stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 182/81 NOVEMBER

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

ON TOP-DOWN PARSING 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: 68F25, 68B20

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

On top-down parsing of ALGOL 68+

by

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

ABSTRACT

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. ALGOL 68+ is a superlanguage of ALGOL 68 which is powerful enough to describe the standard-prelude. An operator-precedence grammar for ALGOL 68+ 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. Both the algorithm for the transduction scheme and the subsequent top-down syntax analysis are discussed, with emphasis on the treatment of incorrect input texts. The LL(1) grammar for ALGOL 68+ is given as well.

KEY WORDS & PHRASES: ALGOL 68+, syntax-directed transduction, top-down parsing, error-recovery

1. INTRODUCTION

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 [1].

ALGOL 68+ is a superlanguage of ALGOL 68 [2] 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 [3]. 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 [4].

A context-free grammar underlying the ALGOL 68+ syntax, such as the one given in [5], is not of type LL(1), but it seems possible to construct an LL(1) grammar for "context-free ALGOL 68+". However, in doing this, the original syntactic structure is lost.

Another possibility is to apply beforehand a simple transduction scheme [6], operating from right to left, which brings the source text in prefix form. For example, the assignation

```
a:= b
```

may be transformed into

```
:= a b .
```

It is now possible to decide on the first character that we are concerned with an assignation. In order to apply this method, the parenthesis skeleton should be correct, for, if this 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 [7]), or decide to abort the parsing process altogether.

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 [8] or [1].) For an operator-precedence grammar, it is possible to construct a transducer which brings the source texts in prefix form, only knowing the precedence relations between the symbols. (It is a straightforward variant of the operator-precedence parsing algorithm given in [9], pp. 170-171.)

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+ has been given in [10].

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 [10]), 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. A complete list of the symbols from this category can be found in [10].
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, [10] 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 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. In [11], an algorithm is derived which transforms ALGOL 68+ texts into sentences produced by the operator-priority grammar. Most of the changes can be taken care of during lexical analysis. Some, however, require knowledge of the types of the various bold words defined in the program, and can therefore not be made until the input routine of the subsequent phase.

In fact, the right-to-left transduction results in a linearized parse tree. Obviously, the same parse tree would have resulted from the standard operator-precedence parsing algorithm. This algorithm, however,

offers rather poor possibilities to handle incorrect input texts. Care has been taken to ensure that, when the right-to-left transduction is applied to the grammar, the resulting grammar is of type LL(1).

In the present report, the algorithm for the syntax-directed transduction is given, and the subsequent top-down syntax analysis is discussed. The emphasis is on the treatment of incorrect input texts. A further discussion of the syntax-directed error-recovery method employed is contained in [12, 13]. The LL(1) grammar which results when the transduction is applied to the operator-priority grammar [10], is given in the Appendix.

2. TOWARDS AN LL(1)-GRAMMAR

Suppose a production rule of the operator-priority grammar contains terminal symbols a_1, ... , a_n, in this order. For a production rule with $n = 1$, the one terminal symbol it contains will be called an "operator". So, in the production rule

assignation: destination, becomes token, source.

the becomes-token (":=") is called an operator. For $n \geq 2$, a_1 will be called an "opener", a_i will be called a "middler" for $2 \leq i \leq n-1$, and a_n will be called a "closer". This terminology is not surprising, since such production rules in general describe parenthesized constructs, like begin ... end, or if ... then ... else ... fi.

The main reason for applying the right-to-left transduction is to allow for a subsequent LL(1) parse of the input texts. Obviously, the prefix transduction is pointless for those rules which already start with a terminal symbol. In an operator-priority grammar, this is the case for all rules containing more than one terminal symbol. So the transduction only applies to the "operators". And even the operators need not all be moved. One may choose a minimal subset such that, after applying the prefix transduction to the grammar, the result is of type LL(1). For the grammar given in [10], we have chosen the following subset:

go on token,
dectag insert,
opdec insert,
and also token,
is defined as token,
at token,
colon mark,
specification token,
becomes token,
identity relator,
routine token,
dyadic operator,
of token,

cast insert,
clice insert.

Apart from one extra feature introduced for error-recovery purposes (the "synchronization symbol", see section 3 below), the LL(1) grammar resulting from the operator-priority grammar as given in [10], can be found in the Appendix. Both the right-to-left transduction of the grammar and the test for the resulting grammar being of type LL(1) were performed mechanically. Obviously, a few iterations were needed before the final result was obtained.

3. RESYNCHRONIZATION

The essence of resynchronization is: if the parsing process gets stuck, skip the source text in some way up to a symbol where parsing may be resumed. For this to be fully effective, two things are required: knowledge about which symbols allow resumption of the parsing process, and a guarantee that such a symbol is indeed present. For, if the cause of the derailment of the parser was the omission or mutilation of some symbol from the source text, the remedy of trying to resynchronize on that symbol is, in general, worse than the disease.

Consider a formula A+B. (Such terms as "formula" will be loosely applied to pieces of source text which superficially resemble a proper formula, but which, on closer inspection, may turn out to be incorrect.) If parsing gets stuck in the operand A, we want it to resume at the operator +. For a top-down parser, the knowledge that A is an operand, and, therefore, may be followed by an operator, implies the knowledge that it is to parse a formula at the start of A+B. This information is supplied by bringing the source text in prefix form, so that the source text reads +AB. But now the symbol at which to resynchronize has disappeared from the point of resynchronization. Fortunately, the right-to-left transducer, which picks up the operator to drop it again somewhere to the left, can leave behind, at the point where it picked it up, a token that this is the point at which to resynchronize. For this purpose, we introduce a new "synchronization symbol", or, for short, "synchro", which we denote by "|". Using this, the prefix form of A+B becomes +A|B, and we may observe that the occurrence of an operator, say +, in the source text implies that it has been dropped there by the transducer, so it has been picked up somewhere to the right. Therefore, the parser can be sure of the future presence of a synchro. Likewise, a synchro can only be present if an operator has been picked up at that place, and that operator must have been dropped somewhere to the left. Since the transducer picks up and drops operators on a "last in - first out" basis, the operators and the corresponding synchros can be viewed as properly balanced and nested parentheses.

In the grammar given in the Appendix, these resynchronization symbols have been included at the appropriate places.

4. THE GAP-INSERT

Introducing the synchronization symbol, however, only solves part of the problem. Remember that the transducer picks up an operator to drop it again somewhere to the left. The exact place where the operator is dropped depends on the table of precedence relations. Consider a source text with a piece of garbage, containing (accidentally) only high-priority operators, followed by a low-priority operator. The prefix transducer will then put that low-priority operator in front of the piece of garbage. So the top-down syntax analyser will take a road, based on that operator, and give error messages accordingly. These error messages may puzzle a human interpreter, who does not know why the parser chose that road. Therefore, a virtual "gap-insert" with a relatively low priority is thought to be placed between something which looks, roughly speaking, like the end of a coherent chunk and something which looks like the start of one (in fact, the gap-insert is placed between any two symbols for which originally no precedence relation was defined). After these gap-inserts have played their role of blocking the leftward motion of operators, they are discarded by the transducer again.

A careful decision has to be taken as regards the priority of the gap-insert. This decision is based on the algorithm for the global parsing of the input text, as incorporated in the lexical phase, see [11]. In there, it has been discussed that each parenthesized construct can be viewed as a series of entities, separated by middlers, completion-tokens, colon-tokens, go-on-tokens and postlude-tokens. Each of these entities has been described there as a "unit-list-or-declaration". On the next lower level, a "unit-list-or-declaration" has been partitioned into smaller entities, separated by and-also-tokens; such a smaller entity may be considered as a "unit-or-declaration-or-definition".

The priority of the gap-insert has been chosen such that each "unit-or-declaration-or-definition" is considered as a coherent piece of text and, therefore, none of its operators should be moved across its boundaries. As a consequence, one would on the one hand wish that

prio(is defined as token) > prio(gap insert) > prio(and also token)

while on the other hand it is desirable that

prio(separate and also token) > prio(gap insert) > prio(go on token).

(These inequalities follow from the table of priorities of operators given in Appendix C of [10]). However, since

prio(and also token) > prio(separate and also token)

these inequalities cannot be fulfilled at one and the same time. Therefore, the priority of the gap-insert is determined by the context at

hand. The precise details follow easily from the algorithm given below.

5. THE TRANSDUCER

In this section, a brief outline of the actual transducer will be given. The algorithm is a straightforward variant of the standard operator-precedence parsing algorithm. The variation concerns the treatment of incorrect input texts. As inputs, the algorithm receives the tables of precedence relations and operator priorities, as given in Appendices B and C of [10], respectively.

The algorithm treats one parenthesized construct, and is called recursively if the start of another such construct is encountered. The algorithm makes use of a stack; each operator to be moved left is pushed onto this stack, and eventually popped off again. The routine 'top' delivers the symbol on top of the stack. The closing symbol of the parenthesized construct is pushed onto the stack as well; it is given the lowest possible priority, so that it serves as a barrier to prohibit any operators from a surrounding construct to be popped off. The closing symbol itself is popped off when the parenthesized construct is finished.

```
PROC transducer = (SYMBOL sym) VOID:
BEGIN output(sym); push(sym);
  WHILE SYMBOL old = sym; sym:= next symbol;
    IF no precedence relation(sym, old)
    THEN # gap-insert #
      WHILE NOT (closer(top) OR and also token(top)
                              OR go on token(top))
      DO output(top); pop OD
    FI;
    IF operator(sym)
    THEN
      WHILE prio(sym) < prio(top) DO output(top); pop OD;
      IF prio(sym) = prio(top)
      THEN (right associative(sym) | output(top); pop);
        output(synchro); push(sym)
      ELIF no prefix transduction(sym) THEN output(sym)
      ELSE output(synchro); push(sym)
      FI;
      TRUE
    ELIF operand(sym) THEN output(sym); TRUE
    ELIF closer(sym) THEN transducer(sym); TRUE
    ELSE # opener or middler #
      WHILE NOT closer(top) DO output(top); pop OD;
      output(sym);
      (middler(sym) | TRUE | pop; FALSE)
    FI
  DO SKIP OD
END.
```

6. ERROR RECOVERY DURING TOP-DOWN PARSING

Except for the treatment of incorrect input texts, the transition from the LL(1) grammar as given in the Appendix to the corresponding recursive descent parser is straightforward. As for incorrect input texts, problems may arise when the input contains symbols that are not expected at a given point.

Consider, e.g., the production rule

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

Typically, such a rule leads to a parser routine like

```
PROC out choice clause = BOOL:
  IF choice out THEN serial clause; TRUE
  ELIF choice again THEN chooser choice clause; TRUE
  ELSE FALSE
  FI;
```

In this scheme, each terminal symbol leads to a boolean routine that attempts to read that terminal symbol from the input stream, and returns true (false) if this attempt succeeds (fails). Each nonterminal symbol leads to a boolean routine similar to the one given above. This boolean routine succeeds if the first member of one of its alternatives succeeds. As soon as the first member of an alternative succeeds, the following members must necessarily succeed. This is the standard backtrack problem for recursive descent parsers. If any of those following members should fail, an error message has to be given, and a place must be found where parsing may be resumed.

It is quite easy to decide on a minimal set of places where error messages may have to be given. Initially, only those routines that correspond to rules which may produce the empty string will always succeed. If any backtrack problem remains, one of the routines that causes problems is augmented by adding an extra (always succeeding) alternative corresponding to an error message. This process is iterated until no more backtrack problems exist. (By using the ALEPH compiler, this test can be performed mechanically. ALEPH [14] is a programming language especially suited for writing recursive descent parsers. Its compiler checks the "backtrack liability" of all routines.)

Knowing all places where error messages must be given, we still have to find ways to bring the parser back on the right track again. Bearing in mind the fact that (i) the parenthesis skeleton is assumed to be correct, and (ii) synchronization symbols mark the original place of occurrence of operators that are moved left by the transducer, it is straightforward to verify that by skipping the input text until the next

closing parenthesis or synchronization symbol in case of an error, the parser will always be back on the track again.

REFERENCES

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

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

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

[4] 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.

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

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

[7] 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).

[8] FLOYD, R.W., Syntactic analysis and operator precedence, JACM 10, 3 (1963), pp 316-334.

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

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

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

[12] MEERTENS, L.G.L.T. & J.C. VAN VLIET, Parsing ALGOL 68 with syntax-directed error recovery, 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 54/75).

[13] MEERTENS, L.G.L.T. & J.C. VAN VLIET, A syntax-directed error-recovery method for parsing ALGOL 68 programs, Report IN 6/73, Mathematical Centre, Amsterdam, 1973.

[14] GRUNE, D., R. BOSCH & L.G.L.T. MEERTENS, ALEPH manual, Report IW 17/74, Mathematical Centre, Amsterdam, 1974.

APPENDIX

```
# The LL(1) grammar for ALGOL 68+ #

# terminal symbols #

open mark;
bold begin token;
big begin token;
choice start;
brief sub token;
loop insert;
def token;
access token;
choice in;
choice again;
choice out;
for token;
from token;
by token;
to token;
while token;
do token;
close mark;
bold end token;
big end token;
choice finish;
brief bus token;
od token;
fed token;
ssecca insert;
egg token;
egg defined as token;
postlude token;
completion token;
go on token;
separate and also token;
public token;
priority token;
mode token;
ldec token;
module token;
dectag insert;
opdec insert;
and also token;
is defined as token;
at token;
colon mark;
specification token;
```

```
becomes token;
identity relator;
routine token;
code token;
dyadic operator;
monadic operator;
of token;
cast insert;
clice insert;
reference to token;
leap token;
structure token;
flexible token;
procedure token;
union of token;
operator token;
go to token;
row insert;
formals insert;
invoke insert;
formal nest token;
language indication;
digit token;
tag token;
format text;
string denoter;
other denoter;
parallel token;
choice token;
defining operator;
mode indication;
module indication;
skip token;
nil token;
synchro.


# production rules #

brief begin token:
    open mark.
brief end token:
    close mark.
style i sub token:
    open mark.
style i bus token:
    close mark.
hole indication:
    string denoter.
```

```
input text:
    big begin token, compilation input, big end token.
compilation input:
    lenclosed clause;
    prelude packet;
    stuffing or definition module packet.
lenclosed clause:
    colon mark, identifier, synchro, lenclosed clause;
    enclosed clause.
prelude packet:
    module declaration.
stuffing or definition module packet:
    egg token, stuffing definition.
stuffing definition:
    hole indication, egg defined as token,
      actual hole or module declaration.
actual hole or module declaration:
    actual hole;
    module declaration.
enclosed clause:
    closed or collateral clause;
    parallel clause;
    choice clause;
    loop clause;
    access clause.
closed or collateral clause:
    begin, inner clause, end.
begin:
    bold begin token;
    brief begin token.
end:
    bold end token;
    brief end token.
inner clause:
    serial clause;
    (joined portrait).
parallel clause:
    parallel token, closed or collateral clause.
serial clause:
    series.
series:
    train, (completion token, series).
train:
    go on token, declun, synchro, train;
    lunit.
declun:
    declaration;
    lunit.
```

```
lunit:
    colon mark, identifier, synchro, lunit;
    unit.
joined portrait:
    and also token, unit or joined portrait, synchro, unit.
unit or joined portrait:
    unit;
    joined portrait.
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 proper;
    united case part.
case part list proper:
    and also token, case part list, synchro, case part.
case part list:
    and also token, case part list, synchro, case part;
    case part.
case part:
    unit;
    united case part.
united case part:
    specification token, single declaration brief pack,
      synchro, unit.
single declaration brief pack:
    brief begin token, single declaration, brief end token.
single declaration:
    dectag insert, declarer, synchro, identifier;
    declarer.
out choice clause:
    choice out, serial clause;
    choice again, chooser choice clause.
loop clause:
    loop insert, 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.
access clause:
    revelation, invoke insert, enclosed clause.
revelation:
    access token, joined module call, ssecca insert.
joined module call:
    module call, (separate and also token, joined module call).
module call:
    (public token), invocation.
invocation:
    module indication.
declaration:
    publety ldecety declaration, (separate and also token,
      declaration).
publety ldecety declaration:
    (public token), ldecety declaration.
ldecety declaration:
    (ldec token), common declaration.
common declaration:
    mode declaration;
    priority declaration;
    identifier declaration;
    operation declaration;
    module declaration.
mode declaration:
    mode token, mode joined definition.
mode joined definition:
    and also token, mode joined definition, synchro,
      mode definition;
    mode definition.
mode definition:
    is defined as token, defined mode indication, synchro,
      declarer or code.
defined mode indication:
    mode indication.
declarer or code:
    declarer;
    code.
priority declaration:
    priority token, priority joined definition.
```

```
priority joined definition:
    and also token, priority joined definition, synchro,
      priority definition;
    priority definition.
priority definition:
    is defined as token, operator, synchro, priority unit.
priority unit:
    digit token.
identifier declaration:
    dectag insert, leapety declarer, synchro,
      identifier joined definition.
leapety declarer:
    (leap token), modine declarer.
modine declarer:
    nonproc declarer;
    modine procedure declarator.
modine procedure declarator:
    procedure token, (formal procedure plan).
identifier joined definition:
    and also token, identifier joined definition, synchro,
      identifier definition;
    identifier definition.
identifier definition:
    identity definition;
    variable definition.
identity definition:
    is defined as token, identifier, synchro, ldecety source.
ldecety source:
    unit or code;
    choice token, ldec source choice list brief pack.
unit or code:
    unit;
    code.
code:
    code token, code string.
code string:
    string denoter.
ldec source choice list brief pack:
    brief begin token, ldec source choice list, brief end token.
ldec source choice list:
    and also token, ldec source choice list, synchro,
      ldec source choice;
    ldec source choice.
ldec source choice:
    colon mark, choice, synchro, unit or code.
choice:
    dyadic operator, synchro, length denoter.
length denoter:
    minus token option, integral denoter.
```

```
minus token option:
    (monadic operator).
integral denoter:
    other denoter.
variable definition:
    becomes token, identifier, synchro, unit;
    identifier.
operation declaration:
    opdec insert, operation heading, synchro,
      operation joined definition.
operation heading:
    operator token, (formal procedure plan).
operation joined definition:
    and also token, operation joined definition, synchro,
      operation definition;
    operation definition.
operation definition:
    is defined as token, operator displayety, synchro,
      ldecety source.
operator displayety:
    operator;
    operator display.
operator display:
    choice token, operator list brief pack.
operator list brief pack:
    brief begin token, operator list, brief end token.
operator list:
    and also token, operator list, synchro, operator;
    operator.
operator:
    defining operator.
module declaration:
    module token, module joined definition.
module joined definition:
    and also token, module joined definition, synchro,
      module definition;
    module definition.
module definition:
    is defined as token, defining indication, synchro,
      module text.
defining indication:
    module indication.
module text:
    (revelation), module series pack.
module series pack:
    def token, module series, fed token.
module series:
    module prelude, (module postlude).
```

```
module prelude:
    go on token, decl or unit, synchro, module prelude;
    decl or unit.
decl or unit:
    declaration;
    unit.
module postlude:
    postlude token, postlude series.
postlude series:
    go on token, unit, synchro, postlude series;
    unit.
declarer:
    nonproc declarer;
    procedure declarator.
nonproc declarer:
    reference to declarator;
    structured with declarator;
    flexible rows of declarator;
    rows of 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:
    common portrayer, (separate and also token, portrayer).
common portrayer:
    dectag insert, declarer, synchro, joined definition of fields.
joined definition of fields:
    and also token, joined definition of fields, synchro,
      field selector;
    field selector.
flexible rows of declarator:
    flexible token, declarer.
rows of declarator:
    row insert, rower bracket, synchro, declarer.
rower bracket:
    brief sub token, rower, brief bus token;
    style i sub token, rower, style i bus token.
rower:
    and also token, rower, synchro, row rower;
    row rower.
row rower:
    colon mark, (unit), synchro, (unit);
    (unit).
procedure declarator:
    procedure token, formal procedure plan.
```

```
formal procedure plan:
    formals insert, joined declarer pack, synchro, declarer;
    declarer.
joined declarer pack:
    brief begin token, joined declarer, brief end token.
joined declarer:
    and also token, joined declarer, synchro, declarer;
    declarer.
union of declarator:
    union of token, joined declarer pack.
identifier:
    tag token.
field selector:
    tag token.
unit:
    assignation;
    identity relation;
    routine text;
    formal hole;
    tertiary.
tertiary:
    formula;
    secondary.
secondary:
    leap generator;
    selection;
    primary.
primary:
    primary one;
    other denoter;
    format text;
    skip token;
    nil token.
primary one:
    slice call;
    cast;
    string denoter;
    identifier;
    jump;
    enclosed clause.
assignation:
    becomes token, tertiary, synchro, unit.
identity relation:
    identity relator, tertiary, synchro, tertiary.
leap generator:
    leap token, declarer.
selection:
    of token, field selector, synchro, secondary.
slice call:
    clice insert, primary one, synchro, indexer bracket.
```

```
indexer bracket:
    brief sub token, indexer, brief bus token;
    style i sub token, indexer, style i bus token.
indexer:
    and also token, indexer, synchro, trimscript;
    trimscript.
trimscript:
    unit;
    at token, (bound pair), synchro, unit;
    (bound pair).
bound pair:
    colon mark, (unit), synchro, (unit).
routine text:
    routine token, routine heading, synchro, unit.
routine heading:
    formals insert, declarative pack, synchro, declarer;
    declarer.
declarative pack:
    brief begin token, declarative, brief end token.
declarative:
    common declarative, (separate and also token, declarative).
common declarative:
    dectag insert, declarer, synchro, parameter joined definition.
parameter joined definition:
    and also token, parameter joined definition, synchro,
      identifier;
    identifier.
formula:
    dyadic formula;
    monadic formula.
dyadic formula:
    dyadic operator, operand, synchro, monadic operand.
monadic formula:
    monadic operator, monadic operand.
operand:
    formula;
    secondary.
monadic operand:
    monadic formula;
    secondary.
jump:
    go to token, identifier.
cast:
    cast insert, declarer, synchro, enclosed clause.
formal hole:
    formal nest token, nest tail.
actual hole:
    enclosed clause.
nest tail:
    (language indication), hole indication.
```