stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IW 54/75 DECEMBER

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

PARSING ALGOL 68 WITH SYNTAX-DIRECTED ERROR RECOVERY

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, 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), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

AMS(MOS) subject classification scheme (1970): 68A10, 68A30

ACM-Computing Reviews-categories: 5.23, 4.22

Parsing ALGOL 68 with syntax-directed error recovery

by

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

ABSTRACT

The generality of ALGOL 68 makes it difficult to obtain good error recovery when the traditional top-down error-recovery method is applied. An error-recovery technique is described for operator-precedence languages, relying on the existence of an algorithm for repairing incorrect parenthesis skeletons. This technique was used to construct an LL(1) grammar for parsing the prefix form of any source text with repaired parenthesis skeleton. The number of places in the source text where "resynchronization" takes place is considerably enlarged. In fact, resynchronization takes place for each terminal symbol involved in the production rule currently applied.

In order to apply the technique to ALGOL-68 programs, the lexical scan has to insert additional operators in the source text, such as a call/slice operator between a primary and a following pack or bracket.

KEY WORDS & PHRASES: *ALGOL 68, syntax-directed error recovery, operator precedence,* LL(1) *grammar, prefix transduction.*

1. INTRODUCTION

The primary purpose of error recovery in the parsing of programs is to minimize the number of runs required to obtain a syntactically correct program. This goal is achieved by continuing the parsing in a "meaningful" way after a syntactic error has been detected, so that pertinent information may be given on errors occurring further on in the source text.

The generality of ALGOL 68 [1] makes good error recovery considerably more difficult than it is, for instance, in ALGOL 60. Investigating this problem, we concluded that the bottleneck for good resynchronization of the parser was formed by the problem of unbalanced parentheses. Therefore, it was decided that in the machine-independent ALGOL-68 compiler which is currently being developed at the Mathematical Centre, incorrect parenthesis skeletons will be repaired before the source text is parsed. This is treated in detail in [2] and [3]. This decision now appears to pay off in a twofold way:

(i) At an early stage it was decided to parse top-down. As a tool for writing our compiler we have at our disposal the language ALEPH [4,5], which is particularly suited for top-down parsing according to a grammar of type LL(1)[6]. The context-free grammar *underlying* [7] the ALGOL-68 syntax 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 [8], operating from right to left, which brings the source text in prefix form. 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.

(ii) The presumption that knowledge about errors in the parenthesis skeleton would alleviate the problems of error recovery was confirmed in a stronger way than we expected: The transduction scheme mentioned above can be amended in such a way that all possible errors in the source text are described syntactically. Error recovery then simply becomes a side effect of syntax-directed parsing.

It is not surprising that the application of a right-to-left transduction scheme opens possibilities for error recovery: it can be viewed as an unbounded lookahead from left to right.

The purpose of this paper is to sketch this error-recovery technique. This is done by describing the technique in an informal way. Subsequently, it is demonstrated on a small example. We conclude with a section on the specific difficulties involved in applying the technique to ALGOL 68.

2. CONVENTIONS, TERMINOLOGY

We shall refrain from giving a formal definition of well-established concepts, such as "context-free grammar", the "language described by" a grammar, "parse tree", "productions", etc. Instead, we shall introduce our conventions in an informal way. Following the ALGOL-68 terminology, non-terminal symbols will be called *notions*, and terminal symbols will be called *symbols*.

2.1. *LL(k)-grammars and top-down parsing*

In this paper, the concept of "LL(k) grammar" is used. For a definition we refer to [6]. We shall only mention here those properties that are relevant to the exposition.

If a grammar is of type LL(k), this means that it is possible to construct a parse tree for a text described by that grammar in the following way: Start with (i) a partial parse tree consisting of only one (top) node, labeled with the start notion, and (ii) that text. The top node is said to be "untreated". In a number of successive steps, the parse tree will be developed by attaching to some bottom node which is labeled with a notion a number of (untreated) descendants, one for each symbol or notion of one of its productions. At the same time, the text will be *accepted* by deleting from left to right successive symbols. Each step has the following form: Take the leftmost untreated node in the partial parse tree (this is always a bottom node). That node is then "treated" as follows: If that node is labeled with a notion, select, on the basis of that notion and the first k symbols of the text, a production for that notion and develop the parse tree

accordingly. (The selection is uniquely determined for an LL(k) grammar.) If that node is labeled with a symbol, it is equal to the first symbol of the remaining text (this is a property of the selection procedure for LL(k) grammars), and that first symbol is deleted.

If the text was indeed produced by the given grammar, this parsing process will terminate with a complete parse tree (all nodes treated) and an empty remaining text. Otherwise, the process terminates with a nonempty remaining text or at some stage in the process no selection is possible.

A parsing method as sketched above is known as a *top-down* method; the fact that the selection is uniquely determined, so that no decisions have ever to be undone, labels this method as *deterministic*. It may be easily implemented by a system of mutually recursive routines, one for each notion. During the parsing process, the untreated part of the tree is reflected in the status of the link stack.

2.2. *Transduction schemes*

The well-known concepts of translator (translation, translation scheme) and transducer (transduction, transduction scheme) are closely connected. The essential difference between a transducer and a translator is that a translator is guaranteed to work only on proper texts, described by a given grammar, whereas a transducer works on a wider class of input texts: in general, on the language Σ^*, where Σ is the set of input symbols. For a comprehensive description of these concepts and their applicability, we refer to [8]. In our case, we are interested only in proper functioning of the transducer on the subset of Σ^* consisting of texts with a correct parenthesis skeleton. Such transducers can easily be constructed for a particular class of operator-precedence grammars, which we have called "operator-parenthesis grammars".

3. TOP-DOWN ERROR RECOVERY AND ALGOL 68

One advantage of top-down parsing is mentioned by KNUTH: "when we are fortunate enough to have an LL(1) grammar, we have more flexibility in applying semantic rules, since we know what production is being used *before*

we actually process its components. This foreknowledge can be extremely important in practise." [6] (Although this remark specifically refers to LL(1) grammars, it seems to hold for LL(k) grammars in general, provided that the k-symbol lookahead is not considered "processing".)

It is not the purpose of this paper to justify our choice for a particular parsing method, but it should be clear that this choice has profound bearings on the error-recovery techniques possible. GRIES: "The nice part about top-down error recovery is that the partially constructed tree conveys much usable information about what should appear next in the source program. This information is not as readily available in the bottom-up method." [9]

A top-down error-recovery technique is sketched in [9]: If, in the partial parse tree at some stage no step is possible (for a node labeled with a notion: no selection is possible; for a node labeled with a symbol: it is not equal to the first symbol of the remaining string), proceed then upwards in the tree until a node is encountered, labeled with an "important" notion, after which the whole tree descending from that node, including the node itself, is considered treated. Delete, then, successive symbols from the remaining string until a next step is possible. The parsing process may now be resumed. For ALGOL 60, an important notion would be, e.g., "statement". If the parsing process gets stuck in a statement, the effect of this technique would be that the source text is skipped up to a semicolon, *end* or *else*, whereupon the parsing continues. Due to the generality of ALGOL 68, this technique is not straightforwardly applicable. The ALGOL-60 concepts of statement and expression are unified in ALGOL 68 into the "*unit*". A typical example is given by

*print(c:= begin real z = exp(x); (z + 1/z) * .5 end)*,

which in ALGOL 60 could be

*begin real z; z:= exp(x); c:= (z + 1/z) * .5 end; print(c).*

The very least thing to do is not to skip to some resynchronizing symbol such as a semicolon, *end* or *else*, but to make an effort to parse parenthesized constructs encountered meanwhile. But even then, it may be expected

that the freedom of expression in ALGOL 68 will give rise to a style of programming compared to which the ALGOL-60 way of cutting into statements will seem short-breathed. It is therefore desirable to increase the number of points where resynchronization may take place. But if this is done at all, it should be done in a systematic fashion; perhaps no error recovery whatsoever is better than an unsurveyable collection of ad-hoc methods, the combined effect of which may easily go beyond our limited ability to grasp complicated processes.

4. RESYNCHRONIZATION AND PREFIX FORM

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 $\alpha + \beta$. (Here, and in the sequel, we loosely apply such terms as "formula" 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 α, we want it to resume at the operator +. For a top-down parser, the knowledge that α 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 $\alpha + \beta$. This information can be supplied by bringing the source text in prefix form, so that the formula reads $+ \alpha \beta$. 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 $\perp$. Using this, the prefix form of $\alpha + \beta$ becomes $+ \alpha \perp \beta$, 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.

We want to have a grammatical treatment of the prefix-form output texts; to this purpose, the transduction may, as it were, be performed on the productions of the original grammar.

4.1. *Operator-parenthesis and synchronized prefix grammars*

Let G be an operator-precedence grammar (for a definition, see [10]). This implies that each production is of the form $\alpha_0\ \tau_1\ \alpha_1 \cdots \alpha_{n-1}\ \tau_n\ \alpha_n$, where each of the α_i is either ε or a notion and where each of the τ_i is a symbol.* For a production with $n = 1$, τ_1 will be called an *operator*; if $n \geq 2$, τ_1 will be called an *opener*, τ_i will be called a *middler* for $2 \leq i \leq n-1$, and τ_n will be called a *closer*. Together, openers, middlers and closers form the *parentheses*.

Between some pairs of symbols σ and τ, a precedence relation ($\lessdot$, $\gtrdot$ or $\doteq$) is defined. We have $\sigma \gtrdot \tau_i$ for all symbols σ such that $\alpha_{i-1} \overset{*}{\rightarrow} \beta\sigma\alpha'$, $\tau_i \lessdot \sigma$ for all symbols σ such that $\alpha_i \overset{*}{\rightarrow} \alpha'\sigma\beta$, and $\tau_i \doteq \tau_{i+1}$ (where α' is again either ε or a notion).

An *operator-parenthesis grammar* is an operator-precedence grammar G such that the operators, openers, middlers and closers of G form mutually disjoint sets. For such a grammar it is possible to construct a translator which brings source texts in prefix form, only knowing the precedence relations between the symbols. This is in fact the technique used in the first ALGOL-60 translator by DIJKSTRA and ZONNEVELD to translate ALGOL-60 programs to reverse Polish notation (= postfix form) [11].

---

* Usually, the restriction is made that no notion produces ε. Since this plays a role only for *parsers* based on operator precedence, and not for translators or transducers, this restriction is dropped here.

In order to construct a transducer for source texts with a correct parenthesis skeleton, it is sufficient to extend the precedence relations to all pairs of symbols σ, τ between which no precedence relation is defined, with the exception of those pairs where σ is an opener or a middler and τ is a middler or a closer. (A confrontation between such symbols with undefined precedence relation can occur only in incorrectly parenthesized texts.) This must be done by taking $\lessdot$ if σ is an opener or a middler, and $\gtrdot$ if τ is a middler or closer. In the other cases (σ is a closer or operator and τ is an opener or operator), we have a free choice between $\lessdot$ and $\gtrdot$. For correct input texts, this transducer is equivalent to the translator: only the original precedence relations are used. The output texts are described by the *synchronized prefix grammar* G_p obtained from G by replacing each production $\alpha_0\ \tau_1\ \alpha_1 \ \cdots\ \alpha_{n-1}\ \tau_n\ \alpha_n$ of G with $n \geq 1$ by $\sigma_{\tau_1 \ldots \tau_n}\ \alpha_0 \perp \alpha_1 \ \cdots\ \alpha_{n-1} \perp \alpha_n$, where $\sigma_{\tau_1 \ldots \tau_n}$ is a symbol uniquely determined by $\tau_1 \ldots \tau_n$.

With some luck, this grammar G_p is of type LL(1) straightaway. Otherwise, it may be necessary to "identify" some notions by replacing them by one and the same notion and by unifying the corresponding productions (with the same initial σ). The best way to do this is to modify the original grammar G.

4.2. *Resynchronization*

The grammar G_p, in this form, may be used to implement the error-recovery method described before. However, instead of simply skipping in order to resynchronize at a synchro, we prefer an attempt to parse the piece of source text concerned. This can be described by the addition of error-production rules to G_p. It is, however, more easily described in an informal way: The first situation in which the parser can get stuck is that the next input symbol is a synchro but is not yet expected. In this case, a direct production "missing" is added to the node being treated, the node is considered treated and parsing may continue. (Note that, in this case, the node cannot be labeled with a symbol! For all symbols other than a synchro occur only at the very start of a production, and if that symbol were not present on the input text, then that production would not have been selected.) In all other situations, some notion is chosen such that a selec-

tion of a production would be possible (there is always at least one such a notion), and an extra error node, labeled with that notion, is inserted in the tree just in front of the node where failure occurred. Parsing is now resumed, starting at that error node.

5. AN EXAMPLE

Before discussing the specific difficulties encountered when this technique is applied to ALGOL 68, it will be demonstrated on a simple example which contains, in a nutshell, the essential problems. Two remarks, however, must first be made:

(i) At those places where G is already "locally" LL(1), there is no need to bring the productions in prefix form. The transducer may be instructed then to leave the corresponding symbols where they are. In that case, the middlers and closers of these productions must be explicitly used as synchronization points.

(ii) The freedom of choice, left by the undefined precedence relations, should be used to obtain a completion which is as consistent as possible with the relations already defined. If the operators can be ordered according to priority, this gives a natural way to define the completion.

Consider the following grammar, in which all symbols end with *token* ("basic token" stands for some recognizable basic item, e.g., an identifier or a denotation, contracted by the lexical scan to one symbol):

```
unit: tertiary, becomes token, unit; tertiary.
tertiary: tertiary, plus token, term; term.
term: term, times token, factor; factor.
factor: plus token, factor; primary.
primary: primary, actual parameter pack;
        open token, unit, close token; basic token.
actual parameter pack: open token, unit, close token.
```

The language described by this grammar contains, in ascending order of priority, constructions resembling *assignations*, *formulas* (with dyadic operators + and × and a monadic operator +), *calls*, *closed clauses* and some basic item.

On inspection, it turns out that this grammar is not operator precedence: there are clashes of precedence caused by the monadic and the dyadic +. Therefore the lexical scan has to distinguish between these, and to replace a monadic + by a new symbol ⊕ ("monadic plus token"). This is possible, since a monadic + is either the first symbol of the source text or is preceded by a ×, +, := or (, whereas a dyadic + is preceded by a) or some basic item.

Also, a production "primary, actual parameter pack" may not occur in an operator-precedence grammar: two notions have to be separated by at least one symbol.* Therefore, the lexical scan has to insert a call operator © ("call insert") between the primary and the actual-parameter-pack of a call. This situation can be recognized by the occurrence of a) or basic item followed by a (.

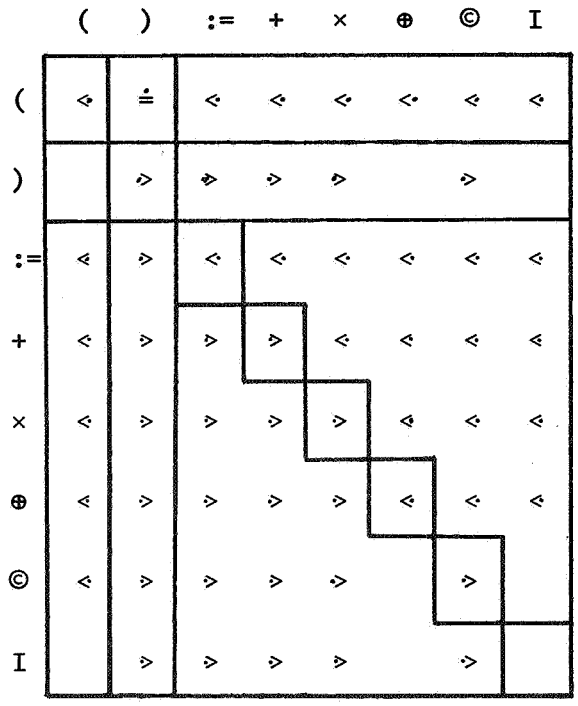
We now have the following grammar:

```
unit: tertiary, becomes token, unit; tertiary.
tertiary: tertiary, plus token, term; term.
term: term, times token, factor; factor.
factor: monadic plus token, factor; primary.
primary: primary, call insert, actual parameter pack;
        open token, unit, close token; basic token.
actual parameter pack: open token, unit, close token.
```

---

* Note that the notion "actual parameter pack" may not be replaced by its production: the prefix grammar thus obtained would not be of type LL(1). Of course, the difficulty can be circumvented by writing the first alternative of "primary" as "basic token, actual parameter pack sequence". But then the structure, which might be needed for semantic purposes, is fully lost.

This is an operator-parenthesis grammar. The precedence relations are given in the following table:



(I stands for "basic token")

The operators in this table are arranged in such a way as to show clearly the fact that they can be ordered according to their priority: the lower left triangle of the operator part of the table contains only relations $\gtrdot$ and the upper right only $\lessdot$.

The prefix grammar corresponding to the above grammar (where only those productions are brought in prefix form which are not already locally LL(1)) is given by

unit: becomes token, tertiary, synchro, unit; tertiary.
tertiary: plus token, tertiary, synchro, term; term.
term: times token, term, synchro, factor; factor.
factor: monadic plus token, factor; primary.
primary: call insert, primary, synchro, actual parameter pack;
open token, unit, close token; basic token.
actual parameter pack: open token, unit, close token.

6. APPLICATION TO ALGOL 68

6.1 *Making ALGOL 68 operator precedence*

Our first step was the construction of a context-free grammar for ALGOL 68. Such a grammar must, of necessity, describe more texts than are proper ALGOL 68. For example, the rule

MODE NEST source: strong MODE NEST unit.

was simplified to

source: unit.

Apart from this type of departure from ALGOL 68, the main differences are:

- in a series, declarations are allowed even if label-definitions have already occurred;
- after *exit*, no label-definition is required;
- an enquiry-clause is treated as a series;
- in choice-clauses, the CHOICE is disregarded;
- in case-clauses, units and united-case-parts may be mixed;
- identity- and variable-definitions are unified, so that *int a = 1, b:= 2* is accepted;
- in routine- and operation-declarations without procedure-plan, the source need not be a routine-text;
- in operation-declarations, an arbitrary number of parameters in the plan or the routine-text is accepted;
- VICTAL is disregarded;
- skips, nihils and jumps are treated as primaries;
- slices and calls are unified.

Consequently, errors linked up with these departures have to be detected separately. With the exception of the last case, and of the use of a goto-less jump in a position where a TERTIARY is required, this can be done mode-independently.

These differences tend to simplify the context-free grammar. An important consideration for incorporating them, however, was a psychological one:

we anticipate that certain errors will be perceived as belonging to the realm of "static semantics", rather than to that of syntax. In such cases, specialized error messages seem in order.

Other departures stem from the fact that the source text has already been submitted to a lexical scan (so that comments are no longer present and denoters are considered one symbol) and that, if necessary, the parenthesis skeleton has been repaired (so that it is no longer necessary to require that corresponding parentheses have matching STYLES). The lexical scan also inserts a symbol loop-insert to mark the start of a loop-clause.

The grammar thus obtained is not operator precedence. By submitting it to a mechanical operator-precedence checker, the trouble spots can be found. 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. It was necessary to distinguish between the equals-token and the is-defined-as-token, between the up-to-/label-token, the specification-token and the routine-token, and between the use of the and-also-token to separate COMMON-declarations or FIELDS-portrayers or PARAMETERS-joined-declarers (the "separate-and-also-token") and other uses.

c. Inserting symbols between notions. These inserts are:

   - dectag-insert, between a declarer and the following TAG-token in an identifier-declaration;
   - opdec-insert, between the operation-heading and the following operator in an operation-declaration;
   - cast-insert, between the declarer and the ENCLOSED-clause of a cast;
   - clice-insert, between the primary and the actual-parameters-pack or indexer-bracket of a call or slice;
   - row-insert, between the ROWS-rower-bracket and the following declarer of a ROWS-of-MODE-declarator;

- formals-insert, between a PARAMETERS-joined-declarer-brief-pack or declarative-brief-pack and the following declarer of a procedure-plan or routine-text.

(The function of the changes in categories a and c is to separate any two notions in a production by at least one symbol, whereas category b serves to resolve clashes in the precedence relations.) The grammar obtained in this way is operator precedence. It is given in Appendix A. From the table of precedence relations (Appendix B), it may be seen that the operators can be ordered according to priority. This means that the prefix transducer need not know the full table, but only the priorities of the operators and their left or right associativity (as indicated by the diagonal). A short survey of this is given in Appendix C.

The construction of the prefix version of the grammar was performed mechanically. For this, no knowledge of priorities is needed, but only knowledge of which operators are moved left and which are left unmoved, which is indicated in Appendix A by marking the operators to be moved with a <. The result is given in Appendix D. A program [12] was used to check this grammar for LL(1)-ness. After the first attempt, only three changes had to be made - two rather trivial, and one nontrivial one: the unification of identity- and variable-declarations.

6.2. *Adjusting the source text accordingly*

The task of making the distinctions of category b or of placing the inserts of category c is part of the duties of the lexical scan. We will not thresh out all problems involved, but only touch upon some general ones. Since the lexical scan is not yet able to parse the source text and has to be able to cope with garbage as well, the decisions are made on the basis of as local as possible information, similar to the way monadic and dyadic + were distinguished and call-inserts were placed in the simple example of Section 5. It is of the utmost importance that mechanical analysis of the grammar (as in [13]) lies at the root of these decisions, since, otherwise, some bizarre but perfectly legal case might easily be overlooked. In some cases information has to be taken into account which is not of a purely local nature. To this purpose, the lexical scan functions as a stack auto-

maton, where the depth of the stack corresponds to the depth of nesting of parenthesized constructs. Within each level, this automaton is a finite-state one.*

A complication is given by the fact that many decisions depend on the distinction between the TAB-tokens which are mode-indications and those which are operators. This distinction can only be made after the completion of the lexical scan, which collects the relevant information from the declarations. The solution to this problem is that the lexical scan, in such cases, places provisional inserts in the output text, which contain a pointer to the TAB-token which determines the actual insert. At the input side of the "backward scan", which performs the prefix transduction, a preprocessor replaces the provisional inserts by actual ones or discards them, as the case requires. This problem is the most complicated when a number of "packs" (parenthesized constructions) follow each other immediately. Consider, e.g., the following text:

```
; (p a) (b) (c) q d;
     ↑   ↑   ↑
     α   β   γ
```

Here, α, β and γ stand for inserts to be placed. Depending on whether *p* and *q* are mode-indications or operators, we have the following possibilities:

- *p* and *q* are mode-indications: α = formals-insert, $\beta = \gamma$ = row-insert;
- *p* is a mode-indication and *q* an operator: this situation is erroneous;
- *p* is an operator and *q* a mode-indication: $\alpha = \beta = \gamma$ = row-insert;
- *p* and *q* are operators: $\alpha = \beta$ = clice-insert, $\gamma = \varepsilon$.

It should be obvious from the above that arbitrarily difficult situations may be constructed.

---

* As may be seen from the grammar in Appendix A, we treat a format-text as a symbol. Actually, the grammar of format-texts has been submitted to a similar process (we had, however, to violate the structure of format-texts more seriously in order to make it meet the requirements). Upon encountering a formatter-symbol, we simply activate a finite-state automaton which is different from the "normal" one.

In such cases, the (actual) inserts after subsequent packs depend on the inserts after previous packs, as determined by some finite-state algorithm. Since the lexical scan does not yet know the actual inserts, but only provisional ones, it cannot perform this algorithm, and has to leave this task to the preprocessor of the backward scan too. But this scan would encounter the provisional inserts in the wrong order. Therefore, in the case of a sequence of packs, the provisional inserts are placed, in reverse order, after the last pack. The preprocessor, in performing the finite-state algorithm, puts the actual inserts on a stack to drop them between the packs as and when required.

There is yet another insert which may be placed by the lexical scan or by the preprocessor of the backward scan. This insert serves to solve a psychological problem which would otherwise arise with the error-recovery technique described here. 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 parser 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 gap-insert with relatively low priority is placed in general between something which looks, roughly speaking, like the end of a coherent chunk and something which looks like the start of one, unless another insert has already been placed there (in fact, 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 backward scan.

6.3. *Actual parsing*

As stated in the introduction, we use ALEPH as our implementation language. Two important kinds of "procedures" (called "rules") in ALEPH are: "predicates" and "actions". The difference between these is that a predicate can "fail", while an action cannot. Both succeeding predicates and actions have "side-effects". (In our case: they both read something from the input text and, possibly, perform some semantic action.) Now, there is a trivial way of transforming an LL(1) grammar (such as the one given in Appendix D)

into an ALEPH program accepting the language described by that grammar:

- write, for each symbol, a predicate labeled with that symbol that succeeds if the next input symbol is equal to that symbol (and then advances the input over one symbol), and fails otherwise;
- write, for each notion, a predicate labeled with that notion and whose right-hand side consists of the direct productions of that notion. E.g., the first rule from our example simply becomes:

```
'predicate' unit: becomes token, tertiary, synchro, unit;
                  tertiary.
```

The ALEPH program thus obtained may be used to determine all places where error productions must be added, since the ALEPH compiler has the nice property that it tests for "backtrack". If, in some right-hand side of a rule, two predicates follow each other, the first one may succeed while the second one fails; in such a case the side-effects of the first (succeeding) predicate would have to be undone. As this is likely to be impossible, a warning is given. In such a case, however, *we* want to add an error production (for, the grammar being LL(1) and thus backtrack-free, the warning indicates that we may somehow get stuck in this rule). In the above example: if a becomes token is read and the rule for tertiary may somehow fail, we want some error production to skip the garbage until the synchro is met, upon which we are back on the track again. In this way, becomes token will be followed by an action: either a tertiary is read, or an error production takes care of the garbage. Thus, we are supplied with a clean mechanical aid in adding a (minimal) number of error productions so as to complete our grammar. As a side-effect of this, the framework of the mode-independent scan results.

As has been stated before, we prefer an attempt to parse the piece of source text which otherwise would be skipped to find the expected synchro. Otherwise, very large pieces of source text, if not virtually the whole program, might just be skipped. Worse yet, skipping a mode-declaration might give rise to many undesirable error messages.

If the input symbol on which the parser gets stuck is a symbol which

may be the start of a unit, a declarer, or a declaration (these three notions have disjoint sets of possible starting symbols), that notion is chosen to label the inserted error node. Other symbols, except, of course, synchros, are skipped (and an error message is given).* If the unexpected symbol is [, it may not be skipped, since this would upset the balance of parentheses (this being the only possible case of an unexpected parenthesis). For this special case, a special error production has been added for unit, and the error message "primary of slice missing" is given.

REFERENCES

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

[2] MEERTENS, L.G.L.T. and J.C. VAN VLIET, *Repairing the State Switcher Skeleton of ALGOL 68 programs*, Report IW 15/74, Mathematisch Centrum, Amsterdam, 1974.

[3] MEERTENS, L.G.L.T. and J.C. VAN VLIET, *Repairing the parenthesis skeleton of ALGOL 68 programs: Proof of correctness*, Report IW52/75, Mathematisch Centrum, Amsterdam, 1975.

[4] BOSCH, R., D. GRUNE and L.G.L.T. MEERTENS, *ALEPH, A Language Encouraging Program Hierarchy*, in A. Günther et al. (eds.) International Computing Symposium 1973, North-Holland Publ. Co., Amsterdam, 1974.

[5] GRUNE, D., R. BOSCH and L.G.L.T. MEERTENS, *ALEPH Manual*, Report IW 17/74, Mathematisch Centrum, Amsterdam, 1974.

---

* Care has to be taken if that symbol is an operator which has been moved to a prefix position, as in *begin* y := @ *end*, which has been changed to *begin* @ := y ⊥ ⊥ *end*. Here, parsing gets stuck on the @, but to the human reader, the parser is only at the y. In such cases, the unexpected symbol is stacked to be complained about at the position of its corresponding synchro - but only, of course, if it was not inserted by the lexical scan; in that case some other error message will demonstrably be given already anyway.

[6] KNUTH, D.E., *Top-down syntax analysis*, Acta Informatica, Vol.1, no.2 (1971) 79-110.

[7] KOSTER, C.H.A., *Affix-grammars*, in: J.E.L. Peck (ed.) ALGOL 68 Implementation, North-Holland Publ.Co., Amsterdam, 1971.

[8] LEWIS II, P.M. and R.E. STEARNS, *Syntax-directed transduction*, Journal of the ACM, Vol.15, no.3 (1968) 465-488.

[9] GRIES, D., *Compiler Construction for Digital Computers*, John Wiley, New York, 1971.

[10] FLOYD, R.W., *Syntactic analysis and operator precedence*, Journal of the ACM Vol.10, no.3 (1963) 316-334.

[11] DIJKSTRA, E.W., *Making a translator for ALGOL 60*, in: R. Goodman (ed.) Annual Review in Automatic Programming, 3, Pergamon Press, Oxford, 1963. (First published in 1961.)

[12] VLIET, J.C. VAN, *The Programs "Relations Concerning a CF-Grammar" and "LL(1)-checker"*, Report IN 4/73, Mathematisch Centrum, Amsterdam, 1973.

[13] GRUNE, D., L.G.L.T. MEERTENS and J.C. VAN VLIET, *Grammar-handling Tools Applied to ALGOL 68*, Report IW 5/73, Mathematisch Centrum, Amsterdam, 1973.

In Appendix A, the operator-precedence grammar of ALGOL 68 is given. D. Grune has done invaluable work in bringing the grammar into its present form. First, a list of all symbols is given, separated by semicolons, the last one followed by a period. If a symbol in this list is preceded by a <, this means that it has to be moved to the left by the prefix transducer. Then, for each notion, a production rule is given by writing, in order, that notion, followed by a colon, followed by the various direct productions of that notion, separated by semicolons, followed by a period. The members of a direct production are separated by commas. Optional parts are enclosed between (and). If the rule for a notion is preceded by a *, it has to be treated as a macro, to be replaced (virtually) in the productions in which it occurs by its direct productions. The grammar is interspersed with comments, written between the symbols [and].

Appendix B lists the precedence relations of the operator-precedence grammar.

Appendix C gives a short account of the proper operators and their left/right associativity.

Appendix D contains the synchronized prefix grammar obtained from the grammar in Appendix A.

APPENDIX A

```
[ opgram, 18-11-75 ]
[9. tokens and symbols.]        [ representation. alternatives are      ]
                                [ separated by spaces. inserts start    ]
                                [ with an apostrophe. three dots        ]
                                [ indicate that only examples are given.]

[ enclosure tokens ]

    open mark;                          [ ( ]
    bold begin token;                   [ .begin ]
    big begin token;                    [ `begin ]
    choice start;                       [ .if .case ( ]
    brief sub token;                    [ [ ]
    loop insert;                        [ `loop ]

    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 ]

    close mark;                         [ ) ]
    bold end token;                     [ .end ]
    big end token;                      [ `end ]
    choice finish;                      [ .fi .esac ) ]
    brief bus token;                    [ ]
    od token;                           [ .od ]


[ priority a tokens ]
    completion token;                   [ .exit ]

[ priority b tokens ]
<   go on token;                        [ ; ]

[ priority c tokens ]
    separate and also token;            [ `sep ]

[ priority d tokens ]
    priority token;                     [ .prio ]
    mode token;                         [ .mode ]

[ priority e tokens ]
<   dectag insert;                      [ `dectag ]
<   opdec insert;                       [ `opdec ]

[ priority f tokens ]
<   and also token;                     [ , ]
```

```
[ priority g tokens ]
<   is defined as token;              [ `idat ]
<   at token;                         [ @ .at ]

[ priority h tokens ]
<   colon mark;                       [ : ]
<   specification token;              [ `spec ]

[ priority i tokens ]
<   becomes token;                    [ := ]
<   identity relator;                 [ :=: .is :≠: :/=: .isnt ]
<   routine token;                    [ `rout ]

[ priority j tokens ]
<   dyadic operator;                  [ +:= .over .xyz ... ]

[ priority k tokens ]
    monadic operator;                 [ + .not .xyz ... ]

[ priority L tokens ]
<   of token;                         [ .of ]

[ priority m tokens ]
<   cast insert;                      [ `cast ]
<   clice insert;                     [ `clice ]

[ priority n 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 ]

[ operands ]
    digit token;                      [ 1 2 3 4 5 6 7 8 9 ]
    tag token;                        [ i ... ]
    parallel token;                   [ .par ]
    format text;                      [ $3zd$ ... ]
    string denoter;                   [ "string" ... ]
    other denoter;                    [ 3.14 .true .empty ... ]
    defining operator;                [ +:= .over .xyz ... ]
    mode indication;                  [ .int ... ]
    skip token;                       [ .skip #tilde ]
    nil token.                        [ .nil #circle ]
```

[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.
```

[10.1.1. program text.]

```
particular program:
    big begin token, lenclosed clause, big end token.
lenclosed clause:
    label definition, lenclosed clause; enclosed clause.
```

[3. clauses.]

```
enclosed clause:
    closed or collateral clause; choice clause; loop clause.
```

[3.1. closed clauses.]

```
closed or collateral clause:
    (parallel token), begin, inner clause, end.
* begin:
    bold begin token; brief begin token.
* end:
    bold end token; brief end token.
inner clause:
    serial clause;
    (joined portrait).
```

[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.


[4. declarations.]

declaration:
    common declaration, (separate and also token, declaration).
common declaration:
    mode declaration; priority declaration;
    identifier declaration; operation 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.
defined mode indication:
    mode indication.

[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, unit.
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, is defined as token, unit.
operator:
    defining operator.

[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; 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.

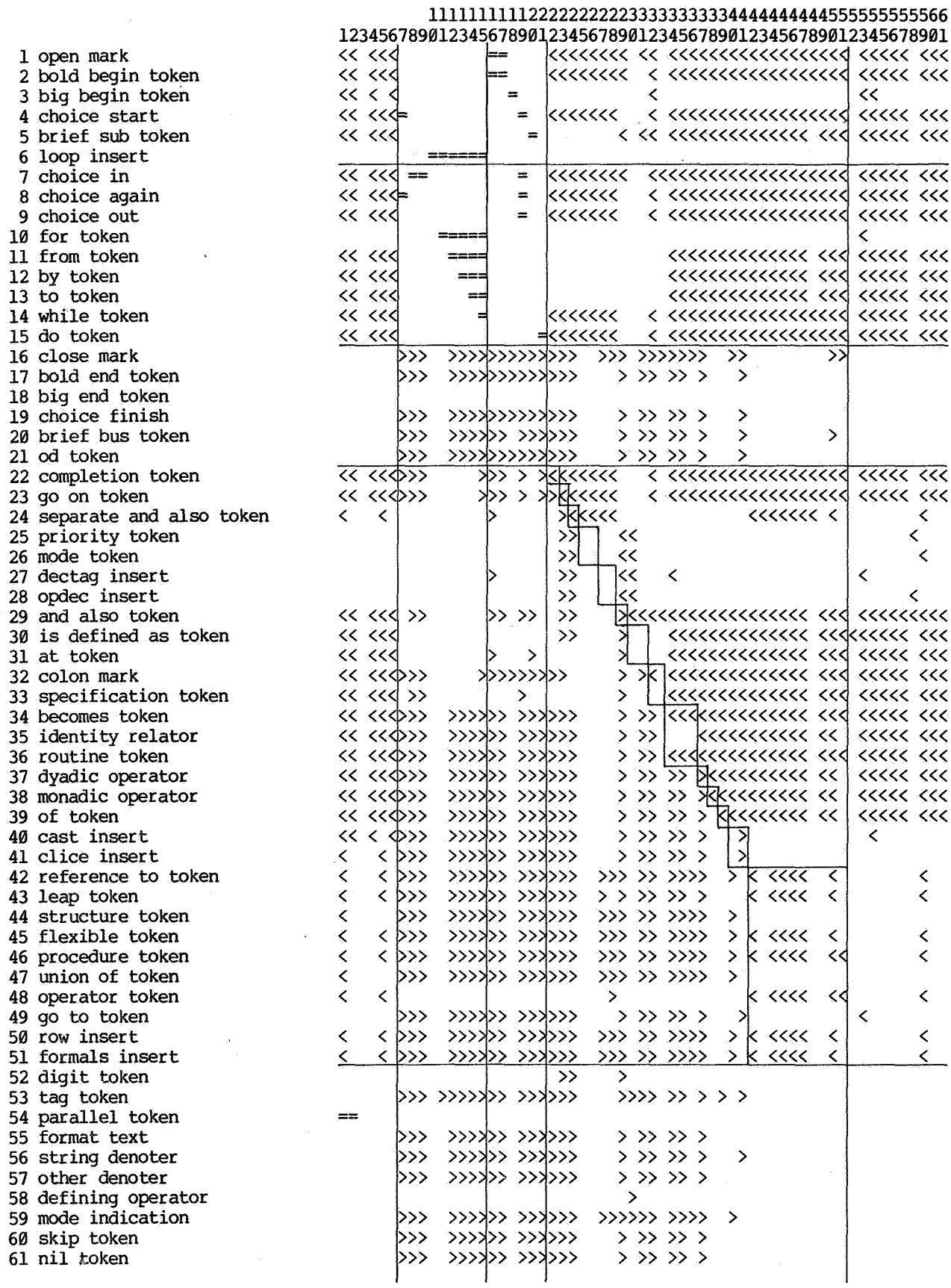
[5.4.3. calls.] [see 5.3.2.]

[5.4.4. jumps.]

jump:
    [()]go to token[()], identifier.

[5.5.1. casts.]

cast:
    declarer, cast insert, enclosed clause.


1 open mark
2 bold begin token
3 big begin token
4 choice start
5 brief sub token
6 loop insert
7 choice in
8 choice again
9 choice out
10 for token
11 from token
12 by token
13 to token
14 while token
15 do token
16 close mark
17 bold end token
18 big end token
19 choice finish
20 brief bus token
21 od token
22 completion token
23 go on token
24 separate and also token
25 priority token
26 mode token
27 dectag insert
28 opdec insert
29 and also token
30 is defined as token
31 at token
32 colon mark
33 specification token
34 becomes token
35 identity relator
36 routine token
37 dyadic operator
38 monadic operator
39 of token
40 cast insert
41 clice insert
42 reference to token
43 leap token
44 structure token
45 flexible token
46 procedure token
47 union of token
48 operator token
49 go to token
50 row insert
51 formals insert
52 digit token
53 tag token
54 parallel token
55 format text
56 string denoter
57 other denoter
58 defining operator
59 mode indication
60 skip token
61 nil token

APPENDIX C

Proper operators

Type: M = monadic, L = left associative, R = right associative.
* indicates "no prefix transduction".

Type	prio	operator
R	140	row insert; formals insert.
M	130	reference to token; leap token; structure token; flexible token; procedure token; union of token; operator token; go to token.
L	120	cast insert; clice insert.
R	110	of token.
M	100	monadic operator.
L	091:099	dyadic operator.
R	080	becomes token; identity relator; routine token.
R	070	colon mark; specification token.
R	060	is defined as token; at token.
L	050	and also token.
R	040	dectag insert; opdec insert.
M	030	priority token; mode token.
R*	020	separate and also token.
R	010	go on token.
R*	000	completion token.

[LL1-grammar, 18-11-75]

```
open mark;
bold begin token;
big begin token;
choice start;
brief sub token;
loop insert;
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;
completion token;
go on token;
separate and also token;
priority token;
mode 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;
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;
digit token;
tag token;
parallel token;
format text;
string denoter;
other denoter;
defining operator;
mode indication;
skip token;
nil token;
synchro.

brief begin token:
    open mark.
brief end token:
    close mark.
style i sub token:
    open mark.
style i bus token:
    close mark.
particular program:
    big begin token, lenclosed clause, big end token.
lenclosed clause:
    colon mark, identifier, synchro, lenclosed clause;
    enclosed clause.
enclosed clause:
    closed or collateral clause;
    choice clause;
    loop clause.
closed or collateral clause:
    (parallel token), begin, inner clause, end.
begin:
    bold begin token;
    brief begin token.
end:
    bold end token;
    brief end token.
inner clause:
    serial clause;
    (joined portrait).
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.
declaration:
    common declaration, (separate and also token, declaration).
common declaration:
    mode declaration;
    priority declaration;
    identifier declaration;
    operation 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.
defined mode indication:
    mode indication.
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, unit.
```

```
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, synchro, unit.
operator:
    defining operator.
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;
    tertiary.
tertiary:
    formula;
    secondary.
secndary:
    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.
```