Generalized Overlap Resolvable Grammars
and their Parsers*

DAVID S. WISE†

Department of Computer Science, University of Edinburgh, Edinburgh, Scotland EH93 JZ

Received March 3, 1972

The class of GOR grammars admits ε-rules and includes a grammar for every
deterministic language. Its simple decision procedure yields pairs of problematic
phrases, if the grammar is not GOR, or tables used to drive a deterministic push-
down parser very rapidly. The parsing algorithm, based on one of Domolki, takes
advantage of the architecture of binary computers by computing state transitions
quickly with logical operations. These computations can be used by a pre-processor to
compute an actual state transition table, if desired. This extension yields a still faster
parser which can be abandoned temporarily by reverting to the state computing
algorithm at any time, if the original grammar and relations need to be modified.

I. INTRODUCTION

Since the approach to definition of ALGOL 60 as the language of a context free
grammar, the mechanization of parser-building for subsets of the context free gram-
mars has become a valuable tool for compiler writing. This tool, itself, has been
formalized by the definition of tranducers [12] as a useful extension of the classic
theoretic models of automata which can only accept or reject input strings.

Classes of grammars (hereafter we presume context-free) have been defined with
three attributes: membership is decidable, every member grammar is unambiguous,
and there is an algorithm to build a practical parser. Because ambiguity for a general
context free grammar is undecidable, the membership problem acts as a semidecision
procedure to screen out the undesirable ambiguous grammars. More important, the
automatically generated parser reduces the problem of interpreting the language to
the semantic pieces associated with each production in the syntax.

Because it is guaranteed to operate in linear time (proportional to the length of the

* This paper is based on the author's dissertation at the University of Wisconsin [17], which
was presented in a preliminary form at the Third Annual ACM Symposium on the Theory of
Computing. (Supported by U.S. Army under contract DA-31-124-ARO-D-462.)
† Present address: Computer Science Department, University of Indiana, Bloomington,
IN 47401.

Copyright © 1972 by Academic Press, Inc.
All rights of reproduction in any form reserved.
sentences), the deterministic push-down acceptor [7, 9] has become a popular frame on which to build parsers. We discuss herein only bottom-up parsers, which parse a sentence by deriving, in reverse order, successive rightmost sentential forms. At any point during the process the push-down store contains information derived from the initial segment of a rightmost sentential form including all nonterminals. On the unread input tape are terminal characters and the end of the leftmost reducible phrase. The parser must locate the next handle and reduce it so that the top of the push-down store describes the prefix of the next rightmost sentential form. The information necessary for this action must be available within the top few symbols on the stack, and from inspection of the next few terminal symbols (or end markers) beyond the proposed handle. If the parser is successful, the start symbol is the last sentential form derived.

Below we shall informally relate the generalized overlap resolvable (GOR) grammars to other published classes of grammars on the basis the decisions of handle identification in such an environment. Except for precedence parsers, all classes mentioned below, including GOR, have parsers which stack more information than a single character.

With a GOR grammar, a handle (including possibly an empty handle) is identified by matching the right side of a production character for character, by ascertaining that the proposed handle is preceded by a character from a set of admissible predecessors, and that it is followed by one from a set of admissible successor characters. The sets of predecessors and successors for a production are mutually independent and based only upon the left-hand-side of the production.

We contrast this power with that of Bounded Right Context (BRC) parser. (By Floyd's definition [6] we are here discussing BRC (1,1),) Under that structure a handle is also identified by a character-for-character match with the right side of a production plus a context check of one character left and right. However, these two context characters must together match one of the pairs associated with that production. These pairs need not form the complete Cartesian products of the set of left context characters and set of right context characters, as is implied by the GOR context.

In an LR(1) parser [11] all information to the left of a handle, within the handle itself, and for one character beyond is available to identify a handle. The information is carried to the top of the push-down-store by providing an elaborate state transition table, which is simplified considerably in the SLR(1) parser [4]. In that algorithm the one character to the right of a proposed handle must only be in the set of permissible right context characters for the production being reduced—independently of what occurs to the left. This test is very similar to the right compatibility check of GOR.

In the precedence schemes a reducible phrase is located, without any character matching, by binary relations defined over the vocabulary. Under invertible simple precedence (ISP) [16] the phrase is isolated by such relations, and a character-for-character scan of the handle can identify the production because all right-hand-sides
are unique. Although the right end of the leftmost reducible phrase is delimited by weak precedence relations (WP) [10], the character scan of it is required to identify its left end, and the production to be reduced, for all right-sides are unique. Under simple mixed strategy precedence (SMSP) [1] identical right-sides are admitted into WP if they can be distinguished by one context character to the left of the handle in question. The test applied for that case is the same as the left check which is always made by an overlap resolvable parser, and so SMSP is the closest of the classes of precedence grammars to GOR.

The discussion above is necessarily sketchy. It says nothing of the decision procedure to see if a grammar is in the class or of the action of the parser on sentences not in the language of the grammar being parsed (i.e., error detection). Moreover, many of the parsers can handle grammars which are not in the class with which they are associated. However, the description should intuitively fit GOR into its place with respect to other grammar classes which we shall discuss later more formally. In particular, the set of GOR grammars contains (properly) the sets of precedence grammars, and is contained in the set of BRC, and hence the set of LR(1) grammars. The difficulty of implementing the GOR decision procedure and parser reflects this containment—more difficult than precedence but easier than the others.

The GOR class of grammars is an extension of Lynch's Overlap Resolvable [13] grammars to allow $\epsilon$-rules (empty right sides). The parser is based on a "bit smashing" algorithm of Domolki [5] which is improved by halving its running time and decreasing space required. Further observations on its behavior make it a flexible state-transition parser.

II. NOTATION AND PRELIMINARY DEFINITIONS

Lower case Roman letter denote integers and characters in a vocabulary. The letters early in the alphabet ($a, b, c, e$) denote terminal symbols; the intermediate letters ($h, i, j, ..., n$) denote indices or integers; the last letters denote arbitrary or nonterminal symbols.

Upper case Roman letters refer to sets and matrices.

Lower case Greek letters will be used for strings and relations. The relations we shall define are $\alpha, \beta, \lambda,$ and $\rho$; since these are defined over characters there will be no confusion with other strings; all other Greek letters denote strings. Of special importance is $\epsilon$, which denotes the empty string. For the empty set we use $\emptyset$.

For a set of characters $V$, called a vocabulary, the set of all strings over $V$ will be denoted $V^*$, and $V^+$ will denote all of $V^*$ but $\epsilon$. If $\xi$ and $\zeta$ are strings then the length of $\xi$ will be denoted $|\xi|$ and the concatenation of $\xi$ and $\zeta$ by $\xi\zeta$. For $\zeta$ and any integer $0 \leq i \leq |\zeta|$ we adopt the notation $i : [\zeta]$ for the prefix string of the first $i$ characters of $\zeta$, and $[\zeta] : i$ for the suffix string of the last $i$ characters of $\zeta$. Thus $0 : [\zeta] = \epsilon =$
[\zeta] : 0 and [\zeta] : | \zeta | = \zeta. If i is out of bounds the meaning is undefined. If | \zeta | = 1 then \zeta may be viewed as a string (\zeta \in \mathcal{V}^*) or a character (\zeta \in \mathcal{V}) as context requires; conversely, all characters are string of length 1. Therefore, we can write [\zeta] : 1 \in \mathcal{V} if \zeta \neq \epsilon, and also \zeta a, which is the string \zeta with an a added at the end. We denote string membership as x \in \zeta, which is defined as

\[
x \in \zeta \iff \exists 1 \leq i \leq | \zeta | \ (x = [i : [\zeta] : 1]).
\]

When we deal with binary relations we will use the infix notation. If x or y denotes an undefined expression then \text{xp} y is always false (due to an undefined operand). If there is some z such that \text{xz} y and \text{xz} y then we may write \text{xpy} or \text{xp^2y}. Extending this notation, the transitive closure of \rho is denoted \rho^+ and the reflexive and transitive closure by \rho^*. We shall denote the inverse of a relation \rho by \rho^T : xpy iff yp^Tx. This notation is suggestive of the Boolean matrix which represents the inverse of \rho; it is the transpose of the matrix for \rho.

A context free grammar is a quadruple \( G = (\mathcal{V}_N, \mathcal{V}_T, s, P) \) where

\[
\begin{align*}
\mathcal{V}_N & \text{ is a set of nonterminal characters;} \\
\mathcal{V}_T & \text{ is a set of terminal characters;} \\
\mathcal{V}_N \cap \mathcal{V}_T & = \emptyset; \quad V = \mathcal{V}_N \cup \mathcal{V}_T; \\
V & \text{ is the vocabulary of } G; \\
s \notin V & \text{ is the start symbol of } G; \\
P & \subseteq (\mathcal{V}_N \cup \{s\}) \times V^* \text{ is the set of productions.}
\end{align*}
\]

We shall often write a production \((y, \xi)\) as \(y \Rightarrow \xi\). The \(\Rightarrow\) relation is defined below but it will be seen that \((y, \xi) \in P\) implies \(y \Rightarrow \xi\). In this example, \(y\) is a left part and \(\xi\) is a right part or phrase.

We index \(P\) by \(1 \leq n \leq | P |; (y_n, \xi_n)\) is the \(n\)-th production; \(\xi_n\) the \(n\)-th right part; \(y_n\) the \(n\)-th left part. The left and right endmarkers, \(\leftarrow\) and \(\rightarrow\), not in \(V\), are introduced by an implicit 0-th right part: \(\xi_0 = \leftarrow s \rightarrow\). A notation to describe the individual characters in any phrase is \(\xi_n = z_{n,1} z_{n,2} \ldots z_{n,|\xi_n|}\) defined by

\[
z_{n,i} = [j : [\xi_n] : 1].
\]

If \(\xi_n = \epsilon\) we call \((y_n, \xi_n) \in P\) an e-rule. If \(| \xi_n | = i\) we call \((y_n, \xi_n)\) an i-production. Hence e-rules could be called 0-productions.

\(P\) yields a relation on \((\mathcal{V} \cup \{s\})^*\) denoted by \(\Rightarrow\) where

\[
\zeta \Rightarrow \sigma \iff \zeta = \psi y \phi \quad \text{and} \quad \sigma = \psi \xi \phi,
\]

where \(\psi, \phi \in (\mathcal{V} \cup \{-\leftarrow, \rightarrow\})^*\) and \((y, \xi) \in P\).

571/6/6-5
We now define a special meaning for $\rightarrow$ which is motivated by the possibility of $\epsilon$-rules: $\xi \rightarrow \sigma$ iff the definition for $\rightarrow$ above holds with $\xi \neq \epsilon$.

We say $\zeta \Rightarrow_{rt} \sigma$ iff the above for $\Rightarrow$ holds, and $\phi \in (V_T \cup \{-\})^*$. By applying the notation for transitive closures of relations we get $\Rightarrow$, $\Rightarrow_{rt}$, $\Rightarrow_{rtt}$, etc. With these "arrow" relations we can easily describe rightmost derivations: $s \Rightarrow_{rt} \sigma$ and $\epsilon$-free derivations: $\zeta \Rightarrow_{rt} \sigma$.

The set of rightmost sentential forms of $G$, denoted $SF(G)$, is $\{\sigma \in V^* \mid s \Rightarrow_{rt} \sigma\}$. The language defined by $G$, $L(G)$, is $SF(G) \cap V_T^*$. We shall assume $G$ is reduced, that is, $\forall y \in V_N (\exists \zeta \in V_T^*(y \Rightarrow \zeta))$ and $\forall z \in V (\exists \sigma \in SF(G)(z \in \sigma))$.

We can derive three relations from $P$; $\alpha$, $\lambda$, and $\rho$ which are read "adjacent to", "left derives", and "right derived from" respectively [2]: the right column below diagrams the definitions.

$$
\alpha uv \iff \exists 0 \leq n \leq |P| (\exists 1 \leq i < |\xi_n|) (u = z_{n,i} \land v = z_{n,i+1}).
$$

$$
\lambda uv \iff \exists 1 \leq n \leq |P| (u = y_n \land v = 1: [\xi_n]).
$$

$$
\rho uv \iff \exists 1 \leq n \leq |P| (u = [\xi_n]: 1 \land v = y_n).
$$

Note the asymmetry of $\lambda$ and $\rho$.

These definitions allow a concise definition [16, 19] of invertible simple precedence (ISP): A grammar $G$ is ISP if $G$ has no $\epsilon$-rules, all phrases in $G$ are unique, and the relations $\alpha$, $\alpha \alpha +$, and $\rho + \alpha +$ defined on $G$ are disjoint.

Because we allow $\epsilon$-rules, there will be need to apply similar relations across sub-strings which disappear ($\Rightarrow \epsilon$).

$$
\alpha uv \iff \exists 0 \leq n \leq |P| (\exists 1 \leq i < |\xi_n|)
(\exists 1 \leq j \leq |\xi_n| - i)
(u = z_{n,i} \land v = z_{n,i+1} \land
[(i + j - 1): [\xi_n]]: j - 1 \Rightarrow \epsilon).
$$

$$
\lambda uv \iff \exists 1 \leq n \leq |P| (\exists 1 \leq i \leq |\xi_n|)
(u = y_n \land v = z_{n,i} \land
(i - 1): [\xi_n] \Rightarrow \epsilon).
$$

$$
\rho uv \iff \exists 1 \leq n \leq |P| (\exists 1 \leq i \leq |\xi_n|)
(u = z_{n,i} \land v = y_n \land
[\xi_n]: (\exists 1 \leq i \Rightarrow \epsilon)).
$$

Thence we define the relations $\beta$ and $\beta$ (read "by") on $V_N \times (V_T \cup \{-\})$. 


The restriction on the domain of $\beta$ will be important. While $\alpha$ is defined over
$$(V \cup \{-\}) \times (V \cup \{-\})$$
we need only define $\lambda$ and $\rho$ over $V_N \times V$ and $V_N \times V_N$, respectively, if we do not neglect to add in equality (on $V$) to $\lambda^*$ and $\rho^*$. This will save space in implementation also.

The relations $\rightarrow, \alpha, \lambda, \rho, \beta$ have the same meaning as $\Rightarrow, \bar{\alpha}, \bar{\lambda}, \bar{\rho}, \bar{\beta}$ but they are defined by excluding $\epsilon$-rules.

### III. GOR AND OTHER GRAMMAR CLASSES

The definition of GOR grammars below is really a decision procedure. Once the relations $\alpha \lambda^*, (\alpha \lambda^*)^T \alpha \lambda^*, \beta \lambda^* T, \beta \lambda^* T, \beta \beta^T \beta$ have been computed, productions can be easily tested pairwise to locate each overlap, classify it, and check that the appropriate relations imply it is resolvable.

Suppose two phrases have a substring in common. When this substring occurs in a sentential form, a parser (under our model) must be able to choose to which of the two phrases the substring belongs, as the end of either phrase occurs. The obvious example is the case where two phrases are identical. We shall give some example diagrams of "overlap" and then define them formally.

```
ab abcde abcde
\| ;  || ;  ||| ;
ab cd abc

aaa abcd aaa
\| ;  || ;  || .
aaa cdefg aaa
```

The empty phrase $\epsilon$ is also included in the definition of overlap, but when it overlaps with a non-$\epsilon$ phrase it does so "internally":
```
\epsilon a bc ab c
\| ;  \| ;  \| .
\epsilon \epsilon \epsilon
```
The following diagrams are not pictures of possible overlaps:

\[
\begin{array}{ccc}
abc & abc & abc \\
\mid & \mid & \mid \\
\epsilon & \epsilon & dbf
\end{array}
\]

The last example is not an example of an overlap because the common substring of an overlap must extend to the extreme of one phrase.

We define \( h \) to be the maximum phrase length, for any grammar \( G = (V_N, V_T, s, P) \):

\[ h = \max_{1 \leq n \leq |P|} |\xi_n|. \]

A \textit{triple} is an element of the set

\[ \{0 \leq m \leq |P|\} \times \{1 \leq n \leq |P|\} \times \{0 \leq i \leq h\}. \]

**DEFINITION.** The triple \((m, n, i)\) is an overlap (Fig. 2) if

\[ |\xi_n| : [\xi_m] : (|\xi_n| + i) = \xi_n \]

or alternatively,

\[ (|\xi_m| - i) : [\xi_m] = [\xi_n] : (|\xi_m| - i) \]

and one of the four conditions below holds:

1. \(|\xi_m| = |\xi_n| \geq i = 0 \) and \( m < n \);
2. \(|\xi_m| > |\xi_n| > i = 0 \);
3. \(|\xi_m| > i > 0 = |\xi_n| \) specifically excluding \( m = 0 \) and \( i = 1 \);
4. \(|\xi_m| > i > 0 < |\xi_n| \).

The definition first says that substrings of \( \xi_n \) and \( \xi_m \) are identical. The four numbered predicates restrict the way in which the phrases \( \xi_m \) and \( \xi_n \) are aligned; then it is the "lapping" substring which must match. This definition is not symmetric in \( m \) and \( n \).

The only way that \( \xi_0 \) can be part of an overlap is in \((0, n, 2)\) where \( \xi_n = \epsilon \), a condition 3 overlap which is designated condition 3a. These "\( \epsilon \)-overlaps" will be treated specially in the definition of right resolvability below.

The pictorial meaning of the definition of overlap is that lining up \( \xi_m \) and \( \xi_n \) so that the last \( i \) characters of \( \xi_m \) extend to the right of \( \xi_n \) (see Fig. 1.) causes the "overlapping" characters to match. The four disjuncts (numbered) in the definition of overlap will be known as conditions 1-4 and are elaborated in Fig. 2. It should be noted that for condition 1, 2, and 3 triples, the first conjunct (unumbered) in the
definition of overlap collapses to $\xi_m = \xi_n$, $[\xi_m]: |\xi_m| = |\xi_n|$, and vacuously true, respectively.

The essential problem with an overlap is the following dilemma: To which of $\xi_m$ or $\xi_n$ does a particular sequence of characters, which may occur in a sentential form and which is common to both (the "overlapping" characters) belong? If the dilemma can always be resolved from the knowledge of one character to the right or one to the left of the common sequence, then a choice is always possible after the string is completely read.

If $\xi_m = \xi = \xi_n$ then we know that between adjacent character pairs within $\xi$ the $\alpha\lambda^*$ relation holds, and within $\omega$ the $\rho^*\lambda^*$ relation holds. In particular $[\psi]: 1 \alpha\lambda^*\alpha$ and $x\beta 1:[\omega]$. In order to resolve overlaps we shall make use of this fact.

A triple is even left iff $|\xi_m| = |\xi_n| + i$ and even right iff $i = 0$. As elaborated in
Fig. 2 condition 1 and 4b are even left; the rest are uneven left. Condition 1 and 2 are even right; the rest are uneven right.

If an overlap is uneven left, then it is resolvable if the \( \alpha \lambda^* \) relation does not hold in a form which would allow the last character in the "overhang to the left" to precede the handle in a rightmost derivation. If the overlap is even left, then it is resolvable if the sets of those allowable preceding characters for the two left parts are disjoint.

**DEFINITION.** A triple \((m, n, i)\) is defined to be resolvable on the left iff

\[
-((1 \leq j \leq i \land \lambda_m(\alpha \lambda^*)^r \alpha \lambda^* y_n) \lor 1 : [\xi_m : (1 \xi_n + i + 1)] \alpha \lambda^* y_n \\
\lor 1 : [\xi_n : (1 \xi_m - i + 1) \alpha \lambda^* y_m]).
\]

This definition applies for all triples; so we may speak of left resolvability even when the triple is not an overlap. In practice, we are only interested in overlaps however. Recall that relations on undefined operands are defined to be false; this convention will always apply to at least two of the three disjuncts in the above definition.

The concept of right resolvability is derived in a similar fashion from the situation \( z_\beta 1 : [\omega] \). However, since \( \epsilon \)-rules may occur (as yet unrecognized) to the right of the handle, uneven right overlaps are resolved in a complicated way. Define a predicate:

\[
R(m, n, i, j) = j[\xi_m : i] \not\geq \epsilon \land (i = j \land \lambda_n \beta \lambda^* y_m \lor \lambda_n \beta \lambda^* y_m 1 : [\xi_m : (i - j)]).
\]

The interpretation of this predicate will be understood from the illustration of the overlap \((m, n, i)\) in Fig. 3 and observing it is false unless \(0 \leq j \leq i\). The character of interest in phrase \( \xi_m \) is \( x = 1 : [\xi_m : (i - j)]\). Everything between the end of the match with \( \xi_n \) and \( x \) could possibly disappear \((\not\geq \epsilon)\) in a sentential form, so that \( x \) could appear adjacent to the end of the match. If this uneven right overlap occurs we are interested in the validity of \( \lambda_n \beta \lambda^* x \). Compare this to the comment \( z_\mu 1 : [\omega] \). Since \( x \) is not always a member of \( \lambda \beta \varepsilon \) we must allow for \( \lambda \beta \varepsilon a \in \lambda \beta \varepsilon \) and \( z_\mu a \).

Recall that \( \beta \) is defined on \( \lambda \beta \varepsilon \), which is consistent with the use of \( \beta \) here.
When \( i = j \), \( x \) is undefined. This indicates an even-right triple whose resolvability will hinge upon \( \gamma_n \beta \beta^* \gamma_m \). For any one triple \((m, n, i)\) we shall check \( R(m, n, i, j) \) for all \( j \), \( 0 \leq j \leq i \), because the various substrings of \( \xi_m \) following the match with \( \xi_n \) may each have potential to disappear in a sentential form, with the exception \( \xi_n = \epsilon \). In the case \( \xi_m \neq \epsilon = \xi_n \), any overlap is uneven right (and uneven left). There is no need to consider local effects of \( \epsilon \)-rules on the right in this case. If they cause a problem, it will show up as an unresolvable condition 1a overlap on the \( n \)-th production. For condition 3 overlaps, right resolvability is restricted to ignore such effects.

**Definition.** A triple \((m, n, i)\) is *resolvable on the right* iff

1. \( |\xi_m| \geq i > |\xi_n| = 0 \) and \( -\gamma_n \beta \lambda^+ \tau_1 : i \) or
2. \( -i (|\xi_m| \geq i > |\xi_n| = 0) \) and \( \forall 0 \leq j \leq i (-R(m, n, i, j)). \)

The first part applies to condition 3 overlaps; the second part to all other overlaps.

**Definition.** A triple is *resolvable* if it is resolvable on the right or resolvable on the left.

**Definition.** A grammar is *generalized overlap resolvable* (GOR) if all overlaps of the grammar are resolvable.

**Definition** [13]. A grammar is *overlap resolvable* (OR) if it is GOR and has no \( \epsilon \)-rules.

If a particular grammar has no \( \epsilon \)-rules, the OR test and parser can be greatly simplified from the GOR requirements. We can ignore condition 1a and 3 overlaps, replace \( \beta \) and \( \lambda \) by \( \beta \) and \( \lambda \) everywhere, and collapse the definition of right resolvability of \((m, n, i)\) to \(-R(m, n, i, 0)\). However, if \( \epsilon \)-rules are included, condition 3 overlaps occur "everywhere" because of the vacuous phrase match. This means that the \( \alpha \lambda^* \) and \( \beta \) relations on the left-parts of such rules ought to be seldom true. If not, the likelihood that the grammar is GOR is severely reduced. While \( \epsilon \)-rules are useful when there are specific semantic requirements, one must be aware that one \( \epsilon \)-rule may generate many unresolvable overlaps.

It should now be clear that every instance in which two phrases have a (nonempty) extreme substring in common has been described by an overlap. Perhaps it is not so clear why triples \((m, n, 0)\) and \( m, n, |\xi_m| \) are excluded from being overlaps when \( \xi_n = \epsilon \neq \xi_m \). If these triples are unresolvable, either there is another overlap which is unresolvable, or the parser makes the right choice anyway. The parsing algorithm will always choose to reduce a non-\( \epsilon \)-rule before reducing an \( \epsilon \)-rule. If \((m, n, 0)\) is unresolvable and there is a sentential form in which \( \xi_n = \epsilon \) should be reduced rather than \( \xi_m \) as chosen, then there will be an unresolvable condition 4 overlap.
(k, m, i) for which \( z_k, \lambda \xi_n \rightarrow_{i+1} \lambda^* y_m \). Similarly, if \((m, n, | \xi_n |)\) is an unresolvable triple, then there is an unresolvable condition 3 overlap \((k, n, i)\) where \( z_k, \lambda \xi_n \rightarrow_{i+1} \lambda^* y_m \). In such cases \( k, i \) are easily found by diagramming the rightmost sentential form in question or by diagramming the left unresolvability of the triple. In the latter case one quickly finds the motivation for allowing condition 3a overlaps, because it is possible that \( k = 0 \).

Similarly the special treatment of condition 3 overlaps in defining right resolvability, and the restriction of \( \beta \) from the domain \( V_N \times V_N \) are motivated by \( \epsilon \)-rules. Were these constraints not imposed, any useful \( \epsilon \)-rule would deny a grammar of the GOR property; under those conditions if \( y_n \Rightarrow \epsilon \), and some \( z_m, \lambda^* y_m \) for \( j > 1 \) then \((m, n, \xi_n \rightarrow_{i} \) would be an unresolvable condition 3 overlap.

With set definitions relating to overlaps set, the definitions of precedence grammars are simplified. We have stated the definition of ISP already to illustrate the \( \alpha, \lambda, \) and \( \rho \) relations.

A grammar \( G \) is weak precedence (WP) \([10]\) if \( G \) has no \( \epsilon \)-rules, the relations \( \alpha \lambda^* \) and \( \rho^+ \alpha \lambda^* \) are disjoint, \( G \) has no condition 1 overlaps, and every condition 2 overlap \((m, n, i)\) satisfies

\[ \neg z_m, \lambda \xi_n \rightarrow_{i+1} \rho^+ \alpha \lambda^* y_m . \]

A grammar \( G \) is simple mixed strategy precedence (SMSP) \([1]\) if \( G \) is WP, but with left resolvable condition 1 overlaps permitted. These definitions are the same as stated in the references cited although expressed in the terminology of overlaps. It is clear from these definitions that every ISP grammar is WP and every WP grammar is SMSP; these subset relations are proper \([1]\).

**Theorem 1.** The SMSP grammars form \( \alpha \) proper subset of the set of OR grammars.

**Proof.** From the definition of SMSP, all condition 1 overlaps are resolvable on the left. All condition 2 overlaps are resolvable of the left because of the WP definition. Without \( \epsilon \)-rules there can be no condition 3 overlaps and the disjointness of \( \alpha \lambda^* \) and \( \rho^+ \alpha \lambda^* \) insure all condition 4 overlaps are resolvable on the right. Hence every SMSP grammar is OR. The grammar described by the \( \rightarrow \) relation below is OR but not SMSP.

\[
G_1: \quad 1 \quad s \rightarrow wab \\
2 \quad w \rightarrow xb \\
3 \quad x \rightarrow a
\]

Note that \( \alpha \rho^+ \alpha \lambda^* b \) and \( \alpha \lambda^* b \) and that the only overlap \((1, 3, 1)\) is resolvable on the left. Of course, an \( \epsilon \)-rule in \( G_1 \) would have violated SMSP also.

In some case there are grammars for which the parser works quite well which are not members of the associated grammar class. (In the definition of RP grammars and
Theorem 2 below, we distinguish the GOR grammars from these for which the parser works.) We next consider the BRC, SLR(1), and LR(1) classes of grammars [6, 4, 11]. These classes are defined by the existence of an operable parsing algorithm. Their decision procedures amount to a successful parser construction, so that the associated parsers can only work for a grammar in that class. Consequently, the definitions of the grammar class is more complex and will not be repeated here. However, following the pattern of definition for LR(k) given in [9, Chap. 12] we define the RP grammars, which form a superset of the GOR grammars, for which the parsers to be proposed work. This class is intended to be descriptive, not useful. On the basis of computational efficiency, one would do better building an SLR(1) parser rather than testing for the RP property.

G is resolvable parseable (RP) if the following is true for all

\[ s \rightarrow \hat{\xi} \Psi_1 x \xi a \omega_1 \quad \text{where} \quad a \omega_1 \in V_T^*\{-\} : \]

If

\[ s \rightarrow \hat{\xi} \Psi_1 x y a \omega_1 \rightarrow \Psi_1 x \xi a \omega_1 , \quad \text{where} \quad y \rightarrow \xi \text{ in } G \]

and

\[ s \rightarrow \hat{\xi} \Phi z \xi \rightarrow \Psi_2 x \xi b \omega_2 \]

where \( \xi, b \omega_2 \in V_T^*\{-\} \) and \( z \in V_N \)
and \( [[\xi]] : (| \xi | - 1) b \omega_2 : | z | = | \xi | \) or \( [b \omega_2] : | z | = | \xi | \)
and \( w \alpha \lambda \gamma \) and \( y \beta \gamma \)

then \( \phi = \Psi_2 x \xi, y = z, \) and \( \xi = b \omega_2. \)

The definition of RP gives us sufficient condition to identify the phrase \( \xi \) within a rightmost sentential form which is the result of the single rule applied in deriving it from another sentential form. That phrase is often called the handle of the sentential form. The process of replacing that handle by the recognized left part \( y \) in the definition above, is called reduction. A parse can be described as a sequential identification of all reductions made in backwards-right-deriving a sentence to the start symbol.

**Theorem 2.** The set of RP grammars properly contains the set of GOR grammars.

**Sketch of Proof.** Containment is seen easily enough: If \( G \) is not RP, the description of the violating sentential forms points toward an unresolvable overlap. \( G_2 \) shows that the containment is proper.

\[ G_2 : \]

\[ 1 \quad s \rightarrow b r y \]
\[ 2 \quad y \rightarrow a g \]
\[ 3 \quad s \rightarrow c z g \]
\[ 4 \quad z \rightarrow r a. \]

In \( G_2 \) the overlap (2, 4, 1) is unresolvable, but \( G_2 \) is RP.
IV. GOR GENERATES THE DETERMINISTIC LANGUAGES

Because every GOR grammar is LR(1) we know that every language with a GOR grammar is a deterministic language by a theorem of Knuth [11]. It is shown below that every deterministic language has a GOR grammar. Aho, Denning, and Ullman [1] independently showed that every deterministic language has an SMSP grammar. Their result follows from one of Graham [18] and easily follows from the construction below.

Define a pushdown automaton (pda) [9] to be a seven-tuple \((K, \Sigma, \Gamma, \delta, q_0, z_0, F)\), where

1. \(K\), \(\Sigma\), and \(\Gamma\) are finite sets of states, input symbols and tape symbols, respectively;
2. \(F \subseteq K\) is the set of final states;
3. \(q_0 \in K\) is the initial state;
4. \(z_0 \in \Gamma\) is the initial contents of the pushdown;
5. \(\delta\) is a mapping from \(K \times (\Sigma \cup \{\epsilon\}) \times \Gamma\) to finite subsets of \(K \times \Gamma\).*

A pushdown automaton is said to be deterministic if for each \(q \in K\), \(a \in \Sigma \cup \{\epsilon\}\), and \(z \in \Gamma\), \(\delta(q, a, z)\) has at most one member, and if \(\delta(q, \epsilon, z)\) is not empty then \(\delta(q, a, z)\) is for all \(a \in \Sigma\).

\(T(M)\) and \(N(M)\) are the sets of sentences accepted by final state and by empty store, respectively, by the pda \(M\).
If $M$ is deterministic then $T(M)$ is a deterministic language.

**LEMMA 1.** For every dpda $M$, we can construct a pda,

$$M' = (K \cup \{e\}, \Sigma, \Gamma, \delta, q_0, z_0, \varepsilon),$$

such that $T(M) = N(M')$ and the following four conditions hold:

1. $\delta$ is a function which has only two possible forms:

   $$\delta(p, e, z) = (q, e) \quad \text{or} \quad \delta(p, a, z) = (q, yz)$$

   for $p, q \in K \cup \{e\}$, $y, z \in \Gamma$, $a \in \Sigma$.

2. $e$ is a distinguished final state which erases all stack characters and is never left once entered:

   $$\delta(e, e, z) = (e, e) \quad \text{for all} \quad z \in \Gamma;$$

   $$\delta(e, a, z) \quad \text{is undefined for all} \quad a \in \Sigma.$$

3. $z_0$ is a distinguished (initial) stack element which can never be erased but by a transition into state $e$.

   $$\delta(p, e, z_0) = (q, e) \implies q = e.$$

4. Nondeterminism is restricted by $\delta$ being a function and by the fact that

   $$\delta(p, e, z) = (q, e) \land q \neq e \quad \text{implies} \quad \delta(p, a, z) \quad \text{is undefined for all} \quad a \in \Sigma.$$

This means that the only nondeterminism involves a move into state $e$ as one of two "choices," the other being a move determined by an input character.

This lemma is proved in [17, Appendix I]. Define $G$ from $M'$ as follows:

$$G = (V_N, V_T, s, P);$$

$$V_N = \{(i; y, z; p, q) \mid p, q \in K \cup \{e\}, y, z \in \Gamma, 0 \leq i \leq 2\}.$$

Really $V_N$ is a set of quintuples $= \{0, 1, 2\} \times \Gamma^2 \times (K \cup \{e\})^2$. The integer $i$ selects either $y$ or $z$ (or neither); that character plus the two states amounts to the triple in the standard construction of a grammar from a pda [9].

$$V_T = \Sigma;$$

$$s = (1; z_0, t; q_0, e) \quad \text{for} \quad t \notin \Gamma, \quad a \text{ a meaningless character};$$

$P$ is derived from the two forms of $\delta$ as follows.
(A) Only if \( \delta(p, \epsilon, z) = (q, \epsilon) \) include in \( P \)
\[
(1; z, x; p, q) \rightarrow \epsilon \quad \forall x \in \Gamma \cup \{t\};
\]
\[
(2; x, z; p, q) \rightarrow \epsilon \quad \forall x \in \Gamma.
\]

(B) Only if \( \delta(p, a, z) = (q, yz) \) include in \( P \)
\[
(1; z, x; p, r_2) \rightarrow (0; y, z; p, q)(1; y, z; q, r_1)(2; y, z; r_1, r_2)
\quad \text{for all } r_1, r_2 \in K \cup \{e\},
\]
\[
\quad \text{for all } x \in \Gamma \cup \{t\};
\]
\[
(2; x, z; p, r_2) \rightarrow (0; y, z; p, q)(1; y, z; q, r_1)(2; y, z; r_1, r_2)
\quad \text{for all } r_1, r_2 \in K \cup \{e\},
\]
\[
\quad \text{for all } x \in \Gamma;
\]
\[
(0; y, z; p, q) \rightarrow a.
\]

The last rule generated is a 1-production used to carry context. These rule formats are really very simple: note that every right hand side of a 3-production has the pattern
\[
(0;,,)(1;,,)(2;,,);
\]
or
\[
(,,)(,,)(,,);
\]
or
\[
(,,)(,,)(,,);
\]
Also, every
\[
(0;,,) \rightarrow a \in \Sigma,
\]
so that, given \( a \in \Sigma, p \in K, z \in \Gamma \) we can find the left part for the one-production from \( \delta(p, a, z) \) uniquely.

**Theorem 3.** \( G', \) defined by reducing \( G \), is GOR.

**Proof.** The proof is given in detail in [17]. Note that \( G' \) has only condition 1 or condition 3 overlaps. The condition 1 overlaps will always be resolvable on the left. When a condition 3 overlap \( (m, n, i) \) is not left resolvable, the \( \epsilon \)-rule concerned, \( y_n \rightarrow \epsilon \), will arise from a definition of \( \delta \) involving state \( e \) as given in Lemma 1. Part 4 of lemma 1 indicates the only right context for this \( y_n \) is \( \rightarrow i \) while

\[
[\xi_m] : i \rightarrow \sigma \in V_{\tau^+}. \quad \blacksquare
\]

Define \( G'' \) from \( G' \) by performing the standard algorithm to remove \( \epsilon \)-rules from
context free grammars [9]: if \( \xi_n \) has \( i \) "disappearing" nonterminals in \( G' \), repeat the \( n \)th rule \( 2^i \) times with the disappearing nonterminals in the right part omitted in all possible combinations. Omit all \( \epsilon \)-rules from the results and reduce to get \( G'' \).

**Theorem 4.** \( G'' \) is OR.

*Proof.* In \( G'' \) we have condition 1 overlaps exactly as in \( G' \). Condition 3 overlaps are impossible but condition 4b overlaps arise between rules in \( G' \) and their images added in \( G'' \) without left parts of \( \epsilon \)-rules. If these new overlaps are not resolvable on the left, then the only allowable right context for the shorter rules is \(-\ldots\). Right resolvability follows. \( \square \)

**Theorem 5.** \( G'' \) is SMSP.

*Outline of proof.* In \( G \), \( \rho^+\alpha\lambda^* \) and \( \alpha\lambda^* \) are clearly disjoint: \( y = (2; ;) \) or \( y \in V_T \) are the only characters \( \rho \) related to anything, but such \( y \) are \( \alpha \) related to nothing. In \( G'' \) we could have \( yp^+x \) where \( y = (0; ;) \) if \( y\alpha x \in V \) we are caught by part 4 of Lemma 1 again. It can only be that \( z = s \) and \( yp^+\alpha\lambda^*-i \). Since there are no condition 2 overlaps and condition 1 overlaps are left resolvable, \( G'' \) is SMSP. \( \square \)

**Corollary.** The set of deterministic languages is the same as the GOR languages. But for the empty sentence, it is the same as the set of OR languages which is the same as the set of SMSP languages.

We recall in passing the known fact that ISP or WP grammars do not generate all those languages [19].

V. THE PARSING ALGORITHM

The example which we shall follow through this section is a representative of the classic algorithms for arithmetic expressions.

\[
G_3: \quad \begin{align*}
1 & \quad e \rightarrow e - t \\
2 & \quad t \rightarrow t \odot f \\
3 & \quad f \rightarrow p \uparrow f \\
4 & \quad p \rightarrow ( e ) \\
5 & \quad s \rightarrow e \\
6 & \quad e \rightarrow t \\
7 & \quad t \rightarrow f \\
8 & \quad f \rightarrow p \\
9 & \quad p \rightarrow a \\
10 & \quad e \Rightarrow e.
\end{align*}
\]
We can assign each character (and endmarker) an index which will be used to access matrices.

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\rightarrow & a & \uparrow & \otimes & \rightarrow & \left\{ \begin{array}{c} p \ f \ t \ e \ s \\
V_T & V_N
\end{array} \right.
\end{array}
\]

The peculiar orders will be justified later. The start symbol is \( s \).

Based on an \( SLR(1) \) grammar by DeRemer [3], \( G_a \) is neither \( LR(0) \) nor SMSP. If the \( \epsilon \)-rule is given the semantic meaning of arithmetic zero, the grammar is provided with a false unary ‘‘−’’ operator. (False because of exponentiation: \( -1^2 \) would be be evaluated to \( -1 \).)

\[
\begin{array}{cccccccc}
\text{a}^{\lambda^*} & p & f & t & e & s \\
\hline
\uparrow & 1 & 1 & 1 & 1 & 1 \\
\downarrow & 0 & 0 & 0 & 0 & 0 \\
a & 0 & 0 & 0 & 0 & 0 \\
ds & 1 & 1 & 0 & 0 & 0 \\
- & 1 & 1 & 1 & 0 & 0 \\
t & 1 & 1 & 1 & 1 & 0 \\
p & 0 & 0 & 0 & 0 & 0 \\
f & 0 & 0 & 0 & 0 & 0 \\
t & 0 & 0 & 0 & 0 & 0 \\
e & 0 & 0 & 0 & 0 & 0 \\
s & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{array}
\]

\[
\begin{array}{cccccc}
\text{t}_{\alpha^*}^{\lambda} & p & f & t & e & s \\
\hline
\uparrow & 1 & 1 & 1 & 1 & 1 \\
\downarrow & 1 & 1 & 1 & 1 & 1 \\
\otimes & 1 & 1 & 1 & 1 & 1 \\
c & 1 & 1 & 1 & 1 & 1 \\
s & 1 & 1 & 1 & 1 & 1 \\
\hline
\end{array}
\]

\[
\begin{array}{cccccc}
\text{t}_{\beta^*}^{\lambda} & p & f & t & e & s \\
\hline
\uparrow & 1 & 0 & 1 & 1 & 0 \\
\downarrow & 1 & 0 & 0 & 1 & 1 \\
\otimes & 1 & 0 & 0 & 1 & 1 \\
e & 1 & 0 & 0 & 0 & 1 \\
s & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}
\]

\[
\begin{array}{cccccc}
\text{t}_{\gamma^*}^{\lambda} & p & f & t & e & s \\
\hline
\uparrow & 1 & 0 & 1 & 1 & 0 \\
\downarrow & 1 & 0 & 0 & 1 & 1 \\
\otimes & 1 & 0 & 0 & 1 & 1 \\
e & 1 & 0 & 0 & 0 & 1 \\
s & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{array}
\]

**Fig. 5.** Relations used to establish grammar \( G_a \) as being GOR.
If we test to see if $G_3$ is GOR, we find the relations displayed in Fig. 5 and the resolvable overlaps listed below:

<table>
<thead>
<tr>
<th>Overlap</th>
<th>Diagram</th>
<th>Condition</th>
<th>Resolvable on</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2,1,2)</td>
<td>$t \oplus f$</td>
<td>4c</td>
<td>right</td>
</tr>
<tr>
<td>(1,5,2)</td>
<td>$e - t$</td>
<td>4b</td>
<td>right</td>
</tr>
<tr>
<td>(1,6,0)</td>
<td>$e - t$</td>
<td>2</td>
<td>left</td>
</tr>
<tr>
<td>(1,10,1)</td>
<td>$e - t$</td>
<td>3b</td>
<td>left &amp; right</td>
</tr>
<tr>
<td>(1,10,2)</td>
<td>$e - t$</td>
<td>3b</td>
<td>left</td>
</tr>
<tr>
<td>(2,6,2)</td>
<td>$t \oplus f$</td>
<td>4b</td>
<td>right</td>
</tr>
<tr>
<td>(2,7,0)</td>
<td>$t \oplus f$</td>
<td>2</td>
<td>left</td>
</tr>
<tr>
<td>(2,10,1)</td>
<td>$t \oplus f$</td>
<td>3b</td>
<td>left &amp; right</td>
</tr>
<tr>
<td>(2,10,2)</td>
<td>$t \oplus f$</td>
<td>3b</td>
<td>left &amp; right</td>
</tr>
<tr>
<td>(3,7,0)</td>
<td>$p \uparrow f$</td>
<td>2</td>
<td>left</td>
</tr>
<tr>
<td>(3,8,2)</td>
<td>$p \uparrow f$</td>
<td>4b</td>
<td>right</td>
</tr>
<tr>
<td>(3,10,1)</td>
<td>$p \uparrow f$</td>
<td>3b</td>
<td>left &amp; right</td>
</tr>
<tr>
<td>(3,10,2)</td>
<td>$p \uparrow f$</td>
<td>3b</td>
<td>left &amp; right</td>
</tr>
<tr>
<td>(4,5,1)</td>
<td>$(e) \uparrow e$</td>
<td>4a</td>
<td>left &amp; right</td>
</tr>
<tr>
<td>(4,10,1)</td>
<td>$(e) \uparrow e$</td>
<td>3b</td>
<td>left</td>
</tr>
<tr>
<td>(4,10,2)</td>
<td>$(e) \uparrow e$</td>
<td>3b</td>
<td>right</td>
</tr>
<tr>
<td>(0,10,2)</td>
<td>$\rightarrow s \rightarrow e$</td>
<td>3a</td>
<td>right</td>
</tr>
</tbody>
</table>

So $G_3$ is GOR. We now consider the parsing algorithm itself.
Domolki [5] has proposed an algorithm which uses the logical operations of “and”, “or”, “shift” and “jump on zero” on a binary computer to keep track of a phrase match with a bit matrix. Each nonzero bit, by its position, indicates that a match has been successful up to that point. These $S$ matrices are placed on a push-down store. The power of his algorithm comes from the fact that logical operations, operating on entire computer words, can compute new $S$ matrices in parallel because these matrices are packed into about $h \cdot |P|/b$ words, where $b$ denotes the bits per word on the computer in question.

Domolki’s bit matrices are one-dimensional. Lynch [14] restructures the $S$ matrix into a $h \times |P|$ rectangular matrix, where the $|P|$ dimension indexes across bits in a computer word. A phrase match is indicated by a 1 bit marching down the column associated with the rule in question. Although more space is needed for each $S$ than in Domolki’s formulation, the exact rule is more easily identified, when any bit appears in the bottom row. A floating point “normalize” operation easily locates the 1 bit within a row (computer word) yielding the index of the rule recognized.

It is the algorithm which is restructured below to run twice as fast and to handle $e$-rules. Note that each phrase is padded to length $h$ with $e \notin V$ on the left; Lynch pads on the right. Let

$$\psi_n = e^{h-|\xi_n|}, \quad \xi_n = e \cdots e z_{n,1} \cdots z_{n,|\xi_n|} = a_{n,1} \cdots a_{n,h},$$

where $e \notin V$.

$$a_{n,j} = [j : [\psi_n]] : 1 \quad \text{is defined} \quad 1 \leq j \leq h.$$

Note that

$$a_{n,j+h-|\xi_n|} = z_{n,j} \quad \text{for} \quad 1 \leq j \leq |\xi_n|.$$

Three Boolean matrices are defined with the index $1 \leq n \leq |P|$ traversing the bits within a few computer words, which together make one row. This allows maximal parallelism using the “and”, “or”, “zero jump”, and “normalize” commands usually available.

$$L_{j,n} = 1 \text{ iff } \quad j = h - |\xi_n| \quad \text{for} \quad 0 \leq j \leq h, \quad 1 \leq n \leq |P|.$$

$$R_{x,n} = 1 \text{ iff } \quad y_n \beta x \quad \text{for} \quad x \in V_T \cup \{-\}, \quad 1 \leq n \leq |P|.$$

$$M_{x,j,n} = 1 \text{ iff } \begin{cases} j = h - |\xi_n| \quad \text{and} \quad x \alpha \lambda^* y_n; \\ j > h - |\xi_n| \quad \text{and} \quad x = b_{n,j} \end{cases} \quad \text{for} \quad x \in V \cup \{-\}, \quad 0 \leq j \leq h, \quad 1 \leq n \leq |P|.$$

Figure 6 illustrates $L$ and $R$ for five of the rules of $G_3$.

$L$ initiates the phrase match on the left, starting the match of the $n$-th phrase in
the appropriate row of $S$ (defined below). $M$ matches characters, including left context. When a 1 bit reaches the bottom row of $S$, the $R$ matrix provides right compatibility checks. Note that $e \notin V$ is used only in the definitions here. Algorithm 1 uses a push-

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
& 1 & 2 & 3 & 4 & 5 & 10 \\
\hline
x & & & & & & \\
\hline
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
2 & 1 & 1 & 1 & 1 & 0 & 1 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 1 & 0 & 0 \\
5 & 0 & 1 & 1 & 1 & 0 & 0 \\
6 & 1 & 1 & 1 & 1 & 0 & 1 \\
7 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
& 1 & 2 & 3 & 4 & 5 & 10 \\
\hline
j & 12345 & 10 \\
\hline
0 & 11110 & 0 \\
1 & 00000 & 0 \\
2 & 00001 & 0 \\
3 & 00000 & 1 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
& x = l & x = ) & x = a & x = \d & x = * \\
\hline
\hline
j & 12345 & 10 & 12345 & 10 & 12345 & 10 & 12345 & 10 \\
\hline
0 & 11110 & 0 & 00000 & 0 & 00000 & 0 & 01110 & 0 & 00110 & 0 \\
1 & 00000 & 0 & 00000 & 0 & 11100 & 0 & 00000 & 0 & 00000 & 0 \\
2 & 00000 & 0 & 00000 & 0 & 00100 & 0 & 00100 & 0 & 01000 & 0 \\
3 & 00000 & 1 & 00100 & 0 & 11101 & 0 & 00000 & 0 & 00000 & 0 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
& x = . & x = \theta & x = \mu & x = f \\
\hline
\hline
j & 12345 & 10 & 12345 & 10 & 12345 & 10 & 12345 & 10 \\
\hline
0 & 01110 & 0 & 11110 & 0 & 00000 & 0 & 00000 & 0 \\
1 & 00000 & 0 & 00010 & 0 & 11100 & 0 & 11000 & 0 \\
2 & 10000 & 0 & 00000 & 0 & 00100 & 0 & 00100 & 0 \\
3 & 00000 & 0 & 00000 & 1 & 11101 & 0 & 11101 & 0 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
& x = t & x = e \\
\hline
\hline
j & 12345 & 10 & 12345 & 10 \\
\hline
0 & 00000 & 0 & 00000 & 0 \\
1 & 10000 & 0 & 10000 & 0 \\
2 & 00010 & 0 & 00010 & 0 \\
3 & 10001 & 0 & 00001 & 0 \\
\end{array}
\]

Fig. 6. Bit matrices for Algorithm 1 to parse six productions of $G_3$. 
down store to hold $h \times |P|$ sized $S$ matrices where an entry is not made into the column associated with $\xi_n$ until only $|\xi_n|$ rows are left beneath and left compatibility has been established. Thereafter, the 1 bit proceeds down the column as $\xi_n$ is matched.

**Algorithm 1**

1. Initialize $i := 0$;

   $$S_{0,j,n} := L_{j,n} \land M_{p-j,n} \quad \text{for} \quad 0 \leq j < h \quad \text{and} \quad 1 \leq n \leq |P|.$$  

Let $w$ be the first character on the input stream which we peek at but do not remove. Set

$$T_n := L_{h,n} \land M_{p-h,n} \land R_{w,n} \quad \text{for} \quad 1 \leq n \leq |P|.$$  

If any $T_n = 1$ go to 8. Note that the plane of $M$ indexed by $\leftarrow$ is not used after this initialization.

2. Set $i := i + 1$ stacking $S_{i-1}$ (and $x$ if desired). Read $x$ from the input stream. If $x = \leftarrow$ then quit indicating an unsuccessful parse. Let $w$ refer to the following character of input which we peek at but do not remove. Go to 4.

3. If $x = s$ quit with $i = 1$.

4. Compute for $1 \leq n \leq |P|$

   $$T_n := S_{i-1,n} \land M_{x,n} \land R_{w,n}.$$  

If, for all $n$, $T_n = 0$ go to 6.

5. Let $n$ be such that $T_n = 1$. Indicate a recognition of $(y_n, \xi_n)$ where $\xi_n \neq \epsilon$. Set $i := i + 1 - |\xi_n|$ popping the stack; $x := y_n$. Set $k := 0$. Go to 3.

6. Compute for $1 \leq j \leq h - 1$ and $1 \leq n \leq |P|$

   $$S_{i,0,n} := L_{0,n} \land M_{x,0,n};$$

   $$S_{i,j,n} := (L_{j,n} \lor S_{i-1,j-1,n}) \land M_{x,j,n}.$$  

7. $T_n := L_{h,n} \land M_{x,h,n} \land R_{w,n}$. If all $T_n = 0$ go to 2.

8. Let $n$ be such that $T_n = 1$. Indicate a recognition of $(y_n, \epsilon)$ between $x$ and $w$. Set $i := i + 1$ stacking $S$; $x := y_n$. Set $k := k + 1$. If $k > |V_N|$ then quit with an error. Go to 3 otherwise.

Figure 7, discussed in more detail below, illustrates the behavior of this algorithm with the tables of Fig. 6.

**Theorem 6.** If $G$ is RP then Algorithm 1 can parse any $\sigma \in L(G)$.
Fig. 7. Diagram of the parse of the sentence $a \rightarrow \gamma (\alpha \odot \beta \cdot \phi)$ using the matrices in Fig. 6. Numbers denote the production recognized. Solid lines are stack pops; dotted lines indicate an overlay of an S matrix not stacked because character is in $P_S$. 

TIME

STACK

$S_0 = (11100) \begin{pmatrix} 00000 \end{pmatrix}$

$S_1 = (10110) \begin{pmatrix} 00000 \\ 00000 \\ 00000 \end{pmatrix}$

$S_2 = (10110) \begin{pmatrix} 00000 \\ 00000 \\ 00000 \\ 00000 \end{pmatrix}$

$S_3 = (10110) \begin{pmatrix} 00000 \\ 00000 \\ 00000 \\ 00000 \\ 00000 \end{pmatrix}$

$S_4 = (10110) \begin{pmatrix} 00000 \\ 00000 \\ 00000 \\ 00000 \\ 00000 \\ 00000 \end{pmatrix}$
Outline of Proof. That \( T \neq 0 \) indicates a handle is a result of constructing a predicate for the meaning of \( S \). We still must show that we eventually find every handle. The only way that some handle would never be completely read (and therefore recognized, by the interpretation of \( S \)) is that the parser loops recognizing \( \epsilon \)-rules, so that the input beyond some point is never reached. This situation cannot happen in any sentence; the proof involves a reduction of cases yielding unresolvable overlaps on an \( \epsilon \)-rule if such \( \epsilon \)-rule is recognized infinitely often in some sentential form.

If \( \sigma \notin L(G) \), however, an infinite loop can occur in Algorithm 1 without the counter \( k \) limiting consecutive reductions of \( \epsilon \)-rules. Consider \( G_4 \) defined by the relations:

\[
G_4: \begin{align*}
1 & \quad s \rightarrow yc \\
2 & \quad s \rightarrow ayz \\
3 & \quad s \rightarrow bzy \\
4 & \quad y \Rightarrow \epsilon \\
5 & \quad z \Rightarrow \epsilon
\end{align*}
\]

\( G_4 \) is GOR but Algorithm 1 would go into an infinite loop "recognizing" \( \epsilon \)-rules if \( \epsilon \notin L(G_4) \) were given as input and the count, \( k \), were omitted. That control on successive \( \epsilon \)-recognitions is necessary when parsing grammars with the following relations:

\[
x_2\alpha\lambda^*x_\delta\lambda^* \cdots \alpha\lambda^*x_m\alpha\lambda^*x_1; \quad \text{and} \\
\exists w \in V_T(x_n\beta w) \quad \text{for} \quad 1 \leq n \leq m, \quad \text{and} \\
x_n = \epsilon \quad \text{for} \quad 1 \leq n \leq m.
\]

Algorithm 1 can parse program-like input in better than one-half the time of Lynch's ICOR parser [14]. By program-like we mean that the grammar is constructed so that the average phrase to be recognized is of length 4 or less. This is a very loose criterion: we are not speaking of the mean phrase length in practical grammars, but whether the long phrases are popular with those who create the input. The padding of \( \epsilon \) on the left of the phrases in the definitions associated with Algorithm 4 (rather than on the right as is Lynch's) results in phrase matches for all productions ending in what would be the \( S_{\ell, h, * \epsilon} \) row, so that there is one less operation (on another bit in \( S_\ell \)) in the computation elements within an \( S_\ell \) matrix. This also makes the expression for \( T \) independent of \( S_\ell \); only \( S_{\ell-1} \) is needed. Most of the time is spent in computing the \( S \) matrices and this is where the saving is:

1. Since \( T \) can be computed before \( S_\ell \), we save the computation associated with one \( S \) matrix (the last) for every non-\( \epsilon \) phrase parsed. For a program-like input we have a 3/4 factor of improvement or better.

2. In each computation of an element of \( S \) we save one logical operation (and
indexing on one operand). Depending on the structure of the hardware we can improve by another factor of $2/3$.

Hence we can expect a 50\% or more time improvement, depending on the input and the machine.

Notice that the power of Algorithm 1 lies in the fact that computation occurs in parallel with respect to $n$, except at steps 5 and 8 which are only executed when a recognition is certain to occur. A sacrifice of this parallelism can yield an $SLR(1)$ parser [17, Algorithm 5].

With an understanding of the basic algorithm, there are many modifications which are possible. Some are trivial and should be considered essential to efficient any implementation. Others require some restructuring of the grammar input, or result in a change in parser behavior. These could be valuable or useless depending on the application. Seven alterations are listed below.

(1) It is important to notice that adding (or deleting) a rule requires (or saves) one more bit in every row of every bit matrix. For an arbitrary grammar $G = (V_N, V_T, S, P)$, unless $|P|$ is an exact multiple of the word size, total space requirements for the bit matrices will not change. In the rare case that $|P|$ is a multiple of word size, however, adding another rule requires an entire word to be added to every row, of which all but one of the bits are not used.

Decreasing the maximal phrase length (defined before as $h$), however, is an action which saves many rows in $M, L$, and every $S$ matrix. If there are only a few long phrases, it may be very easy to split them by adding new productions (increasing $|P|$) whose left parts are new nonterminals in the vocabulary. Of course new overlaps may be introduced, so we must be sure they are resolvable. Therefore, an interactive facility would ease testing hand-made splits.

(2) Let us order the set of productions; so far $P$ has been treated as an unordered set.

**Definition.** $(y_m, \xi_m) \prec (y_n, \xi_n)$ iff

\[
|\xi_m| > |\xi_n| \quad \text{or} \\
|\xi_m| = |\xi_n| \quad \text{and} \quad y_m \xrightarrow{+} y_n.
\]

**Theorem 7.** If $G$ is reduced and unambiguous, $\prec$ is a strict partial ordering on $P$.

**Proof.** It is partial because it is possible $|\xi_m| = |\xi_n|$ but not $y_m \xrightarrow{+} y_n$. It is strict because if $G$ is reduced and unambiguous then $-$ $(y_n \xrightarrow{+} y_m \xrightarrow{+} y_n)$. $\blacksquare$

Let us index $P$ so that $(y_m, \xi_m) \prec (y_n, \xi_n)$ implies $m < n$. $G_0$ is presented in this way. In each column of $S$, the bits above the bits indexed by $j < h - |\xi_n|$ are never
used. Therefore, these wasted bits are gathered to the upper right corner of $S$ by this ordering.

In particular, all $\epsilon$-rules, if any, are pushed to the right, indexed by the highest indices. Let $q$ be such that $\xi_n \neq \epsilon$ for $1 \leq n \leq q$ and $\xi_n = \epsilon$ for $q < n \leq |P|$. Then the $S$ matrices need only be defined for $1 \leq n \leq q$ rather than $1 \leq n \leq |P|$. Since $\epsilon$-rules are only referenced in steps 7 and 8 of Algorithm 1, we can restrict the limits on $n$ to $1 \leq q \leq n$ at the steps where non-$\epsilon$-rules are considered and restrict $n$ to $q < n \leq |P|$ at steps 7 and 8. For $G_3$, $q = 9$.

Indexing bits in a row by $1 < n \leq |P|$ it is appropriate that a word boundary not be crossed unnecessarily. If the space allocation leaves unused bits in every row, these can be inserted between the bits used for the $q$-th and $(q + 1)$-th production to push all the $\epsilon$-rule bits together in the minimal number of words. Then at step 8 when a bit appears in the $T$ vector, the number of the $\epsilon$-rule parsed is derived from the bit location by a constant displacement according to the number of unused bits inserted.

(3) We must also index the vocabulary. In a practical situation the character of $V \cap \{s, \rightarrow, \textarrow, \textarrow, \textarrow\}$ will be represented as integers to index the driver matrices. The order in which this numbering applies is the effective indexing of $V$. Define $V_S$ ($S$ for stacking symbols) as

$$V_S = \{x \in V : \exists y \in V_N(x \alpha y)\}.$$ 

If $x \notin V_S$, then any $S$ matrix computed on $x$ can never again rise to the top of the stack. It will only be popped off as part of a series (associated with a longer phrase) if it is to be removed at all. Clearly, $x \notin V_S$ implies $M_{x,j,n} = 0$ when $j = h - |\xi_n|$, from the definition of $M$. Index $V \cup \{s, \rightarrow, \textarrow, \textarrow\}$ in the following order (where $V_S$ denotes the complement of $V_S$):

<table>
<thead>
<tr>
<th>Set</th>
<th>Upper index</th>
<th>For $G_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${\textarrow, \rightarrow}$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V_S \cap V_T$</td>
<td>$l_1$</td>
<td>2</td>
</tr>
<tr>
<td>$V_S \cap V_T$</td>
<td>$l_2 =</td>
<td>V_T</td>
</tr>
<tr>
<td>$V_S \cap V_N$</td>
<td>$l_3$</td>
<td>6</td>
</tr>
<tr>
<td>$V_S \cap V_N$</td>
<td>$l_4 =</td>
<td>V</td>
</tr>
<tr>
<td>${s}$</td>
<td>$l_5 =</td>
<td>V</td>
</tr>
</tbody>
</table>

The endmakers are treated as one symbol.

With this ordering $x$ can index $R$ within the limits $0 \leq x \leq |V_T|$; $x$ indexes $M$ in the limits $1 \leq x \leq |V|$. The only required exception is $M_{x,j,n}$ which already plays an exceptional role in Algorithm 1 in that it is needed only at initialization. Step 3 tests if $x > |V|$.
As any $S$ is computed with the formula $S := (L \lor S) \land M$ at step 6 in the algorithm, we need only check if $x$, being processed, is in $V_S$. If it is not the new $S$ matrix is not stacked. A corresponding modification of the algorithm occurs whenever a phrase is recognized and reduced. Instead of popping the stack $|\xi_n| - 1$ characters when $\xi_n$ is recognized at step 5, the amount to be popped is appropriately based on the number of characters of $V_S$ which occur in $\xi_n$. Since these lengths can be adjusted before run time, the effective modification to Algorithm 1 is a test for $l_1 < x \leq l_2$ and appropriate computation of $S$ and $T$ depending on the result.

When this modification is applied, we have the equivalent of a technique of DeRemer [4]. If we think of the $S$ matrices as the states of a dpda, and the algorithm as a way of computing state transitions, then the states (S matrices) entered via $x \in V_S$ are DeRemer's states which do not stack their own names.

The space savings in a condensed stack allowed by such a simple test can be considerable. In parsing $G_3$ most $S$ matrices need not be stacked, as indicated in the example of Fig. 7.

(4) In [17, Appendix II] 1-productions are shown to be very useful for carrying context from more than one character to the left of a handle. Other 1-productions can be a nuisance.

With $G_3$, which is representative of the classic grammars for arithmetic expressions, we have derivation such as $e \rightarrow t \rightarrow f \rightarrow p \rightarrow a$ which give no more information for a compiler than the fact $e \rightarrow a$. In the original statement of Domolki's algorithm [5], these productions were handled implicitly by additions to $M$. We shall define $M$ for Algorithm 1 (with $j > h - |\xi_n|$) in terms of $M$, already known:

$$M_{x,i,n} = \bigvee_{y \in V_n \cup \{\varepsilon\}} (M_{y,i,n} \land y \not\rightarrow x).$$

If $M$ is used in place of $M$, the effect is to allow $x$ to match a required $z$ in a phrase when $z \not\rightarrow x$. Domolki then removed all one-productions: this was allowable since his parser could backtrack to recover if such a presumption proved to be improper. Because our parser is deterministic, we are not completely free to ignore 1-productions. Figure 6 illustrates $M$ for some of the productions of $G_3$.

If no 1-productions are removed, the algorithm operates correctly with $M$ replacing $M$. However, it is possible that $n$, such that $T_n = 1$, is not always unique. But each $n(\xi_n \neq \varepsilon)$ identifies a proper recognition. The appropriate one to choose is the one for which $|\xi_n| > 1$ or $y_n \not\rightarrow y_m$ for all $T_m = 1$. If any other reduction were chosen the parser would loop, parsing 1-productions until the described parse was reached. The choice of $n$ is easy if $P$ is ordered by $\gamma$ as in point 2 above. The method of choosing $n$ is then consistent with our earlier rule: find the smallest $n$ such that $T_n = 1$ using, for example, a floating point normalize instruction to find the first $T$ bit set. Note that if we use $M$ not every application of a 1-production is recognized.

The next step is to remove as many 1-productions as possible thus decreasing $P$. 
As noted in point 1 above, lots of productions might have to be removed before we save any space. Two problems constrain the choice of 1-productions which are removable. First, some 1-productions could have semantic meaning to the compiler (for instance an implicit change from type integer to type floating point). Secondly some 1-productions may be required to preserve the GOR property. Lynch notes this problem [14]; in his scheme certain 1-productions are flagged to be saved when the grammar is built. Below is a theorem to find that class of 1-productions he flags. Only the class with semantic meaning must be noted by the person creating the source grammar as essential, once the theorem is implemented.

**THEOREM 8.** Suppose $G$ is a grammar, $P$ is ordered by $<$, $y_j \neq s$, $|\xi_j| = 1$ and all overlaps, or triples $(m, n, i)$ s.t. $m \neq j \neq n$ which would be overlaps if $\xi$ were replaced by $y_j$ throughout $G$, are resolvable even when

$$\forall z \in V(y_jax \supset \xi_jax) \text{ holds.}$$

Then 1-production $(y_j, \xi_j)$ can be removed from indexing the matrices for Algorithm 1, and $G$ can still be parsed. The parsing is done by Algorithm 1 using $\bar{M}$ defined from $G$ and $L$ modified to include any new $\alpha$ relations presumed by the predicate $(y_jax \supset \xi_jax)$ above.

**Proof.** Construct a set of productions $P'$ from $G$ by replacing every occurrence of $y_j$ by $\xi_j$ and removing the 1-production $y_j \rightarrow \xi_j$. Because $\xi_j \in V_T$ possibly, and $y_j$ could be the left part of another production, $P'$ may not describe a formal grammar. Ignoring that momentarily, apply the tests for overlap resolvability to $P'$. If the definition would apply to $P'$ then Algorithm 1 could parse any sentence $\sigma \in L(G)$ according to $P'$.

Assuming that we have run the decision procedure for the GOR property on $G$, what additional computations must be made to decide if $P'$ is GOR? To answer this we observe the implicit validity of the following in $G$.

$$\forall z \in V \ (zax*yz \supset zax*\xi_j);$$

$$\forall z \in V \ (y_j\beta z \supset \xi_jz);$$

$$\forall z \neq y_j \ (z\beta*\lambda^*T y_j \text{ in } G \text{ iff } z\beta\lambda^*T z_j \text{ in } P');$$

similarly $\beta\lambda^*T$.

If we add the constraint to $G$ that

$$\forall z \in V \ (y_jax \supset \xi_jax),$$

then it is also true that

$$\forall z \in V \ (y_j\alpha*\lambda^*z \supset \xi_j\alpha*\lambda^*z).$$

Then to test $P'$ for the GOR property we need only test all overlaps introduced by the
"matching" of \( \xi_j \) and \( y_j \). These overlaps are triples in \( G \) which do not match only because \( \xi_j \neq y_j \). The relations to test resolvability can be exactly those which hold in \( G \) because of the equivalence established by predicates above, with the additional constraint noted.

Note that Theorem 8 applies to productions 6, 7, 8, 9 of \( G_3 \) in succession. Production 5 must remain because its left-part is the sentence prototype which stops the parse. That is why those four productions are omitted from Figs. 6 and 7.

The resolvable triples, newly treated as overlaps, are listed:

<table>
<thead>
<tr>
<th>Triple</th>
<th>Diagram</th>
<th>Would meet condition</th>
<th>Resolvable on</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,1,2)</td>
<td>( e \rightarrow t ) ( e \rightarrow t ) ( e \rightarrow t )</td>
<td>4c left</td>
<td></td>
</tr>
<tr>
<td>(1,2,2)</td>
<td>( e \rightarrow t ) ( t \rightarrow f ) ( e \rightarrow t )</td>
<td>4c left</td>
<td></td>
</tr>
<tr>
<td>(1,3,2)</td>
<td>( p \uparrow f ) ( e \rightarrow t ) ( e \rightarrow t )</td>
<td>4c left</td>
<td></td>
</tr>
<tr>
<td>(3,1,2)</td>
<td>( p \uparrow f ) ( e \rightarrow t ) ( e \rightarrow t )</td>
<td>4c right</td>
<td></td>
</tr>
<tr>
<td>(1,5,0)</td>
<td>( e \rightarrow t ) ( e \rightarrow t ) ( e \rightarrow t ) ( e \rightarrow t )</td>
<td>2 left</td>
<td></td>
</tr>
<tr>
<td>(2,2,2)</td>
<td>( t \rightarrow f ) ( t \rightarrow f ) ( t \rightarrow f )</td>
<td>4c left</td>
<td></td>
</tr>
<tr>
<td>(2,3,2)</td>
<td>( p \uparrow f ) ( t \rightarrow f ) ( t \rightarrow f )</td>
<td>4c left</td>
<td></td>
</tr>
<tr>
<td>(3,2,2)</td>
<td>( p \uparrow f ) ( t \rightarrow f ) ( t \rightarrow f )</td>
<td>4c right</td>
<td></td>
</tr>
<tr>
<td>(2,5,2)</td>
<td>( t \rightarrow f ) ( t \rightarrow f ) ( t \rightarrow f ) ( t \rightarrow f )</td>
<td>4b right</td>
<td></td>
</tr>
<tr>
<td>(2,5,0)</td>
<td>( t \rightarrow f ) ( t \rightarrow f ) ( t \rightarrow f ) ( t \rightarrow f ) ( t \rightarrow f ) ( t \rightarrow f )</td>
<td>2 left</td>
<td></td>
</tr>
<tr>
<td>(3,3,2)</td>
<td>( p \uparrow f ) ( p \uparrow f ) ( p \uparrow f )</td>
<td>4c right</td>
<td></td>
</tr>
<tr>
<td>(3,5,2)</td>
<td>( p \uparrow f ) ( p \uparrow f ) ( p \uparrow f )</td>
<td>4b right</td>
<td></td>
</tr>
<tr>
<td>(3,5,0)</td>
<td>( p \uparrow f ) ( p \uparrow f ) ( p \uparrow f )</td>
<td>2 left</td>
<td></td>
</tr>
</tbody>
</table>

Of course, if the above technique is used \( (y_j, \xi_j) \) will never be recognized explicitly, and hence such recognition cannot be used to trigger semantic actions.
(5) Algorithm 1 has been designed to check independent context one character
to the left and right. Lynch stated his algorithm in a form which allowed inspection
further to either side of the proposed phrase: ICOR(m, n) checks context m characters
to the left and n to the right [14]. This idea is easily worked into Algorithm 1 by using
\((\alpha \lambda^*)^k, \beta^k\) for \(k \leq m\) and \(k \leq n\), respectively, to expand the matrices. Hext and Roberts
[8] have independently applied this technique for extending context to the right to
refine Domolki’s algorithm.

(6) An input not in \(L(G)\) is detected at step 2 of Algorithm 1. The error might be
detected earlier if every \(S\) matrix is tested, as it is computed, to make sure there is a
‘one’ bit somewhere. A disjunction of all rows is easily tested; this can also be done
in parallel with respect to \(n\). Note that the error may not be detected until well after
the point an \(LR(0)\) parser would detect it.

(7) Finally we make a simple observation on Algorithm 1 for OR grammars.
Steps 7 and 8, which only provide for \(\epsilon\)-rules, can be replaced by “Go to 2.” Rows of
\(L\) for \(j = h\) can be eliminated.

Figure 7 gives an example of the behavior of Algorithm 1 with these modifications.
The input sentence being parsed is

\[-a \uparrow ( ) \uparrow (a \circ a) \in L(G_4).\]

Of course we presume \(-\circ\) is suffixed to it. The figure does not show right compatibility
checks. Values of \(x\) are indicated by the character from the vocabulary; when \(x\) is a
terminal, the input stream has been advanced. Because we have used \(M\) and Theorem
9, rules 5–9 are not recognized. Since rule 10 is an \(\epsilon\)-rule, these \(S\) matrices, indexed
across by \(P\), need only provide for the first five rules of \(G_3\).

This example serves well to demonstrate that an abbreviated parse, after pruning
1-productions with Theorem 8, does not fully describe the sentence. The sentence
\(-a \uparrow (( ) \circ a) \uparrow a\) generates the same recognitions in the same order as those in
Fig. 7. Obviously, in a practical application there will be more information put on the
stack, and more output when a rule is recognized, than in this example.

Algorithm 2.

We shall now describe a step by step parser builder, in general terms, which
implements the concepts developed here. Some steps are optional, but all recommended
for an efficient result.

(1) Read in a grammar \(G\) which has all semantically essential 1-productions
flagged.

(2) Reduce \(G\) and in the process find all \(y \Rightarrow \epsilon\) and the relation \(x \Rightarrow y\) defined
over \(V_N \times V\) for use in steps 3 and 9.
(3) Order $P$ by $<$ as in point 2 of the previous discussion.

(4) Construct the $\alpha$ relation and then compute $V_S$ and order $V$ as in point 3 above. Find adjusted length of every phrase according to $V_S$.

(5) Construct the matrix $\beta$ and save it for step 9. Construct $\alpha\lambda^*$, $(\alpha\lambda^*)\alpha\lambda^*$, $\beta\lambda^*\tau$, $\beta\lambda^*\tau$, and $\beta\beta^\tau$. If $G$ has no $\epsilon$-rules, the computations can be greatly abbreviated. For each of the two "left" matrices, $\alpha\lambda^*$ and $(\alpha\lambda^*)\alpha\lambda^*$, set up a 'ghost' matrix with exactly the same dimensions, but initially all zero.

(6) Starting with $i = 1$ thru $|P|$, scan $\xi_i$ against all $\xi_j$ s.t. $j > i$ and locate all overlaps. As each overlap is found, check appropriate matrix for right resolvability. If this fails, check left resolvability, and place a '1' in the 'ghost' matrix of the appropriate left matrix where a '0' was found yielding the overlap resolvable on the left. These 'ghost' entries act as reservations to prevent us from changing any of those entries to one later. When we apply Theorem 8 at the next step we attempt to decrease $P$, at the risk of increasing the number of valid $\alpha\lambda^*$ relations. The 'ghost' matrices will allow us to check quickly if any former resolvability is to be destroyed. Because right resolvability is not threatened by Theorem 8 we need no such 'ghost' matrices. Moreover, we check right resolvability first, even though it is a more difficult test, so that the left test 'ghost' matrices remain as sparse as possible.

If any overlaps are found to be unresolvable, they are listed and the algorithm stops after this step.

(7) If all overlaps are resolvable then remove possible unflagged 1-production $S$ from $|P|$ by applying Theorem 8 to appropriate rules. Look for new overlaps on every production in which the one-character phrase was replaced by the left part of the 1-production in question. Treat as in step 6. Finally, introduce the new $\alpha$ relation and recompute the $\alpha\lambda^*$ and $(\alpha\lambda^*)\alpha\lambda^*$ matrices. If the intersection of these with their 'ghosts' are all zero Theorem 8 applies and the 1-production can be removed. If not, restore both matrices and their ghosts to their former configurations, and try another 1-production until no more changes can be made.

(8) At this point it is appropriate to attempt to split long rules as discussed in point 1. (No conflicting $\alpha\lambda^*$ relations are introduced because new nonterminals are used.) We must verify that any new overlaps are resolvable, but the relevant relations need not be recomputed. An interactive facility would be useful here because the semantic handler may be affected by the syntax being offered in pieces smaller than original phrases.

(9) Construct $M$ from $P$ and $V$ as they now exist but including $\rightarrow$ relation computed at step 2. Construct $R$ and $L$ similarly.

At this point we have all we need to run Algorithm 1. However, we can go one step further.

(10) Starting from the initial $S_0$ matrix compute all possible $S$ matrices. All
characters in $V_T$ are passed against each $S$ matrix. If a "1" appears in what would be the $T$ vector, a potential parse is indicated. If not, one of three kinds of new $S$ matrices is computed: an error matrix (cf. point 6 above), a duplicate to some $S$ matrix calculated

![Diagram](image)

**Fig. 8.** State transition parser for $G_9$ constructed as in step 10 of Algorithm 2 using $M$ (not $\bar{M}$). All ten productions are carried in these $S$ matrices.
before which has already been computed, or a new $S$ matrix which is added to the list.

In this way a transition table among the $S$ matrices is built up. When a parse of $(y_n, \xi_n)$ is detected from some $S$ matrix, $y_n$ is passed against all the $S$ matrices which transfer to the $S$ in question by some sequence of $|\xi_n| - 1$ transitions. (Alternatively, all of $V_N$ can be passed against every $S$ matrix.) Finally, each $S$ matrix which has no eventual transition to some parse indication is treated as an error matrix.

The result of all this is a set of 'useful' $S$ matrices. From $S_i$ and $x$, a transition table can determine which rule might be parsed, or which $S_j$ matrix is to be entered and whether or not to stack it depending on $x$ (cf. point 3 above). With each parse we can associate a table entry giving the left-part, and the number of $S$ matrices to remove from the stack (taking into account the number of characters of $V_S$ in the phrase being recognized).

The calculation yields a state transition parser. Rather than taking an entire $S$ matrix and computing the next action, we can use the state number (index of that matrix) and the transition table to find out what to do next. This state transition version of Algorithm 1 is most similar to DeRemer's parsers. Like DeRemer's $SLR(1)$ and similar $LR$ parsers, this scheme requires an elaborate computation to compute the state table, but is very efficient and compact at parse time.

Figure 8 gives the result of step 10 of Algorithm 2 for $G_3$, using all productions and $M$ as originally defined. This state transition parser has eight intermediate states, indicated by the $S$ matrices beyond the initial stack symbol. There are nine distinct parse states (circular nodes) for non-$\epsilon$-rules, which are entered whenever right compatibility permits; the $R$ matrix is a companion to this state transition parser. The $\epsilon$-rule does not have its own parse state. It is recognized from a state corresponding to an $S$ matrix because whenever an $\epsilon$-rule is parsed the stack must grow by stacking that matrix. Note that only some of the $S$ matrices are stacked, the others are temporary. Whether a state is to be stacked or not need not be recorded. That information is implicit in the index of the character by which the state is entered when $V$ has been indexed to isolate $V_S$.

VI. Extensions

Suppose a parser has been constructed by Algorithm 2, and then the grammar is altered. If production $n$ is deleted it is easy to ignore future recognitions of that rule; set $L_{i,n} = 0$. Even if a production is added the parser can be altered without repeating all of Algorithm 2 if appropriate matrices are available from its first application. To make sure that the amended grammar is GOR we need only test all overlaps on the new rule, and be sure that resolvability of old overlaps is not altered. If the 'ghost' matrices are kept on all five matrices in step 6 of Algorithm 2, then the latter problem is solved if the recomputation of these five matrices, reflecting the new rule, yields no
conflicts with the ghosts. Then these five new matrices are then used to test overlaps on the new rule. Assuming the amended grammar is GOR the tables for Algorithm 1 are easily altered.

Then Algorithm 1 is used as the parser until the alteration to the grammar is canceled when the former parser is recovered. The original parser may have been Algorithm 1 or a state transition version from Algorithm 2. Note that a state number is sufficient to recover the associated $S$ matrix so that such a state transition parser can revert to Algorithm 1 at any time, as long as the stack contents of state numbers are translated to $S$ matrices. For block-structured extendable languages like ALGOL 68 [15] more than the first state translation will not be necessary; the grammar alteration is cancelled just as the stack is restored to the situation in which the alteration occurred.

Moreover, we conceivably could place references to successive altered parser tables, relations, and ghost matrices on the stack to permit repeated changes in block heads and allow recovery at the end of the blocks. All the while Algorithms 1 remains the basic procedure as long as GOR is maintained; only parameters change. The cost of repeated modifications, no doubt, is heavy.

The recovery from a state number back to the $S$ matrix from which it was generated also might be useful when an error occurs. The $S$ matrix contains explicit information about which characters were expected next by the parser: one of the 1 bits should have advanced down its column. Diagnostics can be computed directly from the $S$ and $M$ matrices saving specific error tables. Possibly the bit pattern might even suggest error recovery techniques.

VII. CONCLUSION

The GOR grammars lie between the precedence and LR groups of grammar classes. If a parser is needed for a static programming language with hand coded error recovery, the use of GOR parsers proposed here is not advised. One can be more efficient with a more powerful parser. If a parser is needed for a dynamic programming language, or in a classroom situation where grammar analysis is done repetitively, the efficient decision procedure of GOR can be very inexpensive and yield a parser, albeit large, with little effort.

The OR parsers have proven themselves useful tools in the classroom in compiler writing courses at the University of Wisconsin. Because the decision procedure does not submerge immediately into state tables, when the grammar is unsatisfactory the student obtains intelligible messages. The unresolvable overlaps describe specific confusing phrases, and an analysis of the available intermediate relations on the vocabulary quickly produces the confusing context. The student can see why his grammar was rejected more easily because he has more diagnostics than a precedence
parser-builder would give. Yet all computations are straightforward so that the program is not so expensive as long as he uses efficient algorithms on the binary matrices. If the language is deterministic, in most practical cases a few 1-productions inserted on the left context of unresolvable overlaps can make the grammar overlap resolvable. [17, Appendix II]

Other results in [17] indicate the large storage requirements for the tables of these parsers. DeRemer's parsers [3] are much smaller and more powerful, but Algorithm 1 will probably run as fast computing state transitions as his algorithms, which must look up more intermediate moves between reading or reducing. The time difference arises from taking advantage of the hardware architecture of binary computers.

The two open areas of research suggested by this work are to find effective modifications on Algorithm 1 to attempt error recovery, and to efficiently implement a language extension at parse time. It appears that the GOR system would lend itself to this latter problem: its decision procedure is easy to modify so extensions do not require recomputation, and the parser can be extended by straightforward table amendment.

**Acknowledgments**

I thank T. Pinkerton, E. Robertson, and J. Williams for their encouragement and D. Surballe and L. Foat for their early implementations.

**References**