PROGRAMMING AND COMPUTER SOFTWARE

A translation of *Programmirovanie*

May, 1984

Volume 9, Number 4

July-August, 1983

CONTENTS

The Russian press date (podpisano k pechati) of this issue was 7/9/1983. Publication therefore did not occur prior to this date, but must be assumed to have taken place reasonably soon thereafter.

SYNTAX-DIRECTED PROGRAMMING OF DATA INPUT AND CHECKING

N. N. Bezrukov

UDC 51:681.3.06

The article considers the design of data input and checking programs (DICP) based on the single-pass compiler scheme. A number of standard DICP blocks coded in PL/1 are described. The method supports automatic correction of errors of certain types, shorthand coding of repeating groups of attributes, and availability of comments in the input data. It is especially useful for the design of interactive DICP's.

The programming of data input and checking is the most labor-consuming part in data processing jobs. A data input program generator (DIPG) [1] is intended to simplify this task. However, the DIPG is designed for fairly simple (hierarchic or homogeneous) data structures, restricts the output format, and provides insufficiently informative diagnostic messages in case of errors in the input data. Since the DIPG generates an Assembler input program, while the data processing programs are usually coded in a high-level language, a programmer attempting to use the DIPG must have proficiency in two languages. At the same time, the programming of data input and checking in a high-level language is a sufficiently complicated problem in itself.

The design of data input and checking programs (DICP) is usually considered as an area unrelated to the design of programming language compilers. Yet both subject areas have much in common. The source data may be considered as a program written in some formal language. The syntax of this language may be described using an appropriate syntactic metalanguage (e.g., BNF). Therefore, a data input and checking program may be treated as a simple compiler, and thus created using all the algorithms and methods available in the highly developed area of compiler design [2].

In what follows we consider DICP design following the scheme traditionally used for single-pass compilers: a parser drives a lexic analyzer and a block which performs semantic checking and generates the output file. The proposed method will generate fairly quickly and easily a DICP in a high-level language, capable of processing not only hierarchic and homogeneous data structures, but also more complex structures and providing detailed error diagnostics (much more informative than those in DIPG). The output file may have arbitrary structure. For brevity, we will designate the DICP designed by this method a syntax-directed DICP (SDDICP).

Advisability of Using Lexic Data Types

The most common carrier of data until quite recently was punched paper tape. Data were punched on paper tape with explicit field and record separators, as follows:

```
PUSHKIN A.S.:RUSSIAN:1799:1837 =
LERMONTOV M.Yu.:RUSSIAN:1814:1841 =
```

In this format, the data are without lexic types, or more precisely all the data are of the same lexic type. Therefore this format prevents the use of syntax-directed data input and checking methods, in the same way that the structure of Fortran prevents the use of syntax-directed translation methods.

The recent tendency toward increasing use of interactive data entry methods provides an opportunity for a change of format. The separators displayed on the terminal screen only increase the intrinsically heavy visual load on the operators (who anyhow complain of eye fatigue) and make it more difficult to manipulate character insert and delete instructions (as no blanks are allowed in the middle of data, all the following symbols must be shifted

Translated from Programmirovanie, No. 4, pp. 38-50, July-August, 1983. Original article submitted January 14, 1982.

0361-7688/83/0904-0185$07.50

right or left). It is therefore advisable to use blanks as separators and to enter the data in free-field format, i.e., allowing the data elements to be separated by an arbitrary number of blanks. The free-field format essentially simplifies data correction and significantly improves readability.

The use of blanks as separators necessitates the introduction of lexic data types (identifiers, characters, numbers, separators, etc.), as in all programming languages. The benefits of defining distinct lexic data types were already recognized by the creators of PL/1: list directed and data directed input assume certain lexic types such as identifiers (variable names in data directed input), characters (values of string variables), and numbers. Unfortunately, the control of such input types (by means of CONVERSION conditions) is insufficiently flexible. Nevertheless, these two types of data organization are very convenient and are widely used by PL/1 programmers.

Lexic Analysis of Data

The key aspect of the proposed method is the separation of a lexic level of data and the replacement of direct data input by input with a special lexic analysis subprogram (a scanner). The separation of the lexic data level is conceptually similar to the separation of the lexic level in programming languages: the problem is divided into two conceptual levels, and each can be analyzed and programmed independently.

During a single access, the scanner isolates a single lexic unit (a lexeme) in the source text and enters the corresponding code (so-called type) in a syntax stack, while the text of the lexeme is entered in a parallel semantic stack. Five lexeme types are usually sufficient for source data processing: these are integers (type 9), real numbers (type R), literals or character strings of the form '{*symbol*}' (type X), identifiers (type A), and separators (+, −, =, etc.) for which the type coincides with the particular separator symbol. There should also be a class of symbols ignored by the lexic analyzer (such as the blank and the comma).

Thus, in lexic analysis of the record

'PUSHKIN A.S.', 'EUGENE ONEGIN', BVL, 75, 2.34; the first call to the scanner will push X into the syntax stack and PUSHKIN A.S. into the semantic stack, while the second call will push the pair (X, EUGENE ONEGIN). This will be followed in succession by the pairs (A, BVL), (9, 75), (R, 2.34) and, finally (; , ;).

Note that after the input of this line, the syntax stack will contain the string XXA9R;, which for homogeneous records may be treated as a *syntactic skeleton* of the record (the input of any record of the same structure will generate in the stack a *syntactic template* identical to the template).

The lexic analysis algorithms presented in [2-7] are virtually independent of the computer instruction set. At the same time, the specific features of the instruction set of ES computers (which support the instructions TR and TRT) make it possible to implement simple and efficient lexis analysis algorithms. The algorithm described below simplifies vocabulary modification, and the lexis analyzers based on this algorithms are easy to debug. The algorithm has been in use for several years now and the accumulated experience recommends it as one of the most convenient methods of lexic analysis for PL/1 programming.

The algorithm used by any lexic analyzer should be able to classify the literals into different classes, such as digits, letters, separators, ignored symbols, etc. The classical method uses for this purpose the so-called *classifier vector* [2, 6]. In this method, the bit combination corresponding to each literal is treated as a binary number, and this number constitutes an index under which the classifier stores the information needed to assign the given literal to a particular class (or classes). There are two basic methods of encoding this information [2, 3].

The first method encodes each class by a bit string, which in the simplest case contains ones in places corresponding to the classes which include the given literal [3]. Then a bit mask can be constructed for each class, so that application of logical AND to the classifier element and the mask of the corresponding class performs unique classification. In the simplest case the mask may contain a single one, which is the bit used to encode membership in the given class. For instance, if the class of digits is coded as '10000000'B, and the class of letters as '01000000'B, the check C(B) & '1'B will verify whether the symbol with the code B is a digit; similarly the check C(K) & '01'B will verify whether the symbol with the code K

is a letter; the check C(K) & '11'B will verify membership in the class of alphanumeric characters. This method is highly convenient also for complex lexic structures, i.e., when one character is simultaneously assigned to several classes. Its shortcoming is that the mask cannot be used as an index for jumping to the lexic analyzer fragment that analyzes lexemes of the corresponding type.

The second method encodes each class by an integer. This integer, called the class weight, is also used as an index for jumping to the label of the lexic analyzer fragment that analyzes the given lexeme. The restriction of this method is that each character should belong only to a single class (more precisely, each lexeme should start with a distinct class of character) which is not always feasible. For example, if the numbers are entered in hexadecimal form, the class of digits includes the symbols A-F, i.e., it has a nonempty intersection with the class of letters.

Rapid "symbol-to-class" conversion on the ES computer can be implemented using the instruction TR. To this end, in addition to the given source string, a so-called discriminator string should be created. Each byte in the discriminator represents the class of the character occupying the corresponding position in the input line. A schematic realization of this method is presented in Fig. 1. Here and in what follows we use a version of PL/1 corresponding to the optimizing PL/1 compiler under OS/ES which is essentially superior to the level F compiler judged by all the basic parameters. The discriminator D is created by applying the function TRANSLATE to the source string. Note that the operator D = TRANSLATE-(CARD, C) is translated by the optimizing compiler into the instructions MVC D, CARD and TR D, C.

As we see from Fig. 1, the jump to the lexic analyzer fragment analyzing lexemes of a given type is organized in the form of sequential search and not by means of a switch. This is so because for each indexed label the PL compiler includes an additional initialization instruction in the procedure prologue, thus virtually equating the efficiency of one access to a switch during a single procedure call to that of sequential search. If the classes are ordered by descending frequencies of occurrence of the corresponding lexeme types in the text, the sequential search is actually preferable.

As we know, lexic analysis usually takes up about 2/3 of the total compiling time in single-pass compilers. It is therefore desirable to increase the lexic analyzer efficiency as far as possible in order to allow more highly structured (and therefore less efficient) coding of DICP. Two special techniques can be applied in this context which, although not entirely original, nevertheless do not occur in the literature on lexic analyzers [2-7] examined by the author.

First, checking for end of line (which must be performed for each compound lexeme) has been reduced to checking for the end of a given lexeme (i.e., has been virtually eliminated). This is accomplished by increasing the discriminator length by one symbol (compared with the length of the string being analyzed). This last discriminator symbol is a special character (in Fig. 1 this is the character "7"), which is treated as the class of the special lexeme "end-of-line." The processing of this lexeme reduces to entering a new line. Thus, instead of the usual subprogram to read the next line which lexic analyzers call from several different points, we introduce an additional branch in the CASE operator identifying the type of the current lexeme.

By assigning the class 7 to any character (e.g., the character "?" in Fig. 1), we can include comments in the source data. The importance of this option is that it allows easy and reliable identification of batches of punched cards, paper tape rolls, and particularly magnetic carriers (magnetic tapes and diskettes). Moreover, it becomes possible to include special comments intended to facilitate the operator's job, such as:

```
? EXPERIMENTAL DATA 15.03.82.
PROCESS BY PROGRAM FACTOR
7 12 15 24 36 ? IVANOV A.V.
4, 48 10 30 18 ? PETROV S. N.
```

Second, the classes are arranged so as to optimize the header of the identifier extracting loop DO WHILE(D(K) = LETTER ! D(K) = DIGIT); (here LETTER and DIGIT are the characters representing the classes of letters and digits, respectively). If the class of digits is coded by the character "1" and the class of letters by the character "2," assigning the char-

```
LEXAN: PROC;
DCL((WORD, EXWORD) CHAR(255) VAR, (TYPE, EXTYPE) CHAR(1)),
EXT,
(C CHAR(256)/* CLASSIFIER */ INIT(
'333333333333333333333333333333333333333333333333333333333333333333333333
333333344444443333333334244444433333333436247333333333342254433333 3
333333333333333333333333333333333333333333333333333232222322422222222
222232342222222222332233432222222223223221111111111222223'),
(EOF INIT (0), FROM, N INIT(1), TOTAL, LEN) BIN FIXED,
(CARD, FIELD) CHAR(80), CLASS CHAR(1),
DISCRIMINATOR CHAR(81) INIT('7')) STATIC,
D(81)/* VECTOR = CLASSIFIER*/ CHAR(1) DEF DISCRIMINA-
    TOR; EXWORD=WORD; EXTYPE=TYPE;
@SWITCH: SELECT (D(N));
WHEN ('3') @CASE3: DO; /*BLANK*/
    N = N + 1; DO WHILE(D(N) = '3'); N = N + 1; END;
    GOTO @SWITCH;
    END @CASE3;
WHEN('1') @CASE1: DO; /*DIGIT*/
    TYPE = '9'; FROM = N; N = N + 1;
    DO WHILE(D(N) = '1'); N = N + 1; END;
    WORD = SUBSTR(CARD, FROM, N-FROM);
    END @CASE1;
WHEN('2') @CASE2: DO; /*LETTER*/
    TYPE = 'A'; FROM = N; N = N + 1;
    DO WHILE(D(N) < '3'); N = N + 1; END;
    WORD = SUBSTR (CARD, FROM, N - FROM);
    END @CASE2;
WHEN('4') @CASE4: DO; /*SEPARATOR*/
    WORD, TYPE = SUBSTR(CARD, N, 1); N = N + 1;
    END @CASE4;
WHEN('5') @CASE5: DO; /*QUOTE*/
    TYPE = 'X'; N = N + 1; FROM = N; TOTAL = 0;
@Q: CLASS = D (N);
    IF CLASS = '7'
    THEN CALL MES('%O AFTER % L FOUND UNFINISHED
    LITERAL');
    ELSE IF CLASS ¬='5' THEN DO; N = N + 1; GOTO @Q;
    END;
    /*QUOTE*/ LEN = N - FROM
    SUBSTR(FIELD, TOTAL + 1, LEN) = SUBSTR(CARD,
    FROM, LEN);
    N = N + 1; TOTAL = TOTAL + LEN;
    IF D(N) = '5' THEN D ; /*DOUBLE QUOTE*/
      FROM = N; N = N - 1; GOTO @Q;
      END;
    WORD=SUBSTR(FIELD, 1, TOTAL);
    END @CASE5;
WHEN('6') @CASE6: DO; /*UNRECOGNIZED SYMBOL*/
    CALL MES ('%O UNRECOGNIZED CODE');
    N = N + 1; GOTO @SWITCH;
    END @CASE6;
WHEN('7') @CASE7: DO /*QUESTION MARK (END OF LINE)*/

    IF EOF > 0 THEN GOTO @EOF;
    ON ENDFILE (SYSIN) GOTO @EOF;
    READ FILE(SYSIN) INTO(CARD);
    DISCRIMINATOR = TRANSLATE(CARD,C);
    D(81) = '7'; N = 1;
GOTO @SWITCH;
@EOF: TYPE = 'Z'; EOF = EOF + 1;
    IF EOF > 2
    THEN CALL MES('%O END OF FILE REACHED
    BEFORE END OF PROGRAM');
    END @CASE7;
OTHER CALL MES('%O CLASSIFER ERROR');
END @SWITCH;
END LEXAN;
```

Fig. 1

acter "3-9" to code the remaining classes (in Fig. 1 the class of ignored symbols is coded by the character "3," the class of separators by the character "4," quote by "5," unrecognized symbols by "6"), the header may be written in the form DO WHILE(D(K) <'3'). This header is compiled using the instruction CLI, which ensures fairly fast extraction of identifiers from the text. This arrangement of the classes is preferable to that used in the XPL compiler (see [6], Table 9.3.3).

The instruction TRT provides the most efficient method for detecting the end of compound lexemes on ES computers. Therefore, if the data mainly consist of numbers and identifiers (containing few literals), the lexic analysis can be speeded up by declaring a based variable (e.g., MASK CHAR(80) BASED(CURSOR);) and using the following algorithm to detect the end of a lexeme (e.g., the end of identifiers):

```
WHEN('2') @CASE2: DO;
          TYPE = 'A'; FROM = N; N = N + 1;
          CURSOR = ADDR(D(N));
          N = N + VERIFY(MASK, '12');
          WORD = SUBSTR(CARD,FROM,N - FROM);
          END @CASE2;
```

The shortcoming of this method is that the instruction TRT, which is compiled using the built-in function VERIFY, operates with a 256-byte dictionary, whereas in our case a 7-byte dictionary is quite sufficient (the discriminator does not include symbols other than 1-7).

Freefield data (like programs) contain a substantial number of blanks (about 50%). Under these conditions, it is paramount to maximize the skipping speed over ignored symbols, which also advocates using the instruction TRT.

Within the framework of the proposed method of lexic analysis, we can implement two additional tools which simplify data entry and improve the working conditions. When entering large homogeneous data files, various attributes have fairly high repetition rates. Macros can be used to avoid keying in the same sequence of symbols dozens of times. The simplest macros analogous to the macros of the NEATED editor [8] are easily implemented on the lexic analyzer level.

Macros can be defined by enclosing any sequence of symbols in macrobrackets of the form *%symbol* and *symbol%*, such as

```
%U MEMBER OF THE WRITERS UNION OF THE USSR U%
%L STATE PRIZE LAUREATE L%
```

After the macro is defined, the occurrence in the text of a combination of the form *%symbol* is interpreted as a macro call, e.g.,

```
TVARDOVSKII %U  %L.
```

Macros can be (nonrecursively) nested, and if necessary parameters can be passed to called macros in this way.

Another feature of input data is that they often include sequences of identical numbers, such as

0.008 0.008 0.008 0.01 0.01 0.01 0.02 0.02 0.02 0.02.

To simply the operator's job, it is advisable to introduce a special symbol, e.g., " (double quote), which is treated as a call to a special system macro with a value equal to the last number entered. Then the above sequence can be simply keyed in as

0.008 " " 0.01 " 0.02 """"

This option not only simplifies the operator's task but actually reduces the number of possible errors.

Syntactic Analysis of Raw Data

The syntactic analysis of the source data is intended to check the correspondence of the input to the specific grammar describing the file structure. Syntactic analysis is conveniently applied to check the structure of each record (the presence of separators, the sequence and the total number of attributes, etc.) and the sequence of records (if records of different types occur).

The syntactic structure of the file is conveniently described in BNF. For instance, the structure of a hypothetical file containing information about authors can be described by the grammar G (using a modified BNF accepted for the description of the language ADA [9]):

```
file :: = {record}
record :: = author_name [nationality][year_of_birth];
author_name :: = literal
nationality :: = identifier
year_of_birth :: = unsigned_integer
```

Examples of valid records in the grammar G are the following:

```
'PUSHKIN A.S.'RUSSIAN 1799;
'LERMONTOV M.Yu.' 1814;
```

The following two records are invalid in the grammar G:

```
'BYRON GEORGE' - 1788 1788;
'LOVELACE ADA' ENGLISH WOMAN 1815;
```

Figure 2 presents a simplified program of syntactic analysis of the grammar G.

For files without contextual dependences (e.g., without fields indicating the number of repetitions of the following elements in the record), a syntactic recognizer can be automatically generated in PL/1 from the BNF-description of the grammar. McKeeman's designer [6], in particular, can be adapted for this purpose. Direct programming in PL/1 of the syntactic analyzer for input data grammars is also fairly straightforward. The task can be further simplified by using a standard syntactic analysis block (see below).

The most interesting feature of the syntactic analysis block is that it allows flexible response to errors in the source data. We distinguish between two types of response: *neutralization* and *correction*.

Error neutralization denotes a technique of finding a neutralization point where the analysis can be resumed. The most primitive neutralization method is the so-called panic neutralization: the entire record in which an error is detected is deleted, i.e., the neutralization point is end of record. The advantage of the panic method is that it avoids unnecessary diagnostic messages about nonexisting erors, whereas its shortcoming is the loss of all the information in the error-containing record. Our method allows error neutralization with smaller loss of information.

We developed a simple neutralization method which for simple (e.g., homogeneous) data structures allows continuing the analysis within the erroneous record with minimum loss of information. The method uses the syntactic record template to find the neutralization point. If the string being analyzed has a syntactic skeleton which differs from the template, then the initial part of the string coinciding with the initial part of the template will be called the *skeleton head*, whereas the end of the string coinciding with the end of the template will be called the *skeleton tail*. Everything between the head and the tail is garbage, i.e., this is the part of the record to be neutralized. It can be replaced with a syntactically valid sequence of special neutralizing elements (e.g., each missing number can be replaced with zeros, each missing identifier with a special identifier MISSING_WORD, each missing literal with a special literal 'MISSING DATA').

We will examine the operation of the method in application to the grammar G. The syntactic template of a record in G is the string "XA9;" (we ignore the optional substrings in the grammar since, as we shall see below, they are introduced for the purpose of error correction). For the record " 'BYRON GEORGE' -1788 1788;" the syntactic skeleton is the string "X-99;" with the head X, the tail "9;" and garbage "-9." Replacing the garbage with the neutralizing lexeme (A, MISSING_WORD) required by the template we obtain the following neutralization: " 'BYRON GEORGE' MISSING_WORD 1788;." Similarly, the record " 'LOVELACE ADA' ENGLISH WOMAN 1815;" will be neutralized by removing the second identifier from the skeleton "XAA9;," i.e., it will be transformed to " 'LOVELACE ADA' ENGLISH 1815;."

Automatic error correction is more complex than neutralization, since it employs more complex transformations of the record being analyzed than neutralization does. There are two possible approaches to error correction: 1) using a more complex grammar in order to allow for possible syntax errors; 2) using formal syntax-error correction methods.

The first approach is simple to implement and in fact produces quite acceptable results. For example, the grammar G allows a situation in which the data elements "nationality" or "year of birth" are missing. Therefore the program SYNTAX (Fig. 2) will correctly process

the records " 'LERMONTOV M. Yu.' 1814;" (nationality missing) or " 'SHEVCHENKO T.G.' UKRAINIAN;" (date of birth missing). However, with this method, the designer should foresee the "typical errors" and correspondingly modify the grammar.

The second approach does not require special analysis of the possible errors but it is much more complex to implement. A large number of automatic syntax-error correction techniques are available (see the bibliography in [10]), but most of them are mainly of theoretical value. In our opinion, the most practicable method is the one described in [10-12] which, in historical perspective, constitutes an extension of the method for correction of spelling errors [13], which we will consider in some detail below.

No satisfactory correction methods are available for some of the common errors in input data (a shift from Russian to Latin case and vice versa; lack of separators between data of different types such as RUSSIAN 1814, etc.). These errors are not included in the ordinary methods for the correction of syntactic or semantic errors and require development of new methods.

Semantic Checking and Data Transformation

Compilers characteristically separate between syntactic analysis programs and semantic subprograms. The latter are combined into a semantic block. The application of this approach to DICP yields the same benefits as in compilers: the collection of semantic modules becomes more systematic, the syntax block is simplified, etc. It becomes possible to identify a set of "typical" semantic modules and to develop a standard subprogram including these modules. The variable parts can be tuned with a preprocessor. This is sufficiently simple for modules which a) check the range of the attributes and their significance; b) create output records; c) check arithmetic relationships within the record (horizontal check sums).

The possibilities for the correction of semantic errors are constantly growing. It is difficult to suggest general methods for the correction of semantic errors, since they include many different error categories. Most of the existing methods for the correction of semantic errors utilize the context in which the error is detected.

Practicable correction methods are currently available only for one type of semantic errors, the so-called spelling errors [14]. Suppose that an identifier (or a literal) is not found in the dictionary which lists all the allowed values of this identifier. It is natural to assume that this unrecognized identifier is a distorted image of one of the elements in the dictionary. Several methods can be applied to find the "source" of the distorted element [15]. One of the most successful methods is that described in [16, 17], which is based on dynamic programming techniques and has been developed for speech recognition by computer. The method will correct several errors in one word.

In practical terms, it suffices to correct only single spelling errors. This can be done by the method described in [13, 18], which combines simplicity with fairly high efficiency and allows correcting four categories of spelling errors: 1) one letter keyed in incorrectly; 2) one letter missing; 3) an extra character inserted; 4) two adjoining characters transposed (statistical analysis shows that nearly 80% of all spelling mistakes fall in one of these categories).

The implementation of this method includes three stages: 1) select from the dictionary all the possible candidates for replacement of the source word; 2) eliminate unsuitable candidates, retaining only those which can be obtained by inserting, omitting, changing, or transposing a symbol; 3) if more than one candidate remains, select the best candidate from contextual considerations or from some indirect criteria.

The first round of candidate selection is very fast, using the following criteria: the length of the candidate should not differ from the length of the input word by more than one symbol (since only single errors are being corrected); from among the words of correct length select only those in which either the first or the second letter is the same as in the input word.

The second stage uses the algorithm which is schematically shown in Fig. 3. The function MATCH (see Fig. 3) is called once in order to determine acceptability of the candidate and to identify the type of correction required. Two cases are distinguished: a) the compared words are of the same length; b) the word lengths differ (by one symbol).

```
SYNTAX: PROC OPTIONS(MAIN);
DCL (TYPE, EXTYPE) CHAR(1) EXT;
@READLOOP: CALL LEXAN; /* READ AUTHORS NAME*/
    IF TYPE¬= 'X' THEN GOTO @ERROR;
    CALL LEXAN; /* READ <NATIONALITY> OR
    <YEAR OR BIRTH>*/
    IF TYPE¬= 'A' & TYPE¬='9' THEN GOTO @ERROR;
    IF TYPE = 'A' THEN CALL LEXAN; /*

    IF TYPE ¬ ='9' & TYPE ¬ =';' THEN GOTO @ERROR;
    IF TYPE = '9' THEN CALL LEXAN; /*READ <;>*/
    IF TYPE ¬ =';' THEN  GOTO @ERROR;
    GOTO @READLOOP:
@ERROR: CALL MES ('ERROR IN MESSAGE STARTING
WITH %R');
    DO WHILE(TYPE ¬ =';'); CALL LEXAN; END;
    GOTO @READLOOP;
END SYNTAX;
```

Fig. 2

```
MATCH: PROC RETURNS(BIT(1));
    DCL (WORD, CANDIDAT) CHAR(80) VAR EXT,
        ERTYPE CHAR(1) EXT,
        FIRST BIN FIXED INIT(0),
        (WF, CF) CHAR(80),
        WS(80) CHAR(1) DEF WF,
        CS(80) CHAR(1) DEF CF;
    WF = WORD; CF = CANDIDAT;
    IF LENGTH(WORD) = LENGTH(CANDIDAT)
    THEN /* EQUAL LENGTHS*/
@CASE_A: DO K=1 TO LENGTH(WORD);
        IF WS(K) ¬ = CS(K) THEN DO;
            IF FIRST = 0 THEN DO; FIRST = K; ERTYPE = 'R';
            END;
            ELSE DO;
            IF K¬= FIRST + 1| WS(FIRST) ¬= CS(K)| WS (K)¬=
            CS(FIRST) THEN RETURN('0'B);
            ERTYPE = 'X';
            END;
        END @CASE_A;
    ELSE @CASE_B: DO; /* UNEQUAL LENGTHS*/
        IF LENGTH(WORD) > LENGTH(CANDIDAT)
        THEN ERTYPE = 'I'; ELSE ERTYPE = 'D';
        L = MIN(LENGTH(WORD), LENGTH(CANDIDAT));
        DO K = 1 TO L WHILE(WS(K) = CS (K)); END;
        IF K<=L THEN DO;
            IF ERTYPE = 'I' THEN DO; M = K + 1; N = K; END;
            ELSE DO; M = K; N = K + 1; END;
            L = L - K + 1; /*L ← TAIL LENGTH*/
            IF SUBSTR(WF, M, L) ¬ = SUBSTR(CF, N, L)
            THEN RETURN('0'B);
        END @CASE_B;
    RETURN('1'B);
END MATCH;
```

Fig. 3

For case (a), if the words differ only in one symbol, then we have a replacement error (ERTYPE='R'); if the words differ in two adjoining symbols, we have a transposition error (ERTYPE='X'); otherwise the candidate is dropped.

For case (b), the algorithm locates the first nonmatching symbol from the left and compares the tails of the two words, starting with the first nonmatching symbol in the shorter word. If the tails match, the candidate is acceptable (ERTYPE='D' if a symbol is missing in the input word and ERTYPE='I' otherwise); otherwise the candidate is dropped.

If more than one candidate remains after this screening, a third stage is implemented. It can be based on heuristic techniques exploiting the keyboard layout of the data entry station. For example, correction of transpositions can be ranked according to the distance between the corresponding data entry keys (with allowance for the relevant shift).

Experience shows that the proposed algorithm is quite satisfactory with input words longer than five symbols (some 95% of the errors are corrected correctly). The best results are achieved with correction of natural language words (names of towns, lists of names, etc.).

When working with large data bases, the speed of the second stage can be increased by programming the function MATCH in Assembler. In this case, the first nonmatching symbol is conveniently detected by a sequence of instructions XC ("Exclusive OR") and TRT (after the instruction XC is executed on the candidate and the input word, the first symbol in the result which is not X'00' corresponds to the nonmatching symbol in the compared words).

Generation of Diagnostic Messages

Poor diagnostics is characteristic of most existing data input and checking programs. Yet the information content of the diagnostic messages can be substantially improved by including in the diagnostic subprogram a simple macrogenerator which inserts in the text of the message macro calls of the form *%symbol*, where *symbol* is the macro call name (see Fig. 2). The macro generator will substitute for each macro call it encounters the corresponding value from the input text, e.g., %R is replaced with the text of the current lexeme, %L is replaced with the text of the preceding lexeme, %O is replaced with the text "RECORD record_number CONTAINS AN ERROR:," etc.

For specific applications it is advisable to develop a standard diagnostic message scheme and to identify the words and phrases which are most frequently used in diagnostics. These phrases can be coded as additional macro calls, thus reducing the memory space occupied by the message texts.

Implementation Features

In order to simplify DICP design, we developed a number of ready-made standard blocks (in PL/1) of the previously described procedures. These include the lexic analyzer NEATLEX, the syntax analyzer NEATSYNT, the semantic checking block NEATSEM, and the message generator NEATMES [19].

Some of the variable text fragments in the standard blocks are handled by PL/1 preprocessor operators. The other changes should be made manually using a text editor. This approach is attributable both to the insufficient flexibility of the PL/1 preprocessor (compared with the OS/ES Assembler macro processor) and to the difficulty of foreseeing the changes that will be required in order to adapt the standard blocks.

To simplify the DICP debugging, the standard blocks include a system for printing out the debug information, controlled by a global bit string DEBUG (the output of debug information from each block is controlled by a certain bit in the string). For example, in the NEATLEX block, the debug operators are included in a DO group IF DEBUG &'1'B THEN DO;...END;

Since the debug operators increase the size of each block, they all are additionally enclosed in the preprocessor group %IF DEBUGGING=1%THEN %DO; ...%END;. This preprocessor group also applies to the debugging prefixes SIZE, STRINGSIZE, SUBSCRIPTRANGE, which should be included in each block in the debugging mode.

The blocks communicate mainly through external variables, and not by parameters. This ensures a certain "indicative dump" which prints out the values of the main global variables when an execution error occurs in the program.

Some Advantages of SDDICP

The proposed method of DICP design was applied by the author to develop a system for statistical processing of cost-accounting data. Experience shows that the SDDICP has a number of operating advantages compared with the usual DICP: fewer passes are required to clear the input data from errors; the option of shorthand coding of attributes reduces the number of errors and speeds up data entry; the inclusion of comments helps to prevent confusion with data stored on magnetic media and facilitates the operator's job.

These features are particularly valuable for interactive data entry. Moreover, interactive SDDICP offer more extensive automatic error correction features than batch SDDICP. The most difficult errors can be eliminated by prompting the operator to approve the proposed correction.

An important feature of SDDICP is the unification of input data formats with the format used to input program texts (80-byte records with a sequence field). As a result, the input data can be stored in ordinary text libraries and manipulated by most of the software intended for program text processing. The level of this software is usually an order of magnitude higher than the corresponding data processing software. In particular, data correction using screen editors is faster, more convenient, and more reliable than with ordinary data correction programs (which run in the batch mode and have a very primitive instruction set).

In some cases, both the raw and the checked data can be stored in libraries. This is particularly useful if the data are structured so that a unit of information is representable as a library segment. For example, in the above-described cost-accounting system, the reported data of each plant are entered in a segment identified with the plant code and the processing is done by interpreting the freefield data. In this way the system exploits the power OS/ES and utility tools available for the processing of library data sets (insert, delete, rename, report, etc.).

The user field provided in each element of the table of contents can be used to store information characterizing the logical unit of information on the whole (in our case, a plant report). In this way we can create a whole logging system which at any moment in time will produce information about recorded, checked (with an indication of the number of detected errors), and processed reports. This logging system substantially simplifies the operation.

* * *

The rapid development of computer applications forces us to focus on the creation of more "intelligent" DICPs. Syntax-directed DICPs make it possible to create programs which free the user from the chore of correcting and repeatedly entering data three, four, or more times. As the amount of input data entered on machine-sensible media increases, SDDICPs can be introduced to somewhat alleviate the load experienced by data entry departments.

The use of data banks has been recently expanding on a large scale. While simplifying certain aspects of application program development, data banks create a whole range of new problems. One of these is the loading of data into the base. The data loading programs usually do not ensure adequate checking of the input data, and the format assumed by the loading program may be inconvenient for particular applications.

One of the ways to overcome this difficulty is by creating a preprocessor for the data base loading program. The proposed method of DICP design is fairly convenient for this purpose: the SDDICP output format (the input format of the data base loading program) is fixed, so that we can prepare one set of semantic output modules and use them in all the preprocessors to be designed.

In conclusion, we would like to stress again that the established dichotomy of information into programs and data has always been and still remains quite arbitrary. This applies both to data processing and data storage methods. Therefore, considerable benefits can be gained by overcoming the "program–data" barrier.

LITERATURE CITED

1. V. N. Agafonov et al., A Generator of Data Input Programs for the ES Computer [in Russian], Statistika, Moscow (1976).
2. D. Gries, Compiler Construction for Digital Computers, Wiley, New York (1971).
3. J. F. Gimpel, "The minimization of spatially multiplexed character sets," Commun. ACM, 17, No. 6, 315-318 (1974).
4. L. Baulier, "Compiler design methods," in: Programming Languages [Russian translation], Mir, Moscow (1972), pp. 87-277.
5. W. L. Johnson, J. H. Porter, S. I. Askley, and D. T. Ross, "Automatic generation of efficient lexical processors using finite state techniques," Commun. ACM, 11, No. 12, 805-822 (1968).
6. W. M. McKeeman, J. J. Horning, and D. B.Wortman, A Compiler Generator, Prentice-Hall, Englewood Cliffs, New Jersey (1970).

7. Y. Chu, "A methodology of software engineering," IEEE Trans., Software Eng., SE-1, No. 3, 262-270 (1975).
8. N. N. Bezrukov, "A text editor with a powerful instruction set for OS/ES," Programmirovanie, No. 3, 39-48 (1981).
9. The Programming Language ADA (A Preliminary Description) [Russian translation], Finansy i Statistika, Moscow (1981).
10. K. S. Tai, "Syntactic error correction in programming languages," IEEE Trans., Software Eng., SE-4, No. 5, 414-425 (1978).
11. S. Feyock and P. Lazarus, "Syntax-directed correction of syntax errors," Software Practice Exp., 6, No. 2, 207-219 (1978).
12. F. E. Moth and A. I. Tharp, "Correcting human errors in alphanumeric terminal input," INf. Process. Manag., 13, No. 4, 329-337 (1976).
13. F. Damerau, "A technique for computer detection and correction of spelling errors," Commun. ACM, 7, No. 4, 171-176 (1964).
14. J. L. Peterson, "Computer programs for detecting and correcting spelling errors," Commun. ACM, 23, No. 12, 676-687 (1980).
15. P. Hall and G. Dowling, "Approximate string matching," Computing Surveys, 12, No. 4, 381-402 (1980).
16. T. K. Vintsyuk, "Speech recognition by dynamic programming methods," Kibernetika, No. 1, 81-88 (1968).
17. T. K. Vintsyuk, "Element-by-element recognition of continuous speech using words from a given dictionary," Kibernetika, No. 2, 133-143 (1971).
18. H. L. Morgan, "Spelling correction in system programs," Commun. ACM, 13, No. 2, 90-94 (1970).
19. N. N. Bezrukov, "Translation to a high-level language as a method for realization of problem-oriented languages based on relational algebra," Candidate's Dissertation, IK Akad. Nauk UkrSSR, Kiev (1981).

ONE APPROACH TO THE SPECIFICATION AND VERIFICATION OF TRANSLATORS

V. A. Nepomnyashchii and A. A. Sulimov

UDC 681.142.2:518.5

An approach to specifying and verifying a translator is described using as an example the translator of a subset of BASIC. Facilities are listed for constructing formal specifications of the scanner, syntactic and semantic analyzer, and code generator. The method of inductive statements is used for verification. The correctness of the specifications is machine tested.

Although small programs can be successfully verified (i.e., their correctness proved) by the Floyd-Hoare method, the verification of large programs such as interpreters, translators, or operating systems still remains largely unsolved [1]. For verification a program must first be annotated (i.e., formally specified including the output procedure conditions and loop invariants), and the conditions of correctness must be derived (using the axiomatic semantics of the programming language constructs employed) and proved.

The main difficulties encountered in verification of large programs are associated with the construction of formal specifications which must be complete, compact, and suitable for deriving correctness conditions. The approach to translator specification described in [2] uses abstract data types. It is however not clear if the same approach is suitable for translator verification.

The present paper is a first step towards an analysis of the problem indicated above. As a model for analysis we take the translator of a subset of BASIC (MINIBASIC [3]) into BÉSM-6 computer machine language implemented in PASCAL. The translator is composed of three parts: a scanner, syntactic and semantic analyzer, and code generator whose specifications are separately described. To specify the scanner it was found convenient to use a model of

Translated from Programmirovanie, No. 4, pp. 51-58, July-August, 1983. Original article submitted May 25, 1982.

0361-7688/83/0904-0195$07.50