stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 189/81 DECEMBER

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

ON THE MC ALGOL 68 COMPILER

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, 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.).

1980 Mathematics subject classification: 68B20

ACM-Computing Reviews-category: 4.12

On the MC ALGOL 68 compiler

by

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

ABSTRACT

From 1969 until 1980, research has been done at the Mathematical Centre regarding various aspects of ALGOL 68 implementation. This has resulted in many publications, each treating an aspect in isolation. Several of these publications deal with issues arising in the construction of a portable ALGOL 68 compiler for the full language, including the Standard Hardware Representation and the modules and separate compilation facility. These publications deal especially with the first stages: the construction of a parser, and the last stage: an abstract ALGOL 68 machine. The purpose of the present report is to indicate where these various results would find their place in the construction of an ALGOL 68 compiler.

KEY WORDS & PHRASES: ALGOL 68, compiler, parsing, error recovery, abstract machine

1. INTRODUCTION

From 1969 until 1980, research has been done at the Mathematical Centre regarding various aspects of ALGOL 68 implementation. This has resulted in many publications, each treating an aspect in isolation. Several of these publications deal with issues arising in the construction of a portable ALGOL 68 compiler for the full language, including the Standard Hardware Representation [1] and the modules and separate compilation facility [2]. These publications deal especially with the first stages: the construction of a parser, and the last stage: an abstract ALGOL 68 machine. The purpose of the present report is to indicate where these various results would find their place in the construction of an ALGOL 68 compiler.

The compiler consists of a number of phases, each of which transforms its input stream in an output stream. The first phase has a given source text as input stream, the last phase has runnable code for some target machine as its output stream. The global structure of the compiler is fairly standard: the first phases build the parser, and result in a decorated parse tree. Following phases transform this parse tree into code for the abstract ALGOL 68 machine. Finally, the abstract machine code is transformed into code for some target computer.

In most programming languages, a program is allowed to use certain identifiers which have not been declared in the program, but are somehow known to the system, such as "sqrt" or "print". In the Revised Report on the Algorithmic Language ALGOL 68 [3], these are given as a sequence of declarations in a superlanguage of ALGOL 68. These declarations together constitute the "standard prelude". Since the "transput" part of the standard-prelude is very substantial in size, but can largely, and without undue loss of efficiency, be described in proper ALGOL 68 (see [4]), we have defined a superlanguage of ALGOL 68 suitable for dealing with the standard-prelude. The standard-prelude can then simply be compiled like any other "user-prelude", thus automatically creating a large part of the runtime system. This superlanguage is called ALGOL 68+. It is such that a very minor restriction gives a sublanguage which is identical again to ALGOL 68. The definition is given in [5].

2. THE PARSER

The purpose of the parser is to analyze a given source text, and to produce a decorated parse tree suitable for code generation. In the first phase, the source text is considered as a sequence of individual characters. This sequence is read, and greater units, like identifiers, are built. This phase is known as <u>lexical analysis</u>. Subsequently, it is checked whether this sequence of lexical units conforms to the syntax of the language. This is known as <u>syntax analysis</u>, and the result is a parse tree. Lastly, the meaning of each construct is determined from its context: for each applied occurrence, the corresponding defining occurrence is sought for, and the like. This is known as <u>semantic</u>

analysis, and it adds information to the parse tree: the parse tree is decorated.

At a very early stage in the design of the compiler, we decided to use a top-down parsing method based on an LL(1) grammar. A context-free grammar underlying the ALGOL 68+ syntax, such as the one given in [6], is not of type LL(1), but it seems possible to construct an LL(1) grammar for "context-free ALGOL 68+". However, in doing this the original syntactic structure is lost. Another possibility is to apply beforehand a simple transduction scheme, operating from right to left, which brings the source text in prefix form. This transduction scheme can also be applied to the context-free grammar, resulting in an LL(1) grammar suitable for top-down parsing. In this way, syntax analysis is comprised of two phases: a syntax-directed transduction, followed by a top-down analysis of the transformed input text.

All of the pertinent results obtained have to do with these early phases of the parser. They are further discussed in the sections below. As regards semantic analysis, the following breakdown has been envisaged:

i) Mode-equivalencing: The information gathered on the modes of all constructs are brought into standard form, so that two modes that are described in a different way, but are equivalent, coincide. This algorithm does not transform some input stream into some output stream, but is a transformation of information gathered in tables.

ii) Identification: For each "indicator" occurring in the input text, the corresponding definition is determined. Because of the possibility of overloading operators, context information must be taken into account here.

iii) Coercion: Most constructs in an ALGOL 68 program have both an a priori and an a posteriori mode. During coercion, both these modes and the transformation path in between are determined from the construct and its context.

2.1. Lexical analysis

In 1976, the Standard Hardware Representation for ALGOL 68 was published [1]. This proposal allows for three different stropping regimes: point stropping, upper stropping and res stropping. These stropping regimes mainly differ in the way in which bold words are represented. In point stropping, each bold word is written as a point (".") followed by the letters and digits of the word. In upper stropping, one may in addition represent bold words in the upper-case font. In res stropping, one is allowed to omit the point from the reserved bold words given in section 9.4.1 of the Revised Report.

A finite-state machine accepting text in each of the three stropping regimes (plus a fourth local regime) is given in [7]. It is described in a very simple language, allowing semi-mechanical translation to an arbitrary language. This machine forms the kernel of the lexical analysis phase.

2.2. Error recovery

During the design of the compiler, much attention has been paid to error recovery. Three techniques have been developed: *state switchers*, *parenthesis skeleton* and *synchronization*. Each of these techniques assumes that the previous one has taken place.

ALGOL 68 contains a number of parentheses which play a very special role, viz., ", ¢, #, *co*, *comment*, *pr* and *pragmat*. Not only does one same symbol serve both as opening parenthesis and as closing parenthesis of certain constructions, but, more important, the pieces of text delimited by these symbols (the inside of comments, pragmats and string-denotations) lack syntactical structure. Also, such a piece of text may not contain another similar construction. One may view these parentheses as handles at which the "state" switches from "program text" to, e.g., "comment", and vice versa. An error in the "state switcher skeleton" will in general derail the parser in a serious way: comment will be treated as program text, and program text as comment. A practical algorithm for repairing incorrect state switcher skeletons is given in [8]. It can best be incorporated as a subphase of lexical analysis, to be invoked when an error is found; after the state switcher skeleton has been repaired, lexical analysis must be re-initiated.

On a next higher level, an ALGOL 68 program can be viewed as a sequence of nested parenthesized constructs. Most of these constructs are relevant to the range structure. Therefore, it is not uncommon for compilers to give up if, after lexical analysis, the parenthesis structure of the source text has been found incorrect. An algorithm for repairing incorrect parenthesis skeletons is given in [9]. It can, similar to the algorithm for repairing incorrect state switcher skeletons, be incorporated as a subphase of lexical analysis.

The synchronization method is closely related to the syntax-analysis method chosen. During the transduction phase, the source text is brought into prefix form. For instance, a formula A+B is transformed into +AB. If in the subsequent phase, parsing gets stuck in the operand A, we want it to resume at the place where the operator + occurred. This is in general only possible if some mark is left behind by the transducer at the place where it picked up the operator. Leaving such a "resynchronization symbol" behind has the additional advantage that one can be sure to *always* find a point at which parsing may be resumed. This method is further discussed in [10].

2.3. Syntax analysis

A transducer which brings source texts in prefix form can be easily constructed for an operator-precedence grammar. It is a straightforward variant of the operator-precedence parsing algorithm given in [11], pp. 170-171. This algorithm only needs to know the precedence relations between the symbols of the grammar. In order to let the transducer work for all input texts, it is necessary to define precedence relations for all pairs of symbols. For an arbitrary operator-precedence grammar, it is not at all clear how to fill the empty entries in the table of precedence relations in such a way that a reasonably consistent treatment of incorrect input texts is obtained. Therefore, some further restrictions on the grammar have been introduced, leading to the notion of an operator-priority grammar. Such an operator-priority grammar for ALGOL 68+ has been given in [12].

The syntax-directed transduction is based on the grammar from [12]. Care has been taken to ensure that, when the transduction is applied to the grammar, the resulting grammar is of type LL(1), so that the LL(1) parsing method can be applied next. Both the algorithm for the transduction scheme and the subsequent top-down parsing method are discussed in [10], which also contains the LL(1) grammar for ALGOL 68+.

The operator-priority grammar in [12] differs from the underlying context-free grammar in [6] in various ways. Most notably, symbols represented by the same mark have been distinguished, and various symbols have been inserted in the grammar. When actually parsing, the same modifications must be applied. In [13], an algorithm is derived which transforms ALGOL 68+ texts into sentences produced by the operator-priority grammar. Most of these changes can be taken care of during lexical analysis. Some, however, require knowledge of the types of the various bold words in the program, and can therefore not be made until (the input of) the transduction phase.

3. THE ABSTRACT ALGOL 68 MACHINE

The work on the design of an abstract ALGOL 68 machine, called MIAM, derives its motivation from the objective of portability. However portable a compiler may be, at some stage the actual target computer has to enter the generation of runnable code. A code generator generates MIAM code, which, subsequently, has to be "transformed" into code for an actual target computer. So the abstract machine is the last common stepping stone before the ineluctable divergence. For that reason, the MIAM has been designed such that the solutions for the many problems in code generation that can be solved in a machine-independent way can be expressed in the machine-independent code generation phase. This may sound as a design objective that is easily reached, but in reality it requires a careful analysis. Moreover, a secondary design objective in designing the MIAM was that the design complexity of the whole of code generation and transformation would be reduced. Without such a requirement, the result might be that the primary design objective would be reached at the expense of the machine-dependent stage. This has led to the adoption of a design philosophy, the "Cut Principle", described in [14]. A formal definition of the MIAM is given in [15].

This definition provides for a clear interface between code generation and transformation. By virtue of this interface, it is possible to solve important parts of the problem of realizing the MIAM on an actual computer in a still machine-independent way. The most important of these problems is that of memory management, including garbage collection and compaction. By using a "MIAC", being a "primitive" version of the MIAM with a few instructions only, the algorithms can be expressed in a way that lends to easy realization on actual computers ([16]).

REFERENCES

[1] HANSEN, W.J. & H.J. BOOM, The report on the standard hardware representation for ALGOL 68, SIGPLAN Notices _12_, 5 (May 1977), pp 80-87.

[2] LINDSEY, C.H. & H.J. BOOM, A modules and separate compilation facility for ALGOL 68, ALGOL Bulletin _43_ (1978), pp 19-53.

[3] VAN WIJNGAARDEN, A. et al, Revised Report on the Algorithmic Language ALGOL 68, Acta Informatica _5_ (1975), pp 1-236.

[4] VAN VLIET, J.C., ALGOL 68 transput, Part II: An implementation model, Mathematical Centre Tracts 111, Amsterdam, 1979.

[5] MEERTENS, L.G.L.T. & J.C. VAN VLIET, ALGOL 68+, a superlanguage of ALGOL 68 for processing the standard-prelude, Report IW 168/81, Mathematical Centre, Amsterdam, 1981.

[6] MEERTENS, L.G.L.T. & J.C. VAN VLIET, An underlying context-free grammar of ALGOL 68+, Report IW 171/81, Mathematical Centre, Amsterdam, 1981.

[7] JONKERS, H.B.M., A finite state lexical analyzer for the standard hardware representation of ALGOL 68, ALGOL Bulletin _44_ (1979), pp 16-51.

[8] 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.

[9] MEERTENS, L.G.L.T. & J.C. VAN VLIET, Repairing the parenthesis skeleton of ALGOL 68 programs: proof of correctness, _in_ G.E. Hedrick (Ed.), Proceedings of the 1975 International Conference on ALGOL 68, Oklahoma State University, Stillwater, June 10-12, 1975 (also registered as Mathematical Centre Report IW 52/75).

[10] MEERTENS, L.G.L.T. & J.C. VAN VLIET, On top-down parsing of ALGOL 68+, Report IW 182/81, Mathematical Centre, Amsterdam, 1981.

[11] AHO, A.V. & J.D. ULLMAN, Principles of compiler design, Addison-Wesley, 1977.

[12] MEERTENS, L.G.L.T. & J.C. VAN VLIET, An operator-priority grammar for ALGOL 68+, Report IW 173/81, Mathematical Centre, Amsterdam, 1981.

[13] MEERTENS, L.G.L.T. & J.C. VAN VLIET, Making ALGOL 68+ texts conform to an operator-priority grammar, Report IW 180/81, Mathematical Centre, Amsterdam, 1981.

[14] MEERTENS, L.G.L.T., On the design of an abstract machine for a portable ALGOL 68 compiler, in J.C. van Vliet & H. Wupper (Eds.), Proceedings International Conference on ALGOL 68, Bochum, BRD, March 30-31, 1981, Mathematical Centre Tracts 134, Amsterdam, 1981.

[15] MEERTENS, L.G.L.T., Definition of an abstract ALGOL 68 machine, Report IW 188/81, Mathematical Centre, Amsterdam, 1981.

[16] JONKERS, H.B.M., Abstraction, specification and implementation techniques, with an application to garbage collection, Ph.D. Thesis, Mathematical Centre, Amsterdam, 1982.