$E'	+id$	$E' \to +TE'$
$E'T+	+id$	
$E'T	id$	
$E'T'F	id$	$T \to FT'$
$E'T'id	id$	
$E'T'	$	

<u>Bottom up parsing</u> → Considering the string and deriving the start symbol.

Handle

Handle Pruning

Reduction.

$S \to aABe$
$A \to Abc|b$
$B \to d$

<u>Left most reduction</u>

a<u>b</u>bcde	$A \to b$
<u>aAbc</u>de	$A \to Abc$
aAde	
aABe	$S \to aAbe$
S	

Right most derivation (Top down parsing)

$S \to aABe$ abbcde

$\to aAde$

$\to aAbcde$

$\to abbcde$

- The reduction traceout the right most derivation in reverse order.

<u>Handle</u>

It is a substring that matches the right side of production and whose reduction to the non-terminal on the left-side of the production.

<u>Handle pruning:</u>

- simply it is a rightmost derivation in reverse can be obtained by handle pruning.

$E \to E+E/E*E/(E)/id$

I/p id1 + id2 * id3.

$E \to \underline{E+E}$

$\to E+\underline{E*E}$

$\to E+E*\underline{id3}$

$\to E+\underline{id2}*id3$

$\to \underline{id1}+id2*id3.$

underlined are called Handles.

Shift reduce parsing:

Shift → moving the input symbol to the top of the stack.

– A general type of shift reduce parser. bottomup parser is

Stack implementation of shift reduce parser

$E \rightarrow E+E \mid E*E \mid (E) \mid id$

Stack	i/p	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	$*id_3$ $	Reduce by $E \rightarrow E+E$
$E	$*id_3$ $	shift
$E*	id_3 $	shift
$E*id_3	$	Reduce by $E \rightarrow id$
$E*E	$	Reduce by $E \rightarrow E*E$
$E	$	Accept → successful completion.

SR Conflict

– A parser cannot decide to do shift or reduce operation.

RR Conflict

– A parser cannot decide which reduction to make

RR conflict

$M \rightarrow R+R \mid R+c \mid R$

$R \rightarrow c$

i/p c+c

stack	i/p	Action
$	c+c	shift
$c	+c	Reduce by $R \rightarrow c$
$R	+c	Reduce by $M \rightarrow R$
$M	+c	shift
$M+	c	shift
$M+c	$	Reduce by $R \rightarrow c$
$M+R	$	

stack	i/p	Action
$	c+c	shift
$c	+c	Reduce by $R \rightarrow c$
$R	+c	shift
$R+	c	shift
$R+c	$	Reduce by $M \rightarrow R+c$
$M	$	Accept

(or)

$R+c	$	Reduce by $R \rightarrow c$
$R+R	$	Reduce by $M \rightarrow R+R$
$M	$	Accept

– In order to avoid RR conflict, operator precedence parser, An efficient way of contructing SR parser is called operator precedence parsing.

Operator Precedence parser

- An efficient way of constructing SR parser is called operator precedence parser.

Properties:

- No production on right side is epsilon.
- No two adjacent non-terminals on right side.

Operator precedence relations:

- $a \cdot> b$ → a is higher precedence than b.

  $a <\cdot b$ → a is having less precedence than b.

  $a \doteq b$ → Both have equal precedence

Rules

id, a, b, c --- etc is having highest priority/ than any other precedence

\$ is having lowest precedence

$(\doteq)$

have same precedence.

), +, -, *, / → left associative.

(, ↑ → right associative

left side precedence

$+ \cdot> +$

$* \cdot> *$

$) \cdot>)$

$(<\cdot ($

$\uparrow <\cdot \uparrow$

$\$ \neq \$$, $id \neq id$ } Accept need to be written

Operator precedence algorithm.

1) set i/p to point to the first symbol of w\$

2) Repeat forever.

3) If \$ is on top and i/p points to \$ then

4) Return else Begin

5) Let a be the top and let b be the symbol, pointed to i/p;

6) If $a <\cdot b$ or $a \doteq b$ then begin

7) Push b on to stack

8) Advance i/p to the next i/p symbol end;

9) Else if $a \cdot> b$ then /* reduce */

10) Repeat

11) Pop the stack

12) Until the top is related by $<\cdot$ to the terminal most recently popped

13) Else error()

14) end.

	+	*	)	(	id	\$
+	·>	<·	·>	<·	<·	·>
*	·>	·>	·>	<·	<·	·>
)	<·	<·	·>		<·	·>
(	·>	·>	≐	<·	<·	·>
id	·>	·>	·>	·>	Accept	·>
\$	<·	<·	<·	<·	<·	Accept

empty.

$\$ = \$$

$id \neq id$

$) \neq ($

$(<\cdot ($

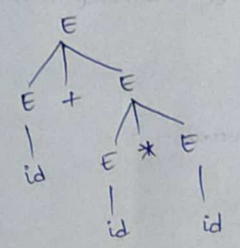
$E \rightarrow EAE / id$ Convert → $E \rightarrow E+E / E*E / id$.

$A \rightarrow + / *$

	+	*	id	$
+	·>	<·	<·	·>
*	·>	·>	<·	·>
id	·>	·>	~~Accept~~	·>
$	<·	<·	<·	Accept

i/p → id+id*id

Stack	Relation	i/p	Action
$	<·	id+id*id$	Shift
$id	·>	+id*id$	Reduce
$E (ignore such this)	<·	+id*id$	Shift
$E+	<·	id*id$	Shift
$E+id	·>	*id$	Reduce
$E+E	<·	*id$	Shift
$E+E*	<·	id$	Shift
$E+E*id	·>	$	Reduce
$E+E*E	·>	$	Reduce. $E \rightarrow E*E$ a bottom up
$E+E	·>	$	Reduce $E \rightarrow E+E$
$E		$	Accept

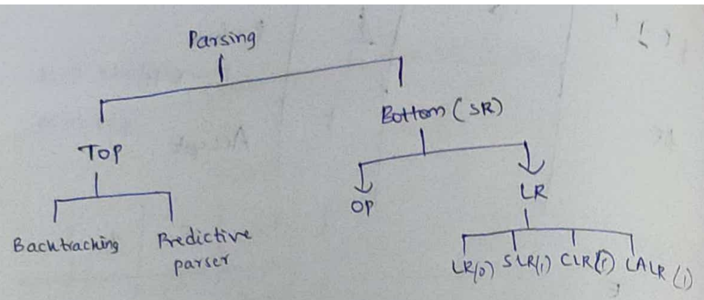


Advantage

- Easy to implement
- The grammar is not preferred during implementation anymore.

Disadvantages

- Only small class of grammars can be parsed
- Hard to implement tokens like minus(-), which has two different precedences.



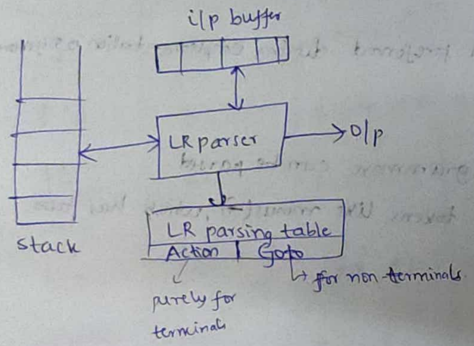
LR(k) → look ahead symbol

LR → left to right scan

R → right most derivation in reverse

LR Parsers:

LR parser structure



LR parsers are basically four types LR(0), SLR(1), LALR(1), CLR(1).

In these 4 types LR(0) is the least powerful parser compared to other LR parsers.

CLR(1) is the most powerful parser compared to all other LR parsers.

SLR → Simple LR parser

LALR → Look ahead LR parser

CLR → Cannonical LR parser

SLR(1)

- It works on smallest class of grammars.
- It have only a few number of states.
- It is simple and fast construction.

LALR

- It works on intermediate size of grammars.
- The number of states are same as SLR.

CLR

- It works on complete set of LR(1) grammar.
- It have large number of states and it is slow construction.

All LR parsers are same but it have the different parsing tables.

NOTE :- 1) To construct LR(0) and SLR(1) parsing tables we use cannonical collection of LR(0) items.

2) To construct LALR(1) and CLR(1) parsing tables we use cannonical collection of LR(1) items.

LR(0)

1. Add augumented grammar (production)
   (Convert given grammar into augumented grammar).
2. Create cannonical collection of LR(0) items.
3. Find closure and goto.
4. Draw DFD (Data flow diagram).
5. Construct LR(0) parsing table.
6. Parse the given string.

① Ex:

Grammar		Augumented grammar
$S \to AB$	$\to$	$S' \to S$
$A \to a$		$S \to AB$
$B \to b$		$A \to a$
		$B \to b$

for start symbol it should be derived from any other. So $S' \to S$ is added to ~~no~~ grammar

LR(0) items

These are the productions of G with a . at some portion of the right side

$S' \to .S$

$S \to .AB$ this, dot is moved to the last

$A \to .a$

$B \to .b$

Closure Operation

- If I is a set of items for a grammar G then closure of I is the set of items construction from I
  (i) Initially every item in I is added to closure (I)
  (ii) If A derives $A \to \alpha . AB$

(ii) If $A \to \alpha . B\beta$ is in closure(I) and $B \to \gamma$ is a production then add the item $B \to .\gamma$ to I if it is not already there. Apply this until no more new items can be added to closure(I)

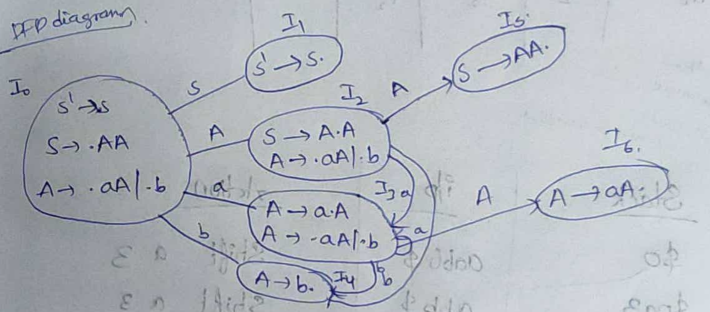
Goto operation

goto(I, X) is defined to be the closure of set of all items ~~such that~~ $A \to \alpha X . \beta$ such that $A \to \alpha . X\beta$ is in I

Example: $S \to AA$
$A \to aA/b$

$S' \to S$ I_0 :- $S' \to .S$
$S \to AA$ $S \to .AA$
$A \to aA/b$ $A \to .aA/.b$

DFD diagram.



3rd step is written fast

goto(I_0, S) : $S' \to S.$

goto(I_0, A) : $S \to A.A$
$S \to .aA / .b.$

goto(I_0, a): $A \to a.A$
$A \to .aA/.b.$

goto(I_0, b) : $A \to b.$

goto(I_2, A) : $S \to AA.$

goto(I_2, a):

goto(I_3, A): $A \to aA.$

goto(I_2, b):

goto (I_3, a) : $A \to a \cdot A$, $A \to \cdot aA / \cdot b$

goto (I_3, b) : $A \to b \cdot$

Parsing table (Shift operation)

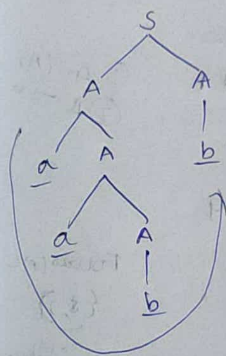
States	a	b	$	S	A
	Action			goto	
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

① $S \to AA$
② $A \to aA$
③ $A \to b$

as there no more so final it need to be reduced

Stack	i/p	Action
$0	aabb$	Shift a 3
$0a3	abb$	Shift a 3
$0a3a3	bb$	Shift b 4
$0a3a3b4	b$	Reduce $A \to b$
$0a3a3A $0a3a3A6	b$	Reduce $A \to aA$
$0a3A → $0a3A6	b$	Reduce $A \to aA$
$0A2	b$	Shift b 4
$0A2b4	$	Reduce $A \to b$
$0A2A5	$	Reduce $S \to AA$
$0S1	$	Accept.

Parse tree



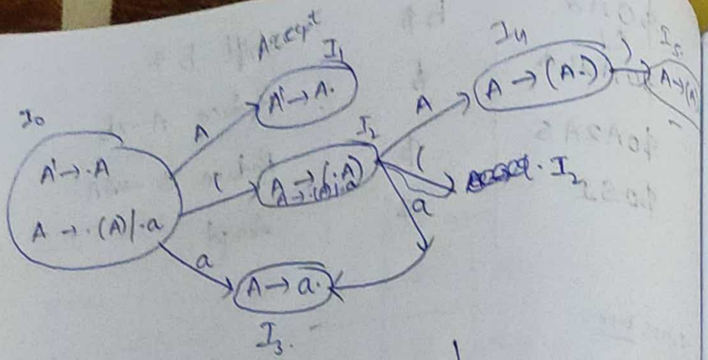
aabb

SLR(1)

- In SLR(1) we place the reduce move only in the follow of left hand side not to entire row.

$A \to (A) | a \longrightarrow A' \to A$
$A \to (A) | a$

I_0 :- $A' \to \cdot A$
$A \to \cdot (A) | \cdot a$



State	(	)	a	$	A
I_0	S_2		S_3		1
I_1				Accept	
I_2	S_2		S_3		4
I_3		r_2		r_2	
I_4		S_5			
I_5		r_1		r_1	

① $A \to (A)$
② $A \to a$

Follow(A) $\{\$,)\}$

So reduce is placed in $ and)

(a)

Stack	i/p	Action
$0	(a)$	Shift (2
$0(2	a)$	Shift a 3
$0(2a3	)$	Reduce $A \to a$
$0(2A		
$0(2A4	)$	Shift) 5
$0(2A4)5	$	Reduce $A \to (A)$
$0A1	$	Accept