a verb, such as INITIALISE, can be defined as:

[INITIALISE assignment command word] (3.10)

Within an imperative statement considerable attention must also be given to prepositions. If we consider the design statement:

ADD A TO B (3.11)

the preposition TO is of special significance since in this instance, it separates the two arguments A and B relating to the arithmetic command word ADD. Consequently it is defined in the dictionary as a separator.

Prepositions are important words in the process of program design analysis, not only for this reason, but also because they can be used to derive the meaning of a design statement. For instance the different meanings of:

DIVIDE A BY B (3.12) and

DIVIDE A INTO B (3.13)

derives purely from the different prepositions used. The same preposition can also be used for more than one imperative, as in:

ADD A TO B AND ASSIGN THE RESULT TO ANS (3.14)

where TO denotes the effect and destination of the verbs ADD and ASSIGN respectively. In this example each occurrence of the preposition is used in connection with the verb immediately preceding it. Conversely a preposition may not be compatible with a particular verb. For instance in statement (3.12) the preposition BY could not be replaced with the preposition TO. In terms of the dictionary definitions used by FAPD, TO is determined to act as a separator for ADD and ASSIGN but not for DIVIDE. The dictionary definition for the word BY will now appear as:

```
[BY   (separator  (INCREASE DECREASE DIVIDE
              INCREMENT DECREMENT MULTIPLY))]      (3.15)
```

which denotes that BY can be used as a separator for any of the verbs INCREASE, DECREASE, DIVIDE, INCREMENT, DECREMENT and MULTIPLY.

A third form of dictionary entry is where a word can have multiple definitions, only one of which is applicable in any given statement. Each definition may be a single item as in (3.10) or an item containing some additional information as in (3.15). The possibility of multiple definitions can be illustrated by comparing the use of SUM in the following two statements:

```
SUM  A  AND  B                        (3.16) and
DIVIDE  THE  SUM  BY  2               (3.17)
```

Statement (3.16) specifies the arithmetic operation which has to be undertaken. However, if a design contains (3.16) followed immediately by statement (3.17) we can surmise the latter use of SUM refers to the arithmetic expression in the previous line. In this respect SUM can be used as either a verb, which means it must be defined within the dictionary as an assignment command word, or as a noun. In the latter case it refers to the result of the preceding arithmetic expression and hence it is defined within FAPD's dictionary as a reference. As a result the dictionary entry for SUM is:

[SUM reference assignment command word] (3.18)

This use of the term reference means that the word can be used to reference objects previously defined. Hence pronouns such as IT and THEM would also fall into this category.

The fourth and final type of dictionary entry is that used for reserved words such as IF and WHILE. A typical definition of a reserved word is:

[IF IF] (3.19)

where the fact that the word and its definition are identical is used to indicate the occurrence of a reserved word.

This section has shown how words are defined by referring to four examples of entries in the dictionary - (3.10), (3.15), (3.18) and (3.19). Although only seven definitions have been considered in this section, other definitions include "adjective" for words such as NEXT and FIRST, "article" for AN and THE and "constant" for a numerical word such as ONE, TWO, or THREE. A comprehensive list of all words recognised by the system together with their definitions appears in Appendix B.

3.2.2.4 The Syntax of a Program Design

The previous section stated that many of the statements within the type of program design being considered, display a similarity to the imperative form of a sentence. Consequently, a statement such as:

INITIALISE FIVEPOUNDS TO 0 (3.20)

can be described by the basic format:

<command word>   <arguments>   (3.21)

In this case INITIALISE is the command word and FIVEPOUNDS and 0 are both arguments. The previous section pointed out that the command word of a statement gives some indication of how that statement can be implemented in code. For this reason (3.20) is defined as an "assignment design statement". Because the syntax of a design statement is based on the imperative form of a sentence FAPD defines read, print and arithmetic design statements as those statements which commence with a read command word, print command word and arithmetic command word respectively.

This research is concerned with analysing only those program designs which can be implemented in a programming language using loops, conditionals, assignment, read and print statements. Hence in terms of the syntactic definitions used by FAPD, a program design must consist of these statements written in their design form together with arithmetic design statements. The reasons for including the latter syntactic unit will now be elaborated.

Within a program it is usually the case that the result of an arithmetic expression will be used by another statement. In the following example:

```
INPUT  THREE  NUMBERS          (3.22)
ADD  THEM  TOGETHER            (3.23)
PRINT  THE  RESULT             (3.24)
```

the arithmetic expression in line (3.23) can only be incorporated into the PRINT statement once the meaning of RESULT has been determined. Consequently the design must be passed as syntactically correct in order for the semantic analyser to combine lines (3.23) and (3.24) into a single statement. Hence the grammar of a program design must allow an arithmetic design statement to be used in the manner illustrated above.

This approach to syntax analysis allows an initial judgement to be made on whether or not a design will result in a syntactically correct program. Thus any program design which contains a programming language construct such as a loop or a conditional in an incorrect format would be analysed as syntactically incorrect. Although a program design comprising of statements (3.22), (3.23) and (3.24) appears valid the correctness of each individual statement cannot be determined at this stage. For instance, the validity of statement (3.23) can only be determined when the meaning of the pronoun THEM is derived. Consequently the checking of individual statements must be left until the semantic analysis phase has derived the meanings of arguments such as THREE NUMBERS, THEM and RESULT.

This section has illustrated how the syntactic definition of a program design has been derived. Because of the method used for analysing a program design (see section 1.4) the syntax is defined in terms of the programming language statements used in its implementation. Now that the syntax of a program design has been discussed we can consider the syntax of individual design statements.

This is outlined in section 3.3.2.6 which traces how an assignment design statement is checked for syntactic correctness.

3.2.2.5 Lexical Analysis

The primary operation within the first phase of the analysis is undertaken by the scanner. This is responsible for reading in a program design and performing lexical analysis. The scanner searches the dictionary for each word and if found forms the appropriate token. If a word has a single definition then the token appears as:

$$\left[\text{reference} \quad (\text{RESULT})\right] \qquad (3.25)$$

indicating that RESULT is defined within the dictionary as a reference. Alternatively a word can have multiple definitions in which case its token has a form similar to the following:

$$\left[\text{adjective} \quad (\text{POSITIVE}) \quad (\text{adjective reference})\right] \quad (3.26)$$

which indicates that the word POSITIVE is used as an adjective within the current context. However in case this is incorrect a list of the alternative definitions of POSITIVE is appended onto the end of the token. Any unrecognised words are given one of two definitions. If the word is a digital representation of a number (i.e. 1, 2 rather than ONE, TWO) it is defined as a constant, otherwise it is assumed to be a user defined variable name.

The token stream for the following statement:

SET A AND B BOTH TO 1 (3.27)

is of the following form:

[assignment command word (SET)] (3.28)

[article (A)] (3.29)

[conjunction (AND)] (3.30)

[variable name (B)] (3.31)

[variable name (BOTH)] (3.32)

[(separator (ADD ASSIGN INITIALISE SET UPDATE))
(TO) [(separator (ADD ASSIGN INITIALISE SET
UPDATE)) boolword-3]] (3.33)

[constant (1)] (3.34)

which shows that SET, A, AND, TO and 1 are all recognised words whereas B and BOTH are not. The syntax analyser, described in the following section, is entered when all the words in the program design have been described in terms of the basic syntactic units.

3.2.2.6 Syntax Analysis

Syntax analysis is responsible for recognising the syntactic structure of the tokens delivered by the scanner. It checks the structure for correctness and if valid it produces a parsed representation of the program design in the form of a syntax tree. If the structure is incorrect, an error is reported. The method of syntax analysis used for a program design relies heavily on backtracking since there is a frequent need to parse a word with multiple definitions. Indeed, since a programmer can use any recognised (but not reserved) word as a variable name, this means most words can have at least two definitions. For instance A can be used as the indefinite article as shown by the following statement:

    INPUT A VALUE INTO X   (3.35)

or as the previous section illustrated, a programmer could use A as a variable, viz:

    SET A AND B BOTH TO 1   (3.36)

Let us consider how FAPD's approach to syntax analysis uses the grammar of a program design and a backtracking mechanism in order to successfully parse (3.36), by re-defining A as a variable name and BOTH as a word that can be ignored. The syntactic format of a program design is specified by a grammar (see Appendix B). The grammar contains a set of rules which can be described concisely in a meta-language called Backus Naur Form (BNF). The grammar of an assignment design statement is defined in modified BNF as:

```
<assignment design statement>::= <assignment
  command word> <arguments>
  [<separator> <separated arguments>
  {<conjunction> <separated arguments>|
   <separator> <separated arguments>)}]  (3.37)
```

As parsing continues from left to right SET will be successfully parsed as the <assignment command word> and the grammar relating to <arguments> allows A to be parsed as an article. An article could be used in this position for statements such as (3.35) and:

    SET A COUNTER TO 0   (3.38)

However this approach to the parsing of statement (3.36) is halted once AND is encountered since the definition of <arguments> does not allow a conjunction to immediately follow an article.

At this point the backtracking mechanism is invoked in order to find an alternative parsing. Because the current focus of attention is the definition of <arguments> , the backtracking mechanism will be confined initially to those tokens successfully parsed within this part of the grammar. If no alternative is found, then backtracking is resumed higher up the tree. The only token successfully parsed according to the definition of <arguments> is that relating to A. Consequently the token for A is changed from:

[article (A)] (3.39)

to [variable name (A) (variable name) article] (3.40)

and parsing according to this new definition is attempted. If the token had been changed to:

[variable name (A) (variable name article)] (3.41)

then the syntax analyser would have parsed it continually as an article, since it looks at all possible definitions, denoted by the list containing variable and article, rather than confining attention to the current definition, which is variable name. The form of definition (3.40) forces A to be parsed as a variable name. At this stage statement (3.36) has only one token that can be redefined. However if there had been more then all possible alternatives would have been tried before reporting failure.

This new definition of A together with the existing definitions of AND and B are now successfully parsed according to the definition of <arguments> .

The grammar of (3.37) states that the next token in the token stream should be a separator. However the next token in the stream is:

$$\left[\text{variable name} \quad (\text{BOTH})\right] \qquad (3.42)$$

which indicates that BOTH is an unrecognised word. Since its definition is inconsistent with the current context and because it is an unrecognised word it can be discarded for the moment. Consequently token (3.42) is altered to:

$$\left[\text{ignorable word } (\text{BOTH})\right] \qquad (3.43)$$

before it is added to the tree. It is important to note that it is not discarded entirely but is retained and may be redefined as a variable name during a future back-up.

Successful parsing of SET A AND B is sufficient evidence of an assignment design statement since (3.37) indicates anything else is optional. The grammar of an assignment design statement has been defined in this way in order to encompass statements such as:

INITIALISE I (3.44)

where no separator or second argument appears. In statement (3.36) the next token is a separator. In order to continue parsing in this part of the tree, we must be able to connect the separator TO with the preceding command word. The token's additional information indicates this is allowed and consequently parsing continues after abbreviating the token by changing it from:

$$\Big[\text{(separator (ADD ASSIGN INITIALISE SET UPDATE)) (TO)} \;\Big[\text{(separator (ADD ASSIGN INITIALISE SET UPDATE)) boolword-3}\Big]\Big] \qquad (3.45)$$

to

$$\Big[\text{(separator (SET)) (TO)} \;\Big[\text{(separator (SET)) boolword-3}\Big]\Big] \qquad (3.46)$$

After successfully parsing the remaining token, the syntax tree for (3.36) is complete and is shown in diagram 6. Once the statement has been parsed successfully, syntax analysis is complete and the second process within the phase of pre-semantic analysis can be entered.

3.2.3 Preparation for Semantic Analysis

Now that the program design has been parsed, the second phase of pre-semantic analysis can be entered. The prime function of this phase is to convert the syntax tree into a series of structures which the semantic analyser can recognise. This series is referred to as an "amended syntax tree". The syntax trees produced for:

SET A AND B BOTH TO 1 (3.47)

INITIALISE SUM TO 0 AND COUNTER TO 0 (3.48)

INITIALISE THE FIRST TWO ELEMENTS OF THE ARRAY (3.49)

are not identical. The semantic analyser however, requires that all statements which are implemented using the same target language construct should have a similar

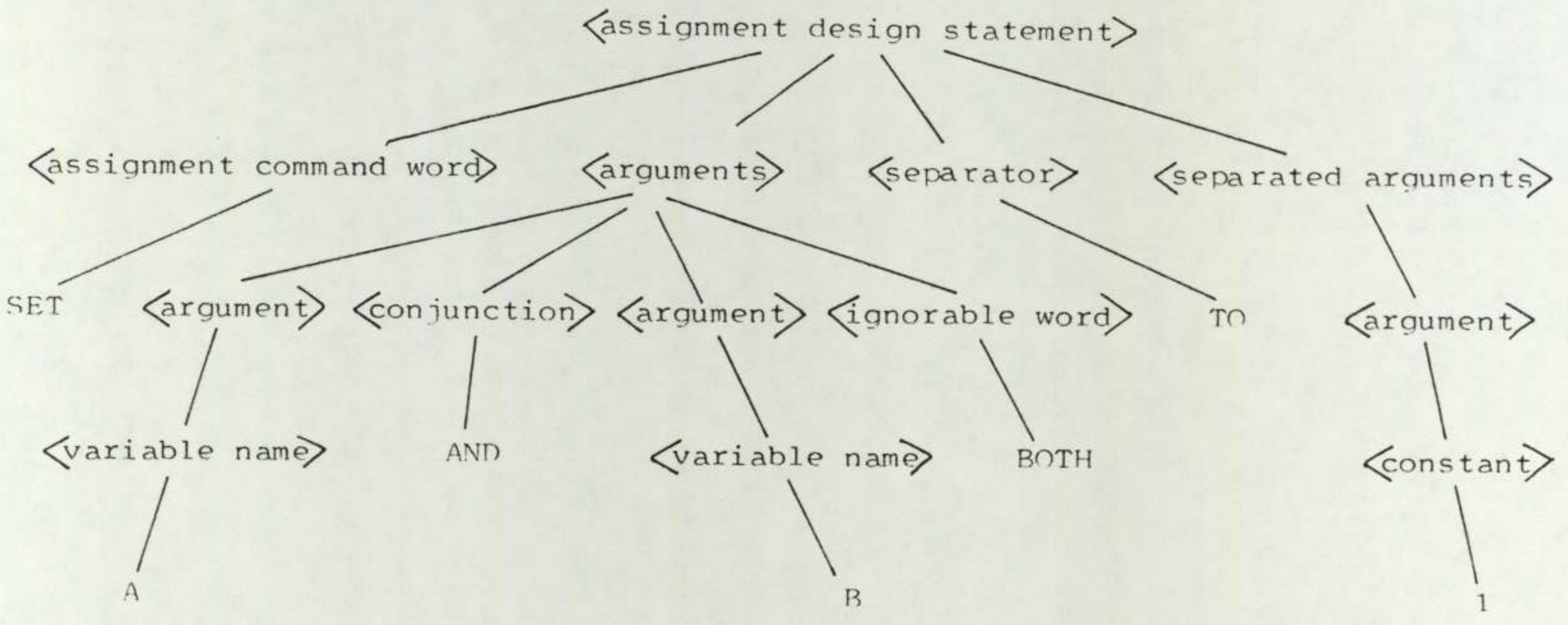


Diagram 6

The Syntax Tree of the Statement "SET A AND B BOTH TO 1" representation. Consequently because statements (3.47), (3.48) and (3.49) will all be implemented as assignment statements they will have the same structure. A structure is used to represent common elements within a program design. FAPD proposes a set of structures, some of which are derived from the syntactic format of a design, and some of which have been specifically developed to aid semantic analysis. Preparation for semantic analysis is concerned solely with producing the former of these.

The general form of a structure is defined as:

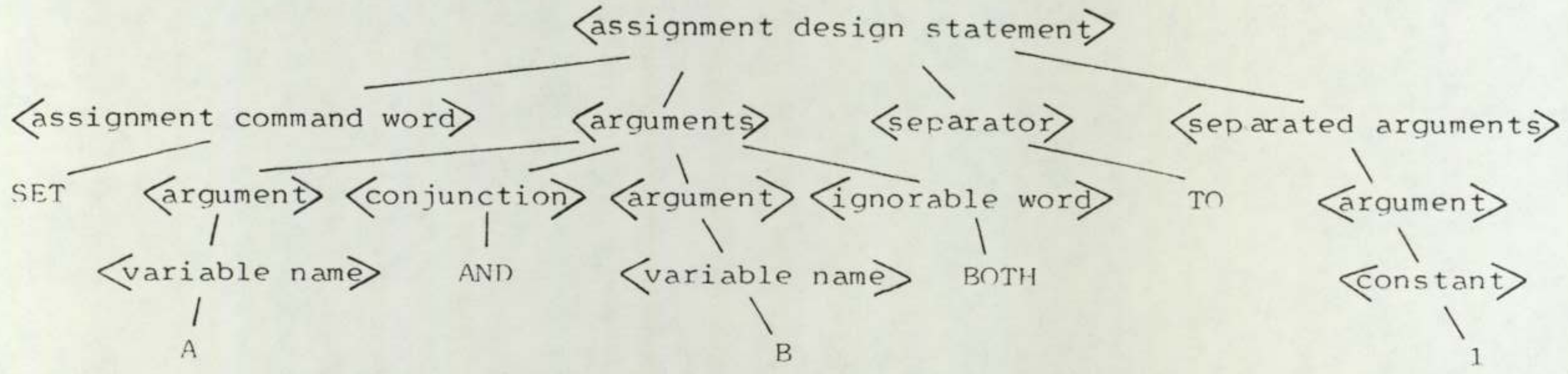
[<name of structure> <one or more structure
  fields>] (3.50)

A typical structure is:

[#ASS <assignment command word> ARGUMENT
  <separator> ARGUMENT] (3.51)

which is that used for the representation of an assignment design statement. Thus the syntax trees for statements such as (3.47), (3.48) and (3.49) can all be represented by the structure shown above. This has been given the structure name #ASS and contains four structure fields. Diagram 7 shows design statement (3.36) together with its syntax and amended syntax trees. The amended tree shows how SET and TO have been entered into the appropriate fields and how ARGUMENT is used to denote a general field which can be filled with other structures.

In order to produce this amended form we need to know how to treat each non-terminal of the grammar.



An amended form of the syntax tree is :

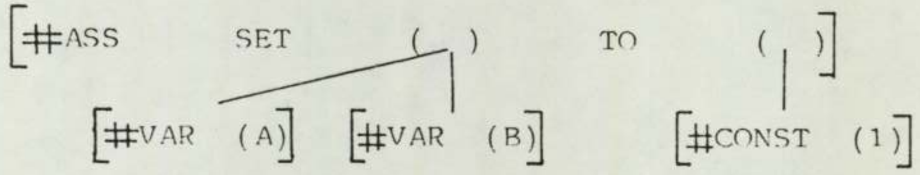


Diagram 7

The Syntax Tree and the Amended Syntax Tree of the Statement "SET A AND B BOTH TO 1"

Within the syntax tree all non-terminals are shown in angled brackets. At the lowest level of the tree, non-terminals such as assignment command word, variable name, conjunction, separator and constant are merely the dictionary definitions of SET, A and B, AND, TO and 1 respectively. At a higher level non-terminals such as assignment design statement and arguments are shown to comprise a series of other non-terminals. The way in which a non-terminal is treated is derived from its semantic definition.

A non-terminal which is comprised of other non-terminals is semantically defined in one of two ways:

a) it can be defined as a structure with multiple fields. Thus structure (3.51) is the definition of an assignment design statement and denotes how each of the non-terminals, assignment command word, arguments, separator and constant, shown in diagram 7 are to be treated; or

b) it may be defined as a non-terminal that can be ignored. This is used for an element of the grammar such as ⟨arguments⟩ which does not require its own structure because it is further defined in terms of other non-terminals. In diagram 7,⟨arguments⟩ is analysed as a series of two structures relating to the variables A and B.

A non-terminal which is a dictionary definition is defined in one of three ways:

a) it can be defined as a structure with a single field. Thus variable name and constant are defined as the classes:

   [#VAR WORD] (3.52) and

   [#CONST WORD] respectively (3.53)

   Diagram 7 shows how the WORDs A, B and 1 have been entered into these classes;

b) a second possibility is when a dictionary definition is defined as a field within a structure. Two examples of this are <assignment command word> and <separator> which are two fields within structure (3.51). In the amended tree these are filled by SET and TO respectively;

c) words which do not make a significant contribution to the semantic context of a sentence can be eliminated. Thus AND and BOTH in statement (3.36) are discarded before the semantic analyser is entered. In this respect we can say that any words which are defined within FAPD as either <ignorable word> or <conjunction> can be eliminated.

Appendix B shows how each non-terminal of the grammar is semantically defined into one of the five classes described above.

This section has based its discussion on the analysis of a single statement. In practice however, a typical program design consists of loop and conditional constructs along with read, print, assignment and arithmetic design statements. Consequently the amended syntax tree of a complete design should contain structures to denote these constructs. The production of an amended tree marks the end of pre-semantic analysis and the design is now in a form suitable for semantic analysis.

3.3 Semantic Analysis

3.3.1 Function of Semantic Analysis

The primary function of semantic analysis is to build a series of assertions which represents a coded form of the program design. Its secondary function is to initiate the processes which detect any implications of forming this representation. These processes run in parallel with the semantic analyser, although any implications are noted as a side effect and do not influence any of the semantic routines. The previous section outlined a set of general structures used for recognising design statements. Semantic analysis is based on the recognition of specific instances of each general structure. The general structure for assignment design statements was shown to be:

[#ASS ⟨assignment command word⟩ ARGUMENT
    ⟨separator⟩ ARGUMENT] (3.54)

and two instances of this general structure are:

[#ASS ASSIGN ⟨first argument⟩ TO ⟨second
    argument⟩] (3.55)

[#ASS INITIALISE ⟨first argument⟩ TO
    ⟨second argument⟩] (3.56)

Attached to each structure is a procedure which translates its structure into a particular programming language. Consequently if ALGOL 68C is the target language then the procedure attached to (3.55) will produce an assertion which denotes the following assignment statement:

    <second argument> := <first argument>(3.57)

Conversely the procedure attached to (3.56) will produce an assertion which denotes the statement has been analysed as having the following implementation:

    <first argument> := <second argument> (3.58)

Because (3.55) and (3.56) have the same structure name (i.e. #ASS) they are defined as both belonging to the same <u>class</u>. Consequently the procedures attached to each of these structures are referred to as <u>class instances</u>.

The results produced by each class instance are determined, to some extent, by the choice of target language. For example, if ALGOL 68R is the target language, then the class instance which recognises the statement:

    READ TEN NUMBERS INTO AN ARRAY   (3.59)

implements this by using a single READ statement in the following manner:

```
READ (ARR01);                (3.60)
```

where ARR01 is the name of the array. Alternatively if PASCAL is the programming language then the same class instance would produce the following implementation:

```
FOR I:= 1 TO 10 DO READ (ARR01 [I]); (3.61)
```

Thus changes in the target language will require that the class instances be re-programmed. However in order to keep alterations of this kind to a minimum, a method of representing a program, irrespective of the target language, has been devised (see section 3.3.2). By doing this the only class instances that need to be re-programmed are those for which the target language uses different constructs. If the target language is changed from ALGOL 68C to LISP, say, then class instances relating to structures (3.55) and (3.56) can be left unaltered since they are implemented as assignment statements in both languages.

At this stage it is important to note that class instances of the type discussed above are incapable of determining the implications, if any, of their results. For example the class instance which recognises statements containing the word INITIALISE cannot differentiate between the following two statements:

```
        INITIALISE   1   TO   4          (3.62)
        INITIALISE   4   TO   1          (3.63)
```

Consequently there is no guarantee that the program produced by analysing a program design will be free of compilation errors. It is felt that the detection of such errors is the responsibility of a compiler and therefore need not be duplicated within FAPD.

This section has given an outline of the aims of semantic analysis and section 3.3.2 now describes the method used for representing the coded version of a program design. This is described in preparation for section 3.3.3 which gives a more detailed account of how a design statement is converted into its coded form.

3.3.2 Representation of a Program

An objective in developing FAPD was to make it, as far as possible, independent of the choice of programming language. Consequently, as long as the same method of program design is used, FAPD should be applicable to examples that are eventually coded into different programming languages, such as PASCAL or ALGOL 68C. In order to achieve this objective, a method of representing a program has been devised which is independent of the target language. This representation is called an assertion language.

FAPD is limited to those examples which, when implemented in a target language, can be represented by FAPD's assertion language. The assertion language has been developed in order to represent the following features of a programming language:

a) loops of the WHILE rather than the FOR variety;

b) conditionals of the IF - THEN - ELSE variety;

c) assignment statements;

d) read statements;

e) print statements;

f) boolean expressions;

g) arithmetic expressions;

h) variables;

i) numerical values; and

j) arrays and array elements

Any program design which does not use a combination of these ten features is beyond the scope of FAPD. From the discussion in section 3.2.2.2 it follows that the main limitations on the variety of examples which can be analysed are the grammar of a program design together with FAPD's assertion language. At this stage it is important to note that any design which uses loops of the FOR variety or special selection statements such as the CASE construct or references to sub-procedures cannot be analysed and will not be processed by the semantic analyser.

In order to represent the ten language features listed above, seventeen different forms of an assertion have been developed. The general form of any assertion is defined as:

[<type of assertion> <one or more assertion fields>
<assertion name>] (3.64)

The <type of assertion> gives some indication of the kind of information contained in the assertion fields. Typical of these are #VAR, #CONST and #COND used to denote assertions containing variable names, numerical values and conditional statements respectively. An assertion field can contain either a string of alpha-numeric characters or a bracketed list of one or more <assertion name>s. Diagram 8 illustrates a program design together with the ten assertions which the semantic analysis routines would use for this particular example.

Because of the hierarchical nature of the assertion language, assertion (AS1) is referred to as the top-most assertion and hence an <assertion name> is not required. It has a single field indicating the design has been analysed as consisting of the read, assignment and print statements, which are represented by assertions (AS2), The program design is as follows :

```
INPUT THREE NUMBERS
ADD THEM TOGETHER AND ASSIGN THE RESULT TO ANSWER
PRINT THE VALUE OF ANSWER
```

The assertions to represent this program design are :

```
(AS1)    [#DESIGN   (RD1  A1  P1)]
(AS2)    [#READ   (V1  V2   V3)   RD1]
(AS3)    [#ASS   (V4)   (E1)   A1]
(AS4)    [#EXPR   +   (V1)   (E2)   E1]
(AS5)    [#EXPR   +   (V2)   (V3)   E2]
(AS6)    [#PRINT   (V4)   P1]
(AS7)    [#VAR   NILL   V1]
(AS8)    [#VAR   NILL   V2]
(AS9)    [#VAR   NILL   V3]
(AS10)   [#VAR   ANSWER   V4]
```

Diagram 8

An Example of the Results Produced by the Semantic Analysis Routines

(AS3) and (AS6) respectively. Assertions (AS4) and (AS5) are the results of analysing the statement ADD THEM TOGETHER. These show than an #EXPRession assertion has three fields, the first of which contains a dyadic arithmetic operator. The operator's arguments are contained in the remaining two fields, which for a correctly formed expression should contain the <assertion name> of either a #CONSTant, #VARiable, array #ELEMENT or an arithmetic #EXPRession assertion.

It is important to notice that the assertions do not contain any information concerning the coding details of a particular language, for example the exact placement of semi-colons or the form of variable declaration statements. Also the assertions are sufficiently general to denote statements in more than one language. For instance (AS3) and its related assertions can be used to represent either the ALGOL 68 statement:

```
        ANSWER := IDR01 + IDR02 + IDR03        (3.65)
```

or even the LISP statement:

```
    (SETQ ANSWER (PLUS IDR01 IDR02 IDR03)) (3.66)
```

The responsibility of converting it into either of these forms can be left until the code generation phase (see section 3.5). Assertions (AS7), (AS8) and (AS9) show that the variables relating to the THREE NUMBERS have been given default names of NILL. Since the semantic analyser uses an <assertion name> rather than an actual name in order to build the assertions, the task of generating suitable identifier names such as IDR01, IDR02 and IDR03 can also be delegated to the phase of code generation. Further details of the assertion language can be found in Appendix B which contains a formal definition of the language showing how it can be used to represent a coded version of a program design.

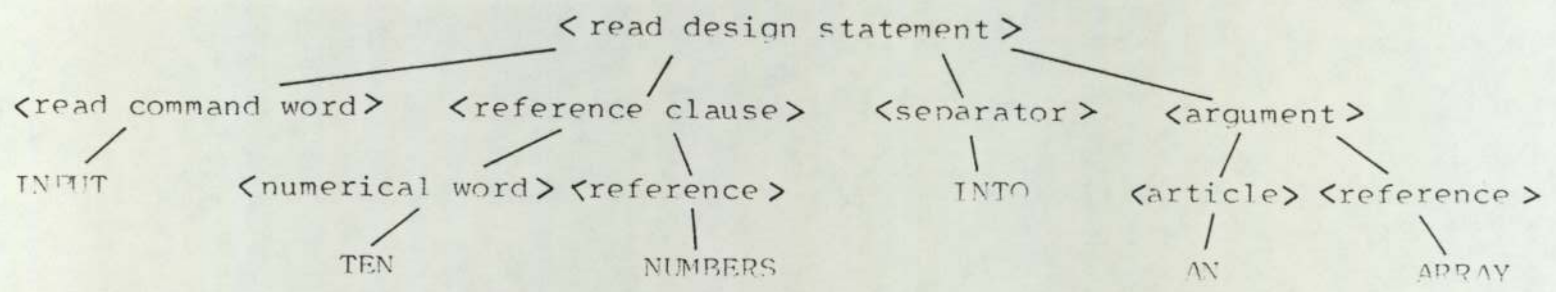
3.3.3 Analysis of a Design Statement

In this section we consider how semantic analysis converts the results of pre-semantic analysis into a series of assertions. Semantic analysis will be discussed by referring to the processes involved in analysing the following design statement:

INPUT TEN NUMBERS INTO AN ARRAY (3.67)

Six class instances are used to produce the assertions which represent this statement's implementation in ALGOL 68C.

Diagram 9 shows modified forms of the syntax and amended syntax trees relating to statement (3.67) which, for it to be analysed, needs a class instance for each of the four structures labelled (C1), (C2), (C3) and (C4). For ease of discussion, this section will refer to the different class instances by using these labels. The amended syntax tree is analysed in a depth first, left to right manner and hence class instance (C1) is the first to be considered. This class instance is used to recognise any design statement containing the read command word INPUT and to produce the appropriate assertion(s) which, in this case, is an ALGOL 68C READ statement. The first operation involves analysing the left hand argument by searching for class instance (C2). A search is then made for class instance (C4). If the search is successful, then the results of these two instances are considered



An amended form of the syntax tree is :

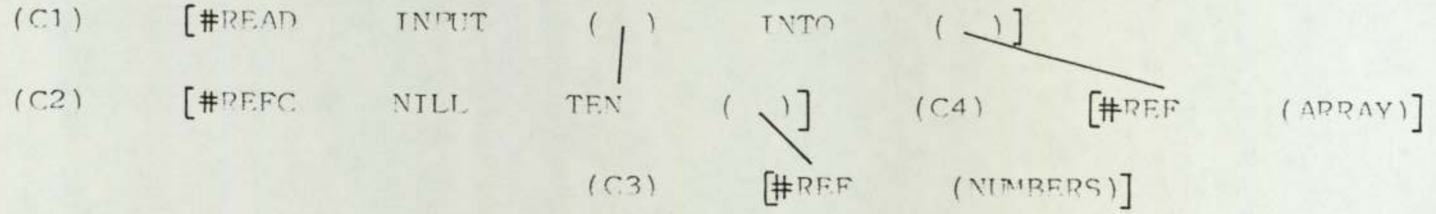


Diagram 9

The Syntax Tree and the Amended Syntax Tree of the Statement "INPUT TEN NUMBERS INTO AN ARRAY" together within the overall context of a read design statement.

Class instance (C2) is concerned with typical phrases likely to be found within design statements such as TWO NUMBERS and FOUR VALUES for example. The first operation of (C2) like that of (C1) involves deriving the meaning of the arguments within the current context. This is achieved by calling class instance (C3) in order to derive the meaning of NUMBERS. Within the scope of FAPD, NUMBERS must refer to a set of numerical values and in terms of FAPD's programming knowledge, a value can be stored in either an array element or a variable. Hence the first attempt at analysis assumes the programmer has used NUMBERS as a reference to a set of variables, the size of that set being undefined. In terms of the assertion language we can say that NUMBERS is analysed as meaning (V1 V2 ... VN) where V1, V2, V3 etc are the names of variable assertions with the following format:

```
        [#VAR   NILL   V1 ]                (3.68)
        [#VAR   NILL   V2 ]                (3.69)
                etc.
        [#VAR   NILL   VN ]                (3.70)
```

In general, any class instance attempts to convert its structure into the appropriate assertions, the names of which represent the results of its analysis. However before leaving class instance (C3) it is necessary to record how NUMBERS has been analysed. This is necessary in case the word is used again within the same design. For instance if a design contained statement (3.67) followed by:

ADD THE NUMBERS TOGETHER (3.71)

then NUMBERS obviously refers to the same variable names. In order to detect this we link the assertions to the design by making an <u>intermediary assertion</u> of the form:

[#REFV NUMBER (V1 V2 ... VN)] (3.72)

An intermediary assertion is defined to be an assertion which either aids semantic analysis or the generation of comments but which is not used by any subsequent phase in the analysis. Consequently intermediary assertion (3.72) is not required by the code generator in order to print a coded version of design statements (3.67) or (3.71).

Statement (3.67) has been specifically chosen as an example because it illustrates how any class instance attempts to analyse how its structure can be implemented in a programming language, even though the results of its analysis are often revised when considered in a wider context. Thus class instance (C2) can now revise the list of variable assertion names from one of indeterminate size to one comprising just ten names. As a side effect intermediary assertion (3.72) is also amended to:

[#REFV NUMBER (V1 V2 ... V10)] (3.73)

and the list (V1 V2 ... V10) now represents the results of analysing the left hand argument of class instance (C1).

Class instance (C1) now attempts to analyse the right hand argument and search for a class instance which recognises the word ARRAY. (C4) is found and an array assertion of the following form is made:

[#ARRAY NILL (LB1) (UB1) A1] (3.74)

where NILL is the default name of an array and (LB1) and

(UB1) are the names of assertions which contain the values of the array's lower and upper bounds respectively. At this stage we have no criterion for determining these values and hence they are assigned default values of 1 and N respectively. These values are represented by the following assertions:

```
        [#LWB    1    LB1]                 (3.75)
        [#UPB    N    UB1]                 (3.76)
```

So far, analysis has used four class instances, all of which are based on the structures formulated by pre-semantic analysis. However semantic analysis often needs to use a series of additional class instances in order to complete its operation. A comprehensive list of all structures defined by FAPD is contained in Appendix C. For the statement under discussion, two additional class instances are required. The first of these has the format:

```
        [#RDARGS    <first argument> <second argument>] (3.77)
```

and it is invoked whenever the first argument of a read design statement is analysed as a list of variable assertion names and when the second argument is analysed as the name of an array assertion.

In terms of statement (3.67) this class instance will perform three operations:

a) Now that the number of values which are to be read in has been determined, the upper bound of the array can be re-defined. Consequently the array is re-defined as one of ten elements by altering assertion (3.76) to:

```
        [#UPB    10    UB1]                (3.78)
```

b) Each of the variable assertions V1 to V10 can be erased and as a result the intermediary assertion (3.73) is altered to:

[#REFV NUMBER (A1)] (3.79)

which indicates that NUMBER now refers to an array of ten elements. If the word is used within a phrase such as FIRST NUMBER, this intermediary assertion is used by a class instance to infer that it means the first element of the array.

c) The following assertion is made which denotes a series of values are to be read into an array:

[#READ (A1) RD1] (3.80)

The second of the additional class instances is called to determine how an assertion such as this can be incorporated into the results of analysing previous statements. It has the following structure:

[DESIGN ARG <assertion names>] (3.81)

and is invoked whenever its argument is the name of a #READ assertion. This class instance makes the appropriate loop and assignment assertions that are necessary when reading values into an array using the programming language ALGOL 68C.

Semantic analysis of statement (3.67) is now complete and diagram 10 summarises the analysis by showing the original design statement together with the results produced by the six class instances. The ALGOL 68C code, also shown, can be derived from knowing that the result of analysing statement (3.67) is represented by a list of just two assertion names - (AS1 LP1). This shows that the design statement has been analysed as comprising an assignment statement (AS1), followed by a loop (LP1).

```
[#ASS     (V1)    (LB1)    AS1]
[#VAR    NILL    V1]
[#LOOP    (B1)    (RD1    AS2)    LP1]
[#BOOLOP    <=    (V1)    (UB1)    B1]
[#READ    (EL1)    RD1]
[#ELEMENT    (A1)    (V1)    EL1]
[#ASS    (V1)    (E1)    AS2]
[#EXPR    +    (V1)    (C1)    E1]
[#CONST    1    C1]
[#ARRAY    NILL    (LB1)    (UB1)    A1]
[#LWB    1    LB1]
[#UPB    10    UB1]
[#REFV    NUMBER    (A1)]
```

A coded form of the statement is :

```
IDR01  := 1 ;
'WHILE  IDR01 <= 10
'DO      READ  (ARR01  [IDR01] )  'OD ;
```

Diagram 10

The Assertions and Program Code Which Represent the Statement "INPUT TEN NUMBERS INTO AN ARRAY"

Because of the hierarchical nature of the assertion language all other information needed to produce the code can be obtained via each of these assertions.

3.3.4 Scope of Semantic Analysis

The phrase "program design analysis" is used throughout this thesis to mean the conversion of a program design into a series of assertions which represent a coded version of that design. Consequently this research has aimed to develop a series of class instances capable of implementing a design in a programming language. The knowledge contained within a class instance has been confined to common programming techniques in much the same way that knowledge is confined to the blocks world in Winograd's system [Winograd 1972].

The kind of examples which FAPD aims to analyse are those which require elementary programming skills in order to be implemented. Hence a typical statement within such a program design could be:

```
      CALCULATE THE TOTAL OF THE VALUES
      OF THE ELEMENTS OF THE ARRAY                  (3.82)
```

The implementation of this statement is important since it demonstrates how a loop structure is often used to index consecutive elements of an array. In terms of FAPD's assertion language, statement (3.82) is represented by:

```
      [#PRED   TOTAL   (A1)   P1 ]                  (3.83)
      [#ARRAY  NILL   (LB1)  (UB1)   A1 ]           (3.84)
```

where assertion (3.83) is an intermediary assertion denoting that the operation TOTAL is to be applied to an array. Thus statements such as:

FIND THE AVERAGE OF THE ELEMENTS OF
THE ARRAY (3.85) and

FIND THE MAXIMUM VALUE OF A AND B AND C (3.86)

can also be represented by similar intermediary assertions such as:

[# PRED AVERAGE (A1) P2] (3.87) and

[# PRED MAXIMUM (V1 V2 V3) P3] (3.88)

where (A1) is the assertion name of an array and V1, V2 and V3 are assertion names relating to the variables A, B and C. Hence, because statements (3.82), (3.85) and (3.86) can all be represented in this manner they are considered to be within the scope of FAPD's semantic analysis.

Generally speaking, a statement is within this scope if all the information required for its implementation can be derived from the following two sources:

a) from a class instance which is capable of translating a common design statement or phrase into a target language. Class instances for predicates such as TOTAL, MAXIMUM and AVERAGE can also be developed since the intuitive meaning of such words is sufficiently explicit to allow their use in more than one design exercise;

b) from the results of analysing previous statements in a program design. Consider a program design which contains a statement for finding the average value of the elements of an array. In this situation the class instance relating to the calculation of an average must be able to determine if the design contains a previous statement which calculated the total of the values held in the array elements. If such a statement exists then the class instance must use the results from analysing this previous statement in order to implement the code for calculating the average.

Conversely a statement is beyond the scope of semantic analysis when some of the information required for its implementation cannot be derived from either of these sources. A statement typical of this is:

PROCESS DATA FOR EMPLOYEE (3.89)

This was referred to in chapter 1 as a general statement covering the various operations used to derive the number of notes and coins a company cashier requires to pay out to an employee on the company's payroll. However this statement could also be found in a program design which uses the number of hours worked by an employee, together with his tax allowances etc. to calculate the total money earned by that employee in any given week. In this respect the meaning of statement (3.89) can only be derived from knowing the domain of discourse or the context within which the statement is made. This information is usually contained in a problem specification and since FAPD makes no use of the specification, then statements such as (3.89) are considered to be beyond the scope of semantic analysis.

Ignorance of the program specification also means that FAPD cannot determine if the design performs as intended. However, since FAPD views the process of analysing a program design as the translation of a program design into code, some of the existing theories of program understanding could be used to determine if the code(and hence the design) agrees with the specification. Although statements such as (3.89) are beyond its scope, semantic analysis should not be prevented from analysing other statements within the same design. Provided a statement, which cannot be analysed is syntactically valid, it is left unattended and attention is diverted to other statements within the design. If this occurs the design is said to be partially analysed.

3.4 Generation of Comments

Semantic analysis is concerned with building a series of assertions which represent a coded form of the program design. As these assertions are constructed it also initiates those processes which detect the implications of forming this representation. This third phase in the analysis process runs parallel to semantic analysis. However any implications are noted only as a side effect and are not used by the semantic routines. One of the main objectives of this phase is to make comments about those statements whose implementation contains a program error. Typical errors are statements which use a variable without first initialising it and statements whose implementation might lead to an array index being out of bounds. These errors are noted and converted into the appropriate English text during the code generation phase. It is hoped that any comments are of a form which a programmer would find useful.

Just as FAPD defines a set of classes for analysing common elements within a program design, it also defines a set of classes (outlined in full in Appendix C) for detecting if the results of semantic analysis are erroneous. Hence these classes are based on the structure of the assertions used to represent a coded version of the design. In this respect, certain errors in an assignment statement are detected by having class instances which are called whenever an assertion of the following form is made:

```
    [#ASS   ( <assignment argument 1>)  (<assignment
                  argument 2>) <#ASS  assertion name >](3.90)
```

Other comments about assignment statements can only be detected by considering an assertion of the form shown in (3.90) within the context of previous lines. For this reason comments about assignment statements are sometimes generated by class instances of the following forms:

```
      [DESIGN       <assertion names>]            (3.91)
      [LOOPBODY  <assertion names>
                   <#LOOP  assertion name >]       (3.92)
```

Class instances with a structure similar to (3.91) are used to consider a particular line within the current context of the program design. The current context in this study is taken to mean the preceding design statements. Similarly class instances with a structure similar to (3.92) will be used whenever the current statement is within a loopbody.

For every comment made about an assignment statement there must be class instances of these forms. Hence whenever the semantic routines incorporate an assertion into a program, all those class instances with the same structure as (3.91) are invoked. If any class instance detects an error then the appropriate information is recorded in a comment assertion which has the following general format:

[#COMM <comment number> <information>
<list of assertion names or a line number>
<assertion name>] (3.93)

where <information> could be a variable's assertion name or an unrecognised statement which is used by the code generator to produce the appropriate English text and the <list of assertion names or a line number> is used to denote where in the coded version of the design, this comment refers. All comments are initially represented in this manner and <comment number> is an integer reference number used to denote the various errors and comments that can be detected and generated.

The results of analysing a statement such as:

```
SET  A  TO  THE  VALUE  OF  B                  (3.94)
```

are represented by assertions such as

```
[#ASS   (V1)   (V2)   AS1]                     (3.95)
[#VAR    A     V1]                             (3.96)
[#VAR    B     V2]                             (3.97)
```

Consequently we use a class instance with structure (3.91) in order to check if all the variables used on the right hand side of the statement (such as B in this example) have been previously defined. Similarly we need a class instance with structure (3.92) to detect the same error for an assignment statement contained within a loop. If the variable B had not been assigned a value then the appropriate class instance would record that fact by making the following assertion:

```
[#COMM   B   (V2)   (AS1)   C1]                (3.98)
```

This contains all the information the code generator needs to inform the programmer which variable (denoted by assertion V2) has been incorrectly used and where (denoted by assertion AS1).

In addition to detecting errors within individual statements, it is also necessary to consider the results obtained from analysing a statement, within the context of previous results. For instance two statements such as:

```
        OUTPUT  THE  SUM  OF  A  AND  B               (3.99) and
        ASSIGN  THE  RESULT  TO  ANSWER               (3.100)
```

are analysed into the following print and assignment statements, both of which are correct, but which together form an inefficient piece of code:

```
        PRINT    (A   +   B);                         (3.101)
        ANSWER :=  A   +   B;                         (3.102)
```

The same operation can be achieved more efficiently by

```
        ANSWER :=   (A  +  B);                        (3.103)
        PRINT   (ANSWER);                             (3.104)
```

Thus whenever an arithmetic design statement is met, a class instance considers the expression produced in the light of any similar expressions previously analysed. Whenever statements such as (3.99) and (3.100) are found this class instance detects that when the two results are combined they display an unnecessary duplication of an arithmetic expression. The information necessary for making an appropriate comment is then recorded for later use. This is achieved by making an assertion similar to:

```
    [#COMM  10  (E2)   (E1   E2)   C2 ]               (3.105)
```

where E1 and E2 are the assertion names corresponding to the arithmetic expressions in statement (3.101) and (3.102).

So far the discussion has been concerned with comments involving erroneous statements, but comments can also be made to show how various statements within the design have been implemented. For example if the design contains a statement such as:

OUTPUT THE VALUES OF THE ARRAY (3.106)

then the fact that this is implemented in ALGOL 68C by using a loop structure is noted using the same form of comment assertion already discussed. A comment which outlines how statements such as (3.106) can be converted into a particular programming language are particularly useful for programmers who still find the implementation of such statements relatively difficult.

The results obtained from analysing a design are now represented by a set of assertions. Each assertion is restricted to one of three forms which indicates where in the analysis process they were produced:

a) an assertion may have been produced during semantic analysis and consequently is used to represent a coded version of the design;
b) alternatively it may have been produced by the phase currently under discussion in which case it represents a comment that will be made about the coded version of the design;
c) a third type of assertion has been termed an intermediary assertion. Assertions of this type are produced by either the semantic routines or the routines responsible for generating any comments. However these routines use intermediary assertions as a method of aiding their own analysis and

consequently this category of assertions is superfluous to code generation.

The task of printing the assertions, described in (a) and (b), in a readable form is the responsibility of the fourth, and final, stage of the analysing process known as code generation.

3.5 Code Generation

FAPD views the process of analysis as the translation of a program design into a series of assertions which represents how statements within the design can be realised in terms of a particular programming language (see section 1.4). In this respect pre-semantic analysis, semantic analysis and generation of comments are the three main processes. The fourth process, known as code generation, is concerned with converting the results of the last process into a computer executable form - namely a coded version of the design. Any comments pertinent to the program design are also converted into a readable form at this stage.

In order to print a program, code generation must take care of the coding details of the target language, such as how variables are declared and where semi-colons and parentheses are needed. An important feature of FAPD is how the results of analysis can be used to represent the same statement in different languages. Thus in order to print the results in different programming languages we need only provide different code generators. At this stage it is important to note that code generation is concerned only with printing a program and does not build a representation of its results in the same way as that achieved by pre-semantic analysis, semantic analysis and generation of comments.

The initial operation of the code generator is to generate oppropriate variable names for any variables or arrays which have been given a default name of NILL (see assertion (3.84) in section 3.3.4). Once this has been done all declarations can be carried out and the program printed. The assertion language has a hierarchical format and an example of a top-level assertion is:

```
    [#DESIGN      (C1   LP1)]                    (3.107)
```

The program can be printed by first of all finding this assertion and then searching for those assertions with names C1 and LP1. In this respect the operation of the code generator can be thought of as a systematic walk through all the assertions.

Conditionals and loops are represented by assertions such as:

```
   [#COND  (B1)   (AS1   AS2)   (AS3  AS4)  C1]   (3.108)
   [#LOOP  (B2)   (AS5   AS6)    LP1 ]            (3.109)
```

where B1 and B2 are boolean expressions. AS1 and AS2 are contained in the first leg of a conditional (i.e. they are executed if B1 is true), AS3 and AS4 are contained in the second leg of a conditional and AS5 and AS6 are both contained in a loopbody. Many ALGOL 68 programs contain compilation errors because the programmer has used the semi-colon as a terminator and not as a separator. Because assertions (3.107), (3.108) and (3.109) provide a convenient method of representing blocks within a program, the correct use of a semi-colon as a continuation character is made easier. Assertion (3.107) shows how the coded version of the design is represented by a conditional - C1 -

and a loop - LP1. This representation allows the code generator to detect that a semi-colon is required after each element in the list (C1 LP1) apart from the last one.

If semantic analysis has failed to analyse a statement, then the amended syntax tree corresponding to this statement is incorporated into the appropriate assertion. For instance if the phrase NOT END OF NUMBERS is not analysed within the following context:

```
        WHILE  NOT  END  OF  NUMBERS
        DO   one or more design statements    OD   (3.110)
```

then the amended tree corresponding to this is included within the appropriate boolean assertion. As the coded version of the design is being printed the code generator can detect that the assertion contains an unrecognised statement and this is then converted into the following comment assertion:

```
    [#COMM  1  (END  OF  NUMBERS)  (3)  C3]     (3.111)
```

where (3) denotes the line number on which the unrecognised statement has been printed.

Whenever a comment such as (3.111) which has a reference number of 1, is produced we say that FAPD has resulted in a partial analysis of the design. Consequently the coded version of such a design cannot be tested on a computer. If analysis had resulted in a complete analysis then the statements for opening and closing input and output channels would need to be inserted before the program could be executed. However for the sake of clarity the code generator does not do this.

This chapter has detailed a framework aimed at analysing a program design. Throughout the discussion attention has been drawn to those factors which impose limitations on FAPD's scope, and discussion of these is continued in the concluding chapter. In order to test FAPD it has been implemented within a system called DACE (which is a Design Analysing and Commenting Environment). The following chapter will now discuss details of this implementation before the results from its analysis are discussed in chapters 5 and 6.

4. IMPLEMENTATION OF DACE

4.1 Relationships between System and FAPD

Before some of the implementation details of DACE are considered it is necessary to determine the relationship between the system itself and FAPD which it tests. Lukey [Lukey 1978] describes his system, PUDSY, as an implementation of a model of part of his theory and DACE can be described in a similar manner.

The preceding chapter described how a set of general classes can be used to categorise the kinds of statements found within a program design. Within each class there are a set of specific instances, called class instances which are used for recognising common statements and phrases. Because the system incorporates a particular set of class instances it is said to represent a model of FAPD. In order to test the validity of FAPD it is considered unnecessary to incorporate within the system a comprehensive library of all statements and phrases which could be recognised.

In order to analyse a design we must have some method for recognising when a class instance appears in a design. Later sections in this chapter describe how this has been achieved by using facilities available in the programming language MICRO-PLANNER. The choice of this language is therefore a decision concerned with how FAPD can best be implemented. As far as FAPD itself is concerned, class instances could be implemented by other programming techniques in other languages. A second implementation decision is the choice of the program design's target language. Preceding chapters have already given reasons why ALGOL 68C has been chosen, but as far as FAPD is concerned, so long as the same method of program design is used and programs in the target language can be represented by the FAPD's assertion language, then other target languages could have been chosen. It is estimated that modifying the system to accommodate a different target language would take six to eight man weeks.

A third detail of this implementation concerns the programming language subset, which is used to implement a program design. The system described in this thesis uses a subset of ALGOL 68C. Hence the fact that this subset contains integer and boolean, but not real variables, is an implementation decision. The assertion language which has been implemented in this study can represent the following features of an ALGOL 68C program:

a) loops of the following format:

```
WHILE -- DO - OD
```

b) conditionals of the following format:

```
IF -- THEN -- ELSE -- FI
```

c) assignment statements

d) read statements

e) print statements

f) boolean expressions

g) arithmetic expressions

h) integer and boolean variables

i) constant integer values

j) one-dimensional integer arrays and array elements

which are considered to be sufficiently comprehensive for analysing a wide variety of program designs.

4.2 Facilities Available

Prior to implementing FAPD, decisions were required about which computer and programming language of those available, were most appropriate for the development of the system. At the time research commenced the following machines were available: the University of Aston's ICL 1904S computer, the Computer Centre's Prime 250 mini-computer, the University of Birmingham's DEC 20/60 computer and the CDC 7600 and ICL 1904S computers at the University of Manchester Regional Computer Centre. Of these, the DEC 20/60 computer was chosen because it is a powerful, interactive machine which also provided three AI programming languages. These were LISP [Bobrow et al 1973, Quam and Diffie 1972, LeFaivre 1978], MICRO-PLANNER [Baumgart 1972] and CONNIVER [McDermott and Sussman 1974]

The programming language LISP provides more comprehensive facilities for word/character handling than languages such as ALGOL 68 and FORTRAN. In addition it also aids the interactive development of a system by providing facilities such as a LISP editor, for editing LISP functions, and powerful TRACE and BREAK packages to aid debugging. The Rutgers/UCI version of LISP, which is available on the DEC, allows functions to be either interpreted or compiled. Compiled functions can improve execution time by a factor of twenty and in addition, take up less memory space.

MICRO-PLANNER is a LISP-based language based on Carl Hewitt's robot language PLANNER [Hewitt 1969]. Like the LISP system, MICRO-PLANNER also provides special editing and tracing packages, however unlike LISP it cannot be compiled. The principal feature of the language is the facility to call functions by name or pattern. Thus every function must have a pattern. An example of a pattern could be:

```
        [#REF           (THV  X)]                (4.1)
```

where (THV X) is a MICRO-PLANNER variable which can take any value. Hence whenever statements of the following forms are made:

```
        (THGOAL   [#REF   TOTAL]  (THTBF THTRUE))   (4.2)
        (THGOAL   [#REF   SUM]    (THTBF THTRUE))   (4.3)
```

then all functions with pattern (4.1) are invoked until the correct function is found. Conversely a statement such as:

```
        (THGOAL   [#VAR   TOTAL]  (THTBF  THTRUE))  (4.4)
```

which does not match pattern (4.1) would not be called. This method of pattern directed invocation has been used to great effect by researchers in AI, such as Winograd [Winograd 1972] and Charniak [Charniak 1973]. Since analysing a program design involves recognising patterns of design statements, MICRO-PLANNER seems an ideal choice for implementing FAPD. The control structure simulates a depth first search of a tree, which backtracks automatically whenever an impasse is reached. Backtracking in this fashion is undirected and could be made more efficient if controlled by the programmer [Bobrow and Raphael 1974].

This criticism led to the development of CONNIVER which allows the programmer to determine how a program should continue once an impasse is reached. Although CONNIVER now seems to be the preferred language, it was decided after an initial investigation to implement FAPD using LISP and where appropriate MICRO-PLANNER. The system does not require the automatic backtracking mechanism of Micro-Planner and so the criticism referred to above is not applicable to this implementation.

4.3 User Interaction

DACE runs on the University of Birmingham' DEC 20/60 computer under the control of the TOPS-20 operating system and takes up 65K words of a 36-bit computer store. Chapter 3 gave details of four phases in the analysing process and diagram 11 shows how they have been implemented in terms of four system modules. A box represents a set of programs and arrows denote how data flows from one to the other. It can be seen that the modules operate in a sequential manner except for semantic analysis and generation of comments which run in parallel. The series of arrows emanating from the former has been used to indicate that whenever a statement or phrase is analysed, the results are passed on to the module for generating comments before the next statement is analysed. Module 1 is written in LISP whereas the other three are all implemented in MICRO-PLANNER. Because the second and third modules run in parallel DACE operates by entering the LISP system once, and the MICRO-PLANNER system twice.

In order to enter a program design the LISP system must be called and module 1 loaded. Whenever the DEC's LISP system is called a special initialisation file is automatically loaded. When operating under the author's usernumber this special file asks the user if he wishes to use DACE and if this is so then the LISP functions

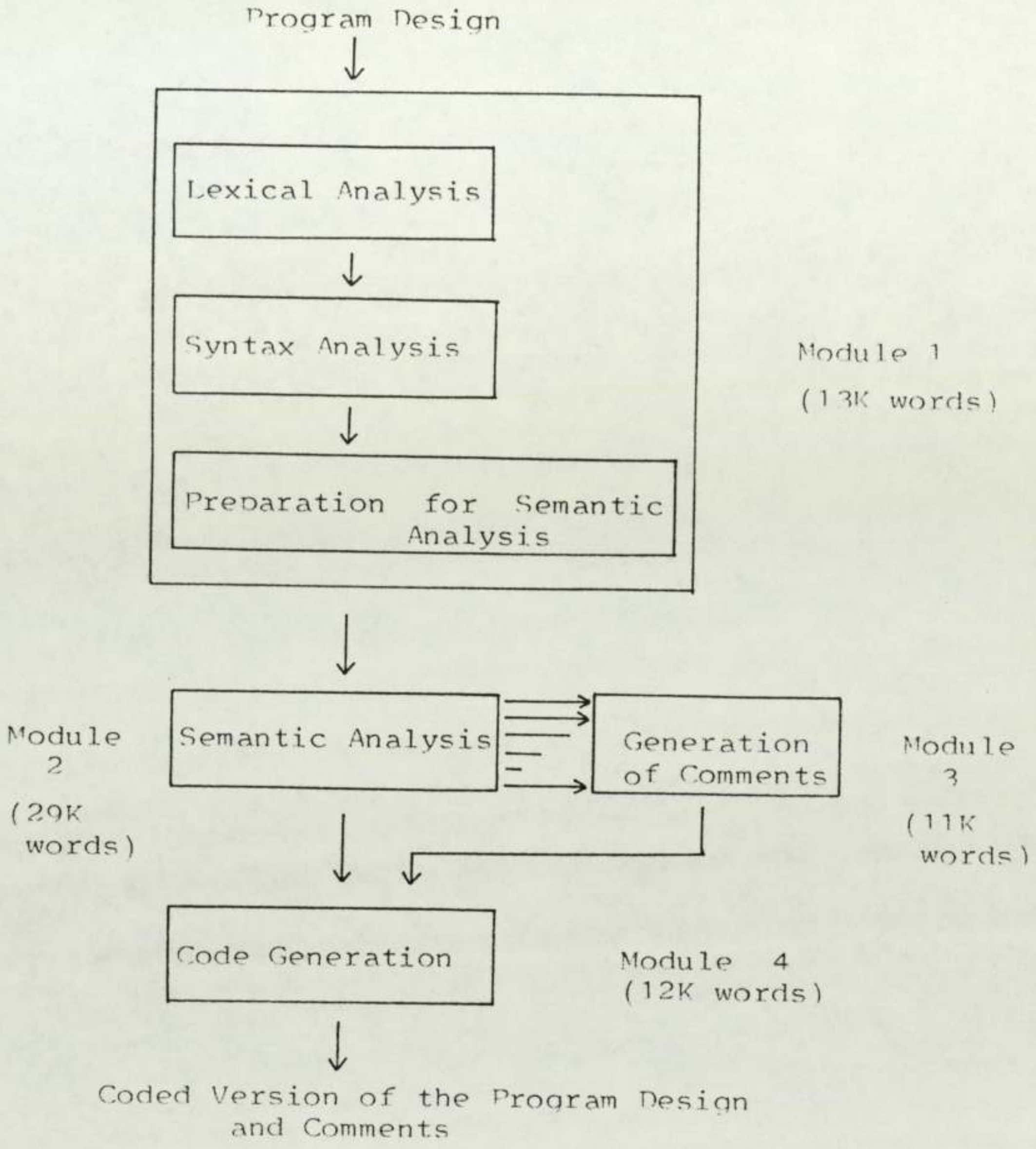


Diagram 11

The Structure of the System

contained in module 1 are entered and the user is invited to enter his design. Diagrams 12, 13 and 14 show how a user interacts with the system and in each diagram the user's responses have been underlined. Diagram 12 shows how the design need not be typed in in any particular format since DACE re-prints it with appropriate indentations.

When entering a design certain rules should be obeyed. These may be summarised as:

a) the design should be terminated by the string *** which must appear at the start of a new line. This string must not be followed by a space, otherwise the design is terminated incorrectly and the user is given another invitation to type. At this point the terminating string should be typed in correctly. DACE now parses the program design up to and including the first occurrence of the terminating string ***. Provided an incorrect termination is rectified in the manner just described it does not prohibit further analysis. However it can lead to distortions in the pretty-printed version of the design. The requirement that the terminating string should not be followed by a space occurs because of the way in which the LISP system reads a line of data.

b) Diagram 12 shows how two or more consecutive design statements should be separated by the string**. This string is used to print consecutive statements on different lines. If the separator string between two statements is omitted then the system will not only print the statements on the same line but will

```
@LISP

Do you wish to use DACE a system for analysing
and commenting
upon some simple program designs ? When the system prints:-
*
please type Y or N followed by the < return > key

*Y

Please input the design when the system types :-
-
At the end of a line press <return> and the system
will again respond with :-
-
In order to terminate the input type the string ***
at the start of a newline followed by <return> and
please ensure no spaces follow that string

-READ N AND INITIALISE I TO 1
-WHILE I IS LESS THAN OR EQUAL TO N
-DO PROCESS DATA FOR EMPLOYEE ** INCREMENT I OD
-***

The design has been entered and
syntax analysis has started

The design is as follows :-

READ N AND INITIALISE I TO 1
WHILE I IS LESS THAN OR EQUAL TO N
DO
      PROCESS DATA FOR EMPLOYEE **
      INCREMENT I
OD
***

Syntax analysis of this design was successful
The syntax tree is being amended
The syntax tree has been successfully amended

Do you wish to carry on ? Please type
Y or N followed by < return >

*Y

When the system types:-
@
please respond by typing PLNR < return >
@
```

Diagram 12

User Interaction With Module 1 of the System also encounter difficulty in differentiating between the statements. The latter difficulty arises because the syntax analyser uses the string as an indication that parsing in the current part of the syntax tree should be complete and thus parsing of the next statement can be initiated. If it is not used, then the syntax analyser first of all tries to parse the next token according to the grammar of the current part of the syntax tree. Failure to do this implies that the current part of the syntax tree has been successfully parsed or that parsing must continue at another point in the syntax tree. Hence the use of ** increases the efficiency of the syntax analyser.

c) ALGOL 68C allows the use of a single quote or a period to denote a reserved word such as IF, WHILE etc. but in a program design these are unnecessary and indeed illegal.

d) All words and characters must be in upper-case.

e) Within a print design statement any sequence of words which the user intends to print as text should start and finish with the character #. ALGOL 68C uses double quotation marks for the same purpose, however this is difficult to implement in a LISP based system because the LISP READ function will read in anything enclosed in double quotation marks as a single item. Consequently, double quotation marks are not recognised and would be treated by the system as a user defined variable name.

Failure to comply with rules (c), (d) or (e) can result in a design being analysed as syntactically incorrect.

After the design has been entered, lexical and syntax analysis are undertaken and, if successful, the syntax tree is amended in preparation for semantic analysis. Diagram 12 shows how module 1 has been successfully completed and the user informed of the steps necessary for entering the next phase. In diagram 13, the second and third modules have been loaded automatically by a special MICRO-PLANNER initialisation file. Typing (START) causes semantic analysis to commence and any comments to be noted. To print the results from this phase in a readable form, the MICRO-PLANNER system must be left and re-entered with module 4 loaded.

Diagram 14 shows how typing (PRINT-CODE) causes the results to be entered before the program and comments are printed at the terminal. The program and comments are also filed so a hard copy is available if desired. Collectively, diagrams 12, 13 and 14 depict a complete terminal session with DACE.

4.4 Pre-Semantic Analysis within DACE

Pre-semantic analysis within DACE is achieved by carrying out lexical analysis, syntax analysis and preparation for semantic analysis in a sequential manner. The principal feature of this implementation is the way the syntax and semantic definitions have been divorced from the procedures that use them. The syntactic format of a program design is specified by its grammar (see Appendix B). This grammar contains a set of rules written in a modified form of BNF. A typical rule of this grammar is:

```
@PLNR

MICRO-PLANNER
>>>  READING (PLNR . INI)
THINIT

When the system prints:-
O*
please respond by typing (START) < return >

>>>  TOP LEVEL
LISTENING  THVAL

O*(START)

The semantic analyser has now been entered


Semantic analysis is now complete



Do you wish to carry on ? Please type
Y or N followed by < return >
*Y
When the system types:-

@

please respond as you have just done
by typing PLNR < return >
@
```

Diagram 13

User Interaction With Modules 2 and 3 of the System

```
@PLNR
MICRO-PLANNER
>>>  READING (PLNR . INI)
THINIT

When the system prints:-
O*
please respond by typing (PRINT-CODE) <return>

>>>  TOP LEVEL
LISTENING  THVAL

O*(PRINT-CODE)

The design is as follows :-

READ N AND INITIALISE I TO 1
WHILE I IS LESS THAN OR EQUAL TO N
DO
      PROCESS DATA FOR EMPLOYEE **
      INCREMENT I
OD
***

A coded form of the design is:-

0   'BEGIN    'INT      I,N;
1             READ (N) ;
2             I := 1;
3             'WHILE  I <= N
4             'DO     < PROCESS DATA FOR EMPLOYEE >;
5                       I := I + 1
6             'OD
7   'END

The following are some comments on the above:-

1  Re line 4 : The design gives insufficient
   detail to analyse
   <PROCESS DATA FOR EMPLOYEE>
2  The design does not contain any output statements
   Before the coded version could be run one or more
   PRINT statements need to be inserted

Analysis is now complete. Your design
together with the coded
version and comments are stored in CODE.RES
Do you wish to leave the MICRO-PLANNER system ?
Please type
Y or N followed by < return >
*Y
@
```

Diagram 14

User Interaction With Module 4 of the System

< loop> ::= <while> <boolean expression> <do>
<series> <od> (4.5)

where items in angled brackets are non-terminals which are further defined elsewhere in the grammar (see Appendix B).

The syntax analyser within DACE uses a technique called top-down analysis. This technique uses the grammar to build a syntax tree by starting from the top-most definition and working downwards in a depth first manner. At each stage an attempt is made to replace the left most non-terminal in the syntax tree by a suitable expression derived from the rules of the grammar.

A basic difficulty in top-down analysis is encountered when a rule employs left recursion. For instance if the definition for a series of design statements was to be written as:

<series> ::= <series> <**><statements> |
<statements> (4.6)

then because the term <series> is recursively defined, searching would continue indefinitely. This problem is overcome by re-writing rules in a right recursive manner. Thus the above definition becomes:

<series> ::= <statements><** ><series> |
<statements> (4.7)

An alternative solution is to rewrite definitions in a modified BNF form which allows the use of two additional features. These are {X} which denotes zero or more occurrences of X and [X] which denotes an occurrence of X is optional. This approach has been used to specify the grammar of a program design and so expression (4.7) can be expressed using iteration instead of recursion as follows:

<series> ::= <statements> {<**> <statements>}   (4.8)

One approach to top-down parsing is recursive descent which involves writing a recursive procedure corresponding to each non-terminal of the grammar. The method does not allow back-up and thus once an item is parsed an alternative parsing cannot be considered. Thus if an impasse is reached a syntax error has been detected and an appropriate error message can be made.

A second approach to top-down parsing uses a set of general procedures driven by a representation of the grammar. In order to implement a syntax analyser, the latter of these two approaches was chosen. The reason for this is that backup must be used whenever a recognised word has been used as a variable name. For instance consider the use of NEXT in the following statements:

```
SET  NEXT  ELEMENT  TO  1                    (4.9)
SET  NEXT  TO  1                             (4.10)
```

In statement (4.9) NEXT is used as an adjective whilst in (4.10) it is used as a variable name. However in statement (4.10) the fact that NEXT is used as a variable is not apparent until the word TO is analysed, at which point it is necessary to back-up and revise the parsing of NEXT.

Although this approach also has the advantage of easy modification, its persistent use of back-up means it is often inefficient. It is also poor at handling errors because it is unable to determine the point at which an error occurred (c.f. top-down analysis using recursive descent). Consequently whenever DACE discovers an error the design is re-printed for the user, together with a general error message indicating a syntax error has been found. In this situation the analysing process is not able to proceed and the user is therefore not able to load and execute the semantic analyser.

If parsing is successful then the syntax tree can be amended in preparation for semantic analysis. In order to achieve this, each non-terminal of the grammar has a semantic definition (see Appendix B). This definition is used by a set of procedures to form a series of classes which the semantic analyser can recognise. By adopting this approach the semantic definitions can be altered without changing the procedures that use them. Consequently as the system was extended in order to analyse an increasing variety of examples, it was modified more easily than it would have been with the definitions procedurally embedded. Once the syntax tree has been amended, the operation of module 1 is complete. The LISP system is now exited and the MICRO-PLANNER system is entered in order to start the semantic analysis and possible generation of comments.

4.5 Semantic Analysis, Generation of Comments and Code Generation within DACE

The remaining three modules of DACE are discussed in this section because they are all implemented in MICRO-PLANNER. Chapter 3 defined a class instance to be a structure which represents statements often found in a program design, together with a function that implements the structure in terms of a particular programming language. Consequently this section is concerned with how a class instance can be coded using a MICRO-PLANNER theorem.

Let us recall (see section 3.2.3) that in the pre-semantic analysis phase, attempts are made to convert the syntax tree for any assignment design statement into the following general form:

```
[#ASS   <assignment command word>  ARGUMENT
                 <separator>  ARGUMENT]                (4.11)
```

Consequently whenever semantic analysis discovers a structure similar to (3.11) in the amended syntax tree, the appropriate class instance must be called to derive its meaning. MICRO-PLANNER allows theorems to be called by a pattern and so commands can be written which have the effect of searching through all the known theorems for any with a pattern which matches (4.11).

The theorems in diagram 15 are typical of those used by DACE. In each case the theorem's pattern has been underlined. From this diagram we can see that the pattern of TC-ASSERT- #ASS matches (4.11) whereas those of TC-ASSERT- #READ and TC- #ASS-ASSIGN do not. Consequently TC-ASSERT - #ASS is invoked. Because DACE is a model of FAPD a particular class instance may not be represented in the set. To overcome this, TC-ASSERT-#READ and TC-ASSERT-#ASS are general theorems which are used to deal with all possible examples of read and assignment design statements respectively. If the amended tree contains something of the form:

```
[#ASS    ASSIGN     <first argument>  TO
                    <second argument>]       (4.12)
```

then the procedure TC-ASSERT- #ASS is called. This

```
(DEFPROP TC-ASSERT-#READ
 (THCONSE (WORD A1 SEP A2 ANS N) (#READ $?WORD $?A1 $?SEP $?A2)
         (THOR [THSETQ $?ANS
                      (THGOAL (#RDM $?WORD $?A1 $?SEP $?A2)
                                  (THNODB)
                                 (THTBF THTRUE))]
               [THAND [THSETQ $?N (GENSYM)]
                      [THASSERT (#READ $?WORD $?A1 $?SEP $?A2 $?N)]
                     [THSETQ $?ANS (LIST $?N)]])
        (THSUCCEED THEOREM $?ANS))
 THEOREM)

(DEFPROP TC-ASSERT-#ASS
 (THCONSE (WORD A1 SEP A2 ANS N) (#ASS $?WORD $?A1 $?SEP $?A2)
       (THOR [THSETQ $?ANS
                      (THGOAL (#ASM $?WORD $?A1 $?SEP $?A2)
                                  (THNODB)
                                 (THTBF THTRUE))]
              [THAND [THSETQ $?N (GENSYM)]
                     [THASSERT (#ASS $?WORD $?A1 $?SEP $?A2 $?N)]
                     [THSETQ $?ANS (LIST $?N)]])
        (THSUCCEED THEOREM $?ANS))
 THEOREM)

(DEFPROP TC-#ASS-ASSIGN
 (THCONSE (A1 A2 SEP ARG1 ARG2 ANS) (#ASM ASSIGN $?A1 $?SEP $?A2)
       (THOR [THAND [THSETQ $?ARG1 $?A2]
                    [THSETQ $?ARG2 $?A1]
                    [THSETQ $?ANS (DF-ASSERT-#ASS)]
                      [THSUCCEED THEOREM $?ANS]]
              [THFAIL THEOREM]))
 THEOREM)
```

Diagram 15

The Class Instances TC-ASSERT- #READ,TC-ASSERT- #ASS and TC- #ASS-ASSIGN procedure alters the class name from #ASS to #ASM (so that it does not call itself) and then searches for a class instance with the following structure:

```
[#ASM    ASSIGN    <first argument>   TO
                   <second argument>]         (4.13)
```

Diagram 15 shows that TC- #ASS-ASSIGN is the class instance which matches this new structure. It analyses the arguments and if successful, makes the necessary assertions. If the statement cannot be analysed fully or the particular class instance is not known then TC-ASSERT- #ASS acts as a safety net. The following assertion, which indicates that semantic analysis has failed, is then generated:

```
[#ASS    ASSIGN    <first argument>   TO
                   <second argument> AS1]  (4.14)
```

This example illustrates how implementation and FAPD differ slightly. Whereas FAPD (see section 3.3.1) states that class instances are represented by, for example:

```
[#ASS    ASSIGN        <first argument>  TO
                       <second argument> ]     (4.15)
[#ASS    INITIALISE    <first argument>  TO
                       <second argument> ]     (4.16)
```

for the reasons outlined above, these are represented within the system as:

```
[#ASM    ASSIGN        <first argument>  TO
                       <second argument> ]    (4.17)
[#ASM    INITIALISE    <first argument>  TO
                       <second argument> ]    (4.18)
```

There are three different forms of MICRO-PLANNER theorems. However the requirements of the system mean that only two of these forms need to be used. These are consequent and antecedent theorems. So far the discussion has concentrated on consequent theorems (indicated by the definition THCONSE on the second line of each theorem). Whenever a search is made through a set of consequent theorems, that search is terminated as soon as a theorem succeeds. Conversely, whenever antecedent theorems (denoted by the definition THANTE) are called by pattern, <u>all</u> theorems are tested for a pattern match regardless of whether any have already succeeded.

Let us now consider how these different attributes can be used within the system. When a design statement is analysed the set of class instances is searched for a particular instance. In this respect consequent theorems provide an ideal method for representing the majority of class instances known to the semantic analyser. For every assertion used to represent a piece of code or a comment, the code generator has a theorem which prints the assertion in a readable form. For this reason consequent theorems are also used as the basis for code generation.

As soon as a design statement has been analysed, the results (in the form of one or more assertions) are passed on to the routines for generating comments. Comments are noted by class instances whose structure matches the assertions produced by semantic analysis. However because a result may provide several implications, the search must continue through <u>all</u> the appropriate class instances. For this reason any class instances used for generating comments are represented by MICRO-PLANNER antecedent theorems.

Section 3.3.3 defined an intermediary assertion to be an assertion which aids semantic analysis or the generation of comments. These assertions are often derived as incidental to the process of analysing a statement or phrase. This is a similar technique to that used to generate any comments and hence any intermediary assertions formed by the semantic analyser are also made by MICRO-PLANNER antecedent theorems.

This concludes a discussion of FAPD and its implementation. The next two chapters give details of the results obtained from the operation of DACE. The final chapter uses these results to draw some conclusions concerning both FAPD and its implementation.

5. RESULTS FROM ANALYSING PROGRAM DESIGNS

This chapter discusses the results obtained from applying DACE to eleven program designs, carefully chosen so as to illustrate the scope of DACE. The first seven of these represent examples which DACE is able to analyse. The eighth contains statements which are beyond the scope of FAPD and consequently have been only partially analysed by DACE. The last three examples represent those designs which cannot be analysed because they do not conform to our definition of a program design. Appendix D contains the results from analysing a further 24 examples.

The examples which have been used to test FAPD and the system have been derived from various sources. Examples 1, 2, 3, 5, 6 and 9 were taken from problems given to computer science students. Examples 4, 7 and 10 were derived by the author and examples 8 and 11 were taken from the literature. Some examples have been modified slightly in order to conform to the requirements of the system. For example, the string ** has been inserted to clearly distinguish between consecutive statements and the loop structures specified in examples 8 and 11 have been altered from the PASCAL to the ALGOL 68C format. Generally speaking, the examples discussed in this chapter have been chosen because they provide a wide ranging examination of the scope of DACE. The format of some results has been modified slightly in order to accommodate the different page size required for this report.

5.1 Example 1

The program design shown in diagram 16 has been produced to meet the following problem specification:

> input two integer values and output the larger value. If the values are equal then a message indicating this should be printed together with the value.

The solution to this problem is important since it involves the use of an elementary programming technique, namely the nested conditional. Since the program design for this problem is relatively simple, it allows us to consider the complete process followed by DACE without having to refer to those processes responsible for analysing more complex aspects of a design.

For the sake of clarity, previous chapters have often shown results in a modified form. Consequently, Appendix A has been included to illustrate the actual results produced for this particular example by the routines for pre-semantic analysis, semantic analysis and generation of comments.

Diagram 16 shows the format of the results produced by the code generator. This format always follows a similar pattern with the user's design being followed by a coded version of the design together with any comments. The line numbers within the design (ie DS1 to DS14) have not been produced by the system but are inserted in all the examples given in this chapter so that the discussion can refer to particular statements. The program design printed in this results section is not necessarily in the same format as that typed in by the user since the code generator re-prints it with consecutive statements on

The design is as follows :-

```
(DS1)    INPUT A AND B
(DS2)    IF A IS LARGER THAN B
(DS3)    THEN
(DS4)         SET ANSWER TO A
(DS5)    ELSE
(DS6)         ASSIGN B TO ANSWER
(DS7)         IF A IS EQUAL TO B
(DS8)         THEN
(DS9)              PRINT # SUITABLE MESSAGE #
(DS10)        FI
(DS11)
(DS12)   FI
(DS13)   OUTPUT ANSWER
(DS14)   ***
```

A coded form of the design is:-

```
0    'BEGIN   'INT     ANSWER, B, A;
1             READ (A, B) ;
2             'IF    A > B
3             'THEN  ANSWER := A
4             'ELSE  ANSWER := B;
5                    'IF     A = B
6                    'THEN  PRINT ("SUITABLE MESSAGE")
7                    'FI
8             'FI  ;
9             PRINT (ANSWER)
10   'END
```

Diagram 16

Results From Analysing a Program Design Which Finds the Larger of Two Values different lines and any loops and conditionals suitably indented. Because the coded version is also pretty printed in a similar manner, adopting this approach makes it easier for the user to see the correspondence between the two forms.

All programs produced by DACE adopt a similar format. They are numbered, starting at line 0, and the opening lines always declare all integer variables, boolean variables and any array used in the design. Consequently, DACE does not have the ability to declare variables locally. The program is printed using upper case characters and any reserved words are preceded by a single quotation mark. Since analysis of this example did not produce any comments, a discussion of these is deferred until a later example.

The syntax analysis of this example was successful which means that DACE is capable of at least partially analysing the design. During this stage DACE has successfully used the grammar of a program design to determine that A has been used throughout as a variable name and not as the indefinite article which is its more common occurrence. The syntax tree has then been successfully amended and the words AND, THAN and TO, which are used in lines (DS1), (DS2) and (DS7) respectively, have all been discarded because they are superfluous to semantic analysis. In this example IS could also have been eliminated from lines (DS2) and (DS7) since the appropriate boolean operator can be derived from knowing the meanings of LARGER and EQUAL. However, if the word IS was to be ignored in a similar fashion to AND, then the meaning of the following statement could not be derived:

```
    IF A IS POSITIVE
        THEN  one or more design statements   FI  (5.1)
```

This is because the meaning of the phrase A IS POSITIVE would be represented by a list similar to (A POSITIVE). In the following statement:

```
    IF A AND POSITIVE ARE GREATER THAN O
        THEN  one or more design statements   FI  (5.2)
```

the meaning of A AND POSITIVE is also represented by an identical list and hence it would have been impossible to differentiate between the two phrases.

The top level of the amended tree will show that the design is comprised of three main items which are the read design statement in line (DS1), the conditional in lines (DS2) to (DS12) inclusive and an output design statement in line (DS13). The way in which these items are processed is shown in diagram 17 which contains the top-level function of the semantic analyser written in an ALGOL-like notation. This shows how the design is analysed from top to bottom and how DACE can analyse the design only if it has three class instances which correspond to the three structures in the amended tree. Once the appropriate class instance has been found and the meaning of a statement or construct has been derived, this meaning must then be considered within the overall context of the design. Diagram 17 shows that the meaning of a statement which is not contained within either a loop or a conditional is derived from class instances with the following structure:

```
TREE := amended syntax tree produced by pre-semantic
          analysis
WHILE  all the structures in TREE have not been
       considered
DO
       NEXT := next structure in TREE which has not
                 been considered
       IF there is a class instance corresponding to NEXT
       THEN
             RESULTS := list of one or more assertion names
                          produced by calling this class
                          instance
             WHILE all the assertion names in RESULTS have
                   not been considered
             DO    NEXT-ASSERTION := next assertion name in
                     RESULTS not yet considered
                   IF there is a class instance
                      corresponding to :
                       [DESIGN ARG  <value of NEXT-
                                       ASSERTION>]
                   THEN
                         TEMP := list of one or more
                          assertion names produced by
                          calling this class instance
                         amend RESULTS by replacing NEXT-
                          ASSERTION with TEMP
                         NEXT-ASSERTION := first assertion
                          name in the list called TEMP
                   FI
                   look for any implications of
                    incorporating NEXT-ASSERTION into
                    the overall design
             OD
       ELSE  c  NEXT is a structure which DACE cannot
                  recognise  c
             RESULTS := list of one or more names of
                          default assertions
       FI
       ANSWER := list of assertion names produced so
                   far from the analysis of TREE
OD
c  a coded version of TREE is now represented by those
   assertions whose names comprise the list ANSWER  c
```

Diagram 17

The Top-Level Function in the Semantic Analyser

[DESIGN ARG <assertion names>] (5.3)

The function for analysing statements in a loopbody is essentially similar to that shown in diagram 17. However instead of considering the meaning of a statement within the context of the overall design it considers the meaning within the context of a loop by looking for class instances of the form:

[LOOPBODY ARG <assertion names>
<#LOOP assertion name>] (5.4)

rather than (5.3). Similarly it will then look for any implications of incorporating the results into the loop rather than the overall context of the program design.

The results in diagram 16 show that each statement in the design can be implemented using a single statement in the target language. However, DACE is able to determine that the meaning of a statement such as:

FIND THE TOTAL OF THE VALUES OF THE ARRAY (5.5)

is described by more than one ALGOL 68C statement. Consequently the inner loop in diagram 17 is used to consider each of these in turn within the overall context of the design.

Now that these general points about DACE have been discussed, let us conclude the discussion of this example by considering the depth of analysis which it achieved in producing the results of diagram 16. The information required to do this has been derived from the following four sources:

a) DACE's vocabulary which at the time of writing consists of 110 words;
b) the grammar of a program design:

c) the semantic definitions of all non-terminals in the grammar; and

d) class instances. At the time of writing, the semantic analysis is based on 128 class instances. Any comments are generated by using the 26 class instances which together with the code generator can produce 18 different comments. Appendix C contains a comprehensive list of all class instances implemented in DACE.

The first three of these (ie a, b and c) are used during pre-semantic analysis to determine how consecutive words and phrases can be combined into meaningful units. The vocabulary and grammar are also used to determine that A is used as a variable name. Thus at the end of this stage, DACE has determined that lines (DS4) and (DS6) will both be implemented as assignment statements although it is not yet able to note the differing effect that SET and ASSIGN have on the treatment of the arguments A, B and ANSWER. Recognising differences such as this is the responsibility of the semantic analyser which uses two class instances of the form:

```
   [#ASM  SET  ARGUMENT  <separator>  ARGUMENT]   (5.6)
   [#ASM  ASSIGN  ARGUMENT   <separator>
                   ARGUMENT]                      (5.7)
```

where the meanings of ARGUMENT and <separator> are as defined in Chapter 3. In addition to class instances such as these, DACE also requires separate class instances for different words or phrases with similar meanings. Thus the meanings of lines (DS9) and (DS13) are derived from the following two instances:

[#PRM PRINT ARGUMENT] (5.8)

[#PRM OUTPUT ARGUMENT] (5.9)

As later examples will illustrate, the results shown in diagram 16 do not illustrate clearly the degree of detailed analysis that had to be undertaken by DACE. For instance it had to determine that the variables A and B have been defined prior to their use in lines (DS2), (DS4), (DS6) and (DS7), and in addition that the value assigned to ANSWER in line (DS6) did not overwrite the value assigned to the same variable in (DS4).

5.2 Example 2

The program design shown in diagram 18 was produced in reply to the following problem specification:

> find the sum and average of a list of integer values. The list is contained in a data file and is terminated by a zero.

The analysis of this example will be discussed by considering each line of the design in turn.

(DS1) has been recognised as a read design statement with two arguments - FIRST VALUE and X. DACE first of all attempts to analyse the word VALUE without taking into consideration the context in which it appears. Since VALUE has not been analysed prior to the current line, it is assumed that it is used in this line as a variable. Hence at this stage, the analysis of VALUE is identical to its analysis of the following statement:

SET VALUE TO 3 (5.10)

The phrase FIRST VALUE is then considered. In this example because VALUE has been analysed as a single

The design is as follows :-

```
(DS1)    GET FIRST VALUE INTO X **
(DS2)    INITIALISE SUM TO 0 AND I TO 1
(DS3)    WHILE X IS NOT EQUAL TO 0
(DS4)    DO
(DS5)          ADD THIS VALUE TO SUM SO FAR **
(DS6)          INCREMENT I **
(DS7)          GET NEXT VALUE
(DS8)    OD
(DS9)    DIVIDE THE SUM OF VALUES
                BY THE NUMBER OF VALUES **
(DS10)   OUTPUT THE RESULT
(DS11)   ***
```

A coded form of the design is:-

```
0    'BEGIN    'INT     I, SUM, X;
1              READ (X) ;
2              SUM := 0;
3              I := 1;
4              'WHILE  X /= 0
5              'DO     SUM := SUM + X;
6                        I := I + 1;
7                        READ (X)
8              'OD  ;
9              PRINT (SUM % I)
10   'END
```

Diagram 18

Results From Analysing a Program Design Which Finds the Average of a List of Values variable, the adjective FIRST is ignored. However given the following pair of statements:

INPUT TWO VALUES (5.11)

MULTIPLY THE FIRST VALUE BY 36 (5.12)

DACE would recognise that in (5.12) the use of FIRST is significant and would use it to distinguish between the two variables in (5.11). The second argument of (DS1) - X - is an unrecognised word and syntax analysis has shown that in this example it has been used as a variable. After analysing these two arguments their meanings are represented by the following three assertions:

[#VAR VALUE V1] (5.13)

[#VAR X V2] (5.14)

[#REFV VALUE (V1)] (5.15)

the last of which is an intermediary assertion indicating that VALUE is used to refer to a variable of the same name. The meanings of FIRST VALUE and X are then considered together, within the context of a read design statement and in so doing it follows that in this context, VALUE is used not as a variable but as a reference to the contents of X. Consequently assertion (5.13) is erased and the intermediary assertion (5.15) is amended to:

[#REFV VALUE (V2)] (5.16)

The meaning of the current line is then denoted by the following representation of an ALGOL 68C read statement:

[#READ (V2) AS1] (5.17)

(DS2) is a design statement which illustrates one of the weaknesses of pre-semantic analysis. The amended syntax tree of this statement is comprised of the following two structures:

```
[#ASS  INITIALISE  (( #VAR(SUM)))  TO
          (( #CONST(0)) ( #VAR(I)))]                   (5.18)
[#ASS  NILL  NILL  TO  (( #CONST(1)))]                 (5.19)
```

However in order to convey the correct meaning the amended syntax tree should contain the following:

```
[#ASS  INITIALISE  ((#VAR(SUM)))  TO
                   ((#CONST(0)))]                      (5.20)
[#ASS  INITIALISE  ((#VAR(I)))    TO
                   ((#CONST(1)))]                      (5.21)
```

The principal reason why (5.20) and (5.21) are not produced is that DACE amends the syntax tree in a single pass. Consequently the decision as to whether an argument appears on the left hand or right hand side of an assignment statement is often unclear at this stage. For instance, because words such as BOTH and RESPECTIVELY are effectively ignored in the following two statements, the use of COUNTER3 is ambiguous until it is considered within the overall context of the statement:

```
INITIALISE COUNTER1 AND COUNTER2 TO
        1 AND COUNTER3 RESPECTIVELY                    (5.22)
INITIALISE COUNTER1 AND COUNTER2 BOTH
        TO 1 AND COUNTER3 TO 2                         (5.23)
```

In statement (5.22) the value of COUNTER3 is being used, viz:

```
COUNTER1:= 1;  COUNTER2 := COUNTER3;                   (5.24)
```

whereas in (5.23), COUNTER3 is being assigned a value, viz:

```
COUNTER1:= 1;  COUNTER2 :=1; COUNTER3:=2;              (5.25)
```

In order to produce structures (5.20) and (5.21) the results would need to be revised at the end of statement (DS2), that is DACE would need to undertake more than one pass of a syntax tree (c.f. single and multiple pass compilers). However if DACE was extended to do this then syntax and semantics may have to be integrated as well. For example, given the statement:

SET A AND B TO C AND D AND E TO 1 (5.26)

it is unclear whether this means:

A := B := C; D := E := 1; (5.27) or

A := C; B := D; E:= 1; (5.28)

A decision between these two alternatives could be based on, for instance, whether or not the variable D had been assigned a value prior to the current line. If it had, but that value had not been used, then statement (5.28) would be chosen instead of (5.27). However this approach is obviously based on a dubious assumption. Since the analysis of statements such as (DS2) and (5.26) is complex, an approach based on this method would require considerable research for its implementation and evaluation.

However, since the amended syntax trees shown in (5.18) and (5.19) are produced, DACE has two class instances which recognise that these particular structures actually mean the same as (5.20) and (5.21). The arguments SUM, O, I and 1 are all considered in a similar manner to the arguments of the previous read design statement. That is, a first attempt is made at their implementation which may be subsequently revised in the light of more information. Hence a first attempt to analyse SUM assumes it is a variable name which in this case happens to be correct. However, if a program design specifies that the values held in the elements of an array are to be added together and then the following statement is met:

OUTPUT THE SUM (5.29)

DACE will revise the initial assumption and will determine correctly that SUM refers to the previous arithmetic operation rather than a variable name.

The word TO in statements (DS2) and (DS3) is analysed differently and hence two definitions of this word must be incorporated into the dictionary. Because (DS2) is an assignment design statement which starts with INITIALISE, TO has been analysed as a separator (see section 3.2.2.3). This form of analysis means that the semantic analyser is able to determine which of the arguments SUM, 0, I and 1 appear on the left hand side and right hand side of an assignment statement. Consequently whenever TO is used as a separator its role is significant and therefore cannot be ignored. However the meaning of (DS3) can still be derived even when IS and TO are ignored. Hence according to FAPD the use of TO in (DS3) is found to be insignificant and as a result it is discarded before the semantic analyser is entered.

(DS5) is a statement which DACE has recognised will be implemented as an arithmetic expression. Again the arguments are considered individually with VALUE being the first to receive attention. In order to be consistent DACE must be able to detect the previous analysis of VALUE.

This is achieved by referring to the intermediary assertion (5.16) which shows that VALUE has previously been used to refer to X. Therefore the assumption is made that VALUE retains the same role in statement (DS5).

Since the words SO and FAR have not been met previously DACE must consider whether they could be user defined variable names. However since they do not comply with the grammar of a program design they are redefined as words that can be ignored and hence they are discarded before semantic analysis is initiated. After successfully forming the ALGOL 68C arithmetic expression corresponding to the current line, it must be incorporated into an assignment statement. Because (DS5) is within a loop, DACE recognises that this describes a summation and therefore requires the following form of an assignment statement:

```
        SUM :=  SUM + X                                  (5.30)
```

In order to do this DACE considers if there are any arithmetic expressions of the following forms which will be executed on each loop iteration:

```
        SUM  +  <variable name>                          (5.31) or
        TOTAL  + <variable name>                         (5.32)
```

Hence if the statement had been, say:

```
        ADD THIS VALUE TO A                              (5.33)
```

then DACE would have created the following statement:

```
        IDR01 :=  X + A                                  (5.34)
```

where IDR01 is a variable name generated by the system.

(DS6) is assumed to mean:

```
        INCREMENT   I  ( BY 1 )                          (5.35)
```

Since this appears within the loopbody, it is assumed by DACE that I is a loopcounter which is used to define the number of times statements (DS5) and (DS7) have been executed.

(DS7) is very similar to (DS1) except that the former does not specify the name of the variable into which the NEXT VALUE should be assigned. Since VALUE has been used previously in connection with the variable X then the NEXT VALUE is obtained by using a READ statement and assigning the inputted value to the same variable X.

(DS9) shows how SUM has been used in a different manner to its use in previous statements. Previously, SUM had been used as a noun but in the current statement its meaning has been derived by considering it within the context of the following phrase:

SUM OF VALUES (5.36)

Its use in this context implies that an arithmetic operation is to be performed. Consequently DACE attempts to form an arithmetic expression in the same way that it would for a statement such as:

(FIND THE) SUM OF A AND B (5.37)

Analysis of (5.36) indicates that a single variable, namely X, is being summed and that this cannot be represented by an arithmetic expression similar to the one considered for example in (5.33). As a result previous lines are considered to see if they can provide information which will facilitate the analysis. In doing so it is found that the variable SUM has been used previously to store the sum of consecutive values read into X. Similarly in order to derive the meaning of NUMBER OF VALUES, DACE looks back for any variables incremented at the same time that a value was read into X in the loopbody. Once I is found the appropriate arithmetic expression can be formed and incorporated into an assignment statement.

(DS10) is a print statement with one argument - RESULT. In a similar way to its analysis of SUM and VALUE, DACE first attempts to analyse it as a variable name. This meaning is rejected when it is considered within the context of the print statement, and an alternative meaning is considered, namely that RESULT refers to some previously defined value. However, because there is no previous reference in the design to RESULT, then the meaning of (DS10) is revised to:

PRINT THE RESULT (OF A PREVIOUS OPERATION) (5.38)

Since the last operation was the arithmetic expression of the previous line DACE assumes that the user intends the result of this operation to be printed. If the results of analysing (DS9) and (DS10) were now printed, the code generator would produce:

```
        IDR01 :=  SUM % I;                          (5.39)
        PRINT  ( IDR01 );                           (5.40)
```

where IDR01 is a variable name generated by DACE. However because the assignment statement has been generated by the system and not specified by the user, DACE combines these two statements into the single statement shown in line 9 of the coded version of the design.

The results of diagram 18 show that the methods used by DACE are adequate for analysing this design. However the following points should be mentioned in conclusion:

a) because VALUE has been used throughout to refer to a single variable, DACE has not used the adjectives FIRST, THIS and NEXT to help in the analysis of lines (DS1), (DS5) and (DS7) respectively. Consequently line (DS7) for instance would have produced a similar result if it had been written:

```
        GET A VALUE                              (5.41)
```

This is perfectly adequate for this example. However if VALUE had been used to refer to more than one variable then DACE would have realised the significance of an adjective much as FIRST and would have included it in the analysis;

b) in order to derive the meaning of SUM OF VALUES and NUMBER OF VALUES in line (DS9) reference had to be made to previous lines in order to associate their meaning with SUM and I respectively. The definition of these variables is then assumed by DACE to take the form shown in lines 5 and 6 of the coded version and any other definitions that may have been given in the program design are ignored;

c) the same representation is given to the meanings of VALUE and VALUES ie (V2) where V2 is the assertion name of the variable X, viz:

```
     [#VAR   X   V2]                              (5.42)
```

Consequently if the loop had been followed by a statement such as:

```
        IF THE VALUES ARE EQUAL TO 5
          THEN  one or more design statements
             FI                                   (5.43)
```

then DACE would have considered this to mean:

```
IF X = 5
    THEN   meaning of the design statements
        FI                                        (5.44)
```

which may not have been what the user intended. To remedy this situation, one possibility is to represent the meaning of VALUES by a list of the form:

```
(V2  V2  ....  V2)                                (5.45)
```

where V2 is the name of assertion (5.42) above. When the meaning of VALUES (ie (5.45)) is considered within the context of the boolean expression in statement (5.43) it is apparent that the user wishes to test whether <u>all</u> the values held by X are equal to 5. By using (5.45) to represent VALUES it is evident that the position of (5.43) is in error and that it should have been incorporated into the loopbody.

5.3 Example 3

Let us now consider a set of results which show that during analysis of a program design, an error has been detected which has led to a comment being generated about the coded version of the design. Comments are also generated:

a) to show the user that the results are inefficient (see Example 4);

b) to show an omission (see comment 5 in Example 5);

c) to indicate that the design cannot be analysed in full (see comment 3 in Example 5); and/or

d) to emphasise the relationship between the program design and the coded version of the design(see comment 6 in Example 6).

Diagram 19 contains a program design which is based on the following problem specification:

design a program which inputs three values representing a measurement in yards, feet and inches. Convert these values into a single measurement in inches and output the result.

Before discussing the results, let us consider the general format of the comments produced by DACE. Any comments are printed after the coded version of the design and are numbered for ease of identification. The majority of them also refer the user to the appropriate line number(s) in the code. The same comment always produces similar text, hence if the same error as that shown in diagram 19 is detected in another example, DACE will produce the same wording apart from different line numbers and variable name. Whenever an assertion representing a comment is generated it is linked to the results of semantic analysis by an assertion name and the appropriate line numbers are detected later during code generation. As each line of the program is printed, the code generator detects whether any comments have been assigned to the line. If so, then the assertion name is replaced with the current line number and the comment name is added to a list of any previous comments. Thus the position of a comment in the list is defined by its order of occurrence. Various lines in the program design are now discussed in order to show how the results have been produced.

(DS1) is recognised as a statement that will be implemented as a READ statement. From the results of

The design is as follows :-

```
(DS1)    INPUT THREE NUMBERS **
(DS2)    MULTIPLY THE FIRST NUMBER BY 36
           AND ASSIGN THE RESULT TO INCHES **
(DS3)    MULTIPLY THE SECOND NUMBER BY 12
           AND ASSIGN THE RESULT TO  INCHES **
(DS4)    ADD THE LAST NUMBER TO THE VALUE OF INCHES
           SO FAR AND OUTPUT THE RESULT
(DS5)    ***
```

A coded form of the design is:-

```
0    'BEGIN    'INT     IDR03, IDR02, IDR01, INCHES;
1              READ (IDR03, IDR02, IDR01) ;
2              INCHES := IDR03 X 36;
3              INCHES := IDR02 X 12;
4              PRINT (IDR01 + INCHES)
5    'END
```

The following are some comments on the above:-

1 Re Lines 2 and 3 : The value assigned
  to the variable< INCHES >
  has been overwritten without being used

Diagram 19

Results From Analysing a Program Design Which Converts Yards, Feet and Inches Into Inches pre-semantic analysis the first item that the semantic analyser attempts to analyse is NUMBERS. Since this has not been analysed prior to the current line, DACE considers it to mean a list of integer variables, the length of which is undefined. How can such a list be suitably represented? Since Dace's analysis of phrases such as ONE NUMBER, TWO VALUES etc. is confined to those which contain words (such as ONE and TWO) whose equivalent numerical value is in the range 1 to 10, a list of variables of undefined length can be represented by a list which is greater than ten elements in length. Consequently a list of undefined length is represented by:

```
        (V1   V2   ....   V11)                    (5.46)
```

where each element is the assertion name of a variable, viz:

```
        [#VAR    NILL    V1  ]                    (5.47)
        [#VAR    NILL    V2  ]                    (5.48)
                 etc.
        [#VAR    NILL    V11 ]                    (5.49)
```

After considering the phrase THREE NUMBERS, this list is shortened to (V1 V2 V3) and all the variables represented by the assertion names V4 to V11 are discarded.

From the coded version of the design it can be seen that DACE has generated the names IDR01, IDR02 and IDR03 to denote the variables relating to THREE NUMBERS. Generally speaking, DACE produces system defined variable names by using a special LISP function which it initialises to IDR00. Because this LISP function restricts all generated names to a length of five characters, any names generated by DACE must lie within the rage IDR01, IDR02 to IDR99.

(DS2), (DS3) and (DS4) are all similar since they actually comprise two design statements joined by the conjunction AND. Pre-semantic analysis has detected this and has divided each of them into two parts. Consequently the semantic analyser has considered each part in turn before combining the results into a single statement in the same way that the results of (DS9) and (DS10) were combined in Example 2.

These three lines also illustrate the point made in the previous section about the significance of adjectives. In this example FIRST, SECOND and LAST are all used to define which of the THREE NUMBERS is being referred to. Both the previous and current examples also contain words and phrases, the meaning of which can only be derived by considering previous lines in the program design. In this example DACE must consider the wider context in order to derive the meaning of RESULT in (DS2), (DS3) and (DS4). To achieve this, the semantic analyser maintains a list of those variable names relating to the last three arithmetic expressions mentioned prior to the current line. By doing so the meaning of a word such as RESULT, which is often used in program designs, can be derived without having to undertake an expensive search of preceding lines each time it is used. In Example 2 the meanings of phrases such as SUM OF VALUES and NUMBER OF VALUES were also derived in a similar manner from notes made by the semantic analyser when the following statements were formed in the loopbody:

```
SUM := SUM + X                                  (5.50)
I := I + 1                                      (5.51)
```

Generally speaking notes are made about those variables which are defined implicitly eg NUMBER OF VALUES in Example 2, rather than through an explicit definition eg INCREMENT I in Example 2.

The principal feature of this example is the way in which the comment relating to the variable INCHES has arisen. An obvious way of doing this would be to watch for consecutive pairs of statements, which assign values to the same variable. Whilst this approach would be adequate for this example DACE uses a more general method which can detect assignments to the same variable even when several statements separate the assignments. The basis of this method is that any variables defined in a read or assignment statement should have these values used before the variables are redefined. For instance in the current example DACE notices from (DS1) that the first value read in is subsequently used in (DS2) and thus any subsequent modification of FIRST NUMBER would be accepted. However since the variable INCHES is assigned a value in (DS2) and then again in (DS3) before the first value has been used, a suitable comment is generated.

The method outlined above is complicated when we come to consider loops and conditionals. For instance consider the use of the variable RESULT in the following fragment of a program:

```
RESULT := a value                          (5.52)
IF condition is true                       (5.53)
THEN                                       (5.54)
     RESULT := a value                     (5.55)
FI;                                        (5.56)
RESULT := a value                          (5.57)
```

DACE does not consider that the assignment in (5.55) overwrites the assignment in (5.52) since the former is contained within a conditional construct and thus the possible execution of (5.55) will be determined at run--time. In this respect statements (5.52) and (5.55) represent alternative values of RESULT. However when statement (5.57) is analysed DACE will make two comments indicating that both the previous values have been overwritten. To accomplish this DACE has noted that statement (5.55) is contained within a conditional whereas statements (5.52) and (5.57) are not. Once it is noted that statement (5.57) is not contained in either a loop or a conditional it is evident that the execution of this statement is unconditional. Consequently the execution of this statement must effectively overwrite any values assigned to the variable RESULT in previous lines. The consequences of this also need to be considered in the following program fragment:

```
RESULT := a value;                         (5.58)
IF  condition is true                      (5.59)
THEN                                       (5.60)
```

```
        variable name := RESULT                    (5.61)
    FI;                                            (5.62)
    RESULT := a value                              (5.63)
```

In this situation DACE gives the user the benefit of the doubt and even though the conditional may not be entered it is assumed the value assigned to RESULT has been used in line (5.61) before the variable RESULT has been redefined in (5.63). Consequently DACE would not generate a comment for a section of code similar to this.

From the discussion of this example four conclusions can be drawn:

a) firstly it has been emphasised how DACE often makes a first attempt at analysis which may be subsequently revised as the context widens. This approach is similar to that adopted by Sussman [Sussman 1975] for his automatic programming system - HACKER;

b) as semantic analysis proceeds, DACE makes assertions, the form of which is unique and predefined, to denote those variable names which it considers may be referred to by words or phrases rather than by name. For instance, a list of variable names is maintained so that whenever RESULT is met in a similar context to that found in lines (DS2), (DS3) or (DS4) the appropriate variable name can be derived. An alternative approach would be to search through preceding lines in an attempt to determine the meaning of such words and phrases. This approach would be more difficult to implement since the first attempt to derive the meaning of a word such as RESULT ignores the context in which it is used. Hence at this stage of the analysis it cannot detect easily those results obtained from analysing preceding lines. A search could be made only when the entire line is considered within the overall context of the design. Furthermore, the results may have to be modified when considered in this wider context;

c) although words similar to RESULT are first considered in isolation, their true meaning can only be derived after considering them in a wider context. For instance in (DS4) it is only after considering RESULT within the context of the statement OUTPUT THE RESULT that it is realised that the user wishes to access some previously defined value. In contrast, in the following statement the context of RESULT indicates that RESULT is being used as a variable name:

SET RESULT TO 0 (5.64)

d) constructs such as the loop and conditional complicate the process of determining whether a variable has been incorrectly overwritten before it is used. Because these constructs alter the top--down execution of consecutive lines the implications of forming an assignment statement say, can only be determined after taking into account whether or not that statement has been found in a loop or a conditional statement.

5.4 Example 4

Diagram 20 shows a program design which has been produced to solve the following problem:

Fibonacci numbers are defined as:

1, 1, 2, 3, 5, 8, 13 etc.

or $U_{N+2} = U_N + U_{N+1}$ where $U_1 = U_2 = 1$. Design a program to generate the first fifteen numbers of this series.

Let us consider the program design in some detail. (DS2) is comprised of an assignment and a print design statement. The first of these constructs is similar to that discussed in Chapter 3. The word BOTH is unrecognised and consequently an attempt is made to analyse it as a variable name. However since LASTRESULT was analysed as a variable name for similar reasons and the grammar specifies that two variable names cannot be used consecutively, it is concluded that BOTH has little significance and thus can be discarded. After analysis of line (DS2) has been completed, DACE has determined that RESULT appears on the left hand side of an assignment statement and consequently it is being used as a variable name. This is in contrast to its use in the previous two examples where it was used to refer to the result of a previous operation. Consequently when RESULT is found

The design is as follows :-

```
(DS1)    INITIALISE J TO 2 **
(DS2)    SET RESULT AND LASTRESULT BOTH TO 1
           AND PRINT THEIR VALUES
(DS3)    WHILE J IS LESS THAN 15
(DS4)    DO
(DS5)          OUTPUT THE TOTAL OF RESULT AND LASTRESULT
                 AND ASSIGN IT TO TEMP **
(DS6)          SET LASTRESULT TO RESULT
                 AND RESULT TO THE VALUE OF TEMP **
(DS7)          INCREMENT J
(DS8)    OD
(DS9)
(DS10)   ***
```

A coded form of the design is:-

```
0    'BEGIN    'INT     TEMP, LASTRESULT, RESULT, J;
1              J := 2;
2              RESULT := 1;
3              LASTRESULT := 1;
4              PRINT (RESULT, LASTRESULT) ;
5              'WHILE  J < 15
6              'DO    PRINT (RESULT + LASTRESULT) ;
7                       TEMP := RESULT + LASTRESULT;
8                       LASTRESULT := RESULT;
9                       RESULT := TEMP;
10                      J := J + 1
11             'OD
12   'END
```

The following are some comments on the above:-

1 Re Lines 6 and 7 : The expression
< RESULT + LASTRESULT >has been
unnecessarily duplicated Only one is needed

<u>Diagram 20</u>

<u>Results From Analysing a Program Design Which Generates the Fibonacci Series</u>

in (DS6), DACE assumes that reference is being made to the same variable.

The analysis of the print design statement is important for the way in which the phrase THEIR VALUES has been analysed. Research into natural language understanding has shown that understanding pronominal references of this sort is very difficult. The method used by DACE is to keep a note of the last subject(s) mentioned in the current block. For instance when THEIR VALUES is analysed, DACE recognises that the variable RESULT and LASTRESULT were both mentioned in the current line and that J was the subject of the preceding line. Since the meaning of THEIR VALUES must be plural, RESULT and LASTRESULT are chosen instead of J.

(DS5) shows how this technique has been used again to determine the meaning of IT. After forming the PRINT statement in line 6, the current subject and the one that IT is assumed to refer to, is then taken as the arithmetic expression contained in this statement. DACE recognises that the addition of RESULT and LASTRESULT has been carried out twice without either variable being assigned a new value. Consequently, a comment to this effect is made, the text of which is sufficiently general to make the user reconsider how the coded version might be improved. Thus the following implementation would be more efficient:

```
          TEMP := RESULT + LASTRESULT;            (5.65)
          PRINT (TEMP);                           (5.66)
```

To make this comment a technique has been used which is similar to that described in the previous section for detecting that the value of a variable has been over--written before it has been used. Whenever an arithmetic expression is formed DACE makes a note of all the variable names used in that expression. If that expression is then subsequently used without any of the constituent variables being redefined, then an identical value ensues. In this case, DACE would generate an appropriate comment regardless of the separation between the two invocations of the expression.

The principal features of this set of results are:

a) that DACE has analysed RESULT correctly as a variable name. This is in direct contrast to its use in previous examples;

b) that DACE has detected a statement in line (DS5) which is computationally inefficient;

c) the way in which the meanings of THEIR VALUES and IT have been derived.

Considering these three features, the derivation of the meanings of pronominal references presents the greatest difficulty. To derive their meaning, the ideal situation would be if DACE made use of the following:

i) a knowledge of the results obtained from analysing previous lines; and

ii) a knowledge of the problem specification.

For example, if we consider the phrase THEIR VALUES in line (DS2) we know from (i) that this phrase must refer to a combination ofJ, RESULT and LASTRESULT and furthermore that RESULT and LAST RESULT are mentioned within the same line. However it is from (ii) that we derive most of the significant information. From the problem specification we know that the Fibonacci series entails adding successive terms. Since RESULT and LASTRESULT denote the first two (and other) consecutive terms we can deduce that it is THEIR VALUES which are to be printed. In terms of the results displayed in diagram 20, DACE has used approach (i) above, but not (ii). Chapter 3 has already stated that a knowledge of the problem specification is outside the bounds of FAPD and other consequences of this are discussed in section 5.8.

5.5 Example 5

So far all the examples discussed in this chapter have been analysed fully. However this fifth example illustrates how DACE may sometimes only partially analyse a design. A partial analysis will result in a coded version of the design together with those design statements and/or phrases which DACE cannot analyse. The example shown in diagram 21 contains an array, which in terms of the constructs known to DACE, represents the most complicated, and therefore the most difficult, programming concept with which it can deal. The program design in diagram 21 is intended to solve the following problem:

a data file comprises eleven integer values.

Determine how many of the first ten values are equal

The design is as follows :-

```
(DS1)    INPUT TEN NUMBERS INTO AN ARRAY **
(DS2)    INITIALISE A COUNTER TO 0
(DS3)    WHILE NOT END
(DS4)    DO
(DS5)
(DS6)           IF NEXT ELEMENT IS EQUAL TO X
(DS7)           THEN
(DS8)                  INCREMENT THE COUNTER
(DS9)           FI
(DS10)
(DS11)   OD
(DS12)
(DS13)   ***
```

A coded form of the design is:-

```
0    'BEGIN     'INT      IDRO1, X, COUNTER;
1               [1:10] 'INT  ARRO1;
2               IDRO1 := 1;
3               'WHILE  IDRO1 <= 10
4               'DO     READ (ARRO1[IDRO1]) ;
5                        IDRO1 := IDRO1 + 1
6               'OD  ;
7               COUNTER := 0;
8               'WHILE  'NOT  < END >
9               'DO     'IF    ARRO1[< UNDEFINED >] = X
10                         'THEN  COUNTER := COUNTER + 1
11                         'FI
12               'OD
13   'END
```

The following are some comments on the above:-

1 Re line 1 : An array < ARRO1 > of 10 elements has been declared
2 Re lines 2 to 6 : These lines have been generated in order to read
values into the elements of the array < ARRO1 >
3 Re line 8 : The design gives insufficient detail to analyse <END>
4 Re line 9 : The design gives insufficient detail to analyse <UNDEFINED>
5 Re line 9 : The variable X has been used but it has not been initialised
6 Re line 10 : The value assigned to the variable < COUNTER > has never been used
7 The design does not contain any output statements Before the coded version could be run one or more PRINT statements need to be inserted

Diagram 21

Results From Analysing a Program Design Which Searches an Array to the eleventh. An array should be used to store the first ten values and an integer variable for the eleventh.

Let us consider the program design in some detail. (DS1) contains the first mention of an array. DACE can only handle a program design which uses a single array and consequently whenever the user refers to an array or an array element in a program design it is assumed that reference is being made to the same array. DACE could be extended to deal with program designs containing more than one array, however this would create problems of ambiguity unless the user specified clearly the array being referred to. To deal with such problems would require further considerable research effort.

Since an array has not been mentioned prior to (DS1) the following assertions are made in its representation:

```
[#ARRAY   NILL   (LB1)   (UB1)   A1]            (5.67)
[#LWB   1   LB1]                                (5.68)
[#UPB   N   UB1]                                (5.69)
```

where assertion (5.67) shows that the array has been given the default name NILL. In the absence of any other information it is assumed that the size of the array will be determined at the time of execution (see Example 6) and hence the default values of the lower and upper bounds are set to 1 and N respectively. However, once DACE considers the two arguments TEN NUMBERS and ARRAY together we can see from line 1 of the coded version that assertion (5.69) has been changed and the value of the upper bound has been revised to 10. The first of the comments illustrates how DACE will always refer the user to this array declaration and that the code generator has named the array ARR01 (c.f. IDR01 for the integer variable). The results of analysing line (DS1) are shown in lines 2 to 6 inclusive of the coded version and the second comment has been produced so that the user can identify easily the code necessary for reading values into an array. Two other array operations known to DACE and for which the ALGOL 68C code can be given are:

a) finding the sum of the values held in the elements of the array; and

b) the printing of these values.

(DS2) and (DS8) are statements which have used A and THE as the indefinite and definite article respectively. Lines 7 and 10 of the coded version show how these articles have been ignored and COUNTER has been analysed as a variable name.

(DS3) represents a statement which DACE can only partially analyse. The third comment informs the user of this fact and lines 8 and 9 of the code show how those items which cannot be translated into the target language are printed in angled brackets. The word END has the same dictionary definition as COUNTER (and RESULT which has been met in previous examples) and so DACE treats them both in a similar fashion. At first DACE assumes that END has been used as a variable name. However from its position in the line (which is in direct contrast to the position of COUNTER in (DS2)), it is recognised that the user wishes to access some value. Consequently it interrogates previous lines for evidence that a variable called END has been assigned a value. Because no evidence is found the assumption that END is a variable is revised and it is left as a word which is too general to be analysed. If END had been defined as a variable name then a comment similar to that of comment 5 would have been produced.

Statement (DS6) has been only partially analysed because of the failure to analyse NEXT ELEMENT. This result is brought to the attention of the user in line 9 of the coded version and comment 4. The term UNDEFINED in line 9 is a general term which DACE inserts into the program code when it cannot be established definitely that an item has been given a value, such as the array index in this case. The fact that it has only partially analysed this phrase can be attributed to two reasons. Firstly, whenever an array is used in connection with a loop DACE assumes the loop is used to access consecutive array elements. Consequently during the scope of the loop, statements of the following form are scanned for:

$$\langle\text{variable name}\rangle := \langle\text{variable name}\rangle + \langle\text{constant}\rangle \qquad (5.70)$$

where ⟨constant⟩ has an integer value of 1 and the ⟨variable name⟩ is used as the array index. However in order to use a variable name for this purpose the assignment (5.70) must not appear within a conditional statement since it must be executed on each iteration of the loop. This is necessary so that a variable such as COUNTER in line (DS8) is not used. This emphasises again the importance of considering statements within the context in which they are found.

After analysing the loopbody, DACE has failed to find a statement similar to (5.70) and so it reconsiders the boolean expression at the start of the loop. If this expression had given a more definite indication that the loop was being used to access successive elements of the array, then an assignment statement similar to that of (5.70) would have been generated so that the array index could have been inserted. The fact that the boolean expression in line (DS3) did not give any indication that the loop was being used to access successive elements of the array is the second reason why this line has been analysed only partially. The opposite case to that found here is dealt with in Example 6. The fifth comment also relates to (DS6) and in particular to the fact that X is treated, and indeed can only be treated, as a variable name. This may be contrasted with the analysis of a word such as END, which may or may not be a variable name, the actual decision depending upon the context in which it is found.

The analysis of line (DS8) indicates that the variable COUNTER is redundant in this program design. This is brought to the attention of the user in comment 6. The technique used to achieve this has been described previously in Example 3. It is assumed that whenever a variable is defined it will never be used throughout the remainder of the program design. An assumption that DACE then tries to disprove. For example when line (DS2) was analysed, DACE noted that COUNTER was simply assigned a value and thus only appeared on the left hand side of an assignment statement. However as soon as the expression:

$$\text{COUNTER} + 1 \qquad (5.71)$$

was formed as a result of analysing (DS8) it was noted that COUNTER was now to be found on the right hand side of an assignment statement. Despite this DACE is able to determine from the single reference to COUNTER in line (DS8) that COUNTER has no significance in this program design.

This method of noting the definition and possible redefinition of variables is complicated by a loop construct because it alters the top-down control flow of the program. Reconsider for example the following loop which DACE produced in the previous example:

```
'WHILE   J < 15                                  (5.72)
'DO      PRINT (RESULT + LASTRESULT);            (5.73)
         TEMP := RESULT + LASTRESULT;            (5.74)
         LASTRESULT := RESULT;                   (5.75)
         RESULT := TEMP;                         (5.76)
         J :=  J +  1                            (5.77)
'OD                                              (5.78)
```

If DACE had disregarded the control structure of the loop then it would have deduced incorrectly that the values assigned to LASTRESULT, RESULT and J had never been used. Consequently at the end of the loop DACE reconsiders the statements higher in the loop in order to detect that the values assigned to LASTRESULT and RESULT in lines (5.75) and (5.76) have been used elsewhere, in this case in line (5.73)and furthermore that the variable J has been used in the boolean expression of line (5.72).

The final comment shows that DACE always expects a program design to include a statement which will be analysed as a PRINT statement. Since DACE recognises that a program design is intended to be implemented as a program, the only way that the results can be described is by printing them. If DACE could analyse a design which a user intended to implement as a procedure say, then no print statements would be necessary since the results are returned to the main body of the program. On the other hand DACE does not require that a program design has an input statement. Example 4 which calculates the Fibonacci series illustrates that such a statement is not a prerequisite of a meaningful program design.

Let us conclude this section by making two points about the method used to deal with arrays:

a) DACE can only handle program designs which use a single, one dimensional integer array. According to the grammar of a program design there is no way that a user can specify the bounds of the array. Consequently it is assumed the lower bound is always 1 and the upper bound is given a default value of N (see Example 6) which may be revised in the light of subsequent analysis. If a program design does not contain a statement of this sort but uses a loop construct to read values into the array, then DACE does not derive the array bounds from information contained in the loop. Instead DACE assumes that in this situation the array bounds will be determined by the value of the variable N at run-time. It is felt that the user will derive the greatest benefit from this approach, since the results obtained from analysing the loop can now be used (see Example 6) to show the user those factors which have influenced the range of elements accessed by the loop ;

b) generally speaking the policy which DACE adopts towards analysing program designs which contain an array is that all elements of the array should be accessed. This reflects the common approach taken to the processing of arrays. Hence the analysis is concerned with operations on an entire array and expects a loop, for example, to access all elements in an array. The consequences of this are illustrated in Example 6.

5.6 Example 6

Diagram 22 shows an example of a program design which calculates the first N terms of the Fibonacci series where the value of N is determined at the time of program execution. The difference between this program design and that of Example 4, which also calculated the Fibonacci series, is that the current example stores the terms in an array before printing them.

(DS1) illustrates that in order for DACE to declare an array it does not necessarily have to meet the word ARRAY in the program design. In this example the word ELEMENTS has led to three array assertions being made which are similar to (5.67), (5.68) and (5.69) in the previous section. However unlike the previous example, line 2 of the coded version shows that the upper bound has not been revised and therefore the default value of N has remained throughout. Line 1 of the coded version illustrates how DACE always expects the value of N to be read in and alternative methods of assignment are not considered.

The design is as follows :-

```
(DS1)    SET THE FIRST TWO ELEMENTS TO 1
(DS2)    WHILE NOT END OF ARRAY
(DS3)    DO
(DS4)          SET THE CURRENT ELEMENT TO THE SUM OF
                   THE PREVIOUS TWO ELEMENTS
(DS5)    OD
(DS6)    OUTPUT THE ARRAY
(DS7)    ***
```

A coded form of the design is:-

```
0    'BEGIN    'INT      IDRO2, IDRO1, N;
1              READ(N) ;
2              [1:N] 'INT  ARRO1;
3              ARRO1[1] := 1;
4              ARRO1[2] := 1;
5              IDRO2 := 3;
6              'WHILE  IDRO2 <= N
7              'DO    ARRO1[IDRO2] := ARRO1[IDRO2 - 1]
                                   + ARRO1[IDRO2 - 2];
8                       IDRO2 := IDRO2 + 1
9              'OD  ;
10             IDRO1 := 1;
11             'WHILE  IDRO1 <= N
12             'DO    PRINT (ARRO1[IDRO1]) ;
13                      IDRO1 := IDRO1 + 1
14             'OD
15   'END
```

<u>Diagram 22</u> (continued on following page)

<u>Results From Analysing a Program Design Which Generates The Fibonacci Series by Using an Array</u>

```
The following are some comments on the above:-

1  Re lines 1 and 2 : An array < ARR01 > of N elements has
   been declared
2  Re line 7 : IDR02 has been used to index the array
   Consequently the first iteration of the loop
   references the  element  ARR01[3]
   If this was not intended or is an incorrect analysis
   change either the initial value of IDR02
   or the index
3  Re line 7 : IDR02 - 1 has been used to index the array
   Consequently the final iteration of the loop
   references the element ARR01[N - 1]
   If this was not intended or is an incorrect analysis
   change either the index or the boolean expression
   following WHILE
4  Re line 7 : IDR02 - 1 has been used to index the array
   Consequently the first iteration of the loop
   references the element ARR01[2]
   If this was not intended or is an incorrect analysis
   change either the initial value of IDR02
   or the index
5  Re line 7 : IDR02 - 2 has been used to index the array
   Consequently the final iteration of the loop
   references the element ARR01[N - 2]
   If this was not intended or is an incorrect analysis
   change either the index or the boolean expression
   following WHILE
6  Re lines 10 to 14 : These lines have been generated
   in order to  print
   the values held in the elements of the array < ARR01 >
```

Diagram 22 (continued from previous page)

Results From Analysing a Program Design Which Generates The Fibonacci Series by Using an Array

(DS2) contains the phrase NOT END OF ARRAY which is recognised as meaning the loop is being used to access consecutive elements of the array. For reasons outlined at the end of the previous section it then expects the first iteration to access the element specified by the lower bound of the array (i.e. ARR01 [1]) and the last iteration to access the element specified by the upper bound of the array (i.e. ARR01 [N]). The previous example illustrates how DACE scans the loopbody for an assignment of the following form:

<variable name> := <variable name> + <constant> (5.79)

where <variable name> is used as an index to the array and <constant> is assigned an integer value of 1 so that consecutive elements of the array may be accessed. It was also stated that if such an assignment was not found, but that there was sufficient evidence to indicate that the loop was being used to index consecutive elements of the array, then an assignment similar to (5.79) would be generated. Since the current line makes an implicit reference to the upper bound of the array this is considered sufficient evidence to produce the assignment. It was not undertaken in the previous example because a phrase such as NOT END would imply that the loop was to be terminated according to some other criterion. DACE analyses the current line as having the following meaning:

<variable name> <= N (5.80)

If a statement such as (5.79) is found in the loopbody then the variable name in (5.80) can be replaced by the actual name.

(DS4) is important because it shows how phrases such as CURRENT ELEMENT and PREVIOUS TWO ELEMENTS are analysed. Because of DACE's expectations about loops and arrays the index of the elements corresponding to these phrases is considered relative to:

ARR01 [<variable name>] (5.81)

where variable name is the same as that defined and found in (5.79). The meanings of CURRENT and NEXT ELEMENT are both represented as (5.81), since the latter is assumed to imply that successive iterations of the loop will consider successive elements of the array. In a similar manner the phrase PREVIOUS TWO ELEMENTS is assumed to mean those elements which were indexed on the previous two iterations of the loop and which therefore can be represented by the following format:

ARR01 [<variable name> - 1] (5.82)

ARR01 [<variable name> - 2] (5.83)

Whenever array elements of this type are met, DACE notes the value of <variable name> that is expected on the first and last iterations of the loop. For example when (5.81) is met it denotes that <variable name> should be initialised to 1 before entry to the loop and contain the value N on the last iteration of the loop. If this situation is found in the program design, then the results of the analysis meet DACE's expectations about arrays and loops. However the expectations of these values have had to be revised by DACE after consideration of (5.82). The variable name should now be initialised to 2 before entering the loop. If these expectations are met then the following situation will hold true:

	Array element accessed on the first iteration of the loop	Array element accessed on the last iteration of the loop
ARR01[<variable name>]	ARR01[2]	ARR01[N]
ARR01[<variable name> - 1]	ARR01[1]	ARR01[N-1]

which illustrates DACE's policy of trying to ensure that somewhere within the loop, the first iteration will access ARR01[1] and the last iteration will access ARR01[N]. From this discussion it is now apparent that once (5.83) has been considered DACE expects the variable name to be initialised to 3 rather than 2. Each of the elements (5.81), (5.82) and (5.83) have only affected the expectation of the initial value of the variable name. However if an element similar to:

ARR01[<variable name> + 1] (5.84)

is met then DACE would expect variable name to have a value equal to N-1 rather than N on the final iteration of the loop. Comments 2, 3, 4 and 5 of the results summarise those features of arrays which concern DACE and which have been described above. Comments 2 and 4 relate to the first iteration of the loop and point out how the index can be affected by either the initial value of the variable or the expression used as the array index. Conversely comments 3 and 5 are concerned with the elements accessed on the last iteration of the loop and point out how these can be altered by changing either the boolean expression at the start of the loop or the expression used as the index.

(DS5) marks the end of the loop and it is at this stage that the variable name will be replaced by the actual name of any variable which has been included in an assignment statement similar to (5.79). The results show this has not been found and hence DACE has created a variable, with the name IDRO2, for this purpose. Line 5 shows how it has been initialised correctly and line 8 shows that the assignment statement has been generated correctly.

(DS6) shows another array operation which DACE is capable of analysing, namely printing the values held in the elements of the array. The final comment has been produced in order to refer the user to that portion of the code which carries out this operation.

In addition to the conclusions drawn at the end of the previous section concerning DACE's policy towards arrays, the following may be added:

a) because the size of the array is undefined, the code generator has automatically included a READ statement prior to its declaration. At present DACE cannot identify if the design contains a statement for this purpose. For instance if the first line of the design had been:

```
        READ A VALUE INTO N                          (5.85)
```

then DACE would not have associated variable N with the size of the array even if this had been the implication of the statement. Generally speaking a statement such as (5.85) is considered too vague to associate with the definition of array size. DACE would require a more specific statement, such as:

```
        READ THE SIZE OF THE ARRAY INTO N            (5.86)
```

in order to associate the variable N with the upper bound of the array;

b) the comments which DACE makes about array elements used in a loop are based on the fact that there is no certain way of deciding whether a phrase such as NEXT ELEMENT means:

```
        ARR01 [ <variable name> ]                    (5.87) or
        ARR01 [ <variable name> + 1 ]                (5.88)
```

since the actual assignment will depend upon the context of the loopbody. Consequently the comments have been chosen deliberately to display this indecision and the final decision on correctness is left to the user. The important point here is that DACE brings to the attention of the user the effect on the array index of choosing certain operations. Since an array index that is outside the bounds of the array is a common error, it is hoped that this form of analysis will help to avoid such a situation.

5.7 Example 7

Diagram 23 is a further example of a program design which shows how DACE deals with arrays. This design has been produced in accordance with the following problem

The design is as follows :-

```
(DS1)     INPUT TEN NUMBERS INTO AN ARRAY **
(DS2)     SET RESULT TO THE VALUE OF THE LAST ELEMENT
             OF THE ARRAY AND INITIALISE FOUND
(DS3)     WHILE I IS LESS THAN 11 AND NOT FOUND
(DS4)     DO
(DS5)           INCREMENT I
(DS6)           IF NEXT ELEMENT IS LARGER THAN X
(DS7)           THEN
(DS8)                 SET FOUND TO TRUE AND RESULT TO
                         THE VALUE OF THE PREVIOUS ELEMENT
(DS9)           FI
(DS10)
(DS11)    OD
(DS12)    IF RESULT IS EQUAL TO X
(DS13)    THEN
(DS14)          PRINT # ARRAY CONTAINS X #
(DS15)    ELSE
(DS16)          PRINT # ARRAY DOES NOT CONTAIN X #
(DS17)    FI
(DS18)
(DS19)    ***
```

A coded form of the design is:-

```
0    'BEGIN    'INT     IDRO1, X, I, RESULT;
1              'BOOL    FOUND;
2              [1:10] 'INT  ARRO1;
3              IDRO1 := 1;
4              'WHILE  IDRO1 <= 10
5              'DO     READ (ARRO1[IDRO1]) ;
6                       IDRO1 := IDRO1 + 1
7              'OD   ;
8              RESULT := ARRO1[10];
9              FOUND := < UNDEFINED >;
10             'WHILE  I < 11 'AND  'NOT  FOUND
11             'DO    I := I + 1;
12                      'IF     ARRO1[I] > X
13                      'THEN   FOUND := 'TRUE ;
14                              RESULT := ARRO1[I - 1]
15                      'FI
16             'OD  ;
```

Diagram 23 (continued on following page)

Results From Analysing a Program Design Which Searches a Sorted Array

```
17             'IF     RESULT = X
18             'THEN   PRINT ("ARRAY CONTAINS X")
19             'ELSE   PRINT ("ARRAY DOES NOT CONTAIN X")
20             'FI
21   'END
```

The following are some comments on the above:-

1 Re line 2 : An array < ARRO1 > of 10 elements has been declared

2 Re lines 3 to 7 : These lines have been generated in order to read values into the elements of the array < ARRO1 >

3 Re line 9 : The design gives insufficient detail to analyse <UNDEFINED>

4 Re lines 10 and 11 : The variable I has been used but it has not been initialised

5 Re line 11 : The variable I has been used but it has not been initialised Variables used in this manner are usually initialise prior to entering the loop

6 Re line 12 : I has been used to index the array Consequently the final iteration of the loop references the element ARRO1[11] This will cause an execution error since the index is outside the bounds of the array In order to rectify change either the index or the boolean expression following WHILE

7 Re lines 12 and 17 : The variable X has been used but it has not been initialised

Diagram 23 (continued from previous page)

Results From Analysing a Program Design Which Searches a Sorted Array specification:

> read ten integer values, which have been sorted into ascending order, from a data file. Another value can then be read in and the program should determine whether or not this value is contained in the sorted list.

The program design shows how this has been solved by reading ten values into an array and by using a loop structure to search through the array elements for the given value. Since the values are in ascending order, the loop shows how the search can be terminated without necessarily considering all the elements.

Let us consider those lines within the design which illustrate points not yet discussed. (DS2) and line 9 in the coded version show that INITIALISE FOUND has been partially analysed. In this respect DACE has inserted a general term - UNDEFINED - to show that the assignment is incomplete.

(DS3) and (DS2) show alternative uses of the word AND. In (DS2) it has been used as a conjunction and consequently has been discarded before semantic analysis was initiated. However lines (DS3) and 10 illustrate AND is always implemented as a boolean operator when it appears in the boolean expression of a loop or conditional.

(DS5) corresponds to line 11 in the coded version and for reasons outlined in previous sections the variable I has been used to index the array elements in lines 12 and 14. The fourth and fifth comments also relate to this variable. The first of these points out that in order to avoid an execution error, I should have been assigned a value prior to the current line i.e. line 10. The fifth comment supplements the fourth and has been made because DACE has recognised that this special form of an assignment statement has been included in the loop-body as a possible array indexing operation.

(DS6) contains the phrase NEXT ELEMENT and because the appropriate variable name was discovered in the previous line, DACE has analysed this to mean:

```
            ARR01  [I]                          (5.89)
```

instead of the more general form:

```
            ARR01 [<variable name>]             (5.90)
```

which was necessary in the previous example. The sixth comment brings to the attention of the user the fact that line 12 of the coded version will cause an execution error on the final iteration of the loop. In making this comment DACE has shown that the techniques discussed in the previous section are sufficiently general to cater for different loop terminating conditions and array indexing operations. More specifically, DACE is capable of recognising that the array elements accessed on the final iterations of the following loops are the same:

```
        'WHILE  I  <  11                          (5.91)
        'DO     I := I + 1;                       (5.92)
                statements which reference
                ARR01  [I]                        (5.93)
        'OD                                       (5.94) and
        'WHILE I   <=  11                         (5.95)
        'DO    statements which reference
               ARR01  [I]                         (5.96)
```

```
            I := I + 1                              (5.97)
        'OD                                         (5.98)
```

The absence of a comment about the array index in line 14 also shows that DACE has correctly analysed that in the final iteration of the loop ARR01 [10] will be accessed. To deduce this it has to take into consideration the actual form of the boolean operator used at the start of the loop, the position of the assignment statement (i.e. the one in line 11) within the loopbody and the arithmetic expression used to index the array.

In addition, although this is not shown by the results, DACE has recognised that in order to reference the first element of the array on the first iteration of the loop, I should be initialised to 2. Because the user has specified the variable I both in the boolean expression in (DS3) and the statement in (DS5) it is considered that the actual initialisation of I should be given by the user. Similarly, although there is no reason why DACE could not be extended to complete the assignment in line 9 it is considered that since the user has partially specified the assignment statement, he should be encouraged to complete the specification.

5.8 Example 8

Some of the program designs previously considered in this chapter have contained statements which DACE is not capable of analysing or for which an incorrect analysis had been made. Let us now consider an example of a program design which contains statements that cannot be analysed because they are beyond the scope of FAPD. This example has been included to show how DACE deals with such a situation. Diagram 24 shows a program design based on the following problem specification :

> calculate the income tax to be paid by each employee of a company. The information available for each employee comprises that employee's earnings, the number of dependents, expenses and type of employee (i.e. whether a man, woman or teenage person). No employee should pay negative tax and a compulsory works charity contribution depends on employee type - (a man pays £5, a woman £2 and a teenager £1). The rate of tax is 35% and a £150 tax free allowance is made for each dependent. Expenses and charity contribution are tax deductible.

This specification together with the program design shown in diagram 24 are based on those used by Wilson and Addyman [Wilson and Addyman 1978] to illustrate programming by stepwise refinement. Let us consider why lines (DS4) to (DS8) inclusive have been only partially analysed.

As was stated in chapter 3 successful analysis of a statement requires that all the information necessary for that analysis must be derived from the following two sources :

a) from a class instance which is used to derive the meaning of a common design statement; and/or

b) from the results of analysing previous statements in the design.

In this respect the meaning of statements such as (DS4) to (DS8) could be derived from the first of these sources. However class instances for analysing terms such as

The design is as follows :-

```
(DS1)    READ N AND INITIALISE I TO 1
(DS2)    WHILE I IS LESS THAN OR EQUAL TO N
(DS3)    DO
(DS4)          READ DATA FOR EMPLOYEE **
(DS5)          CALCULATE CHARITYLEVY **
(DS6)          CALCULATE TOTALEXPENSES **
(DS7)          CALCULATE ALLOWANCE **
(DS8)          CALCULATE TAX **
(DS9)          PRINT TAX OWING **
(DS10)         INCREMENT I
(DS11)   OD
(DS12)
(DS13)   ***
```

A coded form of the design is:-

```
0    'BEGIN    'INT     TAX, ALLOWANCE, TOTALEXPENSES,
                        CHARITYLEVY, I, N;
1               READ (N) ;
2               I := 1;
3               'WHILE  I <= N
4               'DO    READ (< DATA FOR EMPLOYEE >) ;
5                       CHARITYLEVY := < UNDEFINED >;
6                       TOTALEXPENSES := < UNDEFINED >;
7                       ALLOWANCE := < UNDEFINED >;
8                       TAX := < UNDEFINED >;
9                       PRINT (TAX) ;
10                      I := I + 1
11              'OD
12   'END
```

The following are some comments on the above:-

1 Re line 4 : The design gives insufficient detail to analyse <DATA FOR EMPLOYEE>
2 Re line 5 : The design gives insufficient detail to analyse <UNDEFINED>
3 Re line 5 : The value assigned to the variable < CHARITYLEVY > has never been used
4 Re line 6 : The design gives insufficient detail to analyse <UNDEFINED>
5 Re line 6 : The value assigned to the variable < TOTALEXPENSES > has never been used
6 Re line 7 : The design gives insufficient detail to analyse <UNDEFINED>
7 Re line 7 : The value assigned to the variable < ALLOWANCE > has never been used
8 Re line 8 : The design gives insufficient detail to analyse <UNDEFINED>

Diagram 24

Results From Analysing a Program Design Which Calculates Income Tax Payable

CHARITYLEVY, TOTALEXPENSES etc. would be specific to this problem and if implemented for general cases would lead to the problem of combinatorial explosion. Hence the basic approach pursued in this study is to develop class instances which are sufficiently general to be applicable to more than one problem. An alternative approach would be to develop a collection of general class instances which could interrogate the problem specification for information to aid this analysis. However one of the main limitations of FAPD is that it makes no use of this specification to help its analysis. Consequently analysing statements such as those considered in statements (DS4) to (DS8) is beyond the scope of FAPD.

Let us consider DACE's analysis of this program design. (DS4) has been implemented as a READ statement. However DACE does not recognise the phrase DATA FOR EMPLOYEE. It has no comprehension of the concept of an EMPLOYEE, and consequently has tried to analyse it as a variable name. The phrase DATA FOR EMPLOYEE is unrecognised and hence is left in its original form.

(DS5) to (DS8) have all been implemented as assignment statements and CHARITYLEVY, TOTALEXPENSES, ALLOWANCE and TAX are considered to be variable names. From the results we can see that DACE incorrectly expects each of these calculations to produce a single result. Because the initial assumption is that assignment statements are required and then because it is found subsequently that these statements are not required, comments 3, 5 and 7 are generated.

(DS9) has been analysed and a PRINT statement produced. Syntax analysis has determined that OWING can be ignored for the same reason that BOTH was ignored in line (DS2) of Example 4 (see section 5.4). Consequently DACE has realised that the value assigned to the variable TAX in the previous line has in fact been used and as a result no comment is made about line 8 in the coded version.

This example illustrates how DACE has attempted to analyse a program design even though the full meaning of some statements cannot be derived. It is important that examples such as this are tested for two reasons. Firstly, although a program design may not be analysed completely the results from analysing portions of it, such as (DS1), (DS2), (DS9) and (DS10), will provide the user with some benefit. Although the design needs further refinement before it can be analysed completely, comments such as this are still considered to be useful at this stage. Secondly, defining the scope of FAPD and the system is a very difficult problem which can only be clarified by considering results such as those given in diagram 24.

An important point illustrated by this example is that DACE provides assistance in the process of stepwise refinement by indicating those lines which need to be specified in more detail before the design stage is complete. In this example lines 4 to 8 of the coded version show those statements which the user needs to refine further in order to complete the design stage.

5.9 Examples 9, 10 and 11

The results considered so far in this chapter have been produced by subjecting program designs to the four processes of analysis that comprise DACE. However to conclude this chapter, consideration now turns to three examples which do not conform to the grammar of a program design. In this respect they have been rejected after the syntax analysis stage and consequently have not been passed to the pre-semantic routines or subsequent stages. These three examples have been specifically chosen because they represent three different kinds of syntax error.

Diagram 25 is typical of the results produced by DACE whenever it considers a program design is syntactically incorrect. This shows that the user's design has been pretty printed and is followed by a single message (the text of which is always the same) indicating that a syntax error has been found. The design has been stored in a file named CODE.RES so that the user can obtain a hard copy and establish why analysis of the design has failed. Because the syntax error message does not give any indication of why a design has been rejected, any users of DACE would require some details of the possible causes of syntax errors.

Let us now consider why each of these designs has been rejected. Syntax analysis of Example 9 has failed because line (DS2) contains an unrecognised symbol, namely ← . At this point DACE has invoked its back-tracking mechanism in an attempt to find an alternative parsing. Since this has also failed, the design has been rejected as syntactically incorrect. Lines (DS3), (DS6) and (DS10) contain other symbols which will also cause syntax errors. The rejected symbols are <, > and +

```
The design is as follows :-

(DS1)    GET FIRST VALUE OF DATA INTO MAXSOFAR **
(DS2)    COUNTER <- 1
(DS3)    WHILE COUNTER < 1000
(DS4)    DO
(DS5)          GET NEXT VALUE OF DATA INTO N
(DS6)          IF N > MAXSOFAR
(DS7)          THEN
(DS8)              MAXSOFAR <- N
(DS9)          FI
(DS10)         COUNTER <- COUNTER + 1
(DS11)   OD
(DS12)   OUTPUT MAXSOFAR WITH SUITABLE TEXT
(DS13)   ***

A syntax error has been found in this design
```

Diagram 25

Results From Analysing a Program Design Which Contains an Unrecognised Symbol

```
The design is as follows :-

(DS1)    SET MAX AND MIN TO FIRST VALUE **
(DS2)    SET NOCONSIDERED TO 2
(DS3)    WHILE NOCONSIDERED IS LESS THAN 1000
(DS4)    DO
(DS5)          GET NEXT VALUE
(DS6)          IF THE VALUE IS LARGER THAN MAX **
(DS7)          SET MAX TO THIS VALUE
(DS8)          THEN
(DS9)                INCREMENT NOCONSIDERED
(DS10)         ELSE
(DS11)
(DS12)               IF THE VALUE IS LESS THAN MIN **
(DS13)              SET MIN TO THIS VALUE
(DS14)               THEN
(DS15)                     INCREMENT NOCONSIDERED
(DS16)               FI
(DS17)
(DS18)         FI
(DS19)
(DS20)   OD
(DS21)   OUTPUT MAX AND MIN
(DS22)   ***

A syntax error has been found in this design
```

Diagram 26

Results From Analysing a Program Design Which Contains an Unrecognised Form of a Construct respectively. In order for lines (DS2) and (DS3) to be analysed, they should have been written in the following forms:

SET COUNTER TO 1 (5.99) and

WHILE COUNTER IS LESS THAN 1000 (5.100)

Example 10 has failed because of line (DS7) and illustrates a second category of syntax error. This shows that the grammar of a program design does not allow a statement which will be implemented as an assignment statement to appear at the start of a conditional. Consequently this represents those errors where the target language constructs for repetition and choice have not been used as required.

Finally, Example 11 represents a third form of syntax error. Lines (DS2) and (DS4) are based on examples found in Findlay and Watt [Findlay and Watt 1981] and it is the phrase NEXT SUMMAND contained in (DS4) which has caused the error. This has been analysed in the same way that phrases such as THIS VALUE and FIRST ELEMENT are analysed. However whereas DACE contains dictionary definitions of words such as VALUE and ELEMENT, the current implementation of the system does not recognise the word SUMMAND and therefore it has been analysed as a variable name. A phrase comprising an adjective followed by a variable name does not fit the grammar of a program design and consequently the design has been rejected. It is interesting to note that if THE and NEXT were omitted and the line had read:

```
The design is as follows :-

(DS1)    INITIALISE SUM TO 0
(DS2)    WHILE NOT END OF DATA
(DS3)    DO
(DS4)            READ THE NEXT SUMMAND AND ADD IT TO SUM
(DS5)    OD
(DS6)    OUTPUT THE VALUE OF SUM
(DS7)    ***


A syntax error has been found in this design
```

Diagram 27

Results From Analysing a Program Design Which Contains an Unrecognised Phrase

READ SUMMAND AND ADD IT TO SUM (5.101)

analysis of SUMMAND as a variable name would have resulted in the syntax analysis being successful and the program design being analysed.

6. RESULTS FROM USING DACE

6.1 Objectives and Methodology

In the previous chapter, the scope of DACE was described by considering its application to eleven program designs. In this chapter we describe an evaluation of DACE using a group of people with various levels of programming experience. By using people of dissimilar experience, conclusions might be drawn concerning the type of user who derives the greatest benefit from using the system. All the examples and results discussed in this chapter were derived from the evaluation exercise.

For the purposes of this report, all those who participated in the tests are referred to as "users". Eighteen people took part in the experiment and diagram 28 shows how they can be classified. All the students (ie categories 1 to 6) came from the University of Aston and categories 1, 2 and 6 were learning to program. All the undergraduate students were studying for either a single honours degree in computer science or a combined honours degree in computer science and another subject. None of the M.Sc. IT students had degrees in computer science. The primary programming language used by all the students was PASCAL. The others (ie category 7) were graduates from industry, who did not have degrees in computer science, were not professional programmers nor wrote programs on a regular basis. Each user undertook a maximum of five different programming exercises, the solutions to which were submitted to DACE. The total

Category	Description	Number of Users	Number of Program Designs Submitted to DACE	Number of Program Designs Resubmitted to DACE
		(1)	(2)	(3)
1	Combined honours under graduate students - Year 1	2	9	3
2	Single honours undergraduate students - Year 1	2	9	5
3	Single honours undergraduate students - Year 2	4	19	11
4	Single honours undergraduate students - Year 3	3	15	7
5	Combined honours undergraduate students - Year 3	1	5	3
6	M Sc Information Technology (IT) students	4	20	12
7	Others	2	10	3
	TOTAL	18	87	44

Diagram 28

Summary of Results from using DACE

number of different solutions which were submitted by users from each category is shown in column 2 of diagram 28. Column 3 shows the number of solutions which were resubmitted because the original version contained a syntax error.

Before starting the evaluation exercise each user was given handouts containing instructions. As far as possible the names of the handouts are included in this discussion so that the reader can refer to the appropriate material in Appendix G. The experiment took the following form:

a) because of a lack of standard terminology all students were given an "Introduction" handout which explained the phrase program design;

b) users who were not computer science students were given a handout entitled "Notes on Program Design". This was considered necessary since these users may not have been aware of the importance of this part of program development. Although the handout referred to constructs for denoting selection and repetition of actions, the fact that these were ALGOL 68C constructs was not mentioned. Knowledge of the target language is not a prerequisite for using DACE and in an effort not to overburden students with unnecessary detail, any reference to ALGOL 68C was avoided;

c) all users undertook a pre-test and a post-test exercise to solve somewhat similar problems. It was hoped that a comparison between the two solutions would ascertain the effect (if any) that DACE had on a student's performance. The two problems (see Exercises 1a and 1b) were carefully chosen in order to allow the more experienced users to include advanced programming concepts (eg arrays) in the solution and the less experienced users to formulate a solution without using, or indeed knowing, such concepts. The order in which the two problems were tackled was varied so that any difference in problem complexity would be nullified;

d) after the pre-test exercise had been completed users were asked to read the "Introductory Notes for the System User". These notes outlined the basic operation of DACE, the kinds of program designs which could be analysed and some possible causes of syntax errors. A list of system recognised words was also included. For similar reasons to those outlined in (b) above, no reference was made to ALGOL 68C. To sustain their interest users were encouraged to use the system as quickly as possible instead of spending an inordinate amount of time trying to understand every detail within the handout;

e) users were then given a series of exercises which required program designs to be developed for particular problems. The users were requested to write the solution out prior to inputting it into the system. This meant the time spent logged-on to the DEC was kept to a minimum. This was important since the longer a user was logged-on, the more marked was the deterioration in response time. Once the design had been formulated, the system was called up and the user allowed to submit his solution to DACE. The DEC's PHOTO facility recorded all interactions between DACE and the user. DACE then displayed the results of its analysis and the user was given the next exercise in the series;

f) if DACE reported that the program design contained a syntax error, the user was asked to read section 4 of the Introductory Notes which listed some possible causes. The user could then submit a revised solution to the system. If the revised version also contained a syntax error, the user was informed verbally of the cause and was then shown a "Model Solution". These solutions were intended to make the user more aware of the kinds of program design which DACE can accept. It was emphasised that they were not the only solution which the system would accept and numerous variations were possible. The user would then be given the next programming exercise;

g) a set of systematic instructions were given to users whenever they asked for help because they had run into difficulties. The first set ("Instructions 1") was used if a program design was being entered. These instructions could be used when either:
   i) the user had typed a control character which had generated a LISP interrupt; or
   ii) the user wished to correct previous lines in the input.

In both cases it was necessary to reinput the program design and the instructions gave details of how to do this. The second set ("Instructions 2") was used after the program design had been entered but module 1 (see diagram 12) was still in operation. These instructions also requested the program design to be resubmitted. The final set ("Instructions 3") was used whenever difficulties arose with either modules 2 and 3 (see diagram 13) or module 4 (see diagram 14). Typical difficulties here would be a user typing START instead of (START). These instructions gave details for re-entering the current system module;

h) once the exercises had been completed, the users were asked to complete a "Questionnaire" so that their evaluation of the system could be assessed. The text will refer to the results of this questionnaire (see Appendix G);

i) finally, the users were asked to complete the post-test exercise.

6.2 Problem Solutions

The series of programming exercises undertaken by the users was designed to test their ability to deal with some basic programming concepts. The programming exercises involved designing programs for the following problems:

Exercise 1 : Input an integer value which represents a measurement in yards. Output the corresponding number of inches.

Exercise 2 : Input ten integer values. Print each of these values and their total.

Exercise 3 : Input two integer values and print a message stating whether or not the two values are equal.

Exercise 4 : Input ten numbers. Output how many of these numbers have a value greater than 100.

Exercise 5 : A data file contains a set of positive integer values. The end of the set is signified by a 0. Find the total of these values.

When analysing the users' solutions to these exercises DACE detected many errors although some others went undetected. Out of 131 program designs, 77 were rejected by DACE because they contained errors. Some of the factors which caused DACE to reject program designs were :

a) the use of statements which did not conform to the grammar of a program design caused most errors. Typical of these statements are PUT IN LENGTH and RESULT IS INCHES. The former statement is rejected because PUT is not a recognised word whereas the latter is unacceptable because RESULT and IS are used in the wrong context. Errors of this type are difficult to diagnose but using a list of recognised words can help. Thirty-five program designs submitted to DACE contained incorrect statements of this type;

b) incorrect use of ** was also a common error. Out of

131 program designs, 27 used ** incorrectly, although 6 out of 18 users used it correctly at all times. The fact that 8 users thought the instructions on the use of ** were insufficient and 11 found it easier to use with practice suggests that greater tuition is required in this area prior to using the system. Occasionally a user failed to delimit ** with spaces which meant a statement such as READ THE VALUE INTO A** was analysed as READ (A**), where A** was assumed to be a variable. Fifteen program designs contained syntax errors because the character # was not delimited by spaces. Users obviously had similar problems with ** and # and on this basis any future versions of the Introductory Notes should place greater emphasis on the use of spaces;

c) the next section will discuss how the ← key was often used in an attempt to correct mistypings. If it is used during the input of a program design then a control character is read in and a syntax error generated. This occurred in seven of the program designs submitted to DACE. The next section also discusses how using a Lynwood terminal and typing O with the shift - lock on caused a LISP interrupt. On one occasion a user tried to overcome the problem by taking action which did not rectify the situation, but rather yielded an incomplete program design resulting in a syntax error;

d) ten program designs did not specify the correct form of a conditional. Seven of these were due to FI being omitted. Loops seemed to cause fewer problems and OD was <u>never</u> omitted. Two program designs contained loops in an incorrect format. Both of these related to the same user who had specified the following :

```
FOR COUNT EQUAL TO 1  TO  10   DO  ——  OD
```

Errors concerning loops and conditionals were not repeated by the same user on any subsequent exercise. This indicates that the users could adapt quickly to these constructs and the identification of a single error was sufficient to reinforce the system's requirements;

e) finally, the syntax errors in two designs were caused by spelling mistakes. In these cases THEN and LES had been typed instead of THAN and LESS.

Although DACE reported numerous syntax errors, this analysis shows that they fall into a small number of distinct categories. The Introductory Notes contained some causes of syntax errors which the users could try and relate to their program designs. The analysis above could be used to make these notes more succinct and to emphasise those errors which occurred most frequently. Other errors such as not delimiting ** and # with spaces could be overcome by extending DACE to include a prepocessor. This could check the characters within a word in order to identify if spaces had been missed. Thus #INCHES# and A** could be separated into # INCHES # and A ** before parsing was initiated.

The 54 program designs accepted by DACE were inspected by the author to determine if they contained errors of logic. Of these 54 designs, 20 contained errors which were not detected by DACE because of its lack of domain knowledge. These errors may be summarised as follows :

a) the wrong variable was output as the result. A typical example is printing the variable used to count the number of loop iterations instead of the variable used to store the sum of a number series;
b) a conditional statement was incomplete ie there was no ELSE part. This is similar to Miller's observation [Miller 1975] that novice programmers tend to underspecify algorithms and do not specify the actions to be undertaken when a set of conditions is not satisfied;
c) the branches of a conditional were inadvertently reversed such that the actions did not match the results of the condition;
d) loops did not terminate. This was because the variable used to count the number of loop iterations was not updated inside the loopbody or the variable updated within the loopbody was the wrong one;
e) a program design tried to read in more than the specified number of data values. This was because loop and input statements had not been combined correctly.

6.3 System-User Interface

6.3.1 Hardware Considerations

This section is concerned with the system's implementation on the DEC 20/60. Testing the system with the users showed that the following factors affect the usability of the current system :

a) the response time which varied according to the time of day. The best response was obtained before 10.00am and after 6.00pm. For a small program design (ie one of three lines) analysis took appriximately 2 minutes at 8.30am but anything up to 35 minutes at 1.00pm. Because the DEC is used by students at the University of Birmingham, the response times noted above would have been better during vacations. However, the availability of students meant that the experiment had to take place during term time and often when the DEC was used most heavily (ie 10.00am to 6.00pm). Response time is important because one of the primary requirements for an effective system-user interface is speed. If the time which the system takes to respond is excessive, a user could forget information or lose interest. Miller [Miller 1968] has shown that excessive delays in response time seriously affect the performance of computer tasks via terminals;

b) the students who took part in the tests all used the HARRIS 800 computer at the University of Aston. This allows them to use the terminal key marked ← to move the cursor back over previous characters so that mis-typings can be corrected. When using the DEC , the ← key appears to have the same effect because it can be used to backspace and then change characters on the screen. However, this effect is local and using the key actually generates a control character. If it is used during the input of a program design DACE reads this control character which can result in either a syntax error or distortions in the results. The actual result depends upon the context in which it is used. Although the Introductory Notes stated that the DELETE or RUBOUT key should be used, most people still used the ← key. Even when the importance of not using the ← key was stressed (verbally) prior to using DACE, some users still tended to use it "automatically" ;

c) all users accessed DACE via a Lynwood or Newbury 8000 terminal at the University of Aston. The Lynwood terminals caused two problems. Firstly, these terminals did not have a TTY CAPS key. This key allows all <u>letters</u> to be typed as if the shift lock was on. Any key which is not a letter is accepted as if the shift-lock was off. The absence of a TTY CAPS key meant that the shift-key was used continually. Some users found this difficult to adapt to and often switched it on or off at the wrong times. This obviously increased the time spent typing a program design. Secondly, depressing O with the shift-lock on caused a LISP interrupt which is normally used by a LISP programmer in order to break into a program execution. This is obviously confusing for anyone unfamiliar with LISP. Whenever this happened the user was informed of why it had occurred and was returned to the DEC's monitor level. This meant the system had to be re-entered and the program design resubmitted. The instructions for doing this were contained in the handout.

6.3.2 Software Considerations

The system software will obviously affect the usability of the system. This section discusses how users interacted with the programs that comprise DACE. One of the main factors affecting the system-user interface is that a user must type the instructions for calling the system modules (see diagrams 12, 13 and 14). The system was designed in this way so that the results from one module could be listed before the next module was called. This is particularly useful for anyone developing or extending the system but not desirable for normal use. The modules which comprise DACE also print out statements such as "The semantic analyser has now been entered" and "Semantic analysis is now complete". These and similar statements were an aid to system development because they identified how far the analysis of a program design had progressed. The questionnaire showed that 4 out of 18 users found such statements difficult to understand. Du Boulay and O'Shea [du Boulay and O'Shea 1980] emphasise that one of the difficulties facing the novice programmer is to understand what is going on in the computer. Consequently future versions of DACE might benefit from having these statements suppressed. The results from using DACE showed that further consequences of a user having to type instructions for loading and running the system modules are:

a) when users were asked to type (START) and (PRINT-CODE) many responded by omitting the parentheses (diagrams 13 and 14 illustrate when these instructions must be typed). (START) and (PRINT-CODE) each invoke a MICRO-PLANNER theorem and omitting the parentheses causes an error. Errors of this sort and the remedial action to be taken were described in a handout ;

b) diagrams 13 and 14 also show that users are asked if they wish to proceed to the next system module. This facility was used during the development of the system so that any LISP or MICRO-PLANNER functions could be edited before the next module was loaded. The easiest way to do this was to remain in the LISP or MICRO-PLANNER system so that the context editor could be used. Diagram 12 shows that when users first enter the LISP system they are asked if they wish to use DACE. Since a negative reply leaves them in the LISP system the only reason for doing this is again to aid system development. Because the system has retained many features which were included to facilitate its development this meant that users were required to input extra information ;

c) users were often confused about which control level was currently in operation. This caused the following errors :

   i) when DACE asked users if they wished to proceed to the next system module, some tried to list the file containing their program design. This can only be done at the DEC monitor level and not within DACE ;

ii) users tried to input a program design when they were at the DEC monitor level instead of typing LISP (see diagram 12) in order to access DACE;

iii) when the system asked users if they wanted to use DACE some tried to input a program design instead of replying yes or no.

These results are similar to those of Cannara [Cannara 1976] who showed that some students misunderstood the computational context and tried to run a program while it was being edited or vice versa.

Many of the errors noted in (a), (b) and (c) could be eliminated by writing a macro which could load and run the various modules as and when they are required. This would reduce the number of instructions which the user must type. Eisenstadt [Eisenstadt 1983] has implemented a software environment where users are automatically connected to the environment once they are logged on. This minimises their interaction with any other system or monitors. A similar implementation is also applicable to future versions of DACE. One restriction imposed by the current implementation of DACE is that any revisions to a program design can only be achieved by inputting the whole of the revised version. This is necessary because DACE does not load and analyse a program design from a file. If this was possible either a special system editor or the DEC editor could be used to revise an existing program design. The instructions for revising program designs are contained in a handout. One user typed in the following as the final statement of a loopbody :

```
      ADD  NUMBER  TO  TOTAL **
```

and then realised that inputting the loop delimiter OD would generate an error because ** should not be used prior to a reserved word. The error was corrected by typing in a dummy statement of the following form :

```
      ADD  NUMBER  TO  TOTAL **
      OUTPUT #   #
   OD
```

The lack of editing facilities was regarded as a disadvantage by 6 out of the 18 users.

A final point about the input phase concerns the use of the string *** to terminate the program design. If a user types a space after the string the design is terminated incorrectly and the user is given another invitation to type. Although the screen instructions emphasise the importance of doing this correctly mistakes are inevitable. On those occasions when such mistakes did occur the users were able to rectify them.

DACE's analysis of certain statements included in some solutions will now be discussed. Users made statements such as MULTIPLY YARDS BY 36 and ADD 1 TO COUNT to denote YARDS := YARDS * 36 and COUNT := COUNT + 1. However the current implementation of DACE then analysed these statements to mean IDR01 := YARDS * 36 and IDR01 := COUNT + 1 where IDR01 is a variable name generated by the system. This was obviously at variance with the user's intentions. At present an assignment statement such as YARDS := YARDS * 36 can be achieved by stating for example :

```
        MULTIPLY  YARDS  BY  36
        ASSIGN  THE  RESULT  TO YARDS
```

An assignment statement such as COUNT := COUNT + 1 could be effected by stating INCREMENT COUNT BY 1.

Another occurrence which presented difficulties for DACE was for the use of a statement such as OUTPUT NUMBER #INCHES# to mean PRINT (NUMBER, "INCHES"). DACE analysed this statement to mean PRINT (NUMBER). The first reason for this analysis is that the delimiter # and the string INCHES were not separated by spaces and consequently #INCHES# was considered to be a single word. In terms of the grammar of a program design it is used in the same context as SO and FAR in the statement OUTPUT TOTAL SO FAR. Because SO, FAR and #INCHES# are all unrecognised, the context in which they are found allows them to be ignored. The second reason why DACE has ignored #INCHES# is that the original statement should have included AND. The desired effect could have been achieved by the statement OUTPUT NUMBER AND # INCHES #. It is recommended that in future any users of DACE are made more aware of these requirements. A suitable note could be added to the Introductory Notes.

Another interesting occurrence was the use of abbreviations or alternative spellings. Examples of these are INPUT THE NO and INITIALIZE which should have been written as INPUT THE NUMBER and INITIALISE.

The SOPHIE system [Burton 1976] handles these problems by expanding abbreviations and correcting spelling mistakes before parsing is commenced. This is a facility which could be incorporated into a more sophisticated version of DACE. Burton discussed elliptic utterances which were also encountered in this exercise. Consider the following section of a program design :

```
INPUT THE NO
MULTIPLY BY 36
PUT IN LENGTH
```

The first statement is quite explicit whereas the second and third contain an implicit reference to THE NO. Burton solved this problem by using rules in a semantic grammar to identify which concept or class of concepts is possible from the context available in the elliptic utterance. In terms of the statement MULTIPLY BY 36. the two possibilities are :

```
MULTIPLY  <integer number>  BY 36
MULTIPLY  <variable name>   BY 36
```

To distinguish between these possibilities a search could be made through previous lines for an appropriate <integer number> or <variable name>. Although this is beyond the current capability of DACE, it could be achieved by using a modern natural language parser such as that developed by Burton.

Design statements such as those noted above do not contain sufficient detail for DACE to analyse them. Similarly statements such as CALCULATE INCHES and INPUT NUMBERS carry insufficient detail but the nature of the missing detail is quite different. Knowing the problem specification allows us to infer that the latter statement means INPUT TWO NUMBERS. However DACE has no knowledge of the problem specification and so NUMBERS is analysed as a list of undefined length. Analysing a phrase such as CALCULATE INCHES can only be achieved by using the problem specification and real world knowledge about the number of inches in a yard. This gives us an interesting insight into the user's perception of DACE. The Introductory Notes state that the system displays how a program design could be represented in code. If users appreciated this then they obviously thought DACE was more sophisticated than it actually was.

A final software consideration is that of syntax errors. The techniques used for syntax analysis mean that the cause of a syntax error is not known and users always receive the following message :

A syntax error has been found in this design

This does not identify the location or nature of the error and this was commented upon by 8 out of 18 users. Parsing halts as soon as the first syntax error is found which, despite the message above, does not necessarily mean that the design contains only <u>one</u> error. Burton states that an intelligent system should act intelligently when it fails. This is important for naive users, to whom the system should always appear "natural". In this respect any future work on DACE should consider alternative methods of syntax analysis that provide better error diagnostics.

The eighteen users who took part in the experiment submitted 131 program designs to DACE. Of these 77, (58·8%) contained syntax errors and diagram 29 shows how these were related to the five exercises which were undertaken. It is significant that 49·4% of the designs which contained syntax errors were solutions to the first two exercises. This and the very small number of errors in the solution to Exercise 5 indicate that by the end of the experiment users were becoming more aware of the reasons why syntax errors occur. This is also apparent when we consider the program designs which were revised and resubmitted because they contained syntax errors (see columns 3 and 4). For Exercise 2, 10 out of 11 solutions which were revised were also rejected by the syntax analyser. However by the time Exercise 4 was undertaken, 7 out of 11 failed for a second time, but 4 were revised correctly. Similarly all three of the revised solutions to Exercise 5 were passed as syntactically correct.

6.4 Results of the Pre and Post Test Exercises and the Questionnaire

This section discusses the solutions to the pre and post test exercises and the questionnaire. Although we are unable to draw any general conclusions from the analysis of the pre and post test exercises the following observations can be made :

a) 13 out of 18 users described their solutions to the pre test exercise in the expected sense without using PASCAL code. However 16 users wrote out their solutions to the post test exercise in the expected sense; and

Exercise Number	Number of Program Designs Submitted to DACE	Number of Program Designs Containing a Syntax Error	Number of Program Designs Resubmitted to DACE	Number of Resubmissions Containing a Syntax Error
	(1)	(2)	(3)	(4)
1	18	10	9	6
2	18	12	11	10
3	18	11	10	7
4	18	11	11	7
5	15	3	3	0
TOTAL	87	47	44	30

Diagram 29

Syntax Errors Analysed by Exercise Number

b) the post test solutions obtained from 2 of the users showed that their approach was more disciplined than it had been for the pre test exercise. However, the pre and post test solutions from another user were both lacking in discipline.

These results are encouraging since they seem to indicate that DACE had some influence on the student's performance even in the short exercise undertaken.

Some of the results from the questionnaire were discussed in the previous section and the remainder will now be considered. Although the primary programming language for most users was PASCAL and DACE's target language is ALGOL 68C, 16 out of 17 users had no difficulty identifying the relationship between their program design and the coded version. This is probably because the programming exercises were relatively simple and at this level there are only minor differences between ALGOL 68C and PASCAL. The one aspect of the coded version which users did query was the symbol /= which is the relational operator "not equal to" in ALGOL 68C. The corresponding operator in PASCAL is **< >**.

Users were also questioned about the utility of the comments produced by DACE. Six users reported that they had no difficulty in relating all the comments to the coded version of the program design and six others that they had no difficulty with over half the comments. Only two users reported that they found some comments <u>particularly</u> useful, whilst eight users felt that over half the comments produced were useful. The comment which was considered particularly useful concerned the use of a variable not previously initialised. One of the main purposes of DACE is to focus the user's attention on the program design rather than the coding. It was noticeable that some users saw deficiencies in a program design as soon as it was listed on the screen by DACE. These deficiencies, such as specifying the branches of a conditional incorrectly, would not necessarily have been commented upon by DACE. Hence the fact that fifteen users stated they would redesign at least one of their solutions was probably due to other factors besides the comments produced by DACE. The questionnaire also showed that seven users thought there was no need to undertake a program design for <u>any</u> of the problems set but five of these users said they would have redesigned some of their solutions because of the analysis and comments produced by DACE. This indicates that DACE must have had some influence on their thinking.

Of those questionned only two thought that they would spend more time designing programs in the future, the remainder stating that they would not modify their allocation of time. However, seven users thought that DACE had left them better equipped to formulate program designs, two of the users stating that DACE had demonstrated a way of specifying program designs which they found quite useful.

To be applicable to a large audience the program designs which the system accepts should be in a format as close as possible to that which programmers normally use.

This seems to have been achieved to a satisfactory level since all those who used the system reported that they did not have to significantly alter the way they normally wrote out program designs. The modifications which most people had to make concerned the way they normally specified loops and conditionals and restricting the words used to those recognised by the system. The former modification is obviously because the system was developed at a time when students at the University of Aston were taught ALGOL 68 whereas the primary teaching language is now PASCAL. Consequently this restriction is considered to be specific to the current implementation of DACE. The second modification could be overcome to some extent by extending the system dictionary to include additional keywords. One solution to this problem was incorporated into the SOPHIE system [Brown, Burton and de Kleer 1982] which automatically recorded any messages not understood so that the future development of the system was partially prescribed. A similar facility would obviously help any further development of DACE. Users were also questionned about the usefulness of the Introductory Notes. Although some improvements to these notes have already been suggested, it was noted that sixteen out of eighteen users found them sufficiently detailed to use the system.

7. CONCLUSIONS

7.1 Basis of FAPD

The details of FAPD have been given in previous chapters and now two fundamental ideas on which it is based are reconsidered. Firstly, we need to evaluate the benefits of developing a framework which when applied, is capable of analysing program designs. Secondly, we need to consider the implications of representing the results of analysis in the form of a coded version of the program design.

This thesis has viewed the programming process as comprising two related phases, namely the design of a program and the subsequent coding of that design. This research has concentrated on analysing examples produced during the former of these two phases. Because the constructs for repetition and choice are the only aspects of a target language which FAPD accepts, a programmer is forced to delay any decisions concerning the coding details of a design until a later stage. The importance of program design is now well established in the development process of good, structured programs. Although its importance is recognised, difficulties occur in determining when a program design is finalised. The system described in previous chapters can highlight those sections of a program design which need to be refined further. Consequently this emphasises that coding cannot be started until these sections have been specified in greater detail. Since the system is also capable of recognising deficient program designs, the development of working programs can be attained more readily whether by manual or automatic means.

If we accept that these reasons support development of a framework then consideration must be given to evaluating the way in which FAPD analyses a program design. The results from analysing a design are represented in the form of a series of assertions. These assertions are used to represent a coded version of the design from which a program together with any associated comments can be produced. There are several advantages in choosing this form of analysis. The principal advantage is that it provides a convenient format for representing the results of analysis. Since this format is based on a subset of the syntax of a programming language, it is well-defined and furthermore has obviated the need to develop another form of representation. It also means that the results can be printed in a form which is easy to comprehend.

A second advantage of this definition is that the coded version of the design can be analysed to see if it, and hence the design itself, performs as intended. This analysis could be achieved either by executing the program using example input and output pairs or by adapting some of the existing theories of program understanding. Thirdly, for novice programmers who have only just learned the coding details of a programming language, the coded version of a design should illustrate particular language features.

FAPD's method of analysis means that any errors which are detected are referred to the coded version of the design, and brought to the attention of the user for correction. Although FAPD may not detect all errors, even a partial detection is considered beneficial to the user. For such program designs the user may wish to resubmit an improved design taking into account the comments of DACE on the initial design. This process may, of course, be repeated. This is important because the questionnaire on program design showed that 45 out of 85 users would not amend their program design when they found errors of logic in the code. Hence the analysis undertaken by DACE emphasises that designing programs is an iterative process and solutions often need revising.

Both of these basic ideas were discussed in the opening chapter. At the same time a third idea was introduced which was concerned with the kinds of program design FAPD can accept. This idea is discussed in the following section.

7.2 Evaluation of FAPD

The Framework for Analysing Program Designs is comprised of four distinct phases :

a) pre-semantic analysis, the first operation of which is concerned with parsing a program design. Successful parsing means that any target language constructs which may have been used are in their correct format and the statements within that design are of a form that can be analysed. The syntax tree is then converted into a series of structures which the semantic analyser can recognise;

b) semantic analysis is concerned with implementing the structures produced in the pre-semantic analysis phase in terms of a particular programming language.

This implementation is undertaken by a collection of procedures where each procedure defines one recognisable structure. These procedures, which are called class instances, collectively define the body of programming knowledge incorporated within FAPD. Each class instance attempts to implement its own structure in the target language even though the results of doing so may be revised subsequently by other class instances when they are considered in the wider context;

c) comments are generated when the results produced by the semantic analysis routines carry certain implications for the user. Further comments may also be generated by the sets of class instances;

d) code generation, as the name implies, is used to convert the results of the previous two phases into a coded version of the program design. The results from semantic analysis are printed in the form of a program in the particular target language considered. Any comments are converted into the appropriate text and printed after the coded program. Any line numbers given in the comment statements refer to the line numbers of the coded program.

Since FAPD has no knowledge of the problem specification, the degree of analysis possible is obviously limited. Hence the full implication or inference of a statement may go undetected. This particular problem can be illustrated by referring to the following problem specification:

Design a program to find the average
of ten numbers (7.1)

and the following design statement which could be contained within a design which meets this specification:

INPUT THE NUMBERS INTO AN ARRAY (7.2)

The reader will have no difficulty in using the problem specification to infer that the array has ten elements. However, without the benefit of this knowledge, DACE has no means of determining the correct size of the array. Hence the array is analysed as comprising N elements, where the value of N is to be determined at the time of program execution. However, as the discussion of Example 8 in chapter 5 showed, even knowing the problem specification still does not guarantee a complete analysis of the program design. Example 8 was specifically chosen to show that a complete analysis can only be achieved by incorporating into FAPD real world knowledge of concepts such as employee, tax, charity etc..

Let us now conclude this discussion by stating that, at present, FAPD makes use of two sources of information to analyse any statement. These sources may be summarised as:

a) the results obtained from analysing previous lines in the same design; and

b) class instances which recognise and then represent, in terms of the target programming language, particular statements and phrases within the design.

The results discussed in chapters 5 and 6, together with those contained in Appendix D support the claim that these two sources provide sufficient information to undertake some useful analysis. However the depth of analysis could be improved by making use of a third source, namely the problem specification. Because of the problems of combinatorial explosion, limited computer storage space and the difficulties of finding a suitable form of representation it is not feasible for FAPD to make use of real-world knowledge at the present time.

FAPD is not intended to apply to all possible forms of a program design. The general form of the design must be similar to those used in the previous chapter. These contained a limited set of target language constructs interspersed with Enlish-like statements. In order to be analysed, these statements must conform to the grammar of a program design and consequently this grammar, together with the dictionary, define the variety of statements which can be analysed. Let us now evaluate the adequacy of this grammar.

At present, the only target language constructs which the grammar allows are conditionals and loops of the same format as those of the target language. As discussed previously, this prohibits the use of statements such as:

```
        I   <   10                              (7.3) and
        COUNTER  +   1                          (7.4)
```

which must be written in forms such as:

```
        I IS LESS THAN 10                       (7.5) and
        ADD 1 TO COUNTER                        (7.6)
```

Occasionally program designs contain statements of both sorts and so prior to developing FAPD a decision was required on whether or not all these forms should be recognised. As a matter of policy it was decided not to allow target language symbols such as +, -, and >. This is consistent with the policy that the principal beneficiaries of DACE will be novice programmers who might not be expected to know such terms.

The grammar also prohibits the use of punctuation marks such as commas and full-stops. This means that statements cannot be expressed as concisely as they might have been otherwise. For example, a statement such as:

INITIALISE A,B,C AND D (7.7)

must be written as:

INITIALISE A AND B AND C AND D (7.8)

Similarly, although the grammar is adequate for specifying simple operations such as:

ADD A TO B (7.9)

the text required to implement more complex operations such as:

ALLOW := ALLOWANCEPER * DEPENDENTS + EXPENSES (7.10)

is necessarily more protracted.

The assertion language used to represent the results from semantic analysis also limits the scope of FAPD in a similar way to the grammar of a program design. Any program design which cannot be represented by the assertion language is beyond the scope of FAPD. Since it can represent only a subset of a programming language at present, we need to evaluate whether this subset is adequate. As the examples given in chapters 5 and 6 and Appendix D show, the subset is adequate for designing a variety of programs. In this respect it would seem that the limitations it places on the variety of examples which can be analysed are less than those imposed by the grammar of a program design.

FAPD analyses program designs which have been formulated according to the principles of structured programming. However the main vehicle for this technique namely procedures, is excluded from this framework and therefore represents a limitation of DACE. Hence programmers cannot use FAPD to define and test several sub-procedures which comprise a super-procedure. Ideally a programmer should be allowed to decompose a complex task into simpler sub-tasks. These sub-tasks may need further decomposition and so sub-procedures may need to call sub-sub-procedures and so on. Hence, future research could investigate the possibility of extending FAPD so that it could analyse such program constructs. This is comparable to a programmer running and debugging the sub-procedures before testing the procedure which calls them.

This section has evaluated the Framework for Analysing Program Designs and in the following section the performance of DACE will be discussed and evaluated. Because DACE is based on FAPD, the points discussed in this section are also relevant to the system. Consequently, the following section will concentrate on evaluating how FAPD has been implemented.

7.3 Evaluation of DACE

DACE has been used to test the Framework for Analysing Program Designs and was found to be capable of analysing many examples. However the examples on which it has been tested (see chapter 5 and Appendix D) may be thought of as a specification of its capabilities. Section 5.9 contained three program designs which have been rejected because they do not conform to the specified grammar of a program design. However, because of the method used for syntax analysis (see section 4.4), the syntax error message is very general and does not give any indication to the user of the point of occurrence of the error. In an evaluation exercise, the results of which were described in chapter 6, eight out of eighteen users considered this to be a disadvantage. The results from using DACE showed that the main cause of syntax errors were using the separator ** incorrectly and using statements which do not conform to the required format.

Let us now evaluate the programs that comprise the four phases of analysis and give some suggestions for possible improvements. The first operations on a design involve lexical and syntactic analysis. Lexical analysis is undertaken by the scanner which is relatively unsophisticated and merely entails scanning the dictionary for definitions of all words contained in the design. The system does not have the ability to recognise different words of the same derivation and thus such words will go unrecognised unless contained in the dictionary. A more sophisticated method of lexical analysis is obviously needed to overcome the problem.

A scanner for a high-level language often builds a symbol table which contains details of any variable names found. Since DACE does not produce such a symbol table at present, the efficiency of the syntax analyser could also be improved by the inclusion of such a data structure.

For example, after parsing a statement such as:

SET A TO 1 (7.11)

the fact that A has been used as a variable name instead of an article could be recorded in a symbol table. At present, whenever A is met in subsequent lines within the same design, DACE cannot detect how it has been analysed previously and hence it will try to parse it as an article again. In cases such as this, the symbol table could be used so that A is always analysed as a variable name before calling the back-up mechanism to consider other possibilities. The results from using DACE showed that a word which is parsed as a variable name in one line can, in some contexts, be ignored in subsequent lines. In these cases a symbol table could be used to ensure that such words are retained.

The sets of class instances on which semantic analysis and generation of comments are based have been implemented as MICRO-PLANNER consequent and antecedent theorems. These theorems have proved a good choice and have allowed the system to be easily extended in order to cater for a wider variety of examples. The manner in which they are called has also proved adequate for analysing the examples which have been used to test FAPD. At present, as soon as the semantic analyser forms some results, they are passed over to see if any comments can be generated (see diagram 11). Such an approach means that if ever the results from analysing a previous line need to be altered, substantial work is required in order to back-up and erase any assertions made by the class instances responsible for generating comments. For the examples contained in this thesis, considerable backing-up has not been necessary although as the number of examples is increased, consideration should be given to this possibility. In this respect, one possibility would be to run the semantic analyser in isolation and then the results from analysing a complete design could be passed over for the generation of comments. The choice of consequent and antecedent theorems would not be affected but it would mean that a back-up mechanism would be easier to implement. The adequacy of the system-user interface was discussed in the previous chapter. The main difficulties stemmed from the fact that DACE was still in the development stage and the student's lack of familiarity with the DEC 20/60 computer.

In conclusion, we can say that the decisions which have been taken during the development of the system have been justified by the results obtained from the operation of DACE. However, it has been noted that the system could be improved, and some suggestions for improvement have been made above.

7.4 Suggestions for Further Work

Suggestions for further research have been made throughout this chapter. However it is worth summarising the achievements of this research, and in so doing some additional areas which are also worthy of further investigation will be identified. First of all, analysing a program design has been identified as an area of research which should receive just as much attention as the similar area of automatic program understanding and debugging. It has been shown how the Framework for Analysing Program

Designs could be used to complement some of the existing theories of program understanding by using it as a possible front-end to some of these systems. However, to do this a method must be derived for representing and using facts contained in the problem specification. The framework which has been developed in this study can only do this by procedurally embedding such knowledge in the form of class instances. This approach has been rejected as too specific and an alternative approach would be to devise a method for representing this knowledge, which could then be used by a set of general procedures. If this was achieved, analysing a program design would be based on the following three sources of information:

a) class instances which are used to derive how common statements and phrases can be represented in terms of a particular programming language;

b) knowledge of the context derived from analysing preceding lines in the same program design; and

c) knowledge of the problem specification.

Since a class instance could then make use of two sources of information (i.e. (b) and (c)) consideration would have to be given to the organisation and calling of the class instances since both of these sources are equally important.

The problem of organising knowledge in this way is similar to the problem of how the four phases of analysis that comprise FAPD should be organised. Broadly speaking, these phases are called sequentially, with the results from one phase forming the input for the next. However Example 2 in chapter 5 discussed the possibility of integrating syntax and semantics in order to improve the depth of analysis. The benefits of doing so have been discussed by Winograd [Winograd 1972] and hence a possibility for future research would be to investigate the feasibility of this approach.

This research has also advocated how a special procedure, referred to as a class instance, can be used to recognise design statements and to generate any comments about a coded version of those statements. These class instances are the means by which DACE can undertake some useful analysis. They are conceptually similar to Ruth's [Ruth 1976] experts but they are used for different purposes. This research has shown that class instances represent a satisfactory methodology for work of this kind. Consequently the concept of a class instance provides a useful acquisition to the set of tools currently available to researchers in Al.

Finally FAPD has been implemented in a system, the results from which support the contention that FAPD represents a viable approach to the computer analysis of program designs. To evaluate the system it was used by a group of people with various levels of programming experience. This evaluation exercise seemed to indicate that using the system had some influence on their performance. Given the importance of the process of program design in the development of structured programs, in our opinion DACE represents a software environment which provides support to the programmer.