stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 173/81 AUGUSTUS

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

AN OPERATOR-PRIORITY GRAMMAR FOR 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 operator-priority grammar for 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. In this report, such an operator-priority grammar for ALGOL 68+ is given. An account of the differences between the language generated by that grammar, and ALGOL 68+, insofar as these are due to the transition to an operator-priority grammar, is given as well. These differences somehow have to be catered for during the parsing process.

KEY WORDS & PHRASES: ALGOL 68+, operator-precedence grammar, top-down parsing

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 will be introduced, leading to the notion of an operator-priority grammar.

For terminal symbols a and b, we define $a < b$ as: either $a \lessdot b$ or no precedence relation holds between a and b, and similarly for $a = b$ and $a > b$. This relation can be extended to sets of terminal symbols. If A and B are sets of terminal symbols, we define $A < B$ as:

$$A < B \iff \forall a \in A, b \in B: a < b.$$

$A = B$ and $A > B$ are defined in a similar way.

The first restriction that is impcsed can now be stated as follows:

(1) The sets of terminal symbols can be partitioned into sets O, M, C and P satisfying:

i) $O < O$, $O = M$, $O = C$, $O < P$;
ii) $M < O$, $M = M$, $M = C$, $M < P$;
iii) $C > M$, $C > C$, $C > P$;
iv) $P < O$, $P > M$, $P > C$.

Though this restriction looks rather complex, it is in fact quite easily satisfied. Suppose a production rule of the grammar contains terminal symbols $a_1, \ldots, a_n$, 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.

If the grammar is made such that the openers, middlers, closers and operators form mutually disjoint sets, then restriction 1 is automatically fulfilled if we define O, M, C and P to be those sets, respectively. This can easily be verified from the definition of an operator-precedence grammar.

The above observation allows us to partly fill in the empty entries in the table in a consistent way. The remaining empty spots concern the precedence relations between closers and openers, and mutually between operators. However, even for arbitrary (incorrect) input texts, the transducer need never ask for a relation between a closer and an opener (see [10] for further details).

As for the operators, a further restriction is imposed:

(2) The set of operators can be partitioned into sets $A_1, \ldots, A_m$ satisfying:

i) There exists a total linear order << between the sets satisfying, for $i \neq j$:

$$A_i << A_j \Rightarrow A_i < A_j \wedge A_j > A_i.$$

ii) For each of the sets A_i, $1 \leq i \leq m$:

$$A_i < A_i \vee A_i > A_i.$$

Intuitively, the total linear order << implies that if $A_i << A_j$, then elements from A_i always occur at a higher node in the parse tree than those from A_j, provided no parenthesized constructs occur in between. (The operators with higher priorities tend to occur at lower nodes.)

Restriction 2ii) above implies that operators from one and the same set are either all right-associative (<) or all left-associative (>) operators. It should be noted that the partitioning of the set of operators is not necessarily unique. If there is a proper subset of one of the A_i such that no precedence relation holds between any pair of operators from the subset, then the partitioning may be refined. Arguments related to the error-recovery method envisaged (see [10]) may then be applied in order to choose the more desirable partitioning. A short survey of the partitioning we have decided on is given in Appendix C.

If an operator-precedence grammar fulfills the requirements under (1) and (2) above, it will be called an operator-priority grammar. Such an operator-priority grammar for ALGOL 68+ is given in Appendix A. In Appendix B, the corresponding table of precedence relations is given. The terminal symbols have been listed and the sets A_i have been delineated such that the fulfillment of the operator-priority requirements can be readily verified.

The test for the grammar being operator-precedence was performed mechanically. Various initial clashes came to light hereby. The measures taken to make the grammar operator precedence can be distinguished in three 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 Appendix A), 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.

(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, whereas category b serves to resolve clashes in the precedence relations.)

The check for the further restrictions on the grammar was performed manually, by inspecting the table of precedence relations. A few iterations were needed before the final grammar was obtained. This grammar and the table of precedence relations are given in Appendix A and B, respectively. Appendix C gives a list of the priorities of the various operators and their left or right associativity.

When actually parsing ALGOL 68+ texts, the same modifications must be applied. The task of making the distinctions of category b above, and of placing the inserts of category c, can largely be delegated to the lexical pass. A precise description of how the input texts can be made to conform to the operator-priority grammar is given in [11]. In section 2 below, only a short summary of these modifications is given.

The further differences between ALGOL 68+ and the language described by the operator-priority grammar are given in section 3. Some of the changes underlying these differences have been made to resolve clashes in the precedence relations, or to get a proper ordering between the operators. Others are the result of combining rules whose productions partly overlap, such as those for calls and slices. For instance, there is no way to decide whether "a(1)" is a slice or a call without knowing the mode of the identifier "a". In such cases, the syntax has been relaxed by combining the production rules, so as to make possible the top-down parsing method based on an LL(1) grammar. It is mainly the task of the syntax- and semantic-analysis phases of the compiler to take these differences into account. (The differences caused by the transition to a context-free grammar, as given in [5], must be added to the list.)

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). This allows for a unique assignment of semantics to the nodes of the parse tree during the subsequent mode-independent analysis. This top-down parsing method is further discussed in [10].

2. ADJUSTMENTS TO BE MADE DURING LEXICAL ANALYSIS

It is one of the duties of the lexical phase of the parser to cope with the differences listed in this section.

- 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 (i.e., the is-defined-as-token from the stuffing-definition) and operator;
  : as colon-mark, specification-token and routine-token;
  ~ as skip-token and as operator.

- On the lowest level, a distinction 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 an identifier-declaration;
  - the opdec-insert is placed between the operation-heading and the following 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-declarer;
  - 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.

3. FURTHER DIFFERENCES

- Closed-clauses and collateral-clauses are treated alike. This means that the following texts will also be accepted:

```
par ();
par (1);
par (1; 2).
```

  This can easily be dealt with during syntax analysis.

- After exit, no label-definition is required. Obviously, the mode-independent pass can easily catch this case.

- In identifier-declarations and -definitions, identity and variable have been collapsed. As a consequence, the following texts are also accepted:

```
loc int a = 0;
[] int a:= (1, 2);
[1:2] int a = (1, 2);
int a = 1, b:= 2.
```

  The mode-independent pass can use a variable "idvar", with possible values "identity", "variable" and "unknown". The starting value is determined by the leapety-declarer: when a leap-token or apparent actual declarer is encountered, its value is variable, when an apparent formal declarer is encountered, its value is identity, and otherwise it is unknown. For each separate identifier-definition, its actual idvar can be determined.

- For the same reason,

```
bool b:= 'code "GENERATE"
```

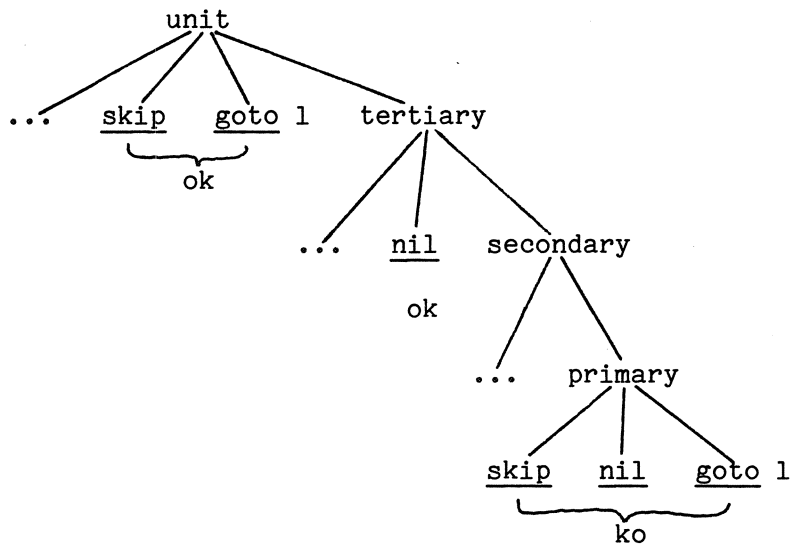
  is also accepted. Here also, the mode-independent pass can easily check this, using the above-mentioned variable "idvar".

- Skips, nihils and jumps are treated as primary, to improve error recovery. Because of this, the following is also accepted:

```
a :=: skip;
b of nil;
1 + goto 1.
```

  This can easily be encompassed in the mode-independent pass by treating them as an erroneous alternative at their present occurrence in the grammar, and by incorporating them as proper occurrence at the original place - where, in recognizing a unit or tertiary, the respective alternatives tertiary and secondary should appear last (note that this

grammar is not of type LL(1) any more):



The case of a wrongly placed goto-less jump, as in 1 + 1, cannot be detected in the mode-independent pass. All those cases will however pop up when doing the coercions: the forbidden jumps never occur in a strong position. Exception: a :=: 1 or 1 :/=: a. This case can be treated separately in the mode-dependent pass. (Note that a :=: (1) is correct!)

- For simplicity's sake the nine priority levels for dyadic operators are reduced to one in the operator-priority grammar.

- Slices and calls have been collapsed. So a construction like sin[1:2] is also accepted. This kind of error can only be detected during mode-dependent analysis.

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, On top-down parsing of ALGOL 68+, Mathematical Centre, Amsterdam, to appear.

[11] VAN VLIET, J.C., Making ALGOL-68+-texts conform to an operator-priority grammar, Mathematical Centre, Amsterdam, to appear.

APPENDIX A -- OPERATOR-PRIORITY GRAMMAR OF ALGOL 68+

In the grammar given below, comments are placed between style-ii-comment-symbols ("#"). The grammar starts with a list of the terminal symbols, separated by semicolons and terminated by 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 a notion, or a list of notions separated by commas and enclosed between the syntactic marks "(" and ")". In the latter case, the direct productions of that member are: empty, and the enclosed list of notions. Obviously, this mechanism does not enlarge the descriptive power, but is merely an expedient way to shorten the syntax.

In the comments in the grammar below, the numbers refer to the corresponding section numbers in [2], as possibly modified by [3]. The representations of the various bold symbols are given in the point-stropping regime. Newly added symbols have a representation which contains the escape-character ("'"). Symbols may have more than one representation; in that case, these are separated by spaces, and three dots indicate that only examples are given.

In an operator-precedence grammar, one in general uses special markers to indicate the start and end of the input text. These markers are then used to properly initialize and terminate the parsing process. In our grammar these are termed "begin of input token" and "end of input token", respectively, and a production rule

```
input text:
    begin of input token, compilation input, end of input token.
```

has been added.

```
# terminal symbols #

# openers #

    open mark;                          # ( #
    bold begin token;                   # .begin #
    begin of input token;               # 'begin #
    choice start;                       # .if .case ( #
    brief sub token;                    # [ #
    loop insert;                        # 'loop #
    def token;                          # .def #
    access token;                       # .access #

# middlers #

    choice in;                          # .then .in | #
    choice again;                       # .elif .ouse |: #
    choice out;                         # .else .out | #
    for token;                          # .for #
    from token;                         # .from #
    by token;                           # .by #
    to token;                           # .to #
    while token;                        # .while #
    do token;                           # .do #

# closers #

    close mark;                         # ) #
    bold end token;                     # .end #
    end of input token;                 # 'end #
    choice finish;                      # .fi .esac #
    brief bus token;                    # ] #
    od token;                           # .od #
    fed token;                          # .fed #
    ssecca insert;                      # 'ssecca #

# proper operators, in ascending order of priority #

# priority a tokens #
    egg token;                          # .egg #

# priority b tokens #
    egg defined as token;               # 'edat #

# priority c tokens #
    postlude token;                     # .postlude #
    completion token;                   # .exit #

# priority d tokens #
    go on token;                        # ; #
```

```
# priority e tokens #
    separate and also token;          # ´sep #

# priority f tokens #
    public token;                     # .pub #

# priority g tokens #
    priority token;                   # .prio #
    mode token;                       # .mode #
    ldec token;                       # .´ldec #
    module token;                     # .module #

# priority h tokens #
    dectag insert;                    # ´dectag #
    opdec insert;                     # ´opdec #

# priority i tokens #
    and also token;                   # , #

# priority j tokens #
    is defined as token;              # ´idat #
    at token;                         # @ .at #

# priority k tokens #
    colon mark;                       # : #
    specification token;              # ´spec #

# priority l tokens #
    becomes token;                    # := #
    identity relator;                 # :=: .is :≠: :/=: .isnt #
    routine token;                    # ´rout #
    code token;                       # .´code #

# priority m tokens #
    dyadic operator;                  # +:= .over .xyz ... #

# priority n tokens #
    monadic operator;                 # + .not .xyz ... #

# priority o tokens #
    of token;                         # .of #

# priority p tokens #
    cast insert;                      # ´cast #
    clice insert;                     # ´clice #
```

```
# priority q tokens #
    reference to token;                 # .ref #
    leap token;                         # .loc .heap #
    structure token;                    # .struct #
    flexible token;                     # .flex #
    procedure token;                    # .proc #
    union of token;                     # .union #
    operator token;                     # .op #
    go to token;                        # .goto .go.to #
    row insert;                         # ´row #
    formals insert;                     # ´formals #
    invoke insert;                      # ´invoke #
    formal nest token;                  # .nest #
    language indication;                # .fortran ... #

# operands #
    digit token;                        # 1 2 3 4 5 6 7 8 9 #
    tag token;                          # i ... #
    format text;                        # $3zd$ ... #
    string denoter;                     # "string" ... #
    other denoter;                      # 3.14 .true .empty ... #
    parallel token;                     # .par #
    choice token;                       # .´choice #
    defining operator;                  # +:= .over .xyz ... #
    mode indication;                    # .int ... #
    module indication;                  # .matrix ... #
    skip token;                         # .skip ~ #
    nil token.                          # .nil ° #

#9.4.1. representations of symbols.#

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

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

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

hole indication:
    string denoter.
```

```
#10.7.1. compilation inputs#

input text:
    begin of input token, compilation input, end of input token.
compilation input:
    lenclosed clause;
    prelude packet;
    stuffing or definition module packet.

lenclosed clause:
    label definition, 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.

#3. clauses.#

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

#3.1. closed clauses.#

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.
```

#3.2. serial clauses.#

serial clause:
    series.
series:
    train, (completion token, series).
train:
    declun, go on token, train; lunit.
declun:
    declaration; lunit.
lunit:
    label definition, lunit; unit.
* label definition:
    identifier, label token.

#3.3. collateral clauses; see also 3.1.#

joined portrait:
    unit or joined portrait, and also token, unit.
unit or joined portrait:
    unit; joined portrait.

#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 proper; united case part.
case part list proper:
    case part list, and also token, case part.
case part list:
    (case part list, and also token), case part.
case part:
    unit; united case part.
united case part:
    specification, unit.
* specification:
    single declaration brief pack, specification token.

```
single declaration brief pack:
    brief begin token, single declaration, brief end token.
single declaration:
    declarer, (dectag insert, identifier).

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

#3.5. loop clauses.#

```
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.
```

#3.6. access clauses.#

```
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.
```

```
#4. declarations.#

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.

#4.2. mode declarations.#

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

#4.3. priority declarations.#

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

```
#4.4. identifier declarations.#

identifier declaration:
    leapety declarer, dectag insert, 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:
    (identifier joined definition, and also token),
        identifier definition.
identifier definition:
    identity definition; variable definition.
identity definition:
    identifier, is defined as token, 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:
    (ldec source choice list, and also token),
        ldec source choice.
ldec source choice:
    choice, up to token, unit or code.
choice:
    dyadic operator, length denoter.
length denoter:
    minus token option, integral denoter.
* minus token option:
    (monadic operator).
integral denoter:
    other denoter.

variable definition:
    identifier, (becomes token, unit).
```

```
#4.5. operation declarations.#

operation declaration:
    operation heading, opdec insert,
        operation joined definition.
operation heading:
    operator token, (formal procedure plan).
operation joined definition:
    (operation joined definition, and also token),
        operation definition.
operation definition:
    operator displayety, is defined as token, 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:
    (operator list, and also token), operator.
operator:
    defining operator.

#4.9. module declarations.#

module declaration:
    module token, module joined definition.
module joined definition:
    (module joined definition, and also token),
        module definition.
module definition:
    defining indication, is defined as token, 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:
    decl or unit, (go on token, module prelude).
decl or unit:
    declaration;
    unit.
```

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

#4.6. declarers.#

```
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:
    declarer, dectag insert, joined definition of fields.
joined definition of fields:
    (joined definition of fields, and also token), field selector.

flexible rows of declarator:
    flexible token, declarer.

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

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

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

#4.8. indicators and field selectors.#

```
identifier:
    tag token.
field selector:
    tag token.
```

#5. units.#

```
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; #go to# jump;
    enclosed clause.
```

#5.2.1. assignations.#

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

#5.2.2. identity relations.#

```
identity relation:
    tertiary, identity relator, tertiary.
```

#5.2.3. generators.#

```
leap generator:
    leap token, declarer.
```

#5.3.1. selections.#

selection:
    field selector, of token, secondary.

#5.3.2. slices.#

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

#5.4.1. routine texts.#

routine text:
    routine heading, routine token, unit.
routine heading:
    (declarative pack, formals insert), declarer.
declarative pack:
    brief begin token, declarative, brief end token.
declarative:
    common declarative, (separate and also token, declarative).
common declarative:
    declarer, dectag insert, parameter joined definition.
parameter joined definition:
    (parameter joined definition, and also token), identifier.

#5.4.2. formulas.#

formula:
    dyadic formula; monadic formula.
dyadic formula:
    operand, dyadic operator, monadic operand.
monadic formula:
    monadic operator, monadic operand.
operand:
    formula; secondary.
monadic operand:
    monadic formula; secondary.

#5.4.3. calls.# #see 5.3.2.#

#5.4.4. jumps.#

jump:
    #(#go to token#)#, identifier.

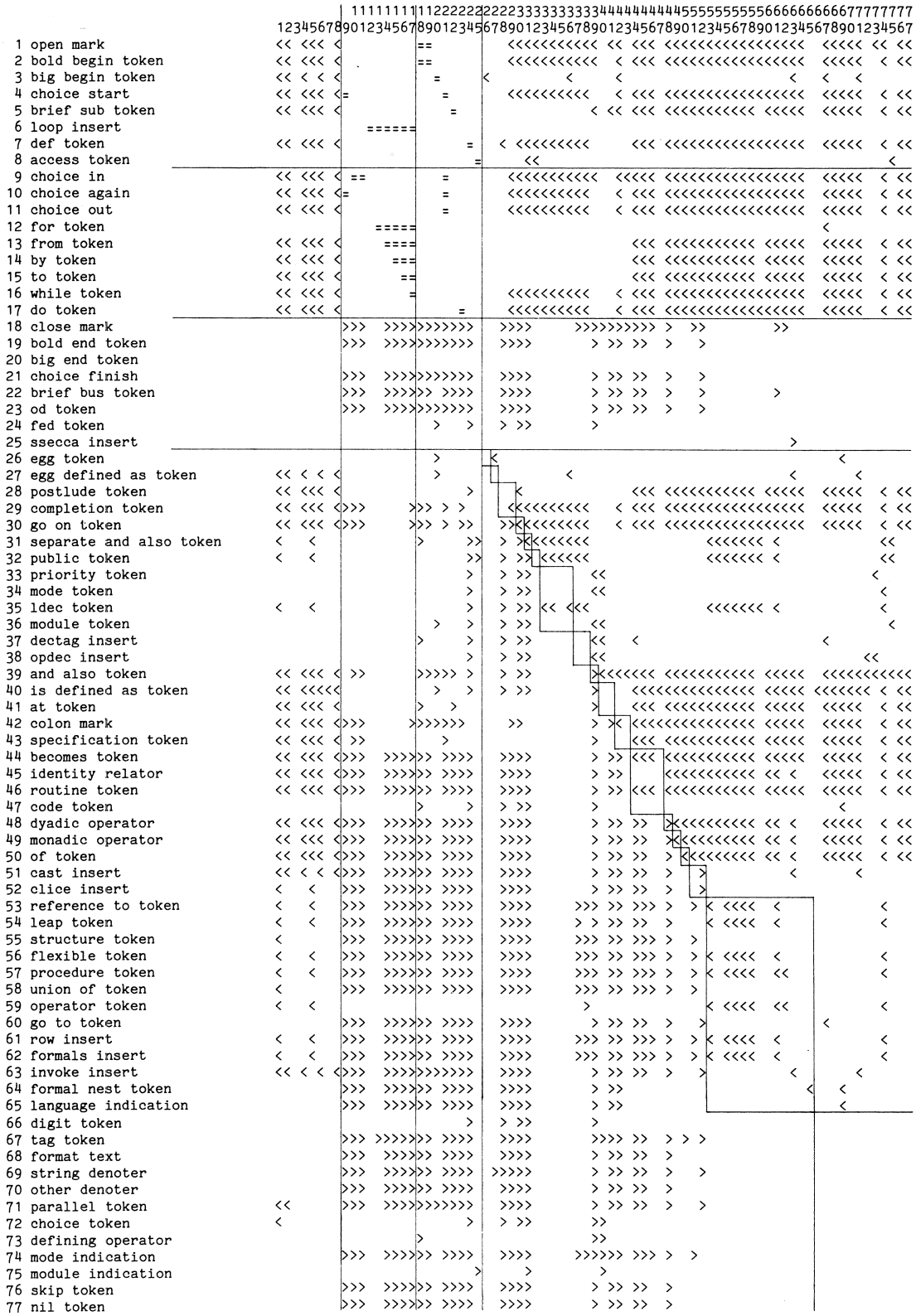
#5.5.1. casts.#

cast:
    declarer, cast insert, enclosed clause.

#5.6. holes.#

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

APPENDIX B — TABLE OF PRECEDENCE RELATIONS



APPENDIX C

Below, a short survey of the various sets of operators is given, in descending order of priority. The set of operators with the highest priority, also termed "operands", is not included in the list below. For each set of operators, it is indicated whether they are left-associative ("L") or right-associative ("R").

R 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.

L cast insert; clice insert.

R of token.

R monadic operator.

L dyadic operator.

R becomes token; identity relator; routine token; code token.

R colon mark; specification token.

R is defined as token; at token.

L and also token.

R dectag insert; opdec insert.

R priority token; mode token; ldec token; module token.

R public token.

R separate and also token.

R go on token.

R postlude token; completion token.

R egg is defined as token.

R egg token.