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 succesive 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 righthand 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 existance 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.

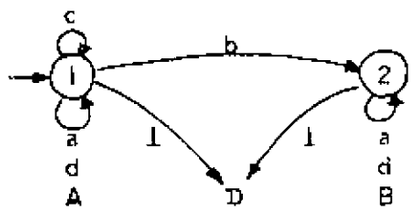
An intermediate situation is obtained if one separates what is to be considered information which has to be carried forward and information that can be obtained from local context. A parser thus constructed will consist of two machines: a forward machine $\mathcal{F}$ and a decision machine $\mathcal{D}$. The parser will work as follows: Initially the control is given to the $\mathcal{F}$ machine. While on $\mathcal{F}$, 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 $\mathcal{F}$ machine can also determine acceptance, an error condition or a call on the $\mathcal{D}$ machine for a decision. The $\mathcal{D}$ 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 $\mathcal{F}$ machine. If a reduce is called for, the right hand side of the production used is removed from the stack, the $\mathcal{F}$ 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

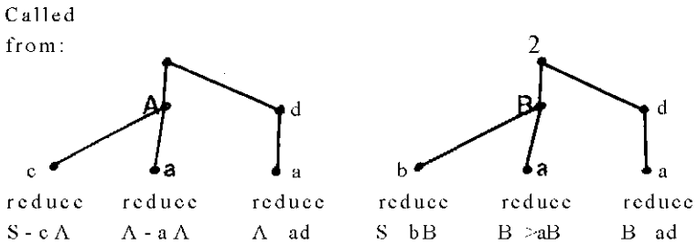
Let G be given by:

S → cA | bB
A → ad | aA
B → ad | aB

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 $\mathcal{F}$ machine could be:



The $\mathcal{D}$ 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 $\mathcal{F}$ from which it was called and it would give a decision on which reduction to make. A diagram for $\mathcal{D}$ 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 7 and *V* 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 accomodate 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.

*<L Labejed 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 *U* 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 < VN. An index i=A_k (I will denote the k-th production whose left hand side is A. If this production is AS we will write i: A & (or A_k: A &). If there is only one production for nonterminal A we will use A instead of Ai as its index. There will be an index 0 to denote an augmented production of the form S' *LSI* (S' *C* 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 < V_n ; a,b,c,d,e,g,r < V_T; fttf,S,c,P,**i**/,P,**cr**,7 < V*; X,Y,Z < 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 2i Let X,Y **C** V. Let C4, **ci**$_2$, a_j, a* € I. Then,

1) X is less than Y under aj, a_{j2}, which we will write as |a*; a_2| : X <• Y, if Vi < c*i, **3** A,B,X,P,iy, such that i : A •> pXBi/ and a_i − {j | B *U* C **cr** , j : C - Y7}.

2) X [s equal than Y under cr_j, which we will write as **[03]** : X − Y, if **03** − {i I i : A > pXYiy}

3) X is greater than Y under α_4, which we will write as $[\alpha_4] : X \gg Y$, if $Y \in V_T$, $\exists\, i \in I$, $i : A \to \rho BD\nu$, $D \overset{*}{\Rightarrow} Y\beta$ and $\alpha_4 = \{j \mid B \overset{*}{\Rightarrow} \sigma C, j : C \to \gamma X\}$

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 → bZg	Y : Y → ag
S_2: S → crY	Z : Z → ra
S_3: S → brX	X : X → a

The labelled precedence relations can be displayed in matrix form:

	S	b	c	g	⊥
S					[∅]:≐
Y					[S_2]:⋗
Z				[S_1]:≐	
X					[S_3]:⋗
a				[Y]:≐	[X]:⋗
				[Z]:⋗	
g					[Y,S_1]:⋗
⊥	[∅]:≐	[∅;S_1,S_3]:⋖	[∅;S_2]:⋖		

	Y	Z	X	a	r
b		[S_1]:≐			[S_1;Z]:⋖
					[S_3]:≐
c					[S_2]:≐
r	[S_2]:≐		[S_3]:≐	[S_2;Y]:⋖	
				[S_3;X]:⋖	
				[Z]:≐	

(We have listed the elements of the sets α, 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 α_1, α_2 such that $[\alpha_1;\alpha_2]:X \lessdot Y$.

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 $\varphi:I \to L$. The original matrix is defined with $L=I$ and φ 1-1. 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 $\perp$, 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 $\emptyset$, and we will say it is incident to the source node $\perp$. 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 $[\alpha_1;\alpha_2]:X \lessdot Y$ and $[\alpha_3]:X \doteq Y$. (There may be more than one label of the form $[\alpha_1;\alpha_2]$ for the $\lessdot$ 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 α_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 $\perp$. There is a stack which will have two channels, subsequently referred as γ_1 and γ_2. $\gamma_1 \in (V \cup \{\perp\})^*$, $\gamma_2 \in Q_F^*$. Initially $\gamma_1 = \perp, \gamma_2 = s_0$. Let $\gamma_1 = \perp\gamma X$ for some $\gamma \in V^*$, $\gamma_2 = \{\emptyset\}\sigma s$ for $\sigma \in Q_F^*$, $|\gamma| = |\sigma|$, be the contents of the stack at some point in the computation. (Thus 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). Let $[\alpha_4]:X \gtrdot Y$. 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 $s \cap \alpha_4 \neq \phi$ we say that a potential conflict occurs. The set of all productions whose indices are in $s \cap (\alpha_4 \cup \alpha_3 \cup \alpha_1)$, for all α_1, 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 $s \cap (\alpha_3 \cup \alpha_1)$, or a reduce to one of the productions in $s \cap \alpha_4$. If a shift is determined, control is transferred to the succesor 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 transfered 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 $\perp$, $\mathcal{T}$ is in its final state and $\gamma_1 = \perp S$.

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

$\mathcal{T}$ is a finite state machine, $\mathcal{T} = (Q_F, V \times V, S_F, \varphi^{-1}(\emptyset), \{\varphi^{-1}(\emptyset)\})$, where Q_F is a subset of the set of all subsets of L, $V \times V$ is the input alphabet, the initial (and final) state is the set containing $\varphi^{-1}(\emptyset)$ and δ_F is defined as follows: Let $s \in Q_F, (X,Y) \in V \times V$. The (X,Y) entry of M contains labels $[\alpha_1;\alpha_2]$, $[\alpha_3]$,$[\alpha_4]$ (there may be many labels of type $[\alpha_1,\alpha_2]$).

$$\delta_F(s,(X,Y)) = \underline{\text{if}}\ s \cap \alpha_4 = \phi \quad \underline{\text{then}}\ (s \cap \alpha_3) \cup \bigcup_{s \cap \alpha_1 \neq \phi} \alpha_2 \quad \underline{\text{else}}\ \mathcal{D}$$

($\mathcal{D}$ in the range of δ_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

Sr'(s,(X,Y)) « (sHa,) IJ I) a.
s H a ^

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 &(V⁻. We denote by i_z an operator such that f,& is the longest prefix of S of length <k We denote by f> * an operator such that f,*& = {f,P| &!>P}. Similarly we define l,S for suffix strings.

Let (Z,s) be an interior symbol of a 2-channel stack (i.e., the stack is ^=(#1,^2)) I ^ I H ^ ^ 1 > and for some n > 1, fil.tfi~Z, fi'n^-s)-

Let i:A ^& be the production whose index is i. If [cq; 02] *. Z <? fi&, sfla^4>, A<ct2 we say that (the distinguished occurrence of) Z leads into production i.

If 3n>l, l„Vi=f,Z&=ZS' and (the distinguished occurrence of) Z leads into production i then (the distinguished occurrence of) £>' is a valid expansion of production i.

If [ar,cc2]:X<:Y or [c<3]:X=Y and for some state s, $sf\langle cxif\rangle cx^{)}$^ then we will say that X leads into Y under s. We will write [s]:X Y.

If iCa and [cr]:X=Y we will sometimes write (i):X=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 3i, yiCsfla*, i:A (3X, n«||3X|+l, l„*i=Z£X and Z leads into i, then "reduce i";

b: ·{'Pi|9'i<t=(sna,) II U a_2 , i:A £XC&, YCf^C,
sU<V<t>

nH0X|+l, l„^i=Z(3X and Z leads into i}"

(when D is called, the parser has Y as input and tfj^crX)

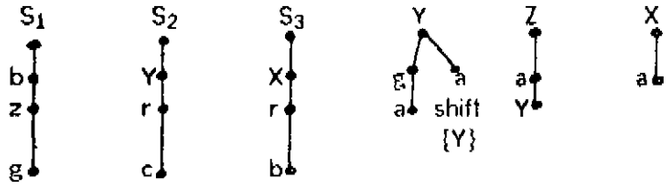
This *1D* machine works as follows: For each production i:A *h* in sfla* it checks that & appears as a valid expansion of i. If so, machine P outputs "reduce i". Also, it may output a state consisting of the set of all labels of productions i:A £XCS such that Y<fi*C, [s]:X Y and such that *(SX* appears 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 7 machines have n states will be called the class of n-state labelled precedence grammars with independent left and riftht context (n-LPI grammars).

Let us compute the machines $\mathcal{T}$ and $\mathcal{D}$ for the grammar in Example 2:

	$\mathcal{T}$ machine States							
(X,Y)	{∅}	$\{S_1,S_3\}$	$\{S_2\}$	$\{S_1\}$	$\{Z,S_3\}$	{X,Z}	{Y}	$\{S_3\}$
⊥S	{∅}							
⊥b	$\{S_1,S_3\}$							
⊥c	$\{S_2\}$							
bZ		$\{S_1\}$						
br		$\{Z,S_3\}$						
cr			$\{S_2\}$					
Zg				$\{S_1\}$				
rY			$\{S_2\}$					
rX					$\{S_3\}$			
ra			{Y}		{X,Z}			
g⊥				$\mathcal{D}(S_1)$			$\mathcal{D}(Y)$	
Y⊥			$\mathcal{D}(S_2)$					
X⊥								$\mathcal{D}(S_3)$
ag						$\mathcal{D}(Z)$	{Y}	
a⊥						$\mathcal{D}(X)$		
S⊥	end							

Whenever a call to the $\mathcal{D}$ machine is given, the set of all i such that $\varphi i \in s \cap (\alpha_4 \cup \alpha_1 \cup \alpha_3)$ is given. The $\mathcal{D}$ machine can be represented as a forest where the root of each tree is labelled by an element l of L and the corresponding tree represents all right hand sides of productions i such that $\varphi i = l$. In this case, L=I and φ is 1-1 so there is one tree for each production.



The parsers constructed as above will be such that their $\mathcal{T}$ machines usually have more states than it is necessary. We can get minimal machines $\mathcal{T}$ as follows: Assume we have a definition for the class of $\mathcal{D}$ 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 $\mathcal{D}$ occurs with state $s \supset \varphi i_1 \supset \varphi i_2$, $\mathcal{D}$ 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 φ 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 $\mathcal{D}$ occurs whenever there is an entry in the matrix M containing a relation $\gg$. The incompatibilities are defined below. Let $\#$ denote incompatibility between productions.

1) $A_i \# C_k$ if $\exists X,Y$ such that $(C_k;B_j):X \ll Y$, $(A_i):X \neq Y$, $A_i:A \to \mu X Y \beta Z$, $B_j:B \to Y \beta Z \nu$ and $(A_i):Z \gg W$ for some $W \ll f_1^* \nu$ or $\nu = \Lambda$ and $\exists W$ such that $(A_i,B_j):Z \gg W$.

2) $C_k \neq D_m$ if there are productions $A_i:A \to Y\beta Z\nu$, $B_j:B \to Y\beta Z$, there is V such that $(C_k;A_i):V \lessdot Y$ and $(D_m;B_j):V \lessdot Y$ and $(B_j):Z \gtrdot W$ for some $W \lessdot f_1^*\nu$ or $\nu=\Lambda$ and $\exists W$ such that $(A_i,B_j):Z \gtrdot W$.

Given the set of incompatible productions, we can define a partition π on the set of productions such that if i_1,i_2 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 $\varphi:I \to L$ such that $\varphi i=\varphi j$ if i and j belong to the same class of π. We can now define the $\mathcal{T}$ and $\mathcal{D}$ machines as before. For some partitions π 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 informalion that has to be carried forward in order to successfully parse the sentences of the language generated by the grammar. It is clear that, for each n, we can define grammars for which the $\mathcal{T}$ machine will have at least n 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 VxV, 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 $A_i:A \to \mu XY\beta Z$, $B_j:B \to Y\beta Z\nu$. Let $\nu \neq \Lambda$ and $W \lessdot f_1^*\nu \cap V_T$. Since $Z \lessdot W$ or $Z \neq W$ we cannot have $Z \gtrdot W$. In particular, we cannot have $(A_i):Z \gtrdot W$. If $\nu=\Lambda$ we cannot have $X \doteq B$ or $X \lessdot B$ so, in particular, there is no index C_k such that $(C_k;B_j):X \lessdot Y$. So no incompatibilities of type 1 can occur. If there are two productions $A_i:A \to Y\beta Z\nu$, $B_j:Y\beta Z$ then again, if $\nu \neq \Lambda$ there can be no $W \lessdot f_1^*\nu \cap V_T$ such that $(B_j):Z \gtrdot W$. If $\nu=\Lambda$ then A_i and B_j have identical right hand sides. So, there is no V such that $(V,A) \lessdot \doteq$ and $(V,B) \lessdot \doteq$. In particular, there are no C_k, D_m such that $(C_k;A_i):V \lessdot Y$ and $(D_m;B_j):V \lessdot Y$. Thus no incompatibilities of type 2 occur. Thus, we can define $\mathcal{T}$ 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 iff every conflict is left or right resolvable. ▌

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

Corolary 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,⊥). For the latter we have that productions Y and S_1 are not of the form occuring in case 1 or 2 for the definition of incompatibility. For the former, we do have that $S_2 \neq Z$. Thus, at least 2 states are required for the $\mathcal{T}$ machine. It turns out that 2 states are sufficient to get a parser for this grammar.

Because we have defined the $\mathcal{D}$ 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_F of states of the unrestricted $\mathcal{T}$ machine, we can define a mapping h from Q_0 to Q_F as follows: $h(S_0)=\{0\}$. Let S_i be a set of LR(0) items. For each symbol $Y \in V$ we can partition S_i in 5 sets, $S_i = S_i^1 \cup S_i^2 \cup S_i^3 \cup S_i^4 \cup S_i^5$, $S_i^1=\{A \rightarrow \alpha X.Y\beta\}$, $S_i^2=\{A \rightarrow \alpha X.Z\beta | Z \neq Y\}$, $S_i^3=\{A \rightarrow \alpha X.\}$, $S_i^4=\{A \rightarrow .Y\beta\}$, $S_i^5=\{A \rightarrow .Z\beta | Z \neq Y\}$. If $h(S_i)=q_i$ then $h(\delta(S_i,Y))=\delta'(q_i,(X,Y))$, where δ' is the transition function of the unrestricted $\mathcal{T}$ machine and $\delta(S_i,Y)=S_j$ is the set of LR(0) items obtained as the GOTO(S_i,Y) (see [7] for undefined terms). Now we make the following claim.

Claim: If S_i is a set of LR(0) items partitioned as above, then $h(S_i)$ contains the indices of all productions in $S_i^1 \cup S_i^2 \cup S_i^3$.

The claim is certainly true for S_0 because $S_0^1=S_0^2=S_0^3=\phi$. Now, assumming the claim holds for S_i, we note that GOTO(S_i,Y) is obtained by taking all productions in $S_i^1 \cup S_i^4$ with the dot shifted over the symbol Y (which becomes the set $S_j^1 \cup S_j^2 \cup S_j^3$), and applying a closure operator to get the set $S_j^4 \cup S_j^5$. But, for every index i of a production in S_i^1 we have (i):X$\doteq$Y, and for every index j of a production in S_i^4, there is an index i of a production in $S_i^1 \cup S_i^2$ such that (i;j):X$\lessdot$Y. Thus, all indices of productions in $S_j^1 \cup S_j^2 \cup S_j^3$ appear in state $h(S_j)$ 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_i of LR(0) items, then the corresponding state of the $\mathcal{T}$ machine will produce a call of the $\mathcal{D}$ 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 LPI grammar. ∎

We note that to generate the $\mathcal{T}$ machine we do not distinguish positions within a production, as an LR(or SLR) parser does. Thus, we are able to get the $\mathcal{T}$ machine faster, but we restrict the class of grammars which can be parsed, excluding those which have productions in which a repeated occurrance 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 $[\emptyset;S_1,S_2]:\perp \lessdot a$, $[S_1,S_2]:a \doteq b$ and $[S_1;A]:b \lessdot d$, $[S_2;B]:b \lessdot d$ and $[A,B]:d \gtrdot \perp$ we have that the $\mathcal{T}$ machine calls the $\mathcal{D}$ machine when in state $\{A,B\}$ and reading symbol $(d,\perp)$. The $\mathcal{D}$ machine gives as output both "reduce A" and "reduce B". This behavior will occur even if the $\mathcal{D}$ 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 $A \cdot X_1X_2...X_{n-1}X_n$ be a production. We will say that this production is free of repetitions (FOR) if for all $1 \leq i,j < n$ we have $i \neq j$ implies $X_i \neq X_j$ (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 $\mathcal{T}$ machine using the identity map $\varphi:I \to L=I$. If G is FOR, the claim stated in the proof of Theorem 3 becomes the following:

Claim: If S_i is a set of LR(0) items partitioned as before,then $h(S_i)$ coincides with the set of indices of all productions in $S_i^1 \cup S_i^2 \cup S_i^3$.

To prove the claim, it suffices to show that there are no indices of productions in $h(S_i)$ which are not in $S_i^1 \cup S_i^2 \cup S_i^3$. This follows from the fact that, if $(i):X \doteq Y$ or $(i;j):X \lessdot Y$ 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 $\mathcal{T}$ 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 $\mathcal{D}$ 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_j if either

1) $(C_j):X \doteq Y$ and either $(C_j):Y \cdot Z$ or $(C_j):Y \gtrdot Z$
or
2) $(C_j;D_k):X \lessdot Y$ and $(D_k):Y \cdot Z$ for some production D_k.

Let $A_i:A\to\delta$ be a production and $\mathcal{P}(A)=\{B|B\overset{*}{\Rightarrow}A\}$. We say that A is a valid reduction for δ within symbols X and Z, and state s if

1) $(C_j;A_i):X\lessdot f_1\delta$ for some $C_j\in s$
2) $\exists Y\in\mathcal{P}(A)$ 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 1: Let $C_j:C\to\gamma X\epsilon$, $\gamma\in V^*,\epsilon\in V^*$. Let $S\overset{*}{\Rightarrow}\alpha C\beta\Rightarrow\alpha\gamma X\epsilon\beta\overset{*}{\Rightarrow}\alpha\gamma XY\epsilon'\beta\overset{*}{\Rightarrow}\alpha\gamma XY Z\epsilon''$, with $\alpha,\beta,\epsilon',\epsilon''\in V^*$ (but $Z\in f_1^*(\epsilon'\beta)$) for some $Y\in\mathcal{P}(A)$ such that $\mathcal{P}(Y)=\phi$. Then A is a valid reduction for δ within symbols X and Z and some state s such that $C_j\in s$.

Proof: We know $C\Rightarrow\gamma X\epsilon\overset{*}{\Rightarrow}\gamma XY\epsilon'$. There are two cases: $\epsilon=Y\epsilon'$ or $\epsilon\neq Y\epsilon',\epsilon'\neq\Lambda$ (since $\mathcal{P}(Y)=\phi$). In the first case, $(C_j):X\doteq Y$. Also, either $Z\in f_1^*(\epsilon')$ or $\epsilon'=\Lambda$ and $Z\in f_1^*(\beta)$. Then, either $(C_j):Y\cdot Z$ or $(C_j):Y\rangle Z$. If $\epsilon\neq Y\epsilon'$ then $\exists D_j:D\to Y\mu$ such that $\epsilon\overset{*}{\Rightarrow}D\mu'\Rightarrow Y\mu\mu'=Y\epsilon'$ with $\mu\neq\Lambda$. Then $Z\in f_1^*(\mu)$ so $(C_j;D_j):X\lessdot Y$ and $(D_j):Y\cdot Z$. In either case, Y is adjacent to symbols X and Z within the context of C_j. Since $Y\overset{*}{\Rightarrow}A\Rightarrow\delta$ we have $(C_j;A_i):X\lessdot f_1\delta$ where $A_i:A\to\delta$. 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 $\mathfrak{D}$ machine. The change will only affect the instruction labelled a. This instruction is changed to:

a: if $\exists i,\ \forall i\in s\cap\alpha_4$, $i:A\to\beta X$, $n=|\beta X|+1$, $l_n\gamma_1=Z\beta X$, Z leads into i and A is a valid reduction for βX within symbols Z and Y and state s, where $s=f_1l_n\gamma_2$ (i.e., the state which appears next to Z) then "reduce i".

We will now construct a parser for a grammar using this machine $\mathfrak{D}$.

Example: Let G be

S_1:	$S\to Aa$	S_3:	$S\to Bb$	A:	$A\to c$
S_2:	$S\to dAb$	S_4:	$S\to dBa$	B:	$B\to c$

The matrix M is:

	S	A	B	a	b	c	d	⊥
S								[Ø]:≐
A				[S_1]:≐	[S_2]:≐			
B				[S_4]:≐	[S_3]:≐			
a								[S_1,S_4]:⟩
b								[S_2,S_3]:⟩
c				[A,B]:⟩	[A,B]:⟩			
d		[S_2]:≐	[S_4]:≐			[S_4;B]:⋖,[S_2A]:⋖		
⊥	[Ø]:≐	[Ø;S_1]:⋖	[Ø;S_3]:⋖			[Ø;A,B]:⋖	[Ø;S_2,S_4]:⋖	

The machine 7 is:

	{9}	{Si}	{S3}	{A,B}	{S2,S4}	{S5}	{S4}
IS	{0}						
iA	{Si} •						
lB	{S3}						
1c	{A,B}						
id	{S2,S4}						
Sl	end						
Aa		{Si}					
Ab						{S2}	
Ba							{S2}
Bb			{S3}				
ca				D({A,[3})			
cb				»({A,B})			
dA					{S2}		
dB							
dc					{A,B}		
al		X>({Si})					
bl			D({S2})			D({S5})	

The forest for machine D is as follows:

{Si}	{S3}	{S5}	{S2}	{A,B}	{A,B}
•	4			a	b
	b	b	a		
	A	B *	B f	c*	
	d		d		

reduce Sireduce S2 reduce S3 reduce S4 d:reduce B reduce A
i:reduce A reduce B

When D is called with {A,B} it knows its lookahead symbol. Assume it" is an "a". Then it checks that the stack contains V and looks at the left context. If it is a (d,{S2,S4}) it checks to see if A or B are valid reductions of c within d and a and state {S5>S4}. 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 7 machine has. In the above example we really didn't need the states in the 7 machine in order to decide the output for the D machine. Thus, we could have built a parser with 1 state in the 7 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 (l-l)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 → aAbAc|aBc
A → d
B → 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 δ within symbols X and Z then XAZ is a substring of some sentential form. Thus, if the $\mathfrak{D}$ 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 $\mathfrak{T}$ 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 $\mathfrak{D}$ 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 $\mathfrak{T}$ machine and which performs the checking that $\mathfrak{D}$ does. (We will also refer to this new machine as the $\mathfrak{D}$ machine.) In this way, the decisions are already taken when the $\mathfrak{T}$ machine requests them.

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

S:	S → DADB	B_1:	B → c
D:	D → aC	B_2:	B → d
A_1:	A → b	C_1:	C → Ce
A_2:	A → c	C_2:	C → e

From the M matrix we can determine the incompatibilities. We find there are none. Thus one state is sufficient for the $\mathfrak{T}$ machine. (In fact, G is an OR grammar,though not an SMSP). The $\mathfrak{T}$ machine is obtained directly from the matrix of (unlabelled) precedence relations. It has only one state, which is denoted by α. A call to $\mathfrak{D}$ is denoted by $\mathfrak{D}$.

Input	Action	Input	Action	Input	Action
IS	a	ae	a	CA	⟫
ID	a	AD	a	Cb	D
Ia	a	Aa	ot	Cc	D
SI	end	BI	D	Cd	D
DA	a	bD		Ce	a
DB	a	ba	D	eA	D
Db	a	cD	D	eb	D
Dc	a	ca	D	ec	D
Dd	a	cI	***V***	ed	D
aC	a	dI	D	ee	

To obtain ***V*** we reduce (using standard techniques of finite state machines) the machine which checks all productions. Since there is only one state in 7, the only information ***J>*** has, to determine its output, is the input from which it is called from 7. 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 7 (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.

State]	A	B	C	D	a	b	c	d	e	Output
1			2	3	1	-	-	-	2	(B,-):S (b,-):A! (e,-):C (c,a):A (c.IhBi (d,-):B
2								-	1	(C,-):D (e,-):C
3	4	x	-	-	-	1	1	x		-
4	-		-	5	1	-	-			
5	x	1	-	-	-	x	1	1	-	-

(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 7 has 1 state we do not show it on the stack. The state of D appears as a second component.

Stack	Input	Action of machine T	D	
⊥	aebaecal⊥			
1		shift		
⊥ a	ebaecal⊥			
1 1		shift		
⊥ a e	baecal⊥			
1 1 2			D	reduce C_2
⊥ a C	baecal⊥			
1 1 2			D	reduce D
⊥ D	baecal⊥			
1 3		shift		
⊥ D b	aecal⊥			
1 3 1			D	reduce A_1
⊥ D A	aecal⊥			
1 3 4		shift		
⊥ D A a	ecal⊥			
1 3 4 1		shift		
⊥ D A a e	cal⊥			
1 3 4 1 2			D	reduce C_2
⊥ D A a C	cal⊥			
1 3 4 1 2			D	reduce D
⊥ D A D	cal⊥			
1 3 4 5		shift		
⊥ D A D c	al⊥			
1 3 4 5 1			D	reduce A_2
⊥ D A D A	al⊥			
1 3 4 5 x				error

Had the last symbol "a" not been there, the last two configurations would have been changed to:

Stack	Input	Action of machine $\mathcal{T}$	$\mathcal{D}$
⊥ D A D c 1 3 4 5 1	⊥	$\mathcal{D}$	reduce B_1
⊥ D A D B 1 3 4 5 1	⊥	$\mathcal{D}$	reduce S
⊥ S	⊥	end	

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 1+5-state parser for this grammar. Because of the simple way the $\mathcal{T}$ and $\mathcal{D}$ 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 postponment techniques [7] but, again, at a substantial savings in the number of states. More work is needed concerning this method of LR decomposition.

References

1. Wirth, N. and H. Weber [1966], EULER - a generalization of ALGOL and its formal definition, Parts 1 and 2," Comm. ACM 9:1, 13-23 and 9:2, 89-99.

2. Ichbiah, J. D., and S. P. Morse [1970], "A technique for generating almost optimal Floyd-Evans productions for precedence grammars," Comm. ACM 13:8, 501-508.

3. Aho, A. V., P. J. Denning, and J. D. Ullman [1972], "Weak and mixed strategy precedence parsing," J.ACM 19:2,225-243

4. Knuth, D. E. [1965], "On the translation of languages from left to right," Information and Control 8:6, 607-639.

5. Wise, D. S. [1971], "Domolki's algorithm applied to generalized overlap resolvable grammars," Proc. Third Annual ACM Symp. on Theory of Computing, 171-184.

6. DeRemer, F. L. [1971], "Simple LR(k) grammars," Comm. ACM 14:7, 453-460.

7. Aho, A. V. and Ullman, J. D. [1972-3], The Theory of Parsing, Translation and Compiling, Prentice-Hall.

8. Ginsburg, S. [1966], The Mathematical Theory of Context-Free Languages, McGraw-Hill, New York.

9. Wirth, N. [1968], "PL360 - a programming language for the 360 computers," J.ACM 15:1, 37-74.

10. Naur, P. (ed.) [1963], "Revised report on the algorithmic language ALGOL 60," Comm. ACM 6:1, 1-17.

11. Korenjak, A. J. [1969], "A practical method for constructing LR(k) processors," Comm. ACM 12:11, 613-623.

12. Harrison, M. A. and Havel I. M., "On the parsing of deterministic languages," to be published.