Translation to and from Polish notation

By C. L. Hamblin

"Reverse Polish" notation is embodied in the instruction languages of two recent machines, and "Forward Polish" notation is of use in mechanized algebra. This article illustrates, using a simple language without detail, some methods of translating between these notations and an "orthodox" one of the kind used in FORTRAN and ALGOL.

The question of efficient translation between an "orthodox" mathematical notation of the kind ordinarily used in writing algebraic formulae (and copied as closely as is practicable in FORTRAN and ALGOL) and "Polish" notation has come to prominence as a result of the use of what is in effect Polish notation as the basic instruction language of two recent computers.*

Polish notation is so-called because of its extensive use in Polish logical writings since its invention by Łukasiewicz (1921, 1929). Łukasiewicz demonstrated that if operators are written always *in front of* their operands, instead of (as in the case of the diadic operators of arithmetic, "$+$", "$-$", "$\times$" and so on) *between* them, there is never any need for brackets to indicate association of terms. Thus if in place of "$a + b$" we write "$+ a b$", and so on, the brackets in an expression such as "$(a + b) \times c$" may be dispensed with in translation, since "$\times + a b c$" indicates unambiguously the result of operating with "$\times$" on "$+ a b$" and "c": for "$a + (b \times c)$" we should instead write "$+ a \times b c$." The resulting notation, in the case of long formulae, is a little harder to read, since brackets aid the eye, but it has some other advantages. In particular *Reverse* Polish—the notation which results if operators are placed *after* operands, as in "$a b +$"—has the property that the operators appear in the order in which they are required in computation. Reverse Polish is hence in some sense a natural notation for an instruction language, each symbol being interpretable as an instruction. (Number variables are "fetch" instructions.) The absence of brackets further makes Polish notation (—either Forward or Reverse, but probably preferably Forward—) useful in mechanized algebra, since it eliminates a continual source of complication in algebraic manipulations.

Machine translation from one notation to another is needed in writing compilers for the new machines, and it is possible to foresee a variety of future uses for it. This article illustrates, using a simple language without detail, some translation methods. In general, translation is extremely simple if done in the right way

It is convenient to distinguish "pure" translation from translation which involves manipulation or rearrangement. A simple way of characterizing this distinction is in terms of the order of the number-denoting symbols (numbers and number variables): in "pure" translation these symbols remain unaltered and in the same order in the translated formula as they were in the original. Thus we shall say that the transformation of "$a + (b \times c)$" into Forward Polish "$+ a \times b c$" is a case of pure translation, whereas its transformation into "$+ \times b c a$," though this is an equivalent form, involves manipulation as well as translation, since the order "$b c a$" of the number variables is different. We confine ourselves to pure translation, so defined, in what follows.

This restriction, however, does not yet entirely remove the possibility that a formula in a given notation should have alternative forms. This is because of the associativity of some arithmetical operators. Thus in orthodox notation "$(a + b) + c$" is equivalent to "$a + (b + c)$," and the brackets are usually omitted; but to these formulae correspond in Forward Polish the formulae "$+ + a b c$" and "$+ a + b c$" respectively. To resolve ambiguity we distinguish two special cases, the *early-operator* and *late-operator* forms respectively of Polish formulae. A Polish formula is in early-operator (late-operator) form if all operator symbols occur as early (late) in it as possible. Thus "$a+b+c+d$" becomes "$+ + + a b c d$" in early-operator Forward Polish, "$+ a + b + c d$" in late-operator Forward Polish, "$a b + c + d +$" in early-operator Reverse Polish, and "$a b c d + + +$" in late-operator Reverse Polish. There are of course intermediate forms such as "$+ + a b + c d$" and "$a b + c d + +$" which, though valid Forward and Reverse Polish respectively, are neither early-operator nor late-operator.

In the case of Reverse Polish for use as an instruction language it is usually the early-operator form that is desirable, since this uses the minimum number of locations in the push-down store.

By *Orthodox A* I shall mean a language constructed with orthodox symbol-order out of the following symbols.

(i) Number-variables a, b, c, d, . . . (The use of actual numerals raises no essential new issues; we need not consider it here.)

(ii) Operators $+$, $-$, *neg*, $\times$, $\uparrow$. Of these, "*neg*" (representing "negative") is monadic, i.e. operates on a single number, and is placed in front of its operand, as in "*neg* a": the others are diadic and stand between their operands, as in "$a + b$". Symbol "$\uparrow$" denotes

* The English Electric KDF9 and the Burroughs B5000. Each of these uses a "push-down" (or "nesting") type of store for arithmetic operands and results, following a scheme suggested by the present author (Hamblin, 1957, 1957, 1960; see also Hamblin, Humphreys, Karoly and Parker, 1960).

exponentiation: thus for "a^b" we write "$a \uparrow b$." (There is, of course, no difficulty in arbitrarily extending the range of permitted operators, but these are enough for our present purpose.)

(iii) Brackets (,). We assume that "+" and "—" are *weaker* (that is, more weakly associative) than "*neg*," which is in turn weaker than "×," which is in turn weaker than "$\uparrow$." Hence the absence of brackets will never actually lead to any ambiguity. For example

$$neg\, a \times b + c \uparrow d \times e + f \times g$$

will mean $\qquad -ab + c^d e + fg.$

Brackets are used to associate symbols into a group when they are not automatically so associated by these rules. (There is, of course, no penalty if brackets are used superfluously.)

It is a trivial matter to convert formulae in a fully orthodox notation to Orthodox A, provided, of course, that they use only the permitted range of mathematical notions. The essential rules are as follows.

(i) Alter "—" to "*neg*" wherever it occurs at the beginning of a formula or immediately following a L.H. bracket.

(ii) Insert "$\uparrow$" wherever there is a change from normal type-face to that used for exponents, and put brackets round the exponent which follows if it contains any operator. Then use the same type-face throughout.

(iii) Insert "×" wherever a number variable or R.H. bracket is followed by a number variable or L.H. bracket.

The Polish notations considered here will have exactly the same range of symbols as Orthodox A, except, of course, the brackets.

The following cases of translation will be considered in detail:

I. Orthodox A to Reverse Polish.
II. Orthodox A to Forward Polish.
III. Forward Polish to Orthodox A.
IV. Forward Polish to Reverse Polish.

These cases provide a survey of the relevant techniques. As will appear, only minor modifications are needed to give the other cases of interest.

I. Orthodox A to Reverse Polish

This is the simplest of the cases. Let $S_1 S_2 \ldots S_m$ be the Orthodox A formula. The symbols of this formula are examined one by one in order from left to right, and the translated formula is written out symbol-by-symbol directly. Number variables are transcribed as soon as they are encountered. Operator-symbols, which can never occur earlier in the sequence of number variables in Reverse Polish than they do in Orthodox A, are held in a "nesting list" N until conditions for their transcription are satisfied. (A "nesting list" is a list operated on the "last-in-first-out" principle. That is, of the entries in the list only the last is available at any one time: if the list has entries $E_1, E_2, \ldots, E_n$, it is necessary to remove E_n before E_{n-1} can be inspected, and so on.)

In detail, the following are the operations to be carried out when symbol S_j of the Orthodox A formula is examined.

(*a*) If S_j is a number variable $a, b, c, d, \ldots$ it is transcribed directly to output.

(*b*) If S_j is a L.H. bracket symbol, it is transcribed to list N.

(*c*) If S_j is an operator symbol, the last entry—call it E—of list N is examined: if E is an operator not weaker than S_j, E is transcribed to output and the next last entry similarly examined; and so on until N is empty or its last entry is a L.H. bracket or an operator weaker than S_j. Then S_j is transcribed to list N.

(*d*) If S_j is a R.H. bracket symbol, entries are transcribed from list N to output until a L.H. bracket symbol is reached: this is deleted.

(*e*) After the last symbol of the Orthodox A formula has been dealt with, the remaining entries of N are transcribed to output.

As described, this procedure gives as output the early-operator form of Reverse Polish. An alteration of detail yields a procedure which gives the late-operator form: paragraph (*c*) is replaced by:

(*c′*) If S_j is an operator symbol the last entry—call it E—of list N is examined: if E is an operator and S_j is weaker than E, or if E is "—" and S_j is "+" or "—," E is transcribed to output and the next last entry similarly examined; and so on until N is empty or its last entry is a L.H. bracket or an operator not as described. Then S_j is transcribed to list N.

The special provisions regarding "+" and "—" here guard against error owing to the incomplete associativity of "—": thus, for example, "$a - b + c$" does not have separate early-operator and late-operator forms, becoming "$a\, b - c\, +$" in either case. Actually, "—" in orthodox notation is *associative to the left*: this corresponds with early-operator Polish directly, but will always lead to a special rule in other cases.

II. Orthodox A to Forward Polish

Translation to or from early-operator (late-operator) Forward Polish is closely the same as translation to or from late-operator (early-operator) Reverse Polish *backwards*, i.e. *from right to left*. In fact the only fore-and-aft asymmetry that occurs is not in the Polish notations, but in the Orthodox A, and then only refers to "*neg*," which appears in front of its operands when the formula is read forwards and after them when the formula is read backwards, and to the associativity properties of "—." Consequently, under this heading two translation methods will be considered, of which the first, which is by far the simpler, is a modification of that described above, used backwards. Circumstances might arise, however, in which it was not desirable to be forced to write and read formulae backwards, and in such cases a method such as the second must be resorted to. The extra complexity of this method is a considerable penalty, but it is unavoidable since in translation from Orthodox A to Forward Polish the operators must be moved forward in the formula, not back; and this cannot be done on-the-run. The alternative of "queueing" the number-variables until the operators are sorted out is not as simple as it sounds, since in most cases *all* the number-variables need to be placed in the queue before a single one is taken out and sent to output, and one might as well have no queue but simply resort to more than one run-through of the formula; for example, to a translation first to Reverse Polish, followed by a translation from Reverse to Forward as described in IV. Method 2, below, would usually be faster than this.

Method 1

Let $S_1S_2 \ldots S_m$ be the Orthodox A formula, and let it contain p bracket symbols: after translation let the resulting Forward Polish formula be $S_1'S_2' \ldots S_n'$, where of course $n = m - p$. Symbols of the Orthodox A formula are taken one by one in the reverse order $S_m, S_{m-1}, \ldots$ and the translated formula is written out symbol-by-symbol in the reverse order $S_n', S_{n-1}', \ldots$ The procedure each time a symbol S_j is examined is the same as in I above, except that if S_j is the symbol "*neg*" it is transcribed to output directly in the same way as a number variable; and that under (*c*) in I, for "not weaker than" it is necessary to read "weaker than", and for "weaker than" it is necessary to read "not weaker than".

This gives the early-operator form. For the late-operator form a comparable, if slightly more complicated, modification of (*c'*) is substituted.

Method 2

Here we first effect a "virtual" reordering of the Orthodox A symbols without rewriting them, by placing against each (other than brackets) the present address of the symbol which is to follow it in the revised order. A separate indication gives the starting-symbol. For example, if symbols "*ABCDEF*" were stored at addresses 18–23 respectively, we could indicate our intention of reordering them "*CBDFEA*" by noting the address (20) of *C* as starting-point, writing against *C* at address 20 the address (19) of *B*, against *B* at address 19 the address (21) of *D*, and so on; thus:

			Start			
Address	18	19	20	21	22	23
	A	*B*	*C*	*D*	*E*	*F*
(next address)		21	19	23	18	22

This can be done in a single run-through of the formula with the aid of a subsidiary list L: each entry of L consists of a symbol and two addresses, indicating a subsequence of the finished formula. L reduces at the end of the process to a single entry.

Let $S_1S_2 \ldots S_m$ be the Orthodox A formula, and $S_1'S_2' \ldots S_n'$ the corresponding formula in Forward Polish. Let $A_1, A_2, \ldots, A_m$ be the addresses of the symbols in the Orthodox A formula. Against each, unless it is a bracket or the final symbol S_n', we write an address, $B_1, B_2, \ldots, B_m$. The final output will then be taken as follows: given that S_{j1} is the starting symbol, it is sent to output and the symbol at address B_{j1} is fetched—let this be S_{j2}: this is sent to output and the symbol at address B_{j2} is fetched—let this be S_{j3}: and so on until a blank address is reached.

Let list L consist at any time of p entries $E_1, E_2, \ldots, E_p$ (where p may, of course, be zero). Each entry E_j consists of a symbol T_j and two addresses C_j and D_j. T_j is one of the symbols a, (, +, −, *neg*, ×, ↑. Every entry stands for a sequence of symbols in the final (Polish) formula: if T_j is a there is a number-denoting expression which can be found in the Orthodox A formula by starting with the symbol at address C_j—call it S_{k1}: taking next the symbol at B_{k1}—call it S_{k2}: and so on until the symbol at address D_j has been taken. If T_j is a diadic operator there is a similar sequence consisting of that symbol followed by a number-denoting expression, its first operand. If T_j is a monadic operator we always have $C_j = D_j$ (there is, as it were, a one-symbol sequence). If T_j is a bracket symbol the entries that follow it are all contained within a bracket-pair in the Orthodox A formula: here C_j and D_j are left blank and are not relevant.

At various stages, to be specified, an entry which is a *merger* of a succession of existing entries is formed. To merge $E_i(= T_iC_iD_i)$, $E_j(= T_jC_jD_j)$, and $E_k(= T_kC_kD_k)$ we replace these entries by a single one, namely by aC_iD_k if T_k is a, otherwise by $T_iC_iD_k$: at the same time against the symbol (in the Orthodox A formula) at address D_i we write the address C_j; and against the symbol at address D_j we write C_k. Similarly for the merger of a longer or shorter sequence of entries.

The procedure for the writing-in of addresses against the symbols of the Orthodox A formula can now be fully specified. The symbols $S_1, S_2, \ldots, S_m$ are examined in order and for each S_j the following action is taken.

(*a*) If S_j is a number variable an entry aA_jA_j is added to the list L.

(*b*) If S_j is a L.H. bracket an entry "(0 0" is added to the list L.

(*c*) If S_j is a monadic operator symbol an entry $S_jA_jA_j$ is added to the list L.

(*d*) If S_j is a diadic operator symbol list L is examined backwards from the last entry (without removing any entries) until either a weaker operator symbol, or a bracket, or the beginning of the list is encountered. Then

(i) if what is encountered (say at E_k) is a weaker operator symbol, E_{k+1} is replaced by the merger of $S_jA_jA_j$, $E_{k+1}, E_{k+2}, \ldots, E_p$; and $E_{k+2}, \ldots, E_p$ are deleted.

(ii) If what is encountered (say at E_k) is a bracket symbol, E_k is replaced by the merger of $S_jA_jA_j$, E_{k+1}, $E_{k+2}, \ldots, E_p$; and $E_{k+1}, \ldots, E_p$ are deleted.

(iii) If what is encountered is the beginning of the list, E_1 is replaced by the merger of $S_jA_jA_j$, $E_1, E_2, \ldots, E_p$; and $E_2, \ldots, E_p$ are deleted.

(*e*) If S_j is a R.H. bracket list L is examined backwards (without removing entries) until a bracket symbol is encountered (say at E_k); and E_k is replaced by the merger of $E_{k+1}, E_{k+2}, \ldots, E_p$, which are deleted.

When all the symbols of the Orthodox A formula have been dealt with, if $p > 1$, E_1 is replaced by the merger of $E_1, E_2, \ldots, E_p$; and $E_2, \ldots, E_p$ are deleted. Now C_1 gives the address of the first symbol in the Polish formula (and D_1 that of the last).

To yield late-operator Forward Polish instead of early-operator, only a minor modification is needed: under (*d*) above, and under (*d*)(i), in place of "a weaker operator symbol" write "a weaker or equally weak operator symbol (other than the symbol '$-$' in case S_j is '$+$' or '$-$')."

III. Forward Polish to Orthodox A

This is a relatively simple case, not unlike I: the operators may similarly be stored up in a nesting list. However, provision must, of course, be made for inserting brackets where necessary; and since the associative influence of an operator extends in the result after it as well as before it the writing of an operator in the output does not mean that it can be cancelled immediately from the nesting list. Hence an extra provision must be made in the nesting list for putting a mark against entries to indicate that they have been "written."

The symbols of the Forward Polish formula $S_1S_2 \ldots S_m$ are examined in order and the following operations are carried out.

(*a*) If S_j is a diadic operator it is transcribed to the nesting list; but a R.H. bracket is written in the nesting list first if S_j is weaker than the operator which is the current last entry in the list, if any: in this case also a L.H. bracket is sent to output.

(*b*) If S_j is the monadic operator "*neg*," and if this is weaker than the operator which is the current last entry in the nesting list, a L.H. bracket is sent to output and a R.H. bracket is written in the nesting list: then, and in any case whether this is so or not, "*neg*" is transcribed to output and also "*neg*" is added to the nesting list, marked "written."

(*c*) If S_j is a number variable, it is transcribed to output: then if the last entry in the nesting list is an operator marked "written," it is cancelled; if it is a R.H. bracket it is transcribed to output. The next last entry is taken from the nesting list and treated in the same way, and so on until an operator not marked "written" is reached: this is sent to output but left in the list, marked "written." If no such operator is reached, the translation is complete.

A closely similar method can naturally be applied, used backwards, to the translation of Reverse Polish to Orthodox A: compare method 1 of II.

IV. Forward Polish to Reverse Polish

The simplest of all methods of converting from Forward Polish to Reverse or vice versa is simply to read the pertinent formula backwards: this is not quite accurate as it stands, since certain operators such as "$-$" and "$\uparrow$" are asymmetrical (Forward Polish "$- \, a \, b$" means "$a - b$" whereas Reverse Polish "$b \, a \, -$" means "$b - a$"), but it may be possible to allow for this in interpretation. The order of the number-denoting symbols is of course reversed. But where this procedure is unacceptable the following method is appropriate.

The Forward Polish formula $S_1S_2 \ldots S_m$ is taken symbol-by-symbol as before, using a nesting list with provision as in III for placing a mark against each entry. In this case a mark placed against an entry indicates that only one operand of the operator concerned remains to be completely written. As each symbol S_j is examined, operations are carried out as follows.

(*a*) If S_j is a diadic operator it is transcribed to the nesting list.

(*b*) If S_j is the monadic operator "*neg*" it is transcribed to the nesting list with a "mark" against it.

(*c*) If S_j is a number variable it is transcribed to output: then the last entry of the nesting list is transcribed to output if it is "marked," and similarly the next last, and so on until an unmarked entry is reached: this is "marked." If there is no unmarked entry translation is complete.

This procedure, perhaps somewhat surprisingly, translates early-operator Forward Polish into early-operator Reverse, and late-operator Forward Polish into late-operator Reverse; and intermediate forms into intermediate forms. A procedure which would translate, say, early-operator Forward into late-operator Reverse, or which would always give early-operator Reverse whatever the form of the original Forward, would need to be rather more complicated.

It is immediate from considerations of symmetry that an identical procedure used backwards—that is, reading and writing the relevant formulae from right to left—translates Reverse Polish to Forward Polish.

References

HAMBLIN, C. L. (1957). "An Addressless Coding Scheme based on Mathematical Notation," *W.R.E. Conference on Computing, Proceedings*, Weapons Research Establishment, Salisbury, South Australia.

HAMBLIN, C. L. (1957). "Computer Languages," *Australian Journal of Science*, Vol. 20, p. 135.

HAMBLIN, C. L. (1960). "GEORGE, an Addressless Coding Scheme for DEUCE," *Australian National Committee on Computation and Automatic Control, Summarised Proceedings of First Conference*, paper C6.1.

HAMBLIN, C. L., HUMPHREYS, H. L., KAROLY, G., and PARKER, G. J. (1960). "Considerations of a Computer with an Addressless Order Code" and "Logical Design for ADM, an Addressless Digital Machine," *Australian National Committee on Computation and Automatic Control, Summarised Proceedings of First Conference*, papers C6.2 and C6.3.

ŁUKASIEWICZ, J. (1921). "Logika dwuwartościowa" (Two-valued logic), *Przegląd Filozoficzny*, Vol. 23, p. 189.

ŁUKASIEWICZ, J. (1929). *Elementy logiki matematyczny* (Elements of mathematical logic), Warsaw.