Ferstl, O. K.; Sinz, E. J.

Designing Structured Cobol Programs

Designing Structured Cobol Programs

O. K. FERSTL AND E. J. SINZ
Universitaet Regensburg, Universitaetsstrasse 31, D–8400 Regensburg

SUMMARY

COBOL is now more than 20 years old and will probably survive to become much older. Since it has some features which are out of date it is desirable to adapt at least the program style to some standards of modern programming languages. The adaption is not only a matter of style but also of costs of program production and program maintenance. This paper presents constructional rules for programming in COBOL which by-pass some of the drawbacks and allow more readable and more maintainable program structures. Finally a postprocessor is presented, that allows verification of the chosen constructional rules and documentation of the resulting programs.

KEY WORDS COBOL Constructional rules Program structures Postprocessor Control structures Documentation Structured programming Abstract types

INTRODUCTION

The programming language COBOL will probably continue to exist for a long time despite all its critics. One reason is the need to maintain existing program systems which are written in COBOL.

There are various proposals to improve COBOL program structures and to adapt them to include those program structures which have been made possible by the use of modern programming languages. An example is the detailed proposal given by Ledgard and Cave.[1]

Most proposals define a subset of COBOL together with constructional rules directed to the intended goal. In most cases the GO TO statement is one of the forbidden language elements. This restriction leads to program structures which are hard to survey. It seems to us that there is no reason to forbid this instruction, because good structured COBOL programming is a matter of discipline anyway.

Allowing the GO TO instruction avoids scattering of connected program parts over the program text. This happens for example if the PERFORM instruction is the only statement to control repetitions.

The proposals presented here are aimed at writing COBOL programs with the following features:

1. The only control structures should be concatenation, selection and repetition. These structures appear in several variants.
2. The process of program development by stepwise refinement should be documented by the program structure. This results in a refinement tree whose leaf nodes are COBOL statements and whose upper levels describe major parts of the program function.
3. Abstract types are instruments for modular programming. Although COBOL has not been designed for this technique at least the concept of abstract types should be simulated.

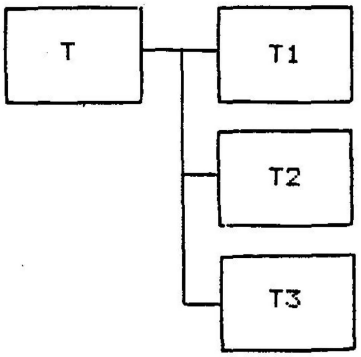
4. It should be possible to use the proposals with COBOL compilers corresponding to the level of COBOL-68 nucleus 2. Module structures according to the constructional principles of abstract types should be possible at least with compilers providing a CALL verb and multi-entries, such as, for instance, IBM COBOL compilers.
5. The resulting COBOL programs should be capable of being processed by a postprocessor that verifies the syntax of the control structures and documents the program by reproducing the refinement tree and separating the comment lines.

BASIC CONCEPTS FOR THE CONSTRUCTIONAL RULES

The following constructional rules are based on a program concept which may be demonstrated by top down flowcharts.[2] A top down flowchart represents a program as a tree with semantic relations between the levels of the tree and sequencing relations between the successors of a node. The flowchart symbols and their meanings may be easily comprehended.

Let T be a task which is divided disjunctively into the subtasks T1, T2, ..., T*n*. The following symbols represent the allowed sequencing structures.

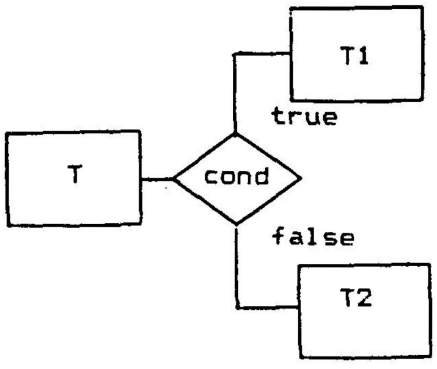
1. Concatenation



Meaning Execute T1, T2, ..., T*n* in the defined sequence.

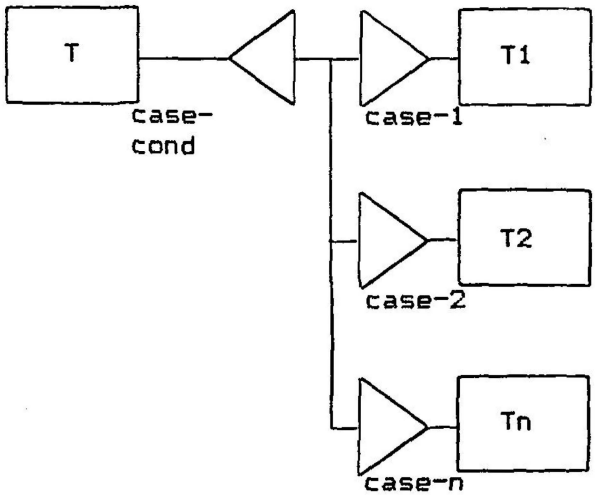
2. Selection

(a) *Selection between two alternatives*



Meaning: T has to be substituted either by T1 or by T2.

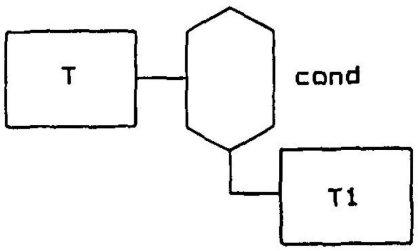
(b) *Selection between n alternatives*



Meaning: T has to be substituted by one of the subtasks T1, T2, ..., T*n*.

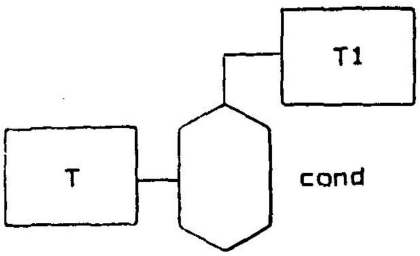
3. Repetition

(a) *Repetition with prechecked condition*



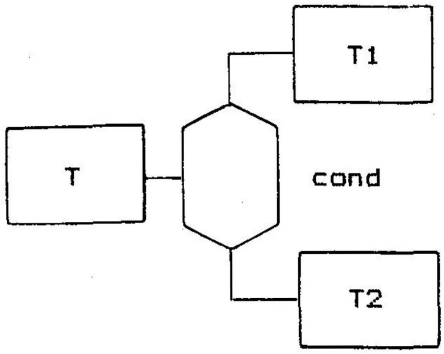
Meaning: T consists of a sequence T1, T1, ..., T1. The length of the sequence is controlled by cond (IF cond = TRUE THEN execute T1). Cond is checked each time before executing T1. The minimal length of the sequence is zero.

(b) *Repetition with postchecked condition*



Meaning: T consists of a sequence T1, T1, ..., T1. The length of the sequence is controlled by cond (execute T1; IF cond = TRUE THEN EXIT). Cond is checked each time after executing T1. The minimal length of the sequence is one.

(c) *Repetition with midchecked condition*



Meaning: T consists of a sequence T1, T2, T1, T2, . . ., T1. The length of the sequence is controlled by cond (execute T1; IF cond = TRUE THEN execute T2). Cond has to be checked each time between the execution of T1 and T2. T1 will be executed once more than T2.

4. Leaving a subtree

Meaning: this node is part of a subtree of T. Stop execution of T and proceed with the task following T.

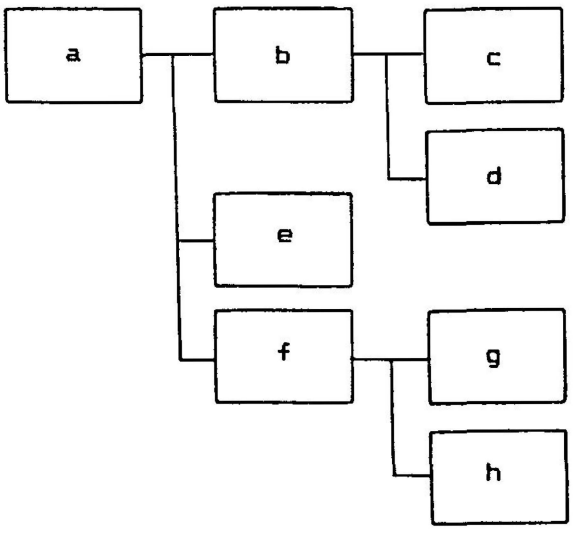
Each of the subtasks T1, T2, . . ., T*n* can be treated recursively like task T. The result is a tree whose root refers to the complete task of a program.

TRANSFERRING TOP DOWN FLOWCHARTS INTO PROGRAMMING LANGUAGES

Transferring a 2-dimensional top down flowchart into a 1-dimensional program text is done by running through the flowchart tree according to a given direction rule. The sequence of nodes passed will be the sequence of the corresponding parts of the program text. Modern programming languages simultaneously support two forms of notation:

1. the block-structured form
2. the list form

Example:



Notation form 1

(a (b (c) (d)) (e) (f (g) (h)))

This is the representation technique of block-structured programming languages like Algol or Pascal. The parentheses (and) are replaced by BEGIN and END. Most programs of this type only contain the leaf nodes of the tree.

Notation form 2

(a)
a: (b, e, f)
b: (c, d)
f: (g, h)

This technique provides a separate representation of each level of the tree. All nodes of the tree are shown. The corresponding instruments in modern programming languages are the procedure and the refinement concept.

TRANSFERRING TOP DOWN FLOWCHARTS INTO COBOL

COBOL has no general block-structuring features to support notation form 1. Hence a top down flowchart has to be transfered into COBOL according to notation form 2.

For each node with a non-empty set of successors this set of successors is composed to a COBOL section. The paragraph construct is reserved exclusively for formulating control structures.

Example

The tree shown above is represented in the following COBOL program.

```
PROCEDURE DIVISION
ROOT SECTION.
paragraphname.
    PERFORM a
    STOP RUN.
a SECTION.
paragraphname.
    PERFORM b
    e
    PERFORM f.
b SECTION.
paragraphname.
    c
    d.
f SECTION.
paragraphname.
    g
    h.
```

The control structures of the top down flowchart are described by the COBOL

components: paragraph, sentence and control statements, which are related as follows*

```
⟨section⟩      ::= ⟨sectionname⟩ SECTION. ⟨paragraph⟩ {⟨paragraph⟩}
⟨paragraph⟩    ::= ⟨paragraphname⟩. ⟨sentence⟩ {⟨sentence⟩}
⟨sentence⟩     ::= ⟨non-conditional statement⟩
                     {⟨non-conditional statement⟩} ⟨dot⟩
                   | {⟨non-conditional statement⟩}
                     ⟨conditional statement⟩ ⟨dot⟩
⟨dot⟩          ::= .
```

This syntax only partially allows block-structuring. Named blocks are sections and paragraphs. Unnamed blocks are sentences or blocks within an IF statement following the THEN clause or the ELSE clause. In the THEN case they are delimited by IF ⟨condition⟩ and ELSE, in the ELSE case by ELSE and ⟨dot⟩. Sections, paragraphs and sentences must not be nested by blocks of the same type.

Notational conventions

It is assumed for the following rules that only COBOL statements without condition clauses are used, for example arithmetic statements without the ON SIZE ERROR clause. Later, the rules will be completed for statements with condition clauses.

Subtasks of a task T are represented by their most general realization in COBOL and can be substituted by a corresponding part of this construct.

Syntax conventions

(a) ⟨n-sequence⟩ ::= ⟨non-conditional COBOL statement⟩
{⟨non-conditional COBOL statement⟩}

(b) ⟨c-sequence⟩ ::= ⟨n-sequence⟩
| {⟨*n-sequence*⟩} ⟨*conditional* COBOL *statement*⟩

(c) ⟨sentence⟩ ::= ⟨c-sequence⟩ ⟨dot⟩

(d) ⟨sentences⟩ ::= ⟨sentence⟩ {⟨sentence⟩}

(e) Task structures which have to be described in more detail are named by upper-case characters.

(f) The words CASE, FOR, REPEAT, UNTIL, WHILE and widenings of these words are reserved.

Constructional rules for COBOL statements without condition clauses

1. *Concatenation of subtasks*

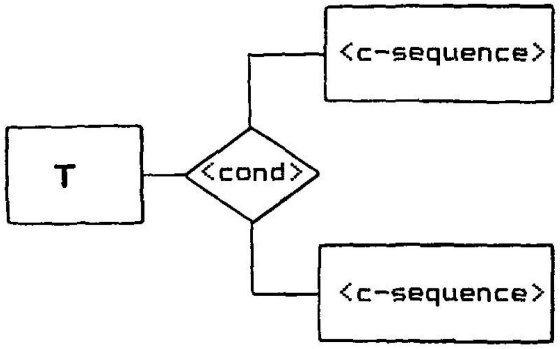
Statements are concatenated to sentences, sentences to paragraphs and paragraphs to sections according to the COBOL syntax shown above.

---

*The notational conventions are described in Backus–Naur form (BNF) as in Reference 3.

2. *Selection of subtasks*

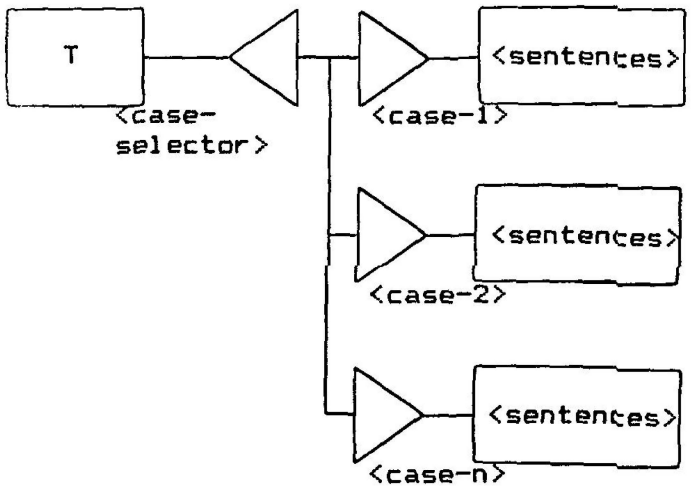
(a) *Selection between two alternatives*



This control structure is translated into

```
T ::= IF ⟨cond⟩
        ⟨c-sequence⟩ | NEXT SENTENCE
      ELSE
        ⟨c-sequence⟩ | NEXT SENTENCE
```

(b) *Selection between n alternatives*



For this control structure COBOL provides two constructions.
Format 1 is part of one sentence:

```
T ::= IF ⟨case-selector⟩ = ⟨case-1⟩
        ⟨c-sequence⟩ | NEXT SENTENCE
      ELSE IF ⟨case-selector⟩ = ⟨case-2⟩
        ⟨c-sequence⟩ | NEXT SENTENCE
      ELSE IF ⟨case-selector⟩ = ⟨case-n⟩
        ⟨c-sequence⟩ | NEXT SENTENCE
```

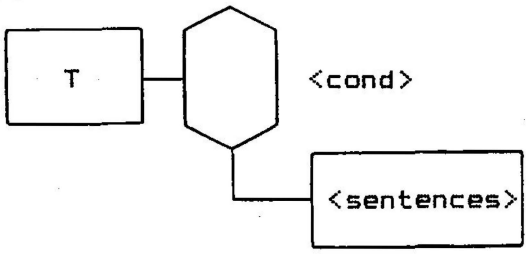
Format 2 includes different paragraphs:

```
T :: = CASE-⟨identifier⟩.
           GO TO ⟨1-identifier⟩ ⟨2-identifier⟩ ... ⟨n-identifier⟩
           DEPENDING ON CASE-⟨selector⟩
           GO TO END-CASE-⟨identifier⟩.
       ⟨1-identifier⟩.
           ⟨sentences⟩
           GO TO END-CASE-⟨identifier⟩.
       ⟨2-identifier⟩.
           ⟨sentences⟩
           GO TO END-CASE-⟨identifier⟩.
       ⟨n-identifier⟩.
           ⟨sentences⟩
           GO TO END-CASE-⟨identifier⟩.
       END-CASE-⟨identifier⟩.
           EXIT.
```

3. *Repetition of subtasks*

Most proposals for structured programming in COBOL prefer the PERFORM UNTIL statement to formulate repetitions, thus removing the body of the loop to a distant position within the program. This method causes program structures which are hard to survey. Another possible effect is repeated long jumps. The following constructions try to avoid removing of the body.

(a) *Prechecked repetition*



This control structure may be translated into one of the following formats.

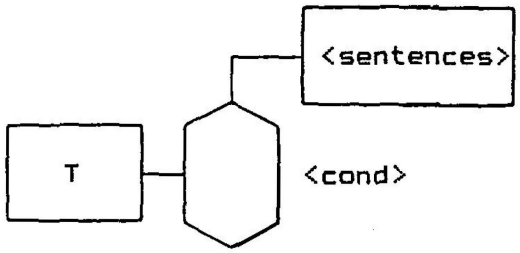
Format 1 needs one paragraph:

```
T :: = WHILE-⟨identifier⟩.
           IF ⟨cond⟩
              ⟨n-sequence⟩
           GO TO WHILE-⟨identifier⟩.
```

Format 2 includes up to four paragraphs. The paragraph ⟨body-end⟩ is optional:

```
T :: = WHILE-⟨identifier⟩.
           PERFORM ⟨body-begin⟩ THRU ⟨body-end⟩ UNTIL NOT ⟨cond⟩
           GO TO END-WHILE-⟨identifier⟩.
       ⟨body-begin⟩.
           ⟨sentences⟩
       ⟨body-end⟩.
           EXIT.
       END-WHILE-⟨identifier⟩.
           EXIT.
```

(b) *Postchecked repetition*



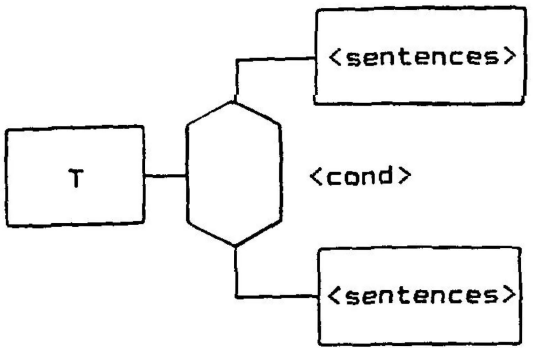
This control structure may be translated into one of the following formats.
Format 1:

```
T :: = UNTIL-⟨identifier⟩.
           ⟨n-sequence⟩
       IF NOT ⟨cond⟩
       GO TO UNTIL-⟨identifier⟩.
```

Format 2:

```
T :: = UNTIL-⟨identifier⟩.
           ⟨sentences⟩
     ⟨body-end⟩.
           EXIT.
     END-UNTIL-⟨identifier⟩.
           PERFORM UNTIL-⟨identifier⟩ THRU ⟨body-end⟩
           UNTIL ⟨cond⟩.
```

(c) *Midchecked repetition*



This control structure may be translated into one of the following formats.
Format 1:

```
T :: = REPEAT-⟨identifier⟩.
           ⟨sentences⟩
       IF ⟨cond⟩
           ⟨n-sequence⟩
     GO TO REPEAT-⟨identifier⟩.
```

Format 2:

```
T ::= REPEAT-⟨identifier⟩.
          ⟨sentences⟩
      IF NOT ⟨cond⟩
      GO TO END-REPEAT-⟨identifier⟩.
          ⟨sentences⟩
      GO TO REPEAT-⟨identifier⟩.
  END-REPEAT-⟨identifier⟩.
      EXIT.
```

(d) *Counted repetition.* A counted repetition is available in programming languages by the FOR-statement. It may be transferred into one of the following COBOL formats. The paragraph ⟨body-end⟩ is optional.

Format 1:

```
T ::= FOR-⟨identifier⟩.
          PERFORM ⟨body-begin⟩ THRU ⟨body-end⟩ ⟨n⟩ TIMES
          GO TO END-FOR-⟨identifier⟩.
      ⟨body-begin⟩.
          ⟨sentences⟩
      ⟨body-end⟩.
          EXIT.
      END-FOR-⟨identifier⟩.
          EXIT.
```

Format 2:

```
T ::= FOR-⟨identifier⟩.
          PERFORM ⟨body-begin⟩ THRU ⟨body-end⟩
          VARYING ⟨count-variable⟩ FROM ⟨lower-bound⟩ BY ⟨increment⟩
          UNTIL ⟨upper-bound-condition⟩
          GO TO END-FOR-⟨identifier⟩.
      ⟨body-begin⟩.
          ⟨sentences⟩
      ⟨body-end⟩.
          EXIT.
      END-FOR-⟨identifier⟩.
          EXIT.
```

4. *Leaving a subtree*

The concept of enclosing the successors of a node within a COBOL section allows one to mark the end of the set of successors by an EXIT paragraph. For leaving a subtree this point has to be reached by a GO TO statement.

The flowchart component 'leaving a subtree' allows one to leave each subtree of which it is part. Therefore the 'leaving' GO TO statement can be directed to the end of each section whose subtree includes this statement.

Constructional rules for COBOL statements with condition clauses

The rules have to be completed for using the statements READ ... AT END, WRITE ... INVALID KEY, SEARCH ... AT END ... WHEN, RETURN ... AT END and arithmetic

statements with the ... ON SIZE ERROR clause. According to the COBOL syntax only non-conditional statements may follow these clauses. There are three situations:

1. The statements following the condition clauses are non-control statements.

They only have local influence on the enclosing control structure and effect thus like the THEN branch of an IF statement. A sequence of statements can be inserted.

Example:

```
ADD A TO B ON SIZE ERROR
    MOVE 1 TO OVERFLOW-FLAG.
```

The sentence has to be closed after the inserted statements. Therefore the enclosing control structure possibly has to be formulated by using format 2 of the constructional rules.

2. One of the statements following the condition clause is a PERFORM section statement.

This statement also has only local influence on the enclosing control structure if the called section is constructed according to the given rules. As in situation 1 format 2 of the rules possibly has to be used.

3. The last statement following the condition clause is a GO TO statement.

This GO TO statement has to be used in the meaning 'leaving a subtree' or as part of a midchecked repetition.

Example:

```
REPEAT-UNTIL-EOF.
        READ FILE
    AT END GO TO END-REPEAT-UNTIL-EOF.
        ⟨n-sequence⟩
    GO TO REPEAT-UNTIL-EOF.
END-REPEAT-UNTIL-EOF.
    EXIT.
```

VERIFYING AND DOCUMENTING OF THE STRUCTURE BY A POSTPROCESSOR

The development of COBOL programs according to the proposed constructional rules can be supported by a postprocessor. The input of the postprocessor is a syntactically correct COBOL program. Its output is the refinement tree showing a top down flowchart for each section of the program.

The postprocessor has two purposes. Purpose one is to check if the constructional rules are strictly obeyed. Thus the postprocessor serves as a static testing tool. Violation of the rules causes error messages. Unintentional constructions are visualized by the refinement tree.

Purpose two is a documentary function. Although the constructional rules provide readable source programs the postprocessor still improves the readability by producing a 2-dimensional documentation of the program with the flowchart symbols shown above. The documentation consists of a list of subtrees each representing a COBOL section. Refinement structures using the PERFORM statement are marked by '-⟩'. Procedure calls are marked by '= ⟩'. Thus the flowcharting technique provides a single description language for design and documentation.

COBOL programs possibly need comments. Comments introduce additional information to a program or explain COBOL statements by another language. Unfortunately comments require separate lines and force when reading the program a continuous change between the two languages COBOL and the chosen comment language. Therefore the postprocessor separates the comment lines. If P is a program line and C a comment line which refers to the preceding program line, then a sequence

P P C P C C

is transferred to the following parallel presentation in the refinement tree:

```
  P
C P
C P
C
```

Finally the postprocessor saves costs and time for documentation and helps to get trouble-free and up-to-date documents.

Compared to preprocessors, which replace the COBOL vocabulary by the control statements of modern languages, a postprocessor has some advantages:

1. A postprocessor only has to run after changing control structures whereas a preprocessor run is needed whenever the program is compiled.
2. A postprocessor is easier to implement because its input is a syntactical correct program. The analysis is restricted to the additional language level defined by the constructional rules.

The postprocessor is written in Pascal and makes extensive use of recursive procedures. It has been implemented on a microcomputer system.

ABSTRACT TYPES IN COBOL

COBOL provides the following data types:

1. Primitive datatypes
   (a) numbers in different code representations
   (b) character strings
2. Composed data types
   (a) records
   (b) tables.

All instructions of COBOL refer to these data types. Beyond that COBOL provides no instruments for handling abstract data types. Since abstract types have proved to be a good concept for modular programming it is desirable to simulate this technique as far as possible. The following method is simple and easy to implement:

Each abstract data object is represented by the form

01 ⟨abstract-data-object⟩ PIC X(⟨length-of-the-object⟩)

and has to be detailed by a record or a table within the LINKAGE SECTION of a program containing operators for this type. A weakness is that the length of the object is dependent on factors like machine type and compiler.

All abstract operators referring to a certain abstract data type are formulated by COBOL subprograms and combined to one COBOL program representing a module.

Outside of a module only the length of an abstract data object and its operators are known. Operators for data objects are usually:

(i) fetching and storing of values of the object
(ii) input/output to external devices
(iii) selection of components of the objects
(iv) application dependent processing of the object.

Instructions already provided by COBOL for this concept are:

(i) the MOVE instruction which permits fetching and storing
(ii) READ/WRITE instructions for standard input/output
(iii) the OF clause and subscripts or indices for the selection of components.

EXAMPLE

The rules may be demonstrated by a simple program for processing data objects of the type DATE. After reading a date and a time distance in days a new date with the given distance should be computed.

The top down flowchart shown in Figure 1 is implemented by two COBOL

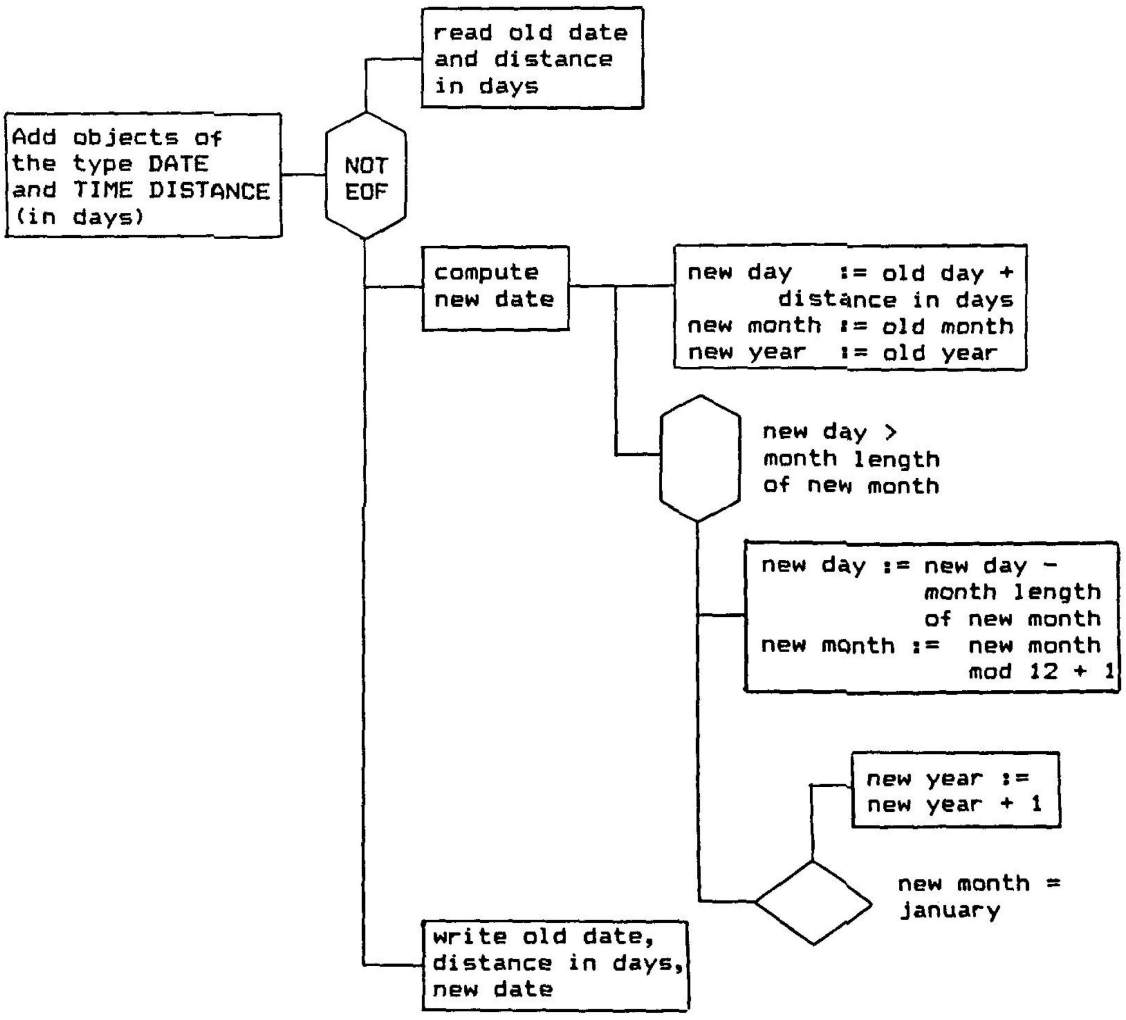


Figure 1. Flowchart for example

programs. Program NEWDATE (Figure 2) uses the abstract type DATE which is implemented by the program DATEOPERATORS (Figure 3).

The postprocessor outputs the charts shown in Figures 4–10.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.     NEWDATE.
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. SIEMENS-7531.
 OBJECT-COMPUTER. SIEMENS-7531.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 EOF-INDICATOR          PIC X.
    88 EOF                 VALUE HIGH-VALUE.
 01 OLD-DATE               PIC X(6).
 01 NEW-DATE               PIC X(6).
 01 DISTANCE-IN-DAYS       PIC 9(5).
*
*
 PROCEDURE DIVISION.
 MAIN SECTION.
*                          this section is the root
*                          of the task
 PAR.
     PERFORM ADD-DATE-DAYS
*                          operator for adding days
*                          to a given date
     STOP RUN.
*
*
 ADD-DATE-DAYS SECTION.
 REPEAT-UNTIL-EOF.
*                          this is version C of repetition
         CALL "READDATE" USING OLD-DATE EOF-INDICATOR
*                          read a correct date or leave
         IF (NOT EOF)
*                          if input is not numeric
*                          EOF-INDICATOR is true (HIGH-VALUE)
            DISPLAY "INPUT DISTANCE IN DAYS:"
            ACCEPT DISTANCE-IN-DAYS
*                          read the distance or leave
            IF (DISTANCE-IN-DAYS NOT NUMERIC)
               MOVE HIGH-VALUE TO EOF-INDICATOR.
      IF (NOT EOF)
         CALL "ADDDATEDAYS" USING NEW-DATE OLD-DATE
                                  DISTANCE-IN-DAYS
*                          new date = old date + distance
         DISPLAY "  OLD DATE:"
         CALL "WRITEDATE" USING OLD-DATE
         DISPLAY "+ DISTANCE:"
         DISPLAY "  " DISTANCE-IN-DAYS
         DISPLAY "= NEW DATE:"
         CALL "WRITEDATE" USING NEW-DATE
*                           write new date
      GO TO REPEAT-UNTIL-EOF.
```

Figure 2. Newdate

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.    DATEOPERATORS.
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. SIEMENS-7531.
 OBJECT-COMPUTER. SIEMENS-7531.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 ERROR-INDICATOR        PIC X.
    88 INPUT-ERROR         VALUE HIGH-VALUE.
 01 DAYS                   PIC 9(6) COMP-3.
 01 MONTH-LENGTH.
    02 LENGTH-INIT         PIC X(24)
       VALUE "312831303130313130313031".
    02 MONTH-LENGTH REDEFINES LENGTH-INIT OCCURS 12 PIC 9(2).
*
 LINKAGE SECTION.
 01 EOF-INDICATOR          PIC X.
    88 EOF                 VALUE HIGH-VALUE.
 01 DATE-1.
    02 MONTH               PIC 9(2).
    02 DAY                 PIC 9(2).
    02 YEAR                PIC 9(2).
 01 DATE-2.
    02 MONTH               PIC 9(2).
    02 DAY                 PIC 9(2).
    02 YEAR                PIC 9(2).
 01 DISTANCE-IN-DAYS       PIC 9(5).
*
*
 PROCEDURE DIVISION.
 READ-DATE SECTION.
*                          read correct date or leave
 ENTRY-PAR.
     ENTRY "READDATE" USING DATE-1 EOF-INDICATOR.
*
 UNTIL-INPUT-OK.
*                          this is version B of repetition
     DISPLAY "INPUT DATE (MMDDYY):"
     ACCEPT DATE-1
*                          input date from console;
*                          if input is not numeric
*                          EOF-INDICATOR is true (EOF);
*                          if date is not a correct date
*                          ERROR-INDICATOR is true (INPUT-ERROR)
     IF  (MONTH OF DATE-1 NOT NUMERIC
        OR  DAY OF DATE-1 NOT NUMERIC
        OR YEAR OF DATE-1 NOT NUMERIC)
        MOVE HIGH-VALUE TO EOF-INDICATOR
     ELSE
        MOVE LOW-VALUE TO EOF-INDICATOR
        IF (MONTH OF DATE-1 < 1 OR > 12)
           MOVE HIGH-VALUE TO ERROR-INDICATOR
        ELSE
           IF (DAY OF DATE-1 < 1
              OR > MONTH-LENGTH (MONTH OF DATE-1))
              MOVE HIGH-VALUE TO ERROR-INDICATOR
           ELSE
              MOVE LOW-VALUE TO ERROR-INDICATOR.
     IF (NOT EOF AND INPUT-ERROR)
        DISPLAY "INPUT ERROR".
 END-UNTIL-INPUT-OK.
     PERFORM UNTIL-INPUT-OK UNTIL (EOF OR NOT INPUT-ERROR).
*
 END-READ-DATE.
     EXIT PROGRAM.
*
```

Figure 3. Dateoperators

```
*
 WRITE-DATE SECTION.
*                              output date to console

 ENTRY-PAR.
      ENTRY "WRITEDATE" USING DATE-1.
      DISPLAY "   " MONTH OF DATE-1 "/" DAY OF DATE-1 "/"
              YEAR OF DATE-1.
      EXIT PROGRAM.
*
*
 ADD-DATE-DAYS SECTION.
*                             date-2 = date-1 + distance-in-days
*                             (no regard to leap-years)
 ENTRY-PAR.
      ENTRY "ADDDATEDAYS" USING DATE-2 DATE-1 DISTANCE-IN-DAYS.
      MOVE MONTH OF DATE-1 TO MONTH OF DATE-2
      COMPUTE DAYS = DAY OF DATE-1 + DISTANCE-IN-DAYS
      MOVE YEAR OF DATE-1 TO YEAR OF DATE-2.
*
 WHILE-DAYS-GT-MONTH-LENGTH.
*                             this is version A of repetition
      PERFORM NEXT-MONTH
              UNTIL (DAYS NOT > MONTH-LENGTH (MONTH OF DATE-2))
      GO TO END-WHILE-DAYS-GT-MONTH-LENGTH.
 NEXT-MONTH.
      SUBTRACT MONTH-LENGTH (MONTH OF DATE-2) FROM DAYS
      ADD 1 TO MONTH OF DATE-2
      IF (MONTH OF DATE-2 > 12)
         MOVE 1 TO MONTH OF DATE-2
         ADD 1 TO YEAR OF DATE-2.
 END-WHILE-DAYS-GT-MONTH-LENGTH.
      EXIT.
*
 END-ADD-DATE-DAYS.
      MOVE DAYS TO DAY OF DATE-2.
      EXIT PROGRAM.
```

Figure 3 (continued)

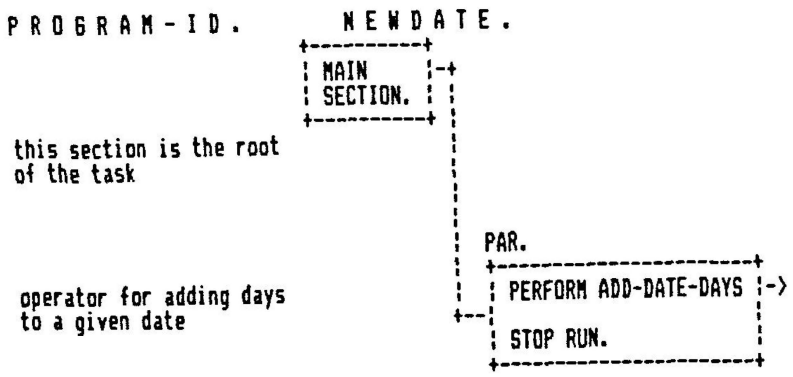


Figure 4. Newdate: chart No. 1

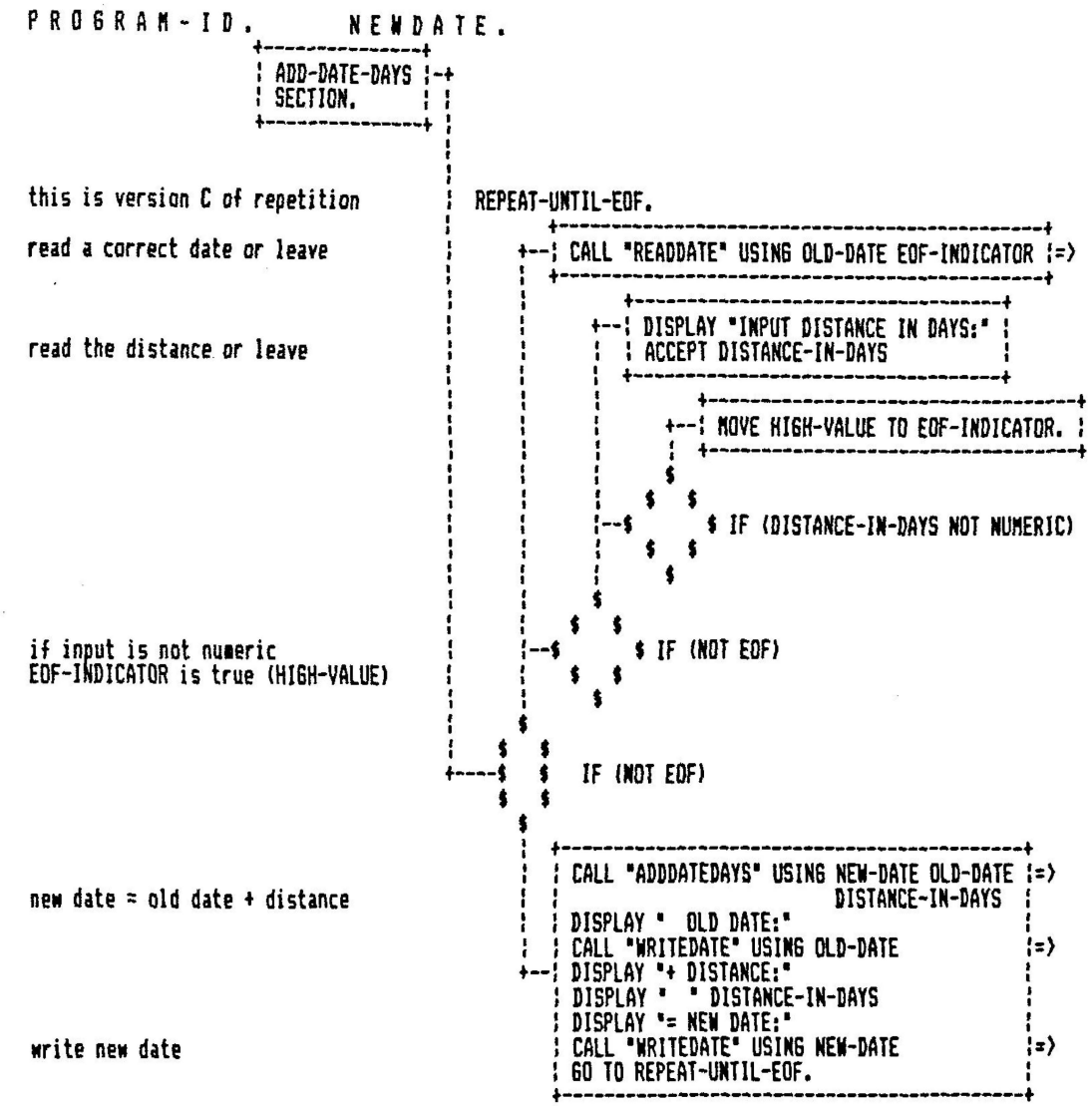


Figure 5. Newdate: chart No. 2

PROGRAM-ID. NEWDATE.

Ascending

Chart No.	Section
01	MAIN
02	ADD-DATE-DAYS

Alphabetic

Section	Chart No.
ADD-DATE-DAYS	02
MAIN	01

Figure 6. Newdate: contents

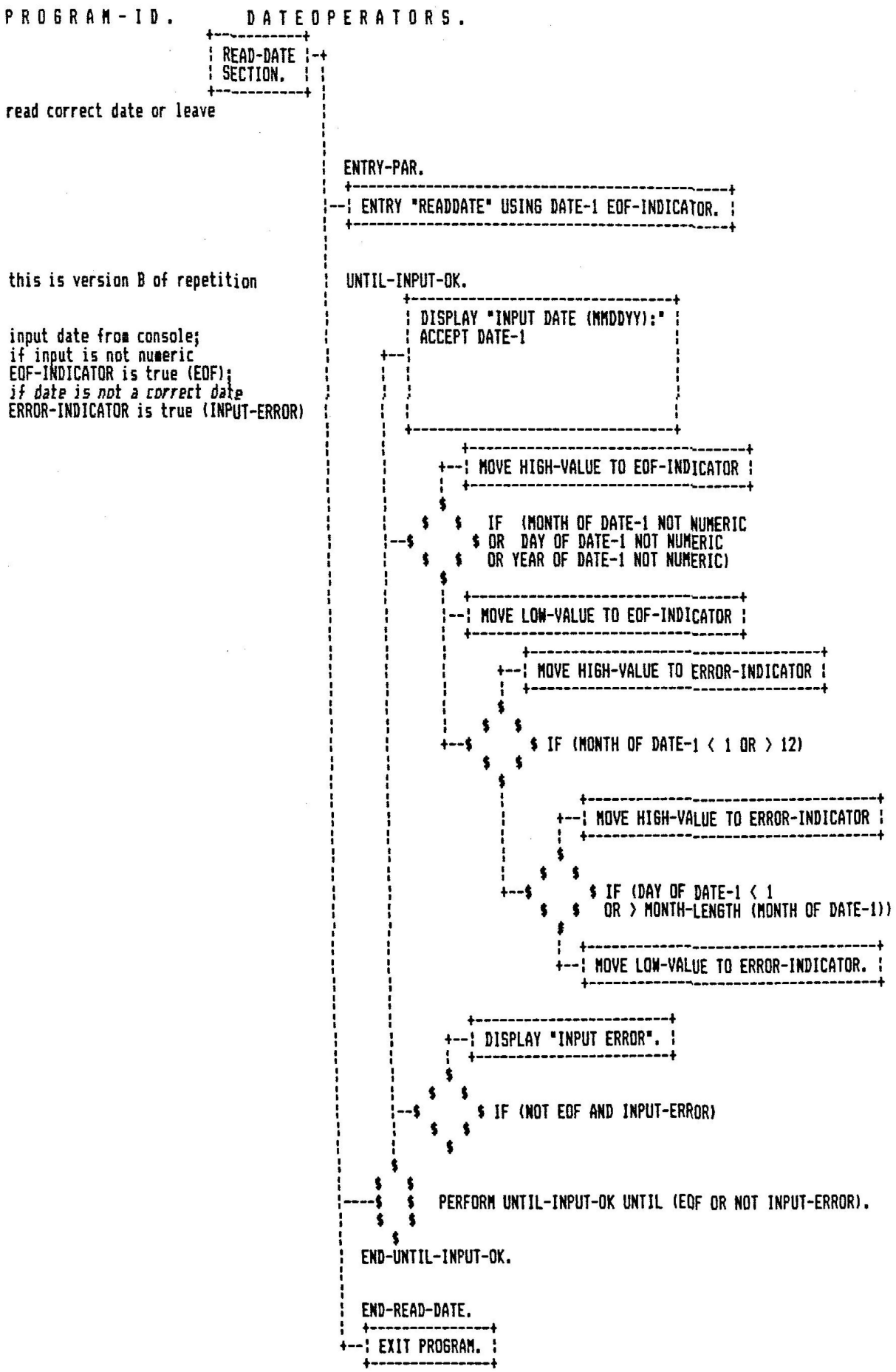


Figure 7. Dateoperators: chart No. 1

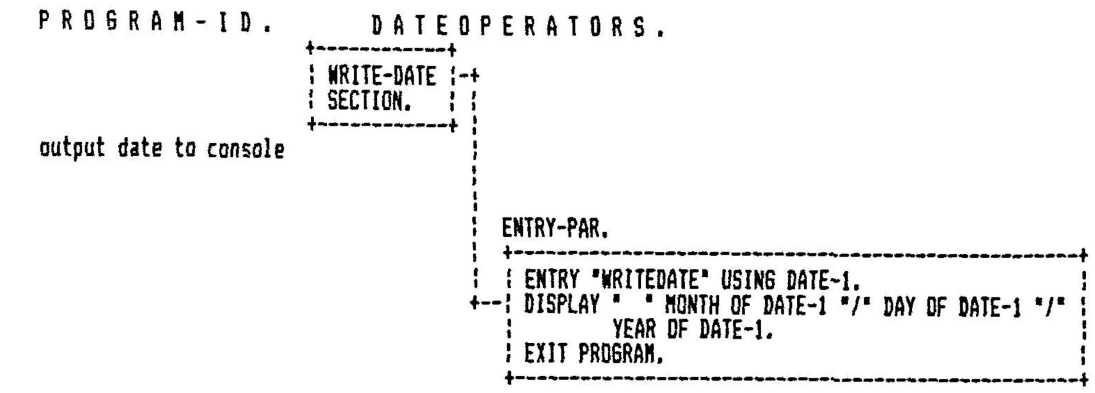


Figure 8. Dateoperators: chart No. 2

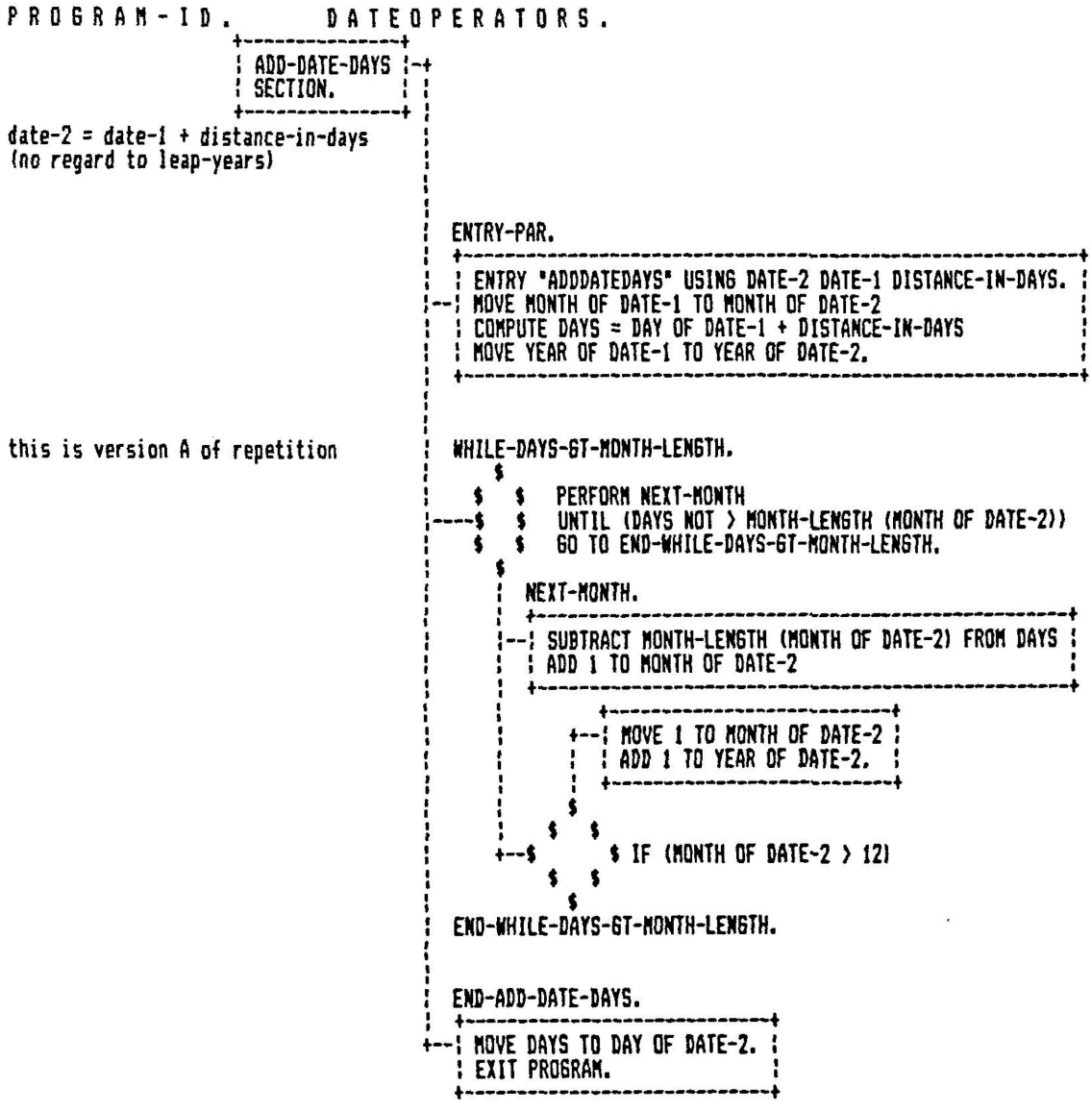


Figure 9. Dateoperators: chart No. 3

```
PROGRAM-ID.     DATEOPERATORS.
```

```
Ascending                          Alphabetic

Chart No.                Section   Section                Chart No.
+-------------------------------+  +-------------------------------+
| 01  READ-DATE                 |  | ADD-DATE-DAYS              03 |
| 02  WRITE-DATE                |  | READ-DATE                  01 |
| 03  ADD-DATE-DAYS             |  | WRITE-DATE                 02 |
+-------------------------------+  +-------------------------------+
```

Figure 10. Dateoperators: contents

CONCLUSIONS

We see the following advantages for COBOL programs written according to the described constructional rules:

(a) The programs are easier to read and to understand. This is especially necessary if several persons are concerned with the program development.

(b) It is easier to maintain the programs because the components of a control structure are put together. Thus it is no problem to substitute a control structure by another one.

(c) The constructional rules possibly avoid long jumps caused by a PERFORM UNTIL instruction which is far away from the body of the loop.

(d) The programs may easily be superseded by programs written in modern programming languages.

(e) The programs allow application of a postprocessor to verify the constructional rules and document the source program. Since this can be done after every change there is no actuality lag between the source program and its documentation.

REFERENCES

1. H. F. Ledgard and W. C. Cave, 'Cobol under control', *CACM*, **19** (11), 601–608 (1976).
2. O. Ferstl, 'Flowcharting by stepwise refinement', *SIGPLAN Notices*, **13** (1), 34–42 (1978).
3. K. Jensen and N. Wirth, *PASCAL, User Manual and Report*, 2nd Edition, Springer Verlag, New York, 1978.
4. L. J. Chmura and H. F. Ledgard, *COBOL with Style*, Hayden Publ. Co., Rochelle Park N.J., 1976.
5. Ch. Floyd, *Strukturierte Programierung—Fuer COBOL—Anwender*, Hoffmann und Campe Verlag, Hamburg 1974.
6. J. F. Gimpel, 'CONTOUR—A method of preparing structured flowcharts', *SIGPLAN Notices*, **15** (10), 35–41 (1980).
7. M. Jackel, 'A formatting parser for PASCAL programs', *SIGPLAN Notices*, **15** (7 & 8), 58–63 (1980).
8. B. Liskov, 'Programming with abstract data types', *SIGPLAN Notices*, **9** (4) 50–59 (1974).
9. C. L. McClure, 'Structured programming in COBOL', *SIGPLAN Notices*, **10** (4), 25–33 (1975).
10. D. D. McCracken, *A Simplified Guide to Structured COBOL Programming*, Wiley, New York, 1976.
11. P. Roy, 'Linear flowchart generator for a structured language', *SIGPLAN Notices*, **11** (11) 58–64 (1976).
12. P. Schnupp, '1st COBOL unsterblich?' in K. Alber, (Hrsg.): Programmiersprachen. 5. Fachtagung der GI, Braunschweig 1978, Springer Verlag, Berlin 1978.
13. H. P. Stevenson, (ed.): *Proc. Symp. Structured Programming in COBOL*, ACM, New York, 1975.