UNIT -3

SYNTAX ANALYSIS

3.1 ROLE OF THE PARSER

Parser obtains a string of tokens from the lexical analyzer and verifies that it can be generated by the language for the source program. The parser should report any syntax errors in an intelligible fashion. The two types of parsers employed are:

1.Top down parser: which build parse trees from top(root) to bottom(leaves)

2.Bottom up parser: which build parse trees from leaves and work up the root.

Therefore there are two types of parsing methods– top-down parsing and bottom-up parsing
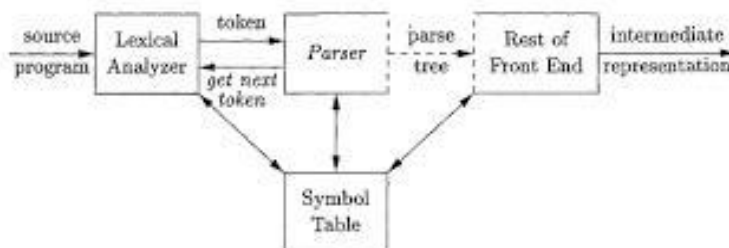

Figure 4.1: Position of parser in compiler model

3.2 TOP-DOWN PARSING

A program that performs syntax analysis is called a parser. A syntax analyzer takes tokens as input and output error message if the program syntax is wrong. The parser uses symbol-look-ahead and an approach called top-down parsing without backtracking. Top-downparsers check to see if a string can be generated by a grammar by creating a parse tree starting from the initial symbol and working down. Bottom-up parsers, however, check to see a string can be generated from a grammar by creating a parse tree from the leaves, and working up. Early parser generators such as YACC creates bottom-up parsers whereas many of Java parser generators such as JavaCC create top-down parsers.

3.3RECURSIVE DESCENT PARSING

Typically, top-down parsers are implemented as a set of recursive functions that descent through a parse tree for a string. This approach is known as recursive descent parsing, also known as LL(k) parsing where the first L stands for left-to-right, the second L stands for

leftmost-derivation, and k indicates k-symbol lookahead. Therefore, a parser using the single symbol look-ahead method and top-down parsing without backtracking is called LL(1) parser. In the following sections, we will also use an extended BNF notation in which some regulation expression operators are to be incorporated.

A syntax expression defines sentences of the form , or . A syntax of the form defines sentences that consist of a sentence of the form followed by a sentence of the form followed by a sentence of the form . A syntax of the form defines zero or one occurrence of the form . A syntax of the form defines zero or more occurrences of the form . A usual implementation of an LL(1) parser is:

> initialize its data structures,
>
> get the lookahead token by calling scanner routines,
>
> and call the routine that implements the start symbol.

Here is an example.

proc syntaxAnalysis()

begin

initialize(); // initialize global data and structures

nextToken(); // get the lookahead token

program(); // parser routine that implements the start

symbol end;

3.4 FIRST AND FOLLOW

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or e can be added to any FIRST set.

1. If X is terminal, then FIRST(X) is {X}.

2. If X->e is a production, then add e to FIRST(X).

3. If X is nonterminal and X->Y1Y2...Yk is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yi) and e is in all of FIRST(Y1),...,FIRST(Yi-1) that is, $Y1.......Yi_{-1}$=*>e. If e is in FIRST(Yj) for all j=1,2,...,k, then add e to FIRST(X). For example, everything in FIRST(Yj) is surely in FIRST(X). If y1 does not derive e, then we add nothing more to FIRST(X), but if Y1=*>e, then we add FIRST(Y2) and so on.

To compute the FIRST(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place $ in FOLLOW(S), where S is the start symbol and $ in the input right endmarker.

2. If there is a production A=>aBs where FIRST(s) except e is placed in FOLLOW(B).

3. If there is aproduction A->aB or a production A->aBs where FIRST(s) contains e, then everything in FOLLOW(A) is in FOLLOW(B).

Consider the following example to understand the concept of First and Follow.Find the first and follow of all nonterminals in the Grammar-

E -> TE' E'-> +TE'|e T -> FT' T'-> *FT'|e

F -> (E)|id Then:

FIRST(E)=FIRST(T)=FIRST(F)={(,id}

FIRST(E')={+,e} FIRST(T')={*,e}

FOLLOW(E)=FOLLOW(E')={),$}

FOLLOW(T)=FOLLOW(T')={+,),$}

FOLLOW(F)={+,*,),$}

For example, id and left parenthesis are added to FIRST(F) by rule 3 in definition of FIRST with i=1 in each case, since FIRST(id)=(id) and FIRST('(')= {(} by rule 1. Then by rule 3 with i=1, the production T -> FT' implies that id and left parenthesis belong to FIRST(T) also.

To compute FOLLOW,we put $ in FOLLOW(E) by rule 1 for FOLLOW. By rule 2 applied toproduction F-> (E), right parenthesis is also in FOLLOW(E). By rule 3 applied to production E-> TE', $ and right parenthesis are in FOLLOW(E').

3.5 CONSTRUCTION OF PREDICTIVE PARSING TABLES

For any grammar G, the following algorithm can be used to construct the predictive parsing table. The algorithm is

Input : Grammar G

Output : Parsing table M

Method

1. 1.For each production A-> a of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(a), add A->a, to M[A,a].
3. If e is in First(a), add A->a to M[A,b] for each terminal b in FOLLOW(A). If e is in FIRST(a) and $ is in FOLLOW(A), add A->a to M[A,$].
4. Make each undefined entry of M be error.

3.6.LL(1) GRAMMAR

The above algorithm can be applied to any grammar G to produce a parsing table M. For some Grammars, for example if G is left recursive or ambiguous, then M will have at least one multiply-defined entry. A grammar whose parsing table has no multiply defined entries is said to be LL(1). It can be shown that the above algorithm can be used to produce for every LL(1) grammar G a parsing table M that parses all and only the sentences of G. LL(1) grammars have several distinctive properties. No ambiguous or left recursive grammar can be LL(1). There remains a question of what should be done in case of multiply defined entries. One easy solution is to eliminate all left recursion and left factoring, hoping to produce a grammar which will produce no multiply defined entries in the parse tables. Unfortunately there are some grammars which will give an LL(1) grammar after any kind of alteration. In general, there are no universal rules to convert multiply defined entries into single valued entries without affecting the language recognized by the parser.

The main difficulty in using predictive parsing is in writing a grammar for the source language such that a predictive parser can be constructed from the grammar. Although left recursion elimination and left factoring are easy to do, they make the resulting grammar hard to read and difficult to use the translation purposes. To alleviate some of this difficulty, a common organization for a parser in a compiler is to use a predictive parser for control

constructs and to use operator precedence for expressions.however, if an lr parser generator is available, one can get all the benefits of predictive parsing and operator precedence automatically.

3.7.ERROR RECOVERY IN PREDICTIVE PARSING

The stack of a nonrecursive predictive parser makes explicit the terminals and nonterminals that the parser hopes to match with the remainder of the input. We shall therefore refer to symbols on the parser stack in the following discussion. An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and the parsing table entry M[A,a] is empty.

Panic-mode error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears. Its effectiveness depends on the choice of synchronizing set. The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice. Some heuristics are as follows

- As a starting point, we can place all symbols in FOLLOW(A) into the synchronizing set for nonterminal A. If we skip tokens until an element of FOLLOW(A) is seen and pop A from the stack, it is likely that parsing can continue.

- It is not enough to use FOLLOW(A) as the synchronizingset for A. Fo example , if semicolons terminate statements, as in C, then keywords that begin statements may not appear in the FOLLOW set of the nonterminal generating expressions. A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being skipped. Often, there is a hierarchica structure on constructs in a language; e.g., expressions appear within statement, which appear within bblocks,and so on. We can add to the synchronizing set of a lower construct the symbols that begin higher constructs. For example, we might add keywords that begin statements to the synchronizing sets for the nonterminals generaitn expressions.

- If we add symbols in FIRST(A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.

➕ If a nonterminal can generate the empty string, then the production deriving e can be used as a default. Doing so may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery.

➕ If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of all other tokens.

3.8 LR PARSING INTRODUCTION

The "L" is for left-to-right scanning of the input and the "R" is for constructing a rightmost derivation in reverse.
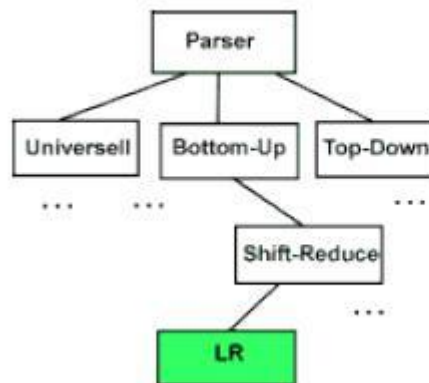

WHY LR PARSING:

✑ LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written.

✑ The LR parsing method is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.

✑ The class of grammars that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive parsers.

✑ An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to- right scan of the input.

The disadvantage is that it takes too much work to constuct an LR parser by hand for a typical programming-language grammar. But there are lots of LR parser generators available to make this task easy.

MODELS OF LR PARSERS

The schematic form of an LR parser is shown below.

LR Parser Table
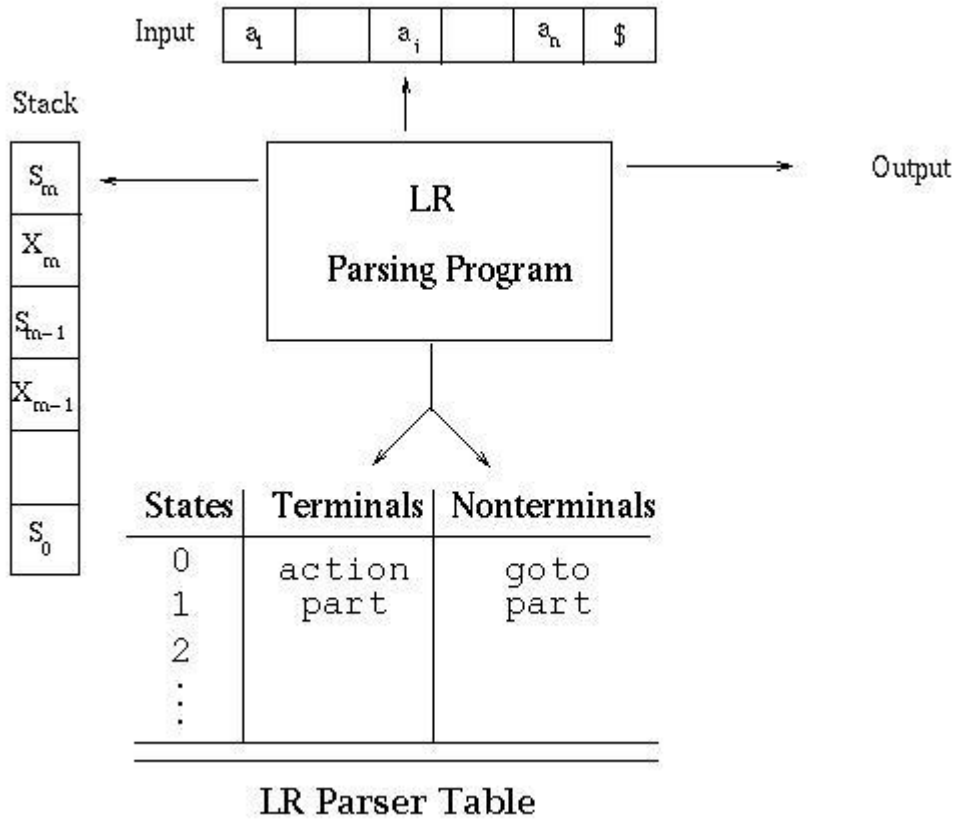
The program uses a stack to store a string of the form s0X1s1X2...Xmsm where sm is on top. Each Xi is a grammar symbol and each si is a symbol representing a state. Each state symbol summarizes the information contained in the stack below it. The combination of the state symbol on top of the stack and the current input symbol are used to index the parsing table and determine the shiftreduce parsing decision. The parsing table consists of two parts: a parsing action function action and a goto function goto. The program driving the LR parser behaves as follows: It determines sm the state currently on top of the stack and ai the current input symbol. It then consults action[sm,ai], which can have one of four values:

ƒ   shift s, where s is a state

ƒ   reduce by a grammar production A -> b

ƒ   accept

ƒ   error

The function goto takes a state and grammar symbol as arguments and produces a state.

For a parsing table constructed for a grammar G, the goto table is the transition function of a deterministic finite automaton that recognizes the viable prefixes of G. Recall that the viable prefixes of G are those prefixes of right-sentential forms that can appear on the stack of a shiftreduce parser because they do not extend past the rightmost handle.

A configuration of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpended input:

(s0 X1 s1 X2 s2... Xm sm, ai ai+1... an$)

This configuration represents the right-sentential

form X1 X1 ... Xm ai ai+1 ...an

in essentially the same way a shift-reduce parser would; only the presence of the states on the stack is new. Recall the sample parse we did (see Example 1: Sample bottom-up parse) in which we assembled the right-sentential form by concatenating the remainder of the input buffer to the top of the stack. The next move of the parser is determined by reading ai and sm, and consulting the parsing action table entry action[sm, ai]. Note that we are just looking at the state here and no symbol below it. We'll see how this actually works later. The configurations resulting after each of the four types of move are as follows:

If action[sm, ai] = shift s, the parser executes a shift move entering the configuration

(s0 X1 s1 X2 s2... Xm sm ai s, ai+1... an$)

Here the parser has shifted both the current input symbol ai and the next symbol.

If action[sm, ai] = reduce A -> b, then the parser executes a reduce move, entering

the configuration,

(s0 X1 s1 X2 s2... Xm-r sm-r A s, ai ai+1... an$)

where s = goto[sm-r, A] and r is the length of b, the right side of the production. The parser first popped 2r symbols off the stack (r state symbols and r grammar symbols), exposing state sm-r. The parser then pushed both A, the left side of the production, and s, the entry for goto[sm-r, A], onto the stack. The current input symbol is not changed in a reduce move.

The output of an LR parser is generated after a reduce move by executing the semantic action associated with the reducing production. For example, we might just print out the production

reduced.

If action[sm, ai] = accept, parsing is completed.

3.9 SHIFT REDUCE PARSING

A shift-reduce parser uses a parse stack which (conceptually) contains grammar symbols. During the operation of the parser, symbols from the input are shifted onto the stack. If a prefix of the symbols on top of the stack matches the RHS of a grammar rule which is the correct rule to use within the current context, then the parser reduces the RHS of the rule to its LHS,replacing the RHS symbols on top of the stack with the nonterminal occurring on the LHS of the rule. This shift-reduce process continues until the parser terminates, reporting either success or failure. It terminates with success when the input is legal and is accepted by the parser. It terminates with failure if an error is detected in the input. The parser is nothing but a stack automaton which may be in one of several discrete states. A state is usually represented simply as an integer. In reality, the parse stack contains states, rather than grammar symbols. However, since each state corresponds to a unique grammar symbol, the state stack can be mapped onto the grammar symbol stack mentioned earlier.

The operation of the parser is controlled by a couple of tables:

ACTION TABLE

The action table is a table with rows indexed by states and columns indexed by terminal symbols. When the parser is in some state s and the current lookahead terminal is t, the action taken by the parser depends on the contents of action[s][t], which can contain four different kinds of entries:

Shift s'

Shift state s' onto the parse stack.

Reduce r

Reduce by rule r. This is explained in more detail below.

Accept

Terminate the parse with success, accepting the input.

Error

Signal a parse error

GOTO TABLE

The goto table is a table with rows indexed by states and columns indexed by nonterminal

symbols. When the parser is in state s immediately after reducing by rule N, then the next state to enter is given by goto[s][N].

The current state of a shift-reduce parser is the state on top of the state stack. The detailed operation of such a parser is as follows:

1. Initialize the parse stack to contain a single state s0, where s0 is the distinguished initial state of the parser.

2. Use the state s on top of the parse stack and the current lookahead t to consult the action table entry action[s][t]:

· If the action table entry is shift s' then push state s' onto the stack and advance the input so that the lookahead is set to the next token.

· If the action table entry is reduce r and rule r has m symbols in its RHS, then pop m symbols off the parse stack. Let s' be the state now revealed on top of the parse stack and N be the LHS nonterminal for rule r. Then consult the goto table and push the state given by goto[s'][N] onto the stack. The lookahead token is not changed by this step.

¾   If the action table entry is accept, then terminate the parse with success.

¾   If the action table entry is error, then signal an error.

3. Repeat step (2) until the parser terminates.

For example, consider the following simple grammar

0) $S: stmt <EOF>

1) stmt: ID ':=' expr

2) expr: expr '+' ID

3) expr: expr '-' ID

4) expr: ID

which describes assignment statements like a:= b + c - d. (Rule 0 is a special augmenting production added to the grammar).

One possible set of shift-reduce parsing tables is shown below (sn denotes shift n, rn denotes reduce n, acc denotes accept and blank entries denote error entries): Parser Tables

Parser Tables

	Action Table					Goto Table	
	ID	**':='**	**'+'**	**'_'**	**\<EOF\>**	**stmt**	**expr**
0	s1					g2	
1		s3					
2					s4		
3	s5	I					g6
4	*acc*	*acc*	*acc*	*acc*	*acc*		
5	r4	r4	r4	r4	r4		
6	r1	r1	s7	s8	r1		
7	s9						
8	s10						
9	r2	r2	r2	r2	r2		
10	r3	r3	r3	r3	r3		

A trace of the parser on the input a:= b + c - d is shown below:

Stack	Remaining Input	Action
0/$S	a:= b + c - d	s1
0/$S 1/a	:= b + c - d	s3
0/$S 1/a 3/:=	b + c - d	s5
0/$S 1/a 3/:= 5/b	+ c - d	r4
0/$S 1/a 3/:=	+ c - d	g6 on expr
0/$S 1/a 3/:= 6/expr	+ c - d	s7
0/$S 1/a 3/:= 6/expr 7/+	c - d	s9
0/$S 1/a 3/:= 6/expr 7/+ 9/c	- d	r2
0/$S 1/a 3/:=	- d	g6 on expr
0/$S 1/a 3/:= 6/expr	- d	s8
0/$S 1/a 3/:= 6/expr 8/-	d	s10
0/$S 1/a 3/:= 6/expr 8/- 10/d	<EOF>	r3
0/$S 1/a 3/:=	<EOF>	g6 on expr
0/$S 1/a 3/:= 6/expr	<EOF>	r1
0/$S	<EOF>	g2 on stmt
0/$S 2/stmt	<EOF>	s4
0/$S 2/stmt 4/	<EOF>	accept

Each stack entry is shown as a state number followed by the symbol which caused the transition to that state.

3.10 SLR PARSER

An *LR(0) item* (or just *item*) of a grammar *G* is a production of *G* with a dot at some position of the right side indicating how much of a production we have seen up to a given point. For example, for the production E -> E + T we would have the following items:

[E -> .E + T]

[E -> E. + T]

[E -> E +. T]

[E -> E + T.]

Stack	State	Comments
Empty	[E'-> .E]	can't go anywhere from here
	e-transition	so we follow an e-transition
Empty	[F -> .(E)]	now we can shift the (
(	[F -> (.E)]	building the handle (E); This state says: "I have (on the stack and expect the input to give me tokens that can eventually be reduced to give me the rest of the handle, E)."

CONSTRUCTING THE SLR PARSING TABLE

To construct the parser table we must convert our NFA into a DFA. The states in the LR table will be the e-closures of the states corresponding to the items SO...the process of creating the LR state table parallels the process of constructing an equivalent DFA from a machine with e-transitions. Been there, done that - this is essentially the subset construction algorithm so we are in familiar territory here.

We need two operations: closure() and goto().

closure()

If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules: Initially every item in I is added to closure(I)

If A -> a.Bb is in closure(I), and B -> g is a production, then add the initial item [B -> .g] to I, if it is not already there. Apply this rule until no more new items can be added to closure(I).

From our grammar above, if I is the set of one item {[E'-> .E]}, then closure(I) contains:

I0: E' -> .E

E -> .E + T

E -> .T

T -> .T * F

T -> .F

F -> .(E)

F -> .id

goto()

goto(I, X), where I is a set of items and X is a grammar symbol, is defined to be the closure of the set of all items [A -> aX.b] such that [A -> a.Xb] is in I. The idea here is fairly intuitive: if I is the set of items that are valid for some viable prefix g, then goto(I, X) is the set of items that are valid for the viable prefix gX.

SETS-OF-ITEMS-CONSTRUCTION

To construct the canonical collection of sets of LR(0) items for

augmented grammar G'.

procedure items(G')

begin

C := {closure({[S' -> .S]})};

repeat

for each set of items in C and each grammar

symbol X such that goto(I, X) is not empty and

not in C do add goto(I, X) to C;

until no more sets of items can be added to C

end;

ALGORITHM FOR CONSTRUCTING AN SLR PARSING TABLE

Input: augmented grammar G'

Output: SLR parsing table functions action and goto for G'

Method:

Construct C = {I0, I1 , ..., In} the collection of sets of LR(0) items for G'.

State i is constructed from Ii:

if [A -> a.ab] is in Ii and goto(Ii, a) = Ij, then set action[i, a] to "shift j". Here a must be

a terminal.

if [A -> a.] is in Ii, then set action[i, a] to "reduce A -> a" for all a in FOLLOW(A). Here

A may

not be S'.

if [S' -> S.] is in Ii, then set action[i, $] to "accept"

If any conflicting actions are generated by these rules, the grammar is not SLR(1) and the

algorithm fails to produce a parser. The goto transitions for state i are constructed for all

*nonterminal*s *A* using the rule: If goto(Ii, A)= Ij, then goto[i, *A*] = j. All entries not defined

by rules 2 and 3 are made "error".

The inital state of the parser is the one constructed from the set of items containing [*S'* ->

.S]. Let's work an example to get a feel for what is going on, An Example

(1) E -> E * B

(2) E -> E + B

(3) E -> B

(4) B -> 0

(5) B -> 1

The Action and Goto Table The two LR(0) parsing tables for this grammar look as follows:

	action					goto	
state	*	+	0	1	$	E	B
0			s1	s2		3	4
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	s5	s6			acc		
4	r3	r3	r3	r3	r3		
5			s1	s2			7
6			s1	s2			8
7	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2		

3.11 LALR PARSER:

We begin with two observations. First, some of the states generated for LR(1) parsing have the same set of core (or first) components and differ only in their second component, the lookahead symbol. Our intuition is that we should be able to merge these states and reduce the number of states we have, getting close to the number of states that would be generated for LR(0) parsing. This observation suggests a hybrid approach: We can construct the canonical LR(1) sets of items and then look for sets of items having the same core. We merge these sets with common cores into one set of items. The merging of states with common cores can never produce a shift/reduce conflict that was not present in one of the original states because shift actions depend only on the core, not the lookahead. But it is possible for the merger to produce a reduce/reduce conflict.

Our second observation is that we are really only interested in the lookahead symbol in places where there is a problem. So our next thought is to take the LR(0) set of items and add lookaheads only where they are needed. This leads to a more efficient, but much more complicated method.

ALGORITHM FOR EASY CONSTRUCTION OF AN LALR TABLE

Input: G'

Output: LALR parsing table functions with action and goto for G'.

Method:

1. Construct C = {I0, I1 , ..., In} the collection of sets of LR(1) items for G'.

2. For each core present among the set of LR(1) items, find all sets having that core and replace these sets by the union.

3. Let C' = {J0, J1 , ..., Jm} be the resulting sets of LR(1) items. The parsing actions for state i are constructed from Ji in the same manner as in the construction of the canonical LR parsing table.

4. If there is a conflict, the grammar is not LALR(1) and the algorithm fails.

5. The goto table is constructed as follows: If J is the union of one or more sets of LR(1) items, that is, J = I0U I1 U ... U Ik, then the cores of goto(I0, X), goto(I1, X), ..., goto(Ik, X) are the same, since I0, I1 , ..., Ik all have the same core. Let K be the union of all sets of items having the same core asgoto(I1, X).

6. Then goto(J, X) = K.

Consider the above example,

I3 & I6 can be replaced by their union

I36:C->c.C,c/d/$

C->.Cc,C/D/$

C->.d,c/d/$

I47:C->d.,c/d/$

I89:C->Cc.,c/d/$

Parsing Table

state	c	d	$	S	C
0	S36	S47		1	2
1			Accept		
2	S36	S47			5

36	S36	S47			89
47	R3	R3			
5			R1		
89	R2	R2	R2		

HANDLING ERRORS

The LALR parser may continue to do reductions after the LR parser would have spotted an error, but the LALR parser will never do a shift after the point the LR parser would have discovered the error and will eventually find the error.

3.12 LR ERROR RECOVERY

An LR parser will detect an error when it consults the parsing action table and find a blank or error entry. Errors are never detected by consulting the goto table. An LR parser will detect an error as soon as there is no valid continuation for the portion of the input thus far scanned. A canonical LR parser will not make even a single reduction before announcing the error. SLR and LALR parsers may make several reductions before detecting an error, but they will never shift an erroneous input symbol onto the stack.

3.12.1 PANIC-MODE ERROR RECOVERY

We can implement panic-mode error recovery by scanning down the stack until a state s with a goto on a particular nonterminal A is found. Zero or more input symbols are then discarded until a symbol a is found that can legitimately follow The situation might exist where there is more than one choice for the nonterminal A. Normally these would be nonterminals representing major program pieces, e.g. an expression, a statement, or a block. For example, if A is the nonterminal stmt, a might be semicolon or }, which marks the end of a statement sequence. This method of error recovery attempts to eliminate the phrase containing the syntactic error. The parser determines that a string derivable from A contains an error. Part of that string has already been processed, and the result of this processing is a sequence of states on top of the stack. The remainder of the string is still in the input, and the parser attempts to skip over the remainder of this string by looking for a symbol on the input that can legitimately follow A. By removing states from the stack, skipping over the input, and pushing GOTO(s, A) on the stack, the parser pretends that if has found an instance of A and resumes normal parsing.

3.12.2 PHRASE-LEVEL RECOVERY

Phrase-level recovery is implemented by examining each error entry in the LR action table and deciding on the basis of language usage the most likely programmer error that would give rise to that error. An appropriate recovery procedure can then be constructed; presumably the top of the stack and/or first input symbol would be modified in a way deemed appropriate for each error entry. In designing specific error-handling routines for an LR parser, we can fill in each blank entry in the action field with a pointer to an error routine that will take the appropriate action selected by the compiler designer.

The actions may include insertion or deletion of symbols from the stack or the input or both, or alteration and transposition of input symbols. We must make our choices so that the LR parser will not get into an infinite loop. A safe strategy will assure that at least one input symbol will be removed or shifted eventually, or that the stack will eventually shrink if the end of the input has been reached. Popping a stack state that covers a non terminal should be avoided, because this modification eliminates from the stack a construct that has already been successfully parsed.