User error analysis and automatic correction for compiling

George Ellwood Hedrick III
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Recommended Citation
https://lib.dr.iastate.edu/rtd/4234
HEDRICK, III, George Ellwood, 1943-
USER ERROR ANALYSIS AND AUTOMATIC CORRECTION
FOR COMPILING.

Iowa State University, Ph.D., 1970
Computer Science

University Microfilms, A XEROX Company, Ann Arbor, Michigan
USER ERROR ANALYSIS AND AUTOMATIC CORRECTION FOR COMPILING

by

George Ellwood Hedrick, III

A Dissertation Submitted to the Graduate Faculty in Partial Fulfillment of The Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

Head of Major Department

Signature was redacted for privacy.

Dean of Graduate College

Iowa State University
Of Science and Technology
Ames, Iowa
1970
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II. BASIC DEFINITIONS</td>
<td>24</td>
</tr>
<tr>
<td>III. FACTORS WHICH INFLUENCE THE DESIGN OF THE SOLUTION TO THE ERROR CORRECTION PROBLEM</td>
<td>32</td>
</tr>
<tr>
<td>IV. FACTORS WHICH INFLUENCE THE IMPLEMENTATION OF AN ERROR CORRECTOR</td>
<td>46</td>
</tr>
<tr>
<td>V. THE DESIGN AND IMPLEMENTATION OF AN ERROR CORRECTOR</td>
<td>61</td>
</tr>
<tr>
<td>VI. STATISTICAL PROCESSING FOR AN ERROR CORRECTOR</td>
<td>82</td>
</tr>
<tr>
<td>VII. RESULTS, SUMMARY, AND SUGGESTED FUTURE WORK</td>
<td>95</td>
</tr>
<tr>
<td>VIII. BIBLIOGRAPHY</td>
<td>97</td>
</tr>
<tr>
<td>IX. ACKNOWLEDGEMENTS</td>
<td>102</td>
</tr>
<tr>
<td>X. APPENDIX A</td>
<td>103</td>
</tr>
<tr>
<td>XI. APPENDIX B</td>
<td>120</td>
</tr>
<tr>
<td>XII. APPENDIX C</td>
<td>203</td>
</tr>
</tbody>
</table>
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 know 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 research­
ers and the students.

In order to start this research certain basic definitions are
required. The research commences with formal definitions of the differ­
et 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, $e_1$, exists at location $a_1$, change the text at $a_1$ such that $e_1$ no longer exists and that no new errors, $e_q$, are introduced. It is unfortunate that it is often more difficult to find $a_1$ than it is to find the existence of $e_1$. 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 BACAIC 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).
Table 1.1 The transformation matrix which is used in the algorithm of Figure 2 as it is applied to the syntax given in Appendix A. Empty entries are treated as zeros. Row and column encoding are given (17).

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
\[ \alpha = \{ a_1, a_2, \ldots, a_i, \ldots \} \]

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) $T_1 \preceq T_2$ if there is a production $U \rightarrow xT_1T_2y$ or $U \rightarrow xT_1U_1T_2y$ where $U_1 \in V$;

2) $T_1 \succ T_2$ if there is a production $U \rightarrow xU_1T_2y$ and a derivation $U_1 \Rightarrow Z$ where $U_1 \in V$ and $T_1$ is the rightmost terminal character of $Z$.

3) $T_1 \prec T_2$ if there is a production $U \rightarrow xT_1U_2y$ and a derivation $U_1 \Rightarrow Z$ where $U_1 \in V$ 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 \textit{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 \textit{a priori} probability of detection, and
2) the \textit{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.*

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 \( \mathcal{A} = \{1_1, 1_2, \ldots, 1_n\} \) 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 \( E^e \) be the set of words in \( \Sigma \) (all finite permutations of the letters in \( \Sigma \)) that are at least a Hamming distance of \( e \) from a valid word in the language, \( L \). Then \( E^e \) 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 \( E = (e_1, e_2, \ldots, e_n) \) 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 \( S = (e_{i_1}, e_{i_2}, \ldots, e_{i_k}) \) 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, $P_n(E)$, $P_{n-k_1}(S_1)$, $P_{n-k_1-k_2}(S_2)$, ..., $P_{n-k_1-...-k_q}(S_q)$; where $E = S_0$, and $S_j$ is a subset of $S_{j-1}$ of dimension $n-k_1-k_2-...-k_j$, and $S$ is the null set; this implies that $n-k_1-k_2-...-k_q = 0$. 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

Legend:

- **P**: the program
- **k₁**: the number of independent errors corrected during the last pass of the corrector
- **n**: the number of errors in the program

**NOTE:** A branch at point A indicates that the program has been totally corrected or repaired. A branch at point B indicates that the program has been partially corrected or repaired.
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, $k \leq n$, \textsc{rscse}
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, \[ \square \]); 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 cation 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. TCI 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.

<table>
<thead>
<tr>
<th>New</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>12</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>4</td>
<td>14</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>4</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
</tbody>
</table>

Legend

<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null string</td>
</tr>
<tr>
<td>1</td>
<td>left hand side</td>
</tr>
<tr>
<td>2</td>
<td>operator</td>
</tr>
<tr>
<td>3</td>
<td>variable</td>
</tr>
<tr>
<td>4</td>
<td>number</td>
</tr>
<tr>
<td>5</td>
<td>exponent</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>letter</td>
</tr>
<tr>
<td>1</td>
<td>number</td>
</tr>
<tr>
<td>2</td>
<td>equal sign</td>
</tr>
<tr>
<td>3</td>
<td>exponent</td>
</tr>
<tr>
<td>4</td>
<td>operator</td>
</tr>
</tbody>
</table>
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 blank characters 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 show 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.
Legend:

n number of independent errors

$P_n^E$ error space

k number of errors corrected during the immediately preceding pass of the error corrector

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 is 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 WAFTR 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 WATFOR 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 WATFOR 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 WATFOR 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 WATFOR compiler for each WATFOR 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 Contro ling error correction module. The basic analysis modules are incorporated into this process.
<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBUG</td>
<td>request debug output</td>
</tr>
<tr>
<td>COPYINPUT</td>
<td>requests that input to the corrector be copied on the printer</td>
</tr>
<tr>
<td>COPYOUTPUT</td>
<td>requests that disk output be copied on the printer</td>
</tr>
<tr>
<td>PRINT</td>
<td>requests additional printed output (unspecified)</td>
</tr>
</tbody>
</table>
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 user's 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 ALGl 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 Creation. $A_c$ 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

<table>
<thead>
<tr>
<th>FLAG</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>end-of-data encountered</td>
</tr>
<tr>
<td>L</td>
<td>data comes from the left hand side of a FORTRAN assignment statement</td>
</tr>
<tr>
<td>R</td>
<td>data comes from the right hand side of a FORTRAN assignment statement</td>
</tr>
<tr>
<td>N</td>
<td>returning data is blank</td>
</tr>
</tbody>
</table>
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 minimiza-
tion, 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.
and it is decided that

<table>
<thead>
<tr>
<th>If it is true that</th>
<th>the corrector worked</th>
<th>the corrector did not work</th>
</tr>
</thead>
<tbody>
<tr>
<td>The error was repaired or corrected</td>
<td>Valid</td>
<td>Invalid</td>
</tr>
<tr>
<td>The error was not repaired or corrected</td>
<td>Invalid</td>
<td>Valid</td>
</tr>
</tbody>
</table>

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

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


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(I) INIT CALL MFILL,
      SS CHAR(80),
      (S(80) DEF SS POS(I),R(0:80),LMDEX(0:16) STATIC INIT
        ('0', '+', 'T', 'I', '2', 'G', 'N', '*', 'I', 'V', 'E', 'U', 'M',
        '(', ')', 'C', 'E'= '1'),
      TMDEX(0:16) STATIC INIT('0', '+', 'T', 'G', 'I', 'S',
        'E', 'M', '#',
        '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(CHAR(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:
------------------------

\[<A.V,> ::= <LPL><AE>;
\]
\[<LPL> ::= <LP>;
\]
\[<LP>::= <V> | <PI> =
\]
\[<V>::= <SIM.V> | <SUB.V>
\]
\[<SUB.V>::= <AI> | 'I' | <SL> %
\]
\[<AI>::= <I>
\]
\[<SL>::= <SE> | <SL>, <SE>
\]
\[<SE>::= <AE>
\]
\[<SIM.V>::= <VI>
\]
\[<VI>::= <I>
\]
\[<AE>::= <SAE>
\]
\[<SAE>::= <T> | <AOP><T> | <SAE><AOP><T>
\]
\[<T>::= <F> | <T><MOP><F>
\]
\[<F>::= <P> | <F><D>
\]
\[<P>::= <U.N.> | <V> | {<AF>}
\]
\[<U.N.>::= <D> | <U.N.><D>
\]
\[<I>::= <I>
\]
\[<I>::= <L> | <I><L> | <I><D>
\]

NOTE 1.
\(<MOP>::=\ast I/
\(<AOP>::=+I-
\(<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. THIS GRAMMAR IS BASED UPON THE DESCRIPTION OF THE ALGOL ASSIGNMENT STATEMENT. THE BRACKETS HAVE BEEN CHANGED TO THE | AND % SYMBOLS, AND \(\ast\) 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(1,1), M(5,1), M(6,3) = '* N'; M(5,3) = '* G';
M(5,2), M(6,2) = '* I'; M(2,7), M(3,7) = '* 2';
M(8,1), M(8,4) = '* I'; M(8,16), M(9,16) = '?';
M(6,7), M(6,8), M(7,7), M(7,9), 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(11,9), 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, LMDEX)
(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 DFILL;

G:
PROC(Q) CHAR(I);
DCL Q CHAR(I);
/*
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) AND ('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 EI;
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 USABLE FORM. */
DO L=1 TO 16 BY 1 WHILE(R(L-1)<=LMDEX(L)); END; L=MOD(L,17);
DO K=1 TO 16 BY 1 WHILE(R(J)<=TMDEX(K)); END; K=MOD(K,17);
F=M(L,K);
IF F=';' 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 EI:
   PUT FILE(SYSPRINT) EDIT('***** ERROR *****')
   (SKIP,A);
   GO TO LOOP1;
STP: RETURN;
IDENTIFICATION:

---------------

PROGRAM-ID: SYNTAX1.
AUTHOR: G. E. HEDRICK.
INSTALLATION: IOWA STATE UNIVERSITY COMPUTATION CENTER.
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 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 <OP> L
L ::= X | Y | Z
N ::= D | N <OP>
D ::= 0 | 1 | A | R
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 IVFSON'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.

4. IVESON, 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) CHAP(P) VAR,
TMP(0:9) CHAR(5) STATIC INIT
('TEMPO', 'TEMPO', 'TEMPO', 'TEMPO', 'TEMPO', 'TEMPO',
  'TEMPO', 'TEMPO', 'TEMPO'),
OPST(0:64) CHAP(4) VAR,
(OP,D,L) ENTRY(CHAR(*)) RETURNS(BIT(1)),
(N,V,E,AS) ENTRY RETURNS(BIT(1));
*/

PROC(CDE) RECURSIVE BIT(1);  
DCL CDE CHAR(*),
CODES(0:4) CHAP(3) STATIC INIT
('ADD', 'SUB', 'MUL', 'DIV'),
OPS(4) CHAR(1) STATIC INIT ('+', '-', '*', '/');
DCL OPX CHAR(4) DEF OPS;

I=I+1;
CDF=CODES(INDEX(OPX,CCARD(I)));  
IF CDE =+ THEN RETURN(I*B);
I=I-1;
```
RETURN("0"');
END DP;

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

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

PROC RECURSIVE BIT(1):
DCL NUM CHAR(P) VAR,
DIG CHAP(1);
/*
 */
/*
 */
NUM="";
IF D(DIG) THEN NUM=DIG; ELSE RETURN("0'B");
R1:
IF D(DIG) THEN 00;
NUM=NUM||DIG;
GO TO R1;
END;
ELSE 00;
K=K+1;
OPST(K)="ENA";
STACK(K)=NUM;
RETURN("1'B");
END N:

/*
*/
V:
PROC RECURSIVE PIT(1);
DCL VAR CHAR(8) VAR,
LETR CHAR(1);
/*
*/
VAR="";
IF L(LETR) THEN VAR=LETR; ELSE RETURN("0'B");
B1:
IF L(LETR) THEN 00;
R2:
VAR=VAR||LETR;
GO TO B1;
END;
IF. D(LETR) THEN GO TO 82;
K=K+1;
STACK(K)=VAR;
OPST(K)="L";
RETURN("1'B");
END V:

/*
*/
E:
PROC RECURSIVE PIT(1);
DCL CP CHAR(3);
DCL KP FIXED PIN(15,0);
/*
*/
IF V\n THEN DO;
  IF OP(CDE1) 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,1,2)=':' THEN 00;
    I=I+1;
    IF E THEN DO:
      IF CCARD(I+1)=' ' THEN GO TO E1;
      RETURN('1'B);
      END;
    END;
  END;
  END;
RETURN('0'B);
END AS;

ON ENDFILE(SYSIN) GO TO R2;
R1:
  READ FILE(SYSIN) INTO(CARD);
  PUT FILE(PRINT) SKIP LIST(CARD);
  NOS,K,T=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;
XI. APPENDIX B
/*
 * GETAPW: ppmc/main optins(main)
 */

/* just update and job transfer program */

#include "header"
? WAYSAMPLE CHAR(12),
? STP CHAR(19) INIT('SAMPLE TAPE NUMBER'),
? STP# CHAR(6) INIT('TP058'),
? SAMP TP CHAR(18) INIT('SAMPLE TAPE NUMBER'),
? SAMP TP# 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(81),
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('/*'),
PARAM CHAR(40) VAR,
WORK1 CHAR(80),
WORK CHAR(80),
SLASH JOB CHAR(4) STATIC INIT('/JOB'),
PRECEDING CARD CHAR(5) STATIC INIT(' '),
(TP058, TP0502, OKS001, 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),
#BATCH FIXED BIN(31,0) INIT(0),
#TRATCHES FIXED BIN(31,0) STATIC INIT(0),
#STARTLOC FIXED BIN(31,0) STATIC INIT(1),
X1 CHAR(81),
X11 CHAR(1) DEF XI,
X12 CHAR(80) DEF XI POS(2),
MODE CHAR(12) VAR,
IDENTIFICATION:

----------------------

PROGRAM-ID: GETSAMPLE.
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 WATFORT 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
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 WATFOR CONTROL
CARD GO TO STEP 9;
6. IF THE CURRENT CARD IMAGE IS A WATFOR JOB CARD THEN UPDATE ITS ACCOUNT FIELD;
7. IF THE CURRENT CARD IMAGE IS A WATFOR /DATA CARD THEN INCREMENT THE WATFOR COUNTERS;
8. IF THE CURRENT CARD IMAGE IS A WATFOR /STOP CARD THEN SAVE THIS FACT FOR FUTURE REFERENCE;
9. IF THE CURRENT CARD IMAGE IS AN OS /STAR CARD THEN UPDATE, STORE, AND REINITIALIZE THE BATCH COUNTERS;
10. WRITE THE UPDATED CARD IMAGE ONTO THE NEW BATCH DATA BASE FILE;
11. REPEAT STEPS 1-10;
12. CLOSE ALL FILES;
13. NUMERICALLY SUMMARIZE THE ACTIONS OF STEPS 1-11 AND OUTPUT THIS SUMMARY ON THE PRINTER;
14. GET A CARD IMAGE FROM THE NEW BATCH DATA BASE FILE;
15. IF AN END-OF-FILE IS ENCOUNTERED GO TO STEP 20;
16. IF THIS IS THE IMAGE OF AN OS CONTROL CARD GO TO STEP 14;
17. WRITE THIS IMAGE ON THE NEW NON-BATCH DATA BASE FILE;
18. WRITE THIS IMAGE ON THE PRINTER;
19. GO TO STEP 14;
20. WRITE A /STOP IMAGE ON THE NEW NON-BATCH DATA BASE FILE;
21. CLOSE ALL FILES AND STOP.

DESCRIPTION OF INPUT:

INPUT MAY COME FROM ONE OF THREE SOURCES: AN INPUT PARAMETER STRING; THE INITIAL DATA BASE FILE; AND/OR THE BATCH DATA BASE FILE.

VALUES FOR ANY OF SEVERAL VARIABLES MAY BE FOUND IN THE INPUT PARAMETER STRING. THE FORM FOR PLACING THESE VALUES IN THE STRING IS VVVVVV=NNNNNN: WHERE, VVVVVV IS THE VARIABLE NAME AND NNNNNN IS THE VALUE TO BE ASSIGNED TO THAT VARIABLE. IF NO VALUE IS GIVEN FOR ANY OF THESE INPUT PARAMETERS THEN...
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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>#IN</td>
<td>100</td>
<td>Specifies the number of batches to be transferred from the initial data base file to the batch and non-batch data base files.</td>
</tr>
<tr>
<td>#PRINTED</td>
<td>3</td>
<td>Specifies the number of transferred batches which will be copied onto the printer.</td>
</tr>
<tr>
<td>MODE</td>
<td>SEQUENTIAL</td>
<td>Specifies the way the batches are to be retrieved from the initial data base.</td>
</tr>
<tr>
<td>SKPBATCH#</td>
<td>0</td>
<td>Specifies which batch in the</td>
</tr>
</tbody>
</table>
BATCH DATA BASE IS NOT TO BE TRANSFERRED TO THE NON-BATCH DATA BASE.

STARTLOC 1 SPECIFIES THE STARTING POSITION FOR THE INITIAL DATA TRANSFER.

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 SEPARATE DEVICES.

THE FOUR OUTPUT CLASSES ARE:
1. COPIES OF PROGRAMS TRANSFERRED FROM THE INITIAL DATA BASE TO THE BATCH DATA BASE;
3. DATA WHICH CREATES THE BATCH DATA BASE;
4. DATA WHICH CREATES THE NON-BATCH DATA BASE.

REFERENCE:
---------

STEPLING, 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.
OPEN FILE(TP0502) RECORD INPUT SEQUENTIAL,
FILE(DSK001) RECORD OUTPUT SEQUENTIAL;
ON ENDFILE(TP0502) GO TO NEXT_STEP;
BATCHT=0;
DO RECORD_COUNT=1 BY 1;
READ FILE(TP0502) INTO (XIN);
IF BATCHT=(SKPTCH*-1) THEN DO;
IF(X2 -= SLASHES )&(X5 -= SLASH_STOP)&(X2^=SLASH_STAR) THEN
WRITE FILE(DSK001) FROM(XIN);
END;
IF X?=SLASH_STAR THEN BATCH=BATCH+1;
END;
NEXT_STEP:
XIN=SLASH_STOP; WRITE FILE(DSK001) FROM(XIN);
XIN=SLASH_STAR; WRITE FILE(DSK001) FROM(XIN);
CLOSE FILE(TP0502), FILE(DSK001);
/*
DEBUG SECTION.
OPEN FILE(DSK001) INPUT RECORD SEQUENTIAL;
ON ENDFILE(DSK001) GO TO THE_END;
DO WHILE(*1'B):
READ FILE(DSK001) INTO(XIN); Z=XIN;
WRITE FILE(PRINT) FROM (Y);
END:
THE_END:
CLOSE FILE(DSK001):
END OF DEBUG SECTION. */
END GETSAMP;
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...
ICACH 030-123  

**PrinErr**

If PrinErr is specified then Proc

**Detectn of this output.**

Proc: NoParagraphs The

Histogram is plotted on The

When Graph Is Specified A

**Parameter (Default Underlined) Significance**

---

**Table of Input Parameters:**

---

**Image.**

Take special action Packer in the reading of the next line

Each Error Which is Encountered Which is

Line tagress Which Are Processed at a time When Possible MCOX4957

- The Data Set Contains The Complet Output For One Entire Sample MD4002X

- The Second Overall The Second Overall The MD4002Y

- When Input Parameters Are Given Below The Second Overall MD4002Z

- The Production Mode. For The Users General The Exact MD4002A

- Input Parameters: All G0 And Or A Second Overall Data Set. The Input MD4002B

- Output This Program Comes From One Of Two Sources. MD4002C

---

**Description of Input:**

- Obtaining tabulation and plots a histogram Of the data.

- Bivariate which are processed a quadratic sort on the previously

- Outputting this processed data on the printer: The

- Processing the data which was gathered in the previous phases MD4003A
OTHERWISE, THIS OUTPUT IS SUPPRESSED.

PRINT_PROG, NOPRINT_PROG

WHEN PRINT_PROG IS SPECIFIED THE WATFOR PROGRAMS WHICH ARE INPUT ARE COPIED ON THE PRINTER.

DEBUG, NODEBUG

WHEN DEBUG IS SPECIFIED A TRACE OF THE EXECUTION OF THE PROGRAM IS GIVEN AS AN AID TO DEBUGGING.

DESCRIPTION OF OUTPUT:

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
PUBLISHING COMPANY. C1968.

FLORES, IVAN. COMPUTER SORTING. ENGLEWOOD CLIFFS, NEW JERSEY. PRENTICE-HALL, INC. C1969.

HOPCROFT, J. E. AND ULLMAN, J. D. ERROR CORRECTION FOR FORMAL LANGUAGES. PRINCETON, NEW JERSEY. PRINCETON UNIVERSITY.


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.

DCL 1 GRAPHX STATIC EXT,
2 HLABLE CHAR(96),
2 DV LABEL CHAR(31) INIT(' 7 7 6 6 5 4 4 3 3 2 2 1 1 0 ')
2 DV LABEL CHAR(31) INIT(' 5 0 5 0 5 0 5 0 5 0 5 0 5 ')
2 DV LABEL CHAR(31) INIT(' 5 0 5 0 5 0 5 0 5 0 5 0 5 '
2 VAXIS CHAR(1) INIT('|'),
2 HAXIS CHAR(96) INIT('|'),
2 H LABEL CHAR(6) INIT('ERRORS'),
2 GRAPH(31,96) CHAR(1),
2 NER(24) CHAR(4) DEF HLABLE POS(1),
2 VLABEL(31) CHAR(1) DEF DV LABEL POS(1),
2 VLABELA(31) CHAR(1) DEF DV LABELA POS(1),
2 VLABEL(31) CHAR(1) DEF DV LABEL POS(1),
2 P POINT, #QFEP(0:24) FIXED BIN(31,0),
2 (I,II,J,K,#QFEP(24) BASED (P)) FIXED BIN(31,0);
DCL ERRORS__ CHAR(33) STATIC INIT
('**EXTENSION**WARNING******ERROR***'),
(TINTAL NUMBER OF COMPILE-TIME ERRORS)

2 LIMIT- CHARACTER  LIMIT

DCL 1 OUTPUT-FILE  UNTAGGED  STATIC

ENTRY'S  UNUSEDED:

XNUMEXP  CHAP(17)  STATIC  INIT

NUMBER  CHAR(I)  DEF  XLINE  POS(105)

(ENTER'S  ARRAY  ARE==)

RAA  CHAR(17)  STATIC  INIT

AA  CHAR(I)  DEF  XLINE  POS(70)

(SEC,  ARRAY  CONE==)

SCLS  CHAR(I)  DEF  XLINE  POS(74)

(SEC,  EXECUTION  TIME==)

SXT  CHAR(I)  DEF  XLINE  POS(25)

CTX  CHAR(13)  DEF  XLINE  POS(2)

ETAE  FIXED  BINARY  INIT(0)

RAGE  FIXED  BINARY  INIT(0)

SAASH-DATA  CHAP(15)  STATIC  INIT

SLASH-JUMP  CHAP(A)  STATIC  INIT

ETAE  CHAR(2)  DEF  XLINE  POS(18)

ETAE  CHAR(4)  DEF  XLINE  POS(18)

ETAE  CHAR(1)  DEF  XLINE  POS(2)

ETAE  CHAR(1)  DEF  XLINE  POS(2)

XC5  CHAR(1)  DEF  XLINE  POS(11)

XC4  CHAR(4)  DEF  XLINE  POS(11)

CAOFN  CHAR(10)  DEF  XLINE  POS(11)

XCHAR(0)  DEF  XLINE  POS(12)

XCHAR(4)  DEF  XLINE  POS(12)

XCHAR(4)  DEF  XLINE  POS(12)

XCHAR(4)  DEF  XLINE  POS(12)

CAOFN=COMPUTAL  CHAP(I)  DEF  XLINE  POS(1)

LINE  CHAR(1)  DEF  XLINE  POS(2)

XLINE  CHAR(1)  DEF  XLINE  POS(2)

E-ERROR  --  CHAR(I)  DEF  ERRORS--POS(23)

E-ERRORS  --  CHAR(1)  DEF  ERRORS--POS(11)

E-WARNING  --  CHAR(I)  DEF  ERRORS--POS(12)
2 #TOT_CERRS FIXED BIN(31,0) INIT(0),
2 ST_2 CHAR(55) INIT
('AVERAGE(MEAN) NUMBER OF COMPILE-TIME ERRORS PER PROGRAM'),
2 #AVE_CERR_PROGRAM FLOAT DEC(15),
2 ST_3 CHAR(49) INIT
('MAXIMUM NUMBER OF COMPILE-TIME ERRORS PER PROGRAM'),
2 #MAX_CERR_PROGRAM FIXED BIN(31,0) INIT(0),
2 ST_4 CHAR(59) INIT
('MINIMUM NUMBER OF COMPILE-TIME ERRORS IN ERRONEOUS PROGRAM'),
2 MIN_CERR_PROGRAM FIXED BIN(31,0) INIT(7000),
2 ST_5 CHAR(45) INIT
('TOTAL NUMBER OF NON-TERMINAL EXECUTION ERRORS'),
2 #TOT_NTEERRS FIXED BIN(31,0) INIT(0),
DCL 1 OUTPUT_RECORD_1A STATIC UNALIGNED,
2 ST_6 CHAR(70) INIT
('AVERAGE(MEAN) NUMBER OF NON-TERMINAL EXECUTION-TIME ERRORS PER PROGRAM'),
2 #AVE_NTE_PROG FLOAT DEC(15),
2 ST_7 CHAR(64) INIT
('MAXIMUM NUMBER OF NON-TERMINAL EXECUTION TIME ERRORS PER PROGRAM'),
2 MAX_NTE_PROG FIXED BIN(31,0) INIT(0),
2 ST_8 CHAR(64) INIT
('MINIMUM NUMBER OF NON-TERMINAL EXECUTION TIME ERRORS PER PROGRAM'),
2 #MIN_NTE_PROG FIXED BIN(31,0) INIT(7000),
2 ST_9 CHAR(46) INIT
('TOTAL NUMBER OF TERMINAL EXECUTION-TIME ERRORS'),
2 #TOT_TERRS FIXED BIN(31,0) INIT(0),
2 ST_10 CHAR(66) INIT
(\"AVERAGE(MEAN) NUMBER OF TERMINAL EXECUTION-TIME ERRORS PER PROGRAM\")

2 #AVE_TERR_PROG FLOAT DEC(6) INIT(0),
2 ST_11 CHAR(61) INIT
(\"MAXIMUM NUMBER OF TERMINAL EXECUTION-TIME ERRORS PER PROGRAM\")

DCL 1 OUTPUT_RECORD_1A UNALIGNED STATIC,
2 ST_12 CHAR(71) INIT
(\"MINIMUM NUMBER OF TERMINAL EXECUTION-TIME ERRORS IN ERRORFUL PROGRAMS\")

2 ST_13 CHAR(22) INIT
(\"TOTAL NUMBER OF ERRORS\")

2 #TOTAL_ERRS FIXED BIN(31,0) INIT(0),
2 ST_14 CHAR(56) INIT
(\"MAXIMUM NUMBER OF ERRORS IN ANY GIVEN ERRORFUL PROGRAM\")

2 #MAXERRS FIXED BIN(31,0) INIT(0),
2 ST_15 CHAR(56) INIT
(\"MINIMUM NUMBER OF ERRORS IN ANY GIVEN ERRORFUL PROGRAM\")

2 #MINERRS FIXED BIN(31,0) INIT(7000),
2 ST_16 CHAR(31) INIT
(\"TOTAL NUMBER OF WATFOR PROGRAMS\")

2 #TOT_PROGS FLOAT DEC(15) INIT(0),

DCL 1 OUTPUT_RECORD_1C STATIC UNALIGNED,
2 ST_17 CHAR(47) INIT
(\"AVERAGE(MEAN) NUMBER OF ERRORS PER PROGRAM\")

2 #AVE_ERRS_PROG FLOAT DEC(15),
2 ST_18 CHAR(34) INIT
(\"TOTAL NUMBER OF WATFOR STATEMENTS\")

2 #TOT_STATEMENTS FLOAT DEC(15) INIT(0),
DCL 1 OUTPUT_RECORD_3 UNALIGNED STATIC,
2 ST_29 CHAR(06) INIT
   ('F/E/J/='), MC0K0298
2 CURRENT_ERR? CHAR(4), MC0K0300
2 ST_30 CHAR(09) INIT
   ('F/E/J/='), MC0K0301
2 PROBH FLOAT DEC(15), MC0K0302
2 ST_31 CHAR(14) INIT
   ('F/E/J/='), MC0K0303
2 PROBH FLOAT DEC(15), MC0K0304
2 ST_32 CHAR(14) INIT
   ('F/E/J/='), MC0K0305
2 PROBH FLOAT DEC(15), MC0K0306
(CURRENT_TOTAL_ERRORS, ETRS, CTRS, NTRS) FIXED BIN(31,0) INIT(0), MC0K0307
DCL #PE STATIC INIT(0299) FIXED BIN(31,0); MC0K0308
DCL ALL_ERRQP_STRING CHAR(364) INIT CALL McC0K0309
DCL ALL_ERROR_STRING2 CHAR(380) INIT MC0K0310
DCL ALL_ERROR_STRING3 CHAR(352) INIT MC0K0311
I-FL1-GL1-TMD-2MD-3MD-4MD-6MD-0MD-1MD-2MD-3MD-4PC-0PC-1PS-0PS-1RE-0RE-1MCDK0334
OSS-1SS-2ST-0ST-1ST-2ST-3ST-4ST-5ST-6ST-7ST-8ST-9ST-ASV-0SV-1SV-2SV-3SV-MCDK0336
-4SV-5SX-0SX-1SX-2SX-3SX-4SX-5SX-6SX-0UN-0UN-1UN-2UN-3UN-4UN-5UN-6UN-7UN-MCDK0337
N-QUN-9'

} STATIC,

    ALL_ERROR_STRING4 CHAR(108) INIT
    ("UV-0UV-1UV-2UV-3UV-4UV-5UV-6VA-0VA-1VA-2VA-3VA-4VA-5VA-6VA-7V"
        MCDK0342
        A-8VA-0VA-AVA-BVA-CVA-DVA-EXT-OXT-1XT-2'"
        MCDK0343
        ) STATIC;

    DCL EXECUTION_TIME_ERROR_STRING CHAR(248) INIT
    ("KO-0KO-1KO-2KO-3KO-4KO-5KO-6KO-7KO-8SR-0SR-1SR-2SR-3SR-4SR-5SR-6SR-7SR-8SR-9SR"
        MCDK0347
        SS-0SS-1SS-2SS-3SS-4SS-5SS-6SS-7SS-8SS-9SS-0SS-1SS-2SS-3SS-4SS-5SS-6SS-7SS-8SS-9SS"
        MCDK0348
        MCDK0349
        ) STATIC;

    DCL SYSPRINTX FILE INT,
    (OSK001, PRINT ) FILE EXT;

    DCL COMPIL_TIME_ERROR_STRING CHAR(1204),
    CTES(299) CHAR(4) DEF COMPIL_TIME_ERROR_STRING POS(1),
    XES(62) CHAR(4) DEF EXECUTION_TIME_ERROR_STRING POS(1),
    NON_TERMINAL_EXEC_ERRORS /* ALSO GARBAGE */
    STATIC CHAR(4);

    DCL (ES1(91) DEF ALL_ERROR_STRING POS(1),
    ES2(95) DEF ALL_ERROR_STRING2 POS(1),
    ES3(98) DEF ALL_ERROR_STRING3 POS(1),
    ES4(27) DEF ALL_ERROR_STRING4 POS(1),
    ES5(299) FIXED BIN(31,0) STATIC,
    ESX CHAR(1204) STATIC,
    ESX(299) CHAR(4) DEF ESX POS(1),
    ((ES1A(91) DEF ESX POS(1),
    ES2A(95) DEF ESX POS(365),
    ES3A(88) DEF ESX POS(745),
    ES4A(27) DEF ESX POS(1001)) CHAR(4),
    #ES(299) FIXED BIN(31,0) STATIC,
    ESX CHAR(1204) STATIC,
    ESX(299) CHAR(4) DEF ESX POS(1),
    (ES1A(91) DEF ESX POS(1),
    ES2A(95) DEF ESX POS(365),
    ES3A(88) DEF ESX POS(745),
    ES4A(27) DEF ESX POS(1001)) CHAR(4),
WORK CHAR(40) STATIC,
(PRINT_ERR, PRINT_PPROG, DEBUG, NOGRAPH) BIT(1),
LKp FIXED BIN(31, 0),
III FIXED BIN(32, 0),
PARM CHAR(40) VAR:

DCL
1 SUBLISTS(18, 18) STATIC UNALIGNED,
   2 A FIXED BIN(32),
   2 B CHAR(4),
1 TMP STATIC UNALIGNED,
   2 A FIXED BIN(32),
   2 B CHAR(4),
1 LEAST_TABLE(18) STATIC UNALIGNED,
   2 A FIXED BIN(32),
   2 B CHAR(4),
XOUTT CHAR(80),
#XNER(24) FIXED BIN(32) DEF #ES,
XNER CHAR(4) DEF ESX POS(1),
RECORD_COUNT_DSK001 FIXED BIN(32),
ULINE CHAR(091) DEF XLINE POS(2),
(N, NN INIT(297)) STATIC FIXED BIN(32):
/
* INITIALIZATION PHASE.
*

GRAPH=' *';
OPEN FILE(DSK001) RECORD INPUT SEQUENTIAL,
FILE(PRINT) STREAM PRINT OUTPUT
PAGESIZE(55) LINESIZE(120),
FILE(SYSPPRIN) STREAM PRINT OUTPUT PAGESIZE(55)
TITLE(*SYSPPRIN*)
LINESIZE(120);
#ES=0;
ON ENDPAGE(SYSPPRIN) PUT FILE(SYSPPRIN) PAGE EDIT((20) /*),
   'DEBUG OUTPUT', (20) /*) (COL(25), 3 A);
ON ENDPAGE(PRINT) BEGIN;
   PUT FILE(PRINT) EDIT((20) /*, 'ERROR CLASSIFICATION',
      (20) /*) (COL(25), 2 A) PAGE;

DUT PÎLE(OPTNT) LTME(3); EMH;
ES?Û=ES1; ES2A=ES2; ES3A=ES3; ES4A=ES4;
CTES=ES;
DO I=1 TO DIM(XES,1); DO J=1 TO #PE: IF CTES(J)=XES(I) THEN
CTES(J)=1: END: END;
WORK=PARM;
NOGRAPH=INDEX(WORK,'NOGRAPH') >0;
PRINT_ERR=(INDEX(WORK,'PRINT_ERR') >0) & (INDEX(WORK,'NPRINT_ERR') =0);
PRINT_PROG=(INDEX(WORK,'PRINT_PROG') >0) &
(INDEX(WORK,'NPRINT_PROG') =0);
DEBUG=(INDEX(WORK,'DEBUG') >0) & (INDEX(WORK,'NODEBUG') =0);
IF DEBUG THEN SIGNAL ENDPAGE(SYSPRINX);
IF PRINT_ERR THEN SIGNAL ENDPAGE(PRINT);
IF DEBUG THEN PUT FILE(SYSPRINX) EDIT(ES)
(SKIP,COL22)10 (A,X(2))1;
/* END OF INITIALIZATION PHASE.

SUMMARIZATION PHASE. */

ON ENDFILE(OSK001) BEGIN;
ON ERROR SNAP GO TO CATCH;
ON ENDPAGE(PRINT)
PUT FILE(PRINT) EDIT((20) '*','WATERED SUMMAR' ,(20) '*')
(COL23),2 (A) PAGE;
PUT FILE(SYSPRINX) EDIT(*EOF AFTER*,RECORD_COUNT.OSK001,
'RECORDS1) (SKIP(DEBUG),A(DEBUG*9),F(DEBUG*6),A(DEBUG*9)));
#AVE_CERR_PROGRAM=#TOT_CERRS/#TOT_PROGS;
#AVE_NTE_PROG=#TOT_NTEERS/#TOT_PROGS;
#AVE_TERP_PROG=#TOT_TERRES/#TOT_PROGS;
#AVE_ERRS_PROG=#TOTAL_ERRS/#TOT_PROGS;
#AVE_ERRS_STMT=#TOTAL_ERRS/#TOT_STATEMENTS;
#AVE_ERRS_CAPD=#TOTAL_ERRS/#TOT_CAPDS;
PROB_OF_ERROR1=#EPPONEOUS_PROG/#TOT_PROGS;
PROB Of_ERROR2=#PROG.Get_ERR/#TOT_PROGs;
#AVE_ERROR_PROG=#TOTAL_PROGS/#ERRONEOUS_PROG;

CATCH:

ON ERROR SYSTEM;
SIGNAL ENDPAGE(PRINT);
PUT FILE(PRINT) LINE(5) EDIT
 (OUTPUT_RECORD_1,OUTPUT_RECORD_1A,OUTPUT_RECORD_1B,
 OUTPUT_RECORD_1C
 (SKIP,COL(2),A,COL(107),F(12,5));
PUT FILE(PRINT) EDIT(OUTPUT_RECORD_1D)
 (SKIP,COL(2),A,COL(110),F(9,7));
PUT FILE(PRINT) SKIP;
DO I=1 BY 1 TO #PE:
IF #ES(I)=0 THEN GO TO LBL1;
CURRENT_EPR=ES(I);
PROB2(#ES(I)/#TOT_PROG; /* P(E/I) */
IF (PROB3=0) OR (PROB4=0) THEN
DO;
SIGNAL ENDPAGE(PRINT);
PUT FILE(PRINT) EDIT(OUTPUT_RECORD_2) (SKIP(3),COL(25),A,
COL(70),A);
END;
DO J=1+1 BY 1 TO #PF WHILE(250=0);
 CURRENT_EPR=ES(J);
PROB4=#ES(J)/#TOT_PROGs; /* P(E/J) */
PROB5=PROB3*PROB4; /* P(E/I |E/J) */
PROB6=PROB5/PROB3; /* P(E/J |E/I) */
IF (PROB4=0) OR (PROB5=0) OR (PROB6=0) THEN
PUT FILE(PRINT) EDIT(OUTPUT_RECORD_3) (SKIP
 COL(17),A,COL(75),A);
KPNKR:
END;
LBL1:
END;
GO TO GRAPHER;
END;

/*
END OF SUMMARIZATION PHASE.

EXISTENCE AND CLASSIFICATION PHASES.

CLASSIFY: BEGIN:
ON ENDPAGE(SYSPRINX) PUT FILE(SYSPRINX) PAGE EDIT
((20) "" /* Waiting for Programs */(20) "" ) (COL(25), 3 A);
IF PRINT_PROG THEN SIGNAL ENDPAGE(SYSPRINX);
DO RECORD_COUNT=SK001=1 BY 1;
READ FILE(DSK001) INTO (XLIN);
#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 X4=SLASH_JOB THEN DO:
   #TOT_STATEMENTS=#TOT_STATEMENTS-1;
   INPGFLG=1:
   GO TO TMPOUT;
END:
IF INDEX(ERRORS, EFLAG1) = 0 THEN
ERROR_LINE:
   DO;
   /* ERROR CLASSIFICATION. */
   II=(INDEXESX,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;
CURRENT_TOTAL_ERRORS=
ERFLG=ERFLG+1;
#TOTAL_ERRS=#TOTAL_ERRS+1;
IF INDEX(EXECUTION_TIME_ERROR_STRING,ES(I))=0 THEN DO;
#TOT_TERPS=#TOT_TERPS+1; ETRS=ETRS+1; END;
IF INDEX(COMPIL_TIME_ERROR_STRING,ES(I))=0 THEN DO;
#TOT_CERRS=#TOT_CERRS+1; CTRS=CTRS+1; END;
IF INDEX(NON_TERMINAL_EXEC_ERRORS,ES(I))=0 THEN DO;
#TOT_NTERPS=#TOT_NTERPS+1; NTRS=NTRS+1; END;
END;
IF (CTX) E(SXT=SXT) & (SOC=SOCI) & (BA=BA) & (UNUSED=UNUSED) THEN DO;
PUT FILE(SYSPPINX) EDIT('EOF ENCOUNTERED');
( (SKIP(DEBUG),A(DEBUG*15)));
IF ERFLG>0 THEN DO:
#ERRONEOUS_PROG=#ERRONEOUS_PROG+1;
#PROG_GT1ERR=#PROG_GT1ERR+(ERFLG>1);
/* MAX & MIN ERROR */
#MAX_CFPR_PROGRAM=MAX(CTRS,#MAX_CFPR_PROGRAM);
IF CTRS=0 THEN
#MIN_CFPR_PROGRAM=MIN(CTRS,#MIN_CFPR_PROGRAM);
#MAX_NTE_PROGRAM=MAX(#MAX_NTE_PROGRAM,NTPS);
IF NTPS=0 THEN
#MIN_NTE_PROGRAM=MIN(#MIN_NTE_PROGRAM,NTPS);
#MAX_TERR_PROGRAM=MAX(#MAX_TERR_PROGRAM,ETRS);
IF ETRS=0 THEN
#MIN_TERR_PROGRAM=MIN(#MIN_TERR_PROGRAM,ETRS);
#MAX_ERRS=MAX(#MAX_ERRS,CURRENT_TOTAL_ERRORS);
#MIN_ERRS=MIN(#MIN_ERRS,CURRENT_TOTAL_ERRORS);
END;
#TOT_CARDS=#TOT_CARDS-1;
#TOT_STATEMENTS=#TOT_STATEMENTS-1;
#TOT_PROGS=#TOT_PROGS+1;
ETRS,CTRS,NTRS,CURRENT_TOTAL_ERRORS,
INPGFLG,
ERFLG=0;
GO TO NXTLINE;
END:

TMPOUT:
OUT FILE(SYSPRINX) 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 NOT GRAPH THEN
DO:
SIGNAL ENDPAGE(SYSPRINX);
P=ADDR('#ONEP(1)));
#ONEP(0)=0;
PUT FILE(SYSPRINX) EDIT('ENTER GRAPHER')
A(DESC(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 ((#ES(I)=0)*PRINT_ERR *4),
X((#ES(I)=0)*PRINT_ERR*X2)
,F((#ES(I)=0)*PRINT_ERR*X4),
X((#ES(I)=0)*PRINT_ERR*X4));
/* SORT TO FIND THE MOST FREQUENT ERRORS. */
/* QUADRATIC SORT. THE TECHNIQUE IS IN FLORES' BOOK. */
PUT FILE(SYSPRINX) EDIT('ENTER SORT.')
(SKIP(DEBUG),A(DEBUG*12));
SUBLISTS.A=20000; SUBLISTS.B=1;
LEAST_TABLE.A=20000; LEAST_TABLE.B=1;
DO I=1 TO 18;
DO J=1 TO 18;
N=18*(I-1)+J;
IF N>NEND THEN GO TO NXTRC;
TMP.A=ES(N);
TMP.B=FS(N);
IF TMP.A<LEAST_TABLE.A(I) THEN LEAST_TABLE(I)=TMP.A BY NAME;
SUBLISTS(I,J)=TMP.B BY NAME;
END;

NXTRC:
END;
PUT FILE(SYSTPLNX) 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(SYSTPLNX) EDIT
/* COMPLETE SORT */
(SKIP(3,6), A(14*DEBUG));
ON ENDPAGE(PRINT) PUT FILE(PRINT) PAGE
   EDIT((201 '*'), 'HISTOGRAM OF ERROR DISTRIBUTION', (20) '*'):
   (COL(25), A);
   NER=XNER; HNFR=#XNER;
   DO K=1 BY 1 TO 24;
      DO J=1 BY 1 TO 96;
         II=32-J;
         III=II*2.85;
         IF III>#NER(K) THEN GO TO NXTPT;
         IF (III<=#NER(K)) & (III>#NER(K-1)) THEN
            GRAPH(I,J)=';
            IF MOD(J,4)=0 THEN GRAPH(I,J)='
NXTPT:
      END;
   END;
   END;
   END;
   SIGNAL ENDPAGE(PRINT);
   PUT FILE(PRINT) LINE(10);
   DO I=1 TO 31;
      PUT FILE(PRINT) EDIT
         (VLABEL(I), VLABEL(I), VAXIS, GRAPH(I,*))
         (SKIP, ?A,X('),07 A);
   END;
   PUT FILE(PRINT) EDIT
      (HAXIS, NER, HLABEL)
      (SKIP, COL(5), A, SKIP, COL(5), 24 A(4), SKIP(2), COL(45), A);
   END;
/* END OF GRAPHING PHASE */
CLOSE FILE(PRINT), FILE(DSK00!), FILE(SYSPRINX);
END ANLY7;
TMAT(0:5,0:4) FIXED BIN(0:15,0) EXIT STATIC

\*

RESET ENTRY

RETURN ENTRY

CERTIFICATION ENTRY \[\text{CHAR}(*)\] RETURN(CHAR(80)) VAR

""""""""""\text{ERRORP}***\text{WARPING}**\text{EXTENSION}***\"

ERROPS \text{CHAR}(33) STATIC INIT

\(1\) FIELD \text{CHAR}(11) DEF XLINE POS(2)

SLASH JOB \text{CHAR}(4) STATIC INIT(\text{JOB})(1)

SLASH STOP \text{CHAR}(5) STATIC INIT(\text{STOP})(1)

SLASH DATA CHAR(15) STATIC INIT(\text{DATA})(1)

XCHAR \text{CHAR}(80) BASE(0)

SAVECHAR \text{CHAR}(80) BASE(0)

FILE INIT / * COPY OUTPUT NEW INPUT

FILE EXIT / * COPY OUTPUT DEBUG OUTPUT

NEW MAP \text{CHAR} DATA SET / *

NEW MAP \text{CHAR} MAP PROGRAM

NEW MAP \text{CHAR} MAP IN

\text{NEW MAP} \text{CHAR} MAP OUT

COPY OUT DEBUG BIT(1)

COPY NEW PROGRAMS / * PRINT NEW PROGRAMS

COPY STDIN BIT(1) / *

COPY OUT DEBUG BIT(1)

COPY OUT DEBUG BIT(1)

COPY OUT DEBUG BIT(1)

COPY OUT DEBUG BIT(1)

I PARMS STATIC UNALIGNED EXIT

DCL \text{CHAR}(80) VAR

DCL \text{CHAR}(80) OPT(D(\text{MAIN}))

SUBPGM SIZE:

\text{ERROR CORRECTION PROGRAM} / *
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.
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
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 CORRECTED FORTRAN STATEMENT ON THE TEMPORARY NEW PROGRAM OUTPUT FILE;
10. IF THERE HAVE BEEN ANY ERRORS FLAGGED IN THE PROGRAM'S OUTPUT THEN SET THE ERROR FLAG;
11. OUTPUT THE PROGRAM'S DATA TO THE NEW PROGRAM FILE;
12. IF THE ERROR FLAG IS SET INVOKE THE CORRECTN MODULE;
   OTHERWISE, INVOKE THE RESET MODULE;
13. RESET THE ERROR FLAG;
14. GO TO STEP 3.

DESCRIPTION OF INPUT:
---------------------

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

DESCRIPTION OF OUTPUT:
----------------------
THERE ARE TWO BASIC CLASSES OF OUTPUT FROM THIS JOB. THE FIRST CLASS IS THE STANDARD OUTPUT. STANDARD OUTPUT IS ALWAYS PRODUCED AND IS IN THE FORM OF A NEW PROGRAM INPUT DATA SET FOR THE WATFOR COMPILER.
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.

TABLE OF INPUT PARAMETERS:

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBUG.</td>
<td>REQUESTS DEBUG OUTPUT.</td>
</tr>
<tr>
<td>COPYINPUT.</td>
<td>REQUESTS THAT ALL INPUT BE COPIED ONTO THE PRINTER.</td>
</tr>
<tr>
<td>COPYOUTPUT.</td>
<td>REQUESTS THAT THE DISK OUTPUT BE COPIED ONTO THE PRINTER.</td>
</tr>
<tr>
<td>PRNT(PRINT).</td>
<td>REQUESTS THAT ADDITIONAL PRINTED OUTPUT BE GIVEN.</td>
</tr>
</tbody>
</table>

DESCRIPTION OF FILES:

<table>
<thead>
<tr>
<th>FILE</th>
<th>USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSK001.</td>
<td>INPUT FILE WHICH CONTAINS WATFOR PROGRAM AND DATA.</td>
</tr>
<tr>
<td>NEWPGM.</td>
<td>TEMORARY NEW PROGRAM FILE; CONTAINS ONE NEW PROGRAM AS IT IS BEING CORRECTED.</td>
</tr>
<tr>
<td>PGMNEW.</td>
<td>THE FILE WHICH (AT THE TERMINATION OF THE CORRECTOR) CONTAINS THE NEW BATCH OF FORTRAN PROGRAMS.</td>
</tr>
<tr>
<td>PRINT,SYSPRINX.</td>
<td>FILES WHICH CONTAIN THE VARIOUS TYPES OF</td>
</tr>
</tbody>
</table>
**DESCRIPTION OF VARIABLES AND LABELS:**

<table>
<thead>
<tr>
<th>VARIABLE OR LABEL</th>
<th>MEANING OR USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CARD, XCARD, SAVECARD</td>
<td>CARD IMAGES.</td>
</tr>
<tr>
<td>COPY, INPUT, COPYOUTPUT</td>
<td>INPUT PARAMETERS.</td>
</tr>
<tr>
<td>COPYOUTPUT, DEBUG, PRNT</td>
<td></td>
</tr>
<tr>
<td>PARMS, PARM.</td>
<td>OTHER CORRECTOR MODULES.</td>
</tr>
<tr>
<td>CRRCTN, CRRCTR, RESET</td>
<td>ERROR FLAGS.</td>
</tr>
<tr>
<td>DFLG, EFLAG1, EFLG</td>
<td>FILES USED WITHIN THE CORRECTOR PROGRAM.</td>
</tr>
<tr>
<td>DSK001, NEWPGM, PGMNEW, PRINT</td>
<td></td>
</tr>
<tr>
<td>SYSPRNX, WTO.</td>
<td>PLACE TO WHICH CONTROL IS TRANSFERRED WHEN AN ERROR LINE IS ENCOUNTERED IN THE PROGRAM LISTING.</td>
</tr>
<tr>
<td>ERROR_LINE.</td>
<td>A LIST OF THE MOST GENERAL WATFOR ERROR FLAGS.</td>
</tr>
<tr>
<td>ERRORS...</td>
<td>LISTING AND OUTPUT LINE IMAGES.</td>
</tr>
<tr>
<td>P, Q.</td>
<td>POINTERS USED FOR OVERLAY DEFINING.</td>
</tr>
<tr>
<td>QUIT, QUITR.</td>
<td>LABELS USED AT JOB TERMINATION.</td>
</tr>
<tr>
<td>RD1, RD2, RD3.</td>
<td>LABELS WHERE SELECTED READ PROCESSES BEGIN.</td>
</tr>
<tr>
<td>S2, S4, S5, S5A, S5B</td>
<td>PLACES WHERE STEPS OF THE ORIGINAL (NOT CURRENT) ALGORITHM BEGAN.</td>
</tr>
<tr>
<td>S11, S15.</td>
<td>WATFOR CONTROL INFORMATION.</td>
</tr>
<tr>
<td>SLASH, SLASH_DATA</td>
<td></td>
</tr>
<tr>
<td>SLASH_JOB, SLASH_STAR</td>
<td></td>
</tr>
<tr>
<td>SLASH_STOP.</td>
<td>THE FIRST N CHARACTERS OF A CARD IMAGE.</td>
</tr>
<tr>
<td>A_CN, N = 1, 2, 3, 4, 5</td>
<td></td>
</tr>
</tbody>
</table>

**SYSPRINT.**

PRINTED OUTPUT.

**WTO.**

INPUT FILE WHICH CONTAINS FORTRAN PROGRAMS' LISTINGS AND OUTPUT.
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;
CALL TFILL;
/* SET UP PARAMETER FLAGS. */
DEBUG=INDEX(PARM,'DEBUG')>0;
COPYOUTPUT=INDEX(PARM,'COPYOUTPUT')>0;
COPYINPUT=INDEX(PARM,'COPYINPUT')>0;
PRNT=INDEX(PARM,'PRINT')>0;
/* OPEN ALL FILES. */
OPEN FILE(WTO) RECORD SEQUENTIAL INPUT, /* WATFOR OUTPUT */
FILE(DSK001) RECORD SEQUENTIAL INPUT, /* WATFOR INPUT */
FILE(NEWPGM) RECORD SEQUENTIAL OUTPUT, /* NEW PROGRAMS */
FILE(PRINT) STREAM PRINT LINESIZE(120) PAGESIZE(55),
FILE(SYSPRINX) STREAM PRINT LINESIZE(120) PAGESIZE(55)
TITLE("SYSPRINX");
/* ENDPAGE ACTIONS. */
ON ENDPAGE(PRINT) PUT FILE(PRINT) PAGE EDIT
((45) "*", 'DEBUG OUTPUT', (45) "*");
(SKIP(2), 3 A);
ON ENDPAGE(SYSPRINX) PUT FILE(SYSPRINX) EDIT
((45) "*", 'COPY FILE', (45) "*");
(SKIP(2), 3 A);
IF COPYOUT|DEBUG THEN SIGNAL ENDPAGE(PRINT);
IF COPYINPUT|PRNT THEN SIGNAL ENDPAGE(SYSPRINX);
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(NEWPGM) FRCM(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
READ FILE(WTO) INTO(XLINE); 00003750
S2:
IF SAVECARD=CARD THEN PUT FILE(PRINT) EDIT('***JCL ERROR***') (SKIP, A); 00003760
(SaveCard, A); 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
S4:
PUT FILE(PRINT) EDIT(*1*,XCARD)
  (SKIP(DEBUG),A(DEBUG),A(80*DEBUG));
WRITE FILE(NEWPGM) FROM(XCARD);
S5:
SAVECARD=CARD;
READ FILE(WTO) INTO(XLINE);
S5A:
PUT FILE(PRINT) EDIT(CARD) (SKIP(COPYINPUT),A(COPYINPUT*80));
PUT FILE(SYSPRINTX) EDIT(LINE)
  (SKIP(COPYOUTPUT),A(COPYOUTPUT*120));
IF XC4=SLASH_JOB THEN GO TO S15;
IF XC5=SLASH_DATA THEN GO TO S11;
IF XC5=SLASH_STOP THEN GO TO S15;
IF XC2=SLASH_STAR THEN GO TO QUIT;
IF INDEX(ERRORS,EFLAG1)=0 THEN
  DO;
    EFLAG='1'B;
    XCARD=CRRCTR(SAVECARD,XLINE);
    PUT FILE(PRINT) EDIT(*2*,XCARD)
      (SKIP(DEBUG),A(DEBUG),A(80*DEBUG));
    IF XCARD=' ' THEN
      WRITE FILE(NEWPGM ) FROM(XCARD);
      SAVECARD=XCARD;
  END;
ERROR_LINE:  DO;
  EFLAG='1'B;
  XCARD=CRRCTR(SAVECARD,XLINE);
  PUT FILE(PRINT) EDIT(*3*,XCARD)
      (SKIP(DEBUG),A(DEBUG),A(80*DEBUG));
      WRITE FILE(NEWPGM ) FROM(XCARD);
      IF XC5=SLASH_DATA THEN
        Go TO S15;
      /* PREVENT MULTIPLE CORRECTIONS FOR DEPENDENT ERRORS. */
      IF INDEX(ERRORS,EFLAG1)>0 THEN GO TO S5B;
      ELSE GO TO S5A;
      END;
S5B:
READ FILE(WTO) INTO(XLINE);
IF XC5=SLASH_DATA THEN GO TO S11;
/* ANY ERRORS IN RESULTS. */
RD1: READ FILE(WTO) INTO(XLINE);
    PUT FILE(SYSPRINX) EDIT(LINE)
        (SKIP(COPYOUT),A(COPYOUT*120));
    IF INDEX(ERRORS,EFLAG1) = 0 THEN EFLG = '1' B;
    IF(XC4 = SLASH_JOB) & (XC5 = SLASH_STOP) THEN GO TO RD1;
    /* OUTPUT DATA TO NEWPROGRAM FILE */
    READ FILE(DSK001) INTO(SAVECARD);
    PUT FILE(PRINT) EDIT(SAVECARD)
        (SKIP(COPYINPUT),A(COPYINPUT*80));
    CARD = SAVECARD;
    IF XC5 = SLASH_DATA THEN GO TO RD2;

RD2: READ FILE(DSK001) INTO(SAVECARD);
    PUT FILE(PRINT) EDIT(SAVECARD)
        (SKIP(COPYINPUT),A(COPYINPUT*80));
    CARD = SAVECARD;
    IF XC5 = SLASH_DATA THEN GO TO RD2;

RD3: READ FILE(DSK001) INTO(SAVECARD);
    PUT FILE(PRINT) EDIT(SAVECARD)
        (SKIP(COPYINPUT),A(COPYINPUT*80));
    CARD = SAVECARD;
    IF XC1 = SLASH THEN DO;
        WRITE FILE(NEWPGM) FROM(SAVECARD);
        GO TO RD3;
    END;

S15: IF EFLG THEN CALL CRRCTN;
    ELSE CALL RESET;
    GO TO S2;

QUIT: RETURN;
    END CORRECT;
EXECUTION ANALYSIS AND CORRECTION MODULE

CRRCTN: PROC;

DCL

DICTIONARY(255) CHAR(11) VAR,
1 PARMS STATIC UNALIGNED EXT,
2 (DEBUG,COPYOUTPUT,COPYINPUT,PRNT) BIT(1),
XDICT 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('O'B),
JvJJ) FIXED BIN(15,0),
PRINT FILE EXT,
SPECIALX CHAR(18) STATIC INIT('®%$*<|&>;:~? '),
SPECIAL(18) CHAR(1) DEF SPECIALX POS(1);
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;
2. READ ONE WORD ON TEXT;
3. IF THE WORD IS THE LEFT HAND SIDE OF A FORTRAN STATEMENT, ENTER THE WORD INTO THE DICTIONARY AND GO TO STEP 2;
4. IF THE WORD IS LESS THAN THREE CHARACTERS, OR IF THE WORD IS ALREADY IN THE DICTIONARY GO TO STEP 2;
5. SET UP THE CHARACTER REGISTER AND CHARACTER COUNT;
6. IF THE CHARACTER COUNT OF THE INPUT WORD AND THE CHARACTER COUNT OF THE DICTIONARY WORD DIFFER BY MORE THAN 2 THEN GO TO STEP 9;
7. IF THE CHARACTER REGISTERS DIFFER IN MORE THAN TWO POSITIONS THEN GO TO STEP 9;
8. GO TO ALGORITHM -1 IF THE ENTRY WORD IS LARGER THAN THE DICTIONARY WORD;
9. UPDATE DICTIONARY COUNT;
10. IF THERE ARE MORE DICTIONARY ENTRIES GO TO STEP 5;
11. IF THERE IS MORE INPUT GO TO STEP 2;
12. RETURN FROM THIS MODULE;

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

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

DESCRIPTION OF INPUT:

INPUT FOR THIS PROGRAM COMES FROM CALLS TO THE GETR PROCEDURE. EACH CALL TO THE GETR PROCEDURE OBTAINS ONE WORD OF TEXT. WHEN THERE IS NO MORE TEXT THIS PROGRAM IS SIGNalled FROM GETR VIA THE FLG PARAMETER (SIDE PARAMETER), AND
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:
-------------------------------------

<table>
<thead>
<tr>
<th>VARIABLE OR LABEL</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALGX,X=0,1,-1.</td>
<td>THE START OF THE X-TH SPELLING CORRECTION ALGORITHM.</td>
</tr>
<tr>
<td>CHARREG,DCREG,DREGC,REGCHAR.</td>
<td>CHARACTER REGISTERS.</td>
</tr>
<tr>
<td>CHARS.</td>
<td>STRING OF POSSIBLE INPUT CHARACTERS.</td>
</tr>
<tr>
<td>CNT.</td>
<td>NUMBER OF CHARACTERS IN THE WORD CURRENTLY UNDER ANALYSIS.</td>
</tr>
<tr>
<td>COPYINPUT,COPYOUTPUT,DEBUG,PRNT,PARMS.</td>
<td>EXTERNAL INPUT PARAMETERS.</td>
</tr>
<tr>
<td>DCNT.</td>
<td>THE LENGTH OF THE CURRENT DICTIONARY WORD.</td>
</tr>
<tr>
<td>DICTIONARY.</td>
<td>THE SYMBOL TABLE.</td>
</tr>
<tr>
<td>DICTLEN.</td>
<td>THE NUMBER OF WORDS IN THE DICTIONARY</td>
</tr>
<tr>
<td>DNO,DN1,GET,GIT,RTRN.</td>
<td>STATEMENT LABEL CONSTANTS.</td>
</tr>
<tr>
<td>FLG.</td>
<td>A FLAG WHICH INDICATES WHETHER OR NOT A CORRECTION HAS BEEN MADE.</td>
</tr>
</tbody>
</table>
I,II,J,JJ.
PRINT.
GETR,PUTR.
REALGOOF.
SET,UNSET.
SIDE.
SPECIAL, SPECIALX.
TEXT.
XCHARS.
XDIC.

REFERENCES:


HOPCROFT, J. E. AND ULLMAN, J. D. ERROR CORRECTION FOR FORMAL LANGUAGES. PRINCETON, NEW JERSEY. PRINCETON UNIVERSITY 1969.


/* CREATE THE DICTIONARY. */
PUT FILE(PRINT) EDIT('ENTER CRRCTN')
   (SKIP(DEBUG),A(13*DEBUG));
CALL SET;
IF DEBUG THEN ON STRG PUT FILE(PRINT) EDIT('***',TEXT,TSTR)
   (SKIP,4 A(11));
DICTIONARY='';
CALL DFILL(DICTIONARY);
/*
I=0;
PUT FILE(PRINT) EDIT('DICTIONARY IS SET UP.')
   (SKIP(DEBUG),A(DEBUG*22));
GO TO GET;
*/
TEXT=TEXT||TSTR;
CALL PUTR(TEXT,FLG);
GET:

FLG='0'B;
CALL GETR(TEXT,SIDE);
PUT FILE(PRINT) EDIT(*T&$='*,TEXT,SIDE)
(SKIP(DEBUG),A(DEBUG*4),A(DEBUG*LENGTH(TEXT)),A(DEBUG));
TSTR=SUBSTR(TEXT,LENGTH(TEXT),1);
TEXT=SUBSTR(TEXT,1,LENGTH(TEXT)-1);
TXT=TEXT;
IF SIDE='E' THEN GO TO RTRN;
IF SIDE='L' THEN DO;
   DICTLEN=DICTLEN+1;
   DICTIONARY(DICTLEN)=TEXT;
   GO TO GIT;
   END;
IF LENGTH(TEXT)<3 THEN GO TO GIT;
XDIC=STRING(DICTIONARY);
IF INDEX(XDIC,TEXT)^=0 THEN GO TO GIT;
/* ACTUAL CORRECTION PROCESS FOLLOWS. */
CNT=LENGTH(TEXT);
DO I=28 TO 3 BY -1;
   REGCHAR(I)=INDEX(TEXT,CHARS(I))>0;
END;
/* FINISH TEXT CHARACTER REGISTER SETUP. */
DO I=0 TO 9 BY 1;
   II=I;
   REGCHAR(2)=(REGCHAR(2))||INDEX(TEXT,II)>0);
END;
/* CHECK FOR SPECIAL CHARACTERS IN TEXT WORD. */
DO I=1 TO DIM(SPECIAL,1);
   REGCHAR(1)=(REGCHAR(1))||INDEX(TEXT,SPECIAL(I))>0);
END; /* OF TEXT CHARACTER REGISTER SETUP. */
/* REPEAT ABOVE PROCESS AS DICTIONARY SEARCH IS MADE. */
DO I=1 TO DICTLEN; /* START DICTIONARY SEARCH. */
   DCNT=LENGTH(DICTIONARY(I));
/* DICTIONARY CHARACTER REGISTER SETUP. */
/* ALPHABETIC TEST. */
ALGORITHM 0: NUMBERS REPRESENT THE POSITION OF A CHARACTER IN THE ALPHABET. THE END OF THE ALGORITHM IS INDICATED BY A RETURN TO THE BEGINNING OF THE PROGRAM.

ALGORITHM 1: DICTIONARY ENTRY LONGER THAN THE INPUT TEXT WORD. A TEMPORARY WORKING VARIABLE IS USED TO HOLD THE DICTIONARY WORD. THE FIRST DIFFERENCE CHARACTER OF THE DICTIONARY WORD IS DISCARDED AND
GO TO DNO;
END;

DNO:
TEXT=SUBSTR(TEXT,1,CNT-1);
/* END OF ALGORITHM 0. */

ALG(0):
/* ALGORITHM 0: NUMBER OF CHARACTERS ARE EQUAL. */
/* ALGORITHM 0 REQUIRES THAT THE INPUT TEXT WORD AND THE DICTIONARY WORD HAVE THE SAME NUMBER OF CHARACTERS. IF THE TWO WORDS DIFFER IN EXACTLY ONE POSITION THE DICTIONARY ENTRY IS ACCEPTED AS THE CORRECT WORD. IF TWO ADJACENT POSITIONS DIFFER THEN THE CHARACTERS IN THESE TWO POSITIONS ARE INTERCHANGED AND COMPARISON PROCEEDS AS ABOVE. */
IF (I=1) MOD(I,25) = 0 THEN
PUT FILE(PRINT) EDIT('ENTER ALG 0')
(SKIP(DEBUG),A(DEBUG*12));
IF SUM((OREGC£(-'REGCHAR') ) )<=1 THEN TEXT=DICTIONARY(I); ELSE DO J=1 TO DCNT-1 WHILE ((LENGTH(TEXT)=LENGTH(DICTIONARY(I))))
IF SUBSTR(TEXT,J,1)=SUBSTR(DICTIONARY(I),J,1) THEN DO; /* CHARACTER INVERSION. */
SIDE=SUBSTR(TEXT, J,1);
SUBSTR(TEXT, J,1)=SUBSTR(TEXT, J+1,1);
SUBSTR(TEXT, J+1,1)=SIDE;
PUT FILE(PRINT) EDIT(TEXT,DICTIONARY(I))
(SKIP(DEBUG),A(DEBUG*LENGTH(TEXT)), A(DCNT*DEBUG));
END;
END;
GO TO REALGOOF;
/* END OF ALGORITHM 0. */

ALG(1):
/* ALGORITHM 1: DICTIONARY ENTRY LONGER. */
/* ALGORITHM 1 IS USED WHEN THE DICTIONARY WORD IS ONE CHARACTER LONGER THAN THE INPUT TEXT WORD. A TEMPORARY WORKING VARIABLE IS USED TO HOLD THE DICTIONARY WORD. THE FIRST DIFFERENCE CHARACTER OF THE DICTIONARY WORD IS DISCARDED AND...*/
THE REMAINING CHARACTERS ARE SHIFTED LEFT ONE POSITION. FROM

This point the algorithm proceeds as above. */

IF (I=1) AND (MOD(I, 25) = 0) THEN

PUT FILE(PRINT) EDIT("ENTER ALG 1")

(SKIP(DEBUG),A(DEBUG*12));

XDIC=DICTIONARY(I);

DO J=1 TO DCNT;

IF SUBSTR(TEXT, J, 1) != SUBSTR(XDIC, J, 1) THEN DO;

DO JJ=J TO DCNT-1;

SUBSTR(XDIC, JJ, 1) = SUBSTR(XDIC, JJ+1, 1);

END;

GO TO DN1;

END;

END;

DN1: TEXT=SUBSTR(XDIC, 1, DCNT-1);

GO TO ALG(0);

/* END OF ALGORITHM 1. */

REALGOOF:

TEXT=TXT;

END; /* DICTIONARY SEARCH. */

GO TO GIT;

RTRN:

PUT FILE(PRINT) EDIT("ABOUT TO CALL UNSET, I='", I)

(SKIP(DEBUG),A(DEBUG*24),F(DEBUG*5));

CALL UNSET;

RETURN;

END CRRCTN;
MASTER DYNAMIC CORRECTION MODULE

CMMT1 INIT {"C ERROR CORRECTION ATTEMPTED IN FOLLOWING CARD"
00008110
D
00008120
CMMT2 INIT {"C ORIGINAL CARD WAS"
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
)
ERROR CORRECTION PROGRAM WHICH THIS MODULE REPRESENTS IS

THE PROGRAMMED SOLUTION FOR THE PART OF THE OVERALL

METHOD OF SOLUTION:

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

DESCRIPTION OF THE PROGRAM:

MEMORY SIZE: DEPENDS ON OTHER CORRECTOR PROCEDURES.
OPERATING SYSTEM: 80/MT.
OBJECT-COMPILER: I8M 360/65.
SOURCE-COMPILER: I8M 360/65.
INSTALLATION: IOWA STATE UNIVERSITY COMPUTATION CENTER.
AUTHOR: C.E. HERRICK.
PROGRAM-ID: CIRC.

------------------------------------------
IDENTIFICATION:

* NCRD CHAR(LENGTH(CARD)) VAR:
CHST(66) CHAR(1) DEF CARO POS(7),
00080000 00080000 00080000 00080000
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 ERRORS;
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 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...
### THE CORRECTED SOURCE PROGRAM STATEMENTS.

### DESCRIPTION OF VARIABLES AND LABELS:

<table>
<thead>
<tr>
<th>VARIABLE OR LABEL</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLBLANKS,BLANKS</td>
<td>VARIABLES WHICH CONSISTS ENTIRELY OF BLANKS; IT IS USED FOR TESTING.</td>
</tr>
<tr>
<td>BAD29.</td>
<td>PROBABLE MISSPUNCHED CHARACTERS WHEN THE FORTRAN PROGRAM HAS BEEN PUNCHED WITH AN IBM 029 KEYPUNCH.</td>
</tr>
<tr>
<td>CARC,CRD.</td>
<td>IMAGE OF THE CARD IN ERROR.</td>
</tr>
<tr>
<td>CMMT1,CMMT2,MARKER</td>
<td>COMMENTS WHICH ARE TO BE PLACED IN THE NEW FORTRAN SOURCE PROGRAM.</td>
</tr>
<tr>
<td>DEBUG.</td>
<td>EXTERNAL PARAMETER.</td>
</tr>
<tr>
<td>ECD.</td>
<td>A COPY OF THE ERROR CODE.</td>
</tr>
<tr>
<td>ERRSET.</td>
<td>A LABEL ARRAY; EACH DIFFERENT ERRSET IS THE START OF A SUBORDINATE CORRECTION MODULE.</td>
</tr>
<tr>
<td>G29,GOOD29.</td>
<td>THE SET OF POSSIBLE CORRECTIONS FOR THE ERRORS IN BAD29.</td>
</tr>
<tr>
<td>I.</td>
<td>A COUNTER WHICH IS USED THROUGHOUT THE PROGRAM.</td>
</tr>
<tr>
<td>LETS.</td>
<td>THE FIRST LETTER OF ERROR CODES.</td>
</tr>
<tr>
<td>LINE,XLINE.</td>
<td>LINE IMAGE OF THE MESSAGE LINE.</td>
</tr>
<tr>
<td>NCARD.</td>
<td>THE NEW CARD IMAGE.</td>
</tr>
<tr>
<td>NEWPGM.</td>
<td>THE TEMPORARY NEW PROGRAM OUTPUT FILE.</td>
</tr>
<tr>
<td>PARMS.</td>
<td>EXTERNAL PARAMETER ARRAY.</td>
</tr>
<tr>
<td>PRINT.</td>
<td>DEBUG OUTPUT FILE.</td>
</tr>
<tr>
<td>RELINQUISH.</td>
<td>THE POINT WHERE THIS MODULE RETURNS CONTROL TO THE CALLING MODULE.</td>
</tr>
<tr>
<td>Z1,Z2,Z3.</td>
<td>FILLERS IN PARMS.</td>
</tr>
</tbody>
</table>
IDENI TICATION OF THE ERRSET MODULES:

<table>
<thead>
<tr>
<th>MODULE NUMBER</th>
<th>MODULE NAME</th>
<th>REFERENCES</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>UNIDENTIFIED ERROR MODULE.</td>
<td>00009560</td>
</tr>
<tr>
<td>01</td>
<td>ASSIGN STATEMENTS AND VARIABLES MODULE.</td>
<td>00009570</td>
</tr>
<tr>
<td>02</td>
<td>BLOCK DATA STATEMENTS MODULE.</td>
<td>00009580</td>
</tr>
<tr>
<td>03</td>
<td>CARD FORMAT AND CONTENTS MODULE.</td>
<td>00009590</td>
</tr>
<tr>
<td>04</td>
<td>DATA/DIMENSION STATEMENT MODULE.</td>
<td>00009600</td>
</tr>
<tr>
<td>05</td>
<td>EQUIVALENCE-COMMON/EXTERNAL MODULE.</td>
<td>00009610</td>
</tr>
<tr>
<td>06</td>
<td>FORMAT/FUNCTION MODULE.</td>
<td>00009620</td>
</tr>
<tr>
<td>07</td>
<td>GO-TO MODULE.</td>
<td>00009630</td>
</tr>
<tr>
<td>08</td>
<td>HOLLERITH MODULE.</td>
<td>00009640</td>
</tr>
<tr>
<td>09</td>
<td>IF-I/O MODULE.</td>
<td>00009650</td>
</tr>
<tr>
<td>10</td>
<td>JCL MODULE.</td>
<td>00009660</td>
</tr>
<tr>
<td>11</td>
<td>DEPENDENT EXECUTION FLAG MODULE.</td>
<td>00009670</td>
</tr>
<tr>
<td>12</td>
<td>LOGICAL/LIBRARY MODULE.</td>
<td>00009680</td>
</tr>
<tr>
<td>13</td>
<td>MODE MODULE.</td>
<td>00009690</td>
</tr>
<tr>
<td>14</td>
<td>PARENTHESIS MODULE.</td>
<td>00009700</td>
</tr>
<tr>
<td>15</td>
<td>RETURN STATEMENT MODULE.</td>
<td>00009710</td>
</tr>
<tr>
<td>16</td>
<td>MULTIPLE ERROR MODULE.</td>
<td>00009720</td>
</tr>
<tr>
<td>17</td>
<td>UNDEFINED OPERATIONS/VARIABLES MODULE.</td>
<td>00009730</td>
</tr>
<tr>
<td>18</td>
<td>VARIABLE NAME MODULE.</td>
<td>00009740</td>
</tr>
<tr>
<td>19</td>
<td>EXTERNAL STATEMENT MODULE.</td>
<td>00009750</td>
</tr>
</tbody>
</table>

REFERENCES:

---------

BLATT, JOHN M. INTRODUCTION TO FORTRAN IV PROGRAMMING: USING THE WATFOR COMPILER. PACIFIC PALISADES, CALIFORNIA. GOODYEAR PUBLISHING COMPANY. C1968.


GRIES, D. USE OF TRANSLATION MATRICES IN COMPILING. ASSOCIATION FOR COMPUTING MACHINERY COMMUNICATIONS 11: 26-34. 1968.

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.

GO TO CONT;

TP=1:

ELSE IF TCT<0 THEN CHST(KT)=SUBSTR(OPER2,TCT,2);

TP=2:

GO TO CONT;

CHST(KT)=TCT;

TP=6:

GO TO CONT;

CROD(T)="."

TP=7:

ACTION(7):=

ACTION(6):=

ACTION(5):=

ACTION(4):=

ACTION(3):=

ACTION(2):=

ACTION(1):=

ACTION(0):=

/*

*/

GO TO ACTION(TP)?

DCL ACTION(6:20) LABEL,TYP,TCT) FIXED BIN(12')O;

/*

MATRICES ARE COMBINED IN ARRAY IMA.

THE TRANSLATION MATRIX HAS ENTERIES AND VICE VERSA. THE TWO

THE ACTION MATRIX. SINCE THE ACTION MATRIX HAS BLANKS WHERE

THE ACTION PROEDURE CONTAINS THE ACTIONS WHICH ARE SPECIFIED BY

PROCEDURE: PROC(TYP)

REVIEW SOFTWARE AG 4, NO. 2: 8-12, 1970.

MORRIS, FRANK, RANDOM INPUT SEARCH TECHNIQUES: A TIMELY
/* END; */

END; /* GET CORRECTIVE ACTION ROUTINE. */

RETURN: CONT;

ACTION(0): CARD=SUBSTR(CARD,1,KT+6)|||SUBSTR(CARD,KT+7);

ACTION(1): CARD=SUBSTR(CARD,1,KT+5)|||SUBSTR(CARD,KT+7);

ACTION(2): CARD=SUBSTR(CARD,1,KT+6)|||SUBSTR(CARD,KT+7);

IF KT=0 THEN CHST(KT)=SUBSTR(OPER2,CHRST(KT));

ACTION(13): CARD=SUBSTR(CARD,1,KT+6)|||SUBSTR(CARD,KT+7);

ACTION(14): CARD=SUBSTR(CARD,1,KT+5)|||SUBSTR(CARD,KT+7);

ACTION(15): CARD=SUBSTR(CARD,1,KT+6)|||SUBSTR(CARD,KT+7);

ACTION(16): CARD=SUBSTR(CARD,1,KT+5)|||SUBSTR(CARD,KT+7);

ACTION(17): CARD=SUBSTR(CARD,1,KT+6)|||SUBSTR(CARD,KT+7);

ACTION(18): CARD=SUBSTR(CARD,1,KT+5)|||SUBSTR(CARD,KT+7);

ACTION(19): CARD=SUBSTR(CARD,1,KT+6)|||SUBSTR(CARD,KT+7);

ACTION(20): CARD=SUBSTR(CARD,1,KT+5)|||SUBSTR(CARD,KT+7);

ACTION(21): CARD=SUBSTR(CARD,1,KT+6)|||SUBSTR(CARD,KT+7);

ACTION(22): CARD=SUBSTR(CARD,1,KT+5)|||SUBSTR(CARD,KT+7);

ACTION(23): CARD=SUBSTR(CARD,1,KT+6)|||SUBSTR(CARD,KT+7);

ACTION(24): /* END; */
IF ECD='CN-6' THEN DO;
  TSX3=SUBSTR(XLINE,35,1)||SUBSTR(XLINE,44,1);
  I=INDEX(CARD,TSX3);
  (NOSUBRG): IF (I=0)&(SUBSTR(XLINE,44,1)='Z')&(CRD(I)='.'
    THEN CRD(I)='.';
END; /* OF ILLEGAL CONSTANT MODULE. */
GO TO RELINQUISH;

ERRSET(04):
ERRSET(05):
ERRSET(06):
ERRSET(07):
ERRSET(08):
ERRSET(09):
ERRSET(10):
  GO TO RELINQUISH;
ERRSET(11):
  /* DEPENDENT EXECUTION FLAG MODULE. */
  IF CRD(I)=='/' THEN CARD='.';
  IF ECD='KO-0' THEN GO TO RELINQUISH;
  GO TO RELINQUISH;
ERRSET(12):
ERRSET(13):
ERRSET(14):
ERRSET(15):
  GO TO RELINQUISH;
ERRSET(16):
  /* MULTIPLE ERROR MODULE. */
  /* SELECTED SYNTAX ERROR MODULE 1. */
  /* DOES NOT OPERATE FOR IF-STATEMENTS. */
  PUT FILE(PRINT) EDIT('ENTER MULTIPLE ERROR MODULE')
    (SKIP(DEBUG),A(DEBUG*36));
  IF INDEX(CARD,'IF()')=0 THEN DO;
    PUT FILE(PRINT) EDIT('ENTER SX MODULE 1,TKT=','TKT')
      (SKIP(DEBUG), A(DEBUG*36), F(DEBUG*6));
    IF DEBUG THEN ON SIZE SNAP BEGIN; (NOSIZE):
      PUT FILE(PRINT) EDIT(TMAT,TEMPCH) (5 F(5,0));
    END;
IF ECD='SX-0' THEN DO I=0 TO 9;
ISXO=I;
KT=INDEX(CARD,ISXO||'\''');
TKT=TKT+1;
IF KT=0 THEN CARD=SUBSTR(CARD,1,KT)||'\'''||SUBSTR(CARD,KT+1);
KT=INDEX(CARD,'\'''||ISXO);
TKT=TKT+1;
IF KT=0 THEN CARD=SUBSTR(CARD,1,KT)||'\'''||SUBSTR(CARD,KT+1);
END;
IF TKT>0 THEN GO TO RELINQUISH;
/* SELECTED SYNTAX ERROR MODULE 2 */
IF ECD='SX-0' THEN DO KT=1 TO 66 BY 1;
IF (CHST(KT)>='A')&(CHST(KT)<='Z') THEN TC1=0;
IF (CHST(KT)>='0') THEN TC1=1;
IF INDEX(OPS,CHST(KT))=0 THEN TC1=4;
IF (((TEMPCH=4)&&(TEMPCH=5))&(CHST(KT)='E')) THEN TC1=3;
PUT FILE(EDIT=TEMPCH) (SKIP(debug),F(debug*3));
TEMPCH=MTGP(TEMPCH,TC1);
PUT FILE(EDIT=TEMPCH) (SKIP(debug),F(debug*3));
IF TEMPCH>5 THEN CALL ACTION(TEMPCH);
IF TEMPCH>5 THEN GO TO RELINQUISH;
END;
IF ECD='SX-0' THEN
GO TO RELINQUISH;
END; /* SX-0 SUB-MODULE. */
/* SUPERFLUOUS OR BAD CHARACTER MODULE. */
IF ECD='SX-3' THEN DO;
TSX3=SUBSTR(XLINE,35,1)||SUBSTR(XLINE,53,1);
I=INDEX(CARD,TSX3);
IF (CRD(I)='\''>>&I=0) THEN CRD(I)='O';
END; /* OF SUPERFLUOUS CHARACTER MODULE. */
/* STATEMENT AND STATEMENT NUMBER MODULE. */
/* PUT FILE(EDIT='ENTER STATEMENT AND NUMBER MODULE')
   (SKIP(debug),A(debug*36));
   IF ECD='ST-1' THEN CRD(6)='4';
   IF (ECD='ST-5') && (ECD='ST-4') THEN CRD(1)='C';
ON SIZE SYSTEM;
/* BAD STATEMENT NUMBER MODULE. */
IF ECD='ST-A' THEN DO;
CARD=SUBSTR(XLINE,25,5)||'CONTINUE';
CMMT3=CARD;
WRITE FILE(NEWPGM) FROM(CMMT3);
END; /* OF BAD STATEMENT NUMBER MODULE. */
GO TO RELINQUISH;
ERRSET(17):
ERRSET(18):
ERRSET(19):
ERRSET(00):
RELINQUISH:
/* TERMINAL ANALYSIS MODULE. */
NCARD=CARD;
PUT FILE(PRINT) EDIT('LEAVE CRRCTR WITH NCARD',NCARD)
(SKIP(DEBUG),A(26*DEBUG),SKIP(DEBUG),A(80*DEBUG));
/* END OF DYNAMIC INNER CORRECTION MODULES. */
RETURN(NCARD);
END CRRCTR;
STREAM INPUT MODULE FOR ERROR CORRECTION. /*

IDENTIFICATION:

PROGRAM-IC: GETR.
AUTHOR: G. E. HEDRICK.
INSTALLATION: IOWA STATE UNIVERSITY COMPUTATION CENTER.

SUBREG,STRG,SIZE):

GETR:

PROC(TXT,FLG) CPTIGNS(REENTRANT) RECURSIVE;

DCL

TXT CHAR(11) VAR,
FLG CHAR(1),
1 PARMS STATIC EXT UNALIGNED,
2 (DEBUG,21,22,23) BIT(1),
1 I O UNALIGNED STATIC EXT,
2 (K INIT(0),L INIT(1), M INIT(7), N INIT(0)) FIXEC 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:

-------------------
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:
--------------------------------------

<table>
<thead>
<tr>
<th>VARIABLE OR LABEL</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLBLANKS</td>
<td>A BLANK CHARACTER STRING WHICH IS USED IN TESTING.</td>
</tr>
<tr>
<td>BUF, BUFR, CARD</td>
<td>INPUT BUFFERS.</td>
</tr>
<tr>
<td>BUF6, BUF72, BUG73</td>
<td>WORKING PORTIONS OF THE BUFFERS.</td>
</tr>
<tr>
<td>DEBUG</td>
<td>EXTERNAL DEBUG FLAG.</td>
</tr>
<tr>
<td>DELIM, DELIMS</td>
<td>FORTRAN DELIMITERS.</td>
</tr>
<tr>
<td>FLG</td>
<td>RETURN PARAMETER; CONTAINS L, R, N OR E.</td>
</tr>
<tr>
<td>FLGR</td>
<td>SPACE HOLDER.</td>
</tr>
<tr>
<td>I, II, J, K, L, M, MM, N</td>
<td>RECORD COUNTERS AND WORK VARIABLES FOR I/O.</td>
</tr>
<tr>
<td>I_O</td>
<td>COMMUNICATION STRUCTURE FOR I/O.</td>
</tr>
<tr>
<td>INPT</td>
<td>LABEL ARRAY. TRANSFERS ARE DEPENDENT</td>
</tr>
</tbody>
</table>

00012710
00012720
00012730
00012740
00012750
00012760
00012770
00012780
00012790
00012800
00012810
00012820
00012830
00012840
00012850
00012860
00012870
00012880
00012890
00012900
00012910
00012920
00012930
00012940
00012950
00012960
00012970
00012980
00012990
00013000
00013010
00013020
00013030
00013040
00013050
00013060
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);

K=1;
/
*/
*/

INPT(1):

IF M=73 THEN GO TO DNN;
IF SUBSTR(BUF72,M)=" " THEN DO;
    DNN:
        L=73; CALL PUTR('A',FLGR); M=7; GO TO INPT(0);
    END;
I=100;
DO J=1 TO LENGTH(DELIMS);
II=INDEX(SUBSTR(BUF,M),DELIM(J));
I=I*(II>=I)+II*(II<1)+I*(II=0);
END;
IF I=100 THEN GO TO INPT(0);
I=11*(I>11)+I*(I<=11);
TXT=SUBSTR(BUF,M,I);
M=M+I;
MM=M-1;
IF CARD(MM)=';' THEN K=0;
IF CARD(MM)='=' THEN FLG='L';

RTRN:

IF LENGTH(TXT)=1 THEN DO; CALL PUTR(TXT,FLGR); GO TO INPT(1); END;

PUT FILE(PRINT) EDIT('LEAVE GETRTXT,FLG,'LEAVE GETR' )
    (SKIP(DEBUG),4 A(12*DEBUG));
RETURN;
END GETR;
IDENTIFICATION:

-----------

PROGRAM-ID: PUTR.

AUTHOR: G. E. HEDRICK.

INSTALLATION: IOWA STATE UNIVERSITY COMPUTATION CENTER.


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.
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:

<table>
<thead>
<tr>
<th>VARIABLE OR LABEL</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUF.</td>
<td>INPUT BUFFER; CHECKED ONLY.</td>
</tr>
<tr>
<td>CLEAR.</td>
<td>POINT AT WHICH THE OUTPUT BUFFERS ARE CLEARED.</td>
</tr>
<tr>
<td>DEBUG.</td>
<td>EXTERNAL DEBUG FLAG.</td>
</tr>
<tr>
<td>DELIMS.</td>
<td>THE FORTRAN DELIMITERS.</td>
</tr>
<tr>
<td>FLG.</td>
<td>UNUSED PARAMETER.</td>
</tr>
<tr>
<td>I_0.</td>
<td>EXTERNAL STRUCTURE WHICH IS USED BY THE I/O ROUTINES FOR COMMUNICATION.</td>
</tr>
<tr>
<td>K,L,M,N.</td>
<td>RECORD COUNTERS FOR I/O.</td>
</tr>
<tr>
<td>OUTREC,RECOUT.</td>
<td>OUTPUT BUFFERS.</td>
</tr>
<tr>
<td>P ARMS.</td>
<td>EXTERNAL PARAMETER LIST.</td>
</tr>
<tr>
<td>PGMNEW.</td>
<td>OUTPUT FILE.</td>
</tr>
<tr>
<td>PRINT.</td>
<td>DEBUG FILE.</td>
</tr>
<tr>
<td>TXT.</td>
<td>STRING TO BE OUTPUT.</td>
</tr>
<tr>
<td>WK,WK73.</td>
<td>WORKING CHARACTER STRINGS.</td>
</tr>
<tr>
<td>Z1,Z2,Z3.</td>
<td>FILLERS.</td>
</tr>
</tbody>
</table>

*/

CLEAR: PROC RECURSIVE;
   PUT FILE(PRINT) EDIT('ENTER CLEAR')
   (SKIP(DEBUG),A(12*DEBUG));
   RECOUT=OUTREC;
   WK73='';
   IF RECOUT<=(80) ' ' THEN
WRITE FILE(PGMNEW) FRCM(RECOUT);
L=1;
CUTREC,RECOUT=' ';
PUT FILE(PRINT) EDIT('LEAVE CLEAR')
(SKIP(DEBUG),A(12*DEBUG));
RETURN;
END CLEAR;

/*
 *
*/
IF DEBUG THEN ON STRG PUT FILE(PRINT) EDIT
(OUTREC,TXT,LENGTH(TXT),L) (SKIP,2 A,X(3),2 F(6,0));
PUT FILE(PRINT) EDIT('ENTER PUTR.',TXT,'ENTER PUTR')
(SKIP( DEBUG),3 A(12*DEBUG));
SUBSTR(OUTREC,L,LENGTH(TXT))=TXT;
L=L+LENGTH(TXT);
WK=SUBSTR(TXT,LENGTH(TXT),1);
IF M<=7 THEN RETURN;
IF L>=72 THEN CALL CLEAR;
ELSE
IF INDEX(DELIMS,WK)=0 THEN CALL CLEAR;
ELSE
IF WK=' ' THEN DO;
IF SUBSTR(BUF,M)='' THEN CALL CLEAR;
END:
PUT FILE(PRINT) EDIT('LEAVE PUTR')(SKIP(DEBUG),A(12*DEBUG));
RETURN;
END PUTR;
TRANSLATION MATRIX FILL ROUTINE. /*

TFILL: PROC;
DCL TMAT(0:5,0:4) FIXED BIN(15,0) STATIC EXT;

THIS ROUTINE SERVES TO FILL THE TRANSLATION MATRIX. THIS
MATRIX IS USED IN THE ERROR ANALYSIS AND CORRECTION ROUTINES
WHICH FOLLOW.

TMAT(0,0) ,TMAT(1,0),TMAT(1,1)=1;
TMAT(2,0) ,TMAT(3,0),TMAT(3,1)=3;
TMAT(1,2),TMAT(3,4),TMAT(4,4)=2;
TMAT(2,1),TMAT(4,1),TMAT(5,1)=4;
TMAT(5,0)=3;
TMAT(0,1)=6; TMAT(0,2)=7; TMAT(0,3)=20; TMAT(0,4)=8;
TMAT(1,4)=9; TMAT(2,2)=10; TMAT(2,4)=11; TMAT(3,2)=12;
TMAT(4,0)=13; TMAT(4,2)=14; TMAT(5,2)=15;
TMAT(5,3)=16; TMAT(5,4)=17;
TMAT(1,3),TMAT(2,3),TMAT(3,3)=20;
RETURN;
END; /* OF TRANSLATION MATRIX FILL ROUTINE. */
/*
SYMBOL TABLE INITIALIZATION ROUTINE. */

DCL DICTIONARY(255) CHAR(11) VAR;
DICTI0NARY(1)='ASSIGN'; DICTIONARY(2)='GOTO';
DICTI0NARY(3)='DIMENSION'; DICTIONARY(4)='END';
DICTI0NARY(5)='SIN'; DICTIONARY(6)='COS'; DICTIONARY(7)='EXP';
DICTI0NARY(8)='MOD'; DICTIONARY(9)='ABS'; DICTIONARY(10)='TAN';
DICTI0NARY(11)='ERF'; DICTIONARY(12)='STOP';
DICTI0NARY(13)='READ'; DICTIONARY(14)='REAL';
DICTI0NARY(15)='DATA'; DICTIONARY(16)='GE.;'
DICTI0NARY(17)='GT'; DICTIONARY(18)='EQ';
DICTI0NARY(19)='NE.'; DICTIONARY(20)='LE.';
DICTI0NARY(21)='OR.'; DICTIONARY(22)='LT.';
DICTI0NARY(23)='SQRT'; DICTIONARY(24)='TABS';
DICTI0NARY(25)='AMOD'; DICTIONARY(26)='ALOG';
DICTI0NARY(27)='ATAN'; DICTIONARY(28)='DMOD';
DICTI0NARY(29)='SIGN'; DICTIONARY(30)='COSH';
DICTI0NARY(31)='SINH'; DICTIONARY(32)='TANH';
DICTI0NARY(33)='DABS'; DICTIONARY(34)='DEXP';
DICTI0NARY(35)='DLOG'; DICTIONARY(36)='DSIN';
DICTI0NARY(37)='DCOS'; DICTIONARY(38)='DTAN';
DICTI0NARY(39)='MAX0'; DICTIONARY(40)='MAX1';
DICTI0NARY(41)='MIN0'; DICTIONARY(42)='MIN1';
DICTI0NARY(43)='CABS'; DICTIONARY(44)='CSIN';
DICTI0NARY(45)='CCOS'; DICTIONARY(46)='ERFC';
DICTI0NARY(47)='DERF'; DICTIONARY(48)='IDIM';
DICTI0NARY(49)='HFIX'; DICTIONARY(50)='DBLE';
DICTI0NARY(51)='AINT'; DICTIONARY(52)='SNGL';
DICTI0NARY(53)='EXIT'; DICTIONARY(54)='FILE';
DICTI0NARY(55)='WRITE'; DICTIONARY(56)='ENTRY';
DICTI0NARY(57)='AND.'; DICTIONARY(58)='ISIGN';
DICTI0NARY(59)='DSIGN'; DICTIONARY(60)='ARCCS';
DICTI0NARY(61)='ARSC'; DICTIONARY(62)='COTAN';
DICTI0NARY(63)='DSQRT'; DICTIONARY(64)='PRINT';
DICTI0NARY(65)='PUNCH'; DICTIONARY(66)='AMAX0';

00014980 00014990 00015000 00015010 00015020 00015030 00015040 00015050 00015060 00015070 00015080 00015090 00015100 00015110 00015120 00015130 00015140 00015150 00015160 00015170 00015180 00015190 00015200 00015210 00015220 00015230 00015240 00015250 00015260 00015270 00015280 00015290 00015300 00015310 00015320 00015330 00015340
DICTIONARY(67) = 'AMIN1';  DICTIONARY(68) = 'AMAX1';  00015350
 DICTIONARY(69) = 'AMIN0';  DICTIONARY(70) = 'DMAXO';  00015360
 DICTIONARY(71) = 'CMAX1';  DICTIONARY(72) = 'DMINO';  00015370
 DICTIONARY(73) = 'DMIN1';  DICTIONARY(74) = 'DTHAH';  00015380
 DICTIONARY(75) = 'CDLOG';  DICTIONARY(76) = 'CDEXP';  00015390
 DICTIONARY(77) = 'ATAN2';  DICTIONARY(78) = 'CDSIN';  00015400
 DICTIONARY(79) = 'CCOS';  DICTIONARY(80) = 'DCOUSH';  00015410
 DICTIONARY(81) = 'CSINH';  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) = 'DSIN';  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';
 DICTIONARY(102) = 'EXTERNAL';  00015520
 DICTIONARY(103) = 'FUNCTION';  00015530
 DICTIONARY(104) = 'NAMELIST';  00015540
 DICTIONARY(105) = 'SUBROUTINE';  00015550
 DICTIONARY(106) = 'EQUIVALENCE';  00015560
 DICTIONARY(107) = 'GRAPH';  DICTIONARY(108) = 'PRINT';  00015570
 DICTIONARY(109) = 'READ';  00015580
 DICTIONARY(110) = 'END';  00015590
 DICTIONARY(111) = 'CALL';  00015600
 RETURN;
 END DFILL;  00015610

/* SELECTED I/O MODULES FOR THE ERROR CORRECTOR. */

(STRG,SUBRG,SIZE):
RESET: PROC;
    DCL (PRINT,PGMNEW,NEWPGM) FILE EXT, CARD CHAR(80);
    DCL 1 PARMS STATIC UNALIGNED EXT,
        2 DEBUG BIT(1), 2 (ZU,ZI,ZO) BIT(1);

IDENTIFICATION:
------------------
PROGRAM-ID: RESET.
AUTHOR: G. E. HEDRICK.
SOURCE-COMPUTER: IBM 360/65.
OBJECT-COMPUTER: IBM 360/65.
OPERATING SYSTEM: OS/MVT.
MEMORY SIZE: DEPENDS ON OTHER CORRECTOR PROCEDURES.

PURPOSE:
--------
THIS PROCEDURE HAS ENTRY POINTS SET, UNSET, AND
RESET. THE PROCESSING WHICH STARTS AT EACH ENTRY POINT IS
USED WHEN A FILE USE MUST BE INITIALIZED OR TERMINATED.

*/

PUT FILE(PRINT) EDIT('ENTER RESET')
    (SKIP(DEBUG),A(CEBUG*11));
CLOSE FILE(NEWPGM);
OPEN FILE(NEWPGM) SEQUENTIAL RECORD INPUT;
ON ENDOFILE(NEWPGM) GO TO RTRNR;
DO WHILE('l'B);
READ FILE(NEWPGM) INTO(CARD);
IF CARD='' THEN
WRITE FILE(PGMNEW) FROM(CARD);
END;
RTRNR:
CLOSE FILE(NEWPGM);
OPEN FILE(NEWPGM) SEQUENTIAL RECORD OUTPUT;
PUT FILE(PRINT) EDIT('NEWPGM—OUTPUT')
(SKIP(DEBUG),A(DEBUG*20));
RETURN;
ENTRY;
PUT FILE(PRINT) EDIT('ENTER SET')
(SKIP(DEBUG),A(DEBUG*11));
CLOSE FILE(NEWPGM);
OPEN FILE(NEWPGM) SEQUENTIAL RECORD INPUT;
IF DEBUG THEN DO;
ON ENDOFILE(NEWPGM) GO TO NXTSET;
PUT FILE(PRINT) SKIP EDIT('NEW PROGRAM:')(A);
DO WHILE('l'B);
READ FILE(NEWPGM) INTO(CARD);
PUT FILE(PRINT) SKIP EDIT(CARD) (A);
END;
NXTSET: CLOSE FILE(NEWPGM);
OPEN FILE(NEWPGM) RECORD SEQUENTIAL INPUT;
PUT FILE(PRINT) SKIP EDIT('END OF PROGRAM.')(A);
END;
PUT FILE(PRINT) EDIT('NEWPGM—INPUT')
(SKIP(DEBUG),A(DEBUG*20));
RETURN;
ENTRY;
PUT FILE(PRINT) EDIT('ENTER UNSET')
(SKIP(DEBUG),A(DEBUG*11));
GO TO RTRNR;
END RESET;
/* CLEANUP ROUTINE FOR ERROR CORRECTION. */ 00016890

(CSUBRG,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 'S999999999',
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
ERROR CORRECTION ATTEMPTED IN FOLL

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) POINTER,
(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 REG(1),
PARM CHAR(40);

IDENTIFICATION:
-----------------
PROGRAM-ID: CLEANUP.
AUTHOR: G. E. HEDRICK.
INSTALLATION: IOWA STATE UNIVERSITY COMPUTATION CENTER.
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;
9. RE-OPEN SCRATCH FILES;
10. RE-INITIALIZE SELECTED VARIABLES;
11. READ TWO CARD IMAGES;
12. READ A THIRD CARD IMAGE;
13. IF AN "ERROR" OCCURS GO TO STEP 19;
14. IF THE FIRST CARD IMAGE IS TRASH GO TO STEP 16;
15. WRITE THE FIRST CARD IMAGE ON THE NEW PROGRAM DATA SET;
16. REPLACE THE FIRST CARD IMAGE BY THE SECOND;
17. REPLACE THE SECOND CARD IMAGE BY THE THIRD;
18. GO TO STEP 12;
19. CLOSE AND RE-OPEN NECESSARY FILES;
20. COPY SEQUENTIAL FILE TO DIRECT FILE;
21. CHECK FORMAT STATEMENTS FOR IMBEDDED GARBAGE;
22. COPY DIRECT FILE TO SEQUENTIAL FILE;
23. GO TO NEXT JOB STEP.

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:
---------------------------

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBUG</td>
<td>REQUESTS DEBUG OUTPUT INCLUDING A</td>
</tr>
</tbody>
</table>
DESCRIPTION OF OUTPUT:
---------------------

THERE ARE THREE KINDS OF OUTPUT FROM THIS PROCEDURE: DEBUG OUTPUT, LIST OUTPUT, AND PROGRAM OUTPUT. DEBUG AND LIST OUTPUT ARE PRINTED AND MUST BE REQUESTED WITH AN INPUT PARAMETER. DEBUG OUTPUT CONSISTS OF SELECTED PRINTOUT WHICH INCLUDES A TRACE OF THE PROGRAM. LIST OUTPUT CONSISTS OF SELECTED LISTINGS OF THE FORTRAN PROGRAM. PROGRAM OUTPUT IS ALWAYS GIVEN. PROGRAM OUTPUT CONSISTS OF THE CLEANSED FORTRAN PROGRAM(5); IT IS PLACED IN THE NEW PROGRAM DATA SET.

DESCRIPTION OF FILES:
-----------------------

FILE USE
-------

PGMNEW. FILE WHICH CONTAINS FORTRAN PROGRAMS BEFORE AND AFTER CLEANUP.
PRINT. FILE WHICH CONTAINS LIST AND DEBUG OUTPUT.
SCRATCH. DIRECT UPDATE SCRATCH FILE USED IN TRASH ELIMINATION.

DESCRIPTION OF VARIABLES AND LABELS:
----------------------------------------

TRACE OF THE PROGRAM EXECUTION. 00018000
REQUESTS LISTINGS OF FORTRAN PROGRAMS 00018010
00018020
00018030
00018040
00018050
00018060
00018070

00018080
00018090
00018100
00018110
00018120
00018130
00018140
00018150
00018160
00018170
00018180
00018190
00018200
00018210
00018220
00018230
00018240
00018250
00018260
00018270
00018280
00018290
00018300
00018310
00018320
00018330
00018340
00018350
00018360
<table>
<thead>
<tr>
<th>VARIABLE OR LABEL</th>
<th>MEANING OR USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>B5tB66</td>
<td>VARIABLE CONSISTING ENTIRELY OF BLANKS; USED FOR TESTING.</td>
</tr>
<tr>
<td>BF1,BF2,BUF1,BUF2,CARD,CRD.</td>
<td>CARD IMAGES.</td>
</tr>
<tr>
<td>BUF1_72</td>
<td>THE FIRST 72 COLUMNS OF BUF1.</td>
</tr>
<tr>
<td>BUF2_50</td>
<td>THE FIRST 50 COLUMNS OF BUF2.</td>
</tr>
<tr>
<td>BUF2_72</td>
<td>THE FIRST 72 COLUMNS OF BUF2.</td>
</tr>
<tr>
<td>CARD72</td>
<td>THE FIRST 72 COLUMNS OF CARD.</td>
</tr>
<tr>
<td>COLS</td>
<td>THE CHARACTER ARRAY FCRM OF CARD.</td>
</tr>
<tr>
<td>CODE,STARS</td>
<td>CODE USED IN TESTING FOR TRASH.</td>
</tr>
<tr>
<td>COL1_5</td>
<td>THE FIRST 5 COLUMNS OF CARD.</td>
</tr>
<tr>
<td>COL7_72</td>
<td>COLUMNS 7 THROUGH 72 OF CARD.</td>
</tr>
<tr>
<td>DATAx</td>
<td>STRUCTURE USED FOR MAINTAINING CERTAIN VARIABLES.</td>
</tr>
<tr>
<td>DEBUG,LIST</td>
<td>INPUT PARAMETERS.</td>
</tr>
<tr>
<td>FMTS</td>
<td>PLACE WHERE FORMAT STATEMENTS ARE EXAMINED.</td>
</tr>
<tr>
<td>I,N</td>
<td>COUNTERS ON LOOPS.</td>
</tr>
<tr>
<td>LP,RP</td>
<td>PARENTHESIS.</td>
</tr>
<tr>
<td>OUT</td>
<td>THE POINT OF TERMINATION OF THIS PROCEDURE.</td>
</tr>
<tr>
<td>P,Q,R</td>
<td>POINTERS USED TO SIMULATE OVERLAY DEFINING.</td>
</tr>
<tr>
<td>PARM,PARMS</td>
<td>SETS OF INPUT PARAMETERS.</td>
</tr>
<tr>
<td>PGMNEW,PRINT,SCRATCH,QUOTE</td>
<td>A SINGLE QUOTE MARK.</td>
</tr>
<tr>
<td>SKPCD</td>
<td>THE POINT TO WHICH CONTROL IS TRANSFERRED WHEN A CARD MUST BE DELETED.</td>
</tr>
</tbody>
</table>

REFERENCES:

---

00018370
00018380
00018390
00018400
00018410
00018420
00018430
00018440
00018450
00018460
00018470
00018480
00018490
00018500
00018510
00018520
00018530
00018540
00018550
00018560
00018570
00018580
00018590
00018600
00018610
00018620
00018630
00018640
00018650
00018660
00018670
00018680
00018690
00018700
00018710
00018720
00018730
THE REFERENCES ARE LISTED WITH THE OTHER MODULES AND IN THE BIBLIOGRAPHY OF THE DISSERTATION WHICH SERVES AS A DOCUMENTATION OF THIS PROGRAM.

```
FILE$S: PROC;
PUT FILE(PRINT) EDIT("ENTER FMT$S.") (SKIP(DEBUG), A(DEBUG*100));
PUT FILE(PRINT) EDIT("LEAVE FMT$S.") (SKIP(DEBUG), A(DEBUG*100));
RETURN;
END FILE$S;
```

```
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(STARTS, 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(I);
    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;
READ FILE(SCATCH) INTO(CARD) KEY(I);
I=1;
READ FILE(SCATCH) INTO(BUF1) KEY(I);
ON KEY(SCATCH) BEGIN;
    PUT FILE(PRINT) EDIT('END PHASE 2. START PHASE 3.'); 00019120
    (SKIP(DEBUG),A(100*DEBUG)); 00019130
    CLOSE FILE(PGMNEW);
    OPEN FILE(PGMNEW) RECORD SEQUENTIAL INPUT;
    PUT FILE(PRINT) EDIT('PHASE 3B.'); 00019140
    (SKIP(DEBUG),A(100*DEBUG)); 00019150
    CLOSE FILE(PGMNEW);
    ON ENDFILE(PGMNEW) BEGIN;
        PUT FILE(PRINT) EDIT('PHASE 3C.'); 00019160
        (SKIP(DEBUG),A(100*DEBUG)); 00019170
        CLOSE FILE(PGMNEW);
        ON KEY(SCATCH) BEGIN;
            PUT FILE(PRINT) EDIT('PHASE 3D.'); 00019180
                (SKIP(DEBUG),A(100*DEBUG)); 00019190
            ON KEY(SCATCH) GO TO OUT;
            PUT FILE(PRINT) EDIT('LISTING FOLLOWS:'); 00019199
                (SKIP(LIST),A(LIST*100)); 00019200
                DO I=0 BY 1 WHILE(COLI_5='/*'); 00019209
                READ FILE(SCATCH) INTO(CARD) KEY(I);
                WRITE FILE(PGMNEW) FROM(CARD);
                IF COLI_5='/*STOP' THEN DO; CARD='/*'; 00019218
                    WRITE FILE(PGMNEW) FROM(CARD);
                END;
                SIGNAL KEY(SCATCH);
            END;
            PUT FILE(PRINT) EDIT(CARD) 00019220
                (SKIP(LIST),A(LIST*100)); 00019230
            END;
            SIGNAL KEY(SCATCH);
        END;
        DO I=0 BY 1;
        READ FILE(SCATCH) INTO(CARD) KEY(I);
        IF INDEX(CARD,'FORMAT')=0 THEN CALL FMTS;
        END;
        SIGNAL KEY(SCATCH);
    END;
    DO I=0 BY 1;
    READ FILE(SCATCH) INTO(CARD) KEY(I);
    IF INDEX(CARD,'FORMAT')=0 THEN CALL FMTS;
    END;
    SIGNAL KEY(SCATCH);

DO I=0 BY 1;
READ FILE(PGMNEW) INTO(CARD);
REWRITE FILE(SRATCH) FROM(CARD) KEY(I);
END;
END;

DO I=2 BY 1 WHILE(COL1_5=''/* '');
READ FILE(SRATCH) INTO(BUF2) KEY(I);
PUT FILE(PRINT) EDIT('ABOUT TO TEST',CARD,BUF1,BF2)
(4 (SKIP(DEBUG),A(DECUB*100)));  
PUT FILE(PRINT) EDIT('CCOLS,*",COL1_5)
(SKIP(DEBUG),73 A(DEBUG),A(INTEGER+5));
IF(COL1_5=''/*DATA''&COL1_5=='/*STOP*)
(COL1_5=''/*JOB*) THEN DO;
IF CCOL(1)=''C' THEN DO N=1 TO 5 BY 1;
   IF INDEX(LEGAL,CCOL(N))=0 THEN GO TO SKPCD;
END;
ENC; ELSE GO TO LBL;
IF COL7_72=B66 THEN GO TO SKPCD;

LBL: IF BUF1_72=STARS THEN DO;
   PUT FILE(PRINT) EDIT('STARS SPOTTED.')
   (SKIP(DEBUG),A(DECUB*100));
   PUT FILE(PRINT) EDIT('TESTING*,BUF2_50,CODE)
   (SKIP(LIST),A(LIST*12),
    2 (SKIP(LIST),A(LIST*100)));
   IF BUF2_50=CODE THEN DO;
   PUT FILE(PRINT) EDIT('GOT ONE')
   (SKIP(DEBUG),A(DEBUG*100));
   GO TO SKPCD;
   END;
   END; WRITE FILE(PGMNEW) FROM(CARD);
SKPCD:
   CARD=BUF1;
   BUFI=BUF2;
   IF COL1_5=''/*STOP'' THEN DO;
   IF SUBSTR(BF1,1,5)=='/*' THEN SIGNAL KEY(SCRATCH);
OUT:

END;
PUT FILE(PRINT) EDIT('END OF TEST LOOP')
(SKIP(DEBUG),A(DEBUG*100));
END;
SIGNAL KEY(SCRATCH);
END;
PUT FILE(PRINT) EDIT('GO...')
(SKIP(DEBUG),A(DEBUG*100));
PUT FILE(PRINT) EDIT('INPUT LISTING FOLLOWS:')
(SKIP(LIST),A(LIST*100));
BF1='';
DO I=0 BY 1 WHILE(COL1_5='/*');
READ FILE(PGMNEW) INTO(CARD);
PUT FILE(PRINT) EDIT(CARD)
(SKIP(LIST),A(LIST*100));
IF SUBSTR(BF1,1,5)='/STOP' THEN DO;
IF COL1_5='/*' THEN SIGNAL ENDFILE(PGMNEW);
END;
WRITE FILE(SCRATCH) FROM(CARD) KEYFROM(I);
BF1=CARD;
END;
SIGNAL ENDFILE(PGMNEW);
OUT:

PUT FILE(PRINT) EDIT('END OF PHASE 3')
(SKIP(DEBUG),A(DEBUG*100));
CLOSE FILE(SCRATCH), FILE(PGMNEW);
PUT FILE(PRINT) EDIT('LEAVE CLEANUP')
(SKIP(DEBUG),A(DEBUG*100));
RETURN;
END CLEANUP;

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

Incorrigible errors

**WATFOR errors (see Blatt (6))**

<table>
<thead>
<tr>
<th>Error</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS-2,...,AS-4,BD-0,BD-1</td>
<td>low probability of occurrence</td>
</tr>
<tr>
<td>CM-0,...,CM-4,CN-0,...,CN-5,CN-8,CN-9</td>
<td>coding of correction modules is difficult</td>
</tr>
<tr>
<td>CP-0,...,CP-4</td>
<td>compiler error</td>
</tr>
<tr>
<td>DA-0,...,DA-9,DM-0,...,DM-4,</td>
<td>coding of correction modules is difficult</td>
</tr>
<tr>
<td>DO-1,...,DO-9</td>
<td></td>
</tr>
<tr>
<td>EC-0,...,EC-9,EN-0,...,EN-3</td>
<td>low probability of occurrence</td>
</tr>
<tr>
<td>EV-0,...,EV-4,EX-0,...,EX-9,</td>
<td>low probability of occurrence</td>
</tr>
<tr>
<td>EY-0,...,EY-6</td>
<td></td>
</tr>
<tr>
<td>FM-0,...,FM-7,FT-0,...,FT-F</td>
<td>coding of correction modules is difficult</td>
</tr>
<tr>
<td>FN-0,...,FN-8,GO-0,...,GO-4</td>
<td></td>
</tr>
<tr>
<td>HO-0,...,HO-4</td>
<td>low probability of occurrence</td>
</tr>
<tr>
<td>IF-0,...,IF-4</td>
<td>coding of correction modules is difficult</td>
</tr>
<tr>
<td>IM-0,...,IM-9</td>
<td>relatively low probability of occurrence</td>
</tr>
<tr>
<td>IO-0,...,IO-K</td>
<td>coding of correction modules is difficult</td>
</tr>
<tr>
<td>JB-1,JB-2,JB-3</td>
<td>not problem program errors</td>
</tr>
<tr>
<td>LC-2,LL-0,...,LL-H</td>
<td>low probability of occurrence</td>
</tr>
<tr>
<td>MD-2,...,MD-6,MO-0,...,MO-4,PC-0,PC-1</td>
<td>coding of correction modules is difficult</td>
</tr>
<tr>
<td>RE-0,...,RE-4,SF-1,SF-2,SF-3</td>
<td>low probability of occurrence</td>
</tr>
<tr>
<td>SR-0,...,SR-A,SS-0,SS-1,SS-2</td>
<td>coding of correction modules is difficult</td>
</tr>
<tr>
<td>UN-0,...,UN-9</td>
<td>violation of installation rules</td>
</tr>
</tbody>
</table>
Corrigible errors

<table>
<thead>
<tr>
<th>errors</th>
<th>ERRSET number</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC-0,...,CC-9</td>
<td>03</td>
</tr>
<tr>
<td>CN-6</td>
<td>03</td>
</tr>
<tr>
<td>ST-0,...,ST-A</td>
<td>16</td>
</tr>
<tr>
<td>SV-0,...,SV-5</td>
<td>16</td>
</tr>
<tr>
<td>SX-0,...,SX-n</td>
<td>16</td>
</tr>
</tbody>
</table>

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