Unit-3

Bottom Up Parsing

• This parsing constructs the parse tree for an input string beginning at the leaves and working up towards the root.
• General style of bottom-up parsing is shift-reduce parsing.

Shift – Reduce Parsing
Reduce a string to the start symbol of the grammar. It simulates the reverse of right most derivation.
In every step a particular sub string is matched (in left right fashion) to the right side of some production and then it is substituted by the non terminal in the left hand side of the production.

For example consider the grammar
S → aABe
A → Abc/b
B → d
In bottomup parsing the string 'abbcde' is verified as

Abbcde
aAbcde
aAde } reverse order
aABe
S

Stack Implementation of Shift-Reduce Parser: The shift reduce parser consists of input buffer, Stack and parse table.
Input buffer consists of strings, with each cell containing only one input symbol.

Stack contains the grammar symbols, the grammar symbols are inserted using shift operation and they are reduced using reduce operation after obtaining handle from the collection of buffer symbols.

Parse table consists of 2 parts goto and action, which are constructed using terminal, non-terminals and compiler items.
Let us illustrate the above stack implementation.
Consider a grammar be
S → AA
A → aA
A → b

Let the input string 'ω' be abab$ i.e.,
ω = abab$

Stack	**Input string**	**Action**
$	abab$	Shift
$a	bab$	Shift
$ab	ab$	Reduce(A → b)
$aA	ab$	Reduce(A → aA)
$A	ab$	Shift
$Aa	b$	Shift
$Aab	$	Reduce (A → b)
$AaA	$	Reduce(A → aA)
$AA	$	Reduce(S → AA)
$S	$	Accept

Rightmost Derivation

S=> aABe =>aAde=> aAbcde=>abbcde
For bottom up parsing, we are using right most derivation in reverse.

Handle of a String: Substring that matches the RHS of some production and whose reduction to the nonterminal on the LHS is a step along the reverse of some rightmost derivation.

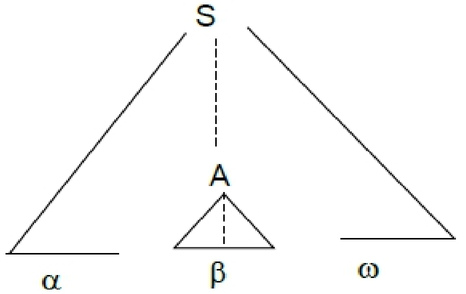
Unit-3

$$S \overset{rm}{\underset{*}{\Rightarrow}} \alpha A r \Rightarrow \alpha \beta r$$

Right sentential forms of a unambiguous grammar have one unique handle.
Ex: For grammar
S → aABe
A → Abc/b
B → d
S=>$\underline{aABe}$=>aA$\underline{d}$e =>a$\underline{Abc}$de=> a$\underline{b}$bcde
(Underline part are handles)

Handle Pruning: The process of discovering a handle and reducing it to the appropriate left hand side is called handle pruning. Handle pruning forms the basis for a bottomup parsing.
To construct the rightmost derivation:
S = r0 => r1 => r2 --- => rn = w
Apply the following simple algorithm:
for i ← n to 1
find the handle Ai → Bi in ri
Replace Bi with Ai to generate ri -1

Consider the cut of a parse tree of a certain right sentential form:



Here A →β is a handle for αβω.

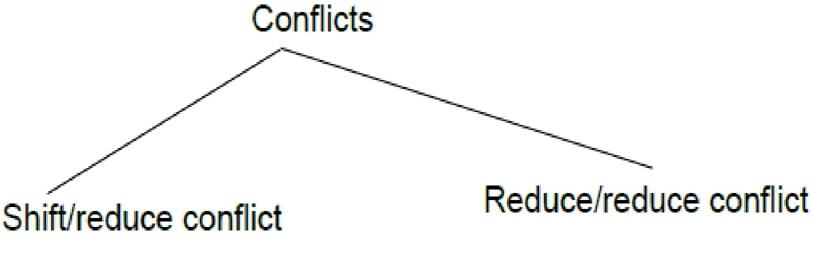
Shift Reduce Parsing with a Stack: There are 2 problems with this technique:
(i) to locate the handle
(ii) decide which production to use

General Construction using a Stack

1. **"Shift"** input symbols onto the stack until a handle is found on top of it.
2. **"Reduce"** the handle to the corresponding non - terminal.
3. **"Accept"** when the input is consumed and only the start symbol is on the stack.
4. **"Errors"** – call an error reporting/recovery routine.

Viable Prefixes: The set of prefixes of a right sentential form that can appear on the stack of a shift reduce parser are called viable prefixes. .

Conflicts



Shift/reduce conflict
Ex: stmt → if expr then stmt | if expr then stmt else stmt | any other statement

Unit-3

If exp then stmt is on the stack, in this case we can't tell whether it is a handle. i.e., "shift/reduce" conflict.

Reduce/reduce conflict

Ex: S → aA/bB
A → c
B →c
W = ac it gives reduce/reduce conflict.

Operating Precedency Parser

Operator Precedence Grammar

In operator grammar, no production rule can have:
- at the right side.
- two adjacent nonterminals at the right side.

Ex 1: E → E + E /E – E/ id is operator grammar.

Ex 2: E → A
A → a } not operator grammar
B → b

Ex 3: E → E0E/id →is an operator grammar

Precedence Relation: If a < b then "b" has higher precedence than "a"

a = b then "b" has same precedence as "a"
a > b then "b" has lower precedence than "a"
Common ways for determining the precedence relation between pair of terminals:

1. Traditional notations of associativity and precedence.
   Ex: x has higher precedence than +
   x ⋗ + (or) + ⋖ ×
2. First construct an unambiguous grammar for the language which reflects correct associativity and precedence in its parse tree.

Operator Precedence Relations from Associativity & Precedence: Let us use $ to mark end of each string. Define $ b and b ⋗ $ for all terminals b. Consider the grammar:

E → E + E/E × E/id

Let the operator precedence table for this grammar:

	id	+	×	$
id		⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
×	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	accept

Unit-3

To find the Handle

1. Scan the string from left until ⋗ is encountered
2. Then scan backwards (to left) over any = until ⋖ is encountered.
3. The handle contains everything to the left of the first ⋗ and to the right of the ⋖ Is encountered.

After inserting precedence relation to a string
$id + id * id $ is
$ ⋖ id ⋗ + ⋖ id ⋗ * ⋖ id ⋗ $

Precedence Functions: Instead of storing the entire table of precedence relations table, we can encode it by precedence functions f and g, which map terminal symbols to integers:

1. f(a) ⋖ f(b) whenever a ⋖ b
2. f(a) = f(b) whenever a = b
3. f(a) > f(b) whenever a ⋗ b

Finding Precedence Functions for a Table

1. Create symbols f(a) and g(a) for each ‘a’ that is a terminal or $.
2. Partition the created symbols into as many groups as possible in such away that a = b then f(a) and g(b) are in the same group
3. create a directed graph

If a < b then place an edge from g(b) to f(a)
If a > b then place an edge from f(a) to g(b)

4. If the graph constructed has a cycle then no precedence function exists.

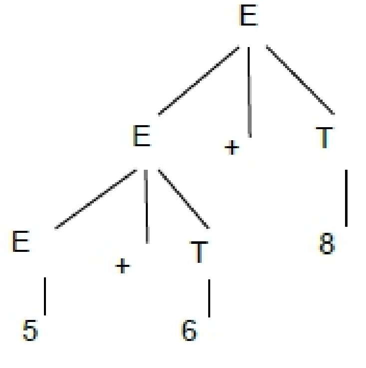
If there are no cycles, let f(a) be the length of the longest path being at the group of f(a).
Let g(a) be the length of the longest path from the group of g(a).

Deciding Associativity of operator from given grammar

(1) If the grammar is Left recursive then the operator is left associative.
(2) If the grammar is Right recursive then the operator is Right associative.

Example 1:

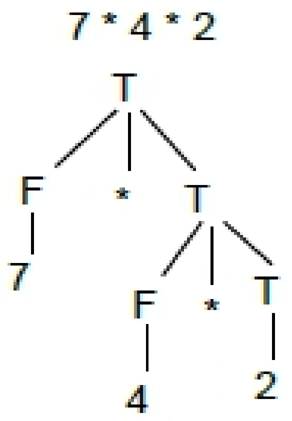
E → E + T
Here ‘E’ is present on LHS of production, therefore the grammar is left recursive.
The operator ‘+’ is left associative.
5 + 6 + 8
Here both operator ‘+’ is at same precedence, then which operation should be performed first is decided by associative 5 is added with 6 first then ‘8’ is added.



Example 2:

T → F * T
The ‘T’ is present on RHS of production, therefore the grammar is Right recursive.
The operator ‘*’ is right associative.

Unit-3

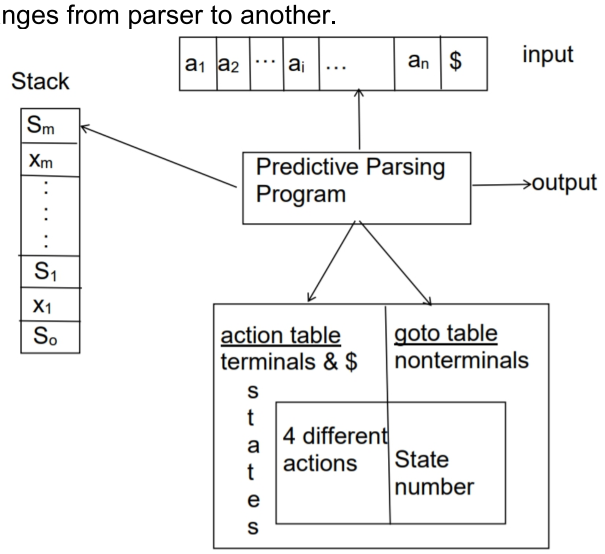


Disadvantages of operator precedence parsing
• It cannot handle unary minus.
• Difficult to decide which language is recognized by grammar.
Advantages:
1. Simple
2. Powerful enough for expressions in programming language.
Error cases:
1. No relation holds between the terminal on the top of stack and the next input symbol.
2. A handle is found, but there is no production with this handle as the right side.

Error Recovery:
1. Each empty entry is filled with a pointer to an error routine.
2. Based on the handle, it tries to recover from the situation.
To recover, we must modify (insert/change)
1. Stack or
2. Input or
3. Both
We must be careful that we don't get into an infinite loop.

LR(K) Parsers

The LR Parsing Algorithm
• It consists of an input, an output, a stack, a driver program and a parsing table that has two parts (action and goto).
• The driver/parser program is same for all these LR parsers, only the parsing table changes from parser to another.



Stack: To store the string of the form,
$S_o x_1 S_1 \ldots x_m S_m$ where

Unit-3

S_m: state
x_m : grammar symbol
Each state symbol summarizes the information contained in the stack below it.
Parsing Table: Parsing table consists of two parts:
1. Action part 2. Goto part

1. ACTION Part:
Let, $S_m \rightarrow$ top of the stack
$a_i \rightarrow$ current symbol
then action [Sm, a_i] which can have one of four values:
(i) shift S, where S is a state
(ii) reduce by a grammar production $A \rightarrow \beta$
(iii) accept
(iv) error

2. GOTO Part:
• If goto (S, A) = X where S → state, A → non-terminal, then GOTO maps state S and Non-terminal A to state X.

Configuration: ($S_o x_1 S_1 x_2 S_2$---$x_m S_m$, $a_i a_{i+1}$--- a_n\$)
The next move of the parser is based on action [S_m, a_i]

The configurations are as follows.

1. If action [S_m, a_i] = shift S
(S_o $x_1 S_1$ $x_2 S_2$--- $x_m S_m$, $a_i a_{i+1}$ --- a_n\$)
2. If action [S_m, a_i] = reduce $A \rightarrow \beta$ then
(S_o $x_1 S_1$ $x_2 S_2$--- $x_{m-r} S_{m-r}$, AS, $a_i a_{i+1}$ --- a_n\$)
Where S = goto [S_{m-r}, A]
3. If action [S_m, a_i] = accept, parsing is complete.
4. If action [S_m, a_i] = error, it calls an error recovery routine

Constructing of LR (K) Parser

Ex 1: Parsing table for the following grammar is shown below:
1. $E \rightarrow E + T$ 2. $E \rightarrow T$ 3. $T \rightarrow T * F$ 4. $T \rightarrow F$ 5. F → € 6. $F \rightarrow id$

Action								goto		
State	id	+	*	(	)	\$		E	T	F
0	S_5			S_4				1	2	3
1		S_6				acc				
2		r_2	S_7		r_2	r_2				
3		r_4	r_4		r_4	r_4				
4	S_5			S_4				8	2	3
5		r_6	r_6		r_6	r_6				
6	S_5			S_4					9	3
7	S_5			S_4						10
8		S_6			S_1					
9		r_1	S_7		r_1	r_1				
10		r_3	r_3		r_3	r_3				
11		r_5	r_5		r_5	r_5				

Moves of LR parser on input string id*id+id is shown below:

Stack	Input	Action
0	id * id + id$	Shift 5
0id 5	* id + id$	reduce 6 means reduce with 6th production F → id andgoto [0, F] = 3
0F 3	* id + id$	reduce 4 i.e T → F goto [0, T] = 2
0T 2	* id + id$	Shift 7
0T2 * 7	id + id$	Shift 5
0T2 * 7 id 5	+ id$	reduce 6 i.e F → id goto [7, F] = 10
0T2 * 7 F 10	+ id$	reduce 3 i.e T → T *F
0T 2	+ id$	goto [0, T] = 2
0E 1	+ id$	reduce 2 i.e E → T & goto [0, E] = 1
0E1 + 6	id$	Shift 6
0E1 + 6 id 5	$	Shift 5
0E1 + 6F 3	$	reduce 6 & goto [6, F] = 3
0E1 + 6T 9	$	reduce 4 & goto [6, T] = 9
0E1	$	reduce 1 & goto [0, E] = 1
0E1	$	accept

Constructing SLR Parsing Table

LR(0) item: LR (0) item of a grammar G is a production of G with a dot at some position of the right side of production.

Ex: A → BCD
Possible LR(0) items are
A → .BCD
A → B.CD
A → BC.D
A → BCD.

A → B.CD means we have seen an input string derivable from B and hope to see a string derivable from CD.

The LR(0) item are constructed as a DFA from grammar to recognize viable prefixes. The items can be viewed as the states of NFA.

The LR(0) item (or) canonical LR(0) collection, provides the basis for constructing SLR parser.

To construct LR (0) items, define
(a) an augmented grammar
(b) closure and goto

Augmented Grammar (G^1): If G is a grammar with start symbol S, G^1 the augmented grammar for G, with new start symbol S1 and production S^1 → S.

Purpose of G^1 is to indicate when to stop parsing and announce acceptance of the input.

Closure Operation: Closure (I) includes

1. Initially, every item in I is added to closure (I)
2. If A → α.Bβ is in closure (I) and β → γ is a production then add B → .γ to I.

Goto Operation: Goto (I, x) is defined to be the closure of the set of all items [A → αX.β] such that [A → α.Xβ] is in I.

Items

Kernel items: S^1 → .S and all items whose dots are not at the left end

Non kernel items: which have their dots at the left end.

Construction of Sets of Items

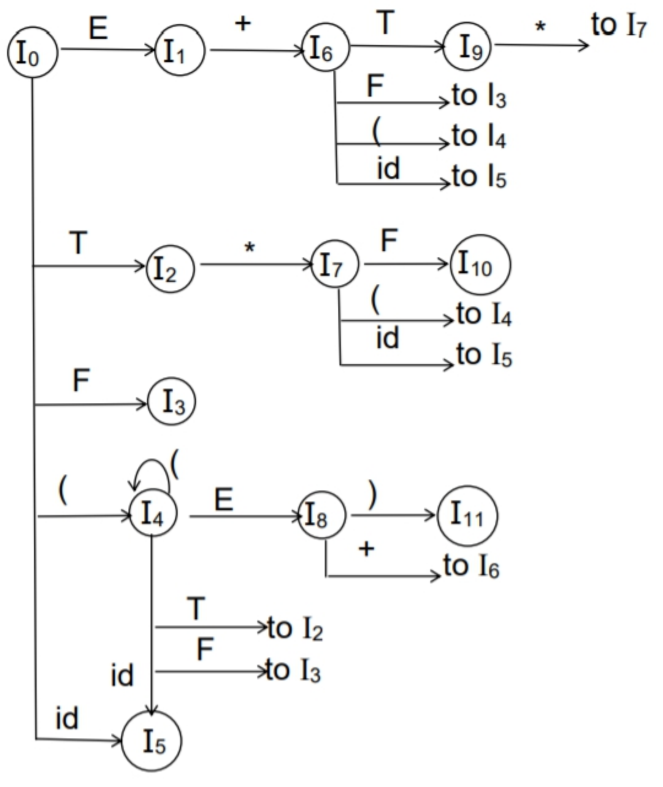
Procedure items (G^1)
begin
C: = closure({[S^1 → .S]});
repeat
for each set of items I in C and each grammar symbol x
such that goto(I, x) is not empty and not in C do add goto (I, x) to C;
until no more sets of items can be added to C, end;

Ex: LR(0) items for the grammar
E
I →E
E → E + T/T
T → T * F/F
F → T * F/F
F → (E)/id
is given below:
I0 :- E
1 → .E
E → .E + T
E → .T
T → .T * F
T → .F
F → .(E)
F → .id
I_1 :- goto (I_0, E)
E^1 → E.
E → E. + T
I_2 :- goto (I_0, T)
E → T.
T → T. * F
I_3 :- goto (I_0, F)
T → F.
I_4 :- goto (I_0, ( )
F → (.E)
E → .E + T
E → .T
E → .T * F
T → .F
F → .(E)
F → .id
I_5 :- goto (I_0, id)
F → id.
I_6 :- goto (I_1, +)
E → E+ .T

T → .T * F
T → .F
F → .(E)
F → .id
I_7 :- goto (I_2, *)
T → T* .F
F →.(E)
F → .id
I_8 :- goto (I_4, E)
F →(E.)
E → E.+ T
I_9 :- goto (I_6, T)
E → E+ T.
T → T.* F
I_{10} :- goto (I_7, F)
T → T* F.
I_{11} :- goto (I_8,))
F →(E).

E → E+ .T
T → .T * F
T → .F
F → .(E)
F → .id
I7 :- goto (I2, *)
T → T* .F
F →.(E)
F → .id
I8 :- goto (I4, E)
F →(E.)
E → E.+ T
I_9 :- goto (I_6, T)
E → E+ T.
T → T.* F
I_{10} :- goto (I_7, F)
T → T* F.
I_{11} :- goto (I_8,))
F →(E).

For viable prefixes construct the DFA as follow



SLR Parsing Table Construction

1. Construct the canonical collection of sets of LR(0) items for G1
.
2. Create the parsing action table as follows:
(a) If a is a terminal and [A → α.aβ] is in Ii, goto (I_i, a) = I_j then action (i, a) to shift j. Here 'a' must be a terminal.
(b) If [A → α.] is in Ii, then set action [i, a] to "reduce A → α" for all a in FOLLOW(A);
(c)If [S^1 → S.] is in Ii then set action [i, $] to "accept".
3. Create the parsing goto table for all nonterminals A, if goto (I_i, A) = I_j then goto [i,A] = j.
4. All entries not defined by steps 2 and 3 are made errors.
5. Initial state of the parser contains S^1 → S.

The parsing table constructed using the above algorithm is known as SLR(1) table for G.
Note: Every SLR (1) grammar is unambiguous, but every unambiguous grammar is not a SLR grammar.
Eg 1. Construct SLR parsing table for the following grammar:

1. S → L = R 2. S → R
3. L → * R 4. L → id
5. R → L

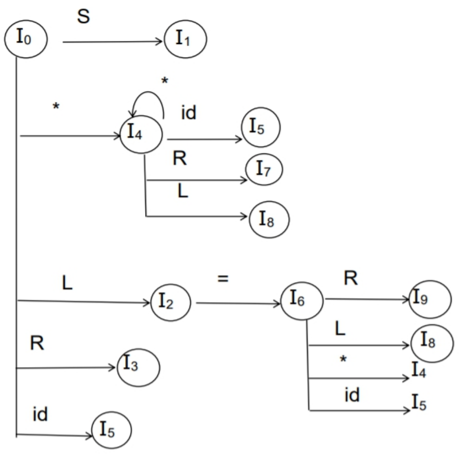
Sol. For the construction of SLR parsing table, add S1 → S production.

S^1 → S
S → L = R
S → R
L → *R
L → id
R → L
LR(0) items will be
I_0 :- S
1 → .S
S → .L = R

Unit-3

S → .R
L → .*R
L → .id
R → .L
I_1 :- goto (I_0, S)
S^1 → S.
I_2 :- goto (I_0, L)
S → L. = R
R → L.
I_3 :- goto (I_0, R)
S → R.
I_4 :- goto (I_0, *)
L → *.R
R → .L
L → .*R
L → .id
I_5 : goto(I_0, id)
L → id.
I6 : goto(I_2, =)
S → L = .R
R → .L
L → .*R
L → .id
I_7 : goto(I_4, R)
L → *R.
I_8 : goto(I_4, L)
R → L.
I9 : goto(I_6, R)
S → L = R.

The DFA of LR(0) items will be



States	action				goto		
	=	*	id	$	S	L	R
0		S4	S5		1	2	3
1				acc			
2	S_6, r_5			r5			
3							
4		S4	S5			8	7
5							
6		S4	S5			8	9
7							
8							
9							

FOLLOW (S) = {$}
FOLLOW (L) = {=}
FOLLOW (R) = {$, =}
For action [2, =] = S6 and r5
Here we are getting shift – reduce conflict, so it is not SLR (1).

Construction of CLR(1) Parsing Table

Construction of the Sets of LR(1) Items
Function closure (**I**):
begin
repeat
for each item [A → α.Bγ, a] in **I**,
each production B → .γ in G^1**,**
and each terminal b in FIRST (βa)
such that [B → .γ,b] is not in **I** do
add [B → .γ,b] to I;
end;
until no more items can be added to **I**;

Eg 1. Construct CLR parsing table for the following grammar:
$S^1 \rightarrow S$
S → CC
C → cC/d

Sol. The initial set of items are
I_0 :- $S^1 \rightarrow .S,\$$
S → .CC, $
A → α.Bβ, a
here A= S, α = ϵ, B = C, β= C and a = $
first (βa) is first (C$) = first (C) = {c, d}
so, add items [C→ .cC, c] [C →.cC, d]
our first set I0:- $S^1 \rightarrow .S,\$$
S → .CC,$
C → .cC, c/d
C → .d, c/d

I_1 :- goto (I_0, X) if X = S

$S^1 \rightarrow S., \$$
I_2 :- goto (I_0, C)
S → C.C,$
C → .cC, $
C → .d, $
I_3 :- goto (I_0, c)
C → c.C, c/d
C → .cC, c/d
C → .d, c/d
I_4 :- goto (I_0, d)
C → d., c/d
I_5 :- goto (I_2, C)
S → CC.,$
I_6 :- goto (I_2, c)
C → c.C, $
C → .cC, $
C → .d, $
I_7 :- goto (I_2, d)
C → d., $
I_8 :- goto (I_3, C)
C → cC., c/d
I_9 :- goto (I_6, C)

C → cC., $

CLR table is:

States	Action			Goto	
	c	d	$	S	C
I_0	S_3	S_4		1	2
I_1			acc		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	r_3	r_3			
I_5			r_1		
I_6	S_6	S_7			9
I_7			r_3		
I_8	r_2	r_2			
I_9			r_2		

Consider the string derivation 'dcd':
S => CC=> CcC=> Ccd=>dcd

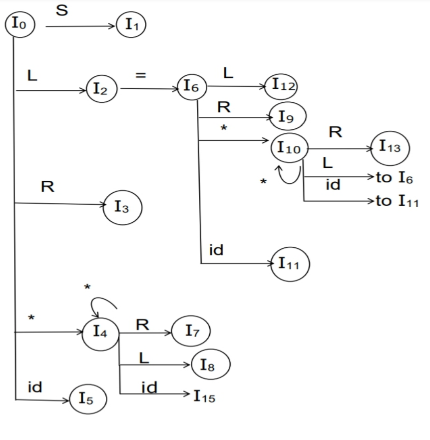
Stack	Input	Action
0	dcd$	shift 4
0d4	cd$	reduce 3 i.e. C → d
0C2	cd$	shift 6
0C2c6	d$	shift 7
0C2c6d7	$	reduce C → d
0C2c6C9	$	reduce C → cC
0C2C5	$	reduce S → CC
0S1	$	

Eg 2. Construct CLR parsing table for the grammar:
1. S → L = R
2. S → R
3. L → *R
4. L → id
5. R → L

Sol. The canonical set of items are
I0 :- S
1 → .S,$
S → .L = R, $
S → .R, $
L → .* R, = / $ [first (= R$) = { =}]
L → .id, = / $
R → .L, $
Note: L → .* R, $
L → .id, $

get added because of $R \rightarrow .L, \$$

I_1 :- goto (I_0, S)
$S^1 \rightarrow S., \$$
I_2 :- goto (I_0, L)
$S \rightarrow L. = R, \$$
$R \rightarrow L.,\$$
I_3 :- goto (I_0, R)
$S \rightarrow R., \$$
I_4 :- goto (I_0, *)
$L \rightarrow *. R, = / \$$
$R \rightarrow .L, = / \$$
$L \rightarrow .* R, = / \$$
$L \rightarrow .id, = / \$$
I_5 :- goto (I_0, id)
$L \rightarrow id. , = / \$$
I_6 :- goto (I_2, =)
$S \rightarrow L = .R, \$$
$R\rightarrow .L, \$$
$L\rightarrow .*R, \$$
$L\rightarrow .id, \$$
I_7 :- goto (I_4, R)
$L \rightarrow *R., = / \$$
I_8 :- goto (I_4, L)
$R \rightarrow L.,= / \$$
I_9 :- goto (I_6, R)
$S \rightarrow L = R., \$$
I_{10} :- goto (I_6, *)
$L \rightarrow *.R, \$$
$R \rightarrow .L, \$$
$L \rightarrow .*R, \$$
$L\rightarrow .id, \$$
I_{11} :- goto (I_6, id)
$L \rightarrow id. , \$$
I_{12} :– goto(I_6, L)
$R \rightarrow L., \$$
I13 :- goto (I10, R)
$L \rightarrow *R., \$$



we have to construct CLR parsing table based on the above diagram.
In this, we are going to have 13 states

The shift –reduce conflict in the SLR parser is reduced here.

States	id	*	=	$	S	L	R
0	S_5	S_4			1	2	3
1				acc			
2			S_6	r_5			
3				r_2			
4	S_5	S_4				8	7
5			r_4	r_4			
6	S_{11}	S_{12}		r_5		12	9
7			r_3	r_3			
8			r_5	r_5			
9				r_1			
10	S_{11}	S_{10}				6	13
11				r_4			13
12				r_5			
13				r_3			

Stack	**Input**	**Action**
0	id = id$	Shift 5
0id5	= id$	reduce 4, L→id
0L2	= id$	Shift 6
0L2 = 6	id$	Shift 11
0L2 = 6id11	$	reduce 4, L→id
0L2 = 6L12	$	reduce 5, L = R
0L2= 6R9	$	reduce 1, S→L=R
0S1	$	Accept

Every SLR (1) grammar is LR(1) grammar.
CLR (1) will have “more number of states” than SLR Parser.

Eg 3. Construct CLR parsing table for the grammar:
S → S + F/F
F → F * P /P
P → x

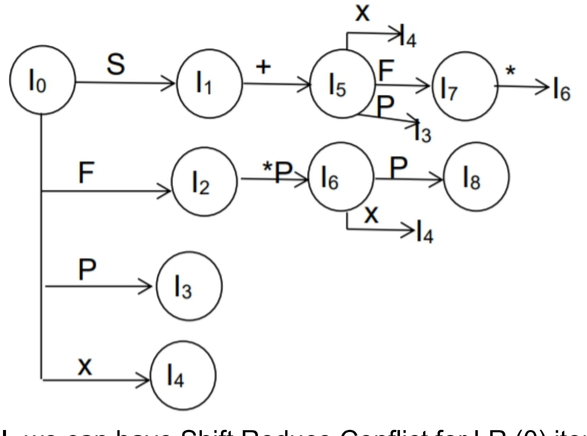
Sol. The canonical set of items are
IO:S
I → .S, $
S →.S + F, $ S → .S + F, +
S → .F, $ S → .F, +
F → .F * P, $/+
F → .P, $/+
F → .F * P, *
F → .P, *
P → .x, *, /$/+

Here we can combine lookahead symbol for F as core set or LR (0) items are same.

F → .F* P, $/*/+
F → .P, $/*/+
Similarly, we can do for S
S → .S + F, $/+
S → .F, $/+
I0: S
I → S, $
S → .S + F, $/+
S → .F, $/+
F → .F * P, $/*/+
F → .P, $/*/+

$P \rightarrow .x, \$/*/, +$

I_1: goto(I_0, S)
$S^I \rightarrow S., \$$
$S \rightarrow S. + F, \$/+$
I_2: goto(I_0, F)
$S \rightarrow F., \$/+$
$F \rightarrow F.*P, \$/+/*$
I_3: goto(I_0, P)
$F \rightarrow P., \$/*/+$
I_4: goto(I_0, x)
$P \rightarrow x., \$/+/*$
I_5: goto(I_1, +)
$S \rightarrow S + .F, \$/+$
$F \rightarrow .F * P, \$/+/*$
$F \rightarrow .P, \$/+/*$
$P \rightarrow .x, \$/+/*$
I_6: goto(I_2, *)
$F \rightarrow F * .P, \$/+/*$
$P \rightarrow .x , \$/+/*$
I_7: goto(I_5, F)
$S \rightarrow S + F., \$/+$
$F \rightarrow F.*P, \$/+/*$
I_8: goto(I_6, P)
$F \rightarrow F * P., \$/+/*$



In I_1 and I_2 we can have Shift Reduce Conflict for LR (0) items, but for LR (1) items no Shift–Reduce conflict will be there.

States	*	+	x	$	S	F	P
0			S_4		1	2	3
1		S_5		ACC			
2	S_6	r_2		r_2			
3	r_4	r_4		r_4			
4	r_5	r_5		r_5			
5			S_4				
6			S_4				
7	S_4	r_1		r_1			
8	r_3	r_3		r_3			

Unit-3

Construction of LALR(1) Parsing Table

LALR Parsing Table

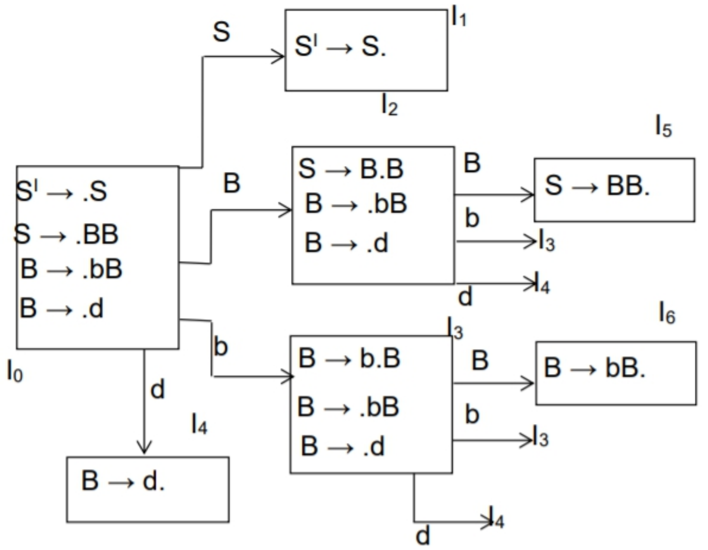
• The tables obtained by it are considerably smaller than the canonical LR table.
• LALR stands for Lookahead LR.
• The number of states in SLR and LALR parsing tables for a grammar G are equal.
• But LALR parsers recognize more grammars than SLR.
• YACC creates a LALR parser for the given grammar.
• YACC stands for "Yet Another Compiler Compiler".
• An easy, but space-consuming LALR table construction is explained below:
1. Construct C = {I_0, I_1, ---I_n}, the collection of sets of LR(1) items.
2. Find all sets having the common core, replace these sets by their union
3. Let C1 = {J_O, J_1 --- J_m} be the resulting sets of LR(1) items. If there is a parsing action conflict then the grammar is not a LALR(1).
4. Let k be the union of all sets of items having the same core. Then goto (J,X) = k
• If there are no parsing action conflicts then the grammar is said to LALR (1) grammar.
• The collection of items constructed is called LALR(1) collection.

Ex: Consider the grammar:
$S^1 \rightarrow S$
$S \rightarrow aAd$
$S \rightarrow bBd$
$S \rightarrow aBe$
$S \rightarrow bAe$
$A \rightarrow c$
$B \rightarrow c$
Which generates strings acd, bcd, ace and bce
LR(1) items are
I_0 :- $S^1 \rightarrow .S,\$$
$S \rightarrow .aAd, \$$
$S \rightarrow .bBd, \$$
$S \rightarrow .aBe, \$$
$S \rightarrow .bAe, \$$
I_1 :- goto (I_0, S)
$S^1 \rightarrow S., \$$
I_2 :- goto (I_0, a)
$S \rightarrow a.Ad, c$
$S \rightarrow a.Be, c$
$A \rightarrow .c,d$
$B \rightarrow .c,e$
I_3 :- goto (I_0, b)
$S \rightarrow b.Bd, c$
$S \rightarrow b.Ae, c$
$A \rightarrow .c,e$
$B \rightarrow .c, d$
I_4 :- goto (I_2, A)
$S \rightarrow aA.d, c$
I_5 :- goto (I_2, B)
$S \rightarrow aB.e,c$
I_6 :- goto (I_2, c)
$A \rightarrow c., d$
$B \rightarrow c., e$
I_7 :- goto (I_3, c)
$A \rightarrow c., e$
$B \rightarrow c., d$
I_8 :- goto (I_4, d)
$S \rightarrow aAd., c$
I_9 :- goto (I_5, e)
$S \rightarrow aBe., c$
If we union I_6 and I_7
$A \rightarrow c., d/e$

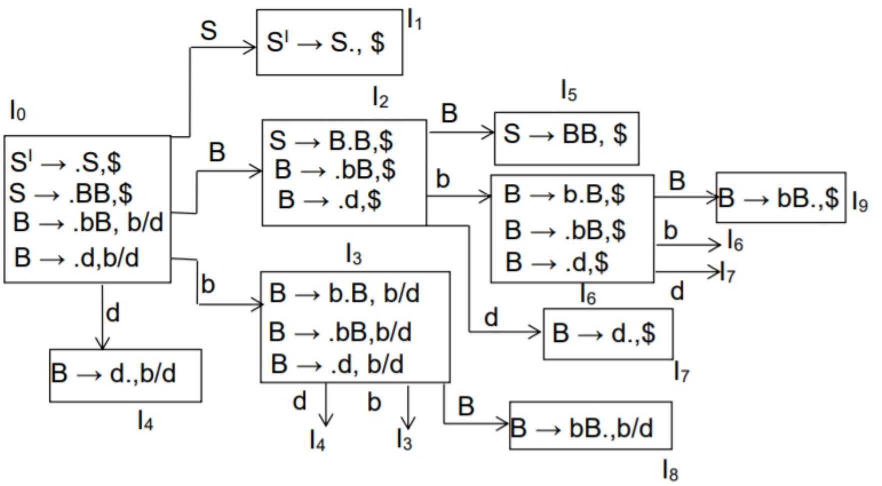
B → c., d/e
It generates reduce/reduce conflict.
Eg 1. Construct LALR parsing table for the following grammar:
$S^1 \rightarrow S$
S → CC
C → cC/d
Sol. We already got LR(1) items and CLR parsing table for this grammar.
After merging I_3 and I_6 are replaced by I_{36}.
I_{36}: C → c.C, c/d/$
C → .cC, c/d/$
C → .d, c/d/$
I_{47}: By merging I_4 and I_7
C → d., c/d/$
I_{89}: I_8 and I_9 are replaced by I_{89}
C → cC., c/d/$
The LALR parsing table for this grammar is given below:

State	Action			goto	
	c	d	$	S	C
0	S_{36}	S_{47}		1	2
1			acc		
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		

Eg 2. Check whether given grammar is LL(1), LR(0), LR(1), SLR(1), LALR(1)
S → BB
B → bB/d
Sol. LL(1)
As b ∩ d = Φ the given grammar is LL(1)
LR(0) items
$S^I \rightarrow .S$
S → .BB
B → .bB
B → .d



No Shift Reduce or reduce – reduce conflict is present.
Therefore the given grammar is LR(0) and SLR(1).
LR(1) items
S
1 → .S, $
S → .BB , $
B → .bB , b/d
B → .d , b/d



As there is no S–R or R–R conflict the given grammar is CLR (1).
The states (I3, I6), (I4, I7) and (I8, I9) can be merged to LALR (1) sets.

Note:
1. The merging of states with common cores can never produce a shift/reduce conflict, because shift action depends only on the core, not on the lookahead.
2. SLR and LALR tables for a grammar always have the same number of states (several hundreds) whereas CLR have thousands of states for the same grammar.
3. The merging of state with common cores may produce a reduce/reduce conflict.

Example:
Consider the following sets in CLR(1)

$A \rightarrow \alpha\,.\,,\ a$
$B \rightarrow \beta\,.\,,\ b$

$A \rightarrow \alpha\,.\,,\ b$
$B \rightarrow \beta\,.\,,\ a$

After merging

$A \rightarrow \alpha\,.\,,\ a/b$
$A \rightarrow \beta\,.\,,\ a/b$

Therefore reduce / reduce conflict will be here in LALR parser.