IA

stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IN 6/73 DECEMBER

Lambert MEERTENS & Hans van VLIET

A SYNTAX-DIRECTED ERROR-RECOVERY METHOD FOR PARSING ALGOL 68 PROGRAMS

Prepublication

IA

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.

ACM - Computing Reviews - category: 4.22, 5.23

A syntax-directed error-recovery method for parsing ALGOL 68 programs [*)]

CONTENTS

*) This paper has been submitted for the IFIP Congress 74 at Stockholm.

ABSTRACT

The generality of ALGOL 68 makes it difficult to obtain good error recovery when the traditional top-down error-recovery method is applied. With the help of a simplified example, an error-recovery technique is sketched, relying on the fact that there is an algorithm for detecting incorrect parenthesis skeletons. The source text is brought into prefix form and an *LL*(1) grammar is constructed such that

(i) the prefix form of each source text satisfying certain restrictions on the parenthesis skeleton can be parsed according to that grammar;

(ii) the number of places in the grammar where resynchronization takes place is considerably enlarged;

(iii) the syntactical errors, although produced by the grammar, are easily identifiable;

(iv) the parse tree obtained for the prefix form of a correct source text is similar in structure to the prefix form of the ALGOL-68 parse tree for that source text;

(v) the resulting grammar is considerably simpler than the original grammar.

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, e.g., in ALGOL 60. Investigating this problem we concluded that good resynchronization of the parser is only then possible if it is known beforehand which opening parentheses are, and which are not, accompanied by a matching closing parenthesis (and vice versa). 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 detected before the source text is parsed. This error detection is treated in detail in [2]. 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 [3], which is particularly suited for top-down parsing according to a grammar of type *LL*(1) [4]. The context-free grammar *underlying* [5] 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 [6], operating from right to left, which brings the source text in prefix form. If, however, this transduction scheme is applied bluntly to a source text with an incorrect parenthesis skeleton, the result is in general unacceptable. The advantage of having first detected errors in the parenthesis skeleton is that the transduction scheme can be constructed in such a way as to yield acceptable results even for improperly parenthesized programs.

(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 in an informal way by applying it to a simplified example.

2. CONVENTIONS, TERMINOLOGY AND DEFINITIONS

We shall refrain from giving a formal definition of well established concepts as "context-free grammar", the "language produced by" a grammar, "parse tree", etc. Instead, we shall introduce our conventions in an informal way.

Nonterminals will be denoted by capital letters (A,B,C,...).

A *terminal production* of a nonterminal will be denoted by the corre-

sponding small letter ($a,b,c,\ldots$).
Terminals will be denoted by their representation ($+,-,\times,\ldots$).
The *empty string* will be denoted by ε.
Other small Greek letters ($\alpha,\beta,\gamma,\ldots$) will be used to denote some -possibly empty- sequence of terminals and nonterminals.
A *production rule* for a nonterminal is written by first giving that nonterminal, then a colon, then the alternative productions for that nonterminal, separated by semicolons, and then a point. E.g., T: $T + X$; X. means that there are two productions for T, $T + X$ and X.
The *start nonterminal* of a grammar will be the nonterminal whose production rule is given first.
As usual, $\alpha \rightarrow \beta$ means that β can be *directly derived* from α by replacing one nonterminal in α by one of its productions, and $\alpha \rightarrow^* \beta$ means that β can be *derived* from α by zero or more derivation steps:
$\alpha = \gamma_0 \rightarrow \gamma_1 \rightarrow \ldots \rightarrow \gamma_n = \beta$, $n \geq 0$.

In this paper, the notions "*LR*(k) grammar" and "*LL*(k) grammar" are used. For a definition, we refer to [4] and [7]. 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 string produced by that grammar in the following way: Start with a partial parse tree consisting of only one (top) node, labelled with the start nonterminal, and that string. 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 labelled with a nonterminal a number of (untreated) descendants, one for each terminal and

nonterminal of one of its productions. At the same time, the string will be *accepted* by deleting from left to right successive terminals. 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 labelled with a nonterminal, select on the basis of that nonterminal and the first k terminals of the string, a production for that nonterminal and develop the parse tree accordingly. (The selection is uniquely determined for an *LL*(k) grammar.) If that node is labelled with a terminal, it is equal to the first terminal of the remaining string (this is a property of the selection procedure for *LL*(k) grammars), and that first terminal is deleted.

If the string was indeed produced by the given grammar, this parsing process will terminate with a complete parse tree (all nodes treated and all bottom nodes terminal) and an empty remaining string. Otherwise, the process terminates with a non-empty remaining string 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 nonterminal. During the parsing process, the untreated part of the tree is reflected in the status of the link stack. For *LR*(k) grammars, there exists a deterministic *bottom-up* parsing method, in which the construction of the parse tree is performed in the opposite direction.

Since for a grammar of type *LL*(k) or *LR*(k) there exists a deterministic parsing method, this implies that for any string produced by that grammar there exists only one parse tree; in other words, such a grammar is *unambiguous*.

Now we need some precise definitions.

Two parse trees T_1 and T_2 are *congruent* if they are the same, apart from the labelling of nodes by nonterminals. An example of two congruent trees is given by T_{7c} and T_{9c} in fig. 1.

A parse tree T_c is the *contracted form* of a parse tree T if T_c may be obtained from T by replacing any nodes with one single descendant by that descendant itself, by deleting any nodes with no descendants, and by repeating this process until no such nodes are left. In fig. 1, T_{7c} and T_{9c} are the contracted forms of T_7 and T_9, respectively.

Two parse trees T_1 and T_2 are *similar* if their contracted forms are congruent. An example is given by T_7 and T_9 in fig. 1. (In fact, all four trees shown are similar, since similarity between trees is an equivalence relation and a tree is always similar to its contracted form.)

A grammar G_e is a *structurally similar extension* of a grammar G if, for each string s produced by G, with parse tree T, there exists a parse tree T_e for s according to G_e such that T and T_e are similar.

Two grammars G_1 and G_2 are *structurally similar* if G_1 is a structurally similar extension of G_2, and vice versa.

Note that the language produced by a grammar G is a (not necessarily proper) subset of the language produced by any structurally similar extension G_e of G. Obviously, two structurally similar grammars are equivalent (in the weak sense that they produce the same language). The equivalence relation of structural similarity is stronger, however; it is closely related to the still stronger notion of *structural equivalence* introduced in [8]. The motivation for the introduction of this weaker form lies in the fact that for purposes of compilation, it is the contracted form rather than the full tree which is important; e.g., in the context-free grammar which we use for ALGOL 68, the appearance in the parse tree of nonterminals like *unit*, *tertiary*, etc., is an artefact, introduced by the step from a Van-Wijngaarden grammar as in [1] to the underlying context-free grammar.

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 practice" [4]. (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 labelled with a nonterminal: no selection is possible; for a node labelled with a terminal: it is not equal to the first terminal of the remaining string), proceed then upwards in the tree until a node is encountered, labelled with an "important" nonterminal, after which the whole tree descending from that node, including the node itself, is considered treated. Delete then successive terminals from the remaining string until a next step is possible. The parsing process may now be resumed. For ALGOL 60, an important nonterminal 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 simply to some resynchronizing terminal 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. A SIMPLE GRAMMAR AND ITS PREFIX FORM

As the syntax of ALGOL 68 is rather complicated, the error-recovery technique will be demonstrated with the help of a simplified grammar, which nevertheless reflects the major difficulties involved. Consider the following grammar, in which I stands for some recognizable basic item, e.g., an identifier or a denotation (the production rule for I is omitted; it will be treated as a terminal):

$$\begin{aligned} &U\text{: } T := U\text{; } T. \\ &T\text{: } T + X\text{; } X. \\ &X\text{: } X \times Y\text{; } Y. \qquad\qquad (1)\\ &Y\text{: } + Y\text{; } P. \\ &P\text{: } P(U)\text{; } (U)\text{; } I. \end{aligned}$$

The language generated by this grammar contains, in ascending order of priority, constructions resembling *assignations*, *formulas* (with dyadic operators + and × and a monadic operator +) , *routine calls*, *closed clauses* and some basic item. The nonterminals U, T and P correspond to the ALGOL-68 *unit*, *tertiary* and *primary*. If the becomes symbol := is considered an operator, and, moreover, an invisible operator ◎ is assumed between the primary P of a call and its parameters pack (U), the prefix form of (1) is given by:

$$\begin{aligned}
&U\colon := T \perp U;\ T.\\
&T\colon + T \perp X;\ X.\\
&X\colon \times X \perp Y;\ Y. \qquad\qquad (2)\\
&Y\colon \oplus Y;\ P.\\
&P\colon \circledcirc P \perp (U);\ (U);\ I.
\end{aligned}$$

In the above grammar, a *synchronization symbol*, denoted by $\perp$, has been introduced. It is superfluous here, but it will become essential in subsequent versions of the grammar. The sequence $+ \perp$ has been replaced by $\oplus$; it thus stands for the monadic + .

In [6] it is proved that any postfix syntax-directed translation of an $LR(k)$ grammar can be performed on a (deterministic) pushdown machine. Grammar (1) is of type $RL(2)$, which simply means that the grammar obtained from it by writing all productions back to front (as in U: U:= T; T.) is of type $LR(2)$ [7]. The mapping from (1) into (2) is a simple *prefix* syntax-directed translation scheme -apart from a trivial modification- so that the required translation can be performed by scanning from right to left. Grammar (2) is of type $LL(1)$, which was the purpose of bringing (1) into prefix form in the first place.

5. INTRODUCING ERROR-PRODUCTIONS

The pushdown machine introduced in the previous section is a translator from the language produced by (1) to the language produced by (2). We now want to extend this translator to all possible input strings.

The grammar

V: VB; ε.
B: := ; + ; × ; I; (V). (3)

generates all strings composed of the symbols of our language with the property that parentheses occur nested. (This restriction corresponds to the fact that errors in the parenthesis skeleton have been detected already at this stage.)

We now extend (1) in the following way: the strings produced at the basic level by P may also be ε (the empty string) or be followed by a sequence of I's.

U: T:= U; T.
T: T + X; X.
X: X × Y; Y.
Y: + Y; P. (4)
P: $P(U)Z$; $(U)Z$; IZ; ε.
Z: IZ; ε.

Clearly, (4) is a structurally similar extension of (1).
We shall prove that (3) and (4) are equivalent, i.e., all strings produced by U are produced by V, and vice versa. In order to simplify the proof, we make a trivial modification to (4) by replacing the last two rules by:

P: $P(U)Z$; $(U)Z$; Z.
Z: ZI; ε. (5)

Proof:

I. If $U \rightarrow^* u$ then $V \rightarrow^* u$. This part is obvious.

II. If $V \rightarrow^* v$ then $U \rightarrow^* v$.

The proof is by induction on the length of v, say n.

For $n = 0$, II is obvious, since $U \rightarrow^* \varepsilon$.

Otherwise, v can be written as $v'b$, $V \rightarrow^* v'$, $B \rightarrow^* b$, where, by the inductive hypothesis, $U \rightarrow^* v'$. We make a distinction by the first derivation step of $U \rightarrow^* v'$.

Case A. $U \rightarrow T{:=}\,U \rightarrow^* v'$. v' can be written as $t{:=}\,u$, with $T \rightarrow^* t$ and $U \rightarrow^* u$. Since $U \rightarrow^* u$, $V \rightarrow^* u$ and, consequently, $V \rightarrow^* ub$.

By the hypothesis, we have $U \rightarrow^* ub$.

We have: $U \rightarrow T{:=}\,U \rightarrow^* t{:=}\,ub = v'b = v$.

Case B. $U \rightarrow T \rightarrow^* v'$. Note that for any v' such that $T \rightarrow^* v'$ each of the following holds: $T \rightarrow \alpha X \rightarrow^* \alpha x = v'$; $T \rightarrow^* \beta P \rightarrow^* \beta p = v'$; $T \rightarrow^* \gamma Z \rightarrow^* \gamma z = v'$. Moreover, $U \rightarrow^* \varepsilon$; $X \rightarrow^* \varepsilon$; $Y \rightarrow^* \varepsilon$. We distinguish the various possibilities for b.

If $b = {:=}$, $U \rightarrow T{:=}\,U \rightarrow^* v'{:=}\,\varepsilon = v'b = v$.

If $b = +$, $U \rightarrow^* T + X \rightarrow^* v' + \varepsilon = v'b = v$.

If $b = \times$, $T \rightarrow \alpha X \rightarrow^* \alpha x = v'$, so $U \rightarrow T \rightarrow \alpha X \rightarrow \alpha X \times Y \rightarrow^* \alpha x \times \varepsilon = v'b = v$.

If $I \rightarrow^* b$, $T \rightarrow^* \gamma Z \rightarrow^* \gamma z = v'$, so $U \rightarrow T \rightarrow^* \gamma Z \rightarrow \gamma Z I \rightarrow^* \gamma z b = v'b = v$.

If $(V) \rightarrow^* b$, $(V) \rightarrow^* (v'') = b$, with $V \rightarrow^* v''$ and therefore, by the hypothesis, $U \rightarrow^* v''$. $T \rightarrow^* \beta P \rightarrow^* \beta p = v'$, so

$U \rightarrow T \rightarrow^* \beta P \rightarrow^* \beta P(U)Z \rightarrow^* \beta p(v'')\varepsilon = v'b = v$.

6. TRANSFORMING THE GRAMMAR

Grammar (4) is not of type $RL(k)$ for any k, since it is ambiguous, unlike

(3). E.g., + (I) is produced in four different ways. Therefore, we need a slight detour, in order to obtain a translator, similar to the one which maps (1) into (2), but which accepts all strings produced by (4): we rewrite (4) eliminating ambiguities (a subscript n for a nonterminal indicates that its terminal productions are of one of the forms α) or αI).

$$
\begin{aligned}
&U\text{: } T\text{:= } U\text{; } T.\\
&T\text{: } T_n + X\text{; } X.\\
&X\text{: } X \times Y\text{; } Y.\\
&Y\text{: } + Y\text{; } P.\\
&P\text{: } P_n(U)Z\text{; } (U)Z\text{; } IZ\text{; } \varepsilon.\\
&Z\text{: } IZ\text{; } \varepsilon.\\
&T_n\text{: } T_n + X_n\text{; } X_n.\\
&X_n\text{: } X \times Y_n\text{; } Y_n.\\
&Y_n\text{: } + Y_n\text{; } P_n.\\
&P_n\text{: } P_n(U)Z\text{; } (U)Z\text{; } IZ.
\end{aligned}
\qquad (6)
$$

The proof that (6) is equivalent to (3), and therefore to (4), runs largely parallel to the proof given above. Grammar (6) is of type $RL(2)$, just as (1); it is obvious that (6) is also a structurally similar extension of (1); the prefix form of this grammar may be constructed in the same way as for (1):

$$U\text{: } := T \perp U;\ T.$$

$$T\text{: } + T_n \perp X;\ X.$$

$$X\text{: } \times X \perp Y;\ Y.$$

$$Y\text{: } \oplus Y;\ P.$$

$$P\text{: } \circledcirc P_n \perp (U)Z;\ (U)Z;\ IZ;\ \varepsilon. \qquad (7)$$

$$Z\text{: } IZ;\ \varepsilon.$$

$$T_n\text{: } + T_n \perp X_n;\ X_n.$$

$$X_n\text{: } \times X \perp Y_n;\ Y_n.$$

$$Y_n\text{: } \oplus Y_n;\ P_n.$$

$$P_n\text{: } \circledcirc P_n \perp (U)Z;\ (U)Z;\ IZ.$$

It is easily verified that the translation scheme which maps (6) into (7) has the property that each string *s* produced by (1) is translated to a string whose parse tree according to (7) is similar to the parse tree according to (2) of the translation of *s* by the translator which maps (1) into (2). (So (7) is a structurally similar extension of (2).) Moreover, each string not produced by (1) is translated to a string which is not accepted by (2). E.g., + *I*:= *II* + is translated to the prefix form := $\circledcirc$ *I* $\perp$ + *II* $\perp$.

Since (7) is still of type *LL*(1), just like (2), we have, in a sense, reached our goal. However, extensive simplifications have come within reach. The important thing to notice is that we know beforehand that any string which will be submitted to be parsed according to our grammar, is produced by (7). What we need is a grammar such that it will produce, for strings accepted by (7), a similar parse tree; if it also accepts strings not accepted by (7), this will do no harm: no such strings will ever be submitted to it. As a first simplification, the step from (4) to (6) is, so to speak, undone:

$$\begin{array}{l}U\text{: } := T \perp U;\ T.\\ T\text{: } + T \perp X;\ X.\\ X\text{: } \times X \perp Y;\ Y.\\ Y\text{: } \oplus Y;\ P.\\ P\text{: } \circledcirc P \perp (U)Z;\ (U)Z;\ IZ;\ \varepsilon.\\ Z\text{: } IZ;\ \varepsilon.\end{array} \qquad (8)$$

Again, (8) is a structurally similar extension of (7); in other words, if a string can be parsed according to (7), then a similar parse tree according to (8) may be given. We must, however, exclude the possibility that, apart from this similar parse tree, another (not similar) parse tree might be found. That this is indeed excluded can be seen immediately by observing that (8) is still of type $LL(1)$, and therefore unambiguous.

Comparison of (8) and (2) shows that the possible errors have been concentrated in two spots in the grammar: if a string is accepted by (8) which is not produced by (2), then this must be the effect of having a Z from a production of P produce a non-empty string or of having P produce ε (or some combination of these). This can be expressed more strikingly by *factoring* the first three alternative productions of the production rule for P, so that we have P: QZ; ε. and Q: $\circledcirc P \perp (U)$; (U); I., thus obtaining a variant which is still a structurally similar extension of (2).

As a final step, we can "contract" this variant of (8):

$$
\begin{array}{ll}
U\colon & := U \perp U;\\
 & + \; U \perp U;\\
 & \times \; U \perp U;\\
 & \oplus \; U;\\
 & QZ;\ \varepsilon.\\
Q\colon & \circledcirc \; U \perp (U);\ (U);\ I.\\
Z\colon & IZ;\ \varepsilon.
\end{array} \tag{9}
$$

Grammar (9) is a structurally similar extension of the variant; as before, the fact that (9) accepts a wider class of strings is not harmful, as only strings accepted by (8) will be submitted for parsing, and the *LL*(1)-ness of (9) guarantees that the parse tree found will be similar to that obtained according to the variant, and, therefore, for strings produced by (2), similar to that obtained according to (7). Compared to (7), however, a not insignificant simplification has been achieved.

7. APPLICABILITY OF THE METHOD

It will be clear that the essence of the method sketched in the previous sections lies in the leaving behind of synchronization symbols at the places where the right-to-left scan picks up operators. It is interesting to note that the top-down error-recovery method sketched in section 3 may be viewed in the same light: if we consider a semicolon an operator, then the prefix form of *begin* *S1; S2; S3* *end*, e.g., is *begin* *;* *S1* $\perp$ *;* *S2* $\perp$ *S3* *end*, and we see how resynchronization may take place at semicolons. The same observation holds for closing parentheses, where resynchronization may take place also. Strictly speaking, the prefix form of (U) should be () $\perp$ U $\perp$, where () may be considered one new indivisible symbol. In these cases, however, nothing is gained by bringing the grammar in prefix form.

The question we ask now is: how generally applicable is this method? It appears that the answer is connected with the theory of *operator precedence grammars*. We shall give a partial answer in the form of a theorem (without proof), but first we need some definitions.

Let G be an operator precedence grammar (for a definition, see [10]). This implies that each production is of the form $\alpha_0\ \tau_1\ \alpha_1\ \ldots\ \alpha_{n-1}\ \tau_n\ \alpha_n$, $n \geq 1$, where each of the α_i is either ε or a nonterminal and where each of the τ_i is a terminal. 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, $2 \leq i \leq n - 1$, will be called a *middler*, and τ_n will be called a *closer*.

The *grammar with error productions* G_e of G is the grammar

V: VB; ε.

B: β_1; ...; β_n. ,

where to each production $\alpha_0\ \tau_1\ \alpha_1 \ldots\ \alpha_{n-1}\ \tau_n\ \alpha_n$ of G there corresponds one β_i of the form $\tau_1\ V \ldots\ V\ \tau_n$.

The *synchronized prefix grammar* G_p of G is obtained from G by replacing each production $\alpha_0\ \tau_1\ \alpha_1 \ldots\ \alpha_{n-1}\ \tau_n\ \alpha_n$ of G by $\sigma_{\tau_1 \ldots \tau_n}\ \alpha_0 \perp \alpha_1\ \ldots\ \alpha_{n-1} \perp \alpha_n$, where $\sigma_{\tau_1 \ldots \tau_n}$ is a terminal uniquely determined by $\tau_1 \ldots\ \tau_n$.

The *error-recovery prefix grammar* G_{pe} of G is the grammar obtained from G_p, with one nonterminal, say U, whose productions are obtained in the following steps:

1. For each production in G_p of the form $\sigma\ \alpha_0 \perp \alpha_1 \ldots\ \alpha_{n-1} \perp \alpha_n$, add a production $\sigma\ U \perp U \ldots\ U \perp U$.
2. Add a production ε.
3. If some production occurs twice or more, delete all occurrences but one.

The operator precedence relations for G determine a translation scheme for the language produced by G.

Theorem:

Let G be an operator precedence grammar and let G_e, G_p and G_{pe} be defined as above. If G satisfies the following restriction:

the operators, openers, middlers and closers of G form mutually disjoint sets,

then it is possible to extend the operator precedence relations for G, for all pairs of terminals 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, by taking for the precedence relation R between σ and τ one of the precedence relations $\lessdot$ and $\gtrdot$, where, if σ is an opener or middler, R must be $\lessdot$, if τ is a middler or closer, R must be $\gtrdot$, and if both σ and τ are operators, the choice between $\lessdot$ and $\gtrdot$ is free, and we have the following result:

Let S be the translation scheme determined by the extended operator precedence relations, and let $S(\alpha)$ denote the result of subjecting a string α to S. Then:

1. If v is produced by G_e, then $S(v)$ is produced by G_{pe}; moreover, if v is also produced by G, then $S(v)$ is equal to the corresponding v_p produced by G_p.
2. G_{pe} is a structurally similar extension of G_p.
3. G_{pe} is of type $LL(1)$.

The context-free grammar underlying ALGOL 68 is certainly not an operator precedence grammar. This holds even for our grammar (1). However, it appears that by small, inessential modifications, such as: recognizing certain basic constructions like identifiers beforehand, resolving precedence clashes by taking into account the immediate context, or separating nonterminals by inserting operators (also on the basis of the immediate context), the grammar may easily be transformed into an operator precedence grammar. The test whether the modified grammar satisfies the restriction of our theorem may be done mechanically. The error productions in the corresponding grammar G_{pe} are not immediately identifiable; this may be done by comparison of G_p and G_{pe}. A sensible way to fill in the empty precedence relations seems to be to make an attempt at consistency with the already defined relations.

REFERENCES

[1] A. van Wijngaarden e.a., Revised Report on the Algorithmic Language ALGOL 68, to be published.

[2] L.G.L.Th. Meertens, and J.C. van Vliet, Repairing the Parenthesis Skeleton of ALGOL 68 programs, report IW 2/73, Mathematical Centre, Amsterdam, February 1973.

[3] D. Grune e.a., ALEPH Manual, Mathematical Centre, Amsterdam, to be published.

[4] Donald E. Knuth, Top-Down Syntax Analysis, Acta Informatica, vol. 1, no. 2, 1971, 79-110.

[5] C.H.A.Koster, Affix-grammars, in ALGOL 68 Implementation, ed. J.E.L. Peck, North-Holland Publishing Co., Amsterdam, 1971.

[6] P.M. Lewis II, and R.E. Stearns, Syntax-Directed Transduction, Journal of the ACM, vol. 15, no. 3, July 1968, 465-488.

[7] Donald E. Knuth, On the Translation of Languages from Left to Right, Information and Control, vol. 8, no. 6, December 1965, 607-639.

[8] Marvin C. Paull, and Stephen H. Unger, Structural Equivalence of Context-Free Grammars, Journal of Computer and System Sciences, vol. 2, no. 4, December 1968, 427-463.

[9] David Gries, Compiler Construction for Digital Computers, John Wiley, New York, 1971.

[10]Robert W. Floyd, Syntactic Analysis and Operator Precedence, Journal of the ACM, vol. 10, no. 3, 1963, 316-334.

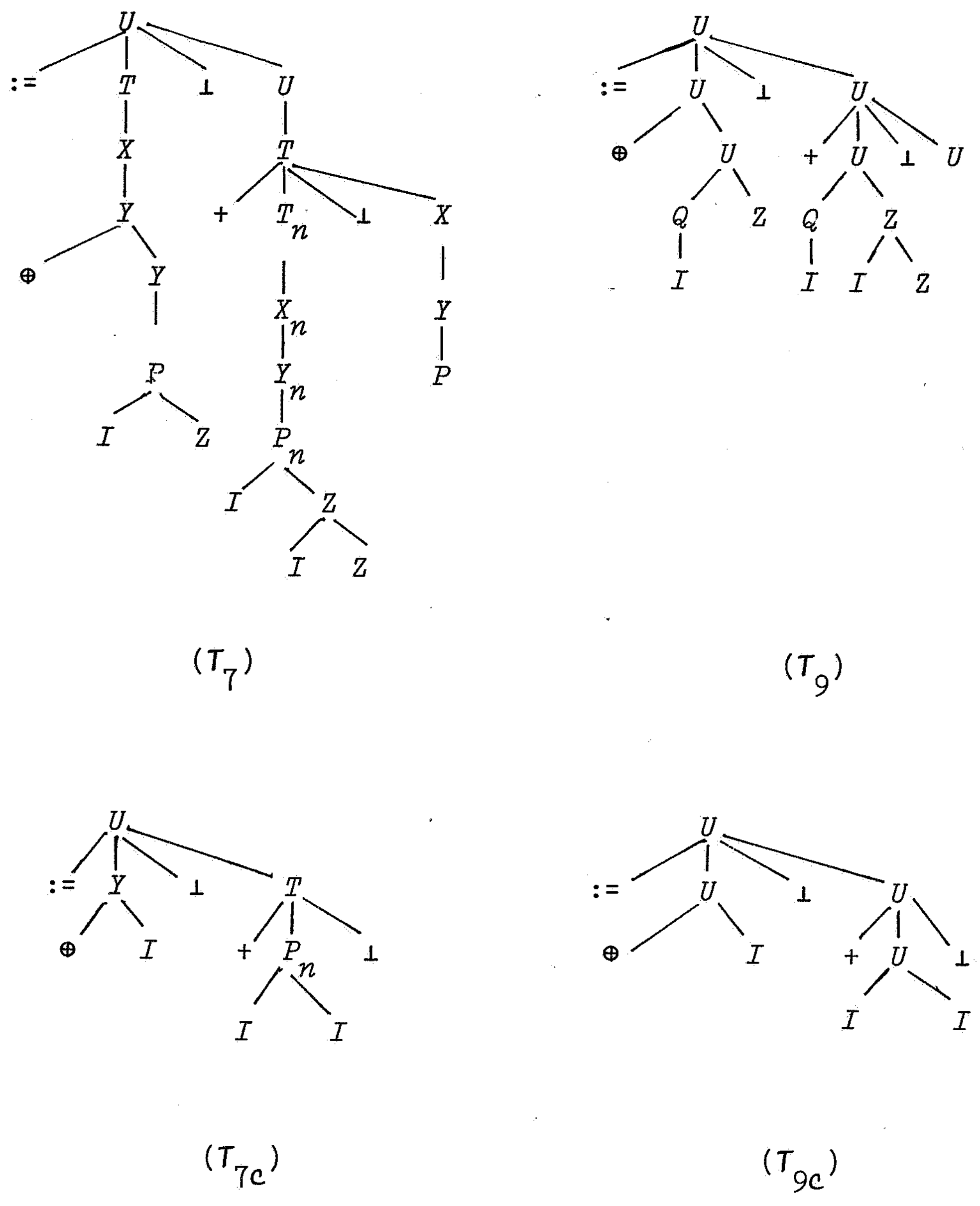

Fig. 1. Two parse trees, according to grammars (7) and (9), for the string := ⊕ I ⊥ + I I ⊥, and their contracted forms. Note that nodes labelled Z, P or U with no descendants correspond to a production ε.