NEATER2: A PL/I Source Statement Reformatter

KENNETH CONROW AND RONALD G. SMITH
Kansas State University, Manhattan, Kansas*

NEATER2 accepts a PL/I source program and operates on it to produce a reformatted version. When in the LOGICAL mode, NEATER2 indicates the logical structure of the source program in the indentation pattern of its output. Logic errors discovered through NEATER2 logical analysis are discovered much more economically than is possible through compilation and trial runs. A number of options are available to give the user full control over the output format and to maximize the utility of NEATER2 as an aid during the early stages of development of a PL/I source deck. One option, USAGE, causes NEATER2 to insert into each logical unit of coding a statement which will cause the number of times each one is executed to be recorded during execution. This feature is expected to provide a major aid in optimization of PL/I programs.

KEY WORDS AND PHRASES: logical analysis of PL/I source, reformatting of PL/I source, documentation aid, execution time usage data
CR CATEGORIES: 1.52, 4.12, 4.19, 4.42

Introduction

The availability of high level languages and the comparative ease of programming in them make it possible to attack problems which would have been beyond solution only a few years ago. For simple numerical problems and repetitive sequential operations, no particular difficulty is experienced in keeping in mind the logical structure of a program, and no particular aid is required to achieve logically correct coding before a program is exposed to the compiler where it is syntactically corrected. However, with a program which responds to a great variety of different input combinations in a great variety of different ways, the complexity of logical paths through the program soon surpasses comprehension and some mechanized programming aid which assists in reviewing and correcting its logical structure becomes economically attractive.

In response to our own need for such a programming aid in the PL/I language, we have developed a PL/I program called NEATER2.[1] NEATER2 accepts a PL/I source program and operates on it to produce a reformatted version. When in the logical mode, NEATER2 indicates the logical structure in the source program by the indentation pattern in its output. The source code is also neatened by omission of nonessential blanks within statements. Figure 1 illustrates NEATER2 output with several simple logical patterns and two more complex logical patterns. Figure 2 provides a more realistic example, which also illustrates the reason for speaking of the program as neatening a source deck.

In Figure 1 an example of NEATER2 output in which a minimum set of key words and delimiters that produce output in the logical format is demonstrated. The output has been rearranged to fit in a smaller space for the illustration; normally, the logical level numbers and the statement numbers appear at the right margin, and the two columns in the illustration appear as two separate pages. A diagnostic indicating an unexpected ELSE in the source stream is included to illustrate one result of the logical analysis.

In Figure 2 the performance of NEATER2 is illustrated in one example where its utility seems particularly high. In this example, a routine which reads in two matrices, A(FD, MD) and B(MD, LD), and forms the product matrix, P(FD, LD), is processed by the PL/I precompiler to assign specific dimensions to the matrices. The precompiler also changes each array into a vector and the precompiler subroutine provides an expression from which the location of each element in the vectors corresponding to each element in the matrices will be calculated. No pretense is made that this is a practical use of the precompiler; it merely illustrates the kind of use it may be put to. The precompiler output is treated by NEATER2, and the neatened version printed out.

The precompiler output is filled with many more unnecessary blanks than were present in its input. The logical structure in the program is far from obvious in either of the first two versions. The complete removal of unnecessary blanks within statements, the construction of logical formatting with the default format, the recognition of the label on the end statement, the behavior of the logical level, and the generation of statement numbers are all illustrated in the NEATER2 output at the bottom.

Utility of NEATER2

We have come to think of NEATER2 as a precompiler since it may be used prior to compilation of a source deck. In most of its modes of operation, NEATER2 is merely a reformatter—it changes the arrangement of the source coding without changing its content. Only when USAGE is on (see below) does NEATER2 act to change the source coding by adding statements in the source stream. NEATER2 is in no sense a replacement for the preprocessor stage of the PL/I compiler since it does an entirely dif-

* Kenneth Conrow is with the Department of Chemistry and Ronald G. Smith is with the Computing Center

[1] An earlier, less versatile, slower program, called NEATER is available from the IBM Corp., Program Information Dept., 40 Saw Mill River Road, Hawthorne, NY 10532, under order number 360D-03.6.018. An erratum is available from the authors.

```
                     KSU'S PL/I NEATENER AND PRECOMPILER              PAGE       1
N2DEMO:                                 1   1
   PROCEDURE(.....);                    1   1
   ....;                                1   2
      IF A....A THEN                    2   3
      .....;                            2   4
   .....;                               1   5
      IF A....A THEN                    2   6
      ......;                           2   7
      ELSE                              2   7
      A.....;                           2   8
   ......;                              1   9
      DO;                               2  10
      .....;                            2  11
      END;                              2  12
   ......;                              1  13
      IF A...A THEN                     2  14
         IF A...A THEN                  3  15
         .....;                         3  16
   ......;                              1  17
      IF A...A THEN                     2  18
         IF A...A THEN                  3  19
         .....;                         3  20
         ELSE                           3  20
         A...;                          3  21
   .....;                               1  22
      IF A...A THEN                     2  23
         IF A....A THEN                 3  24
         .....;                         3  25
         ELSE                           3  25
         A...;                          3  26
      ELSE                              2  26
      A...;                             2  27
   ......;                              1  28
      IF A...A THEN                     2  29
         DO;                            3  30
         ....;                          3  31
         END;                           3  32
   ....;                                1  33
      IF A...A THEN                     2  34
         DO;                            3  35
         ...;                           3  36
         END;                           3  37
      ELSE                              2  37
         IF A...A THEN                  3  38
            DO;                         4  39
            ....;                       4  40
            END;                        4  41
   .....;                               1  42
      BEGIN;                            2  43
         IF A...A THEN                  3  44
            IF A...A THEN               4  45
               DO;                      5  46
               ....;                    5  47
               END;                     5  48
            ELSE                        4  48
               IF A...A THEN            5  49
               ....;                    5  50
      END;                              2  51

   .......;                                                      1  52
***ERROR*** AT SEQNO   51     SRCELS
      ELSE                                                       2  52
      A....;                                                     2  53
      IF A...A THEN                                              2  54
         DO;                                                     3  55
            IF A...A THEN                                        4  56
               IF A...A THEN                                     5  57
               .....;                                            5  58
               ELSE                                              5  58
                  IF A...A THEN                                  6  59
                     IF A...A THEN                               7  60
                        DO;                                      8  61
                        ....;                                    8  62
                        END;                                     8  63
            IF A...A THEN                                        4  64
            ....;                                                4  65
         ....;                                                   3  66
         END;                                                    3  67
      ELSE                                                       2  67
      A...;                                                      2  68
   .....;                                                        1  69
      IF A...A THEN                                              2  70
         IF A...A THEN                                           3  71
         ....;                                                   3  72
         ELSE                                                    3  72
            IF A...A THEN                                        4  73
            ....;                                                4  74
            ELSE                                                 4  74
               IF A...A THEN                                     5  75
                  DO;                                            6  76
                  ...;                                           6  77
                  END;                                           6  78
               ELSE                                              5  78
                  IF A...A THEN                                  6  79
                  ....;                                          6  80
                  ELSE                                           6  80
                     IF A...A THEN                               7  81
                     ....;                                       7  82
                     ELSE                                        7  82
                        IF A...A THEN                            8  83
                           DO;                                   9  84
                           ....;                                 9  85
                           END;                                  9  86
   .....;                                                        1  87
   END N2DEMO;                                                   1  88
KSUCC1I STEP   1 NEAT      EXECUTION TIME =    .001 HRS.     RETURN CODE =     12
KSUOC2I JOB A0008368 EXECUTION TIME =   .001 HRS.
```

Fig. 1

ferent thing from the preprocessor. It does share the property of being a useful program to expose a source deck to before actual compilation is attempted.

At present, in the absence of an aid like NEATER2, one tends to postpone a serious search for logic errors until trial runs of the program have demonstrated its necessity. Such a delay of a search for logic errors until after syntax correction has been made represents a departure from the most desirable progression in program development. Used prior to compilation, NEATER2 makes convenient and economical the process of reducing a program concept into logically correct PL/I coding. NEATER2 will reveal logic errors by producing unexpected indentation patterns and will reveal a certain few syntax errors (e.g. missing colons or semicolons) by peculiarities in indentation pattern or by specifically flagging them (e.g. missing THENs or unexpected ELSEs). This process does not require syntactically correct PL/I since NEATER2 runs on colons, semicolons, quotes comments, and the very few keywords, IF ... THEN, ELSE, DO, BEGIN, PROC, PROCEDURE, END, and ON. (See Figure 1 for example.)

```
        MMULT:PROC;

COMPILE-TIME MACRO PROCESSOR
MACRO SOURCE2 LISTING

  1     MMULT:PROC;
  2     %DCL(FD,MD,LD,FDMMD,MDMLD,FDMLD)FIXED;
  3     %DCL GFSL CHAR;      %GFSL='GET FILE(SYSIN) LIST ';
  4     %DCL LINEAR ENTRY(CHAR,CHAR)RETURNS(CHAR);
  5     %FD=10;%MD=8;%LD=12;
  6     %FDMMD=FD*MD; %MDMLD=MD*LD; %FDMLD=FD*LD;
  7     DCL P(FDMLD),A(FDMMD),B(MDMLD);
  8     GFSL(A,B);
  9     DO I = 1 TO FD; DO K = 1 TO LD; PI = LINEAR(I,K); P(PI) = 0;
 10     DO J = 1 TO MD;
 11     P(PI) = P(PI) + A(LINEAR(I,J)) * B(LINEAR(J,K));
 12     %LINEAR:PROC(L,M)CHAR;
 13     DCL(L,M)CHAR;
 14     IF M='J' THEN RETURN(MD||'*('||L||'-1)+'||M);
 15     IF M='K' THEN RETURN(LD||'*('||L||'-1)+'||M);
 16     %END LINEAR;    END MMULT;

                GENERATED SOURCE STATEMENTS.

MMULT:PROC;                                                          1
DCL P(      120 ),A(       80 ),B(       96 );                       7 1
 GET FILE(SYSIN) LIST  (A,B);                                        8 1
DO I = 1 TO        10 ; DO K = 1 TO        12 ; PI =        12*(I-1)+K  9 1
; P(PI) = 0;                                                         9
DO J = 1 TO         8 ;                                             10 1
P(PI) = P(PI) + A(        8*(I-1)+J ) * B(       12*(J-1)+K );      11 1
     END MMULT;                                                     16


          KSU'S PL/I NEATENER AND PRECOMPILER          PAGE     1

MMULT:  PROC;                                                        1    1
        DCL P(120),A(80),B(96);                                      1    2
        GET FILE(SYSIN)LIST(A,B);                                    1    3
           DO I=1 TO 10;                                             2    4
              DO K=1 TO 12;                                          3    5
              PI=12*(I-1)+K;                                         3    6
              P(PI)=0;                                               3    7
                 DO J=1 TO 8;                                        4    8
                 P(PI)=P(PI)+A(8*(I-1)+J)*B(12*(J-1)+K);             4    9
        END MMULT;                                                   1   13
```

FIG. 2

Since NEATER2 economically reformats a source program, the initial keypunching can be rapidly done in free form with any number of statements per card, and a formatted listing and card deck with one statement per card obtained. The very first proofreading of the source deck can be done on a listing which simultaneously reveals logical structure. Since the programmer has at least a partial conception of the logical structure he requires, he can compare the logical structure of his coding efforts as revealed by NEATER2 with his concept to see whether he has correctly transcribed his intention. Detailed desk checking of the program logic with diverse sets of input data is greatly facilitated by a logically formatted version of the program.

When the program appears logically correct, a freshly neatened source deck may be obtained and syntactic correction accomplished by using the compiler for the first time. Having a freshly formatted and sequenced deck during syntax correction and early trial runs is a great convenience because each source listing appears in logical format. This sequence of program development is especially attractive because of its economy: NEATER2 runs from 3 to 6 times faster than the compiler. Hence, logic errors discovered through NEATER2 are much more economically discovered than those discovered by compilations and trial runs.

Certain terminal errors, namely unmatched quotes and unclosed comments, can cause immediate cessation of compilation by the PL/I compiler. This is a frustrating occurrence because additional submissions are required before the whole of the source deck is scanned. Since NEATER2 is relatively fast and since its object is logical analysis rather than production of compiled coding, continued processing of a source stream which contains these blunders can be tolerated, so NEATER2 was designed to accommodate these source errors in a more constructive way than the compiler does. NEATER2 attempts to localize the problem and to continue its analysis of the logic of the source program from an early point after the detection of the difficulty. The output from

NEATER2 in the event of unmatched quotes or unclosed comments is not suitable for compilation, and the errors are labeled as terminal errors.

In the case of an unmatched quote, the first semicolon after the start of the statement in which an unmatched quote was *recognized* is arbitrarily taken as the end of the statement, and analysis is resumed from that point after an appropriate error message has been given.

In the case of an unclosed comment, illogical handling of statements which intervene between the comment opener and the closure of the next comment in the source stream will be the indication of the omission. In the event that there is no closure within three to four thousand characters, NEATER2 merely isolates the "/*" of the opening, writes an error message, and proceeds to treat the comment as if it were a statement, which again makes obvious the omission.

To increase its effectiveness as a precompiler, NEATER2 has been programmed to return a completion code. If NEATER2 discovers an error (or makes an error which is detected by the PL/I execution time interrupts) then an error code will be returned to the operating system upon completion of NEATER2's execution. The scheme used for the error code is similar to that employed by the PL/I compiler: with a 16 returned for a terminal error; a 12 returned for severe errors; an 8 returned for errors, which it is supposed will not interfere with successful compilation and execution of NEATER2's output; and a 4 for warnings, which seem certain not to interfere with subsequent compilation and execution. If NEATER2 is used as a precompiler and its output is passed to the compiler, condition parameters may be used in the job control language on the compile step to prevent compilation if NEATER2 has detected errors of any specified level of severity. Similarly, a punched reformatted source deck can conditionally be obtained by use of condition codes on a PUNCH step after a NEATER2 step.

While NEATER2 is most useful as a precompiler, it also has important auxiliary utility as a documentation aid. Clearly, a listing of a program in which the logical structure is revealed by an indentation pattern is a far more valuabe documentation of the program than an ordinary, unindented listing. Persons acquainting themselves with a program for the first time will be able to visualize the logical structure without the laborious preparation of flowcharts.

In an academic environment, NEATER2 is a useful aid in precisely demonstrating to students the nature of their logical errors. Teachers and consultants can locate these errors much more rapidly in a neatened listing than in a listing of the student's original source deck.

Requirements for Use of NEATER2

NEATER2 has no special requirements or limitations in the source deck. Any keyword which is important to NEATER2 and which will be correctly identified by the compiler from its context will be correctly identified by NEATER2, as well. Statements like

"IF IF THEN IF THEN THEN A = B;"

are correctly formatted. Comments do not interfere with the logical analysis or upset the usefulness of the formatted version which NEATER2 produces.

While NEATER2 will correctly process any PL/I source deck, the advantage gained from the formatting it effects can be maximized by adopting the following attitude. The source code should make maximum use of DO groups as THEN or ELSE clauses and minimum use of THEN GO TO ... and ELSE GO TO ... statements. The use of DO groups in the THEN or ELSE clauses results in an indentation pattern which very graphically presents the logical situation in the source deck, whereas extensive use of conditional branches minimizes the logical indentation pattern and makes the NEATER2 output graphically less useful.

Widespread use of DO groups and minimal use of statement labels also facilitate the rearrangement of a source deck if it proves necessary to correct logic errors during its development. NEATER2 clearly identifies logical blocks. It puts every statement on a separate card. Therefore, both statements and logical blocks can easily and independently be moved from place to place within a neatened program. The few statement labels which are used are output on the left margin by NEATER2 so that they may easily be spotted in a quick scan of the program, and any changes in them necessitated by a rearrangement of the logical units of the program are easily made.

Parameters to NEATER2

Parameters are given to NEATER2 in the form of a PL/I comment statement beginning in column 2 of a card in the input stream:

/*NEATERPARMS: ...*/,

where the dots are replaced by the desired parameters without spaces but separated by commas. If the user wishes, he may omit a NEATERPARMS card, and NEATER2 will operate using a set of default parameters.

(a) COMPRESS | LOGICAL. When in the compress mode, NEATER2 removes all unnecessary blanks from a source program and outputs it as one massive block of type. By having it punch in this mode, a program deck is produced in its physically most compact form, convenient for storage or shipping to another installation. The logical format can be recovered by a second run through NEATER2 in the logical mode.

The logical mode is the mode which seems the most useful one; it is in this mode that NEATER2 produces logically formatted output and logical level numbers.

The level numbers put out by NEATER2 at the right of each output line when in the logical mode bear little relationship to the level numbers or the nesting level which are given on request by the PL/I compiler. NEATER2 augments the level for each IF, DO, BEGIN,

PROC, or PROCEDURE, and decrements it after each IF sequence is completed and after each END statement is processed. The principal use of the logical levels is to facilitate pairing of IFs and ELSEs or DOs and ENDs, especially when they are separated by a page or more. Logical analysis is not done in the compress mode; so level numbers are not put out in the compress mode.

(b) PRINT | NOPRINT. This option controls the production of printed output by NEATER2. If NEATER2 is to be used as a precompiler, with its "punch" output being passed to the PL/I compiler, then a duplicate listing of the source program would be obtained, and unnecessary time would be spent. Specifying either NOPRINT to NEATER2 or NOSOURCE to the PL/I compiler will eliminate the duplicate listing. When NEATER2 produces an error message, the sequence number of the concerned statement (± 1) is produced as part of the message; so errors can be associated with the statements which produced them even if NOPRINT has been specified, provided the compiler runs. In early runs, if condition codes on the compile step are likely to prevent the compiler from running, then, of course, PRINT should be specified so that it is certain that a listing with which to associate error messages with the source stream statements is available.

If PRINT is specified to NEATER2 and NOSOURCE is specified to the compiler, the compiler's error messages may be associated with the statements in the source stream because the sequence numbers produced by NEATER2 in the logical mode are identical with the statement numbers produced by the compiler on a syntactically correct source program. If syntax errors cause the compiler to insert semicolons or make other changes which alter the compiler's statement count, then, of course, NEATER2's sequence numbers will not correspond with the compiler's statement numbers.

(c) PUNCH|NOPUNCH. This option has two main utilities. When a source deck punched in free format is exposed to NEATER2, it is convenient to have it produce a punched formatted deck to use in further program development. At intervals during the program development as the logical formatting deteriorates due to insertions, deletions, and rearrangements, it is convenient to obtain a freshly punched formatted deck for use for a time until the logic has again changed so much that the formatting must again be repeated. This utility assumes rather infrequent runs through NEATER2.

Alternatively, NEATER2 can be used as a precompiler stage before compilation of a source program under development. In this event, the PUNCH option is used to generate a data set which serves as input to the compiler.

(d) COMMENT|NOCOMMENT. COMMENT has the effect of suppressing the removal of unnecessary blanks from COMMENT statements. Its utility is to maintain formatting which a user may have incorporated in his comments.

(e) DECLARE|NODECLARE. DECLARE is parallel to COMMENT; it has the effect of suppressing the removal of unnecessary blanks from declare statements. This is particularly important with declarations of complex structures, where programmers commonly use formatting to indicate the heirarchical structure. NEATER2 does not format declarations of structures; the DECLARE option merely permits the programmer's formatting to pass through NEATER2 without loss.

(f) USAGE|NOUSAGE. The parameter USAGE causes NEATER2 to insert into each block, group, or clause of the source coding a statement which will cause the number of times each such logical unit is executed to be recorded during execution. This feature is expected to be a valuable aid in optimizing PL/I source programs. If USAGE reveals that a certain segment is executed with extremely high frequency, then attention may be turned to reducing the number of occasions such a region is executed and efforts to optimize the coding can be concentrated in that region. Logical oversights which result in a certain section of coding being unreachable or unnecessary will be revealed by a zero usage. USAGE statistics gathered during execution of an interactive program will gather information about that program's utilization which should aid immensely in the design of a more efficient version. Specifically, USAGE causes additional coding in a PL/I source deck as follows:

(1) After the statement after each label, "UV(n) = UV(n) + 1;" is inserted.
(2) AFTER each DO, BEGIN, PROC, or PROCEDURE, the same statement is inserted. (In case both (1) and (2) are true, only a single insertion is made.)
(3) Each simple then clause or else clause is surrounded by "DO; UV(n) = UV(n) + 1;" and "END;".

The value of the integer n is augmented by 1 before each insertion of the UV ... statement. An initial value of zero is used for n, unless some other value is set by the parameter USAGEINDEX = n.

At the end of a NEATER2 run in which USAGE was enabled, or when it is signalled, NEATER2 puts out a declare statement which properly dimensions and initializes the usage vector (UV), and a put statement which will cause the assumulated usages to be printed. To make the USAGE option maximally useful, PL/I source programs should be so structured that they exit through their end statement. In this event NEATER2's insertion of the declare and put statements just prior to the end statement for the procedure (after any label on the END statement) is accomplished by use of the parameter string /*NEATERPARMS:NOUSAGE, USAGEINDEX =0*/. In this case, the output from NEATER2 is ready to compile. If no such closing parameters are fed, the declare and put statements for UV are generated after the logical end of the program. It will be necessary to manually move them into the correct place in the neatened deck.

(g) PARM|NOPARM. The PARM|NOPARM option

controls whether or not NEATER2 reproduces in its output stream the parameter strings which control it. If PARM is on, NEATER2 reproduces its parameter strings. This feature was incorporated so that the combination PARM,PUNCH would be a useful one. With this combination, the deck which NEATER2 produces will contain the same parameter strings as the deck it was produced from. This means that a deck can be kept in active development over a period of time with several reformattings by NEATER 2 without any further attention being paid to maintenance of the correct parameter set for any special requirements of that deck. The general control PARM can be countermanded on an individual parameter string basis by putting an asterisk in column 1 in front of /* NEATERPARMS: ... */. Neater parameters prefaced in this way are not reproduced in the output stream, even if PARM is on.

(h) DEFAULT. It is not required to give NEATER2 parameters, as it will assume a default parameter set if none are given. If an initial parameter string is given, then any parameters not mentioned in the given string are given default values. If, after a period of processing by NEATER2, it is desired to delete an accumulation of parameters in one command, the parameter DEFAULT is given. DEFAULT implies PRINT, LOGICAL, NOPUNCH, NOCOMMENT, NODECLARE, NOUSAGE, NOPARM, SEQNO=0, INSET=3, LEFTMARGININ =2, RIGHTMARGININ=72, LEFTMARGINOUT=2, RIGHTMARGINOUT=72, STATEMENTMARGIN= 10, and USAGEINDEX=0.

(i) RESET. The keyword RESET causes the logical level and the sequence number to be reset to zero, and a new page to be started. This parameter is useful if a number of procedures are being batched through NEATER2, and if it is desired that each one gives the appearance of having been processed separately.

(j) PAGE. The parameter PAGE causes the printed output to skip to the top of the next page. It should be used instead of RESET if pagination is desired in the middle of a procedure or other block, because it does not alter the logical level or the sequence numbering.

(k) SEQNO. The parameter SEQNO=n permits the initialization to any desired positive integral value of the sequence number (n + 1) which NEATER2 assigns to its initial record of output. This feature is useful if only a portion of a procedure is to be reneatened and if it is desired to have the sequence numbers run through the whole program. To achieve the correct logical history in such a usage, the required preceding labeled do statements, unlabeled do statements, and if statements may be inserted at the beginning of the fragment.

(l) INSET. The parameter INSET=n is one of the parameters available to control the formatting which NEATER2 effects. INSET sets the number of spaces which NEATER2 indents for each logical level. INSETs from 2 to 5 are most satisfactory, but for special purposes an INSET of 0 finds use.

(m) STATEMENTMARGIN. The parameter STATEMENTMARGIN=n is used to control another aspect of the formatting which NEATER2 effects. It sets the column in which the first character of a statement at logical level 1 will appear. In the event that INSET is zero, STATEMENTMARGIN sets the column in which all statements begin.

(n) LEFTMARGININ, RIGHTMARGININ, LEFTMARGINOUT, RIGHTMARGINOUT. Adjustment of input and output margins in NEATER2 has the following advantages. Completely freeform punching of the initial program deck is enabled (except, of course, /* in columns 1 and 2). Such a deck fed to NEATER2 can become a conventionally margined source deck with one statement to a card in a single economical pass, thus saving on keypunching time. Adjustment of the output margins permits utilization of the full width of the output medium: columns 2–80 in punched output and columns 1–132 in printed output. This is principally useful in minimizing the cluttering effect that statements too long to fit in a single output line tends to have on neatened output.

(o) USAGEINDEX. The parameter USAGEINDEX =n may be used to set the value (n + 1) in the first statement of the type "UV(n) = UV(n) + 1;" inserted when USAGE is on. This is useful if several different procedures are being NEATENED separately for eventual combination into a run for gathering usage data. Similarly, if only a portion of a program is being reneatened with USAGE on, and the usage vector indices are to be continuous, this parameter may be used.

Diagnostics Produced

The principal expression of the logical diagnosis which NEATER2 effects is the formatting of the output in the illustrated indentation pattern. Logical errors are discovered by the user when the indentation of the output fails to correspond with his intention. It cannot be overemphasized that NEATER2 output must be scanned for unexpected indentation patterns before it can be assumed that a program is logically correct. Most kinds of logic errors will be expressed as an unexpected indentation pattern and will not be expressed in an error message or a condition code greater than 0.

In those cases in which NEATER2 does produce a diagnostic message (and a nonzero condition code), the message appears in one of the following forms.

```
***ERROR*** AT SEQNO mmmm  CODE
***ERROR*** AT SEQNO mmmm  CODE          mmmmm
***ERROR*** AT SEQNO mmmm  CODE          mmmmm  nnnnn
***ERROR*** AT SEQNO mmmm  CODE  aaaaaaaaaaaaaa
```

The sequence number (±1) of the statement in which the diagnostic arose is printed to aid in the location of the difficulty. The code which follows is a mnemonic which indicates the nature of the error. Of the 40 or so different diagnostic messages which may be produced by NEAT-

ER2, the majority are concerned with consistency in the NEATERPARMS. If NEATER2 has changed the requested formatting because of inconsistencies, the changed format settings are indicated in the numeric output after the error code. If a parameter is not recognized, the unrecognized portion of the parameter string is reproduced to assist the user in making his correction. About 10 distinct error messages are produced which serve to pinpoint logic errors in the source deck.

Performance Notes

We have already remarked that NEATER2 processes a PL/I source deck at 3 to 6 times the speed that the compiler does. This performance was attained by careful choice of PL/I source statements so as to maximize the in-line execution of the compiled program and then to minimize the size and execution time of the compiled program. The source program is long (ca. 1500 statements), but the compiled coding from each statement is typically short as a result of constant revision to optimally tune the program to version 5 of the PL/I compiler. The program gives best results with REORDER specified, and with compilation with OPT=02. The program CSECT is about 22K bytes; the load module about 39K bytes, and the storage area about 18K bytes; so the whole of NEATER2 should run easily in a 60K byte partition.

Most modes of operation of NEATER2 make relatively minor changes in the speed at which it (or, subsequently, the compiler) processes a source deck; so they may be used freely at very little expense. The compiler does process a compressed source deck slightly more rapidly than it processes a logically formatted deck. When USAGE is on, an extensive apparatus to gather usage information is constructed, and marked performance degradation is observed. NEATER2 is slowed about 25 percent, the compiler about 20 percent, and execution about 40 percent when this option is employed.

RECEIVED JUNE, 1970

The Linear Quotient Hash Code

JAMES R. BELL AND CHARLES H. KAMAN
Digital Equipment Corporation, Maynard, Massachusetts

A new method of hash coding is presented and is shown to possess desirable attributes. Specifically, the algorithm is simple, efficient, and exhaustive, while needing little time per probe and using few probes per lookup. Performance data and implementation hints are also given.

KEY WORDS AND PHRASES: hashing, hash code, scatter storage, calculated address, search, table, lookup, symbol table, keys
CR CATEGORIES: 3.74, 4.9

1. Introduction

Hash coding, also known as address calculation or storage scattering, is a way to drastically reduce the time spent by a program in searching tables. Several hash codes are proposed or reviewed in the literature [1, 2, 3].

In this paper we first describe the desired attributes of a hash code, next propose a new hash code to fit these specifications, and finally present data on the performance of this new code.

2. Desired Attributes of a Hash Code

By the term hash code we mean an algorithm which associates with each key an address in a given table. If a match for the key is already in the table, the result should be the index of the match. If not, the result should be the index of some empty location in the table.

In general the hash algorithm makes a sequence of probes into the table. The sequence terminates when a probe selects either the match for the key or an empty location. Alternatively, at some point the sequence may recycle and the table is declared full.

We may list the desired attributes of a hash code as follows:

(1) The algorithm should be simple. This leads to a short, quickly written, easily understood program.

(2) The algorithm should generate an efficient, exhaustive sequence: Every location should be probed exactly once before the table is declared full.

(3) The time per probe should be minimal.

(4) The average number of probes per lookup should be minimal.

It is this set of specifications which will serve as our goal.

3. The Linear Quotient Method

We shall first define the terms necessary for the algorithm. Let n represent a prime number chosen as the length of the table. Let K represent the key sought. Let Q and R represent the quotient and remainder created by the division of K by n. Let h_i be the address of the ith probe into the table, i.e. its index in the table.

We shall now define the linear quotient hash code to