Module 3

Bottom-Up Parsing

Ques 1) Bottom-Up Parsing

What is bottom-up parsing? How they are different from top-down parsing?

Ans: Bottom-Up Parsing
A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (bottom) and working up towards the root (top).

For example, sequence of tree is shown in **figure 3.1**, illustrates a bottom-up parse of the token id*id, with respect to the expression grammar shown by equation (1).

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F \qquad \text{.....(1)}$$
$$F \rightarrow (E) \mid id$$

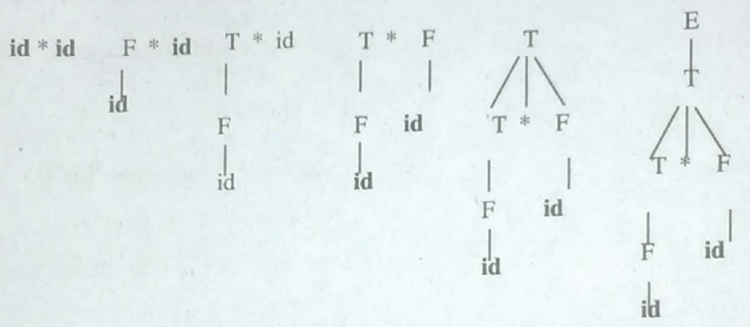


Figure 3.1: A Bottom-up Parse for id*id

Difference between Top-Down Parsing and Bottom-Up Parsing
The comparison between top-down and bottom-up parsing is shown in **table below**:

	Top-Down Parsing	Bottom-Up Parsing
1)	For the construction of parsing tree, it follows top down approval.	For construct of parsing tree, it follows bottom up approach
2)	The parse tree starts from boot to leaves in a Preorder manner.	The parse tree for this, starts from leaves to root.
3)	It top-down parsing non-terminal is expanded to derive the given input string.	In bottom-up parsing, input string is reduced to the starting of non-terminal.
4)	It follows only the hierarchy of forward reference.	It follows only the hierarchy of back truly.
5)	In top-down parsing, we have only the input and corresponding derive output.	In bottom up parsing, input string is reduced to starting non-terminal.

Ques 2) Explain shift-reduce parsing technique.

Ans: Shift-Reduce Parsing
Shift reducing is a form of bottom up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed. It is used to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

At each reduction step a string matching right side of a production is replaced by the symbol on left of that production.

For example, consider the grammar:
$S \rightarrow aABe$
$A \rightarrow Abc / b$
$B \rightarrow d$

The sentence **abbcde** can be reduced to S by the following steps.

abbcde	
aAbcde	∵ $A \rightarrow b$
aAde	∵ $A \rightarrow Abc$
aABe	∵ $B \rightarrow d$
S	∵ $S \rightarrow aABe$

Reduction: Each replacement of the right side of a production by the left side in the process above is called a reduction.

Handle: The right hand side string, which can be reduced to left side production variable called handle. In the above case **b** is the first handle and **Ab** is the second handle and so on.

Note: "If grammar is unambiguous, then every right sentential form of the grammar has exactly one handle".

Ques 3) Define the stack implementation of shift reduce parsing.

Ans: Stack Implementation of Shift Reduce Parsing
A convenient way to implement a shift reduce parser is to use a stack and an input buffer. We use $ to mark the bottom of the stack and also the right end of the input.

Stack	Input
Initial State → $	w$
-	-
(Shift reduce Steps)	
-	-
Final State → $S	$

→ Stack contains the start symbol and input is empty

For example, for the following grammar:
$E \rightarrow E + E \mid E * E \mid (E) \mid id$

We Show the various shift-reduce parsing actions with respect to the input string as $Id_1 + Id_2 * Id_1$.

Stack	Input	Action
$	$id_1 + id_2 * id_3$ $	shift
$id_1	$+ id_2 * id_3$ $	reduce by $E \rightarrow id$
$E	$+ id_2 * id_3$ $	shift
$E +	$id_2 * id_3$ $	shift
$E + id_2	$* id_3$ $	reduce by $E \rightarrow id$
$E + E	$* id_3$ $	shift
$E + E *	Id_3 $	shift
$E + E * id_3	$	reduce $E \rightarrow id$
$E + E * E	$	reduce $E \rightarrow E * E$
$E + E	$	reduce $E \rightarrow E + E$
$E	$	accept

There are possibly four actions a shift reduce parser can make:
1) **Shift**: In a shift action, the next input symbol is shifted to the top of the stack.
2) **Reduce**: In a reduce action; the parser knows the right end of the handle is at the top of the stack. It must then locate the left end of the handle within the stack and decide with what non-terminal to replace the handle.
3) **Accept:** In an accept action, the parser announces successful completion of parsing.
4) **Error:** In an error action, the parser discovers that a syntax error has occurred and calls an error recovery routine.

Ques 4) Write a note on handle pruning.

Ans: Handle Pruning
Bottom-up parsing during a left-to-right scan of the input constructs a right-most derivation in reverse. Informally, a "handle" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.

For example, consider the grammar:
$E \rightarrow E + E / E * E / (E) / id$ and input string $\mathbf{id_1 + id_2 * id_3}$

Reductions made by Handle Pruning		
Right Sequential From	**Handle**	**Reducing Production**
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

Ques 5) Explain operator precedence grammar.

Ans: Operator Precedence Grammars
An operator precedence grammar is an $\in$-free Operator grammar in which the precedence relations $<\!\cdot$, $\doteq$ and $\cdot\!>$ are disjoint, i.e., for any pair of terminals **a** and **b**, never more than one of the relations $a <\!\cdot b$, $a \doteq b$, and $a \cdot\!> b$ is true.

We can define the term "operator-precedence grammar," Let G be an ε-free operator grammar (i.e., no right side is ε and no right side has a pair of adjacent non-terminals). For each two terminal symbols a and b, we say:

1) $a \doteq b$ if there is a right side of a production of the form $s \rightarrow \alpha a \beta b \gamma$, where β is either $\in$ or a single non-terminal. That is $a \doteq b$ (if a appears immediately to the left of b in right side, or if they appear separated by one non-terminal.)

   For example, (a) production $S \rightarrow iCtSeS$ implies that $i \doteq t$ and $t \doteq e$.

   (b) $E \rightarrow (E)$ implies that $(\doteq)$
   ↑↑

2) $a <\!\cdot b$ if for some non-terminal A there is a right side of the form $(S \rightarrow \alpha a A \beta)$, and $A \overset{+}{\Rightarrow} \gamma b \delta$, where γ is either $\in$ or a single non-terminal. That is, $a <\!\cdot b$ (if a non-terminal A appears immediately to the right of a and derives a string in which b is the first terminal symbol.)

   For example, we have i immediately to the left of C in $S \rightarrow iCtS$ and $A \overset{+}{\Rightarrow} b$, so $i <\!\cdot b$. The derivation is one step, and $\gamma \doteq \in$ also, $\$ <\!\cdot b$ if there is derivation $S \overset{+}{\Rightarrow} \gamma bs$ and γ is $\in$ or a single non-terminal.

3) $a \cdot\!> b$ if for some non-terminal A there is a right side of the form $S \rightarrow \alpha A b \beta$ and $A \overset{+}{\Rightarrow} \gamma a \delta$ where γ is either $\in$ or a single non-terminal. That is ,$a \cdot\!> b$ if a non-terminal appearing immediately to the left of b derives a string whose last terminal is a.

   For example, $S \rightarrow iCtS$ and $C \overset{+}{\Rightarrow} b$ imply $b \cdot\!> t$

Ques 6) Consider the grammar:
$\mathbf{E \rightarrow E + E / E * E / (E) / id}$
This grammar is not an operator-precedence grammar because two precedence relations hold between certain pair of terminals.

Ans: We shall show that two precedence relations hold between + and + .
By rule (iii)

$\alpha = \in$, $A = E$,	$b = +$, and $\beta = E$
$E \rightarrow \varepsilon E + E$	$E \rightarrow E + E$
Like $\alpha A b \beta$	$A \xrightarrow{+} \gamma a \delta$
$a > b$ $(+ > +)$	$\gamma = E$, $a = +$, $\delta = E$

Apply rule (ii) definition of <.

$E \rightarrow E + E$	$E \xrightarrow{+} E + E$	$\alpha = E$
$\alpha\, a\, A\, \beta$	$A \xrightarrow{+} \gamma b \delta$	$a = +$
	$A = E$	
	$\beta = \varepsilon$	
$a <\cdot b$	$(+ <\cdot +)$	

Thus, + .> + $ + <. + , so it is not an operator precedence grammar .

Ques 7) Define the concept of operator precedence parsing.

Ans: Operator - Precedence Parsing

For a certain small class of grammars we can easily construct efficient shift reduce parsers by hand. These grammars have the property that:

1) No production right side is ∈ (Unambiguous Grammar)
2) No production has two adjacent non-terminals.

For example, the following grammar for expressions

$E \rightarrow EAE \mid (E) \mid -E \mid id$

$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

is not an operator grammar, because the right side EAE has two consecutive non-terminals.

So substitute A in E → EAE

$E \rightarrow E + E / E - E / E * E / E/E / E \uparrow E / (E) / -E / id$

Now it is an Operator Grammar.

In operator-precedence parsing, we use three disjoint precedence relations, <·, ≐ and ·>, between certain pairs of terminals. These precedence relations guide the selection of handles.

There are two common ways of determining what precedence relation should hold between a pair of terminals.

1) **First method** we discuss is intuitive and is based on the traditional notions of associativity and precedence of operators. **For example,** if * is to have higher precedence than +, we make + <· * and *·>+.

2) **Second method** of selecting operator-precedence relations is to first construct an unambiguous grammar for the language, a grammar which reflects the correct associativity and precedence in its parse trees.

Relation	**Meaning**
a <· b	a "yields precedence to" b
a ≐ b	a "has the same precedence as" b
a ·> b	a "takes precedence over" b

Note: $ < b or b> $ has low priority than all terminals b

Suppose we initially have the right sentential from **id + id * id** and the precedence relations as:

	id	**+**	*****	**$**
id	-	·>	·>	·>
+	<·	·>	<·	·>
*****	<·	·>	·>	·>
$	<·	<·	<·	–

Then the string with precedence relations inserted is

$ <· id ·> + <· id ·> * <· id ·> $

Now the handle can be found by the following process:

Step 1) Scan the string from the left and until the left most ·> is encountered.

Step 2) Then scan backwards (to the left) over any ≐ 's until a <· is encountered.

Step 3) Reduce between <· …. ·>.

Operator-Precedence Parsing Algorithm
Algorithm: Operator-precedence parsing algorithm

Input: The precedence relations from some operator-precedence grammar and an input string of terminals from that grammar.

Output: Strictly speaking, there is no output. We could construct a skeletal parse tree as we parse, with one non-terminal labeling all interior nodes and the use of single productions not shown. Alternatively, the sequence shift-reduce steps could be considered the output.

Method: Let the input string be $a_1a_2..a_n$ \$. Initially, the stack contain \$.

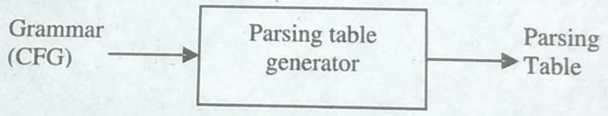
```
repeat forever
if only $ is on the stack and only $ is on the input then
accept and break
else
begin
let a be the topmost terminal symbol on the stack and let b be the current input symbol;
if a <· b or a ≐ b then begin
push  b onto the stack
end;
else if a ·> b then /* reduce */
repeat pop the stack
until the top stack terminal is related by <·
to the terminal most recently popped
else call the error correcting routine
end
```

Figure 3.2: Operator Precedence Parsing Algorithm

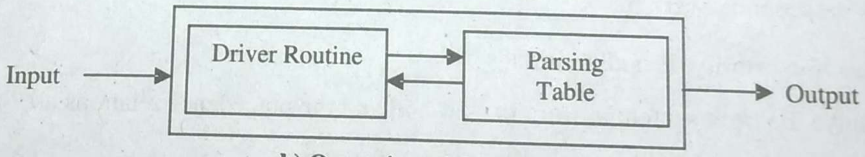
Ques 8) What is LR parsing? Write the algorithm for LR parsing.

Ans: LR Parsing
LR parsing is a bottom - up parsing technique which can used to parse a large class of context free grammars. LR stands for left to right parsers, because it scans input symbols from left to right and constructing a rightmost derivation in reverse that is called **canonical derivations**.

Logically, LR Parser consists of two parts:
1) **Driver Routine:** Same for all LR Parsers
2) **Parsing Table:** Changes from one Parser to another



a) Generating the Parsing Table



b) Operation of the Parser

Figure 3.3: Generating an LR parser

The LR Parser has an input, a stack and a parsing table:

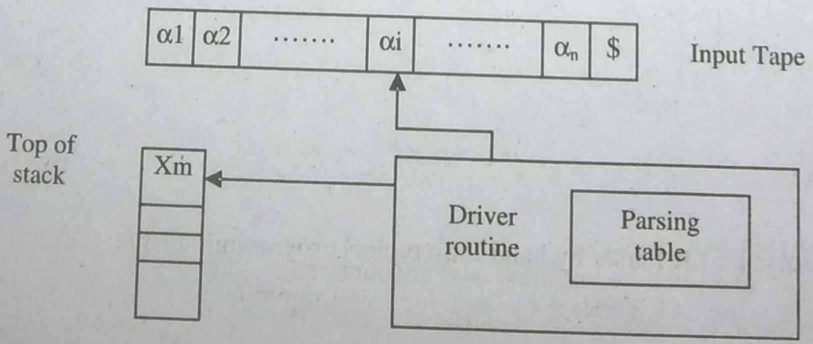


Figure 3.4: L R Parser

Parsing table consists of two parts:

1) **ACTION Function :** The entry ACTION [s_m, a_i] can have one of four values:

i) shift s	ii) reduce A→ β
iii) accept	iv) error

2) **GOTO Function:** It takes a state and grammar symbol as arguments and produces a state.

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

$$(s_0 X_1 s_1 Z_2 s_2 \ldots . X_m s_m, a_i a_{i+1} \ldots . a_n \$)$$

The next move of the parser is determined by reading a_i, the current input symbol, and s_m, the state on top of the stack, and then consulting the parsing action table entry ACTION[s_m, a_i]. The configurations resulting after each of the four types of move are as follows:

i) If ACTION [S_m, a_i] = **shift s,** the parser executes a shift move.
ii) If ACTION [S_m, a_i] = **reduce A→ β,** then the parser executes a reduce move.
iii) If ACTION [S_m, a_i] = **accept,** parsing is completed.
iv) If ACTION [S_m, a_i] = **error**, the parser has discovered an error and calls an error recovery routine.

LR Parsing Algorithm

Input: An input string s and an LR parsing table with functions **action** and **goto** for a grammar G.

Output: If s is in L (G), a bottom - up parse for s; else an error indication.

Method: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and s$ in the input buffer. The parser then executes the algorithm mentioned below until an **accept** or **error** action is encountered.

Algorithm

Set the input pointer to the first symbol of s$;

```
  repeat forever begin
    let x be the state on top of the stack and
        a the symbol pointed by the input pointer;
    If action [x, a] = shift $x_1$ then begin
        push a and then $x_1$ on top of the stack;
    advance the input pointer to the next input symbol
  end
  else if action [x, a] = reduce A → β then begin
        pop (2 * β) symbols off the stack;
        let $x_1$ be the state now on top of the stack;
        push A then goto [$x_1$, A] on top of the stack;
    end
      else if action [x, a] = accept then
          return
      else error ( )
  end
```

Ques 9) What are the advantages and disadvantages of LR Parsers?

Ans: Advantages of LR Parsers

1) These are constructed virtually to recognize all language constructs for which CFG can be written.
2) The LR parsing method is most important non-back-tracking shift-reduce parsing method and easy to implement.
3) The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.
4) 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.

Disadvantages of LR Parsers

1) It is required more time to construct a LR Parser by hand for a typical programming language grammar.
2) Specialized tool is required to construct a LR Parser by hand for a programming language grammar (such as LR Parser generator tool).
3) It is difficult to construct LR Parser generator, if the grammar is ambiguous.

Ques 10) What is difference between LL and LR parsing?

Ans: Difference between LL and LR Parsing
Table below shows the difference between LL and LR parsing:

Table: LL *versus* LR Parsing

LL	LR
Does a leftmost derivation.	Does a rightmost derivation in reverse.
Starts with the root nonterminal on the stack.	Ends with the root nonterminal on the stack.
Ends when the stack is empty.	Starts with an empty stack.
Uses the stack for designating what is still to be expected.	Uses the stack for designating what is already seen.
Builds the parse tree top-down.	Builds the parse tree bottom-up.
Continuously pops a nonterminal off the stack, and pushes the corresponding right hand side.	Tries to recognize a right hand side on the stack, pops it, and pushes the corresponding nonterminal.
Expands the non-terminals.	Reduces the non-terminals.
Reads the terminals when it pops one off the stack.	Reads the terminals while it pushes them on the stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parse tree.

Ques 11) List out the different methods of constructing LR parsing table?

Or

Discuss the method of constructing SLR parsing tables.

Ans: Methods of Constructing LR Parsing Table
There are three methods to construct LR Parsing table for a given grammar:
1) SLR (Simple LR)
2) Canonical LR
3) LALR – Look Ahead LR

Constructing SLR(Simple LR) Parsing Table
SLR or Simple LR is the simplest parser in LR family.

A Simple LR parser or SLR parser is an LR parser for which the parsing tables are generated as for an LR(0) parser except that it only performs a reduction with a grammar rule A → w if the next symbol on the input stream is in the follow set of A.

Rules for Constructing the SLR Parsing Table
1) Let C = {X_0, X_1, ……. X_n}, the states of the parsers are 0, 1, 2, ……n.
2) State i being constructed from X_i. The parsing actions for state i are determined as follows:
   i) If [$A \rightarrow \alpha.a\beta$] is in X_i and GOTO (X_i, a) = X_j then set ACTION [i, a] to "shift j", here "a" is terminal.
   ii) If [$A \rightarrow \alpha.$] is in X_i, then set ACTION [i, a] to "reduce $A\rightarrow \alpha$" for all "a" in FOLLOW (A), here "a" is must not be S'.
   iii) If [$S' \rightarrow S.$] if in X_i, then set ACTION [i, $] to "accept".
3) The GOTO functions for state i, are constructed using the rule:
   If GOTO [X_i, A] = X_i then GOTO [i, A] = j
4) All entries not defined by rule (2) and (3) are made "error".
5) The initial state of the parser is the one constructed from the set of items containing $S'\rightarrow.S$

Example: Construct the SLR parsing table for given grammar, and also find the canonical collection of sets of items.
i) $E \rightarrow E + T$
ii) $E \rightarrow T$
iii) $T \rightarrow T * F$
iv) $T \rightarrow F$
v) $F \rightarrow (E)$
vi) $F \rightarrow id$

Consider X_0: $E' \rightarrow .E$
$E \rightarrow .E + T$
$E \rightarrow .T$

$T \rightarrow .T * F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

Now consider X_1:

$E' \rightarrow E.$
$E \rightarrow E. + T$

Consider X_2:

$E \rightarrow T.$
$T \rightarrow T. * F$

Since FOLLOW(E) ={ $,+,) }, the first item makes action [2 , $]=action[2 , +]=action[2 ,)] = reduce E →T ,the second item makes action [2, *] = Shift 7.

Note: Every SLR (1) grammar is an LR (1) grammar, but for a SLR (1) grammar the canonical LR parser may have more states than the SLR parser for the same grammar.

Ques 12) Explain how to construct the canonical LR parsing tables.

Ans: Constructing the Canonical LR Parsing Tables
For removing such contradictory results as we have seen in above example, it is possible to carry more information in the state that will allow us to rule out some of these invalid reductions.

The extra information is incorporated into the state by redefining items to include a terminal symbols as a second component.

The general form of item becomes [A → α.β, a], where A → α β is a production and **a** is a terminal or a right endmarker $.

We call such an object as LR (1) item. The 1 refers to the length of second component, called the lookahead of the item. The lookahead has no effect in an item of the form [A → α.β, a], where β is not ∈,but an item of the form [A → α. , a] calls for a reduction by A → α only if the next input symbol is **a**.

Rules for Making Canonical LR Parsing Table

i) Construct C = { I_0, I_1, ….. I_n}, the collection of sets of LR (1) items for G_1.
ii) State i for the parser is constructed from Ii, the parsing actions for state i are determined as follows:
   a) If [A → α.aβ, b] is in I_i and goto (I_i, a)= Ij, then set action [i, a] to "shift j".
   b) If [A → α., a] is in I_i, then set action [i, a] to "reduce A → α."
   c) If [S' → S., $] is in I_i then set action [i, $] to "accept".

Ques 13) Consider the following augmented grammar.

S' → S
S → CC
C → cC/d

1) **For starting production add $ as second component**
   S' → S, $
   S' →∈ S ∈, $
   S → .CC, $

2) **Function closure [A → α.Bβ, a] tells us to add [B → .γ, b] for each production B → γ and terminal b in FIRST (βa).**

Ans: The initial set of item is:

X_0:	S' →.S, $	
r_1	S →.CC, $	FIRST(∈ $)
r_2	C → .cC, c/d	FIRST(c $)
r_3	C →.d, c/d	FIRST (c $)

FIRST (C$) = FIRST (C) = terminals c & d

Next set of items are:

X_1: S' → S., $

X_2: S → C. C, $
C → .cC, $
C → .d, $

X_3: C → c.C, c/d
C → .cC, c/d
C → .d, c/d

X_4: C → d., c/d

X_5: S → C C., $

X_6: C → c.C, $
C → .cC, $
C → .d, $

Canonical Parsing Table

State	ACTION			GOTO	
	c	d	$	S	C
0	s_3	s_4		1	2
1			accept		
2	s_6	s_7			5
3	s_3	s_4			8
4	r_3	r_3			
5			r_1		
6	s_6	s_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

For the above table the parsing action and goto functions produced is called LR (1) parsing table.

An LR parser using this table is called a **canonical LR parser**. If the parsing action function has no multiply-defined entries, then the grammar is called on LR (1) grammar.

Note: Every SLR (1) Grammar is an LR (1) grammar, but for an SLR (1) grammar the canonical LR parser may have more states than the SLR parser for the same grammar.

If a conflict results from the above rules, the grammar is said not to be LR (1) and the above rules are said to fail.

Ques 14) Discuss the method of constructing LALR parsing tables.

Ans: Constructing LALR (Look Ahead LR) Parsing Tables
The characteristics of LALR are:
1) Intermediate between SLR and canonical LR.
2) It works for most of the programming languages.

Construction LALR Parsing Tables (Look-Ahead-LR)
The table obtained by it is considerably smaller than canonical LR tables.

Note: SLR and LALR tables for a grammar always have the same number of states.

We can construct the LALR parsing table by using the following algorithm:
Algorithm: Construction of LALR parsing table

Input : Grammar G

Output : Action table and goto table for LALR parser

Method:
1) Construct an augmented grammar G' for the given grammar G.
2) Construct C = {I_0, I_1, ... I_n} of LR(1) items.
3) For each core in the set of LR (1) items, locate all sets having the same core. Replace these sets by their union.
4) Let C' = {k_0, k_1,.... k_m} be the reduced collection of LR{1) items , we can construct the action table in the same way as in LR parser.
5) If K = {I_1U I_2U ... UI_p} then goto [K, X] = r, where r is the union of all sets of items having the same core as goto [I_1, X].

Working Procedure: Consider the following augmented grammar.
S' → S
S → CC
C → cC/d

The initial set of items are:

I_0 S' → .S, $

r_1: S → .CC, $

r_2: C → .cC, c/d

r_3: C → .d, c/d

I_1:   S' → S; $

I_2:   C → C.C, $

C → .cC, $

C → d, $

I_3:   C → c.C, c/d

C → .cC, c/d

C → .d, c/d

I_4:   C → .d, c/d

I_5 :   S → CC., $

I_6 :   C → C.C, $

C → .cC, $

C → .d, $

I_7:   C → d., $

I_8:   C → cC., c/d

I_9:   C → cC., $

In above example, we have calculated LR (1) items. Take a pair of similar looking states, such as I_4 and I_7 → each of these state differ in 2nd component only.

Let us now replace I_4 and I_7 by I_{47}, the union of I_4 and I_7 $C \rightarrow \underbrace{d.}_{\text{core}},c/d, \$$

Merging of states with common core can never produce a shift reduce conflict, because that was not present in one of the original states, since shift action depend on the core, not the look ahead or (second component). It is possible, however that a merger will produce a reduce-reduce conflict. So after applying the above procedure we get:

I_{36}: C → c.C, c/d/$

C → .cC, c/d/$

C → .d, c/d/$

I_{47}: C → d., c/d/$

I_{89}: C → cC., c/d/$

Follow (S) = {$}

Follow (C) = {c, d, $}

LALR Parsing Table

State	ACTION			Goto	
	c	d	$	S	C
0	s_{36}	s_{47}		1	2
1			accept		
2	s_{36}	s_{47}			5
36	s_{36}	s_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		