An empirically-based debugging system for novice programmers

Thesis

An Empirically-Based Debugging System for Novice Programmers.

Tony Hasemer, B.A., M.A.

This thesis is submitted in fulfilment
of the requirements for Ph.D. in Psychology,
30th September 1983.

Date of submission : August 1983

Date of award : 24 November 1983

ACKNOWLEDGEMENTS.

I would like to thank the following for their help:

Marc Eisenstadt, whose enthusiasm, encouragement, and common sense gave me the strength to carry on.

Danny Waite, who built the equipment which recorded my student subjects' interactions with MacSOLO/AURAC. His machine is a mathematical impossibility, and works perfectly.

Hank Kahney, whose experience with detailed protocols helped me turn hours of recorded speech into a sensible chapter 3.

Alexander Kahney, who monitored the "experts" experiments, transcribed all the data, and drew some very elegant diagrams.

Tony Steyger, experimental subject extraordinaire, who caught obsessive hacking from MacSOLO/AURAC and asthma from my cat.

Ann Jones, who shared the traumas of running two series of experiments and who never took the slightest notice of my grumbling.

Martin Levoi, whose willingness to interrupt his own research in order to massage the PDP-11/VAX-750 operating systems saved me many a long hour of frustration.

ABSTRACT

The research described here concerns the design and construction of an empirically-based debugging aid for first-time computer users, integrated into the Open University's SOLO programming environment. Its basis is an account of the processes involved as human experts debug faulty code, which account was later found to be supported by empirical tests on human experts. The account implies that an understanding of the intentions of the programmer is not essential to successful debugging of a certain class of programs. That class comprises programs written in a database-dependent language by users who are initially completely computer-naive and who during their course become competent to write simple programs which embody one or more basic AI techniques such as recursive inference. The debugging system, called AURAC, incorporates an explicit model of the debugging strategies used by human experts. Its understanding, therefore, is of programming in general and of the SOLO environment in particular. We present in the process a broad taxonomy of naive users' errors, showing that they can be divided into types, each type requiring a different debugging approach and indicating a different degree of expertise on the part of the perpetrator. SOLO is a conveniently delimited though nonetheless rich problem

domain.

Also described is a new version of SOLO itself (MacSOLO) which incorporates a large number of traps for the simple errors which plague novices, thus enabling AURAC to concentrate on the more interesting programming mistakes. AURAC is intended to operate after the event rather than whilst a program is actually being written, and is able via analysis of programming cliches and of data flows to isolate errors in the user's code. Where AURAC cannot analyse, or where its analysis yields nothing useful, it describes the corresponding section of code instead, so that the user receives a coherent output.

MacSOLO and AURAC together form a unified system, based upon the principles of Simplicity, Consistency and Transparency. We show how these principles were applied during the design and construction phases.

CHAPTER 1

INTRODUCTION AND OVERVIEW

1.1 THE PROBLEM

When computer programmers write programs, it is often the case that the programs do not work as expected. Experienced programmers have their own personal techniques, often involving the use of special facilities in the programming language concerned, for finding and curing these "bugs". Previous research (see below) has concentrated largely on assisting relatively experienced programmers to produce completely bug-free programs as rapidly as possible: by constraining them to write programs in an approved style; by providing an automated assistant which can derive a representation ("understanding") of the intended purpose or function of the program itself; or by exhaustive demonstration that the completed program will operate as expected under all possible circumstances.

These attempts have not so far been fully successful. Automatic debugging strategies which involve knowing or predicting the behaviour of a program under all possible input conditions can rapidly become very expensive - a combinatorial explosion of possibilities is all too likely to occur. Where use of the system has to be learned, or where it has to be informed of its user's programming intentions via some special meta-language, the load on the programmer can actually be increased rather than eased. Debuggers which derive a representation of the user's intentions for themselves are essentially trying to infer something correct from something (the program) which is known to be incorrect. This is again expensive, and also error-prone.

This thesis reports on the design, construction and behaviour of an automatic debugging system intended for naive users, rather than for relative experts. The system, called AURAC, operates within a new implementation of the Open University's SOLO programming environment, this new implementation being an integral part of the design philosophy behind AURAC itself. AURAC does not attempt, except at the very end of its analysis of a faulty program, to derive any internal representation of the purpose of the program. Instead, it mimics the methods of human experts. As will be seen in chapter 3, human experts can derive very

large amounts of debugging information from the program code alone, and only express a need to know its purpose once a quite high level of debugging has already been achieved.

AURAC embodies our own account of the human debugging process. This is that if the human has no access to the machine, debugging takes place in essentially three stages:

1) "Skim" the faulty code in much the same way as one might "skim" a newspaper article, looking for salient points. In this case the saliences are syntactic errors, including missing data;

2) Recheck the code looking for errors in higher-level segments of it, here identified as programming cliches;

3) Check the code again, attempting to follow data flows in order to establish that these "make sense", and identify the effect of sections of the code in terms of the program's overall purpose, if known.

These steps are not necessarily carried out in the order given, nor are they necessarily applied to the whole of the faulty code at once.

The account was developed from personal experience and observation, and was tested in two ways: (1) via its implementation as the AURAC system, and (2) via empirical work discussed in chapter 3.

An important point is our contention that expert programmers will examine faulty code in terms of its natural flow of control, rather than purely lexically. That is, they consider the individual lines of the main program in their order of execution by the machine, and where one of these lines embodies a call to a subroutine, the lines of the subroutine are investigated at that point. This is contrasted with an approach which would treat the main routine and its subroutines as distinct "units" which could be examined separately. AURAC does the same as the experts, and has modules corresponding to each of the above three steps.

Being an intelligent machine empirically based upon intelligent behaviour by human beings, AURAC claims to be an Artificial Intelligence project whose inspiration comes from psychology. We hope that it also offers lessons for software engineering and some valuable insights for Computer Aided Instruction. The interrelation between the four influences on AURAC is discussed in more detail at the end of chapter 6.

1.2 THE APPROACH

1.2.1 Philosophy

MacSOLO/AURAC can conveniently be regarded as two systems working together. MacSOLO provides the SOLO language itself and supports a comprehensive HELP facility. It also contains a large number of traps for individual low-level errors such as typing mistakes or unbalanced quote-marks. The intention behind its design was that where it is possible for an error to be detected at its time of entry from the keyboard, there should be a trap provided to obviate it, and an available explanation of the mistake. MacSOLO (which term includes the SOLO editor) thus removes from a user's code all the "silly" and uninteresting errors, all of which by their very nature are syntactic, leaving AURAC to handle progressive levels of semantic bugs. AURAC comes into play when the program has been written but fails to work. It consists of three modules each of which looks for a specific type of error rather than for individual errors: Higher Level Syntactic Errors; Cliche Errors; and Algorithmic or Data Flow Errors. (see below.)

The whole system is based on the principles of Compatibility (with existing SOLO implemetations), Consistency, Simplicity and Transparency. These criteria will be discussed further in chapter 2.

Ideally, of course, we too would like our students to be able to write error-free programs in the shortest possible time. There are four broad areas in which we might seek to assist them:

1) Via the language itself. The ideas of Structured Programming (Dijkstra 1972), of Top-Down Design (Mills 1971) and of Development Via Stepwise Refinement (Wirth 1974), were all present in the original version of SOLO, but as the language has evolved these desirable criteria have been to a greater or lesser extent sacrificed in the interests of greater flexibility or of compatibility with other versions.

2) Via program development aids along the lines of the Spadee system (Goldstein and Miller 1976, Miller and Goldstein 1977). We decided against this kind of approach because of its steamroller-like effect: the user is obliged to decompose his/her overall objective in a predetermined way. As will be seen, AURAC allows its users to work upon sub-parts of a program in any order.

3) Via rigorous proof of the correctness of users' programs. The now-classic method originally formalised by Floyd (1969) involves demonstrating, usually with the aid of a general theorem-prover, that the program will validly transform a set of assertions which completely describe its input into a set of assertions which completely describe its desired output. It is difficult to obtain a full and correct set of assertions, especially where a large program must be broken into simpler sections - perhaps requiring the derivation of further assertions (e.g. 'loop invariants' - see Waters 1979).

This has meant in practical terms that proving a program to be correct can be substantially more difficult and expensive than normal, hand-operated debugging processes - although this is not to deny its considerable theoretical potential.

4) Via exhaustive testing of the program in all applicable environment and input modes. This is a alternative to (2) - see the discussion of Persch and Winterstein (1978), below. Chapman (1981) has developed an interesting variant of this idea for programs written in LISP code, which relies on simple interactions with the user to establish a set of tests which can automatically be modified and re-run as the program itself is developed.

AURAC's approach is radically different from all of these: it attempts to simulate the activities of a human expert debugging the faulty code in isolation - that is, without access to normal debugging aids such as steppers or tracers - and in accordance with the above debugging account. In other words, AURAC is in the same position as an actual Open University SOLO tutor. Unlike (3), AURAC generates no rigorous proofs. Its most interesting theoretical result is the extent to which its almost entirely syntactic analysis can detect what would normally be thought of as semantic errors. Many researchers (e.g. Rich, Shrobe and Waters 1979) have commented on the usefulness of a less theoretical and more "human" approach to the problems of debugging; such an approach is here applied in full, to a simple programming environment in which the kinds of bugs which human experts tend to miss

(such as hidden side-effects) are reduced to a minimum.

A large part of the research described here has been to see how much debugging information could, at reasonable cost, be derived from an existing (buggy) program without knowing the program's overall purpose. AURAC therefore does not make a specific attempt to understand the programs on which it works - by translating them into some other representation as, say, Waters (1981) does in his Programmer's Apprentice. Instead, more or less stylised forms of actual SOLO code are stored in AURAC's two libraries (this has the advantage of making it easy and straightforward to augment the libraries), and the process of matching these to the buggy program - the process of mapping superior knowledge onto it - is one of recogniton. Where the recognition only just fails, AURAC assumes that the difference represents an error on the part of the user. As will be seen, user errors can be divided into a hierarchy of types, and this recognition process can be applied at each level in order to detect mistakes of increasing "semantic" content.

A system using Waters' idea of Plan Diagrams and generating a symbolic evaluation of the effects of the user code on a global database has been tried for SOLO (Eisenstadt and Laubsch 1980, Laubsch and Eisenstadt 1981). One of its problems was that the multiple-nested loops permitted in SOLO syntax tended to generate an explosive number of possible cases to be tested; and where the loops themselves contained errors, it was likely that no effect-description - and hence no "understanding" of the user code - could be generated. AURAC employs the following heuristics instead. If the innermost instruction in the nest of loops is not a conditional (CHECK or TEST), and if at least one valid path can be found through the tree of possible cases, then the loops are assumed to be correctly constructed, and alternative paths can be ignored. If the innermost instruction is a conditional test, the nodes of the tree of possible cases are examined in depth-first sequence until such time as the test has both succeeded and failed at least once, or until the cases are exhausted. These heuristics avoid any combinatorial explosion and in practice can spot the bugs of co-operative users.

1.2.2 System Organisation

Broadly, the system operates as follows. MacSOLO processes the user's input line by line as it is entered, auto-correcting or rejecting any line which is not syntactically correct. If the resulting program does not perform as expected, the user may summon AURAC. AURAC examines the faulty code (behind the scenes) in a way analogous to that of a human expert, and is capable of detecting an indeterminate number of errors at three levels of semantic content, in terms of (a) "higher-level" syntactic errors; (b) SOLO cliches; and (c) data flow errors. The underlying taxonomy of novice users' errors will be presented in chapter 3.

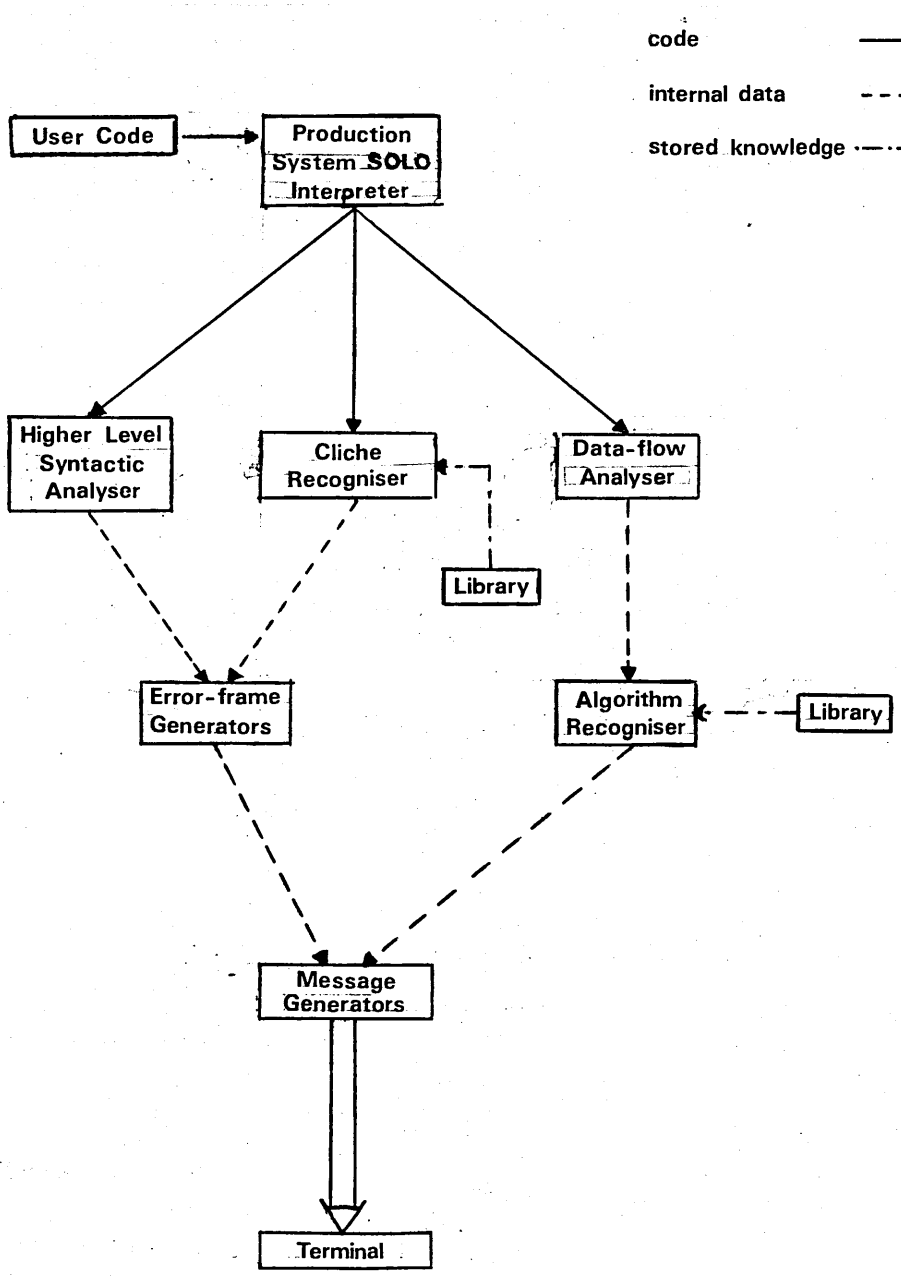

code
internal data
stored knowledge
User Code
Production System SOLO Interpreter
Higher Level Syntactic Analyser
Cliche Recogniser
Data-flow Analyser
Library
Error-frame Generators
Algorithm Recogniser
Library
Message Generators
Terminal

Fig. 1.1

AURAC consists of three modules, based upon our account of human debugging tactics. The original idea was that AURAC should whenever necessary be able to replace the human tutor and, via dialogue with the user if appropriate, perform the same debugging service for the student as the tutor would. In this context it should be stressed that in the experiments the tutors were given a printed listing of the faulty code, together with the statement (if appropriate) that there was a run-time error when the code was executed by SOLO. This is all the information that AURAC has at the start of its analysis, and although in real life the tutor would be able to find out a great deal more by watching the execution, asking questions of the student, or using STEP and TRACE, it was decided that this "cold start" had the advantage of being the most interesting form of the problem from the point of view of debugging algorithms. Furthermore, this approach avoids the need to have natural language parsers to understand the users' comments. Chapter 6 discusses the extent to which AURAC in its current form fulfils the original dream.

The structure of the AURAC system is as follows. The original user code is run through Module 1, the Skimmer, which uses the knowledge in its own production memory plus that of the MacSOLO interpreter to detect Higher Level Syntactic errors. Analysis proceeds line by line, which is a suitable thing for SOLO; there is no inherent reason why it should not proceed differently. Any errors found are stored for later explanation to the user. The lines of code are also passed to Module 2, the Cliche Recogniser, which uses a library of cliche forms to seek out errors at the "word" level. Finally Module 3, the Data Flow Analyser uses a considerable amount of knowledge specific to SOLO syntax in order to find data flow errors. This last module is also in specific cases able to use a second library, of algorithms, to detect algorithmic errors. All four kinds of errors are explained via a selection of canned messages; the selection of these also involves knowledge of SOLO syntax. Lines which contain no errors are "described" in a similar manner.

MacSOLO and all of the modules of AURAC are fully implemented and have been used by a total of thirty-six students and experimental subjects. The range of errors detectable by the Cliche Recogniser and by algorithmic analysis is entirely dependent upon the contents of AURAC's libraries. Since, as will be seen in chapter 4, evolving

full and complete libraries would be a long-term undertaking in itself, AURAC's libraries are limited; but they are sufficient to demonstrate AURAC's power.

1.3 SYSTEM HIGHLIGHTS

There follow some simple examples of the MacSOLO/AURAC system in use.

1.3.1 The SOLO Context

SOLO is a database-dependent language which has been used in the Open University's D303 cognitive psychology course over the last six years. Its purpose in this context is to allow students very rapidly to gain sufficient programming expertise to demonstrate to themselves one of the main points of the course: that computer models of human cognitive processes can be viable and useful. SOLO was therefore designed from the outset (Eisenstadt 1983) to be as simple as possible from the user's point of view, and to be embedded in a very user-friendly environment.

SOLO has strong affinities with LOGO and with a small subset of PROLOG. The user asserts one-way relational triples into a database, and then writes functions which, depending upon the contents of the database, can generate conditional, iterative or recursive inferences. The functions can also, of course, modify the database as they go along. User-defined functions in SOLO are LISP-like, in the sense that they can pass arguments to one another and that the bindings of these arguments are dynamic. SOLO itself handles all the complexities of file input and output; the user types HELLO or BYE in order to load or save his/her entire environment, including the current database.

SOLO is extraordinarily simple to use. It has only eight basic system instructions: NOTE, FORGET, DESCRIBE, CHECK, FOR EACH CASE OF, PRINT, TO and LIST. There is also a syntax-directed editor with its own small set of instructions. SOLO has no numerical primitives.

NOTE and FORGET, as might be expected, assert triples into the database and delete them respectively. These triples are of the form

```
NODE---RELATION-->NODE
```

the arrow signifying the one-way nature of the relation. The first "word" in any SOLO instruction is always a function-call of some sort (inbuilt or user-defined), so that the instruction

```
NOTE FIDO ISA DOG
```

inserts the triple which follows the word NOTE into the database. Similarly, the instruction

```
FORGET FIDO ISA DOG
```

deletes that triple. Several triples may be "hung" from the same first node, as above. The instruction DESCRIBE FIDO will then list all these triples in the standard SOLO format. By analogy with a NOTEpad, the most recent entry is displayed last.

```
FIDO
 '
 '---HAS-->FLEAS
 '
 '---LIKES-->BEER
 '
 '---ISA-->DOG
```

It is permissible to have two or more triples involving the same relation hanging from the same node.

CHECK searches the database for a triple corresponding to that given as its arguments. SOLO functions do not "return" any values as in Lisp, but CHECK used from top level prints PRESENT or ABSENT as appropriate. When used within a user-defined function, CHECK is also an IF-THEN-ELSE conditional construct, taking sublines which specify the further actions to be taken in either case. Each subline includes, as well as the action itself, a CONTROL STATEMENT which has essentially two alternatives: to EXIT from the current user-defined function, or to CONTINUE to its next line. CHECK instructions may not be nested within one another. Here is a simple CHECK line as used within a user-defined function. It DESCRIBEs FIDO if the triple FIDO ISA DOG is present in the database; if not, it prints a message:

```
20 CHECK FIDO ISA DOG
  A If Present: DESCRIBE FIDO ; CONTINUE
  B If Absent : PRINT "Nothing Found" ; EXIT
```

The line-number, the subline prefixes A and B, and the cosmetic "If Present" and "If Absent" are printed automatically when the user-defined function is being defined via TO, EDITed or LISTed. It is up to the user to insert the semicolon separators and appropriate control statements. The "action" part of a subline (DESCRIBE or PRINT in this example) can be left blank, but the control-statements are obligatory. In fact SOLO's editor will not allow CHECK lines or sublines to be written (or

edited so as to be) in an incorrect format.

FOR EACH CASE OF allows iteration through the triples of a single node, where these triples have a common relation. For this purpose it needs a "wildcard" chosen by the user in order to represent the non-specific third item in each matching triple. This wildcard must have a question-mark as its first character, and a corresponding variable-name beginning with an asterisk is bound to the result of each "case". For example,

```
10 FOR EACH CASE OF FIDO ISA ?WHAT
  A PRINT "FIDO ISA" *WHAT
```

is a FOR EACH line with its obligatory subline, the overall effect of which is to print out every database triple which begins "FIDO ISA ...". FOR EACH instructions may be nested up to five deep, and it is permissible to nest a CHECK instruction within a FOR EACH. However, it is not permissible to nest a FOR EACH within a CHECK.

Wildcards are also permitted in CHECK instructions, and the corresponding *-variables are similarly bound. Where there are several possible matches to a wildcard triple, the most recent entry is selected. The bindings in both kinds of instruction are dynamic.

Here is a simple recursive SOLO program which propagates the inference "so-and-so HAS FLU" through a suitable database. The database is shown beneath the program listing:

```
TO INFECT /X/

10 NOTE /X/ HAS FLU

20 CHECK /X/ KISSES ?P
  A If Present: INFECT *P ; EXIT
  B If Absent : EXIT


JOHN---KISSES-->MARY
MARY---KISSES-->ANDY
ANDY---KISSES-->MAUD
MAUD---KISSES-->FRED
FRED---KISSES-->JOAN
```

The top-level call is INFECT JOHN, whereupon the triple JOHN---HAS-->FLU is added to the database by line 10. On line 20 the CHECK instruction gives the value MARY to the variable *P, so that on subline 20A (executed rather than 20B because the CHECK search succeeded) INFECT is recursively called again with MARY as its new argument. The triple MARY---HAS-->FLU is then added to the database. This sequence continues until JOAN is the current argument to INFECT, whereupon the CHECK search fails. Subline 20B is then executed and recursion halts.

The D303 course notes require the student to carry out a few simple experiments which demonstrate the above, and to undertake a number of assignments in which they write simple programs to give them experience of the use of the conditional CHECK form and of recursion. The FOR EACH loop is included only as an appendix to the notes, which students are free to ignore until a later stage if they wish. For all of these exercises Higher Level Syntactic analysis is sufficient to reveal any errors. Beyond that, for the students, comes Summer School.

At Summer School they are offered a choice of four projects each of which is intended to involve them in building a computer model of some aspect of human cognitive activity. The projects reflect areas of psychology which the same students will be studying at other times during the Summer School course, namely Memory, Perception, or Thinking. Students are free to choose some other SOLO project if they wish, but they almost never do so.

1.3.2 MacSOLO

MacSOLO's error traps are a series of fifty-seven demons each of which inspects a single word or line of user input so as to check it for the presence of a specific error. The demons are spread amongst the various modules comprising MacSOLO.

The input-reader is permanently in "line mode". It rejects all characters other than the basic alphanumeric set plus seven SOLO-specific punctuation signs (RH and LH parenthesis, semicolon, double-quote, slash, space and apostrophe). If any other characters appear in the input they are converted into spaces (see below), and extra spaces are auto-inserted if necessary to disambiguate the punctuation, e.g. to convert ISA*D to its only legal SOLO form ISA *D, or to convert 10ABFOO to 10AB FOO. The reader also checks for other auto-correctable errors such as unbalanced quotes or parentheses. However, the design philosophy of MacSOLO (chapter 2) prohibits full auto-correction of these: instead the user is forced to make the correction him/herself before the line will be accepted.

The parser comprises the spelling corrector, the spacing corrector, and the traps for incorrect numbers of arguments. The spelling corrector is semi-automatic, requiring confirmation from the user before the correction is made; the spacing corrector is fully automatic but announces its actions if any; and if the wrong number of arguments is entered, the user is forced to retype that input line.

Each SOLO primitive in MacSOLO is a LISP function containing at least one demon; for example the function PNOTE, executed whenever a NOTE instruction successfully passes through the reader and the parser, always checks that the triple to be NOTEd is not already present in the database (multiple copies of the same triple are not permitted). Other traps occur in the MacSOLO interpreter, for such errors as calls to unknown procedures, or endless recursion.

Finally, a substantial number of the 57 demons mentioned occur in the MacSOLO editor, catering for such editor-specific mistakes as missing line-numbers or missing control-statements. There follows a transcripts condensed from several user sessions in order to demonstrate some of the error-traps in action. Sections in square brackets are annotations. User input is shown in **bold face.**

SOLO: **NOTE FIOD BROTHER ROVER**

[The user types a NOTE instruction to the SOLO prompt, followed by a RETURN. MacSOLO queries the spelling of one of the input words.]

When you typed FIOD, did you mean FIDO? (Y or N) Y

[The user replies with Y, so the corrected version is accepted. The normal database description of the node FIDO results.]

OK...

FIDO
'
'---ISA-->DOG
'
'---BROTHER-->ROVER

[Next, the user tries a PRINT instruction.]

SOLO: **PRINT "FIDO ISA DOG**

[The RETURN key will not work, and a warning message is issued:]

Brackets don't balance!

[This message appears either as a flash across the top of the terminal screen, returning the cursor to the end of the word DOG, or - on printing terminals - as a new line, after which the user's incomplete input is retyped. Once the correct quotes have been entered, the RETURN key will function normally.]

[Now a FORGET instruction:]

SOLO: **FORGET FIDO ISADOG**

Space error(s) assumed.

Corrected input line:

FORGET FIDO ISA DOG

OK...

FIDO
'
'---BROTHER-->ROVER

[Full auto-correction was possible here because

the words FIDO, ISA and DOG, being parts of the database, must also be in the user's dictionary. A similar mistake involving as yet unknown words would have the following result:]

SOLO: **NOTE FIDO HASFLEAS**

Wrong format. Type HELP if you don't understand.

SOLO: **HELP**

The proper way to use NOTE is:

NOTE node1---relation-->node2

See HELP NOTE.

[The user now LISTs and then tries to edit FOO, an existing user-defined procedure:]

SOLO: **LIST FOO**

```
TO FOO /Y/

10 CHECK /Y/ ISA DOG
  A If Present: PRINT /Y/ "IS A DOG" ; CONTINUE
  B If Absent : EXIT
```

SOLO: **EDIT FOO**

edit line...#

[This is the edit prompt, to which the user types:]

NOTE /X/ HAS FLEAS

Missing line number!

[The input is ignored and the prompt returns.]

edit line...# **20 NOTE /X/ HAS FLEAS**

Undeclared parameter!

[This is because the current definition of FOO does not include a declaration of /X/. As above, HELP would generate this explanation.]

edit line...# **20 NOTE /Y/ HAS FLEAS**

[This time, the input is accepted as syntactically

```
        correct, and the user types DONE so as to exit
        from the editor.]
edit line...# DONE
```

Other features of MacSOLO, such as the tracer, single-stepper, spelling corrector and part-word recogniser, are discussed in section 2.3.

1.3.3 Debugging: AURAC

Given that all very simple errors have been removed from the user's code, AURAC applies its three-stage analysis to what remains. Two examples follow. The first uses the INFECT program described earlier in section 1.3.1, but in a buggy version such as might be produced by a student:

```
TO INFECT /X/

10 NOTE /X/ HAS FLU

20 CHECK /X/ KISSES ?P
  A If Present: INFECT /X/ ; EXIT
  B If Absent : EXIT


JOHN---KISSES-->MARY
MARY---KISSES-->ANDY
ANDY---KISSES-->MAUD
MAUD---KISSES-->JOHN
```

The program as shown contains an error: on line 20A the student has arranged for the recursive call to INFECT to have /X/ as its parameter, rather than (as it ahould be) *P. Naturally, if the top-level call to INFECT is e.g. INFECT JOHN, such that the CHECK on line 20 succeeds, the resulting recursion will not terminate. After twenty levels of recursion, MacSOLO halts and delivers the run-time error message "Recursion limit exceeded." It is expected that the user will then summon AURAC to debug the program.

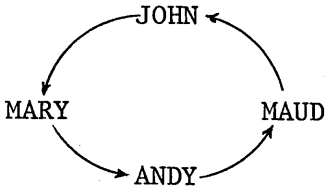
The first stage of AURAC's analysis of the buggy INFECT is a "skim" of the code, using a production system to look for errors such as faulty conditional exits, unreached code, non-existent but referenced data triples - and non-terminating recursion. When the above error is detected, AURAC informs the user:

"Your procedure INFECT repeatedly calls itself with the same parameters:

INFECT JOHN

This caused your run-time error."

(The question of the suitability of AURAC's messages will be returned to later). But, assuming that the student manages to correct line 20A, he/she will still have problems because there is a loop in the accompanying database:



This also results in endless recursion of the program. A second call to AURAC will now result in the message:

"There is a loop in your database via:

JOHN MARY ANDY MAUD JOHN

This caused your run-time error."

The skimmer stage of AURAC's analysis is all that is required to find the two bugs in INFECT.

Fig. 1.2 shows a second simple example. TRY is not a genuine SOLO procedure, and is invented solely for our current purposes. If put through the SOLO interpreter, it would achieve nothing of interest; but discussion of it as below (and in chapter 4) will give a useful "feel" for AURAC's operations.

Line 10 of TRY ascertains whether or not a certain triple is currently present in the database. If so, it FORGETs it; if not, it NOTEs it. Line 20 again checks for the presence of a certain triple, but does nothing with the result of its search. Line 30 is a simple PRINT statement:

```
TO TRY /X/

10 CHECK /X/ IS UP
  A If Present: FORGET /X/ IS UP ; CONTINUE
  B If Absent : NOTE /X/ ISA UP ; CONTINUE

20 CHECK AURAC LOVES TONY
  A If Present: EXIT
  B If Absent : EXIT

30 PRINT /X/ "IS UP."
```

Fig. 1.2

AURAC's skimmer would notice the double EXIT on sublines 20A and 20B, plus the fact that this prevents line 30 from being executed at all. If it happened that either of the referenced triples /X/ IS UP or AURAC LOVES TONY were absent from the database, this fact too would be noted. As a result of this stage of analysis, AURAC reports to the user:

```
Line 10: CHECK-triple does not exist in your database.
Line 20: CHECK-triple does not exist in your database.
Line 20: Double-Exit --> Line 30 not reached.
```

The second stage is an inspection of larger "chunks" of code, here identified as programming cliches. Where these can be detected, they can yield data about errors which would otherwise be difficult to detect. For example, line 10 of TRY is a cliche (called FLIP-FLOP) which FORGETs a given triple if it is present in the database, and NOTEs it if not; but it contains an error: ISA instead of IS on subline 10B. (It is assumed that both IS and ISA appear correctly elsewhere in the user's database, so that MacSOLO's spelling corrector could not resolve the problem.) When AURAC detects the cliche itself it can also point out the error:

```
Line 10 seems to be intended to FORGET the triple /X/ IS UP
  if it exists, and to NOTE it otherwise
  but on line 10B of TRY you have written ISA
  when perhaps you meant to write IS.
On line 20, TRY looks in the database to see if the flag
  AURAC LOVES TONY is Present, and if so EXITs
  Otherwise, it EXITS.
On line 30, TRY prints a message.
```

Notice that AURAC distinguishes between a TRIPLE, which may contain a wildcard, and a FLAG, which is composed entirely of constants.

As can be seen, the printout attempts to explain the discovered cliche without mentioning the fact that it is a cliche. Lines 20 and 30 are merely described, since they contain no cliches at all.

The third stage of analysis is an attempt to follow the flow of input data through the program, in order to be able to say whether or not all data is correctly used (that is, that bound variables and parameters are referred to, that any NOTEd triples are at some point CHECKed and later FORGOTten). This results in the following two messages to the user:

```
..and also the triple NOTEd on line 10B TRY
   is never CHECKed.
..and also the triple NOTEd on line 10B TRY
   is never FORGOTten.
```

As will be described later, the final module of AURAC is also able to check each code line against sample lines from a library of algorithms. When all lines of a given algorithm are found, AURAC states that the program will perform according to that algorithm. This analysis is covered in more detail, along with the other two modules, in chapter 4.

1.4 RELATED RESEARCH

As has already been mentioned, and as will be described in detail in chapters 2 and 3, the design philosophy behind the MacSOLO/AURAC system involves the division of the spectrum of novice errors into four broad categories or "levels".

Wertz's PHENARETE system (1979) also employs a division of error types. He refers to spelling errors, unbalanced parentheses, wrong number of arguments supplied to functions, and the like as "surface" errors. At what he calls a "meta" level of analysis, PHENARETE seeks out such problems in the user's program as unintentional loops, endless recursion, and unreachable code. Unlike most automatic debuggers, PHENARETE requires no knowledge of the programmer's intentions, and is - disregarding the fact that the two systems are designed to operate in different language domains - precisely analogous to MacSOLO/AURAC's first two levels of syntactic analysis. Interestingly, PHENARETE uses

> "a set of pragmatic rules describing general program constructs and stereotyped methods to repair inconsistencies. These rules formalise and express explicitly the knowledge activated by every programmer when he is reading a program." (Wertz 1979, p.953)

The difference between this and AURAC's approach is one of scale. Wertz's rules concern for example the correct form and operation of loops, and are language specific. They are similar to the production rules used in AURAC's Higher Level syntactic analysis. In other words, they represent just one aspect of the knowledge which a human debugger applies to the task. AURAC introduces two other aspects: knowledge of programming cliches and understanding of data flow.

The assumption that only trivial debugging information can be derived in the absence of "understanding" is common to many debugging systems. Ruth (1976) actually defines "understanding" as the system's ability to map an algorithmic description of a process onto the corresponding code. The algorithm in his case is supplied by the user; the mapping ability is inbuilt. Ruth says:

> "The basis for analysis of the type presented here lies in the simple observation that in writing a program the student is guided by a general plan of attack or strategy for effecting the desired purpose. As the usual computational environment admits only the sequential and deterministic execution of operations, such a plan must take the form of a finite set of well-defined steps with unambiguous rules at every point in the execution for determining which step is to be performed next. That is, the plan must be an algorithm." (Ruth 1976, p.66)

Our Summer School students do not come to the terminal with their algorithms ready. Rather, we expect them to discover the process of designing an algorithm by practical experiment and by discussion with their tutors.

Kahney (1983) has produced an in-depth study of the behaviour of novice programmers. He concludes that raw novices will often acquire a knowledge of programming terminology before acquiring any knowledge of how or why the corresponding programming techniques are used. He calls this state of affairs the possession of "empty concepts". Lacking any knowledge of techniques, novices will then use their own world experience to provide meaning for the programming terms. Which of course tends to lead them away from, rather than towards, the construction of a correct algorithm. Sleeman (1977), discussing a similar point as it arises in Computer Assisted Instruction techniques, says:

> "The main problem...is deciding whether or not the answer given by the student is equivalent to that generated by the algorithm." (Sleeman 1977, p.780)

and he concludes that the only satisfactory way of talking about algorithms is via some purposely-designed formal language. The trouble with this approach from our point of view is that our students would then have two formal languages standing between them and the understanding (of artificial intelligence) which we want them to gain.

AURAC's approach is to derive as much debugging information as possible in the absence of any algorithmic criteria - which in the terms described here means in the absence of any "understanding" of the buggy program. Only in the very final stages of analysis does it engage in a (simple) dialogue with the user in order to establish an understanding of the variables used in the program, and hence to decide whether or not the program's code embodies some expected algorithm.

Ruth again:

> "The algorithm description given to the program analyser should be general enough to cover the broadest possible range of programs. But it must be sufficiently specific so that there is no ambiguity concerning the intended procedures and effects." (Ruth 1976, p.67).

Ruth is thinking here of algorithms written in terms of what he calls "universal" constructs and which are in fact programming constructs - loops, conditionals, flags and the like - of which our students are usually quite unaware. There is an added complication in that our students may well (they are encouraged to) add arbitrary print-statements to their code for cosmetic or debugging purposes. Since provision of data for these printouts may incur temporary alterations to the database (a NOTE followed later by a CHECK in which the PRINT instruction is embedded) they can,

if not coded correctly, cause apparent errors on subsequent runs of the same program. However, there is no sense in which they are part of the algorithm for the task in hand (that is, for the program minus its cosmetic printouts), and AURAC must be able to check them separately. It is not safe, as it would be in the LISP-like language used by Ruth's students, simply to ignore them.

Ruth's system uses a "generative semantic grammar", which may be regarded as a set of rules whereby an algorithm is "translated" into code. The translation process is assumed to be perfect, so that if a given algorithm cannot be translated into the program submitted by the student, the latter is said to be incorrect. AURAC, by contrast, does not generate any code at all. Its method is one of recognition: if the steps of the algorithm can be recognised in the user's code, AURAC assumes that the program is doing the correct thing as far as the algorithm is concerned (see section 4.6). As Ruth himself says, the number of possible variations on any given algorithmic theme is not great, so that in AURAC's present limited context it is satisfactory merely to store any possible variants.

However, at some point in the analysis both Ruth's system and AURAC have to ask "is this section of user code (perhaps a variable, perhaps a construct such as a formalised loop or conditional form) equivalent or non-equivalent to my derived or stored version?". SOLO is not flexible enough to allow its formal constructs CHECK and FOR EACH CASE OF to be rewritten in alternative SOLO code, as one might, for example, rewrite the LISP conditional COND construct as an IF-THEN-ELSE. So for AURAC the problem of recognising a sequence of lines as being equivalent to an existing construct does not arise (although its cliche-recognition module does something similar). But the differences between the two methods of ensuring variable-equivalence are interesting. Ruth's analysis proceeds in the context of a current environment, and variables are assumed to be the same variable (one in the user code, one in the model) if their VALUES are the same. AURAC's test for equivalence is in terms of data flow: a variable deep inside a subroutine can be traced back, despite any changes of name, to the point at which it entered the program - usually as an argument in the top-level call. AURAC ascertains by simple yes/no dialogue with the user the SIGNIFICANCE ("meaning") of the value associated with that datum. The stored algorithms are written in terms of significances, so that AURAC is able to recognise a line of user code as being equivalent to a step in the algorithm. In practice, of course, the stored "significance" is merely a special token,

and AURAC associates the corresponding user token (variable name) with it.

Goldstein (1975) also requires from the user an input of "description of intent". This description is written in a special declarative language, which describes the program's intended result - i.e. it is not an algorithm. Goldstein is dealing with LOGO programs designed to operate a turtle (Papert 1971) so that specific pictures are drawn. It is thus, given a complete knowledge of the task domain, possible to infer the intended algorithm from a faulty program.

But this early work of Goldstein's is useful in AURAC's context for the notion of "clearing up". It is desirable that on completion of a subpart of a drawing (corresponding to a subroutine in the program) the turtle be moved if necessary from wherever it has stopped to the start-point of the next subpart; and that at the end of the whole program it be returned to some standard rest position ready for the next program. These interfacing sections are referred to by Goldstein as "preparatory steps", and are analogous in SOLO to such operations as clearing the database of temporary flags, resetting counters to zero, and so on. The principle that overall a program should make no difference to the position or orientation of the turtle is directly applicable

to SOLO's database: if a SOLO program is allowed to leave permanent data changes behind it, it is unlikely that any two runs of the same program will behave in the same way, and as may be imagined such apparent inconsistencies can play havoc with a raw beginner's conceptual model of the SOLO machine.

Persch and Winterstein (1978) have worked on the idea of proving a program by running it under all possible conditions, ensuring that the various supplied sets of input data cause every possible program path to be executed at least once. Such an approach is unsuitable for raw novices, since a mass of data about the behaviour of their programs under input conditions other than those they themselves supply is liable to increase, rather than to reduce, their confusion. In this context a "stepwise" approach is preferable: allowing the user to specify the input conditions and restricting the debugging mechanisms to evaluating the program's performance within that environment.

Osterwell and Fosdick (1976) concerned themselves with analysis of FORTRAN programs. Their handling of variables is interesting. They say that:

> "1. No variable will be used in a computation (referenced) until it has previously been assigned a value (been defined).
>
> 2. A variable, once defined, will subsequently be referenced before the variable is redefined or the program terminates." (Osterwell and Fosdick 1976, p.910)

Their analyser, called DAVE, determines whether or not either of these rules would be violated for "any sequence of statement executions". (The precise meaning of this last phrase is not made clear). Instead of following the flow of control during some kind of re-execution of the faulty program, DAVE constructs for each "unit" of the program a "labelled flow graph" reminiscent of a Plan Diagram (Waters 1976), which carries information concerning any variables referred to or defined within that unit, plus data concerning the unit's position in the overall program. These graphs are then assembled into a higher-order graph so that the interrelations between program units and subroutines can be seen. This larger graph is then explored bottom-up, i.e. from the lowest-level subroutines upwards, so that each lower-order graph is examined exactly once. Anomalies can be detected during this process where they violate one of the above two rules.

Coincidentally, AURAC relies upon the same two rules during its data flow analysis of the progress of variables - though it does so lexically rather than by value - and it detects essentially the same errors. However, AURAC's system of expectations and satisfactions (see section 4.5) is far simpler than the above complex build-and-search method - without, apparently, any loss of useful information.

A somewhat similar, though more comprehensive, approach is adopted by Adam and Laurent (1980). They comment:

> "Often the program, though syntactically correct, still contains semantic errors."

Whilst this is true, it highlights one of AURAC's a priori assumptions: that an error classified in this way as a "semantic" error (for example, the use of a wrong but syntactically correct variable name) may not indicate semantic confusion on the part of the programmer. In trying to work as far as possible without any knowledge of the programmer's intentions, AURAC avoids imputing to the programmer any intentions which he/she did not have. In other words, AURAC does not generate long explanations of ASSUMED semantic misunderstandings, but confines itself to pointing out the coding error (e.g. the wrongly-named variable). AURAC, like LAURA, analyses the surface code of

the buggy program, rather than first converting it into some stylised internal representation.

Lukey (1980) describes a system called PUDSY which embodies his own theory of program understanding and debugging. PUDSY compares a program specification with the program's actual achievement (both described by means of assertions) in order to detect bugs. Unlike AURAC, it cannot handle more than one subroutine at a time, it does not cope with syntax errors, and it cannot deal with recursive programs. (It also cannot handle GOTO statements, but neither can AURAC, since these do not occur in SOLO code). PUDSY's approach is first to "segment" the buggy program into "chunks". This pre-analysis stage is quite elegant, but is not required for SOLO programs since SOLO is designed so that each of Lukey's "chunks" would be a single line of SOLO code, with attached sublines. In other words, SOLO itself performs the "chunking".

Beyond this point, PUDSY and AURAC diverge quite sharply in their analytical methods, but the most obvious difference is that AURAC does not require the provision of an assertion describing the program's specification. As mentioned already, such assertions can be very difficult to generate.

Thus, AURAC attempts to provide the simple explanations of errors which naive users require. In order to do so, it uses the kinds of knowledge, and the debugging techniques, used by tutors who are themselves accustomed to dealing with the work of those same novices. It does not, like some systems designed for novices, require the user to know what he/she is doing at any specific level - for example it does not, as Chapman's system does, demand that user-defined tests be supplied for sections of the program. As will be seen in chapter 6, one possible future for AURAC sees it as the essential kernel element in a larger and more rigourous program-proving system: an element which can very substantially reduce the amount of work required from more theoretically-based debuggers. In SOLO terms alone, to obviate the combinatorial explosion of "cases" referred to above is a considerable achievement.

1.5 OVERVIEW OF REMAINING CHAPTERS

Chapter 2 introduces MacSOLO as an advanced dialect of the original SOLO language: the common kernel of instructions, MacSOLO's design principles, and its advantages over other versions are described. Chapter 3 presents some empirical observations. It gives the frequencies of various error-types as found by MacSOLO/AURAC under practical conditions, and compares them with results found by other researchers in similar studies. It also

describes experiments designed to shed light on expert debugging style, and considers the implications of this for the design of AURAC. Chapter 4 discusses AURAC itself: its organisation, debugging methods, and its results. Chapter 5 comprises a complete session transcript in which a novice tackles a Summer School project using the MacSOLO/AURAC combination. Finally, Chapter 6 offers a critical appraisal of AURAC and describes its future potential.

CHAPTER 2

MACSOLO

2.1 THE SOLO KERNEL

There have so far been four main versions of SOLO. The original, mainframe-only version was implemented in BASIC and is the one from which most of the available data about user response to SOLO has been derived (Eisenstadt 1978). This version was in universal use by our students from 1978-1981. In 1981 Summer School users were offered an improved and microcomputer-based version implemented in UCSD PASCAL (Gawronski and Eisenstadt 1982) and in 1982 a new mainframe version, also written in PASCAL (Eisenstadt 1983) was produced. The latter is now in general use, although it is still not in its final form. The version described in this thesis (Hasemer 1982) is implemented in MacLISP, and is therefore called MacSOLO. It has been used at Summer Schools by nineteen students and at other times in twenty-seven experiments of between one and four days' duration each, involving both OU students and non-students.

It is tempting, but facile, to regard SOLO as a "toy" language, merely because any function or program written in it must inevitably be an extremely simplified version of what could be achieved in, say, PROLOG, LISP or PASCAL. But this simplicity is in fact SOLO's major asset. SOLO instructions and their effects are for the most part direct, straightforward and obvious. Anyone whose first programming language was one such as LISP will remember how difficult it was to understand how to operate the language itself - let alone how to build bug-proof models with it. And in forcing its users to construct very basic models of whatever cognitive ability they are studying, SOLO helps to clarify their thoughts and to improve their understanding. Although it is possible to write quite sophisticated programs in SOLO (we have seen on the one hand a production system, and on the other a model of the Pavlovian responses of a rat), these are in an important sense MIS-uses of the language: it was designed, and operates best, not as a general purpose simulation language but as a means to practical demonstration of elementary AI principles. In this sense SOLO is more akin to an elegant statistics or graphics package: it is not intended to do anything other than what it does, and it does that superlatively well. The SOLO segment of the Summer School course is always enormously popular and heavily over-subscribed.

SOLO's basic instruction set is as follows:

1) NOTE takes three arguments: a "node", roughly equivalent to a noun in English, a one-way "relation" such as ISA, and a second node. NOTE asserts this triple into the database.
For example:

NOTE FIDO---ISA-->DOG

Users are not required to type the arrow, which is printed by SOLO merely as a reminder of the unidirectional nature of the relationship expressed. When NOTE is used from top level, an automatic DESCRIBE (see below) of its effect on the database occurs.

2) FORGET takes similar arguments and deletes the triple from the database. For example:

FORGET FIDO---HAS-->FOURLEGS

FORGET is also followed by an automatic DESCRIBE when used from top level.

3) CHECK takes similar arguments and searches the database to see if the triple represented by those arguments is there. The search is essentially a pattern-match on the triple:

```
CHECK FIDO---ISA-->DOG
```

prints "Present" if the triple is found, "Absent" if it is not. When used within a user-defined procedure, CHECK may be given as its third argument a wildcard (a question-mark followed by zero or more characters forming together a single "word"). If the search succeeds a variable, lexically equal to the wildcard but with the question-mark replaced by a star, is assigned a value which is the result of the search. For example, if the triple FIDO ISA DOG is present in the database, the instruction CHECK FIDO ISA ?WHAT results in the value DOG being bound to the variable *WHAT:

```
10 CHECK FIDO---ISA-->?WHAT
  A If Present: PRINT *WHAT ; CONTINUE
  B If Absent : ; EXIT
```

Within a user-defined procedure, the CHECK instruction has two obligatory sublines, labelled "If Present" and "If Absent", so that CHECK forms a conditional construct. On each subline appears an optional further instruction, followed by a "control statement" which essentially permits EXIT from the current procedure or CONTINUE to its next line. CHECK instructions are not permitted on CHECK sublines.

4) FOR EACH CASE OF is the loop construct, and again takes a triple as its three arguments. It must be given a wildcard as its third argument. The loop iterates until all matching database patterns are exhausted (and allows the operation within the loop to add new such patterns), or until a conditional CHECK within the loop orders an exit. FOR EACH CASE OF takes just one obligatory subline (the "body" of the loop) which may be another FOR EACH CASE OF instruction, up to five deep, or any other legal SOLO line, or a call to a user-defined procedure:

```
10 FOR EACH CASE OF FIDO---ISA-->?WHAT
  A PRINT "Fido isa" *WHAT
```

5) PRINT prints any quoted string or strings. Unquoted items are assumed by PRINT to be either variables or formally-declared parameters whose values are to be printed.

6) BYE causes logout, and writes the user's current data-base to disk.

7) TO summons the editor for definition of a new user-defined procedure.

8) EDIT permits editing of an existing user-defined procedure. The prompts and other formatting during TO or EDIT sessions vary from dialect to dialect of SOLO.

9) DESCRIBE prints a standardised SOLO formatting of the triples which in the current database are associated with a single node, that node being given by the user as DESCRIBE's single argument:
DESCRIBE FIDO causes a printout such as:

```
FIDO
 '---ISA-->DOG
 '
 '---HAS-->FLEAS
```

10) LIST also takes a single argument, the name of a user-defined procedure, and prints out a listing of that definition in a standard indented format including cosmetic additions such as "If Present" and "If Absent" on CHECK sublines.

11) DUMP prints out the entire database, including any user-defined procedures. This is mainly of use for students who wish to take home a hard copy of their work.

SOLO users do not have to concern themselves with any of the complexities of login/logout or of file handling. SOLO responds to the instruction HELLO with a prompt for the user's personal identifier - usually the user's student number, although MacSOLO permits the use of alphabetic login names. If the user has logged into SOLO before, his/her individual database file is found by means of the personal identifier, and contains any database the user may previously have created, plus any previously-defined procedures. BYE writes the user's database and current procedure definitions to the same file. The user thus has at his/her disposal a personal SOLO machine, and is encouraged by the course notes to think of "the computer" in those terms.

Users are also encouraged to view the database as a manipulable, rather than a sacrosanct, object. The instructions SAVE (which does the same as BYE without logging out) and RESTORE (which does the same as HELLO without logging in) allow them to experiment with non-standard databases, subsequently returning to their original version.

2.2 DESIGN PRINCIPLES

As with the earlier versions of SOLO, MacSOLO is more than merely a language: it is a complete programming environment. Within this environment are embedded the intelligent auto-debugging aids which are the main subject of this thesis. Thus, whilst the basic MacSOLO system with its static error-traps will be referred to simply as MacSOLO, the auto-debuggers will be treated separately under their generic name AURAC. The main advantage of MacSOLO from the points of view of the Course Team and of researchers is its flexibility: it is usually a simple matter to make semi-permanent changes to or additions to existing SOLO facilities, in order to try out new ideas.

During the design and implementation of MacSOLO/AURAC, five principles were kept in mind:

1) Compatibility. Radical changes to the original design philosophy were not practicable. In other words, the "model" of the SOLO machine as presented to students in the various course notes had to remain clearly recognisable in any new version of the language. There were two reasons for this restriction: first, it would be unfair to the students themselves, who under OU regulations are allowed to take as many years as they wish to complete a course; and second it

was hoped to be able to compare users' progress using MacSOLO with those using one or other of the other dialects.

2) Consistency. This and the two following principles are drawn from the work of duBoulay, O'Shea and Monk (1981).

Consistency implies that any instruction within a programming language should always do the "same" thing. A glance back at section 2.1 will show that the CHECK instruction is poorly consistent, since it behaves differently according to whether or not its triple contains a wildcard, according to whether it is used from top level or from within a user-defined procedure, and (in older dialects of SOLO) according to any prior use of FOR EACH CASE OF. However, the Consistency principle does not prohibit modification of an instruction's effect via optional arguments (as in HELP).

We add a further important dimension to the principle, not mentioned by DuBoulay et al. This is the requirement that where an English word is used to effect a system instruction, the word and any companion words should reflect as far as possible their normal English usage. In this respect NOTE and FORGET are inconsistent: the instructions should be for example NOTE and ERASE. (REMEMBER and FORGET

would not be suitable since the normal usage of "remember" can imply that the remembered item had been PREVIOUSLY stored - in other words there would be likely semantic confusion between REMEMBER and CHECK).

3) Simplicity. This principle asserts that a language instruction should have just one conceptually integral action - as for example BYE and FOR EACH CASE OF do. And again NOTE (in older dialects) and CHECK are offenders in this respect - see below. However, Simplicity does not forbid an instruction from working on different types of datum, as will be seen in the later description of MacSOLO's Recogniser.

4) Transparency. This is the degree to which the user is able to "see what is going on" inside the SOLO machine as he or she operates it. It has a high importance for beginners, who often find the whole computing environment extremely confusing: it offers reassurance that the user's model of the machine is the correct one. In many cases (as with FORGET, say) this reassurance is given by the machine's own automatic response to the user's input. But there are other occasions - e.g. during the execution of long programs - where continual printouts of each and every database change can be more confusing and irritating than no printouts at all. In the latter case it is preferable to provide a

simple means (e.g. TRACE and STEP) whereby the user can see the detailed machine response only if he or she wishes to do so.

5) It was desirable that MacSOLO be usable in all the contexts in which the other SOLO dialects were used. Since it would evidently differ from them in many respects, it needed a simple method whereby its users could find out what these differences were. This led to the design of the HELP system, which when used from top level is in effect an on-line SOLO manual. It was also desirable that MacSOLO itself - as distinct from AURAC - should trap all those errors which it is possible to trap at their time of entry, so that the success or otherwise of the more intelligent debugging systems could be judged apart from the mass of simple, typing-mistake level errors to which novices are inevitably prone. As will be seen, this led to the design of a number of error traps and aids which are in themselves novel.

Since there is no clear dividing line between the status of novice and that of expert - the one fades into the other - there was from the design point of view a problem concerning the corresponding error messages. Other SOLO systems are verbose with their messages, and students have been seen at Summer School, hard pressed for time and so making lots of "silly" mistakes, brought to screaming point by the machine's relentless and extended explanation of each. MacSOLO's solution to this difficulty is given below, in section 2.3.2.

There is also a problem here related to the Consistency of the user's model of SOLO. If the messages refer (as in other systems they do) to SOLO in the first person, this suggests to a novice user that the machine is far more intelligent than in fact it is. Its failure to behave as intelligently in other contexts can look like mere cantankerousness. MacSOLO/AURAC's messages refer to the SOLO machine in the third person.

2.3 EXTENSIONS

The remainder of this chapter concerns MacSOLO: the changes made to the SOLO machine in the light of Lewis's (1980) work and of subsequent experience, and in the light of the above five criteria.

SOLO's original NOTE instruction prohibited overwriting. For example, if the database already contained the triples

```
FIDO
 '
 '---ISA-->DOG
 '
 '---HAS-->FLEAS
```

then an attempt to NOTE the new triple FIDO HAS FOURLEGS would fail, and would elicit from SOLO an error message to the effect that "FIDO HAS FLEAS already". That is, the triple FIDO---HAS-->FLEAS, which had two elements in common with the new triple, would prevent insertion of the latter.

This was thought to be better from the raw novice's point of view since it prevented him/her from amassing a huge database full of largely redundant data. However, this restriction also prevented any meaningful use being made of the FOR EACH loop. So, once a student had written a procedure using FOR EACH, the NOTE instruction automatically

went into a "multiples permitted" mode, so that databases suitable for the new procedure could then be constructed. In a later version of SOLO, that change was brought under the direct control of the user. But either way it was a flagrant violation of the principle of Simplicity.

It turned out in practice that students do not in fact create the feared huge databases; a "no-multiples" NOTE is therefore unnecessary. MacSOLO's NOTE allows any triple to be entered into the database at any time - provided, of course, that no identical triple is already there. As our students progress, it soon becomes a trivial matter for them to write an overwriting procedure should they need one (i.e. CHECK for the triple, and FORGET it if present). MacSOLO's NOTE is permanently in non-overwrite mode, and is unaffected by any use of FOR EACH.

A somewhat similar problem arose with CHECK. If the database contained more than one triple each of which had two identical elements:

```
FIDO
 '
 '---HAS-->FOURLEGS
 '
 '---HAS-->FLEAS
```

as it would have to have if FOR EACH were to operate upon it, then the instruction

```
CHECK FIDO HAS ?WHAT
```

would have no single result. The original SOLO (automatically) prohibited the use of wildcards in CHECK once FOR EACH had been used in a user-defined procedure, and students produced some extraordinarily tortuous code in their attempts to get round the restriction. A later version of SOLO again brought this change under the user's control (it was Lewis's suggestion in both cases), but MacSOLO's more elegant solution is to allow wildcards in any use of CHECK; if more than one result is possible, that most recently entered is chosen. This is Consistent with the idea of a NOTEpad, on which the most recent entry is the last (and lexically the lowest, as it is when MacSOLO prints out portions of the database as above).

As a convenience for users, FORGET is allowed to take a wildcard in its third-argument position. And after many requests for such a facility from users, an additional control-statement STOP has been included. In contrast to EXIT, which returns control to the next-higher recursive level, STOP returns control immediately to top level. All of the other SOLO instructions mentioned in section 2.1 have been implemented just as they were. For Summer School users a number of extra system instructions are provided. The following are also found in the micro-computer dialect of SOLO:

1) LET used within a user-defined procedure, takes two arguments separated by an "=" sign and assigns a given value to a given variable-name. e.g. LET *X = FIDO. MacSOLO does not normally permit global variables - these can produce weird-looking run-time errors which can baffle even an expert - hence the restriction of LET to use within user-defined procedures.

2) INPUT is again only usable from within user-defined procedures, and for the same reason. It takes any number of arguments, each of which is the name of a variable, e.g. INPUT *A *B *C. At run-time, INPUT causes execution to halt whilst the user inputs one or more values to be bound to these variables. This allows students to write simple interactive procedures.

3) TEST operates like a mini-CHECK. It takes similar sublines, labelled "If Yes" and "If No", and merely ascertains whether or not a given variable has a given value:

```
10 TEST /X/ = FIDO
  A If Yes: PRINT "OK" ; CONTINUE
  B If No : EXIT
```

The alternative of asserting a suitable triple into the database and then retrieving it with CHECK is long-winded and time-consuming.

4) There is also an even faster database-retrieval mechanism involving the apostrophe. If, for example, the database contains the triple FIDO BESTFRIEND ROVER, an instruction such as

NOTE (FIDO'S BESTFRIEND) ISA DOG

will evaluate the parenthesised phrase before making the database assertion, so that the intuitively correct thing happens. In three years' experience we have never known an unaided student to use this construct, and we suspect that this is partly because it violates the principle of Consistency: the necessity in other contexts to write such odd-looking instructions as

NOTE (FIDO'S HAS) ARE INSECTS

lessens the usefulness to naive users of the original good idea.

5) DIR is short for DIRECTORY, and gives a brief listing of all the primary nodes currently held in the database, plus a listing of the names of any user-defined procedures.

6) KILL deletes the definition of a named user-defined function.

The following additions are not found on the micro version of SOLO, and are except for UNDO all facilities for use within SOLO's editor:

7) UNDO operates from either top level or in EDIT mode to nullify the effect on the global database (if any) of the user's most recent action. This latter may be (a) a top-level call to a system primitive; (b) a top-level call to a user-defined procedure; or (c) the editing of one line of a procedure. UNDO will not "undo" its own actions. If there is nothing in UNDO's temporary memory, MacSOLO announces "Nothing to UNDO".

8) RENUMBER. MacSOLO's editor, when summoned via TO rather than via EDIT, supplies automatic line-numbering in multiples of 10. The user is then free to insert additional lines after, before or between any lines already written. RENUMBER simply restores the interval of ten between any two adjacent lines. Although this may seem so trivial a refinement as to be virtually pointless, users do appreciate

it as something which removes one more source of potential confusion from the listings of their programs.

9) RENAME allows, as its own name implies, the name of any user-defined procedure to be changed. Via a brief dialogue with the user, it also allows the formal parameters to be changed if desired. RENAME actually creates a new copy of the original procedure, and gives it the new name and parameters. As the relevant HELP entry (summoned by HELP RENAME or HELP COPY) explains, that fact can be made deliberate use of in cases where the user needs to create several near-identical versions of the same program. This need arises most frequently in the two-column subtraction project.

10) SHOW. Occasionally, students want to insert the normal top-level instructions DESCRIBE or LIST into their programs. In accordance with the principle of simplicity, the best way to accomodate that need is to restrict those instructions to the top level/runtime interpreter, and to provide a separate means of database inspection for use from within the editor. In order not to have two differently-named instructions which do conceptually the "same" thing, SHOW has been made to perform either a LIST or a DESCRIBE of any node or procedure-definition, depending upon the contents of the database at the time.

There follow now brief descriptions of the various user-aids offered by MacSOLO. Unless otherwise stated, there is no directly equivalent facility in any other version of the language.

11) SPELLING CORRECTOR. This was designed largely on the basis of Lewis's (1980)observations concerning the failures of the original mainframe dialect. Its hit-rate measured as a percentage and using an average dictionary is 88%, and its details are discussed in Appendix F.

12) RECOGNISER. This is intended to reduce rather than to correct user errors. Under different conditions, it does conceptually the "same" thing, in accordance with the principle of Consistency. When the user has typed in only part of a word - usually two or three letters are sufficient - depression of the ESCAPE key will cause the word, if it is known and if the typed substring is enough uniquely to identify it, to be filled out automatically. If there is more than one possible match, the recogniser fills out that part common to all of them, and displays the alternatives at the top of the terminal screen (or on a new line). If there is no possible match, it does nothing. A similar facility is found in the TOPS-20 operating system. Another related feature of the MacSOLO recogniser is that when the user is

partway through the typing not of a word but of a complete instruction (i.e. is between words) depression of the ESCAPE key prints out the current state of the syntax-filter - that is, it suggests what type of word (procedure-name, node, relation etc.) should come next.

13) TRACE. The current mainframe dialect of SOLO offers a rudimentary TRACE facility which prints the name of each procedure within a program as control enters it. The microcomputer dialect has a similar arrangement out of the user's control but restricted to the top line of the terminal.

Neither of them has a stepper, and experience has shown that in the latter's absence a much more elaborate tracer is desirable. Given a stepper, a simple tracer is sufficient. MacSOLO's tracer prints the name of the procedure entered plus the values of its formal parameters if any, and prints a note when control exits from that procedure. Announcements concerning subroutines are suitably indented. For example, the recursive INFECT procedure from section 1.3.1 would, when traced, produce the following printout:

```
SOLO: INFECT JOHN

   Enter INFECT JOHN
    Enter INFECT MARY
     Enter INFECT FRED
      Enter INFECT JANE
      Exit INFECT
     Exit INFECT
    Exit INFECT
   Exit INFECT

SOLO:
```

The tracer is intended only as a means of finding out in which subroutine a procedure failed - it is not, as STEP is, meant to be used as a diagnostic tool. It is put into operation by the single-word instruction TRACE, and switched off again by the instruction UNTRACE. There is no provision for tracing individual procedures, since again it is intended that STEP be used for this purpose.

14) STEP. It is expected that STEP will be summoned if and when a user's program has failed at run-time. At such times the single instruction STEP will put MacSOLO into stepping mode, and will re-run the same procedure using the same input data. At other times (no student has used the stepper in this way yet) the stepper can be activated by giving it the full procedure call as arguments:

STEP CONFIRM FIDO ISA DOG

steps the procedure CONFIRM with its normal triplet of

arguments. STEP uses the normal MacSOLO interpreter to re-run the faulty procedure line by line. As throughout the rest of MacSOLO, progress from one line to the next is effected by depression of the RETURN key. At each step, the original user code from the corresponding procedure is printed out, with all variables and parameters evaluated as far as is possible at that stage of execution. Any messages, such as automatic DESCRIBE after NOTE, occur as they would from top level; and where the line assigns a value to a variable (i.e. where it contains a wildcard CHECK) that variable and its new value are explicitly shown.

FOR EACH nests are printed out in a top-down fashion, in the same sequence as their evaluation by SOLO.

It is thus genuinely possible to watch as MacSOLO executes a procedure or program in slow motion. STEP is the most Transparent of MacSOLO's facilities, and on several occasions formerly bemused students have suddenly grasped what is going on as they used the stepper (presumably they suddenly acquire a viable model of the SOLO machine). The stepper is as much a didactic as a debugging tool. On the other hand, as mentioned earlier, to have this degree of transparency constantly available would seriously slow down those users who did not need it. An example of STEP in action is given in chapter 5.

15) UNDO cancels the effect on the database of the user's most recent action. This action may be a top-level call to a system primitive, or a top-level call to a user-defined procedure (which may have any number of subprocedures). UNDO is also available within the editor, where it cancels all edits done during the current editing session.

16) AURAC has three commands: DEBUG, HOLES and INFORM. DEBUG initiates the full analysis as described in chapter 4, and is expected to be used in the same way as the STEP instruction. Analysis terminates with the issuing of the message or messages from the final, algorithmic, section of AURAC. INFORM followed by a procedure name then gives the more detailed results of AURAC's analysis of that particular procedure; and HOLES prints a list of any data triples referred to during execution of the program but absent from the database.

2.3.1 The Editor

As already mentioned, the editor can be summoned in either of two modes: via the instruction TO in order to define a procedure from scratch, or via EDIT in order to modify an existing definition. This is a case where the principle of Consistency requires what is essentially the same underlying suite of (LISP) programs - the editor - to be presented to the user as two separate environments. It

is not at all obvious to a naive programmer that the two operations are the same. In the former mode MacSOLO prints as prompts appropriate line-numbers in multiples of 10, leaving the user free to insert additional lines between existing ones as definition proceeds. In the latter mode the prompt is "edit line... " to remind the user to specifiy which line of the existing procedure is to be modified (probably, for British users, the abbreviation "no." for "number" would be better here than the American hash-sign equivalent).

Where sublines are required (i.e. after CHECK, TEST or FOR EACH CASE OF plus its arguments has been entered) these are automatically prompted for in either mode, with standard cosmetic printouts such as "If Present:". If any necessary control statements are omitted, these are also prompted for: this latter omission counts as an error in all versions of SOLO.

Supplying a line-number followed by a carriage-return deletes the corresponding line, plus its sublines if any. Sublines can be modified independently, but cannot be completely deleted without deletion of the whole of the main line on which they occur.

All of the relevant error traps, plus a number of editor-specific ones, operate during either editing mode, as does the full HELP system. Overall, it actually becomes quite difficult to enter a syntactically incorrect line into SOLO.

2.3.2 Error Traps And HELP

We will not list here all the errors which MacSOLO can trap at their time of entry (for those interested, the errors listed as Simple Syntactic in section 3.1.1 are the ones), but will merely point out that MacSOLO can currently issue a total of some 120 error messages, corresponding to perhaps sixty distinct errors (the 2:1 ratio is explained below). This is a very considerable advance over any other implementation of SOLO, and may approach the full set of Simple Syntactic errors possible in SOLO. In this section we describe how these traps are integrated into MacSOLO's ubiquitous HELP system.

In accordance with the principles of Simplicity and Consistency, the single word HELP summons help from MacSOLO at all times: from top level, whilst editing, whilst stepping and so on. Under non-error conditions, this simplest use of the HELP system will generate something appropriate - such as the information that the user is at top level, or is using the editor - together with pointers

to other and potentially useful pages of the HELP file. If a particular page of HELP is likely to be needed when HELP is typed (for example, when defining a procedure and after typing the semicolon on a CHECK subline, the user probably wants to know about control-statements) that page is printed out. One very important message which is often omitted from HELP systems is that of how to escape from the current environment - e.g. the user might not want to know about control-statements at all, but how to get back from the editor to top level. In MacSOLO this message is always the same, since control-A is its universal escape-to-top-level interrupt, and where appropriate MacSOLO's HELP messages carry this information.

As far as the user is concerned, each page of HELP is identified by the one-word name of its topic, so that a second way to use the system is by giving HELP an argument specifying the page-name: HELP EDIT, HELP NOTE, HELP APOSTROPHE. MacSOLO is able to recognise all the commonly-used synonyms for any HELP page-name; in this context HELP HELP (or HELP ME) generates a list of topics and carries the simple advice "Type HELP folowed by whichever of these seems closest to the help you need and let SOLO guide you from there.". When MacSOLO signals an error, its error message is designed to be as brief as possible consistently with providing enough information

about the error itself. For example, giving one of SOLO's major instructions other than three arguments will result in the simple comment "Wrong format". The point of the emphasis on brevity is that, as already mentioned, the kind of error which MacSOLO without its debugger is designed to trap is the kind which could well be the equivalent of a slip of the pen, and relentless explanations of these are usually counterproductive.

However, the longer explanation is available (which is why the number of possible error messages is roughly twice the number of trappable errors), and as usual carries pointers where appropriate to other potentially useful HELP pages. When one of these Simple Syntactic errors does occur, the HELP system consults a record which it secretly keeps in the user's own database file. The record tells the system whether or not this user has ever read the longer - and more explanatory - version of the message. If not, a reverse-video flash across the top of the terminal (a new line on printing terminals) advertises the existence of the additional HELP: "Type HELP if you don't understand". In this way the HELP system is what we call "ostensive", and this is a useful attribute because novices do not yet have the expert's habit of consulting the machine itself when things go wrong. Changing HELP from a passive to a semi-intrusive format made a very substantial difference to

the amount of use it received. Raw beginners find themselves being constantly reminded about the existence of HELP; as their expertise grows the facility fades into relative obscurity.

There are a few error messages whose meaning is so self-evident that no extended version is required. For example, unbalanced quotes on a string following PRINT cause MacSOLO to refuse to accept the input line (that is, the RETURN key refuses to work) and the announcement at the top of the terminal that "Quotes don't balance". Once the necessary correction has been made, using the RUBOUT if appropriate, the line will be accepted.

MacSOLO's run-time error messages are of necessity somewhat longer, since "enough information about the error" then becomes a larger amount; e.g:

"Unbound variable - *A has no value on line 30 of MATCH"

The longer version is still there, if the user types HELP, but in these cases the pointers to other HELP pages include advice to use the debugging aids described above, such as:

"Use the stepper - HELP STEP tells you how"

2.3.3 Scope Of Variables And System Switches

The scope of variables - this term to include both locally-bound variables and formal parameters - is currently dynamic in MacSOLO as it is in LISP. That is, once a variable has been bound it retains that value during any subroutines, but is unbound once control returns to a higher routine. This decision cannot be justified with any concrete evidence - rather, the evidence points to students having extreme difficulty with the concept of binding itself, never mind its scope. The microcomputer dialect of SOLO uses strictly local binding (i.e. lexical rather than dynamic scope) so that local values must be explicitly passed to subroutines as the values of formal parameters if required. So far, this difference has not been noticed in practice, but - in case it ever should be - one of a series of system switches available only to tutors allows selection of local or global binding as alternatives to the default dynamic.

Other switches allow the tutor to alter such system parameters as the number of permissible lines in a single procedure, the maximum depth of the run-time stack (= depth of recursion), or to turn off the spelling corrector. In other words, to tailor the environment to suit exceptional programming needs. There are also a number of minor modifications, such as improving the readability of SOLO

code by allowing the names of nodes, variables and formal parameters to be of arbitrary length, but these are not important here.

CHAPTER 3

EMPIRICAL OBSERVATIONS

This chapter concerns two things. First we present an error taxonomy for novices' errors, based upon AURAC's actual and achievable results; and second we report on a series of experiments to observe the debugging behaviour of human SOLO experts.

3.1 ERROR TAXONOMY

This is a broad categorisation of the errors which SOLO users actually do make. It is not of course intended as a complete classification of the behaviour of novices, but merely as a useful way of talking about "kinds" of errors. The four categories are: Simple Syntactic Errors, Higher-Level Syntactic Errors, Cliche Errors, and Data Flow Errors.

However, the traditional dichotomy between "syntactic" and "semantic" errors - the former being trappable at compile-time whilst the latter only show up at run-time - does not map perfectly onto AURAC's categories. There are three reasons for this. Firstly, there is a type of error which, although it would normally be classed as semantic, can certainly be trapped at compile-time (in SOLO's case, at its time of entry from the keyboard). As a hypothetical example, consider a FORTRAN user whose program contains

```
Y = X/0
```

in an attempt to assign to Y the result of dividing X by zero. The "divide-by-zero" pattern could certainly be detected by a simple demon long before run-time. This type of error is considered by AURAC to belong in the Simple Syntactic category.

Secondly, there is another type of error (for example, giving the wrong number of arguments to a system primitive or to a user-defined procedure) which can also be detected and explained via simple demons. Whether such errors are trapped at or before run-time depends upon the particular implementation of the error-signalling mechanism in use. But in either case they do not merit the attentions of an intelligent debugging system. This type of error, too, is classed as Simple Syntactic.

Thirdly, it does not seem psychologically plausible to apply the term "semantic" to an error when considered in isolation. Whether or not the error represents any degree of semantic confusion depends entirely upon the degree of understanding of the individual who made it. A novice making the above divide-by-zero mistake might genuinely not understand that dividing by zero is an unreasonable operation to attempt. Conversely an expert would presumably know perfectly well what he/she intended to do, and would type the zero, if at all, merely by accident.

In other words, what may be a "semantic" mistake on the part of a beginner might not indicate any semantic confusion at all on the part of an expert. Therefore we prefer not to classify the errors themselves within a syntactic/semantic spectrum, but to consider instead the expertise or otherwise of the person making them. The four categories presented here are ordered along this dimension. That is to say, errors other than Simple Syntactic are considered to be in themselves some evidence of increasing expertise on the part of their perpetrators. We believe that this gives a more useful way of talking about errors from the point of view of intelligent debugging systems.

3.1.1 Simple Syntactic Errors

Experience during the design and initial evaluation stages of MacSOLO, later confirmed by a reading of Wertz (1979), indicated two subsets of what are normally classed as syntactic errors. The first, which we call Simple Syntactic errors, are characterised by being highly language-specific: such things as unbalanced quotes on printable strings, unbalanced parentheses where these are required, reference whilst defining a procedure to formal parameters which have not been declared, spelling mistakes and so on. All of these can be trapped as they are entered. Although any given one may be entirely language-specific, it is clear that similarly simple traps or demons could be constructed to find analogous errors in other languages. There is nothing here that requires intelligence on the part of the machine. The full list of Simple Syntactic errors trappable by MacSOLO is given in Appendix G.

Simple Syntactic errors arise through misuse of the language's syntax at an elementary level - usually involving single tokens or "words". Above them, and for the most part after them on the average learning curve, come Higher Level syntactic errors, which are specific not to the word-level syntax of the language but to its syntax at the level of programming constructs. Where a construct such as a conditional form or a formalised loop, or a comparable

programming technique such as recursion, is misused, a Higher Level syntactic error can occur.

3.1.2 Higher-Level Syntactic Errors

Higher Level syntactic errors are mistakes which could still be mere slips of the expert's attention, but which from a novice might well betoken a much more serious lack of understanding. For example, faulty exits from a conditional causing non-execution of subsequent code, unbound variables, undefined procedures, and endless recursion. These errors will usually not show up during any "static" analysis of the code, but will reveal themselves only at run-time. And again although the means of automatically detecting them may be language-specific in any given implementation, there is no reason in principle why the same detection algorithms should not be transportable to any other language. As described in chapter 4, AURAC employs a production-system as a substitute SOLO interpreter to re-run the faulty code and to find the following kinds of error in it:

Failure or success of conditional form leaves unreached code.
Faulty conditional exits (four types in SOLO).
Reference to non-existent data triple.
Loop containing ineffective test (two types in SOLO).

Unbound variable - parhaps chained to earlier error.
Endless recursion (five types in SOLO).
Undefined procedure
Incomplete database chain.

The first of the latter group is not necessarily indicative of an error - whether it is or not depends upon the particular input data - and hence is expressed by AURAC as a "possible" error. It was felt desirable to point out the unreached code in any case. Since AURAC's Higher Level syntactic analyser calls upon the normal SOLO interpreter during its simulation of run-time effects, and since MacSOLO is also a stand-alone system, there is a degree of crossover between the two levels of syntactic analysis. That is, the final four errors in the above list can be detected by either module. The difference is that AURAC's explanations of them are potentially richer. For example, there are five causes of endless recursion in SOLO (this will be further explained in chapter 4). These are all the same error in MacSOLO alone; five distinct errors if AURAC is used.

3.1.3 Breakdown Of MacSOLO/AURAC Users' Actual Errors

Matthew Lewis (1980) produced an exhaustive analysis of one year's use of the original SOLO, using a behind-the-scenes computer log of each student's moment-by-moment interactions with the machine. He evaluated the then-existing error traps and made many suggestions for improvement (particularly of the spelling corrector) most of which, where appropriate, have been incorporated into MacSOLO.

MacSOLO has collected data on 3643 user entries, each line of SOLO (plus its sublines if any) being considered a separate entry. Amongst these lines 784 (21.5%) were recorded as "error lines"; as well as genuine user errors, they included lines which invoked some error-related facility such as HELP. MacSOLO's list of potential errors (given above) is substantially longer than Lewis's, but for the sake of comparison only those errors whose frequencies were found to be above two percent are listed again here. The resulting table of frequencies is as follows:

	Actual	Percent
All "errors"	784	100.0
Word recogniser used	266	33.9
Attempt to NOTE existing triple	178	22.7
Unbound variable	98	12.5
Spelling corrector	90	11.5
Missing control statement	76	9.7
Stepper used	75	9.6
Wrong number of arguments	71	9.1
Undefined procedure invoked	61	7.8
HELP system used	60	7.7
Parameter slash error	48	6.1
Attempt to redefine existing procedure	46	5.9
Missing line-number	28	3.6
Attempt to DESCRIBE non-existent node	23	2.9
Attempt to FORGET non-existent node	15	2.3

For comparison, there follows a similar table of the error frequencies found by Lewis. The percentage of error lines he found in his whole sample was 26.1, broken down as follows:

All errors	100.0
Spelling or quotes error	34.4
Wrong number of arguments	25.4
Undefined procedure invoked	9.5
Missing line-number	7.5
Missing control statement	4.8
Non-terminating recursion	2.6
Attempt to redefine existing procedure	2.6
Unbound variable	2.4

Closer comparison is difficult because of the large differences between the original SOLO used by Lewis and MacSOLO. The original version had a rather complicated means of access to such things as its HELP facility, and this contributed a substantial quota of errors. Similarly, such things as illegal characters or spurious control characters were counted as errors - MacSOLO simply ignores them and adjusts inter-word spacings if necessary. In all, roughly one third of the errors noted by Lewis either cannot happen or are auto-corrected in MacSOLO. It is interesting, however, to see from the above tables that whilst the overall proportion of errors remains about the same, (21-26%) their distribution is very different. For example, the "unbound variable" error moves from the bottom of Lewis's list to near the top of MacSOLO's. This error in

itself suggests some degree of progress on the part of the novice, since it implies an attempt on his/her part to make use of a general programming technique - rather than, as Lewis's listed errors do, an attempt to get to grips with the basic SOLO machine. Notice also the considerable use of MacSOLO's recogniser (which is intended to obviate spelling errors) and the much lower incidence of spelling errors in MacSOLO's list.

DuBoulay (1979) made a similar analysis of novice users of LOGO, which has strong similarities with SOLO. His figures are as follows:

ERROR	PERCENTAGE
Call undefined procedure	28
Insufficient arguments	16
No line number	11
Extra text	10
Turtle off drawing area	10
Variables misused	6
Wrong type of argument	4
Command leaves a value	3
Device claiming violation	3
Number too large	3
Stack overflow	2

A direct comparison between the four most common errors in SOLO and LOGO has been produced by Eisenstadt and Lewis (1982), making allowances for differences between the two systems. Their table is reproduced below, with the equivalent MacSOLO figures added:

	SYMPTOM	% OF ALL ERRORS		
		LOGO	SOLO	MACSOLO
1.	Spelling/typing/misquoting	28	34	19.6
2.	Wrong number of arguments passed	18	18	9.1
3.	No line number	12	9	8.2
4.	Call to undefined procedure	12	9	7.8

The interesting points to notice here are (1) that MacSOLO sharply reduces the incidence of the first two types of error - errors which are not implementation-specific - and (2) that MacSOLO's own four most common errors are:

1.	Attempt to NOTE an existing triple	22.7
2.	Unbound variable	12.5
3.	Spelling corrector	11.5
4.	Missing control statement	9.7

That is to say, a user making mistakes in MacSOLO is likely to be making more interesting mistakes than do users of the other two languages. More interesting in the sense that (apart from the ubiquitous spelling error, which plagues

even experts) they have a higher semantic content. MacSOLO's aim of obviating "silly" errors appears to have been achieved. It is unfortunate that there is no equivalent system against which to compare AURAC's more intelligent debugging modules.

3.1.4 Cliche Errors

Thirdly come errors which concern conceptual "chunks" of code, and which we have identified (cf. Brotsky 1981) as programming CLICHES. A cliche is a line or group of lines of code - not necessarily contiguous - which is found repeatedly when large quantities of code are analysed, and which is always used to effect essentially the same purpose.

A cliche can be regarded as a programming CONSTRUCT which has not yet been formalised in the host language. For example, a group of SOLO lines which together have the effect of deleting one triple from the database and replacing it with a similar but different one (such as might be required in a program which accumulated some kind of numeric total) is one of the cliches which we have so far identified. Cliches are, naturally, not only highly language specific but also domain-specific - and could even be user-specific; so it is not to be imagined that the set given in chapter 4 is anything like the full set, even for SOLO.

Brotsky's work on cliches was done for LISP code. Although ours was done on SOLO code, our ability to detect and to make use of them is broadly in line with his. AURAC uses a library of cliche "skeletons" (naturally, the library is known as the Cupboard) which it matches one by one against sections of user code in order to detect cliches. It then looks for a mismatch between the code details and an "example" from the same library in order to pinpoint errors.

There are nine cliches in AURAC's cliche library, and AURAC allows a maximum of two errors per cliche line (see chapter 4). The commonest SOLO line is probably the CHECK line, which typically contains between six and fourteen distinguishable elements (atoms). So in some sense the number of errors potentially detectable via cliche analysis is very large, although in practice they are almost invariably errors in control-statements or in variable-names. Our hypothesis from this is that the data triple is a much more solid concept in our students' minds than either flow of control or variable binding. We have not listed all possible cliche errors because they would be tedious and because they can be inferred immediately by inspection of the cliche library (section 4.3).

3.1.5 Data Flow Errors

Fourthly come data flow errors: errors of control-flow having been analysed as Higher Level Syntactic errors or Cliche errors as above. There is firstly the question of the flow of data into and out of the database. Deriving from Goldstein's idea of "preparatory steps" (section 1.4, above), AURAC expects this flow overall to be nil. Secondly, an error occurs in the flow of data through a procedure when a piece of code is syntactically correct in all respects but is given the wrong data to work upon. This can alternatively be regarded as an algorithm error, and AURAC adopts this approach because of its potential for offering a richer or more comprehensible explanation to the naive user. Data flow analysis also concerns itself with such matters as the presetting of database flags into some known condition, and ensuring by means of clear-up routines that one run of a program does not leave unwanted data in the database which could affect future runs. Data flow analysis involves the use of substantial amounts of knowledge which is specific both to the language and to the problem domain, but it has proved possible to restrict this knowledge to a small number of (LISP) routines within AURAC. The main principle of data flow analysis as described here - the system of balanced "expectations" and "satisfactions" - should be applicable to other database-assertive languages, and as will be described in chapter 6 it is hoped soon to

have the opportunity to test this assumption. Data flow analysis reveals the presence of only a small number of errors: a NOTE instruction not accompanied by a CHECK instruction or not accompanied by a FORGET instruction; a FORGET not accompanied by a NOTE; or a bound but unreferenced variable. (Referenced but unbound variables are trapped elsewhere, as above). Where the algorithm library can be used - i.e. when it contains the appropriate algorithm - this number is potentially increased by the number of steps in the algorithm.

3.2 EXPERT DEBUGGING STYLE

3.2.1 The Sample Program

The following report concerns a series of experiments to observe the debugging behaviour of human SOLO experts, and to find evidence in support of (or against) the account given in chapter 1. For convenience we reprint the account here. It is that if the human expert has no access to the machine, debugging takes place in essentially three stages:

1) "Skim" the faulty code in much the same way as one might "skim" a newspaper article, looking for salient points. In this case the saliences are syntactic errors, including missing data;

2) Recheck the code looking for errors in higher-level segments of it, here identified as programming cliches;

3) Check the code again, attempting to follow data flows in order to establish that these "make sense", and identify the effect of sections of

the code in terms of the program's overall purpose, if known.

Five expert SOLO programmers, all of whom had tutored SOLO at Summer School, were asked to debug the genuine faulty program shown in fig. 3.1, where each line and subline has been separately numbered for the purposes of explication. The program was originally written by a student for an assignment as mentioned in chapter 1. The student's database which accompanied the program is also shown, as fig. 3.2.

```
1.  TO IMPLICATE /X/

2.  10 PRINT /X/ "IS A CRIMINAL"

3.  20 FOR EACH CASE OF /X/ FRIENDLY ?A
4.    A CHECK *A HAS POLICERECORD
5.    AA If Present: IMPLICATE *A ; EXIT
6.    AB If Absent : NEXTCASE

7.  30 FOR EACH CASE OF /X/ PAYS ?B
8.    A CHECK *B HAS POLICERECORD
9.    AA If Present: IMPLICATE *B ; EXIT
10.   AB If Absent : NEXTCASE

11. 40 B CHECK *C PAYS

12. 50 PRINT "THAT SEEMS TO BE THE WHOLE GROUP IDENTIFIED"

13. DONE
```

Fig. 3.1.

```
FRED
 '
 '---ISA-->MAN
 '
 '---LOVES-->MARY
 '
 '---PAIDBY-->BRIAN
 '
 '---PAYS-->COLIN

ADAM
 '
 '---HAS-->POLICERECORD
 '
 '---FRIENDLY-->COLIN
 '
 '---FRIENDLY-->FRED
 '
 '---PAIDBY-->BRIAN

BRIAN
 '
 '---HAS-->POLICERECORD
 '
 '---PAYS-->ERIC
 '
 '---PAYS-->ADAM
 '
 '---PAYS-->FRED

COLIN
 '
 '---HAS-->POLICERECORD
 '
 '---FRIENDLY-->ADAM
 '
 '---PAIDBY-->FRED

DAVID
 '
 '---FRIENDLY-->ADAM

ERIC
 '
 '---PAIDBY-->BRIAN
 '
 '---ISA-->GARDENER
```

Fig 3.2

3.2.2 Program Specification

The experts were not told the purpose of the program, but for reasons of clarity here we will describe it. The problem statement as supplied to the student was as follows:

1) This option asks you to explore the notion of 'propagating' inferences through a database (see Units 3-4, pp 78-82).

2) Suppose that SOLO had the following descriptions stored in its database:

```
LIDDY
  '
  '---ISA-->BURGLAR
  '
  '---WORKSFOR-->MITCHELL


MITCHELL
  '
  '---ISA-->BIGLAWYER
  '
  '---WORKSFOR-->NIXON


NIXON
  '
  '---ISA-->PRESIDENT
```

3) Given the above descriptions, can you define a procedure called 'IMPLICATE' which makes the following inference: if someone is found to be guilty, then whoever that person works for is also guilty.

4) Here is how that procedure might work:

SOLO: IMPLICATE LIDDY

AHA! I'VE CAUGHT LIDDY, SO:

```
LIDDY
  '
  '---ISA-->BURGLAR
  '
  '---WORKSFOR-->MITCHELL
  '
  '---IS-->GUILTY
```

NOW, DOES LIDDY WORK FOR ANYONE?

AHA! I'VE CAUGHT MITCHELL, SO:

```
MITCHELL
  '
  '---ISA-->BIGLAWYER
  '
  '---WORKSFOR-->NIXON
  '
  '---IS-->GUILTY
```

NOW, DOES MITCHELL WORK FOR ANYONE?

AHA! I'VE CAUGHT NIXON, SO:

```
NIXON
  '
  '---ISA-->PRESIDENT
  '
  '---IS-->GUILTY
```

NOW, DOES NIXON WORK FOR ANYONE?

NO, SO I GUESS THAT'S ALL.

5) That's just a simple example. You may want to do something more elaborate - for instance, you may want to include extra CHECKs to see if other conditions are met before someone is IMPLICATED (e.g. is that person a known criminal? etc.). You should feel free to focus on some problem other than the Watergate scandal. As you are writing your IMPLICATE procedure, you should ask yourself: do people really reason this way? If not, can you devise a better model?

Discussing this problem statement, Kahney (1983) says:

> "The problem model which the author of the problem statement intended that the student should abstract.....is graphically represented in Fig. 3.3:

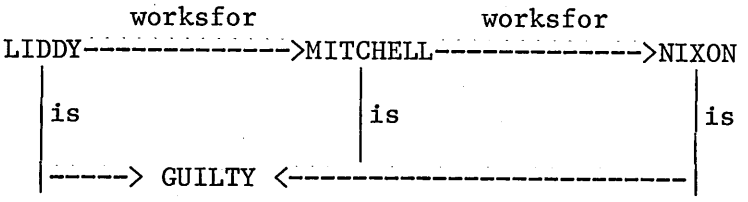


Fig. 3.3

> The problem was meant to be isomorphic with the INFECT problem...Note that the first line of the problem statement contains a pointer to the INFECT program, and it was expected that students would use that procedure as a model for writing their program for this problem." (Kahney 1983, p2-23).

The INFECT program referred to is as follows:

```
TO INFECT /X/

10 NOTE /X/ HAS FLU

20 CHECK /X/ KISSES ?S
  A INFECT *S ; EXIT
  B EXIT
```

And is expected to be used with a database like this:

```
JOHN---KISSES-->MARY

MARY---KISSES-->FRED

FRED---KISSES-->JOAN ...and so on.
```

The isomorphic IMPLICATE program referred to by Kahney is as follows. With it is reprinted the database suggested in the problem statement:

```
TO IMPLICATE /X/

10 NOTE /X/ IS GUILTY

20 CHECK /X/ WORKSFOR ?S
  A IMPLICATE *S ; EXIT
  B EXIT
```

```
LIDDY
  '
  '---ISA-->BURGLAR
  '
  '---WORKSFOR-->MITCHELL
```

```
MITCHELL
  '
  '---ISA-->BIGLAWYER
  '
  '---WORKSFOR-->NIXON
```

```
NIXON
  '
  '---ISA-->PRESIDENT
```

Suppose that the top level call to this program is

IMPLICATE LIDDY

Line 10 of IMPLICATE establishes the GUILT of LIDDY by NOTEing the fact as a normal SOLO database triple. Line 20 CHECKs in the database to see if LIDDY WORKSFOR anybody - in this case he does: he works for MITCHELL. A new recursive call to IMPLICATE is then set up by subline 20A, to

IMPLICATE MITCHELL. Since MITCHELL WORKSFOR NIXON, the latter too will be recursively IMPLICATEd, before the program finally halts at line 20B because NIXON is not recorded as working for anybody.

The student who wrote the program in fig. 3.1 appears to be attempting a more sophisticated inference than is accomplished by this simple isomorph to INFECT. Presumably he/she was motivated by paragraph 5 of the problem statement (page 3-20). In the student's version of IMPLICATE (fig. 3.1) line 1 carries a PRINT statement: PRINT /X/ "IS A CRIMINAL", in place of the NOTE instruction on line 10 of the isomorph. In the student's database (fig. 3.2) the relation FRIENDLY replaces the relation WORKSFOR, but any given person is allowed to be FRIENDLY with more than one other person:

```
ADAM
 '
 '---HAS-->POLICERECORD
 '
 '---FRIENDLY-->COLIN
 '
 '---FRIENDLY-->FRED
 '
 '---PAIDBY-->BRIAN
```

It is as though in the problem statement LIDDY (say) worked for several people at once. The student has correctly placed a FOR EACH loop (line 3 in fig. 3.1) around the basic test-and-recurse section, (lines 4, 5 and 6) which latter remains directly comparable with the INFECT program. If this section of the student's program works correctly, there should be a recursive call to IMPLICATE for each person with whom a given /X/ is recorded as being FRIENDLY. The program will thus propagate the "IS A CRIMINAL" inference through a tree-structure of database triples, rather than merely along a chain of them as does the isomorph. Furthermore, the student has repeated that whole exercise (lines 7-10) using another equivalent of WORKSFOR: this time it is PAYS, giving an alternative link along which the inference may be propagated.

3.2.3 Program Errors

The student's program will be discussed in terms of "segments", each being directly equatable with a SOLO line together with its sublines if any.

The first segment is the line numbered 2 in fig. 3.1. This line is a PRINT statement intended to signal the result of the inference by printing e.g. ADAM IS A CRIMINAL, COLIN IS A CRIMINAL. It contains a possible error, because some SOLO experts regard SOLO as a purely database-manipulative language. By the standards of these experts the result of the inference should be NOTEd into the database as in INFECT or its isomorph: a simple PRINT is not sufficient. The fact that this student uses PRINT is a sign of a possible misconception on his/her part: occasionally, (and presumably before their model of the SOLO machine is anywhere near complete) students will imagine that what appears on the terminal is somehow "in" the computer. They are then unable to grasp the difference between NOTE and PRINT. On the other hand, the problem statement only implies - it does not specify - that guilt is to be effected by the NOTEing of a triple. A correct IMPLICATE program will work in the same way whether its final results are NOTEd or merely printed on the terminal. This possible error is referred to as the PRINT COMMAND error.

The second segment comprises lines 3-6 of fig 3.1, and will be referred to as the FRIENDLY LOOP segment. Via a FOR EACH loop on line 3, it binds *A to, successively, all those database nodes whose triples match the pattern

/X/ FRIENDLY ?A

and then proceeds to apply the test-and-recurse operation (lines 4-6) to each of them. The first occasion on which the test succeeds will result in an eventual EXIT from the program (line 5).

The FRIENDLY LOOP segment is syntactically perfect. However, there is a possible case (which does not arise with this student's particular database) where it will not produce all of the desired results. For example, if:

```
ADAM
 '
 '---FRIENDLY-->COLIN
 '
 '---FRIENDLY-->FRED

COLIN
 '
 '---HAS-->POLICERECORD

FRED
 '
 '---HAS-->POLICERECORD
```

and if the top level call is IMPLICATE ADAM, only ADAM and COLIN, and not FRED, will be announced as criminals. This is because COLIN is the first matching CASE of

/X/--FRIENDLY-->?A. Therefore COLIN gets IMPLICATEd, after which the program EXITs. AURAC would have trapped this error, had it been available to the student, since it regards the code for the FRIENDLY LOOP as a cliche (see IMPLICATE, section 4.3). This error is the SCOPE error.

The third segment comprises lines 7-10 of fig. 3.1, and is identical to the FRIENDLY LOOP except that the FRIENDLY relation is replaced by PAYS. It is referred to as the PAYS LOOP, and is executed if no successful test-and-recurse occurs during execution of the earlier FRIENDLY LOOP. The same database query arises with the PAYS LOOP as with the FRIENDLY LOOP, although again no problems arise in this context with the student's database. Since every expert who detected the SCOPE error in this segment also detected the identical error in the FRIENDLY LOOP, we do not distinguish between the two examples of the same error.

There is also the question of the PAIDBY relations to be seen in the database. These may be no more than hangovers from an earlier attempt at IMPLICATE by the same student, or they may indicate a misunderstanding: students frequently have difficulty in grasping the strictly one-way nature of the relations within SOLO triples. But one can say no more in this individual case without consulting the student him/herself.

The fourth segment consists of line 11 of fig. 3.1. It is syntactic nonsense, and in fact could not have occurred had the student been using the MacSOLO/AURAC system. It seems likely that a slip of the typing finger entered the extraneous B, which then prevented SOLO from seeing the remainder of the line as an incorrect CHECK statement. As it stands, it would certainly cause a run-time error if executed (i.e. if in the database some /X/ was found not to have either of the relations FRIENDLY or PAYS), and SOLO would complain that "the procedure B is undefined". This is the UNDEFINED PROCEDURE error. If set to work on the program as it stands, AURAC would detect the error.

The fifth and final segment consists of line 12 of fig. 3.1. Analogously to the PRINT COMMAND segment, its effect is to print a quoted string on every call to IMPLICATE. It has no syntactic errors, but there will be as many PRINTed statements "THAT SEEMS TO BE THE WHOLE GROUP IDENTIFIED" as there are that /X/ "IS A CRIMINAL". This is referred to as the REPEATING STRING error, although of course it is only classifiable as an error because the subjects know what the words of the string mean. If the line had been

```
50 PRINT "FINISHED IMPLICATING" /X/
```

it is unlikely that anyone would have thought it an error.

Line 13 of fig. 3.1 is not strictly part of the program at all, but is merely a standard token signifying in some SOLO dialects the end of a program or listing.

There remain problems with the student's database. Considering lines 1-6 of the program, and assuming that ADAM is /X/ in this case, the person to be IMPLICATEd via the FRIENDLY LOOP will be COLIN, because in the database ADAM is FRIENDLY with him. On the recursive call, COLIN becomes /X/. And COLIN is described in the database as being FRIENDLY with ADAM. So a third recursive call is set up, with ADAM being /X/ again. This is a database (as opposed to a procedural) loop, and can in fact be entered via more than one route. For example, if the initial call is IMPLICATE FRED, he PAYS COLIN, so that COLIN will be IMPLICATEd via the PAYS LOOP. Whereupon the looping as just described will commence. Again, AURAC could trap this error and inform the student accordingly. This error is referred to as LOOP1. The various relations among the database nodes are shown diagrammatically in fig. 3.4:

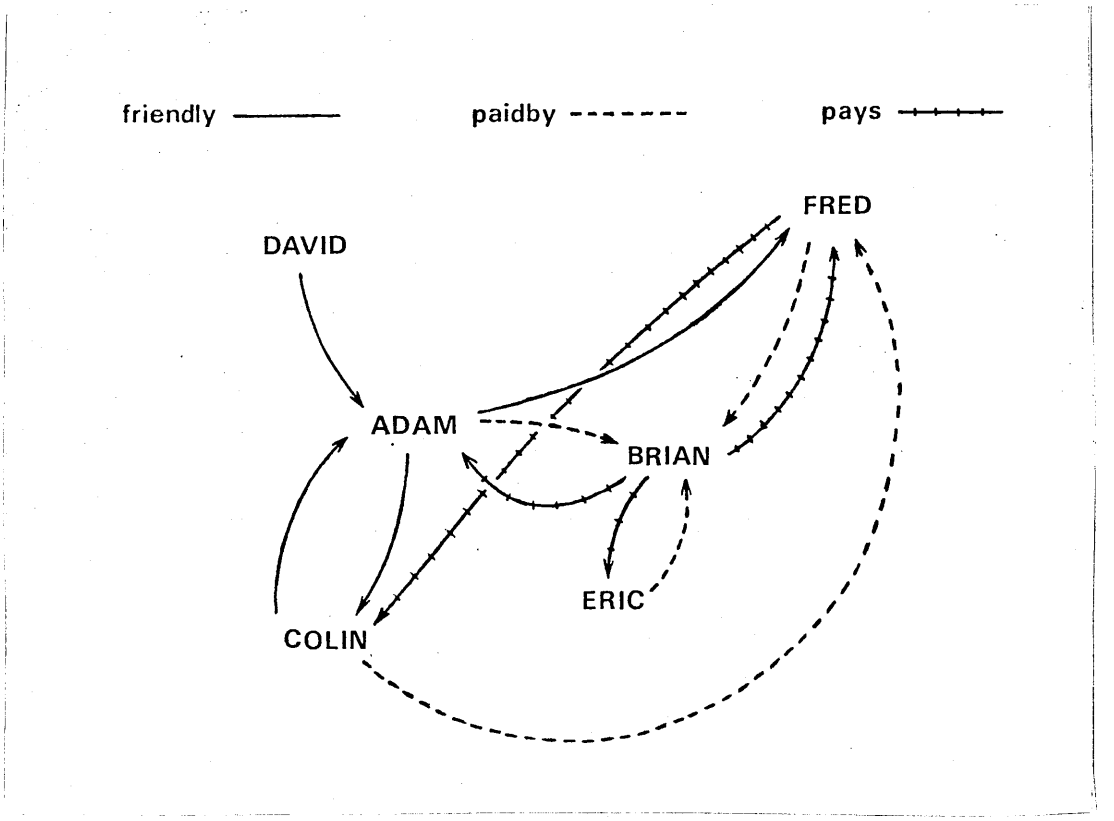


Fig. 3.4

3.2.4 The Protocols

The five experts were tested individually, and asked to think aloud during the process of debugging the program. The experimenter contributed nothing, other than an occasional encouragement to speak up. The subjects' comments were recorded on a Superscope CO-330 cassette tape recorder. Each was given the following instructions:

"In front of you there is a SOLO program written by a D303 student. You should imagine that the student has complained that the program does not work, and that he doesn't understand what is wrong with it. He asks you to tell him what is wrong with it.

I want you to debug the program for the student. Please

speak out loud while you debug the program. I want you to let me know everything you are thinking while you work on the problem. If you actually look at or read anything on the problem page, read it out loud. If you re-read anything, read it out loud. I want to know what information you are using at any particular point in time. So, if I see you looking at the problem, I will assume you are reading something, so always tell me what it is. Also try to tell me why you are doing whatever you do."

The resulting protocols were subsequently transcribed from the tapes, and analysed. Two distinct methods of analysing such data are possible (see Breuker, 1981): Where there is no prior hypothesis concerning what information may be found in the protocols, a bottom-up approach is preferred. This involves searching the protocol to see what information it actually does contain. The other method is top-down and is designed to confirm or disconfirm an existing theory or model, by comparing its predictions with the events recorded in the protocols.

In actual practice a combination of the two is necessary - a bottom-up pass to discover what information the protocols contain, and a series of top-down passes used to ensure that the results of the first are applied consistently, both within a single subject's protocol and also across subjects. The protocols were therefore analysed in the following ways:

(1) One protocol, now referred to as that from subject S2, was split into separate sentences/phrases, each

being given a LABEL to indicate the stated or inferred corresponding mental process on the part of the subject. Over the course of several passes through the protocol these labels were refined and reduced to a manageable number. After the analysis of S2 was completed, the remainder of the protocols (see Appendix E) were analysed using the same labels.

(2) In a similar manner, each protocol was then divided on a larger scale into sections representing different phases of the debugging process. These sections are often based upon the subject's analysis of SEGMENTS of code as specified above.

The significance of the labels used is as follows. Firstly, there are three "levels" of analysis, as predicted by our account:

READ is the label applied when the subject reads lines of code or sections of the database exactly as they were taken from the student (figs. 3.1 and 3.2). The assumption is that one of the things going on in the subject's mind during READ lines is that he/she is on the lookout for obvious (i.e. simple) syntactic errors. It is notable that all subjects (see for example lines 1-14 of the protocol from subject S2, fig. 3.5, below) first notice the UNDEFINED PROCEDURE error on line 11 of the buggy IMPLICATE program during a READ phase of analysis. In order to say that anything more than syntactic analysis is going on at such times, one would expect to find evidence on surrounding lines of other levels of analysis (see below).

ABSTRACT is the label given to protocol lines in which the subject replaces individual items of code - normally SOLO variables - with generalised values. An example is S2's use of "someone" on line 37 of the protocol below. During analysis of line 3 of fig. 3.1 (FOR EACH CASE OF /X/ FRIENDLY ?A"), The subject says: "FOR EACH CASE OF /X/ FRIENDLY with someone", whereas he could have said "FOR EACH CASE OF /X/ FRIENDLY QUESTION-MARK A" (syntactic analysis: READ) or "FOR EACH CASE OF ADAM FRIENDLY COLIN" (SIMULATION: see next). The ABSTRACT level of analysis corresponds to AURAC's cliche analysis.

SIMULATE. Here the mental process involves simulation of a "trace" or "step through" of sections of the program. The subject normally uses actual database items with which to replace items in the code. For example (from S2):

59.	So if you start off with Fred you IMPLICATE Colin and Colin will IMPLICATE...	SIMULATE
60.	Will have POLICERECORD.	SIMULATE
61.	Yeah.....	?
62.	He's FRIENDLY with ADAM, he'll IMPLICATE ADAM.	SIMULATE
63.	ADAM'll 'HAVE POLICERECORD'.	SIMULATE
64.	He's 'FRIENDLY with COLIN'.	SIMULATE
65.	You're going to get, er, by the looks of it something of a loop between Adam and Colin...	SPECIFY
66.	Because Adam'll always get picked up by being FRIENDLY with Colin and vice versa.	SIMULATE
67.	So you're gonna be in a recursive loop there between Adam and Colin	SPECIFY

Here, the subject assumes FRED as the initial value of /X/ in a call to IMPLICATE (line 59). Since FRED PAYS COLIN in the database, COLIN will be IMPLICATED via the PAYS LOOP. The subject then makes sure that COLIN has the necessary POLICERECORD (lines 60-61), and so visualises a new recursive call to IMPLICATE with COLIN as the new value for /X/. This call will in turn IMPLICATE ADAM (line 62). He then realises that because ADAM also has the POLICERECORD and is FRIENDLY with COLIN (lines 63-64), the next recursive call to IMPLICATE will have COLIN as the value of /X/ again: that in fact the program will loop between ADAM and COLIN (lines 65-67) due to a database error.

The SIMULATE level of analysis corresponds to AURAC's Data Flow analysis. However, it should be pointed out that what we have called "levels" of analysis are levels only in terms of the categories described in the first half of this chapter: one level is "higher" than another because the kind of errors it detects are liable to be made by a more experienced user. There is no sense in which, say, a syntactic error will necessarily be detected temporally sooner than a data flow error, if both appear in the same program. As will be seen, this is true of human debuggers. AURAC's three modules are all driven by the same production system, which works through the program code in run-time order. The order in which the three modules are applied to

any given section of code is arbitrary and trivially easy to change; but errors will be detected in essentially line-number order.

Early passes through the protocols revealed the need for six labels representing the results of analysis and the suggestions for curing any bugs found. The corresponding processes occurring in AURAC are indicated for each label.

CERTIFY. The subject states that some part of the program or database is error-free, perhaps after suggesting changes to it. This corresponds to AURAC's message which announces "no errors" for a particular procedure.

IDENTIFY occurs when the subject indicates the presence or probable presence of a bug in the program without stating its precise nature. In the protocols, such lines are (as one would expect) usually closely followed by SPECIFY lines (see next). Where they are not, they correspond to AURAC's "a possible error on ..." message.

SPECIFY is the label given when the subject attempts to state the precise nature of a bug. Within AURAC, as will be seen in chapter 4, detection of a higher-level syntactic or cliche error causes the creation of a stylised "frame" which describes the error in detail. In chapter 6 is mentioned how it is hoped that these frames will eventually permit AURAC to drive a much more sophisticated message generating system than it has at present. For the moment, and from the user's point of view, AURAC SPECIFYs bugs via a series of canned message segments. Data flow errors are not described via frames, but directly via the canned messages.

ADVISE. The subject talks about possible patches or repairs to sections of the code or the database without making specific recommendations. This corresponds to the generalised advice offered by AURAC when it detects, say, endless recursion errors: "You have a loop in your database via JOHN, MARY, FRED, JOAN, JOHN".

PATCH. The subject makes specific suggestions for curing a bug, as AURAC does for cliche errors in particular, e.g. "On line 20 you have written *X when perhaps you meant to write *P".

REFUSE. The subject states that no further analysis of a given section is possible without additional information from the original programmer.

And finally we needed five additional labels, not predicted for by our account of debugging behaviour:

CLASSIFY. Two subjects (S1 and S5) immediately recognise the program as belonging to a class with which they are familiar.

META. Some subjects (for example S5) are practised at giving verbal protocols and so are able to offer specific comments on their own mental processes - in particular to describe their own internal monitoring of their behaviour. In particular, experts sometimes make FALSE STARTS: following a train of thought about the program which they later realise to be wrong. AURAC is not capable of this kind of introspection, in which the experts appear to monitor their own progress at some meta-level of consciousness. However, it should be stressed that AURAC is not intended as a complete or accurate model of experts' behaviour: it is a debugging system whose methods are empirically based on those of experts. This same argument could, of course, be adduced to the discussion of the exact

order in which bugs are detected.

ERROR. Even experts make mistakes during their analyses, and such lines are labelled. Where the errors are also numbered (e.g. ERROR1, ERROR2), this signifies that the subject has made the same error more than once.

FALSE START. Sometimes a subject (see line 45 below) will follow a train of thought for a while but then realise that it is based on a misconception. He then generally retraces an earlier section of analysis (lines 46-50).

LOOPTEST. Subject S5 undertook an extended session of database analysis. This point will be returned to later.

On some 17% of the protocol lines, given the above framework, it is difficult to assign a precise label. The subject often appears to be making comments or partial comments regarding his own progress (see lines 10, 34, and 74 of S2). These lines should perhaps be labelled META lines, but sometimes they look more like temporary embarassment, or mere time-filling whilst some unspecified mental process takes place. Other lines (e.g. lines 76, 92 and 98 of S2) seem to be examples of the subject drawing logical conclusions from his progress so far. All such

lines have been given a question-mark as a label, since it is clearly safer to ignore them than to assign a possibly incorrect label.

The labels eventually arrived at give a good account of the subjects' behaviour, particularly when considered via the larger-scale SEGMENT divisions to be found in the protocols. On an individual level, line labels are often confirmed by their surrounding context rather than by precise correspondence to the label-descriptions given above. As an illustration, here is another short section from Subject S2's protocol:

19.	What does that line 4 mean?.....	IDENTIFY
20.	'B' CHECK *C PAYS.	READ (11)
21.	There must be a typing error there.	SPECIFY
22.	Or printing error there.	SPECIFY
23.	There must be something missed out there.	SPECIFY
24.	On line 4.	SPECIFY
25.	It looks like a CHECK statement, but it's got some unknown piece of whatsit before the CHECK command.. i.e. the 'B'.	SPECIFY
26.	And it's got no 'IF PRESENT','IF ABSENT'.	SPECIFY
27.	So it obviously didn't recognise it as a CHECK statement.	SPECIFY
28.	But I can't see what else it can be.	SPECIFY
29.	So that CHECK statement is wrong.	SPECIFY

One might feel, for example, that line 23 should have been labelled IDENTIFY, since the subject is making no attempt to describe the bug precisely. Or that line 28 should have been labelled REFUSE, since the subject is expressing a lack of sufficient information. But it is clear from the large number of surrounding SPECIFY lines that the subject is

engaged in trying to state precisely what the problem is at this point. Therefore any dubious lines within the immediate vicinity are labelled SPECIFY.

We present now the sample protocol referred to during the above explanation of the labels. It is from Subject S2. The remaining protocols can be found in Appendix E. Numbers in parentheses refer to program lines as shown in fig. 3.1.

This protocol, like those in Appendix E, is divided into sections. These help to clarify what the subject is doing in more general terms. For example, Subject S2 first READs through the entire program (lines 1-14), in the course of which he IDENTIFYs (lines 8-12) a probable error in the program. (This error can be seen on line 11 of fig. 3.1. The number given to this line by SOLO is 40, and this is referred to as segment 4). He then focusses his attention upon segment 4 (at line 19 of the protocol) and goes through a further process of SPECIFICATION (lines 20-29) and ADVICE (lines 30-31), leading to a final REFUSAL (line 32), before returning his attention to the start of the program at line 34 of his protocol.

Subject S2.

SYNTACTIC ANALYSIS & IDENTIFICATION (WHOLE PROGRAM):

1. TO IMPLICATE 'X' PRINT 'X' IS A CRIMINAL. READ (1) (2)

2.	FOR EACH CASE OF 'X FRIENDLY ?A' CHECK *A HAS POLICERECORD.	READ (3) (4)
3.	'IF PRESENT: IMPLICATE A'.	READ (5)
4.	'IF ABSENT: NEXTCASE'.	READ (6)
5.	'FOR EACH CASE OF 'X PAYS ?B' CHECK *B HAS POLICERECORD.	READ (7) (8)
6.	'IF PRESENT: IMPLICATE 'B'.	READ (9)
7.	'IF ABSENT: NEXTCASE'.	READ (10)
8.	'B CHECK'.	READ (11)
9.	'B CHECK *C PAYS'.	READ (11)
10.	Hmmm...	?
11.	'Not too sure what that last bit of code means.	IDENTIFY
12.	'B' CHECK *C PAYS.	READ (11)
13.	PRINT "THAT SEEMS TO BE THE WHOLE GROUP IDENTIFIED".	READ (12)
14.	DONE.	READ (13)

IDENTIFICATION & SPECIFICATION (4TH SEGMENT):

15.	'FRED ISA MAN', Da, Da, Da.	READ (DB)
16.	TO IMPLICATE 'X'.	READ (1)
17.	This program doesn't work.	?
18.	Why doesn't it work?	?
19.	What does that line 4 mean?.....	IDENTIFY
20.	'B' CHECK *C PAYS.	READ (11)
21.	There must be a typing error there.	SPECIFY
22.	Or printing error there.	SPECIFY
23.	There must be something missed out there.	SPECIFY
24.	On line 4.	SPECIFY
25.	It looks like a CHECK statement, but it's got some unknown piece of whatsit before the CHECK command.. i.e. the 'B'.	SPECIFY
26.	And it's got no 'IF PRESENT','IF ABSENT'.	SPECIFY
27.	So it obviously didn't recognise it as a CHECK statement.	SPECIFY
28.	But I can't see what else it can be.	SPECIFY
29.	So that CHECK statement is wrong.	SPECIFY

ADVICE & REFUSAL (4TH SEGMENT):

30.	It looks as if what the student wants to have is CHECK *B PAYS ?C.	ADVISE
31.	IF PRESENT: IMPLICATE *C or something of that form, but I'm not sure.....	ADVISE
32.	I would have to find out from the student what exactly he meant to do with... with... Er.. line 4, because it's certainly not obvious.	REFUSE
33.	Just looks a big mess.	IDENTIFY

READ & CERTIFICATION (1ST SEGMENT):

34.	What do the other preceding lines do?	META
35.	TO IMPLICATE 'X' PRINT 'X' IS A CRIMINAL.	READ (1) (2)
36.	O.K....	CERTIFY

ABSTRACTION & CERTIFICATION (2ND SEGMENT):

37.	FOR EACH CASE OF 'X' FRIENDLY with someone; CHECK that that someone HAS a POLICERECORD.	ABSTRACT (3) (4)
38.	Now if that someone does have a POLICERECORD.	ABSTRACT (3) (4)
39.	Yeah... that IMPLICATES that someone.	ABSTRACT (3) (4)
40.	That goes back there and takes the new [indecipherable].	ABSTRACT (5)
41.	O.K.	CERTIFY
42.	EXIT.	READ (5)
43.	IF ABSENT then CHECK with the NEXT CASE of 'X' being FRIENDLY with someone, so that should work O.K.	ABSTRACT (6)
44.	So then all people that 'X' is FRIENDLY that are going to be IMPLICATED.	ABSTRACT (5) ERROR1
45.	Oh.. if they....	META
46.	If they have a POLICERECORD.	ABSTRACT (5) (6)

ABSTRACTION & CERTIFICATION (2ND & 3RD SEGMENTS):

47.	O.K., FOR EACH CASE OF 'X PAYS B'.....	READ (7)
48.	Oh I see, [indecipherable].	?
49.	FOR EACH CASE OF 'X' PAYS ?B CHECK *B HAS POLICERECORD.	READ (7) (8)
50.	Uh-huh.	CERTIFY
51.	IF PRESENT then IMPLICATE B also.	READ (9)
52.	EXIT, NEXTCASE.	READ (9) (10)
53.	So you're implicating all people that... have POLICERECORDS and are either FRIENDLY with or are PAIDBY 'X', according to statements 2 & 3., commands 2 & 3.	ABSTRACT (3) (4) (5) (6) (7) (8) (9) (10) ERROR1

REFUSAL (4TH SEGMENT):

54.	Now 4.	META
55.	4 just seems to be a complete waste of time.	IDENTIFY
56.	4 doesn't fit in anywhere.	IDENTIFY
57.	It's hard to....	?

58. It's really hard to debug this without knowing what the person actually wanted to do beyond implicating all the people with records who are FRIENDLY and are PAIDBY 'X'.	REFUSE ERROR1

SIMULATION, SPECIFICATION & ADVICE (2ND SEGMENT):

59. So if you start off with Fred you IMPLICATE Colin and Colin will IMPLICATE...	SIMULATE
60. Will have POLICERECORD.	SIMULATE
61. Yeah.....	?
62. He's FRIENDLY with ADAM, he'll IMPLICATE ADAM.	SIMULATE
63. ADAM'll 'HAVE POLICERECORD'.	SIMULATE
64. He's 'FRIENDLY with COLIN'.	SIMULATE
65. You're going to get, er, by the looks of it something of a loop between Adam and Colin...	SPECIFY
66. Because Adam'll always get picked up by being FRIENDLY with Colin and vice versa.	SIMULATE
67. So you're gonna be in a recursive loop there between Adam and Colin	SPECIFY
68. So you probably want in your database to make... Ermm.. the relationships two-way so Adam FRIENDLY with Colin necessarily implies Colin FRIENDLY with Adam without having to explicitly state it in the database.	ADVISE ERROR2
69. As you have at the moment.	ADVISE
70. 'Cos presumably that's gonna lead you to some.... to some loops somewhere, at some point.....	IDENTIFY

IDENTIFICATION, SPECIFICATION & ADVICE (DATABASE RELATIONS):

71. Now you've also got the problem of 'PAIDBY'.	IDENTIFY
72. But you only want to [indecipherable].	?
73. Eric, Adam and Fred.....PAYS Colin	ABSTRACT (DB)
74. Adam.	?
75. After Adam it's gonna go back to Colin, back to Fred.	ABSTRACT (DB)
76. Yes.	?
77. Fred also is gonna be a recursive loop, err...	SPECIFY
78. Because you've got a link between Fred, Colin and Adam which takes you straight back to Fred.	ABSTRACT (DB) ERROR3
79. And you're just gonna keep cycling round those Fred, Colin, Adam loops.	SPECIFY

80. So you wanna get rid of Adam FRIENDLY Colin, Adam FRIENDLY Fred, and just make all your relationships 2-way.	ADVISE ERROR2
81. That will now IMPLICATE....	CERTIFY
82. THAT SEEMS TO BE THE WHOLE GROUP IDENTIFIED.	READ (12)
83. Well, it'll certainly identify Adam, Fred and Colin.	CERTIFY ERROR3
84. Errm.....	?
85. I don't know whether David would be...	IDENTIFY
86. Eric PAIDBY Brian	READ (DB)
87. Well, the whole group wouldn't be IMPLICATED because Eric would be picked up by being paid by Brian...	SPECIFY
88. So how would Brian be picked up?	IDENTIFY
89. Brian would not be picked up because the only relationship he has with anybody is PAIDBY with Fred and PAIDBY isn't recognised as PAYS so Brian would never be picked up because 'PAIDBY' & 'PAYS' do not equate.	SIMULATE
90. The program wouldn't recognise that equation, so Brian would never be picked up which would imply that Eric would never be picked up.	SIMULATE
91. So you cannot equate the relation 'PAIDBY' and 'PAYS'.	?
92. That has to be changed.	ADVISE
93. Ermm...	?
94. So if you do equate them you'll have to specify it in the program somewhere.	ADVISE
95. Errrrr...	?
96. What else have we got?	?
97. David FRIENDLY with Adam.	READ (DB)
98. Adam would be picked up but whether David would be picked up, being the other end of the relationship is doubtful.	? ERROR4
99. So it's doubtful whether Eric, David or Brian would ever be picked up.	IDENTIFY
100. The only ones that would be picked up would be picked up in recursive loops which would be Fred, Adam or Colin.	SPECIFY ERROR5
101. Errmmm...	?

SPECIFICATION & ADVICE (4TH SEGMENT):

102. CHECK....	READ (11)
103. I'm looking again at this statement 4.	?
104. 'B' CHECK C* PAYS.	READ (11)
105. Now I can't see where C comes from, I don't see a question mark, anywhere, above it.	IDENTIFY

106.	So it looks as if....	?
107.	It looks as if it's a complete typographical error.	SPECIFY
108.	Looks as if it should be "CHECK *B PAYS ?C"	ADVISE
109.	Then "IF PRESENT, IF ABSENT", but like I say they aren't even in there.	ADVISE
110.	Err..	?
111.	Given that you're gonna IMPLICATE 'B' I don't see what the point is in having that, anyway, because it's gonna come down later, when it comes to statement 3, "FOR EACH CASE OF 'X' PAYS ?B".	SIMULATE
112.	The last *B will then be 'X'.	SIMULATE
113.	So I can't really see why 4 has to be in there at all.	ADVISE

REFUSAL (4TH SEGMENT):

114.	I'd need to check with what the students were thinking they were doing to find out what that whole line means.	REFUSE

SUMMING-UP:

115. So, I think those were about all the errors that I can identify., based on what my assumptions are on what the program is meant to do.

Fig. 3.5

3.2.5 Results, And Comparison Of Them With The Methods Of AURAC

Protocols were used because they show not only which bugs experts identify but also the main processes - IDENTIFICATION, SPECIFICATION, SIMULATION, PATCH, etc. - they use. Whilst it is apparent that the same processes are used by almost all experts, there are some exceptions - e.g., some experts never go beyond the ABSTRACT level in

analysing program behaviour. Nor do experts uniformly analyse programs in the same fashion. Some READ the entire program, then ABSTRACT. Some both READ and ABSTRACT first, then ABSTRACT and SIMULATE on a second pass through the program, and so on. However, all subjects work through the program in approximately SOLO line-number order. Subject S1 most displays the approach chosen for AURAC: apart from some initial inspection of the database, his method is to deal fully with each error before moving on to the next segment of the program.

The protocol from Subject 3 shows the main drawback of this experimental method: the information available from the protocol is heavily dependent upon the subject's willingness to think aloud. By contrast, the protocol from Subject 5 gives much more information in the way of META comments. Whilst these are fascinating from the point of view of future research, they are not immediately relevant to the current version of AURAC.

The first thing to notice is the wide variation in the order in which the experts detected the various errors. AURAC, of course, being a logical machine, would always detect them in the same order, if asked to debug the above program several times. Its sequence is the same as that in which the program lines are executed. However, it is apparent that the precise order of analysis or of error detection is of no particular importance for a debugging system based upon human expert behaviour. The order in which AURAC itself would find the faults in the program could be changed by completely trivial modifications to its production system.

The second interesting aspect of the results is that not all experts pick up all the errors, and that some report errors (LOOP2) which do not actually exist. There is also some disagreement amongst the experts as to whether or not the PRINT COMMAND is an error. Of the four definite errors found (UNDEFINED PROCEDURE, LOOP1, SCOPE and REPEATING STRING, as described in section 3.2.3) AURAC would detect the first three; in the case of SCOPE it would detect both occurrences of the error. REPEATING STRING is an error which is in any case self-evident at run-time. It also worth noting from the protocols that most of the experts at first completely ignore any PRINT statements in the procedure and mention them, if at all, only after they have

analysed the more complex parts of the code. AURAC does not currently concern itself with PRINT statements.

Thirdly, the experts themselves make a number of mistakes during analysis. These are broadly of two types: confusion between database items and code items (e.g. PAYS vs PAIDBY, lines 47-53 of S2); and incorrect statements about the SOLO machine (e.g. lines 149-150 of S3). As mentioned elsewhere, AURAC is deliberately designed to be very conservative in areas where mistakes of analysis could occur (e.g. during cliche recognition or algorithm matching); it would rather miss a genuine error than announce a non-existent one.

Fourthly, it is clear that the experts sometimes use their own world knowledge (e.g. that the relation FRIENDLY implies knowing - being able to recognise - another person, or that PAYS and PAIDBY are inverse forms of each other) to help them in their search for an abstracted view of the program's operations. AURAC cannot do this, but it is noteworthy that those experts who did use this kind of world knowledge were no more successful at debugging the program than those who did not. The precise usefulness of such world knowledge, if any, in debugging is not revealed by this experiment.

A related point is that three of the subjects (S1, S3 and S5) immediately CLASSIFY the program as belonging to a set of programs with which they are familiar. One of them, S5, spends a long time analysing the database for possible loops (hence the LOOPTEST label - lines 141-350 of S5's protocol). He generalises the database relationships in order to do so, and says in effect "I can 'GET' from A to B if A is either FRIENDLY with B or PAYS B, and if B HAS a POLICERECORD." (lines 252-350). He gradually builds a stylised network representation (on paper) of the GET links so found. Since (apart from those forming LOOP1) no node appears twice on any path through the network, he concludes that no further loops are possible. This is a very specialised analysis, specific to this kind of problem - i.e. the subject was using knowledge selected in response to his CLASSIFICATION of the program. He actually says (line 140) "This I wouldn't do...for detecting the student's bug.".

An early version of AURAC attempted some database analysis. It was capable of detecting three of the database structures most commonly found in SOLO: chains, trees and tables. The process proved to be time-consuming compared with its usefulness. But on the strength of the experience it can be suggested that a module to emulate S5's database analysis might work as follows:

1) Recognise the program as a member of a known class via (a) cliche recognition, or (b) interaction with the

user concerning his/her "project".

2) Predict a suitable database structure.

3) Generate "database fragments" corresponding to the program code, e.g.

```
'                              '
'---HAS-->POLICERECORD         '---HAS-->POLICERECORD
'                              '
'---FRIENDLY---?A              '---PAYS-->?B
```

4) Map these fragments onto each database node in turn, so as to create a deeper representation of any actual database structures.

5) Compare these with the predicted structures so as to detect differences - i.e. errors.

This again sounds as though it might double the computational overheads involved. In the context of a practical debugger it is undesirable to have a large ratio between the time taken to complete an analysis and the results produced by it. Currently, AURAC is expected to be called when a program has failed, If the failure was due to a database loop, AURAC will discover the fact during its simulated re-run of the failed call. This is an efficient way of achieving the desired result if the database has only one loop in it. If, as S5 suspected, it has several, his method might be preferable on efficiency grounds. But, endless recursion is not at all a common error amongst SOLO users, and we believe that multiple database loops are even less common. Therefore, our position concerning S5's unusually meticulous debugging behaviour is that whilst it

is feasible to add a comparable module to AURAC, to do so would not improve AURAC's performance sufficiently to warrant the extra delay suffered by its users.

This subject's additional comment on line 141: "I might do it (the above analysis) when I'm explaining it to the student" raises the large question of how an auto-debugger's results are best presented to the user. We shall return to this question in chapter 6.

Only one of the subjects (S3, lines 115-117 and lines 157-160) goes beyond the three levels of analysis described here to consider whether or not the program conforms to his own model of IMPLICATE. He compares it to something like an algorithm: "There's no increment, or decision-making" (line 160) in a way which suggests that AURAC's step-by-step algorithm matching may be analogous to his own. But of course the evidence here is far too slim for us to claim a confirmation. What is supported is the status of AURAC's algorithm matcher. The matcher is a (useful) by-product of the process of data flow analysis, but is not implied by the criteria on which AURAC is based.

Apart from these two sections from the protocols of S5 and S3, where knowledge specific to the program's CLASS is employed, no subject explicitly uses any level of analysis other then READ, ABSTRACT and SIMULATE. If these prove insufficient to explain a bug, a REFUSAL generally occurs (e.g. lines 20-32 of S1). As described above, a REFUSAL implies a need for extra information from the programmer, rather than a further level of analysis of the existing data.

There is also the question of the kinds of ADVICE and PATCHes suggested by the experts. All of them express puzzlement over the UNDEFINED PROCEDURE error, and several note that in the SOLO dialects they are accustomed to the error could not occur, as it could not in MacSOLO. In general, their suggestion is to discard the line altogether. (It is also worth mentioning that one subject, S5, wished aloud that he had MacSOLO - not AURAC itself - available so as to be able to step through the sample program, rather than having to do a trace on paper).

The error LOOP1 causes several experts to suggest alterations to the database. Subjects 2 and 5 advise changing some relations to their own inverses so that the loop can no longer occur. AURAC does not suggest patches of this sort, but instead points out to the user the fact of the loop, together with the names (FRED, COLIN, ADAM, COLIN in the earlier example) of the database nodes comprising it. Exactly what the user does about it is left to his/her discretion; as can be seen from S2's protocol in particular, having explicit inverse relations in the database can sometimes cause more confusion that it cures (lines 47-53).

The SCOPE error is caused by substitution of EXIT for NEXTCASE as a control-statement. As already mentioned, it is in AURAC's terms a cliche error; which means that AURAC will suggest a direct patch, to replace the incorrect word with the correct one. Only two subjects, S1 and S5, offer a patch for this error, and both of them suggest replacing EXIT with NEXTCASE on lines 5 and 9 of the program.

Finally, there is a marked consistency amongst the experts in terms of their levels of analysis. As already metioned, it is almost invariably during a line labelled READ that the human expert will first notice the syntactic error UNDEFINED PROCEDURE. Lines marked READ are directly comparable to the "skimming" phase of AURAC's analysis, and detect the same kind of errors: Higher Level syntactic. The first 14 lines of the above sample protocol are an excellent example of this phase of analysis. A glance at Appendix E will show that Subject 5 actually says "I want to skim through it" during a META comment at the beginning of his protocol (line 5).

Lines marked ABSTRACT occur when the expert puts some kind of general value (e.g. "someone", "that someone") in place of the user's SOLO variables. The assumption here is that the expert is trying to gain some more abstracted view of the operations of segments of code. The token "someone" resembles a variable to which is assigned some value presumed to be in the program at that moment; and "that someone" similarly resembles retrieval of that value. This is very akin to the detailed matching process in AURAC's cliche analysis: in an ideal case cliche analysis would offer the desired degree of abstraction. There is no evidence as to precisely what would comprise the abstraction the expert seeks, but we offer cliches as suitable material.

As the protocols show, some experts do not go beyond the ABSTRACTION stage in their debugging. Those who do, SIMULATE the program's behaviour, often replacing SOLO variables with suitable but otherwise arbitrary items from the given database, and then following the progress of those data-items during a mental execution of segments or of all of the program. As mentioned above, all of AURAC's analyses, including Data Flow, are effected via a simulated execution of the buggy program. It is clearly during SIMULATION that any missing database triples will show up - errors which AURAC traps during its Higher Level syntactic analysis. So, instead of selecting items from the database, AURAC assumes the correctness of the input data as supplied by the user during an actual (failed) run of the program, and does its tracing - i.e. its Data Flow analysis - on that basis. Approximately 30% of all the protocol lines are labelled READ, ABSTRACT or SIMULATE, and thus correspond to one or other of the analyses carried out by AURAC.

Thus, AURAC's three levels of analysis are clearly seen in comparable human behaviour. It can do what they most frequently do, and often they do not use any further levels. With the exception of REPEATING STRING, it finds all the genuine errors in the program and does not find any non-existent errors (in fact, a relatively minor modification involving an additional type of "endless

recursion" would enable it to find REPEATING STRING). But it is evident also that beyond these three levels of analysis, when a human expert recognises a program as belonging to a known class, he/she is able to apply large amounts of knowledge ranging from the general comments of S3 to the database analysis of S5. (S1 makes no use of the knowledge at all). From the corresponding protocols, one can see that this knowledge includes knowledge of the kinds of subprocesses to be expected in the program - although there is not sufficient data to predict whether the subprocesses more resemble cliches or algorithms - and knowledge of when careful database investigations may be worthwhile. Exactly what is going on here, and how it could usefully be implemented in an auto-debugger, is well worth further research.

It is also reasonable to claim that the current version of AURAC (discounting its algorithm matcher) is an implementation of the SOLO experts' general debugging skills; that is, of the skills they apply when they have no knowledge of the buggy program's specific purpose. It is clear that these skills can generate substantial amounts of debugging information - information which could not be provided by static demons. Knowledge of a program's purpose can be acquired (in the real world) either by asking the programmer or by CLASSIFYing the program, so that S5's long

section of database analysis and S3's short section of algorithmic analysis are comparable with REFUSE lines, on which subjects express a need to consult the programmer over some specific point.

CHAPTER 4

AURAC

This chapter discusses the operations of AURAC in greater detail. It is assumed that a user will call upon AURAC when a program has failed at run-time, and as already mentioned it employs a production system to re-execute the faulty code. The second execution follows the previous run-time sequence and uses the same values for its top-level parameters as were supplied by the user. As it proceeds, the user's code is analysed for Higher Level syntactic errors by the production system, and is also passed for analysis to the Cliche recogniser and Data Flow modules. But before discussing these three modules in detail it is important to describe the means by which errors found by the first two are represented and related within AURAC.

4.1 ERROR FRAMES

Any error found at the syntactic or cliche level is recorded at the moment of its discovery in an error "frame": a standardised data structure having "slots" for various items of information concerning the error. Here is a very simple SOLO routine containing a single error. The CHECK on line 10 fails to find in the database a suitable matching triple, and so the variable *THOUGHT remains unbound. MacSOLO itself sees nothing amiss until line 20, where the unbound variable reveals itself as an error in the NOTE instruction:

```
TO THING

10 CHECK FIDO THINKS ?THOUGHT
  A If Present: CONTINUE
  B If Absent : CONTINUE

20 NOTE TONY HAS *THOUGHT
```

MacSOLO would announce the run-time error, and execution would halt:

"Procedure execution halted in THING because of:
Unbound variable - *THOUGHT has no value on line 20 of THING."

Suppose we were now to call AURAC to debug this procedure. It would first ask for the name of the project concerned (see section 4.7) to which question a carriage-return, NONE or ANY is a suitable reply in this case. There would then be printed the following messages:

```
"Working on THING...1

 An error on line 10 of THING at level 1:

   Your code (CHECK FIDO THINKS ?THOUGHT) is activated
   and that CHECK-triple does not exist in your database.
   So there is also...

 An error on line 20 of THING at level 1:

   Your code (NOTE TONY HAS *THOUGHT) is activated
   and that contains an unbound variable.
   This caused your run-time error.


 Analyses available for

 THING

 type INFORM followed by any of these."
```

The above are the messages from AURAC's Higher Level syntactic module. Notice how the two errors (the failure of the binding and the subsequent failed reference) are chained. This shows in the "So there is also..." segment between the two messages. Use of the INFORM instruction produces a "description" of the THING's execution, including any cliche or data flow errors:

```
"Your procedure THING:

 Lines 10 of THING, 20 of THING
   seem to be intended to carry out an action
   if the triple FIDO---THINKS-->?THOUGHT is present,
   but on line 10B of THING you have written CONTINUE
   where perhaps you meant to write EXIT.

 Level 1> Line 10: CHECK-triple does not exist in your
                   database.
 Level 1> Line 20: contains an unbound variable.
                   RUN-TIME ERROR <<<
```

```
...and also the triple NOTEd on line 20 of THING
   is never CHECKed.
...and also the triple NOTEd on line 20 of THING
   is never FORGOTten."
```

Correct coding of this line would indeed require an EXIT on line 10B so as to avoid, in the absence of the any triple matching the pattern FIDO THINKS ?THOUGHT, precisely the run-time error which has occurred. The next two notes repeat in abbreviated form the information given prior to our use of INFORM; and the final two messages are from the data flow analyser, indicating that THING leaves data behind it in the database. Such added data is frequently not acceptable from a correct program (see section 4.5).

The error frames generated during analysis of THING are as follows:

ERROR4

```
EFFECTS:     (RTE)
CAUSE:       ERROR3
TYPE:        "contains an unbound variable."
UNREACHED:   NIL
EVALUATED:   (NOTE TONY HAS *THOUGHT)
CODE:        (NOTE TONY HAS *THOUGHT)
LINE:        (20)
PROCEDURE:   THING
RECURSION:   NIL
ALTPROCS:    NIL
ALTNODES:    NIL
LEVEL:       1
CLICHE:      NIL
ALTLINES:    NIL
WORD:        NIL
SYMBOL:      NIL
ANNOUNCE:    T
```

```
ERROR3

   EFFECTS:     (ERROR4)
   CAUSE:       NIL
   TYPE:        "CHECK triple does not exist in your database."
   UNREACHED:   NIL
   EVALUATED:   (CHECK FIDO THINKS ?THOUGHT)
   CODE:        (CHECK FIDO THINKS ?THOUGHT)
   LINE:        (10)
   PROCEDURE:   THING
   RECURSION:   NIL
   ALTPROCS:    NIL
   ALTNODES:    NIL
   LEVEL:       1
   CLICHE:      NIL
   ALTLINES:    NIL
   WORD:        NIL
   SYMBOL:      NIL
   ANNOUNCE:    T

ERROR2

   EFFECTS:     NIL
   CAUSE:       NIL
   TYPE:        NIL
   UNREACHED:   NIL
   EVALUATED:   NIL
   CODE:        NIL
   LINE:        (10 B)
   PROCEDURE:   THING
   RECURSION:   NIL
   ALTPROCS:    NIL
   ALTNODES:    NIL
   LEVEL:       1
   CLICHE:      FETCH-DO-M_1
   ALTLINES:    ((10 THING) (20 THING))
   WORD:        CONTINUE
   SYMBOL:      EXIT
   ANNOUNCE:    T


ERROR1

   EFFECTS:     NIL
   CAUSE:       NIL
   TYPE:        NIL
   UNREACHED:   NIL
   EVALUATED:   NIL
   CODE:        NIL
   LINE:        (10 B)
   PROCEDURE:   THING
   RECURSION:   NIL
```

```
ALTPROCS:   NIL
ALTNODES:   NIL
LEVEL:      1
CLICHE:     UPDATE-M_1
ALTLINES:   NIL
WORD:       CONTINUE
SYMBOL:     EXIT
ANNOUNCE:   NIL
```

ERROR1 is discarded by the system - it represents a multi-line cliche of which only one of the library lines could be matched against the code of THING. ERROR2 represents the cliche found and described above, and ERROR3 and ERROR4 together represent the chained unbound-variable errors.

The meanings of the various slots in these frames are as follows:

CAUSE/EFFECTS. When an attempted variable binding fails, that fact is stored temporarily in the production system's working memory, and an error frame is created. Later reference to the same (unbound) variable causes another error frame to be created for this second error, and the data in the working memory enables the two frames to be related in a cause-and-effect chain.

TYPE. There are thirteen "types" of error, as described under Module 1, below. Each "type" is recorded in this slot as an error message segment, such as "CHECK triple does not exist in your database."

UNREACHED. If a control-statement error occurs such that the user's code contains unreachable lines or subroutines, their line-numbers and procedure names are recorded here.

EVALUATED/CODE. These slots are merely for the sake of completeness in the subsequent printouts; so that the message-printers can tell the user how SOLO evaluated any particular line of code - where that line contains variables and where in the light of any earlier errors those variables can be evaluated. For example the CODE slot might contain CHECK FIDO ISA *WHAT, whilst the EVALUATED slot contained CHECK FIDO ISA DOG.

LINE/PROCEDURE. To identify the line of user code to which the current error frame refers.

RECURSION/ALTPROCS/ALTNODES. These slots hold the information necessary to explain endless recursion. The error TYPE in each case (third slot in the frame) is "uses up more than twenty LEVELS.". The RECURSION slot will contain one of five "sub-types", each of them again being an error message segment:

1. "Your chain of database nodes is too long."
2. "Your series of subroutine calls is too long."
3. "self-recursion."
4. "data loop."
5. "procedural loop."

The production system calls itself recursively to handle subroutine calls, and maintains its own push-down stack of useful (to it) data. Amongst this data are records of each subroutine call and of the latter's arguments. Thus, when the inbuilt recursion limit is reached, distinctions can be made between the five sub-types:

1) for example, a recursive SOLO procedure such as INFECT which was given a different argument at each recursive level.

2) disregarding the arguments, where the number of sub-routines, sub-sub-routines and so on exceeds the limit of twenty.

3) where the name of the subroutine and its arguments are identically repeated at sequential levels. This is normally a procedural error, but in the unique case where (e.g.) INFECT is applied to a database consisting only of JOHN---KISSES--->JOHN, AURAC does not distinguish between the two.

4) where the name of the subroutine and its arguments

are identically repeated after a sequence of intervening calls to the same subroutine (as when INFECT is applied to a database such as:

JOHN---KISSES-->MARY MARY---KISSES-->FRED
FRED---KISSES-->JOAN JOAN---KISSES-->JOHN

5) where a sequence of subroutines and their arguments are identically repeated.

Thus, sub-types 1 and 4 will be accompanied by additional information in the ALTNODES slot specifying the database nodes concerned; and sub-types 2 and 5 will carry additional information in the ALTPROCS slot specifying the offending procedure-names.

LEVEL. Recursive or subroutine level at which the current error occurred (derived from the production system's working memory).

CLICHE. The name of the cliche concerned, if the current error is a cliche error. This name is a purely internal marker, and is not told to the user.

ALTLINES. Line-numbers and procedure names of all lines comprising the current cliche, including the current line. These are derived by (rather expensive) heuristics which operate when all possible matches and near-matches between the code line and the cliche library have been found (see section 4.4).

WORD/SYMBOL. Cliche errors are atomic. That is to say, AURAC suggests a patch involving only a single "word" of code. The WORD slot holds the user's version; the SYMBOL slot holds the correct version inferred by AURAC.

ANNOUNCE. The Boolean value of the datum in this slot is adjusted according to whether or not the current error frame needs to be passed to the message printers. Where only some of the lines of a multi-line cliche are found, any error-frames associated with those discovered lines will be redundant, and so will not need to be "announced".

An important feature of this style of representation is that error frames created during code analysis may later be modified, or rejected altogether, during analysis of the reminder of the code.

We shall now discuss the various modules of AURAC in more detail.

4.2 MODULE 1: HIGHER-LEVEL SYNTACTIC ANALYSER

Higher level syntactic analysis is accomplished via a production system which largely replaces the normal, run-time, MacSOLO interpreter. (However, it is important to stress that no implicit claims are made concerning the production system: during implementation, it was convenient formalism, and that is all). The run-time interpreter is called upon, for example, to discover which exit should be taken from a conditional form or to evaluate user-supplied variables. The analyser takes the program code line by line, following the normal run-time sequence where subroutines etc. occur.

The only exception to the normal run-time sequence of execution occurs when the program contains a coded loop. In SOLO, the looping construct is FOR EACH CASE OF, which as its name implies does no more than retrieve from the database a set of cases each of which matches a given wildcard-triple pattern. If a conditional test is required within the loop, this is achieved by nesting the CHECK form within the FOR EACH line. Here are two examples:

```
10 FOR EACH CASE OF FIDO LIKES ?D
  A NOTE *D LIKES FIDO
```

```
10 FOR EACH CASE OF FIDO LIKES ?D
  A CHECK *D ISA DOG
  AA If Present: NOTE *D LIKES FIDO ; NEXTCASE
  AB IF Absent : NEXTCASE
```

Given a suitable database, the first loop will NOTE the inverse relationship for everything which FIDO LIKES; the second will first CHECK to ensure that that something is another dog. Notice the control-statement NEXTCASE, which is only appropriate when CHECK is used within FOR EACH.

FOR EACH lines may be nested up to five deep:

```
10 FOR EACH CASE OF A B ?C
  A FOR EACH CASE OF *C D ?E
  AA FOR EACH CASE OF *E F ?G ...
```

and as before the innermost subline may be a simple SOLO instruction or a conditional test.

Assuming that a FOR EACH loop is correctly constructed in its basic syntax (and MacSOLO will not allow it to be otherwise), the only error which can arise is where one of the triples referrerd to by the code does not actually exist in the database. In analysing a FOR EACH nest, AURAC checks each level in turn to ensure that at least one suitable triple exists. The levels are considered sequentially, in case - as in the last example here - the triples for subsequent levels cannot be specified until earlier levels

have been evaluated. If AURAC finds at least one "case" at each level, it assumes that the nest overall is correct. If the innermost subline is found to be a simple one-line instruction, it is then executed in the context of just one case: this saves on processing time.

However, this short cut is not taken if the innermost subline carries a CHECK instruction: the result "CHECK always succeeds" or "CHECK always fails" despite the conditional test having been applied to the possibly large number of cases generated by a FOR EACH nest is quite likely to be the symptom of an error.

The production system has twelve rules in its production memory - i.e. twelve rules whereby it analyses the SOLO code presented to it, and these rules are briefly described here. Suppose that the line of SOLO code to be analysed consists of a multiple FOR EACH statement within which is nested a CHECK statement, whose A subline carries a call to a user defined function FOO:

```
FOR EACH CASE OF FIDO LIKES ?P
 FOR EACH CASE OF *P BROTHER ?B
  A CHECK *B ISA DOG
  AA FOO *B ; NEXTCASE
  AB ; NEXTCASE
```

This line is initially held in working memory as a Lisp list, each separate statement or subline being a disctinct

element of the list. The first job AURAC must do is ensure that the triples (FIDO LIKES ?P and *P BROTHER ?B) - out of which the tree of possible cases is constructed - do actually exist in the database. So the first production rule says:

1) IF there is a FOR EACH statement in working memory which cannot produce any cases

   THEN create the appropriate error frame;

   deposit FOR into working memory to signify to later rules that a FOR nest has been detected;

   deposit into working memory the *-variable (*P in this case) whose binding has failed.

The last effect of this rule will allow "chaining" of the error if *P is referred to by subsequent user code. If the rule fires (i.e. if the triple was unable to generate any cases) it also resets the production system interpreter so that the same rule is tried again on the next segment of user code. The next segment of user code - the next instruction on the line - is reached by "rotating" the copy of the code held in working memory: the first line (the first member of the list) is moved to its end, so that the head of the list is now the second FOR EACH instruction in the above example.

Rule 2 is the rule which actually generates the cases of any FOR EACH statement. It uses the normal functions in the MacSOLO interpreter to effect evaluation of FOR EACH statements, and so generates all the possible cases of, say, FIDO LIKES ?P. The possible bindings of *P are then deposited in working memory (this is to treat a FOR EACH loop as a generator of aggregate, rather than sequential, results as in Waters (1979).). On the second or any subsequent passes the first of the cases generated during the previous pass is used if necessary for full evaluation of the FOR EACH triple. In the above example, the first value of *P is required when evaluating the triple *P BROTHER ?B.

Production rule 2 says:

2) IF the head of the line in working memory is a FOR EACH
   THEN find its cases and add them to working memory;
        add the token FOR to working memory;
        rotate the line.

The FOR token signifies to other rules that FOR EACH evaluation is in progress, so that the two above rules progressively find each path through the tree of possible cases.

If no further restrictions were to be applied, the system would continue to evaluate the tree of cases in depth-first order. In the interests of saving processing time, only one of the paths is investigated if the innermost instruction is a call to a SOLO primitive or to a user-defined function. When such an instruction is found and executed, all remaining cases in working memory are deleted. However, in the above example, the innermost call is to the conditional form CHECK (TEST is handled identically by the production system); here the cases are worked through one by one until the conditional has both succeeded and failed at least once (or until there are no more cases in working memory).

Rule 3 concerns itself with non-conditional inner instructions. It says:

3) IF there is user code in working memory and if its head is neither a FOR line nor a CHECK line; and if
working memory contains the token FOR; and if
there is at least one case stored in working memory
THEN execute the line with in the context of the case;
delete the line from working memory.

The last effect of this rule halts further evaluation of the tree of cases.

The purpose of rule 4 is to detect a CHECK line and to ensure that the latter's triple exists:

4) IF the head of the line is a CHECK line but there is no CHECK token in working memory;

THEN add CHECK to working memory;

if the CHECK's triple is found in the database, add the token EXISTS to working memory.

Rule 5 looks for the condition where a FOR-nested CHECK line has been evaluated in the presence of sufficient cases for the conditional test to have both succeeded and failed at least once - or for exhaustion of the list of cases. In either of these circumstances no further evaluation of the CHECK line should take place:

5) IF working memory contains both FOR and CHECK; and if

working memory contains either PRESENT with ABSENT or no remaining cases

THEN create an error frame if there are no more cases;

examine the working memory to see if this frame can be chained to earlier error frames;

remove CHECK from working memory;

delete the line from working memory.

Rule 6 handles the intermediate stages of FOR-nested CHECK evaluation - those where the CHECK has not yet both succeeded and failed:

6) IF working memory contains both FOR and CHECK but no more cases

THEN un-rotate the line in order to generate fresh cases from an earlier FOR instruction.

Rules 7 to 9 detect three kinds of control-statement error:

7) IF working memory contains CHECK; and if

both of the current line's sublines carry the STOP control-statement; and if

there is subsequent (unreachable) code.

THEN add STOP to working memory.

8) IF working memory contains CHECK; and if

both of the current line's sublines carry the EXIT control-statement; and if

there is subsequent (unreachable) code.

THEN add EXIT to working memory.

9) IF working memory contains CHECK; and if

one subline of the current line carries the EXIT control statement and one carries STOP; and if

there is subsequent (unreachable) code.

THEN add both STOP and EXIT to working memory.

Rule 10 calls upon the normal MacSOLO interpreter to return the Lisp Boolean values T or NIL according to whether or not a FOR-nested CHECK instruction succeeds in the current context of cases:

10) IF working memory contains both tokens CHECK and FOR

THEN execute the CHECK line in working memory;

place PRESENT or ABSENT in working memory according to the result of this interpretation.

Rule 11 is similar except that it deals with non-nested CHECK lines and so needs to delete the line after interpretation:

11) IF working memory contains CHECK but not FOR

THEN execute the CHECK line in working memory;

place PRESENT or ABSENT in working memory according to the result of the interpretation;

delete the line from working memory.

Finally, Rule 12 executes single (i.e. non-FOR, non-CHECK) lines of SOLO code:

12) IF working memory holds current code, but neither FOR nor CHECK

THEN execute the code and delete it from working memory.

A glance at these rules will reveal that their order within the production cycle is important, and that certain minor bookkeeping details (such as how the various tokens, once deposited in working memory, are cleared out again) have been omitted for the sake of clarity.

These rules tell the analyser how to analyse SOLO code and how to trap the following kinds of error:

a) A FOR EACH triple which either does not exist in the database or cannot produce any CASEs.
b) A CHECK triple which does not exist in the database.
c) Control-statement errors - certain combinations of control-statements on CHECK/TEST sublines can cause subsequent code to be ignored: double STOP, double EXIT, STOP/EXIT or EXIT/STOP. If any of these occurs on a line other than the last line of a procedure, it is an error.
d) A FOR-nested CHECK which either always succeeds or always fails.
e) A non-nested CHECK which either succeeds or fails. This is not, of course, an error - unless it happens to occur in the context of an unbound variable.

As FOR EACH lines are analysed (rule 1) error frames are created if necessary. Rules 10, 11 and 12 call the normal MacSOLO interpreter, which may signal a run-time error. If it does so, an error frame is created by this module of AURAC in the usual way. Usually, this means that a substantial amount more detail about the error can be stored than could be provided by the interpreter alone - for example, see the description of the RECURSION slot in section 4.2. The following run-time errors are handled in this way:

f) Unbound variable. The normal MacSOLO error messages, which specifiy where the unbound variable was discovered, are often not sufficient to pinpoint the actual error. AURAC is able to "chain" together errors such as (a), (d) and (e) in a cause-and-effect sequence.

g) Undefined procedure.

h) Recursion limit exceeded. Unless the user has written a program involving more than ten levels of subroutine, this signifies endless recursion. In SOLO, this can have one of two causes: a procedure which repeatedly calls itself (or a group of procedures which do the same), or a loop in the database. AURAC is able to decide which and to inform the user accordingly.

i) Attempt to DESCRIBE or LIST non-existent database entities.

j) Incomplete database chain. This error arises when the user (quite reasonably) views a certain set of database triples as being in a conceptual "chain", and tries to make use of this fact via the APOSTROPHE construct when in fact the chain is not complete. For example:

```
10 FOR EACH CASE OF FIDO LIKES ?WHO
  A NOTE FIDO LIKES (*WHO'S BROTHER)
```

should assign a value to *WHO on line 10. Suppose that that value is ROVER. On line 10A the APOSTROPHE operation should retrieve from the database the missing element in the triple ROVER BROTHER ..., and should supply this as the third argument to the NOTE instruction. If the triple FIDO LIKES ... is present, but no triple corresponding to ROVER BROTHER ... can be found, the Incomplete Chain error results. Note that use of a CHECK on the FOR EACH subline:

```
10 FOR EACH CASE OF FIDO LIKES ?WHO
  A CHECK *WHO BROTHER ?B
  AA If Present: NOTE FIDO LIKES *B ; NEXTCASE
  AB If Absent : ; NEXTCASE
```

would not generate this error, since the CHECK syntax supplies a specific action in the "If Absent" condition. Notice also the NEXTCASE control statement, which is of course only appropriate when CHECK or TEST is used in a FOR EACH subline.

Any of these errors, if found, will generate an error frame as above. Some of them, such as type (c), have several sub-types. Each is explained, with the help of the corresponding error frame and a set of canned messages (see section 4.2). AURAC also keeps a list of all the "holes" (non-existent triples) it finds in the database, and this list is available to the user after analysis.

Some examples of buggy programs and the corresponding messages resulting from this module of AURAC alone are presented below:

1) This procedure does nothing of any value, other than to demonstrate the error message:

```
TO EXIT2

10 CHECK FIDO ISA DOG
  A If Present: PRINT "Yes" ; EXIT
  B If Absent : PRINT "No" ; EXIT

20 PRINT "And that's that"

30 PRINT "So there."

40 PRINT "Bye."
```

The message from AURAC is:

```
"An error on line 10 of EXIT2 at level 2:
 It has a double-EXIT bug.
 This means that lines 20, 30 and 40 of EXIT2
 are not reached."
```

2) The INFECT procedure from chapter 1, but with a loop in its associated database:

```
TO INFECT /X/

10 NOTE /X/ HAS FLU

20 CHECK /X/ KISSES ?WHO
  A If Present: INFECT *WHO ; EXIT
  B If Absent : EXIT

JOHN---KISSES-->MARY

MARY---KISSES-->ANDREW

ANDREW---KISSES-->URSULA

URSULA---KISSES-->JOHN
```

The message from AURAC is:

> "An error on line 30 of INFECT at level 5:
> You have a loop in your database via
>
> JOHN MARY ANDREW URSULA JOHN
>
> This caused your run-time error."

3) Again, merely a demonstration procedure. The top level call is BLAH JOHN, and there is no suitable triple in the database for the CHECK of line 10. This results in *A being an unbound variable. Reference is then made to this unbound variable on line 20. AURAC is able to "chain" the two errors together and so to announce that the cause of the run-time error on line 20 is to be found on line 10.

```
TO BLAH /X/

10 CHECK /X/ IS ?A
  A If Present: PRINT "YES" ; EXIT
  A If Absent : CONTINUE

20 NOTE /X/ IS *A
```

The message from AURAC is:

```
"A possible error on line 10 of BLAH at level 1:

  Your code CHECK /X/ IS ?A is activated
         as CHECK JOHN IS ?A

  and that CHECK fails.
  So there is also...

 An error on line 20 of BLAH at level 1:

  Your code NOTE /X/ IS *A is activated
         as NOTE JOHN IS *A

  and that contains an unbound variable.
  This caused your run-time error.
```

In the last of these examples, only "a possible error" is announced, even though the error concerned is also noted as being the cause of a run-time error. The "possible" message-fragment is generated when a CHECK or TEST line fails with the user-supplied input data. Under different input conditions it might not do so, so AURAC cannot be sure that a genuine error has occurred. The inappropriateness of the "possible" fragment when further analysis of the

remaining code reveals it to be the chainable cause of some other error, is an example of the undesirable complexities which can arise from a canned-message system (see chapter 6).

Here follows a trace (invisible to the user under normal conditions) of AURAC's first module working on the short program from section 1.3.3. For convenience, the program is reprinted here:

```
TO TRY /X/

10 CHECK /X/ IS UP
  A If Present: FORGET /X/ IS UP ; CONTINUE
  B If Absent : NOTE /X/ ISA UP ; CONTINUE

20 CHECK AURAC LOVES TONY
  A If Present: EXIT
  B If Absent : EXIT

30 PRINT /X/ "IS UP."
```

Items in square brackets are annotations.:

Beginning skimmer's analysis of (TRY FLAG)

```
Original WM for line 10 of TRY:
WM =
((LINE
   (CHECK /X/ IS UP)
   (FORGET /X/ IS UP & CONTINUE)
   (NOTE /X/ IS UP & CONTINUE)))
```

[Working Memory initially holds the user's code for line 10. The production rules are then applied in order.]

P4 fired: CHECK/TEST line found

P11 fired: execute CHECK/TEST line and subline

```
                    Error found on line 10 of TRY:
                    "CHECK-triple does not exist
                     in your database."
```

[This error is actually ERROR2 (ERROR1 represented a cliche - see below). Now that the code for line 10 has been fully dealt with, all that remains in the working memory is a note of the fact that no variables were created during ERROR2, and the notes CHECK and ABSENT which were used in the course of analysis. The production cycle continues.]

```
WM =
((ERRVARS (NIL . ERROR2))
 ABSENT
 CHECK)
```

[Analysis proceeds to line 20. Notice that the ERRVARS information is carried forward into the new "original" working memory. It is used for chaining unbound variable errors back to their causes if any.]

```
Original WM for line 20 of TRY:
WM =
((ERRVARS (NIL . ERROR2))
 (LINE
  (CHECK AURAC LOVES TONY)
  (NIL & EXIT)
  (NIL & EXIT)))

P4 fired: CHECK/TEST line found

P8 fired: Double-exit bug

P11 fired: execute CHECK/TEST line and subline

                              Error found on line 20 of TRY:
                              "CHECK-triple does not exist
                               in your database."

                              Error found on line 20 of TRY:
                              "Double-Exit"
```

[Two errors were found this time: ERROR5 and ERROR6. The other two errors from the sequence, ERROR3 and ERROR4, represent matches found between line 20 and two lines from the multi-line cliches UPDATE-M_1 and FETCH-DO-M_1. They are later discarded by the system.]

```
WM =
((ERRVARS (NIL . ERROR6) (NIL . ERROR5) (NIL . ERROR2))
 ABSENT
 EXIT
 CHECK)

End of skimmer's analysis of TRY
```

[Since analysis follows the normal flow of control, the double exit on line 20 brings the production system to a halt. But this does not prevent AURAC from noticing that line 30 is a portion of unreached code.]

At this point AURAC would be able to report to the user:

Line 10: CHECK-triple does not exist in your database.
Line 20: CHECK-triple does not exist in your database.
Line 20: Double-Exit --> Line 30 not reached.

4.3 MODULE 2: CLICHE ANALYSIS

As each line of the user's code passes through the above production system, it is also passed for analysis to the cliche module and the data flow module of AURAC. Here is the SOLO cliche represented by line 10 of TRY. It FORGETs a triple if it is already present in the database, and otherwise NOTEs it. The internal variables a, b, and c can represent any user-supplied tokens:

```
10 CHECK a b c
  A If Present: FORGET a b c ; CONTINUE
  B If Absent : NOTE a b c ; CONTINUE
```

The structure of this cliche, ignoring possible user variations and cosmetic printouts, is:

```
<n> CHECK <triple>
   A FORGET <same triple> ; CONTINUE
   B NOTE <same triple> ; CONTINUE
```

and it is this "skeleton" which the cliche-analyser looks for in the user code. The skeletons are actually stored in

standardised data objects which have slots into which the various "bones" of the skeleton fit. The bones are:

1) the main instruction on the line (which may be FOR EACH CASE OF, CHECK/TEST, some single-line primitive such as PRINT, or a subroutine call. If it is a nest of FOR EACH CASE OF instructions, these are automatically collapsed into a single FOR EACH CASE OF instruction, in the same way as in the skimmer module. But of course this can only happen if all of the necessary triples are present in the database and can generate new cases.

2) the FOR EACH CASE OF subline if any, which may in turn be a CHECK/TEST instruction, or a single-line call as above.

3) the A and B subline instructions, if any, of a CHECK/TEST instruction, whether the latter was on the main line or on a FOR EACH subline.

4) the A and B control-statements if any.

The current skeleton would be stored like this:

```
MAINLINE:    (CHECK)
FORSUBLINE:  NIL
A-SUBLINE:   (FORGET)
B-SUBLINE:   (NOTE)
A-CONTROL:   (CONTINUE)
B-CONTROL:   (CONTINUE)
```

and that simple structure completely defines the "type" of SOLO line concerned. In the case of multi-line cliches, each line will have its own skeleton, and its own example or "schematic" (see below).

Each skeleton in the cliche library is checked against the line of user code by a pattern-matcher. A cliche skeleton corresponds very closely to the fixed parts of a programming construct such as the MacLISP DO-loop. If the skeleton and the equivalent parts of the user's code are identical, a match is signalled. If there is a single difference (e.g. in one control statement), an error-frame is created as above. If there is more than one difference, a mismatch is signalled and analysis of the code line in terms of that particular cliche ceases. If a match is found, the analyser goes on to look for mismatches between the line of user code and an "schematic" of the cliche line drawn from the same library. The schematic might look like this:

```
((CHECK >N >O ?P)
 (FORGET <N <O *P & CONTINUE)
 (NOTE <N <O *P & CONTINUE))
```

This time the matching is more sophisticated, but again only a single difference will be counted as a detected error. Matching now allows the binding of the temporary variables N, O and P so as to store the arguments to the CHECK instruction, and these values are compared with the arguments to the NOTE instruction in order to ensure that the two user-entered triples are the same. Similarly, the matcher can now accept items such as "ANY" in places where the user might decide to insert an arbitrary node, relation or procedure name, or control-statement, other than any name

already bound to N, O or P. The symbols N, O and P themselves are unrestricted, and can occur in any number of different cliches without confusing the matcher, but of course they must not be LISP special symbols. The operators ">" and "<" will also match to a wildcard or to a starred variable name respectively; similar operators "?" and "*" behave in the same way, and are provided merely to make the writing of the cliche library easier.

If, for example, this second match is perfect except that some symbol other than that already bound to the variable P appears in the NOTE instruction, an error is assumed. Owing to the paucity of SOLO syntax, it is in fact only when a single mismatch is found that an error can be signalled: two mismatches might well signify a different cliche altogether.

Other keywords such as SELF (see the cliche IMPLICATE, below) and SUBR can be inserted into the schematics of recursive or subroutine-calling cliches; and the keyword NO-OP signifies any SOLO instruction such as PRINT or DESCRIBE which cannot affect the database. For example, the INFECT procedure from section 1.3.1 is expressed as a cliche thus:

Line 1: ((note >a >b >c))

Line 2: ((check <a >d ?e)

```
(self <a & exit)
(no-op & exit))
```

With the gradual refining of the library of cliches SELF, and SUBR have dropped out of use. But they have been left as facilities in the matcher for the sake of future expansion. (The changing of the semicolon into an ampersand is done because the semicolon is a special character in LISP - the token remains merely a marker to separate the control-statement from any instruction appearing on the same subline). And the matcher understands cases where although its schematic might specifiy a CONTINUE control statement, the equivalent word in the code is EXIT because the procedure line concerned happens to be the last in that particular procedure.

In more complex cases there will be more than one line comprising the cliche itself, and they may between them contain several wildcards which the matcher has to "understand" as matching the corresponding variables. Each line of code is tried against each line of every cliche in the library. The apparent computational overhead incurred is easily justified by regarding cliche recognition as a conceptually parallel process. In actual fact, the overhead is not unduly burdensome - but, see chapter 6.

When a match or very close match is found, that fact is remembered. Subsequently, the analyser goes through these matches again, and where only part of a multi-line cliche has been found, it rejects all the matches concerned. Thus, so long as any two cliches differ by more than one item per line, they can reliably be distinguished from one another. AURAC always picks the cliche which most closely matches the code as supplied. Here is an example of a procedure which contains a four-line cliche. The cliche replaces one triple with another, the second being inferred from the first, as follows:

```
TO DEM1

10 CHECK FIDO LIKES ?V
  A CONTINUE
  B EXIT

20 CHECK *V BROTHER ?P
  A CONTINUE
  B EXIT

30 FORGET FIDO LIKES *V

40 NOTE FIDO LIKES *P
```

Suppose now that the user inadvertently typed *X instead of *P on line 40. The typical message from this module of AURAC would be:

```
"Lines 10 of DEM1, 20 of DEM1, 40 of DEM1 and 60 of DEM1
   seem to be intended to remove one triple from your
   database and to replace it with another,
   but on line 60 of DEM1 you have written *X
   where perhaps you meant to write *P.
```

The procedure name is given along with the number of each code-line comprising the cliche because, of course, naive

users may spread cliches across procedure boundaries. The cliche detected here is that named UPDATE-M below. A trace of this second module of AURAC in action follows. Again it is working on the program from 1.3.3:

```
TO TRY /X/

10 CHECK /X/ IS UP
  A If Present: FORGET /X/ IS UP ; CONTINUE
  B If Absent : NOTE /X/ ISA UP ; CONTINUE

20 CHECK AURAC LOVES TONY
  A If Present: EXIT
  B If Absent : EXIT

30 PRINT /X/ "IS UP."
```

[AURAC tries the first cliche in its library, attempting to match it against the user's code for line 10. Whenever the skeleton matches the code, an attempt is made to match the schematic against the code. Note that the symbol M occurring in a cliche name merely signifies that the cliche concerned is a multi-line cliche. In such cases a numeral is also added to the name of each line.]

```
                                        Trying UPDATE-M_1

(UPDATE-M_1 TRY (10))

              [The library cliche looks like this...]
((CHECK >A >B >C) (NO-OP & CONTINUE) (NO-OP & EXIT))

              [And line 10 looks like this...]
((CHECK /X/ IS UP) (FORGET /X/ IS UP & CONTINUE)
                   (NOTE /X/ ISA UP & CONTINUE))

              [The first word matches.]
CHECK matches CHECK

              [AURAC goes on to inspect the triple.]
(PHRASE (>A >B >C)
(SUBLINE (/X/ IS UP))

>A matches /X/
>B matches IS
>C matches UP

              [OK so far; now the first subline.]
(PHRASE (NO-OP & CONTINUE))
```

```
(SUBLINE (FORGET /X/ IS UP & CONTINUE))

                [NO-OP doesn't match FORGET /X/ IS UP, so:]

                                        No match

[AURAC runs through the rest of its stored cliches, until it
gets to the one called FLIP-FLOP:

                                        Trying FLIP-FLOP

(FLIP-FLOP TRY (10))

                [The library cliche:]
((CHECK >N >O ?P) (FORGET <N <O *P & CONTINUE)
                   (NOTE <N <O *P & CONTINUE))

                [Line 10 of the user's code:]
((CHECK /X/ IS UP) (FORGET /X/ IS UP & CONTINUE)
                    (NOTE /X/ ISA UP & CONTINUE))

                [The first word matches.]
CHECK matches CHECK

                [Looking at the triple...]
(PHRASE (>N >O ?P))
(SUBLINE (/X/ IS UP))

>N matches /X/
>O matches IS
?P matches UP

                [OK.  Now the first subline.]
FORGET matches FORGET

(PHRASE (<N <O *P))
(SUBLINE (/X/ IS UP))

                [The sudden appearance of error totals
                 is an artifact of the tracer.]
<N matches /X/ with error NIL  TOTAL = 0
<O matches IS with error NIL  TOTAL = 0
*P matches UP with error NIL  TOTAL = 0
& matches &
CONTINUE matches CONTINUE

                [First subline OK.  Now the second.]
NOTE matches NOTE

(PHRASE (<N <O *P) & CONTINUE))
(SUBLINE (/X/ ISA UP & CONTINUE))

<N matches /X/ with error NIL  TOTAL = 0
```

```
<0 matches ISA with error (ISA . IS)  TOTAL = 1
*P matches UP with error NIL  TOTAL = 1
& matches &
CONTINUE matches CONTINUE
```

Imperfect match accepted:
FLIP-FLOP on line 10 of TRY.

The match is "imperfect" because of the single mismatch, between IS and ISA, found on the second subline. This cliche is in fact the only one to be accepted overall. Its "imperfection" is recorded as ERROR1, and the resulting printout to the user is this:

```
Line 10 seems to be intended to FORGET the triple /X/ IS UP
  if it exists, and to NOTE it otherwise
  but on line 10B of TRY you have written ISA
  when perhaps you meant to write IS.
On line 20, TRY looks in the database to see if the flag
  AURAC LOVES TONY is Present, and if so EXITs
  Otherwise, it EXITS.
On line 30, TRY prints a message.
```

Remember that the double-EXIT bug on sublines 20A and 20B is trapped by AURAC's Skimmer module. As mentioned in the previous section, subsequent analysis of subline 20 shows it to match (actually it is a near miss: one control-statement varies in each case) to one line in each of the two multi-line cliches UPDATE-M_1 and FETCH-DO-M_1. But the error frames so created are later ignored in the absence of the companion lines from these multi-line cliches.

Below will be found full details of the nine cliches currently held in AURAC's library. Several more cliches have been discovered in the course of our work with SOLO, but as discussed in chapter 6 near-misses to some SOLO cliches (such as its equivalent of AND and OR) are not reliably distinguishable from one another. Those included in the library are those which can, with a fair degree of consistency, be correctly detected when they arise in user code.

One slot in the skeletons was not mentioned above: the MULTI slot, which records the names of all lines in any cliche, and hence of the companion lines in multi-line cliches. ANNOUNCE is normally initialised to NIL for multi-line cliches - in case the companion lines are not found in the code - and to T for single line cliches. Notice that if slots are not specified, the system assumes a NIL entry.

1) UPDATE-M replaces one triple (a b c) with another similar one (a b d):

```
update-m_1      (mainline (check)
                 forsubline nil
                 a-subline nil
                 b-subline nil
                 a-control (continue)
                 b-control (exit)
                 schematic ((check >a >b >c)
                            (no-op & continue)
                            (no-op & exit))
                 announce nil
                 multi (update-m_1 update-m_2
                        update-m_3 update-m_4))
```

```
update-m_2      (mainline (check)
                 forsubline nil
                 a-subline nil
                 b-subline nil
                 a-control (continue)
                 b-control (exit)
                 schematic ((check <c any >d)
                            (no-op & continue)
                            (no-op & exit))
                 announce nil
                 multi (update-m_1 update-m_2
                        update-m_3 update-m_4))

update-m_3      (mainline (forget)
                 schematic ((forget <a <b <c))
                 announce nil
                 multi (update-m_1 update-m_2
                        update-m_3 update-m_4))

update-m_4      (mainline (note)
                 schematic ((note <a <b <d))
                 announce nil
                 multi (update-m_1 update-m_2
                        update-m_3 update-m_4))
```

2) PSEUDO-P-M is a SOLO "predicate": under conditions specified by its first CHECK, it NOTEs some standard triple. The presence of this triple is used later in the program to enable some action.

```
pseudo-p-m_1    (mainline (check)
                 forsubline nil
                 a-subline (note)
                 b-subline nil
                 a-control (exit)
                 b-control nil
                 schematic ((check any any any)
                            (note >h >i >j & continue)
                            (any & any))
                 announce nil
                 multi (pseudo-p-m_1 pseudo-p-m_2))
```

```
pseudo-p-m_2  (mainline (check)
               forsubline nil
               a-subline nil
               b-subline nil
               a-control nil
               b-control nil
               schematic ((check <h <i <j)
                          (any & any)
                          (any & any))
               announce nil
               multi (pseudo-p-m_1 pseudo-p-m_2))
```

3) INFECT recurses if its CHECK line succeeds. It is most often used, as already explained, for propagating inferences along a database chain.

```
infect-m_1    (mainline (note)
               schematic ((note >a >b >c))
               announce nil
               multi (infect-m_1 infect-m_2))

infect-m_2    (mainline (check)
               forsubline nil
               a-subline (self)
               b-subline nil
               a-control (exit)
               b-control (exit)
               schematic ((check <a >d ?e)
                          (self <a & exit)
                          (no-op & exit))
               announce nil
               multi (infect-m_1 infect-m_2))
```

4) IMPLICATE uses a FOR EACH loop and recurses if an inner CHECK SUCCEEDS. It is the standard SOLO method of searching a database tree, and is named after the course assignment in which such a possibility is likely to arise (see chapter 3).

```
implicate     (mainline (for)
               forsubline (check)
               a-subline (self)
               b-subline nil
               a-control (nextcase)
               b-control (nextcase)
               schematic ((for >a >b ?c)
                          (check *c any any)
                          (self *c & nextcase)
                          (no-op & nextcase))
               announce t
               multi (implicate))
```

5) CLEANUP applies FORGET to all triples matching the pattern (k 1 ?):

```
cleanup       (mainline (for)
               forsubline (forget)
               schematic ((for >k >1 ?m)
                          (forget <k <1 *m))
               announce t
               multi (cleanup))
```

6) FLIP-FLOP FORGETs a triple if it is Present, NOTEs it if Absent:

```
flip-flop     (mainline (check)
              forsubline nil
              a-subline (forget)
              b-subline (note)
              a-control (continue)
              b-control (continue)
              schematic ((check >n >o ?p)
                         (forget <n <o *p & continue)
                         (note <n <o *p & continue))
              announce t
              multi (flip-flop))
```

7) CAUT-NOTE-M NOTEs a triple, but first tests to see if a similar triple already exists, and FORGETs it if so. Such an arrangement is often found when, for example, a database counter is to be augmented.

```
caut-note-m_1 (mainline (check)
               forsubline nil
               a-subline (forget)
               b-subline nil
               a-control (continue)
               b-control (continue)
               schematic ((check >a >b ?c)
                          (forget <a <b *c &continue)
                          (no-op & continue))
               announce nil
               multi (caut-note-m_1 caut-note-m_2))

caut-note-m_2 (mainline (note)
               schematic ((note <a <b any))
               announce nil
               multi (caut-note-m_1 caut-note-m_2))
```

8) FETCH-DO fetches a value to be NOTEd as part of another triple. For example,

```
   10 CHECK FIDO BROTHER ?B
     A If Present: NOTE *B ISA DOG ; CONTINUE
     B If Absent :CONTINUE

fetch-do      (mainline (check)
               forsubline nil
               a-subline (note)
               b-subline nil
               a-control (continue)
               b-control (continue)
               schematic ((check >a >b >c)
                          (note any any <c & continue)
                          (no-op & continue))
               announce t
               multi (fetch-do))
```

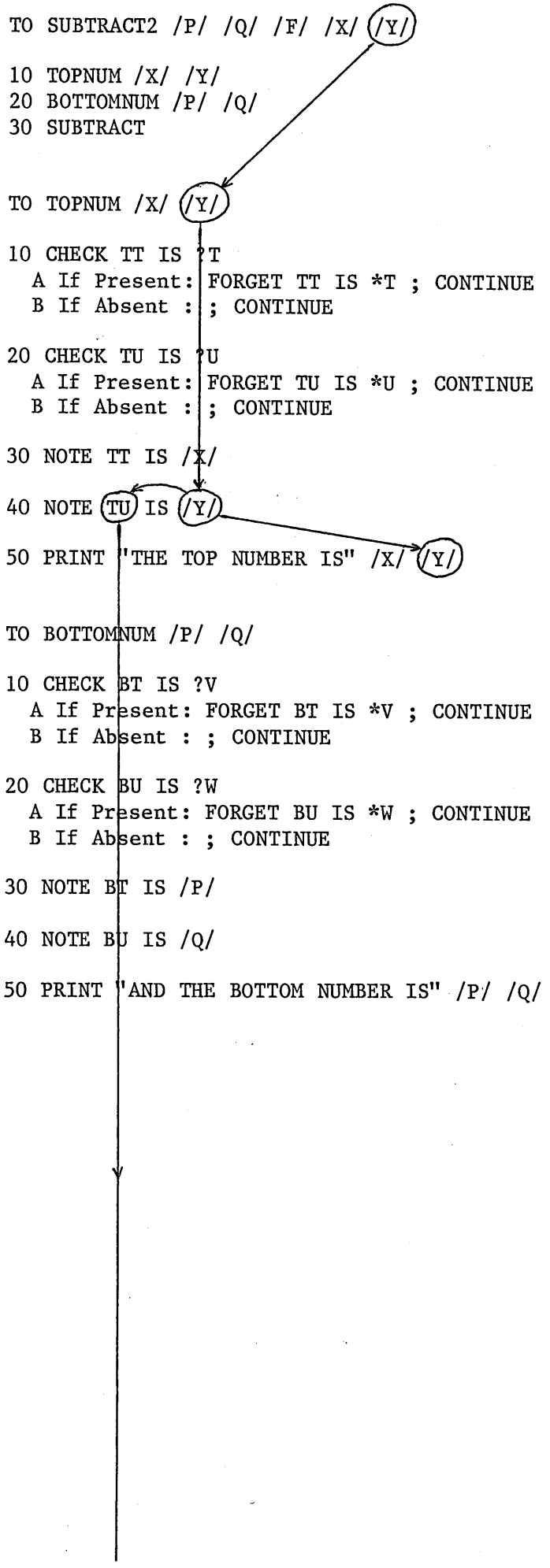
9) FETCH-DO-M does the same thing in two lines (this will be commented upon in chapter 6):

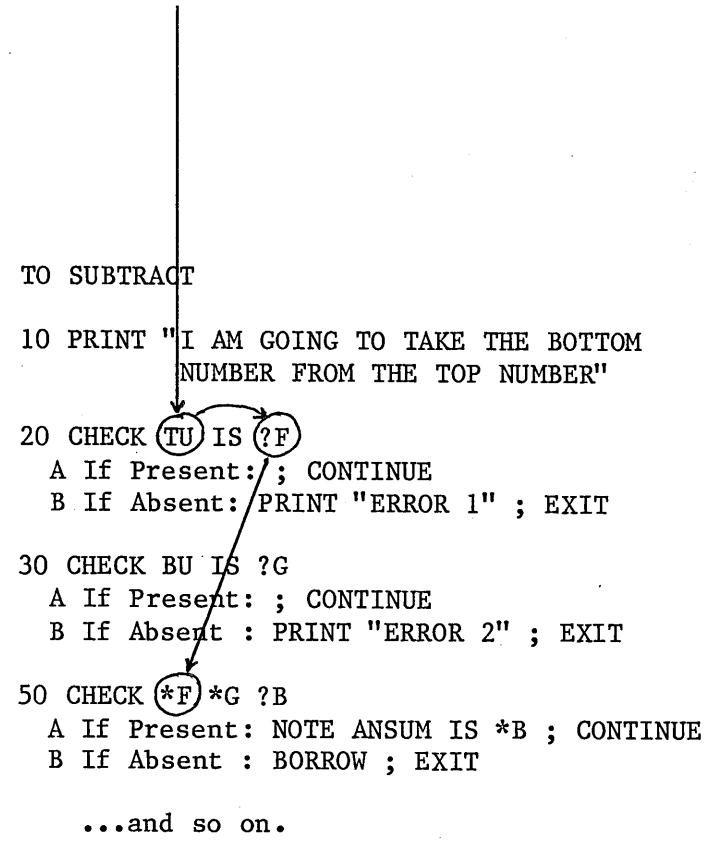
```
fetch-do-m_1  (mainline (check)
               a-subline nil
               b-subline nil
               a-control (continue)
               b-control (exit)
               schematic ((check >a >b >c)
                          (no-op & continue)
                          (no-op & exit))
               announce nil
               multi (fetch-do-m_1 fetch-do-m_2))

fetch-do-m_2  (mainline (note)
               schematic ((note any any <c))
               announce nil
               multi (fetch-do-m_1 fetch-do-m_2))
```

It is normally in cases of variable confusion (the user having accidentally typed the wrong variable name) that cliche recognition appears to be most useful to beginners. Other errors, such as spelling errors, will normally have been handled at a much earlier stage by MacSOLO's spelling corrector. Use of the wrong node or relation name, an equivalent error, has not so far occurred in practice.

4.4 MODULE 3: DATA FLOW ANALYSIS

Data flow analysis involves the setting up of "expectations", and their subsequent "satisfaction". The result conceptually resembles a flow-chart. For example, a CHECK instruction involving a wildcard (and hence binding a variable) sets up the expectation that that variable will be referred to later in the code. Similarly, if there is a call to a user-defined procedure, there is the expectation that the procedure's formally declared parameters will be referred to as execution proceeds. In either case the "expected" items (bound variable, formal parameter) can be equated to the cause of the expectation (result of a CHECK search, argument to the procedure call). AURAC keeps track of both expectations and satisfactions so that at any point during analysis a variable or parameter can be traced back to the point at which it entered the program (often, the original top-level call). For example, here is part of the long subtraction routine used as a demonstration at Summer School (a detailed discussion of which will be found in section 4.6.1). It uses subroutines TOPNUM and BOTTOMNUM, which are actually setup routines ensuring that the correct input digits are entered into the database before subtraction proper begins. One "track" of its data flows is superimposed upon the code:


TO SUBTRACT2 /P/ /Q/ /F/ /X/ /Y/
10 TOPNUM /X/ /Y/
20 BOTTOMNUM /P/ /Q/
30 SUBTRACT
TO TOPNUM /X/ /Y/
10 CHECK TT IS ?T
A If Present: FORGET TT IS *T ; CONTINUE
B If Absent : ; CONTINUE
20 CHECK TU IS ?U
A If Present: FORGET TU IS *U ; CONTINUE
B If Absent : ; CONTINUE
30 NOTE TT IS /X/
40 NOTE TU IS /Y/
50 PRINT "THE TOP NUMBER IS" /X/ /Y/
TO BOTTOMNUM /P/ /Q/
10 CHECK BT IS ?V
A If Present: FORGET BT IS *V ; CONTINUE
B If Absent : ; CONTINUE
20 CHECK BU IS ?W
A If Present: FORGET BU IS *W ; CONTINUE
B If Absent : ; CONTINUE
30 NOTE BT IS /P/
40 NOTE BU IS /Q/
50 PRINT "AND THE BOTTOM NUMBER IS" /P/ /Q/



Ideally, it should not be the case that a SOLO program will make any permanent changes to the database. A correct SOLO program would consist of three sections: a SETUP routine to establish the prior conditions of the database; the program proper; and a CLEARUP routine to restore the database to normal. Therefore, each NOTE instruction "expects" both a CHECK reference to the triple noted, and a FORGET of the same triple. However, it is perfectly legal SOLO usage to put the CLEARUP routine before the program proper, and the SETUP routine at the end. (Messy, but legal). Because of this, it is possible for the FORGET expected by a NOTE lexically to precede the NOTE itself. This is where AURAC's method is superior to symbolic evaluation techniques such as those of Laubsch and Eisenstadt (1981): the latter would derive the wrong

overall net effect from such an inversion of the expected order of operations

Of course, it is not actually illegal to write a SOLO program which does make permanent (but intentional) changes to the database - for example a simple model of learning would have to do so. For this reason, any imbalances which AURAC finds in its system of expectations and satisfactions are merely pointed out to the user, and are not announced as definite errors.

Expectations are placed on a stack as they are generated, in a standardised form which specifies the place (procedure-name, line number) where the expectation arose together with a "descriptive" token such as the word NOTE or the name of a newly-bound variable. When this expectation is found to be satisfied in the code (in the case of NOTE, it is possible for the satisfaction to be already present, waiting to be "expected"!) it is deleted from the expectations stack, but a copy of the same information is placed on the satisfactions stack. A bound variable or a formal parameter may change its name as it is passed into a subroutine, or if it occurs along with a wildcard in a FOR EACH or CHECK triple. Therefore the satisfaction just placed on the stack also carries a note as to the variable's new name.

Thus, by balancing the expectations against the satisfactions during analysis, AURAC can detect such things as bound variables to which no subsequent reference is ever made, or triples which are NOTEd but never CHECKed. So the simplest use of data flow analysis is to be able to issue messages such as:

"The variable *X created on line 20 of FOO is never used."

"The NOTE instruction on line 30 of FOO is unnecessary."

The trace of AURAC's third module analysing the same piece of code from 1.3.3 follows:

```
TO TRY /X/

10 CHECK /X/ IS UP
  A If Present: FORGET /X/ IS UP ; CONTINUE
  B If Absent : NOTE /X/ ISA UP ; CONTINUE

20 CHECK AURAC LOVES TONY
  A If Present: EXIT
  B If Absent : EXIT

30 PRINT /X/ "IS UP."


Line T of TRY expecting /X/ in TRY marker (/X/)

[Line T is the title line.  The marker is the new name, if
any, of the piece of data being followed.]

/X/ satisfied on line 10 of TRY marker (/X/)
remaining expectations:
NIL

IS cannot satisfy existing expectation
UP cannot satisfy existing expectation

TRY 10 (CHECK /X/ IS UP) onto leftover sats list
remaining expectations:
NIL
```

```
[The CHECK on line 10 is remembered because the rule is that
a NOTE expects both a CHECK and a FORGET.  However, these
latter need not occur lexically after the NOTE itself.  In
fact, in TRY, the NOTE occurs on line 10B.  However, a CHECK
without an accompanying NOTE (as on line 20) need not
signify any error.]

/X/ repeated on line 10B of TRY marker (/X/)

ISA cannot satisfy existing expectation
UP cannot satisfy existing expectation

Line 10B of TRY expecting (CHECK /X/ ISA UP) in $ marker NIL
Line 10B of TRY expecting (FORGET /X/ ISA UP) in $ marker NIL
remaining expectations:
((CHECK /X/ ISA UP) (FORGET /X/ ISA UP))

[The NOTE instruction on line 10B generates the expectation
of a matching FORGET and a matching CHECK.]

AURAC cannot satisfy existing expectation
LOVES cannot satisfy existing expectation
TONY cannot satisfy existing expectation

TRY 20 (CHECK AURAC LOVES TONY) onto leftover sats list
remaining expectations:
((CHECK /X/ ISA UP) (FORGET /X/ ISA UP))
```

The above analysis results in the following two messages to the user:

```
...and also the triple NOTEd on line 10B TRY is never CHECKed.
...and also the triple NOTEd on line 10B TRY is never FORGOTten.
```

4.5 CANONICAL ALGORITHMS

There is a secondary use to which the derived information can be put, in cases where some indication of the programmer's intention is known. And this second use depends upon a particular view of algorithms. An algorithm is, of course, a description of a series of steps which the machine is capable of carrying out and which overall achieves some desired objective.

Translating an algorithm into code involves three things: writing lines of code which perform the same actions as those specified in the steps of the algorithm; including in them variables which will be bound to the appropriate data; and assembling those lines into the correct order. AURAC's analysis inverts this process, i.e. it inspects the code with the aim of finding the algorithm distributed throughout it.

A program may of course do a great many other things besides achieving some particular goal; but one would not be correct in saying that the algorithm for the whole program is also the algorithm for achieving that goal. Conversely, if a program does carry out the steps of an algorithm in their correct order, and using the appropriate data, it must achieve (at least) whatever the algorithm

achieves. AURAC has a library of "canonical algorithms", which are each an algorithm for achieving some particular effect. For example, as will be seen below, the canonical algorithm for subtraction represents precisely the four steps necessary to achieve subtraction of two two-digit numbers, provided that the answer is positive but regardless of whether or not "borrowing" is required. If code representing such an algorithm is fed with suitable input data, AURAC concludes that it will successfully subtract.

By combining the knowledge stored in an "algorithm library" with its data flow analysis, AURAC is able to identify individual algorithm lines within the code.

The "canonical" algorithm thus is an algorithm each of whose lines can be expressed as a single line of SOLO code. If it is possible to spread one step over several code lines, these variants must be included in the algorithm library; this does not in practice lead to any combinatorial explosion. But, it is a restriction which, we hope, a later version of AURAC will not impose.

Whatever form the original (input) data may take as program execution proceeds - bound variables or components of database triples - there must at some point be a line of SOLO code which, say, contains the equivalents of both digits of the units column in the format necessary for them to be compared with the subtraction tables in the database:

CHECK /TU/ /BU/ ?ANSU

AURAC's usefulness here is to recognise this line as an algorithmic step.

The crucial point is this: if the user's program, no matter how large or rambling it may be, executes lines which can be shown (during behind-the-scenes evaluation) to correspond to the steps in some stored algorithm, AURAC can say that the program will achieve whatever goal is assured by that algorithm. The program may do other things, but it will achieve that goal. As an example, here is one of the algorithms from AURAC's library: one which achieves the kind of two-column subtraction where "borrowing" involves adding ten to the minuend in the units column and "paying back" requires the subtraction of one from the minuend in the tens column. The algorithm is written in a form which is an abstraction from actual SOLO code:

```
(subtract (check /tu/ /bu/ ?ansu)        <--1
          (check /tu/ plus10 ?newtu)     <--2
          (check /tt/ 1 ?newtt)          <--3
          (check /tt/ /bt/ ?anst))       <--4
```

The steps are labelled here for the sake of the following explanation. The algorithm says (in English): "Try to subtract the two digits in the UNITS column of the problem (step 1). If this is not possible, "borrow" by adding ten to the top digit in the UNITS column (step 2) and by decrementing the top digit in the TENS column by one (step 3). Then subtract the two digits in the TENS column (step 4). When borrowing is required line 1 will obviously need to be repeated, and AURAC is unaffected by whether this is done via recursive call or by a line of code repeating step 1. Exactly how the corresponding program works, and the database it requires, will be made clear in section 4.6.1. An important aspect of data flow analysis is that the system is able to recognise the equivalence between, say, *NEWTT (the top tens digit after borrowing and paying back) and /TT/ (the same digit when borrowing is unnecessary). Thus, the third line of the above algorithm will be detected in either case. The same applies to the first line.

Provided that these four steps are carried out in the correct order and using the correct data, the corresponding SOLO program will subtract (given a suitable database), regardless of how many subroutines it may be divided into or of how many extra lines may appear amongst those shown in the algorithm itself. There is in existence a perfectly sensible SOLO program which, at top level, merely subtracts

a pair of two-digit numbers. It is in fact one of the demonstration programs whose top level effects are shown to Summer School students at the start of their course. As such it of necessity includes a large number of behind-the-scenes routines to guard against faulty or deliberately unco-operative entries by its users. The full running program occupies well over 80 lines of SOLO code (see Appendix B), but the above canonical algorithm approach correctly identifies it as a working two-column subtraction program, and in fact points out a few minor mistakes where certain triples (extraneous to the algorithm itself) are asserted into the database but never cleared out. Fortunately, these triples are not such as to affect subsequent re-runs of the same program. The question of counter-examples is addressed in chapter 6.

By looking down the stack of satisfactions AURAC is able to trace the path of any particular item of data. But notice that the VALUE of any variable is not taken into account: this part of the analysis remains lexical - i.e. syntactic. Data flow analysis enables mapping of the user code onto the stored algorithms; so that AURAC can ask for example:

> "Was /A/ on the title line of SUBTRACT intended to represent the digit on the top row, tens column? (Y or N)"

If the user's answer is Yes, the equivalence between the

user's token /A/ and the algorithm's token /TT/ is established. This may in turn allow AURAC to establish equivalence between the entire line of user code and, say, line 3 of the algorithm. It is not always necessary to ask questions: the likely range of user tokens is limited, and variable-names such as /TOPTEN/ are in fact detected by a "typical menemonic variable-name" recogniser. When AURAC succeeds in identifying lines of code corresponding to all of the lines in the algorithm, it announces that the program will achieve the corresponding objective (two-column subtraction, in this case; the user is asked to sepcify, at the start of analysis, which of the projects he or she is attempting). If one or more of the expected lines is missing, AURAC points this out to the user:

> "There is no line in your program which repays the borrowed ten."

If everything is perfect, and no errors have been discovered throughout all forms of analysis, AURAC announces:

> "Your program will SUBTRACT.
>
> No errors."

AURAC's analysis of a user's actual subtraction program is illustrated in detail in chapter 5.

4.6 THE LIBRARIES

As has been seen, AURAC relies on its libraries for its abilities to produce any useful debugging information above the level of Higher Level Syntactic errors. It has several times been mentioned that AURAC was intended to work within a limited problem area; at this point it is worth describing exactly what those limits are, and how firmly fixed.

The current implementation of AURAC contains the algorithms and their variants for three of the four Summer School projects described below; the reasons why the fourth project is not amenable to AURAC's kind of analysis are given in chapter 6.

4.6.1 Subtraction

The first of the projects is to give SOLO, which has no numerical primitives at all, the ability to perform two-column subtraction. In fact the project is more interesting than this, since having achieved that goal students are asked to treat their programs as models of children's subtraction skills, along the lines of the analyses of Brown and Burton (1978) and of Young and O'Shea (1980). Students are encouraged to create a number of similar but buggy programs, the bugs imitating the

children's (presumed) errors. Having written the initial correct program with the help of MacSOLO/AURAC, students can then use AURAC to reassure themselves that, for example, a buggy program has no errors other than an intentional missing algorithmic step.

For this project students are supplied with a substantial database which contains all the necessary numerical relationships expressed as SOLO triples. For example, the group:

```
3
'
'---3-->0
'
'---2-->1
'
'---1-->2
'
'---0-->3
'
'---plus10-->13
```

gives all the necessary data for subtracting any digit equal to or less than 3 from 3, and this subtraction can be effected using CHECK:

```
CHECK 3 /N/ ?DIFFERENCE
```

where /N/ is a formally declared parameter corresponding to the digit to be subtracted. *DIFFERENCE is then assigned (by CHECK) the correct value. Should the search fail, i.e. if /N/ is greater than 3, a "borrowing" procedure involving adding 10 to 3 must be invoked; hence the final triple

shown above. (Alternative "borrowing" algorithms require a different database, but are equally easy to implement).

A minimal SOLO two-column subtraction program, corresponding exactly to the canonical algorithm reprinted below is as follows:

```
TO SUBTRACT /BT/ /BU/ /TT/ /TU/

 10 CHECK /TU/ /BU/ ?ANSU
  A ; CONTINUE
  B BORROW /BT/ /BU/ /TT/ /TU/ ; EXIT

 20 CHECK /TT/ /BT/ ?ANST
  A PRINT "THE ANSWER IS:" *ANST *ANSU ; EXIT
  B ; EXIT


TO BORROW /BT/ /BU/ /TT/ /TU/

 10 CHECK /TU/ PLUS10 ?NEWTU
  A ; CONTINUE
  B ; EXIT

20 CHECK /TT/ 1 ?NEWTT
  A SUBTRACT /BT/ /BU/ *NEWTT *NEWTU ; EXIT
  B ; EXIT
```

Here again is the canonical algorithm:

```
(subtract (check /tu/ /bu/ ?ansu)
          (check /tu/ plus10 ?newtu)
          (check /tt/ 1 ?newtt)
          (check /tt/ /bt/ ?anst))
```

The tokens given as formal parameters to SUBTRACT here represent the digits in the problem when set out as children normally would; for example /BU/ represents the 7 and /TT/ represents the 4 in this calculation:

```
4 3
2 7
---
1 6
```

SUBTRACT itself merely effects the subtraction of the two individual columns of the calculation. If its first CHECK fails, the subroutine BORROW is called; and this in effect re-writes the problem as

```
3 13
2  7
----
```

before recursively calling SUBTRACT to try again.

There is no necessity for students to write a recursive program: the above second call to SUBTRACT could be included as extra lines of code in BORROW, to mention just one alternative. As far as AURAC is concerned, this would mean that it "saw" at least one line of its corresponding algorithm occurring twice in the code. AURAC does not class that as an error: it signals only missing lines. However, if the student chose to write BORROW so that it "paid back" the borrowed ten like this:

```
4 13
3  7
----
```

that would count as an algorithm-variant and would need to be included in AURAC's library. In other words, so long as the steps of any given algorithm variant are carried out in their correct order, regardless of any reptitions and regardless of flow of control, AURAC recognises the fact. But if a different sequence of steps is carried out, it needs to be told via an addition to its algorithm library.

An alternative strategy, used by Laubsch and Eisenstadt (1981), is to reason about the mathematical equivalence of the two algorithms (e.g. [X-1-Y = X-(Y+1)]). AURAC prefers to store algorithm variants for two reasons: (a) it wishes to mimic the human expert, who will normally employ only one algorithm at a time; and (b) the technique is easily adaptable for new domains which don't involve mathematical reasoning.

The majority of students tackling this project are in fact quite happy to use the database as supplied, and therefore to employ the first of the above algorithms. However, as was seen above, alternative algorithm variants are not a problem. Students may also, of course, choose to code an algorithm differently from the above. A frequently-encountered example is where he/she decides to store intermediate answers in the database, rather than pass them from one routine to another as arguments:

```
TO SUBTRACT /A/ /B/ /C/ /D/

10 CHECK /B/ /D/ ?U                              <--1
  A If Present: NOTE UNITS IS *U ; CONTINUE
  B If Absent : BORROW /A/ /B/ /C/ /D/ ; CONTINUE

20 CHECK TENS IS ?T
  A If Present: PRINTANS ; EXIT
  B If Absent : ; CONTINUE

30 CHECK /A/ /C/ ?T
  A If Present: NOTE TENS IS *T ; CONTINUE
  B If Absent : ; EXIT

40 PRINTANS


TO BORROW /A/ /B/ /C/ /D/

10 CHECK /B/ PLUS10 ?NEWB                        <--2
  A If Present: ; CONTINUE
  B If Absent : ; EXIT

20 CHECK *NEWB /D/ ?U                            <--1
  A If Present: NOTE UNITS IS *U ; CONTINUE
  B If Absent : ; EXIT

30 CHECK /A/ 1 ?NEWA                             <--3
  A If Present: ; CONTINUE
  B If Absent : ; EXIT

40 CHECK *NEWA /C/ ?T                            <--4
  A If Present: NOTE TENS IS *T ; EXIT
  B If Absent : ; EXIT
```

```
TO PRINTANS

10 CHECK UNITS IS ?U
  A If Present: ; CONTINUE
  B If Absent : ; EXIT

20 CHECK TENS IS ?T
  A If Present: PRINT "Answer is: " *T *U ; CONTINUE
  B If Absent : ; EXIT

30 CLEARUP


TO CLEARUP

10 FORGET UNITS IS ?U
20 FORGET TENS IS ?T
```

The marked lines are the algorithm lines corresponding to the four lines of the previous example. Notice that line 20 of BORROW repeats the same algorithmic step as was carried out by line 10 of SUBTRACT. AURAC recognises all programs of this algorithmic type as successful subtraction programs, provided that the user supplies suitable input data. If either of the above programs is run with input data such that borrowing is not required, AURAC will (correctly, following the flow of control) announce that the two corresponding lines from the algorithm are not carried out. Hence the form of its message: "No line activated during that run of your program...". This input-dependent approach can be more useful to novices than a system which analyses regardless of the supplied arguments.

4.6.2 Collins & Quillian

The second project concerns the Collins and Quillian (1969) model of human semantic memory. This is represented as a tree-structure of sets and of supersets: one path might run (from its bottom upwards) JOEY, CANARY, BIRD, CREATURE. The hypothesis is that information is included in the tree only at that point (that node) where it is most general and yet remains true. For example, to discover ("remember") that JOEY can fly, it is necessary to inspect first JOEY's superset CANARY, and then CANARY's superset BIRD, before the information is found. The statement CREATURE CAN FLY is not, of course, necessarily true, so that BIRD is the correct node to which to attach this data. The SOLO algorithm for reaching the same result is a recursive one which first CHECKs for the required information at the lowest node (JOEY) and, if the information is absent, recursively CHECKs each superset until either the information is found or the top of the tree is reached:

```
TO CONFIRM /X/ /Y/ /Z/

10 CHECK /X/ /Y/ /Z/
  A PRINT "YES" ; EXIT
  B ; CONTINUE

20 CHECK /X/ SUPERSET ?S
  A CONFIRM *S /Y/ /Z/ ; EXIT
  B PRINT "NO" ; EXIT
```

In the above example, this procedure is executed a total of

three times, with the formal parameter /X/ acquring successively the values of JOEY, CANARY and BIRD. The /Y/ and /Z/ parameters retain their initial values CAN and FLY on each call. As with the SUBTRACT project, an actual program may contain much else besides these minimal two; but it is to these that the algorithm in AURAC's library corresponds:

```
(candq (check \node\ \relation\ \property\)
       (check \node\ superset ?superset)
       (self *superset \relation\ \property\))
```

Students go on to extend their programs to handle anomalous cases (birds which cannot fly etc.) and individual exceptions (birds with clipped wings), but AURAC is not at present equipped to handle these variations.

4.6.3 Schema Matching

The third project concerns Schema Matching, and its role in object recognition. The idea is that the mind holds representative descriptions of all known classes of object - chairs, people, planets, gods - and that new objects are perceived as belonging or not belonging to one of these classes on the basis of a match of their attributes. This can be modelled in SOLO, for one class of object only, by a FOR EACH loop:

TO MATCH /NEWOBJECT/ /STANDARD/

```
10 FOR EACH CASE OF /STANDARD/ HAS ?ATTRIBUTE
  A CHECK /NEWOBJECT/ HAS *ATTRIBUTE
  AA ; NEXTCASE
  AB ; EXIT

20 PRINT "YES"
```

This is a simple all-or-nothing match: line 20 will only be executed if the loop runs out of cases without detecting a mismatch. A more elaborate version might look something like this:

```
TO MATCH /NEWOBJECT/ /STANDARD/ /DEGREE/

10 RESET /COUNT/

20 FOR EACH CASE OF /STANDARD/ HAS ?ATTRIBUTE
  A CHECK /NEWOBJECT/ HAS *ATTRIBUTE
  AA INCREMENT /COUNT/ ; NEXTCASE
  AB ; NEXTCASE

30 GREATERP /COUNT/ /DEGREE/

40 CHECK GREATERP DID SUCCEED
  A PRINT "YES" ; EXIT
  B PRINT "NO" ; EXIT
```

This allows the match to be taken as perfect if a certain number (= DEGREE) of individual attributes match. COUNT is the node of a database triple such as COUNT IS ..., and GREATERP is a user-defined procedure which ensures that the triple GREATERP DID SUCCEED is removed from the database before either reasserting it or not depending upon its own comparison of the number of matches (COUNT) and the number required (DEGREE). Once again, additions to the basic

matcher can be of arbitrary complexity, but the only differences between the two FOR EACH loops in the above examples are the subroutine call on line AA and the control-statements on line AB. If these are counted as variants of the same algorithm, the existence or absence of the FOR EACH loop can be taken as an indication of whether or not the program has the essentials of a schema-matcher in it.

```
(match (feco /standard/ has ?attribute)
       (check /newobject/ has *attribute)
       (no-op & nextcase)
       (no-op & exit))

(match (feco /standard/ has ?attribute)
       (check /newobject/ has *attribute)
       (subr & nextcase)
       (no-op & nextcase))
```

The subroutine is not analysed further by the library algorithm shown. However, AURAC is perfectly capable of deeper analysis, provided that a suitable canonical algorithm and message generator are added (by the tutor) to the library. It is important to notice that within the canonical algorithms subroutine algorithms are actually macroexpanded in place, just as the BORROW algorithm is within the SUBTRACT algorithm shown above.

The above three examples with their variants are the minimum set of programs which AURAC can handle. As already mentioned, adding new algorithms is in principle a simple matter, but most students are happy enough to have achieved one of the above during the three days allotted to the project at Summer School. Extending, on an experimental basis, AURAC's library to include more complex programs naturally brings problems in its train - especially as regards cliche recognition (see chapter 6). But the skimmer and cliche recognition modules of AURAC are equally effective regardless of the complexity of the code supplied.

AURAC's algorithm library contains algorithms (and variants of algorithms) for these three major projects. All of them are simple and SOLO-like, as shown above. It is therefore almost a trivial matter for a tutor to add new algorithms should this ever prove necessary. So long as the new algorithm is correct and minimal, there will be no problems.

Adding a new cliche is more complicated, since it is necessary to ensure that the new skeleton could not be confused with any existing skeleton. Again, the highly simplified nature of SOLO code makes the reliable discrimination of similar cliches very difficult. This point is covered more fully in chapter 6.

4.7 PRESENTATION OF RESULTS: INFORM

Firstly, a few general points are worth stressing. We are not, except as an incidental means to an end, teaching our students to become expert programmers. Beyond learning how to operate the SOLO machine so as to produce working programs, they are not expected to know what they are doing - certainly not at the level of detail required for them to employ the more traditional stop-and-search debugging methods. Nor would it be practicable to require them to learn a sophisticated meta-language in which to describe their intentions to the machine. This means, as already hinted, that there has until now been no automated debugging system tailored to their needs.

It is perhaps difficult for expert programmers to empathise with the difficulties faced by such students. They are working in what is to many of them a very alien environment, trying to manipulate a machine which can seem at best recalcitrant, if not actually inimical. What they most often need is reassurance, and in this respect Transparency figures very importantly in any list of design priorities. It is notable that the powerful MacSOLO stepper is by far the most successful and the most popular of the various user aids so far implemented for any dialect of SOLO.

The debugging information generated by AURAC must therefore be such as to be useful to such users. (This consideration was in fact the primary motivation behind the original decision to base its methods empirically upon those of SOLO tutors). We can for the time being excuse the fact that its messages, as distinct from the debugging information itself, will sometimes need to be explained for the students' benefit. The crucial point is that the top level of AURAC must appear to understand what the students understand or need to understand - preferably no less, but certainly no more. It would for example clearly be damaging to our students' progress to inform them that "what you have just typed does not hash to any known bucket". It is better to say "that triple does not exist in your database", which

may seem to the novice to be 90% untrue (as when he/she types NOTE /X/ ISA UP instead of NOTE /X/ IS UP) but is far more meaningful in the SOLO context.

As already mentioned, results are presented via canned message-segments. These are selected and combined into reasonable-looking sentences as required, and printed in line-number sequence. From the user's point of view, the whole resembles a description of the program's execution. For the sake of Consistency and Transparency it is desirable that the "gaps" in this description - corresponding to any error-free lines of code - should be filled. Therefore, via a descriptive mechanism involving more canned messages, AURAC inserts into these gaps simple descriptions of the corresponding lines of code. For example:

"Line 30 checks to see if the triple A---B--->C is present.
If so, it CONTINUES. Otherwise, it prints a message."

By this simple addition AURAC becomes not only a fault-finding mechanism but also a source of reassurance that at least some of a user's program is working as expected. However, the current implementation of AURAC's data flow module also uses canned messages with which to ask the user questions concerning the algorithm, and to present its algorithmic results. For each project - i.e. for each major algorithm together with its variants if any - there is a purpose-built (LISP) routine which selects the

message-segments as required, and although to write a new one is not a difficult matter, it does require a knowledge of LISP syntax on the part of the tutor. Failing any better solution, a future version of AURAC will store the required message-segments along with the algorithm lines, so that a general message-printing routine can handle them. The algorithm library will then be considerably more flexible in its ability to accept new algorithms.

AURAC, then, looks for simple answers to known problems. To the extent that it can do that, we judge it a success. As will be seen from a protocol in the next chapter, its accent on being amenable to its users has been so successful that one student was able to use it in an entirely unexpected fashion: as a stepwise program development aid, rather than as an after-the-event analyser.

CHAPTER 5

AURAC AND MACSOLO IN USE: A SESSION TRANSCRIPT

Here is a short protocol from an experimental subject undertaking the subtraction project as described in chapter 4. The subject was a 26 year old male television director (and not an Open University student). He was completely computer-naive and had no knowledge whatever of programming. He first worked through the SOLO primer, and then through the notes and exercises shown in Appendix C. He then successfully attempted the (subtraction) project using the full MacSOLO/AURAC system. His approach was to write each algorithm step as a separate procedure, and then to combine them once each was working correctly. The protocol demonstrates the behaviour of various automatic correctors (for spelling and for unbalanced parameter-slashes); of the stepper, and of the debugger itself. Square brackets below are comments; angle brackets signify messages flashed by MacSOLO at the top of the terminal. User input is shown in **bold face.**

SOLO: **TO SUBTEN /TT/ /BB/**

...10: **CHECK /TT/ /BB/ ?ANST**

...10A If Present: **PRINT "anst is" *ANST ; CONTINUE**

...10B If Absent : **EXIT**

...20: **DONE**

SUBTEN has been successfully defined and added to your pool of procedures.

```
TO SUBTEN /TT/ /BB/

10 CHECK /TT /BB/ ?ANST
  A PRINT "anst is" *ANST ; CONTINUE
  B ; EXIT
```

[Notice that though the subject omitted the semicolon on line 10B, MacSOLO understood the significance of the subsequent control-statement and inserted the semicolon in its own listing]

SOLO: **SUBTEN 25 24**

[Nothing happens, and the SOLO prompt reappears.]

SOLO:

[Subject: "Oh. Single digits."]

SUBTEN 5 4

anst is 1

[Subject: "I'd like to step through that and see what happens."]

SOLO: **STEP**

Enter SUBTEN 5 4 **[RETURN]**

```
10 CHECK 5---4-->?ANST
   A If Present: PRINT "anst is" *ANST ; CONTINUE
   B If Absent : ; EXIT
```

5 4 ?ANST...Present: *ANST = 1 **[RETURN]**

anst is 1

Exit SUBTEN

[Subject: "Now, what does this debugger thing say?"]

SOLO: **DEBUG**

Name of Project:

[Subject: "Subtract"]

SUBTRACT

Working on SUBTEN...1

Was /BB/ on the title line of SUBTEN meant to
subtract 1 from the Top Tens digit? (Y or N) *N

[AURAC does not yet know that /BB/ represents a
database node - it may represent a relation, so
the question must be asked.]

Was /BB/ on the title line of SUBTEN meant to
represent the digit on the Bottom Row, Tens Column?
(Y or N) *Y

** Correct pattern from model found: line 10 of SUBTEN **

No line activated during that run of your program
did the subtraction of the UNITS column.

No line activated during that run of your program
added 10 to the top digit of the UNITS column.

No line activated during that run of your program
decremented the top digit of the TENS column by 1.

Analyses available for

SUBTEN

type INFORM followed by any of these.

SOLO: **INFORM SUBTEN**

On line 10 it looks in the database to see if the triple
/TT/ /BB/ ?ANST is Present,
and if so prints a message and CONTINUEs.
Otherwise, it EXITs.

Level 1> line 10: CHECK succeeds.

[Subject proceeded to write the corresponding SUBUNIT
procedure. He then went on to write his first version of

SUBTRACT, which was to include both SUBUNIT and SUBTEN]

SOLO: TO **SUBTRACT /BT/ /BU/ /DUMMY/ /TT/ /TU/**

...10: **CHECK /TU/ /BU/ ?ANSU**

...10A If Present: **PRINT *ANST**

Control-statement error. <Type HELP if you don't understand>
[MacSOLO reprints the user's incomplete input, plus a semicolon as a prompt:]

...10A PRINT *ANST ;

CONTINUE

...10B If Absent : **EXIT**

...20: **CHECK /TT/ /BT/ ?ANST**

...20A If Present: **PRINT *ANST CONTINUE**

...20B If Absent : **EXIT**

...30 **OK**

[MacSOLO accepts OK for DONE]

SUBTRACT has been successfully defined and added to your pool of procedures.

TO SUBTRACT /BT/ /BU/ /DUMMY/ /TT/ /TU/

10 CHECK /TU/ /BU/ ?ANSU
  A PRINT *ANST ; CONTINUE
  A ; EXIT

20 CHECK /TT/ /BT/ ?ANST
  A PRINT *ANST ; CONTINUE
  B ; EXIT

SOLO: **SUBTRACT 2 1 FROM 4 3**

2
2

SOLO: **EDIT SUBTRACT**

  edit = SUBTRACT

edit line... **10A CONTINUE**

edit line... **20A CONTINUE**

edit line... **30 PRINT "and the right answer is" *ANST *ANSU**

edit line... **OK**

SUBTRACT has been successfully modified:

```
TO SUBTRACT /BT/ /BU/ /DUMMY/ /TT/ /TU/

10 CHECK /TU/ /BU/ ?ANSU
  A ; CONTINUE
  A ; EXIT

20 CHECK /TT/ /BT/ ?ANST
  A ; CONTINUE
  B ; EXIT

30 PRINT 'and the right answer is" *ANST *ANSU
```

SOLO: **SUBTRACT 2 1 FROM 4 3**

and the right answer is 2 2

[Subject: "All RIGHT..."]

SOLO: **DEBUG**

[AURAC remembers the project]

Working on SUBTRACT...1

Was ?ANSU on line 10 of SUBTRACT meant to represent
the answer in the UNITS column? (Y or N) *Y

[In this case, unlike the above one, AURAC is lucky enough
to get the answer Y from the user on its first attempt]

** Correct pattern from model found: line 10 of SUBTRACT **

Was ?ANST on line 20 of SUBTRACT meant to represent
the answer in the TENS column? (Y or N) *Y

** Correct pattern from model found: line 20 of SUBTRACT **

No line activated during that run of your program
added 10 to the top digit of the UNITS column.

No line activated during that run of your program
decremented the top digit of the TENS column by 1.

Analyses available for

SUBTRACT

type INFORM followed by any one of these.

[Subject: "Right. So now it's just the borrowing, and the - er - decrementing." This student immediately found a novel use for AURAC: as a check that individual algorithmic steps have been coded correctly in segments of a growing overall program. He was also highly amused by the inappropriate (and unsatisfactory) use of the word "decremented" in AURAC's printout.]

SOLO: **INFORM SUBTRACT**

[Subject: "This is just a breakdown. A sort of verbal breakdown."]

On line 10 it looks in the database to see if the triple /TU/ /BU/ ?ANSU is Present, and if so CONTINUEs. Otherwise, it EXITs.
On line 20 it looks in the database to see if the triple /TT/ /BT/ ?ANST is Present, and if so CONTINUEs. Otherwise, it EXITs.
On line 30 it prints a message.

Level 1> 10: CHECK succeeds.
Level 1> 20: CHECK succeeds.

The parameter /DUMMY/ on the title line of SUBTRACT is never used.

[Notice that AURAC picked up the unused /DUMMY/ formal parameter. In this case, of course, its purpose was merely cosmetic; that is, an unused variable does not necessarily indicate an error.]

SOLO: **SUBTRACT 2 2 FROM 8 8**

and the right answer is 6 6

[Subject: "OK. Let's have a look at it"]

SOLO: **STEP**

Enter SUBTRACT 2 2 FROM 8 8 **[RETURN]**

10 CHECK 8---2-->?ANSU
A If Present: ; CONTINUE
B If Absent : ; EXIT

```
8 2 ?ANSU...Present: *ANSU = 6      [RETURN]

 20 CHECK 8---2-->?ANST
    A If Present: ; CONTINUE
    B If Absent : ; EXIT

8 2 ?ANST...Present: *ANST = 6      [RETURN]

 30 PRINT "and the right answer is" *ANST *ANSU

and the right anser is 6 6

 Exit SUBTRACT


SOLO: KILL SUBUNITS

When you typed SUBUNITS did you mean SUBUNIT? (Y or N) *Y

SUBUNIT...no longer exists.


SOLO: TO BORROW

 ...10: CHECK /TU/ PLUS10 ?NEWTU

Undeclared parameter!

 ...10: HELP

PARAMETERS - such as /TU/ - which you refer to within one of
your procedures MUST be included (`declared') in the title
line to that procedure.  You present title line to BORROW
is:

              TO BORROW

If you would like to change it (and/or the declared
parameters), look at HELP RENAME.

 ...10: OK

SOLO: TO BORROW /TU/

 ...10: CHECK /TU/ PLUS10 ?NEWTU

 ...10A If Present: CONTINUE

 ...10B If Absent : EXIT

 ...20: /BT/ BU/ <Slashes don't balance>
```

[[System going down in one minute!]]

When the system was restored, this subject successfully completed his SUBTRACT program, saying at the end that "it must be OK now because DEBUG says so." He had clearly acquired, from his use of the system, the habit of employing the (SOLO) machine's own facilities in order to check his evolving program - not only his unexpected use of AURAC, but his frequent recourse to STEP, show that this is so. This we count as a major triumph, since it implies that this user fully understood the differences and relationships between the various parts of the machine. The conclusion is confirmed by the very small number of "silly" mistakes in this protocol. What has been successfully taught here is, from the didactic point of view, of equal importance to the debugging systems themselves. This was his final SUBTRACT program:

```
TO SUBTRACT /BT/ /BU/ /DUMMY/ /TT/ /TU/

10 CHECK /TU/ /BU/ ?ANSU
  A If Present: CONTINUE
  B If Absent : BORROW /BT/ /BU/ /TT/ /TU/ ; EXIT

20 CHECK /TT/ /BT/ ?ANST
  A If Present: CONTINUE
  B If Absent : EXIT

30 PRINT "And the right answer is" *ANST *ANSU


TO BORROW /BT/ /BU/ /TT/ /TU/

10 CHECK /TU/ PLUS10 ?NEWTU
  A If Present: CONTINUE
```

```
  B If Absent : EXIT

20 CHECK /TT/ 1 ?NEWBT
  A If Present: CONTINUE
  B If Absent : EXIT

30 SUBTRACT *NEWBT /BU/ FROM  /TT/ *NEWTU
```

This user also modified his program to employ a different subtraction algorithm, and modified his database accordingly. No assistance was given to him in doing so, either by the notes supplied or verbally. Since the algorithm variant he used was not at the time included in AURAC's library, his "proof" that his new program was correct was AURAC's announcement that one of the algorithm lines was apparently missing. The variant algorithm is the one mentioned in chapter 4, which repays the borrowed ten by adding one to the lower tens digit, rather than by decrementing the upper tens digit:

```
TO SUBTRACT2 /BT/ /BU/ /DUMMY/ /TT/ /TU/

10 CHECK /TU/ /BU/ ?ANSU
  A If Present: CONTINUE
  B If Absent : BORROW2 /BT/ /BU/ /TT/ /TU/ ; EXIT

20 CHECK /TT/ /BT/ ?ANST
  A If Present: CONTINUE
  B If Absent : EXIT

30 PRINT "And the right answer is" *ANST *ANSU


TO BORROW2 /BT/ /BU/ /TT/ /TU/

10 CHECK /TU/ PLUS10 ?NEWTU
  A If Present: CONTINUE
  B If Absent : EXIT
```

```
20 CHECK /BT/ PLUS1 ?NEWBT
  A If Present: CONTINUE
  B If Absent : EXIT

30 SUBTRACT *NEWBTBT /BU/ FROM /TT/ *NEWTU
```

CHAPTER 6

A CRITICAL APPRAISAL OF AURAC

6.1 THE ACHIEVEMENTS

AURAC's main achievement is to show that very substantial amounts of debugging information can be derived from the program code without the need for any elaborate internal representation of the program's intended effect; and that this information can be obtained without the problem of combinatiorial explosion which so bedevils other debugging methods.

Simple Syntactic errors can arise in any programming language (although of course the specific errors may vary), and their common feature is that they are trappable at their time of entry, by simple demons. Simple Syntactic errors do not require any intelligence on the part of the debugging system. Higher Level Syntactic errors cannot be reliably detected until run-time, but even so the first module of AURAC is in essence little more than a multi-purpose demon,

whose abilities and whose scope for trapping errors can be altered by altering the rules in its production memory.

As can be seen from the tables in chapter 3, syntactic errors form a very high proportion of the errors made by novices; the MacSOLO/AURAC combination performs a useful service in clearly distinguishing these from higher-level, more semantic errors.

The kind of errors here referred to as Cliche errors - which, as has already been pointed out, can be equated to errors in the use of programming constructs - may or may not be considered as syntactic errors. If a construct exists in the host language but is wrongly used, that is a syntactic error; but if no suitable construct exists, the error is a semantic one. In the latter case the errors are particularly difficult to pinpoint without the concept of cliches. This is of course an argument in favour of structured programming techniques, but the problem which AURAC has tackled with some degree of success is how to debug programs written in a relatively non-structured language.

Data Flow analysis is not new, but is here used in a novel way: to ensure that overall a SOLO program does not make any permanent changes to the database - changes which might well interfere with subsequent runs of the same program. In the SOLO context this is not only a debugging advantage but also a didactic advantage: it compels students to consider the long-term effects of every addition to or deletion from their databases, and so to move away from a view of the program as a series of discrete steps and towards seeing it as a coherent operation which has predictable effects. Again, this is a move towards the ideas of structured programming.

In the context of a less database-dependent language than SOLO, the no-overall-effect ideal is less useful - if indeed it has any relevance at all. But AURAC has demonstrated that the notion of "canonical algorithms" is well worth further exploration. Appendix B shows two lengthy subtraction programs which have been successfully analysed in this way, and as was seen in chapter 5 AURAC's simple question and answer technique does not impose any undue extra load on the user. Instead, the extra load falls on the tutor, who must supply the library forms of the algorithms and their variants; and from an OU point of view that is precisely what is required.

An unrelated point concerns MacSOLO's partial resolution of the inevitable conflict between its design principles of Transparency and Simplicity. Whilst Transparency requires that as much as possible of the machine's internal workings (especially where error conditions arise) should be displayed to the user, Simplicity requires that all such displays be reduced to a minimum. Earlier versions of SOLO, stressing Transparency, confused their novice users by printing out large amounts of not immediately useful information such as descriptions of non-fatal run-time errors.

MacSOLO's solution is to give the user a measure of (simple!) control over the degree of transparency. This is most evident in two areas: its error messages, which are brief to the point of terseness but which can always be expanded upon via the one-word instruction HELP; and its stepper. The latter, entirely under the user's control as described in chapter 4, prints run-time messages (such as "noted triple already exists" which would be suppressed for the sake of Simplicity during an actual, non-stepped run.

6.2 AREAS FOR FURTHER IMPROVEMENT

6.2.1 Inability To Cope With Certain Projects

There is a fourth Summer School project in which students are asked to model the cognitive behaviour of Sherlock Holmes as he solves a mystery. One solution is to write a program which, for each of a list of suspects, ascertains whether or not he/she has the necessary motive, opportunity, weapon and so on. Unfortunately, the process of ascertaining guilt may involve a loop containing a simple CHECK as in the schema-matching example in chapter 4, or it may involve indirect inferencing of arbitrary complexity. In other words, there is no definable "standard" algorithm (let alone a minimal one) for this project, and so AURAC is quite unable to say whether or not any supposed Sherlock Holmes program is satisfactory.

We count this as a failure, even though AURAC's main analyses still work normally and are of help. It is a failure in terms of the original hope that AURAC would be able fully to debug any Summer School project. However, it should be pointed out in AURAC's defence that the difficulty lies in the imprecise nature of the problem, rather than in any inadequacy in AURAC's strategies. In the terminology of Rich, Schrobe and Waters (1979b) the resulting correct program is a system rather than an algorithm. This suggests

that, when dealing with novices, we have to supply them with (at least):

a) a simple model of programming techniques (i.e. SOLO);

b) instruction concerning the writing of correct algorithms; and

c) problems which are easily amenable to algorithmic translation.

As mentioned in chapter 1, Kahney has produced some valuable new data concerning the third of these points.

6.2.2 Inability To Discriminate Among Certain Cliches

As already mentioned, SOLO cliches tend not to be easily distinguishable from one another owing to the small number of different tokens involved in each one. Cases have been found where two cliches differ only by one CHECK control-statement. For example, the following is the SOLO equivalent of a Boolean AND construct:

```
10 CHECK FIDO ISA DOG
  A ; CONTINUE
  B ; EXIT

20 CHECK FIDO HAS FLEAS
  A ; CONTINUE
  B ; EXIT
```

And the following is the SOLO equivalent of a Boolean OR construct:

```
10 CHECK FIDO ISA DOG
  A ; EXIT
  B ; CONTINUE

20 CHECK FIDO HAS FLEAS
  A ; EXIT
  B ; CONTINUE
```

If the user's code is:

```
10 CHECK FIDO ISA DOG
  A CONTINUE
  B CONTINUE
```

this may be intended as either an AND or an OR line: there is no way of telling. Such cliches have to be rejected from AURAC's library because the system cannot tell whether the user's code as supplied is a correct coding of one cliche, or an erroneous coding of the other. In the (perfectly legal) circumstances of multiple AND/OR lines, or their combinations, the system similarly cannot tell where one cliche ends and the next begins. However, this is a language-specific problem: one which would not necessarily occur in a richer source language.

In this context there is also the question of just how often in real user code one comes across cliches which are perfect apart from the one error (in skeleton or schematic) required if AURAC is to be able to offer a specific patch. In some respects this is a language-specific question - for example it has already been mentioned that the "just one

error" restriction had to be imposed because of the extreme simplicity of SOLO programming constructs (such as CHECK) and the paucity of different symbols therein.

On the other hand, it can already be said that there may be more than one type of cliche. INFECT and IMPLICATE, for example, occur often enough in SOLO code to be counted as cliches - but this is largely because the course notes with which the students work explicitly tell them to write such things. One would reasonably expect to find comparable patterns of code in the work of novices formally learning any computer language. The remainder of the cliches currently in AURAC's library seem to occur spontaneously, and as has been said this is often without the student being aware that his/her piece of code is something commonly found in the work of large numbers of SOLO users. Our suggestion here is that the rate of occurrence of recognisable cliches - the cliche "density" - is likely to vary from language to language, but that the idea of isolating sections of code on this basis and of applying further analysis to them is sensible in any language.

6.2.3 False Alarms

An allied problem, again almost certainly SOLO-specific and due to the extreme simplicity of SOLO code, is that cliches can be detected where no cliche was intended. For example, line 20 of BORROW in the second of the two example subtraction programs given in the chapter 4 is as follows:

```
10 CHECK *NEWB /D/ ?U
  A If Present: NOTE UNITS IS *U ; CONTINUE
  B If Absent : ; EXIT
```

and this is recognised as a faulty (one differing item) version of the FETCH-DO cliche (see section 4.4) whose example is this:

```
((check any any >c)
 (note any any <c & continue)
 (no-op & continue))
```

The assignment operator (>) and the retrieval operator (<) in the matcher permit the corresponding items of code to be respectively a wildcard (?) and its corresponding bound variable (*), whilst the NO-OP symbol must of course match the case where there is genuinely no operation specified on subline B. Therefore the only difference is the EXIT on that subline of the code versus the CONTINUE in the cliche-example, and AURAC duly announces the possible cliche error. However, the code was never intended to represent that cliche, but merely happened quite fortuitously to take the same form.

In his own work with LISP, Shapiro (1981) recognises that programming cliches and Waters-type PLANs are not isomorphic: considerable work is required to derive the one from the other. Cliches tend to be amorphous (just as the AND and OR constructs in LISP can take any number of arguments, so the comparable code in SOLO can be repeated any number of times, and their parts tend to be scattered amongst what would normally be thought of as PLAN-sized clumps of code (a single AND may comprise arbitrary amounts of subordinate code in the form of conditionals, loops, subroutines, each with its own PLANs and Plan Building Methods). AURAC's empirical, and largely lexical, approach ignores the boundaries of PLANs altogether. Although SOLO code can contain the equivalents of the complexities metioned above, library cliches can be reliably detected regardless of their overlapping with divisions between other conceptual "areas" of the program. Our problem, unlike Shapiro's, has not been that cliches are hard to detect but that they can be detected too easily - cliches found where none was intended, or confusion between similar cliches, as above.

There is also the question of the cost of cliche recognition. As already mentioned, it is an expensive process (as implemented in AURAC), involving detailed pattern-matching of user code against a library of cliches. The difficulty is compounded when one includes the possibilities that single lines from multi-line cliches may form near-perfect matches for single lines from other multi-line cliches; that other unrelated lines may be insterspersed amongst the lines of any given cliche; and that multi-line cliches themselves may be intermixed. There is a rapid and explosive growth of potential matches. Thus, unless the cliche library is carefully tailored to contain only cliches which cannot easily be confused with one another, the process can become too slow and too unreliable for use.

Against these considerations has to be balanced the fact that cliche recognition can detect fatal errors of detail - for example, the wrong variable name as in the example in section 4.4 - which would be missed by any of the other systems described here. AURAC's very first "real" success - that is, the first time it found the error in other than a purpose-built buggy routine - was when its cliche recogniser unerringly spotted the single wrong variable name in a fifty line program.

At one time it was fondly hoped that it might be possible to describe all SOLO programs in terms of cliches, and that these in turn could be described in a higher-order terminology; for example, a line such as

CHECK /BT/ PLUS10 ?NEWBT

would be seen as a "database function", since the PLUS10 relation can be regarded as "operating upon" the /BT/ node to generate the resulting *NEWBT binding. It was hoped that ultimately an entire program could be described in terms of database functions and other cliches, and that such a description would closely parallel the symbolic evaluation methods of Eisenstadt and Laubsch. However, the above-mentioned combinatorial explosion put a stop to this idea. (But, see below).

6.2.4 More Sophisticated Reporting Of Analyses

The error frames created by AURAC during analysis often contain far more information than can conveniently be presented to the user - especially to a novice user. The data stored about a single error - especially where it is an error which "chains" back to an earlier cause - can occupy several dozen lines of printout. In the present implementation AURAC selects from this mass of data those items (the most salient) which can be expressed via its canned messages-segments; and these are in turn designed to be understood by a user who conforms to our own notion of an

"average novice".

There are thus three problems concerning the output of AURAC's results:

a) much detailed debugging information is lost; and

b) the information which is presented makes no allowances for the state of expertise of the individual user.

c) canned messages will inevitably conflict with one another (an example was given in chapter 4).

Below is briefly described a system which might remove the first of these diffculties; the second could only be overcome if AURAC maintained some kind of internal model of the individual student's progress. As mentioned in chapter 2, the current implementation keeps only a very rudimentary record of users' behaviour.

The problem is also domain-specific: Rich et al. (1979b) describe how a learner first gains an understanding of the programming language primitives at a computer science level (in LISP, he/she learns about how CONS, CAR and CDR manipulate pointers in memory-space to create, connect and modify CONS-CELLS), and subsequently is able to grasp the more abstracted idea of operating upon lists, each of which may be composed of many cons-cells in a known pattern.

But this is not the hands-on approach. Anyone who learns LISP from Hasemer (1983b) will learn to manipulate lists first, and will only later be told the nuts-and-bolts details. Similarly with our students. They are not expected ever to know how the SOLO primitives actually achieve their effects, neither in terms of the supporting LISP or PASCAL, nor at the memory-address level. They are presented with the abstracted ideas straight away: that there is a "database" where "triples" can be stored, referenced or deleted, that "variables" become "bound" under certain conditions, and so on. However, from the students' point of view it is desirable that they seem to be more than merely apprentice magicians, learning to get the arcane details of their spells right. Hence the emphasis throughout the SOLO environment and its associated course notes on maintaining a consistent and logical model of the SOLO machine. Our students learn to operate SOLO without detailed understanding, as one might learn to operate any complex machine.

This both simplifies the problem of presentation (there is a finite set of terms in which explanations can be couched) and exacerbates it: the terms themselves can only be explained by analogy, which carries the danger of destroying the all-important consistency of the model. The computer as a telephone exchange, or variables as "boxes"

which "contain" values, are analogies which in a different context - or in a more critical mood on the part of the system designer - might seem like no explanation at all.

The third problem can be obviated only temporarily: by a series of ever more complicated and ad hoc adjustments to the (LISP) message-generating functions. This is not only messy and therefore unsatisfactory, but also means that large amounts of highly language-specific information are scattered piecemeal throughout a series of interlocking LISP routines. This, of course, means that extending the system - with the corresponding need for extended printed explanations - becomes more difficult than it otherwise need be. A future version of AURAC will certainly seek to bring some order to this chaos, or to adopt some entirely new way of presenting its results. For the purposes of our present project, however, it was thought more important to concentrate on being able to derive the debugging information itself: presentation was a more or less secondary issue.

An even more glaring example of this kind of problem occurred in the previous chapter, where a message from AURAC read:

> "No line activated during that run of your program decremented the top digit of the TENS column."

If such comments are to make any sense to the user, he/she must be aware of the "picture" of the subtraction problem to which they refer, e.g.:

```
4  3                        Top Tens      Top Units
2  7                        Bottom Tens   Bottom Units
----    is equivalent to    -------------------------
1  6                        Answer Tens   Answer Units
```

and, of course, of the particular subtraction method (algorithm or algorithm variant) which AURAC is expecting. Ideally, AURAC's printouts would require a sophisticated graphics package, which could say for example:

"Is this what you mean? -

```
/TT/  /TU/
/BT/  /BU/
----------
ANST  ANSU "
```

where /TT/ etc. are the student's own coded symbols, and which could actually show an animated version of the algorithmic steps taking place. We have not attempted any such arrangement, partly because as above it would have detracted from the main thrust of our research, but also because - other than at Summer School itself - many of our students have to work with printing terminals.

6.2.5 Alternative Test Inputs

AURAC's analysis relies upon the correctness of its input data: i.e. it relies upon the user to call his/her program with suitable arguments. In many cases its outputs make obvious the fact of any unsuitable arguments, but of course there are anomalous cases, for example when under the given input conditions a FOR-nested CHECK instruction either always succeeds or always fails. This is why certain apparent errors are signalled by AURAC with the phrase "A possible error on line...". AURAC currently makes no attempt automatically to generate alternative test inputs. Within its present context, this would not be hard to do, but (a) in a wider context it would necessitate the construction of a whole new module for this purpose, and (b) the process would no longer conform to tutors' standard strategies as described in chapter 3.

6.2.6 Data Flow Anomalies

Consider again the instruction

CHECK /BT/ PLUS10 ?NEWBT

AURAC attempts to follow the progress of a piece of original input data /BT/ across this triple. It is clear to us as human beings that the value of the variable *NEWBT is merely a new form of /BT/, after it has been operated upon by the database function PLUS10. However, the token PLUS10, the name of the "database function", is arbitrarily chosen by

the user. There will thus be cases where AURAC, despite its parser, cannot be sure that a data-flow graph drawn through a triple such as the above is correct. When that happens, AURAC is obliged to start a new graph from the ?NEWBT token, and to follow the progress of that in subsequent code, at the same time terminating the graph which led from the top-level call to the /BT/ token. It may then have to ask the user

"Was ?NEWBT on line 20 of BORROW intended to represent..."

rather than being able as is preferable to refer all such questions back to the original input data.

This is not a serious drawback, since in any case the ability to trace tokens back to their sources is only a (useful) by-product of AURAC's system of balanced expectations and satisfactions.

6.2.7 Algorithm Variants

There is an interesting analogy between the "paraphrases" suggested by Simmons (1973), needed for the machine to understand verbally different versions of what is essentially the same infomation input, and AURAC's need to establish the precise "significance" of the various user-selected tokens within SOLO code. AURAC has no "paraphrase rules", and is obliged to ask the user "did this token represent that one (or that operation)" in terms of

some standardised format which the user is presumed to share with the machine (e.g. "did /X/ represent the top column, tens digit?). Even in the context of the simple projects described above, AURAC's method has obvious drawbacks - not the least of which is that a new set of canned questions has to be provided for each new project added to the algorithm library.

But the analogy with Simmons' work can be drawn at a higher level also: with the problem of algorithm variants. The latter are different ways of telling the machine to achieve the same overall effect - i.e. different ways of saying the same thing, or paraphrases. They also strongly resemble Rich's (1981) "overlays". It is entirely possible (though no serious work has been done on this) that these ideas might contribute to a much more elegant solution to the problem of variants than AURAC currently employs. Alternatively, a solution might be found via further consideration of the mathematical equivalence of programs and algorithms at a deeper level - somewhat along the lines of Schank's (1973) "conceptual dependencies". We take up this point in more detail in the next section.

6.3 FUTURE DEVELOPMENTS AND EXTENSIONS TO OTHER AREAS

The vast majority of existing automatic programming aids require a knowledgeable, if not knowledge-packed, input from their users. We contend that in the future, when large numbers - if not the majority - of computer users will lack such specialised knowledge, that path is not the most practical to follow. Future users will want aids which, although powerful and intelligent in themselves, can describe bugs and suggest cures for them in layman's terms - or at least in terms of the user's own field of expertise, which is to say in terms of the problem he/she is trying to solve, rather than in terms of the programming language being used to do it. We believe that AURAC's empirical, straightforward methods demonstrate that such aids are feasible. One obvious forward direction for research, therefore, is to investigate AURAC's potential in a richer and more widely applicable programming language. Since AURAC's language-specific knowledge is concentrated into a relatively small proportion of its routines, there would seem to be no obvious reason why its algorithms and methods should not be directly transferable to other languages - particularly to database-dependent language, such as PROLOG or the variants of LOGO.

Plans are in hand to transfer AURAC to a new implementation of PROLOG. It is hoped that this will result in an expansion of AURAC's power, especially where cliches are concerned. The ideas of attempting to describe a program entirely in terms of cliches will be re-examined.

The symbolic-evaluation debugging system of Eisenstadt and Laubsch (see chapter 1) currently fails because of combinatorial explosion of the number of possible paths through the Plan Diagrams it creates. Somewhat ad hoc, after-the-event rules are employed to reduce this number. However, at the level of equating the effect of the user's code to his/her (presumed) intentions it is far more flexible than AURAC, not being dependent upon a library of expected algorithms. Combining the two systems so that AURAC's algorithm-recognition process was replaced by Eisenstadt and Laubsch's symbolic evaluation method would form a very powerful system indeed, capable of handling programs well above (as well as at) the novice level. Such a combination seems entirely possible: although AURAC does not explicitly keep track of a current environment in Ruth's sense, equivalent information is implicitly held in AURAC's working memory. It should therefore be possible for AURAC to "hand on" unexplained sections of code to the Eisenstadt and Laubsch system for deeper analysis. The combination promises to be able to deal with the Sherlock Holmes project

(see chapter 6), which completely defeats the algorithmic approach for the reasons given above.

Cliches and canonical algorithms, as used by AURAC, are both methods of checking the correctness of conceptual "chunks" of code, and there are clear similaritites between the two. Indeed, it would be feasible to add a new cliche to the cliche library - consisting of the same four lines (but not, of course, precisely the same symbols) as those comprising the subtraction model from the algorithm library - and so to have a subtraction program recognised (and analysed) by AURAC's cliche module. The difference between the two methods is of course that a cliche is recognised in isolation, independently of the significance of any variables etc. included within it. The cliche module only checks that suitable "words" occur in the various parts of the cliche, i.e. that its structure is correct. The new subtraction program might be analysed as perfect in this respect, and yet still fail to subtract because the "subtracting cliche" within it was, say, passed the wrong data from some earlier part of the program. By analysing data flows, AURAC establishes the essential links between the input to the program, the various algorithmic steps within it, and the final result.

There is thus an analogy between AURAC and Schank's (1977) method of analysing text via "scripts". Both attempt to understand what is fed to them by using some pre-stored plan as a template against which to match, and hence to abstract the essential elements of, the input. Schank's scripts are analogous to the algorithm (or set of algorithm variants) selected by AURAC once it has established via the user the name of the project being attempted. Both may also fail when the input consists of a deliberately-chosen counter-example. Schank's answer to the problem of counter-examples is that these usefully point out suitable areas for further research, but do not detract from his basic approach. In this respect AURAC has a further practical advantage, in that its users are normally co-operative: their objectives are precisely what AURAC expects them to be (e.g. to write a working subtraction program).

Schank uses an internal representation (conceptual dependency) which, unlike that used by AURAC, is quite different from the surface text it represents. There is obviously a question here as to whether what is essentially raw SOLO code (in AURAC's libraries) is the best representation to use. Other research does not provide any firm answer, and we argue that because of SOLO's bare syntax the language is at least an effective representation of its

own operations. Eisenstadt and Laubsch, whose work has been mentioned several times in this thesis, use a purely symbolic representation of SOLO code, but have commented (personal communication) that at times it seemed that all they had achieved was a very sophisticated SOLO interpreter. Rich (1981) argues that LISP is the best possible form in which to represent LISP programs, and is also the best form in which to represent other languages such as FORTRAN. Adam and Laurent (1980) disagree, and their FORTRAN debugger represents idealised algorithms directly as FORTRAN code. Shapiro (1983) says that PROLOG code can be represented very conveniently in PROLOG. Our contention is that there is nothing to be gained from a re-representation of the raw code unless the new representation (a) contains more information than the raw code itself, or (b) is easier to operate upon or to analyse. Since the majority of AURAC's analyses are carried out via processes of recognition - of pattern matching - the raw code is the obvious choice as far as the second of these criteria is concerned. From the point of view of the amount of stored information, AURAC is a debugging mechanism, and hence needs to "know" very little about sections of the raw code which are not faulty. It creates an error frame (an internal representation of a fault condition) for every error it finds, and these also adequately represent any relevant information derived from non-error code-lines. For the future, the question of representation is interesting, but is not immediately

essential.

Finally, a part of AURAC's interest lies in the fact that it combines elements of psychology, of Artificial Intelligence, of software engineering and of Computer Aided Learning. AURAC would not achieve what it does without its empirical base in psychology - that is, without the account of human debugging behaviour described in chapter 1. The account was originally derived via introspection and by formal observation of SOLO experts, and was later supported by the experiments of chapter 3. The underlying investigation into the behaviour of expert human debuggers has led to a mechanism which is markedly different in its approach from that of any other currently implemented auto-debugger. And, as detailed in chapter 2, the design philosophy behind the whole MacSOLO/AURAC system is also firmly rooted in empirical experience - this time of the problems and difficulties of real novice programmers.

AURAC uses well-known AI techniques such as pattern matching and data-driven searching. In some cases - particularly in the two-stage matching which effects cliche recognition - these basic ideas have been extended in interesting ways. Its efforts to present programming concepts and descriptions of errors in terms suitable for non-programmers are relevant to software engineering. We

remain convinced that use of some of the latter's techniques - bit-mapped terminals, displayed menus, animated graphics - could benefit AURAC and its users enormously. And AURAC's whole conception and design was firmly within the context of (intelligent) Computer Aided Learning. As hinted above, this was very much a process of evolution, the growing system being repeatedly tried out on real representative users and modified if necessary in order to take account of their specific needs and problems. It is clear that machines intended to help novice programmers can benefit from knowledge drawn from all four of the above areas of research.

APPENDIX A

REFERENCES

Adam A. and Laurent J-P. (1980) "LAURA: A System to Debug Student Programs". Artificial Intelligence No.15, pp.75-122

duBoulay B., O'Shea T., and Monk J. "The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices" Int. Journal Man-Machine Studies, 14, pp 237-249.

duBoulay B. (1979) Unpublished Doctoral Dissertation, Dept. of Artificial Intelligence, University of Edinburgh, U.K.

Breuker J., (1981) Availability of Knowledge. COWO - publicatie 81-JB, Amsterdam.

Brown J.S. and Burton R.R. (1978) "Diagnostic Models for Procedural Bugs in Basic Mathematical Skills" Cognitive Science Vol.2 No.2 pp 155-192

Chapman D. (1981) "A Program Testing Assistant" MIT AI Memo No. 651.

Collins A.M. and Quillian M.R. (1969) "retrieval Time from Semantic Memory" Journal of Verbal Learning and Verbal Behaviour Vol.8 pp 240-7

Dijkstra E.W. (1972) "Notes on Structured Programming" in Structured Programming, by O.J. Dahl, E.W. Dijkstra and C.A.R. Hoare, Academic Press, New York, U.S.A.

Eisenstadt M. (1978) The Solo Primer, Units 3/4 of Cognitive Psychology: a Third Level Course. Open University, U.K.

Eisenstadt M. (1983) "Design Features of a Friendly

Software Environment for Novice Programmers". CACM 1983 (in press).

Eisenstadt M. and Lewis M.W., (1982) The Behaviour of Novice Programmers in a Friendly Software Environment. Tech. Report No.4., Human Cognition Research Laboratory, The Open University, U.K.

Eisenstadt M. and Laubsch J. (1980) "Towards an Automated Debugging Assistant for Novice Programmers" AISB-80.

Floyd R.W. (1967) "Assigning Meanings to Programs" Proc. Symposium on Applied Mathematics, AMS Vol 19.

Gawronski A. and Eisenstadt M. (1982) "Micro-SOLO: a Tool for Elementary AI Programming", SWURCC Microprocessor Software Quarterly No.7, South West Universities Regional Computer Centre, University of Bath, U.K.

Goldstein I.P. (1975) "Summary of MYCROFT: A Sytem for Understanding Simple Picture Programs" Artificial Intelligence 6, pp 249-288.

Goldstein I. and Miller M. (1976) "Structured Planning and Debugging: A Linguistic Theory of Design", MIT AI Laboratory Memo No. 387.

Hasemer T. (1982) MacSolo, Computer Assisted Learning Research Group Technical Report No. 24. Open University, U.K.

Hasemer T. (1983) "AURAC - A Debugging Tool for Novice Solo Programmers". In: New Technologies in Distance Education, Jones A., Scanlon E. and O'Shea T. (eds). Harvester Press, Sussex, U.K.

Hasemer T. (1983b) A Beginner's Guide to Lisp, Addison-Wesley, London. (in press).

Hoare C.A.R. (1969) "An Axiomatic Basis for Computer Programming" CACM Vol 12 No. 10.

Kahney H. (1983) An In-Depth Study of the Cognitive Behaviour of Novice Programmers, Human Cognition research Laboratory Tech. Report No.5, Open University, Milton Keynes, U.K.

Laubsch J. and Eisenstadt M. (1981) Domain Specific Debugging Aids for Novice Programmers, Computer Assisted Learning Group Tech Report No. 17, open University.

Lewis M (1980) Improving Solo's User Interface: an Empirical Study of User Behaviour and Proposals for Cost-effective Enhancements to Solo, Computer Asssisted Learning Research Group Technical Report No. 7. Open University, U.K.

Lukey F.J., (1980) "Understanding and Debugging Programs", Int. Journal of Man-Machine Studies, Vol.12., pp.189-202.

Maguire M. (1982) "Computer Recognition of Textual Keyboard Inputs from Naive Users", Behaviour and Information Technology, Vol.1, No.1, pp 93-111.

Miller M. and Goldstein I. (1977) "Structured Planning and Debugging", Proc. Fifth Int. Joint Conf. on Arificial Intelligence.

Mills H.D. (1971) "Top-Down Programming in Large Systems" in Debugging Techniques in Large Systems, R. Rustin (ed), Prentice-Hall, New Jersey, U.S.A. pp 41-45.

Muth F.E. and Tharp A.L. (1977) "Correcting Human Error in Alphanumeric Terminal Input", Information Processing and Management, 13, p.329.

Osterwell L.J. and Fosdick L.D. (1976) "Some Experiences with DAVE - A Fortran Program Analyser" AFIPS Conf. Proc.

Papert (1971) "Teaching Children to be Mathematicians versus Teaching About Mathematics", MIT AI Laboratory Memo 249.

Rich, C. (1981) Inspection methods in programming. Technical Report AI-TR-604. Cambridge, MA: MIT Artificial Intelligence Laboratory.

Rich C., Schrobe H.E., and Waters R.C., (1979) "Overview of the Programmer's Apprentice", Proc. Sixth Int. Joint Conf. on Artificial Intelligence, pp 827-8.

Rich C., Schrobe H.E., and Waters R.C. (1979b) "Computer Aided Evolutionary Design for Software Engineering", MIT AI Memo No. 506.

Ruth G.R. (1976) "Intelligent Program Analysis" Artificial Intelligence 7, pp 65-85.

Schank R.C. and Abelson R.P. (1977) Scripts, Plans, Goals and Understanding&, Lawrence Erlbaum Associates, New Jersey, U.S.A.

Schank R.C. (1973) "Identification of Conceptualizations Underlying Natural Language", in Computer Models of Thought and Language. Schank and Colby (eds), W.H. Freeman and Co., San Francisco, U.S.A.

Shapiro C. (1983) Algorithmic Program Debugging. MIT Press.

Shapiro D. (1981) "Sniffer: a System that Understands Bugs", MIT AI Memo No. 638.

Simmons R.F. (1973) "Semantic Networks: Their Computation and Use for Understanding English Sentences", in Computer Models of Thought and Language. Schank and Colby (eds), W.H. Freeman and Co., San Francisco, U.S.A.

Sleeman D.H. (1977) "A System Which Allows Students to Explore Algorithms", Proc. 5th. Int. Joint Conf. on Artificial Intelligence, pp780-786.

Waters R.C (1976) "A System for Analysing Mathematical FORTRAN Programs" MIT AI Memo No.368.

Waters R.C. (1979) "A Method for Analysing Loop Programs" IEEE Transactions on Software Engineering VOL SE-5 No. 3, pp 237-247.

Waters R.C. (1979b) "A Method for Automatically Analysing Programs" Proc. Sixth Int. Joint Conf. on Artificial Intelligence, pp 935-941.

Waters R.C. (1981) "The Programmer's Apprentice: Knowledge Based Program Editing" IEEE Transactions on Software Engineering, Vol.SE-8 No.1, pp.1-12.

Wertz H. (1979) "Automatic Program Debugging", Proc. Sixth Int. Joint Conf. on Artificial Intelligence, pp 951-3.

Wirth N. (1974) "Program Development by Stepwise Refinement" CACM 14 pp 221-227.

Young R.M. and O'Shea T. (1981) "Errors in Children's Subtraction" Cognitive Science, Vol.5, No.2.

APPENDIX B

TWO SUBTRACTION PROGRAMS

The first of the programs shown below is the demonstration program mentioned in chapter 4. It comprises sixteen separate procedures occupying over eighty lines of code as written here; NUMBERP, ASSERT, WORKON, ASSIGN and DEVALUATE will be called several times each, depending upon whether or not borrowing is required. WORKON is called to do the subtraction of each column, and so its line 70 doubles as both line 1 and line 4 (arrowed) of the algorithm; during analysis, this line is encountered, and the user is queried about it, twice.

```
TO TOPNUM /X/ /Y/

5 CHECK NUMBERP HAS FAILED
 A If Present: FORGET NUMBERP HAS FAILED ; CONTINUE
 B If Absent : CONTINUE
10 NUMBERP /X/ /Y/
20 CHECK NUMBERP HAS FAILD
  A If Present: RETRY TOPNUM ; EXIT
  B If Absent : CONTINUE
30 ASSERT TT IS /X/
40 ASSERT TO IS /Y/


TO BOTTOMNUM /X/ /Y/

5 CHECK NUMBERP HAS FAILED
 A If Present: FORGET NUMBERP HAS FAILED ; CONTINUE
```

```
 B If Absent : CONTINUE
10 NUMBERP /X/ /Y/
20 CHECK NUMBERP HAS FAILD
  A If Present: RETRY TOPNUM ; EXIT
  B If Absent : CONTINUE
40 ASSERT BO IS /Y/


TO ASSERT /X/ /Y/ /Z/

10 CHECK /X/ /Y/ ?
  A If Present: FORGET /X/ /Y/ * ; CONTINUE
  B If Absent : CONTINUE
20 NOTE /X/ /Y/ /Z/


TO SUBTRACT

10 CHECK NUMBERP HAS FAILED
  A If Present: FORGET NUMBERP HAS FAILED ; CONTINUE
  B If Absent : CONTINUE
15 CHECK SAYNUMBERS HAS FAILED
  A If Present: FORGET SAYNUMBERS HAS FAILED ; CONTINUE
  B If Absent : CONTINUE
20 SAYNUMBERS
30 CHECK SAYNUMBERS HAS FAILED
  A If Present: EXIT
  B If Absent : CONTINUE
60 WORKON ONES NOW
70 CHECK AO IS ?F
  A If Present: WORKON TENS NOW ; CONTINUE
  B If Absent : FOULPLAY 70B SUBTRACT ; EXIT
80 CHECK AT IS ?E
  A If Present: SUCCESS *E *F ; EXIT
  B If Absent : FOULPLAY 80B SUBTRACT ; EXIT


TO SAYNUMBERS

10 CHECK TT IA ?A
  A If Present: CONTINUE
  B If Absent : PROTEST 10B SAYNUMBERS ; EXIT
20 CHECK TO IS ?B
  A If Present: CONTINUE
  B If Absent : PROTEST 20B SAYNUMBERS ; EXIT
30 CHECK BT IS ?C
  A If Present: CONTINUE
  B If Absent : PROTEST 30B SAYNUMBERS ; EXIT
40 CHECK BO IS ?D
  A If Present: CONTINUE
  B If Absent : PROTEST 40B SAYNUMBERS ; EXIT
50 PRINT "OK, YOU'VE ASKED ME TO SUBTRACT" *C *D "FROM" *A *B
60 NUMBERP *A *B
```

```
70 NUMBERP *C *D
80 CHECK NUMBERP HAS FAILED
  A If Present: ASSERT SAYNUMBERS HAD FAILED ; EXIT
  B If Absent : CONTINUE


TO NUMBERP /X/ /Y/

10 CHECK /X/ PLUS10 ?
  A If Present: CONTINUE
  B If Absent : ASSERT NUMBERP HAS FAILED ; CONTINUE
20 CHECK /Y/ PLUS10 ?
  A If Present: CONTINUE
  B If Absent : ASSERT NUMBERP HAS FAILED ; CONTINUE
30 CHECK NUMBERP HAS FAILED
  A If Present: CONTINUE
  B If Absent : EXIT
40 PRINT "UH OH!  YOU MUST RESTRICT YOURSELF"
50 PRINT "TO DIGITS ONLY (0,1,2,3,4,5,6,7,8,9)"
60 PRINT "EITHER YOU MADE A TYPING ERROR"
70 PRINT "OR ELSE YOU'RE TRYING TO TRICK ME."


TO WORKON /COL/ /TIME/

10 PRINT "I'M" /TIME/ "WORKING ON THE" /COL/ "COLUMN..."
20 CHECK /COL/ TOP ?T
  A If Present: CONTINUE
  B If Absent : ERROR 20B WORKON ; EXIT
30 CHECK /COL/ BOTTOM ?B
  A If Present: CONTINUE
  B If Absent : ERROR 30B WORKON
40 CHECK /COL/ ANS ?A
  A If Present: CONTINUE
  B If Absent : ERROR 40B WORKON ; EXIT
50 CHECK *T IS ?X
  A If Present: CONTINUE
  B If Absent : ERROR 50B WORKON ; EXIT
60 CHECK *B IS ?Y
  A If Present: PRINT "NOW, DO I KNOW WHAT" *X "MINUS" *Y
                      "IS?" ; CONTINUE
  B If Absent : ERROR 60B WORKON ; EXIT
70 CHECK *X *Y ?R                              <--1,4
  A If Present: ASSIGN *R *A ; EXIT
  B If Absent : PRINT "NO, I'LL HAVE TO BORROW 1." ; CONTINUE
80 CHECK /COL/ NEXTCOL ?N
  A If Present: BORROWFROM *T *N ; CONTINUE
  B If Absent : NEGNUMBER ; EXIT
90 CHECK BORROWFROM HAS FAILED
  A If Present: FOULPLAY 90A WORKON ; EXIT
  B If Absent : WORKON /COL/ /STILL/ ; EXIT
```

```
TO BORROWFROM /PLACE/ /COL/

5 PRINT "HMMM...LOOKING AT THE" /COL/ "COLUMN..."
10 CHECK /COL/ TOP ?T
  A If Present: CONTINUE
  B If Absent : ERROR 10 BORROWFROM ; EXIT
20 CHECK *T IS ?L
  A If Present: CONTINUE
  B If Absent : ERROR 20 BORROWFROM ; EXIT
30 CHECK /PLACE/ IS ?R
  A If Present: CONTINUE
  B If Absent : ERROR 30B BORROWFROM ; EXIT
40 CHECK *L 1 ?X                                <--3
  A If Present: PRINT "BORROWING 1 FROM" *L "LEAVES" *X
                      "THERE, AND" ; CONTINUE
  B If Present: If Absent : ERROR 40B BORROWFROM ; EXIT
50 CHECK *R PLUS10 ?P                           <--4
  A If Present: CONTINUE
  B If Absent : ERROR 50B BORROWFROM ; EXIT
60 ASSERT /PLACE/ IS *P
70 ASSERT *T IS *X


TO ASSIGN /VALUE/ /VARIABLE/

10 PRINT "YES, IT'S" /VALUE/
20 ASSERT /VARIABLE/ IS /VALUE/


TO SUCCESS /FIRSTNUM/ /SECONDNUM/

10 PRINT "FINISHED.  THE ANSWER IS:" /FIRSTNUM/ /SECONDNUM/
20 DEVALUATE TT TO BT
30 DEVALUATE BO AO AT


TO DEVALUTE /X/ /Y/ /Z/

10 FORGET /X/ IS ?
20 FORGET /Y/ IS ?
30 FORGET /Z/ IS ?


TO RETRY /PROCNAME/

10 PRINT "TRY" /PROCNAME/ 'AGAIN, BUT THIS TIME"
20 PRINT "USE TWO DIGITS, E.G." /PROCNAME/ "2 7"
30 FORGET NUMBERP HAS FAILED


TO FOULPLAY /STEPNUM/ /PROC/

10 PRINT "Hey, wait a minute...I think you've"
```

```
20 PRINT "given me a problem which has a funny"
30 PRINT "solution (e.g. a negative number), which"
40 PRINT "I can't handle.  Remember, I'm very dimwitted,"
50 PRINT "so start again with TOPNUM # # and BOTTOMNUM # #."
60 PRINT "(trapped at step" /STEPNUM/ "of" /PROC/ ")"


TO PROTEST /STEPNUM/ /PROC/

10 PRINT "I'm afraid you haven't given me TOPNUM and"
20 PRINT "BOTTOMNUM properly.  Be sure to"
30 PRINT "leave a space between each of the numbers, e.g.:"
40 PRINT "     TOPNUM 4 7"
50 PRINT "     BOTTOMNUM 3 9"
60 PRINT "(trapped at step" /STEPNUM/ "of" /PROC/ ")"
70 ASSERT /PROC/ HAS FAILED


TO ERROR /NUM/ /NAME/

10 PRINT " ==> UH OH, GOOFED AT STEP" /NUM/ "OF" /NAME/


TO NEGNUMBER

10 PRINT "BUT THERE IS NO NEXT COLUMN!!  WELL,"
20 PRINT "YOU MUST HAVE GIVEN ME A PROBLEM WHICH"
30 PRINT "HAS A NEGATIVE NUMBER AS THE SOLUTION."
40 PRINT "I CAN'T COPE WITH THOSE, BUT PERHAPS YOU"
50 PRINT "CAN DEFINE A BETTER 'SUBTRACT' PROCEDURE WHICH CAN."
```

The second program was written by a Summer School student. Rather than using TOPNUM and BOTTOMNUM to enter the problem itself into the database, the programmer has used the SOLO procedure INPUT to allow these values to be inserted directly into a running program, where they are held as the bound values of variables. The algorithm lines are again arrowed, on the assumption that borrowing will occur.

```
TO SUBTRACT
```

```
10 CHECK FLAG IS UP
  A If Present: FORGET FLAG IS UP ; CONTINUE
  B If Absent : CONTINUE
20 PRINT "CAN YOU GIVE ME THE FIRST DIGIT OF THE TOP NUMBER"
30 INPUT *I
40 PRINT *I
50 PRINT "WHAT IS THE SECOND DIGIT OF THE TOP NUMBER"
60 INPUT *J
70 PRINT *J
80 PRINT "NOW I NEED THE FIRST DIGIT OF THE BOTTOM NUMBER"
90 INPUT *K
100 PRINT *K
110 PRINT "WHAT IS THE SECOND DIGIT OF THE BOTTOM NUMBER"
120 INPUT *L
130 PRINT *L
140 ZEROBUG *I *J *K *L
150 CHECK FLAG IS UP
  A If Present: EXIT
  B If Absent : CONTINUE
160 FOOLCHECK *I *K
170 CHECK FLAG IS UP
  A If Present: EXIT
  B If Absent : CONTINUE
180 BORROW *I *J *K *L
190 CHECK FLAG IS UP
  A If Present: EXIT
  B If Absent : CONTINUE
200 LET *Y = (*J'S *L)
210 LET *X = (*I'S *K)
220 PRINT *X *Y


TO ZEROBUG /X/ /Y/ /Z/ /A/

10 TEST /Y/ = /Z/
  A If Yes: CONTINUE
  B If No : EXIT
20 CHECK /A/ /Y/ ?
  A If Present: ZEROBUG1 /Y/ /A/ ; CONTINUE
  B If Absent : EXIT
30 NOTE FLAG IS UP


TO ZEROBUG1 /Y/ /A/

10 TEST /Y/ = /A/
  A If Yes: PRINT "THE ANSWER IS 0!!!" ; EXIT
  B If No : PRINT "THIS NUMBER IS NEGATIVE." ; EXIT


TO FOOLCHECK /Y/ /A/

10 CHECK /A/ /Y/ ?
```

```
  A If Present: CONTINUE
  B If Absent : EXIT
20 NOTE FLAG IS UP
30 PRINT "I REALLY WOULD LIKE TO HELP ETC."


TO BORROW /X/ /Y/ /Z/ /A/

10 PRINT "TT = " /X/ " TO = "/Y/ " BT = " /Z/ " BO = " /A/
20 CHECK /Y/ /A/ ?
  A If Present: PRINT "THIS IS AN EASY ONE!!" ; EXIT
  B If Absent : PRINT "HMM!! I'VE GOT TO BORROW HERE!" ; CONTINUE
30 NOTE FLAG IS UP
40 LET *L = (/Y/'S PLUS10)          <--2
50 LET *B = (/X/'S 1)               <--3
60 LET *X = (*L'S /A/)              <--1
70 LET *Y = (*B'S /Z/)              <--4
80 PRINT *Y *X
```

APPENDIX C

EXERPT FROM EXPERIMENTAL SUBJECT NOTES

Introduction.

This booklet assumes that you have worked through the D303 Course notes (Block 1, Units 3 and 4) concerning SOLO, and have a fair understanding of the following:

1) The computer plus a properly-designed program can together form a dynamic model of some process or other - in our case a model of some human cognitive ability. The model is of course based upon a theory of how the process itself works, and its usefulness lies in the fact that the success or otherwise of running the model provides evidence for or against the validity of the theory.

2) The computer by itself knows NOTHING other than how to "do" SOLO. All the extra knowledge it needs in order to become a model of your theory must be supplied by you. This knowledge can be provided either as static data in a "database", or as procedural instructions in a suite of "procedures" referred to as a "program", or - more usually - as both. In the latter case the program operates (in accordance with your theory) upon the static data so as to produce (predictable) effects.

3) SOLO is a programming language especially designed to help you rapidly to acquire experience of points (1) and (2) in action. It consists of a set of inbuilt procedures which allow you to create and to change a database, and to create and to change procedures of your own. You should by now have a good idea of how to do these things, in particular of how to use the conditonal procedure CHECK, and of how "variables" are given "values" when CHECK is used with a "wildcard".

```
**********************************************************
*                                                        *
*  If ANY of the above isn't clear to you, please ASK -  *
*  otherwise you could waste a great deal of your time.  *
*                                                        *
**********************************************************
```

So, you now feel yourself to be reasonably competent to write working SOLO procedures, and you will understand the cognitive ability which you have chosen to model: perhaps memory-retrieval according to the theory of Collins and Quillian, or schema matching, or two-column subtraction, or the detective skills of Sherlock Holmes. At this point many students are totally bemused as to how to join these two areas of understanding together: how DOES one select from all the possible permutations of NOTE, CHECK, FOR EACH etc. in order to make SOLO behave as a model of anything? The purpose of this booklet is to help you over that hurdle. By the end of it, you'll have the kernel procedures for your project written and working, and can then go on to write for yourself the more interesting additions and variations for which you'll get higher marks.

Algorithms - a How To section.

The word "algorithm" is a piece of programming jargon, but for once it is a useful piece, rather than (as so often) merely a new word to denote something for which a perfectly adequate word exists already. An algorithm is a DESCRIPTION of what a program is to do in order to achieve its (your) goal, and is very much like a recipe, or a section from a car-maintenance manual. That is, it is a set of instructions written in more or less everyday English. The difference is that the individual instructions in an algorithm are instructions which the COMPUTER, rather than a human being, is capable of carrying out; and since the computer is so remorselessly logical the instructions have to follow one another in a a logical and necessary sequence - necessary in the sense that, if carried out in the order given, they INEVITABLY achieve whatever is the goal of the corresponding program. A good algorithm will represent a sequence of actions by the computer which achieve the goal neatly and cleanly, with no waste of computing time.

So, in order to write the algorithm which the computer can follow to achieve two-column subtraction, we first need to decide what are the essential steps in the subtraction process. A handy way of doing this is to introspect: to decide how WE do it. Once we've managed to write those steps down in English, we can go through them pruning out any strictly unnecessary bits, and making sure that they are in the "inevitable" order mentioned above. Finally, we have to make sure that each remaining instruction is one which SOLO is capable of. At that point we'll have our algorithm written, and writing the corresponding SOLO code should be a doddle.

```
**********************************************************
*                                                        *
*        TRY TO WRITE DOWN THE STEPS REQUIRED FOR        *
*        TWO-COLUMN SUBTRACTION BEFORE READING ON        *
*                                                        *
**********************************************************
```

There are several ways of handling the "borrowing and paying back" when the bottom units digit of the subtraction sum is larger than the top units digit. For example, if the sum is like this:

```
  4  3 -
  2  7
  ----
```

Therefore, the following description of how it is done may not be exactly the same as the way you were taught at school. Please look at it carefully, and make sure you're happy that it achieves the same effect as the description you have just written down:

1) Try to take the bottom units digit from the top units digit without borrowing: If that is possible, go to step (2); If not, go to step (3).

2) Subtract the digits in the units column. Go to (4).

3) Borrow 10 from the top tens digit and add it to the top units digit. Go to (2) with this new top units value.

4) If no borrowing was necessary, go to (5). If borrowing was necessary, go to (6).

5) Subtract the digits in the tens column. Go to (7).

6) Pay back the borrowed 10 by subtracting 1 from the top tens digit. Go to (5) with this new top tens value.

7) Write down the result of steps (5) and (2) in that order.

I hope you'll agree that if you follow those seven steps you must inevitably get the right answer to any two-column subtraction sum, provided that no negative numbers are involved (handling negative numbers is one of the refinements you can try to add for yourself later). The arrows at the side, however, show at a glance how tortuous the procedure is. Another clue which tells us that as an algorithm this description is a mess is the fact that steps (1) and (4) actually achieve nothing at all - they just tells us which step to move on to next. We can simplify the description (and shorten it) quite a bit just by rearranging some of its lines: specifically by doing the "paying back" immediately after the "borrowing":

1) Try to subtract the digits in the units column without borrowing: If SUCCESSFUL, go to (4); If UNSUCCESSFUL, go to (2).

2) Borrow by adding 10 to the top units digit. Go to (3), taking this new value with you.

3) Pay back by subtracting 1 from the top tens digit. Go to (1) with this new value and the new value from (3).

4) Subtract the digits in the tens column. Go to (5).

5) Write down the results of steps (4) and (1) in that order.

Notice that step (1) now does the test ("can I subtract the units column as it stands?") AND actually does the subtraction of the units column, either immediately or after the borrowing process embodied in steps (2) and (3). If step (3) is executed, step (1) will be RE-executed, and step (4) will be executed for the first time, using the new numbers at the top of each column. That is to say, our emergent algorithm does this:

```
4  3 -            3  13 -
2  7      ==>     2   7
----              -----
```

To put it another way, it tries to subtract the units digits as they stand, and if that is impossible it changes the digits at the top of both columns and tries again. The second time around, it must (inevitably, because of the borrowing rules) succeed and so can move on to subtract the tens digits.

Right. That's about as good an algorithm as we can get without considering the other essential criterion: the one which says that each and every step comprising the algorithm must represent an action which SOLO is capable of carrying out. Only when that criterion is satisfied will we be able to say that our algorithm is written, and we can go on to write the code. And here we have a problem straight away. If you type into SOLO something like:

```
3 MINUS 2
```

SOLO will reply, in its little electronic way, with the equivalent of "Eh? I don't understand you." because, as I hope you know, the first word in any legal SOLO sentence MUST be the name of a procedure, and SOLO has no inbuilt procedure called "3". Similarly if you type

```
SUBTRACT 3 2
```

you'll get a similar error message from SOLO. In fact, if you type HELP at this stage to see why you got an error message, SOLO will probably suggest that you ought to write the missing procedure "SUBTRACT", which is exactly where we came in. As you probably realised at the start, SOLO knows nothing about arithmetic at all.

So the problem becomes one of how we are to give SOLO a knowledge of the arithmetical difference between 3 and 2 - and, of course, of the arithmetical differences between quite a lot of other pairs of digits as well. Remember what I said in paragraph (2) of the Introduction: SOLO can be given knowledge in either of two forms; as static data or as procedural instructions. What kind of knowledge are we trying to give it now? Is it the knowledge of HOW TO subtract 2 from 3, or knowledge of the FACT that the difference between 3 and 2 is 1?

There is no hard and fast rule here. You, as the programmer, are perfectly entitled to make whatever arbitrary decisions you like as to what knowledge should be represented procedurally and what knowledge should be represented as static data. And the way to make such decisions is to look at the problem (the model of subtraction, in this case) to see what knowledge is needed, and then to look at the tools you have available in whatever programming language you're using (SOLO, of course) to see what methods of representation are most convenient. "Convenient" here can mean that the representations make intuitive sense and so are easy for human beings to understand, or it can simply mean that they lead to the least possible effort for you - programming is probably the

only conscious activity in which laziness can be thought of as creditable.

Now, remember what we're trying to do. We want to write a program which will generate the correct answer to any two-column subtraction problem. It would be perfectly feasible (though I wouldn't advise it) to write a separate procedure for every one of the possible permutations of four digits, and then to run them all on any given problem until one or other of them produced the answer. This would be the way to go about things if you had decided that ALL of the necessary subtraction knowledge should be represented procedurally. It would need something approaching 10,000 procedures, and no programmer in his or her right mind would take on such a huge project (just think how long it would take to find all the typing mistakes). Conversely, it would be equally feasible and equally boring to represent every possible subtraction problem, together with its answer, as a database triple, like this:

```
NOTE 43--27-->16
```

and then to have a single CHECK line:

```
CHECK /X/ /Y/ ?ANS
```

which would dig any required answer out of your gigantic database.

No, both of the above approaches are ludicrously cumbersome, besides being very boring for you and very boring for anyone who reads your work (including, of course, your examiners!). In other words, what we need to do is represent SOME of the knowledge about subtraction procedurally, and SOME of it as data; so as to minimise effort.

Now let's look at SOLO to see what tools we have available. As I've already mentioned, SOLO has no inbuilt procedures to handle arithmetic, so we're going to have to write whatever PROCEDURES we need. On the other hand, it does have inbuilt facilities (NOTE and FORGET) which give us control over what data is to be found in the database, and it has CHECK which enables us to inspect individual database triples. And here SOLO's total ignorance about numbers can be an advantage. You know and I know that the symbol "5" represents a very different KIND of thing from, say, the symbol "FIDO" (one is a number and one is a word), but SOLO knows nothing of the kind, and will be quite happy to accept numbers in its triples as easily as it does words. Remember

also the significance of the arrows on data printouts from SOLO:

FIDO--BROTHER-->ROVER

implies a ONE-WAY relationship. That is, although SOLO knows from that triple that Fido's brother is called Rover, it cannot make the reverse inference and so discover that Rover has a brother called Fido.

Therefore, we can easily put into the database triple which will UNIQUELY express basic data about subtraction. For example, the information that 8 minus 5 is 3 can be written simply as

8--5-->3

without any danger of SOLO getting confused into thinking that 5 minus 3 is 8.

Now let's try the above-mentioned introspection - apart from any other considerations, we have a prime motive for doing so in the fact that we're trying to model a HUMAN cognitive ability, so what could be more sensible than using our own abilities as the basis of the model?

Clearly, we neither work out every possible subtraction problem from first principles, nor do we habitually remember the answers to all 10,000 of them. Perhaps you'll agree, however, that most people don't need to WORK OUT things like the difference between 8 and 5 - they KNOW it, straight away, because they learned their subtraction tables at school. And, given that (static) knowledge of all possible single-column subtraction problems (only 99 of those), we are able to extrapolate via something very like the above algorithm to achieve two, three, four or N column subtraction.

So what I suggest is that we give SOLO very much the same kind of division between static and procedural knowledge. We'll have a database whose triples represent any single-column subtraction problem. Triples like this one:

8--5-->3

There will be similar triples to represent 8 minus any other digit which is equal to or less than 8 itself; and similar sets of triples again to represent subtraction from any of the other digits between 0 and 9.

But unfortunately that's not all we need. Remember that, when the sum requires borrowing, our algorithm generates a new sum something like

```
3  13 -
2   7
-----
```

so we're going in fact to need sets of triples which represent subtraction of any single-digit number from any two digit number up as far as 18 (it is only 18, and not 19, if you think about it). For the borrowing itself, we're going to need triples which tell us what the result is of adding 10 to any single-digit number between 0 and 8. But - our first piece of luck - we'll be able to effect the paying back by using triples such as

```
8--1-->7
```

which already exist.

You may be thinking that even so that sounds like an awful lot of typing, just to create the database. And indeed it is. But don't worry: that's already been done for you. (And is the reason why I was so determined that you should accept one particular algorithm rather than any other). Type, for example,

```
DESCRIBE 5
```

and SOLO will reply with a picture of the triples associated with the node "5" in the usual format:

```
5--5-->0
'
'--4-->1
'
'--3-->2
'
'--2-->3
'
'--1-->4
'
'--0-->
'
'--plus10-->15
```

Get SOLO to DESCRIBE a few other numbers, just to convince yourself that your database is as I said it should be. From this I hope you can see that SOLO has been given the same STATIC data as you and I would use in solving a two-column subtraction problem. The rest of the required knowledge will be given to SOLO in procedural form. Look again at the final version of our algorithm, and remember that it was also written on the basis of how WE do subtraction. You'll see that the algorithm could not possibly work unless that same static data were around for it to work with. Step (1) would always fail, so that the algorithm would try to borrow when it shouldn't; step (2) wouldn't be able to add 10 to anything because it wouldn't know any of the results; and so on. In other words, the algorithm is a description of the PROCEDURAL knowledge required to solve our subtraction problems, and the database holds the STATIC knowledge on which the procedures will operate.

The final stage in the construction of a good algorithm is making sure that each step in it is an instruction which can be carried out by the programming language you're using.

With experience, one can see that at a glance - in fact an experienced programmer has that consideration at the back of her/his mind throughout the writing of any algorithm, and it was certainly at the back of mine as I wrote the algorithm above. I hope you'll be willing to take it on trust that steps (1) to (5) in our algorithm CAN be tramslated directly into working SOLO code. Sections 1 to 4 which follow will show you how this is done. When you reach the end of the Sections, you'll have a working SUBTRACT program, which you'll be able to use by typing for example

SUBTRACT 2 7 from 4 3

in the certainty that SOLO will dutifully reply

16

Stage 1.

Now then. Given the database you've just been looking at, your objective in this section is to write a procedure which will correctly subtract ANY two single-digit numbers, provided that the answer is not negative. In your final program, the code which you will have written here will be the part which subtracts the digits in the units column - i.e. it will be the translation into SOLO code of the major

part of step (1) of the algorithm.

Remember the following: SOLO's database now contains a large number of triples such as

8--5-->3

and SOLO itself has a CHECK procedure with which you could do something like

CHECK 8 5 ?ANSU

(In the particular version of SOLO you're using now, you're allowed to put any letters you like after the question-mark in order to create a variable-name. "ANSU" has been chosen to be mnemonic: to remind you that *ANSU will hold the answer to the subtraction of the units column).

```
**********************************************************
*                                                        *
* CAN YOU WRITE A PROCEDURE CALLED "SUBUNITS" SUCH THAT  *
* YOU CAN TYPE                                           *
*                                                        *
*                  SUBUNITS 8 5                          *
*                                                        *
*    (OR ANY OTHER TWO SINGLE-DIGIT NUMBERS WHERE        *
*    THE FIRST IS LARGER THAN THE SECOND) AND            *
*    SUCH THAT "SUBUNITS" WILL PRINT THE RESULT OF       *
*        SUBTRACTING THOSE TWO NUMBERS?                  *
*                                                        *
**********************************************************
```

If you can, turn to page . If you can't, read on.

Do you remember how to write procedures in SOLO? You type something like

TO SUBUNITS /TU/ /BU/

which puts SOLO into a special mode (called EDIT mode). Anything you type between now and the moment when you type DONE becomes a part of the procedure called "SUBUNITS". You can tell that SOLO is in EDIT mode because you get a hash-sign (#) amongst other things in place of the usual "SOLO:" prompt.

The two parameters /TU/ and /BU/ in the TO-line above are like variables in the sense that they "hold" values. When you have written SUBUNITS and come to try it out, you'll type something like

```
SUBUNITS 8 5
```

and from then on any occurrence WITHIN the procedure SUBUNITS of /TU/ will be understood by SOLO to mean "8", and any occurrence of /BU/ will be under- stood to mean "5". Similarly, if you run SUBUNITS by typing

```
SUBUNITS 9 2
```

then WITHIN the procedure SOLO will understand /TU/ to mean "9" and /BU/ to mean "2".

```
************************************************************
*                                                          *
*              CAN YOU WRITE "SUBUNITS" NOW?               *
*                                                          *
************************************************************
```

If you can, turn to page . If not, read on.

OK. You know about the CHECK procedure which can be used like this

```
CHECK 8 5 ?ANSU
```

to retrieve from the database the third member of whatever triple matches the pattern:

```
8--5-->....
```

Remember also that CHECK creates a variable whose name is the same as that of the wildcard (?ANSU) but which begins with a star (*ANSU) rather than a question-mark. This variable holds whatever was retrieved from the database by the CHECK itself. So you could put into SUBUNITS something like

```
10 CHECK 8 5 ?ANSU
  A If Present: PRINT "ANSU IS" *ANSU; CONTINUE
  B If Absent : EXIT
```

However, WITHIN A PROCEDURE you aren't obliged to write the specific NUMBERS after the word "CHECK". Instead, you have the facility of using the PARAMETERS /TU/ and /BU/, which as

mentioned above will later take on any value you care to give them when you run the procedure.

```
**********************************************************
*                                                        *
*                   WRITE "SUBUNITS"                     *
*                                                        *
**********************************************************
```

Debugging - Stage 1.

The word "debugging" is another bit of programming slang.

Errors and mistakes in written programs are known as "bugs", and the process of finding and rectifying them is known as "debugging".

I'm assuming that at this stage you have what you believe to be a working procedure to subtract any two single-digit numbers. If not, please ASK.

You may have heard or read that SOLO is more than just a programming LANGUAGE, it is a programming ENVIRONMENT. You're about to find out what that means. You've already met some of the many error-messages that SOLO can generate, and I hope that from time to time you've used the HELP system. Both of these are parts of the environment. Now it's time to introduce you to another useful facility: the STEPPER. The stepper allows you to watch, line by line, as your program is executed by SOLO. This is true whether your "program" is a simple one- or two-line procedure, as yours should be at the moment, or a huge suite of procedures and subprocedures. The way to run the stepper is to type, for example:

```
STEP SUBUNITS 8 5
```

that is, you type "STEP" and then, on the same line, whatever you would type to run your procedure in the usual way. SOLO will reply:

```
Enter SUBUNITS 8 5
```

which means that the stepper has begun execution of SUBUNITS with the parameters you specified. From now on, you can move from one line of SUBUNITS to the next by pressing the RETURN key. A correct SUBUNITS will give results similar to

this:

<RETURN>

```
10 CHECK 8 5 ?ANSU
  A If Present: CONTINUE
  B If Absent : EXIT
```

```
Present ... *ANSU = 3                <RETURN>
```

```
20 PRINT "ANSU IS" *ANSU
```

```
ANSU IS 3                            <RETURN>
```

```
Exit SUBUNITS
```

```
SOLO:
```

You may have chosen to put the PRINT statement of Line 20 on line 10A rather than on aline of its own, in place of the CONTINUE above. That's perfectly OK, of course (in fact, is a bit neater), in which case the stepper's printout will look like this:

<RETURN>

```
10 CHECK 8 5 ?ANSU
  A If Present: PRINT "ANSU IS" *ANSU; EXIT
  B If Absent : EXIT
```

```
Present ... *ANSU = 3
```

```
ANSU IS 3                            <RETURN>
```

```
Exit SUBUNITS
```

```
SOLO:
```

If you find any bugs while using the stepper - in particular you should watch out for control-statements (EXIT or CONTINUE) which send SOLO in the wrong direction after a CHECK line - use the SOLO editor to put them right. To do that, type

```
EDIT SUBUNITS
```

and SOLO will reply

```
EDIT = SUBUNITS  (to show you that you're editing the right
procedure)
```

and

edit line...#

Whenever, during editing, you see that last prompt, you can if you like type

SHOW

to have the CURRENT version of your procedure printed out for you. The editor will also prompt you (rather determinedly) for any sublines following a CHECK, should you happen to delete them or anything. To delete a line (including its sublines if any), type the number of the line to the edit prompt, and follow it with a RETURN. To change a line, type its number followed by the new version of the line itself.

YOU MUST ALWAYS GIVE A LINE-NUMBER TO THE EDIT PROMPT

(except when you type HELP or SHOW). Otherwise, you'll get a complaint to that effect from SOLO. When you've finished editing, type
DONE

OK. One final check, to show you that everything really is alright. Run your SUBUNITS. It should of course print out the correct ANSU. Now type

DEBUG

This brings in another part of the SOLO environment, the debugger. The debugger is able to look at quite a number of aspects of programs, and to point out the possible sources of errors. In particular, it can say whether of not a program contains the code necessary to embody each step of the algorithm. And in our case, when the SUBTRACT program is completely written and correct, the debugger will say simply "Your program will subtract". In order to do this, the debugger has to ask you, the programmer, the occsaional question, and of course the first thing it needs to ge told is which of the many algorithms it knows about is the one you're working on. So the first thing you get in response to "DEBUG" is

Name of Project:

to which you type, on the same line of course,

SUBTRACT

Then the debugger will set off doing its stuff, which takes a minute or two. If its asks you any more questions, answer them as best you can. When it has finished, and if your SUBUNITS is correct, the debugger should say:

Your program has no line which adds ten to the top units column.
Your program has no line which subtracts 1 from the top tens column.
Your program has no line which subtracts the digits in the tens column.

Which is quite true, of course - you haven't written those parts yet. What the debugger should NOT say is

Your program has no line which subtracts the digits in the units column.

If it does and you can't see why it should, or if it says anything else which you don't understand, please ASK.

APPENDIX D

SELECTION FROM AURAC CODE

Included here are sample LISP functions from the three modules of AURAC. The first is NOTE-ERROR, which creates the actual error frames each time AURAC's production rules detect a Higher-Level Syntactic error in the user's code. Notice the inefficiency of using strings to denote the various error types; there would be no objection to using, say, mnemonic atoms instead in a future implementation.

```
(defun note-error (type codeflag)   ;create an error frame
  (let ((error-name (symbolconc 'error (setq n (1+ n))))
        (etype (exploden type))
        (var nil)
        (cause nil))
    (push error-name errorlist)   ;remember the error.
                                  ;if the line may contain
                                  ;an unbound variable,
                                  ;note this fact in the
                                  ;PS's working memory.
    (and
      (not (or (equal etype (exploden '|CHECK succeeds|))
               (equal etype
                      (exploden '|CHECK always succeeds|))))
      (wm-augment (cons (genname (ps-is? (headline)))
                        error-name)
                  'errvars))
                                  ;some slots will be
                                  ;empty
    (putprop error-name t 'announce)
    (putprop error-name nil 'symbol)
    (putprop error-name nil 'word)
    (putprop error-name nil 'altlines)
```

```
  (putprop error-name nil 'cliche)
                                ;PS's push-down list
                                ;gives depth of
                                ;recursion.
  (putprop error-name (1+ (length psi-pdl)) 'level)
  (cond
    ((equal (exploden type)
            (exploden '|uses up more than twenty LEVELS.|))
     (note-recursion error-name))
    (t (putprop error-name nil 'altnodes)
       (putprop error-name nil 'altprocs)
       (putprop error-name nil 'recursion)))
                                ;current args to PS
                                ;interpreter give the
                                ;current procedure-name.
  (putprop error-name (car psi-call) 'procedure)
                                ;line-number is global.
  (putprop error-name line-number 'line)
                                ;HEADLINE retrieves
                                ;current line from WM.
  (putprop error-name (headline) 'code)
                                ;AURAC's evaluator can
                                ;evaluate it.
  (putprop error-name (ps-eval (headline)) 'evaluated)
  (putprop error-name codeflag 'unreached)
  (putprop error-name type 'type)
                                ;if the line contains an
                                ;unbound variable, find
                                ;it via HEADLINE and then
                                ;look in the ERRVARS slot
                                ;of WM to see which error
                                ;frame notes its creation.
                                ;That frame and the current
                                ;one can then be chained.
  (putprop
    error-name
    (and (equal etype
                (exploden '|contains an unbound variable.|))
         (setq var (*unbound (headline)))
         (setq cause (cdr (assoc var (errvarsp))))
         (member (get cause 'procedure)
                 (mapcar 'caar psi-pdl))
         cause)
    'cause)
                                ;If the run-time error
                                ;found by AURAC and that
                                ;found by MacSOLO are the
                                ;same, note the fact.
(cond ((and rte (equal rte run-time-error))
         (putprop error-name '(rte) 'effects))
                                ;If not, note the one
                                ;found by AURAC.
       (run-time-error
```

```
          (putprop error-name '(run-time-error) 'effects))
                                ;Otherwise, no run-time
                                ;error.
        (t (putprop error-name nil 'effects)))
                                ;If the current frame was
                                ;found (above) to have a
                                ;cause, note that the
                                ;current frame is that
                                ;frame's effect.
    (addprop (get error-name 'cause) error-name 'effects)))
```

The second module of AURAC, the cliche-recogniser, has a similar function to create error-frames when an error in a cliche is detected:

```
(defun note-cliches (errors)        ;Cliche-recogniser's
                                    ;equivalent of NOTE-ERRORS.
                                    ;ERRORS is a list of:
                                    ;cliche-name, line-number,
                                    ;(word . symbol),
                                    ;procedure-name.
                                    ;WORD is the erroneous
                                    ;word as entered, and
                                    ;SYMBOL is what it ought
                                    ;to be, as retrieved from
                                    ;the EXAMPLE.
  (let ((error-name (symbolconc 'error (setq n (1+ n)))))
                                    ;Allow EXIT or STOP in place
                                    ;of any other control state-
                                    ;ment if the current line is
                                    ;the last of the procedure.
    (cond ((not (and (member (car (third errors))
                             '(exit stop))
                     (null (cdr lines))))
                                    ;remember the error.
          (push error-name errorlist)
                                    ;update ANNOUNCE slot.
          (putprop error-name
                   (announce-slot
                     (get (car errors) 'cliche))
                   'announce)
                                    ;fill the error slots.
          (putprop error-name (cdr (third errors)) 'symbol)
          (putprop error-name (car (third errors)) 'word)
                                    ;the ALTLINES slot is
                                    ;filled later, if the
                                    ;whole of the cliche is
```

```
                          ;found in the user's code.
(putprop error-name nil 'altlines)
                          ;record the cliche-name.
(putprop error-name (car errors) 'cliche)
(putprop error-name nil 'level)
(putprop error-name nil 'altnodes)
(putprop error-name nil 'altprocs)
(putprop error-name nil 'recursion)
(putprop error-name (fourth errors) 'procedure)
(putprop error-name (second errors) 'line)
(putprop error-name nil 'code)
(putprop error-name nil 'evaluated)
(putprop error-name nil 'unreached)
(putprop error-name nil 'type)
(putprop error-name nil 'cause)
(putprop error-name nil 'effects))))
```

In AURAC's third (flow-analysis) module is the function which takes note of each time a SATISFACTION balances an EXPECTATION. Its arguments are the procedure (FN), the line-number (ID) and the name (ARG) of the satisfying item (e.g. NOTE, *P). MARKER is the name of the item expected. An expectation looks like this:

```
(ITEM WHERE-CREATED WHERE-FOUND LINE-NUMBER MARKER)
```

and is referred to in the following code thus:

```
(ARG       FN           POS          ID       MARKER)
```

The point of the marker is to retain information where the name of an item of data changes, for example across the instruction CHECK /TU/ Plus10 ?NEWTU. The function SATISFY is called to inspect each atom (ARG) in the user's code.

```
(defun satisfy (fn id arg)
  (let ((temp nil) (marker nil))
    (cond
                                  ;look down the EXPECTATIONS
```

```
                              ;stack for the most recent
                              ;appearance of ARG within
                              ;FN.
 ((setq temp (fn-assoc arg fn expectations))
                              ;if one is found, establish
                              ;new marker, and
  (setq marker (or (nth 4 temp) (and (atom arg) arg)))
                              ;delete the expectation.
  (setq expectations (delete temp expectations 1))
                              ;record the satisfaction.
  (push
    (list (list arg
                fn
                (third temp)
                id
                (car (last temp)))
          'satisfies
          temp)
    successes)
                              ;in the cases of bound
                              ;variables or expectations
                              ;of complete triples (i.e.
                              ;NOTE expecting CHECK or
                              ;FORGET, look down the
                              ;EXPECTATIONS stack for
                              ;any duplicates.
  (and (or (variablep arg) (pairp arg))
       (others? arg fn id expectations)))
                              ;if none is found, look
                              ;at existing SUCCESSES to
                              ;see if the satisfaction
                              ;is a repeat of one which
                              ;has occurred already.
 ((setq temp (fn-assoc arg fn successes))
  (setq marker (or (nth 4 temp) (and (atom arg) arg)))
  (setq successes (delete temp successes 1))
  (push
    (list (list arg
                fn
                (third temp)
                id
                (car (last temp)))
          'repeats
          temp)
    successes))
                              ;otherwise, the
                              ;SATISFACTION has as yet
                              ;no balancing EXPECTATION.
 (t (push (list arg fn id) satisfactions)
    (setq marker nil)))))
```

APPENDIX E

THE REMAINING PROTOCOLS

Subject 1:

CLASSIFICATION AND SYNTACTIC ANALYSIS (DATABASE):

1. Right, this is Adam's attempt to IMPLICATE or to debug IMPLICATE.	-
2. This is like the Watergate problem. <slurp, slurp>	CLASSIFY
3. IMPLICATE people.	CLASSIFY
4. So first I'm gonna look at the DATABASE.	READ (DB)
5. I'm looking at the various descriptions at the database.	READ (DB)
6. 'PAIDBY' seems to occur in the first two descriptions.	READ (DB)
7. 'PAYS' occurs in the first and third descriptions.	READ (DB)

SYNTACTIC ANALYSIS (1ST SEGMENT):

8. O.K., so what do we want to IMPLICATE?	-
9. PRINT 'X' IS A CRIMINAL.	READ (2)

ASBTRACTION, IDENTIFICATION, SPECIFICATION & PATCH (2ND SEGMENT):

10. And FOR EACH CASE OF 'X' being FRIENDLY to somebody, we're gonna CHECK if that person HAS a POLICERECORD.	ABSTRACT (3) (4)
11. If he HAS we IMPLICATE that person; if not we can loop round.	ABSTRACT (5) (6)

12. So one possible mistake is here.	IDENTIFY
13. When you IMPLICATE somebody, you may want to loop round anyway.	SPECIFY
14. So there is a mistake in 2AA.	SPECIFY
15. I would say 'NEXT CASE', to look for other people to IMPLICATE, having IMPLICATED an instance of somebody.	PATCH
16. Ummm...	?

ABSTRACT AND PATCH (3RD SEGMENT):

17. The next thing we CHECK for is does 'X' PAY money to somebody.	ABSTRACT (7)
18. If that's the case, we CHECK does that person HAVE a POLICERECORD.	ABSTRACT (8)
19. Then we go and IMPLICATE that person, and again I would substitute the EXIT statement with NEXTCASE', to look for other instances.	ABSTRACT (9) PATCH

READ, IDENTIFY, SPECIFY AND REFUSE (4TH SEGMENT):

20. Now, statement 4 is CHECK C PAYS...	READ (11)
21. That's not....	IDENTIFY
22. 4B is a procedure..	IDENTIFY
23. CHECK C PAYS....	READ (11)
24. I think statement 4 is wrong as well...	IDENTIFY
25. 'Cos B is a procedure in it's own right, which isn't defined anywhere.	SPECIFY
26. So the program would stop there with an undefined procedure so we want to CHECK that...	SIMULATE
27. (I'm writing things down)...	?
28. Ermm....	?
29. But I can't see the point of this statement at the moment.	SPECIFY
30. 'Cos we've already CHECKED to see if the person PAYS money to someone else.	ABSTRACT
31. Perhaps the author would look on the relationships which are called 'PAIDBY'.	REFUSE
32. But I'm not asking them what 'PAIDBY' is supposed to do, I'm gonna leave that alone....	REFUSE

ABSTRACT AND PATCH (1ST SEGMENT):

33. But this program can also <solve?> loop.....	SPECIFY
34. Going in turn from the loop we wanna first of all CHECK to see if someone is already a criminal and if they are we wanna NOTE that fact and then stop IMPLICATION at that point	ABSTRACT

and go back up the recursion chain.	
35. So the first thing I would say is 'CHECK that 'X' IS A CRIMINAL: IF PRESENT; EXIT: IF ABSENT; CONTINUE'.	PATCH
36. Then we can NOTE the fact that 'X' IS A CRIMINAL, and we can PRINT out that as we had in the original, just say 'PRINT 'X' ISA CRIMINAL'.	SIMULATE
37. I think I've got most of it here...	META
38. I don't know what 4 is supposed to be doing.	IDENTIFY
39. And that statement 5 will appear a number of times, depending on the level of recursion.	IDENTIFY
40. And I don't know if that isn't required.	IDENTIFY
41. But I'd just chuck '4' out at the moment and see how that works and try that on the terminal.	PATCH
42. See if we get what we expect.	?
43. But that's it.	SUMMARY

Subject 2: is covered in the main text: chapter 3.

Subject 3:

READ (DB):

1. Gotta talk about each line.	?
2. 'Bloody IMPLICATE problem.	CLASSIFY
3. Ermm...	?
4. I'm looking at..	?
5. Ermm...	?
6. The DATABASE....	?
E: Keep talking.	-
7. Well I'm still reading through it.	?
8. I'm just looking at the different relations.	?
9. Err..	?
10. I've noticed already that Fred is an 'ISA' and that Adam, Brian and Colin have 'HAS'.	READ (DB)
11. I see there's a relation between David and Adam.	READ (DB)
12. There's a 'PAIDBY' the gardener.	READ (DB)
E: Could you speak up a bit please.	-
13. O.K...	?
14. Hmmm...	?
15. Right, that makes some sort of sense.	CERTIFY

READ (1ST AND 2ND SEGMENT):

16. "TO IMPLICATE 'X' PRINT 'X' ISA CRIMINAL"....	READ (1) (2)
17. I haven't been told what the student thought wasn't working about this...	REFUSE
18. It would seem that it would PRINT...	SIMULATE
19. Ermm....	?
20. If you say IMPLICATE anything it would immediately PRINT whatever it is ISA CRIMINAL.	ABSTRACT
21. Which means...	?
22. Doesn't seem to make any sense.	?
23. They can't all be...	?
24. But we'll go on.	?

READ (DB):

25. FOR EACH CASE OF 'X' FRIENDLY....	READ (3)
26. HAS POLICERECORD <mumble, mumble>....	READ (4)
27. I'm just checking, umm..	META
28. There were names with a POLICERECORD....	?
29. Relation.	?
30. I'm just checking that all the spellings are correct on 'FRIENDLY', 'cos I can't spell it usually.	META

READ,SIMULATE & IDENTIFY (2ND SEGMENT):

31. FOR EACH CASE OF....	READ (3)
E: Say what you're thinking.	-
32. Ermm...	?
33. I'm just looking at the 'FOR EACH CASE OF'....	READ (3)
34. I'm just imagining in my head that...	?
35. Ermm...	?
36. That Adam was the variable you put in.	SIMULATE
37. So Adam is FRIENDLY with Colin....	SIMULATE
38. You then CHECK whether the....	SIMULATE
39. Ermmm.....	?
40. I see, was it....?	?
41. I'm wondering now, like Adam is FRIENDLY with Colin and Fred....	SIMULATE
42. I'm wondering now if the logic is gonna be right if more than one...	IDENTIFY
43. If there's one FRIENDLY relation and it won't work if there's more than one.	IDENTIFY
44. I'm just checking that.	META

SIMULATE & IDENTIFY (2ND SEGMENT):

45.	IMPLICATE....	?
46.	Well there's recursion within a forward loop.	ABSTRACT
47.	Umm...	?
48.	It will go up that chain..	ABSTRACT
49.	It will find Colin..	SIMULATE
50.	And then go and look for Colin is FRIENDLY,..	SIMULATE
51.	Oh no, ermm..	?
52.	Go back...	SIMULATE
53.	You CHECK if any of them	ABSTRACT
54.	Has Colin got a POLI..	SIMULATE
55.	Yeah, so..	SIMULATE
56.	This one gets in a loop bacause Colin is FRIENDLY with Adam, Adam is FRIENDLY with Colin, Colin HAS a POLICERECORD, so we then recurse with Colin as the parameter.	SPECIFY
57.	And the first thing it will do is that, umm, it will find out that Colin is FRIENDLY with Adam.	SIMULATE
58.	Now Adam HAS a POLICERECORD and is FRIENDLY with Colin,....	SIMULATE
59.	And it will just get into a loop on that first 'CHECK and IMPLICATE'.	SPECIFY
60.	It's 2A and 2AA because...	SPECIFY
61.	Umm...	?
62.	And they could've got round that by putting something in the DATABASE and CHECKING...	ADVISE
63.	And what's that first PRINT...	IDENTIFY
64.	And *&^%($#@%$&&^*#@@!$.	?
65.	But, even if they got that right, that looks like it's gonna get....	IDENTIFY
66.	Just with the two, with Adam and Colin it looks like it would get into an infinite loop.	SPECIFY

ABSTRACT, READ & IDENTIFY (2ND SEGMENT):

67.	I assume that it had been sorted out FOR EACH CASE OF....	IDENTIFY
68.	It EXITS if...	ABSTRACT
69.	Ummm...	?
70.	Well the EXIT after IMPLICATE is going to...	ABSTRACT
71.	I can't think what's gonna happen about that.	?
72.	It's gonna go up and recurse and when it comes back down it's gonna EXIT.	ABSTRACT
73.	It's not gonna do the CHECK of who PAYS.	ABSTRACT
74.	IF ABSENT NEXT.....	READ (6)
75.	It will CHECK all the FRIENDLY ones.	ABSTRACT
76.	I'm looking at 2AB now.	READ (6)
77.	Looking at the NEXT CASE.	READ (6)
78.	FRIENDLY..	READ (3)
79.	If there were more than one FRIENDLY and they all had POLICERECORDS I don't see how this would get it....	IDENTIFY

80. You'll get the first one with a POLICERECORD, and go off IMPLICATING on that one; but not if there's more than one....	IDENTIFY

META-COMMENTS:

81. Right, so that's two problems I can see....	IDENTIFY
82. In the FOR-loop of two.....	?
83. And I'll get three...	?
84. Ermm...	?
85. It's one of those things where a student thinks he can do it in a single procedure.	IDENTIFY
86. 'Cos I assume this is the whole program.	REFUSE
87. They haven't realised that programs have to be in separate procedures.	REFUSE
88. If you were actually tutoring a student you wouldn't give him all the gumph I've just given, you would tell him to go away and think about it in separate procedures.	?
89. I'm now looking at 3.	META
90. Ermm...	?

READ & CERTIFY (3RD SEGMENT):

91. FOR EACH CASE OF 'X' PAYS.	READ (7)
92. I assume there's no case where a person PAYS one person and that person PAYS them back 'cos on the 'PAYS' relation you're not gonna get that infinite loop....	CERTIFY

READ (DB):

93. I assume that.....	?
94. I'm just checking the DATABASE now....	READ (DB)
95. CHECKS through 'PAIDBY' relations.	READ (DB)
96. They've got these 'PAIDBY' relations and they never actually use them.	READ (DB)

META-COMMENTS:

97. I assume it's gonna be one of those things where the students write another procedure or something.....	?
98. And then the 'loves and the 'ISA' relations aren't used either.	ABSTRACT (DB)
99. But I assume they're gonna do something about them..	?
100. Umm....	?

101. <mumble, mumble>	?
102. I'm looking at '3' now, I'm trying to work out this recursion within the FOR-loop.	META
103. I'm trying to work what will happen when it ends.	META
104. And I would tend to debug a program like this myself by actually running it because you're never quite sure what SOLO will do.	META

ABSTRACT & SIMULATE (3RD SEGMENT):

E. What're you doing now?	-
105. I'm just scribbling a notation to myself about what would happen as you go through '3' after each 'FOR' and each 'IMPLICATE'.	ABSTRACT
106. I'm just imagining Fred went to Adam and say Adam went to Brian....	SIMULATE

META-COMMENTS:

107. And in none of this,...	?
108. This is why Hank's examples of...	?
109. Nothing's ever side-effected, nothing's ever put into the DATABASE.....	IDENTIFY

CERTIFY & ABSTRACT (1ST SEGMENT):

110. Oh, I can see why that line 1 has some meaning now....	CERTIFY
111. Because...	?
112. Umm..	?
113. Apart from the first time when it doesn't make much sense, 'cos you're just trying to IMPLICATE the person.,	ABSTRACT
114. It will actually PRINT that the next, umm, person in the relation who was FRIENDLY with 'X' and HAS a POLICERECORD, when it gets to them it will PRINT that they are guilty.	ABSTRACT

META-COMMENTS:

115. Yes, but, ummmm, it doesn't make sense.	IDENTIFY
116. I mean knowing people have POLICERECORDS is just one thing you CHECK.	ABSTRACT
117. This isn't going to IMPLICATE 'X' in any way, it's just CHECKING through the DATABASE very simplistically.....	CLASSIFY

READ, SIMULATE, ABSTRACT & IDENTIFY (3RD SEGMENT:

118.	I'm just going back to think about '3' again.	?
119.	FOR EACH CASE OF F....	READ (7)
120.	IMPLICATE....	READ (8)
121.	Christ....	?
122.	B failed...	ABSTRACT
123.	3AB...	READ (9)
124.	I've got F goes to A, A goes to B.,	SIMULATE
125.	I'm now thinking about when it comes back.	SIMULATE
126.	B is not a...	?
127.	I wonder if B has not got any 'PAID' relations....	SIMULATE
128.	Oh, B is not the same as ?B in 3, B just stands for Brian in my notation.	?
129.	Umm...	?
130.	B....	?
131.	It would just come back into that and then it would EXIT from A....	ABSTRACT
132.	Yeah, there's no way that it's going to CHECK every 'PAID' relation, 'cos when it tries to come back after either there being no 'PAYS' or no POLICERECORD, the FOR-loop isn't.....	ABSTRACT
133.	Recursion is gonna blow up so quickly in SOLO anyway, even if it made sense...	IDENTIFY ERROR1

READ & IDENTIFY (4TH SEGMENT):

134.	Which I can't make sense of.	?
135.	Right, so I've given up thinking about that so I'm thinking about number 4.	?
136.	4B.	READ (11)
137.	CHECK *C PAYS.	READ (11)
138.	I can't, umm.....	?
139.	I've had far more experience the micro-SOLO	?
140.	I can't think how they got it in like that.	?
141.	Maybe the thing just....	?
142.	Whether it would accept it.	?
143.	I can't understand the syntax of the B	IDENTIFY
144.	CHECK *C PAYS.	READ (11)
145.	Well, I mean maybe the B should be after the 'PAYS' or something but I really don't see how they could have produced that line.	IDENTIFY
146.	They couldn't have produced that line on the micro-SOLO because it never would have accepted it, but I assume the original SOLO would have somehow done it.	IDENTIFY
147.	But there's no way a symbol in front of a SOLO command can be interpreted I don't think.	IDENTIFY ERROR2

IDENTIFY (3RD SEGMENT):

148.	I'm now on to '5'.	?
149.	For micro-SOLO you can't have a 'FOR EACH' followed by a 'FOR EACH' anyway.	? ERROR3
150.	It would bomb out.	ERROR3
151.	It looks like a BASIC programmer....	IDENTIFY
152.	You'd have to have the other 'FOR EACH' within procedure within one of the 'IF PRESENT, IF ABSENT' bits of the first 'FOR EACH'.	IDENTIFY
153.	Sort of high level bits are wrong with it...	IDENTIFY
154.	'FOR EACH' and recursion's screwed up..	IDENTIFY
155.	'FOR EACH' within 'FOR EACH, I think, is fairly screwed up.,	
156.	There're relations within the DATABASE that are gonna cause an infinite loop, if you chose those to start with.	IDENTIFY

META-COMMENTS:

157.	It isn't actually very interesting in all it does.....	?
158.	If it did anything it just PRINTED people are guilty if they've got POLICERECORDS.	ABSTRACT
159.	It doesn't seem to do anything to IMPLICATE the first person.	CLASSIFY
160.	There's no increment, or decision-making.	CLASSIFY
161.	I give up here.	
162.	That's it.	

Subject 4:

ABSTRACTION & CERTIFICATION (1ST & 2ND SEGMENTS):

1.	O.K. I'm just reading it, to see what it says.	?
2.	TO IMPLICATE 'X' PRINT 'X' ISA CRIMINAL....	READ (1) (2)
3.	FOR EACH CASE OF 'X' something or other.	ABSTRACT (3)
4.	CHECK that something or another HAS POLICERECORD, IF PRESENT IMPLICATE that, then EXIT.	ABSTRACT (4) (5)
5.	If not, go round again.	ABSTRACT (6)
6.	That looks alright.	CERTIFY

READ (3RD SEGMENT):

7. FOR EACH CASE OF 'X' PAYS B CHECK B HAS a POLICERECORD.	READ (7) (8)
8. IF PRESENT IMPLICATE B.	READ (9)

IDENTIFICATION & SPECIFICATION (4TH SEGMENT):

9. B CHECK C PAYS.	READ (11)
10. That's nonsense, that one.....	IDENTIFY
11. So line 4, anyway, doesn't work.	IDENTIFY
12. I think that's absolute rubbish.	IDENTIFY
13. B CHECK C PAYS.	READ (11)
14. In any case, there's no *C in the program.	SPECIFY

SIMULATION & IDENTIFICATION (2ND SEGMENT):

15. FOR EACH CASE (I've started at the top again).	READ (3)
16. FOR EACH CASE OF 'X' FRIENDLY A there's...	READ (3)
17. Adam's FRIENDLY with two people.	READ (DB)
18. So I'll just assume it's Adam for the moment.	SIMULATE
19. Ummm....	?
20. Adam's FRIENDLY with Colin and Fred, so we'll just see if either of them HAS got a POLICERECORD.	SIMULATE
21. Colin HAS....	?
22. Ummm....	?
23. So it goes around again, starts again.	SIMULATE
24. Colin FRIENDLY Adam.	SIMULATE
25. That's a loop.	SPECIFY
26. 'Cos Adam's FRIENDLY Colin and Colin's FRIENDLY Adam.	SPECIFY
27. So I suppose that line 2 would not actually stop.	SPECIFY
28. That's probably the bug.	IDENTIFY
29. Umm...	?

SIMULATION & CERTIFICATION (3RD SEGMENT):

30. So let's have a look at line 3...	?
31. FOR EACH CASE OF 'X' PAYS...	READ (7)
32. Fred PAYS Colin.	SIMULATE
33. That's alright.	CERTIFY

SUMMARY:

34. I think there's just those two bugs, actually...
35. As far as I can see.
36. That's it.

Subject 5:

READ & CLASSIFICATION (START OF PROGRAM):

1. Okay, TO IMPLICATE X.	READ (1)
2. I immediately realize this is like the standard 'WATERGATE' problem, which in turn should be like the standard 'INFECT' problem.	CLASSIFY
3. That's my expectation.	CLASSIFY

READ & IDENTIFICATION (1ST SEGMENT):

4. One, PRINT X ISA CRIMINAL.	READ (2)
5. And I'm just going to read it now but, my gut feeling is I want to skim through it but some thoughts come to mind instantly so I'll shout them out here.	META
6. I think it ought to be a NOTE here.	IDENTIFY
7. I would say that to myself mentally but I wouldn't point it out to the student.	META

READ, ABSTRACTION, SPECIFICATION & PATCH (2ND SEGMENT):

8. Two, FOR EACH CASE OF X FRIENDLY A CHECK A HAS POLICERECORD. IF PRESENT, IMPLICATE A.	READ (3) (4) (5)
9. So, everyone that X knows who HAS a POLICERECORD is going to recursively get IMPLICATED.	ABSTRACT
10. It's just going to PRINT out more junk about that.	ABSTRACT
11. Uh, FOR EACH CASE OF....	READ
12. Then, it EXIT's on 2AA, which is wrong.	SPECIFY
13. It should be NEXTCASE.	PATCH

CERTIFICATION (2ND SEGMENT):

14. Anyway, go on to line 3.	?
15. I see that 2AB, IF ABSENT, NEXTCASE, is okay.	CERTIFY
16. That's for people who didn't have a POLICERECORD,	ABSTRACT

17. it goes on to the next FRIENDLY.	ABSTRACT

READ, PATCH & CERTIFICATION (3RD SEGMENT):

18. Three, FOR EACH CASE OF X PAYS B, CHECK B HAS POLICERECORD, IF PRESENT, IMPLICATE B, EXIT.	READ (7) (8) (9)
19. Again ought to be NEXTCASE on 3 AA.	PATCH
20. IF ABSENT, 3AB, NEXTCASE, that looks okay.	CERTIFY
21. It's everyone that X PAYS that HAS a POLICERECORD.	ABSTRACT
22. I haven't thought about what those mean yet, I mean sort of semantically in English, but I'll come back to that in a minute.	META

READ & IDENTIFICATION (4TH SEGMENT):

23. Four, B CHECK C PAYS.	READ (11)
24. I don't know what the hell that is.	IDENTIFY
25. There's this sort of multisyntactic stroke typing error which I'll come back to in a second.	IDENTIFY
26. Just a wierd one.	IDENTIFY

READ, IDENTIFICATION & SEPCIFICATION (5TH SEGMENT):

27. Five, PRINT THAT SEEMS TO BE THE WHOLE GROUP IDENTIFIED.	READ (12)
28. DONE.	READ (13)
29. Okay, now five is also wierd.	IDENTIFY
30. It...because, my gut feeling about why it's wierd is that we're going to PRINT THAT SEEMS TO BE THE WHOLE GROUP IDENTIFIED for every turkey on this chain.	SPECIFY

META-COMMENTS & ABSTRACTION (3RD, 4TH AND 5TH SEGMENTS):

31. What we're gonna do, I'm just thinking here...	META
32. It appears, I mean I haven't thought it through, but it appears it's going to go through, well, to discriminate between a fan and a chain where a fan is all the ones emanating out from a node, using FOR EACH, call that a fan.	ABSTRACT
33. And a chain is the one where we go recursively down a chain.	ABSTRACT
34. Now it looks to me like there's going to be a fan of people who are going to get, I'll say	ABSTRACT

	caught, meaning we're going to PRINT X ISA CRIMINAL about them.	
35.	And then basically everytime we get everybody we're going to PRINT out line 5: THAT SEEMS TO BE THE WHOLE LOT IDENTIFIED.	ABSTRACT
36.	Now I might actually be wrong and I'm just going to think it through.	META
37.	Let me look at the database first, and then I'm going to talk through a little trace of it, just to confirm it to myself.	META
38.	I'm going back to the instructions just to see whether the student seems to run the program.	META
39.	I doubt it.	META
40.	(Re-reads instructions).	
41.	So I don't even know what the hell it's supposed to do but I've got my ideas.	META

READ & ABSTRACTION (2ND AND 3RD SEGMENTS):

42.	So let's look at the database.	META
43.	FRED ISA MAN LOVES MARY PAIDBY BRIAN PAYS COLIN.	READ (DB)
44.	I see that PAIDBY is an irrelevant piece of cosmetics.	ABSTRACT (DB)
45.	In fact everything is cosmetic for this particular problem unless it's FRIENDLY or PAYS, but I just see the structure here.	ABSTRACT (DB)
46.	ADAM HAS POLICERECORD.	READ (DB)
47.	Obviously POLICERECORD is important....	ABSTRACT (DB)
48.	ADAM HAS POLICERECORD.	READ (DB)
49.	Sorry, I....	?
50.	That last realisation, I'm commenting on my eye movements here.	META
51.	I noticed that ADAM HAS POLICERECORD and that triggered off: 'Oh, yeah, ADAM HAS POLICERECORD must be important'.	META
52.	And I instantly scanned back to line 3A just to confirm for myself that HAS POLICERECORD is indeed important as I actually knew and I even double checked again by looking further back to line 2A, first to see that there were no typing errors and (b) to see that there were no tricks in the problem as set up by the experimenters.	META
53.	Okay.	?
54.	Now, I'm reading through ADAM here.	READ (DB)
55.	ADAM HAS POLICERECORD, FRIENDLY COLIN, FRIENDLY FRED, PAIDBY BRIAN.	READ (DB)

56. Okay, well PAIDBY BRIAN is irrelevant.	ABSTRACT (DB)
57. The student might have made a mistake or I don't know what.	ABSTRACT
58. It doesn't matter.	?
59. ADAM, well, it may or may not be a mistake, PAIDBY BRIAN may be just cosmetic.	ABSTRACT (DB)
60. BRIAN is the one who PAYS everybody.	ABSTRACT (DB)
61. BRIAN HAS POLICERECORD, PAYS ADAM, PAYS FRED.	READ (DB)
62. COLIN HAS POLICERECORD, FRIENDLY ADAM, PAIDBY FRED.	READ (DB)
63. DAVID FRIENDLY ADAM, ERIC PAIDBY BRIAN, ISA GARDENER.	READ (DB)
64. More cosmetics.	ABSTRACT (DB)
65. So, I can take any one of these as a starting point.	SIMULATE
66. I happened to notice there's an alphabetical group here if you go by ADAM, BRIAN, COLIN, DAVID, ERIC, FRED.	ABSTRACT (DB)
67. We've got the first six letters of the alphabet.	ABSTRACT (DB)
68. I'm going to assume, uh, ADAM and BRIAN make the most natural starting point.	META
69. I'm looking for a test case to try out this crazy program. (snigger, snigger, snigger).	META
70. I'm going to test it out on BRIAN.	META
71. The reason I'm doing this is because it's twenty times easier to work through a test case than to go through it abstractly in my head.	META

SIMULATION (1ST AND 2ND SEGMENTS):

72. So, here I go and this is, this is exactly what I'd do as a tutor.	META
73. I'm going to type in IMPLICATE BRIAN, and I'm writing it down.	SIMULATE
74. I'm probably doing too much work for this thing but this is the only way I can be absolutely sure myself what it does.	META
75. I would in fact do this on a computer if the student was in front of me.	META
76. Okay IMPLICATE BRIAN.	SIMULATE
77. First thing it does (I'm gonna split my page here) and keep a record of the actual printouts as they appear on the computer on the right-hand side of my sheet.	SIMULATE
78. First thing it's gonna do is print BRIAN ISA CRIMINAL.	SIMULATE

79.	Now the only reason I chose BRIAN instead of ADAM; ADAM would be first choice because letter A, but BRIAN has more links emanating from him for me to test out all along this PAYS branch.	META
80.	I'm gonna test ADAM in my head in a second.	META
81.	Ah, looking down through ADAM, I'm gonna get to BRIAN in a sec anyway in the normal course of events.	SIMULATE
82.	So this test, IMPLICATE BRIAN, will be enough to satisfy my feeble brain.	META
83.	Okay, BRIAN ISA CRIMINAL gets printed out at line 1.	SIMULATE
84.	Two, FOR EACH CASE /X/ FRIENDLY A, uh....	SIMULATE
85.	BRIAN's not FRIENDLY with anybody, I can see at a glance so let's zip on down to step 3.	SIMULATE

SIMULATION, IDENTIFICATION & PATCH (3RD SEGMENT):

86.	FOR EACH CASE OF /X/ PAYS B.	READ (7)
87.	Okay BRIAN PAYS ERIC, ADAM and FRED.	SIMULATE
88.	So, we're going to first take ERIC.	SIMULATE
89.	We're going to recursively IMPLICATE ERIC and in the original program we would EXIT.	SIMULATE
90.	Now, I said that's a bug but it might just be the case that the student only wanted to get the first character.	REFUSE
91.	For instance there's a fan of PAYS relations coming out of BRIAN.	ABSTRACT
92.	He may only have wanted to get the first one and then IMPLICATE would recursively, would PRINT out ERIC ISA CRIMINAL and recursively, well, it would get either the person he's FRIENDLY with or with POLICERECORD or etc, the person he pays that HAS a POLICERECORD.	ABSTRACT
93.	I maintain that's not what the student intended.	IDENTIFY
94.	He surely meant NEXTCASE, so when I work through the example here I'm simply changing the EXIT to NEXTCASE.	SPECIFY
95.	At this point I would have to ask the student, did he want to get them all or did he just want to get the first guy who was paid.	REFUSE
96.	I think there might be some halfassed reason, historically, it's the first guy you paid, it's your cohort in crime or something crazy like that.	META
97.	But I doubt it, that's implausible to me.	META
98.	So I'm going to arbritarily change 2AA and 3AA to NEXTCASE, in fact I already did that on the sheet, crossing out EXIT, and I'm working through it on paper in the same way.	PATCH
99.	So we're now recursively going to IMPLICATE	SIMULATE

ERIC.	
100. Now in order to remember where I am I have to keep a little stack, I'm going to have to do this on paper.	META

SIMULATION & META COMMENTS (1ST, 2ND AND 3RD SEGMENTS):

101. So, I'm going to IMPLICATE BRIAN.	SIMULATE
102. I'm writing indented 'IMP ERIC', in fact I'll just keep my own trace here.	META
103. I'm going to write 3AA, just to remind me where the hell I am.	META
104. 3AA, IMP ERIC.	META
105. Obviously if I was using Tony's system this would be a hell of a lot easier because I'd just have it indented for me.	?
106. Now, what the hell's it going to do?	SIMULATE
107. On the right hand side I'm going to print out ERIC ISA CRIMINAL.	META
108. Why? Why should ERIC be a CRIMINAL.	META
109. Wait a minute, in my head I'm saying the guy HAS to have a POLICERECORD.	META
110. ERIC doesn't.	READ (DB)
111. Aw, shit, sorry, I'm thinking three different things at once here.	FALSE START
112. Right, ignore that little digression there.	FALSE START

SIMULATION & SPECIFICATION (1ST, 2ND AND 3RD SEGMENTS):

113. Let us restart the trace.	FALSE START
114. IMPLICATE BRIAN, one PRINT BRIAN ISA CRIMINAL.	READ (1) (2)
115. I'm starting completely from the beginning again.	META
116. BRIAN's not FRIENDLY with anybody so we skip over line 2.	SIMULATE
117. Line 3, X PAYS ERIC.	SIMULATE
118. Now it's already running off at the mouth, starting to IMPLICATE ERIC.	SIMULATE
119. ERIC doesn't have a POLICERECORD.	READ (DB)
120. First, CHECK has a POLICERECORD.	READ (4)
121. So, we get the NEXTCASE of 3AB.	READ (6)
122. What about ADAM?	META
123. Okay, ADAM HAS POLICERECORD.	READ (DB)
124. Hallelieuja, so all the points I made are still valid.	META

125.	Slip of the mind here.	META
126.	So we IMPLICATE ADAM, cross out ERIC on my little trace sheet.	SIMULATE
127.	IMPLICATE ADAM at 3AA.	SIMULATE
128.	Now my print out will say ADAM ISA CRIMINAL.	META
129.	And now, step 2 in my recursive call here, see if ADAM's FRIENDLY with anybody.	SIMULATE
130.	Yes, he is, he's FRIENDLY with COLIN, so let's see whether COLIN's got a POLICERECORD at 2A.	SIMULATE
131.	COLIN does have a POLICERECORD.	READ (DB)
132.	So, we're going to IMPLICATE ADAM.	SIMULATE
133.	So, this is beautiful because you're going to get into an infinite loop here and all sort of screwy things.	SPECIFY
134.	Okay, so, ADAM's FRIENDLY COLIN and COLIN's FRIENDLY ADAM, so we've got a loop in the database.	SPECIFY

META-COMMENTS:

135.	Now, I could've done a loop detection test in the database first, because I know that's a kind of bug that comes up but I think, I know that comes up in some of Hank's examples, but in fact as a real live tutor I'm 99% certain I'd debug it exactly this way, and that's how I'm behaving. I simply go through the trace and during the trace I discover the loop.	META
136.	That to me is simpler.	META
137.	Well, depends on how complicated the database is, but a loop in the database is not a priori reason for alarm, only in the context of this screwy program.	META
138.	So, I haven't even done the trace.	META
139.	This immediately triggers off loop, loop, loopbecause I see that, let me just do the trace now.	SPECIFY
140.	This I wouldn't do for the student, uh, for detecting the student's bug.	META

LOOPTEST & SPECIFY (2ND SEGMENT AND DB):

141.	I might do it when I'm explaining it to the student, but what I do here, uh, I'm doing ADAM here, my first recursive call and I see that ADAM is FRIENDLY with COLIN, at step 2 and at step 2 COLIN HAS POLICERECORD	LOOPTEST

	so I recursively, 2AA IMPLICATE COLIN and now let's trace it out here.	
142.	Step one of my next recursive call I PRINT COLIN ISA CRIMINAL	LOOPTEST
143.	And FOR EACH CASE OF COLIN FRIENDLY with somebody,who's he FRIENDLY with, he's FRIENDLY with ADAM, CHECK ADAM HAS POLICERECORD, yes indeed he does so step 2AA, next level of recursion down, IMPLICATE ADAM, etc.	LOOPTEST
144.	So that's loop in database.	SPECIFIY

ADVICE & PATCH (2ND AND 3RD SEGMENTS):

145.	So, now what I'd do is, I would eliminate the loop.	ADVISE
146.	I mean to debug the student's program.	?
147.	If I'm supposed to.	?
148.	(Rereads instructions).	?
149.	Okay, so I'm looking over to my scratchsheet.	META
150.	So I'm going to debug this thing.	META
151.	What I'm going to do is change the database here.	META
152.	Where it says COLIN FRIENDLY ADAM, ADAM FRIENDLY COLIN, in a sense you only need one and the obvious thing to do is put in an inverse relation and have FRED PAIDBY BRIAN and BRIAN PAYS FRED.	PATCH

SIMULATION, ADVICE & REFUSAL (2ND AND 3RD SEGMENTS):

153.	That little cosmetic trick is a way of keeping out endless loops but immediately an alarm goes off in my head.	META
154.	It's going to be more complicated than that because it's not only that I have to look out for FRIENDLY loops within themselves and PAYS loops within themselves, I could have an indirect loop, I suppose ADAM could PAY X and X could be FRIENDLY with ADAM and that's going to be an additional way of looping, which is a little more subtle. Now let me instantly see if there is such a case.	SIMULATE
156.	Let me just say BRIAN PAYS ERIC for example.	LOOPTEST
157.	If ERIC was FRIENDLY with BRIAN I'm up the creek, but he's not so, jesus christ, having to check all these is actually a pain in the ass but I might as well do it.	?
158.	ADAM FRIENDLY COLIN.	READ (DB)
159.	Let's check that COLIN doesn't get back to	META

ADAM.

160. This seems, now I'm now thinking these indirect loops are going to be even worse because ADAM could pay X, X could PAY Y, Y could PAY Z, Z could be FRIENDLY with ADAM and then I'm still shafted. — META
161. So this looks completely useless. (blows raspberry) — ?
162. So the way I would do it, I mean I know from many AI tricks or from normal lets say graph traversal tricks that anyone knows would be to flag the nodes that have already been hit so you don't do them again. — META
163. I mean that's the simplest thing to do either with trivial node blah already used or the more obvious thing to do is change line 1 to NOTE X ISA CRIMINAL and then if X, then you could CHECK X ISA CRIMINAL and skip over them. — ADVISE
164. So I'd do something like, line 1 CHECK X ISA CRIMINAL, IF PRESENT EXIT, oh sorry, IF PRESENT, ah shit, this is already more complicated, I'll come back to this in a second because I don't want to lose my train of thought. — ADVISE
165. Okay, so that, I'm going to write a note to myself on a sheet of paper here, I would have to ask the student actually what he wanted to do at this point. — REFUSE

ABSTRACTION & PATCH (DB):

166. I would assume that he typically wants to start with ADAM, whip through the database and get everybody who's either FRIENDLY with ADAM, HAS a POLICERECORD, or etc. etc. — ABSTRACT
167. And they're, let's call these people cohorts in crime, he wants to get all the cohorts in crime in the normal way. — ABSTRACT
168. Therefore he wants to avoid looping, he'd want to flag them in someway and I think that's not hard to do. — ADVISE
169. I'll do that at step 1 in a second. — META
170. What I'm doing now, is thinking, I just want to reconfirm that steps 2 and 3 are okay. — META
171. Now, popping back my own goal stack, what I've been working on is looking for loops in the database. — META
172. I'm going to look for simple loops of the type A FRIENDLY B, B FRIENDLY A. — META
173. I'm going to look for simple loops of that kind first. — META
174. I'm simply going to change FRIENDLY to — PATCH

FRIENDOF.

175. I'm going to use this inverse relation trick. — ADVISE
176. So I say COLIN FRIENDOF ADAM. — PATCH

LOOPTEST, CERTIFICATION & PATCH (2ND AMD 3RD SEGMENTS):

177. Okay, ADAM FRIENDLY FRED. — LOOPTEST
178. Let's see if FRED loops back. — LOOPTEST
179. No. — CERTIFY
180. Okay, so let's do BRIAN. — LOOPTEST
181. I'm doing them in alphabetic order. — META
182. PAYS ERIC. — LOOPTEST
183. Does ERIC loop back. — LOOPTEST
184. No, I mean only with this PAIDBY but that's okay, it's this cosmetic one which avoids looping. — CERTIFY
185. BRIAN PAYS ADAM. — LOOPTEST
186. Does ADAM PAY BRIAN? No. — LOOPTEST
187. BRIAN PAYS FRED, good. — LOOPTEST
188. Let's go to COLIN. — LOOPTEST
189. COLIN FRIENDOF and PAIDBY. — LOOPTEST
190. So there's no opportunity for a loop. — CERTIFY
191. DAVID FRIENDLY ADAM. — LOOPTEST
192. Let's see ADAM FRIENDLY DAVID. No. — LOOPTEST
193. I should, to be consistent, add FRIENDOF. — META
194. I'm going to add this inverse here, ADAM FRIENDOF DAVID. — PATCH
195. This is total rubbish, totally cosmetic, but I'm going to add it in anyway. — META
196. Okay, ADAM FRIENDOF DAVID. — LOOPTEST
197. Now, alphabetically here, ERIC PAIDBY BRIAN so we should have BRIAN PAYS ERIC. Correct. — LOOPTEST
198. ISA GARDENER is useless. Leave it in. — LOOPTEST
199. Let's go up to FRED now. — LOOPTEST
200. ISA MAN, LOVES MARY, is useless. — LOOPTEST
201. PAIDBY BRIAN. — LOOPTEST
202. Let's check BRIAN PAYS FRED. Yes. — LOOPTEST
203. FRED PAYS COLIN, so, we've got COLIN PAIDBY FRED. Beautiful. — LOOPTEST LOOPTEST
204. Okay, so, there're no simple loops, in the sense A PAYS B, B PAYS A, or A FRIENDLY B, B FRIENDLY A. — CERTIFY

LOOPTEST & CERTIFY (2ND AND 3RD SEGMENTS):

205. Now I'm going to look for a simple indirect loop of the type A PAYS B and B FRIENDLY A, which I think may shaft me. — META
206. I don't know, I'll look for a couple of cases. — META

207.	Let's go ADAM FRIENDLY COLIN, does COLIN PAY ADAM. No.	LOOPTEST
208.	ADAM FRIENDLY FRED, does FRED PAY ADAM. No.	LOOPTEST
209.	Okay let's go down to BRIAN.	LOOPTEST
210.	BRIAN PAYS ERIC, is ERIC FRIENDLY BRIAN. No.	LOOPTEST
211.	BRIAN PAYS ADAM, is ADAM FRIENDLY BRIAN. No.	LOOPTEST
212.	BRIAN PAYS FRED, is FRED.... No.	LOOPTEST
213.	Okay, COLIN only has these inverse relations.	LOOPTEST
214.	DAVID doesn't have any.	LOOPTEST
215.	Uh, sorry, DAVID has FRIENDLY ADAM.	LOOPTEST
216.	Interesting because DAVID was a one liner.	META
217.	I hallucinated that he was going to be purely cosmetic like ISA BURGLAR or ISA GARDENER, something like that.	META
218.	God knows why.	?
219.	DAVID FRIENDLY ADAM, ADAM should be FRIENDOF DAVID which I put in by hand.	LOOPTEST
220.	So there's no PAYS, there's no indirect loop.	CERTIFY
221.	ERIC is PAIDBY is only an inverse so that's okay.	READ (DB)
222.	FRED is only, ah, PAYS COLIN so we check that COLIN FRIENDLY FRED I'll be in trouble.	READ (DB)
223.	So I'm not, good.	CERTIFY
224.	So I get no simple loops, no two step loops, either PAYS/FRIENDLY or FRIENDLY/PAYS.	CERTIFY
225.	Now I was just thinking in my head it's intrinsically possible for there to be a long distance loop of the kind that, well, it's just obvious what it is.	META
226.	A very long term looping around.	META
227.	We could eventually get from ADAM back to himself.	?
228.	Now let's just see if....	?
229.	It ought to be possible.	?
230.	Let's see if I can get from BRIAN to FRED via PAYS.	LOOPTEST
231.	I can get from FRED to COLIN via PAYS.	LOOPTEST
232.	And I can't get anywhere else because there's only inverses.	LOOPTEST
233.	Ah ha!	?
234.	Interesting. So I've got real terminal nodes here.	CERTIFY
235.	Now let me see, working backwards here in the true means ends analysis I see that I can get to ADAM from DAVID so I'm going to see....	META
236.	Knowing that ADAM's the guy I want to start with, for example, is there anybody that	META

takes me back to him?

237. So I'm looking backwards now. META
238. I see DAVID FRIENDLY ADAM is a potential source of trouble, so how do I get to DAVID? LOOPTEST
239. So I'm looking along the right hand sides [of the database provided] to see how I would get to DAVID in the normal course of events. META
240. And there is no way to get to DAVID in the normal course of events. LOOPTEST
241. So that's good, so that's not a source of trouble. CERTIFY
242. Let's see if there's any other way to get to ADAM. META
243. On the right hand side I see that ADAM, I see BRIAN PAYS ADAM. LOOPTEST
244. I just found ADAM on the right hand side of that. LOOPTEST
245. So how do we get to BRIAN? META
246. Again a means ends analysis here. META
247. How do we get to BRIAN? META
248. On the right hand side, in the normal way, there is no way. LOOPTEST
249. Wow, that's beautiful, that's interesting. ?
250. So BRIAN can't be reached in the normal way unless he's typed in at the top level. CERTIFY
251. ADAM can be reached from BRIAN. LOOPTEST

LOOPTEST CERTIFICATION & PATCH (2ND AND 3RD SEGMENTS)

252. Why don't I draw this out? ?
253. What I'm drawing here is just a tree which says who's reachable. META
254. I don't even care whether it's PAYS or FRIENDLY or what the hell it is. META
255. Doesn't matter to me, I just want to see whether they can be reached at all. LOOPTEST
256. From BRIAN I can get to ERIC, ADAM and FRED. LOOPTEST
257. From ADAM I can get to COLIN and FRED. LOOPTEST
258. From FRED I can get to COLIN. LOOPTEST
259. From.... ?
260. Well I started with BRIAN arbitrarily, well it doesn't matter. META
261. So who haven't I done? META
262. Do it alphabetically here. ?
263. COLIN. LOOPTEST
264. From COLIN I can't get to anybody. LOOPTEST
265. From DAVID I can get to ADAM, but I've got no way to get to DAVID, so I've got.... LOOPTEST
266. So I can get from BRIAN to ADAM or I can get from DAVID to ADAM. LOOPTEST
267. Interesting. ?

268.	So if I start with DAVID I get DAVID, ADAM, COLIN, FRED.	LOOPTEST
269.	So I've got a little sort of converging tree here.	CERTIFY
270.	COLIN I can't get to anybody.	LOOPTEST
271.	Alphabetically now the next one is ERIC.	?
272.	From ERIC I can't get to anybody.	LOOPTEST
273.	BRIAN takes me to ERIC so that's a deadend or terminal node.	LOOPTEST
274.	From FRED, I already did, I can get to COLIN.	LOOPTEST
275.	Interesting, so that's the complete tree here.	CERTIFY
276.	BRIAN is the most prolific source let's say, in this whole damn thing.	?
277.	BRIAN'll get me ERIC, ADAM, FRED.	LOOPTEST
278.	ERIC'll just, uh, because ERIC doesn't have a POLICERECORD I'll skip him.	LOOPTEST
279.	I'm now doing a sort of quicky trace by looking at this tree I've drawn.	META
280.	This is a slightly different way of doing it.	META
281.	I'll get to ADAM, ADAM'll get to COLIN.	LOOPTEST
282.	Now I'm looking back up at my trace.	META
283.	I had this infinite recursion because COLIN took me back to ADAM so by changing that relation, by changing it to FRIENDOF, the thing appears to terminate, because I get BRIAN ISA CRIMINAL, ADAM ISA CRIMINAL, COLIN ISA CRIMINAL.	LOOPTEST
284.	And then I'm looking back at my procedure.	META
285.	COLIN.	?
286.	Now I'm looking in my database.	META
287.	COLIN HAS POLICERECORD.	LOOPTEST
288.	I've got him at....	?
289.	Whereever the hell I got him.	?
290.	At 2AA.	LOOPTEST
291.	And then I would go NEXTCASE.	LOOPTEST
292.	I would try to get the NEXTCASE of....	?
293.	Looking back to my trace here.	META
294.	Of ADAM, who I'm working on.	META
295.	And I cross out my recursive loop where I get IMPLICATE ADAM, on my trace, now that I've corrected my database.	META
296.	It all seems to work okay.	CERTIFY
297.	IMPLICATE COLIN would PRINT out COLIN ISA CRIMINAL and then NEXTCASE would look for the NEXTCASE of who ADAM's friendly with.	LOOPTEST
298.	And I see he's FRIENDLY with FRED.	LOOPTEST
299.	So I'm going to CHECK FRED HAS POLICERECORD.	LOOPTEST
300.	He doesn't so he gets skipped.	LOOPTEST
301.	I'll put a little x next to him.	LOOPTEST
302.	And I tick BRIAN in my tree, I tick ADAM,	LOOPTEST

I tickCOLIN.

303. ERIC gets an x, because he didn't have a POLICERECORD. LOOPTEST
304. FRED gets an x, so nothing happens to him. LOOPTEST
305. I see that, I'm looking in my database who ADAM's FRIENDLY with, just to make sure my NEXTCASE loop gets them all. LOOPTEST
306. I did ADAM FRIENDLY COLIN, ADAM FRIENDLY FRED. LOOPTEST
307. FRED is not, does not have a POLICERECORD. LOOPTEST
308. So I do the NEXTCASE at 2AB. LOOPTEST
309. There is no NEXTCASE. LOOPTEST
310. So I go on to step 3. LOOPTEST
311. FOR EACH CASE OF X PAYS. LOOPTEST
312. ADAM doesn't pay anybody. LOOPTEST

LOOPTEST & SPECIFICATION (1ST, 2ND, 3RD AND 5TH SEGMENTS):

313. So I'm ignoring step 4 for the moment because it's so screwy and.... META
314. Step 5, THAT SEEMS TO BE THE WHOLE GROUP (12) IDENTIFIED. READ
315. Well there's my prediction that it's going to print out that stupid message a little too often because I'm down one level of recursion. IDENTIFY
316. So I'm going to do, having terminated COLIN and FRED, FRED doesn't get anything printed out about him but I then print out THAT SEEMS TO BE THE WHOLE GROUP IDENTIFIED. LOOPTEST
317. I'll just put THAT SEEMS dot dot dot on my sheet. META
318. So I get BRIAN, ADAM, COLIN.... LOOPTEST
319. And since I've hit my terminal node I'll print THAT SEEMS.... LOOPTEST
320. Now I pop up from, I'm popping up from IMPLICATing ADAM. LOOPTEST
321. Why didn't I go down any further with COLIN? ?
322. Because COLIN didn't have any.... LOOPTEST
323. Shit.... ?
324. I'm thinking THAT SEEMS etc should get printed out a lot more, at every terminal node here. SPECIFY
325. I'm sorry, everybody whom I print ISA CRIMINAL should also get THAT SEEMS etc about it. SPECIFY
326. So let me just check this again. META
327. I haven't been worrying about line 5 because it's uninteresting, but just for cosmetics. META
328. I mean, in fact I can see, IMPLICATE BRIAN, LOOPTEST

starting again from the top, I recursively go down to ADAM and I print ADAM ISA CRIMINAL.

329. And then I recursively go down to COLIN, print COLIN ISA CRIMINAL. LOOPTEST
330. And then just before I pop out I print THAT SEEMS TO BE THE WHOLE GROUP IDENTIFIED. Yeah. LOOPTEST
331. So thats, so COLIN ISA CRIMINAL, THAT SEEMS TO BE THE WHOLE GROUP IDENTIFIED. LOOPTEST
332. I pop out of COLIN because COLIN doesn't PAY or FRIENDLY anybody. LOOPTEST
333. So I pop out of COLIN. LOOPTEST
334. Now I try FRED. LOOPTEST
335. Sorry, I think about FRED for a second here, within ADAM, because he doesn't have a POLICERECORD I don't do anything with him. LOOPTEST
336. Finally I pop out of ADAM because there's no more NEXTCASE's. LOOPTEST
337. So I print THAT SEEMS etcetera when popping out of ADAM. LOOPTEST
338. Now I'm in my NEXTCASE loop with BRIAN. LOOPTEST
339. This time to get FRED because BRIAN PAYS FRED. LOOPTEST
340. Now I do this whole routine with FRED.... LOOPTEST
341. No I don't because FRED does not have a POLICERECORD. Good. LOOPTEST
342. So I draw a chain from BRIAN to FRED to COLIN, but that's only a sort of idealized chain along the PAYS and FRIENDLY relations. LOOPTEST
343. It would presuppose that FRED HAS POLICERECORD, that I can even get there. LOOPTEST
344. Because FRED does not have a POLICERECORD, that's a dead chain so I put an x next to FRED for the same reason I did the last time. LOOPTEST
345. I can even cross out that tree to COLIN which makes me feel happier. META
346. I was afraid that COLIN was going to be reached twice, but I was holding that to come back to. META
347. COLIN doesn't get reached twice, so, let's say within my BRIAN loop now, at the top level I'm going to say BRIAN PAYS FRED, CHECK FRED HAS POLICERECORD. LOOPTEST
348. He doesn't, so I'm going to print THAT SEEMS TO BE THE WHOLE GROUP again and pop out. LOOPTEST
349. So that's correct, so I have THAT SEEMS printed out for each level of recursion. SPECIFY
350. And I print it as I pop out so I get that corresponding to each CRIMINAL and I pop out. SPECIFY

CERTIFICATION & PATCH (DB, 2ND, 3RD AND 5TH SEGMENTS):

351. I'm now satisfied that the database is CERTIFY

correct, because I've changed it, changing COLIN FRIENDLY ADAM to COLIN FRIENDOF ADAM and adding ADAM FRIENDOF DAVID just for completeness.

352. I'm going to recommend to the student that he change 2AA and 3AA to NEXTCASE instead of EXIT and I'm going to suggest that line 5 be completely deleted because it's going to get him into trouble. PATCH
353. I'll delete line 5. PATCH
354. If I really wanted to have that line in I'd do it as a separate procedure. ADVISE
355. In other words I'd have a, let's say TO GET, TO GET X, one IMPLICATE X, which would set off this whole nonsense going. PATCH
356. And two, print ALL DONE or THAT SEEMS TO BE THE WHOLE GROUP IDENTIFIED. PATCH
357. So that would be just a cosmetic thing, and I could say GET BRIAN, do the whole chain that I've just traced out, and then when it's all finished it would just print out this message. ADVISE
358. But that's unimportant. I would recommend deleting step five. PATCH

READ, IDENTIFICATION, SPECIFICATION & PATCH (4TH SEGMENT):

359. Finally, step four, which I've saved until last because it's so bizarre. META
360. I haven't even thought about it. ?
361. It says 4 B CHECK STAR C PAYS. READ (11)
362. Well I would just suggest he deletes it. PATCH
363. It looks like.... ?
364. Either it's a typing error on Hank's part, which I doubt, I think it's deliberate just for this protocol. IDENTIFY IDENTIFY
365. I would say probably the student was pissing about. IDENTIFY
366. I know that in the old SOLO, it may be true in the new one, you could get a typing error like this. ?
367. The student was thinking of something wierd and he typed in, he thought he was on a subline of a FOR EACH, he wanted to do two things within that FOR EACH loop. ?
368. Let's say, hallucinating along with him, that he wanted to do the kind of thing to prevent the program looping. META
369. He wanted to CHECK the payment or something like that so he started to type it in and then he changed his mind and he, because META

	he typed 4B CHECK blah blah blah, the old SOLO, because it had an automatic space corrector and space inserter, would've split off the, put in a space between the 4 and the B and invented a procedure named B and ignored the rest of the line.	
370.	So he ends up with a procedure called B which takes three arguments which look like CHECK, C, PAYS.	SPECIFY
371.	But I attribute this to something the student had started to type in, changed his mind, and the old space corrector inserted a space and invented a procedure called B and that's just a wierd one.	SPECIFY
372.	The student, it would cause the student to freak out, I mean it's bizarre, and he probably couldn't remember how to delete it.	?
373.	I'd say, just kill it.	PATCH
374.	Well, I'd obviously ask the student what he was thinking, what he thought he was doing.	REFUSE
375.	I'd delete it.	PATCH

SUMMARY & ADVICE:

376.	The new procedure just has the old step one, the old step two, the old step three, the main ones.	ADVISE
377.	Four and five are deleted.	PATCH
378.	Change 2AA and 3AA to have a NEXTCASE instead of an EXIT.	PATCH
379.	And I believe that's it.	META
380.	I'd only ask the student at step one did he really want to PRINT it out or did he want to NOTE it in the database.	REFUSE
381.	Sometimes it's nice to NOTE it in.	ADVISE
382.	I mean it really depends upon what the student wanted to do.	META
383.	If you NOTE it in then you have the unfortunate consequence of having to clean up the database each time you run the example but if you're doing more complicated things then it's useful to have it in the database.	ADVISE
384.	So that's purely up to the student.	?
385.	So that's my corrections.	?

APPENDIX F

THE SPELLING CORRECTOR

MacSOLO's spelling corrector is essentially a series of filters which progressively reduce the size of the set of possible matches between the input "word" and the current dictionary. The latter is constantly updated as the user types in new node names, relation names and procedure names. The names of variables and parameters are not included in the dictionary, but at logout its current state is saved on the user's own file so that it can later be restored along with the rest of the user's database. The initial dictionary, supplied to new users, contains the names of all the SOLO system words, plus a few nodes and relations (new users are given a handful of preset database triples so that examples in the early part of the course notes will work straight away), plus a set of HELP topic names and common synonyms - such as HELP BRACKET for HELP PARENTHESIS. The size of this initial dictionary is 63 words, stored according to their first letters.

Our spelling corrector is an improvement on that suggested by Maguire in his (1982) paper on recognition of textual keyboard inputs, which was in turn based on the work of Muth and Tharp (1977). We agree with the first three of their four categories of spelling error (wrongly typed letter, missing letter, extraneous letter). However, their fourth category - two adjacent letters interchanged - refers to adjacent letters in a typed word, and is apparently caused by inexpert typists reversing the actions of their two hands during typing. A very similar mistake is caused by a slight shift in the position of the hands, and results in KEYBOARD ADJACENCY errors: for example, U or O instead of I. Our own analysis of spelling errors in Lewis's data shows that horizontal adjacency (along the same row of keys) is by far the commonest type of keyboard adjacency error.

We also agree with Maguire's assumption that the first letter of any typed word will be correct - so long as only novice users are under consideration. We would not agree that this is also the case for experts, or for novices who happen to be expert typists.

MacSOLO's corrector takes the word as entered by the user and immediately selects from the current dictionary the set of words whose first letter is the same as that of the input word. If, of course, the input word exactly matches one of the members of this set, the input is assumed correct and no further corrective action is necessary. The corrector is context-sensitive: owing to the simple nature of SOLO syntax (for example, any legal line MUST begin with an instruction - system supplied or user defined) it is possible to eliminate from the dictionary set all words which would not be legal in the current position. This increases both the corrector's speed and its hit-rate. It rejects from the remainder any members whose length is more than one character greater, or more than one character less, than the length of the input word. Taking each word of this final list in turn, it compares each with the input word, applying a series of filters to find the best match.

The filters check for the following possible errors (imagine the intended input word to have been MEMBER):

MEMMBER	duplicated letter M
MEBMER	reversed pair MB
MEVMER	keyboard adjacency V/B
MEBER	missing M
MEMEBER	extraneous E

The input word is compared with the current dictionary word essentially by successively matching corresponding pairs of letters. The filters are applied in sequence as shown

below. For each letter of the input word:

a) If it is the same as the current letter of the dictionary word, discard the current letter from both words and proceed to the next.
b) If it is the same as the previous letter of the dictionary word (if any), score one error, discard the current letter from the input word and return to (a).
c) If it is the same as the next letter of the dictionary word, whilst the current letter of the dictionary word is the same as the next letter of the input word, score one error, discard two letters from each word and return to (a).
d) If it is "adjacent" to the current letter of the dictionary word, score one error, discard one letter from each word and return to (a).
e) If the next letter of the input word is the same as the current letter of the dictionary word, score one error, discard the current letter from the input word, and return to (a).
f) If the current letter of the input word is the same as the next letter of the dictionary word, score one error, discard the current letter from the dictionary word and return to (a).

Only the lowest-scoring word or words from the dictionary remain in the list of possible matches at the end of filtering. Any dictionary words whose score rises above 35% are rejected immediately. The scores are expressed as percentages of the length of the input word (so many percent of its letters are wrong). The figure of 35% was chosen arbitrarily, but seems to work well in practice. It implies that no word of less than three letters can successfully be spelling-corrected (one wrong letter in three = 33%), and also implies that possible matches to any input word of less than six letters may share the same lowest score. In other words, selecting the lowest scorer(s) as above will sometimes result in several possible matches. And this of

course is a corrector "failure" - it does not recognise the input word as a properly spelt dictionary word, but also cannot suggest a unique dictionary word to replace it. However, many SOLO words are longer than six letters, particularly those used to denote relationships:

NOTE JOHN---LIKESPLAYING-->FRISBEE

so that this kind of failure is less common in a SOLO context.

At the end of filtering, and if there is still more than one candidate in the dictionary set, any members of the latter whose length is not exactly the same as the length of the input word are rejected.

In many cases this will result in a single candidate remaining in the dictionary set. When this is the so, the corrector will query the user, e.g. "When you typed FIOD, did you mean FIDO?". Maguire's system displays a menu of possible matches and invites the user to retype the correct one (leading to the possibility of a repeat mistake). MacSOLO proposes a single candidate match if any and requires only Y or N as the user's reply. If the user gives a positive reply, FIDO replaces FIOD; otherwise FIOD is accepted. But if after all filters have operated the dictionary set still contains more than one possible match (FOO1, FOO2, FOO3) the corrector will not take any further action.

At the start of 1983

the mainframe PASCAL-based SOLO was given a corrector based on these algorithms. But no empirical data as to its behaviour are yet available.

APPENDIX G

SIMPLE SYNTACTIC ERRORS.

This is the full list of simple syntactic errors trappable by MacSOLO. They are in approximate order of increasing complexity.

Specific triple already exists (top level or in STEP mode)
Specific triple not found (top level or in STEP mode)
Node not found
Wrong format
Attempt to redefine or edit system procedure
Procedure - use LIST
Node - use DESCRIBE
Undefined procedure - top level

Unrecognised character(s) removed
Unrecognised extra word(s) removed
Slash error
Quotes error
Parenthesis error
Spacing error
Spelling error

Missing line number
BYE/DONE confusion
SHOW/DESCRIBE/LIST confusion (EDIT mode)
EDIT function (such as RENUMBER) used from top level
DESCRIBE used on non-existent node
Not top level procedure (e.g. FOR EACH, LET)
Attempt to redefine existing procedure
Control statement error
First argument to LET not a node
Third argument to LET not a variable
No arguments given to INPUT

First input word not a procedure name
PRINT omitted before a string
Sublines inappropriate
Inappropriate use of subline syntax
DONE during subline
DONE/BYE confusion (EDIT mode)
Line number at top level
INPUT - wrong number of values
FOR EACH - no wildcard
Inappropriate use of parameter-name
Wildcard inappropriate
Undeclared parameter
Undefined procedure - run-time
Impermissible procedure name (e.g. AB - a reserved subline label)
Impermissible function (e.g. STEP, BYE) in procedure
Duplicated formal parameter
Too many procedure lines

Unbound variable - top level
Excessive FOR nesting
Nothing to undo
CHECK/TEST/FOR nesting not permitted
EDIT etc. used on a string
Inappropriate use of apostrophe