1989

Development of an LALR(1) parser generator

Marie Schneck
Lehigh University

Development of an LALR(1)

Parser Generator

By

Marie Schneck

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

1989

This thesis is accepted and approved in partial fulfillment of the requirements for the Degree of Master of Science.

May 5, 1989
(date)

Samuel L. Gulden
Professor in Charge

Howard J. Hamilton
CS Division Chairman

Lawrence J. Vanasse
CSEE Department Chairman

ACKNOWLEDGEMENTS

I would like to thank Dr. William Seaman, formerly of Muhlenberg College, for introducing me to Compiler Design and to LR grammars and LR parsing. He initiated my interest in the subject, which was further increased by Prof. Samuel Gulden in his Compiler Design Courses at Lehigh. I also greatly appreciate the knowledge gained from both Dr. Edwin Kay and Dr. Robert Barnes in the area of Formal Grammars and Finite Automata, which was invaluable in the the development of this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

Development of an LALR(1) Parser Generator

Marie Schneck

ABSTRACT

LALR(1) grammars are a subset of LR(1) grammars. The techniques used to create LR(1) parsing tables can be directly applied to LALR(1) table construction and the LALR(1) table can be formed by merging states while constructing the LR(1) parsing table. This method is inefficient if one is only looking for LALR(1) grammars as it requires essentially forming the LR(1) machine by computing all the required lookaheads and then merging compatible states.

The method outlined in this thesis and incorporated into the accompaning computer program to generate the LALR(1) parsing action table is due to Frank DeReemer and Thomas Pennello. The method is extremely efficient in that it works from the LR(0) machine and generates *only* the lookaheads needed for production reduction in inconsistent states. No other lookaheads are computed. Thus the number of states to be computed is kept to a minimum by only generating the LR(0) machine and computation of lookaheads is kept to a minimum by only determining the lookaheads needed in inconsistent states.

The one disadvantage to deReemer's and Pennello's method is that if the grammar is not LALR(1) it is not possible to discover if the grammar is LR(1) since all the LR(1) states were never constructed. However, since most grammars which are LR(1) are also LALR(1), the efficiency of the method outweighs this disadvantage. Also, the program gives several diagnostics which show exactly where the grammar is

not LALR(1). These diagnostics can be used to change the grammar to LALR(1), if at all possible.

The parsing action table is input to a shift-reduce parser which is then used to parse strings in the language.

CHAPTER 1

TERMINOLOGY AND DEFINITIONS

1.1 TERMINOLOGY

The following terminology will be used through out the paper.

A context-free grammar, or CFG, G is specified by a quadruple (N, T, P, S), where

N is the finite set of nonterminal symbols

T is the finite set of terminal symbols and N and T are disjoint

$P \subseteq N \times (N \cup T)^*$ is a finite set of productions

S in N is the start symbol

The vocabulary of G refers to $N \cup T$ and is denoted by V. A production in G is denoted by $A \rightarrow \alpha$ where $A \varepsilon N$ and $\alpha \varepsilon V^*$. The empty string is denoted by λ. The length of any string α is denoted by $| \alpha |$.

The following usual conventions will be observed.

S, A, B, C, ... ε N

a, b, c, ... ε T

... x, y, z ε T^*

$\alpha, \beta, \gamma, \ldots$ ε V^*

There exists an augmented production $S' \rightarrow S\#$ where S is the start symbol of the grammar G, # ε T and is considered the end of string of the grammar, and S′ and # appear in no other productions. In the program which generates the LALR(1) parsing action table, this production will be added by the program and is considered an *augmented* production, i.e. not part of the original grammar. Thus the end of string symbol # must not be part of the original grammar.

The notation $\Rightarrow_r$ refers to a rightmost derivation. Thus for all A ε N, $\alpha, \beta \varepsilon$ V^*, y ε T^* and $A \rightarrow \alpha \varepsilon P$, if $S \Rightarrow_r{}^* \beta Ay \Rightarrow_r \beta\alpha y$ is a rightmost derivative in G then both βAy and $\beta\alpha$y are rightmost sentential forms (i.e. sentential forms produced by a rightmost derivative). A *nullable* nonterminal A is one which produces λ (i.e. $A \Rightarrow^* \lambda$) . The language L(G) of the grammar G is the set of all sentences y ε T^* such that $S \Rightarrow^*$ y, where S is the start symbol of grammar G.

1.2 SHIFT-REDUCE PARSING

A shift-reduce parser is a bottom-up parser which operates by shifting input symbols onto a stack until the right hand side of some production is recognized which can be replaced by the left hand side of that production at that point. Since the shift-reduce parser operates by essentially doing an inverse rightmost derivative, the point at which the reduction $A \rightarrow \alpha$ would be done is exactly the point at which A would be replaced by α in the rightmost derivation.

As the parser shifts symbols onto the stack, the current sentential form will always be on the stack combined with the remaining input. The parser must determine when to do a reduction by determining what portion of the current sentential form, if any, is the ***handle*** . The ***handle*** of a right sentential form γ is a production $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right sentential form in a rightmost derivative of γ. Thus if the following derivation exists , $S \Rightarrow_r{}^* \alpha Ay \Rightarrow_r \alpha\beta y = \gamma$ then $A \rightarrow \beta$ in the position following α is a handle of $\alpha\beta$y. The handle for γ may be denoted by the pair (π , i) where $\pi \varepsilon P$, $i \geq 0$ is the position where the replacement occurs. In the previous example, the handle for γ is $(A \rightarrow \beta, |\alpha\beta|)$. The handle is always on the top of the stack.

The stack will always contain the ***viable prefixes*** of the right sentential forms of

the grammar. A string $\alpha\beta$ is called a *viable prefix* if there is some right sentential form containing $\alpha\beta$. In the above example $\alpha\beta$ is a *viable prefix* (and so is αA).

If the shift-reduce parser is to be totally deterministic (no back tracking), the parser must be able to determine at each step what the handle is, if any, and what production reduction to apply.

Thus, a shift-reduce parser will start with an empty stack and an input string to parse and will finish with the start symbol of the grammar on the stack and an empty input string.

Consider the following expression grammar G1 with the augmented production added:

E′	→	E #
E	→	T
E	→	E A T
T	→	F
T	→	T M F
F	→	(E)
F	→	id
F	→	intliteral
A	→	+
A	→	-
M	→	*
M	→	/

Consider the rightmost derivative of the string a + g * d - f / 10 # in G1. Note that the string is terminated by the end of string symbol #.

E′ $\Rightarrow_r$ E # $\Rightarrow_r$ E A T# $\Rightarrow_r$ E A T M F# $\Rightarrow_r$ E A T M 10 #
$\Rightarrow_r$ E A T / 10 # $\Rightarrow_r$ E A F / 10 # $\Rightarrow_r$ E A f / 10 #
$\Rightarrow_r$ E - f / 10 # $\Rightarrow_r$ E A T - f / 10 #
$\Rightarrow_r$ E A T M F - f / 10 # $\Rightarrow_r$ E A T M d - f / 10 #
$\Rightarrow_r$ E A T * d - f / 10 # $\Rightarrow_r$ E A F * d - f / 10 #
$\Rightarrow_r$ E A g * d - f / 10 # $\Rightarrow_r$ E + g * d - f / 10 #
$\Rightarrow_r$ T + g * d - f / 10 # $\Rightarrow_r$ F + g * d - f / 10 #
$\Rightarrow_r$ a + g * d - f / 10 #

The following is the inverse of this rightmost derivative, so it demonstrates the way a shift-reduce parser would correctly parse the string (by doing the proper shifts and reductions).

a + g * d - f / 10 #
F + g * d - f / 10 #
T + g * d - f / 10 #
E + g * d - f / 10 #
E A g * d - f / 10 #
E A F * d - f / 10 #
E A T * d - f / 10 #
E A T M d - f / 10 #
E A T M F - f / 10 #
E A T - f / 10 #
E - f / 10 #
E A f / 10 #
E A F / 10 #
E A T / 10 #
E A T M 10 #
E A T M F #
E A T #
E #
E'

1.3 DEFINITION OF AN LR(K) GRAMMAR

A grammar G is said to be LR(K) if and only if the following conditions apply:

For any w, w', x ε T^* , γ, α, α', β, β' ε V^*, X, X' ε N then if

(1) $S \Rightarrow_r^* \alpha X w \Rightarrow_r \alpha\beta w = \gamma w$

and hence γw has ($X \rightarrow \beta, |\alpha\beta|$) as a handle

(2) $S \Rightarrow_r^* \alpha' X' x \Rightarrow_r \alpha'\beta' x = \gamma w'$

and hence γw' has ($X' \rightarrow \beta', |\alpha'\beta'|$ as a handle

(3) $First_k(w) = First_k(w')$

then

(4) ($X' \rightarrow \beta', |\alpha'\beta'|$) = ($X \rightarrow \beta, |\alpha\beta|$)

Some relationships can be seen from the above definition. Since the two handles are equal in number (4), we have $X = X'$, $\beta=\beta'$, and $|\alpha\beta|=|\alpha'\beta'|$ = some integer i. Also, $\gamma w=\alpha\beta w$, $\gamma w'=\alpha'\beta' x$ where $\gamma=\alpha\beta$. Hence $\alpha\beta w'=\gamma w'=\alpha'\beta' x$. Thus $\alpha\beta=First_i(\alpha\beta w') = First_i(\alpha'\beta' x) = \alpha'\beta'$ since $|\alpha\beta| = |\alpha'\beta'|$ = i. Hence since $\alpha\beta=\alpha'\beta'$ and $\beta=\beta'$ we have $\alpha=\alpha'$. Also, since $\alpha\beta w'=\alpha'\beta' x$ and $\alpha\beta=\alpha'\beta'$ we have $w'=x$.

The above definition does not provide an easy way to tell if a grammar is LR(K). It does, however, provide the basis for determining whether or not a grammar is LR(K) since it clearly states that for a grammar to be LR(K) we must be able to determine at each stage of the parse, with K lookaheads, exactly what production reduction, if any, to apply. There will be one and only one possibility at each stage of the parse.

Looking at the previous string a + g * d - f / 10 # in grammar G1 , which is LR(1), the following demonstrates how an LR(K) parser would correctly parse the string. The productions in G1 have been numbered from 1 to 12.

STACK	ACTION	REMAINING INPUT
λ		a+g*d-f/10#
a	shift	+g*d-f/10#
F	reduce,7	+g*d-f/10#
T	reduce,4	+g*d-f/10#
E	reduce,2	+g*d-f/10#
E+	shift	g*d-f/10#
EA	reduce,9	g*d-f/10#
EAg	shift	*d-f/10#
EAF	reduce,7	*d-f/10#
EAT	reduce,4	*d-f/10#
EAT*	shift	d-f/10#
EATM	reduce,11	d-f/10#
EATMd	shift	-f/10#
EATMF	reduce,7	-f/10#
EAT	reduce,5	-f/10#
E	reduce,3	-f/10#
E-	shift	f/10#
EA	reduce,10	f/10#
EAf	shift	/10#
EAF	reduce,7	/10#
EAT	reduce,4	/10#
EAT/	shift	10#
EATM	reduce,12	10#
EATM10	shift	#
EATMF	reduce,8	#
EAT	reduce,5	#
E	reduce,3	#
E#	shift	λ
E′	reduce,1	λ

CHAPTER 2

LR PARSING

2.1 GENERAL PARSER CONSTRUCTION

Since the accompaning computer program is only concerned with LR(0) and LALR(1) grammars the discussion from now on will refer to 0 or 1 lookaheads. All ideas can be extended to k lookaheads by considering strings of k lookahead symbols. For the moment, the discussion is concerned with LR(0) and LR(1). LALR(1), a subset of LR(1) , will be explained later on.

All LR(1) parsing methods depend upon an item of the form

$A \rightarrow X_1 \ldots X_i \bullet X_{i+i} \ldots X_j \ , \ L$

where $L = \{l_1 l_2 \ldots l_m\}$, $l_i \ \varepsilon T \cup \{\lambda\}$, $A \ \varepsilon \ P$, $X_i \ \varepsilon \ N \cup T$ or, equivalently ,

$A \rightarrow \beta_1 \bullet \beta_2 \ , \ L$ where $\beta_1, \beta_2 \ \varepsilon \ V^*$.

If a grammar is LR(0), the set of lookaheads L is eliminated from the definition of an item since the parse will precede without having to examine any lookaheads for production reductions.

The • symbol indicates how much of the production has been parsed. The symbols on the left hand side of the • have been parsed and are on the stack, and the symbols on the right hand side of the • are still contained in the input string. The set of lookaheads L are the terminal symbols (plus λ) which can follow the production A at this point in the parse. Thus, if there exists a rightmost derivative $S \Rightarrow_r^* \alpha Ax \Rightarrow_r \alpha\beta_1\beta_2 x$ then $L = First_1(x)$, $x \varepsilon T^*$.

In order to construct any LR(1) parser it is necessary to construct a collection of sets of the above items. Each set of items represents a state in the action of the parser. Transitions between these states are determined by constructing a GoTo Function for the grammar and actions (shift or reduce) are determined by constructing

a Parsing Action Table for the grammar. These two tables will drive the shift-reduce parser.

The GoTo Function for a grammar G is the transition function of a deterministic finite state automaton that recognizes the viable prefixes of G. Since the GoTo Function is used to construct the collection of item sets of grammar G, the collection of item sets also recognizes the viable prefixes of G. The GoTo Function Table is indexed by the states in the parse and the vocabulary symbols of the grammar. Thus GoTo(i,X) = j, where i,j are states and X ε N $\cup$ T. Thus, in state i with next symbol X GoTo state j.

The Parsing Action Table indicates if a shift or reduce action should be taken depending on the current state of the parse and the lookahead string. This table is indexed by the states and the terminal symbols. The possible actions indicated by the Parsing Action Table can be determined as follows.

Let P = Parsing Action Function, $\{S_0, S_1, \ldots, S_n\}$ = collection of sets of items (the set of states), A,B ε N , α, β ε V^*, and a,b ε T. Then $P(S_i, a)$ = action in state S_i with lookahead symbol a.

1) if the item $B \rightarrow \alpha \bullet b\beta$, L ε S_i then $P(S_i,b)$ = shift = (s).

2) if the item $B \rightarrow \alpha \bullet$, L ε S_i then for all b ε L ,$P(S_i, b)$ = reduce by $B \rightarrow \alpha$, = (r,j) where j refers to production $B \rightarrow \alpha$.

3) accept is the special case where if the augmented production $S' \rightarrow S \bullet \#$,$\{\lambda\}$ ε S_i then $P(S_i, \#)$ = shift and $P(S_0, S')$ = accept.

4) all other entries are error.

A *configuration* of an LR(1) parser is acturately described as follows, where S_i is a state, X_j ε N $\cup$ T, and a_i ε T.

$$S_0X_1S_1X_2S_2 \ldots X_mS_m \text{ , } a_ia_{i+1} \ldots a_n\#$$

In the above, $S_0X_1 \ldots S_mX_m$ are the contents of the parse stack, and $a_i \ldots a_n\#$ are the rest of the input string. It is ,of course , possible for $a_i \ldots a_n$ to be λ. The above represents the rightmost derivative $S \Rightarrow_r{}^* X_1X_2X_3 \ldots X_m a_i \ldots a_n\#$. The dot position $\bullet$ is between X_m and a_i. At each step of the parse, the parser examines the next input symbol and decides if it should shift the next symbol or if it should reduce by a particular production(i.e. a handle has been recognized).

The relation $\vdash$ is defined as a move from one configuration to another configuration. The relation $\vdash$ can consist of either a shift a ε T (a read) or a reduce by $A \rightarrow \alpha$.

If the parser shifts a symbol, it then examines the GoTo Function to determine what state to go to next. This next state is then shifted onto the stack. If the parser action is to reduce by production $A \rightarrow \alpha$ the parser pops $2 \times |\alpha|$ symbols off the stack if a state and a vocabularly symbol were originally pushed onto the stack. It then examines the GoTo Function for the state at the top of the stack and the nonterminal A. The nonterminal A is now shifted onto the stack, followed by the next state indicated by the GoTo Function.

It is important to note here that the LR(1) parser does not need the vocabularly symbol X_i on the stack. The state symbol and the next lookahead is all that is needed to describe all parsing actions. So the LALR(1) parser program which I have written only pushes states on the stack.

It is also important to note that the GoTo Function and the Parsing Action Table can be more efficiently compacted into one table indexed by the states and the vocabularly symbols. Thus if a ε T indicates a shift, the parser shifts a and goes to state j indicated by the GoTo Function. The appropriate entry in the table becomes (s,j). After a reduction $A \rightarrow \alpha$ the parser shifts on the nonterminal A created by the

reduction and goes to the state indicated by the GoTo Function. The reduction is recorded in the table by (r,k) where k indicates the production A $\rightarrow\alpha$. The appropriate shift action for A is recorded as (s,j), as above. Thus a reduction requires two actions. First in state l with α on the top of the stack, reduce by k, popping $|\alpha|$ symbols off the stack. Second, having popped back to state i, shift A and GoTo j (i.e. (s,j)).

The combined table will be called the Parse Table, or PAT.

If a grammar is LR(0), LR(1), or LALR(1) there will be only one possible action in each state of the parse. An *inconsistent* state will have a conflict. The possible conflicts are a shift-reduce conflict or a reduce-reduce conflict. The former arises when a lookahead symbol indicates a possible shift action and a reduce action. The latter arises when a lookahead symbol indicates two or more possible reduce actions. A grammar is LR(0) if there are no *inconsistent* states when no lookaheads are examined. A grammar is LR(1) or LALR(1) when there are no *inconsistent* states when one lookahead is examined.

2.2 CONSTRUCTION OF THE COLLECTION OF SETS OF ITEMS

In order to construct the collection of item sets , it is necessary to start with the augmented production $S' \rightarrow \bullet\ S\#, \{\lambda\}$. Intuitively this means that at the beginning of the parse we are looking for all symbols that are derived from the start symbol S followed by the end of string #. The augmented symbol S' can only be followed by λ. Also, by including the augmented production $S' \rightarrow \bullet\ S\#$, we have made S# a *viable prefix* of the grammar.

Using the grammar G1 listed earlier, we start with $E' \rightarrow \bullet\ E\#$, $\{\lambda\}$. In this item, since E is a nonterminal it can be expanded to *predict* more productions and more configurations. This is done by the *closure* operation. Every nonterminal that follows a

dot is expanded by *predicting* its production plus any terminal symbols that can follow the production *at this stage of the parse.* Thus from $E' \rightarrow \bullet\ E\#,\{\lambda\}$ we would predict

$E \rightarrow \bullet\ T, \{\#\}$
$E \rightarrow \bullet\ E\ A\ T\ , \{\#\}$

Now since T follows a dot, we further predict

$T \rightarrow \bullet\ F, \{\#\}$
$T \rightarrow \bullet\ T\ M\ F\ , \{\#\}$

Further predict F by

$F \rightarrow \bullet\ (\ E\ )\ , \{\#\}$
$F \rightarrow \bullet$ id, $\{\#\}$
$F \rightarrow \bullet$ intliteral, $\{\#\}$

The second E from $E \rightarrow \bullet\ E\ A\ T, \{\#\}$ must be further expanded by

$E \rightarrow \bullet\ T, \{+\ -\}$
$E \rightarrow \bullet\ E\ A\ T, \{+\ -\}$

This T and the second T from $T \rightarrow \bullet\ T\ M\ F, \{\#\}$ must be further expanded to predict

$T \rightarrow \bullet\ T\ M\ F, \{+\ -\ *\ /\}$
$T \rightarrow \bullet\ F, \{+\ -\ *\ /\}$

This new F must also be expanded to

$F \rightarrow \bullet\ (\ E\ ), \{+\ -\ *\ /\}$
$F \rightarrow \bullet$ id, $\{+\ -\ *\ /\}$
$F \rightarrow \bullet$ intliteral, $\{+\ -\ *\ /\}$

There are no more predictions that will create any new follow symbols, so the above can be condensed to

STATE 0

$E' \rightarrow \bullet\ E\#, \{\lambda\}$
$E \rightarrow \bullet\ T, \{+\ -\ \#\}$
$E \rightarrow \bullet\ E\ A\ T\ , \{+\ -\ \#\}$
$T \rightarrow \bullet\ F\ , \{+\ -\ *\ /\ \#\}$
$T \rightarrow \bullet\ T\ M\ F\ , \{+\ -\ *\ /\ \#\}$
$F \rightarrow \bullet\ (\ E\ )\ , \{+\ -\ *\ /\ \#\}$
$F \rightarrow \bullet$ id $, \{+\ -\ *\ /\ \#\}$
$F \rightarrow \bullet$ intliteral $, \{+\ -\ *\ /\ \#\}$

Basically the two steps performed here were

1. In S_0 = State 0 add the augmenting production $S' \rightarrow \bullet\ S\#, \{\lambda\}$.

2. If $A \rightarrow \bullet\ B\alpha$, L ε S_0 and $B \rightarrow \beta$ is a production in G then add $B \rightarrow \bullet\beta, \{x\}$ to S_0 for all x ε $First_1(\alpha L)$. Note that α can equal λ here. Repeat step 2 until no new items are added to S_0. As above, take the union of all the follow sets to form the final prediction. This is the *closure* of $S' \rightarrow \bullet\ S\#, \{\lambda\}$.

To form the next set of items, form the GoTo Function from State S_0. Consider each vocabularly symbol after the • and go past this symbol to the next state in the parse. Intuitively this means the parse has consumed this symbol and the symbol is on the parse stack. Form the *closure* in each state. Six states will be formed by doing this. They are

STATE 1 = GOTO(0,E)
$E' \rightarrow E \bullet \#, \{\lambda\}$
$E \rightarrow E \bullet A\ T, \{+\ -\ \#\}$
$A \rightarrow \bullet\ +, \{id\ intliteral\}$
$A \rightarrow \bullet\ -, \{id\ intliteral\}$

STATE 2 = GOTO(0,T)
$E \rightarrow T \bullet, \{+\ -\ \#\}$
$T \rightarrow T \bullet M\ F, \{+\ -\ *\ /\ \#\}$
$M \rightarrow \bullet\ *, \{id\ intliteral\}$
$M \rightarrow \bullet\ /, \{id\ intliteral\}$

STATE 3 = GOTO(0,F)
$T \rightarrow F \bullet, \{+\ -\ *\ /\ \#\}$

STATE 4 = GOTO(0,()
$F \rightarrow (\ \bullet\ E\ ), \{+\ -\ *\ /\ \#\}$
$E \rightarrow \bullet\ T, \{+\ -\ )\}$
$E \rightarrow \bullet\ E\ A\ T, \{+\ -\ )\}$
$T \rightarrow \bullet\ F, \{+\ -\ *\ /\ )\}$
$T \rightarrow \bullet\ T\ M\ F, \{+\ -\ *\ /\ )\}$
$F \rightarrow \bullet\ (\ E\ ), \{+\ -\ *\ /\ )\}$
$F \rightarrow \bullet\ id, \{+\ -\ *\ /\ )\}$
$F \rightarrow \bullet\ intliteral, \{+\ -\ *\ /\ )\}$

STATE 5 = GOTO(0,id)
$F \rightarrow\ id \bullet, \{+\ -\ *\ /\ \#\}$

STATE 6 = GOTO(0,intliteral)
$F \rightarrow intliteral \bullet, \{+\ -\ *\ /\ \#\}$

All states are formed by continuing in this way until no new item sets are created. The basic rules used to form the GoTo Sets are

1. If $A \rightarrow \alpha \bullet X\beta$, L ε S_i then add $A \rightarrow \alpha X \bullet \beta$, L to GoTo(i,X) = S_j where X ε N $\cup$ T

2. Take the closure of state S_j. If $A \rightarrow \alpha \bullet B\beta$, L ε S_j and $B \rightarrow \gamma$ is a production in P, add $B \rightarrow \bullet\ \gamma, \{x\}$ to S_j for all x ε $First_1(\beta L)$. As before β can

be λ. Repeat until no new items are created.

The above operations are continued until no new states are formed.

A *kernel* item is defined to be the initial item $E' \rightarrow \bullet\ E\#$, $\{\lambda\}$ and all other items whose dots are not at the left end of the right hand side of the production (these are actually all items formed by moving the dot over and moving to the next state by GoTo(S_i, X)). Thus, all *nonkernel* items have dots at the left end of the right hand side of the production and are actually formed by performing the closure operation.

It is instructive to note that every state is determined by its kernel items. States with identical kernel items will always remain the same after the closure operation is performed. Thus, space could be saved by only storing the kernel items and by performing the closure operation whenever necessary. I did not do this in the computer program since the LALR(1) collection of sets is considerably smaller then the LR(1) collection of sets. It would be extremely time consuming to continually create the closure items.

An efficient way of representing an item is by the trio (i,j,L) where i = a production number, j = the position of the dot and $0 \leq j \leq$ length of production i, and L = set of follow symbols. The LR(0) collection of sets would have no follow sets with the items. It is called the LR(0) *finite state machine.*

2.3 DEFINITION OF LALR(1)

As the LR(1) sets are formed, it becomes very apparent that several states will be the same in the productions and the dot positions, but differ in the follow symbols. For example, in the above expression grammar, the construction of the GoTo Sets would create, among others, the following two states.

STATE 3 = GOTO(0,F)
$T \rightarrow F\ \bullet$, {+ − * / #}

GOTO(4,F)
T → F •, {+ − * /)}

It is very evident that these two states differ only in their follow sets.

Similarly, the LR(1) machine will contain the following two states.

Continuing from State 4 from the kernel item F → (• E), {+ − * / #}
we will create a state with the following item
F → (E) • , {+ − * / #}

Continuing from State 4 from the closure item F → • (E), {+ − * /)}
we will create a state with the following item
F → (E) • , {+ − * /)}

If all states such as the above are combined by taking the union of the follow sets in items in which the dot position and the production numbers are identical the four states above are reduced to two. They are

State I
T → F • , {+ − * /) #}

State J
F → (E) • , {+ − * /) #}

The grammar created in this way is called an LALR(1) grammar if there are no inconsistent states when one lookahead is examined. This means there can be no shift-reduce or reduce-reduce conflicts after all the appropriate states have been merged. Either the entire LR(1) machine can be formed and the appropriate states merged or the states can be merged as the LR(1) machine is constructed.

The LALR(1) machine formed in this way is identical to the LR(0) finite state machine with the appropriate follow sets attached to each item. That is, there are exactly the same number of states and exactly the same items in each state. The parsing actions are determined exactly the same as for the LR(1) machine.

This suggests that it would be much more efficient to construct the LR(0) machine and compute the lookaheads needed in inconsistent states since the LR(0) machine will generally have considerably fewer states then the LR(1) machine. Thus,

considerable time and space is saved by approaching the problem in this way. This is exactly the approach taken by DeReemer and Pennello.

The expression grammar G1 is not only LALR(1), it is also SLR(1). A grammar is SLR(1) if there are no conflicts when the follow sets of the nonterminal symbols in the grammar can be used for the lookahead sets. The LR(0) machine is constructed and all indicated reductions $A \rightarrow \alpha\bullet$ must be legal for a ε Follow(A), where a is the next lookahead symbol. If a grammar is SLR(1), no lookaheads need to be computed as the follow sets of each nonterminal become the second part of the item, i.e. reduce by $A \rightarrow \alpha\bullet$, $\{a_1 \ldots a_n\}$ where a_i ε Follow(A). A shift action is exactly the same as for LR(1) or LALR(1).

The following two grammars will be used to illustrate DeReemer's and Pennello's method for computing lookaheads. The first,G2, is LALR(1) but not SLR(1). The second,G3, is not LALR(1). Their respective LR(0) finite state machines are included below, plus the finite state machine for G1. In all three cases the augmented production has been added.

G2 = (N, T, P, S) where
N = { G, E, T }, T = { =, a, +, * }, S = G,
G′= augmented production symbol, # = end of string symbol
P =

$G' \rightarrow G\#$
$G \rightarrow E = E$
$G \rightarrow a$
$E \rightarrow T$
$E \rightarrow E + T$
$T \rightarrow a$
$T \rightarrow T * a$

G3 = (N, T, P, S) where
N = { Y, Z }, T = { c, b }, S = Y,
Y′ = augmented production symbol, # = end of string symbol
P =

$Y' \rightarrow Y\#$
$Y \rightarrow c\ c\ Z\ b$
$Z \rightarrow \lambda$
$Z \rightarrow c\ Z$
$Z \rightarrow c\ Z\ b$

STATE 0

E′ → • E#
E → • T
E → • E A T
T → • F
T → • T M F
F → • (E)
F → • id
F → • intliteral

STATE 1

E′ → E • #
E → E • A T
A → • +
A → • −

STATE 2

E → T •
T → T • M F
M → • *
M → • /

STATE 3

T → F •

STATE 4

F → (• E)
E → • T
E → • E A T
T → • F
T → • T M F
F → • (E)
F → • id
F → • intliteral

STATE 5

F → id •

STATE 6

F → intliteral •

STATE 7

E → E A • T
T → • F
T → • T M F
F → • (E)
F → • id
F → • intliteral

STATE 8

A → + •

STATE 9

A → − •

STATE 10

E′ → E# •

STATE 11

T → T M • F
F → • (E)
F → • id
F → • intliteral

STATE 12

M → * •

STATE 13

M → / •

STATE 14

F → (E •)
E → E • A T
A → • +
A → • −

STATE 15

E → E A T •
T → T • M F
M → • *
M → • /

STATE 16

T → T M F •

STATE 17

F → (E) •

Figure 2.1 Finite State Machine for G1

STATE 0

$G' \rightarrow \bullet\ G\#$
$G \rightarrow \bullet\ E = E$
$G \rightarrow \bullet\ a$
$E \rightarrow \bullet\ T$
$E \rightarrow \bullet\ E + T$
$T \rightarrow \bullet\ a$
$T \rightarrow \bullet\ T * a$

STATE 1

$G' \rightarrow G \bullet\ \#$

STATE 2

$G \rightarrow E \bullet\ = E$
$E \rightarrow E \bullet\ + T$

STATE 3

$E \rightarrow T \bullet$
$E \rightarrow T \bullet\ * a$

STATE 4

$G \rightarrow a \bullet$
$T \rightarrow a \bullet$

STATE 5

$G' \rightarrow G\# \bullet$

STATE 6

$G \rightarrow E = \bullet\ E$
$E \rightarrow \bullet\ T$
$E \rightarrow \bullet\ E + T$
$T \rightarrow \bullet\ a$
$T \rightarrow \bullet\ T * a$

STATE 7

$E \rightarrow E + \bullet\ T$
$T \rightarrow \bullet\ a$
$T \rightarrow \bullet\ T * a$

STATE 8

$T \rightarrow T * \bullet\ a$

STATE 9

$G \rightarrow E = E \bullet$
$E \rightarrow E \bullet\ + T$

STATE 10

$T \rightarrow a \bullet$

STATE 11

$E \rightarrow E + T \bullet$
$T \rightarrow T \bullet\ * a$

STATE 12

$T \rightarrow T * a \bullet$

Figure 2.2 Finite State Machine for G2

STATE 0

$Y' \rightarrow \bullet\ Y\#$
$Y \rightarrow \bullet\ c\ c\ Z\ b$

STATE 1

$Y' \rightarrow Y \bullet \#$

STATE 2

$Y \rightarrow c \bullet c\ Z\ b$

STATE 3

$Y' \rightarrow Y\# \bullet$

STATE 4

$Y \rightarrow c\ c \bullet Z\ b$
$Z \rightarrow \bullet$
$Z \rightarrow \bullet\ c\ Z$
$Z \rightarrow \bullet\ c\ Z\ b$

STATE 5

$Y \rightarrow c\ c\ Z \bullet b$

STATE 6

$Z \rightarrow c \bullet Z$
$Z \rightarrow c \bullet Z\ b$
$Z \rightarrow \bullet$
$Z \rightarrow \bullet\ c\ Z$
$Z \rightarrow \bullet\ c\ Z\ b$

STATE 7

$Y \rightarrow c\ c\ Z\ b \bullet$

STATE 8

$Z \rightarrow c\ Z \bullet$
$Z \rightarrow c\ Z \bullet b$

STATE 9

$Z \rightarrow c\ Z\ b \bullet$

Figure 2.3 Finite State Machine for G3

CHAPTER 3

COMPUTATION OF LALR(1) LOOKAHEAD SETS

3.1 BACKGROUND

The method outlined in this section to compute the lookahead sets necessary to determine if a grammar is LALR(1) and incorporated into the accompaning computer program is due to DeReemer and Pennello.[1]

Recall that each inconsistent state in the LR(0) Finite State Machine requires lookahead information to resolve the shift-reduce or reduce-reduce conflict. The definition of the lookahead set in an inconsistent state q for the reduction involving the application of the production $A \rightarrow \omega$ is

$$LA(q, A \rightarrow \omega) = \{ a \varepsilon T \mid S \Rightarrow_r^+ \gamma A a w \text{ and } \gamma\omega \text{ accesses } q \}$$

Thus, the parse has preceded to state q, $\gamma\omega$ is on the stack, and LA are all possible terminal symbols which can follow γA in a rightmost sentential form. The item $A \rightarrow \omega \bullet$ will belong to state q. $\gamma\omega$ will be the viable prefix that state q recognizes.

Note in the above that A is a nonterminal in the grammar G. Thus there must be a transition GoTo(p,A) = q′ in the Finite State Machine (FSM) and also the GoTo Table for G. This is the nonterminal transition (p,A) in the FSM. It occurs after a reduction such as $A \rightarrow \omega$. The next step in the parse must be to shift an A. The question becomes : What terminal symbols can follow the nonterminal transition (p,A)?

DeReemer and Pennello have decomposed the computation of the above LA into the following four components.

1) Compute *Direct Read* sets for nonterminal transitions by inspecting the LR(0) machine.

---

[1]Frank DeReemer, Thomas Pennello,"Efficient Computation of LALR(1) Look-Ahead Sets,"*ACM Transactions on Program Languages and Systems*,Vol 4,No.4 (October 1982),pp.615-649.

2) Compute *Read* sets from the Direct Read sets and the *READS* relation. The READS relation is computed by inspecting the LR(0) machine for possible nullable nonterminal transitions.

3) Compute the *Follow* sets of nonterminal transitions from the Read sets.

4) Compute the LA from the Follow sets of nonterminal transitions.

It is important to notice at this point that a single reduce state such as State 5, State 10, and State 12 in the FSM for G2 does not qualify as an inconsistent state. There is no shift-reduce or reduce-reduce conflict. Thus no lookaheads should have to be computed for these reductions. In fact, it is unnecessary to calculate the lookaheads and the states themselves will actually be eliminated from the Parse Table. This is explained in Chapter 4.

The concept of lookahead can be further expanded to mean the following.

Let the symbol $[\gamma\omega]$ represent the path (sequence of states) taken to consume $\gamma\omega$, i.e. starting in state S_0 the path $S_0S_1 \ldots S_m$ is the series of state transitions to consume $\gamma\omega$. Hence there is a path starting at S_0 and ending ar $S_m = q$ with $\gamma\omega$ on the stack, or $\gamma\omega$ accesses q. Note that $\gamma\omega$ on the stack is equivalent to to the series of states $S_0 \ldots q$ being on the stack. Thus a definition for LA where [ ] = start state S_0, x = input string, $z \varepsilon T^*$, $\gamma\omega \varepsilon V^*$, $A \varepsilon N$ is

$$LA(q,A\rightarrow\omega) = \{a\varepsilon T \mid [\,]x \vdash^+ [\gamma\omega]az \vdash_{reduce} [\gamma A]az \vdash^+ [S\#]\lambda\}$$

The definition for lookahead given above is identical to the definition given earlier. The earlier definition relates LA to the rightmost derivative while the latter definition relates LA to the inverse of the rightmost derivative, i.e. a transformation from one configuration to the next in the parse of a sentence beginning in state S_0 with the sentence as input and ending with S# on the stack and no more input. The final step is to reduce by $S'\rightarrow S\#$, dropping back to state S_0 with S' the next symbol. This is

acceptance.

Thus the LALR(1) parser is identical to the LR(0) parser with the concept of reduce in an inconsistent state q. The concept of reduce in state q by production $A \rightarrow \omega$ ε P with lookahead $a \varepsilon T$ can be formulated as

$$\text{Reduce}(q,a) = \{A \rightarrow \omega \mid a \ \varepsilon \ LA(q, A \rightarrow \omega)\}$$

Relating this reduce(q,a) to the combined Parse Table (PAT) entry yields

$$PAT(q,a) = (r, A \rightarrow \omega) \text{ where } a \ \varepsilon \ LA(q, A \rightarrow \omega)$$

Thus, inorder to determine when to reduce(q,a), it is necessary to determine how to compute $LA(q, A \rightarrow \omega)$.

3.2 COMPUTATION OF LA

The calculation of LA is directly related to the concept of *follow sets* of nonterminal transitions. These are defined to be the Follow(p,A) where (p,A) is a nonterminal transition in the FSM. The definition of Follow(p,A) is

$$\text{Follow}(p,A) = \{ a \varepsilon T \mid [\gamma A] az \vdash^* [S\#] \text{ and } \gamma \text{ accesses } p \}$$

Thus 'a' are all the terminal symbols which can follow A in a righmost sentential form with prefix γ. Thus in state p there exists an item of the form $B \rightarrow \gamma \bullet Ax$ and $a = \text{First}_1(x)$. Stated in terms of the derivative

$$\text{Follow}(p,A) = \{ a \varepsilon T \mid S \Rightarrow^+ \gamma A a z \}$$

The LA set will be constructed from the union of these follow sets, or

$$LA(q, A \rightarrow \omega) = \bigcup \{\text{Follow}(p,A) \mid (p,A) \text{ is a transition and } p \text{ —}\cdots\overset{\omega}{\cdots}\rightarrow q\}$$

The symbol p —$\cdots\overset{\omega}{\cdots}\rightarrow$ q means there exists a series of single state transitions from $p \rightarrow r_0 \rightarrow r_1 \rightarrow \cdots \rightarrow q$ which consumes ω. In state q there exists an item of the form $A \rightarrow \omega \bullet$. The reason for the union is that state q can possibly be reached from various states to consume ω. Referring to Figure 3.1,[2] in state q after the reduction $A \rightarrow \omega$ the

parse pops back $|\omega|$ to the appropriate p_i and after reading A must be followed by the corresponding a_i.

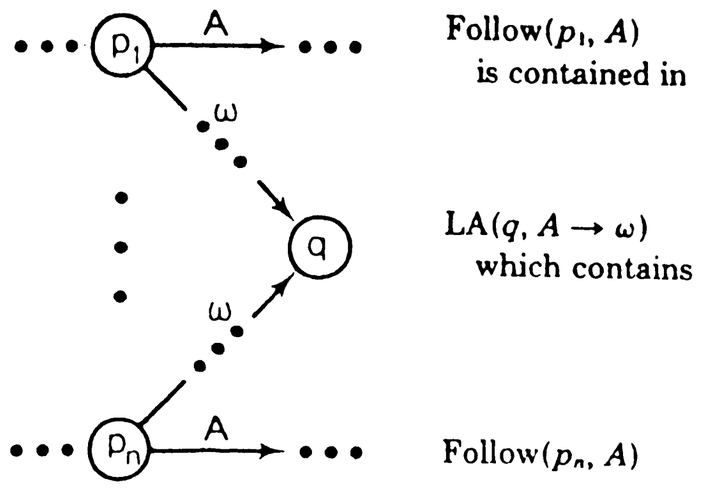


Figure 3.1 Lookahead Set In Terms Of Follow Sets

Thus, $LA(q,A\rightarrow\omega) = \{a_1, a_2 \cdots a_n\}$

The nonterminal transitions (p_i,A) needed for each $LA(q,A\rightarrow\omega)$ can be determined by defining the **lookback** relation. Thus

$(q,A\rightarrow\omega)$ **lookback** (p,A) iff $p \xrightarrow{\omega} q$.

In the above example the lookback for $(q,A\rightarrow\omega)$ contains the following ordered pairs (p_1,A), (p_2,A), $\cdots$ (p_n,A). Thus, each $(q,A\rightarrow\omega)$ must keep a record of all the nonterminal transitions in its lookback relation. In the computer program this is kept as a linked list.

Thus, if in inconsistent state q in Grammar G1, for example, there exists the item $E \rightarrow E\ A\ T\ \bullet$, we must create $(q,E\rightarrow E\ A\ T)$. The (p_i,E) are found by traversing the GoTo Table from state q back thru the states spelling out (backwards) T A E. This involves a recursive tree traversal since from state q, the string T A E may go back to many p_i's. There must exist a (p_i,E) when p_i is reached resulting from the

---

[2]Frank DeReemer,Thomas Pennello,"Efficient Computation of LALR(1) Look-Ahead Sets,"*ACM Transactions on Program Languages and Systems*,Vol 4,No.4(October 1982),p.621.

item $E' \rightarrow \bullet\ E\#$, $F \rightarrow (\ \bullet\ E\ )$, or $E \rightarrow \bullet\ E\ A\ T$ (which must exist in state p_i). Essentially, there must exist an item with an E after the •. The list of ordered pairs is created in this manner.

Applying the above definition for lookback to $LA(q,A\rightarrow\omega)$ produces

$$LA(q,A\rightarrow\omega) = \bigcup \{Follow(p,A) \mid (q,A\rightarrow\omega) \text{ lookback } (p,A)\}$$

The follow sets must now be determined.

3.3 INTERRELATION OF FOLLOW SETS

The follow sets of nonterminal transitions can be related via a new relation called includes. If $B \rightarrow \beta A\gamma$, $\gamma \Rightarrow^* \lambda$, and $p' \xrightarrow{\beta} p$ then

$Follow(p',B) \subseteq Follow(p,A)$.

Thus those symbols which can follow B in state p' can also follow A in state p. The diagram in Figure 3.2[3] illustrates this relationship.

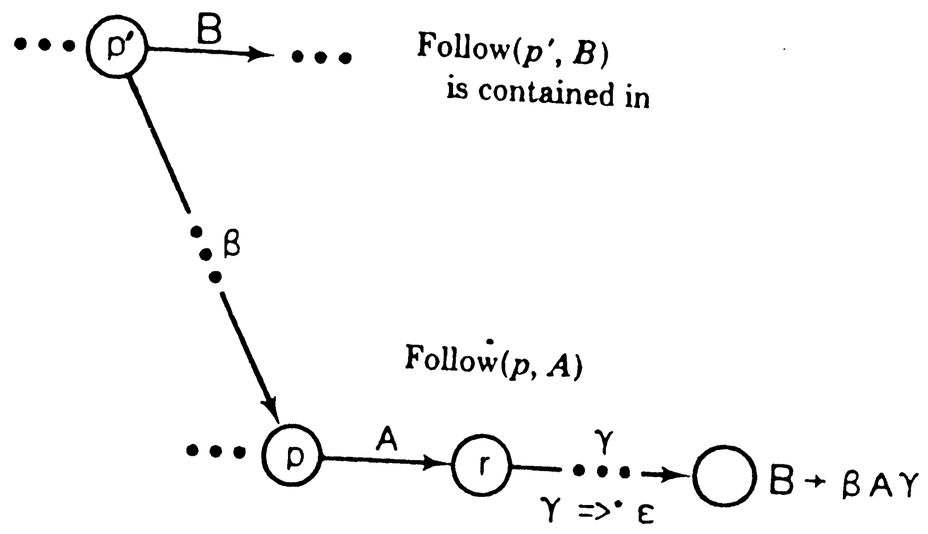


Figure 3.2 Interrelationships Among Follow Sets

In state p there must exist an item $B\rightarrow\beta\bullet A\gamma$ where $\gamma\Rightarrow^*\lambda$, or

(p,A) includes (p',B) iff $B \rightarrow \beta A\gamma$, $\gamma\Rightarrow^*\lambda$, and $p' \xrightarrow{\beta} p$.

---

[3]Frank DeReemer,Thomas Pennello,"Efficient Computation of LALR(1) Look-Ahead Sets,"*ACM transactions on Program Languages and Systems*,Vol 4,No.4(October 1982),p.621.

This means that

Follow(p',B) $\subseteq$ Follow(p,A) if (p,A) **includes** (p',B).

The **includes** relation is determined in the following manner.

For each set of nonterminal transitions (i,A) in the FSM, where i=state, AεN

a) Examine state i in the collection of item sets.

b) In state i there must exist an item of the form B$\rightarrow\beta\bullet$Aγ (or (i,A) would not exist). If γ is the empty string or $\gamma\Rightarrow^{*}\lambda$, add (i,A) **includes** (j,B) where j is determined exactly as in **lookback** by backtracking recursively in the GoTo Table $|\beta|$ states to get j and thus (j,B). Note that β can be the empty string.

c) To (i,A) add (j,B) by a linked list of **include** pairs.

Note that in state i there could be more than one item of the form B$\rightarrow\beta\bullet$Aγ. Thus, there could be other items C$\rightarrow\alpha\bullet$Aω. All must be examined for the **includes** relation.

It is very evident that after a nonterminal transition (p,A), all the terminal symbols which can be read in the next state must belong to the Follow(p,A). This leads to another definition the ***Read***(p,A) where Read(p,A) $\subseteq$ Follow(p,A).

3.4 DETERMINING READ(p,A)

The Read(p,A) is defined as the set of nonterminals which can be read before any phrases containing A are reduced. If there are no empty productions following the nonterminal transition (p,A), the Read(p,A) becomes the DR(p,A) where DR are the *direct read* symbols and are simply

$$DR(p,A) = \{\ a\varepsilon T \mid p \xrightarrow{A} q \xrightarrow{a}\ \}$$

The DR can be obtained very simply by inspecting the GoTo Table of the FSM and recording what terminal symbols can follow in state q where GoTo(p,A) = q. In

state q the possible terminal symbols are obtained directly from the GoTo Table. These come from the items in state q of the form $B \rightarrow \gamma A \bullet az$. Thus in state q the entry for PAT(q,a) = (s,i) = shift and GoTo state i.

Read(p,A) becomes much more complicated if there exists in state q some nullable nonterminal transitions such as (q,C) where $C \Rightarrow^* \lambda$, or, perhaps, a series of possible transitions through several states consuming $\gamma \rightarrow C_1C_2C_3$ and $\gamma \Rightarrow^* \lambda$. Diagrammatically this means

$$p \xrightarrow{A} q_1 \xrightarrow{\gamma} q_m \xrightarrow{a} \text{ and } \gamma \Rightarrow^* \lambda$$

This means that the Read(p,A) must be further defined to include a relation called the **reads** relation where

(p,A) **reads** (t,C) iff $p \xrightarrow{A} t \xrightarrow{C}$ and $C \Rightarrow^* \lambda$.

The **reads** relation computes those symbols which can be indirectly read after (p,A). Now Read(p,A) must be computed by

$$\text{Read}(p,A) = DR(p,A) \cup \bigcup \{ \text{Read}(t,C) \mid (p,A) \textbf{ reads } (t,C) \}$$

Consider Figure 3.3 below.[4] Since

(p,A) **reads** (q_0,B_1), i.e. $B_1 \Rightarrow^* \lambda$

(q_0,B_1) **reads** (q_1,B_2), i.e. $B_2 \Rightarrow^* \lambda$

.....

.....

(q_{m-2},B_{m-1}) **reads** (q_{m-1},B_m), i.e. $B_m \Rightarrow^* \lambda$

Thus a ε Read(p,A) since $DR(q_{m-1},B_m) \subseteq \text{Read}(p,A)$.

DeReemer and Pennello have written a very efficient algorithm called Digraph which traverses a graph inorder to calculate the Read(p,A) from the DR and the **reads** relation. This is included below.

---

[4]Jean-Paul Trembley, Paul G. Sorenson, *The Theory and Practice of Compiler Writing*,(1985), p.380.

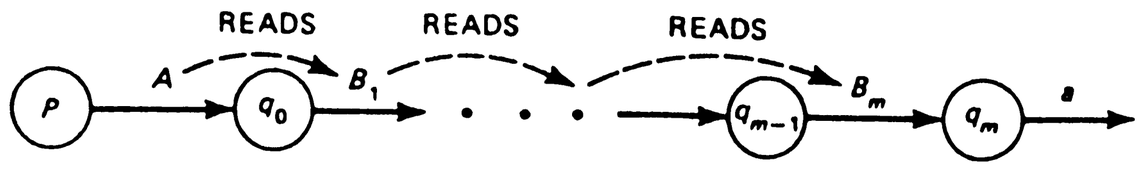


Figure 3.3 The READS relation

Thus the final formula to compute the Follow(p,A) is

$$\text{Follow}(p,A) = \text{Read}(p,A) \cup \bigcup \{ \text{Follow}(p',B) \mid (p,A) \text{ includes } (p',B) \}$$

The algorithm Digraph is also applied to calculate the Follow(p,A).

3.5 APPLYING ALGORITHM DIGRAPH

The following algorithm has as input a set X, a relation R(reads or includes), and a set-valued function F'(a function from X to sets). The output (Read or Follow) is the set valued function F such that

$$F(x) = F'(x) \cup \bigcup \{ F(y) \mid xRy \} \text{ where } x,y \epsilon X.$$

F(x) is computed by traversing the directed graph G = (X,R) induced by the relation R where X denotes the set of vertices in G and R denotes the edges.

The algorithm is applied twice in the program, as follows.

I) To compute the Read sets

NOTE: This is only necessary if there is a possibility of a reads relation, i.e. if there exists some nullable nonterminals, Otherwise, Read = DR and the algorithm is not applied.

X = set of nonterminal transitions = vertices

F' = DR = initial values for Read

R = reads relation = list of nullable nonterminals which can follow a nonterminal transition (p,A). This is obtained by examining the next state q for possible nullable nonterminals. Since so few reads relations actually exist in a grammar

this was made a local linked list in the program and was calculated as needed.

F = output = Read = initial values for Follow(p,A)

II) To compute Follow sets

X = set of nonterminal transitions

F′ = Read = initial values for Follow(p,A)

R = **includes** relation. This was obtained as described previously.

F = output = Follow

What follows is the algorithm as implemented in the program.

Stack is a stack containing elements of X, initially empty. N is a vector of integers (one to each nonterminal transition) with each initially set to 0. F(x) uses the same variable name each time, called ReadFollow , in which to accumulate results. ReadFollow contains Read or Follow depending on which traversal is called. Recall that if the **reads** relation is R, F′(x) = initial F(x) = initial ReadFollow = DR and if the **includes** relation is R, F′(x) = initial F(x) = initial ReadFollow which has been initialized by calculating Read.

Obviously, from the equations, Read is called first.

Algorithm Digraph (R);

```
1. Initialize
     N ← 0
     stack ← nil
2. compute F(x)
     for each x∈X
          if N[x] = 0 then
                    Traverse (x,R)
3. Return
```

The following computes F(x) given a vertex x of the graph. TOPV returns the top element of the stack. MIN returns the smallest of its two arguments. DepthOfStack keeps a constant record of the depth of the stack. y, element, and d are local variables. y and element are nonterminal transitions (εX) and d is an integer.

```
Algorithm Traverse (x, R)

1. Initialize
     Push(x)
     d ← DepthOfStack
     N[x] ← d
2. Complete the Closure
     If relation = reads then
          Compute reads relation locally
     Repeat for each y∈X such that xRy
          If N[y] = 0 then
               Traverse(y,R)
          N[x] ← MIN( N[x], N[y] )
          F(x) ← F(x) ∪ F(y)
     If N[x] = d then
          N[TOPV(stack)] ← ∞
          If x <> TOPV(stack) then
               F(TOPV(stack)) ← F(x)
          element ← POP(stack)
          Repeat while element <> x
               N[TOPV(stack)] ← ∞
               if x <> TOPV(stack) then
                    F(TOPV(stack)) ← F(x)
               element ← POP(stack)
3. Return
```

The vector N[x] serves three purposes. If N[x] = 0, it indicates that x has not been pushed on the stack yet. If $0 < N[x] < \infty$, x is under consideration and is still on the stack. If $N[x] = \infty$, the strongly connected component (SCC) of the vertex x has already been computed. Marking each vertex x avoids recomputation of F(x) if two vertices share the same child. The first time the vertex is encountered it is marked and marked vertices are never traversed again.

When Traverse pushes x on the stack, it records N[x] as the depth of the stack. It then traverses its subtrees (xRy). If an edge is ever encountered from a descendent 'd' to an ancestor 'a' already on the stack, the a and d and the intervening nodes on the stack are part of an SCC (there exists a path from a to d to a). The N[d] is minimized to N[a] to prevent d from being popped as the recursion unwinds.

3.6 STEPS TO COMPUTE LA

The following steps were applied to compute the LALR(1) lookahead sets from the LR(0) FSM.

1) Determine all nullable nonterminals in the grammar.

2) Determine all nonterminal transitions (p,A) from the LR(0) FSM.

3) Determine the inconsistent states and the production reductions for which we need to calculate lookaheads, i.e. LA(q,A→ω).

4) Determine the DR from the LR(0) FSM to initialize each ReadFollow for each nonterminal transition.

5) Determine the **lookback** relation for each LA(q,A→ω).

6) Determine the **includes** relation for all nonterminal transitions.

7) Apply Algorithm Digraph to **reads** to compute Read, if necessary.

8) Apply Algorithm Digraph to **includes** to compute Follow(p,A) for each ReadFollow sets for each nonterminal transition.

9) For each production LA union the follow sets in that productions **lookback** links.

10) Check for conflicts. If none exist, the grammar is LALR(1).

It is instructive to describe two basic structures used in the computation of LA. First, there is an array of production lookaheads. This array of records, called ProdLAS, consists of the following items.

ProdLAS = state number i
number j indicating the actual production applied
Pointer to **lookback** pairs
ProdLAS = (i, j, Ptr)

Second, there is an array of records, called NTTRANS, consisting of the following items.

NTTRANS = state number i
nonterm symbol XεN
StackDepth - N in Algorithm Digraph
ReadFollow = set of terminal symbols

Pointer to includes pairs
NTTRANS= (i, X, N, { follow symbols }, Ptr)

Since it would be a waste of storage to allocate a set of follow symbols for each item in the collection of item sets(the FSM), or even in the ProdLAS array, the follow sets are kept as a local variable and as the union of the follow sets is accomplished (step 9 above) the necessary items indicating reductions are added to the PAT. Consistency is checked at the same time. If an inconsistency is found (step 10 above) an appropriate message is written to a file. All inconsistencies are recorded.

CHAPTER 4

SINGLE REDUCE STATE ELIMINATION

4.1 LR(0) ROW ELIMINATION

A single reduce state has only one action associated with it - reduce by a particular production. In Figure 2.2 for G2 State 5, State 10, and State 12 are single reduce states. Most LR(0) machines have several single reduce states. Suppose the reduction is made in a single reduce state for any lookahead. If the lookahead is incorrect, the error will be detected as soon as an attempt is made to shift the incorrect lookahead. Consider the following:

The lookahead for a production is defined as

$LA(q,A\rightarrow\omega) = \bigcup \{ Follow(p,A) \mid (p,A)$ is a transition and $p \xrightarrow{\omega} q \}$ and

$Follow(p,A) = \{ a \varepsilon T \mid S \Rightarrow_r^* \gamma Aaz \}$

Suppose γ accesses state p_i and $\gamma\omega$ accesses state q as in Figure 3.1. Suppose q is a single reduce state. Allow the parser to perform the reduction $A\rightarrow\omega$ in state q without checking any lookaheads. After dropping to state p_i and reading A, the GoTo transition is made for $GoTo(p_i,A) = r$. In state r there must exist either a shift on symbol a or a reduce on symbol a. If r happens to be a single reduce state, reduction is made as above until a shift is called for. The point is that if a reduction is made with an invalid lookahead, the parser will halt as soon as it attempts to shift the lookahead.

Actually the LALR(1) parsing techniques really use only approximate lookaheads at all times. Consider Figure 3.1 again. Suppose $LA(q,A\rightarrow\omega) = \{a_1,a_2,a_3 \cdots a_n\}$. Suppose the next lookahead is a_j and the parser performs the reduction. Recall that the lookaheads in state q come from the union of the Follow(A) in state $p_1 \cdots p_n$. Suppose the transition in this case was $p_i \xrightarrow{\omega} q$ and a_j does not belong to Follow(A) in p_i. After the parser drops back to p_i, reads A, and then tries to

shift a_j, the error will be detected. So there is often a chance of doing an invalid reduction in a LALR(1) parse. The important point is that the parser will never do an invalid shift.

If you can reduce by any lookahead in a single reduce state, why go to the state at all?

4.2 SHIFT-REDUCE OR L-ACTION

So far we have shift(S) and reduce(R) actions in the parser. We will now replace any parse table entry that indicates a shift(S) followed by a GoTo state i where i is a single reduce state with an L-action and the production number that is in the single reduce state. Thus the parser will pop one less symbol off the stack and drop back to the same state as if it had shifted the terminal symbol onto the stack, gone to the single reduce state, and then done the reduction.

The removal of single reduce states is a simple operation. For this reason, no LA in single reduce states will be calculated, since the state is eliminated from the final Parse Table.

CHAPTER 5

APPLICATION TO GRAMMARS G2 AND G3

5.1 G2 APPLICATION

The finite state machine for G2 is given in Figure 2.2. This is directly transferable to the following GoTo Table. Note that the shift symbol S is used for clarity even though it is really part of the Parsing Action Table (and thus the final combined Parse Table).

ST	LOOKAHEADS							
	G	E	T	=	a	+	*	#
0	S1	S2	S3		S4			
1								S5
2				S6		S7		
3							S8	
4								
5								
6		S9	S3		S10			
7			S11		S10			
8					S12			
9						S7		
10								
11							S8	
12								

FIGURE 5.1 GOTO TABLE FOR G2

The list of nonterminal transitions for G2 is:

(0,G), (0,E), (0,T), (6,E), (6,T), (7,T)

There are only four inconsistent states. They are State 3, State 4, State 9, and State 11. Thus the LA to be computed are:

LA(3, E→T)

LA(4, G→a)

LA(4, T→a)

LA(9, G→E=E)

LA(11, E→E+T)

The DR symbols are obtainable directly from the LR(0) FSM. Since there are no nullable nonterminals in G2 they are the same as Read.

NTTRANS	DR	Read
(0,G)	{ # }	{ # }
(0,E)	{ = + }	{ = + }
(0,T)	{ * }	{ * }
(6,E)	{ + }	{ + }
(6,T)	{ * }	{ * }
(7,T)	{ * }	{ * }

The **lookback** relation for each LA(q,A→ω) is determined as follows:

(3, E→T), $|\omega| = |T| = 1$. Examining the GoTo Table yields two paths back. They are

$$[3] \xrightarrow{T} [6] \text{ and } [3] \xrightarrow{T} [0]$$

(4, G→a), and (4,T→a), $|\omega| = |a| = 1$. There is only one path back here. It is

$$[4] \xrightarrow{a} [0] \text{ for each}$$

(9, G→E=E), $|\omega| = |E=E| = 3$. There is only one path back here, also. It is

$$[9] \xrightarrow{E} [6] \xrightarrow{=} [2] \xrightarrow{E} [0]$$

(11, E→E+T), $|\omega| = |E+T| = 3$. There are two paths back. They are

$$[11] \xrightarrow{T} [7] \xrightarrow{+} [2] \xrightarrow{E} [0]$$

$$\boxed{11} \xrightarrow{T} \boxed{7} \xrightarrow{+} \boxed{9} \xrightarrow{E} \boxed{6}$$

Thus, the **lookback** pairs are:

LA	**lookback** pairs
(3,E→T)	(0,E), (6,E)
(4,G→a)	(0,G)
(4,T→a)	(0,T)
(9,G→E=E)	(0,G)
(11,E→E+T)	(0,E), (6,E)

Interpreting the **lookback** for LA(11,E→E+T) as an example yields the following: the terminal symbols which can follow E in state 0 and E in state 6 can also follow E+T in state 11.

The **includes** relation involves studying each nonterminal transition. Recall that

(p,A) **includes** (p′,B) iff B →βAγ, $\gamma \Rightarrow^* \lambda$ and p′ —$\overset{\beta}{\cdots\cdots}$→ p.

1) nonterminal transition (0,G): Examining state 0 for the items in which the G appears after the • yields the item G′→ •G#. Since a # symbol is after the G, G can never be follow by λ here so the **includes** relation is empty.

2) nonterminal transition (0,E): Examining state 0 in a similar manner yields the items G→•E=E and E→•E+T. Again, since the string "=E" or "+T" follow E here and neither can ever be λ the **includes** relation is empty.

3) nonterminal transition (0,T): This nonterminal transition comes from the two items E→•T and T→•T∗a. In the first item E→•T, the empty string follows T. Thus, using the **includes** formula listed above where β equals the empty string (i.e. $|\beta| = 0$) we stay in the same state which yields (0,T) **includes** (0,E). The second item yields nothing since "∗a" is never λ.

4) nonterminal transition (6,E): This nonterminal transition comes from the two items G→E=•E and E→•E+T. Since the empty string follows E above and $|\beta| = |\ \mathrm{E}=\ | = 2$, traverse back in the GoTo Table as follows

$$[6] \xrightarrow{=} [2] \xrightarrow{E} [0].$$

Thus, (6,E) includes (0,G). The second item yields nothing since "+T" is never empty.

5) nonterminal transition (6,T): The two items important here are E→•T and T→•T∗a. The first item yields (6,E), or (6,T) includes (6,E). The second item yields nothing.

6) nonterminal transition (7,T): This transition comes from E→E+•T and T→•T∗a. Only the first is important. Here $|\beta| = |\ E+\ | = 2$. Going back two states yields

$$[7] \xrightarrow{+} [2] \xrightarrow{E} [0]$$

$$[7] \xrightarrow{+} [9] \xrightarrow{E} [6]$$

Thus, (7,T) includes (0,E) and (6,E)

Interpreting includes using 6) above yields: The terminal symbols which can follow (0,E) and (6,E) can also follow (7,T).

Next compute

$$\text{Follow}(p,A) = \text{Read}(p,A) \cup \bigcup \{\text{Follow}(p',B) \mid (p,A) \textbf{ includes } (p',B)\}$$

NTTRANS	DR-READ	INCLUDES	FOLLOW
(0,G)	{#}	nil	{#}
(0,E)	{= +}	nil	{= +}
(0,T)	{∗}	(0,E)	{∗ = +}
(6,E)	{+}	(0,G)	{+ #}
(6,T)	{∗}	(6,E)	{∗ + #}
(7,T)	{∗}	(0,E), (6,E)	{∗ = + #}

Now apply

$$\text{LA}(q,A\rightarrow\omega) = \cup \{\text{Follow}(p,A) \mid (q,A\rightarrow\omega) \textbf{ lookback } (p,A)\} \text{ to yield}$$

$\text{LA}(3,E\rightarrow T) = \text{Follow}(0,E) \cup \text{Follow}(6,E) = \{= +\} \cup \{+ \#\} = \{= + \#\}$

$\text{LA}(4,G\rightarrow a) = \text{Follow}(0,G) = \{\#\}$

$\text{LA}(4,T\rightarrow a) = \text{Follow}(0,T) = \{* = +\}$

$\text{LA}(9,G\rightarrow E=E) = \text{Follow}(0,G) = \{\#\}$

$\text{LA}(11,E\rightarrow E+T) = \text{Follow}(0,E) \cup \text{Follow}(6,E) = \{= +\} \cup \{+ \#\} = \{= + \#\}$

The follow symbols for the reductions in single reduce states are never

calculated since they will be eliminated by a shift-reduce(L) action.

The following conflicts have been resolved. In state 3 reduce by E→T with lookahead {= + #}. With a "*" shift. In state 4 with lookahead {#} reduce by G→a, with lookahead {* = +} reduce by T→a. In state 9 with lookahead {#} reduce by G→E+E, with lookahead "+", shift. In state 11 with lookahead {= + #} reduce by E→E+T, with lookahead "*", shift.

Adding the appropriate L and R actions gives the final Parse Table for G2 shown in Figure 5.2.

ST	LOOKAHEADS							
	G	E	T	=	a	+	*	#
0	S1	S2	S3		S4			
1								L1
2				S5		S6		
3				R4		R4	S7	R4
4				R6		R6	R6	R3
5		S8	S3		L6			
6			S9		L6			
7					L7			
8						S6		R2
9				R5		R5	S7	R5

Figure 5.2 Final Parse Table For G2

Accept in this combined Parsing Action Table and GoTo Table comes after the L1 in State 1. The parse shifts the # symbol, reduces by G′→G#, and drops back

to State 0. The entry for PAT(0,G′) is accept. This is accept in every Parse Table the program creates. It is added in the shift-reduce parser program to every parse table.

It is interesting to note that G2 is not SLR(1) so the follow symbols of each nonterminal can not be used for the lookahead set. The follow sets of G = {#} and T = {= + # *}. Thus in state 4 if a "#" were the next input symbol the parser would not know what reduction to do. Using the LALR(1) techniques resolves this conflict, since in state 4 T→a cannot be followed by a "#". This can be seen by observing that State 4 can only be reached from State 0. In state 0 the item T→•a comes from expanding G→•E=E via E→•T, E→•E+T, T→•a, T→•T*a. Thus the follow of T here can only be { * = + }. This, of course, is exactly what DeReemer's and Pennello's method calculated for the follow set.

5.2 G3 APPLICATION

The finite state machine for G3 is given in Figure 2.3. The GoTo Table in Figure 5.3 represents the FSM for G3.

The list of the nonterminal transitions for G3 is:

(0,Y), (4,Z), (6,Z).

There are three inconsistent states. They are State 4, State 6, and State 8. The LA to be calculated are

LA(4, Z→λ), LA(6, Z→λ), LA(8, Z→ cZ).

The DR symbols and the Read symbols are computed as follows. Note that after nonterminal transition (0,Y), the parser goes to State 1. State 1 has no transition on Z (the only λ nonterminal), so the **reads** relation for (0,Y) is nil. After (4,Z) the parser goes to State 5 which also has no transition on Z. The same is true for (6,Z). Thus all the **reads** relations are nil and Read = DR.

ST	LOOKAHEADS				
	Y	Z	c	b	#
0	S1		S2		
1					S3
2			S4		
3					
4		S5	S6		
5				S7	
6		S8	S6		
7					
8				S9	
9					

Figure 5.3 GoToTable for G3

The following chart is directly obtainable from the above GoTo Table

NTTRANS	DR	**reads**	Read
(0,Y)	{ # }	nil	{ # }
(4,Z)	{ b }	nil	{ b }
(6,Z)	{ b }	nil	{ b }

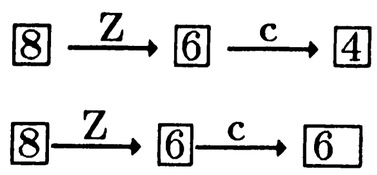
The **lookback** relation for each LA(q,A→ω) is determined as follows.

(4, Z→λ), $|\omega| = |\lambda| = 0$, so (4, Z→λ) **lookback** (4,Z)

(6, Z→λ), $|\omega| = |\lambda| = 0$, so (6, Z→λ) **lookback** (6,Z)

(8, Z→cZ), $|\omega| = |$ cZ $| = 2$. Examining the GoTo Table yields two paths back.

They are



Thus the **lookback** pairs are

LA	**lookback**
(4,Z→λ)	(4,Z)
(6,Z→λ)	(6,Z)
(8, Z→cZ)	(4,Z), (6,Z)

The **includes** relation is calculated as before to yield the following

1) nonterminal transition (0,Y): Examining State 0 yields the item Y′→ •Y#. Obviously Y can never be followed by λ so the **includes** relation is empty.

2) nonterminal transition (4,Z): The only item relevant is State 4 is Y→cc•Zb. Again, since Z is never followed by λ here the **includes** relation is empty.

3) nonterminal transition (6,Z): Two items are of importance here, Z→c•Z and Z→c•Zb. In the former item Z is followed by the empty string and the **includes** relation yields the two paths $\boxed{6} \xrightarrow{c} \boxed{4}$ and $\boxed{6} \xrightarrow{c} \boxed{6}$

Thus, (6,Z) **includes** (4,Z) and (6,Z).

Computing the Follow as before results in the following chart

NTTRANS	READ	INCLUDES	FOLLOW
(0,Y)	{ # }	nil	{ # }
(4,Z)	{ b }	nil	{ b }
(6,Z)	{ b }	(4,Z), (6,Z)	{ b }

Taking the union of the Follow in the **lookback** yields

LA(4, Z→λ) = Follow(4,Z) = { b }

LA(6, Z→λ) = Follow(6,Z) = { b }

LA(8, Z→cZ) = Follow(4,Z) ∪ Follow(6,Z) = { b }

In State 4 and State 6 we reduce by Z→λ if "b" is the next input symbol. There is no conflict in either of these two states. However, in State 8 there is definitely a conflict. When the next input symbol is a "b", the parser calls for a reduction by Z→cZ and a shift because of Z→cZ • b. State 8 is still inconsistent so the grammar is not LALR(1).

Using LR techniques would not have resolved any conflicts in the grammar. The grammar is not LR(k) for any k and can not be parsed by any LR technique. In fact, DeReemer and Pennello have included the following conjector in their article which exactly applies here:

Let (p,A) be a nonterminal transition that is in a nontrivial SCC of the digraph induced by the includes relation. Then the corresponding grammar is not LR(k) for any k if Read(p,A) $\neq \emptyset$.

Note that (6,Z) includes (4,Z) and (6,Z). Thus there is a nontrivial SCC of the digraph. Also note that Read(6,Z) = { b } which $\neq \emptyset$. Not only is the grammar not LALR(1) but it is not LR(k) for any k.

The Parse Table for G1 can be generated by these exact techniques or by simply using the follow symbols of its nonterminals for the lookahead sets. Both methods generate the same follow sets since G1 is not only LALR(1) but is also SLR(1). The final Parse Table for G1 is given in Chapter 6. The diagnostic output for G1, as calculated by the program, is included in Appendix A.

CHAPTER 6

OPTIMIZING PARSE TABLES

6.1 SPARSE MATRIX REPRESENTATION

All parsing action tables and goto tables have many error entries in them. The two-dimensional matrix representation which I have been using to demonstrate the tables is very efficient for table lookup but very space consuming. G1, G2, and G3 are very small grammars but typical programming languages contain hundreds of states. The simple grammar PL0 which has 51 tokens (terminal and nonterminal symbols) and 46 productions (with the augmented production added) generates 88 states for the LR(0) machine. This would require a 51 × 88 matrix, and PL0 is much smaller then most programming languages.

One way to reduce the size of the parse table is to encode the table using a sparse matrix. The sparse matrix will be slower in table lookup but will require much less space since all error entries will be eliminated from the table. A sparse matrix with pointers is used to represent the parse table in the computer program which generates the LALR(1) parse table. The second program (the shift-reduce parser) uses a one-dimensional array to represent the table created by the first program. There are several reasons for using different representations.

As was shown in G2, the final parse table is smaller then the original parse table since single reduce states are eliminated. The final parse table is also static, i.e. it does not change. The original parse table is constantly being changed as R and L actions are inserted into it and single reduce states are eliminated from it. Insertions and deletions are easier to do with pointers. Also, the first program must traverse the table in both directiions - i.e. across the rows (states) and down the columns (vocab symbols). This would be extremely inefficient in the one-dimensional array

representation. Also, the constant table lookup which the shift-reduce parser must do is much more efficient with the one-dimensional array then with a pointer matrix. Since the table created by the first program is input to the second program, there is no need to use the same representation.

The program which generates the LALR(1) parse table uses the following representation:

GoTo Function = array[state] of ParsePtr

Down Function = array[vocab] of ParsePtr

Parse Ptr is a pointer to the following parse record:

```
ParseRec = record
             StNum : state
             VocabNum : vocab
             Action : PACRec
             NextRec : ParsePtr
             DownRec : ParsePtr
             end
```

where Action consists of one of the following
(Si) = shift and go to state i
(Rj) = reduce by j production
(Lj) = L-reduce by j production

The reason for the two sets of pointers (both across and down the matrix) is that sometimes it is necessary to traverse the matrix down a vocabulary symbol through the states rather then across a state through the vocabularly symbols. This is the case when computing the **lookback** and the **includes** relations. The most efficient way to do this traversal is by including a second set of pointers.

Any entry not included in the sparse matrix is, of course, an error condition.

In the shift-reduce parser two one-dimensional arrays are actually used to represent the parse table. The first array references the location of the start of each state in the parse table array. The parse table array consists of records containing the following two items: a vocab symbol, and an Action record which is identical to the one

above except that an Accept action is now added. As mentioned earlier, Accept is always the first entry in the final Parse Table. In state 0 with the next input symbol the augmented symbol, the action is to accept. The shift-reduce parser program automatically inserts this entry as the first item in each Parse Table. When searching the parse table for a particular entry the program does a binary search from the start to the finish of the entries pertinent to the state being searched. Any entry not included is, as before, an error.

The final parse tables for G1 and G2 are shown in section 6.2 and 6.3.

6.2 DEFAULT PARSE TABLE ENTRIES

Observe that in Figure 5.2 of the final parse table for G2, state 3, state 4, and state 9 are *almost* single reduce states. Each contain a reduction by one production on several (three here) lookahead symbols. Since reductions can be made in single reduce states without examining any lookaheads, thereby delaying error detection but not eliminating it, why not do something similar in a state in which there are several reduction entries for the same production number? This is called a default parse table entry.

The default parse table entry must always be a reduction or an error entry. It can never involve a shift action. Therefore, it can never involve an L-action. The reason for the introduction of the error entry is that if we are going to add a default entry, it is easier to have one all the time, which means sometimes the only possible entry will be an error.

Referring back to Figure 5.2, state 0 would contain four shift entries plus a default error entry. State 1 contains one L-entry plus a default error entry. State 2 contains two shift entries plus a default error entry. State 3 contains one shift entry

plus a default reduction entry. The rest of the table is calculated similarly.

The parse table which the shift-reduce parser works with is shown below.

State 0, default = error, (G′, accept), (G, S1), (E, S2), (T, S3), (a, S4)

State 1, default = error, (#, L1)

State 2, default = error, (=, S5), (+, S6)

State 3, default = R4, (*, S7)

State 4, default = R6, (#, R3)

State 5, default = error, (E, S8), (T, S3), (a, L6)

State 6, default = error, (T, S9), (a, L6)

State 7, default = error, (a, L7)

State 8, default = error, (+, S6), (#, R2)

State 9, default = R5, (*, S7)

In larger grammars, such as PL0, several states have 10 - 15 entries for one reduction. The space savings is more apparent then in a small grammar such as G2.

As with single reduce states, if a default reduction is performed with an invalid lookahead, the error will be detected as soon as an attempt is made to shift the lookahead.

6.3 SINGLE-PRODUCTION ELIMINATION

The above techniques help to reduce the size of the parse table in LALR or LR parsing. Single-production elimination helps to *increase* the speed of the parse.

Many programming-language grammars contain productions of the form A→B, where A,B ε N. A good example of this is the expression grammar G1. This grammar contains E→T and T→F where E,T,F ε N. Two other unit productions in the grammar are F→id and F→intliteral, where id, intliteral εT. Suppose as the grammar parses, it

produces a chain of unit reductions. An example of this would be a parse of the simple expression 10#. The parser would reduce as follows: F→intliteral, T→F, E→T. It would obviously speed up the parse to simply reduce by E→intliteral.

The LR(0) machine for grammar G1 is shown in Figure 2.1. Adding the follow symbols produces:

STATE 0

E′ → • E#, {λ}
E → • T, {+ − #}
E → • E A T , {+ − #}
T → • F , {+ − * / #}
T → • T M F , {+ − * / #}
F → • (E) , {+ − * / #}
F → • id , {+ − * / #}
F → • intliteral , {+ − * / #}

After shifting an id or an intliteral the possible lookaheads are {+ − * / #}. If the lookahead is * or /, id or intliteral would be reduced to F, then F to T. If +, −, or # is the lookahead, the chain is F→ id | intliteral, T→F, E→T. The chain could be shortened by including in the next state the pseudo production T→id | intliteral for a *, / lookahead and E→id | intliteral for a +, −, # lookahead.

The obvious problem with the above is two-fold. First, DeReemer and Pennello's method does not calculate all the lookaheads for each item in the LR(0) machine. Second, the next state in the LR(0) machine is a single-reduce state and single-reduce states are eliminated from the final parse table. These are very necessary space saving techniques and are more important then the slight time-saving achieved by keeping the state and essentially changing it from a single reduce state. DeReemer's and Pennello's method does not support this technique for single-production elimination.

In some states a chain of reductions occurs independently of lookaheads. This

can very easily be incorporated into the program.

Recall that the LR(0) machine for G1 contains 18 states. The final parse table contains only 8 states, as shown in Figure 6.1 below. Rewriting and numbering the productions gives:

1) E′ → E#
2) E → T
3) E → E A T
4) T → F
5) T → T M F
6) F → (E)
7) F → id
8) F → intliteral
9) A → +
10) A → −
11) M → *
12) M → /

ST	LOOKAHEADS													
	E	T	F	A	M	(	)	+	−	*	/	id	int	#
0	S1	S2	L4			S3						L7	L8	
1				S4				L9	L10					L1
2					S5		R2	R2	R2	L11	L12			R2
3	S6	S2	L4			S3						L7	L8	
4		S7	L4			S3						L7	L8	
5			L5			S3						L7	L8	
6				S4			L6	L9	L10					
7					S5		R3	R3	R3	L11	L12			R3

Figure 6.1 Parse Table For G1

Note that in state 0 there is a chain reduction starting with L7 and L8. Thus with an id as the next lookahead perform F→id, T→F. With an intliteral perform F→intliteral, T→F. This could be optimized by performing the pseudo reduction T→id

or T→intliteral. A similar situation arises in state 3 and state 4.

To accomplish the pseudo reduction above, it is necessary to create two new productions. They are

13) T → id

14) T → intliteral

The new parse table utilizing these productions is the same as Figure 6.1 except that the entry for PAT(0,id) = L13, PAT(0,int) = L14, PAT(3,id) = L13, PAT(3,int) = L14, PAT(4,id) = L13, and PAT(4,int) = L14.

The parse table used by the shift-reduce parser (with default entries added) is as follows:

State 0, default = error, (E′, accept), (E, S1), (T, S2), (F, L4), ((, S3), (id, L13), (int, L14)

State 1, default = error, (A, S4), (+, L9), (−, L10), (#, L1)

State 2, default = R2, (M, S5), (*, L11), (/, L12)

State 3, default = error, (E, S6), (T, S2), (F, L4), ((, S3), (id, L13), (int, L14)

State 4, default = error, (T,S7), (F, L4), ((, S3), (id, L13), (int, L14)

State 5, default = error, (F, L5), ((, S3), (id, L7), (int, L8)

State 6, default = error, (A, S4), ( ), L6), (+, L9), (−, L10)

State 7, default = R3, (M, S5), (*, L11), (/, L12)

The original parse table had the potential of 18 states × 15 vocab symbols, or 270 entries. The above parse table has 43 entries.

6.4 SIZE OF PL0 PARSE TABLE

As stated earlier, the LR(0) machine for PL0 has 88 states. Since there are 51 vocab symbols the parse table if stored as a two-dimensional matrix would contain

4488 entries. Since the final parse table has only 51 states, this would reduce to 2601 entries.

Using the sparse matrix representation as described above with default reduction entries the parse table which the shift-reduce parser works with for PL0 contains 235 entries. This is a dramatic reduction and indicates that normal programming language parse tables can be stored in a reasonable amount of space.

It is interesting to note that PL0 has 10 nullable nonterminables and 66 nonterminal transitions. Of these 66 nonterminal transitions only 23 have entries in the **reads** relations. This reinforces, as DeReemer and Pennello suggest, that since so few **reads** relations actually exist in a grammar it saves space to make the **reads** relation a local variable in the Digraph procedure and compute the relation as needed.

PL0 has 20 inconsistent states in the LR(0) machine and 20 lookahead productions to compute the follow symbols for. It is LALR(1) as was to be expected, since it is also LL(1).

CHAPTER 7

GENERALIZATIONS

With the introduction of LALR(1) parsing techniques, the size of the parse table needed to parse LR grammars is greatly reduced. Since all LL(1) grammars are LR(1) and almost all are LALR(1), it has become reasonable to consider LALR(1) parsing techniques as a viable alternative to LL(1) techniques.

One advantage to LALR grammars is that, unlike LL grammars, they allow left recursion and productions which share a common prefix. It is generally easier to put a grammar into LALR form then into LL form. Often LALR grammars are easier to write, and frequently they are easier to read. In short, LALR techniques can handle a broader class of grammars then LL techniques.

In constructing an LALR(1) parse table considerable storage is required for the collection of item sets. However, since DeReemer's and Pennello's method only utilizes the LR(0) machine even the space needed to store the collection of sets is reduced. With the utilization of sparse matrix techniques for the parse tables, the size of these tables becomes less significant.

The use of pointers while generating the parse table may slow the construction of the table, but this initial program should only have to be run once. The actual shift-reduce parser is an extremely simple concept and very easy to implement. The use of a one-dimensional array and a binary search routine for the parse table makes the speed of the parser comparable to LL parsing techniques.

Of all the methods used to generate lookahead symbols (such as compatible state merger and propagating symbols through the states from item to item), DeReemer's and Pennello's method definitely seems to be the most efficient both in time and space considerations. Once the concepts are understood, it adapts very well to

implementation on a computer.

CHAPTER 8

PROGRAM IMPLEMENTATION

Both programs, the LALR(1) table generator and the shift-reduce parser are written in Turbo 4 Pascal.

8.1 IMPORTANT DATA STRUCTURES

Inorder to include any of the source code it is necessary to describe in detail the commonly used data structures and constant definitions. The constant definitions apply to both programs and may be changed as needed.

8.1.1 CONSTANT DEFINITIONS

```
MaxVocabSym = 101;          (* max # of Vocab Symbols *)
MaxNumProd = 50;            (* max # of productions *)
MaxProdLen = 7;             (* max length of each production *)
MaxLenVocabTab = 300;       (* max # of chars in all ter and nt combined *)
MaxStateNum = 250;          (* max # for collection of sets *)
MaxNTTrans = 125;           (* max # of non terminal transitions -
                               approximately 1/2 of MaxStateNum  *)
MaxProdLA = 65;             (* max # of production lookaheads to be
                            calculated-approximately 1/4 of MaxStateNum *)
MaxKernelNum = 400;         (* max # of kernel items in all sets combined *)
```

8.1.2 NONTERMINAL AND TERMINAL SYMBOLS

As the nonterminal and terminal symbols are entered into the initial program they are stored in a VocabTab and a 1-1 correspondence is set up between each symbol and a number. Thus,

```
type
  vocab = 1 .. MaxVocabSym;(*number representation of grammar symbols *)
  VocabSymbols = packed array [ 1 .. MaxLenVocabTab ] of char;
  VocabInts = packed array [ 1 .. MaxVocabSym ] of 1 .. MaxLenVocabTab;
var
  VocabTab : VocabSymbols; (* table of all symbols possible in grammar *)
  VocabInt : VocabInts;    (* 1-1 correspondence between grammar
                              symbols and integers *)
  LastNonTerm : vocab;     (* last place in VocabInt for NT symbols *)
  LastTerm : vocab;        (* last place in VocabInt for ter symbols *)
```

All the symbols are kept in the VocabTab. VocabInt[I] represents the vocab symbol I and references the start of the symbol in VocabTab. Thus the symbol represented by I starts in the VocabTab at VocabInt[I] for its character representation and ends at VocabInt[I + 1] - 1. Other then input and output, both programs work only with the number representation of each terminal and nonterminal symbol. For the expression grammar G1 VocabTab is

AUGSYMETFAM()+-*/IDINTLITERAL#

and the VocabInt table is

VocabInt(1) = 1	AUGSYM
VocabInt(2) = 7	E
VocabInt(3) = 8	T
VocabInt(4) = 9	F
VocabInt(5) = 10	A
VocabInt(6) = 11	M
VocabInt(7) = 12	(
VocabInt(8) = 13	)
VocabInt(9) = 14	+
VocabInt(10) = 15	-
VocabInt(11) = 16	*
VocabInt(12) = 17	/
VocabInt(13) = 18	ID
VocabInt(14) = 20	INTLITERAL
VocabInt(15) = 30	# (* end of string symbol *)

8.1.3 PRODUCTION REPRESENTATION

As the productions for the grammar are entered, the parse table generator checks that the left hand side of the production is a nonterminal symbol and also checks that all symbols have been entered into VocabTab, and correspondingly into VocabInt. If an unknown symbol is encountered the program terminates with an error message.

The important structures here are

```
type
   Prod = 1 .. MaxNumProd;          (* production # *)
   OneProdRec = record
                   LHS : vocab;       (* left hand nonterminal sumbol*)
                   NumElem : 0 .. MaxProdLen;    (*# of symbols in RHS *)
```

```
                    RHS: array [ 1 .. MaxProdLen ] of vocab;(*symbols in RHS *)
                    end;
  ProRecords = array [Prod] of OneProdRec;
var
  Productions: ProRecords;
```

An example for grammar G1 is:

```
augsym → E#
  Productions[1].LHS = 1
  Productions[1].NumElem = 2
  Productions[1].RHS[1] = 2
  Productions[1].RHS[2] = 15

E → T
  Productions[2].LHS = 2
  Productions[2].NumElem = 1
  Productions[2].RHS[1] = 3
```

8.1.4 COLLECTION OF SETS

The collection of item sets is represented by

1) a base table indexed by the state number which contains the start of the kernel items for that particular state in the array of kernel items and a pointer to the closure items for the state. The closure items are kept in a pointer list because many states contain only kernel items.

2) An array of all kernel items in all the states.

The important data structures here are:

```
type
  SetRec = record                    (* an item is a
             ProdNum : Prod;          production # and a
             DotPlace: 0 .. MaxProdLen;      dot position *)
           end;
KernelSet = array[ 0 .. MaxKernelNum] 0f SetRec;
ptr = ^ClosureRec;
KernelNums = 0 .. MaxKernelNum;
BaseTab = array [ 0 .. MaxStateNum ] of
             record
               KernelRef : KernelNums;  (* start of kernel elements *)
               ClosurePtr : ptr;          (* pts to closure of each st *)
             end;
ClosureRec = record                 (* closure for each st *)
             ThisSet : SetRec;
             ClosurePtr : ptr;
```

```
                    end;
var
   KernelTab : BaseTab; (* table of references to st of each kernel set for each
                           state *)
   KernelSets: KernelSet;    (* array of kernel items *)
```

As an example of the collection of sets, consider state 1 in the LR(0) machine for grammar G1. The productions for grammar G1 are shown on page 5. State 1 is shown in Figure 2.1. Since state 0 had only one kernel item , the kernel items for state 1 will start in 1. There are two kernel items, E′→ E • # and E → E • A T. The closure items are A→•+ and A→•−. Thus,

KernelTab[1].KernelRef = 1
KernelTab[1].ClosurePtr → (9,0) → (10,0) → nil
KernelSets[1].ProdNum = 1
KernelSets[1].DotPlace = 1
KernelSets[2].ProdNum = 3
KernelSets[2].DotPlace = 1
where (9,0) and (10,0) above represent the ProdNum and the DotPlace in the closure records.

All closure item records have 0 as the dot position, so it would be possible not to include this in a closure record. However, the program works with closure and kernel items in the same procedures, so it is more convenient to record the dot position.

8.1.5 COMBINED PARSE TABLE

The parse table in the initial program is created as a sparse matrix with pointers. As explained in chapter 6, inorder to traverse the parse table in both directions, two pointers are kept in each parse record, one for across the state and one for down the vocab symbols. The important structures here are

```
type
   optype = (R, S, L, AC)
   PACRec = record
                  case op : optype of
                     R, L : (num1: prod);(* reduce by num1 *)
                     S : (num2 : state);       (* shift and goto num2 *)
                     AC : ();                  (* not inc until 2nd program *)
               end;
   ParsePtr = ^ ParseRec;
   ParseRec = record
```

```
                    StNum : state;
                    VocabNum : vocab;
                    Ref : NumNTTrans;  (* used in Digraph algorithm *)
                                       (* ref non terminal trans array *)
                    Action : PACRec;
                    NextRec : ParsePtr;  (* across matrix *)
                    DownRec : ParsePtr;  (* down matrix *)
                    end;
  GoToFunction = array[state] of ParsePtr;
  DownFunction = array[vocab] of ParsePtr;
var
  GoToTab : GoToFunction;   (* used to create combined parse table *)
  DownTab : DownFunction;  (* used in traversal for includes and lookback *)
```

The parse table in the shift-reduce parser is stored in two separate one-dimensional arrays. The first GoToTab contains 1) a reference to the start of the entries for each state in the second table and 2) the default entry for the state. The second table PAT contains the actual parsing action table entries , excluding the default entry. The default entry is actually recorded as a reduction by a number which is greater then the number of productions in the grammar. This indicates an error action. Thus,

```
const
  MaxPATEntry = 750 ;              (* max entries in PAT *)
type
  OpType = (R, S, L, AC, ER );
  PACRec = record
                case op : optype of
                  R, L : (num1 : prod);
                  S : (num2 : state );
                  AC : ();
                  ER : ();
                end;
  ParseRec = record
                VocabNum : vocab;
                Action : PACRec;
                end;
  ParseRecs = array[ 1 .. MaxPATEntry] of ParseRec;
  ParseRef = record
                first : integer;       (* location of start of st in GoToTab *)
                default : prod;        (* default reduction # or error indicator *)
                end;
  GoToFunction = array[state] of ParseRef;
var
  GoToTab : GoToFunction;
```

```
PAT : ParseRecs;
```

Thus to reference the entries for state I look at PAT[GoToTab[I].first] for the first action record up to and including PAT[GoToTab[I + 1].first] -1.

8.1.6 CALCULATING LOOKAHEAD SYMBOLS

The structures for the nonterminal transitions and the structures for the production lookaheads (LA) are as follows:

```
type
  TraPtr = ^ IncList;
  IncList = record    (* used in lookback, includes, and reads relation *)
                Ref : NumNTTrans; (* location in nonterm trans array *)
                Next : TraPtr;
                end;
  NTTransition = record
                  StNum : state;
                  NonTerm : vocab;
                  StDepth : integer; (* stack depth in Digraph algorithm *)
                  ReadFollow : set of vocab;
                  Next : TraPtr;   (* for includes *)
                 end;
  ProdLookAhead = record
                   ProdStNum : state;
                   ProdRed : Prod;
                   Next : TraPtr ; (* for Look-Back *)
                  end;
  NTTran = array[ NumNTTrans ] of NTTransition;
  ProdLA = array[ NumProdLA ] of ProdLookAhead;
  DigraphStack = ^ StackRec;
  StackRec = record     (* stack of nonterm trans in digraph algorithm *)
                Num : NumNTTrans;
                Next : DigraphStack;
                end;
var
  NTTrans : NTTran;   (* nonterm transitions plus includes relation *)
  ProdLAS : ProdLA;       (* Production reductions in inconsistent states
                             plus lookback relation *)
  LastNtTrans : NumNTTrans;
  LastProdLa : NumProdLA;
  stack : DigraphStack;
```

8.2 FORMING THE LR(0) MACHINE

The procedures needed to build the CFSM are as follows:

Procedure Closure Figure 8.1

Procedure GoToSet	Figure 8.2
Function CheckGoTo	Figure 8.3
Procedure FormGoTo	Figure 8.4
Procedure BuildCFSM	Figure 8.5

Procedure BuildCFSM has three basic functions:

1) To build the CFSM. For this it calls GoToSet, FormGoTo, Closure, and CheckGoTo

2) To create the NonTerminal transition array

3) to determine which states are inconsistent and to create the Production Lookaheads (LA) in inconsistent states.

Procedure FormGoTo forms the GoTo Function and PAT for shift actions. It attaches the item in the sparse matrix.

Function CheckGoTo returns true if the state just formed by BuildCFSM is a different kernel state. It returns false if this state already exists in an earlier state.

Procedure GoToSet adds a new kernel item to the next set if the kernel item being checked has the vocab symbol under consideration after the dot. This check is made for every possible vocab symbol and for each item in the current state.

Procedure Closure forms the closure of state I.

8.3 CALCULATING THE LOOKAHEADS

The procedures needed to calculate the lookaheads using DeRemer's and Pennello's method are as follows:

Procedure DirectRead	Figure 8.6
Procedure TraverseBack	Figure 8.7
Procedure Lookback	Figure 8.8
Procedure CkIncludes	Figure 8.9
Procedure Includes	Figure 8.10
Procedure Traverse	Figure 8.11
Procedure Digraph	Figure 8.12
Procedure Union	Figure 8.13

Procedure FindLookAheads Figure 8.14

Procedure DirectRead adds the direct reads to initialize each ReadFollow Set for each nonterminal transition.

Procedure TraverseBack traverses back in the GoTo Function to make the includes list for each nonterminal transition (NTTrans) and to make the lookback list for each production in the list of productions requiring lookaheads calculated (ProdLAS).The procedure is called recursively as there may be many paths back.The traversal is through the vocab symbols (DownRec) pointers.

Procedure LookBack calculates the lookback relation for each production reduction in an inconsistent state. LookBack calls TraverseBack.

Procedure CkIncludes checks if the set record under consideration can go to lambda after the DotPlace + 1 position - i.e. if it should be added to the includes list for NTTrans I. CkIncludes calls TraverseBack.

Procedure Includes calculates the includes relation for all nonterminal transitions. The procedure accesses all nonterminal transitions in NTTrans array. It then checks all items in the state of each nonterminal transition to see if after reading the nonterminal vocab symbol under consideration the remaining symbols can go to lambda (calls CkIncludes).

Procedure Traverse and Procedure Digraph are the implementation of DeReemer's and Pennello's algorithm detailed in Chapter 3.

Procedure Union takes the union (for each production in an inconsistent state) of the follow in that production's lookback relation. Union calls Procedure Attach which attaches the reduction record in the PAT. Union also checks for inconsistencies in the state and reports if an inconsistent LALR(1) state

has been detected. All records are attached, even if an inconsistency has been observed. In this way all possible inconsistencies can be reported.

Procedure FindLookAheads calls all the procedures necessary to calculate the lookaheads for each production in an inconsistent state. If an inconsistency is found (i.e. the grammar is not LALR(1)) the program terminates with an appropriate message.

8.4 FINAL PARSE TABLE MODIFICATIONS

Two procedures are of interest here. They are

Procedure CondensePAT	Figure 8.15
Procedure Elim_Unit_Prods	Figure 8.16

Procedure CondensePat removes all single reduce states from the sparse matrix, renumbering all shift actions as needed.

Procedure Elim_Unit_Prods eliminates unit productions only when there is a chain reduction which stays in the same state. This implys they are independent of lookahead as explained in Chapter 6. Since they are unit productions they must be L-reductions.

8.5 THE SHIFT-REDUCE PARSER

There are three procedures of interest in the shift-reduce parser. They are:

Procedure Shift	Figure 8.17
Procedure Reduction	Figure 8.18
Procedure Parse	Figure 8.19

Procedure Shifts shifts the next state on the parse stack. It is important to realize that only the state is needed on the stack. The terminal or nonterminal symbol is unnecessary.

Procedure Reduction performs either an L-reduction or a straight reduction. This is a recursive procedure as there can be a chain of reductions.

Procedure Parse parses the input string. It calls shift and reduction until an

error or an accept action is reached.

The following pages contain the program segments which represent the above sections of the program. The two programs (the generator for the LALR(1) parsing action table and the shift-reduce parser) are in the possession of Professor Samuel Gulden.

```
Procedure Closure(statenum: state; LastKernelNum: KernelNums);
 (* form closure of state at statenum *)
 var
  I: vocab;
  Next,Last:ptr;
  FirstKernel, NextKernel: KernelNums;
  Added: array[Prod] of boolean;   (* array of boolean indexed by NT symbols *)
                                     (* at most each prod is a different NT symbol *)
                                (* indicate if NT symbol has been added to the closure *)
                                     (* if so, don't repeat *)

  Procedure AddProd(NTSymbol: vocab);
  (* add to closure the NT indicated by NTSymbol *)
  var
   I: Prod;
  begin
   for I:= 1 to NumProd do
    if Productions[I].LHS = NTSymbol then
       begin
       if Last = nil then
          begin  (* first addition *)
          new(KernelTab[Statenum].ClosurePtr);
          Last := KernelTab[Statenum].ClosurePtr;
          end
       else
          begin
          new(Last^.ClosurePtr);
          Last := Last^.ClosurePtr;
          end;
       Last^.ThisSet.ProdNum := I;
       Last^.ThisSet.DotPlace := 0;
       Last^.ClosurePtr := nil;
       end;
end; (* AddProd *)

begin
 for I:= 1 to LastNonTerm do
    Added[I] := false;
FirstKernel := KernelTab[StateNum].KernelRef;
Last := Nil;
for NextKernel := FirstKernel to LastKernelNum do
    (* check kernel items *)
    with KernelSets[NextKernel] do
     if (DotPlace < Productions[ProdNum].NumElem) then
        if (Productions[ProdNum].RHS[DotPlace+1] <= LastNonTerm) then
```

FIGURE 8.1 Procedure Closure

```
            (* add NT to closure if not already in *)
            if Added[Productions[ProdNum].RHS[DotPlace+1]] = false
               then
               begin
               AddProd(Productions[ProdNum].RHS[DotPlace+1]);
               Added[Productions[ProdNum].RHS[DotPlace+1]] := true;
               end;
(* check closure items *)
Next := KernelTab[StateNum].ClosurePtr;
while next <> nil do
  begin
  with next^.ThisSet do
    if Productions[ProdNum].NumElem <> 0 then
       if Productions[ProdNum].RHS[1] <= LastNonTerm then
          (* add NT to closure if not already in *)
          if Added[Productions[ProdNum].RHS[1]] = false
             then
             begin
             AddProd(Productions[ProdNum].RHS[1]);
             Added[Productions[ProdNum].RHS[1]] := true;
             end; (* if*)
     Next := Next^.ClosurePtr;
     end;  (* while *)
  end;  (* closure *)
```

FIGURE 8.1 CLOSURE ,CONTINUED

```
Procedure GoToSet(var OneSet: SetRec; var Temp: KernelNums;
                  I: vocab; LastStateNum: state);
(* Add a new kernel item to the next set if the kernel item being checked
   (OneSet) has the vocab symbol under consideration (I) after the dot.
   This check is made for every possible vocab symbol and for each item
   in the current state. The new set (state) is being created at
   LastStateNum + 1                                              *)

begin
  with OneSet do
    if DotPlace < Productions[ProdNum].NumElem then
      if ( Productions[ProdNum].RHS[DotPlace + 1] = I ) then
        begin   (* go to *)
        If Temp = MaxKernelNum then
           Fatal(13);
        Temp := Temp + 1;
        if LastStateNum = MaxStateNum then
           Fatal(14);
        (* move the dot *)
        KernelSets[Temp].ProdNum := ProdNum;
        KernelSets[Temp].DotPlace := DotPlace +1;
        end;  (* if *)
end;  (* GoToSet *)
```

FIGURE 8.2 Procedure GoToSet

```
Function CheckGoTo(J:state; OldKerNum, Temp:KernelNums): boolean;
(* return true if this state - J - is a different kernel state *)
(* return false if this state - J - already exists in an earlier state *)
(* This procedure checks the state formed by GoToSet                  *)
var
  I1, I2, K1: KernelNums;
  found: boolean;
begin
  found := false;
  I1 := KernelTab[J].KernelRef;
  if J = LastStateNum then
     I2 := LastKernelNum
  else
     I2 := KernelTab[J+1].KernelRef-1;
  if (I2 - I1) <> (Temp - OldKerNum) then
     CheckGoTo := true   (* must be different *)
  else
     begin
     repeat
        k1 := OldKerNum;
        repeat
          if (KernelSets[I1].ProdNum = KernelSets[k1]. ProdNum ) and
             (KernelSets[I1].DotPlace = KernelSets[K1].DotPlace ) then
                found := true;
          K1 := K1 + 1;
        until found or (K1 > Temp);
        I1 := I1 + 1;
     until (not found) or (I1 > I2);
     CheckGoTo := not found;
     end;  (* else *)
end; (* CheckGoTo *)
```

FIGURE 8.3 Function CheckGoTo

```
Procedure FormGoTo( CurrentState, St:state; I: vocab);
(* form the GoTo Function and PAT for shifts *)
(* attach to GoTo Tab - both acroos and down pointers *)

var
  Across: ParsePtr;
  Down:ParsePtr;

begin
  if GoToTab[CurrentState] = nil then
     begin    (* 1st element *)
     new(GoToTab[CurrentState]);
     Across := GoToTab[CurrentState];
     end
  else
     begin
     across := GoToTab[CurrentState];
     while (Across^.NextRec <> nil) do
          across := across^.NextRec;
     new(across^.NextRec);
     across := across^.NextRec;
     end;
with across^ do
     begin
     StNum := CurrentState;
     VocabNum := I;
     Action.op := S;
     Action.num2 := St;
     NextRec := nil;
     DownRec := nil;
     if I <= LastNonTerm then
        Ref := LastNTTrans;  (* add ref to PAT for non term trans *)
     end;
(* add down pointers *)
if DownTab[I] = nil then
     DownTab[I] := across
else
     begin
     Down:= DownTab[I];
     while (Down^.DownRec <> nil ) do
          down := down^.DownRec;
     Down^.DownRec := across;
     end;
end;  (* FormGoTo *)
```

FIGURE 8.4 Procedure FormGoTo

```
Procedure BuildCFSM(var LastStateNum: state; var LastKernelNum: KernelNums);
(* 1) build LR[0] machine
   2) create NonTerminal transition array
   3) Create array of Production Look Aheads in inconsistant states *)

var
   CurrentState, St :state;
            I: vocab;
    TempLastKernel, J, K: KernelNums;
    Across: ptr;
    New: boolean;
    Shift: boolean;  (* is there a shift in a state *)
    Reduce: integer; (* number of reductions in a state *)
    ReduceNum : set of Prod;  (* production # for reductions *)
    P: Prod;

begin
  KernelTab[0].ClosurePtr := nil;   (* create start state *)
  KernelTab[0].KernelRef := 0;
  KernelSets[0].ProdNum :=1;
  KernelSets[0].DotPlace :=0;
  LastKernelNum :=0;
  LastStateNum := 0;
  CurrentState := 0;
  Closure(0,0);
  For I := 1 to LastTerm do
     DownTab[I] := nil;
  LastNtTrans := 1;
  LastProdLA := 1;
  repeat                  (* create all GoTo states *)
   GoToTab[CurrentState] := nil;
   for I:= 1 to LastTerm do
     begin  (* check each symbol after dot to form goto *)
     if CurrentState < LastStateNum then
        K:= KernelTab[CurrentState+1].KernelRef -1
     else
        K:= LastKernelNum;
     TempLastKernel := LastKernelNum;
     (* forming GoTo(CurrentState,I) = LastStateNum   *)
     (* kernel items *)
     for J:= KernelTab[CurrentState].KernelRef to K do
        GoToSet(KernelSets[J],TempLastKernel,I,LastStateNum);
     (* closure items *)
     across := KernelTab[CurrentState].ClosurePtr;
     while across <> nil do
       begin
       GoToSet(across^.ThisSet,TempLastKernel,I, LastStateNum);
       across := across^.ClosurePtr;
       end;
```

FIGURE 8.5 Procedure BuildCFSM

```
(* were any new kernel items added *)
(* if not, no state was formed *)
if (TempLastKernel > LastKernelNum) then
  begin   (* check if have new kernel *)
  St := 0;
  repeat
    new := CheckGoTo(St, LastKernelNum + 1, TempLastKernel);
    St := St + 1;
  until ( not new) or (St > LastStateNum);
  if new then
     begin     (* this is a new state *)
     LastStateNum := LastStateNum +1;
     FormGoTo(CurrentState, LastStateNum, I);
     with KernelTab[LastStateNum] do
        begin
        ClosurePtr := nil;
        KernelRef := LastKernelNum +1;
        end;
      LastKernelNum := TempLastKernel;
      closure(LastStateNum, LastKernelNum);
      end  (* if new *)
    else    (* are going to an earlier, pre-existing state *)
      FormGoTo(CurrentState,St-1,I);
    (* form NonTerminal transitions *)
    if ( I <= LastNonTerm ) then
        begin    (* non terminal transition *)
        NTTrans[LastNTTrans].StNum := CurrentState;
        NTTrans[LastNTTrans].NonTerm := I;
        if LastNtTrans = MaxNTTrans then
          Fatal(15);
        LastNTTrans := LastNTTrans +1;
        end;
    end; (* if *)
 end; (* for I *)
(* check Current State for LR[0] consistency *)
Shift:= false;
Reduce:= 0;
ReduceNum := [];
for J:= KernelTab[CurrentState].KernelRef to K do
     if KernelSets[J].DotPlace <
           Productions[KernelSets[J].ProdNum].NumElem then
            Shift:= true
     else
            begin
            Reduce:= Reduce +1;
            ReduceNum := ReduceNum + [KernelSets[J].ProdNum];
            end;
across:= KernelTab[CurrentState].ClosurePtr;
```

FIGURE 8.5 BuildCFSM, CONTINUED

```
while across <> nil do
     begin
     if across^.ThisSet.DotPlace <
          Productions[across^.ThisSet.ProdNum].NumElem then
            Shift:= true
    else
            begin
            Reduce:= Reduce +1;
            ReduceNum := ReduceNum + [across^.ThisSet.ProdNum];
            end;
    across := across^.ClosurePtr;
    end;    (* while *)
if ((Shift = true) and (Reduce <> 0)) or (Reduce >1) then
    begin     (* have an inconsistent state *)
    Writeln(List,CurrentState:1, ' is an inconsistent state');
    (* add to Prod LookAheads *)
    for P := 1 to NumProd do
       if P IN ReduceNum then
          begin
          ProdLAS[LastProdLA].ProdStNum := CurrentState;
          ProdLAS[LastProdLA].ProdRed := P;
          if LastProdLA = MaxProdLA then
             Fatal(16);
          LastProdLA := LastProdLA + 1;
          end; (* for and if *)
        end; (* if *)
    if ( CurrentState = MaxStateNum) then
           fatal(14);
    CurrentState := CurrentState + 1;
  until CurrentState > LastStateNum;
end;  (* BuildCFSM *)
```

FIGURE 8.5 BuildCFSM, CONTINUED

```
procedure DirectRead;
(* add the direct reads to  initialize each ReadFollow for
   each non terminal transition *)
var
  I: NumNTTrans;
  across: ParsePtr;
  NextSt:state;
  J: vocab;

begin
  for I:= 1 to (LastNTTrans -1) do
      with NTTrans[I] do
         begin
         ReadFollow := [];
         across := GoToTab[StNum];
         while (across^.VocabNum <> NonTerm) do
            across := across^.NextRec;
         NextSt := across^.action.num2;
         across := GoToTab[NextSt];
         while across <> nil do
            begin
            if ( across^.VocabNum > LastNonTerm ) then
                ReadFollow := ReadFollow + [across^.VocabNum];
            across := across^.NextRec;
            end;   (* while *)
         end; (* with *)
end;    (* directly reads *)
```

FIGURE 8.6 Procedure DirectRead

```
procedure TraverseBack(var First, Tree:TraPtr; st:state; num: Prod;
                       length:integer);
(* traverse back in GoTo function to make the include list for each
   non terminal trans and the look-back list for each production in
   the lookahead list. This procedure is called recursively as there
   may be several paths back                                        *)
 var
    across, Ptr: ParsePtr;
    J: vocab;

begin
  if length = 0 then
     (* have gone back all the way *)
     begin              (* are at last state *)
     if tree = nil  then
         begin          (* add to TraPtr List for *)
         new(tree);      (* Lookback or includes *)
         First := tree;
         end
    else
         begin
         new(tree^.next);
         tree := tree^.next;
         end;
    across := GoToTab[st];
    while (across^.VocabNum <> Productions[num].LHS ) do
          across := across^.NextRec;
    tree^.Ref := across^.Ref;    (* Ref - NTTrans arrary element *)
    tree^.next := nil;
    end
 else    (* keep going back *)
    begin
    J:= Productions[num].RHS[length];
    length := length - 1;
    Ptr := DownTab[J];          (* going down vocab symbols *)
    while (Ptr <> nil ) do
      begin
      if ( Ptr^.action.num2 = st) then
         TraverseBack(First,tree ,Ptr^.StNum,num, length);
      ptr := ptr^.DownRec;       (* next state for this vocab symbol *)
      end; (* while *)
    end; (* else *)
end; (* TraverseBack *)
```

FIGURE 8.7 Procedure TraverseBack

```
procedure LookBack;
(* calculate the look-back relation for each production reduction
   in an inconsistent state *)
var
  I: NumProdLA;
  Length: integer;
  First:TraPtr;
begin
  for I:= 1 to (LastProdLA - 1) do
      begin
      length := productions[ProdLAS[I].ProdRed].NumElem;
      ProdLAS[I].Next := nil;
      TraverseBack(First, ProdLAS[I].next , ProdLAS[I].ProdStNum,
                ProdLAS[I].ProdRed, length);
      ProdLAS[I].Next := First;
      end;   (* for *)
end; (* LookBack *)
```

FIGURE 8.8 Procedure LookBack

```
procedure CkIncludes(OneSet:SetRec; I:NumNTTrans; var First:TraPtr);
(* Check if set record OneSet can go to lambda after DotPlace + 1 -
   i.e., if it should be added to the includes relation for NTTrans I *)
var
  J: integer;
  lambda: boolean;

begin
  with OneSet do
    if (DotPlace + 1) = Productions[ProdNum].NumElem then
       (* followed by lambda *)
       TraverseBack(First, NTTrans[I].next, NTTrans[I].StNum,
                 ProdNum, DotPlace)
    else  (* check all productions after DotPlace + 1 *)
       begin
       J:= DotPlace +2; (* first one to check *)
       lambda := true;
       while (( J <= Productions[ProdNum].NumElem ) and lambda ) do
          if ( Productions[ProdNum].RHS[J] In NullNonTerms ) then
             J := J + 1
          else
             lambda := false;
       if lambda then
          (* can go to lambda *)
          TraverseBack(First,NTTrans[I].next, NTTrans[I].StNum,
                    ProdNum, DotPlace);
       end;  (* else *)
end; (* CkIncludes *)
```

FIGURE 8.9 Procedure CkIncludes

```
procedure Includes;
(* calculate includes relation for all non terminal transitions *)
var
  I: NumNTTrans;
  J,K: KernelNums;
  across: ptr;
  St:state;
  First:TraPtr;

begin
  for I := 1 to ( LastNTTrans - 1 ) do
    begin        (* access all elements in nonterminal trans array *)
    First := nil;
    NTTrans[I].next := nil;
    st := NTTrans[I].StNum;
    if st < LastStateNum then
       K := KernelTab[st + 1 ].KernelRef - 1
    else
       K := LastKernelNum;
(* kernel items *)
(* check all items in state of NTTrans[I] to see if after
   reading the nonterminal transition under consideration
   the rest can go to lambda *)
for J:= KernelTab[st].KernelRef to K do
 if Productions[KernelSets[J].ProdNum].NumElem <>
          KernelSets[J].DotPlace then
  if (Productions[KernelSets[J].ProdNum].RHS[KernelSets[J].DotPlace +1]
       = NTTrans[I].NonTerm ) then
           CkIncludes(KernelSets[J], I,First);
(* closure *)
across := KernelTab[st].ClosurePtr;
while across <> nil do
 begin
 if Productions[across^.ThisSet.ProdNum].NumElem <>
          across^.ThisSet.DotPlace then
  if (Productions[across^.ThisSet.ProdNum].RHS[across^.ThisSet.DotPlace+1]
       = NTTrans[I].NonTerm ) then
           CkIncludes(across^.ThisSet,I,First);
  across := across^.ClosurePtr;
  end;
  NTTrans[I].next := First;  (* connect beginning *)
  end; (* for *)
end; (* Includes *)
```

FIGURE 8.10 Procedure Includes

```
procedure Traverse(X: NumNTTrans; var DepthOfSt:integer; Relation: char);
  (* called by Digraph algorithm below *)
  (* adaptation of DeReemer's and Pennello's algorithm *)
var
  NewPtr: TraPtr;      (* used in forming Reads relation locally *)
  RelationPtr: TraPtr; (* accesses Includes or Reads relation *)
  Ptr: ParsePtr;
  Y, element : NumNTTrans;
  d: integer;

    Procedure FormReads;
    (* form reads relation *)
    var
      NextSt: state;
      done: boolean;
    begin
      Ptr := GoToTab[NTTrans[X].StNum];
    while (Ptr^.VocabNum <> NTTrans[X].NonTerm ) do
       Ptr := Ptr^.NextRec;
    NextSt := Ptr^.Action.Num2;
    RelationPtr := nil;
    Ptr := GoToTab[NextSt];
    done := false;
    while not done do
     if ptr = nil then
        done := true
     else if (Ptr^.VocabNum > LastNonTerm ) then
        done := true
     else if (Ptr^.VocabNum IN NullNonTerms ) then
       begin
       if RelationPtr = nil then
         begin
         new(RelationPtr);
         NewPtr := RelationPtr;
         end
       else
         begin
         new(NewPtr^.Next);
         NewPtr := NewPtr^.Next;
         end;
       NewPtr^.Ref := Ptr^.Ref;
       NewPtr^.Next := nil;
       Ptr := Ptr^.NextRec;
       end
     else
       Ptr := Ptr^.NextRec;
end; (* FormReads *)
```

FIGURE 8.11 Procedure Traverse

```
begin  (* Traverse *)
  push(X, DepthOfSt);
  d := DepthOfSt;
  NTTrans[X].StDepth := d;
  (* complete the closure process *)
  if Relation = 'R' then
     FormReads
  else
     RelationPtr := NTTrans[X].Next;   (* Includes *)
  while RelationPtr <> nil do
    begin
    Y := RelationPtr^.Ref;
    if ( NTTrans[Y].StDepth = 0 ) then
      Traverse(Y, DepthOfSt,Relation );
    if (NTTrans[X].StDepth > NTTrans[Y].StDepth ) then
       NTTrans[X].StDepth := NTTrans[Y].StDepth;
   NTTrans[X].ReadFollow := NTTrans[X].ReadFollow + NTTrans[Y].ReadFollow;
    RelationPtr := RelationPtr^.Next;
    end;   (* while *)
  if ( NTTrans[X].StDepth = d ) then
    begin
    NTTrans[Stack^.Num].StDepth := MaxInt;
    if ( Stack^.Num <> X ) then
       NTTrans[Stack^.Num].ReadFollow := NTTrans[X].ReadFollow;
    pop(element,DepthOfSt);
    while ( element <> X ) do
      begin
      NTTrans[Stack^.Num].StDepth := MaxInt;
      if ( Stack^.Num <> X ) then
         NTTrans[Stack^.Num].ReadFollow := NTTrans[X].ReadFollow;
      pop(element,DepthOfSt);
      end; (* while *)
    end; (* if *)
end; (* Traverse *)
```

FIGURE 8.11 Traverse, CONTINUED

```
Procedure Digraph(Relation:char);
(* DeReemer's Digraph algorithm for traversing the Digraph -
   the GoToFunction - to apply the Reads and Includes Relations *)

var
  I: NumNTTrans;
  DepthOfSt: integer;

begin
  for I:= 1 to (LastNTTrans - 1) do
      NTTrans[I].StDepth := 0;
  Stack := nil;
  DepthOfSt := 0;
  for I:= 1 to ( LastNTTrans - 1 ) do
      if NTTrans[I].StDepth = 0 then
            Traverse(I, DepthOfSt, Relation);
end;   (* Digraph *)
```

FIGURE 8.12 Procedure Digraph

```
procedure Union(J: NumProdLA);
(* Take the union - for each production in an inconsistent
   state  - of the follow in that productions's look-Back. *)
(* Also call attach to add the reductions to PAT.
   Check for consistency in each state and report any inconsistent states *)
var
  Follow: set of vocab;
  First: ParsePtr;
  Ptr: TraPtr;
  done: boolean;
  v: vocab;
  I: NumNTTrans;
  st: state;

begin
  Ptr := ProdLAS[J].Next;
  Follow := [];
  writeln(List,'The Follow for Production-LookAheads for Item ', J:1);
while ( Ptr <> nil ) do
  begin
  I := Ptr^.Ref;
  Follow := Follow + NTTrans[I].ReadFollow;
  Ptr := Ptr^.Next;
  end;
(* add to PAT - attach and check LALR[1] consistency *)
(* PRINT ALL FOLLOW PLUS INCONSISTENT STATES *)
st := ProdLAS[J]. ProdStNum;
First := GoToTab[st];
```

FIGURE 8.13 Procedure Union

```
for v:= ( LastNonTerm + 1 ) to LastTerm do
  begin
  if v IN Follow then
    begin
    write(List, v:1,' ,');
    if GoToTab[st] = nil then
      attach(First,v,st,ProdLas[J].ProdRed)
    else
     begin
     done := false;
     while not done do
      begin
      if ((First = GoToTab[st] ) and ( v <= First^.VocabNum )) then
        begin
        done := true;
        if (v = First^.VocabNum) then
          begin
          writeln(List,'Inconsistent LALR[1] st ',St:1,
                ' with production LA ', J:1 );
          LALR1 := false;
          end;
        attach(First, v,st, ProdLAS[J].ProdRed);
        end
      else if (First^.NextRec <> nil) then
       begin
       if ( v <= First^.NextRec^.VocabNum ) then
          begin
          done := true;
          if ( v = First^.NextRec^.VocabNum ) then
             begin
             writeln(List,'Inconsistent LALR[1] st ',St:1,
                      ' with production LA ', J:1 );
             LALR1 := false;
             end;
          attach(First, v, st, ProdLAS[J].ProdRed);
          end;
       end
      else (* are at end of state *)
        begin
        done := true;
        attach(First,v,st,ProdLAS[J].ProdRed);
        end;
       if (not done ) then
          First := First^.NextRec;
       end;      (* while *)
     end;   (* else *)
   end; (* if v IN Follow *)
 end; (* for *)
 writeln(List);
end; (* Union *)
```

FIGURE 8.13 Union, CONTINUED

```
procedure FindLookAheads;
(* calculate lookaheads for each production in an inconsistent state *)
var
  J: NumProdLA;

begin
  DirectRead;
  LookBack;
  Includes;
  if NullNonTerms <> [] then
     Digraph('R');
  Digraph('I');
  LALR1 := true;
  for J:= 1 to ( LastProdLA - 1 ) do
      Union(J);
  if ( not LALR1 ) then
     Fatal(18);
end;
```

FIGURE 8.14 Procedure FindLookAheads

```
procedure CondensePAT;
(* remove all single reduce states from PAT *)

var
  I, J: integer;
  NumSinRedElim: state; (* the number of single reduce states you have
                           eliminated *)
  Reduction : Prod;
  found: boolean;
  Ptr: ParsePtr;

begin
  NumSinRedElim := 0;
  I := 0;
  repeat
    found := false;
    while ( not found ) do
      if I <= LastStateNum then
        if GoToTab[I] = nil then
          found := true
        else
          I := I + 1
      else
        found := true;
  if I <= LastStateNum then
     begin    (* have found a single reduce state *)
     J := KernelTab[I + NumSinRedElim].KernelRef;
     Reduction := KernelSets[J].ProdNum;
```

FIGURE 8.15 Procedure CondensePAT

```
for J := 0 to LastStateNum do
  begin   (* go thru entire PAT  *)
  Ptr:= GoToTab[J];
  while Ptr <> nil do
    begin
    if Ptr^.Action.op = S then
      if Ptr^.Action.num2 = I then
        begin  (* is a shift to the state to be eliminated *)
        Ptr^.Action.op := L;
        Ptr^.Action.num1 := Reduction;
        end
      else
        if Ptr^.Action.num2 > I then
          Ptr^.Action.num2 := Ptr^.Action.num2 - 1;
    Ptr := Ptr^.NextRec;
    end;  (* while *)
  end;  (* for *)
  for J := I to ( LastStateNum - 1 ) do
      GoToTab[J] := GoToTab[J + 1];
  NumSinRedElim := NumSinRedElim + 1;
  LastStateNum := LastStateNum - 1;
  end;  (* if *)
 until(I > LastStateNum);
end;   (* CondensePAT  *)
```

FIGURE 8.15 CondensePAT, CONTINUED

```
Procedure Elim_Unit_Prods;
(* eliminate unit productions only per state for L-reductions
   which are unit and thus will remain in the same state     *)

var
  I: integer;
  UnitProd: set of prod;
  NewNumProd: prod;       (* the elimination creats new productions *)
  s: state;
  Ptr1, Ptr2: ParsePtr;
  changes: integer;      (* indicates if any new changes were made
                            to the parse table and if there is a
                            possibility to add a pseudo-production *)
```

FIGURE 8.16 Procedure Elim_Unit_Prods

```
Procedure Check_Prods;
(* check if a new unit production must be added to our set of productions *)
var
  found: boolean;
  I: integer;

begin
  found := false;
  I := NumProd + 1;
  while ( I <= NewNumProd ) and ( not found ) do
    (* search for production *)
    if ( Productions[I].LHS = Productions[Ptr1^.Action.num1].LHS )
                    and
       (Productions[I].RHS[1] = Ptr2^.VocabNum ) then
          found := true
    else
       I := I + 1;
  if found then
     Ptr2^.Action.num1 := I
  else
     begin     (* make new prod *)
     if (NewNumProd = MaxNumProd ) then
        fatal(8);
     NewNumProd := NewNumProd + 1;
     Productions[NewNumProd].LHS := Productions[Ptr1^.Action.num1].LHS;
     Productions[NewNumProd].NumElem := 1;
     Productions[NewNumProd].RHS[1] := Ptr2^.VocabNum;
     UnitProd := UnitProd + [NewNumProd];
     Ptr2^.Action.num1 := NewNumProd;
     end;    (* else *)
end;  (* Check_Prods *)

begin
  UnitProd := [];
  changes :=0;
  for I := 1 to NumProd do
    if (Productions[I].NumElem = 1 ) then
       begin
       UnitProd := UnitProd + [I];
       if (Productions[I].RHS[1] <= LastNonTerm) then
          changes :=1; (* RHS is a nonterm *)
       end;
  NewNumProd := NumProd;
  if changes > 0 then
    begin
    for s := 0 to LastStateNum do
     repeat
      changes := 0;
      Ptr1 := GoToTab[s];
```

FIGURE 8.16 Elim_Unit_Prods, CONTINUED

```
while ptr1 <> nil do
 begin
 if (Ptr1^.Action.op = L ) then
   if (Ptr1^.Action.num1 IN UnitProd ) then
     if Productions[Ptr1^.Action.num1].RHS[1] <= LastNonTerm then
       begin  (* RHS is a nonterm - look for it *)
       ptr2 := GoToTab[s];
       while ( Ptr2 <> nil ) do
         begin
         If (( Ptr2 <> Ptr1 ) and (Ptr2^.Action.op = L )) then
           if ((Ptr1^.VocabNum = Productions[Ptr2^.Action.num1].LHS)
                        and
              (Ptr2^.Action.num1 IN UnitProd ) ) then
                 begin
                 changes := 1;
                 Check_Prods;
                 end;
               ptr2 := Ptr2^.NextRec;
               end;   (* while Ptr2 *)
             end;   (* if *)
       Ptr1 := Ptr1^.NextRec;
       end;  (* while Ptr1 *)
     until (changes = 0 );  (* are done with a state if no changes made *)
    If NewNumProd > NumProd then
       begin
       writeln(List,'The Pseudo-Productions added are');
       for changes := (NumProd + 1) to NewNumProd do
         Write_Prod(changes, MaxProdLen + 1);
       end;
    NumProd := NewNumProd;
    end;  (* if UnitProd <> nil *)
end;  (* Elim *)
```

FIGURE 8.16 Elim_Unit_Prods, CONTINUED

```
Procedure Shift(NextState:state);
(* shift a terminal symbol or a non-terminal symbol on the stack
   by putting the next state on the stack                    *)
var
  stk: stack;

begin
  new(stk);
  stk^.st := NextState;
  stk^.next := TOS;
  TOS := stk;
end;    (* shift *)
```

FIGURE 8.17 Procedure Shift

```
Procedure Reduction( reduce:prod; var CurrentAction: OpType);
(* perrform either an L-reduction or a straight reduction  *)
(* recursive call *)
(* SearchPAT is the call to the binary search routine for PAT *)
var
  I, NumTimes: integer;
  VocabSym: vocab;
  CurrentState: state;
  entry: integer;

begin
  NumTimes := Productions[reduce].NumElem;
  if (CurrentAction = L ) then
     NumTimes := NumTimes - 1;
  for I := 1 to NumTimes do
     TOS := TOS^.next;          (* pop stack *)
  VocabSym := Productions[reduce].LHS; (* non term *)
  CurrentState := TOS^.st;
  entry := SearchPAT(GoToTab[CurrentState].first,
                     GoToTab[CurrentState + 1].first - 1, VocabSym);
  if ( entry = 0 ) then
     fatal(3);       (* impossible situation *)
  CurrentAction := PAT[entry].Action.Op;
  case CurrentAction of
     S: shift(PAT[entry].Action.num2);
     L: Reduction(PAT[entry].Action.num1, CurrentAction);
     R: fatal(4);     (* impossible situation - cannot have reduce in the
                         non term part of the GoTo table *)
     AC: ;
     end;   (* case *)
end;   (* reduction *)
```

FIGURE 8.18 Procedure Reduction

```
Procedure Parse;
(* parse the input string *)
(* SearchPAT is the call to the binary search rtn for the PAT *)
var
  CurrentAction: OpType;
  CurrentState: state;
I, entry: integer;
  NextState: state;
  begin
   repeat
     CurrentState:= TOS^.st;
     entry := SearchPAT(GoToTab[CurrentState].first,
                        GoToTab[CurrentState + 1].first - 1, NextToken);
     if entry = 0 then
        (* default action *)
        if (GoToTab[CurrentState].default <= NumProd) then
        begin        (* actual reduction *)
        CurrentAction := R;
        Reduction(GoToTab[CurrentState].default, CurrentAction);
        end
     else
        CurrentAction := ER
  else   (* found an entry - no default *)
     begin
     CurrentAction := PAT[entry].Action.op;
     Case CurrentAction of
       S: begin
          NextState := PAT[entry].Action.num2;
          shift(NextState);
          NextSym;
          end;
       R: Reduction(PAT[entry].Action.num1,CurrentAction);
       L: begin
          Reduction(PAT[entry].Action.num1,CurrentAction);
          if ( CurrentAction <> AC ) then
              NextSym; (* go past original terminal in the L-reduction *)
          end;
      AC: ;
      ER: ;     (* never in PAT *)
        end;   (* case *)
      end;    (* else *)
  until ((CurrentAction = ER) or (CurrentAction = AC));
  if (CurrentAction = ER ) then
     List_Errors(CurrentState)
  else
     begin
     writeln(output, 'Parse O.K. ');
     writeln(List,' Parse O.K. ');
     end;
end; (* Parse  *)
```

FIGURE 8.19 Procedure Parse

BIBLIOGRAPHY

Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffrey D. 1987. *Compilers, Techniques, and Tools.* Reading, Mass. : Addison - Wesley.

Barrett, William A. , and Couch, John D. 1979. *Compiler Construction : Theory and Practice.* U.S.A. : Science Research Associates, Inc.

DeReemer, Frank, and Pennello, Thomas. 1982. "Efficient Computation of LALR(1) Look-Ahead Sets." *ACM Transactions on Programming Languages and Systems,* 4(4), pp. 615 - 648.

Fischer, Charles N., and LeBlanc, Jr., Richard J. 1988. *Crafting A Compiler.* Menlo Park, California : The Benjamin/Cummings Publishing Co.

Korenjak, A. J. 1969. "A Practical Method for Constructing LR(K) Processes," *Comm. ACM* 12(11), pp. 613 - 623.

Kristensen, Bent Bruun, and Madsen, Ole L. 1981. "Methods for Computing LALR(K) Lookahead." *ACM Transactions on Programming Languages and Systems,* 3(1), pp. 60 - 82.

Pyster, Arthur B. 1980. *Compiler Design and Construction.* Boston, Mass. : PWS Publishers.

Tremblay, Jean-Paul, and Sorenson, Paul G. 1985. *The Theory and Practice of Compiler Writing.* U.S.A. : McGraw-Hill, Inc.

Wood, Derrick. 1987. *Theory of Computation.* New York, N.Y. : Harper&Row.

APPENDIXES

APPENDIX A - INPUT AND OUTPUT FOR Gen_LALR1_Parse_Table Program

I. File References

1) The user is prompted to enter the file name containing the nonterminal, terminal symbols and productions in the grammar. The program displays on the screen:

Input >

2) After entering the file name containing the grammar symbols, the user is prompted to specify a file name for diagnostic output (collection of sets, etc.). The program displays on the screen:

List File For Sets >

If the output is to be to the screen, the user presses return.

3) If the grammar is found to be LALR(1) the program prompts the user to enter the file name in which to write the parse table and other necessary items which will be input to the shift-reduce parser. For this the program displays on the screen:

The input file for the Parse Program is being created

List File For This File >

If the grammar was not LALR(1) the following message is printed to the screen and no parse table is written:

The grammar is not LALR(1), no table is created.

II. Entering the grammar

The grammar is entered in the following order. Neither the augmented production symbol nor the augmented production are entered. The program creates these. All nonterminal and terminal symbols are converted to upper case letters.

1) Nonterminal Symbols

a) The nonterminal symbols are preceeded by the symbol '<' and terminated by the symbol '>'.

b) The start symbol of the grammar must be entered first.

c) The symbols must be separated by blanks.

d) The list may take as many lines as needed.

2) Terminal Symbols

a) All terminal symbols must be separated by blanks.

b) Entering the symbol id will permit the parse program to input standard user-defined id names (i.e. a, b. c1, etc.) as part of the grammar.

c) entering the symbol intliteral will permit the parser program to input integers as part of the grammar.

d) The list of terminal symbols may span multiple lines.

e) The list of terminal symbols is terminated by the end-of-string character (EOS) defined in the program. Currently this is the '#' symbol. This is defined in the CONST definition of the program and may be changed if needed.

3) Productions

a) Each production starts with the symbol '<<' and is terminated by the string '>>'.

b) The production is entered in the form

<< LHS --> RHS >>

c) LHS must, of course, be a nonterminal. If it isn't, the program terminates with an error message. If LHS is absent, the LHS of the preceeding production is assumed.

d) '-->' is the symbol which must be used for arrowsym.

e) RHS is a series of nonterminal and terminal symbols. If the RHS is absent, Lambda is assumed. Thus, to enter a λ nonterminal X enter

```
<<  X  -->  >>
```

f) RHS may span multiple lines.

g) The list of productions are terminated by the string 'endofprocs'.

Included below is the actual input for the expression grammar G1 and for PL0.

GRAMMAR G1

```
< E  T F A M >
( )  + - * / ID INTLITERAL  #
<<  E --> T  >>
<<  E --> E A T >>
<<  T --> F  >>
<<  T --> T M  F  >>
<<  F --> ( E  )  >>
<<  F --> ID  >>
<<  F --> INTLITERAL >>
<<  A --> +  >>
<<  A --> -  >>
<<  M --> *  >>
<<  m --> /  >>
ENDOFPROCS
```

PL0

```
< program cpart ctail vpart ppart block statement
cstat condition expression sign term factor addop
sexpression sxptail relop ttail mulop vtail      >
.  ,  ;  id  =  := exec  begin  end
 if  then  while  do  odd
<>  <  >  <=  >= + - *  / ( )
 intliteral  const  var  procedure  #
<< program -->  block  .  >>
<< cpart -->  const  id  =  intliteral  ctail  ;  >>
<< cpart -->  >>
<< ctail -->  ,  id  =  intliteral  ctail  >>
<< ctail -->  >>
<< vpart -->  var  id  vtail  ;  >>
<< vpart -->  >>
<< ppart -->  procedure  id  ;  block  ;  ppart  >>
<< ppart -->  >>
<< block -->  cpart  vpart  ppart  statement  >>
<< statement -->  id  := expression  >>
<< statement --> exec  id  >>
<< statement --> begin  statement  cstat  end  >>
```

```
<< statement -->  >>
<< statement --> if condition then statement >>
<< statement --> while condition do statement >>
<< cstat --> ; statement cstat >>
<< cstat -->  >>
<< condition --> odd expression >>
<< condition -->  expression relop expression >>
<< expression -->  sexpression >>
<< expression -->  sign sexpression >>
<< sign -->  + >>
<< sign -->  - >>
<< term -->  factor ttail >>
<< factor -->  id >>
<< factor -->  intliteral >>
<< factor -->  ( expression ) >>
<< addop -->  + >>
<< addop -->  - >>
<< sexpression -->  term sxptail >>
<< sxptail -->  addop term sxptail >>
<< sxptail -->  >>
<< relop -->  = >>
<< relop -->  <> >>
<< relop -->  < >>
<< relop -->  > >>
<< relop -->  >= >>
<< relop -->  <= >>
<< ttail -->  mulop factor ttail >>
<< ttail -->  >>
<< mulop --> * >>
<< mulop --> / >>
<< vtail --> , id vtail >>
<< vtail -->  >>
endofprocs
```

III. Diagnostic Output

The output produced in the file specified by the user after the prompt

List File For Sets >

is as follows:

1) The number representation of each terminal and nonterminal symbol. The start of its location in VocabTab. The actual character string of the terminal or nonterminal. This information is necessary since most of the diagnostic output references the number

representation, not the actual string.

2) The location of LastNonTerm in VocabInt. The location of LastTerm in VocabInt. The last location used in VocabTab (the location for the EOS symbol).

3) All productions in the grammar.

4) The number of null nonterminals, and the null nonterminals, if any.

5) The inconsistent states, if any.

6) A message indicating if the grammar is LR(0). If there are any inconsistent states listed in 5, the grammar is, of course, not LR(0).

7) The LR(0) collection of item sets.

8) The number of nonterminal transitions and the actual nonterminal transitions listed as (state, number representation for nonterminal transition).

9) The number of lookahead productions (LA) for which lookaheads needed to be calculated and the actual list of productions shown as (state, production number).

10) The Directly Reads calculated by the program for each nonterminal transition shown as a list of terminal symbols (their number representation).

11) The lookback calculated for each lookahead production shown as a list of nonterminal transitions (number 8 above).

12) The includes relation, if any, for each nonterminal transition shown as a list of nonterminal transitions (number 8 above).

13) The Follow for each production lookahead shown as a list of terminals (their number representation). If an inconsistent state is found at this point it is reported as

Inconsistent LALR(1) state # with production LA #

14) Any new Pseudo-Productions added by eliminating unit productions.

15) The final parsing action table (without the default entries).

The following is the actual output created for the expression grammar G1.

```
1   1    augsym
2   7    E
3   8    T
4   9    F
5   10    A
6   11    M
7   12    (
8   13    )
9   14    +
10   15    -
11   16    *
12   17    /
13   18    ID
14   20    INTLITERAL
15   30    #
 LastNonTerm = 6
 LastTerm = 15
 EndOfVocab = 30
augsym --> E #
E --> T
E --> E A T
T --> F
T --> T M F
F --> ( E )
F --> ID
F --> INTLITERAL
A --> +
A --> -
M --> *
M --> /
The Number of Null NonTerminals =     0
2 is an inconsistent state
15 is an inconsistent state
The grammar is not LR[0]
****************************************
State    0
augsym --> .E #
E --> .T
E --> .E A T
T --> .F
T --> .T M F
F --> .( E )
F --> .ID
F --> .INTLITERAL
****************************************
State    1
augsym --> E .#
E --> E .A T
A --> .+
A --> .-
****************************************
```

```
State   2
E --> T .
T --> T .M F
M --> .*
M --> ./
*****************************************
State   3
T --> F .
*****************************************
State   4
F --> ( .E )
E --> .T
E --> .E A T
T --> .F
T --> .T M F
F --> .( E )
F --> .ID
F --> .INTLITERAL
*****************************************
State   5
F --> ID .
*****************************************
State   6
F --> INTLITERAL .
*****************************************
State   7
E --> E A .T
T --> .F
T --> .T M F
F --> .( E )
F --> .ID
F --> .INTLITERAL
*****************************************
State   8
A --> + .
*****************************************
State   9
A --> - .
*****************************************
State   10
augsym --> E # .
*****************************************
State   11
T --> T M .F
F --> .( E )
F --> .ID
F --> .INTLITERAL
*****************************************
State   12
M --> * .
*****************************************
```

State 13
M --> / .
****
State 14
F --> (E .)
E --> E .A T
A --> .+
A --> .-
****
State 15
E --> E A T .
T --> T .M F
M --> .*
M --> ./
****
State 16
T --> T M F .
****
State 17
F --> (E) .
The number of NonTerminal Transitions are 13
1 (0,2)
2 (0,3)
3 (0,4)
4 (1,5)
5 (2,6)
6 (4,2)
7 (4,3)
8 (4,4)
9 (7,3)
10 (7,4)
11 (11,4)
12 (14,5)
13 (15,6)

The number of LookAhead Productions are 2
(2,2) (15,3)
The directly reads for non terminal transition (0,2) are :
9 ,10 ,15 ,
The directly reads for non terminal transition (0,3) are :
11 ,12 ,
The directly reads for non terminal transition (0,4) are :

The directly reads for non terminal transition (1,5) are :
7 ,13 ,14 ,
The directly reads for non terminal transition (2,6) are :
7 ,13 ,14 ,
The directly reads for non terminal transition (4,2) are :
8 ,9 ,10 ,
The directly reads for non terminal transition (4,3) are :
11 ,12 ,

The directly reads for non terminal transition (4,4) are :

The directly reads for non terminal transition (7,3) are :
11 ,12 ,
The directly reads for non terminal transition (7,4) are :

The directly reads for non terminal transition (11,4) are :

The directly reads for non terminal transition (14,5) are :
7 ,13 ,14 ,
The directly reads for non terminal transition (15,6) are :
7 ,13 ,14 ,
The Look-Back for (2,2) is :
1 , 6 ,
The Look-Back for (15,3) is :
1 , 6 ,
The Includes relation for (0,2) is:

The Includes relation for (0,3) is:
1 ,
The Includes relation for (0,4) is:
2 ,
The Includes relation for (1,5) is:

The Includes relation for (2,6) is:

The Includes relation for (4,2) is:

The Includes relation for (4,3) is:
6 ,
The Includes relation for (4,4) is:
7 ,
The Includes relation for (7,3) is:
1 , 6 ,
The Includes relation for (7,4) is:
9 ,
The Includes relation for (11,4) is:
2 , 7 , 9 ,
The Includes relation for (14,5) is:

The Includes relation for (15,6) is:

The Follow for Production-LookAheads for Item 1
8 ,9 ,10 ,15 ,
The Follow for Production-LookAheads for Item 2
8 ,9 ,10 ,15 ,
The Pseudo-Productions added are
T --> ID
T --> INTLITERAL

The PAT Is

State 0

2, S1 3, S2 4, L4 7, S3 13, L13 14, L14

State 1

5, S4 9, L9 10, L10 15, L1

State 2

6, S5 8, R2 9, R2 10, R2 11, L11 12, L12 15, R2

State 3

2, S6 3, S2 4, L4 7, S3 13, L13 14, L14

State 4

3, S7 4, L4 7, S3 13, L13 14, L14

State 5

4, L5 7, S3 13, L7 14, L8

State 6

5, S4 8, L6 9, L9 10, L10

State 7

6, S5 8, R3 9, R3 10, R3 11, L11 12, L12 15, R3

IV. The Parse Table To Be Input to the Shift-Reduce Parser

This is the output produced by the user after the prompt

The Input File for the Parse Program is being created

List File for the File >

1) The file name used to input the grammar. This is solely for ease in referencing which grammar this is since most of the input is in number representation.

2) LastNonTerm, LastTerm, EndOf Vocab, LastStateNum, NumProd

3) The VocabTab Table

4) The VocabInt Table

5) The productions in the grammar- LHS, NumElem, RHS for each production

6) The parsing action table with default entries.

Number 6 above is the only instructive item to demonstrate, since items 1 - 5 are really internal representations of the grammar. The Parsing Action Table for G1 is as follows. It is the table used in the shift-reduce parser, with accept added by the parse program itself.

```
6    STATE 0
 2 S1  3 S2  4 L4  7 S3  13 L13  14 L14
15    DEFAULT ERROR
4    STATE 1
 5 S4  9 L9  10 L10  15 L1
15    DEFAULT ERROR
3   STATE 2
 6 S5  11 L11  12 L12
2   DEFAULT REDUCTION
6    STATE 3
 2 S6  3 S2  4 L4  7 S3  13 L13  14 L14
15    DEFAULT ERROR
5    STATE 4
 3 S7  4 L4  7 S3  13 L13  14 L14
15    DEFAULT ERROR
4    STATE 5
 4 L5  7 S3  13 L7  14 L8
15    DEFAULT ERROR
4    STATE 6
 5 S4  8 L6  9 L9  10 L10
15    DEFAULT ERROR
3   STATE 7
 6 S5  11 L11  12 L12
3   DEFAULT REDUCTION
```

APPENDIX B INPUT AND OUTPUT FOR Shift_Reduce_Parser Program

I. File Input

The user is prompted for

1) The file created in IV in Appendix A (the parse table) by

Name of File Containing Parsing Action Table >

2) The file containing the string to be parsed by

Input File Containing String To Parse >

3) The file to be used for the result (output) of the parse

List File for Output of Parse >

Pressing return will put the output to the screen.

The following are the results of several runs of the parse program for the expression grammar G1 and PL0. Note that all strings are terminated with the # (EOS) symbol.

Strings testing G1

```
a * ( b - c / ( e + f ) ) - g #
 Parse O.K.
a + g * d - f / h * g #
 Parse O.K.
a + b + c + d #
 Parse O.K.
a * ( 2 * ( e / ( f * 10 ) ) ) + 6 #
 Parse O.K.
a + bc * 2 #
 Parse O.K.
a + - 10 #
     ^error in Parse
EXPECTED ( ID INTLITERAL
a + ( b * g ) ) #
              ^error in Parse
EXPECTED + - #
a + ( ( b * g ) - 10 #
                  ^error in Parse
EXPECTED ) + -
```

Testing Strings in PL0

```
;  #                    (* parse 1 *)
^error in Parse
EXPECTED  .

.  #                    (* parse 2 *)
 Parse O.K.

if a >= 10 then
   a := b + c ;  #      (* parse 3 *)
                ^error in Parse
EXPECTED  .

if a >= 10 then
   a := b + c .  #      (* parse 4 *)
 Parse O.K.

const
   x = 10 ,
   y = 15 ;
var
   a,b,c, ;             (* parse # 5 *)
         ^error in Parse
EXPECTED  ID

const
   x = 10,
   y = 15,
   z = 25 ;
var
  a, b , c ;
begin
   a := b + c;
   b := 10 - 15;
   ;
   ;
   end .  #             (* parse # 6 *)
 Parse O.K.

var
  a, b , c ;
begin
   a := b + c;
   b := 10 - 15;
   ;
   ;
   end .  #             (* parse # 7 *)
 Parse O.K.
```

```
var
  a, b , c ;
procedure xx( Y , z ) ;     (* parse # 8 *)
                ^error in Parse
EXPECTED  ;

const
   x = 10,
   y = 15,
   z = Y ;                  (* parse # 9 *)
         ^error in Parse
EXPECTED  INTLITERAL
```

```
const
   x = 10,
   y = 15,
   z = 25 ;
var
  a, b , c ;

procedure xx ;
  a := 10;

begin
   a := b + c;
   b := 10 - 15;
   exec xx ;
   ;
   end .  #              (* parse # 10 *)
 Parse O.K.
```

The scanner for the parse program must have all special symbols separated by blanks. This means that the parser can parse the string a:=b without blanks, but will take the string a := 3*(b+c*(d+e)) as incorrect unless there are blanks between the * and the (and between the last two parentheses. It will take the symbols '))' as one special character rather then as two characters, ')' and ')'. The same is true for '*('. Changing this is extremely simple, as it requires adding a simple check for all 'special' characters which are often written together in specific programming languages (without intervening blanks) but which are normally considered separately. This was not

included in the shift-reduce parser so it could apply to 'any' grammar, not a specific one.

BIOGRAPHY

Alma Marie Sechler Schneck was born in Allentown, Pa. Her parents are Alma Marie Fluck and Kenneth Fox Sechler.

Marie Sechler graduated third in her class from Allentown High School in June, 1959. She graduated second in her class from Muhlenberg College in June of 1963 with a B.S. degree in Mathematics. From 1963-1965 she was employed by PP&L in Allentown, Pa. as a mathematician/computer programmer in the Systems Planning Department. Her primary function was to solve engineering problems on the computer. A direct result of one of the projects at PP&L was the publication of the following IEEE paper with Mr. C. W. Watchorn: "A Computer Program for Determining the Economic Size of Pump-Strorage Hydro Sites," *IEEE Transactions on Power Apparatus and Systems,* Vol. PAS-85, No.11, November, 1966 by A. Marie Schneck and C.W. Watchorn, FELLOW, IEEE.

In 1963 Marie Sechler married Henry R. Schneck, Jr. who received his M.S. degree in Civil Engineering from Lehigh in 1965. They have three children, Lisa Marie, Karen Elizabeth, and David Henry.

From 1973-1987 Marie was self-employed, operating her own ballet studio, Marie's School of Ballet Arts, in Allentown, Pa. The school trained children through adults in the art of classical ballet.

In 1987 Marie received a second degree, a Graduate Certificate, from Muhlenberg College in Computer Science. In 1987 she received a fellowship to study for a Master's Degree in Computer Science from Lehigh University. She received the degree in June, 1989.

Marie will be employed by PP&L starting in June of 1989 as an Application Programmer in the Engineering and Scientific Systems Department. She also plans to continue to teach computer science as an adjunct faculty member.