1973

Labelled precedence parsing

Mario Schkolnick
Carnegie Mellon University

Follow this and additional works at: http://repository.cmu.edu/compsci
NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.
LABELLED PRECEDENCE PARSING

Mario Schkolnick

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

July 1973

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-73-C-0074) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.
Abstract

Precedence techniques have been widely used in the past in the construction of parsers. However, the restrictions imposed by them on the grammars were hard to meet. Thus, alteration of the rules of the grammar was necessary in order to make them acceptable to the parser. We have shown that, by keeping track of the possible set of rules that could be applied at any one time, one can enlarge the class of grammars considered. The possible set of rules to be considered is obtained directly from the information given by a labelled set of precedence relations. Thus, the parsers are easily obtained. Compared to the precedence parsers, this new method gives a considerable increase in the class of parsable grammars, as well as an improvement in error detection. An interesting consequence of this approach is a new decomposition technique for LR parsers.

1. Introduction

Among the large variety of techniques used for parsing, one can distinguish the bottom-up parsers, as those which attempt to make successive reductions on a given string so as to eventually get to the starting symbol of the grammar. These parsers can be thought of operating in two modes (or phases). On the detection phase, the parser attempts to determine the portion of a right hand side of a phrase within the string which is being considered. Once this boundary is detected, the parser goes into a reduction phase, consisting of selecting a production which is a handle at the determined position.

If we classify different types of bottom-up parsers according to the amount of information they carry while in the detection phase, we can distinguish two extremes. On one hand we have the precedence parsers, which are characterized by the fact that they carry no information while looking for the right hand side of a phrase and by making its decisions in the reduction phase by using local context only. The parsers obtained are relatively simple but the classes of grammars they can parse is restricted by the existence of local ambiguities.

By varying the amount of context examined one can define different families of
grammars. Among the most popular ones, we have the Wirth-Weber precedence \[1\], the simple weak precedence \[2,3\], and the simple mixed strategy precedence \[3\].

On the other side of the spectrum lie the LR(k) parsers \[4\]. While in the detection phase, they carry enough information so that the decision to reduce can be made immediately after a right hand side is detected. The number of states an LR(k) parser has can become immense. Part of this high number of states is due to the fact that different information that is carried forward has to be further distinguished for the same local context.

An intermediate situation is obtained if one separates what is to be considered information which has to be carried forward and information that can be obtained from local context. A parser thus constructed will consist of two machines: a forward machine \(T\) and a decision machine \(D\). The parser will work as follows: Initially the control is given to the \(T\) machine. While on \(T\), the parser behaves like a precedence parser but every time it shifts an input, it stores in the stack the input symbol together with a symbol denoting the state it is currently in. The decision to shift, which is accompanied by a transition to a new state, is done by examining local context. The \(T\) machine can also determine acceptance, an error condition or a call on the \(D\) machine for a decision. The \(D\) machine determines whether a shift or a reduce has to be performed, by examining local context together with the state information that exists on the pushdown. A shift is performed like the \(T\) machine. If a reduce is called for, the right hand side of the production used is removed from the stack, the \(T\) machine is initialized to the state denoted by the topmost symbol, and the left hand side of the production used is given as input to it (this is like an LR(k) parser). A parser of this type is given in Example 1.

**Example 1**

Let \(G\) be given by:

\[
\begin{align*}
S & \rightarrow cAbB \\
A & \rightarrow adaA \\
B & \rightarrow adaB
\end{align*}
\]

\(G\) is not a member of any of the classes of precedence grammars mentioned above. An LR(1) (or an LR(0)) parser for \(G\) has 10 states. We can see that we really need 2 states to carry information forward (i.e. whether a "c" or a "b" was first seen). The rest of the information can be determined from local context. A diagram for the \(T\) machine could be:

The \(D\) machine would check the contents of the stack to match a right hand side of a subset of the productions, determined by the state of \(T\) from which it was called and it would give a decision on which reduction to make. A diagram for \(D\) can be given as a forest:
In this paper we examine parsers built using this approach. Different classes of parsable grammars can be obtained by applying different criteria for the construction of the \( F \) and \( D \) machines. We will see that any class of precedence grammars can be extended this way, without a significant complication of the parsers and with the big advantage of not having to accommodate the rules of the grammar to satisfy the requirements of the particular precedence method used. Although the intent of this study was to extend precedence parsers, we get as a side effect a decomposition method for LR(k) parsers. This approach is a matter of further study.

2. Labeled Precedence Parsing

In this section we examine the construction of different parsers and the classes of grammars they can parse. We assume the reader is familiar with the terminology for context free grammars [7,8]. Since our original attempt was in the direction of extending precedence techniques, all the grammars considered here will be proper. Extensions to non-\( A \)-free grammars can be studied along the same lines.

Definition 1: A proper context free grammar \( G=(V,V_T,P,S) \) is a reduced, \( A \)-free, cycle-free context free grammar. \( V \) denotes the vocabulary, \( V_T \) is the set of terminals, \( V_N \) is the set of nonterminals. We assume the productions in \( P \) are indexed. The set \( I \) of indices will consist of symbols of the form \( A_k \) where \( A \in V_N \). An index \( i = A_k \in I \) will denote the \( k \)-th production whose left hand side is \( A \). If this production is \( A \cdot S \) we will write \( i : A \cdot S \). If there is only one production for nonterminal \( A \) we will use \( A \) instead of \( A_i \) as its index. There will be an index 0 to denote an augmented production of the form \( S' \cdot 1S \) (\( S' \in V \)). (This is just a convenience to make definitions simpler.)

Except where otherwise noted, the following conventions apply throughout the paper:

\[ A,B,C,D \in V_N \; ; \; a,b,c,d,e,g,r \in V_T \; ; \; \beta_j \in V \; ; \; 
X,Y,Z \in V \]

We will now define certain relations between pairs of symbols in \( V \). These relations will be defined in a similar way as was done in [1] but there will be a label attached to them. The labels will provide information about the way the relation between the symbols was obtained.

Definition 2: Let \( X,Y \in V \). Let \( \alpha_1, \alpha_2, \alpha_3, \alpha_4 \in I \). Then,

1) \( X \) is less than \( Y \) under \( \alpha_1, \alpha_2 \), which we will write as \([\alpha_1; \alpha_2] : X < Y \), if \( \forall i \in \alpha_1, j \in \alpha_2 \), such that \( i : A \cdot \beta_1X_1B_1Y_1 \) and \( \alpha_2 = \{ j \mid j : B_2 \cdot C \cdot \sigma, j : C \cdot \gamma \} \).

2) \( X \) is equal than \( Y \) under \( \alpha_3 \), which we will write as \([\alpha_3] : X = Y \), if \( \alpha_3 = \{ i \mid i : A \cdot \beta_1X_1Y_1 \} \).
3) $X$ is greater than $Y$ under $\alpha_4$, which we will write as $[\alpha_4]: X > Y$, if $Y \in V_T$, $\exists i \in I$, i : $A \cdot \mu_B \nu$, $D \neq \phi$, $Y^\rho$ and $\alpha_4 = \{ j \mid B \neq \sigma \cdot C, j : C \cdot \gamma \cdot X \}$

Notice that, ignoring the labeling, the relations are defined as in [1]. Example 2 shows a grammar together with a matrix of labelled relations.

**Example 2.** Let $G$ be defined by the productions

$S_1$: $S \rightarrow bZg$

$S_2$: $S \rightarrow cY$

$S_3$: $S \rightarrow brX$

$Y: Y \rightarrow a$  

$Z: Z \rightarrow ra$  

$X: X \rightarrow a$

The labelled precedence relations can be displayed in matrix form:

<table>
<thead>
<tr>
<th>S</th>
<th>b</th>
<th>c</th>
<th>g</th>
<th>⊥</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>[S_1]:≠</td>
<td>[S_2]:≠</td>
<td>[S_3]:≠</td>
<td>[S_1;S_2]:≠</td>
</tr>
<tr>
<td>Y</td>
<td>[S_1]:≠</td>
<td>[S_2]:≠</td>
<td>[S_3]:≠</td>
<td>[S_1;S_2]:≠</td>
</tr>
<tr>
<td>Z</td>
<td>[S_1]:≠</td>
<td>[S_2]:≠</td>
<td>[S_3]:≠</td>
<td>[S_1;S_2]:≠</td>
</tr>
<tr>
<td>X</td>
<td>[S_1]:≠</td>
<td>[S_2]:≠</td>
<td>[S_3]:≠</td>
<td>[S_1;S_2]:≠</td>
</tr>
<tr>
<td>a</td>
<td>[S_1]:≠</td>
<td>[S_2]:≠</td>
<td>[S_3]:≠</td>
<td>[S_1;S_2]:≠</td>
</tr>
<tr>
<td>g</td>
<td>[S_1]:≠</td>
<td>[S_2]:≠</td>
<td>[S_3]:≠</td>
<td>[S_1;S_2]:≠</td>
</tr>
</tbody>
</table>

(We have listed the elements of the sets $\alpha$, instead of using the usual set notation.)

The matrix of labelled precedence relations will be denoted by $M$. Note that for two symbols $X$ and $Y$ there may be more than one pair of labels $\alpha_1, \alpha_2$ such that $[\alpha_1;\alpha_2]:X<Y$.

We will later perform reductions on this matrix. These will amount to merging some indices into one. We can think of the set of labels as coming from a set $L$ and having a mapping $\mathcal{P}: I \rightarrow L$. The original matrix is defined with $L=I$ and $\mathcal{P} \mathcal{P}^{-1}$. In general though, we will have a labelled precedence matrix $M$ with labels from a set $L$.

Given a labelled matrix of precedence relations we now define a parser for the grammar. The (forward) states of the parser will be subsets of $L$. 
Informally, the parser can be defined as follows: Define a directed graph whose nodes are the members of V (plus two other nodes, denoted by 1, one of them will be the unique source node, the other, the unique sink node in the graph). An arc exists between nodes X and Y if the X-Y entry of the M matrix is not empty. The initial state will be the set consisting of the label for production 0, and we will say it is incident to the source node 1. Now we perform the following operation at every node: Let state s be incident to node X and let there be an arc from X into Y. Let \([\alpha_1, \alpha_2]_{X \rightarrow Y}\) and \([\alpha_3]_{X \rightarrow Y}\). (There may be more than one label of the form \([\alpha_1, \alpha_2]\) for the \(\prec\) relation.) We then define a state t incident to node Y as \(s \cap \alpha_3\) together with the set of all indices of productions in \(\alpha_2\) such that \(s \cap \alpha_1 \neq \phi\). The state t will be referred to as the successor of state s. When no new states are created the process stops. Note that the computation of the states is done using only boolean operations on sets and that checking if a state has already been created is straightforward. (The whole process can be viewed as a parallel operation at all nodes.)

The set of states so created constitutes the set \(Q_F\) of states of the \(\mathcal{T}\) machine. The underlying fsa will be called the unrestricted \(\mathcal{T}\) machine. The parsing of a word proceeds as follows: Initially the \(\mathcal{T}\) machine is in the initial state \(s_0\), incident to node 1. There is a stack which will have two channels, subsequently referred as \(\gamma_1\) and \(\gamma_2\). \(\gamma_1(V \cup \{1\})^*, \gamma_2(Q_F^*)\). Initially \(\gamma_1=1, \gamma_2=s_0\). Let \(\gamma_1=1, X\) for some \(\in V^*\), \(\gamma_2=\{0\} \cap \gamma_1\). Let \(X\) be the next input symbol (normally this is the next symbol in the input string). The \(\mathcal{T}\) machine is in state s incident to node X. Let Y be the next input symbol (normally this is the next symbol in the input string). If \(s \cap [\alpha_4, \phi]\), a shift is performed. This consists in changing state to the successor state t of s and pushing in the stack the symbols Y on the first channel and t on the second. If \(s \cap [\alpha_4, \phi]\) we say that a potential conflict occurs. The set of all productions whose indices are in \(s \cap (\alpha_3, \phi)\), for all \(\alpha_3\), is made available to the \(\mathcal{D}\) machine which (hopefully) will give a unique decision of what to do.

The \(\mathcal{D}\) machine will either determine a shift, by examining productions in \(\gamma_1(\alpha_3, \gamma_4)\), or a reduce to one of the productions in \(\gamma_4\). If a shift is determined, control is transferred to the successor state of s in the machine \(\mathcal{T}\). If a reduce is determined, the right hand side of the production being reduced is popped up from the stack, control is transferred to the topmost state now appearing on channel 2, and the input symbol fed to machine \(\mathcal{T}\) is the left hand side of the production used. The parser accepts if the input symbol is 1, \(\mathcal{T}\) is in its final state and \(\gamma_1=1, S\).

We will now define the \(\mathcal{T}\) machine.

\(\mathcal{T}\) is a finite state machine, \(\mathcal{T}=(Q_F, \Gamma \times \Gamma, \delta, (\phi, \phi, (\phi, \phi))\), where \(Q_F\) is a subset of the set of all subsets of \(\Gamma\), \(\Gamma \times \Gamma\) is the input alphabet, the initial (and final) state is the set containing \(\phi, \phi\) and \(\delta_F\) is defined as follows: Let \(s \in Q_F, (X, Y) \in \Gamma \times \Gamma\). The \((X, Y)\) entry of M contains labels \([\alpha_1, \alpha_2], [\alpha_3, \alpha_4]\) (there may be many labels of type \([\alpha_1, \alpha_2]\)).

\[
\delta_F(s, (X, Y)) = \begin{cases} 
\text{if } s \cap \alpha_4 = \phi & \text{then } (s \cap \alpha_3) \cup \{0\} \cup \alpha_2 \\
\end{cases}
\]

\(\text{else } \mathcal{D}\)

(\(\mathcal{D}\) in the range of \(\delta_F\) is interpreted as a call to machine \(\mathcal{D}\)). The empty state is interpreted as an error indication. The transition function for the unrestricted \(\mathcal{T}\) machine is
The D machine can be defined in different ways, giving rise to different classes of parsable grammars. We will give some definitions here. For simplicity, we will restrict to local contexts of one symbol, but these constructions can be extended to other contexts. We will need some definitions which we now give:

**Definition 3:** Let $s \in V^*$. We denote by $f_k$ an operator such that $f_k s$ is the longest prefix of $s$ of length $\leq k$. We denote by $f_k^* s = \{f_k p \mid s \Rightarrow^* p\}$. Similarly we define $l_k^*$ for suffix strings.

Let $(Z,s)$ be an interior symbol of a $2$-channel stack (i.e., the stack is $\gamma = (\gamma_1, \gamma_2)$, $|\gamma_1| = |\gamma_2| > 1$, and for some $n > 1$, $f_1 l_n \gamma_1 = Z$, $f_1 l_n \gamma_2 = s$).

Let $i:A \cdot s$ be the production whose index is $i$. If $[\alpha_1 \alpha_2] : Z \Rightarrow_T s \cdot f_1^* \phi$, $A \cdot \alpha_2$ we say that (the distinguished occurrence of) $Z$ leads into production $i$.

If $\exists n > 1$, $l_n \gamma_1 = f_n Z \Rightarrow_T Z \cdot s'$ and (the distinguished occurrence of) $Z$ leads into production $i$ then (the distinguished occurrence of) $s'$ is a valid expansion of production $i$.

If $[\alpha_1 \alpha_2] : X \Rightarrow_T Y$ or $[\alpha_3] : X \Rightarrow_T Y$ and for some state $s$, $s \cdot f_1 \alpha_1 \alpha_2 \cdot s \Rightarrow_T X$ then we will say that $X$ leads into $Y$ under $s$. We will write $[s] : X \Rightarrow_T Y$.

If $\exists \alpha$ and $[\alpha] : X \Rightarrow_T Y$ we will sometimes write $(i) : X \Rightarrow_T Y$. A similar convention holds for the other labels.

Now we can give a definition for the D machine. The D machine is specified as follows:

a: If $i \in P \cap s \cdot f_1^* \phi$, $i : A \cdot \beta X$, $n = |\beta X| + 1$, $l_n \gamma_1 = Z \beta X$ and $Z$ leads into $i$, then "reduce $i"; 

b: "\{\phi\} \cdot \phi(t = (s \cdot f_1^* \phi) \mid l \alpha_2, i : A \cdot \beta X C \Rightarrow_T Y \cdot f_1^* C, s \cdot l_1^* \phi \mid n = |\beta X| + 1, l_n \gamma_1 = Z \beta X$ and $Z$ leads into $i\}" 

(when $D$ is called, the parser has $Y$ as input and $\gamma_1 = \alpha \cdot X$)

This D machine works as follows: For each production $i : A \cdot s$ in $s \cdot f_1^* \phi$ it checks that $s$ appears as a valid expansion of $i$. If so, machine $D$ outputs "reduce $i". Also, it may output a state consisting of the set of all labels of productions $i : A \cdot \beta X C \Rightarrow_T Y \cdot f_1^* C$ such that $Y \Rightarrow^* C$, $[s] : X \Rightarrow_T Y$ and such that $\beta X$ appears as a valid expansion of $i$. Thus, the D machine could produce more than one output. We are interested in deterministic behavior so we will say that a parser is well defined if the D machine has at most one output. (An empty output from $D$ is an indication of error.)

The class of grammars which have deterministic parsers whose D machine are defined as above and whose $T$ machines have $n$ states will be called the class of $n$-state labelled precedence grammars with independent left and right context ($n$-LPI grammars).
Let us compute the machines $I$ and $D$ for the grammar in Example 2:

$I$ machine

<table>
<thead>
<tr>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>${0}$</td>
</tr>
<tr>
<td>${S_1, S_3}$</td>
</tr>
<tr>
<td>${S_2}$</td>
</tr>
<tr>
<td>${S_i}$</td>
</tr>
<tr>
<td>${Z, S_3}$</td>
</tr>
<tr>
<td>${X, Z}$</td>
</tr>
<tr>
<td>${Y}$</td>
</tr>
<tr>
<td>${S_3}$</td>
</tr>
</tbody>
</table>

Whenever a call to the $D$ machine is given, the set of all $i$ such that $\varphi(i) \cap (A_i \cup B_i \cup C_i)$ is given. The $D$ machine can be represented as a forest where the root of each tree is labelled by an element $I$ of $L$ and the corresponding tree represents all right hand sides of productions $i$ such that $\varphi(i) \cap L$. In this case, $L = \{I\}$ and $\varphi$ is $1-1$ so there is one tree for each production. The parsers constructed as above will be such that their $I$ machines usually have more states than it is necessary. We can get minimal machines $I$ as follows: Assume we have a definition for the class of $D$ machines. We then define an incompatibility relation on the set of productions $I$. We will say that two productions $i_1, i_2$, are incompatible if when a call to $D$ occurs with state $s = \varphi(i_1) = \varphi(i_2)$, $D$ will produce more than one output. Once we have determined all incompatible pairs of productions we will define a new set $L$ and a new function $\varphi$ such that if $i_1$ and $i_2$ are incompatible then $\varphi(i_1) \neq \varphi(i_2)$. (In other words we are defining an equivalence relation on $I$.)

Note that a call to $D$ occurs whenever there is an entry in the matrix $M$ containing a relation $\propto$. The incompatibilities are defined below. Let $\not\in$ denote incompatibility between productions.

1) $A_i \not\in C_k$ if $\exists X, Y$ such that $(C_i, B_j) : X < Y, (A_i) : X = Y, A_i : A_0 \cdots \Psi Y Z \beta Z \nu$ and $(A_i) : Z \propto W$ for some $W \in \Gamma_1^* \nu$ or $\nu = \lambda$ and $\exists W$ such that $(A_i, B_j) : Z \propto W$. 

[Diagram of forest with states $S_1$, $S_2$, $S_3$, $X$, $Y$, $Z$, $b$, $c$, $g$, $r$, $a$, $\propto$, $\lambda$, and production rules: reduce $S_1$, reduce $S_2$, reduce $S_3$, reduce $Y$, reduce $Z$, reduce $X$.]


2) \( C_k \neq D_m \) if there are productions \( A_j:A \rightarrow \gamma \beta \gamma Z \cdot \), \( B_j:B \rightarrow \gamma \beta Z \cdot \) there is \( \nu \) such that \( (C_k;A_j):\nu < Y \) and \( (D_m;B_j):\nu \neq Y \) and \( (B_j):Z \neq W \) for some \( W \neq \nu \) or \( \nu = \lambda \) and \( \exists W \) such that \( (A_i;B_j):Z \neq W \).

Given the set of incompatible productions, we can define a partition \( \pi \) on the set of productions such that if \( A_i;A_j \) are incompatible productions they belong to different classes. For each class we define a symbol. Let \( L \) be the set of all these symbols and define the natural map \( \pi: I \rightarrow L \) such that \( \pi = \pi \) if \( i \) and \( j \) belong to the same class of \( \pi \). We can now define the \( \mathcal{T} \) and \( \mathcal{D} \) machines as before. For some partitions \( \pi \) it may happen that \( \mathcal{D} \) will not be well defined. But if the parser defined on the identity partition was well defined, there exist a partition for which the parser is well defined and for which the number of states of the machine \( \mathcal{T} \) is minimal. This number gives an indication on the amount of information that has to be carried forward to successfully parse the sentences of the language generated by the grammar. It is clear that, for each \( \pi \), we can define grammars for which the \( \mathcal{T} \) machine will have at least \( \pi \) states, so this gives a measure of the complexity of the grammar.

As the following result shows, even the simple class of grammars in this hierarchy, i.e., those for which the number of states of the \( \mathcal{T} \) machine is 1, is an extension of the largest class of grammars defined using precedence relations over \( V \times V \), i.e., the class of simple mixed strategy precedence.

**Theorem 1:** The class of SMSP grammars is contained in the class of 1-LPI grammars.

**Proof:** Let \( G \) be a SMSP grammar. Assume there are two productions \( A_j: A \rightarrow \gamma \beta \gamma Z \cdot \), \( B_j: B \rightarrow \gamma \beta Z \cdot \). Let \( \nu \neq \lambda \) and \( \forall (f, \nu) \neq \nu \). Since \( Z \neq W \) or \( Z = W \) we cannot have \( Z \neq W \). In particular, we cannot have \( (A_j):Z \neq W \). If \( \nu = \lambda \) we cannot have \( X \neq B \) or \( X \neq B \) since, in particular, there is no index \( C_k \) such that \( (C_k;B_j):X \neq Y \). So no incompatibilities of type 1 can occur. If there are two productions \( A_j: A \rightarrow \gamma \beta \gamma Z \cdot \), \( B_j: B \rightarrow \gamma \beta Z \cdot \) then again, if \( \nu \neq \lambda \) there can be no \( W \neq \nu \neq \nu \) such that \( (B_j):Z \neq W \). If \( \nu = \lambda \) then \( A_i \) and \( B_j \) have identical right hand sides. So, there is no \( \nu \) such that \( (A_i;A_j):X \neq Y \) and \( (B_j;B_j):Y \neq Z \). In particular, there are no \( C_k, D_m \) such that \( (C_k;A_i):Y \neq Z \) and \( (D_m;B_j):Y \neq Z \). Thus no incompatibilities of type 2 occur. Thus, we can define \( \mathcal{D} \) with one state. It is easy to see the \( \mathcal{D} \) is deterministic.

The class of 1 state labelled grammars with independent left and right context has been presented in the literature under another name as indicated by the following result.

**Theorem 2:** The class of 1 state labelled grammars with independent left and right context coincides with the class of overlap resolvable (OR) grammars [5].

**Proof:** The reader is referred to [5] for the definition of OR grammars. A case analysis shows that \( \mathcal{D} \) has a deterministic behavior if every conflict is left or right resolvable.

Thus we get the following corollary, which answers a conjecture of Wise:

**Corollary 1:** The class of OR languages coincides with the class of deterministic languages.

**Proof:** Follows from the fact that every deterministic language has an SMSP grammar.
Example 2 presented a grammar which failed to be OR. There are two entries in M which can cause incompatibilities, namely M(a,g) and M(g,1). For the latter we have that productions Y and S1 are not of the form occurring in case 1 or 2 for the definition of incompatibility. For the former, we do have that S2#Z. Thus, at least 2 states are required for the I machine. It turns out that 2 states are sufficient to get a parser for this grammar.

Because we have defined the D machine as one which checks left and right context independently we have the following result.

**Theorem 3:** For any n, the class of n-state labelled grammars with independent left and right context is properly included in the class of SLR(1) grammars [6].

**Proof:** Given the set $Q_0$ of sets of LR(0) items for a grammar and the set $Q_1$ of states of the unrestricted I machine, we can define a mapping $h$ from $Q_0$ to $Q_1$ as follows: $h(S_0)=\{0\}$. Let $S_1$ be a set of LR(0) items. For each symbol $YV$ we can partition $S_1$ in 5 sets, $S_1=S_{11} \cup S_{12} \cup S_{13} \cup S_{14} \cup S_{15}$, $S_{11}:=\{A\cdot aX.Y\}$, $S_{12}:=\{A\cdot aX.ZB.Z\neq Y\}$, $S_{13}:=\{A\cdot aX.\}$, $S_{14}:=\{A\cdot aY.\}$, $S_{15}:=\{A\cdot aZB.Z\neq Y\}$. If $h(S_1)=q_i$ then $h(S(S_1,Y))=\delta'(q_{ii},(X,Y))$, where $\delta'$ is the transition function of the unrestricted I machine and $\delta(S_1,Y)=S_1$ is the set of LR(0) items obtained as the GOTO(S1,Y) (see [7] for undefined terms). Now we make the following claim.

**Claim:** If $S_1$ is a set of LR(0) items partitioned as above, then $h(S_1)$ contains the indices of all productions in $S_{11} \cup S_{12} \cup S_{13}$.

The claim is certainly true for $S_0$ because $S_{01}=S_{02}=S_{03}=\emptyset$. Now, assuming the claim holds for $S_1$, we note that GOTO(S1,Y) is obtained by taking all productions in $S_{11} \cup S_{14}$ with the dot shifted over the symbol Y (which becomes the set $S_{11} \cup S_{12} \cup S_{13}$), and applying a closure operator to get the set $S_{14} \cup S_{15}$. But, for every index i of a production in $S_{11}$ we have $(i):X=Y$, and for every index j of a production in $S_{14}$, there is an index i of a production in $S_{11}$ such that $(i;j):X<Y$. Thus, all indices of productions in $S_{11} \cup S_{12} \cup S_{13}$ appear in state $h(S_1)$ and the claim holds.

It is now straightforward to verify that if G is not SLR(1), i.e., if there are two conflicting items in some set $S_1$ of LR(0) items, then the corresponding state of the I machine will produce a call of the D machine which will in turn, give more than one output. Thus the parser will not be a deterministic one and the grammar will not be an n-state LP1 grammar.

We note that to generate the I machine we do not distinguish positions within a production, as an LR(or SLR) parser does. Thus, we are able to get the I machine faster, but we restrict the class of grammars which can be parsed, excluding those which have productions in which a repeated occurrence of a symbol may cause problems, as suggested by the following example:

**Example 3:** Let G have productions

```
S : abcabA | abB
A : d
B : d
```
Since \([\{0;S_1,S_2\}:1\leq a, [S_1,S_2]:a\leq b\) and \([S_1,A]:b\leq d, [S_2,B]:b\leq d\) and \([A,B]:d\leq l\) we have that the \(T\) machine calls the \(D\) machine when in state \{A,B\} and reading symbol (d,l). The \(D\) machine gives as output both "reduce A" and "reduce B". This behavior will occur even if the \(D\) machine checks the left and right context simultaneously as is done later.

On the other hand, it is easily seen that \(G\) is an SLR(1) grammar. Example 3 leads us to the following definition:

**Definition 4:** Let \(A X_1 X_2 \ldots X_{n-1} X_n\) be a production. We will say that this production is **free of repetitions** (FOR) if for all \(1 \leq i,j \leq n\) we have \(i\not=j\) implies \(X_i \not= X_j\) (i.e., there is no repeated occurrence of a symbol among the first \(n-1\) symbols). A grammar will be **free of repetitions** (FOR) if all of its rules are FOR. FOR grammars and FOR productions occur very often. Any grammar in normal 2 form is a FOR grammar and every CF language can be given a trivial FOR grammar. Among the grammars used in programming languages, a quick glance at some reveals that: PL360 as defined in [9, pages 39-53] is FOR; SNOBOL4, as defined in [7, pages 505-507], has only one non FOR rule; ALGOL 60, as defined in [10], has only one non FOR rule (which happens to be a production for the <for list element>!); PAL, as defined in [7, pages 512-514], is FOR.

If we are dealing with FOR grammars, we can strengthen the result of Theorem 3:

**Theorem 4:** If \(G\) is FOR and SLR(1), then it is n-LPI.

**Proof:** Define the \(T\) machine using the identity map \(\varphi: I \to I\). If \(G\) is FOR, the claim stated in the proof of Theorem 3 becomes the following:

**Claim:** If \(S_1\) is a set of LR(0) items partitioned as before, then \(h(S_1)\) coincides with the set of indices of all productions in \(S_1 \cup S_2 \cup S_3\).

To prove the claim, it suffices to show that there are no indices of productions in \(h(S_1)\) which are not in \(S_1 \cup S_2 \cup S_3\). This follows from the fact that, if \((i):X \prec Y\) or \((i,j):X \prec Y\) then, since \(G\) is FOR, there is only one occurrence of \(X\) in the production whose index is \(i\) since an LR(0) item is identified by this symbol, the map \(h\) is 1-1. It is easy to see that the parser constructed is isomorphic to the SLR(1) parser.

Thus, if we restrict our attention to FOR grammars, both classes coincide. Moreover, the SLR(1) parser can be obtained very easily from the \(T\) machine so that a fast procedure for constructing SLR(1) parsers is obtained. As mentioned above, FOR productions and grammars occur frequently in programming languages. Thus, we should take advantage of this fact when constructing parsers for them.

We will now modify the definition of the \(D\) machine so as to make it check for simultaneous left and right context. We need to introduce the following definition.

**Definition 5:** A symbol \(Y\) is adjacent to symbols \(X\) and \(Z\) within the context of a production \(C_i\) if either

1) \((C_j):X \prec Y\) and either \((C_j):Y \prec Z\) or \((C_j):Y \succ Z\)

or

2) \((C_j,D_k):X \prec Y\) and \((D_k):Y \prec Z\) for some production \(D_k\).
Let \( A_t : A \rightarrow \delta \) be a production and \( \varphi(A) = \{ B | B \rightarrow A \} \). We say that \( A \) is a valid reduction for \( \delta \) within symbols \( X \) and \( Z \) and state \( s \) if

1) \( (C_j ; A_j) : X \rightarrow \delta \) for some \( C_j \)'s
2) \( \exists Y : \varphi(A) \) such that \( Y \) is adjacent to symbols \( X \) and \( Z \)
   within the context of production \( C_j \).

Note that we can check the condition of valid reduction by inspecting the matrix \( M \). As the following lemma shows, we get information about possible simultaneous left and right context in which a nonterminal may appear.

**Lemma**: Let \( C_j : X \rightarrow Y \), \( Y \rightarrow \delta \). Let \( S \rightarrow \alpha \beta \Rightarrow \alpha \gamma \delta \beta \Rightarrow \alpha \gamma \delta \epsilon \), with \( \alpha, \beta, \epsilon, \gamma, \delta \) such that \( \varphi(Y) \) for some \( \varphi(A) \) such that \( \varphi(Y) \Rightarrow \delta \). Then \( A \) is a valid reduction for \( \delta \) within symbols \( X \) and \( Z \) and some state \( s \) such that \( C_j < s \).

**Proof**: We know \( C_j : X \rightarrow Y \). There are two cases: \( c = Y \) or \( c \neq Y \) (since \( \varphi(Y) = \delta \)). In the first case, \( C_j : X \rightarrow Y \). Also, either \( Z \rightarrow \delta(c') \) or \( c' = \delta \) and \( Z \rightarrow \delta \). Then, either \( (C_j) : X \rightarrow Y \) or \( (C_j) : Y \rightarrow Z \). If \( c = Y \) then \( (D) : Y \rightarrow Y \) such that \( c = \delta \rightarrow Y \delta = Y \) with \( \delta = Y \). Then \( Z = \delta(c) \) so \( (C_j) : X \rightarrow Y \) and \( (D) : Y \rightarrow Z \). In either case, \( Y \) is adjacent to symbols \( X \) and \( Z \) within the context of \( C_j \). Since \( \varphi(A) = \delta \), we have \( (C_j ; A_j) : X \rightarrow \delta \) where \( A_j = \delta \). Thus we have that conditions 1) and 2) of definition 5 are satisfied.

We are now in a position to specify another class of parsers, by changing the \( D \) machine. The change will only affect the instruction labelled \( a \). This instruction is changed to:

\[ a: \text{if } \exists i, Y = i \Rightarrow A, \quad \text{then } \text{"reduce } i\text{".} \]

We will now construct a parser for a grammar using this machine \( D \).

**Example**: Let \( G \) be

\[
S_1: \quad S \rightarrow Aa \\
S_2: \quad S \rightarrow dAb \\
S_3: \quad S \rightarrow Db \\
S_4: \quad S \rightarrow dBa \\
A: \quad A \rightarrow c \\
B: \quad B \rightarrow c
\]

The matrix \( M \) is:

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>([S_1] = \delta)</td>
<td>([S_2] = \delta)</td>
<td>([S_3] = \delta)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>([S_4] = \delta)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>([S_1, S_4] = \delta)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>([S_2, S_3] = \delta)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>([A, B] = \delta)</td>
<td>([A, B] = \delta)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>([S_3] = \delta)</td>
<td>([S_1] = \delta)</td>
<td>([S_4] = \delta)</td>
<td>([S_1, B] = \delta)</td>
<td>([S_2] = \delta)</td>
<td>([S_1, A] = \delta)</td>
<td>([S_1, B] = \delta)</td>
<td>([S_1, S_2] = \delta)</td>
</tr>
</tbody>
</table>
The machine $T$ is:

\[
\begin{array}{cccccccc}
\emptyset & \{S_1\} & \{S_3\} & \{A,B\} & \{S_2,S_4\} & \{S_2\} & \{S_4\} & \{\hat{S}_4\} \\
\{\emptyset\} & \{S_1\} & \{S_3\} & \{A,B\} & \{S_2,S_4\} & \{S_2\} & \{S_4\} & \{\hat{S}_4\} \\
\{S_1\} & \{S_3\} & \{A,B\} & \{S_2,S_4\} & \{S_2\} & \{S_4\} & \{\hat{S}_4\} \\
\{S_1\} & \{S_3\} & \{A,B\} & \{S_2,S_4\} & \{S_2\} & \{S_4\} & \{\hat{S}_4\} \\
\{S_1\} & \{S_3\} & \{A,B\} & \{S_2,S_4\} & \{S_2\} & \{S_4\} & \{\hat{S}_4\} \\
\{S_1\} & \{S_3\} & \{A,B\} & \{S_2,S_4\} & \{S_2\} & \{S_4\} & \{\hat{S}_4\} \\
\{S_1\} & \{S_3\} & \{A,B\} & \{S_2,S_4\} & \{S_2\} & \{S_4\} & \{\hat{S}_4\} \\
\{S_1\} & \{S_3\} & \{A,B\} & \{S_2,S_4\} & \{S_2\} & \{S_4\} & \{\hat{S}_4\} \\
\end{array}
\]

The forest for machine $D$ is as follows:

\[
\begin{array}{cccccccc}
\emptyset & \{S_1\} & \{S_2\} & \{S_3\} & \{S_4\} & \{A,B\} & \{A,B\} \\
\{S_1\} & \{S_2\} & \{S_3\} & \{S_4\} & \{A,B\} & \{A,B\} \\
\{S_1\} & \{S_2\} & \{S_3\} & \{S_4\} & \{A,B\} & \{A,B\} \\
\{S_1\} & \{S_2\} & \{S_3\} & \{S_4\} & \{A,B\} & \{A,B\} \\
\{S_1\} & \{S_2\} & \{S_3\} & \{S_4\} & \{A,B\} & \{A,B\} \\
\{S_1\} & \{S_2\} & \{S_3\} & \{S_4\} & \{A,B\} & \{A,B\} \\
\{S_1\} & \{S_2\} & \{S_3\} & \{S_4\} & \{A,B\} & \{A,B\} \\
\{S_1\} & \{S_2\} & \{S_3\} & \{S_4\} & \{A,B\} & \{A,B\} \\
\end{array}
\]

reduce $S_1$ reduce $S_2$ reduce $S_3$ reduce $S_4$ $d$:reduce $B$ reduce $A$

When $D$ is called with $\{A,B\}$ it knows its lookahead symbol. Assume it is an "a". Then it checks that the stack contains "c" and looks at the left context. If it is a $d(\{S_2,S_4\})$ it checks to see if $A$ or $B$ are valid reductions of $c$ within $d$ and $a$ and state $\{S_2,S_4\}$. From the matrix $M$ we see that $B$ is valid while $A$ is not. Thus the output "reduce $B$" is given.

We could proceed as before and give a criteria for incompatible productions. We will not do this here, but is clear we again get a hierarchy depending on the number of states the $T$ machine has. In the above example we really didn't need the states in the $T$ machine in order to decide the output for the $D$ machine. Thus, we could have built a parser with 1 state in the $T$ machine. Actually, we have

**Theorem 5:** The class of 1-state labelled precedence grammars with simultaneous left and right context is properly included in the class of (1-1)BRC. If the grammars are restricted to be FOR, these classes coincide.

**Proof:** Because the $D$ machine can check for context of at most one to both left and right of
the right hand side of a production we have that we are within the (1-1)BRC. The following
grammar is (1-1)BRC but not in the class of labelled precedence grammars considered:

\[
S \rightarrow aAbAc\alpha Abc \\
A \rightarrow d \\
B \rightarrow d
\]

It thus remain to be shown that any FOR grammar which is (1-1)BRC is in this class.

This follows from the fact that for a FOR grammar, the converse of lemma 1 holds, i.e., if A
is a valid reduction for \( \varepsilon \) within symbols X and Z then XAZ is a substring of some sentential
form. Thus, if the D machine gives more than one output, it means that knowledge of the
left and right context of a handle of a sentential form does not uniquely determines it. Thus,
G is not (1-1)BRC.

3. A decomposition of LR parsers

So far, we have considered parsers which operate as precedence parsers, in the sense that,
once a reduction could occur (as determined by the T machine) we would check the contents
of the stack to either determine the production to use in the reduction, or to continue the
forward scan.

This sequentiality of actions is clearly not necessary. Since the D machine, when called, only
inspects a bounded amount of tape (not more than one plus the length of the longest right
hand side of any production), we can construct a (definite) machine which can operate in
parallel with the T machine and which performs the checking that D does. (We will also
refer to this new machine as the D machine.) In this way, the decisions are already taken
when the T machine requests them.

Now the parser is behaving exactly as an LR parser, but since we have separated the
functions in the T and D machines, the total number of states is reduced. As an example of
these ideas, consider the following grammar:

\[
S: \quad S \rightarrow DADB \\
D: \quad D \rightarrow aC \\
A_1: \quad A \rightarrow b \\
A_2: \quad A \rightarrow c \\
C_1: \quad C \rightarrow C e \\
C_2: \quad C \rightarrow e
\]

From the M matrix we can determine the incompatibilities. We find there are none. Thus one
state is sufficient for the T machine. (In fact, G is an OR grammar, though not an SMSP).
The T machine is obtained directly from the matrix of (unlabelled) precedence relations. It
has only one state, which is denoted by \( \alpha \). A call to D is denoted by D.
To obtain $D$ we reduce (using standard techniques of finite state machines) the machine which checks all productions. Since there is only one state in $T$, the only information $D$ has, to determine its output, is the input from which it is called from $T$. The following is the transition table for $D$. It has 5 states. Notice that the input to $D$ is taken as the second component of the input to $T$ (i.e., the "new" input symbol, not the one already on top of the stack). The output depends on both.

Next state, under new symbol.

<table>
<thead>
<tr>
<th>State</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>(b,,-):A _</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(a,,-):C _</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td></td>
<td>(c,a):A _</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>(c,,-):B _</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>(d,,-):B _</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>(C,,-):D</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>x</td>
<td>(e,,-):C _</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>5</td>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>x</td>
<td>1</td>
<td></td>
<td>-</td>
<td></td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

(A don't care entry is shown as -. An error entry is shown as x.) The following example shows a sequence of configuration taken by the parser when given an input string. Since $T$ has 1 state we do not show it on the stack. The state of $D$ appears as a second component.
<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action of machine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>aebaecal</td>
<td>shift</td>
</tr>
<tr>
<td>1</td>
<td>aebaecal</td>
<td>shift</td>
</tr>
<tr>
<td>1 a</td>
<td>baecal</td>
<td>D reduce C₂</td>
</tr>
<tr>
<td>1 1 2</td>
<td>baecal</td>
<td>D reduce D</td>
</tr>
<tr>
<td>1 D</td>
<td>baecal</td>
<td>shift</td>
</tr>
<tr>
<td>1 3</td>
<td>aecal</td>
<td>D reduce A₁</td>
</tr>
<tr>
<td>1 3 b</td>
<td>aecal</td>
<td>shift</td>
</tr>
<tr>
<td>1 3 4</td>
<td>aecal</td>
<td>shift</td>
</tr>
<tr>
<td>1 3 4 a</td>
<td>ecal</td>
<td>shift</td>
</tr>
<tr>
<td>1 3 4 1</td>
<td>ecal</td>
<td>shift</td>
</tr>
<tr>
<td>1 3 4 1 2</td>
<td>cal</td>
<td>D reduce C₂</td>
</tr>
<tr>
<td>1 3 4 1 2</td>
<td>cal</td>
<td>D reduce D</td>
</tr>
<tr>
<td>1 3 4 5</td>
<td>cal</td>
<td>shift</td>
</tr>
<tr>
<td>1 3 4 5 1</td>
<td>al</td>
<td>D reduce A₂</td>
</tr>
<tr>
<td>1 3 4 5 x</td>
<td>al</td>
<td>error</td>
</tr>
</tbody>
</table>
Had the last symbol "a" not been there, the last two configurations would have been changed to:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action of machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 D A D c</td>
<td>1</td>
<td>$\mathfrak{T}$, D</td>
</tr>
<tr>
<td>1 3 4 5 1</td>
<td></td>
<td>D reduce $B_1$</td>
</tr>
<tr>
<td>1 D A D B</td>
<td>1</td>
<td>D reduce $S$</td>
</tr>
<tr>
<td>1 3 4 5 1</td>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>

It is interesting to note that this grammar has an 18-state LR(1) parser (constructed a la Knuth), a 14-state parser (using Korenjak's method [11]), and a 10-state SLR(1) parser. By allowing the parser to postpone error detection (as the one above does), Aho and Ullman constructed a 7-state parser [7]. We have shown that using decomposition techniques one can get a 3+5-state parser for this grammar. Because of the simple way the $\mathfrak{T}$ and $\mathfrak{D}$ machines are determined, this decomposition technique appears quite useful.

We should point out here that, although not explicitly mentioned, a similar decomposition technique appears in [12].

4. Conclusions

Keeping track of the possible productions which can be in use at any one time during the operation of a precedence parser can significantly enlarge the class of grammars to which it applies. We have shown how to obtain such parsers and given some ideas about their relative power. An additional feature over conventional precedence parsers is the improved error detection capability. The fact that we have more than one state during the detection phase allows the parser to discover errors before they are detected by conventional precedence parsers. In fact, these parsers look very much like LR parsers, but are easier to obtain, and they are considerably smaller than these. By "reversing" the machine which decides which reduction to perform we were able to get parsers which are equivalent to LR parsers obtained using error postponement techniques [7] but, again, at a substantial savings in the number of states. More work is needed concerning this method of LR decomposition.
References


