Tolerance to inaccuracy in computer programs

E. B. James and D. P. Partridge

College Computer Centre, Imperial College of Science and Technology, Exhibition Road, London SW7 2BX

This paper reports on the performance of a system for recovery from inaccuracies in the specification of program statements. The tolerance to inaccuracy is a natural by-product of the method of adaptive analysis, which is described in detail elsewhere (James and Partridge, 1973).

Probable inaccuracies are detected and accurate reconstructions are explored as an integral part of the process of analysis. Specific examples of inaccuracy are selected from practical program samples and the operation of the analyser is assessed with respect to them.

(Received July 1974)

1. The meaning of inaccuracy

It is difficult to define precisely what is meant by inaccuracy in a computer program. In the broadest sense, any property of a program which prevents it from achieving the objectives which the programmer has in mind will have arisen from some inaccuracy in specification. At a more detailed level, where there exists some accepted definition of 'normal' practice, it may be practical to talk of errors and their correction.

In program statement analysis, we are not aware of any generalised approach to dealing with inaccuracies other than those of a limited syntactic type. Most systems are limited to the detection of four simple types of inaccuracy: one character missing, one extra character inserted, one character wrong or two characters transposed. (Damerau, 1964; Morgan, 1970; Szanser, 1972). Another typical limitation is the expectation of only one mistake in each statement, or in each word in a natural language system. All the generalised systems mentioned aim at *recovering* from the errors detected: they aim to produce a syntactically acceptable structure which will enable the process of analysis to continue. This is in contrast to the idea of *correcting* the errors which can be described loosely as producing 'what the programmer meant' (Irons, 1963). Those systems which aim at correction do so by means of an ad hoc collection of special techniques devised largely on the strength of the system designers' intuition (Conway and Wilcox, 1971; Lafrance, 1971). None of the systems mentioned so far are designed with a view to adaptively improving their method of recovery to produce 'better' corrections.

The method of analysis which we describe here embodies no restriction on the number and type of inaccuracies which can be handled. It is a dynamic system which utilises past experience in the processing of previous 'correct' and 'incorrect' statements to improve its future performance. The use of previous experience enables the system to extend beyond the domain of syntactic inaccuracy, and it becomes possible to mark as suspect those statements which are syntactically correct but so unexpected as to make it seem likely that they have arisen from an error in the specification of a more common type of statement.

2. The adaptive analyser

The adaptive analyser is described in detail in James and Partridge (1973) and Partridge (1972) and only particularly relevant features of the system are outlined here. In the adaptive analyser we have extended the normal process of syntactic analysis in several directions. The general principle is that the tree structure defining the syntax of the language is no longer static but *dynamic*, reflecting changes in the pattern of language usage by variations in the structure. An initial structure is provided and is altered in many different ways as a result of the continuing process of recognising the incoming statements. At any particular point in time therefore, the structure of the tree reflects the total experience of the process of analysis, and it is convenient to call it 'the experience tree'.

2.1. *Improving the experience tree structure*

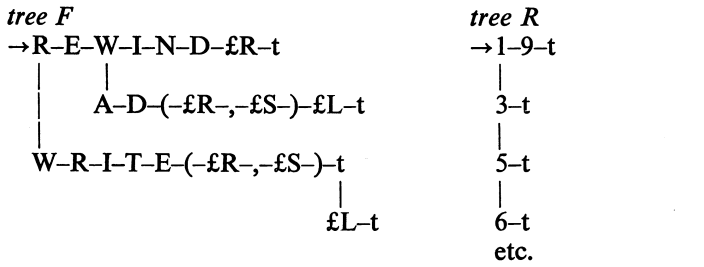
The heuristic principles that are used to restructure the experience tree were designed to exploit the non-uniform use of language features by programmers. This non-uniformity proves to be stable over time, as in the case of natural language use and so the likelihood of future occurrence of the various possible structures can be predicted with a high degree of accuracy. This means that as soon as the experience tree has been altered sufficiently to mirror usage in a particular environment, subsequent alterations that may be necessary will have little effect on the overall processing efficiency. The structure of the experience tree is also adjusted to reflect the fact that all statements contain a certain degree of redundant information which can be ignored if we are to speed the analysis of accurate and 'trivially' inaccurate statements, or utilised if required to assist in the interpretation of defective statements.

2.2. *The automatic restructuring mechanisms*

The restructuring of the experience tree is aimed at minimising the complexity of the total matching process, and so increasing its efficiency. There are two basic restructuring mechanisms.

The 'branch swopping' mechanism The principle behind this technique is that given a series of alternative structures in the language, those that have occurred most frequently in the past are looked for soonest in the future. This can be realised by simply ranking the sets of nodes which represent alternative structures in the experience of the tree so that those most frequently required are nearest the root of the tree and therefore are reached most quickly in the matching process. Naturally, when a set of alternative nodes are reordered in this way, their attachments to subsequent nodes remain undisturbed. All sets of alternative nodes throughout the experience tree are thus ranked in order of precedence with respect to the analysis process.

The 'promotion' mechanism This mechanism can be viewed as an inter-tree branch swopping mechanism. If the few highest precedence items on any tree within the tree-structured hierarchy (the experience tree) are accessed on the vast majority of references to that tree, then these items can be automatically 'promoted' up to a higher level in the total hierarchy, thereby avoiding the time consuming recourse down through the hierarchy via the appropriate references on the occasions when these particular 'promoted' items are accessed. The price to be paid is an increase in the total size of the hierarchy.



key: an upper case character represents an immediate symbol
'£R' is a reference to tree R
't' is a terminal node

Fig. 1

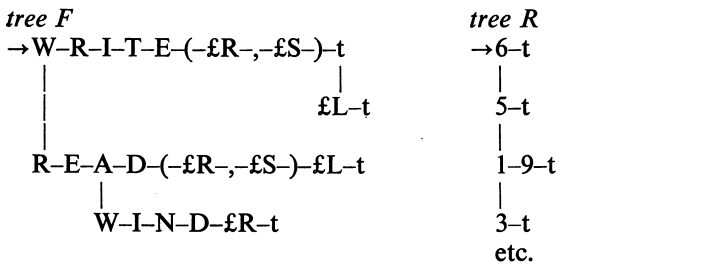


Fig. 2

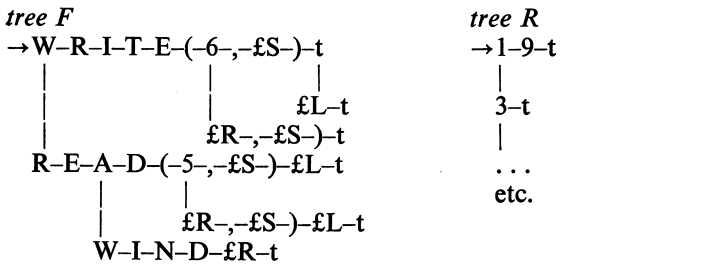


Fig. 3

As an example of these two mechanisms consider the trivial hierarchy of **Fig. 1,** that can match the FORTRAN statements, READ, WRITE and REWIND (tree *F*), all of which require a reference to some input or output device number. These numbers comprise tree *R*. Tree *S* (for statement numbers) and tree *L* (for lists of variable names) are not illustrated. Alternative nodes are connected by vertical lines and successive nodes by horizontal lines.

If the analysis process revealed that the WRITE statement occurred more frequently than the READ statement; and the READ statement more frequently than the REWIND statement and that within tree *R* the item '6' was referenced most frequently, followed by the item '5'; then the branch swopping mechanism would restructure the individual trees as illustrated in **Fig. 2.**

In many installations the usual form of the WRITE statement is WRITE (6 etc. and the READ statement is READ (5 etc. The REWIND statement is seldom accessed, and the vast majority of references to tree *R* match either item '6' or item '5'. In this case, the promotion mechanism would restructure the hierarchy as illustrated in **Fig. 3.**

The matching of the two commonest particular statements, 'WRITE(6, etc.' and 'READ(5, etc.', will be quicker when utilising the tree hierarchy of Fig. 3 than when using that of Fig. 1. The statement beginning, 'WRITE(6 etc.', matched up to the '6' without querying any wrong nodes, as opposed to 4 wrong nodes being processed when utilising Fig. 1. These are the root node of tree *F* containing '*R*' and the first three alternative nodes in tree *R*, containing '1', '3' and '5'. The indirect reference to tree *R* (the node containing '£*R*') does not have to be processed either.

A further important point is that the promotion mechanism is context sensitive: a '6' is promoted into the WRITE statement and a '5' into the READ statement. As we shall see, this is important for the production of good error recoveries as well as the efficient analysis of correct input and output statements.

2.3. *Levels of confidence and the control of statement analysis*

The improving techniques described above imply that there is available a count of the previously successful match frequencies. An obvious place to store and maintain these 'frequency counts' is within the experience tree itself. Thus frequency counts are stored throughout the experience tree, and a frequency count is stored and associated with each item in every tree and subtree that comprises the total hierarchy.

Apart from the use of 'improving' strategies there is an important further use of the frequency counts. They are utilised during the matching of incoming statements to provide a measure of the relative 'likelihood' of success with any particular substructure—or from another viewpoint, a measure of 'confidence' that any particular substructure will prove to be correct. Thus there is available at all times during the processing of a statement, some value representing the confidence in an overall successful match which can provide a criterion for continuing or abandoning the current matching process.

Consideration of these confidence levels has a useful, if not essential part to play in the production of an efficient system which can 'learn' to analyse statements with an arbitrary number of inaccuracies given sufficient learning and processing time.

By neglecting the low likelihood structures within the language we gain not only by directing the interpretation of an incorrect structure to the high likelihood possibilities—but also from the increased efficiency which stems from the effective reduction in size of the working definition of the language to be analysed.

3. Dealing with inaccuracy

3.1. *Structural redundancy and trivial inaccuracy*

It is a fundamental assumption of the work described here that we do not expect perfect specification of the incoming statements. Prime concern is with extracting the 'meaning' of the statements analysed which implies that within the matching process the fit is not necessarily or even usually perfect. The fact that statements can be 'made sense of' (matched) without a perfect fit means that there must be a certain amount of redundancy within them.

As a result of our approximate matching techniques, inaccuracy does not have quite the usual meaning within our analyser. We are not satisfied merely with classifying the incoming statements into one of the two mutually exclusive classes, correct or incorrect, which then invoke the normal processes of syntactic analysis (and later compilation) or error processing techniques respectively. In our system, trivially incorrect statements can be processed, that is, the critical features (the ones that convey the essential meaning) can be extracted with no loss in efficiency over the processing of completely correct statements. A logical consequence of the high efficiency of this technique is that we are not able to indicate the occurrence of trivial mistakes in the incoming statements.

The realisation of this approximate yet discriminating matching is seen in our 'confidence jump' technique. This mechanism quite simply exploits the structural redundancy of the particular language definition that comprises the experience tree. This tree structure definition makes any structurally redundant features of the language conspicuous as nodes that specify immediate symbols and are not associated with any alternative structure.

When analysing the incoming statement, we have at any particular point in the process a current input statement char-

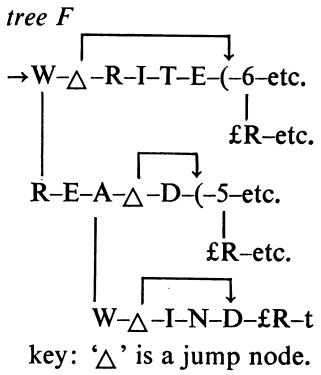


Fig. 4

acter to be matched against an immediate symbol associated directly with or referenced from a current tree node or one of its alternatives. If the current tree node contains an immediate symbol and has no alternative nodes associated with it, then it can be assumed that the match has been successful and controls transferred to the next tree node and input statement character. Clearly this mechanism extends easily and more efficiently to longer sequences of structurally redundant symbols.

As an example, consider the partial syntax tree in Fig. 3. Nodes *R*, *I*, *T*, *E*, and '(' of the top branch are all structurally redundant. If the first character of an input statement is a '*W*' and thus is matched against the first node in the tree then (given the language description of Fig. 3) either the statement is a WRITE statement or it is incorrect. If it is assumed that the statement is a WRITE statement, control is transferred to node 6 in tree *F* and to the appropriate position in the input statement, *without checking* that the intervening characters are '*R*', '*I*', '*T*' and '*E*'.

A special tree node—a 'jump' node—controls this mechanism. Jump nodes are automatically inserted throughout the experience tree. Tree *F* of Fig. 3 with three jump nodes inserted is illustrated in **Fig. 4.**

During the analysis of a statement, when control is passed to a jump node, a decision is taken whether to pass control along the jump path or whether to pass control to the successor node and continue processing scrupulously. The decision is based upon the level of confidence of the total matching of the current statement.

Thus a trivially incorrect statement is defined by the scope of the jump nodes, which is a function of: jump node position, likelihood of current experience tree structure, likelihood of current general strategy (see below) and 'smoothness' of prior matching with the current input statement.

Assuming the total level of confidence is high enough with the experience tree of Fig. 4, the following statements would be trivially incorrect and thus analysed with no loss in efficiency over the completely correct statements:

WITE (6, 10) W(6, 10) WRIGHT(6, 10) .

3.2. *Non-trivial inaccuracy and the strategy list*

The method of analysis described has two components: the experience tree, and an algorithm which matches the incoming statements against this tree structure. The structure of the experience tree is continually optimised and so it seems reasonable to attempt a similar optimisation of the matching algorithm.

Unlike the majority of statement analysers, our system does not immediately consign seriously inaccurate statements to an error exit. Instead we attempt to 'force' a match against the experience tree by modifying the matching algorithm. The various modifications or 'strategies' will in general be different for dealing with different types of error. We can arrange these different strategies in an ordered list similar to sets of alternatives within the experience tree and there seems no reason why the 'strategy list' should not be manipulated in a similar manner. In this way successful strategies can be promoted so that they are tried more often and unsuccessful strategies can be demoted or deleted.

General strategies that might prove useful are, 'assume one character has been omitted from the input statement', or 'assume two successive characters have been altered within the input statement'. One further feature at least would seem to be necessary in the modification of a matching strategy, that is the position within the input statement and the experience tree from which the force match should be initiated. For it is by no means necessary that the error occurs at the position in which it is first detected.

The confidence levels also have a significant contribution to make to the process of recovery when using the force matching technique. The matching strategies are arranged, and will therefore be tried, in order of decreasing likelihood and hence also in decreasing confidence that they can effect a useful correction; the confidence of the matching process at any particular point would seem to be some function of the confidence level of the particular strategy and of the particular experience tree substructure. Thus an attempted matching could be abandoned when some function of these two levels diminishes to some present threshold value and a match has not been obtained. Then, although the force match will be continued with a less likely strategy, the chance that a fit might be obtained on a high likelihood substructure of the experience tree could result in an overall higher confidence correction than would have been obtained by pursuing the previously abandoned matching.

By using the confidence levels in this manner we expect to produce high likelihood corrections which are superior to the initial results of the system, when it has not learned the pattern of language use.

3.3. *The introduction of allowable error*

At all times during the processing of a statement there is available a measure of the likelihood of every structure within the experience tree and of every general strategy within the strategy list.

The possibility arises that a high likelihood correction might be a better result than a very low likelihood analysis with no modification. In other words, very low likelihood syntactic structures might be usefully interpreted, if possible, as high likelihood structures with one of the high likelihood errors. For example, the FORTRAN statement

WRITE (U, 32)

is syntactically correct but very unlikely, and is possibly better interpreted as:

WRITE (6, 32)

which is a very likely statement and only differs from the original in one character, and 'one character wrong' is a very common error.

Thus we have the possibility of simplifying the total analysis and increasing the scope, quality and efficiency of error correction by neglecting the possibility of occurrence of very unlikely structures within the language, as a first approximation. This procedure implies the possibility of making mistakes when the system fails to recognise very unlikely structures when they actually *do* occur.

We may consider such a procedure as a particular example of the more general feature, 'allowable error' (after Bloom, 1970). This is defined as the small proportion of permitted errors introduced into an analysing system to gain a significant increase in the overall processing efficiency of the system. Clearly such a procedure could be fraught with danger and adequate safeguards must be developed concurrently. For

example, we must be able to accommodate a sudden upsurge in the popularity of a previously uncommon feature.

We are now investigating the possibility of a compromise between increasing the efficiency of the analyser and decreasing the number of mistakes that result. The usual view is that the processing efficiency of a system should be improved only so far as to be consistent with maintaining the theoretical number of mistakes that the system can make at zero; no 'allowable error' can be permitted. We relax this stipulation, and consider methods of analysis which admit positive amounts of allowable error to assess the practical feasibility of such a move.

The particular method by which allowable inaccuracy is introduced into the system can be considered as an exploitation of a particular type of redundancy, *statistical* redundancy. Substructures within the experience tree are, to a first approximation, ignored when the defined set of alternatives in the language structure which they represent have been found to occur in 'statistically insignificant' proportions. In practice, for a particular application, 'statistically insignificant' will mean substructures associated with frequency counts below a certain value. This method of introduction of allowable error can be incorporated naturally into the confidence jump mechanism, since that mechanism exploits the *structural* redundancy within the experience tree and clearly, statistically redundant nodes can be treated in exactly the same manner. For the purpose of automatically inserting jump nodes within the experience tree, structural redundancy is considered as a limiting case of statistical redundancy. Structural redundancy *is* statistical redundancy when 'statistically insignificant' takes the value of zero. In the current implementation, 'zero' actually implies a frequency of less than 1 in 10,000.

4. Practical results

The adaptive analyser can be applied to the analysis of statements in any programming language. We chose to analyse programs written in FORTRAN since they were most easily available. A hierarchical description of standard FORTRAN was constructed and applied to 200 programs which were drawn from five different environments. In all 20,121 FORTRAN statements were analysed. The frequency of occurrence of errors was 0·3 per cent; that is three incorrect statements in every thousand. This figure appears to be surprisingly low until it is recast as one statement in every three programs, which probably implies that over one third of the sample programs would have been rejected by a conventional computing system.

A total of 49 statements contained non-trivial syntactic inaccuracies. Of these statements 29 contained a single error which fell in one of the three classes commonly anticipated within error correcting systems: one character wrong, one character inserted and one character omitted. Surprisingly, there were no occurrences of two characters transposed, which is the fourth class of errors commonly catered for in existing systems. Although only 59 per cent of incorrect statements contained a single error involving a single character, more than 90 per cent of the isolated errors were of the single character type. The frequency within each of the three classes of error was: 32 characters missing, 13 characters wrong and 10 characters inserted. 10 statements contained more than one isolated error, whilst in eight statements the error involved a syntactic unit rather than individual character mistakes. We have been unable to find any comparable statistics on errors in computer programs. Most previous work does not include any results of testing a system in depth, but rather the results of a few selected examples. The statements concerned with input and output operations—FORMAT, READ and WRITE statements—account for 75 per cent of incorrect statements, and the assignment statement contributes a further 10 per cent.

Within the total hierarchy, the principal tree *F* specified the overall structure of each of 36 FORTRAN statement types. This does not include the assignment statement which is dealt with separately. 32 jump nodes were automatically inserted within tree *F*, with no allowable error introduced (i.e. only structural redundancy exploited). 100 nodes or 10 per cent of the total experience tree was found to be structurally redundant. Very few trivial errors were encountered, but those that were such as the misspelt keywords 'DIMENTION' and 'FORMIT' were correctly interpreted with no loss in efficiency in comparison with the totally correct versions. An allowable error of less than 0·01 per cent was introduced to exploit the statistical redundancy within tree *F*. Then only 17 jump nodes were inserted but one third of the possible statement types (that is, 12) were effectively removed from the language, as a first approximation to practical usage.

Backspace 0 characters on statement being analysed and attempt to match assuming 0 characters *omitted* whilst expected frequency of occurrence not less than zero (the 'correct' strategy)
↓
Backspace 1 character and search assuming 1 character *wrong*
↓
Backspace 1 character and search assuming 1 character *inserted*
↓
Backspace 2 characters and search assuming 2 characters *wrong*
↓
Backspace 0 characters and search assuming 1 character *omitted*
↓
Etc.

Fig. 5 The strategy list

Restructuring the strategy list and the utilisation of confidence levels in the restoration of non-trivial errors

As described earlier, statements containing non-trivial inaccuracies are analysed by utilising the general strategies in order of precedence from the strategy list until a match is obtained or a complete failure is reported.

The strategy list illustrated below (**Fig. 5**) was utilised in the analysis of Data Set I (a batch of 8164 statements) and although 40 per cent of the incorrect statements were restored, none of the restorations were 'correct'. All but one of the restorations were performed by the second general strategy. This is the first correction strategy since the first strategy matches only correct statements and was given the highest precedence whilst the other strategies were ordered randomly. This particular strategy was always employed because the errors were detected within subtrees which presented several alternatives which were equally likely on the basis of confidence levels. For instance, when a comma is omitted from between two variables the confidence levels within the tree *U* which is the tree that can match a valid variable name, are not sufficiently dissimilar to dictate which of the possible restorations is 'correct'. So in each case the second general strategy is able to produce a syntactically correct structure. The alternative strategies employed are given in Fig. 5.

The first strategy does *not* modify the processing of incoming statements as directed by the experience tree, and the optimisation that can be produced by neglecting the low likelihood experience tree items is not being used: all the strategies continue the search for a correct match in the tree until the expected frequency is 'zero', or less than 1 in 10,000.

The errors detected and restored 'incorrectly' by the method just described during the analysis of Data Set I, were analysed by hand and the strategy tree was restructured to place the

backspace 0 characters, and search assuming 0 characters omitted whilst frequency not less than zero
↓
backspace 0 characters, and search assuming 1 character wrong whilst frequency not less than 100
↓
backspace 0 characters, and search assuming 1 character inserted whilst frequency not less than 125
↓
backspace 1 character, and search assuming 1 character omitted whilst frequency not less than 143
↓
backspace 0 characters, and search assuming 1 character omitted whilst frequency not less than 166
↓
backspace 2 characters, and search assuming 1 character omitted whilst frequency not less than 200
↓
Etc.

Note: No allowable error introduced because the first strategy searches all of the experience tree.

Fig. 6 The strategy list of Fig. 5 restructured as a result of analysing Data Set I

appropriate strategies in order of precedence. Further, a simple linear function of the error frequencies was incorporated in the decision mechanism to direct the force matching to the more likely experience tree items. The restructured strategy list given in **Fig. 6** was utilised in the analysis of Data Sets II to V. The results illustrate a significant proportion of 'correct' modifications, and strategies as far down as the tenth actually performed modifications.

A more direct comparison of the two strategy lists was made with the incorrect statements show in **Fig. 7,** all of which were found within the analysed Data Sets. The restructured strategy list of Fig. 6 is clearly better in performance than the random list of Fig. 5.

Another feature of the introduction of allowable error by mean of the confidence levels is that we can detect inaccuracies which are syntactically correct but nevertheless not 'what was meant'. A portion of these errors, as described earlier, become apparent to the analyser when the analysis is directed by an optimised experience tree, for they match statistically redundant items within the tree.

Seven such statements were detected within the 8,164 statements that comprised Data Set I.

For example, a key punching mistake gave the syntactically correct 'WRITE (0, 2001) L3, N3', which was detected as unlikely as soon as the minimum allowable error (0·1%) was introduced into the first matching strategy. In a practical system it would clearly be useful to flag such a statement as suspect. The system was able to 'recover' most incorrect statements encountered and the use of confidence levels and the strategy list certainly improved the quality of these 'recoveries'.

As all error correction is based on the provision of redundant information whose consistency can be checked, the FORTRAN language as defined is not particularly amenable to error correcting techniques. Nevertheless, the system was able to recover most incorrect statements encountered and by the use of extra information gained from past experience and statement context, was able to effect a large proportion of 'corrections'. A further source of extra information, which our analyser discarded, is the blank characters that commonly occur within program statements. In retrospect it appears that within a significant number of incorrect statements, 'delimiting' blanks indicated the correct structure. Although the system described was able to detect a (presumably) small portion of non-syntactic errors, any significant further advance in this direction requires extension of the hierarchical language description from individual statement structure to total program structure. The current implementation deals only with the internal structure of individual statements, and not with interstatement relationships.

Finally, a few words in defence of the usual charge that tolerant computer systems encourage sloppiness in programmers. We consider most current computer systems as too rigorously pedantic, and so a few carefully considered steps in the direction of relaxing some of the constraints customarily imposed upon programmers does not automatically warrant the charge of sloppiness. In projects which require large amounts of data which will inevitably contain trivial mistakes, it is perhaps a case of coming to terms with features such as imprecision of

	Incorrect statement	*Fig. 5 strategy*	*Fig. 6 strategy*
1.	IFINISH = 0	failed	IFINIS = 0
2.	FORMAT(1X, FK.2)	failed	FORMAT(1X, F1·2)
3.	READ(5, 21)	failed	failed
4.	EDASH = 1.*COS (M)*E)	failed	failed
5.	WRITE(6, 20	failed	WRITE(6, 20)
6.	FORMAT(4H NAME 12X/X	failed	FORMAT(4H NAM 12X/1X)
7.	FORMAT(1X, F10·0/)	failed	FORMAT(1X, F10·1/)
8.	FORMIT(X 22A6)	FORMAT(1X, 2A6)	same
9.	M = IFINISH/2	M = IFINI + H/2	M = IFINIS/2
10.	DO 99 I = ,M	failed	failed
11.	K(J + 1 − 2) = K(J)	failed	K(J + 1, 2) = K(J)
12.	READ(5, 24)KPLUS WAYB	..(..)KPLUSW, AYB	..(..)KPLUSW, YB
13.	FORMAT(I3, X, F9·5)	failed	FORMAT(I3, /, F9·5)
14.	WRITE(6, 21), K(1)	failed	WRITE(6, 21)IK(1)
15.	WRITE(60, 25) (J, K(J), J = 1, M)	failed	WRITE(6, 25) (J, K(J), J = etc.
16.	K(1) = (KPLUS + 3) 14	K(1) = (KPLUS + 3)/4	same
17.	96(M.LT.2)KPLUS = 2	IF(M.LT.2)KPLUS = 2	same
18.	FORMAT (16H-FILE LENGTH I2/(5X 19A1))	failed	..../(5X, 9A1))
19.	GO TO (11, 12) M	GOTO (11, 12), M	same

Fig. 7 The performance of the strategy trees of Figs. 5 and 6

specification and the resulting action or imposing a severe block upon the type of problem that can be solved.

Szanser (1972) states that the Machine Translation project of the National Physical Laboratory constantly ran up against the problem of coping with errors in the input, 'even if the error was trivial by human standards (such as to pass unnoticed —obvious evidence of the existence of 'automatic error correction' in the brain)'.

Again we would emphasise that our position is not to let programmers 'run riot' with the system. A well designed tolerant system will naturally provide incentives to encourage precise programming in that 'correct', that is expected program statements, will be processed faster. It will exploit the redundancy in program specification to increase the chances of recovering from inaccuracy while it will not penalise the correct programs.

References

BLOOM, B. H. (1970). Space/time trade-offs in hash coding with allowable errors, *CACM*, Vol. 13, No. 7, pp. 422-426.

CONWAY, R. W., and WILCOX, T. R. (1971). *Design and implementation of a diagnostic compiler for PL/I*, Research Report 71-107, Department of Computer Science, Cornell University, September 1971.

DAMERAU, F. J. (1964). A technique for computer detection and correction of spelling errors, *CACM*, Vol. 7, No. 3, pp. 171-176.

IRONS, E. T. (1963). An error-correcting parse algorithm. *CACM*, Vol. 6, No. 11, pp. 669-673.

JAMES, E. B., and PARTRIDGE, D. P. (1973). Adaptive correction of program statements, *CACM*, Vol. 16, No. 1, pp. 27-37.

LAFRANCE, J. (1971). *Syntax-directed error recovery for compilers*, Ph.D. thesis, University of Illinois at Urbana-Champaign.

MORGAN, H. L. (1970). Spelling Correction in systems programs, *CACM*, Vol. 13, No. 2, pp. 90-94.

PARTRIDGE, D. P. (1972). *Heuristic methods in the analysis of program statements*, Ph.D. thesis, Department of Computing and Control, Imperial College, University of London.

SZANSER, A. J. (1972). *Automatic error correction in natural texts*, National Physical Laboratory, Part II COM 52 and Supplement COM 63.

Book reviews

FORTRAN to PL/I Dictionary, PL/I to FORTRAN Dictionary by Gary De Ward Brown, 1975. (*John Wiley & Sons, New York*, £5·10).

This book explains FORTRAN and PL/I in terms of each other. The first section consists of alphabetic lists in the style of English-French and French-English dictionaries but the major part of the book is taken with explaining the concepts of the languages from basic statements through data storage and control statements to input/output and debugging aids with where there are differences, as there usually are, FORTRAN to the left and PL/I to the right of the page. There is a chapter on features which have no parallel in FORTRAN, such as multitasking and recursion, and there are appendices on PL/I character sets and abbreviations, interlanguage communication on the IBM 360/370 and answers to the exercises which terminate each chapter. The chapters are ordered as for a reference, not an initial teaching work. There is a good index.

It is assumed that the reader is familiar with the basic concepts of programming, punched cards, and so on and it is visualised that the main use of the book would be to teach one language to a person who knows the other, or to act as a reference for such a person. It is also suggested that the book could be used as a text for a course in which languages are learned in sequence or as a text for a course in comparative languages, or as a language reference.

Of these multifarious targets the one missed by the greatest distance is the function of the single language reference. The FORTRAN covered is described as all of ANS plus 'most of the widely-used non-ANS FORTRAN IV features' including 'those of WATFOR and WATFIV'. What is actually covered is all of ANS plus most of the IBM 360/370 FORTRAN extensions plus a few references to language fragments from other systems, particularly CDC 6000. The earlier chapters are concerned mainly with IBM facilities but later there is more acknowledgement of other systems. The reader, inevitably, must look elsewhere to decide if a particular statement is acceptable to a particular language processor. Similarly the PL/I is unequivocally stated to be IBM version 5 level F with additional features of the IBM checkout and optimising compilers but by Chapter 7 there is an exception indicating a difference between IBM 360/370 and the rest. The book therefore attempts to reference a phantasm.

Further, one must demand accuracy of a reference work and it is not acceptable for example in FORTRAN to use a variable name for both a real and a logical variable in the same statement (p. 48) or to describe the skip/format code as Xw with examples X6 and X10 (p. 127). These are possibly due to printing errors; the general standard of presentation is high, especially given the split-page format, though there are occasional lapses such as using the wrong type face for a word and there is the common error of sometimes continuing a program statement across lines as if it were a natural language sentence.

These minor infelicities will not trouble the experienced programmer who will probably not even notice that page 1 contains an elementary error, a near-illegible example and a spelling mistake. In its basic function of setting out the two languages side by side and line by line with abundant examples the book is excellent and one can imagine it being of great utility to the programmer who is familiar with one of the languages and is rusty at the other and who is prepared to crosscheck with the manual of the processor he is using. *A fortiori*, for comparative language study the book can be used very easily to give an overview of the facilities offered by the two languages.

Those who have read thus far will have gathered that the third of the five aims of the book, to teach the languages in sequence, is not really fulfilled as there are better ways for those who must learn FORTRAN and PL/I from scratch. The author uses the analogy of an English-French dictionary: to give this book to an inexperienced programmer would be to invite translations of the level of 'lettre Française'.

D. T. MUXWORTHY (Edinburgh)

Elementary Algol by A. Brundritt, 1976; 80 pages. (*Macdonald & Evans*, £1·25).

This book may be ideally suited to students on Alan Brundritt's ALGOL courses. I would not recommend it to anyone else. The order in which the language is taught is completely at variance with modern ideas of programming: **goto** is taught early and used heavily throughout the book; and although the safer control statements are introduced, they are badly illustrated (**for** $j := 1, j + 1$ **while** $j > 0$ **do** . . . . . with a **goto** to leave the loop, in one example!). **Comment** is included as an afterthought in the last chapter—it might have helped to explain examples like the one above.

Both input/output and Job Control Language are described without mention of their ICL 1900 dependencies, an omission which is made more grave by the promise in the Introduction that such dependencies would be 'made clear'.

The style is uncomfortably chatty—acceptable in a lecture, but irritating and obtrusive in print.

'There may not be many cheaper books than this, but there are better ones'. Day (1976).

A. C. Day, reviewing Computer Programming/FORTRAN in *Computer Bulletin*, series 2 no. 7 March 1976, page 40.

ANNE ROGERS (Bath)