Construction Methods of LR Parsers

Karl Max Schimpf
University of Pennsylvania

Follow this and additional works at: https://repository.upenn.edu/cis_reports

Part of the Electrical and Computer Engineering Commons

Recommended Citation


This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_reports/725
For more information, please contact repository@pobox.upenn.edu.
Construction Methods of LR Parsers

Abstract
This paper presents five different LR parser generators and an error recovery method which is derived directly from the LR parser. The parsers presented include the original LR (1) parser defined by Knuth. The SLR(1) and LALR(1) parsers defined by DeRemer, and the weak and strong compatible LR parsers presented by Pager. All five parsers have been implemented by the author using two programs. Furthermore, the implementation of the SLR (1) parser generator includes an error recovery method and produces an SLR(1) parser with error recovery built in.

Disciplines
Electrical and Computer Engineering

Comments
A thesis presented to the Faculty of Engineering and Applied Science of the University of Pennsylvania in partial fulfillment of the requirements for the degree of Master of Science in Engineering for graduate work in Computer and Information Science.

Jean H. Gallier

Aravind Joshi
This paper presents five different LR parser generators and an error recovery method which is derived directly from the LR parser. The parsers presented include the original LR(1) parser defined by Knuth, the SLR(1) and LALR(1) parsers defined by DeRemer, and the weak and strong compatible LR parsers presented by Pager. All five parsers have been implemented by the author using two programs. Furthermore, the implementation of the SLR(1) parser generator includes an error recovery method and produces an SLR(1) parser with error recovery built in.
# Table of contents

**Chapter I** : Introduction 1  

**Chapter II** : The construction of LR(1) parsing tables 6  
  II.1 LR(1) grammars 6  
    II.1.1 Derivations 7  
    II.1.2 Languages generated by context free grammars 8  
  II.2 Sentential forms and their viable prefixes 9  
  II.3 LR(1) characteristic automata 10  
  II.4 Construction of LR(1) parsers 17  

**Chapter III** : Methods for reducing states in LR(1) parsers 22  
  II.1 SLR(1) parsers 23  
  II.2 LALR(1) parsers 29  
  II.3 Pager's Weak compatibility 32  
  II.4 Pager's Strong compatibility 40  

**Chapter IV** : An error recovery method for LR parsers 52  

**Chapter V** : Implementation 68  
  V.1 Representation of the parsing tables 69  
  V.2 SLR(1) implementation 76  
    V.2.1 Input grammar 76  
    V.2.2 Running the SLR(1) parser constructor 79  
    V.2.3 Interpretation of the output file 80  
    V.2.4 Conflict resolution 85  
    V.2.5 Size restrictions 91
V.3 LR(1), LALR(1), weak and strong compatible parser generators

V.3.1 Input grammar 92

V.3.2 Running the program 96

Appendix A: Sample PASCAL skeleton for use of the SLR(1) parsing tables 103

Appendix B: Sample PASCAL skeleton for use of the LR(1), LALR(1), Weak and Strong compatibility parser generators 111

References 118
Chapter I

Introduction

It is a well known fact that of all the deterministic string parsers, the class of LR parsers recognize the largest class of context free languages [Knu65]. LR parsers are quite powerful and are able to recognize virtually all programming languages in existence today. These parsers were first introduced by Knuth [Knu65] with his original version known as an LR(1) parser. Unfortunately, his method requires extensive resources and hence is impractical to use for parsing any programming language.

Several alternative parsing methods have since been presented which reduce the resource requirements thus producing more practical LR parsers. Some of these parsers accomplish this result by reducing the class of languages accepted by the parsers. The result is a reduction in the
number of parse states built and hence an overall reduction in the resource requirements. The most common forms of this type of LR parser are the SLR(1) and LALR(1) parsers presented by DeRemer [DeR69].

Another form of resource reduction used by LR parsers is a method of performing state minimization on the LR(1) parser. Two of these state minimization methods have been proposed by Pager [Pag77a, Pag77b] called weak and strong compatible LR parsers. In these parsers, he restricts the state reductions to maintain the power of the LR(1) parser and hence the resultant parser recognizes the same class of languages as the original LR(1) parser.

This paper presents five different LR parser generators and an error recovery method which is derived directly from the LR parser. The parsers presented include the original LR(1) parser defined by Knuth [Knu65], the SLR(1) and LALR(1) parsers defined by DeRemer [DeR69], and the weak and strong compatible LR parsers presented by Pager [Pag77a]. All five parsers have been implemented by the author using two programs. Furthermore, the implementation of the SLR(1) parser generator includes the implementation of an error recovery method and produces an SLR(1) parser with error recovery built in.
The method of construction of the weak and strong compatible LR parsers, presented by Pager [Pag77a], unfortunately only provides a partial explanation of the algorithms which build these parsers. These algorithms also contain minor inconsistancies and omissions which tend to obscure the basic nature of the algorithms. This paper presents Pager's algorithms in a modified notation which simplifies the comprehension of the code. It also provides a more complete explanation of the algorithms, and includes a few minor algorithms omitted by Pager.

The problem with LR parsers, when used in a compiler, is that they are designed as a syntactic method which only decides if the given input string belongs to a language in the class accepted by the LR parser. Hence, once the first illegal input symbol is found, the parser stops reporting failure. However, when a compiler parses a program, it is advantageous to have the compiler report as many additional errors as possible.

In order to improve the LR parser's capabilities for use in a compiler, this paper also presents a purely syntactic error recovery scheme to recognize additional errors. Furthermore, the method has been designed so that it can be directly incorporated into the LR parser. Hence, no additional routines are necessary in order to perform error recovery and parse the rest of the input.
The method used in this paper to handle error recovery is based on the method used by Pennello and DeRemer [P&D79], which has a separate error recovery routine that includes error correction. The control strategy used is to search the remainder of the input, starting from the illegal symbol, and verify that it only consists of "viable fragments" (substrings derivable from its grammar). The error recovery method presented in this paper has been implemented using the SLR(1) parser as its basis. However, the method is general enough that the same method could easily be applied to any of the other LR parsers presented in this paper.

Chapter two starts by setting up preliminary notation for context free languages and derivations. This notation is used to describe the basic strategy used by LR parsers. The last sections of the chapter cover the actual construction methods which will yield the LR(1) parser as its result.

Chapter three describes how each of the other four implemented parser constructors are built. The SLR(1) and LALR(1) construction methods are presented using the LR(0) characteristic automaton as their basis for construction. Pager's notion of compatibility, the definitions of both weak and strong compatibility, and the algorithms used in conjunction with the construction of these two parsers are
also described.

Chapter four discusses the error recovery method and an algorithm which takes in an LR parser and produces an LR parser with error recovery. It also explains how an LR parser is used to parse an input string and decide if the string is derivable from the grammar used to generate the LR parser.

Chapter five concludes the paper by discussing briefly the two programs used for the implementation. One program constructs an SLR(1) parser with error recovery built in. The other program, using our modification of Pager's concept of compatibility, can build either an LR(1), LALR(1), weakly or strongly compatible LR parser.
Chapter II

The construction of the LR(1) parsing tables

This chapter describes how LR(1) parsing tables are created. In order to do this, let me start out by setting up some preliminary notation.

II.1 LR(1) Grammars

A Context-Free Grammar (denoted CFG) $G$ is a quadruple $G = (N, T, P, S)$ where

- $T$ is a finite alphabet of terminal symbols;
- $N$ is a finite alphabet of nonterminal symbols;
- $(N \cup T)$ is the finite set of grammar symbols;
- $S$ is a nonterminal symbol in $N$, called the start symbol; and
- $P$ is a finite set of pairs $(A, a)$, called productions,
such that \( A \in N \) and \( a \in (N \cup T)^* \)

A production \((A, a)\) will be denoted in the form \( A \rightarrow a \). Also there is a special \textit{start} production \( S \rightarrow S' \) where \( S' \in N \) and \( S \) does not occur in any other production in \( P \). There is also a special symbol \( \$ \in T \), which denotes the end of the string being parsed, and does not appear in any production.

For notational convenience, upper case letters will be used to denote nonterminal symbols, lower case letters to denote terminal symbols, underlined upper case letters to denote grammar symbols, and underlined lower case letters to denote strings of grammar symbols (strings in \((N \cup T)^*\)). The symbol \( \epsilon \) will be reserved to denote the empty string.

\subsection*{II.1.1 Derivations}

Given a CFG \( G = (N, T, P, S) \), let the relation \( \Rightarrow : (N \cup T)^* \times (N \cup T)^* \) be defined by the set of pairs

\[
\{ (\text{abc,abc}) \mid B \in N; \ a, b, c \in (N \cup T)^*; \\
\text{and } B \rightarrow b \text{ in } P \}\n\]

In other words, given any string in \((N \cup T)^*\) of the form \( \text{abc} \), with \( B \) a nonterminal symbol in \( N \) and given the production \( B \rightarrow b \) in \( P \), we say that the string \( \text{abc} \) derives the string \( \text{abc} \) in a \textit{one step derivation} using \( B \rightarrow b \). This will be denoted as \( \text{abc} \Rightarrow \text{abc} \). Also, let \( \Rightarrow^+ \) and \( \Rightarrow^* \) denote the transitive and transitive reflexive closures of \( \Rightarrow \)
respectively.

From the above relation, we can define another relation which implies an ordering of the rewrite steps. Let this new relation $\Rightarrow_R : (N \cup T)^* \times (N \cup T)^*$ be defined as the set

$$\{ aBc \Rightarrow_R abc | aBc \Rightarrow abc \text{ and } c \in T^* \}$$

In other words, $\Rightarrow_R$ is the one step derivation, when the derivation is applied to the rightmost nonterminal occurring in the string $aBc$. Let $\Rightarrow^+_R$ and $\Rightarrow^*_R$ denotes the transitive and transitive reflexive closures of $\Rightarrow_R$, respectively.

### 11.1.2 Language generated by a context-free grammar

Given a CFG $G = (N, T, P, S)$, the language $L(G)$ generated by $G$ is the set of strings

$$L(G) = \{ a \mid S \Rightarrow^* a, a \in T^* \}$$

**Note:** The order in which $\Rightarrow$ is applied has no effect on the resulting terminal string produced. Hence the language $L(G)$, generated by $G$, could be alternatively be defined as the set

$$L(G) = \{ a \mid S \Rightarrow^*_R a \text{ where } a \in T^* \}$$

Using the above definitions, an LR(1) grammar can be loosely defined as follows:
An LR(1) grammar is a CFG $G$, such that each string $a \in L(G)$ (derived via a rightmost derivation) can be parsed deterministically in a single scan from left to right, having the ability to look ahead one symbol from the point of scanning.

### II.2 Sentential forms and their viable prefixes

An LR(1) parser, when scanning the input (of a string to be parsed), is essentially looking for a match with one or more strings that can be derived from the CFG’s start symbol. More formally, the LR(1) parser is trying to recognize a sentential form which is an element in the set

$$\{ a \mid S \Rightarrow^* a \text{ and } a \notin (N \cup T)^* \}$$

In recognizing a sentential form, the LR(1) parser is really interested in knowing whether it has scanned enough of the input string such that a reduction can be performed, that is, when the sentential form is the string $s = abc$ where $a, b \in (N \cup T)^*; \ c \in T^*; \text{ and } B \rightarrow b \in P$. Knowing this information, a reduction of $b$ to $B$ can be made to get the rightmost derivation string that $s$ came from. This is known as finding the handle. The handle is defined as the pair $(|ab|, B \rightarrow b)$ such that $S \Rightarrow^* abc$. The $|ab|$ denotes the length of the handle, which states the position where the string $b$ can be reduced to $B$ using $B \rightarrow b$. The string
ab is called the **viable prefix** or characteristic string [A&U77].

Using the above definitions, it is fairly easy to characterize what an LR(1) parser does. It scans the input, from left to right, looking for a viable prefix. Once finding it, the string is reduced with the corresponding production of the viable prefix. Using the reduced string derived from the viable prefix concatenated with the unscanned input, the parser repeats the above process looking for another viable prefix. This continues until either the input has been reduced to the start symbol, or failure occurs by not finding any legal viable prefixes.

**II.3 LR(1) Characteristic Automaton**

It is fundamental result that viable prefixes derived from CFG's are regular. Therefore a deterministic finite automaton, called the **characteristic automaton** for a CFG, can be built to recognize the set of legal viable prefixes. Furthermore, once the characteristic automaton has been built, the LR(1) parser can be directly derived from it.

Let a marked production be of the form \( A \rightarrow a \cdot b \) where \( A \rightarrow ab \) is a production in \( P \), and "." is assumed to be a symbol not in the set of grammar symbols \((N U T)\). These
marked productions will be used to denote "how much" of the production's right hand side has been recognized in the string being scanned. Hence the marked production \( A \rightarrow a \cdot b \) represents the fact that the LR(1) parser has scanned the string \( sa \), where \( s \) is some string that occurred before the string \( a \) in the input.

Expanding this to include a set of look-ahead symbols, let an item be defined as the pair \( [A \rightarrow a \cdot b, LA] \) where \( A \rightarrow a \cdot b \) is a marked production, and \( LA \) is a subset of \( T \) denoting the set of all terminal symbols which can follow the production and is called the set of lookahead symbols. Items, essentially, describe two things:

i) What portion of a production's right hand side can occur at the end of the set of viable prefixes being described

ii) What possible symbols can immediately follow the production's right hand side (and hence what symbols can follow the viable prefix with the given production).

Each state of the characteristic automaton is the set of all items with the same viable prefix. When building a state, there must be a way to insure that all items, for a given state, are included. For example, if there is an item
in the state with the marked production \( A \to a \cdot Bb \) and \( B \to c \) is in \( P \), then there must be an item with the marked production \( B \to \cdot c \) for that state. The viable prefix, formed with the new marked production, will have the same prefix as the original item. The process of including all such items is called closing the state. However, in order to close a state, it is also necessary to describe how to propagate lookaheads to the added items. To do this, define the function \( \text{first}(a) \) as follows:

\[
\text{first}(a) = \{ a \mid a \Rightarrow a_c, a \in T \}
\]

Using the above definition, the closure of a set of items \( I \) (denoted as \( \text{closure}(I) \)) can be constructed using the rules:

i) Every item in \( I \) is also in \( \text{closure}(I) \)

ii) If the item \([A \to a \cdot Bb, LA]\) is in \( \text{closure}(I) \), and \( B \to c \) in \( P \), \( a \in LA \)

then the item \([B \to \cdot c, \text{first}(ba)]\) is in \( \text{closure}(I) \).

**Example 2.1** Let the CFG \( G \) have the set of productions:

\[
\begin{align*}
S & \to A \\
A & \to aA b \\
A & \to e
\end{align*}
\]

where \( S \to A \) is the start production. Then the closure of the item set \( \{[S \to \cdot A , (\$)]\} \) is the set \( \{[S \to \cdot A , (\$)], [A \to \cdot e , (\$)], [A \to \cdot aAb , (\$)]\} \).
The characteristic automaton G is built from the set of states constructed above with the transitions being grammar symbols. The path to a given state will then spell a legal prefix for some sentential form.

The algorithm (shown below) starts by setting the initial state to the closure of the start production, then taking each state just built, determines the transitions from the state as follows:

i) for each grammar symbol \( X \) in \((N \cup T)\) s.t. the item \([A \rightarrow a \cdot Xb, LA]\) is in the state, there is a unique transition, labeled \( X \), to the state containing the item \([A \rightarrow aX \cdot b, LA]\) obtained by shifting the dot across the grammar symbol \( X \).

ii) if \([A \rightarrow a \cdot , LA]\) is in the state, then no transition should be produced for that item.
Algorithm for constructing the characteristic automaton

**input:** a CFG \( G = (N, T, P, S) \)

**output:** a set \( C \), of states, and the function \( \text{GOTO} : \text{(set of items)} \times (N \cup T) \rightarrow \text{(set of items)} \), which defines the characteristic automaton.

**Method:** The two procedures below, initiated by calling \( \text{ITEMS}(G) \);

**procedure** \( \text{ITEMS}(G) \);

\begin{verbatim}
begin
  C := closure([S -> . S',{$}]);
  {where "$" is a unique symbol in T which denotes the end of the string to parse}
  repeat
    for each set of items \( I \) in \( C \), and each grammar symbol \( X \) such that \( J = \text{GOTO}(I,X) \) is not empty and \( J \notin C \)
    do add \( J \) to \( C \);
  until no more sets of items can be added to \( C \)
end;
\end{verbatim}
function GOTO(I, X);
begin
let J be the set of items
[A -> aX . b , LA] such that
[A -> a . Xb, LA] is in I;
return closure(J);
end;

Let the core of a state be the set of items in either of the two following forms:

i) [S -> . S' , {$}] 

ii) [A -> b . c , LA] where b ≠ e

It can be shown that by closing the core of a state, the original state can be retrieved. Hence, all examples in this paper will only show the core of each state.
example 2.2 Construction of a characteristic automaton

Let the CFG $G$ be defined by the same set of productions as in example 2.1. Then, the LR(1) characteristic automaton of the grammar $G$ is as follows:

where the transition arcs are defined by GOTO
II.4 Construction of LR(1) Parsers

Using the characteristic automaton, the LR(1) parser can be directly generated. Let an LR(1) parser be defined as a quintuple $M = (K, \text{action}, \text{goto}, G, \text{start})$ where

- $K$ is a finite set of parser states;
- $\text{action} : K \times T \rightarrow \{\text{shift } j \mid j \in K\} \cup \{\text{reduce } p \mid p \in P\} \cup \{\text{error}\}$ defines the parsing action table;
- $\text{goto} : K \times N \rightarrow K \cup \{\text{error}\}$ defines the parsing goto table;
- $G$ is a CFG such that $L(G)$ is the class of languages to recognize;
- and $\text{start}$ is the initial state.

The set of parser states $K$ contains a special state $\text{accept}$ which is the state $H$, such that $\text{action}(H,\$) = \text{reduce } S \rightarrow S'$. Also, the action and goto parsing tables are enough to define an LR(1) parser.

Using this definition, an LR(1) parser can be constructed using the following algorithm [A&U77,Gal79]:

17
Algorithm for constructing LR(1) parsing tables

input: The characteristic automaton $CG = (C, GOTO)$ for a CFG $G$;

output: a parsing table (possibly with conflicts if the grammar $G$ is not LR(1))

method: Let $C = \{I_1, I_2, \ldots, I_n\}$ be a set of sets of items from the characteristic automaton $CG$. The states of the parser will be labelled $1, 2, \ldots, n$ where state $i$ corresponds to the set of items $I_i$. State 1 is the initial state. The parsing actions are:

i) If $[A \rightarrow \_ \_ ab, LA] \in I_i$ where $a \in T$ and $GOTO(I_i, a) = I_j$; then $\text{action}(i, a) = \text{shift } j$

ii) If $[A \rightarrow \_ \_ , LA] \in I_i$, then for each $a \in LA$, set $\text{action}(i, a) = \text{reduce } A \rightarrow c$

iii) All entries of $\text{action}$ not defined by the above rules are set to $\text{error}$.
The goto transition for state \( i \) is constructed using the two rules:

i) if \( \text{GOTO}(I_i, A) = I_j \), where \( A \) is a nonterminal, then \( \text{goto}(i, A) = j \)

ii) All other entries of goto, not defined by the first rule, are set to error

**Example 2.3** Let the LR(1) characteristic automaton be defined as in example 2.2. Using the above algorithm, the two parsing tables produced are:
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shift 3</td>
<td>error</td>
<td>reduce A→e</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>error</td>
<td>error</td>
<td>reduce S→A</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>shift 4</td>
<td>reduce A→e</td>
<td>error</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>shift 4</td>
<td>reduce A→e</td>
<td>error</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>error</td>
<td>shift 7</td>
<td>error</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>error</td>
<td>shift 8</td>
<td>error</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>error</td>
<td>error</td>
<td>reduce A→aAb</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>error</td>
<td>reduce A→aAb</td>
<td>error</td>
<td></td>
</tr>
</tbody>
</table>
From the above algorithm, one can tell directly when a 
CFG \( G \) does not produce an LR(1) language. This occurs when 
action is not a function but only a relation, or in other 
words, whenever there is more than one possible action for 
some input pair. These multiple entries are known as 
conflicts. The two types of conflicts that can exist are i) 
shift/reduce and ii) reduce/reduce conflicts, which are 
respectively denoted as S/R and R/R.
Chapter III

Methods for reducing states in LR(1) parsers

LR(1) parsers have the nice property that they can be used for parsing most programming languages. Unfortunately, the parsers produced for these grammars, using the method described in the previous chapter, are too large to be considered useful. Hence, several modifications have been proposed which will reduce the size of the parser produced. This chapter discusses four of these methods. Two of the methods (SLR(1) and LALR(1)) reduce the number of states by reducing the size of the language accepted. The other two methods (proposed by Pager [Pag77a]) use conditions for merging states of a LR(1) parser while maintaining the full power to recognize LR(1) languages.
III.1 SLR(1) parsers

The SLR(1) parsing table construction is quite similar to that of the LR(1). The main difference is that the parser produced is based on a characteristic automaton with no lookahead (i.e. an LR(0) automaton). This simplification reduces, in general, the total number of states created.

To build an SLR(1) parser, redefine an item by removing the lookahead set leaving just the marked production. Under this definition, the rules to close a set of SLR items I become:

i) every item in I is also in closure(I);

ii) if the item \( A \rightarrow a \cdot Bc \) is in closure(I), and \( B \rightarrow b \in P \)
    then the item \( B \rightarrow \cdot b \) is also in closure(I);

The procedure to build the characteristic automaton are also simplified. These procedures are as follows:
function GOTO(I, X);
begin

let J be the set of items $A \rightarrow aX \cdot b$ such that

$A \rightarrow a \cdot Xb$ is in I and X is a grammar symbol;

return closure(I);
end;

procedure ITEMS(G);
begin

C := closure($S \rightarrow . S'$);

repeat

for each set of items I in C,

and each grammar symbol X such that

$J = GOTO(I, X)$ is not empty and $J \notin C$

do add J to C;

until no more sets of items can be added to C

end;
example 3.1 Let a CFG G be defined by the set of productions in example 2.1. Then an LR(0) characteristic automaton is:

![Diagram of LR(0) automaton]

The SLR(1) method does not use a lookahead set to decide what reduction to use once a viable prefix has been recognized. Instead, it uses a method to approximate the lookaheads, which in fact guarantees that the set of lookaheads will be included. This is done by the function $\text{FOLLOW} : N \rightarrow 2^T$ which computes all symbols which can follow a given nonterminal symbol. However, in order to compute FOLLOW, the terminal symbol $\$$ must be included. Hence for the definition of FOLLOW, it is assumed that there is an additional production of the form $S' \rightarrow S\$$ where $S'$ is a nonterminal and does not appear in any production in $P$. FOLLOW is defined as

25
\[
\text{FOLLOW}(X) = \{ \alpha \mid Y \Rightarrow \alpha X \beta, \text{ for all } Y \in \mathcal{N} \\
\quad \text{ where } \alpha = \text{first}(b) \}
\]

**Example 3.2** Using the CFG \( G \) described in example 2.1, the FOLLOW sets are:

\[
\begin{align*}
\text{FOLLOW}(S) &= \{ \$ \} \\
\text{FOLLOW}(A) &= \{ \$, b \}
\end{align*}
\]

Using the characteristic automaton and the function FOLLOW the SLR(1) parsing table can be created using the following algorithm:

**SLR(1) parsing table construction algorithm**

**input:** the SLR(1) characteristic automaton \( CG = (C, \text{GOTO}) \) for the CFG \( G \).

**output:** a parsing table (possibly with conflicts if not SLR(1)).

**method:** Let \( C = \{ I_1, \ldots, I_n \} \) be the set of sets of items from the characteristic automaton \( CG \). The states of the parser will be labeled \( 1, 2, \ldots, n \) where state \( i \) corresponds to the set of items \( I_i \). As with LR(1) parsers, let the initial state be state 1.
The parsing actions are defined as follows:

i) If $A \rightarrow a \cdot bc \in I_i$ where $b \in T$ and

$$\text{GOTO}(I_i, b) = I_j$$

then $\text{action}(i, a) = j$

ii) If $A \rightarrow a \cdot$ is in $I_i$ then for each $b \in \text{FOLLOW}(A)$

set $\text{action}(i, b) = \text{reduce } A \rightarrow a$

iii) all entries not defined by i) or ii) are set to $\text{error}$

The $\text{goto}$ transitions are defined by the following two rules:

i) If $\text{GOTO}(I_i, A) = I_j$

then $\text{goto}(i, A) = j$ where $A \in N$

ii) all other entries of $\text{goto}$, not defined by i), are set to $\text{error}$

**Example 3.3** Using the LR(0) characteristic automaton in example 3.1, and the FOLLOW sets in example 3.2, the SLR(1) parser is defined by the following tables:
**action**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shift 3</td>
<td>error</td>
<td>reduce A→e</td>
</tr>
<tr>
<td>2</td>
<td>error</td>
<td>error</td>
<td>reduce S→A</td>
</tr>
<tr>
<td>3</td>
<td>shift 3</td>
<td>reduce A→e</td>
<td>reduce A→e</td>
</tr>
<tr>
<td>4</td>
<td>error</td>
<td>shift 5</td>
<td>error</td>
</tr>
<tr>
<td>5</td>
<td>error</td>
<td>reduce A→aAb</td>
<td>reduce A→aAb</td>
</tr>
</tbody>
</table>

**goto**

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>error</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>3</td>
<td>error</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>5</td>
<td>error</td>
<td>error</td>
</tr>
</tbody>
</table>
III.2 LALR(1) parsers

A second type of simplification similar to the SLR(1) is the LALR(1) parser invented by DeRemmer [DeR69]. Many algorithms for computing LALR(1) parsers have since been presented [LLH71, AEH72, A&U77, DeR72, Alp76, Pag77b]. The main difference from SLR(1) is a concise and more accurate method for computing the set of lookaheads than the function FOLLOW. The same LR(0) characteristic automaton can be used to construct either an LALR(1) of an SLR(1) parser.

The definition of the LALR(1) lookahead function LA : state x P -> { t C T} is defined as follows:

\[ LA(k,A -> a) = \{ t C T \mid S S^* R bA C = R bac \} \]

and \( t = \text{first}(c) \) and the string \( ba \) is a prefix for the state \( k \)

**Example 3.4** Using the CFG \( g \), and the LR(0) characteristic automaton, from example 3.3, the function LA is defined as follows:

<table>
<thead>
<tr>
<th>( LA(1,S-&gt;A) )</th>
<th>( LA(1,A-&gt;aAb) )</th>
<th>( LA(1,A-&gt;e) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( { } )</td>
<td>( { } )</td>
<td>( { $ } )</td>
</tr>
<tr>
<td>( LA(2,S-&gt;A) )</td>
<td>( LA(2,A-&gt;aAb) )</td>
<td>( LA(2,A-&gt;e) )</td>
</tr>
<tr>
<td>( { } )</td>
<td>( { } )</td>
<td>( { } )</td>
</tr>
<tr>
<td>( LA(3,S-&gt;A) )</td>
<td>( LA(3,A-&gt;aAb) )</td>
<td>( LA(3,A-&gt;e) )</td>
</tr>
<tr>
<td>( { } )</td>
<td>( { } )</td>
<td>( { } )</td>
</tr>
<tr>
<td>( LA(4,S-&gt;A) )</td>
<td>( LA(4,A-&gt;aAb) )</td>
<td>( LA(4,A-&gt;e) )</td>
</tr>
<tr>
<td>( { } )</td>
<td>( { } )</td>
<td>( { } )</td>
</tr>
<tr>
<td>( LA(5,S-&gt;A) )</td>
<td>( LA(5,A-&gt;aAb) )</td>
<td>( LA(5,A-&gt;e) )</td>
</tr>
<tr>
<td>( { } )</td>
<td>( { $ , b } )</td>
<td>( { } )</td>
</tr>
</tbody>
</table>
The construction of the LALR(1) parser is exactly the same as an SLR(1) except that the action function is computed as follows:

i) If $A \rightarrow c \cdot ab$ $\notin I_i$ where $a \notin T$ and 
   
   \[
   \text{GOTO}(I_i, a) = I_j \\
   \text{then } \text{action}(i, a) = j
   \]

ii) If $A \rightarrow a$ is in $I_i$ then for each 
   
   \[
   a \notin \text{LA}(i, A \rightarrow A_\cdot) \text{ set } \text{action}(i, a) = \text{reduce } A \rightarrow a
   \]

iii) all entries not defined in i) and ii) are set to error

Example 3.5 Using the LR(0) characteristic automaton in example 3.1, and the function LA as defined in example 3.4, the LALR(1) parsing tables are:
The set of languages defined by $\text{SLR}(1)$, $\text{LALR}(1)$, and $\text{LR}(1)$, are known to form a hierarchy as follows:

$$\text{SLR}(1) \subset \text{LALR}(1) \subset \text{LR}(1)$$
In the previous two sections, restrictions on the class of languages were imposed to reduce the number of states in the LR(1) parser. Pager [Pag77a] shows that the number of states may be reduced without affecting the class of languages accepted.

The modification introduced by weak compatibility is in the construction of the LR(1) characteristic automaton (see section II.3). In the algorithm for constructing the automaton there is the statement:

\[
\text{for each set of items } I \text{ in } C, \text{ and each grammar symbol } X \text{ such that } \text{GOTO}(I,X) \text{ is not empty and } J \notin C \text{ do add } J \text{ to } C; \\
\]

In this statement if two states are similar in form, they can be represented by a single state, and therefore similar copies of a state can be removed. The criterion for deciding whether two states can be combined is called compatibility criterion and the action of combining two states called a \textit{merge}. For the LR(1) construction, two states are compatible if they are similar in form, that is, they contain the same set of items. Pager has found two other forms of compatibility which he calls weak and strong compatibility.
Unfortunately, changing the compatibility criterion from the LR(1) case can cause problems. In particular, when two states satisfy Pager's compatibility criteria, merging the states may necessitate a propagation of lookaheads to states already created, which in turn will modify the merged state which caused the original propagation. However, these problems can be resolved using the following algorithm:

**Algorithm for constructing an LR compatible characteristic automaton**

**input:** a CGF G and a compatibility function `compatible`.

**output:** a set C, of states, and the function 

\[ \text{GOTO} : (\text{set of items}) \times (N \cup T) \rightarrow (\text{set of items}) \], which defines the characteristic automaton.

**method:** the three procedures below, initiated by calling \( \text{ITEMS}'(G) \);
function GOTO(I,X);
begin
let J be the set of items
[A -> aX . b , LA] s.t.
[A -> a . Xb , LA] is in I;
return closure(J);
end;

procedure ITEMS'(G);
begin
C := closure([S -> . S' , {$}]);
repeat
for some set of items I in C,
and each grammar symbol X such that
J = GOTO(I,X) is not empty
do
if there exists a state K in C
such that compatable(K,J)
then insert(J,K,C)
else add J to C
fi
od
until no more sets can be added to C;
end;
procedure insert(S_1,S_2,C);
{merges S_1 into S_2 and updates C accordingly}
begin
S := merge(S_1,S_2);
if S_2 C S
then
    replace the items of state S_2 in C
    by the items of S;
    for each grammar symbol X
        such that GOTO(S_2,X) already defined
        do insert(closure(GOTO(S,X)),
            GOTO(S_2,X),C)
    od
fi
end;

Two states can be merged if and only if they have the
same set of marked productions in their respective item
part. Under this condition, the compatibility criterion is
that merging the states (and therefore the lookahead sets)
will not introduce any R/R conflicts in the resulting state
unless the language is in fact not LR(1). For weak
compatibility, the test is solely based on the two states
being merged, while strong compatibility also uses the set
of productions of the CFG associated with the LR(1) parsing
Let the function merge be defined as follows:

$$\text{merge}(S_1, S_2) = \{ [A \rightarrow a \cdot b , LA_1 \cup LA_2] \mid \]

- $[A \rightarrow a \cdot b , LA_1] \in S_1$
- $[A \rightarrow a \cdot b , LA_2] \in S_2$

and for all items $[A \rightarrow a \cdot b , LA_1] \in S_1$
there exists an item $[A \rightarrow a \cdot b , LA_2] \in S_2$ and
for all items $[A \rightarrow a \cdot b , LA_2] \in S_2$
there exists an item $[A \rightarrow a \cdot b , LA_1] \in S_1$

Then, according to Pager's definition, two states $S_1$ and $S_2$
are weakly compatible if

i) $S_1$ and $S_2$ only have common marked productions in
their item part. That is, if $[A \rightarrow a \cdot b , LA_1] \in S_1$
then there exists an item $[A \rightarrow a \cdot b , LA_2] \in S_2$ and
if item $[A \rightarrow a \cdot b , LA_2] \in S_2$ then there exists an
item $[A \rightarrow a \cdot b , LA_1] \in S_1$

ii) for each pair of items $[A \rightarrow a \cdot b , LA_1] \in S_1$ and
$[B \rightarrow c \cdot d , LA_2] \in S_2$, then at least one of the
following is true:

a) $LA_1 \cap LA_2 = \emptyset$

b) $LA_1 \cap LA_2 \neq \emptyset$ and there exists an item
$[B \rightarrow c \cdot d , LA_1'] \in S_1$ such that
$LA_1 \cap LA_1' \neq \emptyset$
c) $\mathcal{L}A_1 \cap \mathcal{L}A_2 \neq \emptyset$ and there exists an item

$[A \rightarrow a \cdot b \cdot \mathcal{L}A_2', \mathcal{S}_2]$ such that

$\mathcal{L}A_2 \cap \mathcal{L}A_2' \neq \emptyset$

Condition a) states that if there are no items between the states which have a common lookahead symbol, then the merge can not produce any conflicts, and in particular can not produce a R/R conflict. (Note: it is also impossible to introduce S/R conflicts since the states will be merged only if they have common marked productions. Therefore, the result of merging would only produce a S/R conflict if it existed in one of the unmerged states before merging). In condition b) and c) the set of conditions is:

$[A \rightarrow a \cdot b \cdot \mathcal{L}A_1, [B \rightarrow c \cdot d \cdot \mathcal{L}A_1'] \in \mathcal{S}_1$

$[A \rightarrow a \cdot b \cdot \mathcal{L}A_2, [B \rightarrow c \cdot d \cdot \mathcal{L}A_2'] \in \mathcal{S}_2$

$\mathcal{L}A_1 \cap \mathcal{L}A_2 \neq \emptyset$ and either $\mathcal{L}A_1 \cap \mathcal{L}A_1' \neq \emptyset$ or

$\mathcal{L}A_2 \cap \mathcal{L}A_2' \neq \emptyset$

Since $\mathcal{L}A_1 \cap \mathcal{L}A_2 \neq \emptyset$, the only possible conflict is a R/R conflict arising from merging the lookahead on the productions $A \rightarrow ab$ and $B \rightarrow cd$. However, this can occur only if $b \xrightarrow{+} R w$ and $d \xrightarrow{+} R w$, producing a common substate where both productions will be reducible. By condition b)

$\mathcal{L}A_1 \cap \mathcal{L}A_1' \neq \emptyset$, if in addition $b \xrightarrow{+} R w$ and $d \xrightarrow{+} R w, w \in T^*$, then there must already exist a state with a R/R conflict on some symbol $a \in \mathcal{L}A_1 \cap \mathcal{L}A_1'$. Similarly for condition c).

Hence, if the language is indeed LR(1), then it must be the
case that $b \rightarrow_R w; \ d \rightarrow_R w'; \ v, w' \in T^*; \ and \ w \neq w'$, and therefore conditions a), b) and c) are sufficient to insure no conflicts will be produced if the language generated by the grammar is indeed LR(1).

For example, let a CFG be defined with the set of productions in figure 3.1. The LR(1) characteristic automaton contains 38 states (shown in part in figure 3.2). Under weak compatibility, states 8 and 12 can not be merged since the items $[X \rightarrow a. AE, \{d\}] \in 12$ and $[Y \rightarrow a. B, \{d\}] \in 8$ have the common lookahead symbol $d$. However, for example, states 30 and 33 are in fact weakly compatible.

It can be shown that the size of a weak compatible LR(1) parsing table will contain a number of states that is somewhere between that of LALR(1) and LR(1) parsing tables.

$S \rightarrow S'$
$S' \rightarrow aXb$
$S' \rightarrow aYa$
$S' \rightarrow bXd$
$S' \rightarrow bYa$
$S' \rightarrow aAE$
$Y \rightarrow aB$
$X \rightarrow aAE$
$Z \rightarrow aC$
$A \rightarrow aDF$
$Z \rightarrow aC$
$A \rightarrow aDF$
$B \rightarrow b$
$C \rightarrow aDF$
$D \rightarrow d$
$E \rightarrow e$

$F \rightarrow e$

**Figure 3.1**
figure 3.2
III.4 Strong compatibility

Pager's strong compatibility adds one condition to weak compatibility which guarantees the production of a LALR(1) parser if the language generated by the grammar is LALR(1). Otherwise it will produce an LR(1) parsing table with the number of states greater than the number of states produced by the LALR(1) method but less than the number produced by the LR(1) method.

Strong compatibility requires that no two states be merged if they have a common descendant in the LR(1) characteristic automaton which will introduce R/R conflicts when the two states are merged.

For example, the grammar presented by figure 3.1 creates (in part) the LR(1) characteristic automaton in figure 3.2. States 8 and 12 are not weakly compatible because the items \([X \rightarrow a.AE, \{d\}] \in 12\) and \([Y \rightarrow a.B, \{d\}] \in 8\) have a common lookahead symbol "d". If these two states are merged (and hence causing merges of states (20,28), (18,26), (17,25), (16,24), (29,32), (31,34), (30,33), (19,27), (36,38) and (35,37) where each pair are common descendants) the resulting states of the automaton would have no conflicts. Hence these two states, according to Pager's definition, are in fact strongly compatible.
On the other hand, let the grammar be that of figure 3.3 which creates (in part) the LR(1) characteristic automaton in figure 3.4. Merging states 7 and 10 (and hence causing common descendants 14 and 18 to be merged) would result in two R/R conflicts on the symbols "a" and "b" in its descendant state. Hence these states will not be merged under strong compatibility.

\[
\begin{align*}
S & \rightarrow S' \\
S' & \rightarrow aXd \\
S' & \rightarrow aYa \\
S' & \rightarrow bYb \\
Y & \rightarrow ab \\
B & \rightarrow b \\
X & \rightarrow aB \\
S' & \rightarrow bXa 
\end{align*}
\]

\textbf{figure 3.3}
The way in which two items (from different states) can produce a common state with a R/R conflict is if two items can derive the same substring. That is, if the two states $S_1$ and $S_2$ are to be merged such that there exists two items $[A\rightarrow ab, LA_1]$ $\in S_1$ and $[B\rightarrow ab, LA_2]$ $\in S_2$ where $f \in LA_1 \cap LA_2; \text{ b } \xrightarrow{R} w \text{ and } d \xrightarrow{R} w$, then the two states have common descendants such that a merge will introduce R/R conflicts.
For example, the reason that states 7 and 10 (in figure 3.4) could not be merged is that the items \([X \rightarrow a.B, \{d\}] \in 7\) and \([Y \rightarrow a.b, \{d\}] \in 10\) have a common lookahead symbol\(d\), and the strings \(B\) and \(b\) both rewrite to the string \(b\).

The search for a common substring between two states, when necessary to try all possible combinations of rewrites, involves as much work as building all descendant states. However, it is not necessary to expand all possible combinations of rewrite rules. This fact can be seen by understanding how expansion of the nonterminals is performed in building the characteristic automaton. That is, when the item \([A \rightarrow a \cdot Xb \cdot LA]\) is closed, where \(X \rightarrow c \in P\) and \(d \in LA\), it will create the item \([X \rightarrow \cdot c \cdot , \text{first}(bd)]\). If \(b \not\rightarrow_R e\), it is clear that the elements in the lookahead set \(LA\) will be propagated to the new item. On the other hand, if \(b \rightarrow_R e\), the definition of the function \(\text{first}\) indicates that any element \(d \in LA\) is not in \(\text{first}(bd)\). Hence, in this case, the lookaheads defined by \(\text{first}(bd)\) are independent of \(LA\) and does not effect states derived from the new item. Stated differently, the only rewrites that should be performed are those which are applied to the nonterminals which occur at the end of marked productions. This restriction on the number of possible derivations to look at, is what Pager calls a strong rightmost derivation (denoted \(\Rightarrow_{SR}\)) and is defined as:
aBc \rightarrow_{SR} abc \text{ iff}

1) c = e

ii) aBc \rightarrow_R abc

Pager has derived a procedure [Pag77a] which checks if two items, having a common lookahead symbol, will produce a shared descendant containing a R/R conflict. The author feels that the algorithm presented by Pager is opaque, as well as slightly incorrect, and that the algorithm in this paper (see page 49) has been corrected and modified to clarify its nature.

The algorithm is presented using two co-recursive procedures which tries all possible strong rightmost derivations to see if the two given marked productions yield a common descendant state where two different productions will be reduced (since this is the only way that an R/R conflict can be produced). The procedure CHECK looks for trivial cases (i.e., cases where no rewrites are necessary to determine the result) while the procedure nontrivialcheck checks those cases requiring rewrites in order to determine the wanted criteria.

One possibility that procedure CHECK handles is if it is impossible for two items, with or without rewrites, to produce a common descendant. That is, let (1) A \rightarrow a.bXf and (2) B \rightarrow c.bYg be two marked productions where
Assume that these two marked productions can derive a common substring which will produce a \( R/R \) conflict. Then it must be the case that \( Xf \overset{*}{\Rightarrow} R w \) and \( Yg \overset{*}{\Rightarrow} R w \). Since both \( f \) and \( g \) do not derive \( e \), the lookaheads can not propagate through \( f \) and \( g \). But then, by the way LR(1) parsers are generated, the string derived from \( X \) will be reduced to \( X \) before scanning the string derived from \( f \). Hence any string derived from \( Xf \) must be of the form \( Xs \). Similarly, any string derived from \( Yg \) must be of the form \( Yt \). Therefore, since \( X \not\sim Y \), it is impossible for any items of this form to produce a common substring (and hence a common descendant) which will produce \( R/R \) conflicts.

The second trivial check in the procedure \( \text{CHECK} \), is if the two marked productions immediately indicate a common descendant which will produce \( R/R \) conflicts if merged. That is, if the two items are of the form (1) \( A \rightarrow a \cdot bXWf \) and (2) \( B \rightarrow a \cdot bXZg \) where

i) \( f, g \overset{*}{\Rightarrow} R e \)

ii) \( X \in (N \cup T) \) and \( X \not\Rightarrow R e \)

iii) \( W, Z \in N \) and \( W, Z \overset{*}{\Rightarrow} R e \)

It is clear, under the above conditions, that the closure of the items (3) \( [A \rightarrow abX \cdot Wf \cdot LA_1] \) and (4)
[\(B \rightarrow abX, Zg, LA_2\)] will produce the items (5)

\([W \rightarrow e, Q]\) and (6) \([Z \rightarrow e, Q]\) where \(Q = LA_1 \cap LA_2\).

Hence this case will produce a common descendant where conflicts will be produced.

In all other cases, some rewriting is necessary and procedure nontrivialcheck is called to handle these cases.

One possibility, that requires rewriting, is when the two marked productions are of the form (1) \(A \rightarrow a.bXf\) and (2) \(B \rightarrow c.bYg\) where

i) \(X \in N\) and \(X \rightarrow^* e\)

ii) \(f \rightarrow^* e\)

iii) \(Y \in (N U T)\); \(Yg \not\rightarrow^* e\) and \(Y \neq X\)

In this case, \(X\) must rewrite to some string derivable from \(Yg\) in order to produce a common string (and hence a common descendant). However, this the same as testing if there exists a production \(X \rightarrow h\) where \(h \not= e\) such that the items \(X \rightarrow^* h\) and \(B \rightarrow cb.Yg\) will share a common descendant which can produce R/R conflicts.

A second possibility handled in nontrivialcheck are items of the form (1) \(A \rightarrow a.bXf\) and (2) \(B \rightarrow c.bZg\) where
i) $X \in N$

ii) $Z \in (N \cup U T)$; $Zg \not \Rightarrow^*_R e$ and $X \neq Z$

iii) $f \Rightarrow^*_R e$

iv) no production $X \Rightarrow h$, where $h \neq e$, exists such that $X \Rightarrow h$ and $B \Rightarrow cb.Zg$ will have a common descendant

In this case, because of condition iv) and that $X \neq Z$, any common string derivable from $Xf$ must be of the form $Xg$ while any common string derivable from $Zg$ must be of the form $Zt$. But this implies that they can not derive the same string and hence can not have a shared descendant.

The last possibility checked by the procedure nontrivial check is the case when the marked productions are of the form (1) $A \Rightarrow a.bX$ and (2) $B \Rightarrow c.bY$ where $X, Y \in N$ and $X \neq Y$. The only way that these two marked productions can derive a common descendant is if $X \Rightarrow^*_R w$ and $Y \Rightarrow^*_R w$. However, this is the same as testing if there exists two productions of the form $X \Rightarrow s$ and $Y \Rightarrow t$ such that either the marked productions $A \Rightarrow ab.X$ and $Y \Rightarrow t$, or $X \Rightarrow s$ and $B \Rightarrow cb.Y$, will produce a common descendant which can contain an R/R conflict from merging.

For efficiency, the procedure nontrivial check uses a special global function

\text{tried} : N \times (\text{marked productions}) \rightarrow \text{boolean}.

Before the top call to procedure CHECK is made, the function is set to false for all possible inputs, and it will return
false the first time it is called with any given input. After that, anytime the function is again called with the same set of arguments, it will return true. Therefore, this function will prevent the procedure nontrivialcheck from checking if a nonterminal will rewrite to match some particular marked item.

Finally, it is assumed that on the top level call of CHECK(A → a · a', B → b · b') the following two conditions hold:

1) A → a · a' ≠ B → b · b'
2) aa' ≠ e and bb' ≠ e
Co-recursive procedures to check
for a shared descendant

procedure check(A -> a * a_1 a_2 ... a_n,
                B -> b * b_1 b_2 ... b_m): boolean;
(note: a_i, b_i \in \{N \cup T\}; A,B \in N; a,b \in \{N \cup T\} *)
begin
  s:= maximum i s.t.\ a_i a_{i+1} ... a_n \not\in \cdot R \cdot \in ;
  t:= maximum i s.t.\ b_i b_{i+1} ... b_m \not\in \cdot R \cdot \in ;
  match:= maximal i s.t.\ a_i = b_i;
  if match+1 < min(s,t)
    then check:=false
  else if match > max(s,t)
    then check:=true
  else
    if s > t
      then check:=nontrivialcheck(
                      B -> b * b_1 b_2 ... b_m,t
                      A -> a * a_1 a_2 ... a_n,s,match)
    else check:=nontrivialcheck(
                      A -> a * a_1 a_2 ... a_n,s
                      B -> b * b_1 b_2 ... b_m,t,match)
end;
procedure nontrivialcheck(A → a · a₁a₂…aₙ,s,
                 B → b · b₁b₂…bₘ,t,
                 match) : boolean;

{note: s ≤ t}

begin

    terminate:=false;

    repeat

        if (match -(s-1)) < 0) or (s=t)

            then

                nontrivialcheck:=false;  terminate:=true;

            else if (aₛ G N) or

                not tried(aₛ,B → bba,…bₙ₋₁ · bₛ…bₘ)

                then

                    for each production C → c G P

                        s.t. aₛ =C, c ≠ e, and

                        C → c ≠

                        B → bba,…bₙ₋₁ · bₛ…bₘ

                    do

                        if check(C → c,

                            B → bba,…bₙ₋₁ · bₛ…bₙ)

                            then

                                nontrivialcheck:=true;

                                terminate:=true

                            fi

                    od

    fi

end
else if (s=t) and (match-1=s) and \( b_t \in N \)

\[ \text{and check}(B \rightarrow b_b \cdots b_{s-1} \cdot b_s \cdots b_n', \]
\[ A \rightarrow a_{a_1} \cdots a_{s-1} \cdot a_s \cdots a_n) \]

then

nontrivialcheck:=true; terminate:=true

fi;

s:=s+1;

until terminate;

end;

Using the above, two states \( S_1 \) and \( S_2 \) are strong compatible if

1) If the item \([A \rightarrow a \cdot b, LA_1] \in S_1\) then there exists an item \([A \rightarrow a \cdot b, LA_2] \in S_2\) and if the item \([A \rightarrow a \cdot b, LA_2] \in S_2\) then there exists an item \([A \rightarrow a \cdot b, LA_1] \in S_1\)

ii) for each quadruple of items

\([A \rightarrow a \cdot b, LA_1], [B \rightarrow c \cdot d, LA_1'] \in S_1,\]
\([A \rightarrow a \cdot b, LA_2], [B \rightarrow c \cdot d, LA_2'] \in S_2\)

either

a) weak compatibility between the items hold or

b) \( b \) and \( d \) do not share a descendant.
Chapter IV

An Error Recovery Method for LR Parsers

In the previous two chapters, five different constructions were discussed, all of which produce LR parsers. The downfall of all LR parsers is that they are designed only to decide if the given input is legal, that is, belongs to the language generated by its grammar. This causes the unfortunate result that when such a parser is used in a compiler, once the first illegal terminal symbol is found, the parse stops with failure. However, it would be more desirable to have the parse report as many additional errors as possible.

Several people have proposed various error recovery schemes for LR parsers [G&R75, D&R77, P&D79, O’H76, Pen77, P&D78]. This chapter will only deal with one such method, which is a modification of the one presented by DeRemmer and Pennello [P&D79]. The
algorithm presented here differs from theirs in that it is incorporated into the LR parser and does not attempt error correction.

In order to describe error recovery, we first describe how an LR parser works. Let a path be a sequence of states \( q_0 q_1 \ldots q_n \) such that for each state \( q_i \), one of the following conditions hold:

i) \( \text{goto}(q_i, X) = q_{i+1} \) for some \( X \in N \)

ii) \( \text{action}(q_i, a) = q_{i+1} \) for some \( a \in \Sigma \).

A path will be denoted as \([q_0: a]\). That is, if \( a = a_1 a_2 \ldots a_n \) where \( a_i \in (N \cup \Sigma) \) then the path \([q_0: a]\) is the sequence of states such that either \( \text{action}(q_{i-1}, a_i) = q_i \) or \( \text{goto}(q_{i-1}, a_i) = q_i \). Also, let the result of the function \( \text{top}: \text{path} \rightarrow \text{state} \) be defined as the state \( q_n \) where the path is \( q_0 q_1 \ldots q_n \). Finally, whenever the path \([q:a]\) begins from the start state (of the LR parser) it will simply be denoted as \([a]\).

The basic control of a LR parser can be defined by the decision function \( \text{df}: \text{path} \times \Sigma \rightarrow (\text{path} \cup \{\text{reject, accept}\}) \) as follows:

1) \( \text{df}([a], b) = [ab] \) if \( \text{action}(\text{top}([a]), b) = \text{shift} \) j for some state \( j \in \Sigma \).
ii) $df([aw], b) = df([aA], b)$
   if $\text{action(top([aw]), b)} = \text{reduce } A \rightarrow w$, and
   $aw \neq S$ when $b = \$$

iii) $df([S], \$$) = \text{accept}$
     if $\text{action(top([S]), \$$)} = \text{reduce } S \rightarrow S'$

iv) $df$ is defined as $\text{reject}$ for all pairs
    $([a], b)$ not defined by rules i) through iii)

The algorithm to implement the above decision function
is simply as follows:

```plaintext
procedure parse(df, input);
begin
   path:=[start, e];
   repeat
      t:=next terminal symbol from input;
      path:=df(path, t);
      until (path = accept) or (path = reject);
   print path;
end;
```
Note that the variable path is implicitly used as a stack which holds the prefix of sentential forms being recognized by the parser.

The error recovery strategy describes what to do if the parse of an input results in reject. As can be seen from the previous algorithm, LR parsers have the nice property that they stop reading input immediately after the input string is found to be illegal. The best recovery from such an error would be if the parse could somehow be restarted such that all other errors made in the input could be picked up. Unfortunately, this strategy is really unfeasible since it carries the implicit assumption of knowing what the writer meant when he wrote the string to be parsed.

A much more conservative approach is to only state what remaining substrings of the input are impossible according to the given grammar. That is, if the remaining input after the error is a string \( w \in T^* \) and there doesn't exist a rightmost derivation such that \( S \Rightarrow_R awc \) for some \( a \in (N \cup T)^* \) and \( c \in T^* \), then the substring \( w \) should be reported as an error.
For example, consider the two pseudo PASCAL productions

\[
\text{<stmt> } \rightarrow \text{ FOR } <\text{var}> := <\text{exp}> \text{ TO } <\text{exp}> \text{ DO } <\text{stmt}>
\]

\[
\text{<stmt> } \rightarrow \text{ WHILE } <\text{exp}> \text{ DO } <\text{stmt}>
\]

with the erroneous input

\[
\text{FOR X:=1 5 DO BEGIN J:=X; L:=X END;}
\]

where the terminal symbol "TO" has accidentally been left out. Using an LR parse, parsing would stop after reading the symbol "5". As one looks for subsequent errors, it is clear that "5" is a valid substring derivable from \( S \). It is also clear that 5 can occur at the following points in the given productions

\[
\text{<stmt> } \rightarrow \text{ FOR } <\text{var}> := "<\text{exp}>" \text{ TO } <\text{exp}> \text{ DO } <\text{stmt}>
\]

\[
\text{<stmt> } \rightarrow \text{ FOR } <\text{var}> := <\text{exp}> \text{ TO } "<\text{exp}>" \text{ DO } <\text{stmt}>
\]

\[
\text{<stmt> } \rightarrow \text{ WHILE } "<\text{exp}>" \text{ DO } <\text{stmt}>
\]

By expanding the substring to include the next input symbol, the next possible substring to test would be "5 DO". Here, the number of possible positions of this string has been reduced to

\[
\text{<stmt> } \rightarrow \text{ FOR } <\text{var}> := <\text{exp}> \text{ TO } "<\text{exp}>" \text{ DO } <\text{stmt}>
\]

\[
\text{<stmt> } \rightarrow \text{ WHILE } "<\text{exp}>" \text{ DO } <\text{stmt}>
\]

Continuing this process, it is clear that the substring "5 DO BEGIN J:=X; L:=X END" can correspond to the following positions in the productions:

\[
\text{<stmt> } \rightarrow \text{ FOR } <\text{var}> := <\text{exp}> \text{ TO } "<\text{exp}>" \text{ DO } <\text{stmt}>
\]

\[
\text{<stmt> } \rightarrow \text{ WHILE } "<\text{exp}>" \text{ DO } <\text{stmt}>
\]
At this point, the semicolon at the end of the parse string implies that a reduction should be performed by one of the above productions. One possibility is to take the string recognized before the reject point, and to either add or delete symbols to produce a match and therefore decide which reduction to choose. This type of error recovery is in fact the error correction method used by [P&D79]. However, the one chosen by the author assumes that the substring "5 DO BEGIN J:=X; L:=X END" is the maximal deterministic string that could be recognized, and hence remove it from further consideration. That is, it will restart the parse starting with the semicolon.

The above example in fact characterizes the error recovery method described in this chapter. To state the method more explicitly, let me start by defining an error state as a set of LR parser states, where each error state contains the set of LR parser states that the parse might be in. The restart state as a special error state containing all the LR parser states.

The first shift, in error recovery, is a forced shift through the illegal terminal symbol that produced the rejection. This shift can be viewed as a parallel shift, on the error symbol a, from all LR parser states I in the restart state to all states J such that action(I,a) = J. It will then try to parse the input where the parse will start,
simultaneously, from each of the LR parse states $J$ existing after the forced shift through the illegal symbol. If along the way, any of these parses produce an error, it will be dropped from further consideration for simultaneous parsing.

One possible result of the above process is that all parses will be dropped from the set of simultaneous parses. Under this condition, it is clear that there is no derivation such that $S \Rightarrow^*_{R} awc$ for the parsed input $w$. Hence, it is quite legal to assume that the next input symbol input can not occur, and report it as an error. Since this is an error, the algorithm will then restart the recovery method on the next input symbol. Note that the first action on any error is a forced shift. This is done to guarantee that the remaining input is parsed. Also, error recovery should not continue if the illegal terminal symbol was the end of string marker $\$$. 

The second problem is that if the above error recovery process is to be merged into the LR parser, the parallel parses have to be made deterministic. There is no problem with the action function for a set of states, if the result for all possible inputs is a shift entry. In this case it is clear that the action is deterministic, since resulting states can be lumped into a new set of states and hence creating a new error state. The same is true for the goto function. Therefore, nondeterminism can only occur if the
action, for a set of states to be simultaneously parsed, contain either

1) shifts and reductions for the same input symbol
2) reductions for different productions for the same input symbol (as shown in the previous example)

Unfortunately, neither of these cases seem to be resolvable deterministically. If, in either case, the parse was allowed to continue and the next action was performed, the result would produce two different paths. That is, the above two conditions would result in disjoint sentential prefixes. Such conditions will be called overdefined.

However, some decision still has to be made so that the remaining input can be parsed. Again, the conservative approach was taken. Whenever the input string being parsed becomes overdefined, the parser assumes that it is the maximal substring it can recognize, and restarts the whole error recovery process on the next input symbol.

By merging the error-recovery into the LR parser, a new LR parser with error recovery can be built. If an LR parsing table is the tuple $M = (K, \text{action}, \text{goto}, G, \text{start})$, then let the same parser with error recovery be defined as the tuple $M' = (K, K', \text{action}, \text{goto}, G, \text{start}, \text{init-error})$ where
K, G, and \texttt{start} are defined as in M,

\( K' \) is a set of new states called error recovery states

\texttt{init-error} is a state in \( K' \) denoted as the restart state of the error recovery method

\texttt{goto} : \((K \cup K') \times N \rightarrow K \cup K' \cup \{\text{error}\}

\texttt{action} : \((K \cup K') \times T \rightarrow\
\{\text{shift} \ k \mid k \in K\} \cup \{\text{error, overdefined}\} \cup \\
\{\text{reduce} \ p \mid p \in P\}

Furthermore, the \texttt{init-error} state will be so defined that for each \( b \in T \), \( \texttt{action}(\texttt{init-error}, b) = \texttt{shift} \ j \) for some state \( j \). Each recovery state is a set of parsing states in \( K \), such that it is the set of states that can occur simultaneously for the input string being parsed.

Using the above definition, LR parsers with error recovery can be built by the following algorithm:

\textbf{Construction of LR parser with error recovery}

\textit{input:} LR parsing table \( M = (K, \texttt{action}, \texttt{goto}, G, \texttt{start}) \)

\textit{output:} LR parsing table \( M' = (K, K', \texttt{action}, \texttt{goto}, G, \texttt{start}, \texttt{init-error}) \)
method:

begin

{initialize state init-error}

set $K'$ to the single state containing the set $\{s \in K\}$
and label it as init-error;

for each $a \in T$ do

let $s$ be the set

$\{j \in K \mid \text{action}(i,a) = \text{shift } j$

for all $i \in \text{init-error}\}$;

if $s$ is a singleton

then set $s'$ to the element of $s$

else if $s \in K'$

then set $s'$ to that state in $K'$

else add $s$ to $K'$ and label the new state as $s'$;

fi

set action(init-error,a) = $s'$

od
for each $X \in N$ do

let $s$ be the set

\{ $j \in K \mid \text{goto}(t,X) = j$ for all $t \in \text{init-error}$ \}\n
if $s$ is empty

then set $\text{goto}(\text{init-error},X) = \text{error}$

else

if $s$ is a singleton

then set $s'$ to that element of $s$

else if $s \in K'$

then set $s'$ to the state in $K'$ containing $s$

else add $s$ to $K'$, and set $s'$ to its label

fi

set $\text{goto}(\text{init-error},X) = s'$;

fi

od

{build each general error state}

repeat

for each state $i \in K'$ such that the parsing table for that state is still undefined do

for each $a \in T$ do
if there exists two states \( S_1, S_2 \in i \) s.t.

\[ [A \rightarrow a \cdot , LA_1] \in S_1 \text{ where } a \notin LA_1 \]
\[ [B \rightarrow c \cdot d , LA_2] \in S_2 \]

where \( \text{first}(d) = a \)

then set \( \text{action}(i, a) = \text{overdefined} \)

else if there exists two states

\( S_1, S_2 \in i \) s.t.

\[ [A \rightarrow a \cdot , LA_1] \in S_1 \]
\[ [B \rightarrow b \cdot , LA_2] \in S_2 \]

where \( a \notin LA_1 \) \& \( \text{LA}_2 \) and \( A \rightarrow a \neq B \rightarrow b \)

then set \( \text{action}(i, a) = \text{overdefined} \)

else if there exists a state \( s \in i \) s.t.

\[ [A \rightarrow w \cdot , LA] \in s \text{ where } a \notin LA \]

then set \( \text{action}(i, a) = \text{reduce } A \rightarrow w \)

else

let \( s \) be the set

\( \{ j \in K \mid \text{action}(t, a) = \text{shift } j \} \)

for all \( t \in i \);
if s is empty

then set action(i,a) = error

else

if s is a singleton

then set s' to the element in s

else if s ∈ K'

then set s' to the state in K'

containing s

else add s to K', setting s' as the label of the added state;

fi

set action(i,a) = shift s'

fi

fi

od

for each X ∈ N do

let s be the set \{ j ∈ K | goto(t,X) = j \}

for all t ∈ I);

if s is empty

then set goto(i,X) = error
else
    if s is a singleton
        then set s' to the element in s
    else if s ∈ K'
        then set s' to the state in K'
           containing s
    else add s to K', and set s' to its label
fi;
set goto(i,X) = s'
fi
od
od
until no more states can be added to K'
end

Using the resulting LR parser with error recovery, the basic control can be handled using the decision function \( df' \) : path × T → path as follows:

1) \( df'([q:a],b) = [q:ab] \)
   when \( \text{action}(\text{top}([q:a]),b) = \text{shift} j \) for some \( j \in (K \cup K') \)
ii) \( df'([q:\text{aw}], b) = df'([q:\text{aA}], b) \)
when \( \text{action}(\text{top}([q:\text{aw}]), b) = \text{reduce} \ A \rightarrow \text{w} \), and if 
\( \text{aw} = S \) then \( b \neq \$ \)

iii) \( df'([\text{init-error}:\text{w}], b) = df'([\text{init-error}:\text{A}], b) \)
when \( \text{action}(\text{top}([\text{init-error}:\text{w}]), b) \)
\( = \text{reduce} \ A \rightarrow \text{aw} \),
where \( a \neq e \) and \( b \neq \$ \)

iv) \( df'([S], \$) = \text{accept} \)

v) \( df'([\text{init-error}:\text{S}], \$) = \text{Reject} \)
if \( \text{action}(\text{top}([\text{init-error}:\text{S}]), \$) = \text{accept} \) or 
\( \text{overdefined} \)

vi) \( df'([q:\text{a}], \$) = \text{reject} \)
when \( \text{action}(\text{top}([q:\text{a}]), \$) = \text{error} \)

vii) \( df'([\text{init-error}:\text{a}], b) = [\text{init-error}, b] \)
where \( b \neq \$, \) and 
\( \text{action}(\text{top}([\text{init-error}, \text{a}]), b) = \text{overdefined} \)

viii) \( df'([q:\text{a}], b) = [\text{init-error}, b] \)
where \( b \neq \$ \) and \( \text{action}(\text{top}([q:\text{a}]), b) = \text{error} \)
Note that cases vi) or viii) represent that an error has been found in the string being parsed. Hence, any error messages produced are produced at these points.

Finally, an LR parser with error recovery can be implemented simply by calling the procedure parse, using df' as the decision function.
Chapter V

Implementation

This chapter discusses two programs. The first program creates an SLR(1) parser, with error recovery. The second program creates either an LR(1), LALR(1), weakly compatible or a strongly compatible LR parser. The first section discusses the representation of the parsing tables built by both programs. The second section describes the implementation of the SLR(1) parser constructor and how that system is used while the third section does the same for the second parser constructor.
V.1 Representation of the parsing tables

The representation of the parsing tables naturally suggest using arrays. For uniformity of both access and values held in the arrays, all terminal symbols, nonterminal symbols, and productions are provided with an internal code of integers by both programs. For terminal symbols, the codes are defined by the set

\[ \{ i | 0 \leq i < n \text{ where } n \text{ is the number of distinct terminal symbols occurring in the productions} \} \]

where 0 is reserved for the special terminal symbol $$. Nonterminal symbols are encoded using the set

\[ \{ i | -m \leq i < -1 \text{ where } m \text{ is the number of distinct nonterminals occurring in the productions} \} \]

where the start symbol $S$ will always be given the code -1. Productions are coded using the set

\[ \{ i | 1 \leq i < p \text{ where } p \text{ is the number of productions in the grammar} \} \]

where the production $S \rightarrow S'$ is always given the code 1.

In representing the action and goto functions, only non-error values are kept internally since the vast majority of the function values are in fact error. The remaining values are saved in groups, one for for each state, where states having the same set of non-error values will be represented by a single copy of the groups.
For example, the grammar

\[
\begin{align*}
S & \rightarrow E \\
E & \rightarrow E \ast T \\
E & \rightarrow T \\
T & \rightarrow T + F \\
T & \rightarrow F \\
F & \rightarrow \text{id} \\
F & \rightarrow (E) \\
\end{align*}
\]

would produce the following SLR(1) parsing tables:
### Action table

<table>
<thead>
<tr>
<th></th>
<th>$</th>
<th>*</th>
<th>+</th>
<th>id</th>
<th>(</th>
<th>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>S 3</td>
<td>S 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>S 3</td>
<td>S 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>F-&gt;id</td>
<td>F-&gt;id</td>
<td>F-&gt;id</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>T-&gt;F</td>
<td>T-&gt;F</td>
<td>T-&gt;F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>E-&gt;T</td>
<td>E-&gt;T</td>
<td>S 8</td>
<td>E-&gt;T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>S-&gt;E</td>
<td>S 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>S 9</td>
<td>S 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>S 3</td>
<td>S 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>S 3</td>
<td>S 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>F-&gt;(E)</td>
<td>F-&gt;(E)</td>
<td>F-&gt;(E)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>T-&gt;T+F</td>
<td>T-&gt;T+F</td>
<td>T-&gt;T+F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>E-&gt;E*T</td>
<td>E-&gt;E*T</td>
<td>S 8</td>
<td>E-&gt;E*T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>S 9</td>
<td>S 8</td>
<td>S 3</td>
<td>S 2</td>
<td>S 10</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S 3</td>
<td>S 2</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>S-&gt;E</td>
<td>S 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>S 8</td>
<td>S 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

where **shift j** is represented by $S_j$, **reduce p** is represented by $p$, **overdefined** is represented by $O$, and **error** is omitted.
### goto table

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>E</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>7</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

where goto(i,X) = error has been omitted

By elimination of the error values, 58.8% of the above tables does not need to be saved. Also, states 1, 2, 8, and 9 in the previous action table all have the same set of...
values and therefore will be represented by only one group of values.

Each non-error value of the action table will be represented as follows:

i) \(\text{action}(i, a) = \text{shift} \) \(j\) will be represented by the pair \((x, j)\) where \(x\) is the code of the terminal symbol \(a\).

ii) \(\text{action}(i, a) = \text{reduce} \) \(A \rightarrow w\) will be represented by the pair \((x, -p)\) where \(x\) is the code of the terminal symbol \(a\) and \(p\) is the code of the production \(A \rightarrow w\).

iii) \(\text{action}(i, a) = \text{overdefined}\) will be represented by the pair \((x, 0)\) where \(x\) is the code of the terminal symbol \(a\).

The non-error values of the goto table, for some state \(i\), will be represented as the pair \((x, j)\) where \(\text{goto}(i, A) = j\) and \(x\) is the code of the nonterminal \(A\).

For efficiency in retrieving the values from the action and goto tables the integer pairs corresponding to each state are sorted using the relation \(<\) where
(a,b) ≤' (c,d) iff either a < c, or a = c and b < d.

Four integer arrays are used to represent the values of the two parsing tables. The array `parsetable` is a n x 2 array, for some n, which holds all of the non-error values of the two parsing tables. The arrays `actionlist` and `golist` are s x 2 arrays, where s is the number of parse states, and are used to define where the values of the action and goto functions are saved in the array `parsetable`. Each element in these two arrays is the pair (b,t) where b is the starting position of the values saved for that state while t is the number of non-error values of the function for that state. The last array `productionlist` is a p x 2 array where p is the number of productions and, for each production A → w it holds the pair (x,|w|) where x is the code of A and |w| is the length of the string w.

Returning to the previous example, let the codes of the terminals, nonterminals, and productions be as follows:

<table>
<thead>
<tr>
<th>terminals</th>
<th>nonterminals</th>
<th>productions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ : 0</td>
<td>S : -1</td>
<td>1 : S→E</td>
</tr>
<tr>
<td>* : 1</td>
<td>E : -2</td>
<td>2 : E→E*T</td>
</tr>
<tr>
<td>+ : 2</td>
<td>T : -3</td>
<td>3 : E→T</td>
</tr>
<tr>
<td>id : 3</td>
<td>F : -4</td>
<td>4 : T→T+F</td>
</tr>
<tr>
<td>( : 4</td>
<td></td>
<td>5 : T→F</td>
</tr>
<tr>
<td>) : 5</td>
<td></td>
<td>6 : F→id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 : F→(E)</td>
</tr>
</tbody>
</table>
Using the above codes, the **action** and **goto** tables would internally be represented as follows:

<table>
<thead>
<tr>
<th>actionlist</th>
<th>gotolist</th>
<th>parsetable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 3:3</td>
<td>1 3:3</td>
</tr>
<tr>
<td>2</td>
<td>1 6:3</td>
<td>2 4:2</td>
</tr>
<tr>
<td>3</td>
<td>3 13:0</td>
<td>3 -4:4</td>
</tr>
<tr>
<td>4</td>
<td>4 17:4</td>
<td>4 -3:5</td>
</tr>
<tr>
<td>5</td>
<td>5 21:4</td>
<td>5 -2:6</td>
</tr>
<tr>
<td>6</td>
<td>6 23:0</td>
<td>6 -4:4</td>
</tr>
<tr>
<td>7</td>
<td>7 25:0</td>
<td>7 -3:5</td>
</tr>
<tr>
<td>8</td>
<td>8 25:1</td>
<td>8 -2:7</td>
</tr>
<tr>
<td>9</td>
<td>9 26:2</td>
<td>9 0:-6</td>
</tr>
<tr>
<td>10</td>
<td>10 32:0</td>
<td>10 1:-6</td>
</tr>
<tr>
<td>11</td>
<td>11 36:0</td>
<td>11 2:-6</td>
</tr>
<tr>
<td>12</td>
<td>12 40:0</td>
<td>12 5:-6</td>
</tr>
<tr>
<td>13</td>
<td>13 45:0</td>
<td>13 0:-5</td>
</tr>
<tr>
<td>14</td>
<td>14 51:3</td>
<td>14 1:-5</td>
</tr>
<tr>
<td>15</td>
<td>15 57:0</td>
<td>15 2:-5</td>
</tr>
<tr>
<td>16</td>
<td>15 61:0</td>
<td>15 5:-5</td>
</tr>
<tr>
<td>17</td>
<td>17 65:0</td>
<td>17 0:-3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18 1:-3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19 2:8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20 5:-3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>21 0:-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>22 1:9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23 1:9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>24 5:10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>25 -3:12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>26 -4:4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>27 -3:12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>28 0:-7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>29 1:-7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>30 2:-7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>31 5:-7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32 0:-4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>33 1:-4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Productionlist</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1:1</td>
</tr>
<tr>
<td>2</td>
<td>-2:3</td>
</tr>
<tr>
<td>3</td>
<td>-2:1</td>
</tr>
<tr>
<td>4</td>
<td>-3:3</td>
</tr>
<tr>
<td>5</td>
<td>-3:1</td>
</tr>
<tr>
<td>6</td>
<td>-4:1</td>
</tr>
<tr>
<td>7</td>
<td>-4:3</td>
</tr>
</tbody>
</table>

For example, the **action** values held in the above table for state 5 start at position 17 in the array **parsetable** and has 4 non-**error** values. Positions 17 through 20 represent the action values:

\[
\text{\$ : reduce } E\rightarrow T
\]
V.2 SLR(1) implementation

This section describes how to use the SLR(1) parser constructor with error recovery. This implementation has the restriction that no production can be of the form A \rightarrow e. Included in this section is a brief description of the input grammar, how to run the system, and how to interpret the output produced.

V.2.1 Input Grammar

The input for the program is the set of productions defining the CFG which the SLR(1) parsing table is to be constructed from. The input will be parsed in a free style format, that is, no formatting by columns or line boundaries will be used. The end of line character will be treated as a blank character and each symbol on the input file must be separated by one or more blanks.
In general, a terminal symbol is represented by a nonblank string, of 15 characters or less not beginning with the character "<", and is not one of the metasymbols ("->","$", and "."). In the event that the user may use one of the metasymbols used by the program, or a nonblank string beginning with a "<", the quote symbol has been given special meaning. If the quote is followed by a blank character, it will be treated as a terminal symbol. Otherwise, if the quote is followed by a nonblank string, the string following the quote will be treated as the name of the terminal symbol.

Nonterminal symbols are represented as character strings, of 15 characters or less, enclosed by the symbols "<" and ">". The first symbol of the string, if not the empty string, must begin with a nonblank character but blank characters can appear anywhere else in the string. The program also accepts the string "<>", which represents a nonterminal symbol whose name is the empty string.

Productions are represented by writing them in the form A -> w where A is a nonterminal, w is a sequence of grammar symbols, and "->" is a metasymbol recognized by the program. Each production is separated from the next using the metasymbol "." and after the last production, the metasymbol "$" must appear. The productions can be entered in any order except that the first production, on the input file,
must be the start production.

For example, the grammar presented in V.1 could be represented by the following piece of input:

```
<S> -> <e> .
<e> -> <e> * <t> . <e> -> <t> .
<t> -> <t> + <f> . <t> -> <f> .
<f> -> id . <f> -> ( <e> ) $
```

A shorthand notation also exists for productions having the same left hand side (i.e. productions of the form $A \rightarrow w$ where $A$ remains constant between the productions). In these cases, the productions can be entered in the form $A \rightarrow w_1 \, ! \, w_2 \, ! \, \ldots \, ! \, w_n$ where there exists the productions $A \rightarrow w_1 \, , \, A \rightarrow w_2 \, , \, \ldots \, , \, A \rightarrow w_n$.

For example, the grammar in section V.1 could have alternatively been written as:

```
<S> -> <e> .
<e> -> <e> * <t> ! <t> .
<t> -> <t> + <f> ! <f> .
<f> -> id ! ( <e> ) $
```

The order in which productions are found in the input file corresponds to the order in which they will be coded internally. In a similar manner, the terminal and nonterminal symbols will be coded in the order corresponding to their first appearance in the set of productions.
V.2.2 Running the SLR(1) parser constructor

The system can be run on the Vax-11 in the Moore School, by entering the following monitor level procedure call:

`$@[karl]slrbnf`

After invocation, the procedure will ask the user for the files used by the program, and run the program.

The first file to be requested is the file containing the set of productions, and is requested with the prompt:

`input:`

The second file request is for the output file which will contain all diagnostic and informative messages, and is requested with the prompt:

`output:`

The third file request is for the file that the created SLR(1) parsing tables should be saved on, and is requested with the prompt:

`internal representation:`

The last two file requests are for temporary files that can be used by the program, and are both requested with the prompt:

`temporary storage unit:`
Upon completion of the file requests, the program is run. The program will not produce any output, on the user's screen, nor will it ask the user for any further information unless the SLR(1) parsing table was created and contains conflicts (see section V.2.4 for handling this case).

This paper will not mention how to use the file containing the SLR(1) parsing tables except for a PASCAL program skeleton in appendix a.

V.2.3 Interpretation of the output file

The output can be broken into two major sections where the first section describes how the program parsed the input grammar and the second section prints the built SLR(1) parsing tables. However, the second section will be produced only if there were no errors detected in the first section.

The first page of the output is a copy of the input being parsed, along with any error messages indicating illegal syntax. If there were no syntactic mistakes in the input grammar, then this page will be an exact duplicate of the input file. Otherwise, portions of the input file will be written, and will be interspersed with syntactical errors recognized by the program.
For example, the erroneous input:

\[ \langle S \rangle \rightarrow \langle A \rangle . \ \langle A \rangle \rightarrow a \ \langle A \rangle \ b . \ \ A \rightarrow a \ b \ \$

would produce the following output:

\[ \langle S \rangle \rightarrow \langle A \rangle . \ \langle A \rangle \rightarrow a \ \langle A \rangle \ b . \ \ A ***\text{illegal LHS} \]

In this example, the program is reporting that the production has a terminal symbol on the left hand side of the production.

The next three subsections of the output reports the coding scheme of terminals, nonterminals, and productions used by the program.

For example, the input:

\[ \langle S \rangle \rightarrow \langle E \rangle . \ 
\langle E \rangle \rightarrow \langle E \rangle \ * \ \langle T \rangle \ ! \ \langle T \rangle . \ 
\langle T \rangle \rightarrow \langle T \rangle \ + \ \langle F \rangle \ ! \ \langle F \rangle . \ 
\langle F \rangle \rightarrow \text{id} ! \ ( \ \langle E \rangle \ ) \ \$

81
would produce the following output:

**TERMINAL NODES:**

---------

1. *
2. +
3. id
4. ( 
5. )

**NONTERMINAL NODES:**

---------

-1. <S>
-2. <E>
-3. <T>
-4. <F>

**PRODUCTIONS:**

---------

1. <S> → <E> 'EOF MARKER'
2. <E> → <E> * <T>
3. <E> → <T>
4. <T> → <T> + <F>
5. <T> → <F>
6. <F> → id
7. <F> → ( <e> )

The program provides additional information with the coding schemes, that is, if the string "*undef*" proceeds a nonterminal, then that nonterminal does not occur on the
left hand side of any production recognized while parsing the input file.

Below the coding scheme is a diagnostic summary of how well the program did in parsing the given input bnf. If everything is acceptable to the program, it will print the message "successful parse" and attempt to construct the SLR(1) parsing tables. Otherwise, it will give an error summary of why it thought the input was wrong, and abort any further calculations.

Should the input grammar be successfully parsed, the program then attempts to build the SLR(1) parsing tables. To begin with, it computes the first and follow sets for each nonterminal, and prints out these sets. Second, it prints out the sets of SLR(1) items defining the core of each state.
For example, the previous input grammar would produce output for the first five states as follows:

------------------------------- STATE : 1
1)  \( \text{<S> \rightarrow . \text{<E> '<EOF MARKER>'}} \)
------------------------------- STATE : 2
7)  \( \text{<F> \rightarrow ( . \text{<E> }} \)
------------------------------- STATE : 3
6)  \( \text{<F> \rightarrow id .} \)
------------------------------- STATE : 4
5)  \( \text{<T> \rightarrow <F>} \)
------------------------------- STATE : 5
3)  \( \text{<E> \rightarrow <T> .} \)
4)  \( \text{<T> \rightarrow <T> . + <F>} \)

The last section of the output, for a run, is a readable form of the produced parsing table followed by the size of the array `parsetable`. Non-error values, of the parsing tables, for each state are listed separately with the `action` values preceding the `goto` values.
For example, the output produced by the program for the parsing values for the first state would be as follows:

```
-----------------------------
STATE 1
id SHIFT TO 3
( SHIFT TO 2
<F> GO TO 4
<T> GO TO 5
<E> GO TO 6
-----------------------------
```

**V.2.4 Conflict Resolution**

Sometimes, when a CFG G is provided as input to the SLR(1) parser constructor it can not produce a SLR(1) parser for G since L(G) is not in the class of languages of SLR(1). In such cases, the construction method has produced conflicts in the action table.

For example, the grammar in figure 5.1 is an example of a natural grammar for arithmetic expressions with operators + and *. The LR(0) characteristic automaton, for this grammar, and the follow sets are shown in figure 5.2. In states 9 and 10, there will exist S/R conflicts on the
symbols + and * if the SLR(1) parser is built from the characteristic automaton. This can also be seen in the output produced by the program for such an input (see figure 5.3).

\[
\begin{align*}
S &\rightarrow E \\
E &\rightarrow E + E \\
E &\rightarrow E * E \\
E &\rightarrow \text{id} \\
E &\rightarrow (E) \\
\end{align*}
\]

**figure 5.1**

\[
\begin{align*}
1: S &\rightarrow .E \\
2: E &\rightarrow (E) \\
3: E &\rightarrow \text{id} \\
4: S &\rightarrow .E \\
E &\rightarrow E + E \\
E &\rightarrow E * E \\
5: E &\rightarrow (E.) \\
E &\rightarrow E + E \\
E &\rightarrow E * E \\
6: E &\rightarrow E + E \\
7: E &\rightarrow E * E \\
8: E &\rightarrow (E) \\
9: E &\rightarrow E + E \\
E &\rightarrow E + E \\
E &\rightarrow E * E \\
10: E &\rightarrow E * E \\
E &\rightarrow E + E \\
E &\rightarrow E * E \\
\end{align*}
\]

FOLLOW(S) = \{$\}$ 
FOLLOW(E) = \{$\$, $+$, $*$, $\}$

**figure 5.2**
It turns out that these conflicts can be resolved in favor of either a **shift** or a **reduce** action by knowing the precedence and associativity of these two operators. For example, looking at state 9 and the operator *`, the parser is attempting to recognize the sentential form:

```
E + E * E
```

Assuming that * has precedence over +, it is clear that we
want to shift on the input symbol * to first recognize the string E * E and reduce it to the string E producing the new sentential form

E + E

Should the grammar in the input file produce conflicts, the program will arbitrarily pick one of the action definitions for the symbol causing the conflict in the state and discard all other conflicting entries. This choice is reported to the user as shown in figure 5.3. In each case, the "OLD ENTRY: xx" represents the entry chosen by the program while the "CONFLICTING ENTRY: yy" states a discarded entry. Hence, in state 9, the arbitrary choice, for the symbol *, was to reduce on the production labelled 2 (i.e. E->E+E).

To allow the user to change the arbitrary choice made by the program, the program will also become interactive if any conflicts arise in building the SLR(1) parser. That is, the program will prompt the user with the prompt:

ENTER STATE TO RESOLVE:

To this response, two choices are available.

If the user responds with the number 0, the program will stop so that the user can look at the output file in order to identify all existing conflicts in building the SLR(1) parser. If the user feels that these conflicts can
not be resolved, then he is out of luck. Otherwise, the user should rerun the program and when getting the above prompt, he should resolve the conflicts by using the second option.

The second option in responding to the above prompts to type in the state that the user wants to resolve. After the user completes his answer, the program will print out the core of the state, for verification, and will ask the user if it is the state he wanted.

The next request by the program is for the user to provide the integer code of the terminal symbol causing the conflict using the prompt:

ENTER SYMBOL NUMBER TO RESOLVE:

As above, the program will verify the user's response by printing out the terminal's name and asking the user if it is the correct terminal symbol. Again, a "N" response will cause the program to reprompt for a state to resolve while a "Y" response will have to program continue processing the resolution.

The next request, after the symbol request, is for the action function's value for the state and symbol with the prompt:

ENTER NEW ACTION TO TAKE:

If the value provided by the user is a positive integer (and
hence a **shift** action), the program will print out the core of the state the shift is to. If the value given by the user is negative (and hence a **reduce** entry), the program will print out the production associated with the label provided by the user. In either case, it will then ask the user if this was what the user wanted and again verify the user's input.

The program will provide the user one last chance, after the conflict resolution has been specified, to disregard the conflict resolution. A "Y" response by the user will cause the resolution to be processed while a "N" response will disregard the resolution provided by the user. In either case, the program will then request for another conflict resolution with the prompt:

```
ENTER STATE TO RESOLVE:
```

At this point, the whole process repeats unless the user responds with a 0. If a 0 is typed in by the user, then no more conflict resolutions will be processed and the program will build the SLR(1) parser. Note that the program will not produce an SLR(1) parser unless at least one conflict has been resolved.
V.2.5 Size Restrictions

This program contains several size restrictions which are as follows:

i) No more than 100 terminal symbols may be used.

ii) No more than 200 nonterminal symbols may be used.

iii) No more than 300 productions may appear in the input.

iv) No terminal or nonterminal name may exceed 15 characters.

v) For each production $A \rightarrow w$, $w$ can not be a string, of terminal and nonterminal names, exceeding a length of ten names.

vi) The number of parse states, created by the program, must not exceed 600.

vii) The number of SLR(1) items, excluding the items of the form $A \rightarrow . w$, must not exceed 9,999.
The size of the array `parsetable` can not exceed the dimensions of 10,000 x 2.

**V.3 LR(1), LALR(1), Weak and Strong Compatibility**

`parser generators`

This section describes how to use the program which can build either LR(1), LALR(1), weak compatible, or strong compatible parsing tables. Included in this section is a brief description of the input grammar, how to run the program, and how to interpret the output.

**V.3.1 Input Grammar**

The input for the program is the set of productions defining the CGF from which the parsing tables are to be produced. These productions can be optionally preceded with a list of terminals and nonterminals, allowing the user to specify the integer codes given to these symbols.

The input will be parsed in a free style format, that is, no formatting by columns or line boundaries will be used. The end of line character will be treated as a blank
character and each symbol on the input must be separated by at least one blank.

In general, a terminal symbol is any nonempty string of nonblank characters which does not begin with the character "<". However, it can not be any of the metasymbols (i.e. "|", ".", "#$", ":-", ":", or "e"). In the event that the user wants to use one of the metasymbols or a string beginning with a "<", as a terminal symbol, the quote symbol must precede the nonblank string.

Nonterminal symbols are represented as any character string enclosed by the symbols "<" and ">". The characters composing the name of the nonterminal can be any character (including the blank) except the symbol ">", and includes the name composed by the empty string ("<>").

Productions are represented by writing them in the form A \rightarrow w where A is the name of a nonterminal, w is a sequence of terminal and nonterminal names, and "\rightarrow" is a metasymbol recognized by the program. The symbol "e" has been reserved to represent the empty string so that productions of the form A \rightarrow e can be written.

Productions are separated from each other using the metasymbol ".", and no symbols should follow the last production. Productions having the same left hand side, i.e. of the form A \rightarrow w_1, A \rightarrow w_2, \ldots, A \rightarrow w_n, can be
written in the form \( A \rightarrow w_1 \mid A \rightarrow w_2 \mid \ldots \mid A \rightarrow w_n \) where the metasymbol "|" is treated as an "or" symbol.

For example, the grammar

\[
S \rightarrow A \quad A \rightarrow aAb \quad A \rightarrow e
\]

could be entered with the input:

\[
<\text{start symbol}> \rightarrow <A> . \\
<A> \rightarrow a <A> b \mid e
\]

Productions, when parsed, will be coded internally using the order in which they appear on the input. The only restriction on the order in which the productions are written is that the start production must appear first.

Unlike the SLR(1) parser constructor, this program optionally allows the user to specify the coding scheme of the nonterminal and terminal symbols. That is, before the start production the user is allowed to provide a list of terminals, followed by a list of nonterminals, followed by the metasymbol "#". It is not necessary that all terminals and nonterminals appear in these lists, and either of the list may be empty. Elements in these lists will be labeled in the order that they are found (1 for the first terminal, 2 for the second terminal etc. and -1 for the first nonterminal, -2 for the second nonterminal etc.). Any remaining terminals, or nonterminals, not specified by these lists will be labelled according to the order of first
For example, assume using the previous grammar that the user wants the terminal b to be labelled 1 and terminal a to be labelled 2. This could be done by using the input:

\[
\begin{align*}
  b & \ # \\
  \text{<start symbol>} & \rightarrow \text{<A> } . \\
  \text{<A>} & \rightarrow \text{a <A> b } | \text{ e }
\end{align*}
\]

The program described by this section in fact has used the SLR(1) parsing tables (produced by running the SLR(1) program described in section V.2) to parse the input for this program. Hence, the description of the input rules can be formally described by the set of rules used in creating the SLR(1) parsing tables which are as follows:
V.3.2 Running the program

The program can be run on the Vax-11 in the Moore School by entering the following monitor level procedure call:

@[karl]runnewbnf

After invocation, the procedure will ask the user for the files used by the program, and then run the program.

The first file requested by the procedure is the file containing the set of productions, and is requested with the prompt:

BNF FILE:
The second file is request is for the output file which will contain all diagnostic and informatory messages, and is requested with the prompt:

OUTPUT FILE:

The last request is for the file to save the parsing tables created and is requested with the prompt:

TABLE:

Upon completion of the file requests, the program is run. After the program finishes reading the input bnf file, the program will request the user to specify what type of parser should be created with the prompt:

ENTER OPTION
0 - COMPUTE FIRSTS ONLY
1 - BUILD LR(1) PARSE TABLE
2 - BUILD LALR(1) PARSE TABLE
3 - BUILD WEAK COMPATIBLE LR PARSE TABLE
4 - BUILD STRONG COMPATIBLE LR PARSE TABLE

Once the user responds, the program will build the corresponding parse table, printing out "BUILDING STATE X" as it tries to build state X. This completes all interaction the program has with the user.

The first page of the output file is a copy of the input being parsed, along with any error messages describing illegal syntax.
For example, the erroneous input:

\[
\langle S \rangle \rightarrow \langle A \rangle . \ \langle A \rangle \rightarrow a \ \langle A \rangle \ b . \ \ A \rightarrow e
\]

would produce the following output:

**INPUT PARSE OF PRODUCTIONS:**

```
\langle S \rangle \rightarrow \langle A \rangle . \ \langle A \rangle \rightarrow a \ \langle A \rangle \ b . \ \ A \rightarrow e
```

*** 32) PRODUCTION DEFINITION EXPECTED***

The above error is stating that at the beginning on column 32 of the previous input line, the program was expecting to find a production but found something else (i.e. the terminal symbol A).

The next three subsections of the output file, after the parse of the input, reports the coding scheme of the terminals, nonterminals, and productions used by the program.

For example, the input:

```
a \ b \ #
\langle start \ symbol \rangle \rightarrow \langle A \rangle .
\langle A \rangle \rightarrow a \ \langle A \rangle \ b \ | \ e
```
would produce the following output:

**TERMINALS:**
--------
0. $\text{EOF}$
1. a
2. b

**NON-TERMINALS:**
------------------
-1. <start symbol>  *START SYMBOL*  *UNIQUE*
   *NOT USED ON RHS*
-2. <A>

**PRODUCTIONS:**
--------
1<start symbol> $\rightarrow$ <A>
2<A> $\rightarrow$ a <A> b
3<A> $\rightarrow$ e

As can be seen by the above example, additional informational messages about nonterminal symbols are provided, and are as follows:

*START SYMBOL* - States that the nonterminal symbol has been recognized as the start symbol.

*UNIQUE* - States that the start symbol does not occur anywhere else in the productions and hence is a valid start symbol.
*NOT UNIQUE* - States that the start symbol occurs in another production besides the start production and hence is an invalid start symbol.

*NOT USED ON RHS* - States that the nonterminal never appears on the right hand side of any production.

*NT NOT REACHABLE* - States that the nonterminal cannot appear in any of the sentential forms and hence need not be part of the input grammar.

*NT REPRESENTS NO TERMINAL STRINGS* - States that there is not any terminal strings derivable from the nonterminal.

*NT NOT DEFINED* - States that the nonterminal does not appear on the left hand side of any production recognized from the input file.

After the coding schemes, the program will print the first set of each nonterminal.

Finally, if the user selects to have a parser constructed, the program will construct it and print the appropriate parsing tables. The output of the parser will be printed by states where each state will contain its core
(items) and non-error action and goto values.

For example, using the input grammar used above, and if the user chose to build a strong compatible LR parsing table, the parse tables printed would be as follows:

--- STRONG COMPATIBLE L R (1) CHARACTERISTIC TABLE ---

STATE : 1

1) <start symbol> -> . <A>
   LOOKAHEADS:
   $\text{EOF}$

TABLE ENTRIES:

$\text{EOF}$ REDUCE BY 3
a SHIFT TO 3
<A> GO TO 2

STATE : 2

1) <start symbol> -> <A> .
   LOOKAHEADS:
   $\text{EOF}$

TABLE ENTRIES:

$\text{EOF}$ REDUCE BY 1

STATE : 3

2) <A> -> a . <A> b
   LOOKAHEADS:
   $\text{EOF}$
   b

TABLE ENTRIES:

a SHIFT TO 3
b REDUCE BY 3
<A> GO TO 4
STATE : 4

2) <A> -> a <A> . b
LOOKAHEADS:
   $\texttt{EOF}$
   b

TABLE ENTRIES:

b SHIFT TO 5

STATE : 5

2) <A> -> a <A> b .
LOOKAHEADS:
   $\texttt{EOF}$
   b

TABLE ENTRIES:

$\texttt{EOF}$ REDUCE BY 2
b REDUCE by 2
Appendix A

Sample PASCAL skeleton for use of SLR(1) parsing tables

Program doparse(table, {any other files used by program} );

Const
   numberstates    = x;  \{x> of actual parse states\}
   parsetablesize  = y;  \{y> actual size of
                     array parsetable\}
   numberproductions = z;  \{z> actual number
                        of productions\}
   errorvalue      = n;  \{n value not in set of labels\}

type

\{the path will be represented as a stack
using a linear list\}

parsestack = ^stacknode;
stacknode = record
   topstate : integer;
   next     : parsestack
end;

var

   table : file of integer;  \{file containing
                                  parsing tables\}

function push(stack : parsestack;
               newstate : integer) : parsestack;

\{returns stack with new state added in front\}

var temporary : parsestack;

begin
   new(temporary);
   with temporary^ do begin
      topstate:= newstate;
      next:= stack
   end;
   push:= temporary
end;
function pop(stack : parsestack) : parsestack;
    {removes the top element of the stack}
begin
    pop:=stack^.next;
    dispose(stack)
end;

function top(stack : parsestack) : integer;
    {returns state on top of stack}
begin
    top:=stack^.topstate
end;

function empty(stack : parsestack) : parsestack;
    {returns an empty stack}
begin
    while stack<>nil do stack:=pop(stack);
    empty:=nil
end;

function gettoken : integer;
    {This routine returns the label of the next terminal
     occurring in the input file}
end;

procedure semantics(stack : parsestack;
        production : integer);
    {does any semantic routines associated with reducing
     the given production}
end;

procedure errormessages(state , symbol : integer);
    {prints out message corresponding to error value
     for state and symbol}
end;
function parse : boolean;

   {parses input. returns true if no parsing errors are found in parsing the input}

const  eoftoken = 0;

type

{representation of an entry in parsetable}

tableentry = record
   symbol , value : integer
end;

{representation of a reference to a group of entries in parsetable}

stateentry = record
   startposition , size : integer
end;

{representation of a production in productionlist}

productionentry = record
   lhssymbol , rhslength : integer
end;

var

parsetable : array [ 1 .. parsetablesize] of tableentry;

actionlist , gotolist : array [ 1 .. numberstates] of stateentry;

productionlist : array [ 1 .. numberproductions] of productionentry;

{other parameters passed with parsing tables}

topstate, {actual number of parse states}
parsestart, {start state}
errorstart, {forced shift state on error recovery}
errorcontinue, {init-error state}
topoftable, {actual size of parsetable}
productioncount {actual number of productions}
   : integer;
{local variables}  

{next terminal from input}  

{next action to take in parsing input}  

{true when have parsed whole input}  

{true if any parsing errors}  

{holds path}  

{reads in parsing tables}  

{reads in next integer from file table}  

begin  

begin  

reset(table);  

getin(topstate);  

getin(parsestart);  

getin(errorstart);  

getin(errorcontinue);  

getin(topofhtable);  

getin(productioncount);  

for index:=1 to topstate do begin  

with actionlist[index] do begin  

getin(startposition);  

getin(size)  

end;  

end;  

begin;
for index:=1 to toptable do
    with parsetable[index] do begin
        getin(symbol);
        getin(value)
    end;
for index:=1 to productioncount do
    with productionlist[index] do begin
        getin(rhslength);
        getin(lhssymbol)
    end
end;

function clear(stack : parsestack;
    newbottom : integer ) : parsestack;

    {empties stack and put value on bottom of stack}
begin
    clear:=push(empty(stack),newbottom)
end;

function popelements(stack : parsestack;
    amount : integer ) : parsestack;

    {takes the requested amount of states off the stack}
begin
    if (amount = 0) or (stack = nil) then popelements:=stack
    else popelements:=popelements(pop(stack),
        pred(count))
end;

function popoffproduction(stack : parsestack;
    count : integer ) : parsestack;

    {takes the requested amount of states off the stack,
    but if stack underflow occurs, it resets
    the bottom state}
begin
    stack:=popelements(stack,count);
    if stack = nil then popoffproduction:=push(stack,errorcontinue)
    else popoffproduction:=stack
end;
function findvalue(entry: stateentry;
        testsymbol: integer): integer;

{Looks up the value of the function, for
the given state and symbol}

var found: boolean;
    index, outofrange: integer;

begin
    findvalue := errorvalue;
    found := false;
    with entry do begin
        index := startposition;
        outofrange := startposition + size
    end;
    while (index < outofrange) and not found do
        with parsetable[index] do
            if testsymbol > symbol
                then index := succ(index)
            else if testsymbol = symbol
                then begin
                    found := true;
                    findvalue := value
                end
            else index := outofrange
end;

function overdefined(stack: parsestack;
                    var token: integer): parsestack;

{handles overdefined actions}

begin
    if token = eoftoken
        then begin
            overdefined := empty(stack);
            stop := true
        end
    else begin
        overdefined := push(clear(stack, errorcontinue),
                             findvalue(actionlist[errorstart],
                                       token));
        token := gettoken
    end
end;
function unknown(stack : parsestack;  
    var token : integer ) : parsestack;

    {handles error actions}

begin
  parseerror:=true;
  errormessages(top(stack),token);
  unknown:=overdefined(stack,token)
end;

function doshift(stack : parsestack;
    shiftaction : integer;
    var token : integer ) : parsestack;

    {handles performing a shift action}

begin
  doshift:=push(stack,shiftaction);
  token:=gettoken
end;

function doreduction(stack : parsestack;
    production : integer;
    var token : integer ) : parsestack;

    {handles performing a reduction}

var gotovalue : integer;

begin
  gotovalue:=findvalue(gotolist[top(stack)],
    productionlist[production].lhssymbol);
  if gotovalue = errorvalue
    then doreduction:=unknown(stack,token)
  else begin
    semantics(stack,production);
    doreduction:=push(popoffproduction(stack,
        productionlist[production]
        .rhslength),
        gotovalue)
  end
end;
begin
getparsetable;
stack:=push(nil,parsestart);
stop:=false;
errorvalue:= -suc(productioncount);
parseerror:=false;
token:=gettoken;
repeat
    value:=findvalue(actionlist[top(stack)],token);
    if value = errorvalue
        then stack:=unknown(stack,token)
    else if value < -1
        then stack:=doreduction(stack,-value,token)
    else if value = -1
        then stop:=true
    else if value = 0
        then stack:=overdefined(stack,token)
    else stack:=doshift(stack,value,token)
until stop;
parse:= not parseerror
end;
Appendix B

Sample PASCAL skeleton for use of the LR(1), LALR(1), weak compatible, and strong compatible parsing tables

Program doparse(table, {any other files used by program} );

const

numberstates = x; {x ≥ # of actual parse states}
parsetablesize = y; {y ≥ actual size of array parsetable}
numberproductions = z; {z ≥ actual number of productions}

type

{the path will be represented as a stack using a linear list}

parsestack = ^stacknode;
stacknode = record
topstate : integer;
next : parsestack
end;

var

table : file of integer; {file containing parsing tables}
function push(stack : parsestack;
    newstate : integer) : parsestack;

    {returns stack with new state added in front}

var temporary : parsestack;

begin
    new(temporary);
    with temporary^ do begin
        topstate:=newstate;
        next:=stack
        end;
    push:=temporary
end;

function pop(stack : parsestack) : parsestack;

    {removes the top element of the stack}

begin
    pop:=stack^.next;
    dispose(stack)
end;

function top(stack : parsestack) : integer;

    {returns the top of the stack}

begin
    top:=stack.topstate
end;

function empty(stack : parsestack) : parsestack;

    {returns an empty stack}

begin
    while stack <> nil do stack:=pop(stack);
    empty:=nil
end;

function gettoken : integer;

    {This routine returns the label of the next terminal
    occurring in the input file}
end;
procedure semantics(stack : parsestack;
    production : integer);

  {Does any semantic routines associated with reducing
   the given production}
end;

procedure errormessages(state , symbol : integer);

  {prints out message corresponding to error
   value for state and symbol}
end;

function parse : boolean;

  {Parses input. Returns true if no parsing errors
    are found in parsing the input}
const  eoftoken = 0 ;
type

{representation of an entry in parseable}

  tableentry = record
    symbol , value : integer
  end;

{representation of a reference to a group of entries
 in parsing table}

  stateentry = record
    startposition , size : integer
  end;

{representation of a production in productionlist}

  productionentry = record
    lhssymbol,rhslength : integer
  end;
var

parsetable : array [ 1 .. parsetablesize ] of tableentry;

actionlist, gotolist : array [ 1 .. numberstates ]
of stateentry;

productionlist : array [ 1 .. numberproductions ]
of productionentry;

topstate : integer; {actual number of parse states}

{other local variables}

token : integer; {next terminal from input}

errorvalue : integer; {made up number for error values}

value : integer; {next action to take in parsing}

stop : boolean; {true when have finished parsing}

parseerror : boolean; {true if any parsing error occurs}

stack : parsestack; {holds path}

procedure getparsetable;

{reads in parsing tables}

var index, j : integer;

procedure getin(var invalue : integer);

{gets in next integer from file}

begin
  invalue := table^;
  get(table)
end;
begin
  getin(topstate);
  for index:=1 to topstate do begin
    with actionlist[i] do begin
      getin(startposition);
      getin(size);
      for j:=startposition to size do
        with parsetable[j] do begin
          getin(symbol);
          getin(value)
        end
    end
  end
end

function popproduction(stack : parsestack;
                        count : integer ) : parsestack;

  {takes the requested amount of states off the stack}
begin
  while count>0 do begin
    stack:=pop(stack);
    count:=pred(count)
  end
end

function findvalue(entry : stateentry;
                    testsymbol : integer ) : integer;

  {looks up the value of the function, for
   the given state and symbol}

var found : boolean;
  index , outofrange : integer;
begin
    findvalue:=errorvalue;
    found:=false;
    with entry do begin
        index:=startposition;
        outofrange:=startposition + size
    end;
    while (index < outofrange ) and not found do
        with parseable[index] do
            if testsymbol > symbol
                then index:=succ(index)
            else if testsymbol = symbol
                then begin
                    found:=true;
                    findvalue:=value
                end
            else index:=outofrange
    end;

function doshift(stack : parsestack;
    shiftaction : integer;
    var token : integer) : parsestack;

  {handles performing a shift}
begin
    doshift:=push(stack,shiftaction);
    token:=gettoken
end;

function doreduction(stack : parsestack;
    production : integer) : parsestack;

  {handles performing a reduction}
var gotovalue : integer;
begin
gotovalue:=findvalue(gotolist[top(stack)],
  productionlist[production].lhssymbol);
if gotovalue = errorvalue
  then begin
    doreduction:=empty(stack);
    parseerror:=true;
    stop:=true
  end
else begin
  semantics(stack,production);
  doreduction:=push(popoffproduction(stack,
    productionlist[production]
    . rhslength),
    gotovalue)
end
end;

begin
getp Parsetable;
stack:=push(nil,1);
stop:=false;
parseerror:=false;
errorvalue = 0;
token:=gettoken;
repeat
  value:=findvalue(actionlist[top(stack)],token);
  if value = errorvalue
    then begin
      stack:=empty(stack);
      parseerror:=true;
      stop:=true
    end
  else if value < -1
    then stack:=doreduction(stack,-value)
  else if value = -1
    then stop:=true
  else stack:=doshift(stack,value,token)
until stop;
parse:= not parseerror
end;
References


[DeR72] DeRemer, F. - XPL distribution tape containing LALR translator writing system. Information Sciences, UC Santa Cruz, CA. (1972)


[O'H76] O'Hare, M. F. - Modification of the LR(k) parsing technique to include automatic syntactic error recovery. Senior thesis, UC Santa Cruz, Santa Cruz, CA. (1976)


[P&D78] Pennello, T. J. and DeRemer, F. - Practical error recovery for LR parsers. Tech. Rpt. 78-1-002, Information Sciences, UC Santa Cruz, Santa Cruz, CA (1977)

[P&D79] Pennello, T. J. and DeRemer, F. - Practical error recovery for LR parsers. Submitted for publication in TOPLAS, march 1979