OBTAINING A GRAMMAR FROM A LESS
FORMAL LANGUAGE DESCRIPTION

By: Marvin C. Paull

WORK DONE UNDER

Contract No. AF 1863

Air Force Office of Scientific Research
Directorate of Mathematical and
Information Sciences
Arlington, Virginia 22209

June 1971

DCS Technical Report #6
Department of Computer Science
Livingston College
Rutgers University
Introduction

In this paper, we will try to show how CF-grammars can be constructed to describe sets of strings; each string being of a type which is commonly found in programming languages. We will require that these grammars not only describe the syntax of such sets of strings, but more importantly, reflect unambiguously in the derivation of these strings, the ordering or sequencing conventions commonly imposed on such strings. We will show how these grammars can be used to formally describe, at least in part, two aspects of programming language-operations and the sequencing of such operations. We will also be concerned with the case with which the grammars can be used to determine this ordering. The grammars we develop will be one of a number of possible grammars, each different in non-superficial ways, and each adequate to express the class of languages we will discuss.

A large part of the material presented here is tutorial. The material on constructing a grammar to satisfy a set of precedence relations constitutes the main contribution reported here.

We start by considering a non-CFG way of describing ordering conventions in an algebraic expression involving a number of appearances of a group of binary infix operations. This technique - precedence - has immediate appeal because of its familiarity as much, perhaps, because of its simplicity. After discussing precedence in some detail, we will show how some of the same effects can be obtained with CFG's.

Precedence is a common device for describing the order in which operations are to be performed in expressions of the form:

\[ a + b \cdot c \quad \cdot \quad b + c \cdot a \ldots \]

It is useful for a language in which we can meaningfully partition the symbols in the language into a set of binary operations which we will designate:

* References together with some discussion of each reference is included at
\[ 0 = \{ 1, 2, \ldots, n \} \] and variables which we will designate \( v \). The set of all strings in the language, called a simple operator language, is given by the expression \((v0)^*v\). Initially, we will consider two precedence relations whose interpretation follows:

a) \( 1 > 2 \) When \( 1 \) appears immediately to the left of \( 2 \), \( 2 \) is to be done after (not necessarily immediately) \( 1 \).

b) \( 2 < 1 \) When \( 1 \) appears immediately to the right of \( 2 \), \( 2 \) is to be done after \( 1 \).

If a set of such precedence relations, \( P \), is specified for a set of operators, \( Q \), we will say that:

1. It is legitimate, if there is no pair \( x, y \in Q \) for which \( x \to y \) and \( x \nottto y \) are both specified.

2. It satisfies the transitivity relation; if whenever \( x \to y \) and \( y \to z \) are specified, so is \( x \to z \), and similarly whenever \( y \nottto x \), and \( z \nottto y \) are both specified, so is \( z \nottto x \).

For example, if \( 1 > 2 \), and \( 2 > 1 \) are both specified, so also must \( 1 > 1 \) be specified.

3. It is complete; if the specified set of relations is such that for every pair of distinct members of \( Q \), \( x \) and \( y \), either \( x \to y \) or \( x \nottto y \), and either \( y \nottto x \) or \( y \to x \) are specified and for each \( x \in Q \) either \( x \to x \) or \( x \nottto x \) is specified.

Unless otherwise noted we will assume that any set of precedences discussed hereafter is legitimate.

For the familiar uses of precedence, as for example, to give the order in which the operations in an algebraic expression are to be evaluated, the implied set of precedences are complete and transitive. For example, the way we normally interpret an algebraic string involving +, * and ** implies
the set of precedence relations given in the following table in which the existence of the relations \( r \) in the row labeled with the operator \( \times \), and the column labelled \( \odot \), implies \( \times r \odot \).

<table>
<thead>
<tr>
<th></th>
<th>**</th>
<th>*</th>
<th>+</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>*</td>
<td>&lt;</td>
<td>(b)</td>
<td>c</td>
</tr>
<tr>
<td>+</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
</tr>
</tbody>
</table>

(a) This relation, **<**, indicates we evaluate a series of exponentiation from the right. This is the FORTRAN convention. ALGOL does it from the left.

(b) On the other hand, this relation, * > *, indicates we evaluate a series of multiplications from the left.

(c) * > + < * together imply that whenever multiplication and addition appear juxtaposed in any special order in an expression, the multiplication must be done before the addition.

The precedence relations tell us how we may legitimately order the evaluation of the operations in an expression. They do not necessarily specify a complete ordering of the evaluation of all operator appearances in a string, but they will impose a complete partial ordering.

To describe what we mean by a complete partial ordering, we will first describe how bracketing may be used to represent a partial ordering. Then we will define a complete bracketing. A complete partial ordering will be the partial ordering represented by a complete bracketing.
Consider first an example of a partial ordering represented by bracketing in the expression: \([a + b] \times [c \cdot d]\] the bracketing \(1 \times 3\) indicates \(1\) and \(3\) must be done before \(2\) but \(1\) and \(3\) may be ordered either way amongst themselves.

In general, partial orderings of the operators of an operator string will be represented by nested bracketing in which each pair of brackets encloses a sub-string of the form \([Q_1 \times Q_2]\] where \(Q_1\) and \(Q_2\) are bracketed sub-strings or \(v\). A string in which all brackets are as described above is a bracketed operator string.

Corresponding to a complete partial ordering of a string, we have a complete bracketing. If in a bracketed operator string there is a sub-string of the form \(\ldots Q_1 \times Q_2 \ldots\) in which the symbol to the left of \(Q_1\) is not \([\) and the symbol to the right of \(Q_2\) is not \(]\), then \(Q\) is not completely bracketed otherwise it is. Whenever we have sub-strings, \([Q_2 [Q_1 \times Q_2]]\) or \([Q_1 [Q_2] \times Q]\), this corresponds to the ordering: operator \(1\) must be done before \(2\).

We now describe more formally the way in which the precedence relations prescribe a complete partial ordering - or in the terms we will use from now on, a complete bracketing.

Given a string from a simple operator language with a complete set of precedences defined on it, we define the process of evaluating a string in the language as follows:

1. Find an operator appearance \(\times\), called a maximal operator appearance, in the string of one of the following forms:
   a. \(Q \times Q \times \ldots\) with \(\times > \times\)
   b. \(\ldots \times Q \times Q \times \ldots\) with \(\times < \times\)
   c. \(\ldots y Q \times Q \times \ldots\) with \(\times > \times\) and \(y < \times\)
in which Q is either a v or a bracketed substring. If none of the above can be found, the process terminates.

(2) When one of the above situations is found, form a bracketed string as follows:

(a) \[ Q \ x \ Q \] (2) ...
(b) \[ \ldots (2) [Q \ x Q] \]
(c) \[ Q \times Q \]

(In each case we say we have bracketed the operator \( \times \)) and return to (1).

I. The evaluating process we have described always results in complete bracketing.

This follows from the following considerations:

(1) After each bracketing, the string remains of the form:

\((Q0)^*Q\) where 0 is now a variable or a bracketed expression.

(2) Number the unbracketed operator appearances from left to right, with increasing integers 1, 2, ... n. For each operator appearance, put a point on the plane, put the point corresponding to the \( j^{th} \) operator appearance to the right and higher, or to the right and lower than the point corresponding to \( j-1^{th} \) operator appearance according as the \( j^{th} \) operator appearance is \( > \) or \( < \) than the \( j-1^{st} \) operator appearance respectively. Draw straight lines connecting adjacent points, unless the two operators have the relation \( x \).
For example, the resultant curve for the following expression is given below the expression.

\[\text{A} \quad 1 \quad \text{B} \quad 2 \quad \text{C} \quad 3 \quad \text{D} \quad 4 \quad \text{E}\]

assuming \(\text{2} > \text{1} < \text{3} > \text{4}\)

This curve has a maximal point - or a local maximum on the curve. Clearly, every such curve must have at least one local maximum (= a maximal operation) - if not, one extreme point, then somewhere in between. It is instructive to take this example a few steps further, i.e. do what is indicated in step 2 above, assuming \(\text{2} \leq \text{4}\) and bracketing the result of \(\text{3}\), we can replot the precedences thus

\[\text{A} \quad 1 \quad \text{B} \quad 2 \quad [\text{C} \quad 3 \quad \text{D}] \quad 4 \quad \text{E}\]

Now \(\text{4}\) is maximal.

Bracketing the result and replotting the precedences we obtain:

\[\text{A} \quad 1 \quad \text{B} \quad 2 \quad [[\text{C} \quad 3 \quad \text{D}] \quad 4 \quad \text{E}]]\]

Continuing in this way we would get finally the bracketing:

\[[\text{A} \quad 1 \quad [\text{B} \quad 2 \quad [\text{C} \quad 3 \quad \text{D}] \quad 4 \quad \text{E}]]]\]

which itself indicates the ordering in which the operations are to be done.

This example illustrates how in general the precedence rules impose an ordering representable as a bracketing (or alternately a tree) on the strings of the language. When we show how this type of language can be
defined by a CFG, the derivation of that CFG will impose essentially the same bracketing on the strings of its language which may then be given the partial ordering interpretation.

A second significant concern is whether the different orders in which we could bracket operators in a given expression, each order of bracketing consistent with a given set of precedences, might lead to different final bracketings. In such a case, the precedence rules would be ambiguous - they would allow different bracketings indicating different partial orderings of the operators in an expression. Although this can happen if two of the operators have no precedence relation between them, the precedence type of description is particularly valuable because of the following property:

II. The bracketing will be the same no matter in what order the evaluating was done - i.e. the brackets were applied.

(1) Consider an expression bracketed in some standard way, say that leftmost operators which can legitimately be bracketed are always bracketed first. The brackets can be classified into all those which contain no other brackets, called 1-level brackets. Those that contain at least one 1-level brackets, but none of higher level, called 2-level brackets. Those that contain at least one (j-1)-level bracket but none of higher level, called j-level brackets. We define the jth level maximal operators as all those operators which are maximal when all the brackets which are of level j or less are on the expression. The 0th level maximal operators are those that are maximal before any bracketing.

II. No matter what order the bracketing is carried out in, there must be brackets around each 0th level maximum operator and its two surrounding
variables. Say \( 0 \) is such an operator and we have ...
\[ x \circ \circ y \ldots \text{ in } E \text{ with } x < 0 > y, \text{ then if } 0 \text{ were not} \]
bracketed within the brackets for \( x \) and \( y \), it would follow that
the brackets for \( x \) or \( y \), say \( x \) for definiteness, must be within
those for \( 0 \) since \( 0 \) shares an argument with \( x \) and \( y \). If
\( x \) brackets are within \( 0 \) brackets, \( x \) must be bracketed before \( 0 \).
This cannot be because that would imply that when \( x \) was bracketed \( 0 \)
was still unbracketed and since \( x < 0 \) we could not legitimately
bracket \( x \) at that time.
Assume that no matter what order the bracketing is done in, all brackets
of levels 0,1,..., \( j \) are identically placed. Consider a \( j \)th level
operator appearance \( 0 \). No matter in what order the bracketing is
done for \( 0 \) the form of the expression near \( 0 \) just before that
bracket is applied will be:
\[
... \circ \circ 0_1 \circ \circ 0_2 \circ \circ b \circ \circ ...
\]
\( Q_1 \) and \( Q_2 \) being \( j \)th or less level brackets, being by hypothesis independent
of the order of bracketing and as a consequence, so must \( a \) and \( b \).
Since \( 0 \) is a \( j+1 \)th bracket in at least one order of bracketing, it must
be that \( a < 0 > b \). So that for any ordering the same bracketing,
i.e.
\[
... \circ \circ [Q_1 \circ \circ 0_2] \circ \circ b \circ \circ ...
\]
will occur for \( 0 \).

III. It follows that a left-to-right process of the following kind will also
result in correct bracketing.
Move along the operator string with a marker until two successive unbracketed
operator appearances are found of the form:
\[
... Q \circ x_j \circ Q \circ (x_{j+1}) \circ Q ... \text{ such that } x_j < x_{j+1}
\]
(to the left of the arrow is $\alpha_j$ $Q$, to the right is $\alpha_{j+1}$)

Then bracket as follows:

$$\ldots \{ Q \ (\alpha_j \ Q) \} \uparrow \alpha_{j+1} \ Q \ \ldots$$

and continue.

Assume that to the left of the arrow, the sequence of unbracketed operators is $\alpha_1 \ \alpha_2 \ \ldots \ \alpha_j$. These must always have the relations:

$$\alpha_j \ > \ \alpha_2 \ > \ \ldots \ > \ \alpha_j$$

This is so because the arrow can never move past an operator $\alpha_{j+1}$ if $\alpha_j < \alpha_{j+1}$.

Thus the process will continue to pick up maximal sub-expressions, and since, no matter what order it does this in, the resulting bracketing will be the same, this process will work.

Another property of this procedure is that to the right of the arrow, there are no bracketed expressions.

Corresponding Grammar:

Can we use a CFG to describe a simple operator language, and somehow to prescribe an ordering equivalent to that described by a given set of simple precedence relations? We can, but for the first class of grammars we present, unless the set of precedence relations of the operators we are considering satisfies the transitivity relation, that grammar will be ambiguous. This does not, however, preclude the possibility that one may find a general technique for building grammars that correspond to any set of precedence relations and are unambiguous. In fact, we will exhibit one later.

Let us construct a grammar for the operator language $(\mathcal{O})^*\mathcal{V}$, $\mathcal{O} = +,*,$ which reflects the precedence $* > +, * > *, + > +, * < *$. 
This set of relations has the transitivity property.

First of all, we know that such a language will at the last step of evaluation be either of the form:

\[ X + Y \]

or:

\[ Z \times W \]

where \( X, Y, Z \) and \( W \) are evaluated expressions. So initially we have:

\[
\begin{align*}
S & \rightarrow A \\
S & \rightarrow P \\
A & \rightarrow X + Y \\
P & \rightarrow Z \times W
\end{align*}
\]

Now, neither \( Z \) nor \( W \) can at their last step of evaluation be sums, i.e. if \( Z \rightarrow A \) were a rule, then the partial derivation:

\[
S \rightarrow [Z \times W] \rightarrow [[A] \times W] \rightarrow [[X = Y] \times W]
\]

which would not be desirable since it violates \( \times > + \) and \( + < \times \).

Similarly, if \( W \) could be a product, i.e. if \( W \rightarrow P \) were a rule, then:

\[
S \rightarrow [Z \times W] \rightarrow [Z \times [P]] \rightarrow [Z \times [[Z + W]]]
\]

would be a possible derivation and that would violate \( \times > + \).

This leaves only:

\[
\begin{align*}
Z & \rightarrow P \\
Z & \rightarrow v \\
W & \rightarrow v
\end{align*}
\]

The fact that \( > + \) precludes a rule of the form \( Y \rightarrow A \).

So finally we obtain:

\[
\begin{align*}
S & \rightarrow A \\
S & \rightarrow P \\
A & \rightarrow X + Y \\
P & \rightarrow Z \times W \\
W & \rightarrow v \\
Z & \rightarrow v \\
Z & \rightarrow P \\
Y & \rightarrow v \\
X & \rightarrow v \\
X & \rightarrow P \\
X & \rightarrow A
\end{align*}
\]

or simplified:

\[
\begin{align*}
S & \rightarrow A \\
S & \rightarrow P \\
A & \rightarrow X + Z \\
P & \rightarrow Z \times v \\
Z & \rightarrow v \\
Z & \rightarrow P \\
X & \rightarrow Z \\
X & \rightarrow A
\end{align*}
\]
In general, if we have the infix operator set \( \{ 1, 2, \ldots, n \} \)
with a complete set of precedences determining the ordering on the string \((v_0)^*v\).
The equivalent grammar is of the following form:
\[ S \rightarrow X_i \mid i = 1 \text{ to } n \]

[operator formats]
\[ X_i \rightarrow L_i \ 1 \quad R_i \mid i = 1 \text{ to } n \]

[left and right allowable parameters]
\[ L_i \rightarrow X_j \mid i, j \text{ such that } 3 > 2 \]
\[ L_i \rightarrow v \mid i = 1 \text{ to } n \]
\[ R_i \rightarrow X_j \mid i, j \text{ such that } 3 < 2 \]
\[ R_i \rightarrow v \mid i = 1 \text{ to } n \]

Now we will try to justify the assertion that, given a string in the language defined by a grammar generated as described from a complete set of precedence relation, that the parse of that string will produce a bracketing consistent with that which would be produced if the string was evaluated (bracketed) by applying the precedence relation directly.

Consider such a string before bracketing. Any operator may be legitimately bracketed using the grammar we have developed, by bracketing an operator, say the operator \( \otimes \) in the context \( \ldots Q_1 \otimes Q_2 \ldots \) in a string. \( Q_1, Q_2 \) being either both \( v \)'s or being bracketed subexpressions with labels \( Q_1 \) and \( Q_2 \) respectively. In order to legitimately bracket such a subexpression, we need the following three rules in our grammar.

1. \( X_x \rightarrow L_x \ 1 \otimes R_x \)
2. \( L_x \rightarrow Q_1 \)
3. \( R_x \rightarrow Q_2 \)
The result would be the bracketing \( \left[ \left[ Q_1 \left\{ L_x \bigcirc \left[ Q_2 \right] R_x \right\} \right] \right]_x \ldots \)

Now for any operator, we will have the rule of the form (1). It is not always clear that we would have the rules (2) and (3) to go along with (1). Initially, however, \( Q_1 \) and \( Q_2 \) will both be \( v \)'s. So for any of the operators in the string, we could initially bracket it. In particular, we could bracket each of the maximal operators. After doing this, it follows that we would again be able to bracket any unbracketed operator because now, although each \( Q_1 \) and \( Q_2 \) is not a \( v \), it is a bracketed subexpression labelled \( X_x \), having on its left and right, operators \( \bigcirc \) and \( \bigcirc \) i.e. either \( \ldots \left[ Q_1 \bigcirc Q_2 \right] X_x \ldots \) or letting \( \left[ Q_1 \bigcirc Q_2 \right] X_x \ldots \) such that \( \bigcirc < \bigcirc \), and \( \bigcirc < \bigcirc \).

This follows because \( \bigcirc \) was a maximal operator. Therefore, letting \( X_x \) be both bracketed subexpressions labelled \( X_x \), it further follows that if we observe any subexpression of the form \( \ldots \left[ \bigcirc X_{x_1} \bigcirc X_{x_2} \bigcirc \ldots \right] \) in the current string, that we will be able to find rules:

\[
\begin{align*}
(1) & \quad X_y \rightarrow L_y \bigcirc R_y \\
(2) & \quad L_y \rightarrow X_{x_1} \\
(3) & \quad R_y \rightarrow X_{x_2}
\end{align*}
\]

(2) and (3) will exist because \( X_1 > \bigcirc \), \( \bigcirc < \bigcirc \), because maximal operators were just bracketed. If we again use the appropriate set of three rules to bracket each of the maximal operators, we can again conclude that in the resulting string we will be able to bracket any unbracketed operator remaining. Continuing in this way, we will be able to completely bracket or parse the entire string.

Now it should be clear from the way we constructed the above bracketing (parse) and referring again to III of the previous section, that the bracketing
obtained above is consistent with that which would be obtained by applying the precedences reflected in the grammar directly to bracket the string.

We have shown the nature of one of the parses possible with the grammars derived from complete sets of precedence relations. In general, however, using such a grammar will result in there being more than one parse for a given string. Only if the special condition that the grammar is generated from a complete transitive set of precedence relations is satisfied, will there be one parse for each string defined by the grammar. First we illustrate the existence of more than one parse or ambiguity when the grammar is generated from a non-transitive set of precedence relations. Consider the relations:

\[
\begin{align*}
1 & > 2 \\
2 & > 1 \\
1 & < 1 \\
2 & < 2
\end{align*}
\]

The corresponding grammar is:

\[
\begin{align*}
S & \rightarrow X_1 \\
S & \rightarrow X_2 \\
X_1 & \rightarrow L_1 \quad 1 \\
X_2 & \rightarrow L_2 \quad 2 \\
L_1 & \rightarrow X_2 \\
L_1 & \rightarrow v \\
L_2 & \rightarrow X_1 \\
L_2 & \rightarrow v \\
R_1 & \rightarrow X_1 \\
R_1 & \rightarrow v \\
R_2 & \rightarrow X_2 \\
R_2 & \rightarrow v
\end{align*}
\]

The string \( v \ 1 \ v \ 2 \ v \ 1 \) if bracketed by the precedence relations directly results in:

\[
[[[v \ 1 \ v] \ 2 \ v] \ 1 \ v]
\]
But it can be parsed using the generated grammar to give either:

\[ [[[[[[v]_{L_1} \mathbf{1} [v]_{R_1} X_1]_{L_2} \mathbf{2} [v]_{R_2} X_2]_{L_1} \mathbf{1} [v]_{R_1} X_2]_{S} \]

which is consistent with the above or:

\[ [[[v]_{L_1} \mathbf{1} [[[v]_{L_2} \mathbf{2} [v]_{R_2} X_2]_{L_1} \mathbf{1} [v]_{R_1} X_1]_{R_1} X_1]_{S} \]

Next we wish to show that there can be only one parse for each string if the grammar corresponds to a set of complete transitive precedences. We know how one parse can be obtained by initially bracketing (parsing) all maximal operators - and then again bracketing all maximal operators. It is easy to see that any parse complete with our grammar must be representable as a process of bracketing not yet bracketed operators, using an appropriate group of three rules of the form discussed above for each bracketing. We will now show that if a non-maximal operator is bracketed at any stage in the bracketing process then one will not be able to complete the process and thus parsing will not be possible. The uniqueness of the "maximal" parse will therefore be confirmed.

Suppose at some stage in the bracketing process we have:

\[ O_1 \mathbf{X}_1 \ldots \mathbf{X}_{j-1} Q_j \mathbf{X}_j O_{j+1} \ldots O_n \mathbf{X}_{n-1} O_n \]

Suppose that \( \mathbf{X}_j \) is not maximal. Suppose first that \( \mathbf{X}_j \) was a minimal operator. Now assume that we bracketed this minimal operator obtaining:

\[ \ldots \mathbf{X}_{j-1} [O_j]_{X_j} \mathbf{X}_j [O_{j+1}]_{R_{X_j}} \mathbf{X}_{j+1} \ldots \]

with \( \mathbf{X}_{j-1} \prec \mathbf{X}_j \), and \( \mathbf{X}_{j+1} \prec \mathbf{X}_j \). It follows that we will never be able to bracket \( \mathbf{X}_{j-1} \) nor \( \mathbf{X}_{j+1} \) because although there are rules:

\[ X_{X_{j-1}} \rightarrow L_{X_{j-1}} \mathbf{X}_{j-1} R_{X_{j-1}} \]

and

\[ X_{X_{j+1}} \rightarrow L_{X_{j+1}} \mathbf{X}_{j+1} R_{X_{j+1}} \]
but neither of these:

\[ L_{x_{j+1}} \rightarrow x_j \quad \text{because} \quad x_j < x_{j+1} \]

\[ R_{x_{j-1}} \rightarrow x_j \quad \text{because} \quad x_{j-1} > x_j \]

Notice that this fact does not depend on the transitivity of the precedence relations. However, the next point that must be made does depend on transitivity.

Assume now that \( x_j \) is neither minimal nor maximal. We may then assume further, without loss in generality.

\[ x_{j-1} > x_j > x_{j+1} > \ldots > x_{j+k} < x_{j+k+1} \]

That is, that the precedence continually decreases as we move right until a minimum operator, \( x_{j+k} \) is reached, either because it is the rightmost operator, or because \( x_{j+k} < x_{j+k+1} \). If we bracket \( x_j \) we will not be able to complete the bracketing of the string. Because, having bracketed \( x_j \), we will not be able to bracket \( x_{j-1} \) with the bracketed \( x_j \) as its right parameter, i.e. there is no rule \( R_{x_{j-1}} \rightarrow x_j \) because \( x_{j-1} > x_j \).

Thus the bracketed \( x_j \) will have to be the left parameter in the bracketing of \( x_{j+1} \). After bracketing \( x_{j+1} \), again we can conclude that we cannot bracket \( x_{j-1} \) with the just-bracketed \( x_{j+1} \) as its right parameter, i.e. there is no rule \( R_{x_{j-1}} \rightarrow x_{j+1} \) because by transitivity \( x_{j-1} > x_{j+1} \).

This argument can be continued to show that we will have to always bracket \( x_{j+1} \) with a bracket labelled \( x_{j+i} \) as its left parameter, if \( x_{j+i} \) is to be bracketed at all. At each of these steps, \( x_{j-1} \) will remain un-bracketed, and by transitivity, \( x_{j-1} > x_{j+i} \). So we will continue to bracket non-maximal operators until we are forced to bracket a minimal operator, i.e. \( x_{j+k} \). By our previous argument then we will not be able to complete the bracketing.
Non-Transitive Relations

If our set of precedence relations is not transitive, we can still obtain an unambiguous CFG which will define the corresponding operator language. The simple operator language example we have been considering with

\[ 1 > 2 > 1 < 1, \quad 2 < 2 \]

can be shown to be defined unambiguously by the grammar given at the end of this section.

The fact that there does exist a non-ambiguous grammar for any complete legitimate set of precedences is strongly suggested by a theorem of automata theory. This theorem states that any language which can be recognized by a deterministic pushdown automata can be defined by an unambiguous context-free grammar.*

A simple operator language defined with a complete legitimate set of precedences can be recognized by a pushdown automata - given essentially by the left-to-right bracketing algorithm discussed earlier. Therefore, there is a recognition grammar. The theorem is constructive and we can obtain that grammar - however we cannot guarantee that the grammar will bracket the strings of the language in the desired way. So we develop the unambiguous grammar which will do so independent of that theorem.

We will use names for our non-terminal symbols which are intended to describe the set of terminal strings, derivable from that non-terminal symbol. Let the set of operators in our language be:

\{ 1, 2, \ldots, n \}. The NTS \( j_k \) will be the NTS from which all strings in which \( j \) is the leftmost operator and \( k \) is the rightmost operator, i.e. strings of the form \( v j \ldots k v \), are derivable. Each rule with \( j_k \) on the left will be the first rule in the derivation of each terminal string of a subset of all the strings of the \( v j \ldots k v \).

* Reference 4, Chaps. 5 and 12.
For each pair of operators \( j_k \) , \( i_k \) we will have a rule of the form \( j_k X_i \rightarrow L_{jki} \) \( R_{jki} \) . For each operator \( i_k \) . In determining the \( R_{jki} \) and \( L_{jki} \) - rules we require that each string derivable from \( L_{jki} \) have as its first operator \( j_k \) , and \( R_{jki} \) have as its last operator \( k \) . Furthermore, we require that a string containing at least one operator can only be derived from \( L_{jki} \) if the last operator in that string is \( p \) and \( p > i \) . Similarly, a string containing at least one operator can only be derived from \( R_{jki} \) if its first operator is \( e \) , such that \( i < e \) . This information is sufficient to describe how we will define the grammar to handle the situation when \( L_{jki} \) and \( R_{jki} \) derive sets of strings each of which involves at least one operator. For the situation in which \( L_{jki} \) and \( R_{jki} \) might derive simple variables, we require that the rule \( L_{jki} \rightarrow v \) exist iff \( i_k \) is identical to \( j_k \) ; since if \( L_{jki} \) derives \( v \) then we have \( j_k X_i \rightarrow L_{jki} \) \( R_{jki} \rightarrow v \) \( i_k \) \( R_{jki} \) and therefore the leftmost operator in the resultant string is \( i_k \) . Since we wish to only derive strings whose leftmost operator is \( j \) \( j_k \), we must include the rule \( L_{jki} \rightarrow v \) iff \( i = j_k \) . Similarly, the rule \( R_{jki} \rightarrow v \) is included iff \( i = k \) .

Summarizing our procedure for constructing the precedence grammar:

\[
G: V_{T_1} = \{ 1 , 2 , \ldots , n \} \ , \ V = V_{T_1} \cup \{ v \} \\
V_n = \{ j_k X_i \} \ : \ ( j_k , i ) \in V_{T_1}^2 \}
\]

Rules:

\[
\{ j_k X_i \rightarrow L_{jki} \} \quad \{ i_k R_{jki} \} \ : \ ( i_k , j_k , i_k ) \in V_{T_1}^3 \\
\{ L_{jki} \rightarrow X_e \} \ : \ e > i_k \\
\{ R_{jki} \rightarrow X_p \} \ : \ i_k < p_k \\
\{ L_{jki} \rightarrow v \} \ : \ i = j_k \\
\{ R_{jki} \rightarrow v \} \ : \ i = j_k 
\]
Since the rules for $L_{ijk}$ are independent of $k$, and similarly those of $R_{ijk}$ are independent of $j$, we can rewrite these rules as:

$$
\{ jX_k \rightarrow L_{ij} \} \quad R_{ik} : (i, j, k) \in V_0^3 \\
\{ L_{ij} \rightarrow jX_e \} \quad \circ \quad \triangleright \quad i \\
\{ R_{ik} \rightarrow pX_k \} \quad \circ \quad \prec \quad p \\
\{ L_{ij} \rightarrow v \} \quad \circ \quad \equiv \quad j \\
\{ R_{ik} \rightarrow v \} \quad \circ \quad \equiv \quad k
$$

Or again rewriting $jL_i$ instead of $L_{ij}$ and $iR_k$ instead of $R_{ik}$, we have finally:

$$
\{ jX_k \rightarrow jL_i \} \quad i \quad R_k : (i, j, k) \in V_0^3 \\
\{ jL_i \rightarrow jX_e \} \quad \circ \quad \triangleright \quad i \\
\{ iR_k \rightarrow pX_k \} \quad \circ \quad \prec \quad p \\
\{ jL_i \rightarrow v \} \quad \circ \quad \equiv \quad j \\
\{ iR_k \rightarrow v \} \quad \circ \quad \equiv \quad k
$$

From our construction of the grammar we can see that in bracketing an operator we would use a rule of the form: $jX_k \rightarrow L_{ijk} \quad i \quad R_{ijk}$.

From the way we have obtained the rules for $L_{ijk}$ and $R_{ijk}$ we can see that such a rule would only be applicable for bracketing $i$ if the operators to the left and right of $i$ are respectively in their positions relative to $i$, higher precedence than $i$.

This grammar, $G$, does have rules:

$$
\begin{align*}
iX_j \rightarrow L_{iii} & \quad i \quad R_{iii} \\
L_{iii} \rightarrow v \\
R_{iii} \rightarrow v
\end{align*}
$$

for each operator $i$ in $G$. We might therefore initially bracket any operator (with its two surrounding variables) in a given string $s$. But assume we bracket an operator $i$ initially and there is an operator on
its left, say \( k \), and \( k > i \). I.e. we have at some stage in the bracketing: \( \ldots \ k \ [v \ i \ v] \ldots \), with \( k \) not yet enclosed in brackets. In order to enclose \( k \) in brackets at some later stage, we will need rules:

\[
\begin{align*}
(1) & \quad a_h \rightarrow L_{kab} \ k \ R_{iab} \\
(2) & \quad R_{kab} \rightarrow i \ x \ b
\end{align*}
\]

but since \( k > i \) and therefore by legitimacy, it is not true that \( k < i \) and rule 2 will not exist. It follows by extension of this argument to analogous situations that if we legally bracket an operator according to a rule of \( G \) (with it surrounding bracketed expressions or variables) at some stage and the operator on the right or left of this bracketed expression is of lower relative precedence, we will not be able to complete the bracketing. On the other hand, from the exhaustive construction of the grammar we have allowed bracketing of an operator whenever its surrounding unbracketed operators are of relatively lower precedence. These arguments lead to the justification that given any legitimate set of precedences between every pair of operators in a set of \( n \) operators, we can construct an unambiguous CFG which will correspond to the simple operator language thus defined. The construction given above will produce that grammar.

Example:

Given operators \{ 1, 2 \} with \( 1 < 1 \), \( 2 < 2 \), \( 1 > 2 > 1 \)
Then the grammar $G$ which unambiguously defines the corresponding simple
operator language is:

$$
G: \quad V_T = \{ 1, 2 \}, \quad \{ , \}
$$

$$
V_N = \{ X_1', X_1, X_2', X_2 \}, S
$$

and the productions are:

$S \rightarrow v | 1X_1' 1X_1 2X_2 2X_2$

$1X_1' \rightarrow 1L_1 \quad 1R_1 \rightarrow 2L_2 \quad 2R_2$

$1X_2' \rightarrow 1L_1 \quad 1R_2 \rightarrow 2L_2 \quad 2R_1$

$2X_1' \rightarrow 2L_1 \quad 2R_1 \rightarrow 2L_2 \quad 2R_2$

$1L_1 \rightarrow 1X_2' \quad 1L_2 \rightarrow 1X_1$

$2L_1 \rightarrow 2X_2' \quad 2L_2 \rightarrow 2X_1'$

$1R_1 \rightarrow 1X_1 \quad 1R_2 \rightarrow 1X_2$

$1R_2 \rightarrow 1X_2 \quad 2R_1 \rightarrow 2X_1$

$2R_2 \rightarrow 2X_2$

Since from $1L_2$ we can derive the identical strings from $1L_1$ as from
$1X_1$, we can replace every right hand occurrence of $1L_2$ by $1X_1$. Making
all such similar replacements the productions become:

$1X_1' \rightarrow 1L_1 \quad 1R_1 \rightarrow 1X_1 \quad 2X_1$

$1X_2' \rightarrow 1L_1 \quad 1R_2 \rightarrow 2L_2 \quad 2R_2$

$2X_1' \rightarrow 2X_2 \quad 2R_1 \rightarrow 2L_2 \quad 2R_2$

$2X_2' \rightarrow 2X_2 \quad 1X_2 \rightarrow 2L_2 \quad 2R_2$

$1L_1 \rightarrow 1X_2' \quad 2L_2 \rightarrow 2X_2'$

$1R_1 \rightarrow 1X_1 \quad 2R_2 \rightarrow 2X_2$
References:

Precedence grammars were first defined by Floyd. He also showed how to determine if a given grammar was a precedence grammar. We include here a consideration of the inverse problem: i.e. given the precedences find the grammar.

This work was generalized by Weber and Werth

A good survey of work in this area together with an extensive bibliography is contained in:

A general reference on context-free grammar is