stichting
mathematisch
centrum

MC

REKENAFDELING MR 129/72 JANUARY

C.H.A. KOSTER

TOWARDS A MACHINE-INDEPENDENT ALGOL 68 TRANSLATOR

2e boerhaavestraat 49 amsterdam

0. Introduction

The purpose of this paper is to point out some problems encountered, and some techniques used in our attempt to construct a machine-independent ALGOL 68 translator.

The main tool used is a compiler compiler [1], based on "affixgrammars", a two-level extension of Context-Free grammars not unlike the grammar used in the ALGOL 68 Report itself, and furthermore equipped with a macro mechanism allowing the incorporation of primitive actions and functions. The input language of the compiler, called "Compiler Description Language", CDL for short, can be seen as a high-level language whose syntax and semantics are extremely simple, and which is well suited for describing in a machine-independent fashion the parsing and listprocessing which takes place in a translator.

In order to give some flavor of the application of this compiler description language to ALGOL 68, in section 2 a possible treatment of declarers is worked out.

1.0. Some problems

In this section we mention some of the problems encountered in applying syntax-directed methods to the syntax of ALGOL 68 as given in the Report.

1.1. The type of grammar used

The syntax of ALGOL 68 is given in the form of a two-level grammar, which is equivalent to a Context-Free grammar with an infinite number of rules, for which classical parsing methods fail. The report defines how to apply the syntax rules generatively, i.e. how to generate a program. Application of those rules in reverse leads to a parsing process which is at best highly complicated and inefficient, while it is doubtful whether it even terminates.

The solution usually taken is to reduce the parsing problem to the well known and understood parsing problem for Context-Free languages, and take the context-sensitive aspects into account by other means.

From the syntax of the Report is distilled a smallest Context-Free syntax S such that

1) The language of S includes ALGOL 68 properly;
2) The nonterminals occurring in S bear some useful relationship to the notions of ALGOL 68.

A rough approximation is reached by striking from each rule all metanotions, and those parts concerned with mode, and therefore such a syntac S is termed a "mode-independent" syntax of ALGOL 68.

The mode-independent syntax of ALGOL 68 is by no means uniquely determined by those two requirements, and several implementors have each constructed their own mode-independent syntax [2, 3, 4, 5], as we did.

To this mode-independent syntax the actions and functions that comprize the context-sensitive part must then be appended in some way. How this can be done in a clean and elegant way is exemplified in Section 2.

1.2. Incompleteness of the syntax

In the Report, under the heading of Syntax, is given the syntax of the "strict language". In order to treat the full language, one has to incorporate the following into the syntax:

a) Extensions

The Report defines the strict language, and then proceeds to define extensions, which allow one to write other forms for specific language constructs. This extends the range of meaningful constructs, and sometimes enlarges the power (e.g., the "case clauses" are introduced in this way) but paradoxically most extensions have the character of allowing a contraction. Thus,

ref real x = loc real

may be shortened

to *real x*

and *real x, real y*

to *real x, y* .

Pragmatically indispensable, these extensions are a headache for the implementor, because of the curious interactions between various extensions, which give rise to a number of local ambiguities, small, but irritating because they lengthen the syntax and make it less transparant.
Example:

```
struct a = (real a),  | (real b) b
struct a = (real a),  | b | = (real b)
struct a = (real a),  | b | b
```

b) Context conditions

The context conditions may be seen as a collection of syntactical restrictions which have not been included in the syntax itself. To give an example, the identification conditions assure the correspondence in mode between defining and applied occurrences. They might have been included in the syntax of the Report by the addition to a number of rules of an appropriate metanotion to make it aware of the block- and scope-structure of the program. This might have led to an interesting formalization of what is now a number of semantic remarks scattered throughout the Report, but it would also have reduced its readability even further.

A syntax of the full language must include an explicit taking into account of these context conditions. To this end not only an identifier-table must be kept, but also some tables specific to ALGOL 68: mode-indicant-table, operator-indicant-table and declarer-table.

Rather than let the updating of those tables happen by magic outside the syntactic realism, we have chosen to make use of a syntactic formalism which allows one to define the treatment of those tables as syntax-directed transduction, just like the parsing of the program itself.

c) Some of the Semantics

Some of the semantics of the Report have to be included in the syntax, because they have a direct bearing on the compilation phase, rather than on the run phase.

An example is the semantics of "protection" (R6.0.2.d), a concept needed for the description of the identification process which is followed during compilation; it is not normally a process performed at run time.

From these three points it should be clear that the syntax of ALGOL 68 is by no means a complete basis for syntax-directed translation.

1.3. Removal of backtracking

Because we want to apply top-to-bottom parsing techniques efficiently, the

grammar must satisfy certain restrictions, the most important being that it has to be free from backtracking.

A look-ahead of at least one symbol is necessary, as can be seen from:

begin	*l*	*m*
begin	*l*	*:*
begin	*l*	*of t*
begin	*l*	*+*

Upon reading *begin l*, *l* might be either a label-identifier, a field-selector or a mode-identifier, or the first symbol of one of these. The first symbol following it resolves this local ambiguity.

The long-symbol may begin a declarer (R7.1.1.d), a denotation (R5.1.0.1b and R5.2.1.b), a mode-indication (R4.2.1.b) or an operator-indication (R.2.1.e,f). Thus, upon meeting *begin* ¦ *long* ¦ one is rather uncertain about the kind of construction one is about to meet. Especially, another long-symbol might follow, and it is the first symbol following which is not a long-symbol that resolves the local ambiguity. In this case there is no harm in looking ahead past the long-symbols: even though there is no limit to the number of long-symbols, no appreciable efficiency is lost by leaving this local ambiguity in, and backtracking when the occasion arises.

Quite another matter is the local ambiguity exemplified by:

begin	*[1 : 10] real*	*x = loc [1 : 10] real*
begin	*[1 : 10] real*	*y := (1, 2, 3, 4, 5, 6, 7, 8, 9, 0)*
begin	*[1 : 10] real*	*:= (1, 2, 3, 4, 5, 6, 7, 8, 9, 0)*
begin	*[:] real*	*: (1, 2, 3, 4, 5, 6, 7, 8, 9, 0)*

After the begin-symbol, one does not know whether a unit or a declaration starts. Upon meeting a declarer like, e.g., *[1 : 10] real*, this might be a formal-declarer, beginning a strict identity-declaration, an actual-declarer beginning a contracted identity-declaration, an actual-declarer serving as generator or a virtual-declarer beginning a cast. Certainly one is not entitled to look ahead beyond the declarer or to backtrack across it, since the declarer may again contain a whole program.

It is possible to split and rearrange the mode-independent syntax in such a way that these four cases are recognized without backtracking, at the cost

of making the syntax differ more and more from that of ALGOL 68, so that it becomes very difficult to prove afterwards that the language recognized is indeed ALGOL 68.

The most striking example of local ambiguity is the uncertainty one is in after an open parenthesis. An open parenthesis may begin a collateral-clause (R6.2), closed-clause (R6.3), conditional-clause (R6.4), case-clause (R9.4c,d), conformity-case-clause (R9.4.g), not to mention the incredible complication that it also serves as alternative representation of the sub-symbol. In some cases this gives less problem that one might suppose, viz., when the semantics coincide. Thus, if one meets:

(real x; x:= y + 17

one can open a new block, declare x and translate an assignment statement without being bothered by the fact that one does not know whether one has embarked upon a closed-, conditional- or case-clause: it makes no difference, for the time being, as regards the semantic actions to be taken; only upon later meeting with a then-symbol or a close-symbol one has to proceed differently.

The real trouble is given by a case where the semantics differs markedly from that of closed-clauses, e.g. the parameters-pack:

```
m a = | (real a, b, c | ) real : skip
m a = | (real a, b, c | ; 3.14)
```

In the first case a, b, and c envelop the mode 'real' whereas in the second case they envelop the mode 'reference-to-real'. It is not easy to reconcile semantics so different; therefore one likes to know beforehand what open-symbols are the beginning of a formal-parameter-pack. Rather than performing some look-ahead or backtrack, we let a prescan of the ALGOL 68 translator mark those open-symbols, as one of its tasks.

From these few examples it should be clear that the syntax used has to be closely scrutinized, restructured and refined in order to make it sufficiently deterministic.

1.4. Treatment of errors

One is forced to take into account that the translator will be faced

with incorrect programs; to all probability this will be the rule instead of the exception.

We want to translate programs written in a very large sublanguage of ALGOL 68, and to give errormessages for every deviation from that sublanguage. The parser is not a "permissive" one, which accepts, besides correct strings, also whole lots of rubbish without warning: a full syntax check is made. Errors must be reported in as sensible and informative a manner as possible; the poor programmer must get all possible assistance and not be sent into the woods with the information that at the second symbol of the thirtieth line the parser has given up in disgust. Incorrectness has to be reported at the appropriate syntactic level, an attempt at recuperation has to be made, and great care must be taken that no error can wreck the structure or contents of the various tables. If, e.g., we meet *ref x;* we must report that the symbol *ref* is not followed by a declarer; then we may give to the abortive declarer *ref* some special mode 'erroneous' and declare *x* with the mode 'reference-to-erroneous'.

The whole of this error treatment has to be included explicitly in the syntax used.

An example of this will occur in section 2.

The consideration of error treatment again causes differences between the syntax in the Report and the syntax actually used.

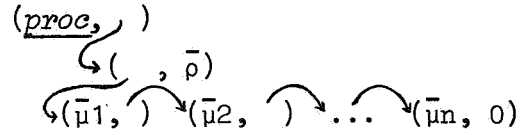
2. An annotated example: declarers

Rather than give a formal description of CDL, the input language of the compiler, an example is given together with some explanation, viz., the treatment of declarers (R7.1).

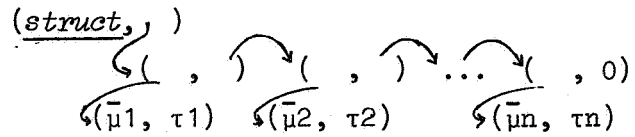
The semantics of 7.1 tells that "a given declarer specifies the mode enveloped by its original". This semantics we will have to put into the syntax: parsing a declarer must yield, as a side effect, its mode. We will record the modes in the declarer-table by (head, tail) pairs, and represent a mode by a numerical key to the recording of a declarer specifying it. For the declarer *int* we record (*int*, 0) and similarly for other primitive declarers. Letting $\bar{\mu}$ stand for the key to the recording of the declarer μ, we enter into the declarer-table the following:

long μ is recorded as (*long*, $\bar{\mu}$)
ref μ as (*ref* , $\bar{\mu}$)
[ ] μ as (*row* , $\bar{\mu}$)

proc (μ1, μ2, ..., μn) ρ is recorded as

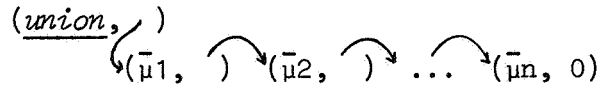


struct (μ1τ1, μ2τ2, ..., μnτn) is recorded as



and

union (μ1, μ2, ..., μn) is recorded as



Furthermore, a mode-indicant μ, standing in a range with range number ν, is recorded as (μ,ν). Thus, a primitive declarator is treated as a mode-indicant with range number zero.

Pairs are added to the declarer-table by the action enter, giving as result a key, taking care that no pair is entered twice, in that case giving the key of the old pair as result. It can easily be shown that declarers which are recorded as the same pair specify the same mode; thus, automatically, some of the equivalent declarers get the same key.

We will show how the syntax of declarers (R7.1) looks when written in Compiler Description Language. The first rule is:

```
declarer + mode - key:
   mode indication + key, enter + key + rangenumber + mode;
   primitive declarator + mode;
   rows declarator + mode;
   long declarator + mode;
   structure declarator + mode;
```

```
reference declarator + mode;
procedure declarator + mode;
union declarator + mode.
```

This rule says that, in order to find a declarer, a number of alternatives are to be tried one after the other; if a mode indication is present, then the pair (key, rangenumber) is entered and a key obtained, which is then the resulting mode of the declarer; otherwise, a primitive declarator is sought, and so on, until some alternative is succesful, or even the last fails; if that is the case, then no declarer was present. On the one hand, this can be seen as a syntactic rule in Van Wijngaarden notation (+ and - being separation marks, mode and key being metanotions); on the other hand, it can be seen as a declaration for a *boolean procedure declarer*, with a parameter *mode* and a local variable *key*. It lends itself to both interpretation as syntax and automatic translation to a *procedure*, performed by a compiler compiler.
More rules:

```
primitive declarator + mode:
   integral symbol, enter + int + 0 + mode;
   real symbol, enter + real + 0 + mode;
   boolean symbol, enter + bool + 0 + mode;
   character symbol, enter + char + 0 + mode;
   format symbol, enter + format + 0 + mode.
```

Here, int, real, bool, char, format and 0 are terminal symbols, i.e., constants.

```
rows declarator + mode - m1:
   subsymbol, rower, rest rows declarator + m1, enter + row + m1 + mode.
```

Here, row is terminal, m1 a local variable.

```
rower:
   bound,
      (up to symbol, bound option;
         error + "up to symbol expected", skip rest list elem);
   up to symbol, bound option; .
```

We will not here define error and skip rest list elem, but only state that the first causes an errormessage to be printed, the second skips symbols until a comma or closing symbol is met.
The notation with the brackets in the rule means the grouping together of a number of alternatives into one; thus, error is only called when a bound is not followed by an up to symbol and a bound option.

```
bound:
   tertiary, (flexible symbol; either symbol;  ).
```

The tertiary is optionally followed by a flexible symbol or either symbol. Note the inclusion in the syntax of R9.2.f.

```
bound option : bound; .
rest rows declarator + mode - m1:
   comma symbol, rower, rest rows declarator + m1, enter + row + m1 + mode;
   bus symbol, must be declarer + mode;
   error + "incorrect row-of- ... declarator", skip rest list elem,
            rest rows declarator + mode.
must be declarer + mode:
   declarer + mode; error + "declarer expected", make + mode + erroneous.
long declarator + mode - m1:
   long symbol,
      (integral symbol, enter + int + 0 + m1, enter + long + m1 + mode;
       real symbol, enter + real + 0 + m1, enter + long + m1 + mode;
       long declarator + m1, enter + long + m1 + mode;
       backtrack, omega).
```

The procedure backtrack backtracks over one symbol, the boolean procedure omega always yields *false*.

```
structure declarator + mode - m1:
   structure symbol, rest structure declarator + m1,
      enter + struct + m1 + mode.
```

```
rest structure declarator + mode - m1 - m2 - m3 - tag:
   open symbol, field + m1 + tag, enter + m1 + tag + m2,
      fields + m1 + m3, enter + m2 + m3 + mode;
   error + "incorrect structured-with- ... declarator",
      make + mode + erroneous.

field + mode + tag:
   must be declarer + mode,
      (identifier + tag;
       error + "missing fieldselector", make + tag + 0).
fields + m1 + mode - m2 - m3 - tag:
   comma symbol,
      (identifier + tag, enter + m1 + tag + m2,
         fields + m1 + m3, enter + m2 + m3 + mode;
       field + m1 + tag, enter + m1 + tag + m2,
                   fields + m1 + m3, enter + m2 + m3 + mode);
   close symbol, make + mode + 0;
   error + "incorrect field pack", skip till closed, make + mode + erroneous.
```

The procedure skip till closed skips symbols up to and including the first closing bracket at the correct bracketlevel.
Notice how extension R9.2.c has been incorporated in the rule for fields.

```
reference declarator + mode - m1:
   reference to symbol, must be declarer + m1, enter + ref + m1 + mode.
procedure declarator + mode - m1:
   procedure symbol, plan + m1, enter + proc + m1 + mode.
plan + mode - m1 - m2:
   parameters + m1, result + m2, enter + m1 + m2 + mode.
parameters + mode:
   declarer list pack + mode; make + mode + 0.
```

The primitive action make assigns the second parameter to the first, i.e., assigns 0 to the parameter mode.

```
declarer list pack + mode - m1 - m2:
   open symbol, must be declarer + m1, rest decl list pack + m2,
                enter + m1 + m2 + mode.
rest decl list pack + mode - m1 - m2:
   comma symbol, must be declarer + m1, rest decl list pack + m2,
                enter + m1 + m2 + mode;
   close symbol, make + mode + 0;
   error + "incorrect declarer list pack", skip till closed,
                make + mode + erroneous.
result + mode:
   declarer + mode; make + mode + void.
union declarator + mode - m1:
   union of symbol, rest union declarator + mode,
                enter + union + m1 + mode.
rest union declarator + mode - m1:
   declarer list pack + m1;
   error + "incorrect union-of- ... declarator",
                make + mode + erroneous.
```

Finally, we will show that the procedure enter itself can also be defined by a rule in CDL, making use of a number of primitive actions and boolean procedures. In CDL, one can add definitions for primitives in the form of macro-definitions. We will use the following primitives:

incr + a	must increment a by one
lseq + a + b	must yield *true* if $a \leq b$, and *false* otherwise
make + a + b	must assign the value of b to a
equal + a + b	must yield *true* if $a = b$
get head + p + d	must put the value of the head of the pair indexed by p, into d
get tail + p + d	idem for the tail of the pair
put head + p + d	must put the value of d into the head of the pair indexed by p
put tail + p + d	idem for the tail of the pair.

The latter four are the means of access for the declarer-table, the others provide the little calculation capability necessary to define the action enter.

We assume global variables min decl and pdecl to point to the first and last pair in the declarer-table respectively.

```
enter + d1 + d2 + mode - x - y - wy:
   make + x + pdecl, incr + pdecl, put head + pdecl + d1,
                put tail + pdecl + d2, make + y + min decl,
   nxy : (get head + y + wy, equal + wy + d1,
          get tail + y + wy, equal + wy + d2,
                (lseq + y + x, make + pdecl + x, make + mode + y;
                 make + mode + pdecl);
          incr + y,: nxy).
```

The notation nxy : stands for the label *nxy*, and : nxy for *goto nxy*. We allow labels and jumps as a means to replace some recursion by the more efficient iteration. Local variables are y, wy and x.
A possible translation into ALGOL 60 is:

```
procedure enter (d1, d2, mode); integer d1, d2, mode;
begin integer   y, wy, x;
      x:= pdecl; pdecl:= pdecl + 1;
      head [pdecl]:= d1; tail [pdecl]:= d2; y:= mindecl;
nxy : wy:= head [y]; if ¬(wy=d1) then goto 1;
      wy:= tail [y]; if ¬(wy=d2) then goto 1;
      if ¬ (y≤x) then goto 2;
      pdecl:= x; mode:= y; goto end;
  2 : mode:= pdecl; goto end;
  1 : y:= y + 1; goto nxy;
end:
end;
```

This is indeed very near to the translation our compiler compiler makes for it.

3. The compiler compiler

The compiler compiler used accepts input in Compiler Description Language, and gives output in one out of a number of object languages. For experimentation puposes we use ALGOL 60 as an object language, whereas other versions will have as object language PL/360 (which runs on some IBM computers), Compass (assembler for some CDC computers) and ELAN (assembler for the Electrologica X8). We will not go here into the properties or workings of the compiler compiler (described in [6]), but only state that it is actively being used in the study and development of compilers for ALGOL 68 by a number of groups. It is our intention to allow other interested groups full access to our tools and results, and contribute in this way to a future widespread availability of good ALGOL 68 translators, which make special-purpose languages like CDL superfluous.

References

[1] J.E.L. Peck (Editor), "Proceedings of a conference on ALGOL 68 implementation", North Holland publishing company, 1969.

[2] H.J. Bowlden, "ALGOL 68 structural flowchart", Westinghouse Research Laboratories, Research Report 69-1c4-comps-R2, October 1969.

[3] G.S. Hodgson, "ALGOL 68 extended syntax", University of Manchester, March 1970.

[4] M. Simonet, "Une grammaire context-free d'Algol 68", Rapport IMAG, Grenoble, June 1969.

[5] P. Branquart, J. Lewi and J.P. Cardinael, "A context-free syntax of ALGOL 68", Technical Note N66 of MBLE, Brussels, August 1970.

[6] C.H.A. Koster, "A compiler compiler", MR 127, Mathematisch Centrum, Amsterdam, November 1971.