RECEIVED BY DTIC OCT 23 1970

i

MASTER

USER ERROR ANALYSIS AND AUTOMATIC
CORRECTION FOR COMPILING

Ph. D. Thesis Submitted to Iowa State University

G. E. Hedrick and R. J. Lambert

Ames Laboratory, USAEC

Iowa State University
Ames, Iowa 50010

PREPARED FOR THE U. S. ATOMIC ENERGY
COMMISSION UNDER CONTRACT NO. W-7405-eng-82

Date Transmitted: October 1970

LEGAL NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Carl

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

IS-2411

LEGAL NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

Printed in the United States of America

Available from

Clearinghouse for Federal Scientific and Technical Information

National Bureau of Standards, U.S. Department of Commerce

Springfield, Virginia 22151

Price: Printed Copy \$3.00; Microfiche \$0.65

USER ERROR ANALYSIS AND AUTOMATIC
CORRECTION FOR COMPILING

George Ellwood Hedrick, III

ABSTRACT

The primary research which is presented in this thesis is a basic theory for the development of an error corrector. The use of this theory should provide one additional debugging tool for the high-level language programmer. Such tools have been few and it is hoped that this theory can be developed to the point where it is useful to the majority of programmers.

Techniques are developed for automatically determining the existence of errors, locating errors, and choosing the most probable correction. Certain new terms are defined in order to facilitate the discussion of the error correction problem. Methods for determining the existence of errors rely heavily on present compiling techniques. Errors are located by using techniques which are modifications of compiling techniques; for example, the translation matrix technique may be used to find the place in the input string where an error has actually occurred. Errors are classified and heuristic techniques are developed for each class of errors. Some of the classes of errors such as spelling errors may be corrected by previously known techniques, but entirely new methods are developed for most classes.

An error corrector was developed for a selected subset of high-level language programs which use the Waterloo FORTRAN [WATFOR] compiler. The corrector makes use of the information which is available to the compiler and the output from the compiler.

The corrector was implemented using the following steps:

- 1) a random sample of programs which use the WATFOR compiler was obtained;
- 2) statistics were gathered so that programs could be described in terms of the errors that programmers make;
- 3) correction algorithms were developed for the most common errors;
- 4) the algorithms were coded as a correction program; and
- 5) the correction program was applied to the WATFOR job stream.

TABLE OF CONTENTS

	Page
ABSTRACT	iii
I. INTRODUCTION	1
II. BASIC DEFINITIONS	24
III. FACTORS WHICH INFLUENCE THE DESIGN OF THE SOLUTION TO THE ERROR CORRECTION PROBLEM	32
IV. FACTORS WHICH INFLUENCE THE IMPLEMENTATION OF AN ERROR CORRECTOR	46
V. THE DESIGN AND IMPLEMENTATION OF AN ERROR CORRECTOR	61
VI. STATISTICAL PROCESSING FOR AN ERROR CORRECTOR	82
VII. RESULTS, SUMMARY, AND SUGGESTED FUTURE WORK	95
VIII. BIBLIOGRAPHY	97
IX. ACKNOWLEDGEMENTS	102
X. APPENDIX A	103
XI. APPENDIX B	120
XII. APPENDIX C	203

I. INTRODUCTION

Although compiling has long been a popular research topic among computer scientists, error analysis and correction for compiling has seldom been approached. In the past, what error analysis and correction that has been done was incorporated into compiler writing yielding relatively inefficient compiling and minimal error correction. In this dissertation a new approach is taken: the error analysis and correction problem is looked at by itself without some of the problems which are inherent in compiler writing. The techniques, which are developed in the following chapters, isolate sets of errors, attempt to correct them, and reinvoke the compiler for the altered source code. Once the new techniques have been perfected then it will be feasible to incorporate them into standard compilers.

The incorporation of error analysis and correction techniques into compilers for the standard high-level languages should concentrate greater power in the compilation phase of programming, thereby decreasing the amount of user effort necessary for obtaining a working program.

The current compilers for existing languages are designed in different ways but have a minimum of error analysis and correction. Most syntax-directed compilers simply discontinue compilation at the point where a syntax error is detected. Those syntax-directed compilers which do not give up quite so easily usually require large tables to be defined or require that compilation proceed in an apparently parallel fashion. Compilers which require large tables include those which use the techniques

of Floyd (17,18)* and the techniques of Wirth and Weber (63,64). The apparently parallel technique was developed by Irons and presented in his 1963 paper (34). All of these techniques are discussed in detail later in this chapter. The apparently parallel technique does require that the algorithm for compilation be logically more complex than the standard syntax oriented compiler.

Most compilers are not syntax oriented; that is, they are heuristically developed rather than making use of the formal definition of the syntax. This type of compiler encounters similar problems. In spite of the fact that the error analysis and correction is better in this case, anyone who has ever used one of the more common FORTRAN compilers knows the frustration of, for example, an omitted DIMENSION statement. This one error may generate several errors such as:

- 1) arithmetic statement function out of sequence;
- 2) improper use of subscripted variable; and
- 3) missing subprogram.

As soon as some of these problems with the compilation errors have been remedied, then the computing power of the computer installation which takes advantage of these solutions should be boosted considerably. This will not be a result of a faster machine or of a faster compiler (In fact the compiler utilizing these methods will probably execute slower than it had previously.), but it will be the result of the fact

* The numbers in parentheses refer to the bibliographic entries.

that the compiler will have performed a large portion of the work currently being done by programmers. This means that less time would be required for each computer program to make the transition from the preliminary analysis phase to the production phase.

One of the problems which arises with the increased sophistication of the error analysis and error correction is that of economics; in order for the method to be truly useful, a realization of the methods must be found such that either the compilation time is competitive with present compilers, or the compiler be invoked only when it is known that errors exist. This is particularly true in a type of environment such as a university where most of the research and student work never reaches the production phase. Unless a way to increase the performance of compilers that implement the proposed techniques can be found, these compilers will be unavailable to those who need them most -- the researchers and the students.

In order to start this research certain basic definitions are required. The research commences with formal definitions of the different types of error. These definitions are then used as a base from which to develop definitions for error spaces. Certain observations are then made about the nature of error spaces and some computational kernels are developed. These computational kernels, which apply specifically to this type of non-numeric processing, can then be used in the development of computational techniques for error correction.

The research continues with an informal description of possible solutions to the error correction problem. This particular phase may be considered to be in the area of preliminary analysis. Following this

preliminary analysis one method is formalized for later implementation. After the formalization, an analysis of it is made as a preparation for future implementation. Finally, an error corrector is developed. This error corrector obeys the principles which were previously developed but is applicable only to one language. Examples show that the error corrector indeed does work in many cases.

The skeletal error correction problem is: given that an error, ϵ_1 , exists at location a_1 , change the text at a_1 such that ϵ_1 no longer exists and that no new errors, ϵ_q , are introduced. It is unfortunate that it is often more difficult to find a_1 than it is to find the existence of ϵ_1 . Thus, a necessary and unwanted part of error correction is the location of an error once its existence is known. Now, the problem can be stated in four parts:

- 1) Find that an error exists;
- 2) Locate and isolate the error within the computer program;
- 3) Examine the possible corrections which do not generate new errors;
- 4) Choose one of these corrections and use it.

Computer scientists have for a long time been aware of the existence of the problem of error analysis and correction. In spite of this fact, it is only recently that the most significant work in the field has been published. The recent literature is certainly a welcome addition to the field, especially in view of the dearth of significant results. This dissertation is an attempt to add to the useful and practical results in this area.

Among the first to consider error analysis were Grems and Porter (23). The BACAIC system which they describe was one of the first higher level languages to go beyond the basic techniques of assembler writing. The technique of analysis that Grems and Porter describe relies heavily on the programmer. When an error occurs in BACIAC the machine halts. Grems and Porter state that "... these stops can be caused by keypunching errors, computing difficulties, or machine malfunction. In order to distinguish which error caused the machine to stop, a machine trail is printed. This 'machine trail' includes a pertinent comment to state the reason for stopping and to indicate corrective measures. It also includes the next instruction to execute after the corrective measures are accomplished." The measures taken in BACAIC differ from those discussed in this dissertation since the new techniques apply strictly to compiling whereas a great deal of the analysis in BACAIC is performed at execution time. Another difference between BACAIC methods and the techniques in this dissertation is that the techniques which are developed herein are automatic and Grems and Porter's techniques require programmer intervention (23).

Block (7) outlines a basic philosophy of automatic error correction. In this 1958 article many general guidelines which are still applicable today are set forth.

"In any technique for automatic error correction, the prerequisite of automatic error detection must exist. ...Certain of these techniques are based upon maneuvers in the area of machine programming" (7). Block continues from this point to discuss hardware error detection techniques. Hardware implementation of the techniques, however, would probably be

implemented by micro-programming.

Block continues to state that "it is natural ... that the issue of reliability and related issues of error detection and error correction have assumed a position of utmost importance from the viewpoint of the users of such computing equipment (7). The "... increase in work load have made it mandatory that not only the rate of error commission be reduced, but that efficient means be found for the treatment of these errors when they do occur. It is in this latter area where the principle of automatic error correction is now coming to the forefront of consideration" (7).

Samelson and Bauer (50) discuss a convenient way to describe the syntax of a programming language. The syntax is stated as a sequence of states of an entity that they call cellar. Their cellar is actually an automaton; admissible state symbol pairs control the transitions from one state to the next. The transition matrix technique, now common in automata theory, is used to represent the admissible state symbol pairs.

By the use of the method which Samelson and Bauer (50) describe it is possible to detect an error by observing an illegal input for a given state. The errors may be detected by respecifying the inputs which cause a transfer to an undefined state such that they transfer instead to some error state. The mechanics of this process may be observed in figure 1.1.

This simple description admits two states. One of the two states is assumed to be the initial state. When the "cellar" is in state 1 the only admissible input is a, while the only admissible input when the "cellar" is in state 2 is b. In figure 1.1a there is no possible way

Figure 1.1a No error detection

Figure 1.1b Error detection

Figure 1.1 Simple cellars

to intercept an illegal input. In the "cellar" of figure 1.1b a third state for error termination is added. This allows the errors to be detected.

In 1960, Floyd (17) described a method to determine whether or not a given input string is consistent with the formation rules of the ALGOL 60 assignment statement. The algorithm that is developed scans the input string from left to right replacing certain character pairs by single characters. If under the allowable transformations the symbol string may be reduced to a specified special character it is a well-formed formula of ALGOL. If this is not the case an error has occurred and the input string is not a well-formed formula. This algorithm is described by the flow chart of figure 1.2 and table 1.1; versions of both are in Floyd's 1960 article (17). A PL/I program description of this algorithm may be found in Appendix A. The syntax used in Appendix A, however, has been slightly altered to facilitate translation on the IBM 360.

Blair (5) discussed a program which implemented a heuristic procedure to correct natural language spelling errors. His method required that the entire vocabulary of properly spelled words be predefined. Any word which was found in the vocabulary was assumed to be spelled correctly. If a word was not in the vocabulary the correction process was applied. Blair made his method context independent so that the same sequence of letters with different sounds would not affect it. His method was based on the pronunciation of words. Words have abbreviations that retain the "kernel" (Blair's term) of the word. Two words are similar if their kernels (or abbreviations) are identical. Letters are assigned values based on their position in a word and the probability of occurrence of

Figure 1.2 An Algorithm Defining Syntactically Correct Assignment Statements (17).

[illegible]

each letter. Abbreviations are determined from these values and the word of the vocabulary with the nearest abbreviation is taken as the correct word.

Irons (34) emphasizes the relationship of a language syntax with the compiler for that language. He defines an object language and a target language. His object language is what today is called a source language, and his target language is what is now called an object language. His use of the term "object language" has now become archaic but his use of the term "target language" remains as it was in 1961.

After discussing the reasons that the language definition is usually imbedded in its compiler, Irons (34) proceeds to describe a method which separates the language definition from its compiler. His method was one of the first to use a meta-language to define the language to be compiled, and to compile that language in a disjoint module of the same program. The first module of such compilers could be called the language specifier and the second part the language translator.

This type of translator demonstrates the great need for error analysis and related techniques. Often, in translators of this type, a simple syntax error may cause premature termination of the compiler program. In addition, the error or errors may not be detected until long after they occur making recovery very difficult.

Samelson(49) studies the definitions and transformation processes of formalized artificial languages. Samelson states that there are two basic definitions which describe such languages:

- "1. A set (at most denumerable) of distinguishable elements

$$\mathcal{A} = \{a_1, a_2, \dots, a_i, \dots\}$$

2) A set of finite sequences of elements or strings over the alphabet which is effectively defined by rules of word formation. These strings are called the words or sentences of the language. The formation rules constitute the syntax of the system."

If an algorithm exists such that the algorithm can effectively decide if an arbitrary string is or is not a member of that language, then the language is called decidable. In such decidable languages one may apply the concept of a cellular automaton (49). This is an extension and formalization of the technique described in Samelson and Bauer (50).

Irons (33) again uses the terminology of translating from an object language into a target language. He describes the structure of a compiling system in which the translator is independent of the translation rules. This is essentially an extension and modification of his 1961 paper in the communications. This paper has a more extensive treatment of the metalanguage for defining the language and of code optimization. This paper also describes in detail one way declarative information can be processed within syntax-oriented translators. The fact that declarative statements must be processed with special considerations is relevant in error analysis and correction as will be observed in later chapters.

Cotton (10) discusses ambiguities in syntactic specification. These ambiguities, such as equal right sides, but different left sides, may cause severe problems in error analysis and error correction since the translation process may sense an error when in reality an ambiguity is the cause of the problem. This can especially be true in those top-down syntax-directed translators which use parallel parsing strategies.

(For example, see [32]). Cotton (10) states certain restrictions on the standard Backus Naur Form (see [45]) metalanguage such that these restrictions are sufficient to prevent ambiguities from occurring in the syntax specification. The "normal forms" that Cotton (10) develops are called "ambiguity preventing", but retain some of the power of the Backus Naur Form.

Cotton (10) also states that the original ALGOL 60 report demonstrated:

- "1) that simple and completely formal methods exist for expressing syntax specifications;
- 2) that in this formulation the individual replacement rules [syntactic definitions] may be made to correspond closely to the semantic units or macro-operations of the object program produced in compilation;
- 3) that, in many practical cases, the rules can be designed to insure that for a given program, or element of the language, only one tree of rule replacements exists to generate (or inversely, to recognize) the program as an element of the language."

Cotton (10) continues to show how to develop recognition trees for languages in these normal forms and develops an algorithm for generation.

Thorelli (58) discusses errors from a different viewpoint. A primary machine (such as a human) is a machine which generates text. Within this generated text there may or may not be errors. A secondary machine is a device which improves the quality of text which was generated by the primary machine. It is important to note that Thorelli's secondary machine is designed to improve the original text, but not necessarily making it error free. The secondary machine corrects errors by making use of the redundancy of the language and restoring parts of the erroneous information by using this redundancy.

Thorelli (58) continues to discuss types of distortion. The simplest case occurs when any one symbol in the text is independent of the status of any other symbol. Unfortunately, errors often occur in clusters, making the situation more difficult. In addition to having the errors occur in clusters, other types of dependency also exist. The dependency of one set of errors upon another set of errors serves to make the task of the error analyzer more difficult.

Davidson (13) discussed the problem of spelling correction when the words in question are people's names. Arbitrary names such as this are similar to the arbitrary words that a programmer uses to name his variables. His procedure involves an automatic scoring technique which matches names in a condensed form. The procedure that Davidson developed compresses the name to be retrieved, codes the name, and searches for a match for ultimate retrieval. Words in the name file which are to be retrieved are coded at the time the search reaches that particular name entry.

Probably the most significant paper which has been written on error analysis and recovery is Iron's 1963 article entitled "An Error-Correcting Parse Algorithm". In this article he states that a "...perplexing problem for many ... parsing algorithms has been what to do about syntactically incorrect object [source] strings. ... most of the ALGOL or FORTRAN 'programs' which a compiler sees are syntactically incorrect. All of the parsing algorithms detect the existence of such errors. Many have difficulty pinpointing the location of the error, printing out diagnostic information, and recovering enough to move on to other correct parts of the object [source] string."

What makes Irons' (32) error-correcting parse algorithm especially attractive is that "when a situation arises when more than one parse is possible for the next few symbols all possible parses are carried along until a symbol is reached which 'selects' one of the parses." This parallel parsing eliminates the necessity of back-up in error detection.

In order to develop his parse algorithm Irons (32) manipulates the standard Backus-Naur Form into his own metalanguage such that it is stripped of its recursive power and iteration is used in place of recursion. Parses vanish shortly after an error has occurred. This causes the following actions to take place:

"1) A list is compiled of all the syntactic elements on basic symbols which might be called for after the error point. The list consists of all elements of SN [SN is the array that holds the elements of a table representing the syntax tree] named by the syntax pointers of all brackets in all parses (just before the error point) and all successors and alternates of these SN elements.

2) The symbols at and after the error point are examined one by one and discarded until one is found which

- a. occurs on the list of 1, or
- b. has an element on its chain which occurs on the list of 1.

3) The bracket from 1 which is selected in 2 is examined in relation to the parses to determine a string of basic symbols which, when inserted at the error point will allow the parse to continue at least one symbol past the inserted string.

4) The string of 3 is inserted into the object [source] string at the error point and the parse is continued. The parse is forced to cover the complete input string by initializing the parse with a 'program' bracket which requires a special symbol ... for its termination."

Floyd (18) largely ignores the problem of error analysis and recovery since this was not the main purpose of his paper. Floyd's 1963 work is of special importance, however, since in it he develops an entirely new concept of bottom-up syntactic analysis and formalizes many

of the concepts of compiling. In particular, Floyd introduces the concept of a precedence grammar and precedence relations. He defines three precedence relations as follows:

"In an operator grammar there are three relations some or all of which may hold between two terminal characters T_1 and T_2 :

- 1) $T_1 \neq T_2$ if there is a production $U \rightarrow xT_1T_2y$ or $U \rightarrow xT_1U_1T_2y$ where $U_1 \in V-T$;
- 2) $T_1 > T_2$ if there is a production $U \rightarrow xU_1T_2y$ and a derivation $U_1 \xrightarrow{*} Z$ where $U_1 \in V-T$ and T_1 is the rightmost terminal character of Z .
- 3) $T_1 < T_2$ if there is a production $U \rightarrow xT_1U_1y$ and a derivation $U_1 \xrightarrow{*} Z$ where $U_1 \in V-T$ and T_2 is the leftmost terminal character of Z ."

Here, V is the vocabulary of the language; T is the set of terminal symbols; U, U_1, U_2 are non-terminal symbols; T_1, T_2, Z are terminal symbols; and x and y are character strings.

What prohibits this method from being very amenable to error analysis and correction is the fact that the precedence relations hold for the terminal symbols only. This makes examination of intermediate steps of translation very difficult. Other methods (62,63) which make use of precedence relations over the entire vocabulary have been developed and prove to be more amenable to error analysis and correction. A simple example employing Floyd's method of syntactic analysis is given in Appendix A.

Others have discussed methods of using a metalanguage to specify the language to be compiled while simplifying the detailed coding of a compiler. Compilers which have been created in this manner have structures such that it is easy to change and extend statements of the language. This method is essentially the same as those which were developed by Irons and, unfortunately, has the same problems with error analysis and correction.

A significant development in the field of error analysis and correction was presented at the Fall Joint Computer Conference in 1964 (20). In this presentation Freeman measures the success of the error analysis and correction processes by evaluating them in the light of three economics:

- 1) Are there fewer re-runs?
- 2) Has the cost of input preparation been reduced? and
- 3) Is less time devoted to tedious hand analysis of errors?

When an error is detected there should be a message to the programmer. This message should describe the relevant "variables, labels, numbers, etc;"; why they are in error and what corrective action the compiler has taken. Inherent in Freeman's philosophy is that errors should be detected as soon as possible in:

- "1) characters within symbols,
- 2) symbols within expressions,
- 3) expressions within statements, ... and
- 4) statements within the sequencing of each program."

Of course, badly garbled statements are deleted and no attempt at error correction is made. Freeman (20) continues with his philosophical discussion of errors by placing errors into three classes: errors belonging to the first class are those which cannot be corrected automatically; the second class includes those errors which are correctable by the problem programmer but by no one or nothing unfamiliar with the problem at hand; and the third class of errors are those which can be detected and corrected by scanning the source text without recourse to the problem definition. Heavy use of context is required to correct some of the type

three errors while no context at all is required for other errors of type three. It is these class three errors that are of primary concern in most error analysis and correction techniques.

A corrigible error is an error for which automatic correction is attempted by the compiler (or monitor). The compiler's "error correction procedures should attempt to maximize the 'expected useful yield' of a program by strategies based on a priori probabilities associated with different errors." Most corrigible errors are detected in the source text by the compiler, but a few may be detected during the use of object programs. Correction procedures must be added for each of these errors yielding some loss in the operating speed and in the core memory available.

Error conditions may be roughly ranked by two criteria:

- 1) the a priori probability of detection, and
- 2) the a priori probability of correction.

Some errors are difficult to correct but may be "repaired" nevertheless. For example, certain misspellings are very common and may be corrected at a high rate, but a subscript omission may only be repaired by supplying a value for the subscript which was not present.

Corrections of spelling errors have been discussed in great detail by Damereau (12) and Blair (5). Freeman (20) relies heavily upon their methods and develops a figure of merit for each possibly misspelled word. The word form with the highest figure of merit is then assumed to be the correct spelling. Dictionary entry and analysis methods are also discussed.

Lietzke (41) developed a syntax checking device which was directly based on the syntax of ALGOL as it was defined in the ALGOL report. Since ALGOL is essentially recursive the syntax checking device was designed as a set of mutually exclusive recursive subprograms. Although she did not desire total error correction as such, she did desire and develop a method which permits most programs to be completely checked in spite of imbedded errors. Her main consideration was in finding a new starting point once an error had been detected. She states that "It would be possible to skip to the next semicolon, end, or else [and] attempt to go from there, but this could leave sections of the program unchecked. To minimize the number of runs necessary to obtain a syntactically correct program, a Resume processor was designed to recognize and check any syntactic unit and then return control to the processor which called it. If on a return from Resume the calling processor is unable to continue, it may call Resume again to check the next unit; if the calling processor does find a legal symbol on return it will continue checking in the normal mode."

Harrison (28) discusses error correction in finite automata. Of particular note is Harrison's definition of "correctable" which is slightly modified for use in this dissertation. He states that a relation, R , is "correctable if there is a machine which corrects R ."

In their two 1966 articles, Wirth and Weber (62,63) develop a method which is an extension of Floyd's (18) 1963 precedence techniques. The method introduces a rigorous relationship between the language structure and its meaning. The method requires that a one-to-one correspondence between syntactic rules and interpretation rules must exist. The actual

interpretation rules are invoked by the use of a bottom-up parse. The method requires that the language must be described by a simple precedence syntax so that a precedence matrix may be used for the parse. It is here that the major departure from Floyd (18) occurs. Instead of the precedence relations holding only for terminal symbols the relations hold for all symbols - terminal and non-terminal. In this case errors may be detected by the encounter of a void entry in the precedence matrix during the parse (62,63).

Hopcroft and Ullman (29) preceded Smith (55) in attempting to find a solution to the error correction problem. Their concern is with the increase of difficulty encountered by incorporating a detector into a translator. In view of this fact, they proceed to develop a theory of correction based on the concept of finite automata. They, like Smith (55) who followed them, did not report the development and testing of any computational algorithm to implement their theory.

Alberga (1) attempted to evaluate several methods for spelling correction. The basic problem considered was whether or not a given string of characters was a misspelling of a given word. The techniques, which were explored, were applied to English words (out-of-context) as they were written by students on spelling examinations.

Torrii, Kasami and Ozaki (59) discuss the detection of whether or not given input strings are grammatical and form a sentence. In addition to the obvious problems of detection and recovery, some subordinate questions are introduced; namely, how much memory is required to implement the technique and how much computation time is required. There can be no application for a detection system that requires two days to detect errors

in a program that would normally execute in ten seconds. It is these subordinate questions that concern these authors.

"A transduction is a mapping from one set of sequences to another (40)." Transduction may be used as a model of part of the translation process that is used by many different compilers. The primary difference between a transducer and a translator is that only the input-output relations need be specified for a transducer whereas a translator requires a translation algorithm. This entire article is an attempt to formalize the previous work by Irons (34), Ingerman (31) and Reeves (47).

Gries (25) describes an algorithm to construct left-right recognizers for sentences which have been described in a suitable Backus Naur Form grammar. The recognizer operates with a pushdown stack and with a transition matrix. This type of transition matrix technique has been used previously by Samelson and Bauer (50). Recognizers of this type tend to be very fast, but they use large amounts of storage. The storage is required for the transition matrix in a switching table which specifies what to do when it is given the current symbol and the next symbol to be processed. Specific error states are discussed for use within this type of compiler.

In spite of their general convenience, compilers which use a syntactic description of the language which they are to translate have one drawback: special well-behaved grammars must be used in order to obtain a tolerable rate of translation. Only sometimes do these special grammars agree with the natural structure of the language under consideration. To avoid this problem Foster (19) has written a program called SID or syntax improving device. This program attempts to transform a natural grammar

into a simpler one which may be parsed easily with the use of an algorithm developed for that purpose. An enhancing feature of SID is that part of an ALGOL compiler produced from it checked syntax completely while a handwritten one did not.

Smith (55) discusses the area of error control for artificial languages. His approach to the problem is through the vehicle of finite automata. He investigates the ability of formal automata to detect and to correct errors in formal languages.

Smith (55) emphasizes the feeling of the professional programmer when he states: "Often, however, a compiler will not continue in the face of an error but rather requires that the input be corrected before continuing past the point of error. This procedure can result in several tries before a program is successfully compiled. It would seem that a feature highly desirable in a compiler would be the ability to correct multiple errors in the input and to complete the compilation in spite of them." This is a statement of one of the primary goals of this dissertation. Contrary to the goals of Smith (55), however, the corresponding goal in this dissertation is to produce a working algorithm for error correction.

Crenshaw (11) discusses methods of locating error and of communicating the location of that error to the user. His views about correction are compatible with most of those currently held. Simple and fast error correction, according to Crenshaw, is complete after an error has been detected and "Control is passed to an independent routine which scans the input elements following the structure in error until an end-of-statement symbol is sensed. At that point, control is returned to the

initial statement processor which indicates the analysis of the next sentence."

Gusev and Smirnova (26,27), and Spanier (56) review the area of languages and its relationship to automata theory. These are very interesting articles which join many of the concepts which have been developed heuristically to those which have been developed formally.

Schneider's (52) translator system is essentially a syntax-directed system. In order to recover from a syntax error the program scans ahead until it finds a semicolon or an END which follows the error. The translator is then reset so that translation may resume from the newly found position in the input string.

Morgan (43) developed spelling correction techniques which closely parallel those which are presented in this dissertation. He describes methods for the use of syntax and semantics, keyword organization, and use of symbol tables for system spelling error correction. Like Freeman (20) Morgan's techniques were designed as part of the system rather than being designed to be applied to a predefined high-level language.

Currently the most popular approach to error recovery and analysis is to design the error recovery process for a particular programming language using the specific characteristics of the language. A more aesthetic approach is to base error analysis and recovery on the formal definition of a language on language class.*

* Leinius, R. P., Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee. A survey of previous work for syntax directed compiling. Private Communication. 1969.

II. BASIC DEFINITIONS

It is helpful in error analysis and correction to divide errors into several classes. To accomplish this the classification system of Freeman (20) is modified and adopted. However, slightly different naming conventions are used.

All errors in artificial procedure-oriented languages may be divided into three basic classes:

- 1) those errors which are incapable of correction, or those errors which are a flagrant misuse of the language but may be correctable by the programmer;
- 2) those errors which are problem-dependent rather than language-dependent;
- 3) those errors which can be corrected with only a knowledge of the source code and of the source language.

Class one errors include those in which the programmer attempts to write a statement that is unsupported by the language in which he is writing. Class two errors include such things as an incorrect numerical constant or a miscopied scientific formula. These errors are correctable only by the programmer. Class three errors are the only ones within the scope of this dissertation. Since only a knowledge of the source code and of the source language is required for correction, such errors are easily adaptable to automatic correction.

There are two subclasses of errors within class three: incorrigible and corrigible. Incorrigible errors are those for which no attempt at correction is made. The main reason that no such attempt is made is that their low probability of occurrence far outweighs the cost in programming overhead for correction. There is a second reason that makes some class

3 errors incorrigible. In spite of a high probability of occurrence there may be a low probability of correction. Some of the errors which occur, but only rarely, in FORTRAN ASSIGN statements are an example of such errors.

An attempt is made to correct corrigible errors. There are two types of corrigible errors. Type I errors are those which are syntax independent while type II errors are those which are syntax dependent. A syntax independent error is an error which may be corrected by changing one syntactically acceptable string into another. This includes those usually inadvertent, keypunch or spelling errors. A syntax dependent error is an error in which a syntactically invalid input string must be changed into one which is syntactically valid. An observation of these types of errors indicates that type I errors tend to be context dependent such as a misspelling, but type II errors are context independent. An example of type II error is an arithmetic statement with a missing operand. The relationships among the various types of errors is shown by the error tree which is given in figure 2.1.

An examination of the error tree given in Figure 2.1 shows that the existence of an error is the root of the whole error problem. Since it is impossible for humans not to make mistakes, an alternative is to make available an automatic error correction device such as the one implemented in this dissertation. This device may be utilized to minimize the effects of these mistakes.

The corrigible errors of a program also may be thought of as independent errors and dependent errors. An independent error is an error which would exist regardless of how the rest (other than the statement in which the error exists) of the program would be modified. A dependent error is

Figure 2.1 A tree showing the relationships among different types of programming errors.

an error which results, directly or indirectly, from some independent error in the program. (Notice the difference between "syntax independent error" and "independent error", and between "syntax dependent error" and "dependent error".)

There is a difference between error correction and error repair as the two terms are used in this dissertation. "The correction of a programming error is defined to be the alteration of relevant source-language symbols to what the programmer truly intended" (20). It may be seen that many errors cannot be corrected. When code is modified to produce an executable program but is dissimilar to the code which the programmer intended to write, then the error has been repaired but not corrected. The algorithms which are developed in this dissertation are designed to correct the error if possible, but to repair it if they cannot correct it (20).

The notation which is introduced in this and the following paragraphs serves as a shorthand to use in the discussion of the processes. The operational compiler is a translator; that is, it requires a specific algorithm to specify how the legal input sequences are to be changed to the output sequences. On the other hand, the error corrector which is developed in this dissertation is a transducer. A transducer does not require the specific translation algorithm that is required by the translator. A transducer merely maps one set of sequences into another. This is the function of the error corrector. A general method is developed but specific algorithms are necessary to apply the techniques to the various languages.

Let $\mathcal{A} = (l_1, l_2, \dots, l_n)$ be some alphabet over which a language is defined. Let the words w_1 and w_2 have a Hamming distance of e if w_1 and w_2 have exactly e different characters. This concept, which is related to the concept of the Hamming distance in a binary code, can be valuable in the analysis of errors as will be seen later in the development of specific algorithms (29).

A burst error is a special type of error which may be said to have length. If a word is erroneous, but differs from some valid word of the specified language with a Hamming distance of e , then a "burst error of length e is one in which all changed symbols occur in a subword of length e or less" (29).

Related to burst errors are e -tuple errors. Let Σ_e be the set of words in Σ (all finite permutations of the letters in Σ) that are at least a Hamming distance of e from a valid word in the language, L . Then Σ_e is the set of e -tuple errors of L (29).

Suppose P is a program which contains n independent errors. In the process of correcting the n independent errors a sequence of partially corrected programs is induced from P . P_n is used to denote the program P and P_{n-k} is used to denote the program after k independent errors have been corrected. With each program of the sequence is associated an error-space. The concept of an error space, as defined below, can be used in the theoretical development of error correction processes. This error space is defined as follows:

Let $E = (e_1, e_2, \dots, e_n)$ be the set of all n independent errors in some program P_n . These errors then form the basis of the error space $P_n(E)$ of P_n . Some subset $S = (e_{i_1}, e_{i_2}, \dots, e_{i_k})$ of E , $S \subseteq E$, defines

an error space $P_k(S)$. The only operation which is defined on an error space is a correction transformation. A correction transformation is an algorithm which is applied to $P_n(E)$ to reduce the number of elements in E . It is desirable then to obtain some sequence of error spaces, $P_n(E)$, $P_{n-k_1}(S_1)$, $P_{n-k_1-k_2}(S_2)$, ..., $P_{n-k_1-\dots-k_j}(S_j)$; where $E = S_0$, and S_j is a subset of S_{j-1} of dimension $n-k_1-k_2-\dots-k_j$, and S is the null set; this implies that $n-k_1-k_2-\dots-k_j = 0$. At the time P_0 is derived from P_n by the above sequence, P_0 should be an executable program.

Existence of errors may be detected in one of two ways. Either the error corrector may use its own recognizer or it may use that of the compiler. If a corrector uses its own recognizer to determine the existence of errors, the method of using such a recognizer should be examined. A recognizer of this type may either be logically separate from the corrector or it may be a part of the corrector itself. The type of corrector that has a recognizer as one of its integral parts, unfortunately, is so logically complex that it would have many of the problems that a compiler has with error correction. If the recognizer is a logically separate entity then many of the compiler problems can be avoided, but much of the compiler is repeated.

Since all compilers have some type of error detection, even in the most minimal sense, it should be possible to make use of this built-in detection facility. At the termination of the compiler's task the loader may be invoked if there are no errors, but the error corrector would be invoked if errors do exist. Notice that the correct program pays no overhead for the error corrector while the incorrect program pays the full price. One way in which this could be accomplished is shown in figure 2.2.

Figure 2.2 One Basic Error Correction Algorithm

This method is discussed in detail in the next chapter.

Thus, errors can be located directly by an error corrector which contains its own recognizer. It is less difficult, however, for a corrector to locate errors which are known to exist by the action of a compiler. This particular phase of the error corrector will usually be biased (if not oriented) toward a specific compiler. Some compilers list all errors at the termination of the source listing, some flag the errors at their occurrences, while still others freely intersperse existence messages throughout the source code. Errors can be located indirectly when it is known where to find these messages. This is a trivial matter when the errors are flagged at the point of occurrence. On the other hand some processing is required when the existence messages occur elsewhere in the program. In this latter case the actual location may be found by examining the message which is output from the compiler. A search for the statement number can be made, then this can be followed by a search for the statement itself. Once the error has been so located the corrector may proceed with its correction algorithm.

III. FACTORS WHICH INFLUENCE THE DESIGN OF THE SOLUTION TO THE ERROR CORRECTION PROBLEM

In the formation of the solution to the error correction problem it is desirable to be cognizant of the four distinct phases of the problem. The reader will recall that these four distinct phases are: existence, location, generation and (ultimately) correction. In this chapter each of these phases is considered, but supplemental and complementary processes are also discussed. These other processes include both those that fall under one of the four basic phases and also those which are requisite for the four basic phases to operate together as a single error corrector. The scope of this chapter is then limited to the four basic phases and these related portions of the error corrector problem.

In order to obtain some insight for the development of the various tasks required in the solution for this problem, it may be helpful to look at some alternate methods of examining the possible solutions to the problem as a whole.

One possible general solution is given in figure 2.2 of the chapter two. This process requires that the program on which the error corrector is to operate be stored on tape or sequentially on disk. The program is fetched from this storage device by the compiler. After the program has been completely compiled, a check is made for the existence of any errors. If there are no errors the program continues through the load and execute steps and terminates as it normally would. If it is determined that errors exist, then the corrector module is invoked. The actual logic of this corrector module is described later in this chapter. The corrector

operates on the erroneous program to produce a new program which contains fewer errors than the original. This new program then replaces the old program on the external data set. This process is then repeated until either the program has no errors or the number of errors can no longer be reduced. Although this can be programmed for use as a production algorithm, its iterative nature is a serious restriction. The fact that it is iterative requires that the same action be repeated for successively fewer corrections. Another restriction of this method is that there is no provision for handling the errors which occur at any time other than compile time.

A second possible general method of solution is shown in Figure 3.1. At first this solution appears to be the same as the one just described. Closer examination, however, reveals that this method overcomes the second restriction of the method which is described above. Although this process has the primary restriction of being iterative like the one above, the processing steps have been rearranged so that compile, load, and execute-time errors can be considered by the corrector. Instead of placing the test for the occurrence of an erroneous program immediately after compile it is placed after the program has completed. Now the test can be valid for execute-time errors and load-time errors as well as compile-time errors. This will facilitate the correction of more errors during each iteration.

Figure 3.2 shows yet another method. This method is essentially the method of Figure 3.1, but it has been stripped of its iterative power thus overcoming the first restriction of the first method and the primary restriction of the second method. In spite of the fact that this

Figure 3.1 Another basic method for error correction. The dotted lines indicate possible paths which may be taken as a result of operating system action.

Figure 3.2 Non-iterative method for correction, $k \leq n$, ~~QCSCE~~

reduction in iterative power causes a subsequent reduction in the correction which is performed by the corrector module itself, it permits the user to make permanent corrections and may permit the corrector to do an adequate job in a shorter period of time. This third method, therefore, is faster and probably more practical than the other two. It is this third method which was actually used in the corrector which is shown in Appendix B. A cursory examination of the method in figure 3.2 shows that the second, third, and fourth processing modules have been duplicated in the second, third, and fourth positions after the determination of the existence of one or more errors. No space in storage is saved; the main benefit of this method is its shorter execution time.

As the preceding paragraphs have indicated, the error correction procedure consists of several modules. These modules perform all of the standard functions as well as analyzing, correcting and causing multiple executions. The corrector proper is only one of several modules of the complete correction procedure. The configuration of these modules may vary even within the same method; however, one logical configuration is shown in figure 3.3. Like the method shown in figure 3.2 this configuration is essentially the one which is used in the correction procedure which is given in Appendix B. There are a few minor variations in that procedure, however. The configuration in figure 3.3 is used for the general process for an arbitrary compiler, but most of these variations are due to the fact that the corrector in the appendix is written for a specific language using a specific compiler. In the configuration shown in figure 3.3 the correction process begins with the standard compile, load, and execute steps that always exist. It is at this point that the

Figure 3.3 Configuration of Modules Within the Corrector. This diagram does not represent the algorithm but shows only the logical positions of the various modules.

modules which have been written specifically for the correction procedure are supplied. The first of these modules is the decision module. This particular module determines whether or not there is any need for further action on the part of the correction procedure. If no further action is required by the correction procedure then the program is output and the correction procedure terminates. After the decision module there are several modules which manipulate the program and its various outputs so that they will be in the proper form to be input to the analyzer and corrector modules. The analyzer and the corrector modules are the most logically complex of the modules which were written specifically for the correction procedure. The analyzer and the corrector operate together as co-routines. Both of these modules make use of the same error code table, but each has separate routines which are dynamically executed. An error analysis table results from the execution of the analyzer. This table is then merged with the results of the corrector module to form an updated program. This updated program is then input to the compiler. The correction procedure then terminates as it started with compile, load, and execute modules. This configuration of modules does match the flow diagram which is shown in the block diagram in figure 2. Similar configurations may be developed for the other possible methods of error correction.

Within each of the modules there are several logical processes which are applicable to the correction process. One of these processes is the process of error location. Location of the occurrence of errors is necessary for the corrector and analyzer modules to operate properly. The

location process resides within the analyzer modules as one of the analyzer routines.

The location process is not invoked unless the correction procedure has the "knowledge" of the occurrence of one or more errors. When this location process is invoked it must first search for the place where the error was messaged. An error message may occur at one of four points:

- 1) immediately following the point of occurrence;
- 2) at the end of the compilation of a routine;
- 3) at the end of the loading process; or
- 4) during the execution of the program itself.

Each of these requires that the locator use a slightly different process to find it. In any implementation an attempt should be made to keep these differences minimal so that location via any of the four types of messages may use as much common code as possible.

Error messages which occur in the line which immediately follows the line in which the error occurred lead to the most straightforward manner of location. Errors which cause the generation of this type of message may be located by a syntactic search of the line that precedes the message. This syntactic search may be made with the aid of a translation matrix such as that given for Floyd's algorithm in Appendix A. For this syntactic searching a companion error action matrix should be used. The error action matrix should specify what action to take when some particular erroneous sequence of characters is encountered.

Errors which cause messages to be generated at the termination of the compilation of a subprogram require a more indirect approach for their location than do the above mentioned type. The message line can be

searched for an indication of the line in the program where the error occurred. When the number of the line which contains the error is found the locator can recall that line and a syntactic search can be made as before. Some additional searching action may be required, however, since these errors may not be errors in the syntax.

Errors which are messaged at the termination of program loading require still more effort for their location. In fact, these errors may be an omission of some type by the programmer and the function of the locator is to pass that fact on to the corrector. Other errors may be errors of commission, but the locator may not be able to determine the exact location of the error. Errors of omission may include missing subprograms, while errors of commission may include subprograms with duplicate names. In this latter case the locator should pass all information that it has about the error to the corrector. Some errors which fit into the categories discussed in the previous paragraphs may be messaged after the loading process. Their handling remains the same as the handling of the cases which were described earlier.

Errors which are messaged at execution time are the last type with which the locator is concerned. These errors also require the most work on the part of the locator. The locator must determine whether a given error message is the result of a programming error or a job control error. If it is a programming error, then the locator must determine if the actual error occurred in the line indicated or if the actual error occurred in some other line and the error was not made manifest until the line which was messaged. If it is a job control error, then the job control must be checked and the locator must "choose" whether the program is

suitable for the correction process to continue.

An identifier may be used as a co-routine with the locator. The identifier is used to classify the various types of errors as they are discovered by the locator. The identifier, like the locator, must perform its processing slightly differently for each different place where an error may be messaged. The identifier should have access to a table which contains all of the possible error codes which may be generated by the compiler for which the corrector is designed to accommodate. This table should supply the identifier with the information about the places where each message can occur. The identifier can then use these tables to create new information tables which the corrector module can use. These new tables may include things such as a partial program symbol table, a statement symbol table, and/or a syntax analysis table.

The corrector module which is a portion of the corrector must rely on the actions of other modules for its own adequate performance. As was mentioned previously the corrector module operates as a co-routine with the analyzer module. The analyzer produces location and identification information. When each item of location or identification information is created it is passed to the corrector which uses the information. The corrector uses the data which is given to it by the analyzer sub-modules and uses it to make heuristic decisions about the actual error correction that it is to perform. If the corrector cannot function with this data alone then it builds a cleanup table which it will use the last time it gains control while a program is being processed.

The corrector must be able to determine properly whether an error is a propagated (dependent) error or an independent error. The tables which

it obtains from the identifier will usually contain all of the information needed for the corrector to make this choice. When these tables do not contain sufficient information, then the corrector must perform some additional probabilistic analysis to generate as much relevant information as possible. After the corrector has obtained this information it places it into the appropriate tables and continues as before. Caution must be exercised in the development of the corrector to assure that the analysis that it performs by default does not supply any data that causes subsequent analysis and correction to fail. The relationship between the analyzer and corrector modules is shown in figure 3.4.

Each of the individual correction modules which fit into the position shown in figure 3.4 must consist of two distinct parts. The first part is a generation phase. During this phase the individual correction modules generate all of the most probable corrections for a given error. These corrections may be stored in a correction generation table. This correction generation table is then used by the second phase of each of the individual error correction modules. Each module must compare the input string with the correction table to produce a new string from that input string. If the action of all of the modules throughout the execution of the corrector has been correct then this new string may be compiled with a minimum of errors.

The correction process which is described in this chapter is not designed for any particular language, but for any language which has adequate error detection facilities of its own. Any corrector which is developed using this basic process must be developed for a specific compiler due to the heuristic decisions which must be made. These

Figure 3.4 Relationship Among The Various Corrector and Analyzer Modules

decisions which are made dynamically by the corrector module must make use of the estimated probabilities of occurrence for each error. In order to obtain these estimates, a random sample of programs written for the compiler in question should be analyzed. Once the sample has been obtained each program in it is processed by the compiler and executed. All of the output from every phase is saved. This output is now ready for use as input to a preliminary analyzer program. This preliminary analyzer examines every occurrence of each error and compares that occurrence with the set of all possible errors. In this manner the preliminary analyzer may gather data and generate estimated statistics which may then be used in the correction process. The diagram in figure 3.5 shows the relationship of the gathering of statistics with the formation of the corrector.

This completes an informal discussion of the solution of the error correction problem. An overall configuration of a general error corrector has been given. This configuration has then been dissected and analyzed by analyzing several of its component parts. The most important of these parts are the analyzer module and the corrector module. The analyzer module contains submodules for location and for identification. The corrector contains modules for the correction of the different types of errors which may be encountered. The analyzer and corrector must have certain statistics available for their development. These statistics may be estimated from a random sample of the type of jobs on which the corrector is to operate.

Figure 3.5 Use of Statistical Sampling in Corrector Development

IV. FACTORS WHICH INFLUENCE THE IMPLEMENTATION OF AN ERROR CORRECTOR*

The language is the most important item to be considered when implementing an error corrector since it has the greatest influence on the ultimate coding of the corrector. The processes described herein have been designed only for certain algorithmic languages, but could, with modifications, be extended to include other languages. Some of the language features which influence the implementation of an error corrector are: input formatting, statement types, keywords, user restrictions, and outputting.

The input format of a language is the specification of how every statement is to be presented to any compiler for that language. For example, the FORTRAN input format specifies that all statements must be written in columns seven through seventy-two of an eighty column input string (usually a card image). The FORTRAN input format also specifies that a statement may be identified by placing a number in columns one through five. A statement may be continued from one input string to another by placing a non-zero mark in column six of the second input string.

Every language has several different types of statements. Each different type of statement must be analyzed and, if it is in error, corrected by a process which is unique for it. For example, the two fundamental classes of statements in the FORTRAN language are non-executable and executable statements. Non-executable FORTRAN statements

* The algorithms for burst error correction which are presented in this chapter were developed simultaneously and independently by Morgan (43).

include the specification statements. An error in a specification statement causes any declaration that follows the error within that statement to be ignored. Such ignored declarations can cause dependent errors throughout the program. If the specification statement is corrected or repaired, then not only will the original independent error be eliminated, but also all dependent errors which have been generated by the original error should disappear. A statistical sampling (see Appendix B) shows that a large number of errors in specification statements are spelling errors. A general spelling correction algorithm may be used to correct these errors. Such an algorithm is discussed later in this chapter.

Executable FORTRAN statements will not often be corrected by spelling correction algorithms. There are two reasons for this: spelling correction algorithms do not work well for short symbols (see figure 4.1, [12]); and errors in executable statements often are not simple spelling errors. Many of the errors in the executable statements of FORTRAN may be syntax errors. Because syntax errors vary among different types of statements, it is necessary to make use of the fact that the class of executable statements can be dissected such that there are arithmetic and non-arithmetic statements; the non-arithmetic statements can be classified still further. One approach to the correction of syntax errors is to use a translation matrix with a companion action matrix. Translation of a statement can proceed in normal fashion until an error condition is encountered (a blank entry in the translation matrix). When such an error condition is encountered, the action matrix may be used to indicate what corrective action is to be taken. An algorithm of this type

Figure 4.1a Algorithm for correcting burst errors. Predefined processes GETR and PUTR are machine dependent I/O routines. Termination is handled by the GETR input processor (12).

Figure 4.1b Flow chart for the predefined process ALG-1, ALGO and ALG1 (12).

is discussed below.

One syntax algorithm which is applicable only to FORTRAN assignment statements may be used as a model to correct other syntax dependent errors. The analysis is performed by examining the statement string from left to right. Each character is examined and the translation, which is indicated by the translation matrix, is performed if possible. If the translation cannot be performed, the action which is specified by the action matrix is performed. This particular algorithm, which is shown in figure 4.2, has been implemented as part of the error corrector which is Appendix B.

The classification of statements discussed above is not unique to FORTRAN; in fact, it is characteristic of almost all computer languages. Any classification of the preceding type serves to make the code to perform error analysis and correction more straightforward, logically speaking, than if the classification could not be made. Such classification allows the most applicable correction algorithms to be used on each of the various types of statements. The greater degree of classification that can be obtained, the easier it is to create the code for error analysis and correction.

The availability of keywords facilitates the correction of that type of error known as a burst error. All of the keywords for a language may be entered into a symbol table (or a dictionary) prior to the execution of the analyzer. During the execution of the analyzer the dictionary can be updated and burst errors can be corrected by the use of the algorithm which is shown in figure 4.1 (12). This algorithm can be applied to any language with the modification of the defined

Figure 4.2 Syntax correction algorithm for syntax errors in FORTRAN assignment statements. T_{c1} is the next column entry in the translation matrix. The translation matrix is given in table 1.

Table 4.1 Combined translation matrix and action matrix for FORTRAN assignment statements. An entry that is a number greater than five indicates that some action must be taken. Action varies among implementations. A blank in the matrix represents an impossible combination.

New Old	0	1	2	3	4
0	1	6	7		8
1	1	1	2		9
2	3	4	10		11
3	3	3	12		2
4	13	4	14	5	2
5	3	4	15	16	17

Legend
Old

0 null string
1 left hand side
2 operator
3 variable
4 number
5 exponent

New

0 letter
1 number
2 equal sign
3 exponent
4 operator

processes GETR and PUTR which are language dependent. Notice that only burst errors of length two or less are considered. This is consistent with the results of Damereau (12) who states that over eighty percent of all burst errors can be corrected in this manner. This particular algorithm tests for burst errors in the symbols, which are user defined, as well as in the predefined keywords, but the keywords serve as the initial set of symbols for the algorithm. This algorithm is more applicable to the total error correction process than is the algorithm which is due to Blair (5), since Blair's algorithm depends in part on the similarity of sound for two words. A complete description of Blair's algorithm is given in his 1960 paper.

A language always places certain restrictions upon a user. The exact restrictions vary from language to language, but any violation of the restrictions of the particular language in use by the user always results in an error. An error analyzer and corrector must expect violations of these restrictions and try to remedy them whenever possible. Restrictions of this type may arise from any of the language features; for example, in FORTRAN statement labels must be positive numbers less than 100000. Other language restrictions include all syntax restrictions, restrictions in the values which are permitted for certain variables, and restrictions concerning the use of blanks. In FORTRAN an integer cannot be used as a parameter in a do loop if it exceed 32768; and blanks are ignored in most cases. The analyzer must be prepared to detect violations of such restrictions and the corrector must be able to make the most likely correction without creating any new errors.

The output formatting may be partially specified by the language definition as well as the compiler and/or the physical devices available. FORTRAN requires that all output lines contain at least eighty print positions, not including printer carriage control, for its program listings. Other languages may require a different number of output positions or be even more specific than FORTRAN. As the error corrector is implemented, these requirements must be considered.

Another item which must be considered for the implementation is the specific compiler which is in use to compile the language on which the error analyzer and corrector is to operate. Each different compiler has its own special control cards and/or symbols, keywords, and other restrictions which are unique to it. In addition, many compilers require special input and have output which is not language defined. For the WATFOR compiler which is in use at the Iowa State installation, the special input is punched on a /JOB card which precedes the actual FORTRAN program. The /JOB card is an example of a special control card which one compiler requires for its own use. Both the compiler and the error analyzer and corrector may use the information which is punched on this card. This information may include many indications about the nature of the FORTRAN program which follows. The compiler output which is not language defined is an aid to the error corrector and analyzer. This output usually includes a preliminary indication of the places where errors have occurred. By examining this output much of the error analysis is reduced and the amount of error detection which must be done by the analyzer and corrector is minimized. If the compiler has elaborate error detection facilities, (as does the WATFOR compiler at

the Iowa State installation) perhaps even some of the analysis can be avoided. Other items by which the special compiler output aids the error analysis and detection may include symbol tables, statements of the time required to perform various processes, and other statements about the user's program.

The final implementation of an error corrector depends upon the implications of the theory on which it is based. For an error corrector which is based on the theoretical considerations of this dissertation the following items must be considered:

- 1) If two or more spaces are symmetric (have the same set of independent errors) what processing should be done to the programs described by those spaces?
- 2) How should two or more equivalent (identical sets of errors) spaces be handled by the error corrector?
- 3) Can the concept of the derivative (the set of dependent errors) of a basis be used as an aid in the implementation of the corrector?

When two or more error spaces are symmetric they have the same set of independent errors but do not necessarily have identical sets of dependent errors. If the dependent errors can be isolated then the identical correction algorithms applied to the programs, which are described by the error spaces, should result in new programs which are also described by symmetric error spaces. If the correction algorithms are applied repetitively then there may be an iteration which produces two equivalent spaces for which the following discussion applies.

When two error spaces are equivalent the correction algorithms should yield the same results when they are applied to the programs which are described by the equivalent spaces.

In the case of an iterative correcting process the criteria for termination of the process may be the fact that programs produced by two successive iterations are symmetric, or even equivalent.

The concept of the derivative of a basis of a space can indeed be utilized in the implementation of an error corrector. If the derivative of the basis can be determined, then the dependent errors which are a part of that derivative can be deleted as they are located, allowing the corrector to save the time required for the processing of such errors. Figures 4.3 and 4.4 shows one way whereby these concepts may be used in an error corrector.

Another thing that must be considered when an error corrector is implemented is the operating system on which the error analyzer and corrector is to be run. The operating system affects the overall configuration of the error analyzer and corrector in that the error analyzer and corrector should be written so that the overall operation is as efficient as possible with this particular operating system combined with its almost unique hardware configuration. The operating system allows only certain compilers to be available. This limits the choice of the language in which to program the error corrector, and in addition it limits the languages to which the corrector may be applied. The overall organization of the entire process is affected as well: on the IBM 360/65 operating under OS/MVT at Iowa State, compilation and execution of the user's program may be followed by the error corrector and analyzer as separate steps of the same job. This is convenient, but is not possible on every operating system of every manufacturer.

Figure 4.3 The use of the symmetry (programs with the same spaces) and equivalence (identical programs) of error spaces within an error corrector.

Figure 4.4 The use of Derivatives (sets of dependent errors) within an error corrector.

The final significant question to be considered is: what language should be used to perform the non-numeric computation required by the error analysis and error correction processes? The language most suitable for the job must have string manipulation capabilities as well as arithmetic capabilities. Any string processing language such as SNOBOL or LISP would suffice to allow the string manipulation capabilities, but the arithmetic features of such languages are severely limited. Languages such as ALGOL or FORTRAN have adequate arithmetic features, but they almost entirely exclude the string manipulation capabilities. Fortunately, the programming language PL/I has the features of both types of languages and becomes the most convenient language to use for the implementation of the error analyzer and corrector.

Special considerations for manipulation involving non-numeric error are required even after the programming language has been chosen. Various types of searches must be performed on character strings and arrays of character strings. For each individual search several questions must be asked:

- 1) Should the built-in functions be used?
- 2) What type of search should be used to obtain maximum speed and storage efficiency?
- 3) How difficult will the updating process be if a particular type of search is used?

The answer to question one varies for each different implementation and can be answered only by the person who actually implements the corrector. The answers to questions two and three also vary among implementations but guidance in obtaining the answers is found in Flores' book. (16).

In summary, the main factors which influence the implementation of an error corrector are:

- 1) the language to which the corrector is applied;
- 2) the compiler for the language;
- 3) the operating system which monitors the execution of the error corrector and the execution of the problem program; and
- 4) the language in which the error corrector is written.

V. THE DESIGN AND IMPLEMENTATION OF AN ERROR CORRECTOR

To demonstrate the applicability of the theory described in chapters one through four an error analyzer and corrector has been implemented for the WAFOR FORTRAN compiler. In this chapter the design and implementation of this error analyzer and corrector is discussed in detail. A method for evaluating the performance of an error analyzer and corrector is examined in the next chapter. The programs which are discussed in this chapter are found in Appendix B.

The basic theory has been developed in the previous chapters and this error corrector is designed utilizing this theory. This basic algorithm which is used in this implementation is the one which is described by the flow chart of figure 3.2 in chapter III. A batch of WAFOR programs is placed on disk by a program which precedes the error analyzer and corrector. The programs on disk are placed in a data set which, via control cards, is made to look like the standard input data set for the WAFOR compiler. The compiler is then invoked for this batch by the use of operating system control cards. Under the control of the compiler every job in the batch of WAFOR programs is compiled and executed. All printed output from compilation and execution is intercepted and forced onto a disk data set; this also is accomplished by the use of operating system control cards. The printed output of the WAFOR compiler for each WAFOR program can now be retrieved by the first part of a program that contains the error analyzer and corrector as its second part. When the output from the compiler is retrieved, the

retrieving program examines it for the existence of errors and passes control to the error analyzer and corrector modules if any errors are found. If no errors are found, the output is passed intact to a new output data set and the next WATFOR program is retrieved. This process is repeated for every program in the WATFOR batch.

When control is passed to the error analyzer and corrector the algorithm, which is shown in the flow chart of figure 5.1, obtains control. This algorithm controls the execution of the corrector and includes the basic error analysis modules. The correction control module and the basic analysis module use interlacing code to partially effect the co-routine interface between the analyzer and the corrector as is shown in figure 3.3 of chapter 3. This corrector control algorithm performs more services for the analyzer and the corrector: it manipulates the WATFOR output so that the statements and error flags can be analyzed separately and merged prior to their being passed to the analyzer and corrector modules.

This correction control algorithm, titled CORRECT, commences with an allocation of the necessary variables. These variables are adjusted so that proper character overlays may be obtained. The character overlays which are used enhance the processing capability of the programmed implementation of this algorithm which is found in Appendix B; they allow for the various sections of a line image to be treated as individual entities rather than a part of the entire line.

After the variable allocation has been made, the input parameter string is examined. The input parameters are given in table 5.1. They are used to request additional output when program modifications must be

Figure 5.1 Controlling error correction module. The basic analysis modules are incorporated into this process.

Table 5.1 Input parameters used by the error corrector

<u>PARAMETER</u>	<u>FUNCTION</u>
DEBUG	request debug output
COPYINPUT	requests that input to the corrector be copied on the printer
COPYOUTPUT	requests that disk output be copied on the printer
PRINT	requests additional printed output (unspecified)

verified. In the programmed implementation the input parameter string is obtained from the PARM.GO field of the EXEC card. The EXEC card is used to specify what procedures are to be executed and the PARM.GO field specifies the character string which is passed to the main program. The input parameter string is examined by the correction control algorithm so that the presence of any keywords (which are recognized by the error analyzer and corrector) within the parameter string are detected. Each keyword parameter that is discovered causes a flag to be set. All of the flags are used throughout the algorithm to specify appropriate user requested actions from the program.

Since the error analyzer and corrector operates on files of programs and files of their compiler outputs, the next section of the implemented program, CORRECT, creates the data necessary to use these and auxiliary files. In the present program one file is required for the FORTRAN input card images and another file is required for the FORTRAN program listing and the FORTRAN program output. Two files are required for output from the error corrector: a temporary output file which contains parts of single programs as they are being corrected, and a new program output file which contains all corrected programs. Several print files are also set up. The print files are used for different kinds of output; the exact kind of output to be printed is specified by the input parameters. All of the files are used intermittently throughout the program.

The termination of the error corrector and analyzer is signaled by the absence of data on the input files. When this condition arises, special action is required for the corrector to terminate normally. The special action is specified immediately after the files have been set

up, but this action is not performed until the end-of-file condition is signaled.

An analysis module is the first of the subordinate modules within CORRECT to gain control. This module is written as an integral part of the controlling module. Immediately following the line image input the card portion of the line is examined for job control information. If the line does contain job control information it is further analyzed to determine whether or not that information is correct. If it is not correct a diagnostic is printed and the user's job is flushed; otherwise processing of the user's job continues. When a job card image is encountered after the basic JCL checking, its analysis is terminated and the card is passed to the temporary new program file. If the end of the user's program is encountered, several of the files must be updated. Both of the input files are updated by positioning them for the analysis of the next job. The temporary new program file is filled, closed, and copied onto the file which contains all corrected programs. A check is then made to determine if any errors were detected during the prior analysis and correction for the job. If errors were detected previously, then a set of post-mortem analysis and correction modules are called. These modules are within the predefined process CRRCTN which is discussed later in this chapter. If this job does not require the use of the post-mortem modules a simple I/O module is invoked. The I/O module is the predefined process RESET. RESET serves to reset the error correction controlling algorithm so that it is ready to process the next incoming job.

As each line of a users job is read, it is analyzed to determine if it is erroneous. At the time a line is found to be erroneous the defined process CRRCTR is invoked. Both analysis and correction modules are used in this predefined process. After the controlling algorithm regains control from CRRCTR the processing commences for the next output line image. This defined process is discussed in detail later in this chapter.

Three defined processes are invoked by the correction control algorithm. The least complex of these defined processes is the one which is implemented as the programmed procedure RESET. The RESET predefined process is described in the flow chart of figure 5.2. The flow chart indicates that the temporary new program output file is closed upon entry to the RESET procedure. It is then changed from an output file to an input file so it may be copied. The output file which contains all programs is opened implicitly during the first call to the RESET procedure; it remains open as an output file through all successive calls to the reset process. The program which has just undergone analysis and correction is then copied from the temporary new program input file onto the output file for all programs. After the copy operation the temporary program file is redefined as an input file and control is returned to the controlling correction module.

The CRRCTN defined process follows the algorithm which is described in the last chapter as closely as possible. A few additions have been made, but these are used to overcome environmental restrictions and to interface the CRRCTN module with the other modules of the error analyzer and corrector. The algorithm which is actually used is described in the flow chart in figure 5.3. The SET and UNSET processes are discussed

Figure 5.2 Defined Process RESET. This process does the housekeeping after the analysis and correction of one program and does the file setup for the processing of the next program.

Figure 5.3 The algorithm for burst error correction and e-tuple error correction as it is currently implemented. The defined processes SET, UNSET, GETR and PUTR are described in detail later. The defined processes ALG-1, ALGO, and ALG1 were discussed in chapter four.

later in this chapter, as are the GETR and PUTR input/output processes. The SET and UNSET procedures are used to interface this module with the rest of the program and the GETR and PUTR procedures are environment-dependent so must be rewritten for different implementations.

After the CRRCTN module has been connected (logically) to the rest of the program, a skeletal dictionary is created. This skeleton contains all of FORTRAN keywords that need to be considered during the execution of the CRRCTN module. The dictionary is completed dynamically during the correction procedures. The dictionary is filled by the use of the same methods that would be used for symbol table building in a translator. Once the skeletal dictionary has been created, the procedure starts to loop through all of the symbols of the program being analyzed and corrected. In other words, the parts of the algorithm which are described below are repeated for every symbol which appears in the source program.

As each symbol is encountered it is either entered into the dictionary or tested to see if it contains a burst error. If it contains a burst error which has length of two or less, correction is attempted. A character register is created as the first step of the correction process (see figure 5.4). The character register contains one bit for each possible character. If a given character is found in the input word the bit for that particular character is set; otherwise, the bit corresponding to that character is reset. A similar character register is created for each dictionary word. The length of the input word and the length of each dictionary word are calculated. The input word is compared to each dictionary word in turn. If it is improbable that the

Figure 5.4 Algorithm for Character Register Creaction. A_c is the string of all possible characters.

dictionary word is the correct spelling of the input word, the dictionary word is ignored and the search continues with the next sequential item in the dictionary. If the dictionary word is a possible correction for the input word then one of three correction algorithms is used. These correction algorithms are discussed in chapter four and seek to find the proper correction for the input word. The correction that they use is found in the dictionary or symbol table.

The SET and UNSET processes are complementary. The SET process sets the temporary new program file to be an input file and UNSET resets the temporary new program output file so that it is ready to start processing the next program. These two procedures are described in figure 5.5.

The SET process must also set up the input character counters and the input buffers. These character counters and input buffers will be used by subsequent input/output procedures.

Two more input/output processes, called GETR and PUTR are required. GETR is used to obtain items from the input string one word at a time. PUTR is used to place items in the output stream one word at a time. These two processes are substantially more complex than the SET and UNSET processes. The details of these processes are described in the flow charts of figures 5.6 and 5.7. A word is defined to be the set of characters which occurs between two delimiters. A delimiter is any character which is called such in the IBM/360 FORTRAN language manual. In some cases a blank may also be used as a delimiter. The GETR procedure requires two parameters: the first parameter is used to pass one word of text to the place from which the GETR procedure was invoked; the second parameter is used to flag the various conditions

Figure 5.5 SET and UNSET Processes

Figure 5.6 The GETR input process conditions are intercepted and the appropriate flags are set.

Figure 5.7 The PUTR Output Procedure

which may have occurred during the execution of the GETR procedure. The possible values that this flag may have are shown in Table 5.2. Each of the values which this flag may assume are alphabetic. A value of "E" indicates that an end-of-data has occurred and that no further calls should be made to the GETR procedure. A value "L" indicates that the word which is returned to the calling process has been obtained from the left hand side of a FORTRAN assignment statement. A value of "R" likewise indicates that the returned word has been obtained from the right hand side of a FORTRAN assignment statement. When the flag has a value of "N" a word of all blanks has been returned to the calling program. This usually applies only when the input data has been obtained from columns one through five of the input card image.

In order to pass one word at a time to the calling procedure GETR uses a buffering process. There are always two card images available to the GETR process. One of these card images is the current card image. All new words are obtained from the current card image. A pointer always points to the next character to be read. As the card image is scanned this pointer is updated. When the end of the card image is encountered the buffered image becomes the current image, a new card is read into the buffer, and the pointer is adjusted for the new current card image. The GETR process is described pictorially in figure 5.6.

Like the GETR process the PUTR process also requires two input parameters. The first parameter contains the text which is to be put into the output buffer. The second input parameter is a flag that is available for general usage. This flag is unused at the time of this writing but may be used as program modifications are made. The PUTR

Table 5.2 Table of Return Flags from Process GETR

<u>FLAG</u>	<u>MEANING</u>
E	end-of-data encountered
L	data comes from the left hand side of a FORTRAN assignment statement
R	data comes from the right hand side of a FORTRAN assignment statement
N	returning data is blank

procedure is unusual since it uses an error condition for normal processing. An output buffer is created incrementally by successive calls to PUTR. This buffer is in the form of a character string. When the buffer is filled a stringrange error condition is signaled. This error condition is intercepted and the buffer emptied before processing is permitted to continue.

When the error condition is intercepted all of the buffers must be reset. This resetting is also required for the internal counters. The intercept processing module is used to effect both of these functions as well as the principle output function. A description of this process is described in figure 5.7.

The CRRCTR process is described in figure 5.8. This module and its subordinate modules attempt to correct at most one erroneous card at a time. This module creates comment flags and places them into the user's new source deck to indicate to the user what corrections have been made.

The first subordinate module is the keypunch correction module. Every card image that is passed to the CRRCTR module undergoes manipulation from the keypunch correction module. The keypunch correction module checks the card image for obvious errors in keypunching and corrects those errors when it finds them. In the current implementation the keypunch correction module is designed to correct errors in cards which were punched on the IBM 029 keypunch but it may be modified so that it corrects errors on cards which were punched by other models. After the keypunch correction module has terminated the error messages which were previously issued by the WATFOR compiler are examined. A

Figure 5.8 Flow of Control within the CRRCTR module. The information gained in the preliminary analysis program is used heavily, although implicitly within the various subordinate modules.

result of the analysis of these messages is that the CRRCTR module can use the analysis to determine which of the individual error correction modules it should next give control. A discussion of all of these modules would be verbose to include here. The actual modules used appear in appendix B. Each of the individual error correction modules has a number by which it is identified; for example, module 03 is the card format and contents module. Any source card formatting error would cause module 03 to obtain control to attempt the correction. After the individual error correction module has completed its job the CRRCTR module again is in control. At this time a terminal analysis module performs any remaining analysis, adjusts the module interfaces, and causes CRRCTR to relinquish control to the controlling error corrector module, CORRECT.

In this chapter a method of creating an actual program to perform user error analysis and correction has been demonstrated. The first phase of such program creation is to design the various algorithms which are to be implemented. In this case these algorithms are those which were developed in the previous chapters, although some subset of those algorithms could have been used. Another phase of the design is the determination of the order of use of the algorithms and the general interfaces among them. The order of use may vary even within the same design but the interfaces should remain the same throughout any chosen design. In the design that is discussed in this chapter each of the algorithms is treated as a module within some modular design.

The second phase of program creation is the actual implementation. During the implementation some problems may arise which cause some of the

program to be redesigned. Design and implementation must be performed concurrently for some period while some modification of the original design is obtained. After the final design is obtained, implementation may proceed directly from this design, but adding the programming conventions which are standard in the language, under the operating system, and for the installation involved.

In the implementation previously described the master module is implemented as the CORRECT procedure. The modules immediately subordinate to the CORRECT module are the CRRCTR and the CRRCTN procedures. Both the CRRCTR and the CRRCTN modules have several submodules which are subordinate to them. In addition there are several input/output modules which may be invoked by many other modules and hence are not subordinate to any one module in particular. The programmed implementation of each of these modules is in the programming language PL/I and PL/I programming conventions are followed. This implementation demonstrates the feasibility of a program in using the theory which was presented earlier in this dissertation.

VI. STATISTICAL PROCESSING FOR AN ERROR CORRECTOR

This chapter describes the post-mortem examination of a group of jobs on which the error corrector has operated. This type of examination can provide fruitful results which are the answer to questions such as:

- 1) Is this type of error correction successful in its operating environment? and
- 2) Can the corrector be maintained in its present state or are modifications desirable?

In order for such testing to be valid special consideration must be given to the design of the experiment, the method of obtaining a sample, how the results are to be analyzed, and what significance the results have.

The data upon which the experiment is performed is collected automatically. The data for the corrector of chapter IV is a random sample of student jobs which use the WATFOR compiler (6). The jobs might be collected by copying all of the jobs of this type onto a storage tape as they are submitted. This, unfortunately, would result in an unwieldy amount of data. To reduce the amount of data a sample was taken of this population of student programs. A time was specified so that all student WATFOR programs which were submitted at that time were copied onto the storage tape. The specified time was chosen so that there would be negligible delay in the students' "turn-around" time.

This method of sampling has the advantage that the costs are kept minimal. The primary cost is computer time. Another advantage is that

no special action is required of the students. Indeed, the computer operator performs the only special action which is required: he must load the copy program and he must mount and dismount the storage tape. This amounts to routine work for the operators.

Another way the costs of obtaining the sample are minimized is by the use of the computer manufacturer's utility program to perform the copy function. The use of this utility initially eliminates the need for any overhead for programming personnel. Thus it can be seen that the costs of data collection are dominated by the cost of computer time. It must be noted that this applies only to data collection and not to preparation of the data for error correction.

By the use of the manufacturer's utility for the copying of the student jobs from the source cards to tape, a single tape containing the sample data is created. This single tape can then be used for basic storage and the work with the sample can be performed with data that has been transferred to a second tape. The method of obtaining the sample and obtaining working data is shown in figure 6.1.

In order to reduce the amount of data which will be used it is possible to modify the transfer of data from the storage tape to the working tape. Two modifications of this basic transfer process have been implemented. These modifications serve primarily to reduce the amount of sample data which must be examined. The first modification of the basic data transfer process consists of taking k batches of student jobs starting from batch i ; where, i and k can be defined at the execution of the transfer. The second modification requires more effort for its implementation but should result in a less biased sample. This

Figure 6.1 Student programs, A, are transferred to storage tape, B, and from the storage tape to the working tape, C.

second modification consists of obtaining k batches of student jobs; where, k is specified at the execution time as in the first method. The fundamental difference, however, is that these k batches are chosen uniformly from all of the student jobs on the storage tape.

There are certain basic factors which must be kept in mind as the experimental design is continued. It is desirable to design the experiment so that the inferences which are drawn are relevant to the evaluation of the corrector. In addition, it is essential to avoid reaching misleading conclusions if it is at all possible. Since the data will primarily be processed by computer, the sample can be large enough to maintain a low probability of reaching an incorrect conclusion.

In order that the inferences which are drawn be relevant, basic facts about the population and the sample must be borne in mind. The total population is considered to be the set of all student jobs which use the WATFOR compiler and which are submitted at the Iowa State University computer center. As a result of the large computer science department at Iowa State University and the large number of other courses which use the Iowa State University computing facilities, most types of student WATFOR programs should be represented. It is not true that the normal production runs for industry would be so represented, but the feasibility of applying the error corrector to that type of industrial program is questionable at best. The penalty for a simple failure in the error corrector is usually so severe in such an industrial environment that the use of a corrector is precluded. Such industrial production programs usually execute for a relatively long time - several hours

as opposed to a few seconds for student runs - so that an attempted correction can cause a loss of several hours of computer time. In summary, it may be said that all of the conclusions drawn apply only to student WATFOR programs.

Design of methods for data collection, statistical error minimization, data summarization, and statistical analysis have been discussed by Sterling and Pollack (58). Using these techniques two programs were written to sample the population of all student WATFOR programs at the Iowa State University Computation Center. This sample was obtained weekdays during the late evening time period for one entire academic quarter.

The two programs which were written to perform the sampling are given in Appendix B. These two programs obtain the sample, manipulate the sample for easier processing, and summarize the data. The data is summarized both pictorially in the form of histograms and numerically by calculating the estimated probability of occurrence of each type of error. The information obtained from the sampling and summarization processes is then incorporated into the error corrector which has been described in the preceding chapters. After the error corrector using this information has been written it may be evaluated by using the decision model of figure 6.2. A complete discussion of this decision model is given in Sterling and Pollack's (58) 1968 book.

Statements in the subsequent position of this chapter apply to the statistical analysis programs which were used in the development of the corrector described in chapter V. Physically obtaining the sample on which the experiment is to work requires a rather large effort.

If it is true that	and it is decided that	
	the corrector worked	the corrector did not work
The error was repaired or corrected	Valid	Invalid
The error was not repaired or corrected	Invalid	Valid

Figure 6.2 Decision model 1 for testing the usefulness of the error corrector (57).

The raw sample is obtained as shown in figure 6.2, but additional processing is desirable. This additional processing allows the installation accounting procedures to operate normally and prepares the data for input to the student compiler. In addition, this processing allows a subset of the initial raw sample to be obtained for use in the performance of the experiment. The conceptual performance of this processor is shown in figure 6.3.

An examination of figure 6.3 indicates that the raw sample is used as input for the processor while two different working samples are output for that same processor. The forms of both working samples are such that they may be used at the Iowa State installation. Working sample A must use a special reader to be used in the experiment but working sample B merely requires an additional job step when it is used as experimental data. As a result, most of the work was performed with working sample B.

The complete processor which is mentioned above is a non-trivial data processing type of program. The listing of this program is given in appendix B and a flow chart for it is given in figure 6.4.

As described in figure 6.4, the processor first initialized several summary counters. These counters are used to gather general statistics about the sample and to summarize the sample itself. This summary counter initialization is followed by the processing function of opening the files. This opening allows the different files to be accessed in prespecified ways and prevents the destruction of the initial data base file. The files which are opened at this point in the program have special functions. One file is the initial data base tape.

Figure 6.3 Initial Processing of the Raw Sample.

Figure 6.4 Statistical Sampling Program

After the above mentioned files have been opened, there is one remaining initialization process. This remaining initialization process consists of obtaining the specified input parameters. If an input parameter is not specified for a particular use then a default value will be assumed for it. There are several of these input parameters. Each of these parameters, its meaning, and its default value are shown in table 1. Of these parameters perhaps the mode parameter needs more explanation. The two modes are random and sequential. Since the data set has sequential organization sequential processing is handled normally but random processing requires special action. To obtain random processing the total number of batches are counted and a random number generator is used to create a random sequence of batch numbers; these batch numbers are then sorted so that processing may continue in sequential order as the unwanted batches are skipped.

After the above initialization has been completed the actual processing takes place. The processing that occurs immediately after the initialization recreates control cards as it transfers selected data from the initial data base to the batch data base. Once this transfer has occurred the two files are closed and the summary of the transfer action is printed. A similar type of processing is performed to obtain the non-batch data base from the batch data base. It should be noted that this final step could sometimes, but not always, be incorporated into the previous steps. The reason that this cannot always be incorporated into the prior steps is that this processing depends on the closure of the batch data base on some occasions. At this time the

Table 6.1 Input Parameters Used in the Raw Sample Processor

Parameter	Default Value	Meaning
#IN	100	The number of batches in the initial data base file which are to be transferred to the other files.
#PRINTED	3	The number of transferred batches that are to be copied on the print file.
MODE	sequential	The way in which the FORTRAN programs are to be obtained from the initial data base file.
SKPBATCH#	0	The number of any one batch that this processor is not to process.
STARTLOC	1	The location in the initial data base where processing is to begin.

raw sample processor may have the logical configuration that is shown in figure 6.5.

In summary the statistical processing consists of four distinct phases:

- 1) the collection of sample data;
- 2) the statistical analysis of sample data;
- 3) the incorporation of the statistical results into the error corrector; and
- 4) the experimental analysis of the results of the error corrector.

Figure 6.5 Logical Configuration of Raw Sample Processor

VII. RESULTS, SUMMARY, AND SUGGESTED FUTURE WORK

The implementation of the error corrector which was discussed in chapter V has been applied to a sample of ten FORTRAN programs. The implementation did not contain a correction module for each type of error, but rather contained only those modules for the most common errors. The error corrector was applied to the programs in the sample of ten with the following results: three programs were corrected or repaired completely; other erroneous programs had some errors corrected or repaired; and no action was taken for any program which was initially correct.

The primary research which is presented in this thesis is a basic theory for the development of an error corrector. The use of this theory should provide one additional debugging tool for the high-level language programmer. Such tools have been few and it is hoped that this theory can be developed to the point where it is useful to the majority of programmers.

Techniques are developed for automatically determining the existence of errors, locating errors, and choosing the most probable correction. Certain new terms are defined in order to facilitate the discussion of the error correction problem. Methods for determining the existence of errors rely heavily on present compiling techniques. Errors are located by using techniques which are modifications of compiling techniques; for example, the translation matrix technique may be used to find the place in the input string where an error has actually occurred. Errors are classified and heuristic techniques are developed for each class of

errors. Some of the classes of errors such as spelling errors may be corrected by previously known techniques, but entirely new methods are developed for most classes.

An error corrector was developed for a selected subset of high-level language programs which use the Waterloo FORTRAN [WATFOR] compiler. The corrector makes use of the information which is available to the compiler and the output from the compiler.

The corrector was implemented using the following steps:

- 1) a random sample of programs which use the WATFOR compiler was obtained;
- 2) statistics were gathered so that programs could be described in terms of the errors that programmers make;
- 3) correction algorithms were developed for the most common errors;
- 4) the algorithms were coded as a correction program; and
- 5) the correction program was applied to the WATFOR job stream.

A significant improvement to the current corrector would be a corrector which incorporates these techniques into the compiler itself. Much work remains to be done in techniques of applying the given methods. The methods can be applied to almost any high-level language but require a great deal of work for any one specific implementation.

Additional work for making a corrector economically feasible is also desirable. Now that the basic techniques have been developed for the correction processes, streamlining these techniques is a logical successor step.

VIII. BIBLIOGRAPHY

1. Alberga, Cyril N. String similarity and misspellings. Association for Computing Machinery Communications 10: 302-313. 1967.
2. Baston, Alan. The organization of symbol tables. Association for Computing Machinery Communications 8: 111-112. 1965.
3. Bell, James R. A new method for determining linear precedence grammars. Association for Computing Machinery Communications 12: 567-569. 1969.
4. Bennet, Richard K. The design of computer languages and software systems: A basic approach. Computers and Automation 18, No. 2: 28-33. 1969.
5. Blair, C. R. A program for correcting spelling errors. Information and Control 3, No. 3: 60-67. 1960.
6. Blatt, John M. Introduction to FORTRAN IV Programming: Using the WATFOR compiler. Pacific Palisades, California, Goodyear Publishing Company. c1968.
7. Block, R. M. The philosophy of automatic error correction. Institute of Radio Engineers, American Institute of Electrical Engineers, Association for Computing Machinery, Western Joint Computer Conference Proceedings 1958: 25-28. c1958.
8. Butnarescu, V. V., ed., Covil, D. L., Zimmerman, W. A., Wilt, J. P. and Pratt, V. W. SLANG: A language and a compiler. Madison, Wisconsin, University of Wisconsin Computing Center Staff Document 4. 1967.
9. Chartres, B. A. and Florentin J. A universal syntax-directed top-down analyzer. Association for Computing Machinery Journal 15: 467-469. 1968.
10. Cotton, Robert M. Ambiguity preventing normal forms for syntax specification. Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia, Pennsylvania. Air Force Directorate of Information Sciences Report AD-285-579. 1962.
11. Crenshaw, John H. Factors influencing the design of a processor for a procedure oriented programming language. Unpublished Master of Science thesis. Carbondale, Illinois, Library, Southern Illinois University. 1968.
12. Damereau, F. J. A technique for computer detection and correction of spelling errors. Association for Computing Machinery Communications 7: 171-176. 1964.

13. Davidson, Leon. Retrieval of misspelled names in an airlines passenger record system. Association for Computing Machinery Communications 5: 169-171. 1962.
14. Feldman, Jerome and Gries, David. Translator writing systems. Association for Computing Machinery Communications 11: 77-113. 1968.
15. Ferguson, H. E. and Berner E. Debugging systems at the source language level. Association for Computing Machinery Communications 6: 430-432. 1963.
16. Flores, Ivan. Computer sorting. Englewood Cliffs, New Jersey, Prentice-Hall, Inc. c1969.
17. Floyd, Robert W. An algorithm defining ALGOL assignment statements. Association for Computing Machinery Communications 3: 170-171. 1960.
18. Floyd, Robert W. Syntactic analysis and operator precedence. Association for Computing Machinery Journal 10: 316-333. 1963.
19. Foster, J. M. A syntax improving program. The Computer Journal 11: 31-34. 1968.
20. Freeman, David N. Error correction in CORC, the Cornell University computing language. American Federation of Information Societies Fall Joint Computer Conference Proceedings 1964 [part 1]: 15-34. Baltimore, Maryland. Spartan Books, Inc. c1964.
21. Ghandour, Ziad Jamil. Formal systems and syntactical analysis. Unpublished Doctor of Philosophy dissertation. New Haven, Connecticut, Library, Yale University. c1968.
22. Green, J., organizer, Opler, A., DiFiorino, A. C. and Gorn, J. Symposium on languages for processor construction. Information Processing 1962. International Federation of Information Processing Societies Congress Proceedings 1962: 513-517. Amsterdam, Netherlands, North Holland Publishing Company. c1963.
23. Grems, Mandalay and Porter, R. E. A truly automatic computing system. Institute of Radio Engineers, American Institute of Electrical Engineers, Association for Computing Machinery Western Joint Computer Conference Proceedings 1956: 10-21. 1956.
24. Gries, D. The use of transition matrices in compiling. Stanford University Computer Science Department Report Number 57. Clearinghouse for Federal Scientific and Technical Information PB 176 765. 1967.

25. Gries, D. Use of transition matrices in compiling. Association for Computing Machinery Communications 11: 26-34. 1968.
26. Gusev, L. A. and Smirnova, I. M. Languages, grammars, and abstract automata models I (survey) [English translation]. Automation and Remote Control USSR 1968: 587-605. 1968.
27. Gusev, L. A. and Smirnova, I. M. Languages, grammars, and abstract automata models II (survey) [English translation]. Automation and Remote Control USSR 1968: 757-774. 1968.
28. Harrison, Michael, A. On the error correcting capacity of finite automata. Information and Control 8: 430-450. 1965.
29. Hopcroft, J. E. and Ullman, J. D. Error correction for formal languages. Princeton, New Jersey. Princeton University Department of Electrical Engineering Digital Systems Laboratory Technical Report 52. 1966.
30. Hopkins, Martin. SABRE PL/I. Datamation 14, No. 12: 35-38. 1968.
31. Ingberman, Peter Zilahy. A syntax-oriented translator. New York, New York, Academic Press Inc. c1966.
32. Irons, Edgar T. An error correcting parse algorithm. Association for Computing Machinery Communications 6: 669-673. 1963.
33. Irons, Edgar T. The structure and use of the syntax directed compiler. Institute for Defense Analysis, Princeton, New Jersey, Working Paper 41. 1962.
34. Irons, Edgar T. A syntax directed compiler for ALGOL 60. Association for Computing Machinery Communications 4: 51-55. 1961.
35. Iverson, Kenneth E. A programming language. New York, New York, John Wiley and Sons, Inc. c1962.
36. Karush, Arnold D. Program quality assurance. Datamation 14, No. 10: 61-66. 1968.
37. Keese, W. M. and Hukey, H. D. An algorithm for the translation of ALGOL statements. Information Processing 1962. International Federation of Information Processing Societies Congress Proceedings 1962. Pp. 498-502. Amsterdam, Netherlands, North Holland Publishing Company. c1963.
38. Knuth, Donald E. On the translation of languages from left to right. Information and Control 8: 607-639. 1965.

39. Laver, Murray. Users influence on computer systems design. *Data-mation* 15, No. 10: 107-116. 1969.
40. Lewis, P. M. and Stearns, R. E. Syntax-directed translation. *Association for Computing Machinery Journal* 15: 465-488. 1968.
41. Lietzke, Marjorie P. A method of syntax checking ALGOL 60. *Association for Computing Machinery Communications* 7: 475-478. 1963.
42. Marsaglia, George and Bray, T. A. One-line random number generators and their use in combinations. *Association for Computing Machinery Communications* 11: 757-759. 1968.
43. Morgan, Howard L. Spelling correction in systems programs. *Association for Computing Machinery Communications* 13: 90-94. 1970.
44. Morris, Frank. Random input search techniques: A timely review. *Software Age* 4, No. 1: 8-12. 1970.
45. Naur, Peter, ed. Revised report on the algorithmic language ALGOL 60. *Association for Computing Machinery Communications* 6: 1-17. 1963. [Published simultaneously in the *Computer Journal* and in *Numerische Mathematik*.]
46. Paul, M. ALGOL 60 processors and a processor generator. *Information Processing 1962. International Federation of Information Processing Congress Proceedings 1962*: 493-497. Amsterdam, Netherlands, North Holland Publishing Company. c1963.
47. Reeves, C. M. Description of syntax-directed translator. *The Computer Journal* 10: 244-255. 1967.
48. Ritland, Steven Rolfe. Implementation of a general purpose simulation language using a syntax directed translator. Unpublished Master of Science thesis. Ames, Iowa, Library, Iowa State University. 1968.
49. Samelson, K. Programming languages and their processing. *Information Processing 1962. International Federation of Information Processing Societies Congress Proceedings 1962*. Amsterdam, Netherlands, North Holland Publishing Company. c1963.
50. Samelson, K. and Bauer, F. L. Sequential formula translation. *Association for Computing Machinery Communications* 3: 76-83. 1960.
51. Schneider, Victor B. Syntax-checking and parsing of context free languages by pushdown store automata. *American Federation of Information Processing Societies Spring Joint Computer Conference 1967*: 685-697. Washington, D. C., Thompson Books. c1967.

52. Schneider, Victor B. A translator system for the EULER programming language. University of Maryland Computer Science Center technical report 69-85. 1969.
53. Shapiro, Robert M. and Zand, Louis J. A description of the input language for the compiler generator system. Wakefield, Massachusetts, Massachusetts Computer Associates. 1963.
54. Sherman, P. M. Table look-at techniques. Association for Computing Machinery Communications 4: 172-175. 1961.
55. Smith, William Bridges. Error control in formal languages. Unpublished Doctor of Philosophy dissertation. Philadelphia, Pennsylvania, Library, University of Pennsylvania. 1968.
56. Spanier, E. Grammars and languages. Mathematical Association of America Monthly 76: 335-342. 1969.
57. Sterling, Theodor D. and Pollack, Seymour V. Introduction to statistical data processing. (Prentice-Hall series in automatic computation) Englewood Cliffs, New Jersey, Prentice Hall, Inc. c1968.
58. Thorelli, Lars Erik. Automatic correction of errors in text. Nordisk Tidskrift for Informations - Behandling [BIT] 2: 45-62. 1962.
59. Torrii, Koji, Kasami, Tadoa and Ozaki, Hiroshi. On the complexity of the recognition algorithm of context-free language. Electronics and Communications in Japan [English edition]. Translated for the Institute of Electrical and Electronic Engineers by Scripta Electronica 50: 102-109. 1967.
60. Ullian, Joseph S. Parial algorithm problems for context free languages. Information and Control 11: 80-101. 1967.
61. Wilkes, M. V. The outer and inner syntax of a programming language. The Computer Journal 11: 260-263. 1968.
62. Wirth, Niklaus and Weber, Helmut. EULER: A generalization of ALGOL and its formal definition: part I. Association for Computing Machinery Communications 9: 13-23, 25. 1966.
63. Wirth, Niklaus and Weber, Helmut. EULER: A generalization of ALGOL and its formal definition: part II. Association for Computing Machinery Communications 9: 89-99. 1966.

TX. ACKNOWLEDGEMENTS

The author wishes to thank Mr. R. A. Sharpe for his guidance and advice and for suggesting the research topic. The author also expresses sincere thanks to Dr. R. J. Lambert for his advice and time spent in the preparation of this dissertation.

Thanks are also due to Mrs. Jody Jacobsen for doing the preliminary typing, to Mrs. Sheila Hilts for typing the manuscript and to the author's wife for proofreading the manuscript.

X. APPENDIX A

```

/*          SYNTACTIC ANALYSIS USING FLOYD'S 1961 MATRIX SCHEME. */
(SUBRG,STRG,SIZE):
ALGOLA: PROC OPTIONS(MAIN);
DCL      M(0:16,0:16) CHAR(1) INIT CALL MFILL,
        SS CHAR(80),
        (S(80) DEF SS POS(1),R(0:80),LMDEX(0:16) STATIC INIT
        ('O','.', 'T','1','2','G','N','*','I','V','E','U','"',
        '(', '|','C','='),
        TMDEX(0:16) STATIC INIT('O','.', 'T','G','I','S',
        'E','"', '#',
        ')','%', ',', ';', 'P','B','Z','Y'),
        F) CHAR(1),
        RSTR CHAR(81) DEF R,
        TITLE_ CHAR(60) STATIC INIT
        ('TRANSFORMATION AND ANALYSIS MATRIX'),
        G ENTRY(CHAP(1)) RETURNS(CHAP(1)), (K,L,I,J) FIXED BIN(15,0),
        MFILL ENTRY;
/*

```

IDENTIFICATION:

PROGRAM-ID: ALGOLA.
 AUTHOR: G. E. HEDRICK.
 INSTALLATION: IOWA STATE UNIVERSITY COMPUTATION CENTER.
 DATE-WRITTEN: 10 OCTOBER 1969.
 SECURITY: UNCLASSIFIED.
 SOURCE-COMPUTER: IBM 360/65.
 OBJECT-COMPUTER: IBM 360/65.
 OPERATING SYSTEM: OS/MVT.
 MEMORY SIZE: HIGH SPEED CORE: 64K BYTES.
 BULK CORE: 0 BYTES.

DESCRIPTION OF THE PROBLEM:

THE PROBLEM TO BE SOLVED IS TO RECOGNIZE WHETHER OR NOT A GIVEN INPUT STRING IS SYNTACTICALLY CORRECT. IF THE INPUT STRING IS IN ERROR A TRANSLATION PROCESS WOULD HAVE TO BE INTERRUPTED AT THE POINT WHERE THE ERROR WAS DETECTED. IF IT IS TOTALLY CORRECT THEN TRANSLATION COULD CONTINUE WITH A SEMANTIC ANALYSIS.

METHOD OF SOLUTION:

THE COMPLETE METHOD OF SOLUTION IS DESCRIBED IN FLOYD'S 1961 ARTICLE IN THE COMMUNICATIONS OF THE ACM. THE BASIC PROCESS IS TO:

1. GET A CHARACTER FROM THE INPUT STRING;
2. TRANSFORM THIS CHARACTER INTO ONE OF THE TYPES WHICH IS RECOGNIZABLE AS AN INDEX OF THE TRANSFORMATION MATRIX;
3. PERFORM A REDUCTION ON THE INPUT STRING, IF IT IS POSSIBLE, USING THE TRANSFORMATION MATRIX;
4. CHECK TO DETERMINE IF THE INPUT STRING HAS BEEN REDUCED TO THE DISTINGUISHED SYMBOL;
5. REPEAT STEPS 1-4 UNTIL EITHER:
 - A) THE INPUT STRING HAS BEEN REDUCED TO THE DISTINGUISHED SYMBOL, OR
 - B) THERE ARE NO MORE INPUT CHARACTERS AND THERE ARE NO MORE REDUCTIONS TO BE MADE.

DESCRIPTION OF INPUT:

ANY EIGHTY BYTE (OR SMALLER) CHARACTER STRING WILL BE ACCEPTED AS INPUT TO THIS PROGRAM. THE ALGORITHM WILL THEN DETERMINE IF THE INPUT STRING IS SYNTACTICALLY CORRECT.

DESCRIPTION OF OUTPUT:

THERE ARE THREE CLASSES OF OUTPUT FROM THIS PROGRAM. THEY ARE OUTPUT IN THE ORDER OF CLASS ONE FOLLOWED BY THE MERGED CLASSES TWO AND THREE. THE THREE CLASSES ARE:

1. THE TRANSFORMATION AND ANALYSIS MATRIX (1 PAGE);
2. A COPY OF THE INPUT STRING (1 LINE);
3. A STATEMENT INDICATING THE CORRECTNESS (OR INCORRECTNESS) OF THE PRECEDING INPUT STRING (1 LINE).

SYNTAX SPECIFICATIONS:

<A.S.>::=<LPL><AE>;
<LPL>::=<LP>
<LP>::=<V>|<PI>=
<V>::=<SIM.V>|<SUB.V>
<SUB.V>::=<AI>'|'|<SL>%
<AI>::=<I>
<SL>::=<SE>|<SL>,<SE>
<SE>::=<AE>
<SIM.V>::=<VI>
<VI>::=<I>
<AE>::=<SAE>
<SAE>::=<T>|<AOP><T>|<SAE><AOP><T>
<T>::=<F>|<T><MOP><F>
<F>::=<P>|<F><P>
<P>::=<U.N.>|<V>|(<AE>)
<U.N.>::=<D>|<U.N.><D>
<PI>::=<I>
<I>::=<L>|<I><L>|<I><D>

NOTE 1.

```

<MNP>::=*|/
<ADP>::=+|-
<L>::=A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<D>::=0|1|2|3|4|5|6|7|8|9

```

NOTE 2.

NOTE 1. '|' IS USED TO REPRESENT | SINCE | HAS A USE IN THE METALANGUAGE.

NOTE 2. THIS GRAMMAR IS BASED UPON THE DESCRIPTION OF THE ALGOL ASSIGNMENT STATEMENT. THE BRACKETS HAVE BEEN CHANGED TO THE | AND % SYMBOLS, AND @ HAS REPLACED THE EXPONENTIATION SYMBOL.

REFERENCE:

FLOYD, ROBERT W. AN ALGORITHM DEFINING ALGOL ASSIGNMENT STATEMENTS. ASSOCIATION FOR COMPUTING MACHINERY COMMUNICATIONS 3: 170-171.

```

*/
/*
*/

```

MFILL: PROC; M='0';

```

/*
THIS ROUTINE IS USED TO CREATE THE TRANSFORMATION MATRIX. THIS
MATRIX IS USED TO SPECIFY THE SYNTAX AND TO TRANSLATE THE
SPECIFIED SYNTAX. THE TRANSLATOR COULD BE MADE MORE GENERAL
BY HAVING THE TRANSLATION MATRIX BE INPUT. THE CURRENT
TRANSLATOR IS KEPT SLIGHTLY MORE EFFICIENT BY HAVING THIS
MATRIX PRESPECIFIED.

```

NOTICE THAT CODE COULD BE GENERATED BY THE USE OF A COMPANION MATRIX TO SPECIFY THE SEMANTICS.

```

*/

```

```

M(2,3),M(3,3),M(4,3),M(7,3)='*';
M(1,3),M(5,1),M(6,3)='N'; M(5,3)='G';
M(5,2),M(6,2)='1'; M(2,7),M(3,7)='2';
M(8,3),M(8,4)='I'; M(8,16),M(9,16)='?';
M(6,7),M(6,8),M(7,7),M(7,8),M(8,7),M(8,8),M(9,7),M(9,8),
M(10,7),M(10,8),M(5,7),M(5,8)='U';
M(5,9),M(6,9),M(7,9),M(8,9),M(9,9),M(10,9),M(11,13),
M(12,13),M(15,13)='P';
M(5,10),M(6,10),M(7,10),M(8,10),M(9,10),M(10,10),
M(11,14),M(12,14),M(15,14)='B';
M(5,11),M(6,11),M(7,11),M(8,11),M(9,11)='C';
M(11,15),M(12,15)='Z';
M(13,13)='E'; M(14,14)='S'; M(16,15)='Y';
M(8,5)='V'; M(8,6)='E';
M(5,12),M(6,12),M(7,12),M(8,12),M(9,12),M(10,12)='Z';
M(10,11)='C';
PUT FILE(SYSPRINT) EDIT(TITLE_,TMDEX)
(PAGE, X(30),A,SKIP(3),X(1),      17 (X(5),A)      );
DO I=0 TO 16;
PUT FILE(SYSPRINT) EDIT(LMDEX(I)) (SKIP(3),A);
PUT FILE(SYSPRINT) EDIT(M(I,*)) (X(5),A);
END;
PUT FILE(SYSPRINT) PAGE;
RETURN;
END MFILL;

```

/*

*/

G:

```

PROC(Q) CHAR(1);
DCL Q CHAR(1);
/*
THIS FUNCTION IS USED TO TRANSFORM INPUT CHARACTERS INTO
SYNTACTIC TYPES WHICH MAY BE RECOGNIZED AND USED VIA THE
TRANSFORMATION MATRIX.
*/
/* A LETTER TRANSFORMS INTO AN I FOR A POSSIBLE IDENTIFIER. */
IF ('A'<=Q)&('Z'>=Q) THEN RETURN('I');ELSE
/* A DIGIT TRANSFORMS INTO A G FOR LATER ANALYSIS. */

```

```

IF ('0'<=Q)&('9'>=Q) THEN RETURN('G'); ELSE
/* OPERATORS HAVE SPECIAL CODES. */
IF (Q='*')|(Q='/')|(Q='@') THEN RETURN(''); ELSE
IF (Q='+')|(Q='-') THEN RETURN('#');
RETURN(Q);
END G;

/*

*/

ON ERROR GO TO E1;
ON ENDFILE(SYSIN) GO TO STP;
R=' ';
LOOP1: READ FILE(SYSIN) INTO (SS);
/* GUARD AGAINST A FORGOTTEN SEMI-COLON. */
S(MAX(INDEX(SS,' '),INDEX(SS,',')))=';';
PUT FILE(SYSPRINT) EDIT(SS) (SKIP,A);
I,J=0; R(0)='0';
LOOP2: I=I+1; J=J+1; R(J)=G(S(I));
BACK1:
/* CHANGE NON-NUMERIC SUBSCRIPTS INTO A USUABLE FORM. */
DO L=1 TO 16 BY 1 WHILE(R(J-1)~=LMDEX(L)); END; L=MOD(L,17);
NXT1: DO K=1 TO 16 BY 1 WHILE(R(J)~=TMDEX(K)); END; K=MOD(K,17);
F=M(L,K);
IF F~='0' THEN DO;
    J=J-1;
    R(J)=F;
    GO TO BACK1;
END;
IF S(I) ~='; ' THEN GO TO LOOP2;
IF R(1)='?' THEN PUT FILE(SYSPRINT)
    EDIT('WELL FORMED SYNTACTICAL FORMULA.')
    (SKIP,A);
ELSE E1:
    PUT FILE(SYSPRINT) EDIT('***** ERROR *****')
    (SKIP,A);
GO TO LOOP1;
STP: RETURN;

```

END ALGOLA;

```
/*                      SYNTAX DIRECTED COMPILING.                      */
(SUBRG,STRG,SIZE):
SYNTAX1:PROC OPTIONS(MAIN,REENTRANT);
/*
```

IDENTIFICATION:

PROGRAM-ID: SYNTAX1.
AUTHOR: G. E. HEDRICK.
INSTALLATION: IOWA STATE UNIVERSITY COMPUTATION CENTER.
DATE-WRITTEN: 14 OCTOBER 1969.
SECURITY: UNCLASSIFIED.
SOURCE-COMPUTER: IBM 360/65.
OBJECT-COMPUTER: IBM 360/65.
OPERATING SYSTEM: OS/MVT
MEMORY SIZE: HIGH SPEED CORE: 64K BYTES.
 BULK CORE: 0 BYTES.

111

DESCRIPTION OF THE PROBLEM:

THE PROBLEM TO BE SOLVED IS TO RECOGNIZE WHETHER OR NOT A GIVEN INPUT STRING IS SYNTACTICALLY CORRECT. IF IT IS THEN THE PROGRAM WILL GENERATE THE PROPER CODE. THIS CODE SHOULD BE A DESCRIPTION OF THE SEMANTICS FOR THAT PARTICULAR SYNTACTIC ENTITY.

METHOD OF SOLUTION:

THE ENTIRE METHOD OF SOLUTION IS GIVEN IN IRON'S ARTICLES.
THE SKELETON OF THE PROCESS IS:

1. SET UP A (SYNTACTIC) GOAL; INITIALLY THIS GOAL WILL BE THE DISTINGUISHED SYMBOL;
2. IF THIS GOAL IS NOT A TERMINAL SYMBOL, STACK THE GOAL AND REPEAT FROM STEP 1;
3. INCREMENT THE INPUT CHARACTER STRING POINTER;
4. WHEN THE INPUT STRING IS EXHAUSTED, STOP;
5. IF THE CURRENT INPUT CHARACTER IS THE ONE WHICH WAS REQUESTED, SET UP THE NEXT GOAL AND REPEAT FROM 1;
6. DECREMENT THE INPUT CHARACTER STRING POINTER;
7. UNSTACK THE SAVED INFORMATION AND REPEAT FROM 1 USING THE NEXT ALTERNATIVE.

DESCRIPTION OF INPUT:

ANY EIGHTY BYTE (OR SMALLER) CHARACTER STRING WILL BE ACCEPTED AS INPUT TO THIS PROGRAM. THE ALGORITHM WILL THEN DETERMINE IF THE INPUT STRING IS SYNTACTICALLY CORRECT AND IF IT IS THE PROPER CODE WILL BE GENERATED.

DESCRIPTION OF OUTPUT:

THERE ARE THREE CLASSES OF OUTPUT FROM THIS PROGRAM. CLASS ONE IS A COPY OF THE INPUT STRING AND IS FOLLOWED EITHER BY CLASS TWO OR BY CLASS THREE. CLASS TWO OUTPUT IS AN INDICATION THAT THERE HAS BEEN A SYNTAX ERROR. CLASS THREE OUTPUT IS THE CODE CORRESPONDING TO A SYNTACTICALLY CORRECT STATEMENT.

SYNTAX:

<ASSIGNMENT STATEMENT>::=<V>:=<E>
<E>::=<N>|<V>|<E><OP><E>
<V>::=<L>|<V><L>|<V><D>
<L>::=X|Y|Z
<N>::=<D>|<N><D>
<D>::=0|1|A|B
<OP>::=+|-|*|/

BRIEF DESCRIPTION OF SEMANTICS:

THE BASIC ASSIGNMENT STATEMENT REQUIRES THAT AN EXPRESSION BE EVALUATED AND STORED IN A SPECIFIC LOCATION. THIS LOCATION IS SPECIFIED BY THE VARIABLE ON THE LEFT-HAND SIDE OF THE REPLACEMENT SYMBOL. EXPRESSIONS CONSIST OF VARIABLES AND NUMBERS JOINED BY ARITHMETIC OPERATORS. THERE IS NO PRECEDENCE AMONG THE ARITHMETIC OPERATORS AND THE EVALUATION OF AN EXPRESSION PROCEEDS STRICTLY FROM LEFT TO RIGHT. NOTICE THAT THE SEMANTICS ARE IN NO WAY DERIVED FROM THE ALGOL ASSIGNMENT STATEMENT, BUT ARE ESSENTIALLY THE INVERSE OF IVERSON'S RIGHT TO LEFT EVALUATION RULE.

REFERENCES:

-
1. IRONS, EDGAR T. A SYNTAX DIRECTED COMPILER FOR ALGOL60. ASSOCIATION FOR COMPUTING MACHINERY COMMUNICATIONS 4: 55-51.
 2. _____ THE STRUCTURE AND USE OF THE SYNTAX DIRECTED COMPILER. INSTITUTUE FOR DEFENSE ANALYSIS WORKING PAPER 41. PRINCETON, NEW JERSEY. 1962.

3. ----- THE STRUCTURE AND USE OF THE SYNTAX
DIRECTED COMPILER. GOODMAN, RICHARD(ED.). ANNUAL REVIEW IN
AUTOMATIC PROGRAMMING. NEW YORK, NEW YORK. THE MACMILLAN
COMPANY. C1963. 207-227.
4. IVERSON, KENNETH E. A PROGRAMMING LANGUAGE. NEW YORK,
NEW YORK. JOHN WILEY AND SONS, INC. C1962.

*/
DCL

```
CARD CHAR(80),
CCARD(80) CHAR(1) DEF CARD POS(1),
PRINT STREAM PRINT OUTPUT EXTERNAL FILE,
(I,K,NOS) FIXED BIN(15,0),
STACK(0:64) CHAR(8) VAR,
TMPS(0:9) CHAR(5) STATIC INIT
('TEMP0','TEMP1','TEMP2','TEMP3','TEMP4','TEMP5','TEMP6',
 'TEMP7','TEMP8','TEMP9'),
OPST(0:64) CHAR(4) VAR,
(OP,D,L) ENTRY(CHAR(*)) RETURNS(BIT(1)),
(N,V,E,AS) ENTRY RETURNS(BIT(1));
```

/*

*/
OP:

```
PROC(COE) RECURSIVE BIT(1);
DCL COE CHAR(*),
CODES(0:4) CHAR(3) STATIC INIT
(' ','ADD','SUB','MUL','DIV'),
OPS(4) CHAR(1) STATIC INIT ('+','-','*','/');
DCL OPX CHAR(4) DEF OPS;
/*
*/
I=I+1;
COE=CODES(INDEX(OPX,CCARD(I)));
IF COE = ' ' THEN RETURN('1'B);
I=I-1;
```

```

        RETURN('0'B);
END DP;

/*
*/
D:  PROC(DIG) RECURSIVE BIT(1);
    DCL  DIG CHAR(*),
        DIGS(4) CHAR(1) STATIC INIT('0','1','A','B');
    DCL DIGX CHAR(4) DEF DIGS;
    /*
    */
    I=I+1;
    IF INDEX(DIGX,CCARD(I))=0 THEN DO; I=I-1; RETURN('0'B);
    END; ELSE DO; DIG=CCARD(I); RETURN('1'B); END;
END D;

/*
*/
L:  PROC(LETR) RECURSIVE BIT(1);
    DCL  LETR CHAR(*),
        LETS(3) CHAR(1) STATIC INIT('X','Y','Z');
    DCL LETX CHAR(3) DEF LETS;
    /*
    */
    I=I+1;
    IF INDEX(LETX,CCARD(I))=0 THEN DO;
        I=I-1; RETURN('0'B); END;
    ELSE DO;
        LETR=CCARD(I);
        RETURN('1'B);
    END;
END L;

/*
*/
N:  PROC RECURSIVE BIT(1);
    DCL  NUM CHAR(8) VAR,
        DIG CHAR(1);
    /*
    */

```

```

      NUM='';
      IF D(DIG) THEN NUM=DIG; ELSE RETURN('0'B);
      IF D(DIG) THEN DO;
B1:      NUM=NUM||DIG ;
          GO TO B1;
          END;
          ELSE DO;
          K=K+1;
          OPST(K)='ENA';
          STACK(K)=NUM;
          RETURN('1'B);
      END N;

/*
*/
V:      PROC RECURSIVE BIT(1);
      DCL  VAR CHAR(8) VAR,
          LETR CHAR(1);
          /*
          */
          VAR='';
          IF L(LETR) THEN VAR=LETR; ELSE RETURN('0'B);
          IF L(LETR) THEN DO;
B1:      VAR=VAR||LETR;
B2:      GO TO B1;
          END;
          IF D(LETR) THEN GO TO B2;
          K=K+1;
          STACK(K)=VAR;
          OPST(K)='L';
          RETURN('1'B);
      END V;

/*
*/
E:      PROC RECURSIVE BIT(1);
      DCL  CDE CHAR(3);
      DCL  KP FIXED PIN(15,0);
          /*

```

```

        */
        IF V IN THEN DO;
        IF OP(CDE) THEN DO;
            KP=K+1;
            IF E THEN DO;
                OPST(KP)=CDE;
                RETURN('1'B);
            END;
        END;
        RETURN('1'B);
    END;
    RETURN('0'B);
END E;

/*
*/
AS:    PROC RECURSIVE BIT(1);
        IF V THEN DO;
            OPST(K)='S';
            I=I+1;
            IF SUBSTR(CARD,I,2)=':=' THEN DO;
                I=I+1;
                IF E THEN DO;
                    IF CCARD(I+1)~=' ' THEN GO TO E1;
                    RETURN('1'B);
                END;
            END;
        END;
        RETURN('0'B);
    END AS;

/*

*/
R1:    ON ENQFILE(SYSIN) GO TO R2;
        READ FILE(SYSIN) INTO(CARD);
        PUT FILE(PRINT) SKIP LIST(CARD);
        NDS,K,I=0;

```

```
IF AS THEN DO I=2 TO K BY 1,1;  
    PUT FILE(PRINT) EDIT  
        (OPST(I),STACK(I))  
    (SKIP,COL(10),A,COL(20),A);  
END: ELSE  
E1: PUT FILE(PRINT) EDIT('***** SYNTAX ERROR *****') (SKIP,A);  
GO TO R1;  
R2: RETURN;  
END SYNTAX1;
```

THIS PAGE
WAS INTENTIONALLY
LEFT BLANK

XI. APPENDIX B

```

/*          JCL UPDATE AND JOB TRANSFER PROGRAM.  */
(SUBRG,STRG,SIZE):
GETSAMP:PROC(PARM) OPTIONS(MAIN);
/*
*/

```

```

DCL      1 OUTPUT_RECORD UNALIGNED STATIC,
          2 TOTBATCHT CHAR(35) INIT
            ('TOTAL NUMBER OF BATCHES TRANSFERRED'),
          2 #BATCHT FIXED BIN(31,0) INIT(0),
          2 TOTWATFT CHAR(39) INIT
            ('TOTAL NUMBER OF WATFOR JOBS TRANSFERRED'),
          2 #WATFT FIXED BIN(31,0) INIT(0),
          2 TOTBATCH CHAR(23) INIT
            ('TOTAL NUMBER OF BATCHES'),
          2 #BATCH FIXED BIN(31,0) INIT(0),
          2 TOTWAT CHAR(27) INIT
            ('TOTAL NUMBER OF WATFOR JOBS'),
          2 #WAT FIXED BIN(31,0) INIT(0),
          2 MXWAT CHAR(39) INIT
            ('MAXIMUM NUMBER OF WATFOR JOBS PER BATCH'),
          2 #MXWAT FIXED BIN(31,0) INIT(0),
          2 MNWAT CHAR(39) INIT
            ('MINIMUM NUMBER OF WATFOR JOBS PER BATCH'),
          2 #MNWAT FIXED BIN(31,0) INIT(1000),
          2 BWAT CHAR(51) INIT
            ('NUMBER OF WATFOR JOBS TRANSFERRED (LAST BATCH ONLY)'),
          2 #BWAT FIXED BIN(31,0) INIT(0),
          2 MEANWAT CHAR(45) INIT
            ('AVERAGE(MEAN) NUMBER OF WATFOR JOBS PER BATCH'),
          2 AVEWAT FIXED BIN(31,0) INIT(0),
          2 STPOS CHAR(40) INIT
            ('TRANSFER OF JOBS PROCEEDED FROM LOCATION'),
          2 STPOS# FIXED BIN(31,0);
DCL      1 OUTPUT_RECORD_A UNALIGNED STATIC,
          2 HOW_SAMPLE CHAR(36) INIT
            ('JOBS WERE OBTAINED FROM STORAGE TAPE'),

```

```

MCDK0002
MCDK0003
MCDK0004
MCDK0005
MCDK0006
MCDK0007
MCDK0008
MCDK0009
MCDK0010
MCDK0011
MCDK0012
MCDK0013
MCDK0014
MCDK0015
MCDK0016
MCDK0017
MCDK0018
MCDK0019
MCDK0020
MCDK0021
MCDK0022
MCDK0023
MCDK0024
MCDK0025
MCDK0026
MCDK0027
MCDK0028
MCDK0029
MCDK0030
MCDK0031
MCDK0032
MCDK0033
MCDK0034
MCDK0035
MCDK0036
MCDK0037

```


2 WAYSAMPLE CHAR(12),
 2 STP CHAR(19) INIT('STORAGE TAPE NUMBER'),
 2 STP# CHAR(6) INIT('TP5058'),
 2 SAMPTP CHAR(18) INIT('SAMPLE TAPE NUMBER'),
 2 SAMPTP# CHAR(6) INIT('TP0502'):
 DCL XIN CHAR(80),
 X1 CHAR(1) DEF XIN,
 X2 CHAR(2) DEF XIN,
 X4 CHAR(4) DEF XIN,
 X5 CHAR(5) DEF XIN,
 ACCOUNT_FIELD CHAR(5) DEF XIN POS(16),
 XOUT CHAR(80),
 Y CHAR(81),
 Z CHAR(80) DEF Y POS(2),
 NEWACCOUNT CHAR(5) STATIC INIT('A0246'),
 SLASH_STOP CHAR(5) STATIC INIT('/STOP'),
 SLASH_DATA CHAR(5) STATIC INIT('/DATA'),
 SLASH CHAR(1) STATIC INIT('/'),
 SLASHES CHAR(2) STATIC INIT('//'),
 SLASH_STAR CHAR(2) STATIC INIT('/:'),
 PARM CHAR(40) VAR,
 WORK1 CHAR(80),
 WORK CHAR(80),
 SLASH_JOB CHAR(4) STATIC INIT('/JOB'),
 PRECEDING_CARD CHAR(5) STATIC INIT(' '),
 (TP5058,TP0502.DSK001,PRINT,SYSPRINT) FILE EXT,
 RECORD_COUNT FIXED BIN(31,0),
 #IN FIXED BIN(31,0) STATIC INIT(100),
 #PRINTED FIXED BIN(31,0) STATIC INIT(003),
 #BWAT FIXED BIN(31,0) STATIC INIT(0),
 SKPBATCH# FIXED BIN(31,0) INIT(0),
 BATCHES_ FIXED BIN(31,0) STATIC INIT(0),
 STARTLOC FIXED BIN(31,0) STATIC INIT(1),
 XI CHAR(81),
 XI1 CHAR(1) DEF XI,
 XI2 CHAR(80) DEF XI POS(2),
 MODE CHAR(12) VAR:

MCDK0038
 MCDK0039
 MCDK0040
 MCDK0041
 MCDK0042
 MCDK0043
 MCDK0044
 MCDK0045
 MCDK0046
 MCDK0047
 MCDK0048
 MCDK0049
 MCDK0050
 MCDK0051
 MCDK0052
 MCDK0053
 MCDK0054
 MCDK0055
 MCDK0056
 MCDK0057
 MCDK0058
 MCDK0059
 MCDK0060
 MCDK0061
 MCDK0062
 MCDK0063
 MCDK0064
 MCDK0065
 MCDK0066
 MCDK0067
 MCDK0068
 MCDK0069
 MCDK0070
 MCDK0071
 MCDK0072
 MCDK0073
 MCDK0074

/*

IDENTIFICATION:

PROGRAM-ID: GETSAMP.
AUTHOR: G. E. HEDRICK.
INSTALLATION: IOWA STATE UNIVERSITY COMPUTATION CENTER.
DATE-WRITTEN: 8 DECEMBER 1969.
SOURCE-COMPUTER: IBM 360/65.
OBJECT-COMPUTER: IBM 360/65.
OPERATING SYSTEM: OS/MVT.
MEMORY SIZE: HIGH SPEED CORE: 128K BYTES.
 BULK CORE: 0 BYTES.

DESCRIPTION OF THE PROBLEM:

THE PRIMARY PROBLEM WHICH THIS PROGRAMMED ALGORITHM IS
DESIGNED TO SOLVE IS THAT OF OBTAINING A SAMPLE OF STUDENT
JOBS. THESE JOBS MAKE USE OF THE WATFOR FORTRAN COMPILER AND
MAY OR MAY NOT CONTAIN ERRORS.

A SECONDARY PROBLEM IS TO MODIFY THE JOB CONTROL CARDS
FOR EACH OF THE STUDENT JOBS SO THAT THEIR EXECUTION TIME IS
CHARGED TO THE RESEARCH ACCOUNT RATHER THAN TO THE STUDENTS'
OWN ACCOUNTS. MORE ACCOUNTING CHANGES ARE MADE ON THE JOB
CARD FOR THE ENTIRE STUDENT BATCH: THE STUDENT OVERHEAD
ACCOUNT NUMBER IS CHANGED TO THE RESEARCH ACCOUNT NUMBER FOR
THE SAME REASON AS ABOVE; THE NAME OF THE JOB IS ALSO CHANGED
TO FACILITATE ITS RETRIEVAL.

SINCE IT MAY BE NECESSARY TO USE THE STUDENT JOBS EITHER
WITH THE BATCH CONTROL CARDS OR WITHOUT THEM, A THIRD PART OF
THE PROBLEM IS INTRODUCED. THIS PART OF THE PROBLEM IS TO

MCDK0075
MCDK0076
MCDK0077
MCDK0078
MCDK0079
MCDK0080
MCDK0081
MCDK0082
MCDK0083
MCDK0084
MCDK0085
MCDK0086
MCDK0087
MCDK0088
MCDK0089
MCDK0090
MCDK0091
MCDK0092
MCDK0093
MCDK0094
MCDK0095
MCDK0096
MCDK0097
MCDK0098
MCDK0099
MCDK0100
MCDK0101
MCDK0102
MCDK0103
MCDK0104
MCDK0105
MCDK0106
MCDK0107
MCDK0108
MCDK0109
MCDK0110
MCDK0111

CREATE DATA FILES WHICH CONTAIN THE STUDENT JOBS FOR BOTH OF THE ABOVE STATED CASES.

METHOD OF SOLUTION:

AN EXAMINATION OF THE PROBLEM DESCRIPTION REVEALS THAT THE PROBLEM HAS THREE DISTINCT PARTS:

1. OBTAINING A SAMPLE;
2. CHANGING ACCOUNTING INFORMATION; AND
3. CREATING TWO FORMS OF THE NEW DATA BASE.

EACH OF THESE PARTS COULD BE CONSIDERED IN TURN BUT THE ARRANGEMENT OF THE INITIAL DATA BASE ALLOWS THE FIRST TWO STEPS TO BE COMPLETELY COMBINED WITH HALF OF THE THIRD. THE CREATION OF THE SECOND NEW FILE IN THE THIRD STEP REMAINS THE ONLY THING TO BE PERFORMED SEPARATELY. THE CHANGING OF ACCOUNTING INFORMATION AND THE CREATION OF THE FIRST FILE OF STEP THREE ARE INCORPORATED INTO THE PHASE OF THE PROBLEM WHERE THE SAMPLE IS OBTAINED. BY THIS ANALYSIS THE PROBLEM CAN BE REDUCED TO TWO PARTS:

1. OBTAINING A SAMPLE;
2. CREATING A NEW DATA BASE.

THE INITIAL DATA BASE RESIDES ON AN INPUT TAPE WHICH CONTAINS A VERY LARGE SAMPLE OF STUDENT JOBS. THE SAMPLE WHICH IS OBTAINED IS TO BE A REDUCTION OF THIS INITIAL DATA BASE. TO OBTAIN THIS INDICATED REDUCTION AND TO CREATE THE NEW FILES THE FOLLOWING STEPS ARE USED:

1. READ A CARD IMAGE FROM THE INITIAL DATA BASE FILE;
2. IF AN END-OF-FILE IS ENCOUNTERED GO TO STEP 12;
3. IF THE CURRENT CARD IMAGE IS NOT AN OS CONTROL CARD GO TO STEP 5;
4. IF THE CURRENT CARD IMAGE IS AN OS CONTROL CARD THEN CHANGE THE IMAGE BY CHANGING THE ACCOUNT FIELD AND BY CHANGING THE NAME FIELD;
5. IF THE CURRENT CARD IMAGE IS NOT A WATER CONTROL

MCDK0112
MCDK0113
MCDK0114
MCDK0115
MCDK0116
MCDK0117
MCDK0118
MCDK0119
MCDK0120
MCDK0121
MCDK0122
MCDK0123
MCDK0124
MCDK0125
MCDK0126
MCDK0127
MCDK0128
MCDK0129
MCDK0130
MCDK0131
MCDK0132
MCDK0133
MCDK0134
MCDK0135
MCDK0136
MCDK0137
MCDK0138
MCDK0139
MCDK0140
MCDK0141
MCDK0142
MCDK0143
MCDK0144
MCDK0145
MCDK0146
MCDK0147
MCDK0148

CARD GO TO STEP 9;	MCDK0149
6. IF THE CURRENT CARD IMAGE IS A WATFOR JOB CARD THEN	MCDK0150
UPDATE ITS ACCOUNT FIELD;	MCDK0151
7. IF THE CURRENT CARD IMAGE IS A WATFOR /DATA CARD	MCDK0152
THEN INCREMENT THE WATFOR COUNTERS;	MCDK0153
8. IF THE CURRENT CARD IMAGE IS A WATFOR /STOP CARD	MCDK0154
THEN SAVE THIS FACT FOR FUTURE REFERENCE;	MCDK0155
9. IF THE CURRENT CARD IMAGE IS AN OS /STAR CARD	MCDK0156
THEN UPDATE, STORE, AND REINITIALIZE THE BATCH COUNTERS;	MCDK0157
10. WRITE THE UPDATED CARD IMAGE ONTO THE NEW BATCH	MCDK0158
DATA BASE FILE;	MCDK0159
11. REPEAT STEPS 1-10;	MCDK0160
12. CLOSE ALL FILES;	MCDK0161
13. NUMERICALLY SUMMARIZE THE ACTIONS OF STEPS 1-11 AND	MCDK0162
OUTPUT THIS SUMMARY ON THE PRINTER;	MCDK0163
14. GET A CARD IMAGE FROM THE NEW BATCH DATA BASE FILE;	MCDK0164
15. IF AN END-OF-FILE IS ENCOUNTERED GO TO STEP 20;	MCDK0165
16. IF THIS IS THE IMAGE OF AN OS CONTROL CARD GO TO	MCDK0166
STEP 14;	MCDK0167
17. WRITE THIS IMAGE ON THE NEW NON-BATCH DATA BASE	MCDK0168
FILE;	MCDK0169
18. WRITE THIS IMAGE ON THE PRINTER;	MCDK0170
19. GO TO STEP 14;	MCDK0171
20. WRITE A /STOP IMAGE ON THE NEW NON-BATCH DATA	MCDK0172
BASE FILE;	MCDK0173
21. CLOSE ALL FILES AND STOP.	MCDK0174
DESCRIPTION OF INPUT:	MCDK0175
-----	MCDK0176
	MCDK0177
INPUT MAY COME FROM ONE OF THREE SOURCES: AN INPUT	MCDK0178
PARAMETER STRING; THE INITIAL DATA BASE FILE; AND/OR THE	MCDK0179
BATCH DATA BASE FILE.	MCDK0180
VALUES FOR ANY OF SEVERAL VARIABLES MAY BE FOUND IN THE	MCDK0181
INPUT PARAMETER STRING. THE FORM FOR PLACING THESE VALUES IN	MCDK0182
THE STRING IS VVVVVV=NNNNNN; WHERE, VVVVVV IS THE VARIABLE	MCDK0183
NAME AND NNNNNN IS THE VALUE TO BE ASSIGNED TO THAT VARIABLE.	MCDK0184
IF NO VALUE IS GIVEN FOR ANY OF THESE INPUT PARAMETERS THEN	MCDK0185

A DEFAULT VALUE WILL BE ASSUMED FOR THEM. THESE VARIABLES, THEIR DEFAULT VALUES, AND THEIR SIGNIFICANCE ARE GIVEN BELOW.

INPUT FROM THE INITIAL DATA BASE FILE IS IN THE FORM OF CARD IMAGES. THESE CARD IMAGES MAY REPRESENT OS CONTROL CARDS, WATFOR CONTROL CARDS, FORTRAN PROGRAMS, OR DATA FOR FORTRAN PROGRAMS. BOTH OS CONTROL CARDS AND WATFOR CONTROL CARDS REQUIRE SPECIAL PROCESSING, BUT THE OTHER CARDS ARE PASSED TO THE NEXT DATA FILE WITH NO PROCESSING.

INPUT FROM THE BATCH DATA BASE FILE IS ALMOST IDENTICAL TO THAT FROM THE INITIAL DATA BASE FILE. THE PRIMARY DIFFERENCE IS THE WAY IN WHICH THE INPUT CARD IMAGES ARE PROCESSED AFTER THEY HAVE ENTERED THE SYSTEM. ANOTHER DIFFERENCE IS THE FACT THAT THE OS CONTROL CARDS ARE DISCARDED UPON ENTRY FROM THIS FILE.

TABLE OF INPUT PARAMETERS:

PARAMETER	DEFAULT	SIGNIFICANCE
#IN	100	SPECIFIES THE NUMBER OF BATCHES TO BE TRANSFERRED FROM THE INITIAL DATA BASE FILE TO THE BATCH AND NON-BATCH DATA BASE FILES.
#PRINTED	3	SPECIFIES THE NUMBER OF TRANSFERRED BATCHES WHICH WILL BE COPIED ONTO THE PRINTER.
MODE	SEQUENTIAL	SPECIFIES THE WAY THE BATCHES ARE TO BE RETRIEVED FROM THE INITIAL DATA BASE.
SKPBATCH#	0	SPECIFIES WHICH BATCH IN THE

MCDK0186
MCDK0187
MCDK0188
MCDK0189
MCDK0190
MCDK0191
MCDK0192
MCDK0193
MCDK0194
MCDK0195
MCDK0196
MCDK0197
MCDK0198
MCDK0199
MCDK0200
MCDK0201
MCDK0202
MCDK0203
MCDK0204
MCDK0205
MCDK0206
MCDK0207
MCDK0208
MCDK0209
MCDK0210
MCDK0211
MCDK0212
MCDK0213
MCDK0214
MCDK0215
MCDK0216
MCDK0217
MCDK0218
MCDK0219
MCDK0220
MCDK0221
MCDK0222

		BATCH DATA BASE IS NOT TO BE	MCDK0223
		TRANSFERRED TO THE NON-BATCH	MCDK0224
		DATA BASE.	MCDK0225
			MCDK0226
STARTLOC	1	SPECIFIES THE STARTING POSITION	MCDK0227
		FOR THE INITIAL DATA TRANSFER.	MCDK0228
			MCDK0229
			MCDK0230
			MCDK0231
			MCDK0232
			MCDK0233
			MCDK0234
			MCDK0235
			MCDK0236
			MCDK0237
			MCDK0238
			MCDK0239
			MCDK0240
			MCDK0241
			MCDK0242
			MCDK0243
			MCDK0244
			MCDK0245
			MCDK0246
			MCDK0247
			MCDK0248
			MCDK0249
			MCDK0250
			MCDK0251
			MCDK0252
			MCDK0253
			MCDK0254
			MCDK0255
			MCDK0256
			MCDK0257
			MCDK0258
			MCDK0259

DESCRIPTION OF OUTPUT:

THERE ARE FOUR CLASSES OF OUTPUT FROM THIS PROGRAM. CLASSES ONE AND TWO ARE ON THE PRINTER AND CLASSES THREE AND FOUR ARE ON SEQUENTIAL DATA SETS SUCH AS TAPE. CLASS TWO OUTPUT FOLLOWS CLASS ONE OUTPUT ON THE PRINTER. CLASS THREE AND CLASS FOUR OUTPUT GO TO PHYSICALLY SEPERATE DEVICES. THE FOUR OUTPUT CLASSES ARE:

1. COPIES OF PROGRAMS TRANSFERRED FROM THE INITIAL DATA BASE TO THE BATCH DATA BASE;
2. A SUMMARY OF THE ACTION WHICH TOOK PLACE IN THE FIRST PART OF THE PROGRAM;
3. DATA WHICH CREATES THE BATCH DATA BASE;
4. DATA WHICH CREATES THE NON-BATCH DATA BASE.

REFERENCE:

STERLING, THEODOR D. AND POLLACK SEYMOUR V. INTRODUCTION TO STATISTICAL DATA PROCESSING. (PRENTICE-HALL SERIES IN AUTOMATIC COMPUTATION) ENGLEWOOD CLIFFS, NEW JERSEY. PRENTICE-HALL, INC. 1968.

*/

```

/*      PART I      */
X11=' ';
Y=' ';
MODE='SEQUENTIAL';
OPEN FILE(TP5058) RECORD INPUT SEQUENTIAL,
      FILE(TP0502) RECORD OUTPUT SEQUENTIAL,
      FILE(PRINT) RECORD OUTPUT SEQUENTIAL,
      FILE(SYSPPRINT) STREAM PRINT PAGESIZE(55) LINESIZE(121);
WORK=PARM||' ';
GET STRING(WORK) DATA(#IN,#PRINTED,STARTLOC,MODE,SKPBATCH#);
STPOS#=STARTLOC;
IF MODE='SEQUENTIAL' THEN WAYSAMPLE='SEQUENTIALLY';
ELSE DO;
      WAYSAMPLE='RANDOMLY';
      ON ENDFILE(TP5058) GO TO XRDM;
      DO WHILE('1'B);
      READ FILE(TP5058) INTO(XIN);
      IF X2=SLASH_STAR THEN TBATCHES_=TBATCHES_+1;
      END;
      CLOSE FILE(TP5058);
      OPEN FILE(TP5058) RECORD INPUT SEQUENTIAL;
      END;
      ON ENDFILE(TP5058) GO TO CLEANUP;
      DO RECORD_COUNT=1 BY 1;
      READ FILE(TP5058) INTO(XIN);
      IF X2=SLASHES THEN
      DO;
      WORK=SUBSTR(XIN,INDEX(XIN,' '));
      DO WHILE(SUBSTR(WORK,1,1)=' '); WORK=SUBSTR(WORK,2); END;
      IF SUBSTR(WORK,1,3)='JOB' THEN DO;
      SUBSTR(XIN,INDEX(XIN,'')+1,5)=NEWACCOUNT;
      SUBSTR(XIN,INDEX(XIN,'',STUDENT')+1,7)='HEDRICK';

```

XRDM:

MCDK0260
MCDK0261
MCDK0262
MCDK0263
MCDK0264
MCDK0265
MCDK0266
MCDK0267
MCDK0268
MCDK0269
MCDK0270
MCDK0271
MCDK0272
MCDK0273
MCDK0274
MCDK0275
MCDK0276
MCDK0277
MCDK0278
MCDK0279
MCDK0280
MCDK0281
MCDK0282
MCDK0283
MCDK0284
MCDK0285
MCDK0286
MCDK0287
MCDK0288
MCDK0289
MCDK0290
MCDK0291
MCDK0292
MCDK0293
MCDK0294
MCDK0295
MCDK0296

END;	MCDK0297
END;	MCDK0298
IF X1=SLASH THEN DOS: DO;	MCDK0299
IF X4=SLASH_JOB THEN ACCOUNT_FIELD=NEWACCOUNT;	MCDK0300
IF X5=SLASH_DATA THEN DO; #WAT=#WAT+1;	MCDK0301
#BWAT=#RWAT+1;	MCDK0302
IF #BATC<#IN THEN #WATFT=#WATFT+1; END;	MCDK0303
IF X5=SLASH_STOP THEN PRECEDING_CARD=SLASH_STOP;	MCDK0304
IF X2=SLASH_STOP THEN DO;	MCDK0305
IF #RWAT=0 THEN	MCDK0306
#MNWAT=MIN(#MNWAT, #BWAT); #MXWAT=MAX(#MXWAT, #BWAT);	MCDK0307
IF #BWAT=0 THEN	MCDK0308
#BWATF=#BWAT;	MCDK0309
#BWAT=0;	MCDK0310
IF (PRECEDING_CARD=SLASH_STOP)	MCDK0311
& (#BATC<#IN) THEN DO; WORK1,	MCDK0312
Z=SLASH_STOP;	MCDK0313
IF #BATC<#PRINTED THEN	MCDK0314
WRITE FILE(PRINT) FROM(Y);	MCDK0315
WRITE FILE(TP0502) FROM(WORK1);	MCDK0316
END;	MCDK0317
IF #BATC<#IN THEN #BATC=#BATC+1;	MCDK0318
#BATCH=#BATCH+1;	MCDK0319
PRECEDING_CARD='';	MCDK0320
IF #BATC>#IN THEN GO TO CLEANUP;	MCDK0321
END;	MCDK0322
END DOS;	MCDK0323
XI2=XIN;	MCDK0324
IF #BATC<#IN THEN WRITE FILE(TP0502) FROM(XIN);	MCDK0325
IF #BATCH <#PRINTED THEN WRITE FILE(PRINT) FROM(XI);	MCDK0326
END;	MCDK0327
CLEANUP:	MCDK0328
AVEWAT=#WATFT/#BATC;	MCDK0329
CLOSE FILE(TP0508), FILE(TP0502), FILE(PRINT);	MCDK0330
PUT FILE(SYSPRINT) EDIT(OUTPUT_RECORD)	MCDK0331
(SKIP,COL(10),A,COL(66),F(6,0));	MCDK0332
PUT FILE(SYSPRINT) EDIT(OUTPUT_RECORD_A)	MCDK0333

(SKIP, COL(10), A, COL(60+(MODE='RANDOM')*4), A,	MCDK0334
SKIP, COL(10), A, COL(66), A, SKIP, COL(10), A, COL(66), A1;	MCDK0335
CLOSE FILE(SYSPRINT);	MCDK0336
/* PART II */	MCDK0337
OPEN FILE(TP0502) RECORD INPUT SEQUENTIAL,	MCDK0338
FILE(DSK001) RECORD OUTPUT SEQUENTIAL;	MCDK0339
ON ENDFILE(TP0502) GO TO NEXT_STEP;	MCDK0340
BATCHT=0;	MCDK0341
DO RECORD_COUNT=1 BY 1;	MCDK0342
READ FILE(TP0502) INTO (XIN);	MCDK0343
IF BATCHT=(SKPBATCH-1) THEN DO;	MCDK0344
IF (X2 /= SLASHES) & (X5 /= SLASH_STOP) & (X2 /= SLASH_STAR) THEN	MCDK0345
WRITE FILE(DSK001) FROM(XIN);	MCDK0346
END;	MCDK0347
IF X2=SLASH_STAR THEN BATCHT=BATCHT+1;	MCDK0348
END;	MCDK0349
NEXT_STEP:	MCDK0350
XIN=SLASH_STOP; WRITE FILE(DSK001) FROM(XIN);	MCDK0351
XIN=SLASH_STAR; WRITE FILE(DSK001) FROM(XIN);	MCDK0352
CLOSE FILE(TP0502), FILE(DSK001);	MCDK0353
/* DEBUG SECTION.	MCDK0354
OPEN FILE(DSK001) INPUT RECORD SEQUENTIAL;	MCDK0355
ON ENDFILE(DSK001) GO TO THE_END;	MCDK0356
DO WHILE('1'B):	MCDK0357
READ FILE(DSK001) INTO(XIN); Z=XIN;	MCDK0358
WRITE FILE(PRINT) FROM (Y);	MCDK0359
END;	MCDK0360
THE_END:	MCDK0361
CLOSE FILE(DSK001);	MCDK0362
END OF DEBUG SECTION. */	MCDK0363
END GETSAMP;	MCDK0364
	MCDK0365

```
(SURRG,STRG,SIZE):
ANLYZ1: PROC(PARM) OPTIONS(MAIN);
/*
```

DESCRIPTION OF THE PROBLEM:

THIS PROGRAM IS DESIGNED TO ANALYZE THE ERRORS WHICH HAVE OCCURRED IN A RANDOM SAMPLE OF FORTRAN PROGRAMS. THESE FORTRAN PROGRAMS HAVE BEEN EXECUTED UNDER THE WATFOR STUDENT COMPILER SYSTEM AS IT IS USED AT THE IOWA STATE UNIVERSITY INSTALLATION.

THE VARIOUS TYPES OF ERRORS ARE ANALYZED TO DETERMINE THEIR EXISTENCE, THEIR TYPE, AND THE PROBABILITY OF OCCURRENCE OF EACH TYPE OF ERROR. THE PROGRAM ALSO SUMMARIZES THE ERRORS WHICH IT ANALYZES; PLOTS (ON THE PRINTER) A HISTOGRAM OF THE FREQUENCY OF OCCURRENCE OF THE MOST COMMON ERRORS; AND AIDS IN THE DETERMINATION OF ERROR DEPENDENCY.

METHOD OF SOLUTION:

THE SOLUTION WHICH WAS DEVISED FOR THIS PROBLEM HAS SEVERAL DISTINCT PHASES:

1. INITIALIZATION,
2. DETERMINATION OF ERROR EXISTENCE,
3. ERROR CLASSIFICATION,
4. SUMMARIZATION,
5. GRAPHING.

WITH THE EXCEPTION OF THE INITIALIZATION PHASE, EACH OF THE ABOVE PHASES IS SUBSEQUENTLY DIVIDED INTO ADDITIONAL PARTS. THE DETERMINATION OF ERROR EXISTENCE REQUIRES AN EXAMINATION OF THE ERROR MESSAGE OUTPUT LINE; ERROR CLASSIFICATION INVOLVES SEARCHING FOR ERROR TYPES AND TABULATING EACH OCCURRENCE OF EACH DIFFERENT TYPE; SUMMARIZATION CONSISTS OF

```
MCDK0001
MCDK0002
MCDK0003
MCDK0004
MCDK0005
MCDK0006
MCDK0007
MCDK0008
MCDK0009
MCDK0010
MCDK0011
MCDK0012
MCDK0013
MCDK0014
MCDK0015
MCDK0016
MCDK0017
MCDK0018
MCDK0019
MCDK0020
MCDK0021
MCDK0022
MCDK0023
MCDK0024
MCDK0025
MCDK0026
MCDK0027
MCDK0028
MCDK0029
MCDK0030
MCDK0031
MCDK0032
MCDK0033
MCDK0034
MCDK0035
MCDK0036
MCDK0037
```

PROCESSING THE DATA WHICH WAS GATHERED IN THE PREVIOUS PHASES
AND OUTPUTTING THIS PROCESSED DATA ON THE PRINTER; THE
GRAPHING PHASE PERFORMS A QUADRATIC SORT ON THE PREVIOUSLY
OBTAINED TABULATION AND PLOTS A HISTOGRAM OF THIS DATA.

DESCRIPTION OF INPUT:

INPUT FOR THIS PROGRAM COMES FROM ONE OF TWO SOURCES: AN
INPUT PARAMETER STRING AND/OR A SEQUENTIAL DATA SET. THE
PARAMETERS ARE NOT USUALLY NECESSARY WHEN THE PROGRAM IS RUN
IN THE PRODUCTION MODE. FOR THE USER'S BENEFIT THE EXACT
USAGES OF THE INPUT PARAMETERS ARE GIVEN BELOW. THE SEQUENTIAL
DATA SET CONTAINS THE COMPLETE OUTPUT FOR ONE ENTIRE SAMPLE
OF WATER JOBS. THE INFORMATION ON THIS DATA SET CONSISTS OF
LINE IMAGES WHICH ARE PROCESSED ONE AT A TIME WHERE POSSIBLE.
EACH ERROR WHICH IS ENCOUNTERED REQUIRES THAT THIS PROGRAM
TAKE SPECIAL ACTION PRIOR TO THE READING OF THE NEXT LINE
IMAGE.

TABLE OF INPUT PARAMETERS:

PARAMETER (DEFAULT UNDERLINED) SIGNIFICANCE

GRAPH, NOGRAPH

WHEN GRAPH IS SPECIFIED A
HISTOGRAM IS PLOTTED ON THE
PRINTER. NOGRAPH FORCES THE
DELETION OF THIS OUTPUT.

PRINT_ERR, NOPRINT_ERR

IF PRINT_ERR IS SPECIFIED THEN
A LINE OF PRINT IS GIVEN AS
EACH ERROR IS ENCOUNTERED.

MCDK0038
MCDK0039
MCDK0040
MCDK0041
MCDK0042
MCDK0043
MCDK0044
MCDK0045
MCDK0046
MCDK0047
ANMCDK0048
MCDK0049
MCDK0050
MCDK0051
MCDK0052
MCDK0053
MCDK0054
MCDK0055
MCDK0056
MCDK0057
MCDK0058
MCDK0059
MCDK0060
MCDK0061
MCDK0062
MCDK0063
MCDK0064
MCDK0065
MCDK0066
MCDK0067
MCDK0068
MCDK0069
MCDK0070
MCDK0071
MCDK0072
MCDK0073
MCDK0074

	OTHERWISE, THIS OUTPUT IS SUPPRESSED.	MCDK0075 MCDK0076 MCDK0077
PRINT_PROG, NO PRINT_PROG -----	WHEN PRINT_PROG IS SPECIFIED THE WATFOR PROGRAMS WHICH ARE INPUT ARE COPIED ON THE PRINTER.	MCDK0078 MCDK0079 MCDK0080 MCDK0081
DEBUG, NO DEBUG -----	WHEN DEBUG IS SPECIFIED A TRACE OF THE EXECUTION OF THE PROGRAM IS GIVEN AS AN AID TO DEBUGGING.	MCDK0082 MCDK0083 MCDK0084 MCDK0085 MCDK0086 MCDK0087 MCDK0088 MCDK0089 MCDK0090
DESCRIPTION OF OUTPUT: -----		MCDK0091 MCDK0092 MCDK0093 MCDK0094 MCDK0095 MCDK0096 MCDK0097 MCDK0098 MCDK0099 MCDK0100 MCDK0101 MCDK0102 MCDK0103 MCDK0104 MCDK0105 MCDK0106 MCDK0107 MCDK0108 MCDK0109
THERE ARE FOUR DIFFERENT CLASSIFICATIONS OF OUTPUT FROM THIS PROGRAM. THE OUTPUT CLASSES ARE:		
1. DEBUG OUTPUT,		
2. ERROR OCCURRENCE OUTPUT,		
3. SUMMARY OUTPUT,		
4. GRAPHIC OUTPUT.		
ALL OF THIS OUTPUT EXCEPT CLASS 1 OUTPUT IS PRINTED ON THE PROGRAM PRINT FILE. CLASS 1 OUTPUT IS PRINTED ON THE SYSPRINX PRINT FILE. SPECIAL INFORMATION MUST BE PROVIDED IN THE INPUT PARAMETER STRING TO OBTAIN CLASS 1 OR CLASS 2 OUTPUT. CLASS 4 OUTPUT MAY BE SUPPRESSED VIA AN INPUT PARAMETER. CLASS 3 OUTPUT IS ALWAYS GIVEN AND CONTAINS THE STATISTICS NECESSARY FOR THE DEVELOPMENT OF FUTURE ALGORITHMS.		
REFERENCES: -----		
BLATT, JOHN M. INTRODUCTION TO FORTRAN IN PROGRAMMING: USING THE WATFOR COMPILER. PACIFIC PALISADES, CALIFORNIA. GOODYEAR		MCDK0110 MCDK0111

PUBLISHING COMPANY. C1968.

MCDK0112

MCDK0113

FLORES, IVAN. COMPUTER SORTING. ENGLEWOOD CLIFFS, NEW JERSEY. MCDK0114

PRENTICE-HALL, INC. C1969. MCDK0115

MCDK0116

HOPCROFT, J. E. AND ULLMAN, J. D. ERROR CORRECTION FOR FORMAL MCDK0117

LANGUAGES. PRINCETON, NEW JERSEY. PRINCETON UNIVERSITY MCDK0118

DEPARTMENT OF ELECTRICAL ENGINEERING DIGITAL SYSTEMS LABORATORY MCDK0119

TECHNICAL REPORT NUMBER 52. AMES LABORATORY TECHNICAL REPORT MCDK0120

NUMBER ALCTN-78. C1966. MCDK0121

MCDK0122

STERLING, THEODOR D. AND POLLACK, SEYMOUR V. INTRODUCTION TO MCDK0123

STATISTICAL DATA PROCESSING. (PRENTICE-HALL SERIES IN AUTOMATIC MCDK0124

COMPUTATION) ENGLEWOOD CLIFFS, NEW JERSEY. PRENTICE-HALL, MCDK0125

INC. C1968. MCDK0126

MCDK0127

*/MCDK0128

MCDK0129

DCL 1 GRAPHX STATIC EXT, MCDK0130

2 HLABEL CHAR(96), MCDK0131

2 DVLABEL CHAR(31) INIT(' 7 7 6 6 5 5 4 4 3 3 2 2 1 1 0 ') MCDK0132

, 2 DVLABELA CHAR(31) INIT MCDK0133

(' 5 0 5 0 5 0 5 0 5 0 5 0 5 0 5 '), MCDK0134

2 DVLABLE CHAP(31) INIT MCDK0135

(' NUMBER OF OCCURENCES '), MCDK0136

2 VAXIS CHAR(1) INIT(' '), MCDK0137

2 HAXIS CHAP(96) INIT((96) ' '), MCDK0138

2 HLABEL CHAR(6) INIT('ERRORS'), MCDK0139

2 GRAPH(31,96) CHAR(1), MCDK0140

NER(24) CHAR(4) DEF HLABEL POS(1), MCDK0141

VLABEL(31) CHAR(1) DEF DVLABEL POS(1), MCDK0142

VLABELA(31) CHAR(1) DEF DVLABELA POS(1), MCDK0143

VLABLE(31) CHAR(1) DEF DVLABLE POS(1), MCDK0144

P POINTER, #QNER(0:24) FIXED BIN(31,0), MCDK0145

(I,II,J,K,#NER(24) BASED (P)) FIXED BIN(31,0); MCDK0146

DCL ERRORS___ CHAR(33) STATIC INIT MCDK0147

('*EXTENSION***WARNING*****ERROR***'), MCDK0148

```

F_WARNING__ CHAR(11) DEF ERRORS___ POS(12),
FEXTENSION_ CHAR(11) DEF ERRORS___ POS(1),
F__ERROR___ CHAR(11) DEF ERRORS___ POS(23),
XLINE CHAR(133) STATIC,
LINE CHAR(132) DEF XLINE POS(2),
CARRIAGE_CONTROL CHAR(1) DEF XLINE POS(1),
X4 CHAR(4) DEF XLINE POS(2),
X5 CHAR(5) DEF XLINE POS(2),
CARDIN CHAR(80) DEF XLINE POS(11),
XC4 CHAR(4) DEF XLINE POS(11),
XC5 CHAR(5) DEF XLINE POS(11),
EFLAG1 CHAR(11) DEF XLINE POS(2),
EFLAG2 CHAR(4) DEF XLINE POS(18),
EFLAG3 CHAR(5) DEF XLINE POS(18),
SLASH_JOB CHAR(4) STATIC INIT('/JOB'),
SLASH_DATA CHAR(5) STATIC INIT('/DATA'),
COL6 CHAR(1) DEF XLINE POS(16),
SLASH_STOP CHAR(5) STATIC INIT('/STOP'),
INPGFLG FIXED BIN(31,0) STATIC INIT(0),
ERFLG FIXED BIN(31,0) STATIC INIT(0),
CT CHAR(13) STATIC INIT('COMPILE TIME='),
CTX CHAR(13) DEF XLINE POS(2),
SXT CHAR(19) DEF XLINE POS(25),
SXXT CHAR(19) STATIC INIT
('SEC,EXECUTION TIME='),
SOC CHAR(16) DEF XLINE POS(54),
SOXT CHAR(16) STATIC INIT
('SEC,OBJECT CODE='),
BAA CHAR(17) DEF XLINE POS(79),
RXAA CHAR(17) STATIC INIT
('BYTES,ARRAY AREA='),
UNUSED CHAR(13) DEF XLINE POS(105),
XUNUSED CHAR(13) STATIC INIT
('BYTES,UNUSED=');
1 OUTPUT_RECORD_1 UNALIGNED STATIC,
2 ST_1 CHAR(35) INIT
('TOTAL NUMBER OF COMPILE-TIME ERRORS'),

```

```

MCDK0149
MCDK0150
MCDK0151
MCDK0152
MCDK0153
MCDK0154
MCDK0155
MCDK0156
MCDK0157
MCDK0158
MCDK0159
MCDK0160
MCDK0161
MCDK0162
MCDK0163
MCDK0164
MCDK0165
MCDK0166
MCDK0167
MCDK0168
MCDK0169
MCDK0170
MCDK0171
MCDK0172
MCDK0173
MCDK0174
MCDK0175
MCDK0176
MCDK0177
MCDK0178
MCDK0179
MCDK0180
MCDK0181
MCDK0182
MCDK0183
MCDK0184
MCDK0185

```

```

2 #TOT_CERRS FIXED BIN(31,0) INIT(0), MCDK0186
2 ST_2 CHAR(55) INIT MCDK0187
('AVERAGE(MEAN) NUMBER OF COMPILE-TIME ERRORS PER PROGRAM'), MCDK0188
, 2 #AVE_CERR_PROGRAM FLOAT DEC(15), MCDK0189
2 ST_3 CHAR(49) INIT MCDK0190
('MAXIMUM NUMBER OF COMPILE-TIME ERRORS PER PROGRAM'), MCDK0191
2 #MAX_CERR_PROGRAM FIXED BIN(31,0) INIT(0), MCDK0192
2 ST_4 CHAR(59) INIT MCDK0193
('MINIMUM NUMBER OF COMPILE-TIME ERRORS IN ERRONEOUS PROGRAM'), MCDK0194
MS' MCDK0195
MCDK0196
), 2 #MIN_CERR_PROGRAM FIXED BIN(31,0) INIT(7000), MCDK0197
2 ST_5 CHAR(45) INIT MCDK0198
('TOTAL NUMBER OF NON-TERMINAL EXECUTION ERRORS'), MCDK0199
2 #TOT_NTERRS FIXED BIN(31,0) INIT(0); MCDK0200
DCL 1 OUTPUT_RECORD_1A STATIC UNALIGNED, MCDK0201
2 ST_6 CHAR(70) INIT MCDK0202
('AVERAGE(MEAN) NUMBER OF NON-TERMINAL EXECUTION-TIME ERRORS'), MCDK0203
S PER PROGRAM' MCDK0204
MCDK0205
), MCDK0206
2 #AVE_NTE_PROG FLOAT DEC(15), MCDK0207
2 ST_7 CHAR(64) INIT MCDK0208
('MAXIMUM NUMBER OF NON-TERMINAL EXECUTION TIME ERRORS PER MCDK0209
PROGRAM' MCDK0210
MCDK0211
), MCDK0212
2 #MAX_NTE_PROG FIXED BIN(31,0) INIT(0), MCDK0213
2 ST_8 CHAR(64) INIT MCDK0214
('MINIMUM NUMBER OF NON-TERMINAL EXECUTION TIME ERRORS PER MCDK0215
PROGRAM' MCDK0216
MCDK0217
), MCDK0218
2 #MIN_NTE_PROG FIXED BIN(31,0) INIT(7000), MCDK0219
2 ST_9 CHAR(46) INIT MCDK0220
('TOTAL NUMBER OF TERMINAL EXECUTION-TIME ERRORS'), MCDK0221
2 #TOT_TERRS FIXED BIN(31,0) INIT(0), MCDK0222

```

```

2 ST_10 CHAR(66) INIT MCDK0223
('AVERAGE(MEAN) NUMBER OF TERMINAL EXECUTION-TIME ERRORS PER PROGRAM' MCDK0224
), MCDK0225
2 #AVE_TERR_PROG FLOAT DEC(6) INIT(0), MCDK0226
2 ST_11 CHAR(61) INIT MCDK0227
('MAXIMUM NUMBER OF TERMINAL EXECUTION-TIME ERRORS PER PROGRAM' MCDK0228
RAM' MCDK0229
2 #MAX_TERR_PROG FIXED BIN(31,0) INIT(0); MCDK0230
DCL 1 OUTPUT_RECORD_1B UNALIGNED STATIC, MCDK0231
2 ST_12 CHAR(71) INIT MCDK0232
('MINIMUM NUMBER OF TERMINAL EXECUTION-TIME ERRORS IN ERROR PROGRAMS' MCDK0233
), MCDK0234
2 #MIN_TERR_PROG FIXED BIN(31,0) INIT(7000), MCDK0235
2 ST_13 CHAR(22) INIT MCDK0236
('TOTAL NUMBER OF ERRORS'), MCDK0237
2 #TOTAL_ERRS FIXED BIN(31,0) INIT(0), MCDK0238
2 ST_14 CHAR(56) INIT MCDK0239
('MAXIMUM NUMBER OF ERRORS IN ANY GIVEN ERRONEOUS PROGRAM' MCDK0240
), MCDK0241
2 #MAXERRS FIXED BIN (31,0) INIT(0), MCDK0242
2 ST_15 CHAR(56) INIT MCDK0243
('MINIMUM NUMBER OF ERRORS IN ANY GIVEN ERRONEOUS PROGRAM' MCDK0244
), MCDK0245
2 #MINERRS FIXED BIN(31,0) INIT(7000), MCDK0246
2 ST_16 CHAR(31) INIT MCDK0247
('TOTAL NUMBER OF WATFOR PROGRAMS'), MCDK0248
2 #TOT_PROGS FLOAT DEC(15) INIT(0); MCDK0249
DCL 1 OUTPUT_RECORD_1C STATIC UNALIGNED, MCDK0250
2 ST_17 CHAR(42) INIT MCDK0251
('AVERAGE(MEAN) NUMBER OF ERRORS PER PROGRAM'), MCDK0252
2 #AVE_ERRS_PROG FLOAT DEC(15), MCDK0253
2 ST_18 CHAR(34) INIT MCDK0254
('TOTAL NUMBER OF WATFOR STATEMENTS'), MCDK0255
2 #TOT_STATEMENTS FLOAT DEC(15) INIT(0), MCDK0256
MCDK0257
MCDK0258
MCDK0259

```


	2 ST_19 CHAR(44) INIT	MCDK0260
	('AVERAGE(MEAN) NUMBER OF ERRORS PER STATEMENT'),	MCDK0261
	2 #AVE_ERRS_STATEMENT FLOAT DEC(15),	MCDK0262
	2 ST_20 CHAR(44) INIT	MCDK0263
	('TOTAL NUMBER OF CARDS WITHIN WATERFOR PROGRAMS'),	MCDK0264
	2 #TOT_CARDS FLOAT DEC(15) INIT(0),	MCDK0265
	2 ST_21 CHAR(39) INIT	MCDK0266
	('AVERAGE(MEAN) NUMBER OF ERRORS PER CARD'),	MCDK0267
	2 #AVE_ERRS_CARD FLOAT DEC(15);	MCDK0268
DCL	1 OUTPUT_RECORD_1D STATIC UNALIGNED,	MCDK0269
	2 ST_22 CHAR(43) INIT	MCDK0270
	('NUMBER OF PROGRAMS WITH MORE THAN ONE ERROR'),	MCDK0271
	2 #PROG_GT1ERR FIXED BIN(31,0) INIT(0),	MCDK0272
	2 ST_23 CHAR(51) INIT	MCDK0273
	('NUMBER OF PROGRAMS WHICH CONTAIN AT LEAST ONE ERROR'),	MCDK0274
	2 #ERRONEOUS_PROG FIXED BIN(31,0) INIT(0),	MCDK0275
	2 ST_24 CHAR(47) INIT	MCDK0276
	('AVERAGE NUMBER OF ERRORS PER ERRONEOUS PROGRAM'),	MCDK0277
	2 #AVE_ERRON_PROG FLOAT DEC(15),	MCDK0278
	2 ST_25 CHAR(79) INIT	MCDK0279
	('PROBABILITY THAT A PROGRAM CONTAINS AT LEAST ONE ERROR (P	MCDK0280
	R(P/N/(E)-P/O/(PHI)))'	MCDK0281
	1,	MCDK0282
	2 PROB_OF_ERROR1 FLOAT DEC(15),	MCDK0283
	2 ST_26 CHAR(105) INIT	MCDK0284
	('PROBABILITY THAT A PROGRAM CONTAINS MORE THAN ONE ERROR W	MCDK0285
	HEN IT IS ERRONEOUS (PR(N>1 P/N/(E)-P/O/(PHI)))'	MCDK0286
	1,	MCDK0287
	2 PROB_OF_ERROR2 FLOAT DEC(15);	MCDK0288
DCL	1 OUTPUT_RECORD_2 UNALIGNED STATIC,	MCDK0289
	2 ST_27 CHAR(105) INIT	MCDK0290
	('E/I/='), 2 CURRENT_ERR1 CHAR(4),	MCDK0291
	2 P_C_E1 CHAR(9) INIT('PR(E/I/)='),	MCDK0292
	2 PROB3 FLOAT DEC(15),	MCDK0293
	2 ST_28 CHAR(28) INIT	MCDK0294
		MCDK0295
		MCDK0296

```

                ('PR(E/I/|P/N/(E)-=P/O/(PHI))='),
2  PROB4 FLOAT DEC(15);
DCL 1 OUTPUT_RECORD_3 UNALIGNED STATIC,
2  ST_29 CHAR(06) INIT
    ('E/J/='),
2  CURRENT_ERR2 CHAR(4),
2  ST_30 CHAR(09) INIT
    ('PR(E/J/)='),
2  PROB4 FLOAT DEC(15),
2  ST_31 CHAR(14) INIT
    ('PR(E/I/&E/J/)='),
2  PROB5 FLOAT DEC(15),
2  ST_32 CHAR(14) INIT
    ('PR(E/I/)|E/J/|'),
2  PROB6 FLOAT DEC(15),
    (CURRENT_TOTAL_ERRORS,ETRS,CTRS,NTRS) FIXED BIN(31,0) INIT(0),
    #PE STATIC INIT(0209) FIXED BIN(31,0);
DCL ALL_ERROR_STRING CHAR(364) INIT
    ('AS-2AS-3AS-4AS-5BD-6BD-1CC-2CC-3CC-4CC-5CC-6CC-7CC-8CMCDK0315
C-9CM-0CM-1CM-2CM-3CM-4CN-5CN-6CN-7CN-8CN-9CP-0CP-1CP-2CP-3MCDK0316
CP-4DA-0DA-1DA-2DA-3DA-4DA-5DA-6DA-7DA-8DA-9DA-ADA-BDM-0DM-1DM-2DM-3DM-MCDK0317
4DD-0DD-1DD-2DD-3DD-4DD-5DD-6DD-7DD-8DD-9EN-0EN-1EN-2EN-3EQ-0EQ-1EQ-2EQMCDK0318
-3EQ-4EQ-5EQ-6EQ-7EQ-8EQ-9EQ-AEV-0EV-1EV-2EV-3EV-4EX-0EX-1EX-2EX-3EX-4EMCDK0319
X-5EX-6EX-7EX-8EX-9'
MCDK0320
MCDK0321
MCDK0322
MCDK0323
) STATIC,
    ALL_ERROR_STRING2 CHAR(380) INIT
    ('EY-0EY-1EY-2EY-3EY-4EY-5EY-6FM-0FM-1FM-2FM-3FM-4FM-5FM-6FM-7FMCDK0324
T-0FT-1FT-2FT-3FT-4FT-5FT-6FT-7FT-8FT-9FT-AFT-BFT-CFT-DFT-EFT-FFN-0FN-3MCDK0325
FN-4FN-5FN-6FN-7FN-8GO-0GO-1GO-2GO-3GO-4HO-0HO-1HO-2HO-3HO-4IF-0IF-1IF-MCDK0326
2IF-3IF-4IM-0IM-1IM-2IM-3IM-4IM-5IM-6IM-7IM-8IM-9IO-0IO-2IO-3IO-4IO-6IOMCDK0327
-8IO-9IO-AIO-BIO-CIO-DIO-EIO-FIO-GIO-HIO-JIO-KJB-0JB-1JB-2JB-3KO-0KO-1KMCDK0328
O-2KO-3KO-4KO-5KO-6KO-7KO-8KO-9LG-2'
MCDK0329
MCDK0330
MCDK0331
MCDK0332
) STATIC,
    ALL_ERROR_STRING3 CHAR(352) INIT
    ('LI-0LI-1LI-2LI-3LI-4LI-5LI-6LI-7LI-8LK-9LI-ALI-BLI-CLI-DLI-ELMCDK0333

```

```

I-FLI-GLI-HMD-2MD-3MD-4MD-6MD-0MD-1MD-2MD-3MD-4PC-0PC-1PS-0PS-1RE-0RE-1MCDK0334
PE-2RE-3RE-4SF-1SF-2SF-3SR-0SR-1SR-2SR-3SR-4SR-5SR-6SR-7SR-8SR-9SR-ASS-MCDK0335
OSS-1SS-2ST-0ST-1ST-2ST-3ST-4ST-5ST-6ST-7ST-8ST-9ST-ASV-0SV-1SV-2SV-3SVMCDK0336
-4SV-5SX-0SX-1SX-2SX-3SX-4SX-5SX-CSX-DUN-0UN-1UN-2UN-3UN-4UN-5UN-6UN-7UMCDK0337
N-8UN-9* MCDK0338
) STATIC, MCDK0339
    ALL_ERROR_STRING4 CHAR(108) INIT MCDK0340
    ('UV-0UV-1UV-2UV-3UV-4UV-5UV-6VA-0VA-1VA-2VA-3VA-4VA-5VA-6VA-7V MCDK0342
A-8VA-9VA-AVA-BVA-CVA-DVA-EXT-0XT-1XT-2' MCDK0343
) STATIC; MCDK0344
DCL EXECUTION_TIME_ERROR_STRING CHAR(248) INIT MCDK0345
    ('KO-0KO-1KO-2KO-3KO-4KO-5KO-6KO-7KO-8SR-2SR-4SR-5SR-6SR-7SS-0SMCDK0347
S-1EX-0EX-2EX-3EX-6EX-7EX-8EX-9FT-0FT-1FT-2FT-3FT-4FT-5FT-6FT-7FT-8FT-9MCDK0348
FT-AFT-BFT-CFT-DFT-EFT-FFM-OFM-5FM-6GO-2GO-4UN-0UN-1UN-2UN-3UN-4UN-5UN-MCDK0349
6UN-7UN-8UN-9UV-0UV-1UV-2UV-3UV-4UV-5UV-6DO-5' MCDK0350
), MCDK0351
    SYSPRINX FILE INT, MCDK0352
    (DSK001,PRINT ) FILE EXT; MCDK0353
DCL COMPILE_TIME_ERROR_STRING STATIC CHAR(1204), MCDK0354
    CTES(299) CHAR(4) DEF COMPILE_TIME_ERROR_STRING POS(1), MCDK0355
    XES(62) CHAR(4) DEF EXECUTION_TIME_ERROR_STRING POS(1), MCDK0356
    NON_TERMINAL_EXEC_ERRORS /* ALSO GARBAGE */ MCDK0357
    STATIC CHAR(4): MCDK0358
DCL (ES1(91) DEF ALL_ERROR_STRING POS(1), MCDK0359
    ES2(95) DEF ALL_ERROR_STRING2 POS(1), MCDK0360
    ES3(88) DEF ALL_ERROR_STRING3 POS(1), MCDK0361
    ES4(27) DEF ALL_ERROR_STRING4 POS(1)) CHAR(4), MCDK0362
    *ES(299) FIXED BIN(31,0) STATIC, MCDK0363
    ESX CHAR(1204) STATIC, MCDK0364
    ES(299) CHAR(4) DEF ESX POS(1), MCDK0365
    (ES1A(91) DEF ESX POS(1), MCDK0366
    ES2A(95) DEF ESX POS(365), MCDK0367
    ES3A(88) DEF ESX POS(745), MCDK0368
    ES4A(27) DEF ESX POS(1097)) CHAR(4), MCDK0369
    MCDK0370

```

	WORK CHAR(40) STATIC,	MCDK0371
	(PRINT_ERR,PRINT_PROG,DEBUG,NOGRAPH) BIT(1),	MCDK0372
	LKP FIXED BIN(31,0),	MCDK0373
	III FIXED BIN(31,0),	MCDK0374
	PARM CHAR(40) VAR;	MCDK0375
DCL	1 SUBLISTS(18,18) STATIC UNALIGNED,	MCDK0376
	2 A FIXED BIN(31,0),	MCDK0377
	2 B CHAR(4),	MCDK0378
	1 TMP STATIC UNALIGNED,	MCDK0379
	2 A FIXED BIN(31,0),	MCDK0380
	2 B CHAR(4),	MCDK0381
	1 LEAST_TABLE(18) STATIC UNALIGNED,	MCDK0382
	2 A FIXED BIN(31,0),	MCDK0383
	2 B CHAR(4),	MCDK0384
	XOUTT CHAR(80),	MCDK0385
	#XNER(24) FIXED BIN(31,0) DEF #ES,	MCDK0386
	XNER CHAR(4) DEF ESX POS(1),	MCDK0387
	RECORD_COUNT_DSK001 FIXED BIN(31,0),	MCDK0388
	ULINE CHAR(091) DEF XLINE POS(2),	MCDK0389
	(N,NN INIT(297)) STATIC FIXED BIN(31,0);	MCDK0390
/*		MCDK0391
	INITIALIZATION PHASE.	MCDK0392
*/		MCDK0393
		MCDK0394
	GRAPH=' ';	MCDK0395
	OPEN FILE(DSK001) RECORD INPUT SEQUENTIAL,	MCDK0396
	FILE(PRINT) STREAM PRINT OUTPUT	MCDK0397
	PAGESIZE(55) LINESIZE(120),	MCDK0398
	FILE(SYSPRINX) STREAM PRINT OUTPUT PAGESIZE(55)	MCDK0399
	TITLE('SYSPRIN')	MCDK0400
	LINESIZE(120);	MCDK0401
	#ES=0;	MCDK0402
	ON ENDPAGE(SYSPRINX) PUT FILE(SYSPRINX) PAGE EDIT((20) '*',	MCDK0403
	'DEBUG OUTPUT',(20) '*') (COL(25),3 A);	MCDK0404
	ON ENDPAGE(PRINT) BEGIN;	MCDK0405
	PUT FILE(PRINT)EDIT((20) '*', 'ERROR CLASSIFICATION',	MCDK0406
	(20) '*') (COL(25),3 A) PAGE;	MCDK0407

PUT FILE(PRINT) LINE(3);	MCDK0408
END;	MCDK0409
ES1A=ES1; ES2A=ES2; ES3A=ES3; ES4A=ES4;	MCDK0410
CTES=ES;	MCDK0411
DO I=1 TO DIM(XES,1); DO J=1 TO #PE; IF CTES(J)=XES(I) THEN	MCDK0412
CTES(J)=' ' ; END; END;	MCDK0413
WORK=PARM;	MCDK0414
NOGRAPH=INDEX(WORK,'NOGRAPH') >0;	MCDK0415
PRINT_ERR=(INDEX(WORK,'PRINT_ERR')=0)&(INDEX(WORK,	MCDK0416
'NOPRINT_ERR')=0);	MCDK0417
PRINT_PROG=(INDEX(WORK,'PRINT_PROG')=0)&	MCDK0418
(INDEX(WORK,'NOPRINT_PROG')=0);	MCDK0419
DEBUG=(INDEX(WORK,'DEBUG')=0)&(INDEX(WORK,'NODEBUG')=0);	MCDK0420
IF DEBUG THEN SIGNAL ENDPAGE(SYSPRNX);	MCDK0421
IF PRINT_ERR THEN SIGNAL ENDPAGE(PRINT);	MCDK0422
IF DEBUG THEN PUT FILE(SYSPRNX) EDIT(ES	MCDK0423
(SKIP,COL(22),10 (A,X(2))));	MCDK0424
/* END OF INITIALIZATION PHASE.	MCDK0425
	MCDK0426
	MCDK0427
	MCDK0428
SUMMARIZATION PHASE. */	MCDK0429
	MCDK0430
ON ENDFILE(DSK001) BEGIN;	MCDK0431
ON ERROR SNAP GO TO CATCH;	MCDK0432
ON ENDPAGE(PRINT)	MCDK0433
PUT FILE(PRINT) EDIT((20) '*', 'WATER SUMMARY', (20) '*')	MCDK0434
(COL(23),3 A) PAGE;	MCDK0435
PUT FILE(SYSPRNX) EDIT('EOF AFTER',RECORD_COUNT_DSK001,	MCDK0436
' RECORDS.') (SKIP(DEBUG),A(DEBUG*9),F(DEBUG*6),A(DEBUG*9));	MCDK0437
#AVE_CERR_PROG=#TOT_CERRS/#TOT_PROGS;	MCDK0438
#AVE_NTE_PROG=#TOT_NTERRS/#TOT_PROGS;	MCDK0439
#AVE_TERR_PROG=#TOT_TERRS/#TOT_PROGS;	MCDK0440
#AVE_ERRS_PROG=#TOTAL_ERRS/#TOT_PROGS;	MCDK0441
#AVE_ERRS_STATEMENT=#TOTAL_ERRS/#TOT_STATEMENTS;	MCDK0442
#AVE_ERRS_CAPD=#TOTAL_ERRS/#TOT_CARDS;	MCDK0443
PROB_OF_ERROR1=#ERRONEOUS_PROG/#TOT_PROGS;	MCDK0444

	PROB_QF_ERROR2=#PROG_GT1ERR/#TOT_PROGS;	MCDK0445
	#AVE_ERRON_PROG=#TOTAL_ERRS/#ERRONEOUS_PROG;	MCDK0446
CATCH:		MCDK0447
	ON ERROR SYSTEM;	MCDK0448
	SIGNAL ENDPAGE(PRINT);	MCDK0449
	PUT FILE(PRINT) LINE(5) EDIT	MCDK0450
	(OUTPUT_RECORD_1,OUTPUT_RECORD_1A,OUTPUT_RECORD_1B,	MCDK0451
	OUTPUT_RECORD_1C	MCDK0452
	(SKIP,COL(2),A,COL(107),F(12,5));	MCDK0453
	PUT FILE(PRINT) EDIT(OUTPUT_RECORD_1D)	MCDK0454
	(SKIP,COL(2),A,COL(110),F(9,7));	MCDK0455
	PUT FILE(PRINT) SKIP;	MCDK0456
	DO I=1 BY 1 TO #PE;	MCDK0457
	IF #ES(I)=0 THEN GO TO LBLI;	MCDK0458
	CURRENT_ERR1=ES(I);	MCDK0459
	PROB3=#ES(I)/#TOT_PROGS;	MCDK0460
	IF (PROB3<=0) (PROB4<=0) THEN	MCDK0461
	DO;	MCDK0462
	SIGNAL ENDPAGE(PRINT);	MCDK0463
	PUT FILE(PRINT) EDIT(OUTPUT_RECORD_2) (SKIP(3),COL(25),A,	MCDK0464
	COL(70),A);	MCDK0465
END;		MCDK0466
	DO J=1+I BY 1 TO #PE WHILE (PROB3<=0);	MCDK0467
	CURRENT_ERR2=ES(J);	MCDK0468
	PROB4=#ES(J)/#TOT_PROGS;	MCDK0469
	PROB5=PROB3*PROB4;	MCDK0470
	PROB6=PROB5/PROB3;	MCDK0471
	IF (PROB4<=0) (PROB5<=0) (PROB6<=0) THEN	MCDK0472
	PUT FILE(PRINT) EDIT(OUTPUT_RECORD_3) (SKIP	MCDK0473
	COL(117),A,COL(75),A);	MCDK0474
KPNKR:		MCDK0475
	END;	MCDK0476
LBLI:		MCDK0477
	END;	MCDK0478
	GO TO GRAPHER;	MCDK0479
	END;	MCDK0480
/*		MCDK0481

END OF SUMMARIZATION PHASE.

EXISTENCE AND CLASSIFICATION PHASES.

*/

CLASSIFY: BEGIN:

ON ENDPAGE(SYSPRINX) PUT FILE(SYSPRINX) PAGE EDIT
((20) '*', 'WATER PROGRAMS', (20) '*') (COL(25), 3 A);

IF PRINT_PROG THEN SIGNAL ENDPAGE(SYSPRINX);

DO RECORD_COUNT_DSK001=1 BY 1;

READ FILE(DSK001) INTO (XLINE);

#TOT_CARDS=#TOT_CARDS+1;

IF COL6 = ' ' THEN

#TOT_STATEMENTS=#TOT_STATEMENTS+1;

IF XC5=SLASH_DATA THEN DO;

#TOT_STATEMENTS=#TOT_STATEMENTS-1;

INPGFLG=0;

END;

IF XC4=SLASH_JOB THEN DO;

#TOT_STATEMENTS=#TOT_STATEMENTS-1;

INPGFLG=1;

GO TO TMPQUT;

END;

IF INDEX(ERRORS___, EFLAG1)~=0 THEN

ERROR_LINE:

DO;

/* ERROR CLASSIFICATION. */

II=(INDEX(ESX, EFLAG2)+3)/4;

PUT FILE(PRINT) EDIT

('ERROR NUMBER', II, ' ', ES(II), ' ENCOUNTERED.')

(SKIP(PRINT_ERR), COL(25), A(PRINT_ERR*12), X(PRINT_ERR*2),

F(PRINT_ERR*4), A(PRINT_ERR), X(PRINT_ERR*2),

A(PRINT_ERR*4), X(PRINT_ERR*2), A(PRINT_ERR*12));

#ES(II)=#ES(II)+1;

#TOT_STATEMENTS=#TOT_STATEMENTS-1;

#TOT_CARDS=#TOT_CARDS-1;

MCDK0482

MCDK0483

MCDK0484

MCDK0485

MCDK0486

MCDK0487

MCDK0488

MCDK0489

MCDK0490

MCDK0491

MCDK0492

MCDK0493

MCDK0494

MCDK0495

MCDK0496

MCDK0497

MCDK0498

MCDK0499

MCDK0500

MCDK0501

MCDK0502

MCDK0503

MCDK0504

MCDK0505

MCDK0506

MCDK0507

MCDK0508

MCDK0509

MCDK0510

MCDK0511

MCDK0512

MCDK0513

MCDK0514

MCDK0515

MCDK0516

MCDK0517

MCDK0518

CURRENT_TOTAL_ERRORS,	MCDK0519
ERFLG=ERFLG+1;	MCDK0520
#TOTAL_ERRS=#TOTAL_ERRS+1;	MCDK0521
IF INDEX(EXECUTION_TIME_ERROR_STRING,ES(II))=0	MCDK0522
THEN DO;#TOT_TERRS=#TOT_TERRS+1; ETRS=ETRS+1; END;	MCDK0523
IF INDEX(COMPILE_TIME_ERROR_STRING,ES(II))=0	MCDK0524
THEN DO; #TOT_CERRS=#TOT_CERRS+1; CTRS=CTRS+1; END;	MCDK0525
IF INDEX(NON_TERMINAL_EXEC_ERRORS,ES(II))=0	MCDK0526
THEN DO; #TOT_NTERRS=#TOT_NTERRS+1; NTRS=NTRS+1; END;	MCDK0527
END;	MCDK0528
IF (CT=CTX)&(SXT=SXXT)&(SOC=SOXT)&(BAA=BXAA)&(UNUSED=XUNUSED)	MCDK0529
THEN DO ;	MCDK0530
PUT FILE(SYSPRNX) EDIT('EOJ ENCOUNTERED')	MCDK0531
(SKIP(DEBUG),A(DEBUG*15));	MCDK0532
IF ERFLG>0 THEN DO;	MCDK0533
#ERRONEOUS_PROG=#ERRONEOUS_PROG+1;	MCDK0534
#PROG_GTIERR=#PROG_GTIERR+(ERFLG>1);	MCDK0535
/* MAX & MIN ERROR */	MCDK0536
#MAX_CERR_PROGRAM=MAX(CTRS,#MAX_CERR_PROGRAM);	MCDK0537
IF CTRS=0 THEN	MCDK0538
#MIN_CERR_PROGRAM=MIN(CTRS,#MIN_CERR_PROGRAM);	MCDK0539
#MAX_NTE_PROG=MAX(#MAX_NTE_PROG,NTRS);	MCDK0540
IF NTRS=0 THEN	MCDK0541
#MIN_NTE_PROG=MIN(#MIN_NTE_PROG,NTRS);	MCDK0542
#MAX_TERR_PROG=MAX(#MAX_TERR_PROG,ETRS);	MCDK0543
IF ETRS=0 THEN	MCDK0544
#MIN_TERR_PROG=MIN(#MIN_TERR_PROG,ETRS);	MCDK0545
#MAXERRS=MAX(#MAXERRS,CURRENT_TOTAL_ERRORS);	MCDK0546
#MINERRS=MIN(#MINERRS,CURRENT_TOTAL_ERRORS);	MCDK0547
END;	MCDK0548
#TOT_CARDS=#TOT_CARDS-1;	MCDK0549
#TOT_STATEMENTS=#TOT_STATEMENTS-1;	MCDK0550
#TOT_PROGS=#TOT_PROGS+1;	MCDK0551
ETRS,CTRS,NTRS,CURRENT_TOTAL_ERRORS,	MCDK0552
INPGFLG,	MCDK0553
ERFLG=0;	MCDK0554
GO TO NXTLINE;	MCDK0555


```

                                END;
TMPQUT:
    PUT FILE(SYSPRNX) EDIT(ULINE)
    (SKIP(PRINT_PROG),COL(15),A(091*PRINT_PROG));
NXTLINE:
    END;
    END CLASSIFY;
/*
    END OF EXISTENCE AND CLASSIFICATION PHASES.

    GRAPHING PHASE.
*/
GRAPHER:
    IF -NOGRAPH THEN
    DO;
    SIGNAL ENDPAGE(SYSPRNX);
    P=ADDR(#QNER(1));
    #QNER(0)=0;
    PUT FILE(SYSPRNX) EDIT('ENTER GRAPHER')
    (A(DEBUG*14)) SKIP(DEBUG);
    /* IF PRINT_ERR THEN OUTPUT THE OCCURRENCES OF EACH ERROR. */
    PUT FILE(PRINT) SKIP(PRINT_ERR);
        PUT FILE(PRINT) EDIT
        ((ES(I),#ES(I) DO I=1 TO #PE))
        ( SKIP,
          6 (A((#ES(I)-=0)*PRINT_ERR *4),
            X((#ES(I)-=0)*PRINT_ERR*2)
            ,F((#ES(I)-=0)*PRINT_ERR*4),
            X((#ES(I)-=0)*PRINT_ERR*4))) );
    /* SORT TO FIND THE MOST FREQUENT ERRORS. */
    /* QUADRATIC SORT. THE TECHNIQUE IS IN FLORES' BOOK. */
    PUT FILE(SYSPRNX) EDIT('ENTER SORT.')
    (SKIP(DEBUG),A(DEBUG*12));
    SUBLISTS.A=20000; SUBLISTS.B=' ';
    LEAST_TABLE.A=20000; LEAST_TABLE.B=' ';

```

MCDK0556
MCDK0557
MCDK0558
MCDK0559
MCDK0560
MCDK0561
MCDK0562
MCDK0563
MCDK0564
MCDK0565
MCDK0566
MCDK0567
MCDK0568
MCDK0569
MCDK0570
MCDK0571
MCDK0572
MCDK0573
MCDK0574
MCDK0575
MCDK0576
MCDK0577
MCDK0578
MCDK0579
MCDK0580
MCDK0581
MCDK0582
MCDK0583
MCDK0584
MCDK0585
MCDK0586
MCDK0587
MCDK0588
MCDK0589
MCDK0590
MCDK0591
MCDK0592

```

DO I=1 TO 18;
DO J=1 TO 18;
N=18*(I-1)+J;
IF N>#PE THEN GO TO NXTRC;
TMP.A=#ES(N);
TMP.B=ES(N);
IF TMP.A<LEAST_TABLE.A(I) THEN LEAST_TABLE(I)=TMP,BY NAME;
SUBLISTS(I,J)=TMP,BY NAME;
END;
NXTRC:
END;
PUT FILE(SYSPRINX) EDIT(SUBLISTS,LEAST_TABLE)
(SKIP(DEBUG),COL(10),10 (F(DEBUG*6),A(DEBUG*4)));
SORT:
DO NN=#PE BY -1 TO 1 WHILE(N->=0);
LKP=20000; N=0;
DO I=1 TO 18 WHILE(LKP>0);
IF LEAST_TABLE.A(I)<LKP THEN DO;
N=I;
LKP=LEAST_TABLE.A(I);
END;
END;
ES(NN)=LEAST_TABLE.B(N);
#ES(NN)=LEAST_TABLE.A(N);
LEAST_TABLE.A(N),LKP=20000;
I=N;
DO J=1 TO 18 WHILE((LKP>0)&(I>0));
TMP=SUBLISTS(I,J),BY NAME;
IF TMP.A<LKP THEN DO;
LEAST_TABLE(I)=SUBLISTS(I,J),BY NAME;
LKP=TMP.A;
END;
END;
SUBLISTS.A(I,J)=20000;
END;
END_OF_SORT:
PUT FILE(SYSPRINX) EDIT

```

```

MCDK0593
MCDK0594
MCDK0595
MCDK0596
MCDK0597
MCDK0598
MCDK0599
MCDK0600
MCDK0601
MCDK0602
MCDK0603
MCDK0604
MCDK0605
MCDK0606
MCDK0607
MCDK0608
MCDK0609
MCDK0610
MCDK0611
MCDK0612
MCDK0613
MCDK0614
MCDK0615
MCDK0616
MCDK0617
MCDK0618
MCDK0619
MCDK0620
MCDK0621
MCDK0622
MCDK0623
MCDK0624
MCDK0625
MCDK0626
MCDK0627
MCDK0628
MCDK0629

```

('COMPLETE SORT.')	MCDK0630
(SKIP(DEBUG),A(14*DEBUG));	MCDK0631
ON ENDPAGE(PRINT) PUT FILE(PRINT) PAGE	MCDK0632
EDIT((20) '*, 'HISTOGRAM OF ERROR DISTRIBUTION', (20) '*,	MCDK0633
(COL(25),2 A);	MCDK0634
NER=XNER; #NER=#XNER;	MCDK0635
DO K=1 BY 1 TO 24;	MCDK0636
DO J=1 BY 1 TO 96;	MCDK0637
DO I=1 BY 1 TO 31;	MCDK0638
II=32-I;	MCDK0639
III=II*2.5;	MCDK0640
IF III>#NER(K) THEN GO TO NXTPT;	MCDK0641
IF (III<=#NER(K))&(III>#QNER(K-1)) THEN	MCDK0642
GRAPH(I,J)='_';	MCDK0643
IF MOD(J,4)=0 THEN GRAPH(I,J)=' ';	MCDK0644
	MCDK0645
END;	MCDK0646
END;	MCDK0647
END;	MCDK0648
SIGNAL ENDPAGE(PRINT);	MCDK0649
PUT FILE(PRINT) LINE(10);	MCDK0650
DO I=1 TO 31;	MCDK0651
PUT FILE(PRINT) EDIT	MCDK0652
(VLABEL(I),VLABELA(I),VAXIS,GRAPH(I,*))	MCDK0653
(SKIP,2 A,X(21,97 A);	MCDK0654
END;	MCDK0655
PUT FILE(PRINT) EDIT	MCDK0656
(HAXIS,NER,HLABEL)	MCDK0657
(SKIP,COL(5),A,SKIP,COL(5),24 A(4),SKIP(2),COL(45),A);	MCDK0658
END;	MCDK0659
/* END OF GRAPHING PHASE. */	MCDK0660
CLOSE FILE(PRINT),FILE(DSK001),FILE(SYSPRINX);	MCDK0661
END ANLYZ1;	MCDK0662

```

/*          ERROR CORRECTION PROGRAM          */
(SUBRG,STRG,SIZE):
CORRECT: PROC(PARM) OPTIONS(MAIN);
DCL      PARM CHAR(60) VAR,
        1 PARMS STATIC UNALIGNED EXT,
        2 DEBUG BIT(1),          /* DEBUG INFORMATION */
        2 COPYOUTPUT BIT(1),     /* COPY WTO */
        2 COPYINPUT BIT(1),      /* COPY DSK001 TO PRINT */
        2 PRNT BIT(1),           /* PRINT NEW PROGRAMS */
COPYOUT DEF COPYOUTPUT BIT(1),
(WTO, /* WATFOR OUTPUT */
DSK001, /* WATFOR INPUT */
NEWPGM, /* NEW WATFOR PROGRAM */
SCRO01, /* SCRATCH DATA SET. */
PGMNEW /* ALL NEW WATFOR PROGRAMS. */ ,
PRINT /* COPY OUTPUT, DEBUG OUTPUT */
) FILE EXT,
SYSPRINX /* COPY OUTPUT, NEW INPUT */ FILE INT,
(P,Q) POINTER,
XLINE CHAR(133),
LINE CHAR(122) DEF XLINE POS(2),
CARD CHAR(80) DEF XLINE POS(11),
SAVECARD CHAR(80) BASED(P),
XCARD CHAR(80) BASED(Q),
SLASH_STAR CHAR(2) STATIC INIT('/*'),
SLASH_DATA CHAR(5) STATIC INIT('/DATA'),
SLASH_STOP CHAR(5) STATIC INIT('/STOP'),
SLASH_JOB CHAR(4) STATIC INIT('/JOB'),
EFLAG1 CHAR(11) DEF XLINE POS(2),
(EFLG,DFLG) BIT(1),
ERRORS___ CHAR(33) STATIC INIT
('***ERROR*****WARNING***EXTENSION*'),
CRRCTN ENTRY,
CRRCTR ENTRY(CHAR(*),CHAR(*)) RETURNS(CHAR(80) VAR),
RESET ENTRY,
TFILL ENTRY,
TMAT(0:5,0:4) FIXED BIN(15,0) EXT STATIC,

```

```

*/00000870
00000880
00000890
00000900
00000910
00000920
00000930
00000940
00000950
00000960
00000970
00000980
00000990
00001000
00001010
00001020
00001030
00001040
00001050
00001060
00001070
00001080
00001090
00001100
00001110
00001120
00001130
00001140
00001150
00001160
00001170
00001180
00001190
00001200
00001210
00001220
00001230

```

```

SLASH CHAR(1) STATIC INIT(''),
XC1 CHAR(1) DEF XLINE POS(11),
XC2 CHAR(2) DEF XLINE POS(11),
XC4 CHAR(4) DEF XLINE POS(11),
XC5 CHAR(5) DEF XLINE POS(11);

```

```

/*
*/
/*

```

IDENTIFICATION:

```

PROGRAM-ID:  CORRECT.
AUTHOR:  G. E. HEDRICK.
INSTALLATION:  IOWA STATE UNIVERSITY COMPUTATION CENTER.
DATE-WRITTEN:  14 FEBRUARY 1970.
SOURCE-COMPUTER:  IBM 360/65.
OBJECT-COMPUTER:  IBM 360/65.
OPERATING SYSTEM:  OS/MVT.
MEMORY SIZE:  HIGH SPEED CORE:  128K.
                BULK CORE:  96K.

```

DESCRIPTION OF THE PROBLEM:

THIS ENTIRE PROGRAM IS DESIGNED TO CORRECT SOURCE ERRORS IN FORTRAN PROGRAMS. THIS MODULE SERVES AS THE CONTROLLING ANALYSIS AND CORRECTION MODULE. ITS CONTROL FUNCTION IS MADE EASIER BY THE FACT THAT IT IS THE MAIN PROGRAM FOR THE ENTIRE SET OF PL/1 PROCEDURES.

MUCH OF THE ERROR ANALYSIS HAS BEEN PERFORMED PREVIOUSLY. THIS PRIOR ANALYSIS IS INSTALLATION DEPENDENT AND MUST BE REPEATED AT EACH INSTALLATION WHERE THE CORRECTOR IS USED. THIS ANALYSIS IS INDEPENDENT OF THE CORRECTOR PROGRAM ITSELF. THE EARLY ANALYSIS INCLUDES THE GATHERING OF STATISTICS AND

```

00001240
00001250
00001260
00001270
00001280
00001290
00001300
00001310
00001320
00001330
00001340
00001350
00001360
00001370
00001380
00001390
00001400
00001410
00001420
00001430
00001440
00001450
00001460
00001470
00001480
00001490
00001500
00001510
00001520
00001530
00001540
00001550
00001560
00001570
00001580
00001590
00001600

```

THE CALCULATION OF ESTIMATED PROBABILITIES FOR EACH OF THE
VARIOUS TYPES OF ERROR. IT IS THE CALCULATION OF THESE
STATISTICS AND ESTIMATED PROBABILITIES THAT MAY BE PERFORMED
INDEPENDENTLY.

ADDITIONAL ERROR ANALYSIS IS PERFORMED WITHIN THIS
MODULE AND THE OTHER MODULES WHICH IT INVOKES, EITHER DIRECTLY
OR INDIRECTLY. THE ANALYSIS WHICH IS DONE WITHIN THIS ERROR
CORRECTOR IS NEEDED AT THE TIME THAT IT IS DONE AND CANNOT BE
DONE IN A PRECEDING EXTERNAL PROGRAM. THE GENERAL OUTLINE OF
THE ANALYSIS IS INCORPORATED WITH THE GENERAL OUTLINE OF
CORRECTION IN THE NEXT SECTION AND IN THE OTHER MODULES.

THE ACTUAL METHOD USED BY THE CORRECTION CONTROL MODULE
IS DESCRIBED BELOW. THE OVERALL CORRECTION PROCESS MUST,
HOWEVER, CORRECT THE ERRORS IN THE SOURCE PROGRAM BY USING
THE INFORMATION WHICH IS FOUND IN THE PROGRAM LISTING, ON THE
ORIGINAL SOURCE CARDS, ON THE DATA CARDS, AND IN THE PROGRAM'S
OUTPUT. THIS INFORMATION IS ACTUALLY USED TO EFFECT THE FINAL
CORRECTION.

METHOD OF SOLUTION:

THE METHOD USED BY THE CONTROLLING CORRECTION MODULE
CONSISTS OF THE STEPS LISTED BELOW:

1. SET UP THE PARAMETER FLAGS;
2. OPEN ALL REQUIRED FILES;
3. READ THE WATFOR JOB CARD;
4. TEST FOR JCL ERRORS AND CREATE A JOB CARD FOR THIS PROGRAM;
5. READ A LINE OF THE PROGRAM LISTING;
6. IF THE LINE IS A /JOB IMAGE GO TO STEP 12;
7. IF THE LINE IS A /DATA IMAGE GO TO STEP 10;
8. IF THE LINE IS AN ERROR MESSAGE LINE SET THE ERROR FLAG
AND TRANSFER TO THE CRRCTR MODULE;
9. WRITE THE CORRECT(ED) FORTRAN STATEMENT ON THE TEMPORARY
NEW PROGRAM OUTPUT FILE;

00001610
00001620
00001630
00001640
00001650
00001660
00001670
00001680
00001690
00001700
00001710
00001720
00001730
00001740
00001750
00001760
00001770
00001780
00001790
00001800
00001810
00001820
00001830
00001840
00001850
00001860
00001870
00001880
00001890
00001900
00001910
00001920
00001930
00001940
00001950
00001960
00001970

10. IF THERE HAVE BEEN ANY ERRORS FLAGGED IN THE PROGRAM'S	00001980
OUTPUT THEN SET THE ERROR FLAG;	00001990
11. OUTPUT THE PROGRAM'S DATA TO THE NEW PROGRAM FILE;	00002000
12. IF THE ERROR FLAG IS SET INVOKE THE CORRCTN MODULE;	00002010
OTHERWISE, INVOKE THE RESET MODULE;	00002020
13. RESET THE ERROR FLAG;	00002030
14. GO TO STEP 3.	00002040

DESCRIPTION OF INPUT:

-----	00002050
	00002060
	00002070
	00002080
	00002090
	00002100
THE PRIMARY INPUT FOR THIS PROGRAM COMES FROM ONE OF TWO	00002110
SOURCES: THE WTO FILE OR THE DSK001 FILES. THE INPUT ARRIVES	00002120
EITHER AS A LINE IMAGE OR AS A CARD IMAGE. INPUT FROM THE	00002130
WTO FILE IS IN THE FORM OF LINE IMAGES OF THE FORTRAN PROGRAM	00002140
LISTING AND OF LINE IMAGES OF THE FORTRAN PROGRAM'S OUTPUT.	00002150
THE INPUT FROM THE DSK001 FILE IS IN THE FORM OF CARD IMAGES	00002160
OF THE ORIGINAL FORTRAN PROGRAM AND ITS DATA.	00002170

ONE ADDITIONAL FORM OF INPUT IS PARAMETER INPUT. THIS	00002180
INPUT IS OBTAINED FROM THE PARM.GO FIELD OF THE EXEC CARD.	00002190
THE INPUT PARAMETERS INDICATE WHAT TYPE OF OUTPUT IS DESIRED	00002200
FROM THE COMPLETE CORRECTOR PROGRAM. THERE ARE SEPARATE	00002210
PARAMETERS TO REQUEST DEBUG OUTPUT, THAT THE INPUT BE COPIED	00002220
ONTO THE PRINTER, AND THAT THE DISK OUTPUT BE COPIED ONTO THE	00002230
PRINTER.	00002240

DESCRIPTION OF OUTPUT:

-----	00002250
	00002260
	00002270
	00002280
	00002290
	00002300
THERE ARE TWO BASIC CLASSES OF OUTPUT FROM THIS JOB.	00002310
THE FIRST CLASS IS THE STANDARD OUTPUT. STANDARD OUTPUT IS	00002320
ALWAYS PRODUCED AND IS IN THE FORM OF A NEW PROGRAM INPUT DATA	00002330
SET FOR THE WATFOR COMPILER.	00002340.

THE SECOND CLASS OF OUTPUT IS THE REQUESTED OUTPUT. THIS OUTPUT MUST BE REQUESTED WITH AN INPUT PARAMETER AND IS PUT ON THE VARIOUS PRINT DATA SETS. ITS PRIMARY FUNCTION IS TO SERVE AS AN AID IN PROGRAM DEBUGGING.

00002350
00002360
00002370
00002380
00002390
00002400
00002410
00002420
00002430
00002440
00002450
00002460
00002470
00002480
00002490
00002500
00002510
00002520
00002530
00002540
00002550
00002560
00002570
00002580
00002590
00002600
00002610
00002620
00002630
00002640
00002650
00002660
00002670
00002680
00002690
00002700
00002710

TABLE OF INPUT PARAMETERS:

PARAMETER	FUNCTION
DEBUG.	REQUESTS DEBUG OUTPUT.
COPYINPUT.	REQUESTS THAT ALL INPUT BE COPIED ONTO THE PRINTER.
COPYOUTPUT.	REQUESTS THAT THE DISK OUTPUT BE COPIED ONTO THE PRINTER.
PRNT(PRINT).	REQUESTS THAT ADDITIONAL PRINTED OUTPUT BE GIVEN.

DESCRIPTION OF FILES:

FILE	USE
DSK001.	INPUT FILE WHICH CONTAINS WATFOR PROGRAM AND DATA.
NEWPGM.	TEMPORARY NEW PROGRAM FILE; CONTAINS ONE NEW PROGRAM AS IT IS BEING CORRECTED.
PGMNEW.	THE FILE WHICH (AT THE TERMINATION OF THE CORRECTOR) CONTAINS THE NEW BATCH OF FORTRAN PROGRAMS.
PRINT, SYSPRINX,	FILES WHICH CONTAIN THE VARIOUS TYPES OF

SYSPRINT.	PRINTED OUTPUT.	00002720
WTO.	INPUT FILE WHICH CONTAINS FORTRAN	00002730
	PROGRAMS' LISTINGS AND OUTPUT.	00002740
		00002750
		00002760
		00002770
		00002780
DESCRIPTION OF VARIABLES AND LABELS:		00002790
-----		00002800
VARIABLE OR LABEL	MEANING OR USE	00002810
-----	-----	00002820
		00002830
CARD,XCARD,SAVECARD.	CARD IMAGES.	00002840
COPYINPUT,COPYOUTPUT,INPUT PARAMETERS.		00002850
COPYOUT,DEBUG,PRNT,		00002860
PARMS,PARM.		00002870
CRRCTN,CRRCTR,RESET.	OTHER CORRECTOR MODULES.	00002880
DFLG,EFLAG1,EFLG.	ERROR FLAGS.	00002890
DSK001,NEWPGM,	FILES USED WITHIN THE CORRECTOR PROGRAM.	00002900
PGMNEW,PRINT,		00002910
SYSPRINX,WTO.		00002920
ERROR_LINE.	PLACE TO WHICH CONTROL IS TRANSFERRED	00002930
	WHEN AN ERROR LINE IS ENCOUNTERED IN THE	00002940
	PROGRAM LISTING.	00002950
ERRORS_____.	A LIST OF THE MOST GENERAL WATFOR ERROR	00002960
	FLAGS.	00002970
LINE,XLINE.	LISTING AND OUTPUT LINE IMAGES.	00002980
P,Q.	POINTERS USED FOR OVERLAY DEFINING.	00002990
QUIT,QUITR.	LABELS USED AT JOB TERMINATION.	00003000
RD1,RD2,RD3.	LABELS WHERE SELECTED READ PROCESSES	00003010
	BEGIN.	00003020
S2,S4,S5,S5A,S5B,	PLACES WHERE STEPS OF THE ORIGINAL (NOT THE	00003030
S11,S15.	CURRENT) ALGORITHM BEGAN.	00003040
SLASH,SLASH_DATA,	WATFOR CONTROL INFORMATION.	00003050
SLASH_JOB,SLASH_STAR,		00003060
SLASH_STOP.		00003070
XCN,N=1,2,3,4,5.	THE FIRST N CHARACTERS OF A CARD IMAGE.	00003080

REFERENCES:

THE REFERENCES ARE LISTED WITH THE OTHER MODULES AND IN THE BIBLIOGRAPHY OF THE DISSERTATION WHICH SERVES AS A DOCUMENTATION OF THIS PROGRAM.

*/

Q=ADDR(CARD); ALLOCATE SAVECARD;	00003090
CALL TFILL;	00003100
/* SET UP PARAMETER FLAGS. */	00003110
DEBUG=INDEX(PARM,'DEBUG')>0;	00003120
COPYOUTPUT=INDEX(PARM,'COPYOUTPUT')>0;	00003130
COPYINPUT=INDEX(PARM,'COPYINPUT')>0;	00003140
PRNT=INDEX(PARM,'PRINT')>0;	00003150
/* OPEN ALL FILES. */	00003160
OPEN FILE(WTO) RECORD SEQUENTIAL INPUT, /* WATFOR OUTPUT */	00003170
FILE(DSK001) RECORD SEQUENTIAL INPUT, /*WATFOR INPUT */	00003180
FILE(NEWPGM) RECORD SEQUENTIAL OUTPUT, /* NEW PROGRAMS */	00003190
FILE(PRINT) STREAM PRINT LINESIZE(120) PAGESIZE(55),	00003200
FILE(SYSPRINX) STREAM PRINT LINESIZE(120) PAGESIZE(55)	00003210
TITLE('SYSPRIN');	00003220
/* ENDPAGE ACTIONS. */	00003230
ON ENDPAGE(PRINT) PUT FILE(PRINT) PAGE EDIT	00003240
((45) '*', 'DEBUG OUTPUT', (45) '*')	00003250
(SKIP(2), 3 A);	00003260
ON ENDPAGE(SYSPRINX) PUT FILE(SYSPRINX) EDIT	00003270
((45) '*', 'COPY FILE', (45) '*')	00003280
(SKIP(2), 3 A);	00003290
IF COPYOUT DEBUG THEN SIGNAL ENDPAGE(PRINT);	00003300
IF COPYINPUT PRNT THEN SIGNAL ENDPAGE(SYSPRINX);	00003310
	00003320
	00003330
	00003340
	00003350
	00003360
	00003370
	00003380
	00003390
	00003400
	00003410
	00003420
	00003430
	00003440
	00003450

PUT FILE(PRINT) EDIT('FILES OPENED') (SKIP(DEBUG), A(DEBUG*14));	00003460
IF DEBUG THEN PUT FILE(PRINT) DATA(PARMS) SKIP;	00003470
/* ENDFILE ACTIONS. */	00003480
ON ENDFILE(WTO) BEGIN;	00003490
IF XC4=SLASH_JOB THEN GO TO QUIT;	00003500
ELSE XC4=SLASH_JOB;	00003510
END;	00003520
ON ENDFILE(DSK001) GO TO QUIT;	00003530
ON FINISH BEGIN;	00003540
ON ERROR SNAP SYSTEM;	00003550
ON TRANSMIT(NEWPGM) SNAP GO TO QUITR;	00003560
ON TRANSMIT(PGMNEW) SNAP GO TO QUITR;	00003570
SAVECARD=SLASH_STOP; XCARD=SLASH_STAR;	00003580
WRITE FILE(PGMNEW) FROM(SAVECARD);	00003590
WRITE FILE(PGMNEW) FROM(XCARD);	00003600
CLOSE FILE(PRINT), FILE(SYSPRINX), FILE(WTO),	00003610
FILE(NEWPGM),	00003620
FILE(PGMNEW),	00003630
FILE(DSK001);	00003640
QUITR:	00003650
END;	00003660
IF DEBUG THEN ON ERROR SNAP BEGIN;	00003670
PUT FILE(PRINT) EDIT(ONCODE) (SKIP,A);	00003680
CLOSE FILE(NEWPGM);	00003690
OPEN FILE(NEWPGM) RECORD SEQUENTIAL OUTPUT;	00003700
END;	00003710
/* READ A JOB CARD */	00003720
READ FILE(DSK001) INTO(SAVECARD);	00003730
READ FILE(WTO) INTO(XLINE);	00003740
	00003750
IF SAVECARD=CARD THEN PUT FILE(PRINT) EDIT('***JCL ERROR***') (SKIP, A);	00003760
	00003770
IF COPYINPUT THEN PUT FILE(PRINT) EDIT(SAVECARD)(SKIP,A);	00003780
PUT FILE(SYSPRINX) EDIT(LINE)	00003790
(SKIP(COPYOUTPUT),A(COPYOUTPUT*120));	00003800
/* WRITE JOB CARD */	00003810
	00003820

S2:

S4:

PUT FILE(PRINT) EDIT('1',XCARD)	00003830
(SKIP(DEBUG),A(DEBUG),A(80*DEBUG));	00003840
WRITE FILE(NEWPGM) FROM(XCARD);	00003850
S5:	00003860
SAVECARD=CARD;	00003870
READ FILE(WTO) INTO(XLINE);	00003880
S5A:	00003890
PUT FILE(PRINT) EDIT(CARD) (SKIP(COPYINPUT),A(COPYINPUT*80));	00003900
PUT FILE(SYSPRINX) EDIT(LINE)	00003910
(SKIP(COPYOUTPUT),A(COPYOUTPUT*120));	00003920
IF XC4=SLASH_JOB THEN GO TO S15;	00003930
IF XC5=SLASH_DATA THEN GO TO S11;	00003940
IF XC5=SLASH_STOP THEN GO TO S15;	00003950
IF XC2=SLASH_STAR THEN GO TO QUIT;	00003960
IF INDEX(ERRORS____,EFLAG1)≠0 THEN	00003970
ERROR_LINE: DO;	00003980
EFLG='1'B;	00003990
XCARD=CRRCTR(SAVECARD,XLINE);	00004000
PUT FILE(PRINT) EDIT('2',XCARD)	00004010
(SKIP(DEBUG),A(DEBUG),A(80*DEBUG));	00004020
IF XCARD≠' ' THEN	00004030
WRITE FILE(NEWPGM) FROM(XCARD);	00004040
SAVECARD=XCARD;	00004050
S5B:	00004060
READ FILE(WTO) INTO(XLINE);	00004070
IF XC5=SLASH_DATA THEN GO TO S11;	00004080
/* PREVENT MULTIPLE CORRECTIONS FOR DEPENDENT ERRORS. */	00004090
IF INDEX(ERRORS____,EFLAG1)>0 THEN GO TO S5B;	00004100
ELSE GO TO S5A;	00004110
END;	00004120
S11:	00004130
PUT FILE(PRINT) EDIT('3',XCARD)	00004140
(SKIP(DEBUG),A(DEBUG),A(80*DEBUG));	00004150
WRITE FILE(NEWPGM) FROM(XCARD);	00004160
IF XC5≠SLASH_DATA THEN	00004170
GO TO S5;	00004180
/* ANY ERRORS IN RESULTS. */	00004190

RD1:	READ FILE(WTO) INTO(XLINE);	00004200
	PUT FILE(SYSPRINX) EDIT(LINE)	00004210
	(SKIP(COPYOUT),A(COPYOUT*120));	00004220
	IF INDEX(ERRORS___,EFLAG1)≠0 THEN EFLG='1'B;	00004230
	IF(XC4≠SLASH_JOB)&(XC5≠SLASH_STOP) THEN GO TO RD1;	00004240
	/* OUTPUT DATA TO NEWPROGRAM FILE. */	00004250
RD2:	READ FILE(DSK001) INTO(SAVECARD);	00004260
	PUT FILE(PRINT) EDIT(SAVECARD)	00004270
	(SKIP(COPYINPUT),A(COPYINPUT*80));	00004280
	CARD=SAVECARD;	00004290
	IF XC5≠SLASH_DATA THEN GO TO RD2;	00004300
RD3:	READ FILE(DSK001) INTO(SAVECARD);	00004310
	PUT FILE(PRINT) EDIT(SAVECARD)	00004320
	(SKIP(COPYINPUT),A(COPYINPUT*80));	00004330
	CARD=SAVECARD;	00004340
	IF XC1≠SLASH THEN DO;	00004350
	WRITE FILE(NEWPGM) FROM(SAVECARD);	00004360
	GO TO RD3;	00004370
	END;	00004380
S15:	IF EFLG THEN CALL CRRCTN;	00004390
	ELSE CALL RESET;	00004400
	GO TO S2;	00004410
QUIT:		00004420
	RETURN;	00004430
	END CORRECT;	00004440

```

/*          EXECUTION ANALYSIS AND CORRECTION MODULE          */
(SUBRG,STRG,SIZE):
CRRCTN: PROC;
DCL    DICTIONARY(255) CHAR(11) VAR,
        1 PARMS STATIC UNALIGNED EXT,
        2 (DEBUG,COPYOUTPUT,COPYINPUT,PRNT) BIT(1),
        XDIC CHAR(2805) VAR,
        TEXT CHAR(11) VAR,
        TXT CHAR(11) VAR,
        (DICTLEN INIT(111),CNT,I,DCNT) FIXED BIN(15,0),
        (CHARREG,DCREG) BIT(28),
        (REGCHAR DEF CHARREG POS(1), DREGC DEF DCREG POS(1)) (28)
        BIT(1),
        SIDE CHAR(1),
        TSTR CHAR(1),
        XCHARS CHAR(28) STATIC INIT(' ZYXWVUTSRQPONMLKJIHGFEDCBA'),
        CHARS(28) CHAR(1) DEF XCHARS POS(1),
        ALG(-1:1) LABEL,
        II PICTURE '9',
        FLG BIT(1) INIT('0'B),
        (SET,UNSET) ENTRY,
        GETR ENTRY(CHAR(11) VAR,CHAR(1)),
        PUTR ENTRY(CHAR(11) VAR,BIT(1)),
        DFILL ENTRY((255) CHAR(11) VAR),
        (J,JJ) FIXED BIN(15,0),
        PRINT FILE EXT,
        SPECIALX CHAR(18) STATIC INIT('#@%,*._|&>;~?" '),
        SPECIAL(18) CHAR(1) DEF SPECIALX POS(1);
/*
IDENTIFICATION:
-----

PROGRAM-ID:  CRRCTN.
AUTHOR:  G. E. HEDRICK.
INSTALLATION:  IOWA STATE UNIVERSITY COMPUTATION CENTER.
DATE-WRITTEN:  4 FEBRUARY 1970.

```

```

*/ 00004460
00004470
00004480
00004490
00004500
00004510
00004520
00004530
00004540
00004550
00004560
00004570
00004580
00004590
00004600
00004610
00004620
00004630
00004640
00004650
00004660
00004670
00004680
00004690
00004700
00004710
00004720
00004730
00004740
00004750
00004760
00004770
00004780
00004790
00004800
00004810
00004820

```

SOURCE COMPUTER: IBM 360/65.
OBJECT COMPUTER: IBM 360/65.
OPERATING SYSTEM: OS/MVT.
MEMORY SIZE: DEPENDS ON OTHER PROCEDURES.

00004830
00004840
00004850
00004860
00004870
00004880
00004890
00004900
00004910
00004920
00004930
00004940
00004950
00004960
00004970
00004980
00004990
00005000
00005010
00005020
00005030
00005040
00005050
00005060
00005070
00005080
00005090
00005100
00005110
00005120
00005130
00005140
00005150
00005160
00005170
00005180
00005190

DESCRIPTION OF THE PROBLEM:

THIS PROGRAM IS DESIGNED TO CORRECT SOURCE ERRORS IN FORTRAN PROGRAMS. THIS PARTICULAR MODULE ATTEMPTS TO CORRECT THOSE ERRORS WHICH MAY BE VIEWED AS SPELLING ERRORS. THE PRESENCE OF SUCH ERRORS MAY BE INDICATED BY ANY OF SEVERAL ERROR MESSAGES FROM THE FORTRAN PROGRAM. THE MOST COMMON ERRORS WHICH ARE FLAGGED IN THIS WAY, ARE UNDEFINED VARIABLES, ARITHMETIC OVERFLOW, AND ARITHMETIC UNDERFLOW.

METHOD OF SOLUTION:

THE SOLUTION OF THE PROBLEM WHICH IS IMPLEMENTED IN THIS PROGRAM RELIES ON TWO ASSUMPTIONS:

1. A WORD WHICH IS IN ERROR HAS AT MOST ONE ERROR;
2. IF AN ERROR OCCURS IT FALLS INTO ONE OF THE

FOLLOWING FOUR CATEGORIES:

- A. A SINGLE LETTER IN ERROR,
- B. A MISSING LETTER,
- C. AN EXTRA LETTER,
- D. A SINGLE INVERSION, POSSIBLY WITH A SINGLE LETTER IN ERROR.

DAMEREAU STATES THAT 80% OF ALL SPELLING ERRORS SATISFY THESE ASSUMPTIONS AND THAT 95% OF SUCH ERROR CAN BE CORRECTED BY THE USE OF ONE OF HIS THREE ALGORITHMS.

UNDER THE PRECEDING ASSUMPTIONS THE OVERALL PROCESS

CONSISTS OF THE FOLLOWING STEPS:

1. CREATE THE DICTIONARY; 00005200
2. READ ONE WORD ON TEXT; 00005210
3. IF THE WORD IS THE LEFT HAND SIDE OF A FORTRAN 00005220
STATEMENT, ENTER THE WORD INTO THE DICTIONARY AND
GO TO STEP 2; 00005230
4. IF THE WORD IS LESS THAN THREE CHARACTERS, OR 00005240
IF THE WORD IS ALREADY IN THE DICTIONARY GO TO STEP 2; 00005250
5. SET UP THE CHARACTER REGISTER AND CHARACTER COUNT; 00005260
6. IF THE CHARACTER COUNT OF THE INPUT WORD AND THE 00005270
CHARACTER COUNT OF THE DICTIONARY WORD DIFFER BY MORE 00005280
THAN 2 THEN GO TO STEP 9; 00005290
7. IF THE CHARACTER REGISTERS DIFFER IN MORE THAN TWO 00005300
POSITIONS THEN GO TO STEP 9; 00005310
8. GO TO ALGORITHM -1 IF THE ENTRY WORD IS LARGER THAN 00005320
THE DICTIONARY WORD; 00005330
- GO TO ALGORITHM 0 IF THE WORDS HAVE THE SAME LENGTH; 00005340
- GO TO ALGORITHM 1 IF THE DICTIONARY WORD IS LONGER; 00005350
9. UPDATE DICTIONARY COUNT; 00005360
10. IF THERE ARE MORE DICTIONARY ENTRIES GO TO STEP 5; 00005370
11. IF THERE IS MORE INPUT GO TO STEP 2; 00005380
12. RETURN FROM THIS MODULE; 00005390

EACH OF THE THREE CORRECTION ALGORITHMS IS DESCRIBED AT ITS 00005400
POINT OF INVOCATION. 00005410

NOTE THAT MUCH OF THE DATA THAT WAS OBTAINED WITH THE SEPARATE 00005420
SAMPLING AND ANALYSIS PROGRAMS IS IMPLICITLY BUILT INTO THIS 00005430
PROCEDURE. 00005440

DESCRIPTION OF INPUT:

INPUT FOR THIS PROGRAM COMES FROM CALLS TO THE GETR 00005450
PROCEDURE. EACH CALL TO THE GETR PROCEDURE OBTAINS ONE WORD 00005460
OF TEXT. WHEN THERE IS NO MORE TEXT THIS PROGRAM IS SIGNALLED 00005470
FROM GETR VIA THE FLG PARAMETER (SIDE PARAMETER), AND 00005480
00005490
00005500
00005510
00005520
00005530
00005540
00005550
00005560

THIS PROCEDURE RELINQUISHES CONTROL.

DESCRIPTION OF OUTPUT:

ALL OUTPUT FROM THIS PROCESS GOES ONTO A NEW WATFOR INPUT DATA SET VIA CALLS TO THE PUTR PROCEDURE. WHEN THE PROBABLE CORRECT SPELLING OF EACH INPUT WORD IS DISCOVERED, IT IS PASSED TO PUTR WHICH CREATES THE NEW DATA SET AND EDITS THE OUTPUT.

DESCRIPTION OF VARIABLES AND LABELS:

VARIABLE OR LABEL	MEANING	
ALGX,X=0,1,-1.	THE START OF THE X-TH SPELLING CORRECTION ALGORITHM.	00005570
CHARREG,DCREG,DREGC,REGCHAR.	CHARACTER REGISTERS.	00005580
CHARS.	STRING OF POSSIBLE INPUT CHARACTERS.	00005590
CNT.	NUMBER OF CHARACTERS IN THE WORD CURRENTLY UNDER ANALYSIS.	00005600
COPYINPUT,COPYOUTPUT,DEBUG,PRNT,PARMS.	EXTERNAL INPUT PARAMETERS.	00005610
DCNT.	THE LENGTH OF THE CURRENT DICTIONARY WORD.	00005620
DICTIONARY.	THE SYMBOL TABLE.	00005630
DICTLEN.	THE NUMBER OF WORDS IN THE DICTIONARY	00005640
DNO,DNI,GET,GIT,RTRN.	STATEMENT LABEL CONSTANTS.	00005650
FLG.	A FLAG WHICH INDICATES WHETHER OR NOT A CORRECTION HAS BEEN MADE.	00005660
		00005670
		00005680
		00005690
		00005700
		00005710
		00005720
		00005730
		00005740
		00005750
		00005760
		00005770
		00005780
		00005790
		00005800
		00005810
		00005820
		00005830
		00005840
		00005850
		00005860
		00005870
		00005880
		00005890
		00005900
		00005910
		00005920
		00005930

I,II,J,JJ.	COUNTERS ON LOOPS.	00005940
PRINT.	DEBUG FILE.	00005950
GETR,PUTR.	I/O MODULES.	00005960
REALGOOF.	LOCATION TO WHICH CONTROL IS PASSED	00005970
	WHEN NO CORRECTION CAN BE MADE.	00005980
SET,UNSET.	SUPPORTING I/O MODULES AND INTERFACE	00005990
	MODULES.	00006000
SIDE.	I/O CONDITION FLAG.	00006010
SPECIAL, SPECIALX.	THE SPECIAL CHARACTERS OF FORTRAN.	00006020
TEXT.	THE TEXT CURRENTLY BEING CORRECTED.	00006030
XCHARS.	FORTRAN CHARACTER SET.	00006040
XDIC.	THE SYMBOL TABLE.	00006050
		00006060
		00006070
		00006080
		00006090

REFERENCES:

-----		00006100
		00006110
ALBERGA, CYRIL N. STRING SIMILARITY AND MISSPELLINGS.		00006120
ASSOCIATION FOR COMPUTING MACHINERY COMMUNICATIONS 10:		00006130
302-313. 1967.		00006140
		00006150
BASTON, ALAN. THE ORGANIZATION OF SYMBOL TABLES. ASSOCIATION		00006160
FOR COMPUTING MACHINERY COMMUNICATIONS 8: 111-112. 1965.		00006170
		00006180
HOPCROFT, J. E. AND ULLMAN, J. D. ERROR CORRECTION FOR		00006190
FORMAL LANGUAGES. PRINCETON, NEW JERSEY. PRINCETON UNIVERSITY		00006200
BLAIR, C. R. A PROGRAM FOR CORRECTING SPELLING ERRORS.		00006210
INFORMATION AND CONTROL 3, NO. 3: 60-67. 1960.		00006220
		00006230
DAMEREAU, F. J. A TECHNIQUE FOR COMPUTER DETECTION AND		00006240
CORRECTION OF SPELLING ERRORS. ASSOCIATION FOR COMPUTING		00006250
MACHINERY COMMUNICATIONS 7: 171-176. 1964.		00006260
		00006270
FERGUSON, H. E. AND BERNER E. DEBUGGING SYSTEMS AT THE SOURCE		00006280
LANGUAGE LEVEL. ASSOCIATION FOR COMPUTING MACHINERY		00006290
COMMUNICATIONS 6: 430-432. 1963.		00006300

DEPARTMENT OF ELECTRICAL ENGINEERING DIGITAL SYSTEMS LABORATORY	00006310
TECHNICAL REPORT 52. AMES LABORATORY TECHNICAL REPORT	00006320
ALCTN-78. 1966.	00006330
	00006340
	00006350
MORGAN, HOWARD L. SPELLING CORRECTION IN SYSTEMS PROGRAMS.	00006360
ASSOCIATION FOR COMPUTING MACHINERY COMMUNICATIONS 13:	00006370
ASSOCIATION FOR COMPUTING MACHINERY COMMUNICATIONS 13:	00006380
90-94. 1970.	00006390
	00006400
SHERMAN, P. M. TABLE LOOK-AT TECHNIQUES. ASSOCIATION FOR	00006410
COMPUTING MACHINERY COMMUNICATIONS 4: 172-175. 1961.	00006420
	00006430
THORELLI, LARS ERIK. AUTOMATIC CORRECTION OF ERRORS IN TEXT.	00006440
NORDISK TIDSKRIFT FOR INFORMATIONS-BEHANDLING(BIT) 2: 46-62.	00006450
1962.	00006460
	00006470
	00006480
	00006490
	00006500
/* CREATE THE DICTIONARY. */	00006510
PUT FILE(PRINT) EDIT('ENTER CRRCTN')	00006520
(SKIP(DEBUG),A(13*DEBUG));	00006530
CALL SET;	00006540
IF DEBUG THEN ON STRG PUT FILE(PRINT) EDIT('***',TEXT,TSTR)	00006550
(SKIP,4 A(11));	00006560
DICTIONARY='';	00006570
CALL DFILL(DICTIONARY);	00006580
	00006590
	00006600
I=0;	00006610
PUT FILE(PRINT) EDIT('DICTIONARY IS SET UP.')	00006620
(SKIP(DEBUG),A(DEBUG*22));	00006630
GO TO GET;	00006640
	00006650
TEXT=TEXT TSTR;	00006660
CALL PUTR(TEXT,FLG);	00006670

*/

/*

*/

GIT:

```

GET:  FLG='0'B;                                00006680
      CALL GETR(TEXT,SIDE);                      00006690
      PUT FILE(PRINT) EDIT('T&S=',TEXT,SIDE)    00006700
          (SKIP(DEBUG),A(DEBUG*4),A(DEBUG*LENGTH(TEXT)),A(DEBUG)); 00006710
      TSTR=SUBSTR(TEXT,LENGTH(TEXT),1);          00006720
      TEXT=SUBSTR(TEXT,1,LENGTH(TEXT)-1);        00006730
      TXT=TEXT;                                  00006740
      IF SIDE='E' THEN GO TO RTRN;               00006750
      IF SIDE='L' THEN DO;                       00006760
          DICTLEN=DICTLEN+1;                     00006770
          DICTIONARY(DICTLEN)=TEXT;              00006780
          GO TO GIT;                             00006790
      END;                                       00006800
      IF LENGTH(TEXT)<3 THEN GO TO GIT;           00006810
      XDIC=STRING(DICTIONARY);                  00006820
      IF INDEX(XDIC,TEXT)~=0 THEN GO TO GIT;     00006830
      /* ACTUAL CORRECTION PROCESS FOLLOWS. */    00006840
      CNT=LENGTH(TEXT);                         00006850
      DO I=28 TO 3 BY -1;                       00006860
          REGCHAR(I)=INDEX(TEXT,CHARS(I))>0;    00006870
      END;                                       00006880
      /* FINISH TEXT CHARACTER REGISTER SETUP. */ 00006890
      DO I=0 TO 9 BY 1;                         00006900
          II=I;                                  00006910
          REGCHAR(2)=(REGCHAR(2))|(INDEX(TEXT,II)>0); 00006920
      END;                                       00006930
      /* CHECK FOR SPECIAL CHARACTERS IN TEXT WORD. */ 00006940
      DO I=1 TO DIM(SPECIAL,1);                 00006950
          REGCHAR(1)=(REGCHAR(1))|(INDEX(TEXT,SPECIAL(I))>0); 00006960
      END; /* OF TEXT CHARACTER REGISTER SETUP. */ 00006970
      /* REPEAT ABOVE PROCESS AS DICTIONARY SEARCH IS MADE. */ 00006980
      /*                                         00006990
      /*                                         00007000
      DO I=1 TO DICTLEN; /* START DICTIONARY SEARCH. */ 00007010
      DCNT=LENGTH(DICTIONARY(I));               00007020
      /* DICTIONARY CHARACTER REGISTER SETUP. */ 00007030
      /* ALPHABETIC TEST. */                   00007040

```

```

DO J=28 TO 3 BY -1;                                00007050
DREGC(J)=INDEX(DICTIONARY(I),CHARS(J))>0;          00007060
END; /* ALPHABETIC TEST */                          00007070
/* DIGIT TEST. */                                    00007080
DO J=0 TO 9 BY 1;                                    00007090
II=J;                                                 00007100
DREGC(2)=(DREGC(2))|(INDEX(DICTIONARY(I),II)>0);    00007110
END; /* DIGIT TEST. */                              00007120
DO J=1 TO DIM(SPECIAL,1);                            00007130
DREGC(1)=(DREGC(1))|(INDEX(DICTIONARY(I),SPECIAL(J))>0); 00007140
END; /* CHARACTER REGISTER SETUP. */                00007150
/*                                                    00007160
*/                                                    00007170
IF (I=1)|(I=127)|(I=255) THEN                      00007180
PUT FILE(PRINT) EDIT('DEBUG CRRCTN',DCNT,CNT,DCNT-CNT,'CC') 00007190
(SKIP(DEBUG),A(DEBUG*20),3 F(DEBUG*6),A(DEBUG*20));    00007200
IF ABS(DCNT-CNT)>=2 THEN GO TO REALGOOF;              00007210
IF SUM((REGCHAR&DREGC))<2 THEN GO TO REALGOOF;        00007220
GO TO ALG(DCNT-CNT);                                00007230
/*                                                    00007240
*/                                                    00007250
ALG(-1):                                             00007260
/* ALGORITHM -1; TEXT ENTRY LONGER THAN DICTIONARY ENTRY. */ 00007270
/* IN ORDER TO USE ALGORITHM -1 THE TEXT ENTRY WORD MUST BE 00007280
ONE CHARACTER LONGER THAN THE DICTIONARY WORD. THIS ALGORITHM 00007290
DISCARDS THE FIRST DIFFERENCE CHARACTER OF THE TEXT WORD AND 00007300
SHIFTS THE REMAINING CHARACTERS LEFT ONE POSITION. AN EQUALITY 00007310
COMPARE IS MADE AND IF IT IS SUCCESSFUL THE NEW WORD IS USED; 00007320
OTHERWISE THE DICTIONARY SEARCH CONTINUES. */         00007330
IF (I=1)|(MOD(I,25)=0) THEN                          00007340
PUT FILE(PRINT) EDIT('ENTER ALG-1')                 00007350
(SKIP(DEBUG),A(DEBUG*12));                            00007360
DO J=1 TO DCNT;                                       00007370
IF SUBSTR(TEXT,J,1)~=SUBSTR(DICTIONARY(I),J,1) THEN DO; 00007380
DO JJ=J TO CNT-1;                                    00007390
SUBSTR(TEXT,JJ,1)=SUBSTR(TEXT,JJ+1,1);              00007400
END;                                                  00007410

```

GO TO DNO;	00007420
END;	00007430
END;	00007440
DNO: TEXT=SUBSTR(TEXT,1,CNT-1);	00007450
/* END OF ALGORITHM -1. */	00007460
ALG(0):	00007470
/* ALGORITHM 0: NUMBER OF CHARACTERS ARE EQUAL. */	00007480
/* ALGORITHM 0 REQUIRES THAT THE INPUT TEXT WORD AND THE	00007490
DICTIONARY WORD HAVE THE SAME NUMBER OF CHARACTERS. IF THE	00007500
TWO WORDS DIFFER IN EXACTLY ONE POSITION THE DICTIONARY ENTRY	00007510
IS ACCEPTED AS THE CORRECT WORD. IF TWO ADJACENT POSITIONS	00007520
DIFFER THEN THE CHARACTERS IN THESE TWO POSITIONS ARE	00007530
INTERCHANGED AND COMPARISON PROCEEDS AS ABOVE. */	00007540
IF (I=1) (MOD(I,25)=0) THEN	00007550
PUT FILE(PRINT) EDIT('ENTER ALG 0')	00007560
(SKIP(DEBUG),A(DEBUG*12));	00007570
IF SUM((DREGC&(1-REGCHAR)))<=1 THEN TEXT=DICTIONARY(I);	00007580
ELSE DO J=1 TO DCNT-1 WHILE	00007590
((TEXT<=>DICTIONARY(I))&	00007600
(LENGTH(TEXT)=LENGTH(DICTIONARY(I))));	00007610
IF SUBSTR(TEXT,J,1)<=>SUBSTR(DICTIONARY(I),J,1) THEN	00007620
DO; /* CHARACTER INVERSION. */	00007630
SIDE=SUBSTR(TEXT,J,1);	00007640
SUBSTR(TEXT,J,1)=SUBSTR(TEXT,J+1,1);	00007650
SUBSTR(TEXT,J+1,1)=SIDE;	00007660
PUT FILE(PRINT) EDIT(TEXT,DICTIONARY(I))	00007670
(SKIP(DEBUG),A(DEBUG*LENGTH(TEXT)),A(DCNT*DEBUG));	00007680
END;	00007690
END;	00007700
GO TO REALGOOF;	00007710
/* END OF ALGORITHM 0. */	00007720
ALG(1):	00007730
/* ALGORITHM 1: DICTIONARY ENTRY LONGER. */	00007740
/* ALGORITHM 1 IS USED WHEN THE DICTIONARY WORD IS ONE	00007750
CHARACTER LONGER THAN THE INPUT TEXT WORD. A TEMPORARY WORKING	00007760
VARIABLE IS USED TO HOLD THE DICTIONARY WORD. THE FIRST	00007770
DIFFERENCE CHARACTER OF THE DICTIONARY WORD IS DISCARDED AND	00007780

THE REMAINING CHARACTERS ARE SHIFTED LEFT ONE POSITION.	FROM	00007790
THIS POINT THE ALGORITHM PROCEEDS AS ABOVE. */		00007800
IF (I=1) (MOD(I,25)=0) THEN		00007810
PUT FILE(PRINT) EDIT('ENTER ALG 1')		00007820
(SKIP(DEBUG),A(DEBUG*12));		00007830
XDIC=DICTIONARY(I);		00007840
DO J=1 TO DCNT;		00007850
IF SUBSTR(TEXT,J,1)≠SUBSTR(XDIC,J,1) THEN DO;		00007860
DO JJ=J TO DCNT-1;		00007870
SUBSTR(XDIC,JJ,1)=SUBSTR(XDIC,JJ+1,1);		00007880
END;		00007890
GO TO DN1;		00007900
END;		00007910
END;		00007920
DN1: TEXT=SUBSTR(XDIC,1,DCNT-1);		00007930
GO TO ALG(0);		00007940
/* END OF ALGORITHM 1. */		00007950
REALGOOF:		00007960
TEXT=TXT;		00007970
END; /* DICTIONARY SEARCH. */		00007980
GO TO GIT;		00007990
RTRN:		00008000
PUT FILE(PRINT) EDIT('ABOUT TO CALL UNSET, I=',I)		00008010
(SKIP(DEBUG),A(DEBUG*24),F(DEBUG*5));		00008020
CALL UNSET;		00008030
RETURN;		00008040
END CRRCTN;		00008050

```

/*          MASTER DYNAMIC CORRECTION MODULE          */
(SUBRG,STRG,SIZE):
CRRCTR: PROC(CARD,XLINE) CHAR(80) VAR;
DCL      (MARKER INIT((080) '*'),
          CMMT1 INIT ('C      ERROR CORRECTION ATTEMPTED IN FOLLOWING CAR
D      (      )      '),
          CMMT2 INIT ('C      ORIGINAL CARD WAS
                        '))
) STATIC CHAR(80),
CMMT3 STATIC CHAR(80),
1 PARMS STATIC UNALIGNED EXT,
2 (DEBUG,Z1,Z2,Z3) BIT(1),
CARD CHAR(*),
(BAD29 INIT('#@%$<_ |&>:;-''?' ' '),
GOOD29 INIT(',W,*.WRYP$DFGHXCB')) STATIC CHAR(17),
(OPERR1 INIT('0125Q81'),
OPERR2 INIT('01ZSOBL')) STATIC CHAR(7),
LETS CHAR(19) STATIC INIT('ABCDEFGHIJKLMPSUVX'),
ERRSET(0:19) LABEL,
XLINE CHAR(*),
LINE DEF XLINE POS(2) CHAR(120),
CRD(LENGTH(CARD)) CHAR(1) DEF CARD POS(1),
G29(17) CHAR(1) DEF GOOD29 POS(1),
COLS105 CHAR(5) DEF CARD POS(1),
BLANKS CHAR(5) STATIC INIT(' '),
ALLBLANKS CHAR(LENGTH(CARD)),
ECD CHAR(4) STATIC,
(NEWPGM,PRINT) FILE EXT,
I FIXED BIN(15,0),
OPS CHAR(04) STATIC INIT('+-*/' ),
(KT,TC1) FIXED BIN(15,0),
TMAT(0:5,0:4) FIXED BIN(15,0) EXT STATIC,
ACTION ENTRY(FIXED BIN(15,0)),
TEMPCH FIXED BIN(15,0),
ISX0 PICTURE '9',
TSX3 CHAR(2),
TKT FIXED BIN(15,0) INIT(0),

```

```

*/ 00008070
00008080
00008090
00008100
00008110
00008120
00008130
00008140
00008150
00008160
00008170
00008180
00008190
00008200
00008210
00008220
00008230
00008240
00008250
00008260
00008270
00008280
00008290
00008300
00008310
00008320
00008330
00008340
00008350
00008360
00008370
00008380
00008390
00008400
00008410
00008420
00008430

```


CHST(66) CHAR(1) DEF CARD POS(7),
NCARD CHAR(LENGTH(CARD)) VAR;

/*

IDENTIFICATION:

PROGRAM-ID: CRRCTR.
AUTHOR: G. E. HEDRICK.
INSTALLATION: IOWA STATE UNIVERSITY COMPUTATION CENTER.
DATE-WRITTEN: 6 FEBRUARY 1970.
SOURCE-COMPUTER: IBM 360/65.
OBJECT-COMPUTER: IBM 360/65.
OPERATING SYSTEM: OS/MVT.
MEMORY SIZE: DEPENDS ON OTHER CORRECTOR PROCEDURES.

DESCRIPTION OF THE PROBLEM:

THIS PROGRAM IS DESIGNED TO CORRECT SOURCE ERRORS IN
FORTRAN PROGRAMS. THIS MODULE ATTEMPTS TO CORRECT THOSE
ERRORS WHICH ARE FLAGGED AT THE POINT OF OCCURRENCE. THE
ERROR MESSAGE LINE IMMEDIATELY FOLLOWS THE COPY OF THE INPUT
CARD WHICH CONTAINS THE SOURCE ERROR. THE TYPE OF ERROR
MUST BE DETERMINED FROM THE MESSAGE LINE SO THAT APPROPRIATE
ACTION MAY BE TAKEN. EACH CLASS OF ERROR HAS A DIFFERENT
CORRECTION PROCESS SO SOME SUBDIVISION IS REQUIRED.

METHOD OF SOLUTION:

THE PROGRAMMED SOLUTION FOR THE PART OF THE OVERALL
ERROR CORRECTION PROBLEM WHICH THIS MODULE REPRESENTS IS

00008440
00008450
00008460
00008470
00008480
00008490
00008500
00008510
00008520
00008530
00008540
00008550
00008560
00008570
00008580
00008590
00008600
00008610
00008620
00008630
00008640
00008650
00008660
00008670
00008680
00008690
00008700
00008710
00008720
00008730
00008740
00008750
00008760
00008770
00008780
00008790
00008800

RELATIVELY STRAIGHTFORWARD. THE SOLUTION CONSISTS OF THE
 FOLLOWING STEPS:

1. CREATE ERROR FLAG STRINGS;
2. WRITE THE ERROR FLAG STRINGS AS CARD IMAGES;
3. INVOKE THE KEYPUNCH CORRECTION MODULE;
4. CLASSIFY THE ERROR(S);
5. INVOKE THE APPROPRIATE INDIVIDUAL CORRECTION MODULE;
6. PERFORM THE TERMINAL ANALYSIS;
7. RETURN TO THE CALLING PROCEDURE.

NOTE THAT EACH OF THE INDIVIDUAL CORRECTION MODULES HAS A ONE
 LINE DESCRIPTION AT THE POINT WHERE IT OBTAINS CONTROL.

DESCRIPTION OF INPUT:

THERE ARE TWO INPUT PARAMETERS FOR THIS PROCEDURE. THESE
 TWO PARAMETERS ARE THE VALUES IN CARD AND IN XLINE. THERE IS
 NO OTHER INPUT TO THIS MODULE. CARD CONTAINS THE IMAGE OF THE
 CARD WHICH IS IN ERROR AND XLINE CONTAINS THE IMAGE OF THE
 ERROR MESSAGE LINE.

DESCRIPTION OF OUTPUT:

THERE ARE TWO CLASSES OF OUTPUT FROM THIS PROCEDURE. THE
 FIRST CLASS OF OUTPUT IS COMMENT OUTPUT. COMMENT OUTPUT IS
 PLACED DIRECTLY ON THE TEMPORARY NEW PROGRAM OUTPUT FILE.
 COMMENT OUTPUT CONSISTS OF THOSE COMMENTS WHICH ARE PLACED
 IN THE FORTRAN PROGRAM TO CALL THE PROGRAMMER'S ATTENTION TO
 AN ATTEMPTED CORRECTION. THE SECOND CLASS OF OUTPUT IS THE NEW
 SOURCE STATEMENT OUTPUT. NEW SOURCE STATEMENT OUTPUT IS
 RETURNED TO THE CALLING PROCEDURE TO BE PLACED IN THE TEMPORARY
 NEW PROGRAM OUTPUT FILE. NEW SOURCE STATEMENT OUTPUT INCLUDES

00008810
 00008820
 00008830
 00008840
 00008850
 00008860
 00008870
 00008880
 00008890
 00008900
 00008910
 00008920
 00008930
 00008940
 00008950
 00008960
 00008970
 00008980
 00008990
 00009000
 00009010
 00009020
 00009030
 00009040
 00009050
 00009060
 00009070
 00009080
 00009090
 00009100
 00009110
 00009120
 00009130
 00009140
 00009150
 00009160
 00009170

THE CORRECTED SOURCE PROGRAM STATEMENTS.

DESCRIPTION OF VARIABLES AND LABELS:

VARIABLE OR LABEL

MEANING

ALLBLANKS, BLANKS.

BAD29.

CARC, CRD.

CMMT1, CMMT2, MARKER.

DEBUG.

ECD.

ERRSET.

G29, GOOD29.

I.

LETS.

LINE, XLINE.

NCARD.

NEWPGM.

PARMS.

PRINT.

RELINQUISH.

Z1, Z2, Z3.

VARIABLES WHICH CONSISTS ENTIRELY
OF BLANKS; IT IS USED FOR TESTING.
PROBABLE MISSPUNCHED CHARACTERS WHEN
THE FORTRAN PROGRAM HAS BEEN PUNCHED
WITH AN IBM 029 KEYPUNCH.
IMAGE OF THE CARD IN ERROR.
COMMENTS WHICH ARE TO BE PLACED IN
THE NEW FORTRAN SOURCE PROGRAM.
EXTERNAL PARAMETER.
A COPY OF THE ERROR CODE.
A LABEL ARRAY; EACH DIFFERENT ERRSET
IS THE START OF A SUBORDINATE
CORRECTION MODULE.
THE SET OF POSSIBLE CORRECTIONS FOR
THE ERRORS IN BAD29.
A COUNTER WHICH IS USED THROUGHOUT
THE PROGRAM.
THE FIRST LETTER OF ERROR CODES.
LINE IMAGE OF THE MESSAGE LINE.
THE NEW CARD IMAGE.
THE TEMPORARY NEW PROGRAM OUTPUT
FILE.
EXTERNAL PARAMETER ARRAY.
DEBUG OUTPUT FILE.
THE POINT WHERE THIS MODULE RETURNS
CONTROL TO THE CALLING MODULE.
FILLERS IN PARMS.

00009180
00009190
00009200
00009210
00009220
00009230
00009240
00009250
00009260
00009270
00009280
00009290
00009300
00009310
00009320
00009330
00009340
00009350
00009360
00009370
00009380
00009390
00009400
00009410
00009420
00009430
00009440
00009450
00009460
00009470
00009480
00009490
00009500
00009510
00009520
00009530
00009540

COLS1T05.

THE FIRST FIVE COLUMNS OF THE CARD
IMAGE.

IDENTIFICATION OF THE ERRSET MODULES:

MODULE NUMBER

MODULE NAME

00	UNIDENTIFIED ERROR MODULE.
01	ASSIGN STATEMENTS AND VARIABLES MODULE.
02	BLOCK DATA STATEMENTS MODULE.
03	CARD FORMAT AND CONTENTS MODULE.
04	DATA/DIMENSION STATEMENT MODULE.
05	EQUIVALENCE-COMMON/EXTERNAL MODULE.
06	FORMAT/FUNCTION MODULE.
07	GO-TO MODULE.
08	HOLLERITH MODULE.
09	IF-I/O MODULE.
10	JCL MODULE.
11	DEPENDENT EXECUTION FLAG MODULE.
12	LOGICAL/LIBRARY MODULE.
13	MODE MODULE.
14	PARENTHESIS MODULE.
15	RETURN STATEMENT MODULE.
16	MULTIPLE ERROR MODULE.
17	UNDEFINED OPERATIONS/VARIABLES MODULE.
18	VARIABLE NAME MODULE.
19	EXTERNAL STATEMENT MODULE.

REFERENCES:

00009550
00009560
00009570
00009580
00009590
00009600
00009610
00009620
00009630
00009640
00009650
00009660
00009670
00009680
00009690
00009700
00009710
00009720
00009730
00009740
00009750
00009760
00009770
00009780
00009790
00009800
00009810
00009820
00009830
00009840
00009850
00009860
00009870
00009880
00009890
00009900
00009910

BENNET, RICHARD K. THE DESIGN OF COMPUTER LANGUAGES AND SOFTWARE SYSTEMS: A BASIC APPROACH. COMPUTERS AND AUTOMATION 18, NO. 2: 28-33. 1969.	00009920 00009930 00009940 00009950
BLATT, JOHN M. INTRODUCTION TO FORTRAN IV PROGRAMMING: USING THE WATFOR COMPILER. PACIFIC PALISADES, CALIFORNIA. GOODYEAR PUBLISHING COMPANY. C1968.	00009960 00009970 00009980 00009990
FLOYD, ROBERT W. AN ALGORITHM DEFINING ALGOL ASSIGNMENT STATEMENTS. ASSOCIATION FOR COMPUTING MACHINERY COMMUNICATIONS 3: 170-171. 1960.	00010000 00010010 00010020 00010030
FREEMAN, DAVID N. ERROR CORRECTION IN CORC, THE CORNELL UNIVERSITY COMPUTING LANGUAGE. AMERICAN FEDERATION OF INFORMATION PROCESSING SOCIETIES FALL JOINT COMPUTER CONFERENCE PROCEEDINGS 1964 (PART 1): 15-34. BALTIMORE, MARYLAND. SPARTAN BOOKS, INC. C1964.	00010040 00010050 00010060 00010070 00010080 00010090
GHANDOUR, ZIAD JAMIL. FORMAL SYSTEMS AND SYNTACTICAL ANALYSIS. XEROX COPY. DOCTOR OF PHILOSOPHY DISSERTATION. NEW HAVEN, CONNECTICUT. LIBRARY, YALE UNIVERSITY. 1968. ANN ARBOR, MICHIGAN. UNIVERSITY MICROFILMS 68-14,844. C1968.	00010100 00010110 00010120 00010130 00010140
GRIES, D. USE OF TRANSLATION MATRICES IN COMPILING. ASSOCIATION FOR COMPUTING MACHINERY COMMUNICATIONS 11: 26-34. 1968.	00010150 00010160 00010170 00010180
HOPCROFT, J. E. AND ULLMAN, J. E. ERROR CORRECTION FOR FORMAL LANGUAGES. PRINCETON, NEW JERSEY. PRINCETON UNIVERSITY DEPARTMENT OF ELECTRICAL ENGINEERING DIGITAL SYSTEMS LABORATORY TECHNICAL REPORT 52. AMES LABORATORY TECHNICAL REPORT ALCTN-78. 1966.	00010190 00010200 00010210 00010220 00010230 00010240
IRONS, EDGAR T. AN ERROR CORRECTING PARSE ALGORITHM. ASSOCIATION FOR COMPUTING MACHINERY COMMUNICATIONS 6: 669-673. 1963.	00010250 00010260 00010270 00010280

MORRIS, FRANK. RANDOM INPUT SEARCH TECHNIQUES: A TIMELY
REVIEW. SOFTWARE AGE 4, NO. 1: 8-12. 1970.

*/

ACTION: PROC(TYP);

/* THIS PROCEDURE CONTAINS THE ACTIONS WHICH ARE SPECIFIED BY
THE ACTION MATRIX. SINCE THE ACTION MATRIX HAS BLANKS WHERE
THE TRANSLATION MATRIX HAS ENTRIES AND VICE VERSA, THE TWO
MATRICES ARE COMBINED IN ARRAY TMAT.

*/

DCL ACTNO(6:20) LABEL,(TYP,TC2) FIXED BIN(15,0);
GO TO ACTNO(TYP);

/*

*/

ACTNO(6):

ACTNO(12):

TC2=INDEX(OPERR1,CHST(KT));

IF TC2=0 THEN CHST(KT)=SUBSTR(OPERR2,TC2,1);

ELSE CHST(KT)='I';

TYP=1;

GO TO CONT;

ACTNO(7):

ACTNO(8):

CRD(1)='C';

TYP=6;

GO TO CONT;

ACTNO(9):

CHST(KT)='=';

TYP=2;

GO TO CONT;

ACTNO(10):

TC2=INDEX(OPERR1,CHST(KT));

IF TC2=0 THEN CHST(KT)=SUBSTR(OPERR2,TC2,1);

ELSE CHST(KT)='I';

TYP=3;

GO TO CONT;

00010290
00010300
00010310
00010320
00010330
00010340
00010350
00010360
00010370
00010380
00010390
00010400
00010410
00010420
00010430
00010440
00010450
00010460
00010470
00010480
00010490
00010500
00010510
00010520
00010530
00010540
00010550
00010560
00010570
00010580
00010590
00010600
00010610
00010620
00010630
00010640
00010650

```

ACTNO(11):
CARD=SUBSTR(CARD,1,KT+6)||'('||SUBSTR(CARD,KT+7);
CRD(72)=')';
TYP=6;
GO TO CONT;
ACTNO(13):
CARD=SUBSTR(CARD,1,KT+5)||'*'||SUBSTR(CARD,KT+7);
TYP=6;
GO TO CONT;
ACTNO(14):
ACTNO(15):
TC2=INDEX(OPERR2,CHST(KT));
IF TC2=0 THEN CHST(KT)=SUBSTR(OPERR1,TC2,1);
ELSE CHST(KT)='1';
TYP=4;
GO TO CONT;
ACTNO(16):
CARD=SUBSTR(CARD,1,KT+5)||SUBSTR(CARD,KT+7);
TYP=5;
GO TO CONT;
ACTNO(17):
CARD=SUBSTR(CARD,1,KT+6)||'+00'||SUBSTR(CARD,KT+7);
TYP=4;
KT=KT+3;
GO TO CONT;
ACTNO(18):
ACTNO(19):
ACTNO(20):
CONT:
RETURN;
END; /* OF CORRECTIVE ACTION ROUTINE. */

```

```

/*
*/

```

```

PUT FILE(PRINT) EDIT('ENTER CRRCTR WITH CARD=',CARD,
'AND XLINE=',LINE)
(SKIP(DEBUG),A(DEBUG*24),SKIP(DEBUG),A(DEBUG*80),
SKIP(DEBUG),A(DEBUG*11),

```

```

00010660
00010670
00010680
00010690
00010700
00010710
00010720
00010730
00010740
00010750
00010760
00010770
00010780
00010790
00010800
00010810
00010820
00010830
00010840
00010850
00010860
00010870
00010880
00010890
00010900
00010910
00010920
00010930
00010940
00010950
00010960
00010970
00010980
00010990
00011000
00011010
00011020

```

SKIP(DEBUG),A(DEBUG*120));	00011030
SUBSTR(MARKER,1,1)='C';	00011040
SUBSTR(CMMT1,62,4)=SUBSTR(XLINE,18,4);	00011050
WRITE FILE(NEWPGM) FROM(MARKER);	00011060
WRITE FILE(NEWPGM) FROM(CMMT1);	00011070
WRITE FILE(NEWPGM) FROM(CMMT2);	00011080
CMMT3=CARD;	00011090
SUBSTR(CMMT3,1,1)='C';	00011100
PUT FILE(PRINT) EDIT('CMMT3=',CMMT3)(SKIP(DEBUG),	00011110
A(10*DEBUG),A(80*DEBUG));	00011120
WRITE FILE(NEWPGM) FROM(CMMT3);	00011130
/* INNER DYNAMIC CORRECTION MODULES. */	00011140
/* KEYPUNCH CORRECTION MODULE. */	00011150
PUT FILE(PRINT) EDIT('ENTER KEYPUNCH CRRCTR MODULE.')	00011160
(SKIP(DEBUG),A(DEBUG*35));	00011170
DO I=1 TO 72;	00011180
IF INDEX(BAD29,CRD(I))=0 THEN	00011190
CRD(I)=G29(INDEX(BAD29,CRD(I)));	00011200
END;	00011210
ECD=SUBSTR(XLINE,18,4);	00011220
GO TO ERRSET(INDEX(LETS,SUBSTR(XLINE,18,1)));	00011230
/* INDIVIDUAL ERROR CORRECTION MODULES BY ERRSET NUMBER. */	00011240
ERRSET(01):	00011250
ERRSET(02):	00011260
GO TO RELINQUISH;	00011270
ERRSET(03):	00011280
/* CARD FORMAT AND CONTENTS MODULE. */	00011290
IF SUBSTR(ECD,1,2)='CC' THEN DO;	00011300
IF COLS1TO5=BLANKS THEN DO;	00011310
SUBSTR(CARD,7,72)=SUBSTR(CARD,6,72);	00011320
CRD(6)='';	00011330
END;	00011340
ELSE COLS1TO5=BLANKS;	00011350
IF CARD=ALLBLANKS THEN CARD='';	00011360
GO TO RELINQUISH;	00011370
END; /* OF CC MODULE. */	00011380
/* ILLEGAL CONSTANT MODULE. */	00011390

IF ECD='CN-6' THEN DO;	00011400
TSX3=SUBSTR(XLINE,35,1) SUBSTR(XLINE,44,1);	00011410
I=INDEX(CARD,TSX3);	00011420
(NOSUBRG): IF (I=0)&(SUBSTR(XLINE,44,1)<='Z')&(CRD(I)='.') THEN CRD(I)=',';	00011430
END; /* OF ILLEGAL CONSTANT MODULE. */	00011440
GO TO RELINQUISH;	00011450
ERRSET(04):	00011460
ERRSET(05):	00011470
ERRSET(06):	00011480
ERRSET(07):	00011490
ERRSET(08):	00011500
ERRSET(09):	00011510
ERRSET(10):	00011520
GO TO RELINQUISH;	00011530
ERRSET(11):	00011540
/* DEPENDENT EXECUTION FLAG MODULE. */	00011550
IF CRD(1)~='/' THEN CARD=' ';	00011560
IF ECD='KO-0' THEN GO TO RELINQUISH;	00011570
GO TO RELINQUISH;	00011580
ERRSET(12):	00011590
ERRSET(13):	00011600
ERRSET(14):	00011610
ERRSET(15):	00011620
GO TO RELINQUISH;	00011630
ERRSET(16):	00011640
/* MULTIPLE ERROR MODULE. */	00011650
/* SELECTED SYNTAX ERROR MODULE 1. */	00011660
/* DOES NOT OPERATE FOR IF-STATEMENTS. */	00011670
PUT FILE(PRINT) EDIT('ENTER MULTIPLE ERROR MODULE')	00011680
(SKIP(DEBUG),A(DEBUG*36));	00011690
IF INDEX(CARD,'IF(')=0 THEN DO;	00011700
PUT FILE(PRINT) EDIT('ENTER SX MODULE 1,TKT=',TKT)	00011710
(SKIP(DEBUG),A(DEBUG*36),F(DEBUG*6));	00011720
IF DEBUG THEN ON SIZE SNAP BEGIN; (NOSIZE):	00011730
PUT FILE(PRINT) EDIT(TMAT,TEMPCH) (5 F(5,0));	00011740
END;	00011750
	00011760

IF ECD='SX-0' THEN DO I=0 TO 9;	00011770
ISX0=I;	00011780
KT=INDEX(CARD,ISX0 '(');	00011790
TKT=TKT+1;	00011800
IF KT=0 THEN CARD=SUBSTR(CARD,1,KT) '*' SUBSTR(CARD,KT+1);	00011810
KT=INDEX(CARD,'') ISX0);	00011820
TKT=TKT+1;	00011830
IF KT=0 THEN CARD=SUBSTR(CARD,1,KT) '*' SUBSTR(CARD,KT+1);	00011840
END;	00011850
IF TKT>0 THEN GO TO RELINQUISH;	00011860
/* SELECTED SYNTAX ERROR MODULE 2 */	00011870
IF ECD='SX-0' THEN DO KT=1 TO 66 BY 1;	00011880
IF(CHST(KT)>='A')&(CHST(KT)<='Z') THEN TC1=0;	00011890
IF(CHST(KT)>='0') THEN TC1=1;	00011900
IF INDEX(OPS,CHST(KT))=0 THEN TC1=4;	00011910
IF((TEMPCH=4) (TEMPCH=5))&(CHST(KT)='E') THEN TC1=3;	00011920
PUT FILE(PRINT) EDIT(TEMPCH) (SKIP(DEBUG),F(DEBUG*3));	00011930
TEMPCH=TMAT(TEMPCH,TC1);	00011940
PUT FILE(PRINT) EDIT(TEMPCH) (SKIP(DEBUG),F(DEBUG*3));	00011950
IF TEMPCH>5 THEN CALL ACTION(TEMPCH);	00011960
IF TEMPCH>5 THEN GO TO RELINQUISH;	00011970
END;	00011980
IF ECD='SX-0' THEN	00011990
GO TO RELINQUISH;	00012000
END; /* SX-0 SUB-MODULE. */	00012010
/* SUPERFLUOUS OR BAD CHARACTER MODULE. */	00012020
IF ECD='SX-3' THEN DO;	00012030
TSX3=SUBSTR(XLINE,35,1) SUBSTR(XLINE,53,1);	00012040
I=INDEX(CARD,TSX3);	00012050
IF (CRD(I)='-')&(I=0) THEN CRD(I)='0';	00012060
END; /* OF SUPERFLUOUS CHARACTER MODULE. */	00012070
/* STATEMENT AND STATEMENT NUMBER MODULE. */	00012080
PUT FILE(PRINT) EDIT('ENTER STATEMENT AND NUMBER MODULE')	00012090
(SKIP(DEBUG),A(DEBUG*36));	00012100
IF ECD='ST-1' THEN CRD(6)=' ';	00012110
IF (ECD='ST-5') (ECD='ST-4') THEN CRD(1)='C';	00012120
ON SIZE SYSTEM;	00012130

/* BAD STATEMENT NUMBER MODULE. */	00012140
IF ECD='ST-A' THEN DO;	00012150
CARD=SUBSTR(XLINE,25,5) 'CONTINUE';	00012160
CMMT3=CARD;	00012170
WRITE FILE(NEWPGM) FROM(CMMT3);	00012180
END; /* OF BAD STATEMENT NUMBER MODULE. */	00012190
GO TO RELINQUISH;	00012200
ERRSET(17):	00012210
ERRSET(18):	00012220
ERRSET(19):	00012230
ERRSET(00):	00012240
RELINQUISH:	00012250
/* TERMINAL ANALYSIS MODULE. */	00012260
NCARD=CARD;	00012270
PUT FILE(PRINT) EDIT('LEAVE CRRCTR WITH NCARD',NCARD)	00012280
(SKIP(DEBUG),A(26*DEBUG),SKIP(DEBUG),A(80*DEBUG));	00012290
/* END OF DYNAMIC INNER CORRECTION MODULES. */	00012300
RETURN(NCARD);	00012310
END CRRCTR;	00012320

```

/*          STREAM INPUT MODULE FOR ERROR CORRECTION.  */
(SUBRG,STRG,SIZE):
GETR:  PROC(TXT,FLG) CPTIONS(REENTRANT) RECURSIVE;
DCL    TXT CHAR(11) VAR,
        FLG CHAR(1),
        1 PARMS STATIC EXT UNALIGNED,
        2 (DEBUG,Z1,Z2,Z3) BIT(1),
        1 I_O UNALIGNED STATIC EXT,
        2 (K INIT(0),L INIT(1), M INIT(7), N INIT(0)) FIXED BIN(15,0),
        2 BUF CHAR(80),
        2 DELIMS CHAR(12) INIT(' +-*/( )''.=;, '),
        2 OUTREC CHAR(80) INIT((80) ' '),
        CARD(80) CHAR(1) DEF BUF,
        BUFR CHAR(80),
        BUF72 CHAR(72) DEF BUF,
        BUF6 CHAR(6) DEF BUF,
        BUF73 CHAR(1) DEF BUF POS(73),
        BUF1 CHAR(1) DEF BUF,
        DELIM(12) CHAR(1) DEF DELIMS,
        ALLBLANKS CHAR(72) STATIC INIT((72) ' '),
        FLGR BIT(1),
        MM FIXED BIN(15,0),
        (I,J,II) FIXED BIN(15,0),
        (PRINT,NEWPGM,PGMNEW) FILE EXT,
        PUTR ENTRY(CHAR(11) VAR,BIT(1)),
        INPT(0:1) LABEL;

```

/*

IDENTIFICATION:

PROGRAM-ID: GETR.
 AUTHOR: G. E. HEDRICK.
 INSTALLATION: IOWA STATE UNIVERSITY COMPUTATION CENTER.
 DATE-WRITTEN: 26 FEBRUARY 1970.

00012340
 00012350
 00012360
 00012370
 00012380
 00012390
 00012400
 00012410
 00012420
 00012430
 00012440
 00012450
 00012460
 00012470
 00012480
 00012490
 00012500
 00012510
 00012520
 00012530
 00012540
 00012550
 00012560
 00012570
 00012580
 00012590
 00012600
 00012610
 00012620
 00012630
 00012640
 00012650
 00012660
 00012670
 00012680
 00012690
 00012700

SOURCE-COMPUTER: IBM 360/65.
 OBJECT-COMPUTER: IBM 360/65.
 OPERATING SYSTEM: OS/MVT.
 MEMORY SIZE: DEPENDS ON OTHER CORRECTOR PROCEDURES.

PURPOSE:

 THIS PROGRAM SIMULATES STREAM OUTPUT FROM A RECORD INPUT
 FILE. THE RECORD INPUT FILE IS FILE NEWPGM. INPUT IS ASSUMED
 TO BE FORTRAN STATEMENTS. DELIMITERS IN THE INPUT STRING ARE
 THE STANDARD FORTRAN OPERATORS. NEW INPUT IS RETURNED VIA
 THE OUTPUT PARAMETER TXT.

DESCRIPTION OF VARIABLES AND LABELS:

VARIABLE OR LABEL	DESCRIPTION	
-----	-----	
ALLBLANKS	A BLANK CHARACTER STRING WHICH IS	00012710
	USED IN TESTING.	00012720
BUF,BUFR,CARD.	INPUT BUFFERS.	00012730
BUF6,BUF72,BUG73.	WORKING PORTIONS OF THE BUFFERS.	00012740
DEBUG.	EXTERNAL DEBUG FLAG.	00012750
DELIM,DELIMS.	FORTAN DELIMITERS.	00012760
FLG.	RETURN PARAMETER; CONTAINS L,R,N OR	00012770
	E.	00012780
FLGR.	SPACE HOLDER.	00012790
I,II,J,K,L,M,MM,N.	RECORD COUNTERS AND WORK VARIABLES	00012800
	FOR I/O.	00012810
I_O.	COMMUNICATION STRUCTURE FOR I/O.	00012820
INPT.	LABEL ARRAY. TRANSFERS ARE DEPENDENT	00012830
		00012840
		00012850
		00012860
		00012870
		00012880
		00012890
		00012900
		00012910
		00012920
		00012930
		00012940
		00012950
		00012960
		00012970
		00012980
		00012990
		00013000
		00013010
		00013020
		00013030
		00013040
		00013050
		00013060
		00013070

NEWPGM.	UPON THE NEED FOR NEW DATA.	00013080
OUTREC.	RECORD INPUT FILE.	00013090
PARMS.	OUTPUT BUFFER.	00013100
PGMNEW.	EXTERNAL PARAMETER STRUCTURE.	00013110
PRINT.	OUTPUT FILE.	00013120
PUTR.	DEBUG FILE.	00013130
RTRN.	OUTPUT ROUTINE.	00013140
TXT.	LAST LOCATION PRIOR TO RETURN.	00013150
Z1,Z2,Z3.	INPUT STRING.	00013160
	FILLERS.	00013170
		00013180
		00013190
		00013200
		00013210
		00013220
		00013230
		00013240
		00013250
		00013260
		00013270
		00013280
		00013290
		00013300
		00013310
		00013320
		00013330
		00013340
		00013350
		00013360
		00013370
		00013380
		00013390
		00013400
		00013410
		00013420
		00013430
		00013440

*/

```

IF DEBUG THEN ON STRG PUT FILE(PRINT) EDIT
  ('*****',L,M,BUF72)
  (SKIP,A,2 F(6,0),A);
PUT FILE(PRINT) EDIT('ENTER GETR')
  (SKIP(DEBUG), A(DEBUG*12));
ON ENDFILE(NEWPGM) BEGIN; FLG='E'; TXT=''; L=73;
  CALL PUTR('E',FLGR);
  GO TO RTRN;
END;

```

```

FLG='R';
GO TO INPT(K);
/*
*/

```

INPT(0):

```

READ FILE(NEWPGM) INTO(BUFR);
PUT FILE(PRINT) EDIT(BUFR) (SKIP(DEBUG),A(80*DEBUG));
BUF=BUFR;
IF (BUF1='C')|(BUF1='/' ) THEN DO;
  WRITE FILE(PGMNEW) FROM(BUFR);
  GO TO INPT(0);
END;
BUF73='';
N=N+1;

```

CALL PUTR(BUF6,FLGR);	00013450
K=1;	00013460
/*	00013470
*/	00013480
INPT(1):	00013490
IF M=73 THEN GO TO DNN;	00013500
IF SUBSTR(BUF72,M)=' ' THEN DO;	00013510
DNN:	00013520
L=73; CALL PUTR('A',FLGR); M=7; GO TO INPT(0);	00013530
END;	00013540
I=100;	00013550
DO J=1 TO LENGTH(DELIMS);	00013560
II=INDEX(SUBSTR(BUF,M),DELIM(J));	00013570
I=I*(II>I)+II*(II<I)+I*(II=0);	00013580
END;	00013590
IF I=100 THEN GO TO INPT(0);	00013600
I=11*(I>11)+I*(I<=11);	00013610
TXT=SUBSTR(BUF,M,I);	00013620
M=M+I;	00013630
MM=M-1;	00013640
IF CARD(MM)=';' THEN K=0;	00013650
IF CARD(MM)='' THEN FLG='L';	00013660
RTRN:	00013670
IF LENGTH(TXT)=1 THEN DO; CALL PUTR(TXT,FLGR); GO TO INPT(1);	00013680
END;	00013690
PUT FILE(PRINT) EDIT('LEAVE GETR',TXT,FLG,'LEAVE GETR')	00013700
(SKIP(DEBUG),4 A(12*DEBUG));	00013710
RETURN;	00013720
END GETR;	00013730

```

/*          STREAM OUTPUT MODULE FOR ERROR CORRECTION.          */
(SUBRG,STRG,SIZE):
PUTR:  PROC(TXT,FLG) CPTIONS(REENTRANT) RECURSIVE;
DCL    1 PARMS STATIC EXT UNALIGNED,
        2 (DEBUG,Z1,Z2,Z3) BIT(1),
        1 I_O UNALIGNED STATIC EXT,
        2 (K INIT(0),L INIT(1),M INIT(7),N INIT(0)) FIXED BIN(15,0),
        2 BUF CHAR(80),
        2 DELIMS CHAR(12) INIT(' +-*/(())'.'.=;, '),
        2 OUTREC CHAR(80)      INIT((80) ' '),
        RECOU STATIC CHAR(80),
        TXT CHAR(11) VAR,
        FLG BIT(1),
        WK73 CHAR(8) DEF RECOU POS(73),
        WK CHAR(1);

/*

IDENTIFICATION:
-----

PROGRAM-ID:  PUTR.
AUTHOR:  G. E. HEDRICK.
INSTALLATION:  IOWA STATE UNIVERSITY COMPUTATION CENTER.
DATE-WRITTEN:  26 FEBRUARY 1970.
SOURCE-COMPUTER:  IBM 360/65.
OBJECT-COMPUTER:  IBM 360/65.
OPERATING SYSTEM:  OS/MVT.
MEMORY SIZE:  DEPENDS ON OTHER PROCEDURES.

PURPOSE:
-----

THIS PROCEDURE SIMULATES STREAM OUTPUT ON A RECORD

```

```

00013750
00013760
00013770
00013780
00013790
00013800
00013810
00013820
00013830
00013840
00013850
00013860
00013870
00013880
00013890
00013900
00013910
00013920
00013930
00013940
00013950
00013960
00013970
00013980
00013990
00014000
00014010
00014020
00014030
00014040
00014050
00014060
00014070
00014080
00014090
00014100
00014110

```


FILE. THE RECORD OUTPUT FILE IS FILE PGMNEW. PORTIONS OF
THE NEW OUTPUT RECORD ARRIVE VIA THE INPUT PARAMETER TXT.

DESCRIPTION OF VARIABLES AND LABELS:

VARIABLE OR LABEL

DESCRIPTION

BUF.	INPUT BUFFER; CHECKED ONLY.
CLEAR.	POINT AT WHICH THE OUTPUT BUFFERS ARE CLEARED.
DEBUG.	EXTERNAL DEBUG FLAG.
DELIMS.	THE FORTRAN DELIMITERS.
FLG.	UNUSED PARAMETER.
I_O.	EXTERNAL STRUCTURE WHICH IS USED BY THE I/O ROUTINES FOR COMMUNICATION
K,L,M,N.	RECORD COUNTERS FOR I/O.
OUTREC,RECOU.	OUTPUT BUFFERS.
PARMS.	EXTERNAL PARAMETER LIST.
PGMNEW.	OUTPUT FILE.
PRINT.	DEBUG FILE.
TXT.	STRING TO BE OUTPUT.
WK,WK73.	WORKING CHARACTER STRINGS.
Z1,Z2,Z3.	FILLERS.

*/

```

CLEAR:  PROC RECURSIVE;
        PUT FILE(PRINT) EDIT('ENTER CLEAR')
        (SKIP(DEBUG),A(12*DEBUG));
RECOU=OUTREC;
WK73=' ';
IF RECOU-=(80) ' ' THEN

```

00014120
00014130
00014140
00014150
00014160
00014170
00014180
00014190
00014200
00014210
00014220
00014230
00014240
00014250
00014260
00014270
00014280
00014290
00014300
00014310
00014320
00014330
00014340
00014350
00014360
00014370
00014380
00014390
00014400
00014410
00014420
00014430
00014440
00014450
00014460
00014470
00014480

WRITE FILE(PGMNEW) FROM(RECOUT);	00014490
L=1;	00014500
OUTREC,RECOUT=' ';	00014510
PUT FILE(PRINT) EDIT('LEAVE CLEAR')	00014520
(SKIP(DEBUG),A(12*DEBUG));	00014530
RETURN;	00014540
END CLEAR;	00014550
/*	00014560
*/	00014570
IF DEBUG THEN ON STRG PUT FILE(PRINT) EDIT	00014580
(OUTREC,TXT,LENGTH(TXT),L) (SKIP,2 A,X(3),2 F(6,0));	00014590
PUT FILE(PRINT) EDIT('ENTER PUTR.',TXT,'ENTER PUTR')	00014600
(SKIP(DEBUG),3 A(12*DEBUG));	00014610
SUBSTR(OUTREC,L,LENGTH(TXT))=TXT;	00014620
L=L+LENGTH(TXT);	00014630
WK=SUBSTR(TXT,LENGTH(TXT),1);	00014640
IF M<=7 THEN RETURN;	00014650
IF L>=72 THEN CALL CLEAR;	00014660
ELSE	00014670
IF INDEX(DELIMS,WK)=0 THEN CALL CLEAR;	00014680
ELSE	00014690
IF WK=' ' THEN DO;	00014700
IF SUBSTR(BUF,M)=' ' THEN CALL CLEAR;	00014710
END;	00014720
PUT FILE(PRINT) EDIT('LEAVE PUTR')(SKIP(DEBUG),A(12*DEBUG));	00014730
RETURN;	00014740
END PUTR;	00014750

```

/*          TRANSLATION MATRIX FILL ROUTINE. */          00014770
TFILL:  PROC;          00014780
DCL      TMAT(0:5,0:4) FIXED BIN(15,0) STATIC EXT;      00014790
/*          00014800
          THIS ROUTINE SERVES TO FILL THE TRANSLATION MATRIX.  THIS  00014810
          MATRIX IS USED IN THE ERROR ANALYSIS AND CORRECTION ROUTINES  00014820
          WHICH FOLLOW.          00014830
/*          00014840
          TMAT(0,0),TMAT(1,0),TMAT(1,1)=1;          00014850
          TMAT(2,0),TMAT(3,0),TMAT(3,1)=3;          00014860
          TMAT(1,2),TMAT(3,4),TMAT(4,4)=2;          00014870
          TMAT(2,1),TMAT(4,1),TMAT(5,1)=4;          00014880
          TMAT(5,0)=3;          00014890
          TMAT(0,1)=6; TMAT(0,2)=7; TMAT(0,3)=20; TMAT(0,4)=8;      00014900
          TMAT(1,4)=9; TMAT(2,2)=10; TMAT(2,4)=11; TMAT(3,2)=12;      00014910
          TMAT(4,0)=13; TMAT(4,2)=14; TMAT(5,2)=15;          00014920
          TMAT(5,3)=16; TMAT(5,4)=17;          00014930
          TMAT(1,3),TMAT(2,3),TMAT(3,3)=20;          00014940
          RETURN;          00014950
          END; /* OF TRANSLATION MATRIX FILL ROUTINE. */          00014960

```

```

/*          SYMBOL TABLE INITIALIZATION ROUTINE.          */
DFILL:  PROC(DICTIONARY);
/*  THIS ROUTINE IS USED TO CREATE THE DICTIONARY WHICH
IS USED BY THE SYMBOL TABLE CORRECTION ALGORITHM.  */
DCL    DICTIONARY(255) CHAR(11) VAR;
DICTIONARY(1)='ASSIGN'; DICTIONARY(2)='GOTO';
DICTIONARY(3)='DIMENSION'; DICTIONARY(4)='END';
DICTIONARY(5)='SIN'; DICTIONARY(6)='COS'; DICTIONARY(7)='EXP';
DICTIONARY(8)='MOD'; DICTIONARY(9)='ABS'; DICTIONARY(10)='TAN';
DICTIONARY(11)='ERF'; DICTIONARY(12)='STOP';
DICTIONARY(13)='READ'; DICTIONARY(14)='REAL';
DICTIONARY(15)='DATA'; DICTIONARY(16)=' .GE.';
DICTIONARY(17)=' .GT.'; DICTIONARY(18)=' .EQ.';
DICTIONARY(19)=' .NE.'; DICTIONARY(20)=' .LE.';
DICTIONARY(21)=' .OR.'; DICTIONARY(22)=' .LT.';
DICTIONARY(23)='SQRT'; DICTIONARY(24)='IABS';
DICTIONARY(25)='AMOD'; DICTIONARY(26)='ALOG';
DICTIONARY(27)='ATAN'; DICTIONARY(28)='DMOD';
DICTIONARY(29)='SIGN'; DICTIONARY(30)='COSH';
DICTIONARY(31)='SINH'; DICTIONARY(32)='TANH';
DICTIONARY(33)='DABS'; DICTIONARY(34)='DEXP';
DICTIONARY(35)='DLOG'; DICTIONARY(36)='DSIN';
DICTIONARY(37)='DCOS'; DICTIONARY(38)='DTAN';
DICTIONARY(39)='MAX0'; DICTIONARY(40)='MAX1';
DICTIONARY(41)='MIN0'; DICTIONARY(42)='MIN1';
DICTIONARY(43)='CABS'; DICTIONARY(44)='CSIN';
DICTIONARY(45)='CCOS'; DICTIONARY(46)='ERFC';
DICTIONARY(47)='DERF'; DICTIONARY(48)='IDIM';
DICTIONARY(49)='HFIX'; DICTIONARY(50)='DBLE';
DICTIONARY(51)='AINT'; DICTIONARY(52)='SNGL';
DICTIONARY(53)='EXIT'; DICTIONARY(54)='FILE';
DICTIONARY(55)='WRITE'; DICTIONARY(56)='ENTRY';
DICTIONARY(57)=' .AND.'; DICTIONARY(58)='ISIGN';
DICTIONARY(59)='DSIGN'; DICTIONARY(60)='ARCCOS';
DICTIONARY(61)='ARSIN'; DICTIONARY(62)='COTAN';
DICTIONARY(63)='DSQRT'; DICTIONARY(64)='PRINT';
DICTIONARY(65)='PUNCH'; DICTIONARY(66)='AMAX0';

```

DICTIONARY(67)='AMIN1';	DICTIONARY(68)='AMAX1';	00015350
DICTIONARY(69)='AMINO';	DICTIONARY(70)='DMAXO';	00015360
DICTIONARY(71)='CMAX1';	DICTIONARY(72)='DMINO';	00015370
DICTIONARY(73)='DMIN1';	DICTIONARY(74)='DTANH';	00015380
DICTIONARY(75)='CDLOG';	DICTIONARY(76)='CDEXP';	00015390
DICTIONARY(77)='ATAN2';	DICTIONARY(78)='CDSIN';	00015400
DICTIONARY(79)='CDCOS';	DICTIONARY(80)='DCOSH';	00015410
DICTIONARY(81)='DSINH';	DICTIONARY(82)='DERFC';	00015420
DICTIONARY(83)='FORMAT';	DICTIONARY(84)='RETURN';	00015430
DICTIONARY(85)='COMMON';	DICTIONARY(86)='.TRUE.';	00015440
DICTIONARY(87)='DFLOAT';	DICTIONARY(88)='CDSQRT';	00015450
DICTIONARY(89)='ALOG10';	DICTIONARY(90)='DLOG10';	00015460
DICTIONARY(91)='DCOTAN';	DICTIONARY(92)='DARCOS';	00015470
DICTIONARY(93)='DARSIN';	DICTIONARY(94)='REWIND';	00015480
DICTIONARY(95)='DEFINE';	DICTIONARY(96)='INTEGER';	00015490
DICTIONARY(97)='LOGICAL';	DICTIONARY(98)='.FALSE.';	00015500
DICTIONARY(99)='COMPLEX';	DICTIONARY(100)='CONTINUE';	00015510
DICTIONARY(101)='IMPLICIT';		00015520
DICTIONARY(102)='EXTERNAL';		00015530
DICTIONARY(103)='FUNCTION';		00015540
DICTIONARY(104)='NAMELIST';		00015550
DICTIONARY(105)='SUBROUTINE';		00015560
DICTIONARY(106)='EQUIVALENCE';		00015570
DICTIONARY(107)='GRAPH';	DICTIONARY(108)='PRINT';	00015580
DICTIONARY(109)='READ';		00015590
DICTIONARY(110)='END';		00015600
DICTIONARY(111)='CALL';		00015610
RETURN;		00015620
END DFILL;		00015630

```

/*          SELECTED I/O MODULES FOR THE ERROR CORRECTOR. */ 00015650
(STRG,SUBRG,SIZE):                                           00015660
RESET:  PROC;                                                00015670
    DCL (PRINT,PGMNEW,NEWPGM) FILE EXT, CARD CHAR(80);      00015680
    DCL 1 PARMS STATIC UNALIGNED EXT,                        00015690
        2 DEBUG BIT(1), 2 (ZU,ZI,ZO) BIT(1);                 00015700
/*                                                           00015710
                                                           00015720
                                                           00015730
                                                           00015740
IDENTIFICATION:                                              00015750
-----                                                    00015760
                                                           00015770
PROGRAM-ID:  RESET.                                         00015780
AUTHOR:  G. E. HEDRICK.                                     00015790
DATE-WRITTEN:  14 FEBRUARY 1970.                            00015800
SOURCE-COMPUTER:  IBM 360/65.                                00015810
OBJECT-COMPUTER:  IBM 360/65.                                00015820
OPERATING SYSTEM:  OS/MVT.                                   00015830
MEMORY SIZE:  DEPENDS ON OTHER CORRECTOR PROCEDURES.        00015840
                                                           00015850
                                                           00015860
                                                           00015870
PURPOSE:                                                    00015880
-----                                                    00015890
                                                           00015900
        THIS PROCEDURE HAS ENTRY POINTS SET,UNSET, AND      00015910
        RESET.  THE PROCESSING WHICH STARTS AT EACH ENTRY POINT IS 00015920
        USED WHEN A FILE USE MUST BE INITIALIZED OR TERMINATED. 00015930
                                                           00015940
                                                           00015950
                                                           00015960
                                                           00015970
/*                                                           00015980
    PUT FILE(PRINT) EDIT('ENTER RESET')                     00015990
        (SKIP(DEBUG),A(DEBUG*11));                            00016000
    CLOSE FILE(NEWPGM);                                       00016010
    OPEN FILE(NEWPGM) SEQUENTIAL RECORD INPUT;

```

ON ENDFILE(NEWPGM) GO TO RTRNR;	00016020
DO WHILE('1'B);	00016030
READ FILE(NEWPGM) INTO(CARD);	00016040
IF CARD=' ' THEN	00016050
WRITE FILE(PGMNEW) FROM(CARD);	00016060
END;	00016070
RTRNR: CLOSE FILE(NEWPGM);	00016080
OPEN FILE(NEWPGM) SEQUENTIAL RECORD OUTPUT;	00016090
PUT FILE(PRINT) EDIT('NEWPGM--OUTPUT')	00016100
(SKIP(DEBUG),A(DEBUG*20));	00016110
RETURN;	00016120
/*	00016130
*/	00016140
SET: ENTRY;	00016150
PUT FILE(PRINT) EDIT('ENTER SET')	00016160
(SKIP(DEBUG),A(DEBUG*11));	00016170
CLOSE FILE(NEWPGM);	00016180
OPEN FILE(NEWPGM) SEQUENTIAL RECORD INPUT;	00016190
IF DEBUG THEN DO;	00016200
ON ENDFILE(NEWPGM) GO TO NXTSET;	00016210
PUT FILE(PRINT) SKIP EDIT('NEW PROGRAM:')(A);	00016220
DO WHILE('1'B);	00016230
READ FILE(NEWPGM) INTO(CARD);	00016240
PUT FILE(PRINT) SKIP EDIT(CARD) (A);	00016250
END;	00016260
NXTSET: CLOSE FILE(NEWPGM);	00016270
OPEN FILE(NEWPGM) RECORD SEQUENTIAL INPUT;	00016280
PUT FILE(PRINT) SKIP EDIT('END OF PROGRAM.')(A);	00016290
END;	00016300
PUT FILE(PRINT) EDIT('NEWPGM--INPUT ')	00016310
(SKIP(DEBUG),A(DEBUG*20));	00016320
RETURN;	00016330
/*	00016340
*/	00016350
UNSET: ENTRY;	00016360
PUT FILE(PRINT) EDIT('ENTER UNSET')	00016370
(SKIP(DEBUG),A(DEBUG*11));	00016380

GO TO RTRNR;
END RESET;

00016390
00016400


```

/*          CLEANUP ROUTINE FOR ERROR CORRECTION.          */
(SUBRG,STRG,SIZE):
CLEANUP: PROC(PARM) OPTIONS(MAIN);
/*
*/
DCL      1 PARMS STATIC EXT UNALIGNED,
          2 (DEBUG,LIST,UNUSED) BIT(1),
          1 DATAX STATIC EXT UNALIGNED,
          2 (I,N) PICTURE '99999999',
          2 (CRD,BF1,BF2) CHAR(80),
          2 LEGAL CHAR(11) INIT(' 0123456789'),
          2 QUOTE CHAR(1) INIT(''),
          2 LP CHAR(1) INIT('('),
          2 RP CHAR(1) INIT(')'),
          2 STARS CHAR(72) INIT((72) '*'),
          2 CODE CHAR(50) INIT('C
OWING CARD'),
          CARD72 CHAR(72) DEF CRD,
          BUF1_72 CHAR(72) DEF BF1,
          BUF2_72 CHAR(72) DEF BF2,
          BUF2_50 CHAR(50) DEF BF2,
          COL1_5 CHAR(5) DEF CRD,
          COL7_72 CHAR(66) DEF CRD POS(7),
          CCOLS(72) CHAR(1) UNALIGNED DEF CRD,
          CCOL(72) CHAR(1) UNALIGNED DEF CRD,
          (P,Q,R) PCINTER,
          (CARD BASED(P),BUF1 BASED(Q), BUF2 BASED(R)) CHAR(80),
          B66 CHAR(66) INIT((66) ' '),
          B5 CHAR(5) DEF B66,
          (PRINT,PGMNEW) FILE EXT,
          SCRATCH FILE EXT KEYED ENV(
          PARM CHAR(40);
          REGIONAL(1)),
/*
IDENTIFICATION:
-----

```

```

00016890
00016900
00016910
00016920
00016930
00016940
00016950
00016960
00016970
00016980
00016990
00017000
00017010
00017020
00017030
00017040
00017050
00017060
00017070
00017080
00017090
00017100
00017110
00017120
00017130
00017140
00017150
00017160
00017170
00017180
00017190
00017200
00017210
00017220
00017230
00017240
00017250

```

PROGRAM-ID: CLEANUP.
AUTHOR: G. E. HEDRICK.
INSTALLATION: IOWA STATE UNIVERSITY COMPUTATION CENTER.
DATE-WRITTEN: 17 MARCH 1970.
SOURCE-COMPUTER: IBM 360/65.
OPERATING-SYSTEM: OS/MVT.
MEMORY-SIZE: HIGH SPEED CORE: 96K BYTES.
 BULK CORE: 0 BYTES.

DESCRIPTION OF THE PROBLEM:

THE COMPLETE JOB IS DESIGNED TO CORRECT SOURCE ERRORS IN
FORTRAN PROGRAMS. THIS PROCEDURE IS DESIGNED TO OPERATE AS
A SEPARATE STEP OF THE ENTIRE CORRECTION PROCESS. THE
FUNCTION OF THIS PROCESS IS TO COLLECT AND TO ELIMINATE THE
TRASH THAT REMAINS IN THE FORTRAN PROGRAM AFTER OTHER
CORRECTION MODULES HAVE OPERATED ON IT.

METHOD OF SOLUTION:

THE METHOD OF SOLUTION IS TO DETERMINE WHICH OF THE CARD
IMAGES IN THE PROGRAM ARE TRASH AND DELETE THESE RECORDS. THE
STEPS BY WHICH THIS PROCESS IS EFFECTED ARE:

1. SET UP PARAMETER FLAGS;
2. OPEN ALL REQUIRED FILES;
3. INITIALIZE VARIABLES;
4. READ A CARD IMAGE FROM THE FORTRAN PROGRAM;
5. IF THERE IS AN END OF FILE GO TO STEP 8;
6. WRITE THE CARD IMAGE ON A SCRATCH DIRECT ACCESS DATASET;
7. GO TO STEP 4;
8. CLOSE SCRATCH FILES;

00017260
00017270
00017280
00017290
00017300
00017310
00017320
00017330
00017340
00017350
00017360
00017370
00017380
00017390
00017400
00017410
00017420
00017430
00017440
00017450
00017460
00017470
00017480
00017490
00017500
00017510
00017520
00017530
00017540
00017550
00017560
00017570
00017580
00017590
00017600
00017610
00017620

9. RE-OPEN SCRATCH FILES;	00017630
10. RE-INITIALIZE SELECTED VARIABLES;	00017640
11. READ TWO CARD IMAGES;	00017650
12. READ A THIRD CARD IMAGE;	00017660
13. IF AN "ERROR" OCCURS GO TO STEP 19;	00017670
14. IF THE FIRST CARD IMAGE IS TRASH GO TO STEP 16;	00017680
15. WRITE THE FIRST CARD IMAGE ON THE NEW PROGRAM DATA SET;	00017690
16. REPLACE THE FIRST CARD IMAGE BY THE SECOND;	00017700
17. REPLACE THE SECOND CARD IMAGE BY THE THIRD;	00017710
18. GO TO STEP 12;	00017720
19. CLOSE AND RE-OPEN NECESSARY FILES;	00017730
20. COPY SEQUENTIAL FILE TO DIRECT FILE;	00017740
21. CHECK FORMAT STATEMENTS FOR IMBEDDED GARBAGE;	00017750
22. COPY DIRECT FILE TO SEQUENTIAL FILE;	00017760
23. GO TO NEXT JOB STEP.	00017770

DESCRIPTION OF INPUT:

THERE ARE TWO SOURCES OF INPUT TO THIS PROGRAM: THE INPUT PARAMETER STRING AND THE FORTRAN NEW PROGRAM DATA SET. THE INPUT PARAMETER STRING COMES FROM THE EXEC CARD AND CONTAINS A LIST OF OPTIONS FOR THIS MODULE. THE NEW PROGRAM DATA SET CONTAINS THE NEW FORTRAN PROGRAMS WITH SOME TRASH THAT REMAINS FROM THE OTHER CORRECTION MODULES.

TABLE OF INPUT PARAMETERS:

PARAMETER	FUNCTION
-----	-----

DEBUG.

REQUESTS DEBUG OUTPUT INCLUDING A

00017630
00017640
00017650
00017660
00017670
00017680
00017690
00017700
00017710
00017720
00017730
00017740
00017750
00017760
00017770
00017780
00017790
00017800
00017810
00017820
00017830
00017840
00017850
00017860
00017870
00017880
00017890
00017900
00017910
00017920
00017930
00017940
00017950
00017960
00017970
00017980
00017990

	TRACE OF THE PROGRAM EXECUTION.	00018000
LIST.	REQUESTS LISTINGS OF FORTRAN PROGRAMS	00018010
		00018020
		00018030
		00018040
DESCRIPTION OF OUTPUT:		00018050
-----		00018060
		00018070
THERE ARE THREE KINDS OF OUTPUT FROM THIS PROCEDURE:		00018080
DEBUG OUTPUT, LIST OUTPUT, AND PROGRAM OUTPUT. DEBUG AND LIST		00018090
OUTPUT ARE PRINTED AND MUST BE REQUESTED WITH AN INPUT		00018100
PARAMETER. DEBUG OUTPUT CONSISTS OF SELECTED PRINTOUT WHICH		00018110
INCLUDES A TRACE OF THE PROGRAM. LIST OUTPUT CONSISTS OF		00018120
SELECTED LISTINGS OF THE FORTRAN PROGRAM. PROGRAM OUTPUT IS		00018130
ALWAYS GIVEN. PROGRAM OUTPUT CONSISTS OF THE CLEANSED FORTRAN		00018140
PROGRAM(5); IT IS PLACED IN THE NEW PROGRAM DATA SET.		00018150
		00018160
		00018170
		00018180
DESCRIPTION OF FILES:		00018190
-----		00018200
		00018210
FILE	USE	00018220
----	----	00018230
		00018240
PGMNEW.	FILE WHICH CONTAINS FORTRAN PROGRAMS	00018250
	BEFORE AND AFTER CLEANUP.	00018260
PRINT.	FILE WHICH CONTAINS LIST AND DEBUG	00018270
	OUTPUT.	00018280
SCRATCH.	DIRECT UPDATE SCRATCH FILE, USED IN	00018290
	TRASH ELIMINATION.	00018300
		00018310
		00018320
		00018330
DESCRIPTION OF VARIABLES AND LABELS:		00018340
-----		00018350
		00018360

VARIABLE OR LABEL -----	MEANING OR USE -----	
B5,B66.	VARIABLE CONSISTING ENTIRELY OF BLANKS; USED FOR TESTING.	00018370
BF1,BF2,BUF1,BUF2, CARD,CRD.	CARD IMAGES.	00018380
BUF1_72.	THE FIRST 72 COLUMNS OF BUF1.	00018390
BUF2_50.	THE FIRST 50 COLUMNS OF BUF2.	00018400
BUF2_72.	THE FIRST 72 COLUMNS OF BUF2.	00018410
CARD72.	THE FIRST 72 COLUMNS OF CARD.	00018420
CCOLS.	THE CHARACTER ARRAY FCRM OF CARD.	00018430
CODE,STARS.	CODE USED IN TESTING FOR TRASH.	00018440
COL1_5.	THE FIRST 5 COLUMNS OF CARD.	00018450
COL7_72.	COLUMNS 7 THROUGH 72 OF CARD.	00018460
DATAX.	STRUCTURE USED FOR MAINTAINING CERTAIN VARIABLES.	00018470
DEBUG,LIST.	INPUT PARAMETERS.	00018480
FMTS.	PLACE WHERE FORMAT STATEMENTS ARE EXAMINED.	00018490
I,N.	COUNTERS ON LOOPS.	00018500
LP,RP.	PARENTHESIS.	00018510
OUT.	THE POINT OF TERMINATION OF THIS PROCEDURE.	00018520
P,Q,R.	POINTERS USED TO SIMULATE OVERLAY DEFINING.	00018530
PARM,PARMS.	SETS OF INPUT PARAMETERS.	00018540
PGMNEW,PRINT,SCRATCH.	FILES.	00018550
QUOTE.	A SINGLE QUOTE MARK.	00018560
SKPCD.	THE POINT TO WHICH CONTROL IS TRANSFERRED WHEN A CARD MUST BE DELETED.	00018570
		00018580
		00018590
		00018600
		00018610
		00018620
		00018630
		00018640
		00018650
		00018660
		00018670
		00018680
		00018690
		00018700
		00018710
		00018720
		00018730
REFERENCES: -----		

THE REFERENCES ARE LISTED WITH THE OTHER MODULES AND
IN THE BIBLIOGRAPHY OF THE DISSERTATION WHICH SERVES AS A
DOCUMENTATION OF THIS PROGRAM.

*/

```

FMTS: PROC;
PUT FILE(PRINT) EDIT('ENTER FMTS.') (SKIP(DEBUG),A(DEBUG*100));
PUT FILE(PRINT) EDIT('LEAVE FMTS.') (SKIP(DEBUG),A(DEBUG*100));
RETURN;
END FMTS;
P=ADDR(CRD); Q=ADDR(BF1); R=ADDR(BF2);
DEBUG=(INDEX(PARM,'DEBUG')>0)&(INDEX(PARM,'NODEBUG')=0);
LIST=(INDEX(PARM,'LIST')>0)&(INDEX(PARM,'NOLIST')=0);
OPEN FILE(PGMNEW) INPUT RECORD SEQUENTIAL,
      FILE(SCRATCH) OUTPUT RECORD SEQUENTIAL,
      FILE(PRINT) STREAM PRINT OUTPUT LINESIZE(120)
      PAGESIZE(55);
PUT FILE(PRINT) EDIT('START OF PROGRAM. FILES OPENED.')
      (SKIP(DEBUG),A(100*DEBUG));
SUBSTR(STARS,1,1)='C';
CRD,BF1,BF2=(80) ' ';
PUT FILE(PRINT) EDIT('START OF PHASE 1.')
      (SKIP(DEBUG),A(100*DEBUG));
PUT FILE(PRINT) EDIT('OPTIONS IN EFFECT ARE:',PARM)
      (SKIP(LIST),2 A(LIST*40));
ON ENDFILE(PGMNEW) BEGIN;
      PUT FILE (PRINT) EDIT('EOF PGMNEW')
      (SKIP(DEBUG),A(DEBUG*100));
      CARD='/*'; WRITE FILE(SCRATCH) FROM(CARD) KEYFROM(1);
      CLOSE FILE(PGMNEW), FILE(SCRATCH);
      OPEN FILE(SCRATCH) DIRECT UPDATE RECORD KEYED,
      FILE(PGMNEW) RECORD SEQUENTIAL OUTPUT;
      PUT FILE(PRINT) EDIT('END OF PHASE 1. PHASE 2 INITIAL')
      (SKIP(DEBUG),A(100*DEBUG));

```

I=0;	00019120
READ FILE(SCRATCH) INTO(CARD) KEY(I);	00019130
I=1;	00019140
READ FILE(SCRATCH) INTO(BUF1) KEY(I);	00019150
ON KEY(SCRATCH) BEGIN;	00019160
PUT FILE(PRINT) EDIT('END PHASE 2. START PHASE 3.')	00019170
(SKIP(DEBUG),A(100*DEBUG));	00019180
CLOSE FILE(PGMNEW);	00019190
OPEN FILE(PGMNEW) RECORD SEQUENTIAL INPUT;	00019200
PUT FILE(PRINT) EDIT('PHASE 3B.')	00019210
(SKIP(DEBUG),A(100*DEBUG));	00019220
ON ENDFILE(PGMNEW) BEGIN;	00019230
PUT FILE(PRINT) EDIT('PHASE 3C.')	00019240
(SKIP(DEBUG),A(100*DEBUG));	00019250
CLOSE FILE(PGMNEW);	00019260
ON KEY(SCRATCH) BEGIN;	00019270
PUT FILE(PRINT) EDIT('PHASE 3D.')	00019280
(SKIP(DEBUG),A(100*DEBUG));	00019290
ON KEY(SCRATCH) GO TO OUT;	00019300
PUT FILE(PRINT) EDIT('LISTING FOLLOWS:')	00019304
(SKIP(LIST),A(LIST*100));	00019305
DO I=0 BY 1 WHILE(COLL_5~='/*');	00019310
READ FILE(SCRATCH) INTO(CARD) KEY(I);	00019320
WRITE FILE(PGMNEW) FROM(CARD);	00019330
IF COLL_5='/*STOP' THEN DO; CARD='/*';	00019331
WRITE FILE(PGMNEW) FROM(CARD);	00019332
SIGNAL KEY(SCRATCH);	00019333
END;	00019333
PUT FILE(PRINT) EDIT(CARD)	00019334
(SKIP(LIST),A(LIST*100));	00019335
END;	00019340
SIGNAL KEY(SCRATCH);	00019345
END;	00019350
DO I=0 BY 1;	00019360
READ FILE(SCRATCH) INTO(CARD) KEY(I);	00019370
IF INDEX(CARD,'FORMAT')~=0 THEN CALL FMTS;	00019380
END;	00019390

END;	00019400
DO I=0 BY 1;	00019410
READ FILE(PGMNEW) INTO(CARD);	00019420
REWRITE FILE(SCRATCH) FROM(CARD) KEY(I);	00019430
END;	00019440
END;	00019450
DO I=2 BY 1 WHILE(COL1_5-='/* ');	00019460
READ FILE(SCRATCH) INTO(BUF2) KEY(I);	00019465
PUT FILE(PRINT) EDIT('ABOUT TO TEST',CARD,BUF1,BF2)	00019470
(4 (SKIP(DEBUG),A(DEBUG*100)));	00019480
PUT FILE(PRINT) EDIT(CCOLS,'*',COL1_5)	00019490
(SKIP(DEBUG), 73 A(DEBUG), A(DEBUG*5));	00019491
IF(COL1_5-='/DATA')&(COL1_5-='/STOP')&	00019500
(COL1_5-='/JOB') THEN DO;	00019510
IF CCOL(1)-='C' THEN DO N=1 TO 5 BY 1;	00019520
IF INDEX(LEGAL,CCOL(N))=0 THEN GO TO SKPCD;	00019530
END;	00019540
END; ELSE GO TO LBL;	00019550
IF COL7_72=866 THEN GO TO SKPCD;	00019560
LBL:	00019565
IF BUF1_72=STARS THEN DO;	00019570
PUT FILE(PRINT) EDIT('STARS SPOTTED.')	00019580
(SKIP(DEBUG),A(DEBUG*100));	00019590
PUT FILE(PRINT) EDIT('TESTING',BUF2_50,CODE)	00019600
(SKIP(LIST),A(LIST*12),	00019610
2 (SKIP(LIST),A(LIST*100)));	00019620
IF BUF2_50=CODE THEN DO;	00019630
PUT FILE(PRINT) EDIT('GOT ONE')	00019640
(SKIP(DEBUG),A(DEBUG*100));	00019650
GO TO SKPCD;	00019660
END;	00019670
END;WRITE FILE(PGMNEW) FROM(CARD);	00019680
SKPCD:	00019690
CARD=BUF1;	00019700
BUF1=BUF2;	00019710
IF COL1_5='/STOP' THEN DO;	00019714
IF SUBSTR(BF1,1,5)-='/* ' THEN SIGNAL KEY(SCRATCH);	00019715

END;	00019716
PUT FILE(PRINT) EDIT('END OF TEST LOOP')	00019720
(SKIP(DEBUG),A(DEBUG*100));	00019730
END;	00019740
SIGNAL KEY(SCRATCH);	00019745
END;	00019750
PUT FILE(PRINT) EDIT('GO...')(SKIP(DEBUG),A(100*DEBUG));	00019760
PUT FILE(PRINT) EDIT('INPUT LISTING FOLLOWS:')	00019770
(SKIP(LIST),A(LIST*100));	00019780
BF1=' ';	00019785
DO I=0 BY 1 WHILE(COL1_5~='/* ');	00019790
READ FILE(PGMNEW) INTO(CARD);	00019800
PUT FILE(PRINT) EDIT(CARD)(SKIP(LIST),A(LIST*100));	00019810
IF SUBSTR(BF1,1,5)='/STOP' THEN DO;	00019814
IF COL1_5~='/* ' THEN SIGNAL ENDFILE(PGMNEW) ;	00019815
END;	00019816
WRITE FILE(SCRATCH) FROM(CARD) KEYFROM(I);	00019820
BF1=CARD;	00019825
END;	00019830
SIGNAL ENDFILE(PGMNEW);	00019835
OUT:	00019840
PUT FILE(PRINT) EDIT('END OF PHASE 3.')	00019850
(SKIP(DEBUG),A(DEBUG*100));	00019860
CLOSE FILE(SCRATCH), FILE(PGMNEW);	00019870
PUT FILE(PRINT) EDIT('LEAVE CLEANUP.')	00019880
(SKIP(DEBUG),A(DEBUG*100));	00019890
RETURN;	00019900
END CLEANUP;	00019910

XII. APPENDIX C

Summary of the corrigible and incorrigible errors for the implementation of the error corrector shown in Appendix B.

Incorrigible errors

WATFOR errors (see Blatt (6))	Reason
AS-2,...,AS-4,BD-0,BD-1	low probability of occurrence
CM-0,...,CM-4,CN-0,...,CN-5,CN-8,CN-9	coding of correction modules is difficult
CP-0,...,CP-4	compiler error
DA-0,...,DA-3,DM-0,...,DM-4, DO-1,...,DO-9	coding of correction modules is difficult
EC-0,...,EC-9,EN-0,...,EN-3	low probability of occurrence
EV-0,...,EV-4,EX-0,...,EX-9, EY-0,...,EY-6	low probability of occurrence
FM-0,...,FM-7,FT-0,...,FT-F FN-0,...,FN-8,GO-0,...,GO-4	coding of correction modules is difficult
HO-0,...,HO-4	low probability of occurrence
IF-0,...,IF-4	coding of correction modules is difficult
IM-0,...,IM-9	relatively low probability of occurrence
IO-0,...,IO-K	coding of correction modules is difficult
JB-1,JB-2,JB-3	not problem program errors
LG-2,L1-0,...,L1-H	low probability of occurrence
MD-2,...,MD-6,MO-0,...,MO-4,PC-0,PC-1	coding of correction modules is difficult
RE-0,...,RE-4,SF-1,SF-2,SF-3	low probability of occurrence
SR-0,...,SR-A,SS-0,SS-1,SS-2	coding of correction modules is difficult
UN-0,...,UN-9	violation of installation rules

UV-1,...,UV-6,VA-0,...,VA-E

coding of correction modules
is difficult

XT-0,XT-1,XT-2

low probability of occurrence

Corrigible errors

errors	ERRSET number
CC-0,...,CC-9	03
CN-6	03
ST-0,...,ST-A	16
SV-0,...,SV-5	16
SX-0,...,SX-D	16

Errors which are not in one of the above two lists are usually dependent errors.