Note on coding Reverse Polish expressions for single-address computers with one accumulator

By A. J. T. Colin

This note gives a general method of coding Reverse Polish expressions into machine orders for single-address computers with one accumulator, and includes the description of an algorithm which improves the efficiency of the compiled program, by manipulating the expression before translation.

Since the subject of this note is the manipulation of Reverse Polish algebraic expressions, it may be as well to explain certain terms used in connection with the latter.*

The Reverse Polish notation is a method of writing algebraic expressions so that operators *follow the operands they refer to.* For example, the sum of two numbers a and b (normally written $a + b$) is, in Reverse Polish $ab +$; or "the square root of x" is written $x\surd$.

Operands may be either explicitly named, as in the examples above, or they may be the results of some previous operations on different operands. We may put, for $m + \surd n$ in conventional notation, $mn\surd +$ in Reverse Polish. The two operands of the operators "$+$" are m and $n\surd$.

This process of using general operands may be extended to any depth, thus avoiding the need to use brackets.

In general there are two types of operator: *monadic*, which refer to only one operand (such as $\surd$, sin, log, or mod) and *diadic*, which refer to *two* operands (such as $+$, $\times$, or $\div$). As an example, we shall give the Reverse Polish form of the expression

$$\log\,[a + bc(d - e\surd f/g)]$$

It is:

$$abc \times def\surd \times g \div - \times + \log$$

It is well known that the coding of Reverse Polish expressions for computers with nesting-store accumulators is particularly easy, because each symbol in the expression can be represented by one machine order. On the other hand, coding these expressions for conventional computers is more difficult, because it is generally impossible to establish a one-to-one correspondence between symbols and machine orders. When such coding is being done, it is most convenient to split the expression into four types of "molecules" as follows. (It is assumed that the Reverse Polish expressions considered do not contain the operator "becomes.")

(*a*) "Simple operations," which each consist of an operand followed by a diadic operator.
(*b*) Monadic operators.
(*c*) and (*d*) Operands and diadic operators which cannot be paired into simple operations.

*See also the paper by A. B. Hamlin which begins on p. 210 of the October 1962 issue of this *Journal.—Ed.*

When translating the expression, a stack of consecutive registers, numbered 1 upwards, is set aside for storing intermediate results, and a pointer (P) is initially set to zero. Each molecule of the string is then considered in turn, and coded by the following rules.

(1) Simple operation:

The appropriate machine order (which is usually available) is compiled. For example, the molecule "$\alpha +$" would be compiled as

"Add α"

(2) Monadic operator:

Instructions to carry out the indicated operation on the contents of the accumulator are compiled.

(3) Operand:

If the current value of the stack pointer is 0, the instruction compiled is simply "Clearadd Operand." Otherwise it is necessary to compile the two instructions

```
Store    (P)
Clearadd   Operand
```

In both cases P is increased by 1.

(4) Diadic Operator:

Instructions are compiled to carry out the specified operation using the contents of the accumulator and of storage register $(P - 1)$ as operands. P is decreased by 1.

To illustrate this technique, consider the expression $a - b + xy\surd z$, whose Reverse Polish form is

$$ab - xy \times z\surd \times +.$$

The coding process for this expression is as shown in Table 1.

This code will produce the value of the expression in the accumulator.

It is clear that the inefficiencies in this code are introduced as a result of the storage operation which is necessary whenever any operand after the first is compiled. The ideal form of a Reverse Polish expression to be compiled for a single-address computer is therefore

Table 1

MOLECULE CONSIDERED	FINAL VALUE OF POINTER	CODE PRODUCED
a	1	Clearadd a
$b-$	1	Subtract b
x	2	$\left\{\begin{array}{l}\text{Store (1)}\\ \text{Clearadd } x\end{array}\right.$
$y\times$	2	Multiply y
z	3	$\left\{\begin{array}{l}\text{Store 2}\\ \text{Clearadd } z\end{array}\right.$
$\surd$	3	Acc ′ = : $\surd$ Acc
$\times$	2	Multiply (2)
$+$	1	Add (1)

one operand, followed only by simple operations and monadic functions.

For example, $ab + c - d \times e +$ and $z\surd x \times y \times a + b -$ are ideal forms, but $abc \times \times$ is not.

It is frequently possible to manipulate a Reverse Polish string so that its mathematical meaning is not changed, but its form is brought closer to the ideal. The manipulations which can be used consist of interchange of sequences of symbols, according to certain rules. The circumstances under which these interchanges can be made are set out below.

We shall define a "general operand" as "The subject of an operator." General operands can clearly consist of one or more symbols; so that, for example, in the expression $ab \times cd - f \div +$ both $ab\times$ and $cd - f\div$ are general operands (as they are both the subjects of the operator $+$). We shall also define a "general operation" as a "general operand followed by a diadic operator."

Rules for possible interchanges are:

(1) Adjacent general operations may be interchanged whenever the order in which they are carried out is immaterial. For example, adjacent additions and/or subtractions, or adjacent multiplications and/or divisions, may be carried out in any order; and adjacent operations involving these groups of operators can be interchanged.

(2) The two general operands of a diadic operator may be interchanged provided that the operator itself is replaced by a mirror-image operator, which specifies the same operation with the operands taken in the *reverse* order.

Clearly, only symmetrical operators, such as addition or commutative multiplication are identical to their mirror images. Below, we shall denote mirror image operators by circling them; so that we may write:

$$(ab-) \equiv (ba \ominus).$$

The rules by which actual interchanges are selected from all possible ones can be deduced from the following consideration. First, single operands should be moved towards the right, for there they may combine with diadic operators to produce operations. Secondly, operations which include monadic functions should be moved to the left, because the subjects of these functions can never form operations, and the only place in the string where two consecutive operands can be efficiently dealt with is at the extreme left.

These two ideas can be combined in a single rule which states: "Whenever an interchange of two groups is possible, arrange for the *longer* group to be on the left, and the shorter on the right." This rule should be applied repeatedly to the interchange of operations and then to the interchange of operands.

Two examples of the application of this rule are given below.

The brackets in the Reverse Polish expressions which follow have no mathematical significance, but are inserted to show which groups of symbols are to be interchanged.

(1) The Reverse Polish form of the expression $a - b + xy\surd z$ is $ab - xy \times z\surd \times +$.

By interchange of operations,

$$a(b-)(xy \times z\surd \times +) \equiv axy \times z\surd \times + b - ;$$
$$ax(y\times)(z\surd\times) + b - \equiv axz\surd \times y\times + b -;$$

By interchange of operators,

$$a(x)(z\surd) \times y \times + b - \equiv az\surd x \otimes y \times + b - ;$$
$$(a)\,(z\surd x \otimes y \times) + b - \equiv z\surd x \otimes y \times a \oplus b -.$$

(Ideal form).

(2) The Reverse Polish of $a + b(c - de/f)$ is $abcde \times f \div - \times +$. By interchange of operands,

$$ab(c)(de \times f \div) - \times + \equiv abde \times f \div c \ominus \times + ;$$
$$a(b)(de \times f \div c \ominus) \times + \equiv ade \times f \div c \ominus b \otimes + ;$$
$$(a)(de \times f \div c \ominus b \otimes) + \equiv de \times f \div c \ominus b \otimes a \oplus$$

(Ideal form)

The system described in this note is liable to produce alteration in the order in which operands are used. In most cases this is immaterial, but where the order does matter,* this could be indicated to the compiler by the use of a special "becomes" symbol or similar device.

Lastly, it is often possible to improve coding by the introduction of purely "local" operators which correspond to special instructions available on the target computer. An example of such an operator is negative multiplication. It can be shown that the sequence "$\times -$" can be replaced by the sequence "$\overline{\times} +$", where $\overline{\times}$ denotes the operation of negative multiplication. This device is often useful in getting rid of mirror-image subtraction, which can be difficult to code. For example, consider the expression $a - bc$: which in Reverse Polish is $abc \times -$. Without substitution, this transforms to $bc \times a \ominus$, which may lead to some awkwardness in coding; but with substitution we obtain

$$\begin{aligned} & abc \times - \\ \equiv\ & abc\, \overline{\times} \oplus \\ \equiv\ & bc\, \overline{\times}\, a \oplus \\ \equiv\ & bc\, \overline{\times}\, a + \end{aligned}$$

which is easily coded.

* E.g. if $x_n \ll y$, the evaluation of $x_1 + x_2 + x_3 + x_4 + \ldots + y$ and of $y + x_1 + x_2 + x_3 + x_4 + \ldots$ on a floating-point computer might give significantly different results.