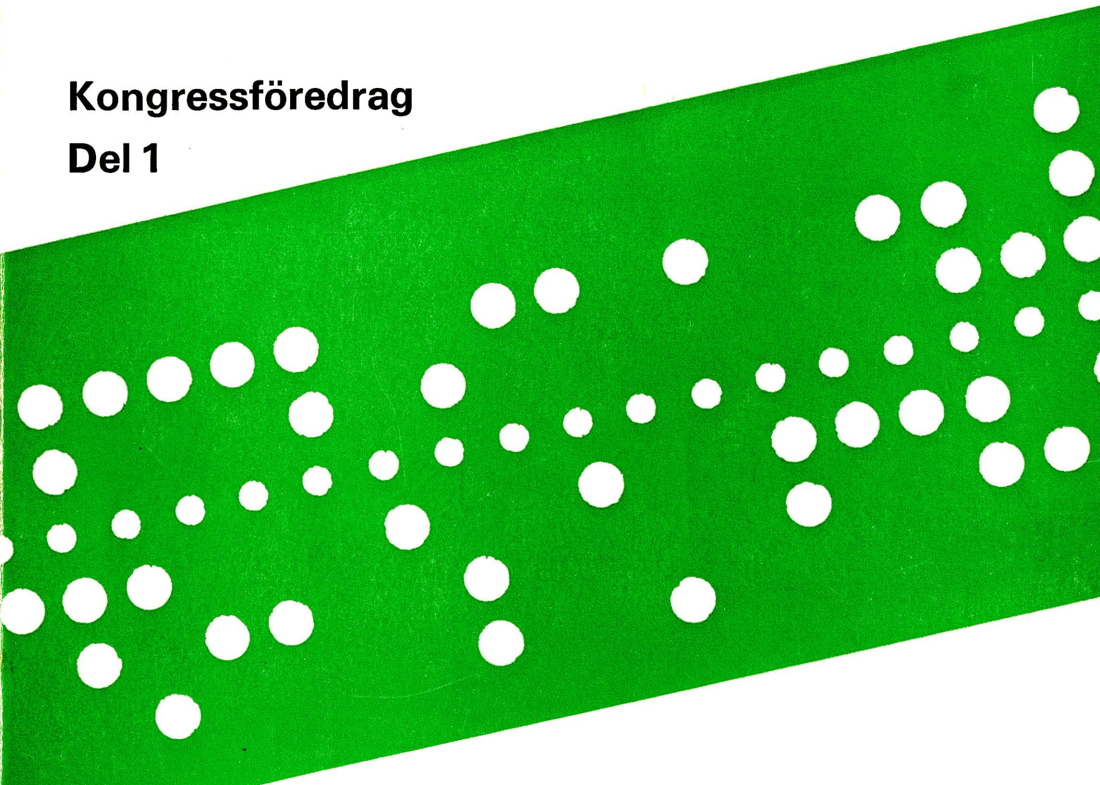
NordDATA-69

11-14 juni · Stockholm

Kongressföredrag

Del 1



ON ARITHMETIC EXPRESSIONS AND TREES

Roman R Redziejowski, IBM Nordic Laboratory, Lidingö, Sweden (presently on leave, at the Laboratory for Pulse and Digital Techniques, The Technical University of Denmark, Lyngby, Denmark).

1. INTRODUCTION.

The structure of an arithmetic expression is often represented in the form of a tree. By projecting the vertices of such a tree on a straight line, one can represent also an order of evaluating this expression, and thus a certain program of the computation. As shown below, some properties of the program, such as its length and storage requirement are then represented in a quite straightforward manner. This may help to a better understanding of certain relationships, and be of some use for the compiler design.

2. BASIC CONCEPTS.

To evaluate an expression, e.g. ((a+b)-c×d)/(e×(f-g)), one has to compute a number of partial results: A=c×d, B=a+b, C=B-A, D=f-g, E=e×D, and the final result F=C/E, in a number of elementary steps, or operations. This process can be represented by a directed graph, as in Fig.1. Each vertex of this graph represents one elementary operation. Each arc represents one partial result, and always leads from the place where the partial result is used to the place where it is computed.

In the following, the operations and the vertices corresponding to them are denoted by letters p,q,r,...,x,y,z, sometimes followed by indices, e.g. x[1],x[2],x[i],etc. The arcs are written in the form (x,y), where x is the initial, and y the terminal vertex of the arc.

A vertex r having the property that for every vertex x≠r there exists a path from x to r is called here a root of the graph; it is easy to see that the vertex corresponding to the final operation is such a root.

A vertex that is not the initial vertex of any arc is called here a leaf of the graph; it is easy to see that each operation performed entirely on initial data is represented by such a leaf.

A finite graph with a root r such that:
(a) r is not the terminal vertex of any arc,
(b) every vertex x≠r is the terminal vertex of exactly one arc
is called here a tree. It is easy to see that the graph such as in Fig.1 will always be a tree.

The structure of a tree (as well as that of the expression) is known to be recursive. In particular, every vertex of a tree is the root of some subtree (every partial result is the final result of some partial computation). In the following, the subtree having a vertex x as its root is called the x-subtree, and denoted T(x).

By removing from a tree T its root, we obtain a collection of $n \geq 0$ disjoint trees. In the following, we call these trees the main subtrees of T.

3. LINEUP OF A TREE.

Choosing a feasible order for the operations is equivalent to a topological sorting of the tree of Fig.1, i.e. to placing its vertices in such a sequence that a vertex x precedes a vertex y if there is an arc from y to x, for every pair x,y. In the following, a sequence satisfying this requirement is called a lineup of the tree. It can be conveniently represented as in Fig.2. The diagram of Fig.2 can be regarded as a timescale representation of the computation; it reflects not only the succession of operations, but also the history of all partial results.

4. INTERMEDIATE STORAGE.

Suppose we take an instantaneous picture of the computation while executing an operation x. Such a picture will show some results of the previous operations being stored for further use and not participating in x. In Fig.2, these stored results are represented by arcs passing over the vertex x. If the number of these arcs is m, we must have at least m+1 storage elements at our disposal at the considered moment: m for the results already stored, and one for the result of x.

The size of the storage we have to reserve for the entire computation is thus determined by the maximum number of arcs passing over a single vertex in Fig.2; we call here this number the >width< of the lineup. It can be formally defined as follows.

DEFINITION. Let L=(x[1],x[2],...,x[n]) be a lineup of a tree T. For this lineup, let m[i] be the number of arcs (x[j],x[k]) such that k<i<j, for i=1,2,...,n. The width of L, denoted W(L), is then defined as the maximum of m[i] for i=1,2,...,n.

The problem of minimizing the intermediate storage is thus reduced to one of finding a minimum width lineup of a given tree.

5. THE >MINIMUM WIDTH< THEOREM.

The minimal width we can arrive at by ordering a tree in different ways obviously depends upon the complexity of the tree. For our purpose, we can characterize this complexity by a number that is defined as follows.

DEFINITION. The _complexity_ of a tree T, denoted C(T), is defined recursively as:
C(T)=max(0,C(T[0])+0,C(T[1])+1,...,C(T[k])+k),
where T[0],...,T[k] are all the main subtrees of T, in such an order that $C(T[0]) \geq C(T[1]) \geq \ldots \geq C(T[k])$.

There exists the following relationship between the complexity C(T) and the width of any lineup of T:

THEOREM 1. For every tree T and its lineup L holds $W(L) \geq C(T)$.
Proof of this theorem can be found in [1].

If T is understood to represent a computation, the theorem states that, whatever is the order of operations, we must have at our disposal at least C(T)+1 storage cells for the partial results.

6. MINIMUM WIDTH LINEUP.

A lineup having the minimal width stated by Theorem 1 can be constructed for a given tree T as follows.

ALGORITHM 1. Proceeding from leaves towards the root, construct, for each vertex x, a lineup of the subtree T(x) as stated under (a) and (b) below.
(a) For a leaf x, take (x) as the lineup.
(b) For a non-leaf x, take the concatenation of L[0], L[1],...,L[k] and (x), where L[0],...,L[k] are previously obtained lineups of all the main subtrees of T(x), ordered according to their decreasing widths. (For illustration see Fig.3).

If T is understood to represent a computation, the procedure can be viewed as constructing, step by step, some partial programs. In the rule (b), L[0],...,L[k] represent programs for evaluating the arguments of x; we combine them in one program in the order of their decreasing storage requirements. This agrees well with the common sense: the available storage will shrink as we evaluate and store the arguments of x.

7. LENGTH OF PROGRAM.

Let us imagine now the following idealized one-address machine. The storage of the machine consists of a number of _memory cells_ and one cell called the _accumulator_. The memory cells are identified by their _addresses_, here denoted by letters: a,...z,A,...Z. The machine can execute the following instructions (c denotes an arbitrary address):

- load accumulator from the cell c;
- store the contents of accumulator in the cell c;
- add the contents of c to that of accumulator;
- subtract the contents of c from that of accumulator;
- multiply the contents of accumulator by that of c;
- divide the contents of accumulator by that of c.

In the following, these instructions are written as: [c, c], +c, -c, ×c, and /c, respectively. A _program_ is writen as a sequence of such instructions.

An individual operation of our computation can be now programmed as a sequence of three instructions, e.g. [c×dA], [a+bB], etc. Some operations can be programmed in two alternative ways, e.g. [a+bB] and [b+aB], [e×DE] and [D×eE], etc. A complete program for the computation represented by Fig.2 may thus appear as follows:

[c×dA][a+bB][B-AC][f-gD][e×DE][C/EF].

However, in this program the pair B][B is redundant. We can keep B in the accumulator in between the operations:

[c×dA][a+b-AC][f-gD][e×DE][C/EF],

thus reducing the total length of the program (and its execution time).

It is easy to see that some more such pairs can be created, and the program further improved, if we execute the operations in a slightly changed order. It is of an obvious interest to find an order that would maximize the number of such pairs, and thus minimize the length of the program.

To find such an order, let us mark in our tree every argument of an operation that can be placed in the accumulator prior to the operation. In our case, these marked arcs will be B,C and D (see Fig.4). Choosing some order of operations, we obtain a diagram such as in Fig.5. It is easy to see that in Fig.5 every partial result that need not be transferred to a memory cell and back is represented by a marked arc appearing in the same line as all the vertices. In the following, we call such an arc a >bridge<.

DEFINITION. Let T be a tree with some arcs marked. Let L=(x[1],x[2],...,x[n]) be a lineup of T. A _bridge_ is defined as a marked arc of the form (x[i-1],x[i]), where 1<i≤n.

The problem of constructing the optimal order of operations is thus reduced to one of constructing, for a given tree T and marking of its arcs, a lineup containing the greatest possible number of bridges.

8. THE >BRIDGE< THEOREM.

The maximum possible number of bridges is stated by the following theorem.

THEOREM 2. For every tree T and marking of its arcs, the number of bridges in any lineup of T is not greater than n-k, where n is the number of vertices in T, and k the number of leaves in the graph <u>T</u> obtained from T by removing all unmarked arcs.
<u>Proof</u>. Let L=(x[1],x[2],...,x[n]) be a lineup of T. Consider all pairs (x[i-1],x[i]). There are n-1 such pairs. If x[i] is a leaf of <u>T</u>, the pair is not a marked arc, and thus not a bridge. This must be true for at least k-1 such pairs; thus, there cannot be more than n-k bridges.

If T is understood to represent a computation, the theorem states that the program must contain at least k pairs of >store-load< instructions (since initially there is n such pairs, one for each operation).

9. LINEUP WITH MAXIMAL NUMBER OF BRIDGES.

A lineup containing exactly n-k bridges can be constructed as follows.

ALGORITHM 2. Starting from the root of T, follow the arcs as far as possible, and remember all arcs going >aside< from the followed path. Upon coming to a leaf, return to one of the remembered arcs, and repeat the procedure until all arcs have been traversed. During the whole travel, observe the following rule: if a vertex x is the initial vertex of a marked arc, always leave x by means of this arc. The sequence of vertices visited in this way gives the required lineup, right-to-left.

In terms of our interpretation, the above procedure corresponds to writing the program backwards. We program first the final operation, and ask what we would like to have in the accumulator to start this operation with (this corresponds to finding a marked arc leaving r). We find then the operation that places in the accumulator exactly what we need, program it as one before the last and so on.

10. FINAL REMARKS.

(a) The results presented here may be applied to evaluate the effectivness of some compiling algorithms already in use. One can see, in particular, that the order of operations generated by the stack method, or the Reverse Polish notation, is usually not the optimal one, in both respects discussed here.

(b) The problems discussed seem to be common for all situations where a tree structure is being treated in a sequential manner. Thus, the width of a lineup has an obvious relationship to the size of a stack neede to traverse a tree structure in a certain order, while the ideas appearing in sections 7 through 9 are related to techniques of storing tree structures in a linear memory.

REFERENCE:
1. R.R.REDZIEJOWSKI: On Arithmetic Expressions and Trees, Comm.ACM, Vol.12, No 2 (February 1969), page 81.

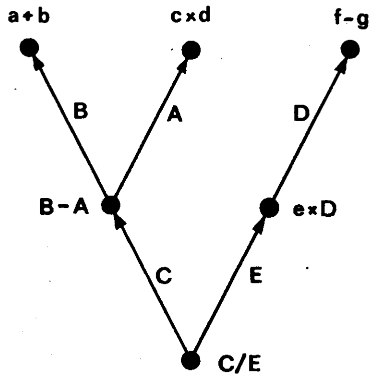


Fig. 1

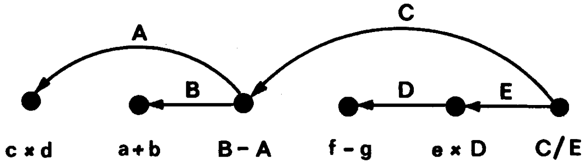
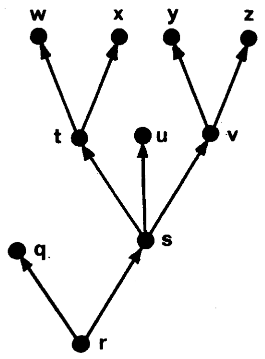


Fig. 2



vertex	lineup	width
w	w	o
x	x	o
y	y	o
z	z	o
t	wxt	1
u	u	o
v	yzv	1
s	wxtyzvus	2
q	q	o
r	wxtyzvusqr	2

Fig. 3

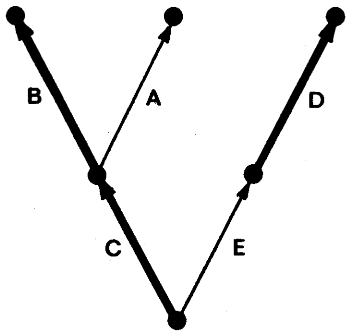


Fig. 4

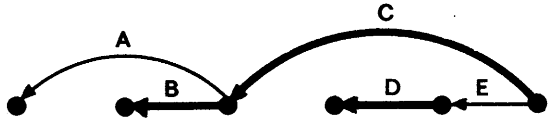


Fig. 5