NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

UNIVERSITY OF PENNSYLVANIA
THE MOORE SCHOOL OF ELECTRICAL ENGINEERING
SCHOOL OF ENGINEERING AND APPLIED SCIENCE

CONSTRUCTION METHODS OF LR PARSERS

Karl Max Schimpf

Philadelphia, Pennsylvania

May 1981

A thesis presented to the Faculty of Engineering an Ap
Science of the University of Pennsylvania in partia
fulfillment of the requirements for the degree of Maste
Science in Engineering for graduate work in Computer an
Information Science.

Jean H. Gallier

Jean H. Gallier

Aravind Joshi

abstract

This paper presents five different LR parser generat in error recovery method which is derived directly fr< ,R parser. The parsers presented include the origina i parser defined by Knuth, The SLR(l) and LALR(l) >rs defined by DeRemer, and the weak and strong itible LR parsers presented by Pager* All five parse been implemented by the author using two programs• termore, the implementation of the SLR(l) parser rator includes an error recovery method and produces .) parser with error recovery built ±n*

Chapter I

Introduction

It is a well known fact that of all the determi tring parsers, the class of LR parsers recogn .argest class of context free languages [Knu65]. LR ire quite powerful and are able to recognize virtua rogramming languages in existance today. These vere first introduced by Knuth [Knu65] with his o version known as an LR(1) parser. Unfortunately, his requires extensive resources and hence is impractical or parsing any programming language.

Several alternative parsing methods have sinc resented which reduce the resource requirement roducing more practical LR parsers. Some of these ccomplish this result by reducing the class of la umber of parse states built and hence an overall red n the resource requirements. The most common forms a ype of LR parser are the SLR(l) and LALR(l) p resented by DeRemer [DeR69].

Another form of resource reduction used by LR p s a method of performing state minimization on th€ arser* Two of these state minimization methods have roposed by Pager [Pag77a, Pag77b] called weak and ompatible LR parsers* In these parsers, he restict tate reductions to maintain the power of the LR(1) nd hence the resultant parser recognizes the same els anguages as the original LR(1) parser*

This paper presents five different LR parser gene nd an error recovery method which is derived direct! he LR parser. The parsers presented include the ot R(l) parser defined by Knuth [Knu65], the SLR< ALR(l) parsers defined by DeRemer [DeR69], and the we trong compatible LR parsers presented by Pager [Pa 11 five parsers have been implemented by the author wo programs. Furthermore, the implementation of the arser generator includes the implementation of an ecovery method and produces an SLR(l) parser witt ecovery built in.

›atible LR parsers, presented by Pager [Pag7
›rtunately only provides a partial explanation of
›rithms which build these parsers. These algorithms
:ain minor inconsistancies and omissions which tend
:ure the basic nature of the algorithms. This p
sents Pager's algorithms in a modified notation w
›lifies the comprehension of the code. It also prov
›re complete explanation of the algorithms, and incl
ew minor algorithms omitted by Pager.

The problem with LR parsers, when used in a compi
that they are designed as a syntactic method which
ldes if the given input string belongs to a language
class accepted by the LR parser. Hence, once the f
egal input symbol is found, the parser stops repor
lure. However, when a compiler parses a program, i
antageous to have the compiler report as many additi
ors as possible.

In order to improve the LR parser's capabilities
in a compiler, this paper also presents a pu
tactic error recovery scheme to recognize additi
ors. Furthermore, the method has been designed so
can be directly incorporated into the LR parser. He
additional routines are necessary in order to per
or recovery and parse the rest of the input.

>ased on the method used by Pennello and DeRemer [P&D
:h has a separate error recovery routine that incl
>r correction. The control strategy used is to se
remainder of the input, starting from the ill
>ol, and verify that it only consists of '"vi
jments" (substrings derivable from its grammar).
>r recovery method presented in this paper has
Lemented using the SLR(l) parser as its basis. Howe
method is general enough that the same method c
Lly be applied to any of the other LR parsers prese
:his paper.

Chapter two starts by setting up preliminary nota
context free languages and derivations. This nota
lsed to describe the basic strategy used by LR pars
last sections of the chapter cover the ac
struction methods which will yield the LR(1) parser
result.

Chapter three describes how each of the other
Lemented parser constructors are built. The SLR(l)
1(1) construction methods are presented using the L
racteristic automaton as their basis for construct
sr's notion of compatibility, the definitions of
< and strong compatibility, and the algorithms use
junction with the construction of these two parsers

described.

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

Chapter five concludes the paper by discussing brie
two programs used for the implementation. One prog
tructs an SLR(1) parser with error recovery built
other program, using our modification of Pager's conc
ompatibility, can build either an LR(1), LALR(1), wea
trongly compatible LR parser.

Chapter II

The construction of the LR(1) parsing tables

This chapter describes how LR(1) parsing tab created* In order to do this, let me start out by 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 U T) is the finite set of *grammar symbols*;

S is a nonterminal symbol in N, called the start symbol; and

A production (A,a.) will be denoted in the form A there is a special start production S -> S' wher S does not occur in any other production in P* also a special symbol $ 6 T, which denotes th string being parsed, and does not appear in any

For notational convenience, upper case lett used to denote nonterminal symbols, lower caa denote terminal symbols, underlined upper case denote grammar symbols, and underlined lower caa denote strings of grammar symbols (strings iu The symbol ja will be reserved to denote the empt

11.1*1 Derivations

Given a CFG G « (N , T , P , S) , let t «> : (N U T)* x (N U T)* be defined by the set c

{ (aBc.abc) I B 6 N; a,jb,jc (N U T)*;
and B -> b. in P}

In other words, given any string in (NUT) c ^aBc., with B a nonterminal symbol in N at production B -> b^ in P, we say that the string the string abc in a one step derivation using I will be denoted as jaBj: »> abc * Also, let $\overset{+}{*}$> anc the transitive and transitive reflexive clo

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

$$\{\underline{a}B\underline{c} \Rightarrow_R \underline{abc} \mid \underline{a}B\underline{c} \Rightarrow \underline{abc} \text{ and } \underline{c} \in T^*\}$$

In other words, $\Rightarrow_R$ is the one step derivation, derivation is applied to the rightmost nonterminal in the string $\underline{a}B\underline{c}$. Let $\overset{+}{\Rightarrow}_R$ and $\overset{*}{\Rightarrow}_R$ denotes the and transitive reflexive closures of $\Rightarrow_R$, respecti

II.1.2 Language generated by a context-free gramma

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

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

Note: The order in which $\Rightarrow$ is applied has no eff resulting terminal string produced. Hence th L(G), generated by G, could be alternatively be d the set

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

Using the above definitions, an LR(1) gramma loosely defined as follows:

$\underline{a} \in L(G)$ (derived via a rightmost derivat parsed deterministically in a single scan fr right, having the ability to look ahead one the point of scanning.

II.2 Sentential forms and their viable prefixes

An LR(1) parser, when scanning the input (of to be parsed), is essentially looking for a mat or more strings that can be derived from the symbol. More formally, the LR(1) parser is recognize a sentential form which is an element

$\{ \underline{a} \mid S \overset{*}{\Rightarrow}_R \underline{a} \text{ and } \underline{a} \in (N \cup T)^* \}$

In recognizing a sentential form, the LR(1) really interested in knowing whether it has sca of the input string such that a reduction can be that is, when the sentential form is the str where $\underline{a},\underline{b} \in (N \cup T)^*$; $\underline{c} \in T^*$; and $B \to \underline{b} \in$ this information, a reduction of $\underline{b}$ to B can be the rightmost derivation string that $\underline{s}$ came from. known as finding the handle. The handle is def pair $(|\underline{ab}|, B \to \underline{b})$ such that $S \overset{*}{\Rightarrow}_R \underline{abc}$. The $|\underline{a}$ the length of the handle, which states the pos the string $\underline{b}$ can be reduced to B using $B \to \underline{b}$.

ab is called the viable prefix or characte [A&U77].

Using the above definitions, it is fai characterize what an LR(1) parser does. It sca from left to right, looking for a viable p finding it, the string is reduced with the production of the viable prefix. Using the re derived from the viable prefix concatenat unscanned input, the parser repeats the a looking for another viable prefix. This co either the input has been reduced to the start failure occurs by not finding any legal viable

II.3 LR(1) Characteristic Automaton

It is fundemental result that viable pref from CFG's are regular. Therefore a determi automaton, called the characteristic automaton can be built to recognize the set of legal via Furthermore, once the characteristic automat built, the LR(1) parser can be directly derived

Let a marked production be of the form where A -> ab is a production in P, and "." is

auction's right hand side has been recognized in
ing being scanned. Hence the marked produc
> *a* . *b* represents the fact that the LR(1) parser
nned the string *sa*, where *s* is some string that occu
ore the string *a* in the input.

Expanding this to include a set of look-ahead symb
an *item* be defined as the pair [A -> *a* . *b* , LA] w
> *a* . *b* is a marked production, and LA is a subset o
›ting the set of all terminal symbols which can fo
production and is called the *set of lookahead symb*
ns, essentially, describe two things:

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

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

Each state of the characteristic automaton is the
all items with the same viable prefix. When buildin
e, there must be a way to insure that all items, for
n state, are included. For example, if there is an i

B -> $\underline{c}$ is in P, then there must be an item with the production B -> . $\underline{c}$ for that state. The viable formed with the new marked production, will have th prefix as the original item. The process of includi such items is called <u>closing</u> the state. However, i to close a state, it is also necessary to describe propagate lookaheads to the added items. To do this, the function first($\underline{a}$) as follows:

$$\text{first}(\underline{a}) = \{\ a \mid \underline{a} \overset{*}{\Rightarrow} a\underline{c},\ a \in T\}$$

Using the above definition, the closure of a items I (denoted as closure(I)) can be constructed us rules:

i) Every item in I is also in closure(I)

ii) If the item [A -> $\underline{a}$. B$\underline{b}$, LA] is in closure
and B -> $\underline{c}$ in P, a $\in$ LA
then the item [B -> . $\underline{c}$, first($\underline{b}$a)]
closure(I).

<u>example</u> <u>2.1</u> Let the CFG G have the set of produc

S -> A
A -> a A b
A -> $\underline{e}$

where S -> A is the start production. Then the of the item set {[S -> . A , {\$}]} is t {[S -> . A ,{\$}], [A -> . $\underline{e}$, {\$}], [A -> . aAb

The characterisitc automaton G is built from the set
es constructed above with the transitions being gran
ols. The path to a given state will then spell a le
ix for some sentential form*

The algorithm (shown below) starts by setting
ial state to the closure of the start production, t
ng each state just built, determines the transit!
the state as follows:

i) for each grammar symbol X in (N U T) s•t• the i
I*> ~$^>$ £. • 2Lk. > **A] $*^3$ $*^n$ the state, there is a uni
transition, labeled X, to the state containing the i
[A -> aX . b. , LA] obtained by shifting the dot act
the grammar symbol X*

ii) if [A -> a. . , LA] is in the state, then
transition should be produced for that item*

gorithm for constructing the characteristic automaton

put: a CFG G • (N , T , P , S)

tput: a set C, of states, and the function

GOTO : (set of items) x (N U T) ->(set of items),

defines the characteristic automaton.

thod; The two procedures below, initiated by c*

EMS(G);

```
ocedure ITEMS(G);
 begin
    C :» closure((S -> • S',{$}]);
         {where ff$fl is a unique symbol in T which  dt
         the end of the string to parse}
    repeat
       for each set of items I in C, and each gi
       symbol X such that J • GOTO(I,X) is not empl
       J £ C
          djp add J to C;
    until no more sets of items can be added to C
 end;
```

```
unction 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

: the two following forms:

i) [S -> . S′ , {$}]

ii) [A -> b . c , LA] where b $\neq$ e

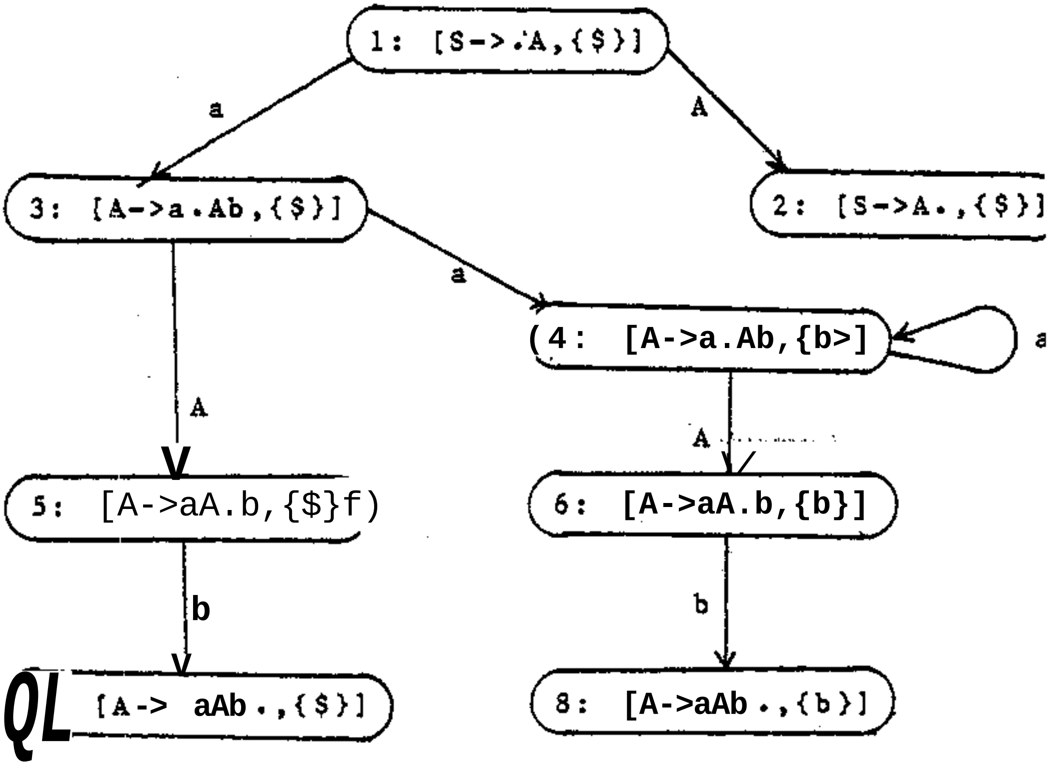
It can be shown that by closing the core of a

e origonal state can be retrieved. Hence, all examp

is paper will only show the core of each state.

xample 2.2 Construction of a, characteristic automaton

et the CFG G be defined by the same set of production s in example 2.1. Then, the LR(l) characteristi utomaton of the grammar G is as follows:



here the transition ars are defined by GOTO

.4 *Construction of LR(1) Parsers*

Using the characteristic automaton, the LR(1) p
n be directly generated. Let an *LR(1) parser* be de
a quintuple M = (K , *action* , *goto* , G , *start*) wh

K is a finite set of *parser* *st*

action : K x T -> {*shift* j | j ∈ K}

U {*reduce* p | p ∈ P} U {*error*}

defines the *parsing action table*;

goto : K x N -> K U {*error*} defines the

parsing goto table;

G is a CFG such that L(G) is the class of

languages to recognize;

and *start* is the *initial state*.

The set of parser states K contains a special s
ept which is the state H, such
ion(H,$) = *reduce* S -> S′. Also, the *action* and
sing tables are enough to define an LR(1) parser.

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

Algorithm for constructing LR(1) parsing tables

Input: The characteristic automaton CG * (C,GOTO)
for a CFG G;

putput t a parsing table (possibly with conflicts
grammar G is not LR(1))

nethod: Let C * {I.$_t$I$_n$, ••• ,1 } be a set of sets of
Erom the characteristic automaton CG* The states
parser will be labelled 1>2, ••• ,n where st
corresponds to the set of items I$_i$• State 1 is the
state. The parsing actions are:

i) If [A ->c_ . ajb_ , LA] 61$_±$ where a S T
GOTO(I$_i$,a) -I$_j$; then action(i,a) * shift j

ii) If [A -> c • ,LA] in I,, then for each a 6 L
action(i»a) * reduce A -> £

iii) All entries of action not defined by the
rules are set to error«

oto transition for state i is constructed using the t

:

i) if GOTO(I_i,A) = I_j, where A is a nonterminal, th

goto(i,A) = j

ii) All other entries of *goto*, not defined by the fir

ule, are set to *error*

xample *2.3* Let the LR(1) characteristic automaton b

efined as in example 2.2. Using the above algorithm

he two parsing tables produced are:

action

	a	b	$
1	shift 3	error	reduce A-
2	error	error	reduce S-
3	shift 4	reduce A->e	error
4	shift 4	reduce A->e	error
5	error	shift 7	error
6	error	shift 8	error
7	error	error	reduce A-
8	error	reduce A->aAb	error

goto

	S	A
1	*error*	2
2	*error*	*error*
3	*error*	5
4	*error*	6
5	*error*	*error*
6	*error*	*error*
7	*error*	*error*
8	*error*	*error*

From the above algorithm, one can tell directly w
FG G does not produce an LR(1) language. This occur
ction is not a function but only a relation, or in
ords, whenever there is more than one possible acti
ome input pair. These multiple entries are kno
onflicts. The two types of conflicts that can exist
hift/*reduce* and ii) *reduce*/*reduce* conflicts, whic
espectively 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 ca
ed for parsing most* programming languages. Unfortuna
e parsers produced for these grammars, using the tr
scribed in the previous chapter, are too large
nsidered useful. Hence, several modifications have
oposed which will reduce the size of the parser prod
is chapter discusses four of these methods. Two of
thods (SLR(l) and LALR(l)) reduce the number of stat
ducing the size of the language accepted. The other
thods (proposed by Pager [Pag77a]) use conditions
rging states of a LR(1) parser while maintaining the
wer to recognize LR(1) languages.

II.1 SLR(1) parsers

The SLR(1) parsing table construction is quite s o that of the LR(1). The main difference is th arser produced is based on a characteristic automato o lookahead (i.e. an LR(0) automaton). implification reduces, in general, the total numb tates created.

To build an SLR(1) parser, redefine an item by re he lookahead set leaving just the marked production. his definition, the rules to close a set of SLR it ecome:

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

ii) If the item $A \rightarrow \underline{a} \cdot B\underline{c}$ is in closure(I),

and $B \rightarrow \underline{b} \in P$

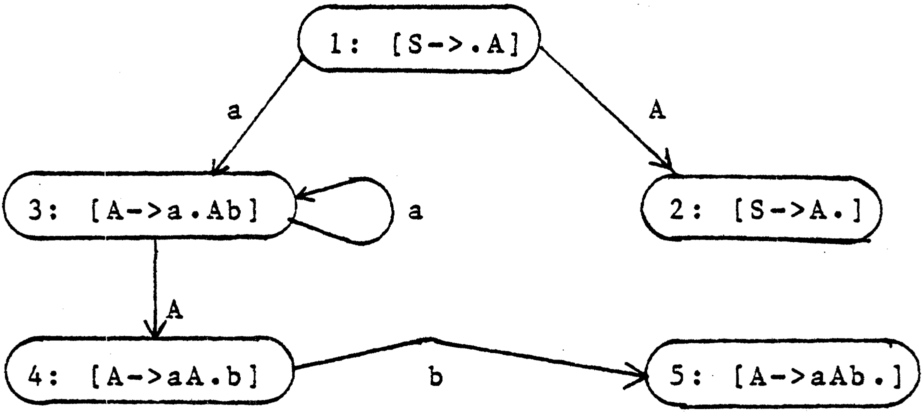
then the item $B \rightarrow \cdot \underline{b}$ is also in closure(I

The procedure to build the characteristic automat lso simplified. These procedures are as follows:

```
unction GOTO(I,jC);
  begin
     let J be the set of items A -> aJC • b. such that
        A -> &. • JLk *s *n * anc* 2[ is a grammar symbol
     return closure(I);
  end;


rocedure ITSMS(G);
  begin
     C :» closure(S -> • 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 $ C
           jio 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
productions in example 2.1. Then an LR
characteristic automaton is:



The SLR(1) method does not use a lookahead set
de what reduction to use once a viable prefix has b
gnized. Instead, it uses a method to approximate
aheads, which in fact guarantees that the set
aheads will be included. This is done by the funct
OW : N -> 2^T which computes all symbols which can fol
ven nonterminal symbol. However, in order to comp
OW, the terminal symbol $ must be included. Hence
definition of FOLLOW, it is assumed that there is
tional production of the form S'' -> S$ where S'' i
erminal and does not appear in any production in
OW is defined as

where a « first(b)}

example 3«2 Using the CFG G described in example

the FOLLOW sets are:

FOLLOW(S) - {$}
FOLLOW(A) - {$,b}

Using the characteristic automaton and the fi

OLLOW the SLR(l) parsing table can be created usd

ollowing algorithm:

LR(1) parsing table construction algorithm

nput: the SLR(l) characteristic automaton CG « (C,GC

for the CFG G.

»utput: a parsing table (possibly with conflicts j

:LR(D)

lethod: Let C » {L , ••• ,1 } be the set of sets of
^ ^

rom the characteristic automaton CG. The states

iarser will be labeled 1,2, ••• ,n where state i corre

:o the set of items I ^{i}• As with LR(1) parsers, *I*

.nitial state be state 1.

The parsing actions are defined as follows:

i) If $A \rightarrow \underline{a} \; . \; b\underline{c} \in I_i$ where $b \in T$ and
$GOTO(I_i, b) = I_j$ then *action*(i,a)=j

ii) If $A \rightarrow \underline{a} \; .$ is in I_i then for each b ∈
set *action*(i,b) = *reduce* $A \rightarrow \underline{a}$

iii) all entries not defined by i) or ii) ar
error

The *goto* transitions are defined by the following

i) If $GOTO(I_i, A) = I_j$
then *goto*(i,A) = j where $A \in N$

ii) all other entries of *goto*, not defined by
set to *error*

example 3.3 Using the LR(0) characteristic au
example 3.1, and the FOLLOW sets in exampl
SLR(1) parser is defined by the following tab

action

	a	b	$
1	shift 3	error	reduce A-
2	error	error	reduce S-
3	shift 3	reduce A->e	reduce A-
4	error	shift 5	error
5	error	reduce A->aAb	reduce A->

goto

	S	A
1	error	2
2	error	error
3	error	4
4	error	error
5	error	error

LALR(1) parsers

A second type of simplification similar to the SLR(
he LALR(l) parser invented by DeRemmer [DeR69]* Ma
ithms for computing LALR(l) parsers have since be
nted [LLH71,AEH72,A&U77,DeR72,Alp76,Pag77b] . The ma
rence from SLR(l) is a concise and more accurate meth
computing the set of lookaheads than the functi
W* The same LR(0) characteristic automaton can be us
nstruct either an LALR(l) of an SLR(l) parser*

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

LA(k,A •-> a)«< t C T | S$ ~~=~~*>- bAc *>- bac

and t * first C^) and the string bji

a prefix for the state k>

example 3*4 Using the CFG g, and the LR(
characteristic automaton, from example 3.3, t
function LA is defined as follows:

LA(1, S->A)-{>	LA(l,A->aAb)»{>	LA(1f A->JB)-<
LA(2,S->A)»<$>	LA(2,A->aAb)»{ >	LA (2 , A->e) -<
LA(3,S->A)»<>	LA(3,A->aAb)»<}	LA (3 , A->.e) *<
LA(4,S->A)»{>	LA(4,A->aAb)*< >	LA(4 , A->.e) «<
LA(5,S->A)»<>	LA(5,A->aAb)»{$,b>	LA(5 , A->,e) *<

he construction of the LALR(1) parser is exactly the s an SLR(1) except that the action function is compu ollows:

i) If $A \to \underline{c} \cdot a\underline{b} \in I_i$ where $a \in T$ and

GOTO(I_i,a)=I_j

then *action*(i,a)=j

ii) If $A \to \underline{a} \cdot$ is in I_i then for each

a ∈ LA(i, A-> A_) set *action*(i,a) = *reduce*

iii) all entries not defined in i) and ii) are se

error

example 3.5 Using the LR(0) characteristic automa example 3.1, and the function LA as defined in e 3.4, the LALR(1) parsing tables are:

action

	a	b	$
1	shift 3	error	reduce A-
2	error	error	reduce S-
3	shift 3	reduce A->je	error
4	error	shift 5	error
5	error	reduce A->aAb	reduce A->

goto

	S	A
1	error	2
2	error	error
3	error	4
4	error	error
5	error	error

The set of languages defined by SLR(l), LALR(l)
1(1), are known to form a hierarchy as follows:

•3 ~~rager s weatc compaciDincy~~

In the previous two sections, restrictions on the c languages were imposed to reduce the number of state LR(1) parser. Pager [Pag77a] shows that the number tes may be reduced without affecting the class guages accepted*

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

```
for each set of items I in C, and each grammar symt
such that GOTO(I,X.) is not empty and J 0 C
    do add J to C;
```

this statement if two states are similar in form, be represented by a single state, and therefore sin ies of a state can be removed. The criterion iding whether two states can be combined is *ce* patibility criterion and the action of combining tes called a merge. For the LR(1) construction, tes are compatible if they are similar in form, that y contain the same set of items. Pager has founc er forms of compatibility which he calls weak and *st* patibility.

Unfortunately, changing the compatibility cri
rom the LR(1) case can cause problems. In particular
wo states satisfy Pager's compatibility criteria, m
he states may necessitate a propagation of lookahe
tates already created, which in turn will modify the
tate which caused the original propagation. However,
roblems can be resolved using the following algorithm

Algorithm for constructing an LR compatible characteristic automaton

nput: a CGF G and a compatibility function compatibl

utput: a set C, of states, and the function
OTO : (set of items) x (N U T) -> (set of items), whi
efines the characteristic automaton.

ethod: the three procedures below, initiated by call
TEMS'(G);

```
unction 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;

rocedure 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;
```

```
    {merges $S_1$ into $S_2$ and updates C accordingly}
  begin
    S :» merge($S_1$,$S_2$);
    if S. C S
       then
          replace the items of state S^ in C
             by the items of S;
          for each grammar symbol X,
                such that GOTO($S_2$,j[) already define*
             do insert(closure(GOTO($S_9$X)),
                        GOTO($S_2$,X),C)
             SA
    £1
  end;
```

Two states can be merged if and only if they ha
>ame set of marked productions in their respecti^
•art. Under this condition, the compatibility criter:
:hat merging the states (and therefore the lookahea<
fill not introduce any R/R conflicts in the resulting
inless the language is in fact not LR(1)» Foi
compatibility, the test is solely based on the two
>eing merged, while strong compatibility also uses i
)f productions of the CFG associated with the LR(1)]

Let the function merge be defined as follows:

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

$[A \rightarrow \underline{a} \cdot \underline{b} , LA_1] \in S_1$

$[A \rightarrow \underline{a} \cdot \underline{b} , LA_2] \in S_2$

and for all items $[A \rightarrow \underline{a} \cdot \underline{b} , LA_1] \in S_1$

there exists an item $[A \rightarrow \underline{a} \cdot \underline{b} , LA_2] \in S_2$

for all items $[A \rightarrow \underline{a} \cdot \underline{b} , LA_2] \in S_2$

there exists an item $[A \rightarrow \underline{a} \cdot \underline{b} , LA_1] \in S_1\}$

en, according to Pager's definition, two states S_1 ar

e weakly compatible if

i) S_1 and S_2 only have common marked productions i

their item part. That is, if $[A \rightarrow \underline{a} \cdot \underline{b} , LA_1] \in$

then there exists an item $[A \rightarrow \underline{a} \cdot \underline{b} , LA_2]$ e S_2

if item $[A \rightarrow \underline{a} \cdot \underline{b} , LA_2] \in S_2$ then there exists

item $[A \rightarrow \underline{a} \cdot \underline{b} , LA_1] \in S_1$

ii) for each pair of items $[A \rightarrow \underline{a} \cdot \underline{b} , LA_1] \in S_1$

$[B \rightarrow \underline{c} \cdot \underline{d} , LA_2] \in S_2$, 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

$[B \rightarrow \underline{c} \cdot \underline{d} , LA_1'] \in S_1$ such that

$LA_1 \cap LA_1' \neq \emptyset$

$[A \rightarrow \underline{a} \cdot \underline{b} , LA_2'] \in S_2$ such that

$LA_2 \cap LA_2' \neq \emptyset$

Condition a) states that if there are no items b he states which have a common lookahead symbol, th erge can not produce any conflicts, and in particula ot produce a R/R conflict. (Note: it is also impo o introduce S/R conflicts since the states will be nly if they have common marked productions. Therefo esult of merging would only produce a S/R conflict xisted in one of the unmerged states before mergin ondition b) and c) the set of conditions is:

$[A \rightarrow \underline{a} \cdot \underline{b} , LA_1],[B \rightarrow \underline{c} \cdot \underline{d} , LA_1'] \in S_1$

$[A \rightarrow \underline{a} \cdot \underline{b} , LA_2],[B \rightarrow \underline{c} \cdot \underline{d} , LA_2'] \in S_2$

$LA_1 \cap LA_2 \neq \emptyset$ and either $LA_1 \cap LA_1' \neq \emptyset$ or

$LA_2 \cap LA_2' \neq \emptyset$

Since $LA_1 \cap LA_2 \neq \emptyset$, the only possible conflict /R conflict arising from merging the lookaheads roductions $A \rightarrow \underline{ab}$ and $B \rightarrow \underline{cd}$. However, this can nly if $\underline{b} \overset{+}{\Rightarrow}_R \underline{w}$ and $\underline{d} \overset{+}{\Rightarrow}_R \underline{w}$, producing a common s here both productions will be reducible. By conditi $A_1 \cap LA_1' \neq \emptyset$, if in addition $\underline{b} \overset{+}{\Rightarrow}_R \underline{w}$ and $\underline{d} \overset{+}{\Rightarrow}_R \underline{w}$, hen there must already exist a state with a R/R confl ome symbol $a \in LA_1 \cap LA_1'$. Similarly for conditi ence, if the language is indeed LR(1), then it must

that JD $^+$*>_ w; d^ $^+$»>_ w' ; w>w' S T* ; and w + w' ,
efore conditions a),b) and c) are sufficient to inss conflicts will be produced if the language generated grammar is indeed LR(1)«

For example, let a CFG be defined with the set uctions in figure 3»1* The LR(1) characteris maton contains 38 states (shown in part in figure 3* r weak compatibility, states 8 and 12 can not be mer e the items [X->a*AE,{d}] S 12 and [Y->a#B$_f${d}] S 8 h common lookahead symbol d« However, for example, *ata* nd 33 are in fact weakly compatible.

It can be shown that the size of a weak compata) parsing table will contain a number of states that where between that of LALR(l) and LR(1) parsing table

S -> S'
S' -> aZa
S' -> bZc
Z -> aC
C -> aDF

S' -> aXb
S' -> bXd
X -> aAE
A -> aDF
D -> d
F -> *e*

S' -> aYd
S' -> bYa
Y -> aB
B -> b
E -> e

figure 3.1

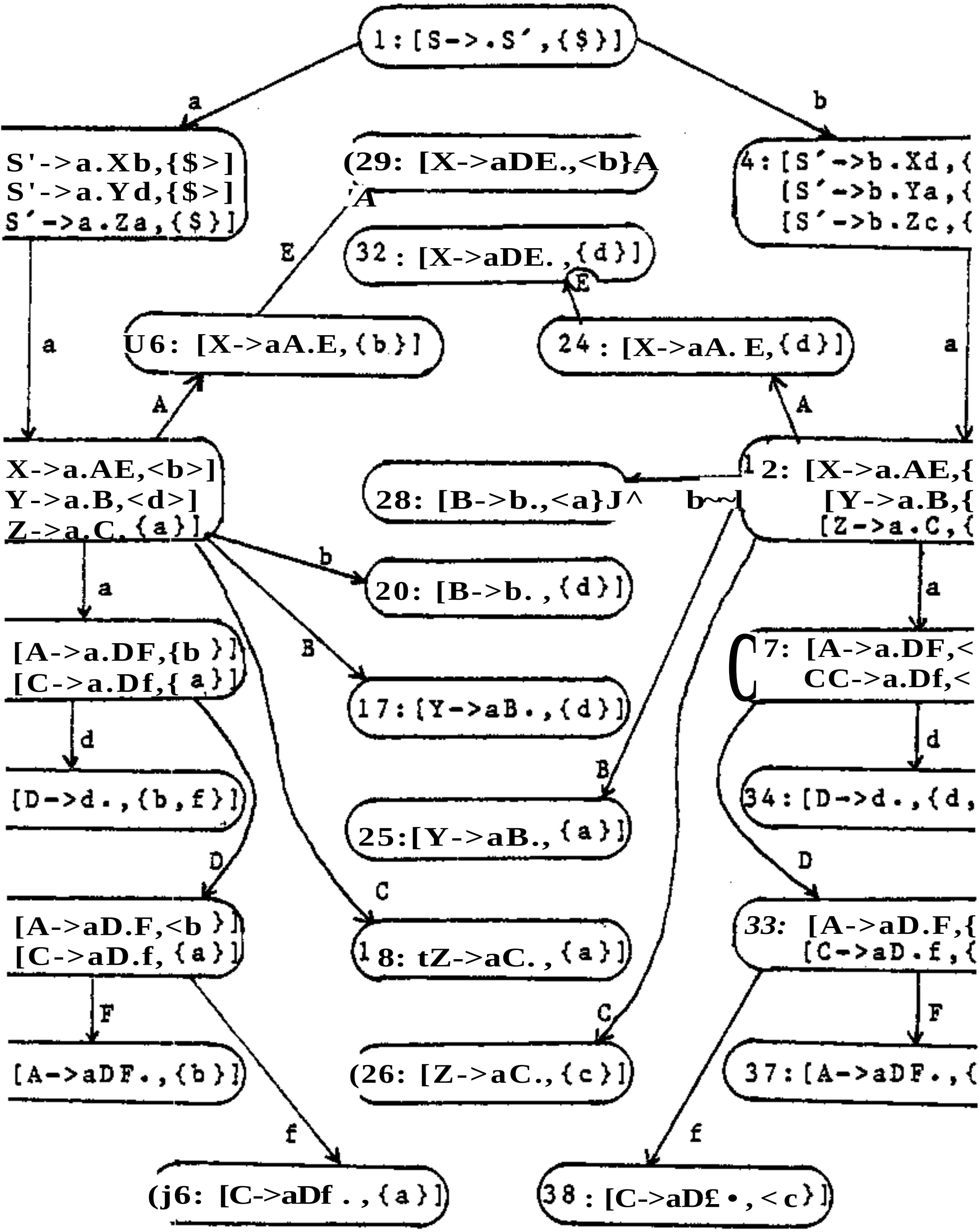

1:[S->.S´,{$}]
a
b
S'->a.Xb,{$>]
S'->a.Yd,{$>]
S´->a.Za,{$}]
(29: [X->aDE.,<b}A
'A
4:[S´->b.Xd,{
[S´->b.Ya,{
[S´->b.Zc,{
E
32 : [X->aDE. ,{d}]
E
a
U6: [X->aA.E,{b}]
24 : [X->aA. E,{d}]
a
A
A
X->a.AE,<b>]
Y->a.B,<d>]
Z->a.C,{a}]
28: [B->b.,<a}J^
b~~l
12: [X->a.AE,{
[Y->a.B,{
[Z->a.C,{
b
20: [B->b. ,{d}]
a
a
B
[A->a.DF,{b }]
[C->a.Df,{ a}]
17:{Y->aB.,{d}]
C7: [A->a.DF,<
CC->a.Df,<
d
d
B
[D->d.,{b,f}]
25:[Y->aB.,{a}]
34:[D->d.,{d,
D
D
C
[A->aD.F,<b }]
[C->aD.f,{a}]
18: tZ->aC. ,{a}]
33: [A->aD.F,{
[C->aD.f,{
F
C
F
[A->aDF.,{b}]
(26: [Z->aC.,{c}]
37:[A->aDF.,{
f
f
(j6: [C->aDf . ,{a}]
38 : [C->aD£ • , < c}]

figure 3.2

III.4 Strong compatibility

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

Strong compatibility requires that no two sta
merged if they have a common descendant in th
characteristic automaton which will introduce R/R co
when the two states are merged.

For example, the grammar presented by figu
creates (in part) the LR(1) characteristic autom
figure 3.2. States 8 and 12 are not weakly com
because the items [X->a.AE,{d}] ∈ 12 and [Y->a.B,{
have a common lookahead symbol "d". If these two sta
merged (and hence causing merges of states (20,28), (
(17,25), (16,24), (29,32), (31,34), (30,33), (19,27),
and (35,37) where each pair are common descendan
resulting states of the automaton would have no con
Hence these two states, according to Pager's definiti
in fact strongly compatible.

On the other hand, let the grammar be that of figt which creates (in part) the LR(1) characterisi laton in figure 3.4. Merging states 7 and 10 (and hei .ng common descendants 14 and 18 to be merged) woi .t in two R/R conflicts on the symbols "a" and "b" tescendant state. Hence these states will not be merj 1 strong compatibility.

```
S  -> S'         S'-> aXd         S'-> bXa
S'-> aYa        S'-> bYb         X  -> aB
Y  -> ab        B  -> b
```

<u>figure</u> 3.3

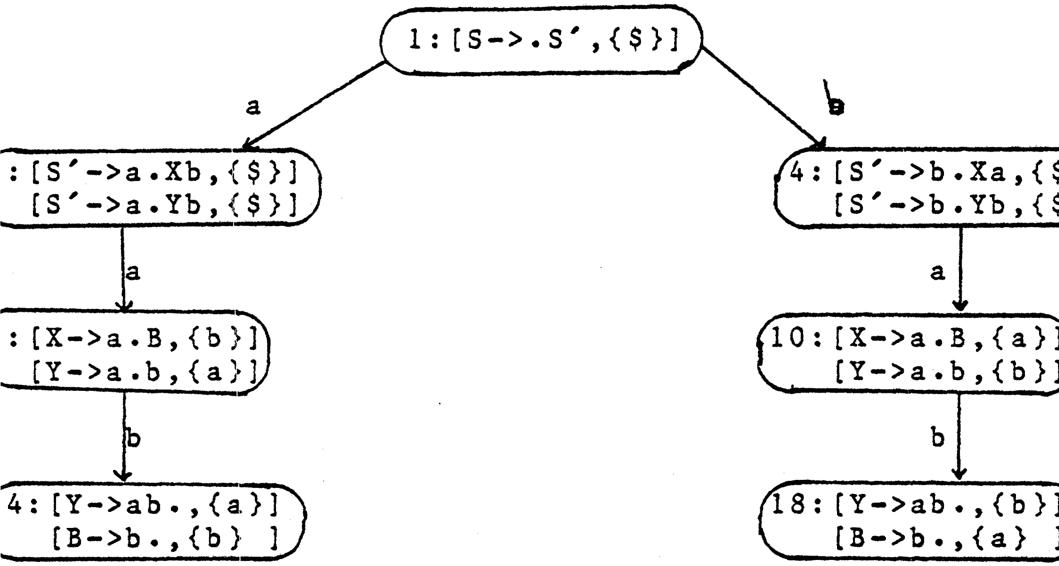


figure 3.4

The way in which two items (from different states roduce a common state with a R/R conflict is if tw an derive the same substring. That is, if the two $_1$ and S_2 are to be merged such that there exists tw A -> $\underline{a}$. $\underline{b}$, LA_1] ∈ S_1 and [B -> $\underline{c}$. $\underline{d}$, LA_2] ∈ S_2 ∈ $LA_1 \cap LA_2$; $\underline{b} \overset{+}{=}>_R \underline{w}$ and $\underline{d} \overset{+}{=}>_R \underline{w}$, then the two ave common descendants such that a merge will introd onflicts.

4) could not be merged is that the items [X->a.B,{d} l [Y->a.b,{d}] $\in$ 10 have a common lookahead symbol d, e strings B and b both rewrite to the string b.

The search for a common substring between two st en necessary to try all possible combinations of rewr volves as much work as building all descendant st wever, it is not necessary to expand all pos mbinations of rewrite rules. This fact can be see derstanding how expansion of the nonterminals is perf building the characteristic automaton. That is, whe em [A -> $\underline{a}$. X$\underline{b}$, LA] is closed, where X -> $\underline{c}$ in $\in$ LA, it will create the item [X -> . $\underline{c}$, first($\underline{b}$d)]. $\overset{*}{=}>_R$ $\underline{e}$, it is clear that the elements in the lookahea will be propagated to the new item. On the other $\underline{b}$ $\overset{*}{\neq}>_R$ $\underline{e}$, the definition of the function first indi at any element d $\in$ LA is not in first($\underline{b}$d). Hence, in se, the lookaheads defined by first($\underline{b}$d) are independe and does not effect states derived from the new ated differently, the only rewrites that shoul rformed are those which are applied to the nonterm ich occur at the end of marked productions. striction on the number of possible derivations to , is what Pager calls a strong rightmost deriv enoted $=>_{SR}$) and is defined as:

i) $\underline{c} = \underline{e}$

ii) $\underline{a}B\underline{c} \Rightarrow_R \underline{abc}$

Pager has derived a procedure[Pag77a] which chec wo items, having a common lookahead symbol, will pro hared descendant containing a R/R conflict. The eels that the algorithm presented by Pager is opac ell as slightly incorrect, and that the algorithm in aper (see page 49) has been corrected and modif larify its nature.

The algorithm is presented using two co-rec rocedures which tries all possible strong rig erivations to see if the two given marked productions common descendant state where two different produ ill be reduced (since this is the only way that a onflict can be produced). The procedure CHECK loo rivial cases (i.e. cases where no rewrites are nec o determine the result) while the procedure nontrivia hecks those cases requiring rewrites in order to det he wanted criteria.

One possibility that procedure CHECK handles is s impossible for two items, with or without rewrit roduce a common descendant. That is, let (1) A -> nd (2) B -> $\underline{c}$.$\underline{bYg}$ be two marked productions where

ii) X ± Y * £

iii) I,& ?>$_R$ e.

ume that these two marked productions can derive a co string which will produce a R/R conflict* Then it

the case that Xf $\overset{*}{»>}$_ w and Yg $\overset{*}{»>}$- w . Since both f a not derive e, the lookaheads can not propagate throu £• But then, by the way LR(1) parsers are genera string derived from *X* will be reduced to *X* be nning the string derived from £• Hence any *st* ived from X£ must be of the form *]££.* Similarly, ing derived from jf£ must be of the form Y£« Theref ce ^ r X> ^^{c} *-^{s} impossible for any items of this fox duce a common substring (and hence a common descend ch will produce R/R conflicts.

The second trivial check in the procedure CHECK, is two marked productions immediately indicate a cc cendant which will produce R/R conflicts if merged, if the two items are of the form (1) A -> a^bjnf£ anc -> ji.bJCZj^ where

i) ItA *>$_R$ A

ii) X G (N 0 T) and X ?>- e,

iii) W,Z S N and W,Z $\overset{*}{->}$,. e

is clear, under the above conditions, that the closui : items (3) [A -> abX . W£ , LA$_1$3 and

abX . Zg , LA_2] will produce the items (
. e , Q] and (6) [Z -> . e , Q] where Q = $LA_1 \cap LA$
this case will produce a common descendant whe
icts will be produced.

In all other cases, some rewriting is necessary a
dure nontrivialcheck is called to handle these cases.

One possibility, that requires rewriting, is when t
marked productions are of the form (1) A->a.bXf and (
bYg where

i) X ∈ N and $X \overset{*}{\Rightarrow}_R e$

ii) $f \overset{*}{\Rightarrow}_R e$

iii) Y ∈ (N U T); $Yg \overset{*}{\not\Rightarrow}_R e$ and Y ≠ X

is case, X must rewrite to some string derivable fr
n order to produce a common string (and hence a comm
ndant). However, this the same as testing if the
s a production X -> h where h ≠ e such that the ite
and B->cb.Yg will share a common descendant which c
ce R/R conflicts.

A second possibility handled in nontrivialcheck a
of the form (1) A->a.bXf and (2) B->c.bZg where

ii) Z 6 (N U T); Z& *>$_R$ e, and X ≠ Z

iii) f *»>$_R$ e

iv) no production X->]i, where h.^> exists such

X->.Ji and B->j2j£.Zjz will have a common descendant

this case,, because of condition iv) and that X^j£,

mon string derivable from Xj[must be of the form X£ x*

common string derivable from Z& must be of the form

this implies that they can not derive the same st

hence can not have a shared descendant.

The last possibility checked checked by the proce

trivialcheck is the case wb<sn the marked productions

the form (1) A->jL.bX and (?> 5->c«bY where X,Y 6 N

• The only way that t· ·,^o two marked productions

ive a common descendant xs if X *>- w and Y *>

ever, this is the sama as testing if there exists

ductions of the form X->£ and Y->£ such that either

ked productions A->a.b.X aid Y->.£, or X->.£ and B->^c

1 produce a common descendant which can contain an

flict from merging.

For efficiency, the procedure nontrivialcheck us*

cial global function

tried : N x (marked productions) -> boolean,

ore the top call to procedure CHECK is made, the func

set to false for all possible inputs, and it will re alse the first time it is called with any given Eter that, anytime the function is again called wi ame set of arguments, it will return true. Therefore mction will prevent the procedure nontrivialchec lecking if a nonterminal will rewrite to match articular marked item.

Finally, it is assumed that on the top level ca 3ECK(A -> a. ._a/ , B ->_ b. _. V) the following Dnditions hold:

i) A -> a, . at' + B -> ɪD . b.'

ii) jLa' + e. and bjb' ,' <e

Co-recursive procedures to check for a shared descendant

```
procedure check(A -> $\underline{a}$ . $a_1 a_2 ... a_n$,
                B -> $\underline{b}$ . $b_1 b_2 ... b_m$ ) : boolean;
{note:  $a_i$,$b_i$ $\in$ (N U T);  A,B $\in$ N;  $\underline{a}$,$\underline{b}$ $\in$ (N U T)$^*$
begin
    s:= maximum i s.t.  $a_i a_{i+1} ... a_n \overset{*}{\not\Rightarrow}_R \underline{e}$;
    t:= maximum i s.t.  $b_i b_{i+1} ... b_m \overset{*}{\not\Rightarrow}_R \underline{e}$;
    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 -> $\underline{b}$ . $b_1 b_2 ... b_m$,t
                                A -> $\underline{a}$ . $a_1 a_2 ... a_n$,s,match
                else check:=nontrivialcheck(
                                A -> $\underline{a}$ . $a_1 a_2 ... a_n$,s
                                B -> $\underline{b}$ . $b_1 b_2 ... b_m$,t,match
end;
procedure nontrivialcheck(A -> a • $a_1$^$a_2$^.$a_n$,s,
                            B -> b • b.$b_o$.•.$b_m$ ,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 (a  6 N) or
            not tried($a_s$ ,B -> bb$_1$*..b$_{s-1}$ • b$_s$ ...b$_m$)
            then
                for each production C -> c^ 6 P
                    s.t. a $^a$C, ^c £ £, and
                    C -> . c ≠
                        B -> bb$_1$««*b$_{s-1}$ • b$_s$•••b$_m$
                do
                    if check(C ->•€.,
                            B -> bb$_1$»««b$_{s-1}$ • b$_s$•••b$_n$)
                        then
                            nontrivialcheck:*true;
                            terminate:»true
                    fi
```

```
      else if (s=t) and (match-1=s) and $b_t \in N$
            and check($B \to \underline{b}b_1 \ldots b_{s-1} \cdot b_s \ldots b_n$,
                  $A \to \underline{a}a_1 \ldots a_{s-1} \cdot a_s \ldots a_n$)
         then
            nontrivialcheck:=true;  terminate:=true
      fi;
      s:=s+1;
   until terminate;
end;
```

Using the above, two states S_1 and S_2 ar
compatible if

i) If the item $[A \to \underline{a} \cdot \underline{b}, LA_1] \in S_1$ then ther
an item $[A \to \underline{a} \cdot \underline{b}, LA_2] \in S_2$ and if
$[A \to \underline{a} \cdot \underline{b}, LA_2] \in S_2$ then there exists
$[A \to \underline{a} \cdot \underline{b}, LA_1] \in S_1$

ii) for each quadruple of items
$[A \to \underline{a} \cdot \underline{b}, LA_1], [B \to \underline{c} \cdot \underline{d}, LA_1'] \in S_1$,
$[A \to \underline{a} \cdot \underline{b}, LA_2], [B \to \underline{c} \cdot \underline{d}, LA_2'] \in S_2$
either

a) weak compatibility between the items h

b) $\underline{b}$ and $\underline{d}$ do not share a descendant.

Chapter IV

An Error Recovery Method for LR Parsers

In the previous two chapters, five dif onstructions were discussed, all of which produ arsers. The downfall of all LR parsers is that the esigned only to decide if the given input is legal s, belongs to the language generated by its grammar. auses the unfortunate result that when such a par sed in a compiler, once the first illegal terminal s found, the parse stops with failure. However, it e more desirable to have the parse report as dditional errors as possible.

Several people have proposed various error re chemes for LR p G&R75,D&R77,P&D79,O'H76,Pen77,P&D78]. This chapter nly deal with one such method, which is a modificat algorithm presented here differs from thiers in tha incorporated into the LR parser and does not attempt correction.

In order to describe error recovery, we first d how an LR parser works. Let a *path* be a sequence of $q_0q_1\ldots q_n$ such that for each state q_i, one of the fo conditions hold:

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

ii) *action*$(q_i,a) = q_{i+1}$ for some $a \in T$.

A path will be denoted as $[q_0:\underline{a}]$. That is, if $\underline{a} = a_1$ where $a_i \in (N \cup T)$ then the path $[q_0:\underline{a}]$ is the sequ states such that either *action*$(q_{i-1},a_i) = q$ *goto*$(q_{i-1},a_i) = q_i$. Also, let the result of the f top : path -> state be defined as the state q_n whe path is $q_0q_1\ldots q_n$. Finally, whenever the path $[q:\underline{a}]$ from the start state (of the LR parser) it will sim denoted as $[\underline{a}]$.

The basic control of a LR parser can be defined decision function df : path x T -> (path U{*reject*,*a* as follows:

i) df($[\underline{a}]$,b) = $[\underline{a}b]$ if *action*(top($[\underline{a}]$),b) = *shif* some state j ∈ K.

ii) df([aw],b) = df([aA],b)

if action(top([aw]),b) = reduce A

aw ≠ S when b = $

iii) df([S],$) = accept

if action(top([S]),$) = reduce S -> S′

iv) df is defined as reject for all pairs

([a],b) not defined by rules i) throug

The algorithm to implement the above decisi

is simply as follows:

```
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;
```

i that the variable path is implicitly used as a s :h holds the prefix of sentential forms being recogn :he parser.

The error recovery strategy describes what to do if 5e of an input results in <u>reject</u>. As can be seen previous algorithm, LR parsers have the nice prop : they stop reading input immediately after the i Lng is found to be illegal. The best recovery from error would be if the parse could somehow be resta i that all other errors made in the input could be pi Unfortunately, this strategy is really unfeasible s carries the implicit assumption of knowing what ter meant when he wrote the string to be parsed.

A much more conservative approach is to only state aining substrings of the input are impossible accor the given grammar. That is, if the remaining input a error Is. a string <u>w</u> € T and there doesn't exi

*

htmost derivation such that S «>. awe for

J\ ———

(N U T) and £ 6 T , then the substring $\overline{w}$ shoul orted as an error.

rur example, cuasiutr uie two paeuau rdo^AL pro<

<stmt> -> FOR <var> :* <exp> TO <exp> DO <stmt>

<stmt> -> WHILE <exp> DO <stmt>

with the erroneous input

FOR X:-l 5 DO BEGIN J:»X; L:-X END;

where the terminal symbol "TO^{11} has accidently been 1< Using an LR parse, parsing would stop after rea< symbol ^{lf}5^{lf}. As one Looks for subsequent errors, it : that "5" is a valid substring derivable from S* It clear that 5 can occur at the following points in thi productions

<stmt> -> FOR <var> :* ^{lf}<exp>^{tf} TO <exp> DO <stm

<stmt> -> FOR <var> ;* <exp> TO ^{fl}<exp>^{tf} DO <stm

<stmt> -> WHILE ^{fl}<exp>^{1f} DO <stmt>

By expanding the substring to include the next input the next possible substring to test would be "5 DO^{11} the number of possible positions of this string h reduced to

<stmt> •-> FOR <var> : * <exp> TO "<exp> DO^{11} <stm

<stmt> ·-> WHILE ^{ft}<exp> DO^{11} <stmt>

Continuing this process, it is clear that the subst DO BEGIN J:*X; L:*X END" can correspond to the f positions in the productions:

<stmt> -> FOR <var> :* <exp> TO "<exp> DO <stmt

<stmt> «•> WHILE ^{ff}<exp> DO <stmt>$^{!l}$

g implies that a reduction should be performed by o
e above productions. One possibility is to take t
g recognized before the reject point, and to either a
lete symbols to produce a match and therefore deci
reduction to choose. This type of error recovery
act the error correction method used by [P&D79
er, the one chosen by the author assumes that t
ring "5 DO BEGIN J:=X; L:=X END" is the maxim
ministic string that could be recognized, and hen
e it from further consideration. That is, it wi
rt the parse starting with the semicolon.

The above example in fact characterizes the err
ery method described in this chapter. To state t
d more explicitly, let me start by defining an err
as a set of LR parser states, where each error sta
ins the set of LR parser states that the parse might
The restart state as a special error state containi
he LR parser states.

The first shift, in error recovery, is a forced shi
gh the illegal terminal symbol that produced t
tion. This shift can be viewed as a parallel shift,
error symbol a, from all LR parser states I in t
rt state to all states J such that action(I,a) = J.
then try to parse the input where the parse will star

ter the forced shift through the illegal symbol. I
e way, any of these parses produce an error, it wi
opped from further consideration for simultaneous pa

One possible result of the above process is tha
rses will be dropped from the set of simultaneous p
der this condition, it is clear that there i
rivation such that S $\overset{*}{=>}_R$ <u>awc</u> for the parsed in
nce, it is quite legal to assume that the next
mbol input can not occur, and report it as an
nce this is an error, the algorithm will then resta
covery method on the next input symbol. Note th
rst action on any error is a forced shift. This is
guarantee that the remaining input is parsed.
ror recovery should not continue if the illegal te
mbol was the end of string marker $.

The second problem is that if the above error re
ocess is to be merged into the LR parser, the pa
rses have to be made deterministic. There is no p
th the <u>action</u> function for a set of states, if the
r all possible inputs is a shift entry. In this ca
clear that the action is deterministic, since res
ates can be lumped into a new set of states and
eating a new error state. The same is true for th
nction. Therefore, nondeterminism can only occur

ion, for a set of states to be simultaneously par
tain either

i) shifts and reductions for the same input symbol

ii) reductions for different productions for the input symbol (as shown in the previous example)

ortunately, neither of these cases seem to be resolv erministically. If, in either case, the parse owed to continue and the next action was performed, ult would produce two different paths. That is, ve two conditions would result in disjoint senten fixes• Such conditions will be called overdefi ever, some decision still has to be made so that aining input can be parsed. Again, the conserva roach was taken* Whenever the input string being pa omes overdefined, the parser assumes that it is imal substring it can recognize, and restarts the *a or recovery process on the next input symbol.

By merging the error-recovery into the LR parser, a parser with error recovery c n be built* If a sing table is the t (K ,actian , goto , G , start) , then let the ser with error recovery be defined as the t » (K , K' , action , goto , G , s tart , init-ei re

K,G, and *start* are defined as in M,

K′ is a set of new states called error recovery

init-error is a state in K′ denoted as the

state of the error recovery method

goto : (K U K′) x N -> K U K′ U {*error*}

action : (K U K′) x T ->

{*shift* k | k $\in$ K} U {*error*,*overde*

{*reduce* p | p $\in$ P}

Furthermore, the *init-error* state will be so defin for each b $\in$ T, *action*(*init-error*,b) = *shift* j f state j. Each recovery state is a set of parsing st K, such that it is the set of states that c simultaneously for the input string being parsed.

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

Construction of LR parser with error recovery

input: LR parsing table M = (K,*action*,*goto*,G,*start*)

output: LR parsing table M′ = (K , K′ ,*action* ,*got*
start , *init-error*)

```
nethod:
  begin
    {initialize state init-error}
    set K′ to the single state containing the set {
    and label it as init-error;
    for each a ∈ T do
      let s be the set
        {j ∈ K | action(i,a) = shift j
                     for all i ∈ init-error};
      if s is a singleton
        then set s′ to the element of s
      else if s ∈ K′
        then set s′ to that state in K′
        else add s to K′ and label the new state
      fi
      set action(init-error,a) = s′
    od
```

```
for each $X \in N$ do
    let s be the set
        $\{j \in K \mid goto(t,X) = j$
                for all $t \in init\text{-}error\}$;
    if s is empty
        then set goto(init-error,X) = 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′ cont
                s
                else add s to K′, and set s′ to its
            fi
            set goto(init-error,X) = s′;
    fi
od
{build each general error state}
repeat
    for each state $i \in K'$ such that the parsing
    for that state is still undefined do
        for each $a \in T$ do
```

```
      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 th
            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
```

```
if there exists two states $S_1, S_2 \in i$ s.t.
        [A -> a .  , $LA_1$] $\in S_1$ where a $\in LA_1$
        [B -> c . d , $LA_2$] $\in S_2$
            where first(d) = a
    then set action(i,a) = overdefined
else if there exists two states
        $S_1, S_2 \in i$ s.t.
        [A -> a .  ,$LA_1$] $\in S_1$
        [B -> b .  ,$LA_2$] $\in S_2$
        where a $\in LA_1$   $LA_2$ and A->a $\neq$ B->b
    then set action(i,a) = overdefined
else if there exists a state s $\in$ i s.t.
        [A -> w . ,LA] $\in$ s where a $\in$ LA
    then set action(i,a) = reduce A -> w
else
    let s be the set
    {j $\in$ K | action(t,a) = shift j
    for all t $\in$ i};
```

```
                  if s is a singleton
                     then set s' to the element in s
                  else if s 8 K'
                     then set s' to the state in K'
                     containing s
                     else add s to K', and set s' to
                     label
                  Hi
                  set goto(i»X) » s'
            fj,
         od.
      od
   until no more states can be added to K'
end
```

Using the resulting LR parser with error recovery, sic control can be handled using the decision fun ' : path x T -> path as follows:

i) df'([q:aj,b) » [q:£b]
when action(top([q:a])>b) * shift j for some
j S (K U K')

ii) df′([q:*aw*],b) = df′([q:*a*A],b)

when *action*(top([q:*aw*]),b) = *reduce* A -> *w*, and

aw = S then b ≠ $

iii) df′([*init-error*:*w*],b) = df′([*init-error*:A],b)

when *action*(top([*init-error*:*w*]),b)

= *reduce* A -> *aw*,

where *a* ≠ *e* and b ≠ $

iv) df′([S],$) = *accept*

v) df′([*init-error*:S],$) = *R*eject

if *action*(top([*init-error*:S]),$) = *accept* or

overdefined

vi) df′([q:*a*],$) = *reject*

when *action*(top([q:*a*]),$) = *error*

vii) df′([*init-error*:*a*],b) = [*init-error*,b]

where b ≠ $, and

action(top([*init-error*,*a*]),b) = *overdefined*

viii) df′([q:*a*],b) = [*init-error*:b]

where b ≠ $ and *action*(top([q:*a*]),b) = *error*

that cases vi) or viii) represent that an error *h*
found in the string being parsed. Hence, any err
ges produced are produced at these points.

Finally, an LR parser with error recovery can
mented simply by calling the procedure parse% using *c*
e decision function.

Chapter V

Implementation

This chapter discusses two programs* The first prc
ates an SLR(l) parser, with error recovery. The se
gram creates either an LR(1), LALR(l), weakly compat
a strongly compatible LR parser. The first sec
cusses the representation of the parsing tables built
h programs. The second section describes
lementation of the SLR(l) parser constuctor and how
tern is used while the third section does the same fot
ond parser constructor.

The representation of the parsing tables nat
uggest using arrays. For uniformity of both acce
alues held in the arrays, all terminal symbols, nonte
ymbols, and productions are provided with an interna
f integers by both programs. For terminal symbols
odes are defined by the set

{i | $0 \le i \le n$ where n is the number of distinct term
symbols occurring in the productions}

here 0 is reserved for the special terminal symb
onterminal symbols are encoded using the set

{i | $-m \le i \le -1$ where m is the number of distinct
nonterminals occurring in the productions}

here the start symbol S will always be given the cod
roductions are coded using the set

{i | $1 \le i \le p$ where p is the number of productions
in the grammar}

here the production S -> S' is always given the code

In representing the *action* and *goto* functions,
on-*error* values are kept internally since the vast ma
f the function values are in fact *error*. The rem
alues are saved in groups, one for for each state,
tates having the same set of non-*error* values wi
epresented by a single copy of the groups.

For example, the grammar

```
S -> E               T -> F
E -> E * T           F -> id
E -> T               F -> ( E )
T -> T + F
```

i produce the following SLR(l) parsing tables:

Action table

	$	*	+	id	(	)
1				S 3	S 2	
2				S 3	S 2	
3	F->id	F->id	F->id			
4	T->F	T->F	T->F			
5	E->T	E->T	S 8			E->
6	S->E	S 9				
7		S 9				S
8				S 3	S 2	
9				S 3	S 2	
0	F->(E)	F->(E)	F->(E)			F->(
1	T->T+F	T->T+F	T->T+F			T->T
2	E->E*T	E->E*T	S 8			E->E
3		S 9	S 8	S 3	S 2	S
4	0	0	0	S 3	S 2	0
5	S->E	S 9				S
6	0	0	S 8			S
7	0	0	0			

where *shift* j is represented by S j,
reduce p is represented by p,
overdefined is represented by 0, and
error is omitted.

	S	E	T	F
1		6	5	4
2		7	5	4
3				
4				
5				
6				
7				
8				11
9			12	4
10				
11				
12				
13				
14		15	16	17
15				
16				
17				

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

By elimination of the *error* values, 58.8% of the tables does not need to be saved. Also, states 1,2, in the previous *action* table all have the same same values.

Each non-*error* value of the *action* table wil

presented as follows:

i) *action*(i,a) = *shift* j will be represented

by the pair (x,j) where x is the code of

terminal symbol a.

ii) *action*(i,a) = *reduce* A -> *w* will be represente

by the pair (x,-p) where x is the code of

terminal symbol a and p is the code c

production A -> *w*.

iii) *action*(i,a) = *overdefined* will be representec

by the pair (x,0) where x is the of the ter

symbol a.

The non-*error* values of the *goto* table, for

ate i, will be represented as the pair (x,j)

to(i,A) = j and x is the code of the nonterminal A.

For efficiency in retrieving the values from the *a*

d *goto* tables the integer pairs corresponing to each

e sorted using the relation $\underline{<}'$ where (a,b) ≤/ (c,d) iff either a<c, or a»c and b<d.

Four integer arrays are used to represent the valv e two parsing tables. The array parsetable is a ray, for some n, which holds all of the non-error A the two parsing tables. The arrays actionlis tolist are s x 2 arrays, where s is the number of ates, and are used to define where the values c tion and goto functions are saved in the array parset ch element in these two arrays is the pair (b,t) wt the starting position of the values saved for that ile t is the number of non-error values of the fut r that state. The last array productionlist is a ray where p is the number of productions and, foi oduction A -> *r it holds the pair (x,|w;|) where x is de of A and Iw| is the length of the string w.

Returning to the previous example, let the codes c rminals, nonterminals, and productions be as follows:

terminals	
$	0
*	1
+	2
id	3
(	4
)	5

nonterminals	
S	-1
E	-2
T	-3
F	-4

productions	
1	S->E
2	E->E*T
3	E->T
4	T->T+F
5	T->F
6	F->id
7	F->(E)

ally be represented as follows:

ctionlist
1:2
1:2
9:4
13:4
17:4
21:2
23:2
1:2
1:2
28:4
32:4
36:4
40:5
45:6
54:3
57:4
61:4

roductionlist
-1:1
-2:3
-2:1
-3:3
-3:1
-4:1
-4:3

gotolist	
1	3:3
2	6:3
3	13:0
4	17:0
5	21:0
6	23:0
7	25:0
8	25:1
9	26:2
10	32:0
11	36:0
12	40:0
13	45:0
14	51:3
15	57:0
16	61:0
17	65:0

parsetable			
1	3:3	34	2:-4
2	4:2	35	5:-4
3	-4:4	36	0:-2
4	-3:5	37	1:-2
5	-2:6	38	2:8
6	-4:4	39	5:-2
7	-3:5	40	1:9
8	-2:7	41	2:8
9	0:-6	42	3:3
10	1:-6	43	4:2
11	2:-6	44	5:10
12	5:-6	45	0:0
13	0:-5	46	1:0
14	1:-5	47	2:0
15	2:-5	48	3:3
16	5:-5	49	4:2
17	0:-3	50	5:0
18	1:-3	51	-4:17
19	2:8	52	-3:16
20	5:-3	53	-2:15
21	0:-1	54	0:-1
22	1:9	55	1:9
23	1:9	56	5:10
24	5:10	57	0:0
25	-3:12	58	1:0
26	-4:4	59	2:8
27	-3:12	60	5:0
28	0:-7	61	0:0
29	1:-7	62	1:0
30	2:-7	63	2:0
31	5:-7	64	5:0
32	0:-4	65	*:*
33	1:-4		

or example, the *action* values held in the above tabl
ate 5 start at position 17 in the array *parsetable* ar
non-*error* values. Positions 17 through 20 represen
tion values:

$: *reduce* E->T

```
* : reduce E->T

+ : shift 8

) : reduce E->T
```

2 SLR(1) implementation

This section describes how to use the SLR(1) p nstructor with error recovery. This implementati e restriction that no production can be of the -> e. Included in this section is a brief descripti e input grammar, how to run the system, and h terpret the output produced.

2.1 Input Grammar

The input for the program is the set of produ fining the CFG which the SLR(1) parsing table is nstructed from. The input will be parsed in a free rmat, that is, no formatting by columns or line boun ll be used. The end of line character will be treat blank character and each symbol on the input file m parated by one or more blanks.

>noianic string, or i: cnaraccers or less not oeginnm
Le character M<lf f and is not one of the metas
•-.>","$", and "•'")• In the event that the user may u
: the metasymbols used by the program, or a nonblank
^ginning with a fl<M, the quote symbol has been
>ecial meaning* If the quote is followed by a
laracter, it will be treated as a terminal s
:herwise, if the quote is followed by a nonblank s
le string following the quote will be treated as th
: the terminal symbol.

Nonterminal symbols are represented as cha
:rings, of 15 characters or less, enclosed by the s
c" and ">". The first symbol of the string, if no
apty string, must begin with a nonblank character but
laracters can appear anywhere else in the string*
rogram also accepts the string "<>" which repres
>nterminal symbol whose name is the empty string*

Productions are represented by writing them in th
-> w; where A is a nonterminal, * is a sequence of g
rmbols, and "->" is a metasymbol recognized by the pr
ich production is separated from the next usin
•tasymbol "." and after the last production, the meta
?ff must appear* The productions can be entered
:der except that the first production, on the input

For example, the grammar presented in V.1 cou
presented 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
e same left hand side (i.e. productions of th
-> $\underline{w}$ where A remains constant between the producti
these cases, the productions can be entered in th
-> $\underline{w}_1$! $\underline{w}_2$! ... ! $\underline{w}_n$ where there exists the produ
-> $\underline{w}_1$, A -> $\underline{w}_2$, ... , A -> $\underline{w}_n$.

For example, the grammar in section V.1 could
ternatively 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
le corresponds to the order in which they will be
ternally. In a similar manner, the terminal
nterminal symbols will be coded in the order corresp
their first appearance in the set of productions.

.2.2 Running the SLR(1) parser constructor

The system can be run on the Vax-11 in the :hool, by entering the following monitor level pro all:

```
$<3[karl]slrbnf
```

Eter invocation, the procedure will ask the user fo Lies used by the program, and run the program.

The first file to be requested is the file cont le set of productions, and is requested with the prom

```
input:
```

The second file request is for the output file ill contain all diagnostic and informatory messages, aquested with the prompt:

```
output:
```

The third file request is for the file that the c LR(1) parsing tables should be saved on, and is req ith the prompt:

```
internal representation:
```

The last two file requests are for temporary file an be used by the program, and are both requested wi romp t:

n. The program will not produce any output, on the
reen, nor will it ask the user for any futher infor
less the SLR(1) parsing table was created and co
nflicts (see section V.2.4 for handling this case).

This paper will not mention how to use the
ntaining the SLR(1) parsing tables except for a
ogram skeleton in appendix a.

2.3 Interpretation of the output file

The output can be broken into two major sections
e first section describes how the program parsed the
ammar and the second section prints the built
rsing tables. However, the second section wi
oduced only if there were no errors detected in the
ction.

The first page of the output is a copy of the
ing parsed, along with any error messages indi
legal syntax. If there were no syntactic mistakes i
put grammar, then this page will be an exact duplic
e input file. Otherwise, portions of the input file
written, and will be interspersed with syntactical
cognized by the program.

For example, the erroneous input:

```
    <S> -> <A> •  <A> -> a <A> b .  A -> a b $
```

ould produce the following output:

```
    <S> -> <A> •  <A> -> a <A> b •  A ***illegal LHS
```

a this example, the program is reporting that roduction has a terminal symbol on the left hand s tie production.

The next three subsections of the output report oding scheme of terminals, nonterminals, and produ sed by the program*

For example, the input:

```
    <S> -> <E> .
    <E> -> <E> * <T> J <T> .
    <T> -> <T> + <F> ! <F> .
    <F> -> id ! ( <E> ) $
```

```
 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 t g schemes, that is, if the string "*undef*" procedes rminal, then that nonterminal does not occur on t eft hand side of any production recognized while p he input file.

Below the coding scheme is a diagnostic summmary ell the program did in parsing the given input bn verything is acceptable to the program, it will prin essage "successful parse" and attempt to constru LR(1) parsing tables. Otherwise, it will give an ummary of why it thought the input was wrong, and abo urther calculations.

Should the input grammar be successfully parsed rogram then attempts to build the SLR(1) parsing t o begin with, it computes the first and follow set ach nonterminal, and prints out these sets. Seco rints out the sets of SLR(1) items defining the co ach state.

For example, the previous input grammar would p

itput for the first five states as follows:

```
---------------------------------------- STA

     1>   <S> -» . <E> '<EOF MARKER>

---------------------------------------- STA

     7)   <F> -> ( . <E> )

---------------------------------------- STA

     6)   <F> -> id •

---------------------------------------- STA

     5)   <T> -> <F>

---------------------------------------- STA

     3)   <E> -> <T> •

     4)   <T> -> <T> • + <F>
```

The last section of the output, for a run, iadable form of the produced parsing table followed '.ze of the array parsetable• Non-error values, o irsing tables, for each state are listed separatel ie action values preceeding the goto values.

ing 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

-----------------------------------------------
```

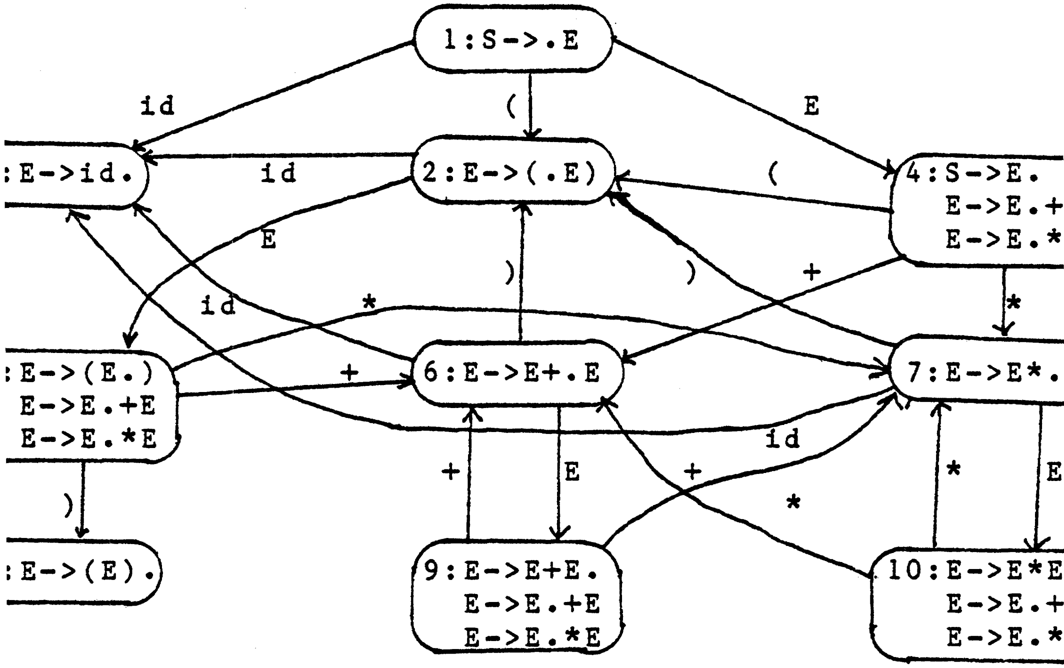
.4 Conflict Resolution

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

For example, the grammar in figure 5.1 is an exampl natural grammar for arithmetic expressions with opera nd *. The LR(0) characteristic automaton, for mmar, and the follow sets are shown in figure 5.2. tes 9 and 10, there will exist S/R conflicts on ymbols + and * if the SLR(1) parser is built fr haracteristic automaton. This can also be seen i utput produced by the program for such an input (see .3).

```
S -> E                  E -> id
E -> E + E              E -> ( E )
E -> E * E
```

figure 5.1



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

figure 5.2

```
---------------------------------------------- STATE : 9

    <E> -> <E> + <E> .

    <E> -> <E> . + <E>

    <E> -> <E> • * <E>

JCE/SHIFT CONFLICT ON SYMBOL +
CRY:         -2 CONFLICTING ENTRY:           6

ICE/SHIFT CONFLICT ON SYMBOL *
CRY:         -2 CONFLICTING ENTRY:           7
---------------------------------------------- STATE : 10

    <E> -> <E> . + <E>

    <E> -> <E> * <E> •

    <E> -> <E> . * <E>

ICE/SHIFT CONFLICT ON SYMBOL +
:RY:         -3 CONFLICTING ENTRY:           6

ICE/SHIFT CONFLICT ON SYMBOL *
:RY:         -3 CONFLICTING ENTRY:           7
```

figure 5.3

: turns out that these conflicts can be resolved i of either a shift or a reduce action by knowing th ince and associativity of these two operators* Fo *i,* looking at state 9 and the operator *, the parse

.ng E * E and reduce it to the string E producing the
:ential form

E + E

Should the grammar in the input file produce confli
program will arbitrarily pick one of the ac
.nitions for the symbol causing the conflict in the s
discard all other conflicting entries. This choic
›rted to the user as shown in figure 5.3. In each c
"OLD ENTRY: xx" represents the entry chosen by
;ram while the "CONFLICTING ENTRY: yy" states
:arded entry. Hence, in state 9, the arbitrary cho
the symbol *, was to reduce on the production labell
:. E->E+E).

To allow the user to change the arbitrary choice
the program, the program will also become interactiv
conflicts arise in building the SLR(1) parser. That
program will prompt the user with the prompt:

ENTER STATE TO RESOLVE:

this response, two choices are available.

If the user responds with the number 0, the pro
l stop so that the user can look at the output fil
er to identify all existing conflicts in building
(1) parser. If the user feels that these conflicts

should rerun the program and when getting the a
ipt, he should resolve the conflicts by using the se
.on.

The second option in responding to the above prompt
: in the state that the user wants to resolve. After
: completes his answer, the program will print out
: of the state, for verification, and will ask the
Lt is the state he wanted.

The next request by the program is for the user
vide the integer code of the terminal symbol causing
:lict using the prompt:

ENTER SYMBOL NUMBER TO RESOLVE:

above, the program will verify the user's response
ıting out the terminal's name and asking the user i
the correct terminal symbol. Again, a "N" response
se the program to reprompt for a state to resolve whi
response will have to program continue processing
olution.

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

ENTER NEW ACTION TO TAKE:

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

The.program will provide the user one last after the conflict resolution has been specif disregard the conflict resolution. A "Y^{lf} response user will cause the resolution to be processed whi response will disregard the resolution provided by t In either case, the program will than request fox conflict resolution with the prompt:

ENTER STATE TO RESOLVE:

At this point, the whole process repeats unJ user responds with a 0. If a 0 is typed in by t then no more conflict resolutions will be processed program will build the SLR(l) parser. Note that th€ will not produce an SLR(l) parser unless at l€ conflict has been resolved.

Size Restrictions

lis program contains several size restrictions whic follows:

) No more than 100 terminal symbols may be used.

i) No more than 200 nonterminal symbols may be used*

ii) No more than 300 productions may appear in tin nput•

v) No terminal or nonterminal name may exceed 1 haracters•

) For each production A -> w, £ can not be a string £ terminal and nonterminal names, exceeding a lengt f ten names*

i) The number of parse states, created by the prograi mst not exceed 600.

'ii) The number of SLR(l) items, excluding the items < :he form A -> . w, must not exceed 9,999* 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
.d either LR(1), LALR(1), weak compatible, or st
itable parsing tables. Included in this section i
ef description of the input grammar, how to run
gram, and how to interpret the output.

.1 Input Grammar

The input for the program is the set of product
ining the CGF from which the parsing tables are t
luced. These productions can be optionally prece
a list of terminals and nonterminals, allowing the
specify the integer codes given to these symbols.

The input will be parsed in a free style format,
no formatting by columns or line boundaries wil
i. The end of line character will be treated as a b

eas t UUC blank.

In general, a terminal symbol is any nonempty strin; dank characters which does not begin with the chara< However, it can not be any of the metasymbols ("•", "#", "->". "'", or "e")- In the event that ' wants to use one of the metasymbols or a st .nning with a ft<ff, as a terminal symbol, the quote syi : preceed the nonblank string*

Nonterminal symbols are represented as any chara .ng enclosed by the symbols ff<lf and ">"• The charac >osing the name of the nonterminal can be any chara rluding the blank) except the symbol fl>ff, and incl name composed by the empty string ("<>")•

Productions are represented by writing them in the • w where A is the name of a nonterminal, w is a sequ :erminal and nonterminal names, and fl->ft is a metasy >gnized by the program* The symbol ffelf has been rese represent the empty string so that productions of a A -> ji can be written.

Productions are separated from each other using isymbol "•", and no symbols should follow the iuction. Productions having the same left hand s • of the form A -> £•$_1$, A -> w^, •• , A -> w_n, ca ne metasymbol "|" is treated as an "or" symbol.

For example, the grammar

```
S -> A          A -> aAb     A -> e
```

ould be entered with the input:

```
<start symbol> -> <A> .
<A> -> a <A> b | e
```

Productions, when parsed, will be coded inte
sing the order in which they appear on the input. Th
estriction on the order in which the production
ritten is that the start production must appear first

Unlike the SLR(1) parser constructor, this p
ptionally allows the user to specify the coding sch
he nonterminal and terminal symbols. That is, befor
tart production the user is allowed to provide a l
erminals, followed by a list of nonterminals, follow
he metasymbol "#". It is not necessary that all ter
nd nonterminals appear in these lists, and either o
ist may be empty. Elements in these lists will be l
n the order that they are found (1 for the first ter
for the second terminal etc. and -1 for the
onterminal, -2 for the second nonterminal etc.).
emaining terminals, or nonterminals, not specified by
ists will be labelled according to the order of

ppearance in the set of productions.

For example, assume using the previous grammar th ser wants the terminal b to be labelled 1 and termina e labelled 2. This could be done by using the input:

```
b a #
<start symbol> -> <A> .
<A> -> a <A> b | e
```

The program described by this section in fact has ie SLR(1) parsing tables (produced by running the rogram described in section V.2) to parse the inpu iis program. Hence, the description of the input rul e formally described by the set of rules used in cr ie SLR(1) parsing tables which are as follows:

```
<> -> <input grammar> .
<input grammar> -> <start prod> ´. <other prods>
                 ! <symbol defns> <start prod>
                   ´. <other prods> .
<start prod> -> nonterminal ´-> nonterminal .
<other prods> -> <production>
               ! <other prods> ´. <production> .
<production> -> nonterminal ´-> <rhs> .
<rhs> -> e-rule
       ! <symbols>
       ! <rhs> | e-rule
       ! <rhs> | <symbols> .
<symbols> -> terminal
           ! nonterminal
           ! <symbols> terminal
           ! <symbols> nonterminal .
<symbol defns> -> <terminals> <nonterminals> #
                ! <terminals> #
                ! <nonterminals> # .
<terminals> -> terminal
             ! <terminals> terminal .
<nonterminals> -> nonterminal
                ! <nonterminals> nonterminal $
```

.3.2 Runing the program

The program can be run on the Vax-11 in the chool by entering the following monitor level pro ll:

```
@[karl]runnewbnf
```

ter invocation, the procedure willask the user fo les used by the program, and then run the program.

The first file requested by the procedure is the second file is request is for the output file which ain all diagnostic and informatory messages, an ested with the prompt:

```
OUTPUT FILE:
```

last request is for the file to save the parsing ta ted and is requested with the prompt:

```
TABLE:
```

Upon completion of the file requests, the program . After the program finishes reading the input bnf f program will request the user to specify what type ser 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 responding parse table, printing out "BUILDING STAT it tries to build state X. This completes eraction the program has with the user.

The first page of the output file is a copy of ut being parsed, along with any error messages descri egal syntax.

For example, the erroneous input:

```
<S> -> <A> .  <A> -> a <A> b .  A -> e
```

produce the following output:

```
 INPUT PARSE OF PRODUCTIONS:
 ----- ----- -- ------------

<S> -> <A> . <A> -> a <A> b . A -> e
                               ^

 *** 32) PRODUCTION DEFINITION EXPECTED
```

bove error is stating that at the beginning on colu f the previous input line, the program was expecting a production but found something else (i.e. t nal symbol A).

The next three subsections of the output file, aft parse of the input, reports the coding scheme of t nals, nonterminals, and productions used by t am.

For example, the input:

```
a b #
<start symbol> -> <A> .
<A> -> a <A> b | e
```

```
TERMINALS:
----------

   0.  $EOF$
   1.  a
   2.  b


NON-TERMINALS:
--------------

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


PRODUCTIONS:
------------

   1<start symbol> -> <A>
   2<A> -> a  <A>  b
   3<A> -> e
```

As can be seen by the above example, addi nformational messages about nonterminal symbol rovided, and are as follows:

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

UNIQUE - States that the start symbol does not anywhere else in the productions and hence valid start symbol.

NOT UNIQUE - States that the start symbol occur another production besides the start prod and hence is an invalid start symbol.

NOT USED ON RHS - states that the nonterminal n appears on the right hand side of any produc

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

NT REPRESENTS NO TERMINAL STRINGS - States that is not any terminal strings derivable fro nonterminal.

NT NOT DEFINED - States that the nonterminal do appear on the left hand side of any prod recognized from the input file.

After the coding schemes, the program will prin irst set of each nonterminal.

Finally, if the user selects to have a nstructed, the program will construct it and pri

.terns) and non-error action and goto values.

For example, using the input grammar used above, ie user chose to build a strong compatible LR p ible, the parse tables printed would be as follows:

```
              STRONG COMPATIBLE L R (1) CHARACTERISTIC
------------------------------STATE : 1----------------------------

    l)<start symbol> ->   . <A>
     LOOKAHEADS:
          $EOF$

CABLE ENTRIES:

?EOF$   REDUCE BY 3
i  SHIFT TO 3
CA>   GO TO 2
------------------------------STATE : 2 ---------------------------

    l)<start symbol> -> <A> .
     LOOKAHEADS:
          $EOF$

CABLE ENTRIES:

?EOF$   REDUCE BY 1
---------------------------- STATE : 3 ----------------------------

    2)<A> -> a   . <A> b
     LOOKAHEADS:
          $EOF$
          b

rABLE ENTRIES:

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

```
--------------------- STATE : 4 ----------------------

   2)<A> -> a <A>  . b
    LOOKAHEADS:
        $EOF$
        b

ABLE ENTRIES:

  SHIFT TO 5
--------------------- STATE : 5 ----------------------

   2)<A> -> a <A> b  .
    LOOKAHEADS:
        $EOF$
        b

ABLE ENTRIES:

EOF$  REDUCE BY 2
  REDUCE by 2
```

Appendix A

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

```
ogram doparse(table, {any other files used by progra

nst
   numberstates      = x; {x≥   of actual parse state
   parsetablesize    = y; {y≥ actual size of
                                  array parsetable}
   numberproductions = z; {z≥ actual number
                                  of productions}
   errorvalue        = n; {n value not in set of labe

pe

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

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

r

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

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

   {returns stack with new state added in front}

r temporary : parsestack;

gin
 new(temporary);
 with temporary^ do begin
    topstate:=newstate;
```

```
motion pop(stack : parsestack) : parsestack;

    {removes the top element of the stack)

egin
  pop:»stack~«next;
  dispose(stack)
Li;

tnetion top(stack : parsestack) : integer;

    {returns state on top of stack)

egi
  top:=stack~«topstate
Li;

tnction empty(stack : parsestack) : parsestack;

    {returns an empty stack)

egin
  while stackonil dD stack:*pop(stack);
  empty:«nil
Li;

mction gettoken : integer;

  {This routine returns the label of the next terminal
   occuring in the input file)

Li;

'ocedure semantics(stack : parsestack;
                      production : integer);

  {does any semantic routines associated with reducinj
   the given production)

Li;

-ocedure errormessages(state , symbol : integer);

  {prints out message corresponding to error value
   for state and symbol)

Li;
```

```
function parse : boolean;

      {parses input, returns true if no parsing err-
       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 en
     in parsetable}

    stateentry ª record
       startposition , size : integer
    end;

    {representation of a production in productionli

    productionentry « record
       lhssymbol , rhslength : integer
    end;

var

    parsetable : array [ 1 .• parsetablesize ] oj[ t

    actionlist , gotolist : array [ 1 .. numberstat
                                    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 re
    errorcontinue,    {init-error state}
    topoftable,       {actual size of parsetable}
    productioncount   {actual number of productions}
         : integer;
```

```
{local variables}

token : integer ; {next terminal from input}

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

stop : boolean; {true when have parsed whole input}

parseerror : boolean; {true if any parsing errors}

stack : parsestack; {holds path}

procedure getparsetable;

     {reads in parsing tables}

var index : integer;

   procedure getin(var invalue : integer);

        {reads in next integer from file table}

   begin
        invalue:=table^;
        get(table)
   end;

begin
   reset(table);
   getin(topstate);
   getin(parsestart);
   getin(errorstart);
   getin(errorcontinue);
   getin(topoftable);
   getin(productioncount);
   for index:=1 to topstate do begin
      with actionlist[index] do begin
         getin(startposition);
         getin(size)
      end;
      with gotolist[index] do begin
         getin(startposition);
         getin(size)
      end
   end;
   for index:=1 to topoftable 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  ) : parsest

     {empties stack and put value on bottom of s

begin
   clear:=push(empty(stack),newbottom)
end;

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

     {takes the requested amount of states off t

begin
   if (amount = 0) or (stack = nil)
      then popelements:=stack
      else popelements:=popelements(pop(stack),
                                        pred(cou
end;

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

     {takes the requested amount of states off t
      but if stack underflow occurs, it resets
      the bottom state}

begin
   stack:=popelements(stack,count);
   if stack = nil
      then popoffproduction:=push(stack,errorco
      else popoffproduction:=stack
end;
```

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

     {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 ) : parsestac

     {handles overdefined actions}

begin
   if token = eoftoken
      then begin
         overdefined:=empty(stack);
         stop:=true
      end
      else begin
         overdefined:=push(clear(stack,errorcontinu
                     findvalue(actionlist[errorstar
                                     token));
         token:=gettoken
      end
end;
```

```
function unknown(stack : parsestack;
                    var token : integer ) : parsest

     {handles error actions}

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

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

     {handles performing a shift action}

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

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

     {handles performing a reduction}

var gotovalue : integer;

begin
   gotovalue:=findvalue(gotolist[top(stack)],
                 productionlist[production].lhssymb
   if gotovalue = errorvalue
      then doreduction:=unknown(stack,token)
      else begin
         semantics(stack,production);
         doreduction:=push(popoffproduction(stack,
                              productionlist[produc
                                            . rhsle
                           gotovalue)
      end
end;
```

```
egin
  getparsetable;
  stack:=push(nil,parsestart);
  stop:=false;
  errorvalue:= -succ(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,toke
  until stop;
  parse:= not parseerror
nd;
```

Appendix J3

Sample PASCAL skeleton for use of the

LR(1), LALR(l), weak compatible, and

stong compatible parsing tables

```
ogram doparse(table, {any other files used by prograt

nst

 numberstates      * x; {x >. # °f actual parse states
 parsetablesize    » y; {y j> actual size of
                                 array parsetable}
 numberproductions * z; {z j> actual number of
                                        productions>

pe

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

 parsestack » ~stacknode;
 stacknode  * record
      topstate : integer;
      next     : parsestack
 end;

r

 table : file oj! integer; {file containing
                            parsing tables)
```

```
nction push(stack : parsestack;
         newstate : integer) : parsestack;

   {returns stack with new state added in front}

r temporary : parsestack;

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

action pop(stack : parsestack) : parsestack;

   {removes the top element of the stack}

gin
 pop:=stack^.next;
 dispose(stack)
l;

iction top(stack : parsestack) : integer;

   {returns the top of the stack}

 in
 top:=stack.topstate
 ;

 ction empty(stack : parsestack) : parsestack;

   {returns an empty stack}

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

 ction gettoken : integer;

   {This routine returns the label of the next termina
    occuring in the input file}

 ;
```

```
rocedure semantics(stack : parsestack;
                    production : integer);

    {Does any semantic routines associated with reduc
     the given production)

Li;

rocedure errormessages(state , symbol : integer);

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



mction parse : boolean;

    {Parses input. Returns true if no parsing errors
     are found in parsing the input}

>nst  eoftoken * 0 ;

ype

  {representation of an entry in parsetable)

  tableentry » record
     symbol , value : integer
  end;

  {representation of a reference to a group of entrie
   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 ] oj[ tat
   actionlist , gotolist : array [ 1 •• numberstates
                                      of stateentry;
   productionlist : array [ 1 •• numberproductions ]
                                 of productionentry;
   topstate : integer; {actual number of parse state
   {other local variables)
   token : integer; {next terminal from input}
   errorvalue : integer; {made up number for error \
   value : integer; {next action to take in parsing)
   stop : boolean; {true when have finished parsing)
   parseerror : boolean; {true if any parsing error
   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:*l J^o topstate jd£ begin
      with actionlist [i] dj£ begin
         getin(startposition);
         getin(size);
         for j:*startposition J^o size do
            with parsetable [j ] jd£ begin
               getin(symbol);
               getin(value)
            end
      end;
      with gotolist [index] d^D begin
         getin(startposition);
         getin(size);
         for j:*startposition J^o size do
            with parsetable [j ] djD begin
               getin(symbol);
               getin(value)
            end
      end
   end
end;

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

     {takes the requested amount of states off

begin
   while count>0 d^o begin
      stack:*pop(stack);
      count:*pred(count)
   end
end;

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

     {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 doshift(stack : parsestack;
                     shiftaction : integer;
                     var token : integer) : parsest

     {handles performing a shift}

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

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

     {handles performing a reduction}

ar gotovalue : integer;
```

```
   begin
      gotovalue:=findvalue(gotolist[top(stack)],
                    productionlist[production].lhssy
      if gotovalue = errorvalue
         then begin
            doreduction:=empty(stack);
            parseerror:=true;
            stop:=true
         end
         else begin
            semantics(stack,production);
            doreduction:=push(popoffproduction(stack,
                                 productionlist[produ
                                                . rhsl
                              gotovalue)
         end
   end;

begin
   getparsetable;
   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
 nd;
```

References

[AEH72] Anderson, T.; Eve, J.; and Horning, J. - Effi LR(1) parsers, Acta Informatica 2, p12-39 (19

[Alp76] Alperb, Bowen, Martin Chaney, Micheal Fay, Th Pennello, and Rachel Radin. - Translator writ system for the Burroughs B5700. Information S , UC Santa Cruz, Santa Cruz, CA. (1976)

[A&U77] Aho, A. V. and Ullman, J. D. - Principles of compiler design. Addison-Wesley Publishing Co Reading Mass. (1979)

[DeR69] DeRemer, F. - Practical translators for LR(k languages. Ph.D. thesis, Dept. of Electrical Engineering, M.I.T, Cambridge, Mass. (1969)

[DeR71] DeRemer, F. L. - Simple LR(k) grammars. Comm. p453-460 (1971)

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

[D&R77] Druseikis, F. and Ripley, G. D. - Extended SI parsers for error recovery and repair. Dept. Computer Science, Univ. of Arizona, Tuscon Az.(1977)

[Gal79] Gallier, J. H. - Class notes for CSE341, Dep Computer and Information Science, University Pennsylvania, Philadelphia, PA. (1979)

[G&R75] Graham, S. and Rhodes, S. - Practical syntac error recovery. Comm. ACM 18, p639-650 (1975

[Knu65] Knuth, D. E. - On the translation of languag left to rigth. Information and Control 8, p6

[LLH71] LaLonde, W. R.; Lee, E. S.; and Horning, J. LALR(k) parser generator. Proc. IFIP congres

6] 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)

7a] Pager, D. - A practical general method for constructing LR(k) parsers. Acta Informatica 7, p249-268 (1977)

7b] Pager, D. - The lane tracing algorithm for constructing LR(k) parsers and ways of enhancing i efficeincy. Information Sciences 12, p19-42 (1977)

7] Pennello, T. J. - Error recovery for LR parsers. Master's thesis, Tech. Rpt. 77-7-002, Information Sciences, UC Santa Cruz, CA (1977)

8] 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)

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