UNIT -5

5.1 CANONICAL LR PARSING

By splitting states when necessary, we can arrange to have each state of an LR parser indicate exactly which input symbols can follow a handle a for which there is a possible reduction to *A*. As the text points out, sometimes the FOLLOW sets give too much informationand doesn't (can't) discriminate between different reductions.

The general form of an LR(k) item becomes [A -> a.b, s] where A -> ab is a production and s is a string of terminals. The first part (A -> a.b) is called the core and the second part is the lookahead. In LR(1) |s| is 1, so s is a single terminal.

A -> ab is the usual righthand side with a marker; any a in s is an incoming token in which we are interested. Completed items used to be reduced for every incoming token in FOLLOW(A), but now we will reduce only if the next input token is in the lookahead set s..if we get two productions A -> a and B -> a, we can tell them apart when a is a handle on the stack if the corresponding completed items have different lookahead parts. Furthermore, note that the lookahead has no effect for an item of the form [A -> a.b, a] if b is not e. Recall that our problem occurs for completed items, so what we have done now is to say that an item of the form [A -> a., a] calls for a reduction by A -> a only if the next input symbol is a. More formally, an LR(1) item [A -> a.b, a] is valid for a viable prefix g if there is a derivation

S =>* s abw, where g = sa, and either a is the first symbol of w, or w is e and a is $.

5.2 ALGORITHM FOR CONSTRUCTION OF THE SETS OF LR(1) ITEMS

Input: grammar G'

Output: sets of LR(1) items that are the set of items valid for one or more viable prefixes of G'

Method:

```
closure(I)
begin
repeat
for each item [A -> a.Bb, a] in I,
each production B -> g in G',
and each terminal b in FIRST(ba)
such that [B -> .g, b] is not in I do
add [B -> .g, b] to I;
until no more items can be added to I;
end;
```

5.3 goto(I, X)

```
begin
let J be the set of items [A -> aX.b, a] such that
[A -> a.Xb, a] is in I
return closure(J);
end;
procedure items(G')
begin
C := {closure({S' -> .S, $})};
repeat
for each set of items I 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;
```

An example,

Consider the following grammer,
S’->S
S->CC
C->cC
C->d
Sets of LR(1) items
I0: S’->.S,$
S->.CC,$
C->.Cc,c/d
C->.d,c/d

I1:S’->S.,$
I2:S->C.C,$
C->.Cc,$
C->.d,$

I3:C->c.C,c/d
C->.Cc,c/d
C->.d,c/d

I4: C->d.,c/d

I5: S->CC.,$

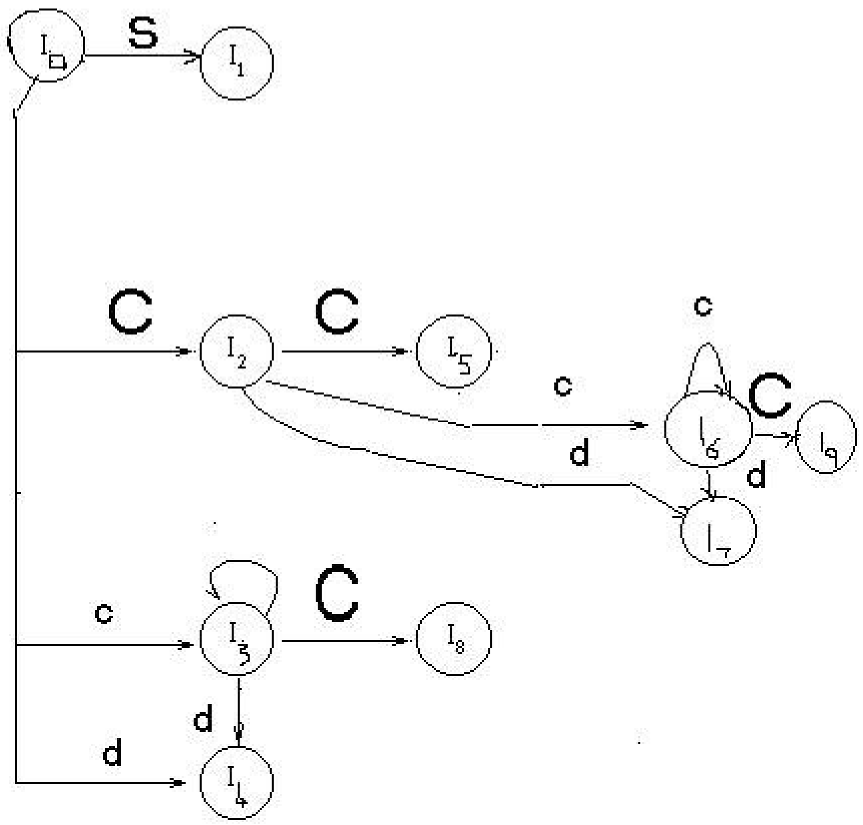
I6: C->c.C,$
C->.cC,$
C->.d,$

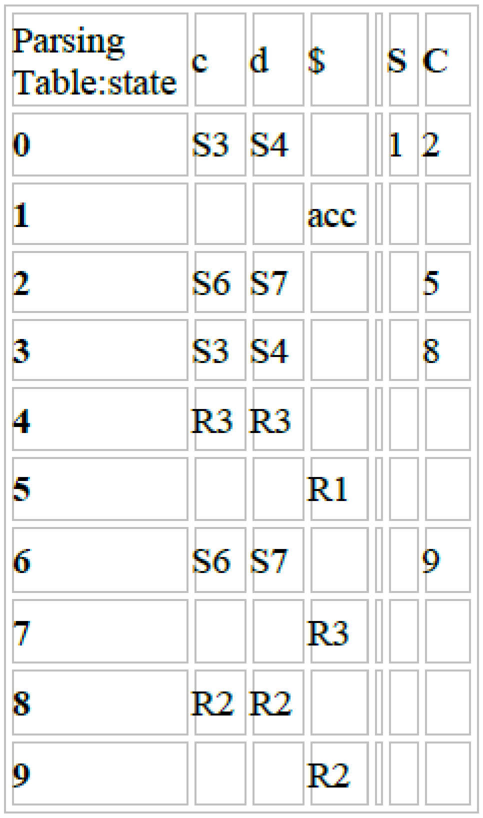
I7:C->d.,$

I8:C->cC.,c/d

I9:C->cC.,$

Here is what the corresponding DFA looks like





Parsing Table:state	c	d	$		S	C
0	S3	S4			1	2
1			acc			
2	S6	S7				5
3	S3	S4				8
4	R3	R3				
5			R1			
6	S6	S7				9
7			R3			
8	R2	R2				
9			R2			

5.4 ALGORITHM FOR CONSTRUCTION OF THE CANONICAL LR PARSING TABLE

Input: grammar G'
Output: canonical LR parsing table functions action and goto

1. Construct C = {I0, I1 , ..., In} the collection of sets of LR(1) items for G'.State i is constructed from Ii.
2. if [*A* -> a.ab, b>] is in Ii and goto(Ii, a) = Ij, then set action[i, a] to "shift j". Here a must be a terminal.
3. if [*A* -> 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*'.
4. if [*S*' -> *S*.] is in Ii, then set action[i, $] to "accept"
5. If any conflicting actions are generated by these rules, the grammar is not LR(1) and the algorithm fails to produce a parser.
6. 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.
7. All entries not defined by rules 2 and 3 are made "error".
8. The inital state of the parser is the one constructed from the set of items

   containing [*S*' -> .S, $].

5.5.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.

5.6 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		

5.7HANDLING 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.

5.8 DANGLING ELSE

The dangling else is a problem in computer programming in which an optional else clause in an If–then(–else) statement results in nested conditionals being ambiguous. Formally, the context-free grammar of the language is ambiguous, meaning there is more than one correct parse tree.

In many programming languages one may write conditionally executed code in two forms: the if-then form, and the if-then-else form – the else clause is optional:

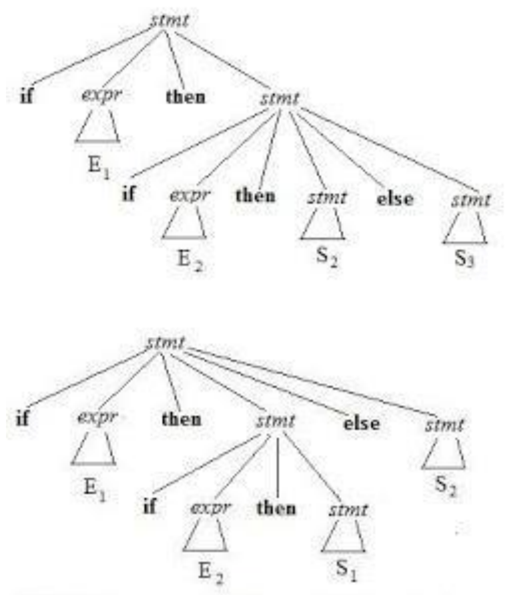


Fig 2.4 Two parse trees for an ambiguous sentence

Consider the grammar:

S ::= E $

E ::= E + E

| E * E

| (E)

| id

| num

and four of its LALR(1) states:

I0: S ::= . E $?

E ::= . E + E +*$

E ::= . E * E +*$

E ::= . (E) +*$

E ::= . id +*$

E ::= . num +*$

I1: S ::= E . $?

E ::= E . + E +*$

E ::= E . * E +*$

I3: E ::= E * E . +*$

E ::= E . + E +*$

I2: E ::= E * . E +*$

E ::= . E + E +*$

E ::= . E * E +*$

E ::= . (E) +*$

E ::= . id +*$

E ::= . num +*$

E ::= E . * E   +*$

Here we have a shift-reduce error. Consider the first two items in I3. If we have a*b+c and we parsed a*b, do we reduce using E ::= E * E or do we shift more symbols? In the former case we get a parse tree (a*b)+c; in the latter case we get a*(b+c). To resolve this conflict, we can specify that * has higher precedence than +. The precedence of a grammar production is equal to the precedence of the rightmost token at the rhs of the production. For example, the precedence of the production E ::= E * E is equal to the precedence of the operator *, the precedence of the production E ::= (E) is equal to the precedence of the token), and the precedence of the production E ::= if E then E else E is equal to the precedence of the token else. The idea is that if the look ahead has higher precedence than the production currently used, we shift. For example, if we are parsing E + E using the production rule E ::= E + E and the look ahead is *, we shift *. If the look ahead has the same precedence as that of the current production and is left associative, we reduce, otherwise we shift. The above grammar is valid if we define the precedence and associativity of all the operators. Thus, it is very important when you write a parser using CUP or any other LALR(1) parser generator to specify associativities and precedence's for most tokens (especially for those used as operators). Note: you can explicitly define the precedence of a rule in CUP using the %prec directive:

```
 E ::= MINUS E   %prec UMINUS
```

where UMINUS is a pseudo-token that has higher precedence than TIMES, MINUS etc, so that -1*2 is equal to (-1)*2, not to -(1*2).

Another thing we can do when specifying an LALR(1) grammar for a parser generator is error recovery. All the entries in the ACTION and GOTO tables that have no content correspond to syntax errors. The simplest thing to do in case of error is to report it and stop the parsing. But we would like to continue parsing finding more errors. This is called *error recovery*. Consider the grammar:

```
S ::= L = E ;
   | { SL } ;
   | error ;
SL ::= S ;
    | SL S ;
```

The special token error indicates to the parser what to do in case of invalid syntax for S (an invalid statement). In this case, it reads all the tokens from the input stream until it finds the first semicolon. The way the parser handles this is to first push an error state in the stack. In case of an error, the parser pops out elements from the stack until it finds an error state where it can proceed. Then it discards tokens from the input until a restart is possible. Inserting error handling productions in the proper places in a grammar to do good error recovery is considered very hard.

5.9LR 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.

5.10 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 A. The parser then stacks the state GOTO(s, A) and resumes normal parsing. 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.

5.11 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.