ERROR CORRECTION IN CORC, 
THE CORNELL COMPUTING LANGUAGE

David N. Freeman
IBM General Products Division Development Laboratory
Endicott, New York

I. INTRODUCTION

CORC, the Cornell Computing Language, is an experimental compiler language developed at Cornell University. Although derived from FORTRAN and ALGOL, CORC has a radically simpler syntax than either of these, since it was designed to serve university students and faculty. Indeed, most of the users of CORC are "laymen programmers," who intermittently write small programs to solve scientific problems. Their programs contain many errors, as often chargeable to fundamental misunderstandings of the syntax as to "mechanical errors." A major objective of CORC is to reduce the volume of these errors. This objective has been achieved to the following extent: the average rate of re-runs for 4500 programs submitted during the fall semester of 1962 was less than 1.1 re-runs/program.

Three features of CORC have enabled it to achieve this low re-run rate:

1. Inherent simplicity of the syntax;
2. Closed-shop operation of the Cornell Computing Center on CORC programs, including keypunching, machine operation, and submission/return of card decks;
3. A novel and extensive set of error-correction procedures in the CORC compiler/monitors.

The CORC language is briefly described below; it is more fully documented elsewhere. The current paper describes the error-correction procedures in greater detail.

II. THE CORC LANGUAGE

CORC was designed by a group of faculty and students in the Department of Industrial Engineering and Operations Research at Cornell. This group has coded and tested two similar compiler/monitor systems, one for a medium scale decimal computer and the other for a large binary computer.

During the definition of the language, the design group surrendered potency to simplicity whenever the choice arose. Certain redundancies have been included in CORC, serving two functions: to facilitate error-correction during source-deck scanning, and to aid novice programmers' grasp of compiler-language syntax. Excepting these redundancies, CORC is quite frugal with conventions. For example, all variables and arithmetic expressions are carried in floating-point form, avoiding the confusing notion of "mode." At the same time, programmers are spared all knowledge of floating-point arithmetic.

Each CORC card deck is divided into three required sub-decks plus an optional sub-deck of data cards:

a. The preliminary-description cards supply heading data for each page of the output listing.
(b) The dictionary cards declare all variables used in the program, simple as well as subscripted.

(c) Each statement card may have an indefinite number of continuation cards. Statements may bear labels having the same formation rules as variables. Continuation cards may not be labelled.

Variables, labels, numbers, reserved words, and special characters comprise the symbols of CORC. Each symbol is a certain string of at most eight, non-blank characters. Numbers may have up to twelve digits; decimal points may be leading, trailing, or imbedded in the numbers. There are forty-three reserved words in CORC, e.g., LET, and ten special characters: + − * $ = (). The character string defining each label, variable, or reserved word is terminated by the first blank space or special character. The character string defining each number is terminated by the first character that is neither a digit nor a decimal point. Each special character is a distinct symbol. There are forty-six legal characters in CORC: letters, digits, and special characters.

A subset of the reserved words is the set of fifteen first-words: LET, INCREASE, INC, DECREASE, DEC, GO, STOP, IF, REPEAT, READ, WRITE, TITLE, NOTE, BEGIN, and END. The first symbol in each statement should, if correct, be one of these first-words.

There are eight executable-statement types, plus a NOTE statement for editorial comments on the source-program listing. (NOTE statements may be labelled; in this case, they are compiled like FORTRAN "CONTINUE" statements.) To simplify the description of the statement types, single letters denote entities of the CORC language:

V ...... a variable, simple or subscripted
E ...... an arithmetic expression, as defined in FORTRAN
L ...... a statement label
B ...... a repeatable-block label (see below)
R ...... one of the six relational operators: EQL, NEQ, LSS, LEQ, GTR, and GEQ. A relational expression is a predicate comprising two arith-

The statement types are as follows:

(1) LET V = E, and two variants INCREASE V BY E and DECREASE V BY E. (INCREASE may be abbreviated to INC, DECREASE to DEC.)

(2) IF E₁, R, E₂
    THEN GO TO L₁
    ELSE GO TO L₂, and two variants
    IF E₁₁, R₁, E₁₂
    AND E₂₁, R₂, E₂₂
    OR E₃₁, R₃, E₃₂
    .
    .
    AND Eₙ₁, Rₙ, Eₙ₂
    THEN GO TO L₁
    ELSE GO TO Lₙ
    ELSE GO TO L₂.

(3) GO TO L.

(4) STOP, terminating execution of a program.

(5) READ V₁, V₂, . . . , bringing in data cards during the execution phase. Each data card bears a single new value for the corresponding variable.

(6) WRITE V₁, V₂, . . . , printing out the variable names, the numerical values of their subscripts for each execution of the WRITE statement, and the numerical values of these variables.

(7) TITLE (message), printing out the remainder of the card and the entire statement fields of any continuation cards.

(8) REPEAT B . . . , comprising four variants
    (8a) REPEAT B E TIMES,
    (8b) REPEAT B UNTIL E₁₁, R₁, E₁₂
        AND E₂₁, R₂, E₂₂
        .
        .
        AND Eₙ₁, Rₙ, Eₙ₂,
    (8c) REPEAT B UNTIL E₁₁, R₁, E₁₂
        OR E₂₁, R₂, E₂₂
        .
        .
        OR Eₙ₁, Rₙ, Eₙ₂,
A principal tenet of the CORC philosophy is to detect errors as early as possible in:

(1) characters within symbols,
(2) symbols within expressions,
(3) expressions within statements, e.g., the left and right sides of an assignment statement, and
(4) statements within the sequencing of each program.

An explicit message for each error is printed on the output listing. This listing is the only output document from a CORC program; all programs are compiled and executed, and machine code is never saved on tape or punched cards.

After detecting a statement-card error, CORC always "repairs" the error by one of the two following actions:

(a) CORC refuses to compile a "badly garbled" statement. Instead, CORC replaces it with a source-program "message statement" reminding the programmer of the omitted statement.

(b) CORC edits the contents of a "less badly garbled" statement into intelligible source language. The edited statement is subsequently compiled into machine code.

Errors in cards other than statement cards are repaired by similar techniques.

Thus, the machine code produced by CORC is always executable, and compilation-phase and execution-phase error messages are provided for every program. By continuing compilation in the presence of errors, CORC provides diagnostic data simultaneously on structural levels (1)–(4) cited above. By also executing these programs, CORC detects additional errors in program flow, subscript usage, improper function arguments, etc.

The correction of a programming error is defined to be the alteration of relevant source-language symbols to what the programmer truly intended. Under this operational definition, many errors are incapable of "correction," e.g., the programmer may have intended a statement or expression not even offered in CORC. Other errors are capable of "correc-
tion" by the programmer himself but by no critic unfamiliar with the complete problem-definition; an incorrect numerical constant is again an example.

A third class of errors can be corrected by an intelligent critic after scanning the source-deck listing, without recourse to the problem definition. Some errors in this class require a profound use of context to elicit the programmer's true intention. Other errors in this class can be detected and corrected with little use of context, e.g., the omission of a terminal right parenthesis.

The author defines a corrigible error to be one whose correction is automatically attempted by the CORC compiler/monitor. Thus, this definition is by cases, for a specific version of CORC. CORC may correct one error and fail to correct a second, nearly-identical error. Error correction is a fundamentally probabilistic phenomenon; the CORC error-correction procedures attempt to maximize the "expected useful yield" of each program by strategies based on a priori probabilities associated with the different errors.*

The majority of corrigible errors are detected during the scanning of source decks by the CORC compiler. A few corrigible errors are detected during the execution of object programs. For each error, one or more correction procedures have been added to CORC, representing certain investments in core memory and operating speed.

The following paragraph discusses the selection of corrigible errors, and section IV catalogues these errors. The catalogue will be somewhat peculiar to the structure of CORC, a population of novice programmers, and the operation of a university computing center. However, the discussion of control-statement errors, arithmetic-expression errors, and mispellings is relevant to most compiler languages.

The author has roughly ranked various error conditions by two criteria: a priori probabilities* of their occurrence, and a priori probabilities of their correction (if correction is attempted). Correction procedures were designed for some errors, while other chronic errors had such low a priori probabilities of correction that only explicit error-detection messages were printed out. For example, omission of a subscript is a common error which is difficult to correct, although easy to detect and "repair." CORC "repairs" a subscript-omission error by supplying a value of 1.

On the other hand, misspellings are common errors whose a priori probabilities of correction are high if sophisticated procedures are used. The author hopes to achieve at least 75 percent correction of misspellings with the current procedures; many have not yet been tested in high-volume operation.†

IV. ERROR CORRECTION DURING SCANNING

First, the general procedures for card scanning will be described. The second, third, and fourth subsections deal with dictionary cards, data cards, and statement cards, respectively. The last subsection describes the error-correction phase which follows scanning, i.e., after the last statement card has been read but before machine code is generated by the compiler.

A. CARD SCANNING

Each CORC source deck should have all cards of one type in a single sub-deck:

1. Type 1, preliminary description cards
2. Type 5, dictionary cards
3. Type 0, statement cards
4. Type 4, data cards (if used).

The type of each card is defined by the punch in column 1 (although CORC may attempt to correct the type of a stray source card).

At the beginning of each new source program, CORC scans the card images (usually on magnetic tape) for the next type 1 card, normally a tab card bearing any non-standard time limit and page limit for this program. (The tab cards are used to divide the decks, facilitating batch processing and other handling.) This scanning procedure skips any extraneous data cards from the previous program.*

References 2 and 5 also propose probabilistic correction of misspellings.

† Probabilities in the sequel are estimates based on human scrutiny of several hundred student programs.

‡ Damereau has achieved over 95% correction of misspellings in an information-retrieval application.
gram deck. If the preceding deck was badly shuffled, misplaced dictionary cards and statement cards will also be skipped.

An indefinite number of type 1 cards may be supplied: CORC inserts data from the first two cards into the page headings of the output listing. This serves to label all output with the processing date and programmer name, avoiding losses in subsequent handling.

The problem identification should be duplicated into each deck; any deviations from this identification generate warning messages. The serialization of cards is checked, although no corrective action is taken if the cards are out of sequence. If the serialization is entirely omitted, CORC inserts serial numbers into the print-line image of each card, so that subsequent error messages can reference these print lines without exception.

The general procedure on extraneous or illegal punches is as follows: illegal punches are uniformly converted to the non-standard character "≠"; extraneous punches are ignored except in non-compact variable/label fields and in the statement field of type 0 cards, where all single punches are potentially meaningful. Rather than discard illegal punches, CORC reserves the possibility of treating them as misspellings. Likewise, any non-alphabetic first character of a variable/label field must be erroneous and is changed to "≠," furnishing a later opportunity to treat this as a misspelling. All hyphen punches are converted to minus signs during card reading; the keyboard confusion of these two characters is so chronic—and harmless—that CORC even refrains from a warning message.

B. DICTIONARY CARDS

Although the dictionary and data cards are processed in entirely different phases of a CORC program, their formats are identical—with the exception of column 1—and common procedures are used to scan them. As mentioned in the preceding subsection, non-alphabetic first characters are changed to "≠." Embedded special characters are similarly changed with the following exception: character strings of the form "(1)" or "(I,J)" are omitted. Fixed-column subscript fields have already been provided and students consistently and correctly use them. However, a common student error is to supply redundant parenthesized subscripts in the label field; these are ignored by CORC, although a warning message is supplied.

Non-numeric characters in the subscript fields and the exponent field are changed to "I"s. Vector subscripts can appear in either the first-subscript field or the second-subscript field. These subscripts need not be right-justified in their respective fields. After an array has been defined, subsequent subscripts of excessive magnitude are not used; the corresponding data entries are put into the highest legal cell of the array.

C. DATA CARDS

All of the foregoing procedures apply with these exceptions: if a data card has its variable field blank or, in the case of subscripted variables, its subscript fields blank, the data can still be entered with a high probability of correcting the omission. Information in the READ statement overrides incorrect or missing entries on the corresponding data cards. CORC insists on exact agreement of the variables and subscripts if warning messages are to be avoided. Symbolic subscripts may be used in READ statements, but their execution-phase values must agree with the numeric subscripts on the type 4 cards.

D. STATEMENT CARDS

Correction of erroneous statement cards is a complex technique—and the most fruitful of those currently implemented in CORC. Statement cards comprise over 80% of student source decks, on the average. Students commit the overwhelming majority of their errors in communicating imperative statements to a compiler, rather than header statements, declarative statements, or data cards. Statement-card errors fall into two major categories: those detectable at compilation time and those detectable only at execution time. The second category is discussed in section V. Some of the most useful correction techniques for the first category—tested and modified during the past two years of CORC usage—are described in the following eight sub-sections.
(1) Misspellings\(^{1,5}\)

At the end of Section III, misspellings were cited as a class of errors that both occur frequently and have attractively high \textit{a priori} probabilities for correction. Accordingly, CORC now contains a subroutine that compares any \textit{test word} to any list of words (each entry being denoted a \textit{list word}), determining a "figure of merit" for the match of each list word to the test word. Each figure of merit can be considered as the \textit{a posteriori} probability that the test word is a misspelling of this particular list word. The list word with the highest figure of merit is selected as the spelling of the test word "most likely" to be correct.

Various categories of misspelling are defined in CORC; to each category is assigned an \textit{a priori} probability of occurrence. When the test word and a list word match within the scope of a category, i.e., the test word is some particular misspelling of the list word, the \textit{a priori} probability for this category is added to the figure of merit for this list word. Actually, the figures of merit are integers rather than probabilities; they can be converted to probabilities by the usual normalization, but this is unnecessary—they are used merely to rank the possible misspellings.

All increments used in misspelling analyses reflect the number \(N\) of non-blank characters in the test word, as follows: a certain base-value increment is specified for each misspelling; if a match is found, this base value is multiplied by the ratio \(N/8\), then added to the corresponding figure of merit.

(a) A \textit{concatenation misspelling} occurs when a delimiting blank is omitted between two symbols, e.g., "LET X . . ." is a concatenation misspelling of "LET X . . .". When such a misspelling is detected, any relevant list of words is compared against the concatenated symbol. The increment to the figure of merit for each list word is computed as follows:

(i) If the list word and the test word do not have at least their initial two characters in common, the increment is 0.

(ii) For every consecutive character in common with the list word (after the first character), an increment of 2 is added to the figure of merit.

Example: Assume that the test word is ENTRYA and that two of the list words are ENT and ENTRY. The corresponding figures of merit are 6 and 10, respectively. The higher figure reflects the more exact agreement of ENTRY to ENTRYA.

(b) Single-character misspellings provide four different increments to the figure of merit, corresponding to mutually exclusive possibilities:

(i) A \textit{keypunch-shift misspelling} occurs when the IBM 026 keypunch is improperly shifted for the proper keystroke, e.g., a "1"-"U" error. There are fourteen possible misspellings of this type, corresponding to the seven letter-number pairs on the keyboard. The special character row, including "0," does not seem susceptible to misspelling analysis, since special characters are always segregated, never imbedded in symbols.

For each list word which agrees within a single keypunch-shift misspelling with the test word, an increment of \((20N/8)\) is added to the corresponding figure of merit, where \(N\) is the number of nonblank characters in the test word.

(ii) An \textit{illegal-character misspelling} occurs either (a) when a variable/label has previously required a "single-letter perturbation" using the character "\(\neq\)" or (b) when an illegal punch in the card is changed to "\(\neq\)." Single-letter perturbations are used when the same symbol occurs at both a variable and a label, or when a reserved word is used as a variable or label. In either case, conflicting usage cannot be tolerated, and CORC appends "\(\neq\)" to the symbol for the current usage. In subsequent searches of the symbol dictionary, one may wish to recognize the origi-
inal spelling. Thus, for each list word which agrees within a single illegal-character misspelling with the test word, an increment of \((20N/8)\) is added to the corresponding figure of merit, where \(N\) is as above. This increment is higher than that for a random misspelling, reflecting the peculiar origins of the character “\(≠\).”

(iii) A resemblance misspelling occurs whenever any of the following character pairs is confused: “1”-“1,” “O” (the letter)-“0” (the number) and “Z”-“2.” For each list word which agrees within a single resemblance misspelling with the test word, an increment of \((40N/8)\) is added to the corresponding figure of merit, where \(N\) is as above.

(iv) A random misspelling occurs when any other single character is misspunched in a symbol. For each list word which agrees within a single random misspelling with the test word, an increment of \((10N/8)\) is added to the corresponding figure of merit, where \(N\) is as above.

(c) A permutation misspelling provides a single increment to a figure of merit whenever the test word matches the corresponding list word within a pair of adjacent characters, this pair being the same but permuted in the two words, e.g., LTE is a permutation misspelling of LET. For each list word which agrees within a single permutation misspelling with the test word, an increment of \((20N/8)\) is added to the corresponding figure of merit, where \(N\) is as above. Other permutations may deserve consideration at some future date, but adjacent-pair permutations seem to have the highest \(a\) priori occurrence probabilities.

(d) Simple misspellings of the foregoing types have high probabilities of successful correction insofar as the following conditions are met:

(i) The list of words does not contain many nearly-identical entries. Otherwise, there will be many reasonable misspelling possibilities from which the program may select only one.

(ii) Neither test words nor list words are single-character symbols. The program excludes such list words from consideration during a misspelling analysis; experience has shown that only a small proportion—perhaps 10 percent—of single-character symbols are successfully corrected.

(iii) Context can be extraordinarily helpful. Associated with each list word is a set of attributes such as the count of its usage in the current program, its function (variable, label, constant, reserved word, etc.), and any peculiar usages already detected (such as being an undeclared variable). Certain misspelling possibilities can be immediately discarded if the context associated with the corresponding list words does not match the context of the test word. For example, if an arithmetic statement is being analyzed, any test for misspelled variables can immediately discard all misspelled label possibilities.

The first two of these three conditions are controlled by the vocabulary of the source-deck programmer; CORC gives far better assistance to programs using only a few variables and labels of highly distinctive spelling with at least three characters apiece.

(e) The increments corresponding to different misspellings were arbitrarily selected; they can be readily raised or lowered as experience indicates. The current values reflect the following observations:

(i) The weakest communication link is between the handwritten coding sheets and their interpretation by the keypunch operator. Hence, the
largest increment is assigned to resemblance misspellings.

(ii) In lieu of exact information, permutation misspellings and key-punch-shift misspellings have been judged equally probable.

(iii) Illegal punches in a card image arise from three sources: illegal hole patterns, improper use of a character (e.g., non-alphabetic character beginning a first word, or the duplicate use of a symbol as two entities), and card-reading failures. Lacking other evidence, the author considered the increment to be approximately the same as in (ii).

(iv) Other single-character misspellings seem only half as likely to occur.

Examples of the current CORC misspelling analyses may be found at the end of subsection E on Post-Scanning Spelling Corrections.

(2) Subscripts

Correction attempts for subscript errors have low success probabilities, on the whole. Isolated omission of one or both subscripts seems almost hopeless. CORC edits such an omission by appending "(1)" to a vector variable and "(1, 1)" to a matrix variable. Likewise, if a matrix variable has other than two subscripts, CORC uses primitive editing techniques to produce executable machine code. Excessive commas are changed to "÷" signs, and "(E)" is changed to "(E, 1)," where "E" is the arithmetic expression for the first subscript of a matrix variable.

Missing right parentheses are supplied and extra right parentheses are deleted as necessary, although not always correctly.

Definition of new array variables after the dictionary is complete (i.e., after all type 5 cards have been processed) is an attractive—if difficult—error-correction procedure. Most algebraic compilers scan source decks several times; they have a leisurely opportunity to accumulate evidence for undeclared array variables. If such evidence is overwhelming, i.e., if every usage of a certain variable is immediately followed by a parenthesized expression, these compilers could change the status of this variable before the final code-generation scan.

To reduce compilation time, the current version of CORC scans each source statement once and must make an immediate decision when it finds a left parenthesis juxtaposed to a supposedly simple variable: should "V(. . .)" be changed to "V*(. . .)," i.e., implied multiplication, or should it be treated as a subscript (and re-designate "V" as an array variable)? The present error-correction procedure is to encode "V(. . .)" into the intermediate language without change; special counters for usage as a vector/matrix variable are incremented, depending on one/two parenthesized arguments. At the conclusion of scanning, these usage counters are tested for all "simple" variables. Any variable used preponderantly as a vector variable causes CORC to test for the misspelling of some declared vector variable. Failing this, CORC changes the status of the variable to a vector of 100 cells. Any variable used preponderantly as a matrix variable causes CORC to test for the misspelling of some declared matrix variable. Failing this, CORC changes the status of the variable to a matrix of 2500 cells, comprising a $50 \times 50$ array.

If a variable is infrequently juxtaposed to parenthesized expressions, CORC treats these juxtapositions as implied multiplications. Deferral of this decision necessitates a procedure for inserting the multiplication operator during the conversion of intermediate language to machine code, together with the appropriate message. This error-correction procedure is one of the few in the code-generation phase. The message appears at the end of the source-deck listing rather than adjacent to the offending card image; the gain in error-correcting power seems to justify deferring the message.

The a priori probabilities of omitted array-variable declarations and implied multiplications are both high. Since the two possibilities are mutually exclusive, CORC bases its choice on the percentage occurrence of the ambiguous usage. If the usage is chronic, i.e., comprising more than 50 percent of the total usage of some variable, an undeclared array variable seems more probable. If the ambiguous usage is a
small percentage of the total usage, implied multiplication seems more probable.

(3) Arithmetic and relational expressions

The rules for analyzing and correcting arithmetic expressions are as follows:

(a) Extraneous preceding plus signs are deleted, and preceding minus signs are prefixed by zero, i.e., “−E” becomes “0−E.”

(b) Thereafter, “+,” “−,” “∗,” and “/” are all binary operators. If an operand is missing before or after a binary operator, the value “1” is inserted. This merely preserves the coherence of the syntax; to correct this error seems hopeless.

(c) If an expression using two binary operators might be ambiguous (irrespective of the formal syntax), CORC prints out its resolution of the ambiguity, e.g., “A/B∗C IS INTERPRETED AS (A/B)∗C.”

(4) LET, INCREASE-BY, and DECREASE-BY

Four components are essential to each correct statement in this category: the first-word, the assigned variable, the middle symbol, and the right-hand-side (RHS) arithmetic expression.

(a) The first-word of the statement has been identified by a generalized pre-scan of the statement. If “LET” has been omitted but “=” has been found, CORC furnishes the former symbol.

(b) The assigned variable may be subscripted; if so, CORC supplies any missing arguments, commas, and right parentheses when “=” or “BY” terminates the left-hand-side (LHS) of the statement. If other symbols follow the assigned variable but precede “=” or “BY,” they are ignored.

(c) “EQU,” “EQL,” and “EQ” are erroneous but recognizable substitutes for “=”.

(d) Any arithmetic expression is legal for the RHS.

(5) GO TO, STOP, and IF

(a) With one exception—(b) just below—all unconditional branches begin with “GO,” followed by an optional “TO.”

(b) STOP is a complete one-word statement. Also, it may be used in the conditional-branch statement, e.g., “IF . . . THEN STOP ELSE GO TO . . . .”

(c) A conditional branch always follows one or more relational expressions in an IF or REPEAT statement. For IF statements, the first incidence of “THEN,” “ELSE,” “GO,” “TO,” or “STOP” terminates the last relational expression; missing operands, commas, and right parentheses are then inserted as needed. Thereafter, the two labels are retrieved from any “reasonable” arrangement with two or more of the above five words.

Missing labels are replaced by dummy “next statement” labels, which later inhibit the compilation of machine-code branches. Thus, if an IF statement lacks its second label, the falsity of its predicate during execution will cause no branch. At the end of scanning, certain labels may remain undefined; here also, CORC inhibits the compilation of machine-code branches.

(6) REPEAT

(a) If the repeated label is omitted, e.g., in the statement REPEAT FOR ARG = 2, CORC scans the label field of the following source card. Programmers often place repeatable blocks directly after REPEAT statements using these blocks: Hence, any label on this following card is likely to be the missing repeated label: it is inserted into the REPEAT statement. If no such label is found, CORC creates a dummy label for the repeatable block. During the execution of the program, usage of this erroneous REPEAT statement can be monitored by this dummy label.

(b) If the REPEAT-FOR variant is used, CORC tests for three components in addition to the repeated label:

(i) The bound variable, i.e. ARG in the example in 6(a).
(ii) The character "=" or its erroneous variants "EQU," "EQL," and "EQ."

(iii) Any collection of iteration triples and single arithmetic expressions, separated by commas. In any iteration triple, CORC will supply a single missing argument with value "1."

(c) As in IF statements, an indefinite number of relational expressions can be used in REPEAT-UNTIL statements.

(7) BEGIN and END

REPEAT statements and repeatable blocks require consistent spelling of labels and matching BEGIN/END pseudo-statements. Through misunderstanding or carelessness, novice programmers commit grievous errors in using REPEAT statements and their blocks. CORC attempts to correct a certain subset of errors whose correction probabilities are attractively high:

(a) If the label of a BEGIN pseudo-statement is missing, the preceding and following cards are tested for clues:

(i) if the preceding card was a REPEAT statement using a yet-undefined label, this label is supplied to the BEGIN pseudo-statement.

(ii) If (i) fails to hold and if the following card is labelled, this label is shifted to the BEGIN pseudo-statement.

(iii) Otherwise, a dummy label is supplied, awaiting further clues to the identity of the repeatable block. If such clues never appear, the block is closed by a CORC-supplied END pseudo-statement after the last statement card of the deck.

Should an unpaired END pseudo-statement be subsequently found, the dummy label (on the BEGIN pseudo-statement) is changed to match this unpaired END label.

(b) If the label for an END pseudo-statement is missing, CORC tests for the existence of a "nest" of unclosed blocks. If so, the label of the innermost unclosed block is used in the current END pseudo-statement. Otherwise, the card is ignored.

(c) If the label in an END pseudo-statement does not match the label of the innermost unclosed block, the current label is tested against the labels of the entire nest of blocks. If a "crisscross" has occurred, i.e.,

\[
\text{A \cdot BEGIN} \nonumber \\
\cdot \\
\text{B \cdot BEGIN} \nonumber \\
\cdot \\
\text{A \ END,} \\
\]

CORC inserts the END pseudo-statement for block B before the current END pseudo-statement for block A.

(d) If the preceding test fails, CORC again tests the current label against the nest, looking for a misspelling. If the current label is misspelled, procedure (c) is used. If the misspelling tests fail, CORC ignores the END pseudo-statement.

(e) If the student has programmed an apparent recursion, CORC prints a warning message but takes no further action. Although unlikely, there may be a legitimate use for the construction:

\[
\text{A \ BEGIN} \nonumber \\
\cdot \\
\text{REPEAT A \ldots} \nonumber \\
\cdot \\
\text{A \ END.} \\
\]

In this situation, CORC makes no attempt to preserve the address linkages as a truly recursive routine would require. Thus, the program is likely to terminate in an endless loop.

(8) READ and WRITE

Only simple or subscripted variables can appear in READ statements. The subscripts can
be any arithmetic expressions. If a label appears in the argument list of a WRITE statement, the current count of the label usage will be printed. Constants, reserved words, and special characters are deleted from the argument lists of READ/WRITE statements.

E. POST-SCANNING SPELLING CORRECTIONS

The misspelling of labels and variables is corrected—insofar as CORC is capable—after scanning an entire deck, with the exceptions mentioned in section D. After scanning, much usage and context data have been accumulated. CORC attempts to resolve suspicious usages by equating two or more symbols to the same entity.

When the implementation of CORC was originally under study, heavy weight was given to the potential benefits from correcting misspellings. Efficient correction of misspellings seemed to require one of the following similar strategies:

(a) Two or more complete scans of the source deck, the first serving primarily for the collection of data on suspicious usages such as possible misspellings.

(b) Encoding of the source deck into an intermediate language which is tightly packed and substantially irredundant but which also permits re-designation of labels and variables after misspelling analyses.

A third alternative to these strategies was to compile the source deck directly into machine code, then attempt to repair this code after determining the set of corrigible misspellings. However, this procedure seemed less flexible to use and more difficult to program than the first two strategies; it was rejected from consideration.

The second alternative was selected and appears in both current implementations of CORC. Details of the strategy are as follows:

(a) Each new simple variable entered into the dictionary is paralleled by a pointer-cell containing the address of a second cell. This address is ordinarily used during machine code-generation to represent the variable in question. Since any misspelled variable is equated to a properly-spelled variable after scanning but before code generation, CORC corrects the misspelling merely by giving the variables identical pointer-cell contents.

(b) Each new array variable is paralleled by a pointer-cell containing the base address of the array. As for simple variables, only one pointer cell is changed if this variable is equated to another array variable.

(c) To each label corresponds a pointer-cell containing a branch instruction to the appropriate machine location (when the latter becomes defined during the generation of machine code). For an undefined label equated to some other label, its cell is filled with a branch instruction to the pointer-cell for the other label. Thus, execution of GO TO LABELA, where LABELA is a defined label, requires two machine-language branch instructions; if LABELA is an undefined label equated to LABELB, three machine-language branch instructions are required.

The penalty in compilation speed for using the intermediate language is modest: the average time to complete compilation for CORC programs—after the last statement card has been read—is less than one second; few decks require more than two seconds.

(1) Correction of misspelled labels

If a label has been referenced but never defined in a label field, it is tested for being a possible misspelling of some defined label. The defined label with the highest figure of merit is selected and the following message is printed:

LABELA IS CHANGED TO LABELB,

where LABELA and LABELB are the undefined and defined labels, respectively. If no defined label has a non-zero figure of merit with respect to the undefined label, the following message is printed:

LABELA IS UNDEFINED

Subsequently, all references to this label during the generation of machine language are
treated as "next-statement" branches. At execution time, any GO TO or REPEAT statements referencing this label cause the following messages, respectively:

IN STATEMENT __________ ,  
GO-TO NOT EXECUTED.

IN STATEMENT __________ ,  
REPEAT NOT EXECUTED.

(2) Correction of misspelled simple variables

(a) If an undeclared variable is never used in suspicious juxtaposition to parenthesized expressions (cf. subsection D(2) above), CORC attempts to find a declared simple variable meeting the following criteria:

(i) The undeclared variable is a potential misspelling of the declared variable.

(ii) The LHS–RHS usage of the declared variable is complementary to that of the undeclared variable.

By LHS–RHS usage is meant the following two frequencies:

(aa) Usage on the LHS of an assignment statement, in a READ statement, or in the initial dictionary. This usage corresponds to assigning the variable a new value.

(bb) Usage on the RHS of an assignment statement, in a relational expression, or in a WRITE statement. This usage corresponds to using the current value of the variable.

The motivation for LHS–RHS analysis is the following: if two variables are spelled almost identically, if one has a null RHS usage and the other a null LHS usage, then the a priori probability that the programmer intended a single entity is higher than the probabilities for most alternative misspellings.

CORC does not use LHS–RHS analysis alone to determine the best misspelling possibility. Instead, an increment of 5 is added to the figure of merit of each declared variable whose null usage complements any null usage of the current test word, i.e., undeclared variable. Undeclared variables can be equated only to declared variables, not to other undeclared variables.

(b) If a declared variable has a null RHS usage, it may be an erroneous dictionary spelling of some variable which is thereafter consistently spelled. However, CORC will announce that the dictionary spelling is "correct" in this case, after it detects the misspelling; all "misspelled" incidences of the variable are equated to the declared variable.

(3) Examples

Four groups of nearly-matching symbols are illustrated in Table I. In the first group, the label ABC requires testing for misspelling. The label ABCDE is a concatenation misspelling, figure of merit (FOM) = 6. The label ABD is a random misspelling, FOM = 3. The label BAC is a permutation misspelling, FOM = 7. The label AB≠ is an illegal-character misspelling, FOM = 7. Thus, CORC would choose at random between BAC and AB≠ for the defined label to which ABC should be equated.

In the second group, the defined label DEI has FOM = 15 with respect to the undefined label DEL.

In the third group, three simple variables have not been declared in the dictionary and require testing for misspelling. One should remember that only declared simple variables, i.e., XYZ and XYU, are eligible for identification with the undeclared variables. With respect to XYV, XYZ has misspelling FOM = 3; to this must be added the null-RHS increment of 5, making a total FOM = 8. Since XYU has only the misspelling FOM of 3 with respect to XYV, XYV is equated to XYZ.

With respect to YXZ, XYZ has a misspelling FOM of 7, plus the null-RHS increment of 5, making a total FOM of 12: since XYU has a zero FOM for YXZ, CORC equates YXZ to XYZ.

With respect to YXW, neither XYZ nor XYU has a positive FOM; thus, YXW is not equated to a declared variable.

In the fourth group, GHI was invariably used as a vector variable. Since it is a resem-
TABLE I. SAMPLE PROBLEMS IN POST-SCANNING SPELLING CORRECTIONS

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Declared/Defined?</th>
<th>LHS Usage</th>
<th>RHS Usage</th>
<th>Usage as Vector</th>
<th>Usage as Matrix</th>
<th>Total Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC</td>
<td>label</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABCDE</td>
<td>label</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABD</td>
<td>label</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BAC</td>
<td>label</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AB=</td>
<td>label</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEI</td>
<td>label</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEI</td>
<td>label</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XYZ</td>
<td>simple variable</td>
<td>yes</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>XYU</td>
<td>smp. var.</td>
<td>yes</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>XYV</td>
<td>smp. var.</td>
<td>no</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>YXZ</td>
<td>smp. var.</td>
<td>no</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>YXW</td>
<td>smp. var.</td>
<td>no</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>GHI</td>
<td>vector variable</td>
<td>yes</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>GH1</td>
<td>smp. var.</td>
<td>yes</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>GHJ</td>
<td>smp. var.</td>
<td>yes</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>GHK</td>
<td>smp. var.</td>
<td>no</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

balance misspelling of the declared vector variable GHI, it is equated to this variable and its status changed to a vector. GHJ was used 67 percent of the time as a vector variable; since it is a random misspelling of GHI, it is equated to the latter. GHK has a positive figure of merit with respect to each of the three preceding entries. However, GHK was never used as a vector variable. Since the GHJ and GHI have been set to vector status, GHK can no longer be equated to either of them; it thus remains a distinct, undeclared variable.

V. ERROR MONITORING DURING EXECUTION

CORC prefaces each compiled statement by a sequence of machine language instructions to monitor object-program flow. Additional "overhead" instructions for monitoring appear in four types of statements: labelled statements, statements containing subscripted variables, REPEAT statements, and READ statements. The monitoring effort has three objectives:

(a) Prevent the object program from overwriting the CORC compiler/monitor or itself;

(b) Continue the execution phase through untested code when the flow of the object program becomes confused (through misuse of REPEAT statements or incomplete GO TO, IF, and REPEAT statements);

(c) Provide explicit diagnostic messages for each error detected at execution time, followed by an unconditional post-mortem dump of simple-variable values and other helpful data.§

A. THE GENERAL MONITOR

(1) CORC accumulates a count of all statements executed, the statement count. This count is printed in the post-mortem dump, together with the number of errors committed during the entire program and the total elapsed time for the program. The statement count has two minor functions: to aid debugging of

§ Many debugging languages such as BUGTRAN (cf. 6) furnish trace and snapshot information if requested by the programmer. CORC furnishes such diagnostic information unconditionally; the overhead instructions cannot be suppressed after programs are debugged.
short programs in conjunction with the “label tallies” (see (3) below) and—looking towards future CORC research—to exhibit the different speeds of execution for various programs, e.g., with/without heavy subscript usage. The per-statement overhead of the statement count is 13.2 microseconds, comprising a single “tally” instruction.

(2) Before executing each statement, its source-card serial number (converted to a binary integer) is loaded into an index register. Execution-phase messages resulting from this statement retrieve the serial number and print it as an introductory phrase to each message, e.g.,

IN STATEMENT 1234, THE PROGRAM IS STOPPED.

Each load-index instruction requires 3.3 microseconds. The percentage of execution time devoted to items (1) and (2) is usually less than 3 percent; see (5) below.

(3) The execution of each labelled statement is tallied, by label. These tallies are printed in the post-mortem dump; they show the progress of the program, which branches were never taken, endless loops, etc. Each tally instruction requires 13.2 microseconds.

(4) At each labelled statement, a two-position console switch is interrogated. In the normal position, the switch has no effect on program flow. If set, the switch causes the program to terminate at once, printing the message,

IN STATEMENT ________, THE PROGRAM IS MANUALLY INTERRUPTED,

followed by the usual post-mortem dump.

Thus, any endless loop can be manually interrupted without stopping the computer, although this is rarely necessary. (Cf. the subsequent section on Terminations.) The switch interrogation is required only at labelled statements, since endless loops must include at least one label. Each switch interrogation requires 7.2 microseconds. The percentage of execution time devoted to items (3) and (4) is usually less than 1 percent, as exhibited by the following analysis.

(5) Assuming that 100,000 statements are executed per minute, an average statement requires some 600 microseconds. Since items (1) and (2) aggregate 16.5 microseconds per statement, the overhead for these items is 2.75 percent. Assuming that every fourth statement is labelled, items (3) and (4) are incurred once every 2400 microseconds on the average; since these times aggregate 20.4 microseconds, their overhead is approximately 0.8 percent.

(6) No tracing features are offered in CORC. If a student requires more diagnostic data than is already furnished, he is encouraged to use WRITE and TITLE statements generously. However, he is also warned to print such data compactly:

(a) If two consecutive pages print less than 30 percent of the 14,400 character spaces available (2 pages × 60 lines/page × 120 characters/line), CORC prints out the following message:

——TRY TO USE MORE EFFICIENT WRITE AND TITLE STATEMENTS AND AVOID WASTING SO MUCH PAPER——

(b) A page-count limit is set for all normal programs; when this limit is reached, the program is terminated at once.

(7) Each untranslatable source card has been replaced by a TITLE card during scanning, bearing the following message:

CARD NO. ________, NOT EXECUTED, SINCE UNTRANSLATABLE.

These messages remind the programmer of omitted actions during the execution phase.
B. MONITORING ARITHMETIC ERRORS

CORC uses conventional procedures for arithmetic overflow/underflow errors, but somewhat novel procedures for special-function argument errors. The machine traps of the computer detect overflow/underflow conditions, which are then interpreted into CORC messages:

(1) IN STATEMENT EXPONENT UNDERFLOW. (CORC zeros the accumulator and proceeds.)
(2) IN STATEMENT EXPONENT OVERFLOW. (CORC sets the accumulator to 1 rather than to some arbitrary, large number. This tends to avoid an immediate sequence of identical messages, allowing the execution phase to survive longer before termination from excessive errors.)
(3) IN STATEMENT DIVISION BY ZERO. Assume quotient of 1.0.

For each special function error, CORC creates an acceptable argument and proceeds, instead of taking drastic action, e.g., immediate program termination, as many monitor systems do.

(4) IN STATEMENT EXP SIN EXPRESSION TOO LARGE. THE RESULT IS SET TO 1.
(5) IN STATEMENT LN 0 YIELDS (or . . . LOG 0 YIELDS) 1.
(6) IN STATEMENT LOG SQRT OF NEGATIVE ARGUMENT. THE ABSOLUTE VALUE IS USED.
(7) IN STATEMENT ZERO TO NEGATIVE POWER—ASSUME 1.
(8) IN STATEMENT $—NEGATIVE ARGUMENT. THE RESULT IS SET TO 1.

C. TERMINATIONS

Two abnormal terminations were discussed in the General Monitor section. Altogether, there are five terminations, caused by the following events:

(1) Console switch set.
(2) Page count limit exceeded.
(3) Time limit exceeded. Overflow of the real-time clock produces a machine trap which is intercepted by CORC. For each program, a time limit (ordinarily of sixty seconds) is set. (The tab cards separating the source decks can bear any non-standard page-count and time limits.||) When this time is exhausted, the program is terminated with the following message preceding the post-mortem dump:

IN STATEMENT THE TIME IS EXHAUSTED.

Endless loops are terminated by this procedure, avoiding the necessity of operator intervention with the console switch.

(4) Error count too high. After each program has been compiled, the total error count is interrogated. When it exceeds 100, then or thereafter, the program is terminated with the appropriate message.

(5) Normal execution of STOP. The message

IN STATEMENT THE PROGRAM IS STOPPED

identifies which STOP statement—possibly of several such statements—has been met. For all terminations, the post-mortem dump includes the following:

(a) The final values of all simple variables. Since arrays may comprise thousands of cells, CORC cannot afford paper or machine time to dump them too.
(b) The usage tallies for all labels.
(c) The first fifteen (or fewer) data card images.
(d) The error-count, statement-count, and elapsed-time figures.

|| Ordinarily the tab cards are blank. A special rerun drawer is used for programs which require unusual output volume or running time; the computing center inserts special tab cards with non-standard page-count and time limits before these decks.
D. MONITORING SUBSCRIBED VARIABLES

One of CORC's most radical innovations is the universal monitoring of subscripts. CORC is attempting to trade execution efficiency for two other desiderata:

(a) Protection of the in-core compiler/monitor against accidental overwriting by student programs.

(b) Provision of complete diagnostics on all illegal subscripts: in which statements, for which variables, and the actual erroneous values of the subscripts.

CORC's excellent throughput speed has depended on infrequent destruction of the in-core compiler/monitor; in the author's opinion, subscript monitoring is CORC's most important protective feature.

Criterion (b)—full diagnostic information on subscript errors—is also of significance, since erroneous subscript usage comprises at least 30 percent of all execution-phase errors. Students quickly learn that these errors are among the easiest to commit—although they are spared the hardship of their detection and isolation.

Subscript usage is monitored as follows:

(1) Each reference to a subscripted variable incurs a load-index instruction corresponding to the dictionary entry for this variable. If subsequent troubles arise in the subscripts, CORC can retrieve the name and other particulars of the variable by using this index register.

(2) The subscript is an arithmetic expression, whose floating point value is transmitted in the machine accumulator to a closed subroutine for unfloating numbers.

(3) The latter subroutine checks for a positive, integral subscript.

(a) 0 is changed to 1 with the following message:

IN STATEMENT _____, SUBSCRIPT FOR VARIABLE _____ IS 0. IT IS SET TO 1.

(b) Negative numbers are also changed to 1:

IN STATEMENT _____, SUBSCRIPT FOR VARIABLE _____ IS NEGATIVE. IT IS SET TO 1.

(c) If non-integral, the subscript is rounded to an integer. If the round-off error is less than 10^-9, no error message is incurred; earlier calculations may have introduced small round-off errors into a theoretically exact subscript. If the round-off error exceeds 10^-9, the following message appears:

IN STATEMENT _____, SUBSCRIPT FOR VARIABLE _____ IS NON-INTEGRAL. IT IS ROUNDED TO AN INTEGER.

(d) After verifying (or changing to) a positive, integral subscript, the closed subroutine for unfloating subscripts returns control to the size test peculiar to this variable.

(4) The subscript is tested for exceeding the appropriate dimension of the array variable. Thus, the first subscript of a matrix variable is tested against the declared maximum number of rows, and the second subscript is tested against the declared maximum number of columns; a vector subscript is tested against its declared maximum number of elements. An excessive value incurs one of the three following messages:

IN STATEMENT _____, _____ IS THE \{FIRST SECOND \} SUBSCRIPT FOR THE VARIABLE _____ SINCE IT IS EXCESSIVE, IT IS REPLACED BY THE VALUE _____.

The second blank in the message is filled with the current execution-phase value of the subscript. The third and fourth blanks are filled with the variable name and its maximum allowable subscript. This action serves to repair the erroneous subscript but hardly to correct it.
The overhead for each error-free usage of a subscript is 85 microseconds. With obvious waste of effort, this overhead is incurred six times for the statement:

\[ \text{LET } A(I,J) = B(I,J) + C(I,J). \]

Future versions of CORC may treat such repeated usage of identical subscripts with more sophistication. However, one must remember that "A," "B," and "C" could have different maximum dimensions, in this example. A row subscript legal for "A" might be excessive for "B," etc. Also, in statements such as

\[ \text{LET } A(I) = A(I+1), \]

one must corroborate the legal size of "(I+1)" as well as that of "I."

The per-program overhead of subscript monitoring varies between 0 percent and 90 percent of the execution time, as one might guess. An average overhead of 15 percent has been measured for a representative batch of programs.

E. MONITORING REPEATED BLOCKS

1. Each repeatable block is legally used only as a closed subroutine. Hence, the exit instruction from the block—machine code generated by its END pseudo-statement—can be used to trap any illegal prior branch to an interior statement of the block. (One cannot enter a block by advancing sequentially through its BEGIN pseudo-statement. However, one can illegally branch to an interior statement of a repeatable block from a statement physically outside the block.) When the block is properly entered by a REPEAT statement, the address of the exit instruction is properly set; after the repetitions have been completed, a trap address is set into this exit instruction before the program advances beyond the REPEAT statement.

Thus, program flow can physically leave and re-enter a repeatable block in any complex pattern, as long as the block has been properly "opened" by a REPEAT statement and has not yet been "closed" by completion of the repetitions. In this respect, CORC allows more complex branching than most compilers.

When the exit instruction traps an illegal prior entry, CORC prints the following message:

IN STATEMENT ________, AN ILLEGAL EXIT FROM BLOCK _______ HAS JUST BEEN DETECTED. IN SOME PREVIOUS GO-TO STATEMENT, THE BLOCK WAS ILLEGALLY ENTERED. THE PROGRAM CONTINUES AFTER THE END STATEMENT OF THIS BLOCK.

2. To protect against various illegal usages of the bound variable in REPEAT-FOR statements, CORC pre-calculates the number of repetitions and conceals this count from the repeatable block; the count is fetched, decremented, and tested only by the REPEAT statement. This discussion is amplified in (d) below.

Consider the statement: REPEAT B FOR V = (E1, E2, E3):

(a) If \( E_1 \leq E_3 \), the block is executed once.

(b) Otherwise, if \( E_2 \) is zero, CORC prints the following message: IN STATEMENT ________, IN REPEAT-FOR TRIPLE, SECOND ARGUMENT IS 0. THE REPEAT IS EXECUTED ONCE.

(c) Otherwise, if \( (E_3 - E_1)/E_2 \) is negative, CORC prints the following message:

IN STATEMENT ________, IN REPEAT-FOR TRIPLE, SECOND ARGUMENT HAS WRONG SIGN. THE REPEAT IS EXECUTED ONCE.

(d) Otherwise, CORC uses the count \[ \left\lfloor \frac{E_3 - E_1}{E_2} \right\rfloor \] to determine the number of repetitions. This count is reduced by 1 for each iteration, irrespective of the subsequent values of "V," "E_2," or "E_3." Novice programmers often manipulate "V" inside repeatable blocks; CORC pre-
vents many potentially endless loops by ignoring this manipulation.

F. MONITORING DATA-CARD INPUT

The reading and checking of data cards was introduced in Section IV. In brief, a READ statement causes the following steps to occur.

1. A new card is read in; if it is of type 1, CORC assumes it to be the first card of the next source deck. Thereupon, the following messages appear:

   THE INPUT DATA HAS BEEN EXHAUSTED. IN STATEMENT _____, CORC SUPPLIES A DATA CARD FOR THE VARIABLE ____ WITH VALUE 1.0.

   Thus, CORC enters a value of 1 for the READ variable and proceeds with the program; subsequent READ statements incur only the second message above.

2. If the new card is neither type 1 nor type 4 (i.e., the correct type), CORC prints this message:

   IN STATEMENT _____, THE CARD IS ASSUMED TO BE A DATA CARD.

3. If the new card is type 4—possibly as the result of (2) above—CORC checks the variable field against the variable name in the READ statement. If they disagree, CORC considers the name in the READ statement to be correct; the following message is printed:

   IN STATEMENT _____, THE VARIABLE ____ WAS READ FROM THE CARD. THE VARIABLE IN THE READ STATEMENT WAS _______.

4. When the variable names have been reconciled CORC checks for none, one, or two subscripts on the card, as appropriate to the READ variable. Missing or erroneous subscripts incur the following message:

   IN STATEMENT _____, THE SUBSCRIPT (______,______) WAS READ FROM THE CARD. THE SUBSCRIPT IN THE READ STATEMENT WAS (______,______).

In every case, CORC uses the value in the READ statement.

VI. CONCLUSIONS

A. EXPERIENCE IN PRACTICE

Throughout the 1962–63 academic year, CORC was in "pilot project" status; in 1963–64 CORC was established as the fundamental computing tool for undergraduate engineering courses at Cornell. In the spring semester of 1964, over 15,000 CORC programs were run, peaking at 2500 programs in one week.

The performance of CORC programmers far surpassed the preceding years’ performance by ALGOL programmers at Cornell in such respects as speed of language acquisition, average number of re-runs per program, and average completion time for classroom assignments.

Actual processing time can be evaluated from the following figures, which are rough estimates based on last year’s experience with CORC programs:

(a) Average processing time (tape/tape configuration)—500 programs per hour.

(b) Average machine-code execution rate—100,000 source-language statements per minute, for a random sample of twenty student programs.

(c) Average compilation time for CORC programs—less than two seconds.

(d) Turnaround time for programs—one day or less, with rare exceptions.

The author has automated the operation of the compiler/monitor to the following degree: only a random machine malfunction can cause the computer to halt. Since programming errors cannot produce object code that erroneously diverts control outside the CORC system, the
role of the machine operator is merely to mount input tape reels and remove output tapes: the computer console needs almost no attention.

A few error-detection procedures were altered during 1962-64, primarily to make diagnostic messages increasingly explicit. A new CORC manual was prepared for instructional use in 1963–64; this manual omitted any catalogue of errors, since the author expected that the compiler/monitor systems could describe the errors—and the corresponding remedial actions—in satisfactory detail.

CORC has imposed a modest load on the two computers at Cornell. The computing center is satisfied that neither FORTRAN nor ALGOL can lighten this load, which is rarely as much as two hours of CORC runs daily. (FORTRAN and ALGOL systems have greater capability but require more facility in programming. The class of problems for which CORC has been developed would not warrant the expenditure of time required to program in the advanced languages.) In the author’s opinion, this small commitment of resources is well-justified by the educational value of the CORC project.

B. POTENTIAL UTILITY OF CORC

The author feels that many universities and technical colleges can profitably utilize CORC for introductory instruction. The designers of CORC are convinced that a simple language is well suited for initial study; many Cornell students have already easily advanced to FORTRAN or ALGOL after mastering CORC.

With respect to the error-detection and error-correction features, CORC demonstrates the modest effort required to furnish intelligible messages and how little core memory and machine time are consumed. Many CORC error-monitoring procedures deserve consideration in future implementations of compiler languages: unconditional counts of statement labels (or statement numbers), source-program citations in diagnostic messages, and brief dumps following all program terminations. The monitoring of subscripts would not be burdensome if the latter were carried as integers—index registers are used in most current compiled codes. Ninety percent of the CORC subscript-

C. POTENTIAL IMPROVEMENTS IN CORC

Four areas for significant improvements in CORC are as follows:

1. Identification of integer-mode variables by their context. Index registers can then be used for arrays and loop counting as in FORTRAN.

2. A problem-grading mechanism. Each instructor can assign a scale of penalties for various errors. CORC will process his batch of student programs and assign the appropriate grades.

3. A permanent file for tabulating errors. Each time that CORC programs are run, an auxiliary output device—paper tape or punched cards—will record the serial number of each error committed. Periodically, these tapes or cards will be summarized. This data will furnish statistical estimates for the a priori occurrence probabilities of the errors.

4. Remote consoles. These are much discussed in current computer literature, and they hold unusual promise for high-volume university operation. Students would type in their programs from keyboards distributed around a campus covering hundreds of acres. Either these programs would interrupt a large computer programmed for real-time entry, or they would be stacked on tape/disk by a satellite computer. Perhaps results could be printed/typed at these remote stations by the satellite computer.

The author and his colleagues are well aware of shortcomings in the language. However, they intend to resist changes which increase the power of the syntax at the expense of linguistic simplicity. Changes on behalf of additional simplicity or clarity are willingly accepted. Continuing efforts will be made to improve the clarity and explicitness of the diagnostic messages, so that classroom instruction can be further integrated with output from the computer.

From the collection of the Computer History Museum (www.computerhistory.org)
VII. ACKNOWLEDGMENTS
The author is a former student of Professors Conway and Maxwell; he gratefully acknowledges their assistance to the error-correction project.

Other contributors were R. Bowen, J. Evans, C. Nugent, J. Rudan, and R. Sanderson.

VIII. REFERENCES


4. Ibid.
