Right-to-Left Code Generation for Arithmetic Expressions

Dušan M. Velašević
Faculty of Electrical Engineering, University of Belgrade, Bulevar Revolucie 73, PO Box 816, 11001 Belgrade, Yugoslavia

This paper presents a method of right-to-left code generation for arithmetic expressions given in the postfix notation. The number of generated instruction lines is equal to the one obtained from the binary tree structure. However, the right-to-left code generator is faster and requires less memory than the code generator from the binary tree. These results were obtained by introducing a vector, called vector-generatrice, assigned to every postfix string. A translation grammar which generates the postfix string and its associated vector-generatrice is defined. An experiment has been performed over a set of arithmetic expressions to compare the right-to-left code generator and the code generator from the binary tree.

INTRODUCTION

In this paper, we shall concentrate on the design of an efficient code generator which produces an assembly language code for arithmetic expressions. The input to the code generator is an arithmetic expression. The output is an equivalent assembly language program. We would like the resulting assembly language program to be good under some cost function such as the number of assembly language instructions, or the number of memory fetches. The cost function of the code generator depends only on the internal representation of expressions in compilers. The well-known internal structures are parenthesized notation, quadruples, triples, postfix and prefix notation, and the binary tree. The parenthesized form of expressions was used in the first FORTRAN compilers.[1] This is not a suitable internal structure for an efficient code generation. In particular, the transformation to such a structure is not compatible with the design of modern compilers, as far as a single processor is concerned. Stone gives the one-pass algorithm which transforms the expressions into a parenthesized form for parallel processors.[2] In quadruple notation, the operations appear in the order in which they are to be executed. The main problem is to keep track of the contents of the accumulator, in which all arithmetic is done at run-time, in order to eliminate unnecessary load and store instructions.[3] The main disadvantage of quadruples is that a description of each temporary value is maintained throughout the compile-time. This problem does not exist in the triple notation first introduced by Sheridan.[4] We do not need a description of each temporary when using triples; it need only be maintained while code which references it is being generated. If the ranges of temporaries are pairwise disjoint or nested, we can use a compile-time stack to hold descriptions of temporaries; otherwise, a more complex scheme will be necessary to allocate and delete space for them. If the triples are not going to be processed once the code has been generated, we can replace the triple by a description of its result. Consequently, we would not need a stack, and we would not have to delete descriptions. Obviously, the handling of temporaries in triple notation is more complex than in the quadruple notation. Thus, the code generator from the quadruple notation is more efficient. On the other hand, the memory space for triples is smaller since they require three fields per entry. The triples are more suitable for the code optimization since they can easily be rearranged for this purpose. Furthermore, the code generated from the triple notation generally uses fewer temporaries.

A number of languages are based on a concept known as a Polish notation; this has advantages for machine but it is difficult for human digestion. It is so called because it was first introduced by the Polish philosopher Jan Lukasiewicz in connection with the formulae of symbolic logic. A variation of it is called the reverse Polish notation. These notations are also known as the prefix and postfix notation, respectively.[1] The transformation to prefix notation, and the code generation from it, requires the analysis of expressions from right to left. This differs from the previously mentioned methods where the analysis is performed from left to right. In postfix notation, the analysis is also performed from left to right. Formally, the difference between postfix and prefix notation lies in the operator structure; reverse Polish notation is based on postfix operators—⟨operand⟩ ⟨operand⟩ ⟨operator⟩—while direct Polish notation is based in a similar way on prefix operators—⟨operator⟩ ⟨operand⟩ ⟨operand⟩. Since the writing and interpretation of expressions are usually performed from left to right, the prefix notation is inappropriate for the code generation (with the exception of APL, for example). Compared to the triple notation, the postfix notation requires less space and generates fewer temporaries; the code generation from it is very simple.[3,5] However, the postfix notation is not a structure particularly suitable for code optimization either at the single expression level or at the level of a group of expressions.

Code generation from a binary tree is probably the most discussed method in this field. Anderson has described an algorithm for the code generation for a one-register machine.[6] Nakata proposed a similar algorithm.[7] The number of registers required to generate code for an expression tree has been investigated by Nakata, Redziejowski, and Sethi and Ullman.[7–9] Beatty[10] and Frailey[11] discuss extensions involving the unary minus operator. The code generated from the binary tree is the shortest-length program to compute a given expression.

CCC-0010-4620/82/0025-0316 $05.50

This was shown by Aho and Ullman.[12] Besides the shortest-length program, the binary tree uses fewer temporaries and offers a more suitable structure for the straight-line code optimization than the postfix notation. On the other hand, the postfix notation requires less space and is more efficient from the transformation and code generation point of view.

There are other works dealing with the problem of register allocation in a sequence of expression evaluations (e.g. Refs 13–15), translating arithmetic expressions into code for parallel computers (e.g. Refs 2, 16–19), or basic algorithms for the code generation.[20–23]

This paper presents a new algorithm for the code generation without optimization. The work was motivated by the results of the investigation of the distribution of executable statements,[24] and by the examination of works concerning the code optimization. It was found that the assignment and IF statements together have over 60% of static distribution.[24] Furthermore, the existing internal structures of expressions are not particularly suitable for the code optimization and register allocation at the single expression level or at the level of a group of expressions. With respect to other algorithms in this field, the proposed algorithm gives faster code generation than the binary tree but for the same length of the code. Nevertheless, the internal structure of expressions is suitable for the code optimization and register allocation. It offers an opportunity to investigate the efficient code optimization techniques.

The internal structure of an expression is composed of two substructures: (1) well-known postfix notation, and (2) vector-generatrice (VG), describing the postfix notation in a certain way. Its generation is discussed through a translation grammar for arithmetic expressions. Although the transformation to such an internal structure is performed from left to right, the code is generated from right to left. The algorithm is presented in Algol-like form and discussed. It is proved that the algorithm produces the code of the same length as the binary tree. All considerations are based on a single accumulator machine.

Theoretical results are illustrated by a worked and output example. The output example is extracted from an experiment done for a set of arithmetic expressions to examine the characteristics of the algorithm and compare them to that obtained with the binary tree. It was found that the proposed algorithm is about 20% faster. When the transformation to internal structure is taken into account, this time gain reduces to 13%. The required memory space and number of instructions necessary to realize the transformation and code generation are approximately the same.

VECTOR-GENERATRICE

In the generation of VG-based data structure we shall use the concept of the translation grammar.[25] A translation grammar is a context-free grammar in which the set of terminal symbols is partitioned into a set of input symbols and a set of action symbols. The action symbols are used to represent more general *ad hoc* routines. The strings in the language specified by a translation grammar are called activity sequences.

Let a translation grammar G^T be given generally by

$$G^T = (IS, AS, NS, P, SN) \quad (1)$$

where IS = input symbols, AS = action symbols, NS = nonterminal symbols, P = productions and SN = starting nonterminal.

The following metanotation is used for G^T:

$\langle, \rangle$ = matched pair of angle brackets enclosing a nonterminal,
$\rightarrow$ = production operator,
$\{, \}$ = matched pair of curly brackets enclosing an action symbol

An example of a translation grammar is:

IS = a, b, c
AS = x, y, z
NS = $\langle A\rangle, \langle B\rangle$
SN = $\langle A\rangle$
(1) P = $\langle A\rangle \rightarrow a\langle A\rangle\{x\}\langle B\rangle$
(2) $\langle A\rangle \rightarrow \{z\}$
(3) $\langle B\rangle \rightarrow \langle B\rangle c$
(4) $\langle B\rangle \rightarrow b\{y\}$

For the input sequence *abc*, the activity sequence is:

$$a\{z\}\{x\}b\{y\}c.$$

It is obtained by the following parsing:

(1) $\langle A\rangle \rightarrow a\langle A\rangle\{x\}\langle B\rangle$
(2) $\langle A\rangle \rightarrow a\{z\}\{x\}\langle B\rangle$
(3) $\langle A\rangle \rightarrow a\{z\}\{x\}\langle B\rangle c$
(4) $\langle A\rangle \rightarrow a\{z\}\{x\}b\{y\}c$

If each action symbol represents a routine which outputs the symbol within the curly brackets (and possibly performs anything else), then a given translation grammar is intended to describe the translation of input strings into output strings. For the given example, the output string is *zxy*. It is readily derived from the obtained activity sequence. This concept of the translation grammar will be used in the derivation of VG-based data structure.

As the first step toward this goal, we devise a simplified translation grammar for arithmetic expressions:

IS = $I, N, +, -, *, /, \$$
AS = I, N,, $+$, $-$, #, $*$, $/$, $\$$ (,, denotes the empty symbol)
NS = $\langle S\rangle$ = (start expression)
$\langle E\rangle$ = (expression)
$\langle T\rangle$ = (term)
$\langle P\rangle$ = (primary)
$\langle V\rangle$ = (variable)
$\langle N\rangle$ = (unsigned number)
SN = $\langle S\rangle$

(1) P = $\langle S\rangle \rightarrow \{\}\langle E\rangle\$\{\$\}$
(2) $\langle E\rangle \rightarrow \langle E\rangle + \langle T\rangle\{+\}$
(3) $\langle E\rangle \rightarrow \langle E\rangle - \langle T\rangle\{-\}$
(4) $\langle E\rangle \rightarrow \langle T\rangle$
(5) $\langle T\rangle \rightarrow \langle T\rangle * \langle P\rangle\{*\}$
(6) $\langle T\rangle \rightarrow \langle T\rangle/\langle P\rangle\{/\}$
(7) $\langle T\rangle \rightarrow \langle P\rangle$ (2)
(8) $\langle P\rangle \rightarrow +\langle P\rangle$
(9) $\langle P\rangle \rightarrow -\langle P\rangle\{\#\}$
(10) $\langle P\rangle \rightarrow \langle V\rangle\{I\}$
(11) $\langle P\rangle \rightarrow \langle N\rangle\{N\}$
(12) $\langle P\rangle \rightarrow (\langle E\rangle)$
(13) $\langle V\rangle \rightarrow I$
(14) $\langle N\rangle \rightarrow N$

It should be noted that the unary operators + and # (unary minus) are given the highest precedence. Each operator can be followed by a string of unary operators. The input sequences are restricted to symbols I (representing a variable), N (representing an unsigned number), (,) and +, −, *, / only for the sake of the clear and concise presentation of the algorithm. The function designator and the operator ↑ could be readily included in the grammar. Their presence or absence does not influence the final conclusion on the algorithm characteristics.

If we assign to each action symbol an action routine for performing special functions, then the application of the translation grammar given by Eqn (2) to an input sequence will produce a standard postfix string with associated VG. There is a very simple practical explanation of VG. Let E_i be an arithmetic expression in the infix notation, transformed to postfix string E_p:

$$E_p = a_1 a_2 \ldots a_m b_1 b_2 \ldots b_j a_{m+1} a_{m+2} \ldots a_{m+i} b_{j+1} b_{j+2} \ldots \ldots b_{j+k} \ldots a_{m+i+n} a_{m+i+n+1} \ldots \ldots a_{m+i+M} b_{j+k+q} b_{j+k+q+1} \ldots b_{j+k+N} a_{m+i+M+1} \quad (3)$$

where $a_1, a_2, a_3, \ldots a_m, \ldots a_{m+i+M+1}$ are the operands (or their addresses), and $b_1, b_2, b_3, \ldots b_j, \ldots b_{j+k+N}$ the operators (or their indices in delimiter table). The dollar sign \$ denotes the end of E_p, i.e.

$$a_{m+i+M+1} = \$ \quad (4)$$

Using Eqn (3), VG can be generated in the following way. Firstly, we define the substrings of E_p. A substring is a sequence of all successive operands or operators. If the number of operand substrings is N_1, the number of operator substrings N_2 must be

$$N_2 = N_1 \quad (5)$$

The substrings of E_p in Eqn (3) are:

$a_1 a_2 \ldots a_m$	(operands)
$b_1 b_2 \ldots b_j$	(operators)
$a_{m+1} a_{m+2} \ldots a_{m+i}$	(operands)
$b_{j+1} b_{j+2} \ldots b_{j+k}$	(operators)
⋮	
$a_{m+i+n} a_{m+i+n+1} \ldots a_{m+i+M}$	(operators)
$b_{j+k+q} b_{j+k+q+1} \ldots b_{j+k+N}$	(operators)
$a_{m+i+M+1}$	(operand)

The total number of substrings N_3 (including the endmarker as an operand substring) is

$$N_3 = 2N_1 + 1 \quad (6)$$

By assigning to each substring of operands a successor substring of operators, one obtains a pair of connected substrings CS_l, where l is its ordinal number.

For example, $a_1 a_2 \ldots a_m b_1 b_2 \ldots b_j$ forms a pair of connected substrings CS_1. $a_1 a_2 \ldots a_m$ represents a substring of operands followed by a successor substring of operators $b_1 b_2 \ldots b_j$. The number of pairs N_{CS} is

$$N_{CS} = \lfloor N_3/2 = N_1 \quad (7)$$

For each pair of connected substrings we create a 3-tuple (N_l^O, N_l^B, P_l), where N_l^O = number of operands in CS_l, N_l^B = number of operators in CS_l, P_l = pointer to beginning of CS_l in E_p. For CS_1, the 3-tuple is $(m, j, 1)$ because $N_1^O = m$, $N_1^B = j$ and $P_1 = 1$. VG is formed now by concatenating all 3-tuples of connected substrings by their ordinal numbers into a vector of dimension $(N_{CS} + 1) \times 3$:

$$VG_1 \mathbf{cat} VG_2 \mathbf{cat} \ldots \mathbf{cat} VG_{N_{CS}+1} \quad (8)$$

where **cat** is the concatenation operator. The endmarker is considered as an operand substring connected to a dummy operator substring. Its corresponding 3-tuple is given by $(1, 0, P_{N_{CS}+1})$.

ACTION ROUTINES

In generating VG-based data structure, the action routines play the principal role. In order to describe them, we associate with each action symbol a mnemonic name representing a routine intended to perform a particular function:

Action symbol	Name of routine
{}	NULL
{*I*}	VARIABLE
{*N*}	UNSIGNED_NUMBER
{+}	PLUS
{−}	MINUS
{#}	UNARY_MINUS
{*}	MULTIPLY
{/}	DIVIDE
{$}	END_MARKER

We shall proceed now with the presentation of these routines as Algol procedures.

```
NULL
  procedure NULL (VG, current_VG_index, current_index_of_output_string);
  integer array VG;
  integer current_VG_index, current_index_of_output_string;
    begin integer i, n;
    comment: because the empty symbol is to be outputted, no output
             operation is performed;
    current_VG_index := 1;
    current_index_of_output_string := 1;
    comment: reset VG array;
    for i := 1 step 1 until n do
    VG[i] := 0;
    VG[3] := 1;
    end NULL;
```

The routine *NULL* resets the space allocated to *VG* (defined as a vector), and initializes the indices of the *VG* and output string array (E_p).

```
VARIABLE
  procedure VARIABLE (VG, current_VG_index, class_of_previously_outputted_
                      _symbol, output_string, current_index_of_output_
                      _string);
  integer array VG; string array output_string; integer current_VG_
                                                index, class_of_
                                                _previously_
                                                outputted_symbol,
                                                current_index_of_
                                                output_string;
    begin
      if current_VG_index ≠ 1 ∧ class_of_previously_outputted_
        symbol ≠ 0
      then
        begin
          VG[current_VG_index + 5] := VG[current_VG_index] +
                                    + VG[current_VG_index + 1] +
                                    + VG[current_VG_index + 2];
          current_VG_index := current_VG_index + 3;
        end;
        comment: output the action symbol I into the output string array;
        output_string[current_index_of_output_string] = 'I';
        current_index_of_output_string := current_index_of_output_
                                          _string + 1;
        comment: count the number of outputted symbols of the
                 class 0;
        VG[current_VG_index] := VG[current_VG_index] + 1;
        class_of_previously_outputted_symbol := 0;
    end VARIABLE;
```

The routine *VARIABLE* performs three functions: (1) compares the class of the previously outputted symbol to the current one (*I* or *N*) for *current_VG_index* ≠ 1; the class of outputted symbol is *class_of* (*I*) = 0, *class_of* (*N*) = 0, *class_of* (*other action symbols*) = 1; (2) outputs the action symbol *I* into the output string array (defined as a vector); (3) counts the number of outputted symbols of the class 0.

UNSIGNED_NUMBER

The body of this procedure is identical to *VARIABLE* except for the statement

output_string [*current_index_of_output_string*] := '*I*';

which has to be replaced by

output_string [*current_index_of_output_string*] := '*N*';

```
PLUS
  procedure PLUS (VG, current_VG_index, output_string, current_index_
                  _of_output_string, class_of_previously_outputted_
                  _symbol);
  integer array VG; string array output_string; integer current_VG_
                                                _index, current_
                                                _index_of_output_
                                                _string, class_
                                                _of_previously_
                                                _outputted_
                                                _symbol;
    begin
        comment: output the action symbol + into the output string array;
        output_string [current_index_of_output_string] := '+';
        current_index_of_output_string := current_index_of_output_string + 1;
        comment: count the number of outputted symbols of the
                 class 1;
        VG [current_VG_index + 1] := VG [current_VG_index + 1] + 1;
        class_of_previously_outputted_symbol := 1;
    end PLUS;
```

The routine *PLUS* performs two functions: (1) outputs the action symbol + into the output string array, and (2) counts the number of outputted symbol of the class 1.

MINUS

The body of this procedure is identical to *PLUS* except for the statement

output_string [*current_index_of_output_string*] := '+';

which has to be replaced by

output_string [*current_index_of_output_string*] := '−';.

UNARY MINUS

The body of this procedure is identical to *PLUS* except for the statement

output_string [*current_index_of_output_string*] := '+';

which has to be replaced by

output_string [*current_index_of_output_string*) := '#';.

MULTIPLY

The body of this procedure is identical to *PLUS* except for the statement

output_string [*current_index_of_output_string*] := '+';

which has to be replaced by

output_string [*current_index_of_output_string*] := '*';.

DIVIDE

The body of this procedure is identical to *PLUS* except for the statement

output_string [*current_index_of_output_string*] := '+';

which has to be replaced by

output_string [*current_index_of_output_string*] := '/';.

```
END MARKER
  procedure END_MARKER (VG, current_VG_index, output_string,
                        current_index_of_output_string);
  integer array VG; string array output_string; integer current_
                                                _index_of_
                                                _output_string,
                                                current_VG_
                                                _index;
    begin
        comment: output the action symbol $ into the output
                 string array;
        output_string [current_index_of_output_string] := '$';
        comment: terminate VG by a special three words marker;
        current_VG_index := current_VG_index + 3;
        VG [current_VG_index] := 1;
        VG [current_VG_index + 1] := 0;
        VG [current_VG_index + 2] := VG [current_VG_index − 1] +
                                   + VG [current_VG_index − 2] +
                                   + VG [current_VG_index − 3];
    end END_MARKER;
```

It is obvious that the procedures *VARIABLE*, *UNSIGNED_NUMBER*, and the procedures *PLUS*, *MINUS*, *UNARY_MINUS*, *MULTIPLY*, *DIVIDE* can be replaced (in the domain of realization) by two procedures: (1) *OPERAND*, and (2) *OPERATOR*. This is obtained easily by including the output symbol in the list of formal parameters.

```
OPERAND
  procedure OPERAND (VG, current_VG_index, class_of_previously_
                     _outputted_symbol, output_string, current_
                     _index_of_output_string, output_symbol);
  integer array VG; string array output_string; integer current_VG_
                                                _index, class_
                                                _of_previously_
                                                _outputted_
                                                _symbol, current_
                                                _index_of_
                                                _output_string;
  string (1) output_symbol;
    begin
        if current_VG_index ≠ 1 ∧ class_of_previously_outputted_
                                  _symbol ≠ 0
        then
            begin
                VG [current_VG_index + 5] := VG [current_VG_index] +
                                           + VG [current_VG_index + 1] +
                                           + VG [current_VG_index + 2];
                current_VG_index := current_VG_index + 3;
            end;
            comment: output the action symbol (output_symbol) into
                     the output string array;
            output_string [current_index_of_output_string] := output_
                                                              _symbol;
            current_index_of_output_string := current_index_of_
                                              _output_string + 1;
            comment: count the number of outputted symbols of the
                     class 0;
            VG [current_VG_index] := VG [current_VG_index] + 1;
            class_of_previously_outputted_symbol := 0;
    end OPERAND;
```

The procedure *OPERAND* replaces the procedures *VARIABLE* and *UNSIGNED_NUMBER*.

```
OPERATOR
  procedure OPERATOR (VG, current_VG_index, output_string, current_
                      _index_of_output_string, output_symbol,
                      class_of_previously_outputted_symbol);
  integer array VG; string array output_string; integer current_VG_
                                                _index, current_
                                                _index_of_output_
                                                _string, class_
                                                _of_previously_
                                                _outputted_symbol;
```

```
string (1) output_symbol;
  begin
    comment: output the action symbol (output_symbol) into
             the output string array;
             output_string [current_index_of_output_string] :=
                          output_symbol;
             current_index_of_output_string := current_index_of_
                                              _output_string + 1;
    comment: count the number of outputted symbols of the class 1;
    VG [current_VG_index + 1] := VG [current_VG_index + 1] + 1;
    class_of_previously_outputted_symbol := 1;
  end OPERATOR;
```

The procedure *OPERATOR* replaces the procedures *PLUS*, *MINUS*, *UNARY_MINUS*, *MULTIPLY* and *DIVIDE*.

As an illustration, consider the following arithmetic expression

$$I + N - (- (- (N + I))/(I * I - N/I))/I * I + I\$ \quad (9)$$

whose VG-based data structure is to be found. The parsing sequence is:

(1) $\{\}\langle E\rangle\,\$\,\{\$\}$
(2) $\{\}\langle E\rangle + \langle T\rangle\{+\}\,\$\,\{\$\}$
(3) $\{\}\langle E\rangle - \langle T\rangle\{-\} + \langle T\rangle\{+\}\,\$\,\{\$\}$
(2) $\{\}\langle E\rangle + \langle T\rangle\{+\} - \langle T\rangle\{-\} + \langle T\rangle\{+\}\,\$\,\{\$\}$
(5) $\{\}\langle E\rangle + \langle T\rangle\{+\} - \langle T\rangle * \langle P\rangle\{*\}\{-\} + \langle T\rangle\{+\}\,\$\,\{\$\}$
(6) $\{\}\langle E\rangle + \langle T\rangle\{+\} - \langle T\rangle/\langle P\rangle\{/\} * \langle P\rangle\{*\}\{-\} + \langle T\rangle\{+\}\,\$\,\{\$\}$
(7) $\{\}\langle E\rangle + \langle T\rangle\{+\} - \langle P\rangle/\langle P\rangle\{/\} * \langle P\rangle\{*\}\{-\} + \langle T\rangle\{+\}\,\$\,\{\$\}$
(12) $\{\}\langle E\rangle + \langle T\rangle\{+\} - (\langle E\rangle)/\langle P\rangle\{/\} * \langle P\rangle\{*\}\{-\} + \langle T\rangle\{+\}\,\$\,\{\$\}$
(4) $\{\}\langle E\rangle + \langle T\rangle\{+\} - (\langle T\rangle)/\langle P\rangle\{/\} * \langle P\rangle\{*\}\{-\} + \langle T\rangle\{+\}\,\$\,\{\$\}$
(6) $\{\}\langle E\rangle + \langle \mathrm{T}\rangle\{+\} - (\langle T\rangle/\langle P\rangle\{/\})/\langle P\rangle\{/\} * \langle P\rangle\{*\}\{-\} + \langle T\rangle\{+\}\,\$\,\{\$\}$
(7) $\{\}\langle E\rangle + \langle T\rangle\{+\} - (\langle P\rangle/\langle P\rangle\{/\})/\langle P\rangle\{/\} * \langle P\rangle\{*\}\{-\} + \langle T\rangle\{+\}\,\$\,\{\$\}$
(12) $\{\}\langle E\rangle + \langle T\rangle\{+\} - (\langle P\rangle/(\langle E\rangle)\{/\}/\langle P\rangle\{/\} * \langle P\rangle\{*\}\{-\} + \langle T\rangle\{+\}\,\$\,\{\$\}$
(3) $\{\}\langle E\rangle + \langle T\rangle\{+\} - (\langle P\rangle/(\langle E\rangle - \langle T\rangle\{-\})\{/\}/\langle P\rangle\{/\} * \langle P\rangle\{*\}\{-\} + \langle T\rangle\{+\}\,\$\,\{\$\}$
(4) $\{\}\langle E\rangle + \langle T\rangle\{+\} - (\langle P\rangle/(\langle T\rangle - \langle T\rangle\{-\})\{/\})/\langle P\rangle\{/\} * \langle P\rangle\{*\}\{-\} + \langle T\rangle\{+\}\,\$\,\{\$\}$
(5) $\{\}\langle E\rangle + \langle T\rangle\{+\} - (\langle P\rangle/(\langle T\rangle * \langle P\rangle\{*\} - \langle T\rangle\{-\})\{/\})/\langle P\rangle\{/\} * \langle P\rangle\{*\}\{-\} + \langle T\rangle\{+\}\,\$\,\{\$\}$
(6) $\{\}\langle E\rangle + \langle T\rangle\{+\} - (\langle P\rangle/(\langle T\rangle * \langle P\rangle\{*\} - \langle T\rangle/\langle P\rangle\{/\}\{-\})\{/\})/\langle P\rangle\{/\} * \langle P\rangle\{*\}\{-\} + \langle T\rangle\{+\}\,\$\,\{\$\}$
(9) $\{\}\langle E\rangle + \langle T\rangle\{+\} - (- \langle P\rangle\{\#\}/(\langle T\rangle * \langle P\rangle\{*\} - \langle T\rangle/\langle P\rangle\{/\}\{-\})\{/\})/\langle P\rangle\{/\} * \langle P\rangle\{*\}\{-\} + \langle T\rangle\{+\}\,\$\,\{\$\}$
(12) $\{\}\langle E\rangle + \langle T\rangle\{+\} - (- (\langle E\rangle)\{\#\}/(\langle T\rangle * \langle P\rangle\{*\} - \langle T\rangle/\langle P\rangle\{/\}\{-\})\{/\})/\langle P\rangle\{/\} * \langle P\rangle\{*\}\{-\} + \langle T\rangle\{+\}\,\$\,\{\$\}$
(4) $\{\}\langle E\rangle + \langle T\rangle\{+\} - (- (\langle T\rangle)\{\#\}/(\langle T\rangle * \langle P\rangle\{*\} - \langle T\rangle/\langle P\rangle\{/\}\{-\})\{/\})/\langle P\rangle\{/\} * \langle P\rangle\{*\}\{-\} + \langle T\rangle\{+\}\,\$\,\{\$\}$
(7,9) $\{\}\langle E\rangle + \langle T\rangle\{+\} - (- (- \langle P\rangle\{\#\})\{\#\}/(\langle T\rangle * \langle P\rangle\{*\} - \langle T\rangle/\langle P\rangle\{/\}\{-\})\{/\})/\langle P\rangle\{/\} * \langle P\rangle\{*\}\{-\} + \langle T\rangle\{+\}\,\$\,\{\$\}$
(12) $\{\}\langle E\rangle + \langle T\rangle\{+\} - (- (- (\langle E\rangle)\{\#\})\{\#\}/(\langle T\rangle * \langle P\rangle\{*\} - \langle T\rangle/\langle P\rangle\{/\}\{-\})\{/\})/\langle P\rangle\{/\} * \langle P\rangle\{*\}\{-\} + \langle T\rangle\{+\}\,\$\,\{\$\}$
(2) $\{\}\langle E\rangle + \langle T\rangle\{+\} - (- (- (\langle E\rangle + \langle T\rangle\{+\})\{\#\})\{\#\}/(\langle T\rangle * \langle P\rangle\{*\} - \langle T\rangle/\langle P\rangle\{/\}\{-\})\{/\})/\langle P\rangle\{/\} * \langle P\rangle\{*\}\{-\} + \langle T\rangle\{+\}\,\$\,\{\$\}.$

Applying now to each nonterminal $\langle E\rangle$ the production sequence (4), (7), (10), (11), (13) or (14) (depending on the operand type), to each nonterminal $\langle T\rangle$ the production sequence (7), (10), (11), (13) or (14) (depending on the operand type), and to each nonterminal $\langle P\rangle$ the production sequence (10), (11), (13) or (14) (depending on the operand type), one obtains the following activity sequence:

$\{\}I\{I\} + N\{N\}\{+\} - (- (- (N\{N\} + I\{I\}\{+\})\{\#\})\{\#\}/(I\{I\} * I\{I\}\{*\} - N\{N\}/I\{I\}\{/\}\{-\})\{/\})/I\{I\}\{/\} * I\{I\}\{*\}\{-\} + I\{I\}\{+\}\,\$\,\{\$\}.$

If we extract from the activity sequence all input symbols by the order of their appearance, we obtain Eqn (9). The application of the action routines for each action symbol by the order of its appearance in the activity sequence, gives, finally, VG-based data structure (see Table 1).

Each underlined substring in the output_string (E_p), in Table 1, corresponds to one connected substring; i.e. + # #. Two substrings form CS_l if they are continuous in E_p, and the first substring consists of operands; i.e., $CS_2 = NI + \#\ \#$, 3-tuple is (2, 3, 4). Each underlined 3-tuple in VG corresponds to a CS_l in E_p by the order of its appearance.

RIGHT-TO-LEFT CODE GENERATION

The term *right-to-left code generation* designates the code generation from the right end of E_p. One of the key points in introducing the vector-generatrice is just to enable the right-to-left code generation. The algorithm is based on the following principles: (i) the elementary unit for the code generation is a pair of connected substrings CS_l; (ii) the code generation from an elementary unit is the same as for any standard postfix string; (iii) the code is generated until $N_l^O = 1 \vee N_l^B = 0$; (iv) when the current CS_l becomes temporary inactive, i.e. $N_l^O = 1 \vee N_l^B = 0$, the next $CS_{l'}$ is taken, where $l' = l + 1 \vee l' = l' - 1$ respectively; (v) the separate stack for operands is not needed; instead, the space for E_p itself is used for N_{CS} stacks, each preset already with an operand substring in the phase of transformation to E_p, and (vi) the transition from one CS_l to another $CS_{l'}$ is performed by means of $P_{l'}$ given in a 3-tuple for $CS_{l'}$.

Before we proceed to detailed presentation of algorithm, we shall explain the procedure for the code generation from an elementary unit CS_l.

Firstly, we shall define a hypothetical machine, necessary for this purpose, as a single accumulator (ACC) machine with the following basic operations:

- LDA A; move the contents of the location A into ACC,
- STA A; move the contents of ACC into the location A,
- NEG; the contents of ACC is two's complemented,
- ADD A; the contents of the location A are added to the contents of ACC and the result placed in ACC,
- SUB A; the contents of the location A are subtracted from the contents of ACC and the result placed in ACC,
- MUL A; the contents of ACC are multiplied by the

Table 1. Generation of VG-based data structure for expression I+N−(−(−(N+I))/(I*I−N/I))/I*I+I$

ENTRY	ACTIVITY	OUTPUT_STRING	Current_ _index of_ _output_ _string	VG	Class_of_ _previously_ outputted_ _symbol	
{}	NULL		1	00100	0	—
{I}	VARIABLE	I	2	10100	0	0
{N}	UNSIGNED_NUMBER	IN	3	20100	0	0
{+}	PLUS	IN+	4	21100	0	1
{N}	UNSIGNED_NUMBER	IN+N	5	2111040	0	0
{I}	VARIABLE	IN+NI	6	2112040	0	0
{+}	PLUS	IN+NI+	7	2112140	0	1
{#}	UNARY_MINUS	IN+NI+#	8	2112240	0	1
{#}	UNARY_MINUS	IN+NI+##	9	2112340	0	1
{I}	VARIABLE	IN+NI+##I	10	211234109	0	0
{I}	VARIABLE	IN+NI+##II	11	211234209	0	0
{*}	MULTIPLY	IN+NI+##II*	12	211234219	0	1
{N}	UNSIGNED_NUMBER	IN+NI+##II*N	13	2112342191012	0	0
{I}	VARIABLE	IN+NI+##II*NI	14	2112342192012	0	0
{/}	DIVIDE	IN+NI+##II*NI/	15	2112342192112	0	1
{−}	MINUS	IN+NI+##II*NI/−	16	2112342192212	0	1
{/}	DIVIDE	IN+NI+##II*NI/−/	17	2112342192312	0	1
{I}	VARIABLE	IN+NI+##II*NI/−/I	18	21123421923121017	0	0
{/}	DIVIDE	IN+NI+##II*NI/−/I/	19	21123421923121117	0	1
{I}	VARIABLE	IN+NI+##II*NI/−/I/I	20	211234219231211171019	0	0
{*}	MULTIPLY	IN+NI+##II*NI/−/I/I*	21	211234219231211171119	0	1
{−}	MINUS	IN+NI+##II*NI/−/I/I*−	22	211234219231211171219	0	1
{I}	VARIABLE	IN+NI+##II*NI/−/I/I*−I	23	2112342192312111712191022	0	0
{+}	PLUS	IN+NI+##II*NI/−/I/I*−I+	24	2112342192312111712191122		1
{$}	END_MARKER	IN+NI+##II*NI/−/I/I*−I+$		2112342192312111712191122 1024		

contents of the location A and the result placed in ACC, and

DIV A; the contents of ACC are divided by the contents of the location A and the result placed in ACC.

Let CS_l be given by

$$CS_l = a_1^l a_2^l a_3^l \ldots a_m^l b_1^l b_2^l b_3^l \ldots b_n^l \qquad (10)$$

The space occupied by the operand substring $a_1^l a_2^l a_3^l \ldots a_m^l$ has to be considered as a stack preset with $a_1^l a_2^l a_3^l \ldots a_m^l$. Starting from the top element of the stack, T.OPD(a_m), and the first operator in the connected operator substring (b_1^l), we generate the following assembly code:

```
LDA T.OPD
NEG
```

if b_1^l is the unary operator (#), or

```
LDA BT.OPD
OPT T.OPD
```

if b_1^l is binary operator (+, −, *, /), where BT.OPD designates generally the below-the-top element of the stack, and OPT is a symbolic representation of operators (ADD, SUB, MUL, DIV, NEG).

If any of succeeding operators is unary, an NEG is generated. Otherwise, the operator is checked for the commutativity. If it is a comutative one, an instruction line

```
OPT T.OPD
```

is generated. The noncommutative operator requires the generation of the code

```
STA T
```

with

T.OPD ← 'T'

and then

```
LDA BT.OPD
OPT T.OPD
```

where T is a temporary variable. The code generation is continued in this way until $N_l^O = 1 \vee N_l^B = 0$. Then, the next $CS_{l+1 \vee l-1}$ is taken.

```
procedure code generator (EP, VG, IL, NCS);
comment: The procedure code generator generates the assembly code for
         arithmetic expressions starting from the right end of the postfix
         string with an associated vector-generatrice.
         EP = postfix string,
         VG = vector-generatrice,
         IL = generated instruction lines,
         NCS = number of elementary units,
         NlO = number of operands in the current elementary unit,
         NlB = number of operators in the current elementary unit,
         CSl = current elementary unit,
         CSlPLUSONE = elementary unit right to the current one,
         CSlMINUSONE = elementary unit left to the current one,
         l = pointer to the beginning of a 3-tuple in VG,
         lc = pointer to the leftmost 3-tuple currently handled,
         OPT = symbolic representation of operators (ADD, SUB, MUL, DIV,
               NEG),
         OPT1 = standard representation of operator (+, −, *, /, #) in the
                operator substring,
         T.OPD = top element of the stack for CSl,
         BT.OPD = below-the-top element of the stack for CSl,
         T = symbolic representation of temporary variable;
begin
  lc := (NCS − 1) * 3 + 1;
  NlO := VG[lc];
  NlB := VG[lc + 1];
  if NlB = 0 then begin
                    generate_code_line LDA_T.OPD; goto exit;
                  end
  else L1: l := lc; if OPT1 = '#' then begin
                                         generate_code_lines_LDA_
                                         T.OPD_and_NEG; goto L5;
                                       end
  else if NlO = 1 then
       L2: begin
             comment: terminate code generation from
                      an elementary unit CSl and access
                      the next one CSlMINUSONE;
                      lc := lc − 3; NlO := VG[lc]; NlB := VG[lc + 1];
                      goto L1;
           end
```

```
else
  L3: begin
        comment: BT.OPD is located on the stack belonging to CSl
                 pointed to by lc;
        generate_code_line_LDA_BT.OPD;
        comment: OPT is taken from CSl pointed to by l, and T.OPD
                 from CSl pointed to by lc;
      L4: generate_code_line_OPT_T.OPD;
        comment: lc is a pointer to NlO;
          NlO := NlO - 1;
        comment: l + 1 points to NlB;
      L5: NlB := NlB - 1;
      end
        comment: l + 1 points to NlB;
        if NlB = 0 then goto L6
        else if OPT1 = '≠' then begin
                                  generate_code_line_NEG; goto L5;
                                end
else
      comment: lc is a pointer to NlO;
      if NlO = 1 then begin
      comment: lc is a pointer to NlO;
                                generate_code_line_STA_T; NlO := 0;
      comment: T.OPD is located on the stack belonging to CSl
               pointed to by l; T.OPD := 'T';
                                goto L2;
                    end
else if OPT1 = '+' ∨ '*' then goto L4
else begin
      generate_code_line_STA_T;
      comment: T.OPD is located on the stack belonging to CSl
               pointed to by l; T.OPD := 'T'; goto L3;
    end
L6: begin
      comment: terminate code generation from an elementary
               unit CSl and access the next one CSlPLUSONE;
               l := l + 3; NlO := VG[l]; NlB := VG[l + 1];
      comment: l + 1 points to NlB and l to NlO;
    if NlB ≠ 0 then begin
      comment: OPT or NEG and T.OPD are taken from CSl pointed
               by l;
                  generate_code_line_OPT_T.OPD_or_NEG; NlO := 0;
                end
      comment: T.OPD is taken from CSl pointed to by l;
else if T.OPD = '$' then begin
                           VG [1] := 0;
                           VG [l] := 0;
                         end;
else goto L6;
    end
exit: end code generator;
```

It should be noted that the algorithm is presented in a descriptive form. Many irrelevant details are dropped in order to provide a clear and concise presentation of the basic idea underlying the algorithm. For example, there is only one temporary variable (T) although there can be more. Furthermore, the variables *NlO* and *NlB* are introduced for convenience and readability, although they were not necessary. Each change of *NlO* implies, in fact, the change of VG [K], and each change of *NlB* implies the change of VG [$K + 1$] in an elementary unit pointed to by *lc* or *l*. The use of pointers *lc* or *l* is not given in explicit form. Instead, their strategic use is described by comments which make an essential part of the algorithm description. The algorithm provides an automatic clearing of VG space with the exception of VG [$K + 2$] component in each of its 3-tuples. This is of significance in the algorithm implementation.

As an example, we shall apply the algorithm to VG-based data structure whose generation is already described. Each step in the code generation corresponds to the current elementary unit. It includes the starting values of CS, the part of VG for that elementary unit (indicated by VG_l and CS_l, $l = 1, 2, 3, \ldots 8$), and the results of operations performed until the current elementary unit is changed. Due to interdependence of elementary units, every item is denoted by a subscript to provide the identification of the elementary unit to whom an item belongs. The generated instruction lines are underlined in the resumé of each step. When they are extracted from there, one obtains the following code:

LDA *N*
DIV *I*
STA *T*
LDA *I*
MUL *I*
SUB *T*
STA *T*
LDA *N*
ADD *I*
NEG
NEG
DIV *T*
DIV *I*
MUL *I*
STA *T*
LDA *I*
ADD *N*
SUB *T*
ADD *I*

(1) $VG_7 = 1\ 1\ 22$
$CS_7 = I+$
$lc = 19$
$NlO_7 = 1$
$NlB_7 = 1$
$l = 19$
$lc = 16$

(2) $VG_6 = 1\ 2\ 19$
$CS_6 = I*-$
$NlO_6 = 1$
$NlB_6 = 2$
$l = 16$
$lc = 13$

(3) $VG_5 = 1\ 1\ 17$
$CS_5 = I/$
$NlO_5 = 1$
$NlB_5 = 1$
$l = 13$
$lc = 10$

(4) $VG_4 = 2\ 3\ 12$
$CS_4 = NI/-/$
$NlO_4 = 2$
$NlB_4 = 3$
$l = 10$
LDA N
DIV I
$NlO_4 = 1$
$NlB_4 = 2$
STA T
$NlO_4 = 0$
$'T' \rightarrow \text{stack}_4$
$lc = 7$

(5) $VG_3 = 2\ 1\ 9$
$CS_3 = II*$
$NlO_3 = 2$
$NlB_3 = 1$
$l = 7$
LDA I
MUL I
$NlO_3 = 1$
$NlB_3 = 0$
$l = 10$

(6) $VG_4 = 0\ 2\ 12$
$CS_4 = T-/$
$NlO_4 = 0$
$NlB_4 = 2$
SUB T
$NlO_4 = 0$
$NlB_4 = 1$
STA T
$NlO_3 = 0$
$'T' \rightarrow \text{stack}_4$
$lc = 4$

(7) $VG_2 = 2\ 3\ 4$
$CS_2 = NI + \#\ \#$
$NIO_2 = 2$
$NIB_2 = 3$
$l = 4$
LDA N
ADD I
$NIO_2 = 1$
$NIB_2 = 2$
NEG
$NIB_2 = 1$
NEG
$NIB_2 = 0$
$l = 7$

(8) $VG_3 = 0\ 0\ 9$
$CS_3 = \text{empty}$
$NIO_3 = 0$
$NIB_3 = 0$
$l = 10$

(9) $VG_4 = 0\ 1\ 12$
$CS_4 = T/$
$NIO_4 = 0$
$NIB_4 = 1$
DIV T
$NIO_4 = 0$
$NIB_4 = 0$
$l = 13$

(10) $VG_5 = 1\ 1\ 17$
$CS_5 = T/$
$NIO_5 = 1$
$NIB_5 = 1$
DIV I
$NIO_5 = 0$
$NIB_5 = 0$
$l = 16$

(11) $VG_6 = 1\ 2\ 19$
$CS_6 = I*-$
$NIO_6 = 1$
$NIB_6 = 2$
MUL I
$NIO_6 = 0$
$NIB_6 = 1$
STA T
$NIO_2 = 0$
$'T' \rightarrow \text{stack}_6$
$lc = 1$

(12) $VG_1 = 2\ 1\ 1$
$CS_1 = \text{IN}+$
$NIO_1 = 2$
$NIB_1 = 1$
$l = 1$
LDA I
ADD N
$NIO_1 = 1$
$NIB_1 = 0$
$l = 4$

(13) $VG_2 = 0\ 0\ 4$
$CS_2 = \text{empty}$
$NIO_2 = 0$
$NIB_2 = 0$
$l = 7$

(14) $VG_3 = 0\ 0\ 9$
$CS_3 = \text{empty}$
$NIO_3 = 0$
$NIB_3 = 0$
$l = 10$

(15) $VG_4 = 0\ 0\ 12$
$CS_4 = \text{empty}$
$NIO_4 = 0$
$NIB_4 = 0$
$l = 13$

(16) $VG_5 = 0\ 0\ 17$
$CS_5 = \text{empty}$
$NIO_5 = 0$
$NIB_5 = 0$
$l = 16$

(17) $VG_6 = 0\ 1\ 19$
$CS_6 = T-$
$NIO_6 = 0$
$NIB_6 = 1$
SUB T
$NIB_6 = 0$
$l = 19$

(18) $VG_7 = 1\ 1\ 22$
$CS_7 = \text{I}+$
$NIO_7 = 1$
$NIB_7 = 1$
ADD I
$NIO_7 = 0$
$NIB_7 = 0$
$l = 22$

(19) $VG_8 = 1\ 0\ 24$
$CS_8 = \$$
$NIO_8 = 1$
$NIB_8 = 0$
$NIO_1 = 0$
$NIO_8 = 0$
exit

Theorem. The binary tree and the postfix string with associated vector-generatrice give the code of the same length.

Proof. Let CS_l be given by Eqn (10). It is evident that CS_l can be represented generally as a *partial* subtree. The partial subtrees are those subtrees which have not all their nodes resolved. This uncompleteness is the consequence of the fact that the relation

$$N_l^O = N_l^B + 1 \tag{11}$$

is not always satisfied. Bearing in mind the procedure for the binary tree derivation, the graph representation of CS_l must form a partial subtree in the binary tree structure. Thus, the code generated from CS_l is of the same length as that obtained from the corresponding partial subtree belonging to the binary tree. If the following relation for CS_l

$$N_l^O \leq N_l^B \tag{12}$$

is valid, then, according to the given algorithm, CS_{l-1} will be taken, causing an instruction line

STA T

to be generated. When the code is generated from the binary tree, each node is examined in order to verify whether it contains a commutative operator. If it contains it, the left or right subtree of the node can be chosen. Otherwise, the right subtree must be chosen. Upon generation of the code from the chosen subtree, the alternative subtree is taken, with an instruction line

STA T

being generated. We can conclude that the choice of the left or right subtree for commutative operators does not influence the length of code obtained for that node. Thus, the fact that the given algorithm always chooses CS_{l-1}, i.e. the nearest (minimum number of nodes to be passed) right partial subtree if Eqn (12) is satisfied, will not influence the number of STA and LDA instructions to be generated. Now, we can state the following: (1) number of instruction lines generated for each CS_l is equal to that obtained from the corresponding partial subtree, and (ii) the number of STA and LDA instruction lines generated due to transition from CS_l to CS_{l-1}, CS_{l-1} to CS_{l-2}, ... etc., or due to transition from one to another corresponding partial subtree, ... etc., are equal as long as Eqn (12) is satisfied.

To finish the proof, we shall prove that for

$$N_l^O > N_l^B \tag{13}$$

one obtains the same length of code as with the binary tree moving to the right in VG space until $N_{lc}^O = 1 \wedge N_{lc+1}^O = 0 \wedge N_{lc+2}^O = 0 \wedge \ldots N_{lc+i}^O = 0 \wedge N_{lc+i}^B \neq 0$. It is clear that when a CS_l satisfying Eqn (13) is encountered, one or several partial subtrees satisfying Eqn (12) are already handled. These partial subtrees will complete the partial subtree satisfying Eqn (13) to obtain the resulting subtree satisfying Eqn (11). CS_l which satisfies Eqn (13) corresponds to a left subtree in the binary tree. Thus, moving to the right in VG space will not cause any paired instructions STA and LDA to be generated due to transitions. Because we have to generate the code for a node having as its left side a partial subtree and as its right side the leefs, paired instructions STA and LDA

will not be generated, with the exception of the surplus of operands in CS_l which can be covered by non-commutative operators left over during the handling of previous CS_l satisfying Eqn (12). Thus, the code generated from a CS_l satisfying Eqn (13) will be of the same length as the code generated from the corresponding partial left subtree and the leefs. This closes the proof.

In fact, the code generator based on the vector-generatrice simulates the code generation from the binary tree with only one exception that at each node the right subtree is firstly handled independently from the commutativity of operators. The consequence of that can be only the different generated code.

EXPERIMENTAL RESULTS

An experiment has been performed over a set of arithmetic expressions transformed to postfix strings, binary trees and postfix strings with associated vector-generatrices in order to examine the time performances of algorithms and the difference in length of the generated code. The set of arithmetic expressions is divided in subsets each comprising arithmetic expressions of the same length. The initial length was 6 and the final one 72. The length is varied in increments of 6. The end marker is accounted for in the length. Each subset contained 15 expressions. Furthermore, two additional subsets are taken into consideration: (1) a subset consisting of a single operand (i.e., $A\$$) and (2) a subset consisting of two operands (i.e., $A+B\$$, $A*B\$$, etc). In order to achieve a uniform distribution in length space, these subsets are filled to also contain 15 expressions (length 2 and 4). The time performances are obtained by evaluating the mean values for each subset. Two classes of comparisons are performed: (1) postfix string–binary tree, and (2) binary tree–postfix string with associated vector-generatrice.

Firstly, the performances of postfix string and binary tree are compared. The comparisons are based on parameters D_L, D_T^s, D_T^c and D_T defined in the following way:

$$D_L = \frac{L_{ps} - L_{bt}}{L_{bt}} \tag{14}$$

$$D_T^s = \frac{\sum_1^m (T_{bt}^s - T_{ps}^s)}{\sum_1^m T_{ps}^s} \tag{15}$$

$$D_T^c = \frac{\sum_1^m (T_{bt}^c - T_{ps}^c)}{\sum_1^m T_{ps}^c} \tag{16}$$

$$D_T = \frac{\sum_1^m (T_{bt}^s + T_{bt}^c) - \sum_1^m (T_{ps}^s + T_{ps}^c)}{\sum_1^m T_{ps}^s + T_{ps}^c} \tag{17}$$

where m = number of subsets,
L_{ps} = length of code generated from postfix string,
L_{bt} = length of code generated from binary tree,
D_L = relative difference in length of generated code,
T_{ps}^s = mean time spent to syntax analysis and transformation to postfix string,
T_{bt}^s = mean time spent to syntax analysis and transformation to binary tree,
D_T^s = relative difference in time spent to syntax analysis and transformation,
T_{bt}^c = mean time spent to code generation from binary tree,
T_{ps}^c = mean time spent to code generation from postfix string,
D_T^c = relative difference in time spent to code generation,
D_T = total relative difference in time spent to code generation, syntax analysis and transformation.

It was found that D_L was between about 0–0.4. This fact becomes significant in highly repetitive loops with a large number of calculations. The code obtained from the binary tree is shorter because the whole arithmetic expression is in a form which provides the alternatives in particular points during the code generation.[5] There are no such alternatives in the postfix string, which is scanned strictly sequentially, thus eliminating any possibility of economizing in the code generation.

The results obtained for D_T^s, D_T^c and D_T (0.15, 0.33 and 0.24 respectively) show that these parameters are practically independent for expression length, in a statistical sense, when evaluated separately for each subset ($m = 1$). The memory requirement for transformation to the binary tree was 50% higher and approximately the same for the phase of the code generation.

The second class of comparisons made between the binary tree and postfix string with the associated vector-generatrice represents the key point in the experiment. The results are presented in Table 2.

Table 2. The results of the comparison of binary tree to postfix string with vector-generatrice

D_L	D_T^s	D_T^c	D_T	D_M^s	D_M^c	D_I^s	D_I^c	D_I
0	0.03	0.21	0.13	−0.12	0.12	≃0	≃0.06	0.024

D_L, D_T^s, D_T^c and D_T are defined analogously to Eqns (14), (15), (16) and (17):

$$D_L = \frac{L_{bt} - L_{vg}}{L_{bt}} \tag{18}$$

$$D_T^s = \frac{\sum_1^m (T_{bt}^s - T_{vg}^s)}{\sum_1^m T_{bt}^s} \tag{19}$$

$$D_T^c = \frac{\sum_1^m (T_{bt}^c - T_{vg}^c)}{\sum_1^m T_{bt}^c} \tag{20}$$

$$D_T = \frac{\sum_1^m (T_{bt}^s + T_{bt}^c) - \sum_1^m (T_{vg}^s + T_{vg}^c)}{\sum_1^m T_{bt}^s + T_{bt}^c} \tag{21}$$

It was found also that the parameters D_T^s, D_T^c and D_T were independent on expressions length ($m = 1$).

D_M^s, D_M^c, D_M, D_I^s and D_I^c are defined as follows:

$$D_M^s = \frac{M_{bt}^s - M_{vg}^s}{M_{bt}^s} \tag{22}$$

$$D_M^c = \frac{M_{bt}^c - M_{vg}^c}{M_{bt}^c} \tag{23}$$

$$D_M = \frac{(M_{bt}^s + M_{bt}^c) - (M_{vg}^s + M_{vg}^c)}{M_{bt}^s + M_{bt}^c} \tag{24}$$

$$D_I^s = \frac{I_{bt}^s - I_{vg}^s}{I_{bt}^s} \tag{25}$$

$$D_I^c = \frac{I_{bt}^c - I_{vg}^c}{I_{bt}^c} \tag{26}$$

$$D_I = \frac{(I_{bt}^s + I_{bt}^c) - (I_{vg}^s + I_{vg}^c)}{I_{bt}^s + I_{bt}^c} \tag{27}$$

where M_{bt}^s = memory space necessary for syntax analysis and transformation to binary tree,
M_{vg}^s = memory space necessary for syntax analysis and transformation to postfix string with associated vector-generatrice,
M_{bt}^c = memory space necessary for code generation from binary tree,
M_{vg}^c = memory space necessary for code generation from vector-generatrice,
I_{bt}^s = number of instruction lines of syntax analyser (binary tree),
I_{vg}^s = number of instruction lines of syntax analyser (vector-generatrice),
I_{bt}^c = number of instruction lines of code generator (binary tree),
I_{vg}^c = number of instruction lines of code generator (vector-generatrice).

The experimental results obtained show a significant decrease of code generation time (21%) when the vector-generatrice is applied. The overall improvement in time domain is reduced to 13% when the syntax analysis is taken into account. Other improvements, expressed through parameters D_M and D_I are practically negligible. The code obtained from a vector-generatrice uses one or two more temporaries than the code obtained from a binary tree, even for very long expressions (in the case when the left substrees are chosen for commutative operators).

Because VG space is cleared (with the exception of VG [K + 2]) after each execution of the code generator, the action routine *NULL* reduces only to the initialization of variables *current_VG_index*, *current_index_of_output_string* and *VG* [3].

In order to obtain an efficient code generator, several changes are done over the postfix string: (1) commutativity of operators is indicated by a special character (blank-operator is commutative, $-operator is not commutative); (2) unary operator is indicated by the numeric # (function call could be also considered as a unary operator); (3) each operator is replaced by a symbolic-machine code during the syntax analysis. The same changes are performed over the binary tree structure.

The postfix string in the example gets the following form after performing these changes:

*IN*bADD*NI*bADD # NEG # NEG*II*bMUL*NI*$
DIVSUBDIV*I*$DIV*I*bMUL$SUB*I*bADD$

where b = blank.

ADD, SUB, MUL, DIV, NEG—symbolic-machine representation of operators +, −, *, / and unary minus.

It should be noted that the syntax analysis was done informally through the top-down or bottom-up processing of the translation grammar. The informal syntax analysis is more easily programmed for the given translation grammar.

The machine output of the example includes the input infix string, the modified output postfix string, the vector generatrice, the syntax analysis time (*T1*), the code generation time (*T2*), the total time *T*(*T1* + *T2*) and the generated code. The input string of the example is modified to include different, single character variables (not only *I* and *N*),

$$A + B - (- (- (B + C))/(D * E - G/F))/M * N + G\$,$$

but the structure of the input string is left unchanged. The output is as follows:

```
INFIX STRING IS:
A+B-(-(-(B+C))/(D*E-G/F))/M*N+G$
POSTFIX STRING IS:
AB ADDBC ADD # NEG # NEGDE MULGF$DIV$SUB$DIVM$DIVN MUL$SUBG ADD$
VECTOR GENERATRICE IS:
2  1  0  2  3  6  2  1 20  2  3 26  1  1 40
1  2 45  1  1 54  1  0 59
        SYNTAX ANALYSIS TIME IS: T1 = 3.033 MSEC
        GENERATED CODE IS:
        LDA G
        DIV F
        STA Z
        LDA D
        MUL E
        SUB Z
        STA Z
        LDA B
        ADD C
        NEG
        NEG
        DIV Z
        DIV M
        MUL N
        STA Z
        LDA A
        ADD B
        SUB Z
        ADD G
        CODE GENERATION TIME IS: T2 = 1.490 MSEC
        TOTAL TIME IS: T = 4.523 MSEC.
```

The character *Z* denotes the symbolic address of a temporary variable.

The same code is generated from the binary tree with the following times:

T1 = 3.012 MSEC
T2 = 1.961 MSEC
T = 4.973 MSEC

Besides this general experiment, a subexperiment is performed over two subsets of expressions of length 2 and 4 (single and two operand subsets) with uniform distribution. The same benefit is observed in the domain of the code generation, but the overall time gain (when the syntax analysis is taken into account) is reduced to 8%.

CONCLUSION

The right-to-left generator has demonstrated better characteristics than the ones obtained with a binary tree. The length of the code generated from a vector-generatrice is equal to one obtained from binary tree. The compilation time for arithmetic expressions (syntax analysis + code generation) is appreciably shorter. This benefit is more significant for the stand-alone code generator. The total memory space (syntax analyser + code generator) was approximately the same. The memory space required by the syntax analyser, based on the vector-generatrice, was greater but it was balanced by the decrease of the memory requirement for the corresponding code generator. With respect to programming, the code generator based on the vector-generatrice

was programmed with a smaller number of instruction lines, but not significantly.

It would be interesting to extend the investigations on the application of the vector-generatrice to all operator precedence grammars and especially to the code optimization.

REFERENCES

1. F. Genuys, *Programming Languages*, pp. 99–122, Academic Press, London (1968).
2. H. Stone, One-pass compilation of arithmetic expressions for a parallel processor, *Communications of the ACM* **10**, 220–223 (1967).
3. D. Gries, *Compiler Construction for Digital Computers*, pp. 337–374, Wiley, New York (1971).
4. P. Sheridan, The arithmetic translator-compiler of the IBM FORTRAN automatic coding system, *Communications of the ACM* (February, 1959).
5. F. Hopgood, *Compiling Techniques*, pp. 89–118. Macdonald, London (1970).
6. J. Anderson, A note on some compiling algorithms, *Communications of the ACM* **7**, 149–150 (March, 1964).
7. I. Nakata, On compiling algorithms for arithmetic expressions, *Communications of the ACM* **12**, 81–84 (February, 1967).
8. R. Redziejowski, On arithmetic expressions and trees, *Communications of the ACM* **12**, 81–84 (February, 1969).
9. R. Sethi and J. Ullman, The generation of optimal code for arithmetic expressions, *Journal of the ACM* **17**, 715–728 (April, 1970).
10. J. Beatty, An axiomatic approach to code optimization for expression, *Journal of the ACM* **19**, 4 (April, 1972).
11. D. Frailey, Expression optimization using unary complement operators, *ACM SIGPLAN Notices*, 67–85 (July, 1970).
12. A. Aho and J. Ullman, The theory of parsing, translation and compiling, *Computing* **11**, 878–902, Prentice-Hall, Englewood Cliffs, New Jersey (1973).
13. L. Horwitz *et al.*, Index register allocation, *Journal of the ACM* **13**, 43–61 (January, 1966).
14. K. Kennedy, Index register allocation in straight-line code and simple loops, in *Design and Optimization of Compilers* (Ed by R. Rustin), pp. 51–64. Prentice-Hall, Englewood Cliffs, New Jersey (1972).
15. R. Sethi, *Validating register allocations for straight-line programs*, Phd Thesis, Department of Electrical Engineering, Princeton University (1973).
16. R. Allard *et al.*, Some effect of the 6600 computer on language structures, *Communications of the ACM* **7**, 112–127 (February, 1964).
17. H. Hellerman, Parallel processing of algebraic expressions, *IEEE Transactions on Electronic Computers* **EC-15**, 82–91 (January, 1966).
18. J. Baer and D. Bovet, Compilation of arithmetic expressions for parallel computations, *Proceedings of the IFIP Congress*, B4–B10 (1968).
19. R. Graham, Bounds on multiprocessing anomalies and related packing algorithms, *Proceedings of the AFIPS Spring Joint Computer Conference*, Vol. 40, pp. 205–217. AFIPS Press, Montvale, New Jersey (1972).
20. E. Hawkins and D. Huxtable, A multi-pass translation scheme for Algol 60, *Annual Review in Automatic Programming* **3**, 163–168 (1963).
21. H. Kanner *et al.*, The structure of yet another Algol compiler, *Communications of the ACM*, 427–430 (July, 1965).
22. B. Randell and L. Russell, *Algol 60 Implementation*, Academic Press, London (1964).
23. R. Graham, Bounded context translation, *Proceedings AFIPS, SJCC*, Vol. 25, pp. 17–29 (1964).
24. A. Tanenbaum, Implications of structured programming for machine architecture, *CACM*, March, pp. 237–246 (1978).
25. P. Lewis *et al.*, Compiler design theory, Addison-Wesley Publishing Company, London, pp. 181–226 (1976).

Received April 1980

Books Reviewed in this Issue