PROGRAMMING AND COMPUTER SOFTWARE

A translation of *Programmirovanie*

Volume 5, Number 4 July-August, 1979

CONTENTS

The Russian press date (podpisano k pechati) of this issue was 7/19/1979. Publication therefore did not occur prior to this date, but must be assumed to have taken place reasonably soon thereafter.

A MODIFIED FLOYD — EVANS LANGUAGE

N. N. Bezrukov

UDC 51:681.3.06

The MFEL is described, which is a modification of the Floyd – Evans language FEL for writing parsers.

In the early 1950s Markov [1] defined an algorithmic system that became known as a system of normal Markov algorithms. Although Markov suggested normal algorithms as a technique for examining insolubility in mathematical logic, it was found that the concept of reducing complex test transformations to various simple structural transformations (canonical substitutions) made it possible to program operations upon lines of characters. Later on, various languages were devised for processing characters on this basis (Comit [2], Snobol [3], et al.), which received the general name of languags of Markov-algorithm type [4].

Floyd [5] proposed a similar approach in the field of parsing, when in 1961 he devised a compact and convenient notation for parsing algorithms. In that case, an algorithm for parsing an arithmetic expression, which was previously described by a long sequence of block diagrams [6], could be written by means of a set of rules for reduction of the initial text. Floyd's notation was developed to the level of a language for describing parsing algorithms by Evans, who in 1963 published a description of his ALGOL-60 compiler [7].

The parsing in Evans' compiler was performed by interpreting the parsing algorithm written as a program in a language later known as the Floyd – Evans language FEL. In the 1960s, FEL was used with minimal changes as a syntactic metalanguage in various compiler-construction systems, including Feldman's FSL system of 1964 [8]. FEL is interpreted in FSL directly during the parsing, as in Evans' compiler. The need for high interpretation speed was responsible for features of the treatment such as positional notation, rigid restrictions on the structure of statements, and the minimum number of auxiliary words.

A new stage in the development of FEL began in 1970 with the publication [9] of an assembler version of an FEL optimizing compiler. The method of compiling FEL programs described there increased the rate of execution by a factor of 10 in comparison with interpretation. This provided syntactic recognizers comparable in store volume and speed to those written by hand. In [9], an ELSE part was introduced into the FEL statements, which was absent from previous versions of FEL, and it was suggested that a queueing system should be used to organize the repeated scans of the input text in parsing. Further, names could be used for semantic functions, so semantic information could be used during the parsing. The queueing also meant that FEL could be used to simulate Markov algorithms, and the class of languages that could be analyzed in that form coincides with the class of recursively denumerable languages.

In 1971 Gries [10] discussed FEL in his seventh chapter; although Gries' version was designed for interpretation, it did provide a very good generalization of 10 years' experience with FEL.

From 1965 onwards, various algorithms have been published for translating from BNF grammars for some subsets of context-free languages, particularly to give an FEL program that constitutes a recognizer [11-13]. In 1972 an algorithm was published for translating a large subclass of affix grammars into an analysis program in FEL [14].

The development of FEL occurred in isolation from papers on the use of Markov algorithms and papers related to string-processing languages. It would therefore seem that Haynes and Schütte's idea has remained virtually overlooked, although it is readily transferred to the context of application of Markov algorithms and of string-processing languages. In particular, the recognizer suggested in 1972 was based directly on Markov algorithms [15], and although it had the power of Haynes' version it was much less convenient and did not provide efficient parsers. In application to Snobol, optimization by compilation of standards (patterns in Snobol terms) to give machine code directly was described only in 1977 [16]. Also, the introduction of queues into FEL makes the latter similar to the Comit language, and therefore it is possible to envisage the use of FEL in string processing and further that languages of Markov-algorithm type could be used in syntactic recognizers.

Translated from Programmirovanie, No. 4, pp. 53-64, July-August, 1979. Original article submitted August 10, 1978.

Various researchers working independently of Floyd considered the representation of parsing as a sequence of structural transformations; e.g., in 1970 Vel'bitskii and Yushchenko [17] suggested a metalanguage for context-sensitive grammars that was particularly intended for describing a recognizer with one or more pushdown stores. A form of such grammar convenient for computer input (the SM language) was proposed in 1972 by Lavrishcheva and Yushchenko [18]. In 1973 Vel'bitskii suggested the R-grammar metalanguage [19], which he considered as an extension of the SM grammars.

An interpreter for a language analogous to R grammars has also been described [20]. The two metalanguages are similar to FEL if a single pushdown store is employed, but the standard consists of a single element.

We have developed the MFEL language as a modification of FEL, which is an attempt to extend Gries' version via Haynes and Schütte's concepts. A difference from previous versions is that MFEL uses the notation employed in languages of ALGOL type.

MFEL is a specialized language for writing parsers, so it does not have means for describing the semantics of the language. However, an MFEL program is relatively readily translated into assembler or into virtually any high-level language. Therefore, it is convenient to write the parser in MFEL and then employ a compiler from MFEL to get an equivalent program in the language that is employed in the syntactic subroutine. A compiler from MFEL to PL/1 has been written for the ES OS system. A comparison is made below between MFEL and Gries' version of FEL.

Basic Language Concepts

Any MFEL program works with an input text and a stack, which holds the codes for the lexical units (lexemes), which in MFEL are called atoms. The topmost atom in a stack is called the window, and the group of top atoms is called the top of the stack. In the initial state, the stack always contains the atom ⟨!−⟩.

The MFEL statements provide for execution of four operations upon the stack: insertion of atoms into the stack, comparison of the top of the stack with a certain atomic configuration, replacement of the top of the stack by a given configuration, and elimination of atoms from the stack. The description of the MFEL syntax is given in the Appendix.

Lexical units are recognized by a lexical analyzer, which is called by the SCAN statement in MFEL, and which isolates from the text one lexeme and inserts the corresponding atom into the stack. For simplicity we assume that the standard lexical analyzer in MFEL distinguishes the following lexical categories: identifiers, which correspond to the atom ⟨I⟩; auxiliary words (some subset of identifiers), in which the atom for each coincides with the word itself (e.g., if IF is an auxiliary word for the input language, then it corresponds to the atom ⟨IF⟩); the separators +, /, etc., for each of which the atom is the separator itself; literals (of the form of any string of letters), each of which corresponds to the atom ⟨L⟩; and integers, which correspond to the atom ⟨9⟩.

For example, consider an MFEL program that consists only of SCAN statements and that analyzes the line (A + E)/H; during the execution of the first SCAN statement the lexical analyzer inserts the atom ⟨(⟩ into the stack, and on the second it inserts ⟨I⟩ (identifier), on the third ⟨+⟩, etc. The analyzer inserts ⟨−!⟩ into the stack on reaching the end of the input text.

The name "standard" is given to a configuration of atoms to be compared with the top of the stack. The top of the stack coincides with the standard if all the positions in the standard coincide with atoms at the top of the stack. The comparison is performed from right to left, so the window in the stack is compared with the rightmost atom in the standard. The comparison of the top with the standard is specified in MFEL by the IF statement.

The top of the stack can be replaced in MFEL only after comparison with some standard (in the conditional statement). Execution of the replacement statement causes the deletion from the stack of a number of atoms equal to the number in the standard. Atoms from the replacement statement are then inserted into the stack. The leftmost atom from the replacement statement is inserted first into the stack.

The POP statement is used in deleting atoms from the stack in MFEL; the POP statement can specify either the number of atoms to be deleted or the standard to be used in the replacement. If neither the number of atoms to be deleted nor the standard is given, then only a single atom is deleted from the stack. For example, the statement

POP ⟨!—⟩;

denotes deletion of all the atoms present in the stack, because the atom ⟨! – ⟩ is the left limiter for the input line.

Examples of MFEL Programs

Example 1. Program ¤ EXPRES (Fig. 1). This program is used in analyzing arithmetic expressions in accordance with grammar Γ:

```
      Γ :: =! —  ⟨EXP⟩  —!
  ⟨EXP⟩  :: = ⟨TERM⟩ ! ⟨EXP⟩  (+ ! — ) ⟨TERM⟩.
 ⟨TERM⟩  :: =  ⟨SET⟩   !  ⟨TERM⟩ (* !/)  ⟨SET⟩
    ⟨SET⟩ :: = ⟨I ⟩ ! ⟨9⟩ ! «(»  ⟨EXP⟩ «)»
```

We first consider the structure of the IF statement with label @T, which specifies comparison of the top of the stack with the standard ⟨TERM⟩ (⟨*⟩! ⟨/⟩) ⟨SET⟩ ANY; the right-hand element in the standard is the auxiliary word ANY, which means that the corresponding position in the stack does not participate in the comparison. The standard is scanned from right to left. Before ANY we find the atom ⟨SET⟩, and the third element in the standard is the class (⟨*⟩! ⟨/⟩), which consists of the two atoms ⟨*⟩ and ⟨/⟩. This element in the stack coincides with this class in the standard only if it is one of the atoms entering into that class. The fourth and last element in the standard is the atom ⟨TERM⟩. Therefore, this standard may coincide with various configurations of atoms at the top of the stack.

When the top of the stack coincides with the standard, the statements appearing in the THEN group of the IF statement are executed. In the present case, the THEN group consists of a single statement, the statement for replacing the top of the stack by ⟨TERM⟩ANY, where ANY in the replacement statement corresponds to the atom termed the substitution atom, which is not involved in the comparison. Therefore, when the standard coincides with the top of the stack the top four atoms are deleted from the stack, and then the stack receives the atom ⟨TERM⟩ and the substitution atom.

If coincidence does not occur, the ELSE group is executed, which begins with the auxiliary word ELSIF, which, as in ALGOL-68, is a contraction of ELSE IF. Therefore, the ELSE group contains a nested IF statement, which specifies comparison of the top of the stack with the ⟨SET⟩ ANY standard. The THEN group in this statement also consists of a single statement, namely that for replacement of the top of the stack. The auxiliary word FI terminates the statement.

The ¤EXPRES program also contains the three statements ER, GO, and STOP. The ER statement is intended to output diagnostic messages during parsing, while GO controls the jump to the corresponding label, and STOP halts execution.

We now consider the operation of ¤ EXPRES.

```
 1      ¤EXPRES: PROC MAIN;
        /* THIS PROGRAM PARSES        */
        /* ARITHMETIC EXPRESSIONS     */
 2         SCAN;
 3      @M:
 5         IF (⟨I⟩ ! ⟨9⟩) THEN  ⟨SET⟩; SCAN;
 6  1          ELSIF ⟨(⟩ THEN SCAN; GO @M;
 9  1          ELSE ER [' ERROR    1.'];
10  1          FI;
11      @T:
           IF ⟨TERM⟩(⟨*⟩!⟨/⟩⟨ ⟨SET⟩ ANY THEN ⟨TERM⟩ANY;
13  1          ELSIF ⟨SET⟩   ANY THEN ⟨TERM⟩ ANY;
15  1          FI;
16         IF ⟨TERM⟩(⟨*⟩!⟨/⟩) THEN SCAN; GO @M; FI;
20         IF ⟨EXP⟩ (⟨+⟩!⟨—⟩)⟨TERM⟩ANY THEN  ⟨EXP⟩ ANY;
22  1         ELSIF ⟨TERM⟩ ANY THEN  ⟨EXP⟩ ANY;
24  1         FI;
25         IF ⟨EXP⟩ (+⟩!⟨—⟩) THEN SCAN; GO @M;
28  1         ELSIF ⟨(⟩ ⟨EXP⟩ ⟨)⟩ THEN  ⟨SET⟩; SCAN; GO @T;
32  1         ELSIF ⟨!—⟩⟨EXP⟩  ⟨—!⟩; THEN ⟨Γ⟩; STOP;
35  1         ELSE ER[' ERROR   2.'];
36  1         FI;
37      END ¤EXPRES;
```

Fig. 1

```
***START TRACE ¤EXPRES
STACK AFTER OPER.  2(SCAN;)            |—  (
STACK AFTER OPER.  7(SCAN;)            |—  (     I
STACK AFTER OPER.  5(REPLACE TOP)      |—  (     SET
STACK AFTER OPER.  5(SCAN;)            |—  (     SET i   +
STACK AFTER OPER. 15(REPLACE TOP)      |—  (     TERM    +
STACK AFTER OPER. 24(REPLACE TOP)      |—  (     EXP     +
STACK AFTER OPER. 26(SCAN;)            |—  (     EXP     +     I
STACK AFTER OPER.  5(REPLACE TOP)      |—  (     EXP     +     SET
STACK AFTER OPER.  5(SCAN;)            |—  (     EXP     +     SET)
STACK AFTER OPER. 15(REPLACETOP)       |—  (     EXP     +     TERM)
STACK AFTER OPER. 22(REPLACE TOP)      |—  (     EXP)
STACK AFTER OPER. 30(REPLACE TOP)      |—  SET
STACK AFTER OPER. 30(SCAN;)            |—  SET    /
STACK AFTER OPER. 15(REPLACE TOP)      |—  TERM   /
STACK AFTER OPER. 17(SCAN;)            |—  TERM   /    I
STACK AFTER OPER.  5(REPLACE TOP)      |—  TERM   /    SET
STACK AFTER OPER.  5(SCAN;)            |—  TERM   /    SET    —|
STACK AFTER OPER. 13(REPLACE TOP)      |—  TERM   —|
STACK AFTER OPER. 24(REPLACE TOP)      |—  EXP    —|
STACK AFTER OPER. 34(REPLACE TOP)      Γ
*** END TRACE ¤EXPRES.
```

Fig. 2

The program begins by scanning the first lexeme and inserting the corresponding atom into the stack. Statements 3 and 6 check whether this atom begins a set. If it cannot, statement 9 outputs an error message. If one of the atoms ⟨I⟩ or ⟨9⟩ is encountered, then this is replaced by ⟨SET⟩. Then the next lexeme is scanned and control is transferred to the next statement, i.e., statement 11. If the set begins with a left parenthesis, then statement 8 transfers control to statement 3. Before the execution of statement 13, the set always contains ⟨SET⟩ ANY and statement 14 is obliged to replace ⟨SET⟩ by ⟨TERM⟩, but first of all it is necessary to check whether the following rule is applicable:

⟨TERM⟩ :: = ⟨TERM⟩ (* ! /) ⟨SET⟩.

When one of these reductions has been performed, a check is made whether one of the atoms ⟨*⟩ or ⟨/⟩ occurs in the window. If the answer is yes, the following lexeme is scanned and a return is made to statement 3 in order to prepare for the reduction ⟨TERM⟩: : = ⟨TERM⟩ (* ! /) ⟨SET⟩, for which purpose it is necessary to find a phrase that is reducible to ⟨SET⟩. If the window contains an atom different from ⟨*⟩ and from ⟨/⟩, then either the appropriate reduction is performed for ⟨EXP⟩ or an error message is output.

Figure 2 illustrates the tracing of expression (A + E)/H by the ¤ EXPRES program as implemented in the MFEL translator to facilitate program debugging. During the tracing, the contents of the stack are printed out when any change is made, as well as the number and type of the statement causing the change.

Example 2. The ¤PALINDROM Program (Fig. 3). We suppose it is necessary to write a recognizer for a language described by grammar Γ:

Γ :: = |— ⟨P⟩ —|

⟨P⟩ :: = + ⟨P⟩ + ! — ⟨P⟩ — ! + ! —

This grammar describes a palindrome of characters + and – with an unlabeled center [21]. Note that Γ is not an LR(K) grammar. The simplest technique for parsing this grammar is a two-pass algorithm that on the first pass determines the degree of the palindrome and in the second finds the center and performs the successive reduction. Figure 3 shows this algorithm as the ¤PALINDROM program.

This program uses various constructions absent from the previous example. These include a repetition statement, a looping statement, and the phrases TO and FROM in the SCAN statement. We first consider the repetition statement. This introduces elements of four types: the semantic block SEMAN, which consists of one semantic action # COUNT, the filter block TEST, which consists of the single filter #MIDDLE, the queue BUFFER, and the syntactic class %SIGN, which includes the two atoms ⟨+⟩ and ⟨–⟩.

In most current compilers there is a clear distinction between the software that parses the input program and the software that generates the object code (semantic procedures). This distinction between syntax and semantics provides for formalization and automation of the parsing, together with the use of a more systematic approach in implementing the semantics. Usually the semantic procedures, each with its own number, are combined into a semantic block. The number of the cell procedure is transmitted as a parameter when the

```
1      ¤PALINDROM: PROC MAIN;
2      DCL SEMAN ACTION(#COUNT : 1),
          TEST FILTER(#MIDDLE : 1),
          BUFFER QUEUE,
          %SIGN CLASS (⟨+⟩ | ⟨—⟩);
       /* READ INPUT TEXT INTO    BUFFER */
3         DO FOREVER;
4   1         SCAN TO BUFFER; #COUNT;
6   1         IF %SIGN THEN;
8   2            ELSIF ⟨—|⟩ THEN POP⟨|—⟩; GO @MIDDLE;
11  2            ELSE ER['NOT PALINDROME  .']; STOP;
13  2            FI;
14  1         END;
15     @MIDDLE : /* SEARCH FOR CENTER OF PALINDROME */
          DO UNTIL #MIDDLE;
16  1         SCAN FROM BUFFER;
17  1         END;
       /* REDUCTION OF CENTER OF PALINDROME  */
18        IF %SIGN THEN ⟨W⟩; SCAN; FI;
22        DO FOREVER;
23  1         IF ⟨+⟩ ⟨W⟩ ⟨+⟩ THEN ⟨W⟩;
25  2            ELSIF ⟨—⟩ ⟨W⟩ ⟨—⟩ THEN ⟨W⟩;
27  2            ELSIF ⟨|—⟩ ⟨W⟩ ⟨—|⟩ THEN ⟨Γ⟩; STOP;
30  2            ELSE ER['NOT PALINDROME']; STOP;
30  2            FI;
33  1         SCAN FROM BUFFER;
34  1         END;
35     END ¤PALINDROM;
```

Fig. 3

semantic block is called. The overall volume of the semantic block may be very large, so one commonly employs several semantic blocks, each of which contains perhaps 10-100 semantic procedures. The large number of procedures means that it becomes difficult to recall what number and what block relates to a particular semantic procedure. Therefore, the semantic procedures in MFEL (which are called semantic actions or simply actions) have names, and the block identifier and the number of a procedure in the block are specified in the DCL statement.

Therefore, the phrase SEMAN ACTION (# COUNT: 1) means that the semantic action # COUNT is in the semantic block SEMAN as number 1.

The phrase TEST FILTER (# MIDDLE: 1) is the call to the block of filters TEST, which consists of the single filter # MIDDLE; the filter concept (semantic function) is analogous to the action concept. The difference is that filter may be an element of a standard and return a signal to the calling point (which takes two values, which are called TRUE and FALSE, although there are no Boolean quantities in MFEL). This signal is used in comparing the standard with the top of the stack, and it is considered that there is no coincidence if the filter returns the signal FALSE.

The queue call in MFEL serves to organize repeated scanning of the input text during translation.

The class call puts some sets of atoms into correspondence with the class name. In the ¤ EXPRES program, there is a class called by context. In this program %SIGN is the name of an explicitly called class, which has the two atoms ⟨+⟩ and ⟨−⟩. The name of the class can be used in the standard, in the statement for replacement of the top of the stack, and in the conditional-jump statement.

We now consider the operation of ¤ PALINDROM; the DO FOREVER statement specifies infinite repetition of the statements appearing within it. These statements 4-13 implement the first pass of the text. The statement SCAN TO BUFFER provides for transferring an atom to the stack and storing the current lexeme in the queue having the name BUFFER. In general, if the TO phrase is used in the SCAN statement, the current lexeme and the corresponding atom are entered into the queue, and when FROM appears in the SCAN statement then the queue is used as the source of input text. Both phrases may be given in a single statement, and in that case they may both point to the same queue, in which case the current lexeme is transferred to the end of the queue after the corresponding atom has been inserted into the stack. This means that multiple passes can be organized.

Statement 5 specifies execution of the semantic action # COUNT; it is assumed that this action consists in counting the number of elements in the palindrome. When the end of the text is reached, statement 9 clears

```
1        ¤REPLACE: PROC MAIN;
2        DCL OUTPUT ACTION(#WRITE: 1);
3           DO FOREVER;
4   1           SCAN;
5   1           DO UNTIL(<;> ! <—|>); SCAN; END;
8   1           IF <—|> THEN STOP;
10  2              ELSIF <|—> <PUT> <EDIT> <(> <I> <)> <(> <A> <)> <;>
                   THEN <|—> <CALL> <OUT> <(> <I> <)> <;>;
12  2              FI;
13  1            #WRITE; POP<|—>;
15  1            END;
16        END ¤REPLACE;
```

Fig. 4

the stack completely and statement 10 passes control to the statement with label @MIDDLE, which indicates looping.

Apart from an infinite loop statement in MFEL, there are two looping statements of other types: the WHILE statement, which is executed while the top of the stack coincides with the standard, and the UNTIL statement, which is executed until the top of the stack coincides with the standard. If on the first execution the top of the stack coincides (does not coincide) with the standard, then the body of the UNTIL(WHILE) loop is not executed.

In the present case, the standard consists of the filter #MIDDLE, which returns information on the first pass in accordance with the #COUNT action, where the first M accesses (M is the degree of the palindrome) result in the signal FALSE, i.e., until the scanned atom is the center of the palindrome.

Statement 18 reduces the center of the palindrome. Statements 23-32 reduce the stack in accordance with Γ. If the atoms equidistant from the center do not coincide, then statement 30 outputs an error message.

Example 3. The Program ¤REPLACE (Fig. 4). MFEL is in the main intended for writing syntactic recognizers, but it can be used for writing string-processing programs. As an example of the latter use we consider context-dependent text editing. In some PL/1 programs we have to replace all the statements

PUT SKIP EDIT (⟨IDENTIFIER⟩) (A);

by statement

CALL OUT (⟨IDENTIFIER⟩);

Figure 4 shows a program for performing this substitution. It is assumed that the lexical analyzer recognizes the lexeme PUT, SKIP, EDIT, CALL, OUT, A as atoms.

The operation begins with scanning the first lexeme. The loop consisting of statements 5-7 provides for scanning the input text as far as the semicolon. If an end-of-file condition is detected, statement 9 completes the operation. If this is not so, statement 10 makes a check on the top of the stack for a configuration corresponding to the statement to be replaced. If the top of the stack coincides, a replacement is made by the configuration corresponding to the statement

CALL OUT (⟨IDENTIFIER⟩);

The semantic action #WRITE performs output of the statement corresponding to the stack configuration, which passes to the output file. It is possible to recover the statement from the stack configuration because any implementation of the Floyd – Evans language uses at least two parallel stacks [10]. We have already discussed the syntactic stack, and the second one, which is called the semantic stack, contains either the lexeme itself or else a reference to the line in the table that holds this lexeme. More details of the semantic stack are given somewhat later on. An important point here is that recovery of the initial text (apart from comments) is possible from the configuration of the atoms in the stack.

Comparison of MFEL with Gries' FEL Version

MFEL contains all the facilities that exist in Gries' version; for example, the automatic selection of one out of several semantic subroutines in calling a semantic action (CLASSNO in Gries' version) is implemented in MFEL by transferring to the semantic actions a parameter that may be a literal, an atom, or a class. The resemblance of MFEL to Gries' version is particularly clear if one compares the ¤EXPRES program (Fig. 1) with the program shown in Fig. 7.1 of [10].

On the other hand, MFEL differs in structure from the FEL of [10] and is closer to languages of ALGOL type, since MFEL allows statement nesting, and unconditional jumps are defined explicitly, along with the call statement, the ELSE part of the conditional statement, and looping statements.

The facilities of the conditional statement and of the statement for replacing the top of the stack are extended in MFEL by virtue of the filter technique (it is possible to use semantic information on comparing a standard with the top of the stack and in replacing the top of the stack).

Some of the MFEL statements are also absent from [10], which include the statement for clearing the stack, the write statement, the statement for skipping a lexeme, and the statements for operating with a queue. Also, programming of algorithms for the weak-precedence method [13] is facilitated by the conditional statement and the use of compound labels, together with an efficient compilation method (see below).

On the whole one can say that MFEL provides not only a more convenient notation but also various features that extend the applicability of the language and improve the performance of the recognizers. The class of grammars analyzed by MFEL coincides with the class of recursively denumerable grammars, whereas the class of grammars analyzed by FEL is restricted to deterministic context-free languages. However, this is attained at the expense of complicating the language and therefore the implementation.

Some Implementation Features

It is assumed that there is some algorithmic language linked to FEL in which the semantic procedures can be written when one comes to write syntactic recognizers (Assembler, Pascal, PL/1, or any other suitable language). It is therefore simpler and more convenient to translate the Floyd–Evans language not into machine code but into the language in which the semantic actions are written. This facilitates debugging the compiler and simplifies the link between the syntactic and semantic blocks.

Here we take PL/1 as that language. We have chosen PL/1 not only on account of its growing popularity and the good performance of compilers written for the ES computers, but also because it is possible to write an MFEL compiler in PL/1. This compiler is of one-pass type and uses recursive descent in parsing. The volume of the translator is about 1500 PL/1 statements.

The performance of recognizers written in MFEL is very much dependent on how efficiently the operations are performed upon the syntactic stack. In the MFEL system for the OS ES system, the syntactic stack is simulated by the array BINARY FIXED (31), which makes it possible to implement the comparison of the top of the stack with a standard without using procedure calls. The semantic stack has been implemented as an array of POINTER type, which contains pointers to the lexeme table. The latter is organized by hashing (a chaining method is used [10]).

The queues are organized by means of the store techniques of PL/1; each list element consists of an atom field, a field indicating the pointer to the lexeme in the table, and a field containing a pointer to the next link in the list. Each queue also has two pointers that define the start and end of the queue.

Composite labels are compiled into indexed labels in PL/1. The atoms are mapped into indexes by means of a hashing function constructed by means of Sprugnoli's algorithm [22].

The procedure for outputting diagnostic messages is constructed as a simple macrogenerator, which provides better messages. The method is most readily illustrated on an example: let %L denote a macrocall with a body corresponding to the preceding lexeme, while %R denotes a macrocall whose body corresponds to the current lexeme, and %P denotes a macrocall with a body AFTER %L INSTEAD OF; %O denotes a macrocall with a body FOUND %R. Then execution of the statement ER['IEO54 %P IDENTIFIER %O']; results in the output of the following diagnostic message provided that the preceding lexeme was 02 and the current lexeme is TEMPLATE:

IEO54 AFTER 02 INSTEAD OF IDENTIFIER FOUND TEMPLATE

Although the compiler is of small volume, it implements various functions that facilitate debugging MFEL programs. These include the scope for indentation in the listing of loops and the conditional statements (the NEST option), a cross-reference table forming part of the table of program elements (the ATR option), the scope for generating accesses to the MFEL debugger after certain types of statements, in particular, after statements that operate with the stack (the TRACE option), and certain others.

The compiler also provides local optimization of the program in accordance with the method of [9] (the OPT option).

* * *

The MFEL compiler has been operating since May 1977; in particular, MFEL has been used to write a syntactic recognizer for a compiler working with the Ryaod language [23] for the ES computers. The accumulated experience has shown that this modification of FEL facilitates research and use, and also raises the level of the language somewhat in the sense that an MFEL program is somewhat shorter than the equivalent FEL program.

In MFEL programming, one can use different parsing methods for the different parts of the language, which is the optimal strategy for parsing. For example, in the Ryaod compiler the technique is top downwards parsing down to the statement level and bottom upwards for the statements themselves.

As the analysis is programmed in MFEL, it is simple to add additional operations to define error types more closely or to overcome errors in the syntax analyzer. This makes it possible not only to output valuable diagnostic messages but also to improve diagnosis on the basis of experience with the translator. Here we must note that it is extremely difficult to provide high-grade diagnostics in an automatic recognizer compiler.

Programming a recognizer in MFEL requires more effort than in automatic recognizer generation from the grammar, but this is offset by the greater freedom from the need to match the analysis to generation of the object code, and also by the ease of making changes. In addition, the recognizers are fast.

MFEL also allows one to experiment with the syntax of possible constructions without altering the semantics, or vice versa. Implementation of a language can also go in parallel with improvement of the language itself, and also with definition of efficient compilation methods. A difference from encoding directly from a high-level language is that fewer decisions are required that later turn out to be difficult to alter. This makes MFEL an attractive means of designing specialized languages.

APPENDIX

Description of MFEL Syntax

This description is given to enable the reader to obtain a fuller conception of the language. In many respects it is not as accurate as it might be, since it is assumed that the reader is familiar with PL/1 and does not want to be concerned with the description of obvious and unimportant details. A modified BNF is employed in the description [10]. In this version of BNF the repetition of a string is indicated by the use of braces (null and higher repetitions are permitted). The choice of one out of several strings (factorization) is denoted by parentheses, while the strings themselves are separated by exclamation marks. Auxiliary strings are shown in square brackets. The MFEL characters that are the same as the BNF metacharacters are enclosed in double quote marks.

```
⟨identifier⟩ :: = ⟨letter apart from ⧺, @, ¤⟩{ ⟨letter⟩ ! ⟨digit⟩}
⟨simple label⟩ :: = å⟨identifier⟩
⟨action name⟩ :: = ⧺⟨identifier⟩
⟨procedure name⟩ :: ¤⟨identifier⟩
⟨class name⟩ :: = % ⟨identifier⟩ ! % {⟨character⟩} %
⟨atom⟩ :: = "⟨"⟨character⟩{ ⟨character⟩} ")"
⟨literal⟩ :: = {⟨character apart from quote⟩}
⟨comment⟩ :: = /* ⟨character⟩} */
⟨procedure⟩ :: = ⟨procedure name⟩: PROC [RECUR] [MAIN]; {⟨statement⟩} END
                ⟨procedure name⟩;
⟨label⟩ :: = ⟨simple label⟩ ! ⟨compound label⟩
⟨compound label⟩ :: = ⟨simple label⟩ ⟨atom⟩
⟨call statement⟩ :: = DCL ⟨call⟩{, ⟨call⟩}
⟨call⟩ :: = ⟨call semantic block⟩
   ! ⟨call filter block⟩
   ! ⟨call queue⟩ ! ⟨call class⟩
⟨call semantic block⟩ :: = ⟨identifier⟩ ACTION
   "(" ⟨action name⟩: ⟨integer⟩ { , ⟨action name⟩: ⟨integer⟩ } ")"
```

```
⟨call filter block⟩ :: = ⟨identifier⟩ FILTER
  "(" ⟨filter name⟩: ⟨integer⟩ {, ⟨filter name⟩: ⟨integer⟩ } ")"
⟨call queue⟩ :: = ⟨queue name⟩ QUEUE
⟨call class⟩ :: ⟨class name⟩ CLASS "(" ⟨atom⟩{ "!" ⟨atom⟩} ")"
⟨conditional statement⟩ :: = IF ⟨standard⟩ THEN ⟨statement⟩ { ⟨statement⟩}
                                             ELSE ⟨statement⟩ { ⟨statement⟩}
                                             FI;
⟨standard⟩ :: = ⟨element⟩ { ⟨element⟩ }
⟨element⟩ :: = ⟨atom⟩ ! ANY ! ⟨affix⟩ ! ⟨class name⟩
  ! ⟨context-called class⟩
⟨context-called class⟩ :: = "(" ⟨atom⟩ { "!" ⟨atom⟩ } ")"
⟨affix⟩ :: = ⟨filter name⟩ ! "(" ⟨atom⟩ & ⟨filter name⟩ ")"
⟨loop statement⟩ :: = ⟨loop head⟩ { ⟨statement⟩ } END;
⟨loop head⟩ :: = DO FOREVER; ! DO ( WHILE ! UNTIL ) ⟨standard⟩;
⟨unconditional jump statement⟩ :: = GO ⟨label⟩;
⟨conditional jump statement⟩ :: = GO ⟨label⟩ ⟨class name⟩;
⟨procedure call statement⟩ :: = ⟨procedure name⟩;
⟨return statement⟩ :: = RETURN;
⟨stop statement⟩ :: = STOP;
⟨input statement⟩ :: = SCAN [FROM ⟨queue name⟩] [TO ⟨queue name⟩];
⟨write statement⟩ :: = ADD ⟨standard⟩ [TO ⟨queue name⟩];
⟨skip statement⟩ :: = SKIP "(" ⟨integer⟩ ")" [IN ⟨queue name⟩];
  ! SKIP ⟨standard⟩ [IN ⟨queue name⟩];
⟨lexeme-replacement statement⟩ :: = { ⟨atom⟩ ! ANY ! ⟨class name⟩ ! ⟨affix⟩};
⟨stack-clear statement⟩ :: POP "(" ⟨integer⟩ ")"; ! POP ⟨standard⟩;
⟨semantic-action statement⟩ :: = ⟨action name⟩ "[" ⟨parameter⟩ "]";
⟨parameter⟩ :: = ⟨atom⟩ ! ⟨class name⟩ ! ⟨literal⟩
⟨diagnostic-message output statement⟩ :: = ER "[" ⟨literal⟩ "]";
```

LITERATURE CITED

1. A. A. Markov, "Algorithm theory," Tr. Mat. Inst., 38, Izd. Akad. Nauk SSSR, Moscow (1954).
2. W. H. Ingve, "A language for programming jobs in machine translation," in: Cybernetic Collection, Issue 6 [Russian translation], IL, Moscow (1963).
3. R. E. Criswold, J. F. Poage, and I. P. Polonsky, The Snobol 4 Programming Language, Prentice-Hall, Englewood Cliffs, New Jersey (1971).
4. A. Caracciolo de Forine, "String processing languages and generalized Markov algorithms," Symbol Manipulation Languages and Techniques, North-Holland, Amsterdam (1968).
5. R. W. Floyd, "An algorithm for coding efficient arithmetic operations," Commun. Assoc. Comput. Mach., 4, No. 1 (1961).
6. R. W. Floyd, "A descriptive language for symbol manipulation," J. Assoc. Comput. Mach., 8, No. 4 (1961).
7. A. Evans, "An ALGOL-60 compiler," Annu. Rev. Autom. Program., No. 4 (1964).
8. J. Feldman, "A formal semantic for computer languages and its application in a compiler," Commun. Assoc. Comput. Mach., 9, No. 1 (1966).
9. H. R. Haynes and L. J. Schütte, "Compilation of optimized syntactic recognizers from Floyd – Evans productions," ACM SIGPLAN Notices, 5, No. 7 (1970).
10. D. Gries, Compiler Construction for Digital Computers, Wiley, New York (1971).
11. J. Earley, "Generating a recognizer for a BNF grammar," Computational Center Report, Carnegie-Mellon Univ., Pittsburgh (1965).
12. F. L. De Remer, "Generating a recognizer for a BNF grammar," AFIPS Nat. Comput. Conf. Expo., Conf. Proc., 34 (1969).
13. J. D. Ichbian and S. P. Morse, "A technique for generating almost optimal Floyd – Evans productions for precedence grammars," Commun. Assoc. Comput. Mach., 13, No. 8 (1970).
14. D. Crowe, "Generating parsers for affix grammars," Commun. Assoc. Comput. Mach., 15, No. 8 (1972).
15. J. Katzenelson, "The Markov algorithm as a language parser: linear bounds," J. Syst. Comput. Sci., 6, No. 5 (1972).

16. J. N. Brownlee, "An ALGOL-based implementation of Snobol 4 patterns," Commun. Assoc. Comput. Mach., 20, No. 7 (1977).
17. I. V. Vel'bitskii and E. L. Yushchenko, "A metalanguage oriented for syntactic analysis and monitoring," Kibernetika, No. 2 (1970).
18. E. M. Lavrishcheva and E. L. Yushchenko, "A method of program analysis by means of the SM language," Kibernetika, No. 2 (1972).
19. I. V. Vel'bitskii, "A metalanguage for R grammars," Kibernetika, No. 3 (1973).
20. A. A. Krasilov, "A machine-command system for grammatical analysis," Programmirovanie, No. 2 (1978).
21. A. O. Slisenko, "Symmetry recognition in a predicate in a multihead Turing machine with input," Tr. Mat. Inst., Izd. Akad. Nauk SSSR, 129, Nauka, Moscow (1973).
22. R. Sprugnoli, "Perfect hashing functions: a single probe retrieving method for static sets," Commun. Assoc. Comput. Mach., 20, No. 11 (1977).
23. A. A. Stognii, E. L. Yushchenko, V. I. Voitko, E. I. Mashbits, L. V. Vernik, and N. N. Bezrukov, "A data-processing system designed for nonprofessional computer users," in: Algorithms and Economic-Job Organization, Issue 14 [in Russian], Statistika, Moscow (1979).

ANALYSIS OF WORD SIMILARITY IN SPELLING CORRECTION SYSTEMS

A. A. Sidorov

UDC 681.3.068

Automatic spelling correction in programming systems is considered. A measure of word similarity is introduced and an algorithm for computing this measure is proposed.

All current translators from artificial languages (programming languages, operating-system task-control languages, etc.) detect syntax errors and issue appropriate diagnostics. Spelling mistakes are one of the error types detected by translators. Over 80% of all spelling errors arise when one or several symbols are distorted in a word in the source program [1]. These errors often can be corrected automatically by the compiler if it is equipped with a spelling correction procedure [2]. When a spelling error is detected, the compiler calls the spelling correction procedure which compares the wrong word with a symbol table and attempts to determine (using the specific language features) whether the offending word is a distortion of one of the table words.

This error-correction scheme suggests that one of the crucial aspects of spelling correction is the development of an effective word-similarity measure. This measure should provide a numerical index for word matching, yet its calculation should not excessively tax the computer resources (memory and compilation time).

Various approaches have been tried to solve this problem. A similarity measure in the CORC language is defined as the probability that the two words match, considering the number of identical letters and the possibility of pairwise transpositions of symbols [2]. A similar approach is described in [3]. The shortcoming of this similarity measure is that it assumes known the probability of one symbol being substituted for another for all pairs of symbols. The resulting spelling correction procedure is thus dependent on the particular alphabet used.

Morgan's spelling correction procedure [1] has been incorporated in the CUPL and DPL compilers and in the Cornell-HASP operating system for the IBM-360. Morgan's measure is the number of elementary errors in the input word. Elementary errors include omission of a single symbol, distortion of a symbol, transposition of two adjoining symbols, and insertion of an extra symbol into the word. A distorted word is correctible if it contains a single elementary error. This approach will correct up to 80% of spelling errors [4]. Having narrowed down the scope of spelling correction, Morgan succeeded in creating an effective algorithm whose execution time in general is proportional to word length.

Wagner [5] expanded Morgan's method and proposed as a measure the minimum number of editing operations needed to restore the distorted line to a syntactically correct line. Wagner distinguishes between three

Translated from Programmirovanie, No. 4, pp. 65-68, July-August, 1979. Original article submitted June 6, 1978.