mathematisch centrum

AFDELING INFORMATICA

IW 31/75 JANUARY

L. AMMERAAL

AN IMPLEMENTATION OF AN ALGOL 68 SUBLANGUAGE

Prepublication

amsterdam 1975

stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IW 31/75 JANUARY

L. AMMERAAL

AN IMPLEMENTATION OF AN ALGOL 68 SUBLANGUAGE

Prepublication

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.

AMS (MOS) subject classification scheme (1970): 68A00, 68A30

An implementation of an ALGOL 68 sublanguage *)

by

L. Ammeraal

ABSTRACT

For various reasons, attention should not only be paid to the implementation of the complete language ALGOL 68 but also to the definition and implementation of ALGOL 68 sublanguages. Such a sublanguage has been implemented and some details of the object code and of the compiler are reported. For a machine independent discussion of code generation and of a simple optimization method, some virtual machine instructions are introduced.

KEY WORDS & PHRASES: *Compilers, ALGOL 68, code generation, syntactic analysis.*

---

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

INTRODUCTION

There are some elements in ALGOL 68 [1] that are both useful and easily implementable. To illustrate this in connection with some other languages, consider e.g. the following ALGOL 60 expression

```
p + q * (if x > 0 then a + b else c - d) / r .
```

This example shows a conditional expression, a useful concept, which is unfortunately not present in e.g. PL/I [2], or PASCAL [3]. In these latter languages the programmer is forced to express this in a more primitive way, splitting it up into two statements or defining a function for the conditional form if an expression is required by the context.

In ALGOL 68, on the other hand, the notions "expression" and "statement" are very elegantly generalized in the concept "clause"; the programmer is even allowed to write things like

```
if x < y then v else w fi :=
p + q * if x < 0
        then a := 3; a + (b:=4;b)
        else while c < d
             do c := c + 1
             od;
             c - d
        fi / r
```

Of course, for specific applications, one can construct more interesting examples of the flexible and convenient way algorithms can be expressed in ALGOL 68. It is perhaps not always realized, however, that to some

extent a programmer can express algorithms in this way, even if he is restricted to a very small and hence well-implementable subset of this language. The main argument of considering the implementation of rather small ALGOL 68 sublanguages is the practical wish that useful and easily implementable specific ALGOL 68 features will become available for users of small and medium size machines. Another argument is the idea of "bootstrapping": a more general ALGOL 68 compiler could be written in a sublanguage if a compiler for this sublanguage is available. In the first step of such a bootstrapping process some already implemented language has to be used, of course. A simple language, without concepts that are far beyond the scope of the ALGOL 68 sublanguage under consideration, is an attractive candidate as a medium to formulate the first compiler, since this will reduce the amount of work involved in rewriting the compiler in the chosen sublanguage. In the implementation that will be reported, a restricted use of ALGOL 60 was made for this purpose.

To avoid any misunderstanding, it is stated explicitly that this paper deals only with a small project, undertaken by one man; it should not be interpreted as a criticism of full ALGOL 68 implementation projects, which should be encouraged as well.

The sublanguage L_0.

For reasons of brevity the implemented ALGOL 68 sublanguage will be called L_0. It does not include structured values, united modes, mode declarations, operator declarations and heap generators. These and some other restrictions leave a language that is very poor compared with ALGOL 68 but still more powerful than some conventional programming languages. Very roughly

speaking, L_0 has the level of ALGOL 60, but the implementer of L_0 has neither to bother about less fortunate ALGOL 60 elements nor to solve immediately the implementation problems related to some more advanced ALGOL 68 concepts such as "infinite modes".
On the other hand, L_0 includes e.g. the ALGOL 68 loop construction

for ... from ... by ... to ... while ... do ... od

with all its abbreviated forms, constructions such as mentioned in the introduction, "dynamic array bounds" (a severe lack in PASCAL as pointed out by Haberman [8]) and objects whose modes start with an arbitrary number of times "reference to".
A precise grammatical definition of L_0 could be given but one can more informally obtain an impression of L_0 from the preceding remarks and from all sample pieces of ALGOL 68 text in this paper since this ALGOL 68 text is at the same time L_0 text. This impression will suffice to understand some remarks on the implementation of L_0 which is the main topic of this paper.

Some virtual machine instructions

The object code physically produced by the compiler is assembly code for one specific machine. To facilitate the discussion on the object code and to make this discussion machine independent the idea of virtual machine instructions, also called macros, is introduced. Each macro is a shorthand notation for a number of assembler instructions that are actually generated by the compiler. So there is no need for a macro processor. Within the compiler a macro corresponds to a code generating procedure.

The design of a set of virtual machine instructions should be done very carefully in order to permit the application of a simple and well-known optimization technique as is stepwise shown by the following example. If macros are to be generated for

```
x := yy + 1,                                                    (1)
```

where x and yy have modes specified by ref int and ref ref int, respectively, a first attempt could yield

```
VALUE (1,1);
VALUE (1,2);
DEREF;
DEREF;                                                          (2)
DENOTATION (1);
ADDIT
ASSIGN .
```

Each identifier is addressed by a number pair consisting of the number of a range and an ordinal number relative to be beginning of the range. This is described in detail for ALGOL 60 by Randell and Russell [5]. The instructions VALUE and DENOTATION put a value on the top of a stack, increasing the stackpointer beforehand by 1. DEREF replaces a value which is an address by the contents of that address, operating on the top of the stack and leaving the stack pointer unaltered. ADDIT fetches the value that is on the top of the stack, subsequently decreases the stackpointer by 1 and then adds the fetched value to the value that is on the new top of the stack. ASSIGN fetches a value in the same way as ADDIT and stores it in

the address which is the value on the new top of the stack.
This first approximation (2) can easily be generated from the source text (1). It is, however, a very inefficient translation of (1) if a machine with at least one register is used. This fact and a simple means to improve efficiency is shown by the following decomposition of the macros.

```
proc VALUE = (int rn, on) void : (VAL(rn,on);PUSH);
proc DEREF = void : (POP;DRF;PUSH);
proc DENOTATION = (int i) void : (DEN(i);PUSH);
proc ADDIT = void : (POP;ADD);
proc ASSIGN = void : (POP;ASS).
```

Here a number of new and more elementary macros are introduced; simple transport to and from the stack is made explicit by PUSH and POP. The unnecessarily laborious definition of DEREF will fit perfectly in the optimization rule to be mentioned.
The new macros replace the macros of (2); they are defined by

```
proc PUSH = void : S[n +:= 1] := R;
proc POP = void : (R := S[n]; n -:= 1);
proc VAL = (int rn,on) void : R := ACCESS(rn,on)
                        # ACCESS is left undefined here #;
proc DEN = (int i) void : R := i;
proc DRF = void : R := G[R];
proc ADD = void : S[n] +:= R;
proc ASS = void : G[S[n]] := R.
```

The macro VAL places a value, being the internal object possessed by an

identifier, into register R; this value is an address, i.e. an index of the generator stack G, if the mode of the identifier begins with "reference to". The contents of this address may again be an address as happens to be the case for yy in the example.

Replacing each macro of (2) by (one line with) its more elementary macros yields

```
VAL (1,1); PUSH;
VAL (1,2); PUSH;
POP; DRF; PUSH;
POP; DRF; PUSH;                                    (3)
DEN (1); PUSH;
POP; ADD;
POP; ASS .
```

The simple optimization method mentioned before is now very obviously obtained from (3). It consists very simply indeed in deleting all occurrences of the sequence

```
PUSH; POP;.
```

Thus (3) reduces to

```
VAL (1,1); PUSH;
VAL (1,2);
DRF;
DRF; PUSH;                                         (4)
DEN (1);
ADD;
POP; ASS.
```

It follows from this example that in evaluating the merits of virtual object code one should be careful : at first sight the ten macros in (4) do not seem to be a better translation of (1) than the seven macros in (2). Yet they are, as was shown by expanding them and by looking at the resulting combination of elementary instructions that originated from two consecutive macros. For ALGOL 60, the application of similar optimization rules are described by Kruseman Aretz [4].

The compiler

The compiler consists of two parts : the rather simple lexical analysis part and the more interesting syntactic analysis and code generation part.

Top-down syntactic analysis is performed by means of a set of boolean procedures each of which corresponds to a notion at the left hand side of a production rule. A call of such a "syntactical procedure", corresponding to some notion, yields true if a terminal production of that notion is present, starting at x[p] in the text array x and false otherwise. The read pointer p is advanced if and only if the call yields true. To specify information about other side effects each procedure is given three parameters, called q, sort and moid.
In a call of a syntactical procedure there are three possibilities for q:

f : neither code generation nor bookkeeping side effects are allowed,

g : normal bookkeeping and code generation side effects are required,

h : only bookkeeping of the mode of formal declarers is required.

The second parameter, sort, may be given the values:

strong : the coercions deproceduring, dereferencing and voiding are allowed for coercends,

soft : only deproceduring is allowed.

The third parameter moid is either given the value of an index in the mode table and then represents a mode, or it is given some negative value, indicating that the mode is not fully prescribed but is to be determined and delivered by the procedure through this parameter.

The following example illustrates how modes are represented at compile time.

Suppose the above mentioned index value is 4 and the mode table is filled as follows

	mode table		
	mark	repl	next
1	bool	0	0
2	int	0	0
3	void	0	0
→ 4	proc	2	5
5	void	0	0
6	bool	0	0
7	ref	2	8
8	row	3	2

Then the index value represents the mode specified by

proc (bool, ref ref[, ,] int) void

The column "next" may contain another mode table index. The column "repl" specifies the number of parameters if mark = proc, the dimension of a multiple value if mark = row or it is a "replication factor" if mark = ref. The following declarations may give an idea how the compiler works. Although the first version of the compiler was written in ALGOL 60, the compiler fragments are given here in ALGOL 68, or more specifically, in the implemented sublanguage L_0. So the following text is part of the compiler and at the same time part of a sample program accepted by the compiler.

```
int strong = -12, soft = -13, ref = 96, proc = 99,
    f = -3, g = -6, h = -7 #and so on#;

proc unit = (int q, sort, ref int moid) bool :
if loop(q, sort, moid) then true
elif skip(q, sort, moid) then true
elif routine text(q, sort, moid) then true
elif assignation(q, sort, moid) then true
else tertiary(q, sort, moid)
fi;

proc tertiary = (int q, sort, ref int moid) bool :
if formula(q, sort, moid) then true
elif local generator(q, sort, moid) then true
else primary(q, sort, moid)
fi;
```

```
proc primary = (int q, sort, ref int moid) bool :
if istag(xp) #is x[p] a tag?#
then if x[p+1] = open symbol then call or slice(q, sort, moid)
                             else identifier(q, sort, moid)
     fi
elif xp = open symbol
then if x[x[p+1]] = open symbol #see remark below#
     then call or slice(q, sort, moid)
     else enclosed clause(q, sort, moid)
     fi
else denoter(q, sort, moid)
fi;
```

Each open symbol is followed by a pointer p' pointing to the symbol x[p'] immediately after the corresponding close symbol. This considerably reduces the amount of work in looking forward. The above formula

```
x[x[p+1]] = open symbol
```

is used to detect a construction like the slice after the first go on symbol in

```
[ ] int a = (10, 20, 30); (int i = 2; print(i); a) (3) #
```

```
proc assignation = (int q, sort, ref int moid) bool :
if look 2 (tertiary(f, dummy, dummy),
           input(becomes symbol)) #look 2 resets p#
then int tdest, tsource;
     tdest := -ref #see remark below#;
     tertiary(q, soft, tdest);
     input(becomes symbol);
     if q = g
     then tsource := ask #yields a new mode table index#;
          copy(tdest, tsource);
          #the source mode is now constructed from the destination
           mode#
           if repl[tsource] > 1
           then repl[tsource] -:= 1
           else tsource := next[tsource]
           fi
     fi;
     if unit(q, strong,tsource) then skip else error fi;
     if q = g
     then if stock then stock := false else output(pop) fi;
          output(ass)  #see remark below#;
          if coercion(tdest, sort, moid) then skip else error fi;
     fi; true
else false
fi;
```

In the above call

tertiary(q, soft, tdest),

the value tdest = -ref indicates that the mode of the destination must begin with "reference to" but is otherwise not prescribed. After this call tdest represents the mode of the destination. This explains why the mode of the third parameter of the syntactical procedures is reference to integral.

In the following remarks on the generation of object code, the "output of macros" should be interpreted, symbolically, as a means to restrict ourselves to the essential points. In reality not these macros are output by the compiler but the assembly instructions they represent.
The simple optimization rule mentioned before is implemented by using a global boolean variable stock, which is initially given the value false. Each time the macro PUSH would be output if we applied a straightforward translation method yielding code like (3), this is deferred and stock is given the value true. Each time POP would be output, stock is inspected. If it is true, POP is not output and stock is set to false; otherwise POP is output. Each time VAL or DEN is output, this is preceded by the inspection of stock and by outputting PUSH and resetting stock to false if it is true. In the above syntactical procedures the method of code generation is shown by "assignation". Here first code is generated for the destination and the source as side effects of "tertiary" and "unit", respectively.

Then the macro ASS is output, preceded by POP if stock was false, as happens in a case like (4). In the assignation

```
x := 0 ,
```

however, where x has the mode reference to integral, ASS would not be preceded by POP, because here the last macro was DEN and stock would have the value true, so the result is

```
VAL(...,...);
PUSH;
DEN(0);
ASS.
```

Since an assignation is a "coercend", whereas e.g. a unit is not, the syntactical procedure "assignation" contains a call of "coercion"; this procedure differs from syntactical procedures in the meaning of the first parameter. This first parameter of coercion represents the a priori mode of the coercend, the second and third parameter stand for the "syntactic position" sort and the required a posteriori mode. The generation of coercion macros such as DRF is performed by this procedure.

Results

The present version of the syntactic analysis and code generation part of the compiler consists of 15 pages of ALGOL 60 text. It generates COMPASS assembler code for the Control Data Cyber computer, but no special features of this machine were exploited, and, if necessary, the generation of the

group of instructions that correspond to each macro can easily be replaced to generate code for a different machine. A number of test programs, including an ALGOL 68 version of Knuth's "Man or boy?" [6], was compiled and excuted succesfully.

Although extensive timing measurements have not yet been made, the first impression is that execution speed of the generated code can compete with some other implementations of high-level languages.

References

[1] A. van Wijngaarden (ed.), *Report on the Algorithmic Language ALGOL 68*, Numer. Math. 14, 79-218 (1969)

[2] *PL/I : Language Specifications*, IBM, File no.S360-29

[3] N. Wirth, *The programming Language PASCAL*, Acta Informatica 1, 35-63 (1971)

[4] F.E.J. Kruseman Aretz, P.J.W. ten Hagen, H.L. Oudshoorn, *An ALGOL 60 Compiler in ALGOL 60*, Mathematical Centre Tracts Nr.48, Amsterdam 1973.

[5] B. Randell and L.J. Russell, *ALGOL 60 Implementation*, Academic Press, 1964.

[6] D.E. Knuth, *Man or boy?* ALGOL Bulletin No.17 (page 7), 1964.

[7] H.J. Lane, *An ALGOL 68 machine and translator*, University of California, Los Angeles, September 1973.

[8] A.N. Haberman, *Critical Comments on the Programming Language PASCAL*, Acta Informatica 3, 47-57 (1973.