A note on compiling arithmetic expressions

J. S. Rohl and J. A. Linn

Department of Computer Science, The University, Manchester M13 9PL

This note describes a technique for compiling arithmetic expressions to minimise temporary storing on a single accumulator machine.

(Received March 1970, Revised July 1971)

Most algorithms for compiling arithmetic expressions regard the operators + and − as being essentially the same: in the ALGOL Report they are given the same precedence, in an Operator Precedence context they have the same precedence relations. They have, though, some important differences: minus is neither commutative ($a - b \not\equiv b - a$), nor associative ($a - (b - c) \not\equiv (a - b) - c$). Similar differences apply between * and /.

In the field theory sense, there are two binary operators + and *; the 'reciprocal' operators − and / are unary. The expression $a - b$ is a convenient shorthand for $a + -b$, and a/b a shorthand for $a*/b$. (Note that this automatically gives the ALGOL interpretation to $a/b/c$.) These two binary operators are both commutative and associative so that terms of an expression can be reordered at will, as can the factors of a term. Consider first the terms of an expression. For example, using the symbol ≡ to show the equivalence between short and long-hand, and ⇛ to indicate transformation:

$$a - b - c*d \equiv a + -b + -c*d \Rrightarrow$$
$$-c*d + -b + a \equiv -c*d - b + a$$

It is the purpose of this note to point out that the same transformation can be produced by regarding all operators as being unary, and an expression as a concatenation of 'signed' terms. First it is necessary to insert, where applicable, an implicit + in front of the first term.

$$a - b - c*d \equiv +a - b - c*d \Rrightarrow -c*d - b + a$$

The implicit + is added in the examples below.

Phase 1

There are, of course, a number of transformations of any expression: the object is to produce a transformation which can be easily compiled into optimum code in the sense of minimising the storage of temporary results. The transformation may be performed explicitly and the transformed expression compiled by some other algorithm; alternatively, it may be merged with the compiling algorithm so that code corresponding to the transformation is produced. When we consider the transformation process separately then the optimum transformation is one which can be compiled optimally by a left-to-right algorithm.

Such a transformation algorithm is straightforward:

1. If the expression is a single term then the transformed expression is the same.
2. If the first term is not simple† (i.e. not a scalar, array element or constant) then the transformed expression is the same.
3. Otherwise, treat the rest of the terms as an expression (using these rules recursively) and then append the first term.

Thus, assuming all the named quantities to be scalars:

$$+z \Rrightarrow +z$$
$$+a + b \Rrightarrow +b + a$$
$$+2*x + 1 \Rrightarrow +2*x + 1$$
$$+1 + 2*x \Rrightarrow +2*x + 1$$
$$-a + b + c*d \Rrightarrow +c*d + b - a$$
$$+f0 + 4*f1 + f2 \Rrightarrow +4*f1 + f2 + f0$$
$$+x - y^2 \Rrightarrow -y^2 + x$$

Note that even when a transformation does actually occur (as in the second example above) the resulting expression does not necessarily give rise to the better code. Note also that, as in the last example above, the algorithm is as likely to produce a leading negated term as it is to eliminate it. We will return to this point later.

Similar manipulations can be performed at a lower level on the factors of a term. We must, however, postulate (for the moment) the existence of a unary multiplication operator and again will add it to all our examples. Thus we regard the term $c*d*e^2$ as $*c*d*e^2$, which is a combination of the signed factors $*c$, $*d$, $*e^2$ which may be reordered at will. A similar algorithm to that used on the terms of an expression would produce for the above example $*e^2*d*c$, which is the optimal form for compilation.

This manipulation is likely to produce an initial division operator. For example, $*2*x/y \Rrightarrow /y*x*2$. Clearly this is not in an optimal form (even less so than the original) unless a machine is provided with a 'load the reciprocal' order and (remembering that each term is signed) with a 'load the negative of the reciprocal' order. This is, of course, unlikely, but in any case a further transformation avoids the problem.

Phase 2

Phase 2 is required to transform any term whose initial factor includes a division sign.

This initial division can be replaced by a multiplication and its effect promulgated as follows:

1. Replace the initial division and all those divisions up to the next multiplication by multiplications.
2. Replace this multiplication (there will always be one) by a reverse divide operator (ϕ).
3. Leave all following operators unaltered.

Thus

$$*2*x/y \Rrightarrow /y*x*2 \Rrightarrow *y\phi x*2 \ .$$

An interesting side effect is that strings of divisions can be replaced by one reverse division and a string of multiplications. As an (unlikely) example:

$$*a/b/c/d/e \Rrightarrow /e/d/c/b*a \Rrightarrow *e*d*c*b\phi a.$$

The final result is the same as that obtained from the more usual way of writing such an expression, since

$$*a/(*b*c*d*e) \Rrightarrow /(*e*d*c*b)*a \Rrightarrow *(*e*d*c*b)\phi a$$

†More precisely a simple term is one which does not require the accumulator of the machine for its evaluation.

and the brackets in this final form are trivially recognised as redundant.

The initial minus can be eliminated by an equivalent procedure so that, for example $-1 + i \Rightarrow 1\,\theta\,i$. There is no guarantee, however, that there will be a plus sign to convert to a reverse minus. Consider, for example, $-b - a$. The algorithm would never reach beyond stage 1 and would produce $a + b$ which is, of course, the negative of the required expression. This situation is comparatively rare and the simplest solution is to negate the result. This may be necessary, for example, where the expression is to be exponentiated or used as an exponent. More often the effect of this negation can be propogated. There are two cases:

1. The expression is enclosed within brackets as an operand of another larger expression. The negation can be effected by reversing the plus or minus sign of the term containing this expression. For example:

$$+y + (-a - b) \Rightarrow +(-b - a) + y \Rightarrow (b + a)\,\theta\,y$$
$$+y - (-a - b) \Rightarrow -(-b - a) + y \Rightarrow (b + a) + y$$

   where the brackets on the final expression are of course redundant.

2. The expression is to be assigned (either explicitly in an assignment statement or implicitly in the parameter of a procedure call).

First we expand our view from expressions to assignments. The left-hand side variable is moved to the right-hand side with the store operator. Thus:

$$z = +x + y \Rightarrow +y + x \Rightarrow z$$

The rules for phase 2 as applied to the terms of an expression can be more explicitly stated:

1. Replace the initial minus sign and all those minus signs up to a plus or store by plus signs.
2. Replace this plus or store by reverse minus or store negative respectively.
3. Leave any following operator above.

For example:

$$x = -a - b \Rightarrow -b - a \Rightarrow x \Rightarrow +b + a \mathrel{-\!\!\Rightarrow} x$$

Thus the full algorithm assumes an order code with operators for LOAD, $+$, $-$, $*$, $/$, θ, ϕ, $\Rightarrow$ and $\mathrel{-\!\!\Rightarrow}$.

Implementation

As noted earlier, the transformation described above may not necessarily be performed: instead the compilation may be such that object program corresponding to the transformation is produced. The algorithm in this form has been implemented in a compiler to validate the ease of compilation into the order code of a new machine, MU5 (Kilburn, Morris, Rohl, and Sumner, 1968) being built in this department, and for an Atlas Autocode Compiler for the ICL 1900 Series. Since it requires a structural knowledge of any statement (whether a term is simple and so on) the algorithm is suited to a syntax directed approach, and the compilers have been written using the Compiler Compiler facilities of Atlas Autocode (Brooker, Morris, and Rohl, 1967) and SPG (Morris, Wilson, and Capon, 1970) respectively. Measurements on the first compiler indicate that compiling time generally increases by 10%, as compared with a straightforward, less efficient algorithm.

Some of the compilers for the MU5 machine will use the algorithm directly, since all the compilers will produce a common target language (CTL) as output rather than machine code. This CTL is at a fairly high level but contains no precedence rules so that the algorithm will be modified to insert brackets as necessary.

Conclusions

As indicated above the algorithm above requires the instructions LOAD, $+$, $-$, θ, $*$, $/$, ϕ, $\Rightarrow$ and $\mathrel{-\!\!\Rightarrow}$. (In fact, this note may be looked upon as a partial design of an order code.) Where the reverse operators are not available (e.g. the 1900 fixed point orders and the Atlas A orders) then phase 1 of the algorithm can be modified to avoid their use by a little more testing of the context.

For example, rule (2) of phase 1 may be modified to:

2. If the first term is not simple, or if it is but the second term is negative, then the transformed expression is the same.

This would in some cases result in less than optimum code.

Where a store negative is not provided, then two orders—a negate followed by a store—must be used. Since this can arise only in an expression with no $+$'s it is likely to be rare. It can, however, be serious in a statement such as $a = -b$, since the algorithm does not accommodate a load negative order.

The algorithm suffers from the disadvantage that the resulting code bears a complicated relationship to the source code. For example, even a straightforward expression $a + b + c$ is translated as $c + b + a$. Since the same algorithm applies to all subexpressions nested within brackets, the resulting translation is, in general, difficult to unravel. This should not affect the main body of users, who are not concerned with the translation, but will add a little difficulty to those who scan object code to see whether an inner loop can be speeded up by handcoding.

The rearrangement does have one further advantage, however. If we consider $i = i + 1$ as a typical incrementing instruction (rather than $i = 1 + i$) then the algorithm converts this to $1 + i \Rightarrow i$, and it is trivially easy to recognise this case where orders which leave their result in the store are available (as in the 1900 Series).

Acknowledgement

We should like to thank the Science Research Council who have supported one of us (JAL) in this work. It has been pointed out by the referee and others that this algorithm is quite similar to that used by Sheridan (1959).

References

Brooker, R. A., Morris, D., and Rohl, J. S. (1967). Compiler Compiler Facilities in Atlas Autocode, *The Computer Journal*, Vol. 9, No. 4.

Kilburn, T., Morris, D., Rohl, J. S., and Sumner, F. H. (1968). A System Design Proposal, Proc. IFIP 1968.

Morris, D., Wilson, I. R., and Capon, P. C. (1970). A System Program Generator, *The Computer Journal*, Vol. 13, No. 3.

Sheridan, P. B. (1959). The Arithmetic Translator—Compiler of the IBM Fortran Automatic Coding System, *CACM*, Vol. 2, No. 2.