URBAN EDGE 28/1/20

MODULE-3

Bottom-Up Parsing

- A bottom-up parser creates the parse tree of the given i/p starting from leaves towards the root.
- It tries to find the right-most derivat^n of given i/p in reverse order.
- It is also known as shift-reduce parsing because its 2 main act^{n}s are shift & reduce
- → At each shift, current i/p symbol is pushed onto stack
- → " " reduct^n, the symbols at the top of the stack will be replaced by the non-terminal at the left-side of that product^n
- → There are also 2 more act^{n}s: accept & error.

Shift-Reduce Parsing

- A shift-reduce parser tries to reduce the given i/p string into the starting symbol

Handle

- A handle of a right sentential form $\gamma (\equiv \alpha\beta w)$ is a product^n rule $A \to \beta$ & a posit^n of γ where the string β may be found & replaced by A to produce the previous right-sentential form in a rightmost derivat^n of γ:

$$S \Rightarrow \alpha A w \Rightarrow \alpha \beta w$$ (rm)*

- If the grammar is unambiguous, then every right-sentential form of the grammar has exactly 1 handle.

URBAN EDGE 28/1/20

eg: $E \Rightarrow E+E$ — Handles

$\Rightarrow E+E*E$ — Bottom-up — id

$\Rightarrow E+E*id$ — parsing — id

$\Rightarrow E+id*id$ — handles — id

$\Rightarrow id+id*id$ — E*E

$w = id+id*id$ — E+E

- Another name of bottom-up parsing is handle pruning.
- 2 types of shift-reduce parsing:

(i) Operator Precedence Parser (Special case of bottom-up parser)

(ii) LR Parser → SLR → Simple LR parser
→ CLR → Canonical " " / LR parser
→ LALR → Look Ahead " "

Operator Precedence Parser

$E \to E+E \mid E*E \mid id$

Here parsing table will be a square matrix of terminal symbols.

	id	+	*	$
id		$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	$\doteq$ accept

$ id + id * id $

$ $<\cdot$ id $\cdot>$ + $<\cdot$ id $\cdot>$ * $<\cdot$ id $\cdot>$ $

Symbol btw $<\cdot$ & $\cdot>$ is considered as Handle

Remove all handles

$ $<\cdot$ + $<\cdot$ * $\cdot>$ $ → * is handle. Removing $<\cdot$ * $\cdot>$

$\$ \lessdot + \gtrdot \$$ ⇒ + is a handle. Remove it.

$\$ \doteq \$$

30/1/20

~~$id + id * \$$~~

~~$\$ \lessdot id \gtrdot + \lessdot id \gtrdot * \$$~~

~~$\$ + * \$$~~

~~$\$ \lessdot + \lessdot * \gtrdot \$$~~

~~$\$ \lessdot + \$$~~

~~$\$ \lessdot + \gtrdot \$ \Rightarrow \$ \doteq \$$~~

Operator Precedence Parsing Algorithm

Initialize: Set ip to point to 1st symbol of w$

Repeat: Let X be top stack symbol, & a the symbol pointed to by ip.

if $ is on the top of stack & ip points to $ then return

else

Let a be the top terminal on stack, & b the symbol pointed to by ip

if $a \lessdot b$ or $a \doteq b$ then

push b onto stack

advance ip to the next i/p symbol

else if $a \gtrdot b$ then

repeat

pop the stack

until the top stack terminal is related by $\lessdot$ to the terminal most recently popped

else error()

end

Stack Implementation of Shift-Reduce Parsers

- 4 possible act's of shift-reduce parser:

(i) Shift:- The next i/p symbol is shifted onto the top of the stack

(iii) Reduce

(iv) Accept

(v) Error

LR Parsing Algorithm Configuratn

- A config of a LR parsing is:

$(S_0 X_1 S_1 \cdots X_m S_m, a_i a_{i+1} \cdots a_n \$)$

Stack ↑ Rest of i/p

- S_m & a_i decide the parser actn by consulting the parsing actn table. (Initial Stack contains just S_0)
- A config of a LR parsing represents the right sentential form:

$X_1 \cdots X_m a_i a_{i+1} \cdots a_n \$$

Actn's of a LR Parser

- shift s → shifts the next i/p symbol & state s onto stack

$(S_0 X_1 S_1 \cdots X_m S_m, a_i a_{i+1} \cdots a_n \$) \Rightarrow (S_0 X_1 S_1 \cdots X_m S_m a_i s, a_{i+1} \cdots a_n \$)$

- reduce $A \to \beta$ (or rn where n → productn no.)

→ pop $2|\beta|$ items from stack;

→ then push A & s where $s = goto[S_{m-r}, A]$

$(S_0 X_1 S_1 \cdots X_m S_m, a_i a_{i+1} \cdots a_n \$) \Rightarrow (S_0 X_1 S_1 \cdots X_{m-r} S_{m-r} A s, a_i \cdots a_n \$)$

→ O/p is the reducing productn reduce $A \to \beta$

- Accept → Parsing successfully completed
- Error → Parser detected an error (an empty entry in the actn table)

(SLR) Parsing Tables for Expression Grammar

eg. 1) E → E+T
2) E → T
3) T → T*F
4) T → F
5) F → (E)
6) F → id

id + id * id $

(0, id + id * id $)
↑

Corresponding to 0 & id ⇒ s5

S_5 ⇒ shift opeat^n & change state to 5

(0 id 5, + id * id $)

Corresponding to 5 & + ⇒ r6

r6 ⇒ reduce opeat^n & refer product^n 6 ⇒ F → id

⇒ (0 F 3, + id * id $)

⇒ (0 T 2, + id * id $)

⇒ (0 E 1, + id * id $) ⇒ (0 E 1 + 6, id * id $) ⇒ (0 E 1 + 6 id 5, * id $)

⇒ (0 E 1 + 6 F 3, * id $) ⇒ (0 E 1 + 6 T 9, * id $) ⇒ (0 E 1 + 6 T 9 * 7, id $)

⇒ (0 E 1 + 6 T 9 * 7 id 5, $) ⇒ (0 E 1 + 6 T 9 * 7 F 1, $) ⇒ valid i/p

⇒ accept

Act^n Table

state	id	+	*	(	)	$
0	s5			s4		
1		s6				acc
2		r2	s7		r2	r2
3		r4	r4		r4	r4
4	s5			s4		
5		r6	r6		r6	r6
6	s5			s4		
7	s5			s4		
8		s6			s11	
9		r1	s7		r1	r1
10		r3	r3		r3	r3
11		r5	r5		r5	r5

Goto Table

E	T	F
1	2	3
8	2	3
	9	3
		1
		0

Constructing SLR Parsing Tables - LR(0) Item

URBAN EDGE 30/1/20

- LR(0) item of grammar G_1 → product^n of G_1 with a dot at some posit^n on right side

eg: $A \to aBb$: $A \to \cdot aBb$
(4 different possibilities) $A \to a \cdot Bb$
$A \to aB \cdot b$
$A \to aBb \cdot$

- Sets of LR(0) items will be states of act^n & goto table of the SLR parser.
- A collect^n of sets of LR(0) items is the basis for constructing SLR parsers.
- Augmented Grammar:- Augmented grammar G' is G with a new product^n rule $S' \to S$ where S' is the new starting symbol.

The Closure Operat^n

- If I is a set of LR(0) items for a grammar G, then closure(I) is the set of LR(0) items constructed from I by 2 rules:

(i) Initially, every LR(0) item in I is added to closure(I).

(ii) If $A \to \alpha \cdot B\beta$ is in closure(I) & $B \to \gamma$ is a product^n rule of G; then $B \to \cdot\gamma$ will be in the closure(I). We will apply this rule until no more new LR(0) items can be added to closure(I)

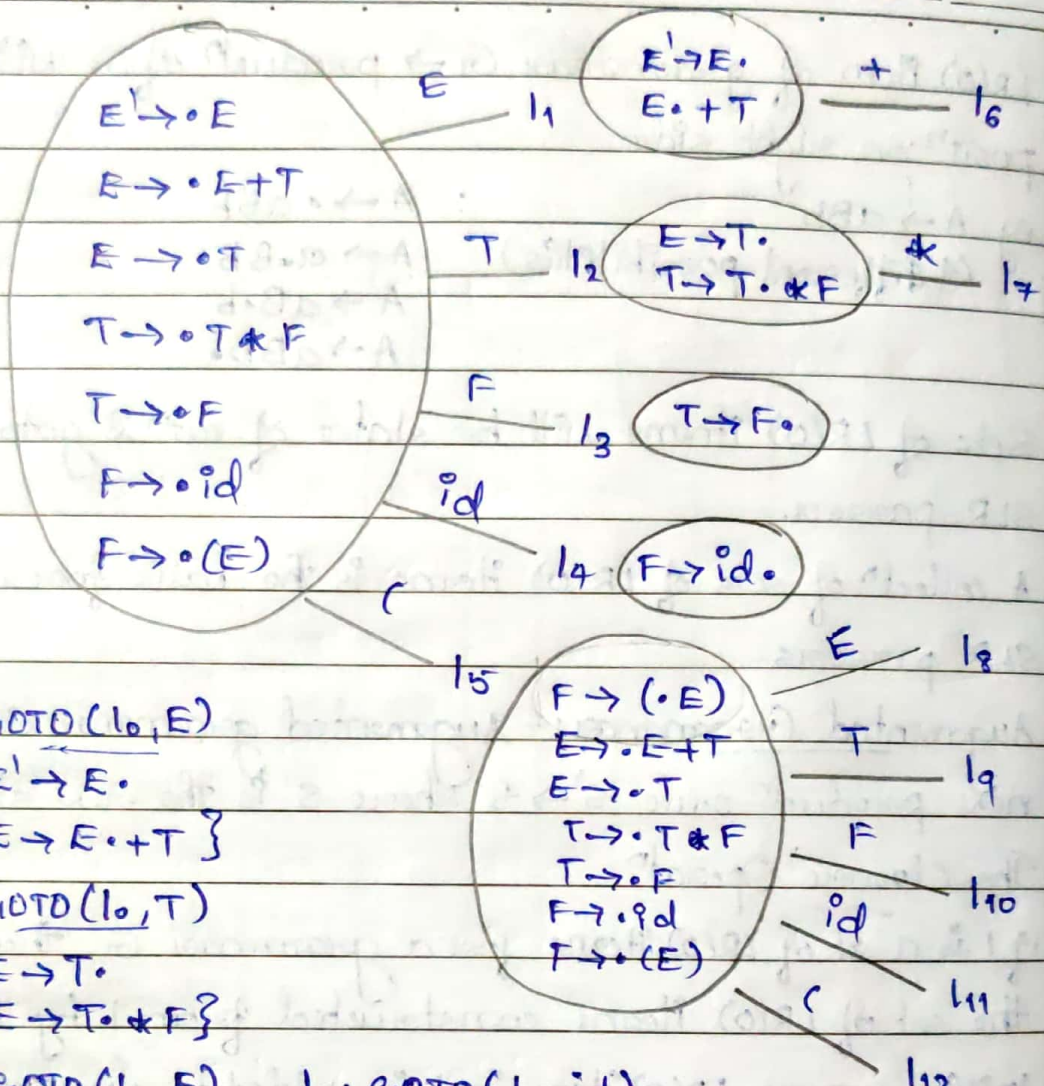
eg: $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to id \mid (E)$

Canonical Set(E^1) = I_0

$E' \to \cdot E$ $E \to \cdot T$ $T \to \cdot F$ ~~$F \to (E)$~~
$E \to \cdot E+T$ $T \to \cdot T*F$ $F \to \cdot id$ $F \to \cdot (E)$

} Entire set is state 0

URBAN EDGE 30/1/20



31/1/20

I_1: GOTO(I_0, E)
{ $E' \to E \cdot$
$E \to E \cdot +T$ }

I_2: GOTO(I_0, T)
{ $E \to T \cdot$
$E \to T \cdot *F$ }

I_3: GOTO(I_0, F)
{ $T \to F \cdot$ }

I_4: GOTO(I_0, id)
{ $F \to id \cdot$ }

I_5: GOTO(I_0, ()
{ $F \to (\cdot E)$
$E \to \cdot E+T$
$E \to \cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot id$
$F \to \cdot (E)$ }

I_6: GOTO(I_1, +)
{ $E \to E+ \cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot id$
$F \to \cdot (E)$ }

~~GOTO~~ GOTO(I_5, T)
{ $E \to T \cdot$
$T \to T \cdot *F$ }
} Already exists $\Rightarrow I_2$
~~So not possible~~ So not new state
$\Rightarrow$ GOTO(I_5, T) $= I_2$

I_7: GOTO(I_2, *)
{ $T \to T* \cdot F$
$F \to \cdot id$
$F \to \cdot (E)$ }

I_8: GOTO(I_5, E)
{ ~~$E' \to E \cdot$~~ $F \to (E \cdot)$
~~$E \to E$~~ $E \to E \cdot +T$ }

URBAN EDGE 31/1/20

GOTO (I_5, F)

$\{T \to F\cdot\} \Rightarrow$ GOTO $(I_5, F) = I_3$

GOTO (I_5, id)

$\{F \to id\cdot\} \Rightarrow$ GOTO $(I_5, id) = I_4$

GOTO $(I_5, ()$

$\{F \to (\cdot E)$

$E \to \cdot E+T$

$E \to \cdot T$

$T \to \cdot T*F$

$T \to \cdot F$

$F \to \cdot id$

$F \to \cdot (E)\}$ $\Rightarrow$ GOTO $(I_5, () = I_5$

GOTO (I_7, F)

$\{T \to T*F\cdot\} \Rightarrow I_{10}$

GOTO $(I_8,))$

$\{F \to (E)\cdot\} \Rightarrow I_{11}$

~~GOTO $(I_{10},$~~

~~GOTO (I_4, E)~~

~~$\{F \to (E\cdot)$~~

~~$E \to E\cdot+T\}$~~

GOTO $(I_8, +)$

~~$\{E \to E+\cdot T$~~ $\Rightarrow I_6$

GOTO (I_6, T)

$\{E \to E+T\cdot$

$T \to T\cdot*F\} \Rightarrow I_9$

GOTO (I_6, F)

$\{T \to F\cdot\} \Rightarrow I_3$

GOTO (I_6, id)

$\{F \to id\cdot\} \Rightarrow I_4$

GOTO $(I_6, ()$

$\Rightarrow I_5$

GOTO (I_7, id)

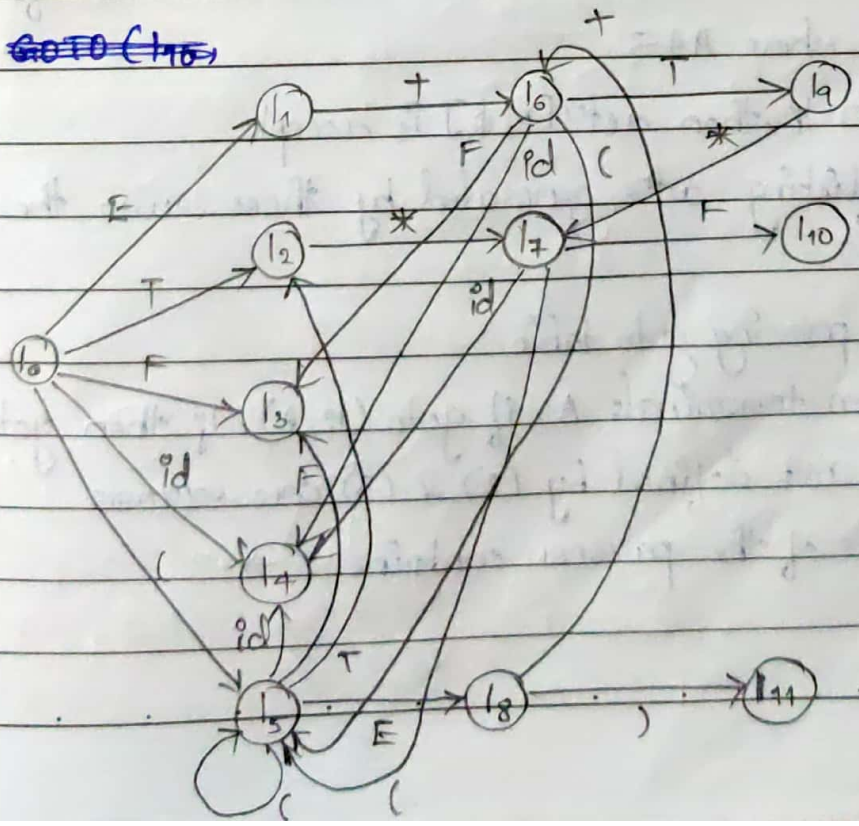
$\Rightarrow I_4$

GOTO $(I_7, ()$

$\Rightarrow I_5$

GOTO $(I_9, *)$

$\Rightarrow I_7$



URBAN EDGE 31/1/20

Construct^{n} of The Canonical LR(0) Collect^{n}

- Algorithm:

C is { closure ({S' → .S}) }

repeat the following until no more set of LR(0) items can be added to C

for each I in C & each grammar symbol X

if goto(I, X) is not empty & not in C

add goto(I, X) to C

- goto funct^{n} is a DFA on the sets in C.

Constructing an SLR Parsing Table

- Construct canonical collect^{n} of sets of LR(0) items for G'.

C ← {$I_0, \ldots I_n$}

- Create the parsing act^{n} table as follows:

→ If a is a terminal, A → α.aβ in I_i & goto(I_i, a) = I_j then act^{n}[i, a] is shift j

→ If A → α. is in I_i, then act^{n} [i, a] is reduce A → α for all a in FOLLOW(A) where A ≠ S'.

→ If S' → S. is in I_i, then act^{n} [i, $] is accept.

→ If any conflicting act's generated by these rules, the grammar is not SLR(1)

- Create the parsing goto table

→ For all non-terminals A, if goto(I_i, A) = I_j then goto[i, A] = j

- All entries not defined by (2) & (3) are errors.
- Initial state of the parser contains S' → .S

URBAN EDGE 31/1/20

LR Parsing Algorithm

```
let a be the 1st symbol of w$;
while(1) { /* repeat forever */
    let s be the state on top of stack;
    if (ACTION[s,a] = shift t) {
        push a then t onto stack;
        advance ip to the next i/p symbol;
    }
    else if (ACTION[s,a] = reduce A→β) {
        pop 2*|β| symbols of the stack;
        let state t now be on top of the stack;
        push A then GOTO[t, A] on top of the stack;
        output the production A → β;
    }
    else if (ACTION[s,a] == accept) break; /* parsing is done */
    else call error-recovery routine;
}
```

① S → L=R | R

L → *R | id

R → L

URBAN EDGE 31/1/20

ans. $S \to L=R$

$S \to R$

$L \to *R$

$L \to id$

$R \to L$

Augmented grammar ⟹

$S' \to S$

(i) $S \to L=R$

(ii) $S \to R$

(iii) $L \to *R$

(iv) $L \to id$

(v) $R \to L$

$I_0 = \{S' \to \cdot S$
$S \to \cdot L=R$
$S \to \cdot R$
$L \to \cdot *R$
$L \to \cdot id$
$R \to \cdot L\}$

$I_1 = \{S' \to S \cdot\}$

$I_2 = \{S \to L \cdot = R$
$R \to L \cdot\}$

$I_3 = \{S \to R \cdot\}$

$I_4 = \{L \to * \cdot R$
$R \to \cdot L$
$L \to \cdot id$
$L \to \cdot *R\}$

$I_5 = \{L \to id \cdot\}$

GOTO(I_2, =)
$\{S \to L= \cdot R \Rightarrow I_6$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot id\}$

GOTO(I_4, R)
$\{L \to *R \cdot\} \Rightarrow I_7$

GOTO(I_4, L)
$\{R \to L \cdot\} \Rightarrow I_8$

GOTO(I_4, id)
$\{L \to id \cdot\} = I_5$

GOTO(I_4, *)
$\{L \to * \cdot R$
$R \to \cdot L$
$L \to \cdot id$
$L \to \cdot *R\} = I_4$

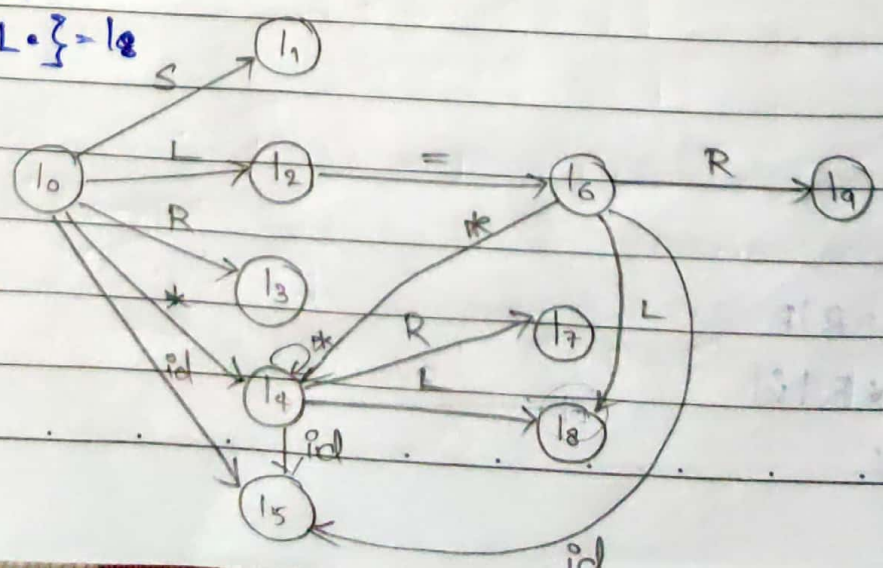
GOTO(I_6, R)
$\{S \to L=R \cdot\} \Rightarrow I_9$

GOTO(I_6, *)
$\{L \to * \cdot R$
$R \to \cdot L$
$L \to \cdot id$
$L \to \cdot *R\} = I_4$

GOTO(I_6, id)
$\{L \to id \cdot\} = I_5$

~~GOTO(~~

GOTO(I_6, L)
$\{R \to L \cdot\} = I_8$



URBAN EDGE 31/1/20

$C \leftarrow \{I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9\}$

FIRST(S) = {id, *} FOLLOW(S) = {$}

FIRST(L) = {*, id} FOLLOW(L) = {$, =}

FIRST(R) = {*, id} FOLLOW(R) = {$, =}

Action table | Goto table

state	id	*	=	$	S	L	R
0	s5	s4			1	2	3
1				acc			
2			r5, s6	r5			
3			r2	r2			
4	s5	s4				8	7
5			r4	r4			
6	s5	s4				8	9
7			r3	r3			
8			r5	r5			
9				r1			

$L \rightarrow \cdot *R \Rightarrow$ shift to state 4 $\Rightarrow s4$

$L \rightarrow \cdot id \Rightarrow$ " " " 5 $\Rightarrow s5$

} state 0

$S' \rightarrow S \cdot \Rightarrow$ accept } state 1

$S \rightarrow L \cdot = R \Rightarrow$ shift to state 6 ~~} state 2~~

$R \rightarrow L \cdot \Rightarrow$ reduce $R \rightarrow L \Rightarrow r5$ in FOLLOW(R)

} state 2 → shift-reduce conflict

This is an ambiguous grammar.

∴ We cannot construct Parsing table

URBAN EDGE 31/1/20

S→R· ⇒ reduce S→R ⇒ r2 in FOLLOW(R) } State 3

L→·id $ ⇒ shift to state 5
L→·*R $ ⇒ " " " 4 } State 4

L→id· ⇒ reduce L→id ⇒ r4 in FOLLOW(L) } State 5

L→·*R ⇒ shift to state 4
L→·id ⇒ shift to state 5 } state 6

L→*R· ⇒ reduce L→*R ⇒ r3 in FOLLOW(L) } State 7

R→L· ⇒ reduce R→L → r5 in FOLLOW(R) } State 8

S→L=R· ⇒ reduce S→L=R ⇒ r1 in FOLLOW(S) } State 9

4/2/20

Canonical LR Parser (CLR / LR)

Canonical LR(1) Items

- In LR(1) items each item is in the form: [A→αβ, a]
- An LR(1) item [A →* α·β, a] is valid for a viable prefix γ if there is a derivation

  S =rm⇒ δAw ⇒ δαβw, where

  γ = δα & a → either 1st symbol of w, w is ε & a is $.

- eg: S'→S

  S→BB

  B→aB

  B→b

  $I_0 = \{ [S'→·S, \$]$
  $[S→·BB, \$]$
  $[B→·aB, a/b]$
  $[B→·b, a/b] \}$

  $I_1 = GOTO(I_0, S)$

  $I_2 = GOTO(I_0, B)$

  $I_3 = GOTO(I_0, a)$

  $I_4 = GOTO(I_0, b)$

  $I_1 = \{ [S'→S·, \$] \}$

  $I_2 = \{ [S→B·B, \$]$
  $[B→·aB, \$]$
  $[B→·b, \$] \}$

  $I_3 = \{ [B→aB·, a/b]$
  $[B→·aB, a/b]$
  $[B→·b, a/b] \}$

  $I_4 = \{ [B→b·, a/b] \}$

URBAN EDGE 5/2/20

GOTO $(I_2, B) = \{[S \to BB\cdot, \$]\} = I_5$

GOTO $(I_2, a) = \{[B \to a\cdot B, \$]$
$[B \to \cdot aB, \$]$
$[B \to \cdot b, \$]\} = I_6$

GOTO $(I_3, B) = \{[B \to aB\cdot, a/b]\} = I_8$

GOTO $(I_3, a) = \{[B \to a\cdot B, a/b]$
$[B \to \cdot aB, a/b]$
$[B \to \cdot b, a/b]\} = I_3$

GOTO $(I_3, b) = \{[B \to b\cdot, a/b]\} = I_4$

GOTO $(I_2, b) = \{[B \to b\cdot, \$]\} = I_7$

GOTO $(I_6, B) = \{[B \to aB\cdot, \$]\} = I_9$

GOTO $(I_6, a) = \{[B \to a\cdot B, \$]$
$[B \to \cdot aB, \$]$
$[B \to \cdot b, \$]\} = I_6$

GOTO $(I_6, b) = \{[B \to b\cdot, \$]\} = I_7$

Canonical LR(1) Parsing Table

- Method

→ Construct $C = \{I_0, I_1, \ldots I_n\}$, the collection of LR(1) items for G'.

→ State i is constructed from state I_i:

* If $[A \to \alpha\cdot a\beta, b]$ is in I_i & Goto $(I_i, a) = I_j$, then set ACTION $[i, a]$ to "shift j"
* If $[A \to \alpha\cdot, a]$ is in I_i, then set ACTION $[i, a]$ to "reduce $A \to \alpha$"
* If $[S' \to S\cdot, \$]$ is in I_i, then set ACTION $[i, \$]$ to "Accept"

URBAN EDGE 5/2/20

	Action Table			Goto Table	
State	a	b	$	S	B
0	s3	s4		1	2
1			accept		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7	r3	r3	r3		
8	r2	r2			
9			r2		

LALR (Look Ahead) Parser

- It reduces the no. of states of a CLR parser.

eg: I_4 : GOTO (I_0, b) I_7 : GOTO (I_2, b)

$\{B \to b\cdot, a/b\}$ $\{B \to b\cdot, \$\}$

Combine I_4 & I_7 to form I_{47} : $\{B \to b\cdot, a/b/\$\}$