stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IW 52/75 NOVEMBER

L.G.L.T. MEERTENS & J.C. VAN VLIET

REPAIRING THE PARENTHESIS SKELETON OF ALGOL 68 PROGRAMS: PROOF OF CORRECTNESS

Prepublication

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

AMS(MOS) subject classification scheme (1970): 68A20, 68A10

ACM-Computing Reviews-categories: 4.22, 5.24, 5.25

Repairing the parenthesis skeleton of ALGOL 68 programs: proof of correctness [*)]

by

L.G.L.T. Meertens & J.C. van Vliet

ABSTRACT

The error-correction problem for structures of nested parentheses bears some resemblance to the problem of finding the shortest path in a directed graph. Not surprisingly, the algorithm for determining the optimal "reparation" lends itself to a neat and concise (min, +)-algebraic notation, thus permitting abstraction from the representation used in a practical implementation. The correctness proof for this abstract algorithm may then be given with only a few inductive assertions.

KEY WORDS & PHRASES: *ALGOL 68, parenthesis structure, error-correction, minimax algebra, directed graph, correctness proof, inductive assertions.*

---

*) This paper is not for review; it is meant for publication elsewhere.

0. INTRODUCTION

In ALGOL 68 an abundance of parenthesized constructs is available which may occur nested within one another. Most of these constructs are relevant to the range structure. Therefore, it is not uncommon for compilers to give up if, after the first pass, the parenthesis structure of the source program has been found incorrect. This has the unpleasant consequence that the opportunity is lost to give further error messages that may assist in reducing the number of runs before the program is syntactically correct.

Unavoidably, a compiler which does not give up has to impose some "interpretation" on an incorrect source text. It is convenient, to say the least, if this interpretation is the same for successive passes. One obvious way to achieve this is to "correct" the source text so as to comply with the chosen interpretation. Since the compiler cannot, of course, find out what is correct in terms of the intention of the programmer, we prefer the less suggestive term "repairing" of incorrect source texts.

The way in which the repairing algorithm presented here was obtained is remarkable enough to merit some words. Dissatisfied with an earlier algorithm [2] , which appeared to perform similar computations over and over, thus wasting time and space, we tried to contract similar states to generalized ones. This resulted in an obscure network of molecular actions connected by jumps. By studying the flow of control, we brought the network into a shape where the only control structures were simple selection and repetition. In spite of its seeming simplicity, however, the algorithm evaded our attempts to show convincingly its correctness, the bottle-neck being the formulation of invariants and other assertions, which grew into algorithms themselves.

A break-through occurred when we hit upon the idea of viewing the algorithm not as manipulating a graph, but as performing operations on a matrix representing that graph. With a suitable notation, the correctness proof turned out to be feasible.

1. FORMULATION OF THE PROBLEM

In this paper, we concentrate on the parenthesis skeleton, i.e., the succession of parentheses occurring in the source text. It is assumed that these parentheses are numbered from 1 to n. Moreover, there is a predicate "matching" which tells if two given parentheses match each other. For the sake of convenience, the predicate is defined on a pair of indices i and j, $1 \le i < j \le n$, rather than on the parentheses themselves. As a final assumption, the correctness of a parenthesis skeleton is determined by the following syntax:

```
correct skeleton: enclosure sequence option.
enclosure: STYLE opening parenthesis,
           correct skeleton,
           STYLE closing parenthesis.
```

Here, of course, a STYLE-opening-parenthesis and a STYLE-closing-parenthesis match. It should be noted that this formulation does not require that the two sets of marks representing opening and closing parentheses, are disjoint. (There are several problems in bringing ALGOL-68's parentheses in this framework; for the time being, however, we abstract from these complications.)

A "reparation" consists of the marking of a number of parentheses such that deletion of these parentheses from the skeleton results in a correct-skeleton. (Whether in practice these parentheses will be actually deleted or matching ones should be inserted, is a matter which falls outside the scope of this paper.)

An "optimal reparation" is a reparation which marks the least possible number of parentheses. The problem is to determine an optimal reparation of a given parenthesis skeleton.

A "segment" of a skeleton may be coded as a pair of integers (a, b), with $0 \le a \le b \le n$, and then comprises the parentheses numbered from a+1 to b (so the whole skeleton is given by (0, n) and each segment (a, a) is empty). It is called "neutral" if it may be produced by the above syntax for correct-skeleton.

The "error value" of a segment is the smallest number of parentheses that has to be deleted from it in order to neutralize it.

Obviously, there is a connection between the error value of the segment (0, n) and the optimal reparation. In the sequel, we concentrate on algorithms for determining the error value; an optimal reparation is then easily obtained as a byproduct.

2. ERROR VALUE AS SHORTEST PATH

In order to determine the error value of a segment, we may proceed as follows:

If the segment is empty, its error value is 0.

Otherwise, draw an arc over each parenthesis and label it with a length of 1. Moreover, for each pair of matching parentheses, determine the error value of the (shorter) segment enclosed between these parentheses and draw an arc over the whole segment, including the matching parentheses, labelled with that error value. Then the error value of the segment is the length of the shortest path obtainable by following arcs from the beginning to the end, where the length of a path is the sum of the lengths of the individual arcs involved. An example is given in figure 1.

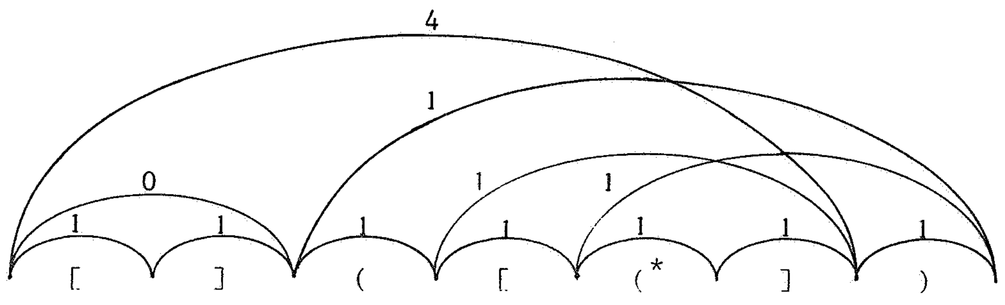

Fig. 1. A segment with error value 1. It can be neutralized by deleting the marked parenthesis.

Obviously, the error value of a neutral segment is 0, and the arcs of the shortest path parse it as a sequence of enclosures. That in general the error value, say e, determined by this procedure is correct may easily be proved (by induction on the length of the segments) by observing, first, that a neutral segment may indeed be obtained by deleting all parentheses jumped over in a single-parenthesis arc and by neutralizing in the same way all inner segments of larger arcs — which leads obviously to the deletion of exactly e parentheses--and, second, that each neutralizing action deleting d parentheses determines a path of length d, so $e \le d$.

It can be seen that this procedure consists of the construction of an acyclic directed graph whose edges (the arcs) are labelled with the length of the shortest path in a subgraph. It is not difficult to formulate it as an algorithm which avoids recomputing the error value of subsegments. However, further optimizations are possible (and desirable), and for the purpose of establishing the correctness of these optimizations a different representation is introduced.

3. (min, +)-ALGEBRAIC NOTATION

A convenient way to express algorithms for finding the shortest path in a directed graph is to represent the graph by a matrix. Let each edge of the graph be labelled with a length (a nonnegative real--or, in our application, integral--number). In order to obtain a uniform process, unconnected vertices are considered connected by an edge of length ∞, so that the set of possible lengths is given by $\{x \in \mathbb{R} : x \ge 0\} \cup \{\infty\}$. Also, each vertex is connected to itself by an edge of zero length. Now we can represent the graph as an $(n+1) \times (n+1)$ matrix A by numbering the vertices from 0 to n and taking as entry A[i, j] the length of the edge from vertex i to vertex j. An example of a graph and its matrix representation is given in figures 2a and 2b.

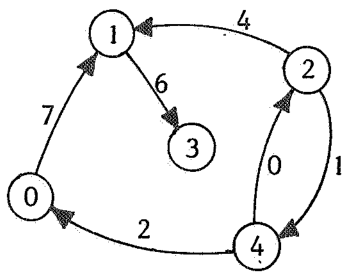

Fig. 2a. A directed graph.

$$\begin{array}{c|ccccc|}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & 0 & 7 & \infty & \infty & \infty \\
1 & \infty & 0 & \infty & 6 & \infty \\
2 & \infty & 4 & 0 & \infty & 1 \\
3 & \infty & \infty & \infty & 0 & \infty \\
4 & 2 & \infty & 0 & \infty & 0 \\
\hline
\end{array}$$

Fig. 2b. Matrix representation of the graph in fig. 2a.

The most important operations on lengths are: taking their minimum and taking their sum. For these operations we introduce the following notation:

> For two lengths x and y, the minimum of x and y is denoted $x + y$ and their sum is denoted $x \times y$, where, of course, $x + \infty = \infty + x = x$ and $x \times \infty = \infty \times x = \infty$.*)

It is easily checked that the operations + and × obey the laws of commutativity, associativity and even distributivity, reason why this notation is called algebraic:

$$x + y = y + x$$
$$x + (y + z) = (x + y) + z$$
$$x \times y = y \times x$$
$$x \times (y \times z) = (x \times y) \times z$$
$$x \times (y + z) = x \times y + x \times z$$

From the identities $x + \infty = x \times 0 = x$ and $x \times \infty = \infty$, it is seen that ∞ takes the role of the zero element and 0 that of the unity element.

*) For indices, "+" retains its conventional meaning. It will always be clear from the context which operation is intended.

It is useful to keep also the following facts in mind:

If $x + y = z$ then $z \le x$ and $z \le y$; moreover, $z = x$ or $z = y$.
If $x \le y$, then $x + y = x$.

4. MATRIX OPERATIONS

Using this notation, we can now also define "addition" and "multiplication" for matrices:

$A + B$ is defined by $(A + B)[i, j] = A[i, j] + B[i, j]$.
$A \times B$ is defined by $(A \times B)[i, j] = \sum_k A[i, k] \times B[k, j]$,
where the summation $\sum$ is performed using the + operation introduced in section 3.

As is to be expected, matrix addition is associative and commutative, matrix multiplication is associative but not commutative, and together the operations are distributive. Also, the matrix I having entries 0 on the main diagonal and entries ∞ elsewhere, satisfies $I \times A = A \times I = A$, and so serves as the identity matrix. From the fact that all matrices have entries 0 on the main diagonal, it also follows that $I + A = A$.

The powers of a matrix can be defined, as usual, by $A^0 = I$, $A^{i+1} = A^i \times A$. The most important operation, for our purposes, is given by

$$A^* = A^0 + A^1 + \dots + A^n.$$

This can be interpreted as: $A^*[i, j]$ is the length of the shortest path from i to j by taking 0, 1, ... or n steps. Since the shortest path between any two vertices in a graph with n+1 vertices consists of at most n edges, $A^*[i, j]$ is the length of the shortest path from i to j. The operation * is usually called "transitive closure".

From $I + A = A$, it follows that $A^i + A^{i+1} + \dots + A^n = A^i \times (I + A) + \dots + A^n = A^i \times A + \dots + A^n = A^{i+1} + \dots + A^n$, so $A^* = A^n$.

In the graphs of our application, there never is an edge from i to j if $i > j$, so the lower triangles of the matrices are filled with ∞'s. (This is equivalent to the statement that the graphs are not only acyclic, but also topologically sorted.) This property, which is preserved by the operations $+$, $\times$ and *, will be tacitly assumed from now on. (In fact, in the algorithm no reference is ever made to an element $A[i, j]$ with $i > j$.)

From this property we derive

LEMMA 1. $A^*[i, j]$ *depends only on the sub-matrix* $A[i : j, i : j]$.

PROOF. Since $A^* = A^n$, it suffices to prove that $(A \times B)[i, j]$ depends only on $A[i : j, i : j]$ and $B[i : j, i : j]$.

$$(A \times B)[i, j] = \sum_{k=0}^{n} A[i, k] \times B[k, j] = \sum_{k=i}^{j} A[i, k] \times B[k, j],$$

since all terms with $i > k$ or $k > j$ have a factor equal to ∞ and, therefore, do not contribute to the sum.

Since $A^* = A^n$, $A^*[i, j]$ can be expressed as

$$\sum_{(p)} A[i, p_1] \times A[p_1, p_2] \times \dots \times A[p_{n-1}, j],$$

where the summation is performed over all $(n+1)$-tuples (p_k) with $i = p_0 \le p_1 \le p_2 \le \dots \le p_{n-1} \le p_n = j$. Since $A[x, x] = 0$, the unit element of multiplication, factors with equal indices may be omitted, so we obtain

$$A^*[i, j] = \sum_{m} \sum_{(p)} A[i, p_1] \times \dots \times A[p_{m-1}, j],$$

where the summation is performed over all $(m+1)$-tuples (p_k) with $i = p_0 < p_1 < \dots < p_{m-1} < p_m = j$, for $m \ge 2$ and, for $m = 1$, over $i = p_0 \le p_1 = j$. (We need $\le$ here, rather than $<$, since, otherwise, all factors would be omitted from the product if $i = j$, leaving "nothing" to be summed.) In the sequel, whenever the above notation is used, the summation convention mentioned here is understood.

For the sum of the terms of the above summation, but now excluding the term $A[i, j]$, the notation

$$A^+[i, j] = \sum_{m\geq 2} \sum_{(p)} A[i, p_1] \times \ldots \times A[p_{m-1}, j]$$

will be used [*]. So we have

$$A^*[i, j] = A[i, j] + A^+[i, j].$$

For $i = j$ or $i+1 = j$, no (p_k) satisfies the conditions of the summation convention, so in that case $A^+[i, j] = \infty$, from which we also conclude that then $A^*[i, j] = A[i, j]$.

For the $^+$ operation, we can prove

LEMMA 2. *If* $A^*[p, q] = B^*[p, q]$ *for all* p, q *with* $i \leq p \leq q \leq j$, *with the possible exception of* $p = i$ *and* $q = j$, *then* $A^+[i, j] = B^+[i, j]$.

PROOF. $A^+[i, j] = \sum_{m\geq 2} \sum_{(p)} A[i, p_1] \times \ldots \times A[p_{m-1}, j] \geq$

$\sum_{m\geq 2} \sum_{(p)} (A[i, p_1] + A^+[i, p_1]) \times \ldots \times (A[p_{m-1}, j] + A^+[p_{m-1}, j]) =$

$\sum_{m\geq 2} \sum_{(p)} A^*[i, p_1] \times \ldots \times A^*[p_{m-1}, j] = \sum_{m\geq 2} \sum_{(p)} B^*[i, p_1] \times \ldots \times B^*[p_{m-1}, j] =$

$\sum_{m\geq 2} \sum_{(p)} (\sum_{m'} \sum_{(p')} B[i, p'_1] \times \ldots \times B[p_{m'-1}, p_1]) \times \ldots \times$

$(\sum_{m'} \sum_{(p')} B[p_{m-1}, p'_1] \times \ldots \times B[p'_{m'-1}, j]) \geq$

$\sum_{m\geq 2} \sum_{(p'')} B[i, p''_1] \times \ldots \times B[p''_{m-1}, j] = B^+[i, j].$

(The unequalities are all special cases of $x \geq x + y$). Similarly, we find $B^+[i, j] \geq A^+[i, j]$, so $A^+[i, j] = B^+[i, j]$.

[*] The A^+, thus defined, does not satisfy $I + A^+ = A^+$, so it is not the matrix representation of a graph. It is purely introduced for proof-technical reasons.

5. NOTATIONS FOR INDUCTIVE ASSERTIONS AND VERIFICATION RULES

The correctness proof of the algorithm is given by inserting a number of assertions in the text and by verifying next a number of verification conditions, generated by verification rules. These rules have been chosen such that the number of inserted assertions is limited.

Each assertion is enclosed between { and }. Compound assertions are written, e.g., $\{A, B, C\}$, meaning $\{A \wedge B \wedge C\}$. If A is an assertion, then $A \varsigma (v := e)$ stands for A with e substituted for v. This substitution should be taken intentionally rather than literally. For example, if A stands for $a[4] \times y = z$, then $A \varsigma (a[2 \times 2] := x + 1)$ stands for $(x + 1) \times y = z$.

In a series, the statements are not separated by go-on-symbols, but by assertions. The correctness of a series is checked by verifying the correctness of each statement with respect to its surrounding assertions, or, in the notation of the verification rules:

$$\{A_0\}\ S_1\ \{A_1\}\ \ldots\ \{A_{p-1}\}\ S_p\ \{A_p\}:$$

$$\{A_0\}\ S_1\ \{A_1\},\ \ldots\ ,\ \{A_{p-1}\}\ S_p\ \{A_p\}.$$

For the other types of statements occurring in our algorithm, we have the following verification rules:

$\{A\}\ v := e\ \{B\}:\ \{A\}\ \{B \varsigma (v := e)\}.$

$\{A\}\ \underline{\text{if}}\ C\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}}\ \{B\}:$

  $\{A, C\}\ S_1\ \{B\},\ \{A, \neg C\}\ S_2\ \{B\}.$

$\{A\}\ \underline{\text{for}}\ v\ \underline{\text{from}}\ f\ \underline{\text{by}}\ -1\ \underline{\text{to}}\ t\ \{B\}\ \underline{\text{do}}\ S\ \underline{\text{od}}\ \{C\}:$

  $f \geq t-1,$

  $\{A\}\ \{B \varsigma (v := f)\},$

  $\{B, v \leq f, v \geq t\}\ S\ \{B \varsigma (v := v-1)\},$

  $\{B \varsigma (v := t-1)\}\ \{C\}.$

These rules have to be applied repeatedly, until the case is reached where no statement separates the assertions (as in the rule for assignations). The verification condition is then that the first assertion ("antecedent") implies the second one ("consequent"). It should be remarked that the last rule given is not the most general one, but that it is tailored to the loop-clause with a step of -1 and with $f \geq t-1$. Also, the rule for the conditional-clause is only given for clarity's sake; it plays no role in the proof. (Of course, these rules are related to those of Hoare's axiomatic method [1], but instead of deriving correctness of the whole text by bottom-up application of rules, verification conditions are generated by top-down application.)

In order to benefit more fully from the rule for assignations, a collateral assignation, like

$$v_1 := e_1,\ v_2 := e_2$$

is used, with the syntactic position of a statement. It will only be used if $v_1 :\neq: v_2$ and e_1 does not depend on v_2 nor e_2 on v_1, or, in other words, only if $A ; (v_1 := e_1,\ v_2 := e_2) \equiv A ; (v_1 := e_1) ; (v_2 := e_2) \equiv A ; (v_2 := e_2) ; (v_1 := e_1)$. Furthermore,

$$\forall\ i \in [p : q]\colon v_i := e_i$$

is used, with the meaning

$$v_p := e_p,\ v_{p+1} := e_{p+1},\ \dots,\ v_q := e_q.$$

6. THE ALGORITHM

In order that an algorithm for determining the error value can be formally proved correct, a more formal definition of error value is needed. Using the (min, +)-algebraic notation, we can write:

$$\text{error value}[i,\ j] = Q^*[i,\ j], \text{ where}$$

$$Q[i,\ j] = \begin{cases} 0 \text{ if } i = j, \\ 1 \text{ if } i+1 = j, \\ Q^*[i+1,\ j-1] \text{ if } i+1 < j \wedge \text{matching}(i+1,\ j), \\ \infty \text{ otherwise.} \end{cases}$$

Obviously, Q is the matrix representation of the graph introduced in section 2, where the edges are directed from left to right. That the recursion in this definition is well grounded follows from Lemma 1, using induction on $j - i$.

The algorithm computes, column by column, a matrix R such that R = Q. It runs, to a first level of approximation, as follows:

```
R[0, 0]:= 0
{R[0 : 0, 0 : 0] = Q[0 : 0, 0 : 0]}
for i from 1 by 1 to n
{R[0 : i-1, 0 : i-1] = Q[0 : i-1, 0 : i-1]}
do Addition of parenthesis(i) od
{R[0 : n, 0 : n] = Q[0 : n, 0 : n]}
R:= $R^*$
{R = $Q^*$}.
```

It is the part "Addition of parentheses(i)" that will be worked out further. The verification condition for this part, using the obvious equivalent for a step +1 of the loop-clause rule given above, is

```
{R[0 : i-1, 0 : i-1] = Q[0 : i-1, 0 : i-1], $i \ge 1$, $i \le n$}
Addition of parenthesis(i)
{R[0 : i, 0 : i] = Q[0 : i, 0 : i]}.
```

For simplicity, within this part $1 \le i \le n$ will be taken as a global fact, so there is no need to drag it along all through the proof.
The correctness proof given below uses the notations R', Q' and R_i. These notations are introduced to let the proof do double duty; for the time being, they should be read as R, Q and R, respectively.

Let RQ stand for

$$R'[0 : i-1, 0 : i-1] = Q'[0 : i-1, 0 : i-1]$$

and PT for

$$R'[j : i, i] = Q'[j : i, i],$$

$$\forall k \in [1 : j]: T[k] + \sum_{z=k+1}^{j} R_i[k, z] \times R_i^*[z, i-1] = R_i^*[k, i-1].$$

(In view of the fact that the lower triangles are understood to contain entries ∞ , RQ can be considered a convenient abbreviation for:

$$R'[p, q] = Q'[p, q] \text{ for all } p, q \text{ such that } 0 \leq p \leq q \leq i-1.)$$

The part "Addition of parenthesis(i)" may then be written as follows:

```
{RQ}
R[i-1 : i]:= (1, 0), ∀ k ∈ [1 : i-2]: T[k]:= ∞, T[i-1]:= 0
{RQ, PT ς (j:= i-1)}
for j from i-1 by -1 to 1
{RQ, PT}
do R[j-1, i]:= (matching(j, i) | T[j] | ∞),
   ∀ k ∈ [1 : j-1 : T[k]:= T[k] + $R_i[k, j] \times T[j]$
od
{RQ ς (i:= i+1)}
```

7. VERIFICATION

By applying the verification rules, performing substitutions and simplifying the result where appropriate, we obtain the following five verification conditions, each time followed by a proof of the non-obvious parts:

V1: {RQ} {RQ, (R'[i-1 : i, i] = Q'[i-1 : i, i]) ς (R[i-1 : i, i]:= (1, 0)),

$$\forall k \in [1 : i-2]: \infty + \sum_{z=k+1}^{i-1} R_i[k, z] \times R_i^*[z, i-1] = R_i^*[k, i-1],$$

$$0 + \sum_{z=i}^{i-1} R_i[i-1, z] \times R_i^*[z, i-1] = R_i^*[i-1, i-1]\}.$$

Since R' = R and Q' = Q, the assertion R'[i-1 : i, i] = Q'[i-1 : i, i] stands for R[i-1 : i, i] = Q[i-1 : i, i], which, after substitution of (1, 0) for R[i-1 : i, i], reads as (1, 0) = Q[i-1 : i, i]. That Q[i-1, i] = 1 and Q[i, i] = 0 follows immediately from the definition of Q.

$R^*_i[k, i-1]$ can be written as $\sum_m \sum_{(p)} R_i[k, p_1] \times \ldots \times R_i[p_{m-1}, i-1]$.

Since $k < i-1$, we have $k < p_1$ for each p_1 also. By summing first over each $p_1 = z$, $k < z \le i-1$ and factoring out $R_i[k, z]$, we obtain

$$R^*_i[k, i-1] = \sum_{z=k+1}^{i-1} R_i[k, z] \times (\sum_m \sum_{(p)} R_i[z, p_2] \times \ldots \times R_i[p_{m-1}, i-1]) =$$

$$\sum_{z=k+1}^{i-1} R_i[k, z] \times R^*_i[z, i-1].$$

(The term ∞ appearing in V1 does not contribute to the sum.)

Since $R^*_i[i-1, i-1] = 0$ and $0 + x = 0$ for any x, the last component of V1's consequent holds also.

V2: $i-1 \ge 1-1$.

This follows from the global fact that $1 \le i \le n$.

V3: {RQ, PT ς (j:= i-1)} {RQ, PT ς (j:= i-1)}.

V4: {RQ, $R'[j : i, i] = Q'[j : i, i]$,

$$\forall k \in [1 : j]: T[k] + \sum_{z=k+1}^{j} R_i[k, z] \times R^*_i[z, i-1] = R^*_i[k, i-1],$$

$j \le i-1$, $j \ge 1$}

{RQ, $(R'[j-1, i] = Q'[j-1, i])$ ς $(R[j-1, i] := (\text{matching}(j,i) \mid T[j] \mid \infty))$,

$R'[j : i, i] = Q'[j : i, i]$,

$$\forall k \in [1 : j-1]: T[k] + R_i[k, j] \times T[j] + \sum_{z=k+1}^{j-1} R_i[k, z] \times R^*_i[z, i-1] =$$

$$R^*_i[k, i-1]\}.$$

From $T[j] + \sum_{z=j+1}^{j} R_i[j, z] \times R^*_i[z, i-1] = R^*_i[j, i-1]$, we deduce

$T[j] = R^*_i[j, i-1]$ (the summation is empty).

Since $R' = R$ and $Q' = Q$, we have to show

$$Q[j-1, i] = (\text{matching}(j, i) \mid T[j] \mid \infty).$$

Using $T[j] = R^*_i[j, i-1]$, $R_i = R$ and $R[0 : i-1, 0 : i-1] = Q[0 : i-1, 0 : i-1]$ (RQ), we obtain the condition

$$Q[j-1, i] = (\text{matching}(j, i) \mid Q^*[j, i-1] \mid \infty),$$

which follows immediately from the definition of Q.
The last component of V4's consequent follows from the third component of its antecedent, together with $T[j] = R^*_i[j, i-1]$.

V5: $\{R'[0 : i-1, 0 : i-1] = Q'[0 : i-1, 0 : i-1],$
$R'[0 : i, i] = Q'[0 : i, i],$
$\forall k \in [1, 0]: \ldots\}$
$\{R'[0 : i, 0 : i] = Q'[0 : i, 0 : i]\}.$

8. OPTIMIZATION OF THE ALGORITHM

Just like a graph may be represented by a matrix, we may represent a matrix by a graph. Usually, the graph representation is more efficient if the matrix is sparse, i.e., contains many elements equal to ∞. Since the matrices occurring in the process are sparse indeed, a graph representation is used in the algorithm on a lower level of abstraction, where a matrix A is represented by associating with parenthesis j, say, the set E(A, j) of pairs (i, A[i, j]) with $i < j$ and $A[i, j] < \infty$. So, in this representation, the arcs as in figure 1 are pointing backwards rather than forwards. Logically, this is indifferent, but the effect is that the concrete implementation of the abstract algorithm is much more efficient than it would be otherwise. For example, the assignation

$$\forall k \in [1 : j-1]: T[k] := T[k] + R_i[k, j] \times T[j]$$

is implemented as

$$\forall (k, e) \in E(R_i, j): T[k] := T[k] + e \times T[j].$$

It may be seen that the graph representation, for sparse matrices R_i, is not only more efficient in space, but also in time.

Now we can make the following observation. Our algorithm computes $R = Q$, but we are not so much interested in Q, but rather in Q^*. For that reason, any R such that $R^* = Q^*$ will serve our purpose equally well. Hopefully, such an R can be made sparser even than Q.

Now, if we look at figure 3a, we see a skeleton where an arc is found of at most error value $a \times b \times 1$, given the fact that the two inner segments indicated by "..." have error values of a and b, respectively. This arc corresponds to a finite entry in Q. As can be seen from figure 3b, however, this entry is superfluous in R if we are only interested in $R^* = Q^*$, since the entry in R^* for this segment is already at most $a \times b \times 1$. Moreover, the two paths correspond to different interpretations (indicated by marking the incorrect brace with an asterisk), and of these, from a human point of view the one in figure 3b is preferable to that in figure 3a.

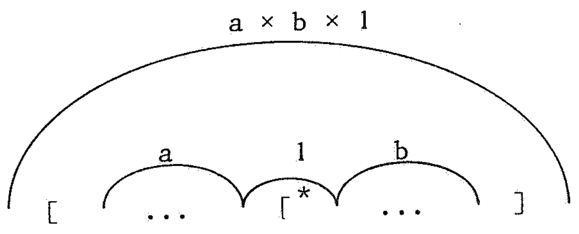

Fig. 3a. A segment with error value $\leq a \times b \times 1$.

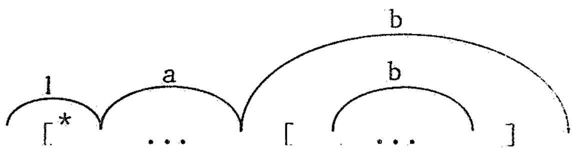

Fig. 3b. A different way of exhibiting that the segment in figure 3a has error value $\leq a \times b \times 1$.

In order to implement the improvement suggested by these considerations, we merely have to redefine the meaning of R_i:

$$R_i[p, q] = \begin{cases} \infty \text{ if } p+1 = q \text{ and matching}(q, i) \text{ or equal}(q, i), \\ \\ R[p, q] \text{ otherwise.} \end{cases}$$

Here, "equal" is any predicate such that matching(i, j) and equal(k, j) implies matching(i, k) for $i < k$.

The same algorithm with the same verification conditions as in the preceding section applies now, but R' has now to be read as R^* and Q' as Q^*. Obviously, the verification itself goes through at all places where no use is made of the meaning of R', Q' or R_i, so only the two places where this is not the case have to be rechecked.

The first occasion is in the verification of V1. As we have seen in section 4, from the identity $A^*[i, j] = A[i, j] + A^+[i, j]$ it follows that $A^*[i, j] = A[i, j]$ if $i = j$ or $i+1 = j$. Consequently, the meaning of $R'[i-1 : i, i]$ is indifferent to R' standing for R or for R^*. The analogous result holds for $Q'[i-1 : i, i]$, so the verification of V1 is left intact.

The second occasion is in V4, where we now have to verify (taking the essential part, and using $T[j] = R^*_i[j, i-1]$):

$$\begin{aligned} &\{R^*[0 : i-1, 0 : i-1] = Q^*[0 : i-1, 0 : i-1], \\ &\quad R^*[j : i, i] = Q^*[j : i, i], j \le i\} \\ &\{(R^*[j-1, i] = Q^*[j-1, i]) \varsigma (R[j-1, i] := \\ &\qquad\qquad (\text{matching}(j, i) \mid R^*_i[j, i-1] \mid \infty))\}. \end{aligned}$$

Using $R^* = R + R^+$ and $Q^* = Q + Q^+$, the consequent can be written as

$$\begin{aligned} &(\text{matching}(j, i) \mid R^*_i[j, i-1] \mid \infty) + R^+[j-1, i] = \\ &Q[j-1, i] + Q^+[j-1, i]. \end{aligned}$$

Since $R^*[p, q] = Q^*[p, q]$ for all p, q such that $j-1 \le p \le q \le i$, with the possible exception of $p = j-1$ and $q = i$, we have, from Lemma 2, $R^+[j-1, i] = Q^+[j-1, i]$. From the definition of Q, we have $Q[j-1, i] =$

(matching(j, i) $|$ $Q^*[j, i-1]$ $|$ ∞). Using these identities, we may rewrite the consequent as

$$\begin{aligned}&(\text{matching}(j, i) \mid R_i^*[j, i-1] \mid \infty) + Q^+[j-1, i] = \\ &(\text{matching}(j, i) \mid Q^*[j, i-1] \mid \infty) + Q^+[j-1, i].\end{aligned}$$

If $\neg$ matching(j, i), the truth of this assertion is obvious. We therefore concentrate on the case where matching(j, i), and have to show

$$R_i^*[j, i-1] + Q^+[j-1, i] = Q^*[j, i-1] + Q^+[j-1, i].$$

It is sufficient to establish the existence of a value x, such that

$$\begin{aligned}&R_i^*[j, i-1] + x = R^*[j, i-1] \text{ and} \\ &x + Q^+[j-1, i] = Q^+[j-1, i],\end{aligned}$$

since in that case we find

$$\begin{aligned}&R_i^*[j, i-1] + Q^+[j-1, i] = R_i^*[j, i-1] + (x + Q^+[j-1, i]) = \\ &(R_i^*[j, i-1] + x) + Q^+[j-1, i] = R^*[j, i-1] + Q^+[j-1, i] = \\ &Q^*[j, i-1] + Q^+[j-1, i].\end{aligned}$$

The second equation involving x suggests that x should be taken as large as possible. From the first equation, we see that we can take for x the sum of those terms in

$$R^*[j, i-1] = \sum_m \sum_{(p)} R[j, p_1] \times \ldots \times R[p_{m-1}, i-1]$$

which have "disappeared" in

$$R_i^*[j, i-1] = \sum_m \sum_{(p)} R_i[j, p_1] \times \ldots \times R_i[p_{m-1}, i-1]$$

by the transition from R to $\underset{i}{R}$. These terms must involve some factor $R[p_{k-1}, p_k] \neq \underset{i}{R}[p_{k-1}, p_k]$ (and, therefore, $\underset{i}{R}[p_{k-1}, p_k] = \infty$, so the term does disappear indeed). This implies, by the definition of $\underset{i}{R}$, that $p_{k-1} = p_k-1$ and matching(p_k, i) or equal(p_k, i).
Taking together the terms having a fixed factor R[z-1, z] in common, we obtain

$$x_z = \sum_m \sum_{(p)} R[j, p_1] \times \ldots \times R[z-1, z] \times \ldots \times R[p_{m-1}, i-1] =$$

$$(\sum_{m'} \sum_{(p')} R[j, p_1'] \times \ldots \times R[p_{m'-1}', z-1]) \times R[z-1, z] \times$$

$$\times (\sum_{m'} \sum_{(p')} R[z, p_1'] \times \ldots \times R[p_{m'-1}', i-1]) =$$

$$R^*[j, z-1] \times R^*[z-1, z] \times R^*[z, i-1] =$$

$$Q^*[j, z-1] \times Q^*[z-1, z] \times Q^*[z, i-1] = Q^*[j, z-1] \times 1 \times Q^*[z, i-1].$$

If matching(z, i), $Q[z-1, i] = Q^*[z, i-1]$. Also, $Q[j-1, j] = 1$, so

$$x_z = Q^*[j, z-1] \times 1 \times Q^*[z, i-1] =$$

$$Q[j-1, j] \times Q^*[j, z-1] \times Q[z-1, i] =$$

$$Q[j-1, j] \times (\sum_m \sum_{(p)} Q[j, p_1] \times \ldots \times Q[p_{m-1}, z-1]) \times Q[z-1, i] \geq$$

$$\sum_{m\geq 2} \sum_{(p)} Q[j-1, p_1] \times \ldots \times Q[p_{m-1}, i] = Q^+[j-1, i].$$

Similarly, if equals(z, i), which implies matching(j, z), we find

$$x_z = Q^*[j, z-1] \times 1 \times Q^*[z, i-1] =$$

$$Q[j-1, z] \times Q^*[z, i-1] \times Q[i-1, i] \geq Q^+[j-1, i].$$

So, if matching(z, i) or equals(z, i), $x_z + Q^+[j-1, i] = Q^+[j-1, i]$.
Since x is the sum of all these x_z, we have

$$x + Q^{+}[j-1, i] = (\sum x_z) + Q^{+}[j-1, i] = Q^{+}[j-1, i].$$

This completes the verification of the algorithm under the optimization introduced by the new definition of R_i.

9. FURTHER OPTIMIZATIONS

Another quite important optimization can be obtained if we can distinguish between opening and closing parentheses. Let predicates "maybe opening parenthesis" and "maybe closing parenthesis" be defined with the properties:

if $\neg$ maybe opening parenthesis(i), then $\neg$ matching(i, k) for all $k > i$;

if $\neg$ maybe closing parenthesis(i), then $\neg$ matching(k, i) for all $k < i$.

Two adjacent parentheses, with indices p and p+1, say, may be omitted from the skeleton without change to the error value, if

matching(p, p+1),
$\neg$ maybe closing parenthesis(p) and
$\neg$ maybe opening parenthesis(p+1).

This can be proved as follows:
Take any reparation of the skeleton. First, we show that there is another reparation, which is at least as good, in which neither p nor p+1 is marked. For, suppose that both are marked. Obviously, unmarking both gives a better reparation.
If, on the other hand, only one is marked, say k, the other one must match, in some parse of the correct skeleton obtained by deleting all marked parentheses, a third one. Let these matching parentheses be i and j. From the properties of the predicates given above, it follows that $i < k < j$. But this situation corresponds exactly to that treated in the previous section (see figures 3a and 3b), and the reparation obtained by

transferring the mark from k to i or j--whichever is appropriate--is just as good.
Now, there is an obvious one-one correspondence between reparations in which neither p nor p+1 is marked, and reparations of the skeleton obtained by omitting p and p+1, and this correspondence preserves the number of marked parentheses. From this, the claim follows immediately.

After omitting parentheses p and p+1, two parentheses which were not yet adjacent may become so, and may possibly be omitted together also. By repeating this process, we obtain an "extract" of the skeleton on which the repairing algorithm has to be performed. This extract can be determined by a simple stack algorithm:

```
stack s:= empty;
for i to n
do if (s = empty | false |
          matching(top(s), i) ∧
          ¬maybe closing parenthesis(top(s)) ∧
          ¬maybe opening parenthesis(i))
   then pop(s)
   else push(s, i)
   fi
od.
```

At the end, the stack contains the extract. For those wishing to give a correctness proof: the essential invariant is

$$\text{error value (skeleton}[0 : n]) = \text{error value (stack + skeleton}[i : n]).$$

The importance of the optimization of only subjecting the extract to the repairing algorithm becomes obvious if the classes of "maybe opening" and "maybe closing" parentheses are disjoint. In that case, a correct skeleton has an empty extract, so that the whole repairing process takes linear time.

A final optimization is found by observing that

```
        Addition of parenthesis(i)
```

may be written

```
        if maybe closing parenthesis(i)
        then Addition of parenthesis(i)
        else Addition of parenthesis(i)
        fi,
```

where, in the else part, the piece of program as developed in section 6 may be greatly simplified, resulting in an overall improvement in speed of about a factor of 2. For, in that case, matching(j, i) cannot hold, so the assignation to R[j-1, i] can be simplified to R[j-1, i]:= ∞. Since then the values of T are no longer used, the assignations to elements of T can be omitted. Factoring out the common start of the then- and the else-part, and performing an obvious abbreviation, we now have obtained (writing again go-on-symbols):

```
        R[i-1 : i, i]:= (1, 0);
        if maybe closing parenthesis(i)
        then ∀ k ∈ [1 : i-2]: T[k]:= ∞, T[i-1]:= 0;
             for j from i-1 by -1 to 1
             do R[j-1, i]:= (matching(j, i) | T[j] | ∞),
                ∀ k ∈ [1 : j-1]: T[k]:= T[k] + R[k, j] × T[j]
                                               i
             od
        else ∀ k ∈ [0 : i-2]: R[k, i]:= ∞
        fi.
```

Under the representation discussed in section 8, the above else-part entails no action at all.

10. APPLICATION TO ALGOL 68.

The parenthesized constructs of ALGOL 68 are:

```
begin ... end   if ... then ... {elif ... then ...}* {else ...} fi
( ... )         case ... in ... {ouse ... in ...}* {out ...} esac
[ ... ]         ( ... | ... {|: ... | ...}* {| ...} )
$ ... $         {for ...} {from ...} {by ...} {to ...} {while ...} do ... od
```

Here, {...} means an optional part, and $\{...\}^*$ means an arbitrary number of repetitions.

A first complication lies in the double nature, closing and opening, of such parentheses as then. It is easily overcome by considering these as a pair of different parentheses, one closing followed by one opening. Another problem stems from the use of one symbol | with two meanings, $|_{in}$ and $|_{out}$. An easy way out is to consider incorrect constructions like (... | ... | ... | ...) correct in terms of the parenthesis structure, and to postpone detection of this error to the actual parsing phase. A more sophisticated solution is to adapt the definition of Q and the computation of the elements of T in such a way that the corresponding incorrect path will not be constructed.

The fact that $ occurs both as opening and as closing parenthesis presents no problem, since the repairing algorithm is perfectly able to deal with such cases. It only means that two $'s do not simply cancel each other in determining the extract (section 9). However, in almost all cases it can be determined from the immediate context if a given $ is an opening or a closing parenthesis.

The final complication is best explained by an example:

a) while + loc v do s od; + log w do s od;
b) while + log v do s od; + loc w do s od.

In line a, + loc v is an enquiry-clause, whereas w is a void-mode-indication and log an operator with a cast as operand, so the parenthesis structure is that suggested by (while do od) (do od). In line b, v is the void-mode-indication, and the parenthesis structure corresponds to (while (do od) do od). So the question matching(i, j) for these parentheses depends on the context of parenthesis j. Fortunately, the decision can be taken on the basis of the two tokens preceding the parenthesis: In the context $\lambda\mu\pi$, where π is <u>from</u>, <u>by</u>, <u>to</u>, <u>while</u> or <u>do</u>, π is an opening parenthesis only, if and only if μ is one of the tokens

:	,	;	<u>begin</u>	<u>if</u>	<u>then</u>	<u>elif</u>	else	
<u>case</u>	<u>in</u>	<u>ouse</u>	<u>out</u>	<u>while</u>	<u>do</u>	(	|	|:

or if μ is a bold-TAG-token or void-token and λ is not one of the tokens

<u>loc</u> <u>heap</u> <u>ref</u>)] <u>proc</u> <u>flex</u>.

In [2], the term "parenthesis" was used for a wider class of symbols, which also comprised ", ¢, #, <u>co</u>, <u>comment</u>, <u>pr</u> and <u>pragmat</u>. The algorithm presented here assumes that the skeleton of these "state switchers" has already been repaired. An efficient algorithm is given in [3].
It is possible to adapt the present algorithm in such a way that it can deal with this wider class of parentheses, but this would result in a great loss of efficiency since then, in general, the extracts from section 9 become much longer.

11. SPEED OF THE PROCESS

From the algorithm it should be obvious that the time taken by the process has an upper bound of the form $O(n^3)$. We have not been able to derive sharper bounds. However, we have determined an empirical formula approximating the times needed for some actual skeletons by an implementation of the algorithm written in ALEPH and run on a CYBER 73. These skeletons were obtained by generating, with a random process, correct skeletons and deleting its parentheses with a probability ε. In total, 184 skeletons were generated, with n varying from 200 to 2000 and ε from 0 to .1, and a formula of the form $t = c_1\varepsilon^p n^q + c_2 n$ was fitted to the results. A good fit was obtained by

$$t = .0013\ \varepsilon^{.82} n^{2.5} + .65\ n \text{ msec.}$$

A typical observed time, for $n = 1000$ and $\varepsilon = .01$, is $t = 1449$ msec.

REFERENCES

[1] HOARE, C.A.R., *An axiomatic basis for computer programming*, Comm. ACM 12 (1969) 576-580.

[2] MEERTENS, L.G.L.T. & J.C. VAN VLIET, *Repairing the parenthesis skeleton of ALGOL 68 programs*, Report IW 2/73, Mathematical Centre, Amsterdam (1973).

[3] MEERTENS, L.G.L.T. & J.C. VAN VLIET, *Repairing the state switcher skeleton of ALGOL 68 programs*, Report IW 15/74, Mathematical Centre, Amsterdam (1974).