stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IA IN 5/73 OCTOBER

L.G.L.T. MEERTENS and J.C. van VLIET
REPAIRING THE PARENTHESIS SKELETON
OF ALGOL 68 PROGRAMS (Extended abstract)

2e boerhaavestraat 49 amsterdam

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

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

ACM - Computing Reviews - category: 4.12, 4.22

Repairing the parenthesis skeleton of ALGOL 68 programs

Lambert Meertens, Hans van Vliet

Summary

Good error-recoverability for an ALGOL 68 parser is only possible if the errors in the parenthesis skeleton are known beforehand. Two algorithms for repairing such errors are discussed.

Text of a paper presented at the Third Semi-annual Western-North American Informal Conference on ALGOL 68, Los Angeles, September, 8-9, 1973.

1. Introduction

The degree to which compilers for high-level languages like ALGOL 68 are able to recover from errors in the source text and to give meaningful error messages, i.e.,error messages which are interpretable for the human programmer, varies considerably in practise. In our current effort to construct a machine-independent ALGOL 68 compiler, one of the design objectives is to reach a relatively high level of error-recoverability. Here, we will discuss one of the concrete aspects of the syntax of ALGOL 68 [1], the parenthesis structure.*

If, somewhere in a piece of source text which starts with an opening parenthesis, an error occurs which causes the parser to derail, one may hope to use the closing parenthesis to bring it back in its track. Should, however, this closing parenthesis be missing (which might be the cause of the derailment in the first place), then this strategy is not particularly helpful. It may appear that the solution would be to insert, as it were, the matching closing parenthesis in the source text when a different closing parenthesis is met, but then, if the source text contains an extra closing parenthesis, we are even worse off. The conclusion is that a good resynchronization of the parser is only then possible if it is known beforehand which opening parentheses are accompanied by a matching closing parenthesis, and which are not (and vice versa). This holds especially for some parentheses, such as the quote-symbol, that have one same representation for opening and closing parenthesis.

The term "parenthesis" is used here to denote a wider class of sym-

*) A detailed treatment is given in [2]. However, the revision of ALGOL 68 has been taken only partially into account there. It is hoped that a "revised" version of [2] will appear in due time.

bols than is usual: we shall use this term to stand for the following symbols:

"braces": $, (,), *begin*, *end*, [,], |, |:, *case*, *in*, *out*, *ouse*, *esac*, *if*, *then*, *else*, *elif*, *fi*, *for*, *from*, *by*, *to*, *while*, *do*, *od*, and
"state switchers": ", ¢, #, *co*, *comment*, *pr* and *pragmat*.

The role played by the state switchers is so special as to warrant a special treatment. Not only does one same symbol serve both as "opener" and as "closer" of certain constructions, but, which is more important, the "item sequences" which are embraced by these symbols lack syntactical structure and may contain braces in an arbitrary fashion that otherwise would have to occur "nested". Therefore, it is a hopeless task to treat the braces before it is known which parts of the program are item sequences, and which are not, and, consequently, which braces have to be disregarded and which have to be taken into account.

Apart from this, the treatment of both types of parentheses runs largely in parallel. In general, errors in the parenthesis skeleton are repaired by marking a number of parentheses such that, by deleting these parentheses, a correct skeleton is obtained.

2. The treatment of state switchers

An ALGOL 68 program can be thought of as consisting of a sequence of (possibly empty) segments, separated by state switchers. To each of these segments a "state" may be assigned, which is either "neutral" or one of the state switchers. For a correct program, it is possible to assign these states in such a way that the first and the last segment are neutral and that at each state switcher we have a correct transition, i.e., the state switches to that state switcher if it was neutral and to neutral if the present state switcher is equal to the state, and otherwise the state is not affected. To give an example:

segments: ~~~~~ " ~~~~~ # ~~~ " ~~~~~~~ ¢ ~~ *co* ~~~ " ~~ ¢ ~~~~~
states: neutral " " neutral ¢ ¢ ¢ neutral

Note that the segments which have a state switcher as state are precisely those segments that are, or are contained in, an item sequence. Obviously, if such an assignment of states is not possible, the program is incorrect.

It is necessary to refine our definition of a correct transition slightly further. Although state switchers have one same representation for openers and closers, it is possible in some cases to derive from the context that a given state switcher, which then must be a quote-symbol, cannot be an opener or a closer. E.g., in the context of *d = "monday";* it can be shown that the first quote-symbol must be an opener and the second one a closer. Now, for a state switcher which has shown to be a non-opener, the transition from the state neutral to the state " is not considered correct. A similar restriction applies to state switchers which have been shown to be non-closers.

The task of the algorithm for correcting the state-switcher skeleton can be formulated approximately as follows: assign states to each of the segments in such a way that the number of incorrect transitions is kept, in some sense, as low as possible.

The elementary actions consist of the marking of one state switcher, indicating that it should be disregarded in order to obtain a correct state-switcher skeleton. This implies that the state should not switch at such a state switcher. Therefore, for an incorrect transition to be admissible in such cases, it is necessary that the state does not change.

In the general case, there will be more than one admissible interpretation for a given sequence of segments. The problem is, therefore, to give a criterion to which one of these interpretations can be chosen as, hopefully, the best. A simple criterion would be to count the number of incorrect transitions. We have chosen, however, for a more sophisticated criterion. Rather than having all incorrect transitions weigh equally, different "error values" have been assigned to the various types of incorrect transitions, based upon estimates of the likelihood of these transitions. The thought underlying the computation of those values is that of Bayesian analysis.

2.1 The algorithm

The task of the algorithm can be stated thus: find among all admissible interpretations an optimal one, i.e., one with minimal total error value. Obviously, it is impractical to generate all admissible interpretations one by one, as their number will grow exponentially with the number of state switchers in the source text. By applying the principle of dynamic programming, however, it is possible to derive a practical algorithm.

3. The treatment of braces

After, as a result of the treatment of the state switchers, states have been assigned to all of the segments, it is known which parts of the program are item sequences, and, consequently, which braces have to be disregarded. It is the task of the algorithm for repairing the brace skeleton to try to match as much as possible the braces and to mark the remaining ones. The criterion for comparing two alternatives will simply be: which one has the smaller number of marked braces.

First, it is obvious that braces like (and) or *begin* and *end* match. However, in the case of, e.g., | the potential for matching is much higher: this brace is able to match simultaneously to the left with either of the braces (, | and |:, and to the right with |, |:, and). In spite of this seeming complication, a satisfactory and yet simple solution is given by systematically replacing | by)|(and |: by)|:(, and then to require the matching of (and), after which | and |: are no longer considered braces. Similarly, *then* is replaced by *fi then if*, and so on. Obviously incorrect skeletons, such as (|||), are not indicated as such by this algorithm.

Similar to the state switchers, it is sometimes possible to tell whether a $ serves as an opener or a closer. This information, if available, is taken into account.

3.1 The algorithm

A correct brace skeleton, i.e., one in which all braces match properly, can be characterized algorithmically as follows:

Start with an empty stack. Scan the braces from left to right. Upon meeting an opening brace, it is put on the stack. When a closing brace is met, it matches the top of the stack (otherwise the brace skeleton was incorrect) and that brace is "matched away", i.e., removed from the stack top. After having processed all braces, the stack is again empty.

An alternative way of viewing this can bring the notion of "correct brace skeleton" in a framework quite similar to that employed for dealing with state switchers: The stacks are *states* that are assigned to the segments between the braces. For a correct program, it is possible to assign these states in such a way that the first and the last segment are neutral (i.e., have an empty stack) and that at each brace we have a correct transition, i.e., if the brace is an opening brace the new state consists of the old state with that brace put on top, and if it is a closing brace, the old state consists of the new state with the matching opening brace put on top.

The definition of an admissible incorrect transition then becomes: a transition is admissible incorrect if the old state and the new state are equal.

The algorithm sketched above is rather impractical. It is possible to refine the process in such a way that the number of branches is kept below a certain reasonable limit.

References

[1] A. van Wijngaarden e.a., Revised Report on the Algorithmic Language ALGOL 68.

[2] L.G.L.T. Meertens and J.C. van Vliet, Repairing the Parenthesis Skeleton of ALGOL 68 Programs, IW2/73, Mathematisch Centrum, Amsterdam (1973).