stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 171/81 JULI

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

AN UNDERLYING CONTEXT-FREE GRAMMAR OF ALGOL 68+

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: 68F05, 68F25, 68B20

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

An underlying context-free grammar of ALGOL 68+

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. In the defining documents, ALGOL 68 is defined using a two-level grammar. This type of grammar is not well suited for standard parsing techniques. The present report describes the construction of an underlying context-free grammar for ALGOL 68+. The differences introduced by this transition are discussed. Finally, the resulting context-free grammar is given.

KEY WORDS & PHRASES: ALGOL 68, context-free grammar, syntax-analysis

1. INTRODUCTION

ALGOL 68+ is a superlanguage of ALGOL 68, which is obtained by combining three documents:

i) The Revised Report on the Algorithmic Language ALGOL 68 [1],
ii) The official IFIP modules and separate compilation facility [2], and
iii) A number of changes and additions to [1] and [2] in order to be able to process a version of the standard-prelude. These are described in [3].

ALGOL 68+ is described in these three documents by means of a two-level Van Wijngaarden grammar, in which the rules as they are presented (the "hyper-rules") may contain "metanotions", which yet have to be replaced by one of their terminal metaproductions. In this way, from a single given rule an infinity of new rules may be derived, corresponding to the infinite number of terminal metaproductions for some metanotions, such as, e.g., "MOID". A typical rule where this mechanism is applied, is rule 4.6.1.u from [1]:

MOID NEST joined declarer: formal MOID NEST declarer.

Because of this feature, Van Wijngaarden grammars are essentially more powerful than context-free grammars. The relationship between the "underlying" context-free grammar developed here, and ALGOL 68+, is that the set of ALGOL-68+-particular-programs is a strict subset of the set of programs generated by this grammar.

In section 2 below, the method is discussed by which the context-free grammar is obtained from the two-level grammar used in the defining documents [1, 2, 3]. The differences introduced through this transition are discussed in section 3. Where appropriate, hints are given as to where during the parsing process these differences can be dealt with in a reasonable way. Finally, section 4 contains the context-free grammar in a self-explanatory format. Though this grammar has been developed with a very specific (top-down) parsing method in mind, it is believed to be general enough in order to be useful as a starting point for a wide variety of parsing algorithms.

2. CONSTRUCTING AN UNDERLYING CONTEXT-FREE GRAMMAR

Before the "removal" of the metanotions from the syntax of ALGOL 68+ was undertaken, a slightly amended version of the grammar of ALGOL 68+ has been constructed. The notions ´tag token´, ´bold tag token´, ´sizety standard token´, ´other tao token´, ´denoter´ and ´format text´, for which a production rule is given were considered primitive symbols, the reason being that the former five are expected to be treated at the lexical level rather than the syntax-analysis level, while the latter has an (essentially context-free) syntax rather different from the remainder of ALGOL 68+ and had consequently better be analyzed by a completely separate routine.

The remaining primitive (i.e., terminal) symbols largely correspond to the symbols listed in section 9.4.1 of the Revised Report. This means that various representations of the same symbol have been contracted, like goto and go to, or @ and at. On the other hand, if one and the same mark is used to denote different symbols (which is the case for ":", "=", "(", ")", "~" and "|"), the mark is given as terminal, while production rules for the symbols are given. Note also that 'stick colon mark' (whose representation is "|:") is given as terminal, while production rules are given for the brief-else-if-token and brief-ouse-token. Also, pragmats and comments are not taken into account.

In short, the terminal symbols of the context-free grammar are the units that are likely to result from lexical analysis (see, e.g., [4]). Section 4 starts with a list of these terminal symbols; they are separated by semicolons, and the list is terminated with a period.

In order to get rid of the metanotions, several strategies were applied. Some of the metanotions serve no other purpose than mere shortening of the syntax. In such cases, it is possible to write out the rule for the several terminal productions of the metanotion. In most of these cases, however, a new notion has been introduced, as in the case of

```
enclosed clause:
    closed clause; collateral clause; parallel clause;
    choice clause; loop clause; access clause.
```

The new notion 'enclosed clause' replaces, e.g., a hypernotion 'SORT MOID ENCLOSED clause'.

Some metanotions, such as 'ADIC', exist only for semantical reasons. These, and their terminal productions, were simply struck out. The same fate befell the metanotions concerned with nests, modes and coercions, such as 'NEST', 'MOID' and 'SORT', and their (possibly partial) terminal productions. Thus, a rule like

```
REF to MODE NEST assignation:
    REF to MODE NEST destination, becomes token, MODE NEST source.
```

was reduced by this process to

```
assignation: destination, becomes token, source.
```

A special role is played by the metanotion 'NOTION'. There is no one simple means by which the rules containing 'NOTION' can be brought into context-free form. For notions of the form 'NOTION list', e.g., the production rule has been written out in most of the cases. In order to treat notions of the form 'NOTION option', the syntactical description mechanism has been enriched with a new construct: a list of notions enclosed between the syntactic marks "(" and ")" may be used as one member in another list of notions. The direct productions of such a

member are: empty, and the enclosed list of notions. It goes without saying that this does not enlarge the descriptive power of the syntactical description mechanism, but is merely an expedient way to shorten the syntax.

A peculiar application of 'NOTION' is found in the so-called "predicates". Predicates are used in the syntax to enforce certain restrictions, such as that each applied-indicator should identify a uniquely determined defining-indicator. When applied thus, they serve to describe syntactically what usually is considered "static semantics". A more modest use is to reduce the number of rules by grouping similar cases as alternatives in one rule (as in the example given below). A predicate either holds, in which case it produces an empty string, or it does not hold, in which case each attempt to produce something runs into a blind alley.

All occurrences of predicates in the syntax have been removed, i.e., they are treated as if they always hold. Thus a rule like

```
MODE NEST source for MODINE:
    where (MODINE) is (MODE), MODE NEST source;
    where (MODINE) is (routine), MODE NEST routine text.
```

is reduced to

```
MODE NEST source for MODINE:
    MODE NEST source;
    MODE NEST routine text.
```

As a result, the second alternative can now be removed, since 'source' already produces 'routine text'. Because of the above simplification, e.g., "proc p = q" is accepted as a valid identity-definition.

As a last step, many shortcuts have been made where notions were introduced in the ALGOL 68 syntax in behalf of the semantics. Thus, the three rules

```
assignation: destination, becomes token, source.
destination: tertiary.
source: unit.
```

were combined into one rule:

```
assignation: tertiary, becomes token, unit.
```

3. DIFFERENCES CAUSED BY THE TRANSITION TO A CONTEXT-FREE GRAMMAR

It is not simple to characterize the programs that are generated by the given context-free grammar, but are not ALGOL-68+-particular-programs. Informally, they may be termed "programs with mode errors". This is not surprising: many of the metanotions which have been removed serve to pass mode information from one part of a production to another. On the other hand, this terminology might lead on astray. Typical examples of programs that are accepted might look as follows:

```
begin ref int i = true; if (1, 2) then skip:= j fi end;

begin real z:= to 3 do newline od; 1 + par (2, 3) end;

if v of "a" then (1 :=: goto m)[loc union (int)] fi.
```

It is possible, at the cost of considerable complications, to refine the grammar in such a way that many of the possible "errors" are prevented. From a more semantically oriented point of view, however, it is not at all clear that this would be advantageous. There seems to be no reason why, at this level, a condition like (1, 2) should be excluded when [] int r = (1, 2); r will still be generated. Moreover, many of these "errors" can often quite easily be detected in a different way during either syntax analysis (by letting the metanotion act as an "attribute", or "affix", to the production rule) or semantic analysis, where the more difficult cases have to be dealt with anyway. A list of the differences and some suggestions as to the stage of the parsing process where they can be dealt with in a reasonable way is given below.

- The metanotions SORT, MOID and NEST have been removed. Thus, a rule like

  ```
  MODE NEST source: strong MODE NEST unit.
  ```

  is implemented as

  ```
  MOID1 NEST1 source: SORT MOID2 NEST2 unit.
  ```

  Examples of accepted constructs are:

  ```
  real (true);
  void x = empty;
  s of do skip od.
  ```

  The checks will mainly have to be performed in the coercion and identification phases, which are part of semantic analysis. Part of the related-test (cf section 7.1.1 of the Revised Report [2]), as for real x; int x, can already be done in the mode-independent pass of the compiler.

- The predicates are always treated as if they always hold, except for the general predicates defined in section 1.3.1 of the Revised Report (cf [2]). For a list of these, see section 12.3 of the Revised Report; this list must be extended with the predicates 'revealed by', 'filters ... out of', 'absent from' and 'collected properties from', which are used in the part on modules and separate compilation (cf [3]), and the predicate 'descriptors from' used in one of the extensions from [4]. If there is anything to test here, this will mainly fall under the heading of well-formedness and mode-equivalencing (i.e., these tests have to be performed during semantic analysis).

- It is in general not required any more that parentheses match in style. Thus,

  begin real x; get(x); put(x)), and

  if a in b)

  are accepted as valid enclosed-clauses. The correctness of the parenthesis skeleton can easily be tested during lexical analysis, or during syntax analysis by using an attribute "style". Note, however, that the parentheses of the indexer-bracket of a slice are required to match (except that it saves two production rules, there is no deep reason for this).

- In a series, declarations will be accepted where units are allowed. So declarations are allowed after a label-definition has been encountered (though they are not allowed just before a completer, or as the last unit of the series). The mode-independent (syntax-analysis) pass can easily test for this by means of a flag "label definition encountered".

- An enquiry-clause is treated as a series, so label-definitions are accepted. Again, a flag "label definitions permitted" (true for a series, false for an enquiry-clause) solves this case.

- In choice-clauses, the CHOICE is ignored, so that for instance the following is accepted:

  if b then 1, 2 fi.

  This must be treated in the mode-dependent pass (i.e., during semantic analysis). Note also that a single unit is allowed as the in-part of a choice-using-integral-clause (where a proper list of units is actually required). Since a single unit is (and should be) allowed as the in-part of a choice-using-boolean-clause also, the production rule for in-part-of-choice is ambiguous in the present grammar.

- In case-clauses, units and united-case-parts are accepted intermixed, as in

```
case z in 1, (real x): 2 esac.
```

  This can probably best be catered for in the mode-dependent pass, though a solution for the mode-independent pass is not difficult.

- In COMMON-declarations, 'ldec and pub are always accepted. It is not necessary to test for 'ldec; only the standard-prelude author can and may use it, and it is his responsibility to use this facility in a proper way. The acceptability of pub can easily be tested in the mode-independent pass. A flag "pub allowed" is true for the series of a module-text, and false otherwise.

- 0 (zero) is also accepted as priority-unit (unless digit-token does not produce 0). This should be treated during lexical analysis. Suggestion: accept all integral-denoters, but issue an error message in case the number of digits exceeds 1 or the numerical value equals 0 (and take some reasonable value instead).

- In identifier- and operation-declarations, no link is made between a missing procedure-plan and the source being a routine-text. So, the following is accepted:

```
proc p = q;
proc p;
proc p:= q;
op + = q.
```

  This can easily be tested in the mode-independent pass; it is already necessary to determine the tentative mode in order to fill the indicant tables.

- Only in ldec-identity- and ldec-operation-declarations the source is allowed to be of the form

```
choice token, ldec source choice list brief pack.
```

  This need not be tested. The much too weak syntax of ldec-sources is harmless as well.

- An arbitrary number of parameters is accepted in the plan or routine-text of an operation-declaration. So the following is accepted:

```
op real + = 1;
op ? = (bool b, int x, y) int: (b | x | y).
```

  This can easily be tested in the mode-independent pass.

- VICTAL is ignored, and the bounds in a row-rower are also uncoupled, so that the following is accepted:

```
ref [1:3] real;
mode vec = [] real;
proc flex [] char;
[1:, :2] real.
```

  Also here, a solution is not difficult: a variable "victal", with possible values "virtual", "actual", "formal", "unknown", and "clash". These values are mostly inherited, but for identifier-declarations they are derived if loc or heap is missing. The value virtual occurs only inherited (after ref) and is used to allow for flex.

- For simplicity's sake, the priorities of operators are ignored in formulas. This leads to a highly ambiguous syntax. A solution for the mode-independent pass is straightforward (given that some information on the priorities of operators has been collected during lexical analysis).

4. AN UNDERLYING CONTEXT-FREE GRAMMAR OF ALGOL 68+

In the following context-free grammar, comments are placed between style-ii-comment-symbols ("#"). Optional members are placed between the symbols "(" and ")"; see also section 2 above. The grammar starts with a list of the terminal symbols, separated by semicolons and terminated with a period. Production rules consist of a left-hand-side and a colon followed by one or more alternatives, separated by semicolons. The last alternative is followed by a period. An alternative consists of one or more members, separated by commas. A member is either an optional list of notions, or a notion.

In the comments in the grammar below, the numbers refer to the corresponding section numbers in [1], as possibly modified by [2]. The representations of the various bold symbols are given in the point-stropping regime (cf. [5]).

The rules of the grammar which (partly) refer to pieces of syntax from [2] and [3] are marked in the margin by straight and wavy lines, respectively. In this way, a context-free grammar for ALGOL-68-proper can easily be extracted.

```
# terminal symbols #

#9. tokens and symbols.#

#9.4.1.c. operator symbols.#

    becomes token;                          # := #

#9.4.1.d. declaration symbols.#

    reference to token;                     # .ref #
    local token;                            # .loc #
    heap token;                             # .heap #
    structure token;                        # .struct #
    flexible token;                         # .flex #
    procedure token;                        # .proc #
    union of token;                         # .union #
    operator token;                         # .op #
    priority token;                         # .prio #
    mode token;                             # .mode #

#9.4.1.f. syntactic symbols.#

    bold begin token;                       # .begin #
    bold end token;                         # .end #
    and also token;                         # , #
    go on token;                            # ; #
    completion token;                       # .exit #
    parallel token;                         # .par #
    bold if token;                          # .if #
    bold then token;                        # .then #
    bold else if token;                     # .elif #
    bold else token;                        # .else #
    bold fi token;                          # .fi #
    bold case token;                        # .case #
    bold in token;                          # .in #
    bold ouse token;                        # .ouse #
    bold out token;                         # .out #
    bold esac token;                        # .esac #
    brief sub token;                        # [ #
    brief bus token;                        # ] #
    at token;                               # @  .at #
    is token;                               # :=:  .is #
    is not token;                           # :≠:  :/=:  .isnt #
    nil token;                              # ∘  .nil #
    of token;                               # .of #
    go to token;                            # .go .to  .goto #
```

```
#9.4.1.g. loop symbols.#

    for token;                          # .for #
    from token;                         # .from #
    by token;                           # .by #
    to token;                           # .to #
    while token;                        # .while #
    do token;                           # .do #
    od token;                           # .od #

#not defined#

    equals mark;                        # = #
    open mark;                          # ( #
    close mark;                         # ) #
    stick mark;                         # | #
    stick colon mark;                   # |: #
    colon mark;                         # : #
    bold skip mark;                     # .skip #
    tilde mark;                         # ~ #
    digit token;                        # like, e.g., 1 #
    tag token;                          # like, e.g., i #
    bold tag token;                     # like, e.g., .m #
    sizety standard token;              # like, e.g., .int #
    other tao token;                    # like, e.g., + #
    format text;                        # like, e.g., $3zd$ #
    denoter;                            # like, e.g., 3.14 #

#a68+ symbols#

    code token;                         # .'code #
    ldec token;                         # .'ldec #
    choice token;                       # .'choice #

#sep comp symbols#

    module token;                       # .module #
    access token;                       # .access #
    def token;                          # .def #
    fed token;                          # .fed #
    public token;                       # .pub #
    postlude token;                     # .postlude #
    formal nest token;                  # .nest #
    egg token.                          # .egg #
```

```
# production rules #

#9.4.1. representations of symbols.#

    is defined as token:            equals mark.

    open token:                     open mark.
    close token:                    close mark.

    brief begin token:              open mark.
    brief end token:                close mark.

    brief if token:                 open mark.
    brief then token:               stick mark.
    brief else if token:            stick colon mark.
    brief else token:               stick mark.
    brief fi token:                 close mark.

    brief case token:               open mark.
    brief in token:                 stick mark.
    brief ouse token:               stick colon mark.
    brief out token:                stick mark.
    brief esac token:               close mark.

    style i sub token:              open mark.
    style i bus token:              close mark.

    label token:                    colon mark.
    colon token:                    colon mark.
    up to token:                    colon mark.
    routine token:                  colon mark.

    skip token:                     bold skip mark; tilde mark.


#4.8. indicators and field selectors.#

identifier:
    tag token.
mode indication:
    bold tag token; sizety standard token.
operator:
    bold tag token; equals mark; tilde mark; other tao token.
field selector:
    tag token.
module indication:
    bold tag token.
hole indication:
    denoter.
```

```
#9.1.1. choice clause tokens.#

choice start:
    bold if token; brief if token; bold case token; brief case token.
choice in:
    bold then token; brief then token; bold in token; brief in token.
choice again:
    bold else if token; brief else if token;
    bold ouse token; brief ouse token.
choice out:
    bold else token; brief else token; bold out token; brief out token.
choice finish:
    bold fi token; brief fi token; bold esac token; brief esac token.


#10.7.1. compilation inputs.#

compilation input:
    #language# stuffing packet;
    definition module packet;
    particular program;
    prelude packet.

#10.6.1. packets.#

#algol 68# stuffing packet:
    egg token, hole indication, is defined as token, actual hole.

definition module packet:
    egg token, hole indication, is defined as token,
        module declaration.

particular program:
    label definition, particular program;
    enclosed clause.

prelude packet:
    module declaration.


#3. clauses.#

enclosed clause:
    closed clause; collateral clause; parallel clause; choice clause;
    loop clause; access clause.
```

#3.1. closed clauses.#

```
closed clause:
    begin, serial clause, end.
begin:
    bold begin token; brief begin token.
end:
    bold end token; brief end token.
```

#3.2. serial clauses.#

```
serial clause:
    series.
series:
    unit, go on token, series;
    declaration, go on token, series;
    label definition, series;
    unit, completion token, label definition, series;
    unit.
label definition:
    identifier, label token.
```

#3.3. collateral and parallel clauses.#

```
collateral clause:
    begin, (joined portrait), end.
joined portrait:
    unit, and also token, unit or joined portrait.
unit or joined portrait:
    unit; joined portrait.

parallel clause:
    parallel token, begin, joined portrait, end.
```

#3.4. choice clauses.#

```
choice clause:
    choice start, chooser choice clause, choice finish.

chooser choice clause:
    enquiry clause, alternate choice clause.
enquiry clause:
    series.

alternate choice clause:
    in choice clause, (out choice clause).
```

```
in choice clause:
    choice in, in part of choice.
in part of choice:
    serial clause; case part list.
case part list:
    case part, (and also token, case part list).
case part:
    (specification), unit.
specification:
    brief begin token, declarer, (identifier),
        brief end token, colon token.

out choice clause:
    choice out, serial clause;
    choice again, chooser choice clause.

#3.5. loop clauses.#

loop clause:
    for part, (from part), (by part), (to part), repeating part.
for part:
    (for token, identifier).
from part:
    from token, unit.
by part:
    by token, unit.
to part:
    to token, unit.
repeating part:
    (while part), do part.
while part:
    while token, enquiry clause.
do part:
    do token, serial clause, od token.

#3.6. access clauses.#

access clause:
    revelation, enclosed clause.
revelation:
    access token, joined module call.
joined module call:
    module call, (and also token, joined module call).
module call:
    (public token), module indication.
```

#4. declarations.#

declaration:
    (public token), ldecety common declaration,
        (and also token, declaration).
ldecety common declaration:
    (ldec token), common declaration.
common declaration:
    mode declaration; priority declaration; identity declaration;
    variable declaration; operation declaration;
    module declaration.

#4.2. mode declarations. #

mode declaration:
    mode token, mode joined definition.
mode joined definition:
    mode definition, (and also token, mode joined definition).
mode definition:
    mode indication, is defined as token, declarer or code.
declarer or code:
    declarer;
    code.

#4.3. priority declarations.#

priority declaration:
    priority token, priority joined definition.
priority joined definition:
    priority definition, (and also token, priority joined definition).
priority definition:
    operator, is defined as token, digit token.

#4.4. identifier declarations.#

identity declaration:
    modine declarer, identity joined definition.
modine declarer:
    declarer; procedure token.
identity joined definition:
    identity definition,
        (and also token, identity joined definition).
identity definition:
    identifier, is defined as token, ldecety source.

```
ldecety source:
    unit or code;
    choice token,
        brief begin token, ldec source choice list, brief end token.
unit or code:
    unit;
    code.
code:
    code token, denoter.

ldec source choice list:
    ldec source choice,
        (and also token, ldec source choice list).
ldec source choice:
    relational, length denoter, colon token, unit or code.
relational:
    operator.
length denoter:
    (other tao token), denoter.

variable declaration:
    (leap token), modine declarer, variable joined definition.
variable joined definition:
    variable definition,
        (and also token, variable joined definition).
variable definition:
    identifier, (becomes token, unit).

#4.5. operation declarations.#

operation declaration:
    operator token, (formal procedure plan),
        operation joined definition.
operation joined definition:
    operation definition,
        (and also token, operation joined definition).
operation definition:
    operator displayety, is defined as token, ldecety source.
operator displayety:
    operator;
    choice token,
        brief begin token, operator list, brief end token.
operator list:
    operator, (and also token, operator list).
```

#4.9. module declarations.#

```
module declaration:
    module token, module joined definition.
module joined definition:
    module definition, (and also token, module joined definition).
module definition:
    module indication, is defined as token, module text.
module text:
    (revelation), def token, module series, fed token.
module series:
    module prelude, (module postlude).
module prelude:
    unit, (go on token, module prelude);
    declaration, (go on token, module prelude).

module postlude:
    postlude token, postlude series.
postlude series:
    unit, (go on token, postlude series).
```

#4.6. declarers.#

```
declarer:
    reference to declarator; structured with declarator;
    flexible rows of declarator; rows of declarator;
    procedure declarator; union of declarator; mode indication.

reference to declarator:
    reference to token, declarer.

structured with declarator:
    structure token, portrayer pack.
portrayer pack:
    brief begin token, portrayer, brief end token.
portrayer:
    declarer, joined definition of fields, (and also token, portrayer).
joined definition of fields:
    field selector, (and also token, joined definition of fields).

flexible rows of declarator:
    flexible token, declarer.
```

```
rows of declarator:
    rower bracket, declarer.
rower bracket:
    brief sub token, rower, brief bus token;
        style i sub token, rower, style i bus token.
rower:
    row rower, (and also token, rower).
row rower:
    unit, up to token, unit;
    unit;
    (up to token).

procedure declarator:
    procedure token, formal procedure plan.
formal procedure plan:
    (joined declarer pack), declarer.
joined declarer pack:
    brief begin token, joined declarer, brief end token.
joined declarer:
    declarer, (and also token, joined declarer).

union of declarator:
    union of token, joined declarer pack.



#5. units.#

unit:
    assignation; identity relation; routine text; jump; skip token;
    formal hole; tertiary.
tertiary:
    formula; nil token; secondary.
secondary:
    leap generator; selection; primary.
primary:
    slice; call; cast; denoter; format text; identifier;
    enclosed clause.

#5.2.1. assignations.#

assignation:
    tertiary, becomes token, unit.

#5.2.2. identity relations.#

identity relation:
    tertiary, identity relator, tertiary.
identity relator:
    is token; is not token.
```

```
#5.2.3. generators.#

leap generator:
    leap token, declarer.
leap token:
    local token; heap token.

#5.3.1. selections.#

selection:
    field selector, of token, secondary.

#5.3.2. slices.#

slice:
    primary, indexer bracket.
indexer bracket:
    brief sub token, indexer, brief bus token;
    style i sub token, indexer, style i bus token.
indexer:
    trimscript, (and also token, indexer).
trimscript:
    unit;
    (unit), up to token, (unit), (revised lower bound);
    (revised lower bound).
revised lower bound:
    at token, unit.

#5.4.1. routine texts.#

routine text:
    (declarative pack), declarer, routine token, unit.
declarative pack:
    brief begin token, declarative, brief end token.
declarative:
    declarer, parameter joined definition, (and also token,declarative).
parameter joined definition:
    identifier, (and also token, parameter joined definition).

#5.4.2. formulas.#

formula:
    (operand), operator, operand.
operand:
    formula; secondary.
```

#5.4.3. calls.#

call:
    primary, actual parameters pack.
actual parameters pack:
    brief begin token, actual parameters, brief end token.
actual parameters:
    unit, (and also token, actual parameters).

#5.4.4. jumps.#

jump:
    (go to token), identifier.

#5.5.1. casts.#

cast:
    declarer, enclosed clause.

#5.6. holes.#

formal hole:
    formal nest token, language indication, hole indication.
actual hole:
    enclosed clause.
language indication:
    (bold tag token).

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] JONKERS, H.B.M., A finite state lexical analyzer for the standard hardware representation of ALGOL 68, ALGOL Bulletin 44 (1979), pp 16-51.

[5] BOOM, H.J. & W.J. HANSEN, The report on the standard hardware representation for ALGOL 68, SIGPLAN Notices 12, no 5 (1977), pp 80-87.