Syntactic Analysis and Operator Precedence*

ROBERT W. FLOYD

Computer Associates, Inc., Woburn, Massachusetts

Abstract. Three increasingly restricted types of formal grammar are phrase structure grammars, operator grammars and precedence grammars. Precedence grammars form models of mathematical and algorithmic languages which may be analyzed mechanically by a simple procedure based on a matrix representation of a precedence relation between character pairs.

Introduction

The central problem in constructing a compiler for a nontrivial programming language is the syntactic analysis by mechanical means of formulas in the language. Such mechanical analyzers are conventionally constructed "by hand"; freedom from syntactic errors is achieved with difficulty, and subsequent additions or modifications to the source language, even though readily incorporated in the formal grammar, may necessitate extensive alterations to the analyzer.

Efforts have been made [1] to construct analyzers capable of analyzing any of a wide class of phrase structure languages, operating interpretively under control of a tabular representation of the grammar for the current language. Such "syntax directed" analyzers presently seem to be inefficient, although detailed information is lacking.

There is a significant and useful class of formal languages, however, for which the design of an efficient mechanical analyzer may be determined in a simple way from the grammar of the language itself. It will be our purpose to characterize this class of languages, to describe the construction of an analyzer and to verify the correctness and uniqueness of the resulting analysis.

Definitions and Conventions

Numerous results used in the paper, but not essential to an intuitive understanding of the new techniques presented, are proved in Appendix A. References to Appendix A appear as [T1] or [C1], designating Theorem 1 and Corollary 1, respectively.

The properties of *characters* and *strings* of characters are well-known and will be used informally. Characters will be represented by Latin capitals, strings by lower case Latin letters, and the null string by Λ. If x and y are strings, xy is the *concatenation* of x and y, formed by writing x followed by y. If $z = xy$ is a string, x is a *head* and y a *tail* of z. A head of z consisting of a single character is the *initial character* of z; a tail of z consisting of a single character is the *final character* of z.

* Received September, 1962. The research reported in this paper was sponsored in part by the Air Force Cambridge Research Laboratories, Office of Aerospace Research, under Contract AF19(628)-419.

A *language* is a set of strings, and a *grammar* a means of defining a language and ascribing structure to its strings. The set of characters under consideration is the vocabulary V, and the set of strings over a vocabulary is W_V.

A *production* $U \rightarrow u$ is an ordered pair consisting of a character U and a nonempty string u. The relation $w \rightarrow v$ holds (with respect to a set P of productions) if there are strings x and y and a character U such that $w = xUy$, $v = xuy$ and $U \rightarrow u$ is a production of P. The relation $w \Rightarrow v$ holds (with respect to P) if there is a finite sequence $w = w_0, w_1, \cdots, w_n = v$ $(n \geqq 0)$ such that $w_{i-1} \rightarrow w_i$ $(1 \leqq i \leqq n)$. The sequence $w_0, w_1, \cdots, w_n$ is said to be a *derivation*[1] of v from w, and v is a *w-derivative*. The relation "$\Rightarrow$" is reflexive and transitive; $x \Rightarrow x$; and if $x \Rightarrow y$ and $y \Rightarrow z$, then $x \Rightarrow z$. If $x_i \Rightarrow y_i$ $(1 \leqq i \leqq n)$ then $x_1x_2 \cdots x_n \Rightarrow y_1y_2 \cdots y_n$. Conversely, if $x \Rightarrow y$ and $x = x_1x_2 \cdots x_n$, then there are strings $y_1, y_2, \cdots, y_n$ such that $y = y_1y_2 \cdots y_n$ and $x_i \Rightarrow y_i$ $(1 \leqq i \leqq n)$ [4, T1, C1]. If $x \Rightarrow y$, then y is nonempty if and only if x is nonempty.

If P is a set of productions containing exactly one character S which appears only on the left of "$\rightarrow$", and a nonempty set T of (terminal) characters which appear only on the right of "$\rightarrow$", then P is a *phrase structure grammar* [3, 4]. The derivatives of S (with respect to P) are *sentential forms* and the sentential forms in W_T are the *sentences* of the *phrase structure language* L_P. Without restricting the set of phrase structure languages, we shall assume that for each character Y of P there are strings x, z and t such that $S \Rightarrow xYz$, $Y \Rightarrow t$ and $t \in W_T$; that is, every character may be used in deriving some sentence.

If no production of the phrase structure grammar P takes the form $U \rightarrow xU_1U_2y$, where U_1 and U_2 are nonterminal characters (NTC), then P is an *operator grammar* and L_P is an *operator language*. It is readily shown by induction [T3, C3] that if P is an operator grammar then no derivative of any character, and in particular no sentential form, contains two adjacent NTC.

In an operator grammar, there are three relations some or all of which may hold between two terminal characters T_1 and T_2:

(1) $T_1 \doteq T_2$ if there is a production $U \rightarrow xT_1T_2y$ or $U \rightarrow xT_1U_1T_2y$, where $U_1 \in V - T$;

(2) $T_1 \gtrdot T_2$ if there is a production $U \rightarrow xU_1T_2y$ and a derivation $U_1 \Rightarrow z$ where $U_1 \in V - T$ and T_1 is the rightmost terminal character of z;

(3) $T_1 \lessdot T_2$ if there is a production $U \rightarrow xT_1U_1y$ and a derivation $U_1 \Rightarrow z$ where $U_1 \in V - T$ and T_2 is the leftmost terminal character of z.

A *precedence grammar* is an operator grammar for which no more than one of the three relations holds between any ordered pair T_1, T_2 of terminal symbols. The relations are called *precedence relations*. The goal of the paper as a whole is to provide techniques to recognize precedence grammars, to construct matrix representations of their precedence relations and to analyze sentences by means of such precedence matrices.

[1] Strictly speaking, a derivation of v from w must specify, for each step $w_{i-1} \rightarrow w_i$, which character of w_{i-1} is operated on and by which production. We shall make implicit use of this requirement.

General Discussion

An example of a precedence grammar P_1 is the following:

$$S \rightarrow A$$
$$A \rightarrow A + B$$
$$A \rightarrow B$$
$$B \rightarrow B * C$$
$$B \rightarrow C$$
$$C \rightarrow (A)$$
$$C \rightarrow \lambda$$

In P_1, $T = \{+, *, (,), \lambda\}$ and $V - T = \{S, A, B, C\}$. If $C \Rightarrow z$, then the leftmost terminal character (LTC) of z must be (or λ. If $B \Rightarrow z$, the LTC of z must be (, λ, or $*$. Continuing in the same manner, we arrive at a table (Figure 1) showing the possible left- and rightmost terminal characters (LTCD and RTCD) of the derivatives of each NTC.

Because of the production $C \rightarrow (A)$, the relation ($\doteq$) holds in P_1. Because $A \rightarrow A + B$ and $*$ is a RTCD of A, $*\gtrdot +$. Because $C \rightarrow (A)$ and (is a LTCD of A, $(\lessdot)$. The complete matrix of precedence relations is shown in Figure 2.

If $s = xyz$ is a sentential form, then y is a *phrase* of s provided there is a sentential form $x'Yz'$ for which $x' \Rightarrow x$, $Y \Rightarrow y$, and $z' \Rightarrow z$, where $Y \in V - T$. Equivalently, y is a phrase of $s = xyz$ if there is a sentential form xYz for which $Y \Rightarrow y$, $Y \in V - T$. A *prime phrase* is a phrase which contains at least one terminal character, but no prime phrase other than itself. Thus in L_{P_1} the sentential form $C * C * (\lambda + B)$ contains only the prime phrases $C * C$ and λ; each other phrase either consists of a NTC or contains one of the prime phrases. Every sentential form containing more than one character contains at least one prime phrase [T13].

In a precedence language, there is a simple technique for recognizing prime phrases [T12]. If a string x appears in the context BxC within a sentential form s, where B and C are terminal characters, x contains the terminal characters

NTC	LTCD	RTCD
C	$(\lambda$	$)\lambda$
B	$(\lambda*$	$)\lambda*$
A	$(\lambda*+$	$)\lambda*+$
S	$(\lambda*+$	$)\lambda*+$

FIG. 1

	(	λ	*	+	)
)			$\gtrdot$	$\gtrdot$	$\gtrdot$
λ			$\gtrdot$	$\gtrdot$	$\gtrdot$
*	$\lessdot$	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$

FIG. 2

$A_1, A_2, \cdots, A_n$ $(n \geqq 1)$, and $B \lessdot A_1$, $A_n \gtrdot C$, $A_i \doteq A_{i+1}$ $(1 \leqq i \leqq n-1)$, then x is a prime phrase. (Note that the nonterminal characters of x are ignored in this decision.) If x is not preceded by any character in s, we consider $B \lessdot A_1$ to be true; similarly if no character of s follows x, we consider $A_n \gtrdot C$ to be true. This convention is implemented automatically if each sentential form s is written in the augmented form $\vdash s \dashv$, with $\vdash \lessdot A$ and $A \gtrdot \dashv$ for all A in T.

The process of syntactic analysis of a sentential form s may now be described. The string is written in the augmented form $s_0 = \vdash s \dashv$. If s_0 contains a prime phrase, the prime phrase is replaced by the NTC N_1, yielding the string s_1. The procedure is iterated, some prime phrase of s_i being replaced by N_{i+1} to obtain s_{i+1}. Since each step diminishes the number of terminal characters in s_i, the process must in a finite number of steps result in $s_m = \vdash N_m \dashv$. Denoting s_i' as the string such that $\vdash s_i' \dashv = s_i$, the sequence $S \Rightarrow s_m' \Rightarrow s_{m-1}' \Rightarrow \cdots \Rightarrow s_1' \Rightarrow s_0' = s$ is a skeletal representation of a derivation of s. The details of the derivation may usually be filled in without difficulty, substituting the appropriate nonterminal symbol of P for each N_i, and determining the derivation of $s_i' \Rightarrow s_{i-1}'$ for each value of i. Since each of these derivations takes the form $xN_iz \Rightarrow xyz$, where y is a prime phrase, and since the number of possible prime phrases is finite, this determination is essentially trivial.

Consider by way of example the sentence $(\lambda + \lambda) * \lambda$ of P_1. Since $(\lessdot \lambda \gtrdot +$, the first occurrence of λ in $s_0 = \vdash (\lambda + \lambda) * \lambda \dashv$ is a prime phrase, and s_1 is $\vdash (N_1 + \lambda) * \lambda \dashv$. Continuing in this manner, we have the skeletal derivation of Figure 3, where the leftmost prime phrase of s_i is chosen for analysis at each step.

To complete the derivation, we must establish derivations of:

$$\begin{array}{lll} S \Rightarrow N_6 & N_4 \Rightarrow (N_3) & N_2 \Rightarrow \lambda \\ N_6 \Rightarrow N_4 * N_5 & N_3 \Rightarrow N_1 + N_2 & N_1 \Rightarrow \lambda \\ N_5 \Rightarrow \lambda & & \end{array}$$

For example:

$$S \rightarrow A \rightarrow B = N_6$$

$$\begin{array}{ll} N_6 = B \rightarrow B * C = N_4 * N_5 & N_3 = A \rightarrow A + B = N_1 + N_2 \\ N_5 = C \rightarrow \lambda & N_2 = B \rightarrow C \rightarrow \lambda \\ N_4 = B \rightarrow C \rightarrow (A) = (N_3) & N_1 = A \rightarrow B \rightarrow C \rightarrow \lambda \end{array}$$

i	s_i	*Prime phrases*
0	$\vdash(\lambda + \lambda) * \lambda\dashv$	$\lambda, \lambda, \lambda$
1	$\vdash(N_1 + \lambda) * \lambda\dashv$	λ, λ
2	$\vdash(N_1 + N_2) * \lambda\dashv$	$N_1 + N_2, \lambda$
3	$\vdash(N_3) * \lambda\dashv$	$(N_3), \lambda$
4	$\vdash N_4 * \lambda\dashv$	λ
5	$\vdash N_4 * N_5\dashv$	$N_4 * N_5$
6	$\vdash N_6\dashv$	

FIG. 3

The table which shows for each nonterminal character the leftmost terminal characters of its derivatives may be constructed as follows:

(1) For each production $U_1 \rightarrow T_1x$ or $U_1 \rightarrow U_2T_1x$ ($T_1 \in T$, $U_2 \in V - T$), enter T_1 as a LTCD of U_1.

(2) For each production $U_1 \rightarrow U_2x$, enter every LTCD of U_2 as a LTCD of U_1.

(3) Repeat Step 2 until, in a finite number of steps, the process converges.

An analogous procedure constructs a table of rightmost terminal characters of the derivatives of each nonterminal character.

A flow chart is shown in Figure 4 which performs a syntactic analysis of a string in a precedence language, processing phrases from left to right. This flow chart could be taken as the nucleus of a simple compiler, although other sequences of processing the phrases may potentially yield better coding in many instances [2].

Let us consider several instances of phrase structure grammars which are not precedence grammars and show how they may be treated as precedence grammars by minor adjustments.

In ALGOL 60 [5], the definition of an identifier (I) is equivalent to

$$\begin{aligned} I &\rightarrow L \\ I &\rightarrow IL \\ I &\rightarrow ID \\ L &\rightarrow a \\ L &\rightarrow b \\ &\vdots \\ L &\rightarrow z \\ D &\rightarrow 0 \\ D &\rightarrow 1 \\ &\vdots \\ D &\rightarrow 9 \end{aligned}$$

Because of the production $I \rightarrow IL$, ALGOL 60 is not an operator language. If, however, the categories L and D are eliminated, we have:

$$\begin{array}{lll} I \rightarrow a & I \rightarrow Ia & I \rightarrow I0 \\ I \rightarrow b & I \rightarrow Ib & I \rightarrow I1 \\ \vdots & \vdots & \vdots \\ I \rightarrow z & I \rightarrow Iz & I \rightarrow I9 \end{array}$$

This set of productions is consistent with a precedence grammar. It is, however, unwieldy. We may achieve the same effect by the set of productions:

$$\begin{aligned} I &\rightarrow \lambda \\ I &\rightarrow I\lambda \\ I &\rightarrow I\delta \end{aligned}$$

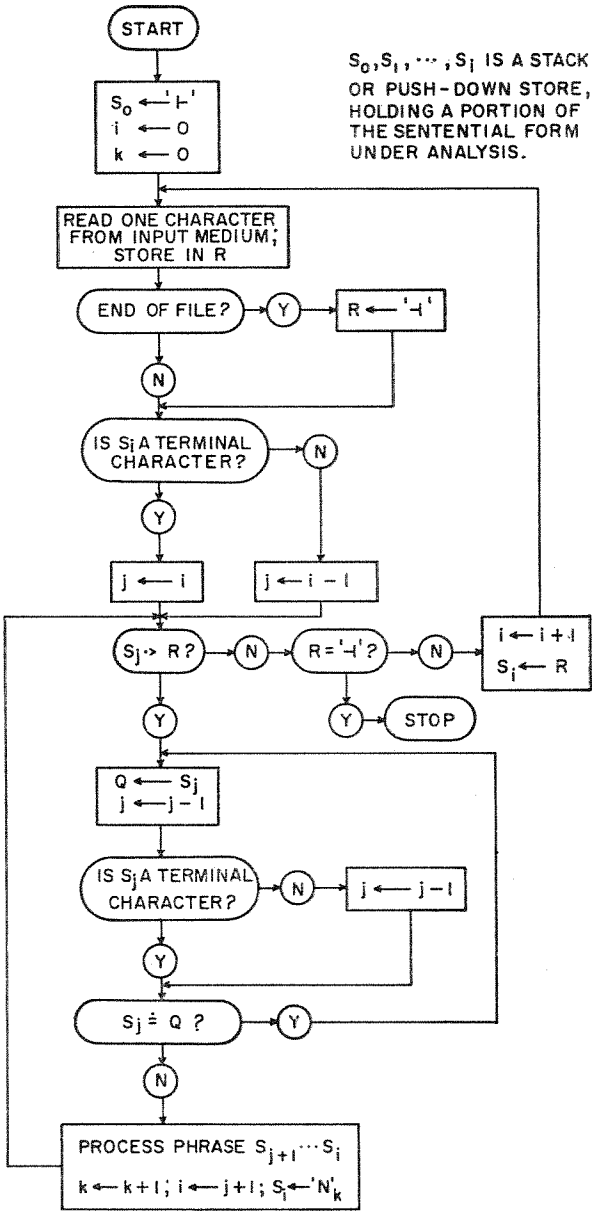

FIG. 4

where λ and δ are terminal symbols representing an arbitrary letter and an arbitrary digit, respectively, We speak of λ as a *representative* of the set of letters.

Again in ALGOL 60, the definition of a for statement is:

$$\langle \text{for statement} \rangle \longrightarrow \langle \text{for clause} \rangle \langle \text{statement} \rangle.$$

While this is not consistent with an operator grammar, one may modify the definition of the for clause from

$$\langle \text{for clause} \rangle \longrightarrow \textbf{for}\ \langle \text{variable} \rangle := \langle \text{for list} \rangle\ \textbf{do}$$

to

$$\langle \text{for clause} \rangle \longrightarrow \textbf{for}\ \langle \text{variable} \rangle := \langle \text{for list} \rangle$$

so that the definition of a for statement becomes

$$\langle \text{for statement} \rangle \longrightarrow \langle \text{for clause} \rangle\ \textbf{do}\ \langle \text{statement} \rangle$$

which is consistent with a precedence grammar.

A simple formula language may be defined by P_2 :

$$\begin{aligned}
S &\longrightarrow A \\
A &\longrightarrow A - B \\
A &\longrightarrow B \\
B &\longrightarrow B * C \\
B &\longrightarrow C \\
C &\longrightarrow -D \\
C &\longrightarrow D \\
D &\longrightarrow (A) \\
D &\longrightarrow \lambda
\end{aligned}$$

While P_2 is an operator grammar, both $*\gtrdot -$ and $*\lessdot -$, so that P_2 is not a precedence grammar. If the unary and binary minus signs were distinct characters, however, we would have a precedence grammar P_3 , with the precedence matrix of Figure 5:

$$\begin{aligned}
S &\longrightarrow A \\
A &\longrightarrow A - B \\
A &\longrightarrow B \\
B &\longrightarrow B * C \\
B &\longrightarrow C \\
C &\longrightarrow \theta\, D \\
C &\longrightarrow D \\
D &\longrightarrow (A) \\
D &\longrightarrow \lambda
\end{aligned}$$

We may construct a matrix (Figure 6) showing the allowable character pairs of sentences in P_3 .

In a sentence of P_3 , the character preceding any occurrence of $-$ must be) or λ. That preceding any occurrence of θ must be $*$, $-$, (, or $\vdash$. Since these two sets of characters are disjoint, the characters $-$ and θ may be distinguished by inspection of the preceding character, even when $-$ and θ are represented by the same symbol, as in P_2 . A compiler for a language like P_2 might make

NTC	LTCD	RTCD
S	$-\ *\ \theta\ (\ \lambda$	$-\ *\ \theta\ )\ \lambda$
A	$-\ *\ \theta\ (\ \lambda$	$-\ *\ \theta\ )\ \lambda$
B	$*\ \theta\ (\ \lambda$	$*\ \theta\ )\ \lambda$
C	$\theta\ (\ \lambda$	$\theta\ )\ \lambda$
D	$(\ \lambda$	$)\ \lambda$

	(	λ	θ	*	−	)
)				⩾	⩾	⩾
λ				⩾	⩾	⩾
θ	⩽	⩽		⩾	⩾	⩾
*	⩽	⩽	⩽	⩾	⩾	⩾
−	⩽	⩽	⩽	⩽	⩾	⩾
(	⩽	⩽	⩽	⩽	⩽	≐

FIG. 5

	(	λ	θ	*	−	)
)				x	x	x
λ				x	x	x
θ	x	x				
*	x	x	x			
−	x	x	x			
(	x	x	x			

FIG. 6

use of an input program which would at all times remember the last character read, and would change a $-$ to a θ if its predecessor was one of $*$, $-$, $($, or $\vdash$. The remainder of the compiler would then process the precedence language P_3. If a grammar represents the absolute value function by $|x|$, the two uses of the vertical line may be distinguished in the same way.

The determination of the allowable character pairs of the sentential forms of a language proceeds as follows:

(1) For each character X, enter X in a table as a leftmost character of a derivative (LCD) of X.
(2) For each production $U \longrightarrow Xy$, enter each LCD of X as a LCD of U.
(3) Repeat Step 2 until in a finite number of steps the process converges.
(4) By a process analogous to steps 1–3, determine the rightmost characters of the derivatives (RCD) of each character.
(5) For each character pair XY which occurs in the string on the righthand side of a production, if X_1 is a RCD of X and Y_1 a LCD of Y then X_1Y_1 is an allowable character pair in sentential forms of the language. Those allowable character pairs both of which are terminal characters may occur in sentences of the language.

The representation of a precedence matrix requires a table with n^2 entries, where n is the number of terminal characters in V. It is possible, however, in many instances to represent the useful content of the matrix far more compactly. Suppose there are two functions $f(X)$ and $g(X)$ mapping terminal symbols into numbers, such that if $X \lessdot Y$ then $f(X) < g(Y)$, if $X \doteq Y$ then $f(X) = g(Y)$, and if $X \gtrdot Y$ then $f(X) > g(Y)$. Assuming that exactly one of $X \lessdot Y$, $X \doteq Y$, and $X \gtrdot Y$ is true, one may determine which by evaluating and comparing $f(X)$ and $g(Y)$. The tables representing f and g require only $2n$ positions. For example, the precedence matrix of Figure 5 may be represented by Figure 7.

X	$f(X)$	$g(X)$
)	5	1
λ	5	6
θ	5	6
*	5	4
−	3	2
(	1	6

FIG. 7

If such precedence functions exist, we may compute instances of them from the precedence matrix in the following manner:

(1) Initially set $f(X) = g(X) = 1$ for all $X \in T$.

(2) For each instance of $X \gtrdot Y$, if $f(X) \leqq g(Y)$ set $f(X)$ equal to $g(Y) + 1$.

(3) For each instance of $X \lessdot Y$, if $f(X) \geqq g(Y)$ set $g(Y)$ equal to $f(X) + 1$.

(4) For each instance of $X \doteq Y$, if $f(X) \neq g(Y)$ set the smaller of $f(X)$ and $g(Y)$ equal to the larger.

(5) Iterate Steps 2, 3 and 4 until the process converges. If any value of f or g becomes greater than twice the number of terminal characters in V, the process will never converge, and no pair of functions has the desired qualities.

There are more efficient ways of computing f and g, but the technique given clearly converges only to functions satisfying the desired conditions. In the process of analyzing sentential forms in a precedence language by the algorithm of Figure 4, one considers the precedence relations only of terminal characters T_1 and T_2 for which T_1T_2 or $T_1U_1T_2$ occurs in some sentential form. Therefore [T9], exactly one of $T_1 \lessdot T_2$, $T_1 \doteq T_2$, and $T_1 \gtrdot T_2$ occurs, according as $f(T_1) < g(T_2)$, $f(T_1) = g(T_2)$, or $f(T_1) > g(T_2)$, respectively.

The existence of a pair of precedence functions representing a precedence matrix seems very unlikely *a priori*, and one readily constructs examples of precedence grammars for which precedence functions of the type described here do not exist. Yet the author has found that insofar as programming languages or natural languages can be represented by precedence grammars, they can also be represented by a pair of precedence functions without doing violence to the intuitive phrase structure of the language. This suggests that the intuitive perception of phrase structure may be based on a mechanism similar to the use of precedence functions.

Applications

Several uses for the theory of precedence languages present themselves. Some existing programming languages are, in fact, precedence languages and could be compiled using the flow chart of Figure 4 as an analyzer. Probably more existing languages are nearly precedence languages; only for a small number of terminal character pairs does more than one precedence relation hold. The analyzer above could be adapted to test for these special cases, and refer to an appropriate subroutine to resolve each ambiguity.

If a phrase structure grammar generates ambiguous sentences, the precedence matrix will show terminal character pairs for which more than one precedence relation holds. Such entries in the matrix are guides toward detecting the presence of ambiguous sentences, although they do not by themselves prove the grammar to be ambiguous.

Designers of new programming languages might do well to consider precedence languages; the restriction, other things being equal, benefits both the translator and the user of the language. The grammar P_4 shown in Appendix B defines a language closely comparable to ALGOL 60 [5], departing from it only for sufficient reason. Because it contains none of the syntactic ambiguities of ALGOL 60 [6], is much more concise (using 43 nonterminal characters rather than 109), and assigns more natural constructions to several grammatical forms, it might be taken as a model for the eventual successor to ALGOL 60. It is included here because it exemplifies most of the difficulties which may be encountered in analyzing an operator language by precedence techniques.

In the grammar of P_4, the form $U \rightarrow x \mid y \mid z$ is an abbreviation for the productions $U \rightarrow x$, $U \rightarrow y$, and $U \rightarrow z$. The form $U \rightarrow x\{y\}z$ is an abbreviation for the infinite set of productions $U \rightarrow xz$, $U \rightarrow xyz$, $U \rightarrow xyyz$, $U \rightarrow xyyyz$, etc. Only the first and third productions of such a set need be considered in deriving precedence relations. Observe that while the notation $x\{y\}z$ does not increase the class of describable languages it may reduce the number of productions required by the grammar and is the only way of introducing prime phrases of arbitrarily great length. The form $U \rightarrow x\{y\}^a z$ is an abbreviation for the set of productions $U \rightarrow xyy \cdots yz$, where the number of occurrences of y is no greater than a.

While P_4 is not an operator grammar, omitting the first seven sets of productions converts it to an operator grammar. The characters λ, δ, β, π, μ, ρ and τ are then terminal characters serving as representatives of letters, digits, truth values, signs, multiplying operators, relational operators and type designators, respectively.

In P_4, the same problem concerning the signs $+$ and $-$ arises as arose in P_2. It may be solved in exactly the same way.

The precedence relations **else** $\gtrdot$: and **else** $\lessdot$: both hold in P_1. In the first instance, the relation applies to the **else** of Production 18 and the colon of Production 27; in the other, the relation applies to the **else** and colon of Productions 40 and 41. The two uses of the colon may be distinguished by counting the preceding left and right brackets; if a colon is preceded by more left brackets than right brackets, it is of the type in Production 27. If an input program distinguishes the two types of colon and the two usages of plus and minus signs, P_4 is a precedence grammar, with the precedence matrix shown in Figure 8. In the figure, $\odot$ represents the colon of Productions 36, 40 and 41, and θ represents the π of Production 15. Figure 8 also gives values of the two functions $f(X)$ and $g(Y)$ for P_4; for the precedence relation **array** $\doteq$ [, however, $f(\textbf{array}) < g([)$, so that this case must be detected and treated separately.

These departures from the definition of precedence grammars are inessential;

X \ Y	δ	λ	•	]	)	go to	10θ μ ↑	π	for constant	function	else	ρ	¬	∧	∨	⊃
$f(X)$	22	22	20	20	20	19	18	16	15	15	14	14	12	12	10	8
δ	>	>	<			<	<	<	<	<	<	<	<	<	<	<
λ	>	>					<	<			<	<	<	<	<	<
•	>	>				=	<	<		<	<	<	<	<	<	<
]	>	>				=	<	<			<	<	<	<	<	<
(	>	>					<	<			<	<	<	<	<	<
↑	>	>					<	<			<	<	<	<	<	<
μ	>	>	>	>	>		>	<			<	<	<	<	<	<
begin go to for comment											<					
if := ⊙	>	>		>				.	=	=	<					
π	>	>	>	>	>		>	>			<	<	<	<	<	<
¬													<			
β													<	<	<	<
ρ	>	>	>	>	>		>	>			>	>	<	<	<	<
∧ ∨ ⊃ ≡	>	>	>	>	>		>	>				>	>	>	<	<
:	>	>	>	>	>		>	>			>					
step while	>	>	>	>	>		>	>			>					
array																
value																
]) op ,	>	>	>	>	>		>	>			>	>	>	>	>	>
	>	>	>	>	>		>	>			>	>	>	>	>	>
	>	>	>	>	>		>	>			>	>	>	>	>	>
	>	>	>	>	>		>	>			>					
constant function switch procedure																
else	>	>	>	>	>	>	>	>			>	>	>	>	>	>
until	>	>	>	>	>		>	>			>					
then	>	>	>	>	>		>	>			>	>	>	>	>	>
; end	>	>	>	>	>	>	>	>			>	>	>	>	>	>

	$g(Y)$	21	20	19	19	19	19	19	17	15	15	15	13	13	13	11	9	7	5	4	4	3	3	3	3	3	2	2	1	1	1	1	1
≡	6																			>	>	>	>	>	>	>		>	>	>	>	>	>
β	12																			>	>	>	>	>	>	>		>	>	>	>	>	>
until	4		>	>	>	>	>	>	>		>		>	>	>	>	>	>	>														
while	4		>	>	>	>	>	>	>		>		>	>	>	>	>	>	>	>	>	>	>	>	>	>		>	>	>	>	>	>
comment	3										>		>	>	>	>	>	>	>									>	>	>	>	>	>
end																																	
value	3		>	>	>	>	>	>	>																								
τ	3		>	>	>	>	>	>	>																								
array	3		>	>	>	=	>	>	>																								
do ⊙	3		>	>	>	>	>	>	>		>	>	>	>	>	>	>	>	>														
[	3		>	>	>	>	>	>	>		>	>	>	>	>	>	>	>	>	>	>	=	=	=	=	=	=						
(	3		>	>	>	>	>	>	>		>	>	>	>	>	>	>	>	>	>	>		=	=	=	=	=						
,	3		>	>	>	>	>	>	>		>	>	>	>	>	>	>	>	>	>	>			=	=	=	=						
:=	3		>	>	>	>	>	>	>		>	>	>	>	>	>	>	>	>	>	>				=	=	=						
switch	2		>	>	>	>	>	>	>	>										>	>												
then	2		>	>	>	>	>	>	>		>	>	>	>	>	>	>	>	>	>	>							=	>	>	>	>	>
step	1		>	>	>	>	>	>	>		>	>	>	>	>	>	>	>	>										=	>	>	>	>
if	1		>	>	>	>	>	>	>		>	>	>	>	>	>	>	>	>	>	>								=	>	>	>	>
; begin	1		>	>	>	>	>	>	>		>	>	>	>	>	>	>	>	>									<	<	=	=	=	=
procedure	1		>	>	>	>	>	>	>	>																					=	=	=	=

Fig. 8

they illustrate liberties which may be taken with the abstract structure at slight cost. In principle, the terminal symbols serving as representatives may be eliminated from the grammar entirely, substituting for each representative all the characters which it may represent. For example, the production

$$\text{factor} \longrightarrow \pi\ \text{factor}$$

becomes

$$\text{factor} \longrightarrow +\ \text{factor}$$
$$\text{factor} \longrightarrow -\ \text{factor}.$$

The difference between the precedence relations of the unary and binary plus and minus signs occurs because the unary signs are introduced in the definition of factor, rather than in that of simple arithmetic expression. In ALGOL, FORTRAN and most other programming languages the two usages of the sign satisfy the same precedence relations. The double use of the vertical line in representing the absolute value function $|\,x\,|$ is, however, an example from conventional mathematical notation of a multiple usage of a character in which the precedence relations satisfied by the character depend upon its context: the immediately preceding character.

The relation **else** $\gtrdot$: may be eliminated by redefining

27 limit pair $\longrightarrow$ simple arithmetic expression : arithmetic expression.

A slight reformulation of the definition of array declaration results in **array** $\lessdot$ [, in accordance with $f(\textbf{array}) < g([)$.

APPENDIX A. THE THEORY

Proofs follow for most of the properties of phrase structure grammars (PSG), operator grammars (OG) and precedence grammars (PG) used in the text. Most of the proofs are based on the *Induction Principle* (PSG):

To prove a general proposition of the form $P(x, y)$ for the strings x, y such that $x \Longrightarrow y$, it is sufficient to prove $P(x, x)$ and to prove that if $P(x, z)$ and $x \Longrightarrow z \longrightarrow y$, then $P(x, y)$.

The induction principle may be verified by consideration of the length n of a derivation $(x = x_0 \longrightarrow x_1 \longrightarrow \cdots \longrightarrow x_{n-1} \longrightarrow x_n = y)$ of y from x. If $n = 0$ then $x = y$, and it is sufficient to prove $P(x, x)$. If $n > 0$, we may assume the theorem true for the derivation $x \Longrightarrow x_{n-1} = z$, which is of length $n - 1$; therefore, in proving $P(x, y)$ we may assume $x \Longrightarrow z \longrightarrow y$ and $P(x, z)$.

THEOREM 1 (PSG). *If $x_1x_2 \Longrightarrow y$, then there are strings y_1 and y_2 such that $x_1 \Longrightarrow y_1$, $x_2 \Longrightarrow y_2$ and $y_1y_2 = y$.*

PROOF. If $y = x_1x_2$, let $y_1 = x_1$ and $y_2 = x_2$. Otherwise, by the induction principle $x_1x_2 \Longrightarrow z \longrightarrow y$, and $x_1 \Longrightarrow z_1$, $x_2 \Longrightarrow z_2$, $z_1z_2 = z$. The relation $z_1z_2 \longrightarrow y$ implies one of two situations:

(1) z_1 takes the form z_3Uz_4 such that $U \longrightarrow u$, $y = z_3uz_4z_2$; then $x_1 \Longrightarrow z_1 \Longrightarrow z_3uz_4 = y_1$, $x_2 \Longrightarrow z_2 = y_2$, $y_1y_2 = y$.

(2) z_2 takes the form z_3Uz_4, etc.; $x_1 \Longrightarrow z_1 = y_1$, $x_2 \Longrightarrow z_2 \Longrightarrow z_3uz_4 = y_2$, $y_1y_2 = y$.

COROLLARY 1 (PSG). *If $x_1x_2 \cdots x_n \Longrightarrow y$, then there are strings $y_1, y_2, \cdots, y_n$ such that $x_i \Longrightarrow y_i$ $(1 \leqq i \leqq n)$ and $y = y_1y_2 \cdots y_n$.*

PROOF. By induction on Theorem 1. If a detailed derivation of y from $x_1, x_2, \cdots, x_n$ is given, the strings $y_1, y_2, \cdots, y_n$ are uniquely determined.

THEOREM 2 (PSG). *If $x \Longrightarrow y$ and the initial character of x is terminal, then the initial character of y is terminal.*

PROOF. If $x = y$, the conclusion is immediate. Otherwise, by the induction principle $x \Longrightarrow z \longrightarrow y$ where the initial character of z is terminal. The production $U \longrightarrow u$ which transforms z into y operates on a nonterminal character U of z, and therefore leaves the initial character unchanged.

COROLLARY 2 (PSG). *If $x \Longrightarrow y$ and the initial character of y is nonterminal, then the initial character of x is nonterminal.*

PROOF. Obvious from Theorem 2.

THEOREM 3 (OG). *If $x \Longrightarrow y$ and x nowhere contains two adjacent nonterminal characters, then y nowhere contains two adjacent nonterminal characters.*

PROOF. If $x = y$, the conclusion is immediate. Otherwise, by the induction principle $x \Longrightarrow z \longrightarrow y$, and z nowhere contains two adjacent nonterminal characters. Since $z \longrightarrow y$, $z = vUw$, $U \longrightarrow u$, and $y = vuw$. Neither v nor w contains two adjacent NTC; nor, by the definition of an operator grammar, does u contain two adjacent NTC. Since U is nonterminal, the final character of v and the initial character of w must be terminal; so a pair of adjacent NTC can occur nowhere in vuw.

COROLLARY 3 (OG). *No sentential form contains two adjacent nonterminal characters.*

PROOF. Every sentential form is derived from S; apply Theorem 3.

THEOREM 4 (OG). *If T_1U_1 occurs in a sentential form s, where T_1 is terminal and U_1 is nonterminal, then any phrase of s containing T_1 also contains U_1.*

PROOF. Suppose $s = xyz$ where y is a phrase. Then $S \Longrightarrow x'Uz'$, $x' \Longrightarrow x$, $U \Longrightarrow y$, $z' \Longrightarrow z$. If y contains T_1 but not U_1, then U_1 is the initial character of z, and by Corollary 2 the initial character of z' is nonterminal. A pair of adjacent NTC therefore occurs in the sentential form $x'Uz'$, contradicting Corollary 3. If y contains T_1, it then must contain U_1.

COROLLARY 4 (OG). *If U_1T_1 occurs in a sentential form s ($T_1 \in T$, $U_1 \in V - T$), then any phrase of s containing T_1 also contains U_1.*

PROOF. Similar to the proof of Theorem 4, considering the final character of x'.

THEOREM 5 (OG). *No phrase in a sentential form is immediately preceded or followed by nonterminal characters.*

PROOF. If y is a phrase in $xyz = s$ and the initial character of z is nonterminal, then the final character of y is terminal, by Corollary 3. This contradicts Theorem 4. Similarly, the final character of x cannot be nonterminal. Of course, either of x and z may be null strings.

THEOREM 6 (OG). *If T_1U_1 occurs in a sentential form s ($T_1 \in T$, $U_1 \in V - T$) and if T_2 is the leftmost terminal character of some derivative v of U_1, then $T_1 \lessdot T_2$.*

PROOF. Since s is a sentential form and $s \neq S$, there is a string z for which $S \Longrightarrow z \longrightarrow s$. By the induction principle, we may assume the theorem true for the sentential form z. Suppose $z = xUy$, $U \longrightarrow u$, and $s = xuy$. The occurrence of T_1U_1 in s may take four forms:

(1) T_1U_1 occurs in x or y, thus in z; by the induction principle, $T_1 \lessdot T_2$.

(2) T_1U_1 occurs in u, where $U \longrightarrow u$. By the definition of the relation $\lessdot$ itself, $T_1 \lessdot T_2$.

(3) T_1 is the final character of x; U_1 the initial character of $u = U_1v_1$; $U \Longrightarrow U_1v_1 \Longrightarrow vv_1$; T_2 is the leftmost terminal character of a derivative vv_1 of U, and T_1U occurs in z; by the induction principle $T_1 \lessdot T_2$.

(4) T_1 is the final character of u; U_1 the initial character of y. Then UU_1 occurs in the sentential form xUy, contrary to Corollary 3.

COROLLARY 6 (OG). *If U_1T_1 occurs in a sentential form s ($T_1 \in T$, $U_1 \in V - T$) and if T_2 is the rightmost terminal character of some derivative v of U_1, then $T_2 \gtrdot T_1$.*

PROOF. Apply to Theorem 6 the obvious symmetries of $\lessdot$ and $\gtrdot$.

THEOREM 7 (OG). *If the substring $T_1U_1T_2$ occurs in the sentential form s ($T_1 \in T$, $U_1 \in V - T$, $T_2 \in T$), then at least one of the three relations $T_1 \lessdot T_2$, $T_1 \doteq T_2$, or $T_1 \gtrdot T_2$ holds.*

PROOF. Since $s \neq S$, we may assume by the induction principle that $S \Longrightarrow z \longrightarrow s$ and that the theorem holds for the sentential form z; $z = xUy$, $U \longrightarrow u$, and $s = xuy$.

Six cases arise:

(1) $T_1U_1T_2$ occurs in x or in y, thus in z; by induction, one of the three relations holds.

(2) $T_1U_1T_2$ occurs in u; by the definition of the relation, $T_1 \doteq T_2$.

(3) T_1U_1 is a tail of x; then U_1U occurs in z, contrary to Corollary 3.

(4) T_1 is a tail of x; two cases arise:

(4a) U_1T_2 is a head of u, so that T_2 is the leftmost terminal character of a derivative of U. By Theorem 6, $T_1 \lessdot T_2$.

(4b) $U_1 = u$, and T_2 is a head of y. Then T_1UT_2 occurs in z, and by induction one of the three relations holds.

(5) T_1U_1 is a tail of u, T_2 a head of y. Since u is a derivative of U, application of Corollary 6 yields $T_1 \gtrdot T_2$.

(6) T_1 is a tail of u, U_1T_2 a head of y; then UU_1 occurs in z, contrary to Corollary 3.

COROLLARY 7 (OG). *If the substring T_1T_2 occurs in the sentential form S ($T_1 \in T$, $T_2 \in T$), then at least one of the three relations $T_1 \lessdot T_2$, $T_1 \doteq T_2$, or $T_1 \gtrdot T_2$ holds.*

PROOF. A proof may be modeled upon that of Theorem 7.

THEOREM 8 (OG). *If s is a sentential form containing at least one terminal character, then $\vdash s \dashv$ contains a substring AxC for which $A \lessdot B_1 \doteq B_2 \doteq \cdots \doteq B_n \gtrdot C$ ($n \geqq 1$), where the B_i are the terminal characters of x.*

PROOF. By Theorem 7 and Corollary 7, every pair of adjacent terminal characters (whether or not separated by a nonterminal character) satisfies at least one of the three precedence relations; also $\vdash \lessdot T_1$, $T_m \gtrdot \dashv$ where T_1 and T_m are the left- and rightmost terminal characters of s. Let $T_0 = \vdash, T_1, T_2, \cdots, T_m, T_{m+1} = \dashv$ be the terminal characters of s. Let j be the smallest integer for which $T_j \gtrdot T_{j+1}$; such a j exists, since $T_m \gtrdot T_{m+1}$; $1 \leqq j \leqq m$. Let i be the greatest integer such that $i < j$ and $T_i \lessdot T_{i+1}$; such an i exists, since $T_0 \lessdot T_1$. Now for all k such that $i < k < j$, it is not possible that $T_k \gtrdot T_{k+1}$, by the definition of j; nor can $T_k \lessdot T_{k+1}$, by the definition of i. By elimination, we must have $T_k \doteq T_{k+1}$ ($i < k < j$), so that the conclusion of the theorem is true in particular for $A = T_i$ and $C = T_{j+1}$.

THEOREM 9 (PG). *If $T_1U_1T_2$ or T_1T_2 occurs in the sentential form s ($T_1 \in T$, $U_1 \in V - T$, $T_2 \in T$), then exactly one of the three relations $T_1 \lessdot T_2$, $T_1 \doteq T_2$, or $T_1 \gtrdot T_2$ holds.*

PROOF. By Theorem 7 or Corollary 7, at least one of the relations holds; by the definition of a precedence grammar, no more than one holds.

THEOREM 10 (PG). *If $T_1U_1T_2$ or T_1T_2 occurs in a sentential form s and $T_1 \lessdot T_2$, then there is a phrase in s to which T_2, but not T_1, belongs.*

PROOF. There is a derivation $S = s_0 \longrightarrow s_1 \longrightarrow \cdots \longrightarrow s_n = s$ ($n \geqq 1$). Let i be the greatest integer for which T_2 does not occur in s_i ($0 \leqq i \leqq n - 1$). Then T_2 occurs in s_{i+1}, and $s_i = xUy$, $U \longrightarrow u_1T_2u_2$, $s_{i+1} = xu_1T_2u_2y$. Three cases arise:

(1) T_1 first occurs in s_{i+1}; then it is the rightmost terminal character of u_1, and $T_1 \doteq T_2$, contradicting $T_1 \lessdot T_2$.

(2) T_1 first occurs in s_j ($j > i + 1$); then it is the rightmost terminal character of some derivative of u_1, so that $T_1 \gtrdot T_2$ by an application of Corollary 6, contradicting $T_1 \lessdot T_2$.

(3) T_1 already occurs in s_i, so that the phrase of s derived from U contains T_2 but not T_1.

COROLLARY 10 (PG). *If $T_1U_1T_2$ or T_1T_2 occurs in a sentential form s and $T_1 \gtrdot T_2$, then there is a phrase in s to which T_1, but not T_2, belongs.*

PROOF. Apply the obvious symmetries of $\gtrdot$ and $\lessdot$ to the proof of Theorem 10.

THEOREM 11 (PG). *If $T_1U_1T_2$ or T_1T_2 occurs in the sentential form s, and $T_1 \doteq T_2$, then every phrase containing T_1 contains T_2, and every phrase containing T_2 contains T_1.*

PROOF. Suppose a phrase z contains T_1 but not T_2. It must also contain U_1 in the first case, by Theorem 4, so that $S \Longrightarrow x'Uy'$, $x' \Longrightarrow x$, $U \Longrightarrow z$, $y' \Longrightarrow y$, $xzy = s$, where T_2 is the leftmost character of y and T_1 the rightmost terminal character of z; then $S \Longrightarrow xUy \Longrightarrow xzy$, and by Corollary 6, $T_1 \gtrdot T_2$, contradicting $T_1 \doteq T_2$. Thus every phrase containing T_1 contains T_2. The converse is obvious by symmetry.

THEOREM 12 (PG). *If x is a phrase whose terminal characters are $T_1, T_2, \cdots, T_n$ $(n \geqq 1)$ and $T_i \doteq T_{i+1}$ $(1 \leqq i \leqq n-1)$, then x is a prime phrase.*

PROOF. Any prime phrase in x contains a terminal character T_i. By Theorem 11, it contains each T_i $(1 \leqq i \leqq n)$. By Theorem 4 and Corollary 4, it contains all the characters of x. Thus x is a prime phrase.

THEOREM 13 (PG). *If $T_0\, x\, T_{n+1}$ occurs in the sentential form s, where the terminal characters of x are $T_1, T_2, \cdots, T_n$ $(n \geqq 1)$, and if $T_0 \lessdot T_1$, $T_n \gtrdot T_{n+1}$, and $T_j \doteq T_{j+1}$ $(1 \leqq j \leqq n-1)$, then x is a prime phrase of s.*

PROOF. Assume $S = s_0 \longrightarrow s_i \longrightarrow \cdots \longrightarrow s_n = s$. Let i be the greatest integer for which T_1 does not occur in s_i. By Theorem 11, s_i is also the last step in the derivation of s for which T_j does not occur $(1 \leqq j \leqq n)$. By the proof of Theorem 10, T_0 and T_{n+1} both occur in s_i, so that there is a phrase z containing T_j $(1 \leqq j \leqq n)$ but not T_0 or T_{n+1}; by Theorem 4 and Corollary 4, z must contain with T_j any adjacent nonterminal characters, so that $z = x$. By Theorem 12, x is a prime phrase.

COROLLARY 13 (PG). *If $T_0\, x\, T_{n+1}$ occurs in $\vdash s \dashv$, where s is a sentential form, and the terminal characters of x are $T_1, T_2, \cdots, T_n$ $(n \geqq 1)$, and if $T_0 \lessdot T_1$, $T_n \gtrdot T_{n+1}$, and $T_j \doteq T_{j+1}$ $(1 \leqq j \leqq n-1)$, then x is a prime phrase of s.*

PROOF. Four cases arise:

(1) $T_0 \neq \vdash$, $T_{n+1} \neq \dashv$; apply Theorem 13.

(2) $T_0 = \vdash$, $T_{n+1} \neq \dashv$; by Corollary 10, there is a phrase containing T_n but not T_{n+1}; by Theorem 11, Theorem 4 and Corollary 4, the phrase contains exactly the characters of x; by Theorem 12, x is a prime phrase of s.

(3) $T_0 \neq \vdash$; $T_{n+1} = \dashv$; similar to (2).

(4) $T_0 = \vdash$, $T_{n+1} = \dashv$; $x = s$, so that x is a phrase; by Theorem 12, it is a prime phrase.

THEOREM 14 (PG). *Every sentential form s either is a nonterminal character or contains a prime phrase.*

PROOF. If s is a terminal character, it is itself a prime phrase. If s contains more than one character, then by Corollary 3, it contains a terminal character. By Theorem 8 and Corollary 13, s contains a prime phrase.

APPENDIX B. THE GRAMMAR P_4

1. $\lambda \rightarrow a|b|c| \cdots |y|z$
2. $\delta \rightarrow 0|1| \cdots |8|9$
3. $\beta \rightarrow$ **true** | **false**
4. $\pi \rightarrow + \mid -$
5. $\mu \rightarrow \times \mid / \mid \div$
6. $\rho \rightarrow <|\leqq|=|\geqq|>|\neq$
7. $\tau \rightarrow$ **real** | **integer** | **Boolean**
8. identifier $\rightarrow \lambda$ | identifier λ | identifier δ
9. digit string $\rightarrow \delta$ | digit string δ
10. literal constant $\rightarrow$ digit string | digit string .|. digit string | digit string . digit string
11. subscripted variable $\rightarrow$ identifier [{arithmetic expression,} arithmetic expression]
12. variable $\rightarrow$ identifier | subscripted variable
13. function designator $\rightarrow$ identifier ({expression,} expression)
14. primary $\rightarrow$ function designator | variable | literal constant | (arithmetic expression)
15. factor $\rightarrow$ primary | primary $\uparrow$ factor |$\imath\jmath$factor | π factor

16. term → factor | term μ factor
17. simple arithmetic expression → term | simple arithmetic expression π term
18. arithmetic expression → simple arithmetic expression | **if** Boolean expression **then** arithmetic expression **else** arithmetic expression
19. relation → arithmetic expression ρ arithmetic expression | relation ρ arithmetic expression
20. Boolean primary → β | variable | function designator | relation | (Boolean expression)
21. Boolean secondary → Boolean primary | $\neg$ Boolean primary
22. conjunction → Boolean secondary | conjunction $\wedge$ Boolean secondary
23. disjunction → conjunction | disjunction $\vee$ conjunction
24. implication → disjunction | implication $\supset$ disjunction
25. Boolean expression → implication | Boolean expression $\equiv$ implication
26. expression → arithmetic expression | Boolean expression
27. limit pair → arithmetic expression : arithmetic expression
28. name part → identifier ({identifier,} identifier)
29. specifier → τ{**value**}^{1} {identifier,} identifier | {τ}^{1} **array** {identifier,} identifier
30. go statement → **go to** identifier | **go to** identifier [arithmetic expression]
31. assignment → variable := expression | variable := assignment
32. type declaration → τ {identifier,} identifier | **constant** identifier := expression
33. array declaration → {τ}^{1} **array** {identifier,} identifier [{limit pair,} limit pair]
34. switch declaration → **switch** name part
35. procedure declaration → **procedure** name part {; specifier} {; statement} **end**
36. function declaration → **function** name part : statement
37. procedure call → identifier ({expression,} expression)
38. compound statement → **begin** {declaration;} {statement;} statement **end**
39. for list element → arithmetic expression | arithmetic expression **step** arithmetic expression **until** arithmetic expression | arithmetic expression **while** Boolean expression
40. closed statement → go statement | assignment | procedure call | identifier : {closed statement}^{1} | **comment** | compound statement | **for** identifier := {for list element,} for list element **do** closed statement | **if** Boolean expression **then** closed statement **else** closed statement
41. open statement → identifier : open statement | **for** identifier := {for list element,} for list element **do** open statement | **if** Boolean expression **then** closed statement **else** open statement | **if** Boolean expression **then** statement
42. statement → closed statement | open statement
43. declaration → type declaration | array declaration | switch declaration | procedure declaration | function declaration

Remarks. The words **comment** and **end** may be followed by any string not containing a semicolon, **begin**, **end**, or **else**, without affecting the meaning of the program. Since such strings are typically comments in a natural language, they have the structure assigned them in that language, and cannot be assigned a structure by P_4.

Production 9. The only literal constants explicitly permitted are digit strings optionally containing a decimal point. The ALGOL 60 constant $123_{10}4$ is ruled out as not in accord with conventional usage; the expression 123×10^4, however, is allowed, ane it is the intent of the design that all phrases having constant value be evaluated during compilation.

Production 15. The form 10^x is intended to mean the same as $10 \uparrow x$. Exponentiation associates from the right, since left association plays no useful role for this operation. As a result, such common forms as $e \uparrow -x \uparrow 2$ may be written without parentheses. Note also the greater freedom allowed the unary minus sign.

Production 19. Conditions like $1 \leqq x < 10$ or $1 < x < y < z$ occur with such frequency that it seems worthwhile to permit the construction.

Production 29. Type and value declarations in procedures are combined for brevity. Specifiers are required for all parameters.

Production 32. The constant declaration assigns to an identifier a constant value. The value must be known during translation, and the expression for the constant may therefore contain identifiers only of standard functions and of previous constants. It is not necessary that a type be assigned to a constant; a type specifies a possible set of values and is redundant for an object having a known value.

Productions 35, 36. A procedure declaration plays much the same role as in ALGOL 60, with the restrictions that each formal parameter must be declared in a specifier and that a procedure declaration does not define a function.

A function declaration defines a mathematical function of the values of its parameters. All variables are locally declared within the statement. A function designator is evaluated by assigning the values of its parameters to the corresponding variables in the name part of the function name and executing the statement. The value of the function is then the final value of the variable having the same name.

Functions may be recursive. A function is not recursive if it refers only to functions occurring earlier in the program, none of which are recursive.

REFERENCES

1. IRONS, E. T. A syntax directed compiler for ALGOL 60. *Comm. ACM 4* (Jan. 1961), 51–55.
2. FLOYD, R. W. An algorithm for coding efficient arithmetic operations. *Comm. ACM 4* (Jan. 1961), 42–51.
3. CHOMSKY, N. On certain formal properties of grammars. *Inform. Contr. 2* (1959), 137–167, 393–395.
4. BAR-HILLEL, Y., PERLES, M., AND SHAMIR, E. On formal properties of simple phrase structure grammars. *Zeit. Phonetik, Sprachwissen. Kommunik. 14* (1961), 143–172.
5. NAUR, P. (Ed.) Report on the algorithmic language ALGOL 60. *Comm. ACM 3* (May 1960), 299–314.
6. CONWAY, M. (Letter to the editor). *Comm. ACM 4* (Oct. 1961), 465.