stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 180/81 OKTOBER

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

MAKING ALGOL 68+ TEXTS CONFORM TO AN OPERATOR-PRIORITY GRAMMAR

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: 68F25, 68B20

ACM-Computing Reviews-category: 5.23, 4.22, 4.12

Making ALGOL 68+ texts conform to an operator-priority grammar

by

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

ABSTRACT

ALGOL 68+ is a superlanguage of ALGOL 68 which is powerful enough to describe the standard-prelude. An operator-precedence grammar can, through a simple right-to-left transduction scheme, be made to be of type LL(1). If, in addition, the grammar is an "operator-priority" grammar, an easy and consistent error-recovery mechanism can be applied. In an earlier report, an operator-priority grammar of ALGOL 68+ has been given. The main difference between this grammar and an underlying context-free grammar of ALGOL 68+ is that (i) symbols represented by the same mark have been distinguished, and (ii) various symbols have been inserted in the grammar. The present report gives a detailed account of how these changes can be taken care of during the first phases of an ALGOL 68+ implementation.

KEY WORDS & PHRASES: ALGOL 68+, lexical analysis, syntax-directed transduction

1. INTRODUCTION

ALGOL 68+ is a superlanguage of ALGOL 68 [1] which is powerful enough to describe the standard-prelude. Besides this, ALGOL 68+ also encompasses the official IFIP modules and separate-compilation facility as given in [2]. The changes and additions to the language needed to be able to process a version of the standard-prelude are of a fairly simple nature; they are described in [3].

For an operator-precedence grammar, at most one of three relationships (denoted by $\lessdot$, $\doteq$, or $\gtrdot$) may hold between each pair of terminal symbols. These relationships are called the precedence relations. (For a formal treatment of operator-precedence grammars, see [4].) For an operator-precedence grammar, it is possible to construct a transducer [5] which, operating from right to left, brings the source text in prefix form, only knowing the precedence relations between the symbols.

In general, a number of entries in the table of precedence relations is empty, i.e., there is no precedence relation between certain pairs of terminal symbols. For correct input texts, this is no problem, since the transducer will never need them. For incorrect input texts, however, the transducer might well ask for them. In order to let the transducer work for all input texts, it is therefore necessary to define precedence relations for the empty spots as well. For an arbitrary operator-precedence grammar, it is not clear how to fill these empty spots 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+ is given in [6].

In order to apply the above-mentioned right-to-left transduction scheme, the parenthesis skeleton should be correct, for, if the transduction scheme is applied bluntly to a source text with an incorrect parenthesis skeleton, the result is in general unacceptable. To this end, one can either try to repair the parenthesis skeleton during lexical analysis if it turns out to be incorrect (e.g., using the algorithm given in [10]), or decide to abort the parsing process altogether. In the discussion below (and especially in section 2.8), it is assumed that all parentheses match properly.

The right-to-left transduction scheme can also be applied to the operator-priority grammar. Care has been taken to ensure that the prefix-form of that grammar is of type LL(1). If a grammar is of type LL(1), this easily leads to a parsing method for that grammar, implemented by a set of mutually recursive routines, one for each non-terminal of the grammar. Using such a parser, there is no need to back up, since it is decidable which rule to apply (i.e., which routine to call) by looking at most one symbol ahead. A more formal treatment of LL(1) grammars and parsers based on them can be found in [4].

This combined scheme (a syntax-directed transduction based on an operator-priority grammar and a subsequent top-down syntax analysis), together with the associated consistent treatment of erroneous input texts, is further dealt with in [7]. The emphasis in this report is on the derivation of an algorithm which transforms ALGOL 68+ texts into sentences of the language produced by the operator-priority grammar.

The measures taken to make the grammar operator-priority can be distinguished in four categories:

a. Trivial rearrangements of the syntax. This has mainly been done by considering some notions as macros, to be replaced (conceptually) in the productions in which they occur by their direct productions. Obviously, this trick can only be used for nonrecursive notions. In the grammar (see [6]), these notions are indicated by prefixing their production rules with an asterisk.
b. Distinguishing symbols represented by the same mark. For instance, it was necessary to distinguish between the up-to-/label-token, the specification-token and the routine-token. For a complete list of this category, see section 2 below.
c. Various symbols have been inserted between notions. For instance, a "dectag-insert" is placed between a declarer and the following TAG-token in an identifier-declaration. Again, section 2 contains a complete account of the modifications from this category.
d. Relaxations in the grammar. For instance, closed-clauses and collateral-clauses are treated alike.

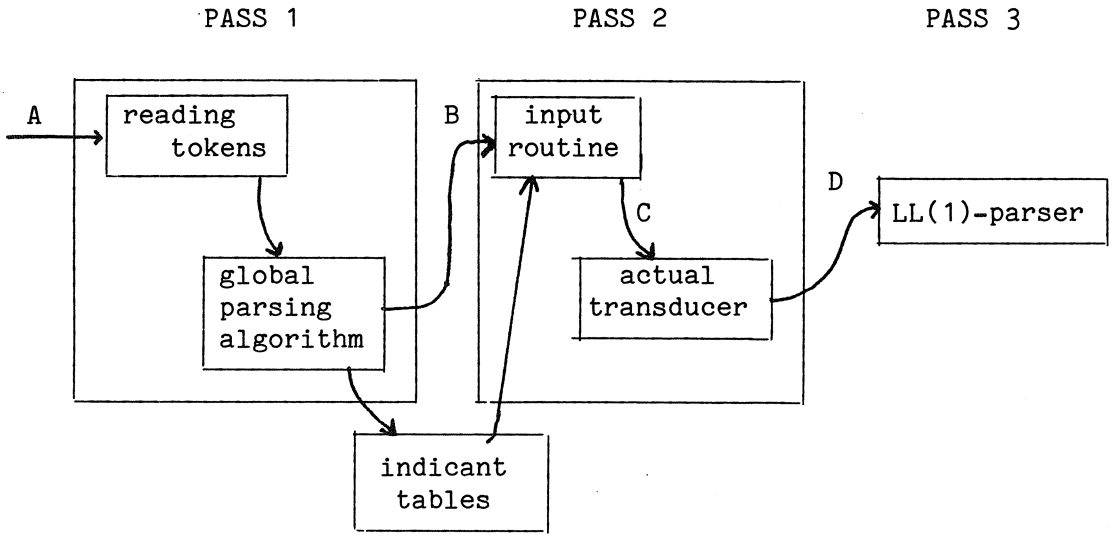
(The function of the changes in categories a and c is to separate any two notions in a production rule by at least one terminal symbol, which is mandatory in an operator-precedence grammar. The changes in category b serve to resolve clashes in the precedence relations. The changes in category d mainly serve to fulfill the operator-priority requirements and to allow for the top-down parsing method using the prefix-form of the operator-priority grammar.)

When actually parsing ALGOL 68+ texts, the same modifications must be made. Category a does not change the language generated, while category d only enlarges the set of accepted sentences (which must then again be catered for during further syntax-analysis). In this report, a description is given of how the distinctions from category b and the insertion of additional symbols from category c can be handled.

Some of these changes can be dealt with quite easily during lexical analysis. Others, however, require a more global knowledge of the input-text. For example, in a context like "p a;", a "dectag-insert" must be placed between "p" and "a" only if p is a mode-indication. Much more complicated examples can be found when various constructs enclosed by the symbols "(" and ")" are considered. In such cases, as much information as possible is gathered during lexical analysis, and the final decision as to which change applies can then be made in the input routine of the actual transducer, by inspecting the various indicant tables. (The indicant tables must be partly filled by the lexical analyzer with

information concerning defining occurrences of mode-indications, module-indications and operators. They may be pre-filled if pieces of a program are compiled separately.)

The global (but very incomplete) scheme of the first three passes of the parser now looks as follows:



A,B,C and D are streams:
A contains the ALGOL 68+ input text;
B contains lexical units (like identifiers), and is the partly transformed version of the input text;
C contains the completely transformed version, i.e., conforms to the operator-priority grammar;
D contains the prefix-form of C and can thus be parsed top-down.

Taking the example "(p a)", the various streams might look as follows:

```
A: ... (p a) ...
B: ... (p 'dectag(p) a) ...
C: ... (p 'dectag a) ...        or      ... (p a) ...
D: ... ('dectag p a) ...        or      ... (p a) ...
```

Here, "'dectag (p)" stands for: place the dectag insert 'dectag if p is a mode-indication, and ignore this otherwise.

In the next section, a detailed analysis is given of how and where the various changes and inserts should be effected. In order to be able to fully appreciate this analysis, a fairly thorough knowledge of the syntax of ALGOL 68+ is necessary. Section 3 combines the results of these analyses into a global parsing algorithm to be included in the lexical analyzer.

2. RECOGNIZING SPECIFIC TOKENS AND PLACING INSERTS

The adjustments to be made before the actual transduction scheme can be applied are the following (see also [6]):

- On the lowest level, a distinction is made between
  (as open-mark and as choice-start;
  | as choice-in and choice-out;
) as close-mark and as choice-finish;
  = as is-defined-as-token, egg-defined-as-token and operator;
  : as colon-mark, specification-token and routine-token;
  ~ as skip-token and as operator.

- On the lowest level, a distiction is also made between defining occurrences of operators (in priority- and operation-declarations) and applied occurrences (in formulas and ldec-sources).

- Besides the and-also-token, which separates the individual elements of a list, there is a variant, the separate-and-also-token, which separates lists.

- The grammar contains inserts:
  - the loop-insert marks the beginning of a loop;
  - the ssecca-insert marks the end of the revelation of an access-clause;
  - the dectag-insert is placed between a declarer and the following TAG-token in a declarative, FIELDS-portrayer or identifier-declaration;
  - the opdec-insert is placed between the MODINE-plan and the following defining-operator in an operation-declaration;
  - the cast-insert is placed between the declarer and the ENCLOSED-clause of a cast;
  - the clice-insert is placed between the primary and the actual-parameters-pack or indexer-bracket of a call or slice;
  - the row-insert is placed between the ROWS-rower-bracket and the following declarer of a ROWS-of-MODE-declarator;
  - the formals-insert is placed between a PARAMETERS-joined-declarer-brief-pack or declarative-brief-pack and the following declarer of a PROCEDURE-plan or routine-text;
  - the invoke-insert is placed between the revelation and the following ENCLOSED-clause in an access-clause.

2.1 Recognizing choice-symbols

Obviously, when an input-text of the form

(... | ...)

is encountered, one can not decide that this concerns a choice-clause until the symbol "|" is met. An easy way to solve this is to distinguish choice-in- and choice-out-symbols (both of which may be represented by "|") and to recognize the choice-finish-symbol represented by ")" during lexical analysis, which is straightforward. Since the transducer operates from right-to-left, the "(" can subsequently be recognized by its input routine in a similar way.

2.2 Placing the loop-insert

The loop-insert marks the start of a loop-clause. A loop-clause may start with one of the symbols for, from, by, to, while and do. Except for the first one, all those symbols may also appear in the middle of a loop-clause. If one of these symbols, say by, is encountered, we want a simple procedure to decide which case applies. In a context like " ... ; by ... " it clearly marks the start of a loop-clause, while we are obviously in the middle of one in the context " ... +2 by ... ".

In general, the following can be stated: the symbol by marks the start of a loop-clause if it is the first symbol of an enclosed-clause, and is thus preceded by a symbol which may appear just before an enclosed-clause. Obviously, the same holds for the symbols from, to, while and do.

On the other hand, the symbol by does not mark the start of a loop-clause if it is preceded by a for-part or from-part, and thus by a tag or a unit, respectively. Something similar holds for the symbols to and while; the symbol from may only be preceded by a tag in this case. If the symbol do does not indicate the start of a loop-clause, it must be preceded by a tag, unit or enquiry-clause, the last one of which ends in a unit again. It is therefore reasonable in all cases to test for symbols which may mark the end of a unit.

An enclosed-clause may be preceded by one of the following symbols:

:=[1]	:=:[1]	:≠:[1]	=[1]	:	,	;	[[1]	@[1]	begin
if	then	elif	else	case	in	ouse	out	of[1]	from[1]
by[1]	to[1]	while	do	(	|	|:	)[1]	def	postlude
operator[1]		mode-indication			module-indication				

Remarks:
1) The symbols with superscript 1 may only precede a SORT MODE ENCLOSED CLAUSE. Since a loop-clause is only allowed in a (strong-) void context, they are disallowed here. Therefore, the symbol ~, when used to represent the operator not, is not allowed in this context.
2) A program [8] was used to determine the above set of symbols from the context-free grammar of ALGOL 68+ as given in [9]. The set of symbols with superscript 1 was determined manually by inspecting the original syntax of ALGOL 68+.
3) It should be noted here that pragmats are not taken into account. In ALGOL-68 terminology this means that we consider 'tokens', rather than 'symbols'.

A unit may end with one of the following symbols:

end	fi	esac	]	nil	od	)	skip {~}
tag	format-text		denotation			mode-indication.	

Taking remark 1 into account, the two sets may be called disjoint, except for the mode- and module-indication (which can not be distinguished at this level). In order to give a decisive answer in the case of a mode- or module-indication, a more complicated reasoning is needed.

Given the context at hand (a bold word followed by, say, to), there are three possibilities:
i) we are concerned with an access-clause, as in "... access m to ...";
ii) we are concerned with a cast, as in "... ; m to ...";
iii) we are concerned with a generator, as in "... loc m to ...".

In the first two cases, the start of a loop-clause is indicated; in the latter case we are inside a loop-clause. It is possible to decide which case applies by considering the symbol immediately preceding the bold word.

In the case of an access-clause, the module-indication is preceded by one of the following symbols

access , pub

In the case of a cast, the mode of the declarer of that cast is VOID, so the declarer consists of a single mode-indication. That mode-indication therefore is the first symbol of the cast, and is preceded by a symbol which may immediately precede a cast. The symbols that may precede a cast are the same as those that may precede an enclosed-clause, with the exception of the close-symbol, mode-indication and module-indication.

In the case of a generator, the set of symbols which may precede the mode-indication consists of the symbols loc and heap, plus those symbols which may immediately precede a mode-indication in an actual-MODE-declarer. By inspecting section 4.6.1 of the Revised Report, we arrive at the following set:

loc heap ref)] proc flex

In this way, the loop-insert can be placed at the symbol-level during lexical analysis, by expecting the two preceding symbols. In case one cannot give a decisive answer (i.e., there is an error in the input text, as in the case of "... op by ..."), we have decided to place a loop-insert provisionally. During an eventual correction phase of the parenthesis skeleton (see [10]), this provisional insert can be removed again, if such comes out better.

2.3 Recognizing the separate-and-also-token

The separate-and-also-token serves to separate common-declarations, common-declaratives, common-portrayers and module-calls. To be able to distinguish these, it is necessary to know which of the bold words that are defined in the program are mode-indications, module-indications and operators, respectively. Since this is in general not known until at the end of the lexical phase, this problem can most easily be dealt with in (the input routine of) the next phase.

An and-also-token must then be changed into a separate-and-also-token if it is followed by one of the symbols mode, op, prio, module, pub and 'ldec, or a construct of the form

declarer, dectag insert, identifier.

This last case can be recognized if some additional information (viz., the fact that a dectag-insert has been placed) is obtained from the actual transducer. The joined-module-call must be treated in a special way; it can easily be dealt with during the input routine of the transduction phase (see section 2.4 below).

2.4 Ssecca-insert and invoke-insert

In a context like

module a = access b, c def ... fed, module d = ...

both and-also-tokens will be transformed into a separate-and-also-token. However, they occur at different levels in the parse tree. So, in order to let the transducer work properly, we must ensure that the first and-also-token is viewed to occur within some nested parenthesized construct.

We may consider a parenthesized construct of the form access ... def ... fed. Since the revelation of a module-text (the part "access b, c") is optional, we then have to recognize the start of a module-text, just like we had to recognize the start of a loop-clause (section 2.2 above).

Revelations may also appear in ENCLOSED-clauses, for instance in a context

int i = access b, c (...), real z:= ...

Again, both and-also-tokens will be transformed into a separate-and-also-token, and again they occur at different levels in the parse tree. If we consider a construct of the form "access ... (...)" as one parenthesized construct, it will be necessary to recognize the start of almost every parenthesized construct, which is clearly undesirable.

We therefore decided to introduce an explicit closing parenthesis to match access (and termed it ssecca-insert). This in turn leads to problems with regard to the operator-precedence requirements. Therefore, an additional invoke-insert is placed after the ssecca-insert.

Both inserts can be placed already during lexical analysis. However, technical complications then arise when trying to place some of the other inserts between parenthesized constructs. We therefore decided to place only the ssecca-insert during lexical analysis, and to incorporate the invoke-insert in the scheme used to handle sequences of parenthesized constructs (see section 2.8 below).

2.5 Recognizing the egg-defined-as-token

The egg-defined-as-token is the equals-mark from the stuffing-definition (see [2]). Therefore, the equals-mark must be recognized in a context like

egg "a" = ...

This can easily be accomplished during lexical analysis.

2.6 Dectag-insert, opdec-insert and is-defined-as-token

These are all concerned with the begin pieces of "declarations" (which also includes declaratives and portrayers). The dectag-insert is placed between a declarer and the following identifier in a declarative, portrayer, identity- and variable-declaration. The opdec-insert is placed immediately after the MODINE-plan in an operator-declaration. The is-defined-as-token replaces the equals-mark when it is used as such in the grammar of ALGOL 68+ (except in a stuffing-definition; see section 2.5 above).

Since it is generally not known during lexical analysis whether a bold word is used as mode-indication, module-indication or operator, both the dectag-insert and the is-defined-as-token are in general placed conditionally. In the input routine of the next phase, this condition is known and the decision can be taken.

In a number of cases the decision can be taken on the basis of the immediate context:

```
; real a      -->     ; real 'dectag a
mode m =      -->     mode m  'idat
```

In other cases, like " ... , i = ... ", the input text (and especially declarations) must be analyzed globally. A precise description of this parsing algorithm is given in section 3 below.

2.7 Recognizing the specification-token and routine-token

The specification-token is the colon-mark from the specification of a choice-using-UNITED-clause; the routine-token is the colon-mark from a routine-text. The specification-token can not be recognized until the transduction phase, since the type of the parenthesized construct just preceding it determines whether or not it concerns a specification (see also section 2.8 below). The routine-token is in general placed conditionally during lexical analysis. The condition here is: is the bold word just preceding it a mode-indication? If it is preceded by a "visible" declarer (like, e.g., real) the routine-token can be placed during lexical analysis unconditionally.

2.8 Cast-insert, clice-insert, row-insert, formals-insert and invoke-insert

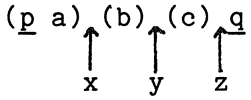
The cast-insert serves to separate the declarer and the enclosed-clause of a cast, like in "real(x)". The clice-insert is placed between the primary and the actual-parameters-pack or indexer-bracket of a call or slice, like in "sin(3.14)" or "a[1]". The row-insert separates the ROWS-rower-bracket and the following declarer of a ROWS-of-MODE-declarator, as in "[1:3] int i". The formals-insert is placed between a PARAMETERS-joined-declarer-brief-pack and the subsequent declarer of a PROCEDURE-plan or routine-text, as in "(real x) void: p". Finally, the invoke-insert serves to separate a revelation from the following enclosed-clause in an access-clause, like in "access a (...)".

These inserts have two aspects in common: Firstly, they are all concerned with parenthesized constructs. In general, a sequence of parenthesized constructs must be considered, and a sequence of (possibly different) symbols must be inserted. For example, in

```
ref [] real ( ... ) [ ... ]
```

a row-insert, a cast-insert and a clice-insert must be placed, in the order from left to right.

Secondly, the precise types of the inserts to be placed often depend on the fact whether a given bold word is a mode-indication or not. Consider, for example, an input text of the form



Depending on the type of p and q, different combinations of symbols must be inserted at the places indicated by x, y and z:

i) if both p and q are mode indications, then: x=formals-insert, y=z=row-insert;
ii) if p is an operator and q is a mode-indication, then: x=y=z=row-insert;
iii) if p and q are both operators, then: x=y=clice-insert, z=empty;
iv) if p is a mode-indication and q is an operator, or any of p and q is a module-indication, then the input-text is erroneous.

The type of the various bold words is in general not known during lexical analysis. In order to preclude the very complicated situations that may arise because of this, these inserts will be placed by the input routine of the transduction phase. The information necessary to decide which inserts must be placed once the type of each bold word is known, is gathered during lexical analysis and placed after the sequence of parenthesized constructs. For each sequence of parenthesized constructs $P_1 \ldots P_n$, this information consists of:

i) The "protostate" just prior to P_1, and
ii) The "prototype" of each P_i, $1 \leq i \leq n$.

The prefix "proto" serves to emphasize that the information depends on the type of the bold words involved. These types are known at the start of the next phase, so its input routine can immediately turn each "protostate" and "prototype" into a "state" and "type", respectively. The finite-state automaton given below is driven by the states and types thus obtained.

Given the initial state b_1, and types $p_1, \ldots, p_n$ for $P_1, \ldots, P_n$, a sequence $a_1, \ldots, a_{n+1}$ of inserts will now be determined by a finite-state automaton as follows: The tuple (b_1, p_1) determines the insert a_1 and a new state b_2. Subsequently, the tuple (b_2, p_2) determines the insert a_2 and a new state b_3, and so on. Finally, the state b_{n+1} determines the insert a_{n+1}. For $1 \leq i \leq n$, a_i will be inserted just prior to P_i'; a_{n+1} will be inserted just after P_n', where P_i' is the result of applying the transduction to P_i.

In the sequel, the term "pack" will be used, rather than "parenthesized construct". It will be used to denote any construct of the form

access	... ´ssecca
def	... fed
´loop	... od
if	... fi
case	... esac
begin	... end
(	... {|} ...)
[	...]
par	

Remarks:
1) The single parallel-symbol is considered as a pack. This leads to a reasonably simple, albeit somewhat ad hoc, treatment.
2) The constructs struct (...) and union (...) are supposed to be transformed into some kind of mode-standard during lexical analysis; they are not taken into account by the scheme developed here.
3) A pack of the form "def ... fed" will be termed a "module-pack" in the sequel.

If, in the underlying context-free grammar of ALGOL 68+ [9] and in the corresponding operator-priority grammar [6], each parenthesized construct occurring in the right-hand-side of a production rule is replaced by some terminal symbol, regular languages L and L´ are obtained, respectively. The finite-state automaton given below is precisely the automaton which transforms L into L´. In the discussion below, only the various possible states and types will be given, together with the transition-matrix which drives the finite-state automaton. Most, if not all, of these transitions will be obvious.

As concerns the state just prior to the pack-sequence, the following cases are distinguished:

i) "cliceable", i.e., there occurs a simple primary: an identifier or a string-denotation;
ii) "decl", i.e., there occurs a mode-indication (which includes the mode-standards!);
iii) "decpref", i.e., we are clearly about to start a declarer, as for instance after loc or heap;
iv) "modtext", i.e., we are about to start a module-text, the right-hand-side of a module-declaration;
v) "rest", all other cases (which also includes the possibility that we are about to start a declarer which is not yet recognized as such).

As concerns the type of a pack, the following cases are distinguished:

i) "par", for a pack consisting of a single parallel-symbol;
ii) "formals", for a pack consisting of a list of declarers, portrayers or declaratives, surrounded by an open- and close-mark;
iii) "brief pack", for any other construct surrounded by an open- and close-mark;
iv) "subbus", for a construct surrounded by a brief-sub- and brief-bus-symbol;
v) "revel", for a revelation, i.e., a construct of the form access ... 'ssecca;
vi) "deffed", for a module-pack, i.e., a construct of the form def ... fed;
vii) "bold pack", for any other pack.

In certain cases, this information obviously depends on the type of a given bold word. In such cases, the bold word is included in the information to be passed on to the input routine of the transduction phase, which then determines the actual state or type. This in fact means that the state "decl" and the type "formals" are conditional. If the bold word in question turns out to be an operator or module-indication, they will be transformed into the state "rest" and type "brief pack", respectively.

From the five possibilities given above for the state just prior to the pack-sequence, only the state "cliceable" is left as a possible state after the pack P_1. The other four possibilities only serve as possible entries for the automaton. However, seven new possibilities occur as a possible state after the first pack of the pack-sequence:

i) "par", i.e., we have just treated a single parallel-symbol;
ii) "rower", i.e., starting with a state "decpref" we have processed a rower (a pack with type "subbus" or "brief pack");
iii) "formals", i.e., we have just treated a pack with type "formals";
iv) "cliceable or rower" (or "cor" for short), i.e., we cannot yet decide between "cliceable" and "rower". The final decision will depend on the fact whether or not the pack-sequence is followed by a mode-indication. A temporary insert 'clicerow' is placed; we will come back to this case later on;
v) "acliceable", i.e., we have just treated a revelation; eventually, there has to follow a call or slice, but any number of revelations is allowed in between;
vi) "deffed", i.e., we have just treated the revelation of a module-declaration;
vii) "done", i.e., the pack-sequence should be ended; we will come into this state after a pack following par, and after a module-pack.

type of pack ->
state
|
v

	par par	(real a) formals	(a;b) brief pack	[ ] subbus	begin end bold pack	access a revel	def...fed deffed
decl	´cast par		´cast cliceable		´cast cliceable	´cast acliceable	
rest	ε par	ε formals	ε cor	ε cor	ε cliceable	ε acliceable	
decpref		ε formals	ε rower	ε rower			
modtext						ε deffed	ε done
cliceable			´clice cliceable	´clice cliceable			
par			ε done		ε done		
rower			´row rower	´row rower			
formals			´formals rower	´formals rower			
cor			´clicerow cor	´clicerow cor			
acliceable	´invoke par		´invoke cliceable		´invoke cliceable	´invoke acliceable	
deffed							´invoke done
done							

For each entry, the top line indicates the insert to be placed, while the bottom line gives the new state

Table 1

The finite-state automaton which, given an initial state b_1 and types $p_1, \ldots, p_n$, determines the inserts $a_1, \ldots, a_n$, is driven by the transition matrix given in Table 1. The insert a_{n+1} is determined in a special way from the final state b_{n+1}; this will be further dealt with below. Obviously, this automaton is only capable of handling pack-sequences which are correct at this level; a slight modification which allows a reasonable treatment of erroneous pack-sequences is given at the end of this section. Entries in Table 1 which are marked with an ε indicate that no insert is placed; this is only possible when it concerns the insert just prior to the pack-sequence, or after a parallel-symbol.

For the tuple ("rest", "subbus"), Table 1 indicates a transition to the state "cor". However, it is sometimes possible to distinguish between states "cliceable" and "rower" here. We have decided not to make this refinement; rather, the decision on which insert is to be placed is based on whether or not the pack-sequence is followed by a mode-indication. This probably leads to a better treatment of incorrect input texts. Suppose the input contains something like

```
; [3]:= x;
```

Using the scheme of Table 1, and the algorithm for determining the final insert a_{n+1} and for refining the temporary clicerow-inserts, which is given below, the above text will be treated as

```
; 'wrongtag 'clice [3]:= x;
```

This type of error-recovery needs further investigation.

What remains now is the algorithm to determine the insert a_{n+1} from the final state b_{n+1}. We may end in any state except the ones that serve as an entry to the automaton: "cliceable", "par", "rower", "formals", "cor", "acliceable", "deffed" and "done".

If the final state is "done", there is no symbol to be inserted after the pack-sequence, so $a_{n+1} = \varepsilon$.

Ending in one of the states "par", "deffed" and "acliceable" means that there definitely is something wrong: there should at least have followed yet another pack. The further treatment of these cases should be done during syntax analysis, and $a_{n+1} = \varepsilon$.

If the final state is "cliceable", there also is no symbol to be inserted after the pack-sequence, so $a_{n+1} = \varepsilon$.

If the final state is "rower", an additional row-insert must be placed: a_{n+1} = 'row.

If the final state is "formals", three cases are distinguished:

i) The pack-sequence is followed by a mode-indication or otherwise visible declarer (like "ref ... "). It then obviously concerns a procedure-plan or routine text, and a_{n+1} = 'formals;
ii) When the pack-sequence is followed by a colon-symbol, it concerns a specification, so $a_{n+1} = \varepsilon$. Moreover, that colon-symbol must be transformed into a specification-token (see also section 2.7);
iii) In all other cases there is something wrong. One (reasonable) possibility is to assume that a mode-indication is missing, so a_{n+1} = 'formals.

If the final state is "cor", two cases are distinguished:

i) The pack-sequence is followed by a mode-indication or otherwise visible declarer. We may then decide that it concerns a rower, so a_{n+1} = 'row. Moreover, each clicerow-insert must be changed into a row-insert;
ii) In all other cases we may assume that it concerns a call or slice, so $a_{n+1} = \varepsilon$. Now each clicerow-insert is replaced by a clice-insert.

As mentioned earlier, the transition scheme given in Table 1 is only capable of handling correct pack-sequences. The changes needed to handle incorrect pack-sequences also are fairly simple, however. It is reasonable, and in any case consistent, to partition the pack-sequence $P_1 \ldots P_n$ into two sequences $P_1 \ldots P_{i-1}$ and $P_i \ldots P_n$ as soon as no transition is possible for a state b_i and type p_i, where b_i and p_i are the state arrived at after pack P_{i-1} and the type of pack P_i, respectively. We may then act as follows:

1) The sequence $P_1 \ldots P_{i-1}$ must be finished off, i.e., we must decide on a final insert a_i. The algorithm for determining the final insert a_{n+1}, as given above, can be applied here. In the final states "formals" and "cor" we now have to choose a_i = 'formals (case iii) and $a_i = \varepsilon$ (case ii), respectively. If i = 1, i.e., the entry state is wrong already, there is no need to place a final insert;
2) The sequence $P_i \ldots P_n$ is further treated, starting in a state "rest", since there is no further information.

The above scheme can be implemented straightforwardly. It is also possible to fill in the empty entries from the transition matrix in such a way that the effect is the same. For each empty entry (b, p), the inserts follow from the algorithm above, and the new state is that given in Table 1 for the entry ("rest", p), with the addition that (b, "deffed") leads to a state "done" for each b. The thus adjusted scheme is given in Table 2.

type of pack ->
state
|
v

	par par	(real a) formals	(a;b) brief pack	[ ] subbus	begin end bold pack	access a revel	def...fed deffed
decl	´cast par	ε formals	´cast cliceable	ε cor	´cast cliceable	´cast acliceable	ε done
rest	ε par	ε formals	ε cor	ε cor	ε cliceable	ε acliceable	ε done
decpref	ε par	ε formals	ε rower	ε rower	ε cliceable	ε acliceable	ε done
modtext	ε par	ε formals	ε cor	ε cor	ε cliceable	ε deffed	ε done
cliceable	ε par	ε formals	´clice cliceable	´clice cliceable	ε cliceable	ε acliceable	ε done
par	ε par	ε formals	ε done	ε cor	ε done	ε acliceable	ε done
rower	´row par	´row formals	´row rower	´row rower	´row cliceable	´row acliceable	´row done
formals	´formals par	´formals formals	´formals rower	´formals rower	´formals cliceable	´formals acliceable	´formals done
cor	ε par	ε formals	´clicerow cor	´clicerow cor	ε cliceable	ε acliceable	ε done
acliceable	´invoke par	ε formals	´invoke cliceable	ε cor	´invoke cliceable	´invoke acliceable	ε done
deffed	ε par	ε formals	ε cor	ε cor	ε cliceable	ε acliceable	´invoke done
done	ε par	ε formals	ε cor	ε cor	ε cliceable	ε acliceable	ε done

Table 2

The transition scheme, capable of handling incorrect pack-sequences also.

3. THE GLOBAL PARSING ALGORITHM

In this section, the global parsing algorithm to be included in the lexical analysis phase of the compiler is described in some detail. The loose ends of it, such as the various mode-declarations, input- and output-routines, are not given; they suggest themselves quite easily from the given texts.

Many routines, like 'go on token', will return true if the symbol(s) suggested by the name of the routine appear next in the input stream, and false otherwise. If the routine returns true, the lexical unit(s) it stands for will be copied to the output stream. As a consequence, a routine like 'pack sequence' will consume a complete pack-sequence, etc.

One of the tasks of the algorithm is to collect information on mode-definitions, operator- (& priority-) definitions and module-definitions. This information is collected in "indicant tables", which are subsequently inspected (and amplified) by the following phases of the compiler. During lexical analysis, minimal information on these defining occurrences is collected:

- for each bold word or operator defined, it is recorded whether it is a mode-indication, module-indication or operator. For operators, the priority is recorded as well. For modes and operators, it is recorded whether they are declared public. Finally, the module-indications in a revelation are recorded together with their publicity.
- for each of these, the range in which they occur is recorded.

The range is not really determined; rather, for each opener or middler a new range is started. The precise structure of the indicant tables, and therefore the body of routines like 'put in mode table', is not given. It is easy to verify that the above information is sufficient to associate the proper defining occurrence with each applied occurrence of an operator, mode- or module-indication during the subsequent phases of the compiler.

The most important entity that must be paid attention to is the pack-sequence. As has been explained in section 2.8, information on the state just prior to the pack-sequence and the type of each of its packs must be gathered. Assuming that some output stream is produced which contains the tokens recognized, this information might as well be incorporated in the output stream also. Since the transducer processes that stream in reverse order, it is convenient to output the information in reverse order as well. This leads to:

```
PROC pack sequence = (STATE state) BOOL:
 IF TYPE p; pack(p)
 THEN treat remaining sequence;
    leave info(p); leave info(state); TRUE
 ELSE FALSE
 FI;
```

```
PROC treat remaining sequence = VOID:
 IF TYPE p; pack(p)
 THEN treat remaining sequence; leave info(p)
 FI;
```

Inside a pack, it is necessary to recognize declarations. In the scheme given below, a pack is viewed as a series of entities, separated by middlers (symbols like "|"), completion-tokens, colon-tokens, go-on-tokens and postlude-tokens, and surrounded by parentheses. Each of these entities then potentially is a declaration, and may be described as a "unit-list or declaration".

A declaration can be further partitioned into COMMON-declarations. This partitioning cannot easily be accomplished during lexical analysis. It is not necessary either, as long as we partition a declaration into pieces separated by and-also-tokens and keep track of some information which determines the type of COMMON-declaration we are concerned with. (In parsing a text "i = int" in a context "mode r = real, i = int", it is important to know that it concerns a mode-declaration.) By partitioning "unit-list or declaration" into smaller entities, separated by and-also-tokens, each of these smaller entities may be considered as a "unit or definition".

The following "types" of a "unit or definition" are distinguished:

i) "mode" , i.e., something of the form "mode ...";
ii) "op" , i.e., something of the form "op ...";
iii) "prio" , i.e., something of the form "prio ...";
iv) "module" , i.e., something of the form "module ...";
v) "decl" , i.e., something of the form "d ...", where "d" has been recognized as a declarer;
vi) "m" , i.e., something of the form "m ...", where "m" is a bold word whose type is yet unknown. Eventually, this will reduce to case v) or vii);
vii) "rest", for all other cases, i.e., it concerns a unit.

The information which is transported from one "unit or definition" to the next can be viewed as the status in which we are going to parse it. It is manipulated as follows:

1) At the start of a pack, and at a middler, completion-token, colon-token, go-on-token or postlude-token, it is set to "rest";
2) At an and-also-token, the status is updated if the "unit or definition" just treated has a type which is not "rest". (This is not surprising since we have to parse "i = z" in a status "decl" in the context "real a, b, i = z", though the type of the entity just treated ("b") is "rest".)

We now arrive at the following (in which, for the sake of legibility, an obvious extension of the ALGOL 68 case-clause is used):

```
PROC pack = (REF TYPE p) BOOL:
 IF parallel token THEN p:= "par"; TRUE
 ELIF opener
 THEN BOOL no decl pack:= FALSE;
   UDTYPE status:= rest, type;
        # UDTYPE stands for "unit or definition type" #
   WHILE
     WHILE unit or definition(status, type);
       IF and also token
       THEN (type ≠ "rest" | status:= type); TRUE
       ELSE public:= FALSE;
         mode token ahead OR operator token ahead OR
         priority token ahead OR module token ahead
       FI
     DO SKIP OD;
     IF middler OR completion token OR colon token OR go on token
                OR postlude token
     THEN status:= rest; no decl pack:= TRUE
     ELSE FALSE
     FI
   DO SKIP OD;
   p:= CASE closer IN
     "["           : "subbus",
     "'secca"      : "revel",
     "fed"         : "deffed",
     ")"           : IF no decl pack THEN "brief pack"
                     ELIF (type ≠ "rest" | status:= type);
                       status = "decl"
                     THEN "formals"
                     ELIF is bold word(status) THEN status
                     ELSE "brief pack"
                     FI
    OUT "bold pack"
    ESAC;
   TRUE
 ELSE FALSE
 FI;
```

As has been mentioned before, we must keep track of the openers, middlers and closers, in order to associate the various defining occurrences of mode- and module-indications and operators with the range in which they occur. To this purpose, it is convenient to maintain a stack of "open" ranges. With each entry of this stack, a list of definitions in the corresponding range is associated. At each opener or middler, a new elemant is pushed on the stack with an initially empty list. For each defining occurrence of a mode- or module-indication or operator, the list associated with the topmost element of the stack is updated. This is taken care of by the routines 'put in mode table', 'put in priority table', 'put in operator table' and 'put in module table'. At each middler or closer, one or more elements are popped off the stack.

The number of elements popped off depends on the contruct at hand: in a context like

IF ... THEN ... ELIF ...THEN ... ELSE ... FI,

three elements must be popped off when the symbol "fi" is encountered. This number can easily be determined if for each entry the middler or opener that the corresponding range started with is maintained as well. Then elements can be popped off up to and including the first opener encountered. Obviously, the corresponding lists must somehow be saved for later use.

This stack manipulation can be taken care of by the routines ´opener´, ´middler´ and ´closer´. The routine ´opener´ can also take care of the "ssecca insert": if the topmost element of the stack conforms to a revelation and the next input symbol is an opener, then a ssecca-insert is "inserted" in the input stream (and the routine returns false). In a similar way, it can take care of the loop-insert. The routines ´middler´ and ´closer´ can deal with the transformation of choice-symbols, as described in section 2.1 above. These routines largely have a clerical task; their bodies will not be given here.

As can be seen from the text of the routine ´pack´ above, the symbols "mode", "op", "prio" and "module" are intercepted at a high level. These symbols are considered extremely important; even in an erroneous input text, they will be treated as starting symbols of a declaration.

The next important routine is ´unit or definition´. Its main task is to have a close look at the first part of such an entity; the remaining part is only very globally analyzed. The first part determines whether it concerns a (potential) definition. To this end, the following patterns are recognized:

i) "mode m = ..."
ii) "prio + = ..."
iii) "op + = ..."
iv) "module m = ..."
v) "loc m a = ..." or "´ldec m a = ..."
vi) "real a = ...", "real: ...", "real(...)",
"(...)m a = ...", "(...)m ...", "(...)m(...)"
vii) "m = ...", when the status is "mode", "op", "prio" or "module",
"m a = ...", "m: ...", "m(...)"
viii) "a = ..." when the status is "decl" or "m",
"a" {since it may concern a call or slice}
ix) "egg "a" = ..."

We thus arrive at the following:

```
PROC unit or definition = (UDTYPE status, REF UDTYPE type) VOID:
 BEGIN
  PROC idat = BOOL:
    (input symbol = "=" | skip symbol; leave(is defined as token); TRUE
                        | FALSE);

  PROC idat cond = (TYPE m) BOOL:
    (input symbol = "=" | skip symbol; leave(is defined as token);
                          leave(m); TRUE
                        | FALSE);

  PROC mode definition = VOID:
    IF type:= "mode"; bold(m)
    THEN put in mode table(m, public);
      (idat | state:= "decpref")
    FI;

  PROC priority definition = VOID:
    IF type:= "prio"; operator(m)
    THEN put in priority table(m, public, (idat | priority unit | 1))
    FI;

  PROC operation definition = VOID:
    IF type:= "op"; operator displayety(m)
    THEN put in operator table(m, public); idat
    FI;

  PROC module definition = VOID:
    IF type:= "module"; bold(m)
    THEN put in module table(m, public);
      (idat | pack sequence("modtext"))
    FI;

  PROC tag equals = (TYPE m) BOOL:
    IF tag ahead
    THEN leave(dectag insert); leave(m); tag;
      (idat cond(m) | SKIP | state:= "cliceable");
      type:= m; TRUE
    ELSE FALSE
    FI;

  STATE state:= "rest",  # initial state for pack sequence #
  type:= "rest";

  IF public token THEN public:= TRUE;
  BOOL ldec = ldec token;

  IF mode token THEN mode definition
  ELIF priority token THEN priority definition
  ELIF operator token
```

```
  THEN
    (NOT operator displayety ahead | pack sequence("decpref"); declarer);
    leave(opdec insert); operation definition
  ELIF module token THEN module definition
  ELIF leap token OR ldec
  THEN declarer;
    IF tag ahead
    THEN leave(dectag insert); tag; idat; type:= decl
    FI
  ELIF
    IF visible declarer
    THEN m:= "decl"; TRUE
    ELIF pack sequence("rest")
    THEN (visible declarer or bold(m) | SKIP | m:= "rest"); TRUE
    ELSE FALSE
    FI
  THEN
    IF tag equals(m) OR routine token(m) OR pack sequence(m)
    THEN SKIP
    ELSE type:= m
    FI
  ELIF bold ahead OR operator displayety ahead
  THEN
    IF status = "mode" THEN mode definition
    ELIF status = "prio" THEN priority definition
    ELIF status = "op" THEN operation definition
    ELIF status = "module" THEN module definition
    ELIF NOT bold(m)
    THEN SKIP
    ELIF tag equals(m) OR routine token(m) OR pack sequence(m)
         OR in revelation(public, m)
            # 'in revelation' just inspects the previously mentioned
              stack of open ranges. If the topmost element corresponds to
              a revelation, the module indication m is recorded together
              with its publicity #
    THEN SKIP
    ELSE type:= m
    FI
  ELIF tag
  THEN
    IF ((status = "decl" | TRUE | bold(status:= m))
       | idat cond(status) | FALSE )
    THEN type:= status
    ELSE type:= "rest"; state:= "cliceable"
    FI
  ELIF egg token
  THEN denoter( SKIP ); egg defined as token
  FI;
  WHILE junk(state) DO state:= "rest" OD
END # unit or definition #;
```

The routine 'junk' treats the remaining part of a 'unit or definition'. It has to watch for pack-sequences, calls or slices, and declarers; the remaining symbols are just copied to the output stream.

```
PROC junk = (STATE state) BOOL:
 IF pack sequence(state) THEN TRUE
 ELIF DENTYPE t; denoter(t)
 THEN (t = "string" OR t = "char" | pack sequence(cliceable)); TRUE
 ELIF tag THEN pack sequence(cliceable); TRUE
 ELIF STATE m; visible declarer or bold(m)
 THEN (pack sequence(m) | SKIP | routine token(m)); TRUE
 ELIF leap token THEN declarer; TRUE
 ELIF operator THEN TRUE
       # Note that this will be a non-bold operator #
 ELIF format text THEN TRUE
 ELIF becomes token THEN TRUE
 ELIF at token THEN TRUE
 ELIF identity relator THEN TRUE
 ELIF nil token THEN TRUE
 ELIF skip token THEN TRUE
 ELIF of token THEN TRUE
 ELIF go to token THEN TRUE
 ELIF code token THEN TRUE
 ELIF formal nest token THEN TRUE
 ELSE FALSE
 FI;
```

Most of the remaining routines are of no interest to the algorithm under discussion. If we assume that a routine 'visible declarer' exists which is able to cope with declarers that start with one of the symbols ref, proc, flex, union, struct or 'l, or consist of a single mode standard, then the two remaining routines which affect the algorithm are 'declarer' and 'visible declarer or bold':

```
PROC declarer = VOID:
 BEGIN pack sequence(decpref);
       visible declarer or bold(LOC STATE)
 END;

PROC visible declarer or bold = (REF STATE m) BOOL:
 (visible declarer | m:= "decl"; TRUE | bold(m));
```

REFERENCES

[1] VAN WIJNGAARDEN, A. et al, Revised Report on the Algorithmic Language ALGOL 68, Acta Informatica 5 (1975), pp. 1-236.

[2] LINDSEY, C.H. & H.J. BOOM, A modules and separate compilation facility for ALGOL 68, ALGOL Bulletin 43 (1978), pp. 19-53.

[3] 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.

[4] AHO, A.V. & J.D. ULLMAN, The Theory of Parsing, Translation and Compiling, Vol I: Parsing, Prentice-Hall, 1972.

[5] LEWIS II, P.M. & R.E. STEARNS, Syntax-directed transduction, JACM 15, 3 (1968), pp. 465-488.

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

[7] MEERTENS, L.G.L.T. & J.C. VAN VLIET, On top-down parsing of ALGOL 68+, Mathematical Centre, Amsterdam, to appear.

[8] VAN VLIET, J.C. The programs "Relations concerning a cf-grammar" and "LL(1)-checker", Report IN 4/73, Mathematical Centre, Amsterdam, 1974.

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

[10] 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).