LR Parser
Module 3_Part 2

Prepared by

Jesna J S

Assistant Professor, CSE TKMCE

LR Parser

- LR parsing is one type of Bottom Up parsing. It is used to parse the large class of grammars.
- Also called LR(k) parsing
- In the LR parsing, "L" stands for left-to-right scanning of the input.
- "R" stands for constructing a right most derivation in reverse.
- "K" is the number of input symbols of the look ahead used to make number of parsing decision

Why LR Parsing is used?

- LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written.
- The LR parsing method is the most general nonbacktracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
- 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.
- 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.

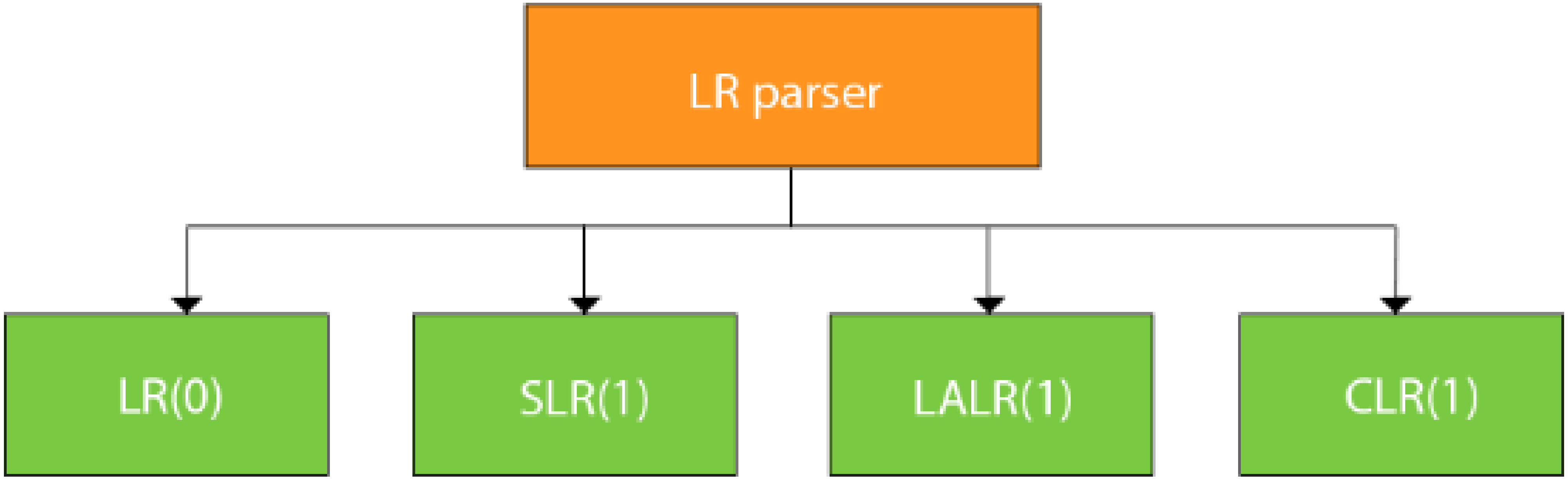


Fig: Types of LR parser

Simple LR (SLR)

- Easiest to implement
- Least powerful
- It may fail to produce parsing table for certain grammars on which the other methods succeed.

Canonical LR

- Most powerful
- Most expensive

LALR

- Work on most programming language grammars and, with some effort, can be implemented efficiently.

Algorithm for finding closure of items

```
function closure ( I );
begin
        J := I;
        repeat
                for each item A → α·Bβ in J and each production
                        B → γ of G such that B → ·γ is not in J do
                                add B → ·γ to J
        until no more items can be added to J;
        return J
end
```

Fig. 4.33. Computation of *closure*.

The Sets-of-Items Construction

We are now ready to give the algorithm to construct C, the canonical collection of sets of LR(0) items for an augmented grammar G'; the algorithm is shown in Fig. 4.34.

```
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
```

Fig. 4.34. The sets-of-items construction.

Algorithm for constructing SLR parsing table

Algorithm 4.8. Constructing an SLR parsing table.

Input. An augmented grammar G'.

Output. The SLR parsing table functions *action* and *goto* for G'.

Method.

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(0) items for G'.

2. State i is constructed from I_i. The parsing actions for state i are determined as follows:

   a) If $[A \rightarrow \alpha \cdot a\beta]$ is in I_i and $goto(I_i, a) = I_j$, then set $action[i, a]$ to "shift j." Here a must be a terminal.

   b) If $[A \rightarrow \alpha \cdot]$ is in I_i, then set $action[i, a]$ to "reduce $A \rightarrow \alpha$" for all a in FOLLOW(A); here A may not be S'.

   c) If $[S' \rightarrow S \cdot]$ is in I_i, then set $action[i, \$]$ to "accept."

Examples

SLR Parsing

Eg: $S \rightarrow AA$

$A \rightarrow aA$

$\quad\quad / b$

Step - 1

Construct augmented grammar

$S' \rightarrow S\$$

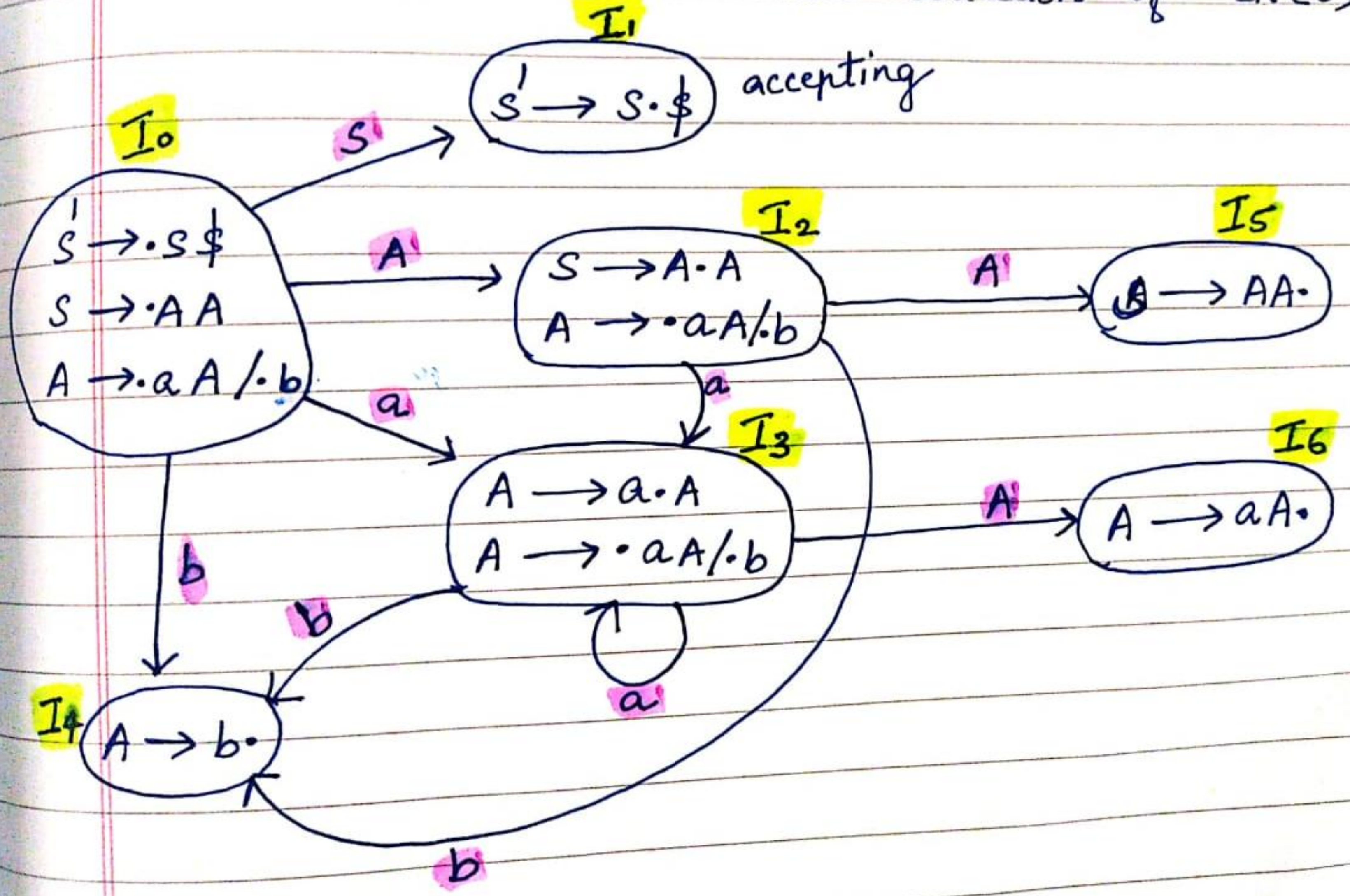
$S \rightarrow AA$

$A \rightarrow aA$

$\quad\quad / b$

step - 2

Construct canonical collection of LR(0) items



step-4

Number the productions

$S \longrightarrow AA$ —— r_1

$A \longrightarrow aA$ —— r_2

$\quad\;\; | \; b$ —— r_3

step-5

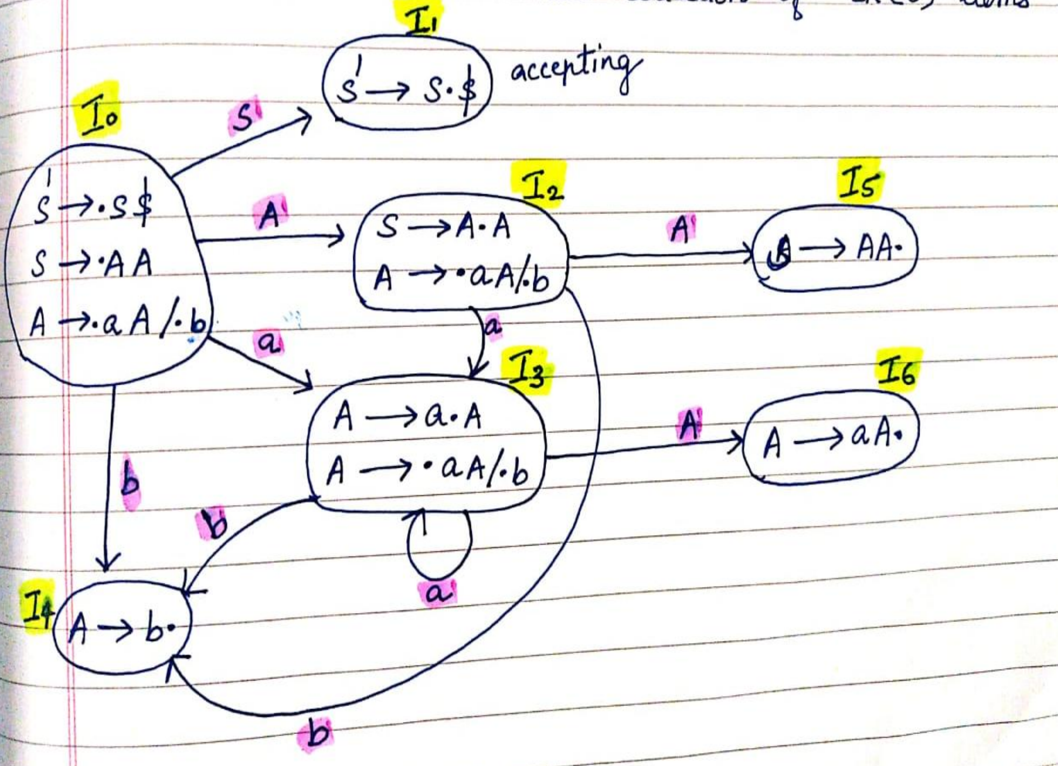
Construct the parsing table

State	Action			Goto	
	a	b	$	S	A
0	S3	S4		1	2
1			Acc		
2	S3	S4			5
3	S3	S4			6
4	r3	r3			
5			r1		
6	r2	r2			

follow (S) = { $ }
follow (A) = { a, b }

step-2

Construct canonical collection of LR(0) items



Question

→ Construct SLR parsing table for below grammars

(1)

$S \rightarrow dA$
$\quad / aB$
$A \rightarrow bA$
$\quad / c$
$B \rightarrow bB$
$\quad / c$

(2)

$E \rightarrow T + E / T$
$T \rightarrow id$

3) $E \rightarrow E+T | T$
$T \rightarrow T*F | F$
$F \rightarrow (E) | id$

$S \to dA$
$\quad / aB$
$A \to bA$
$\quad / c$
$B \to bB$
$\quad / c$

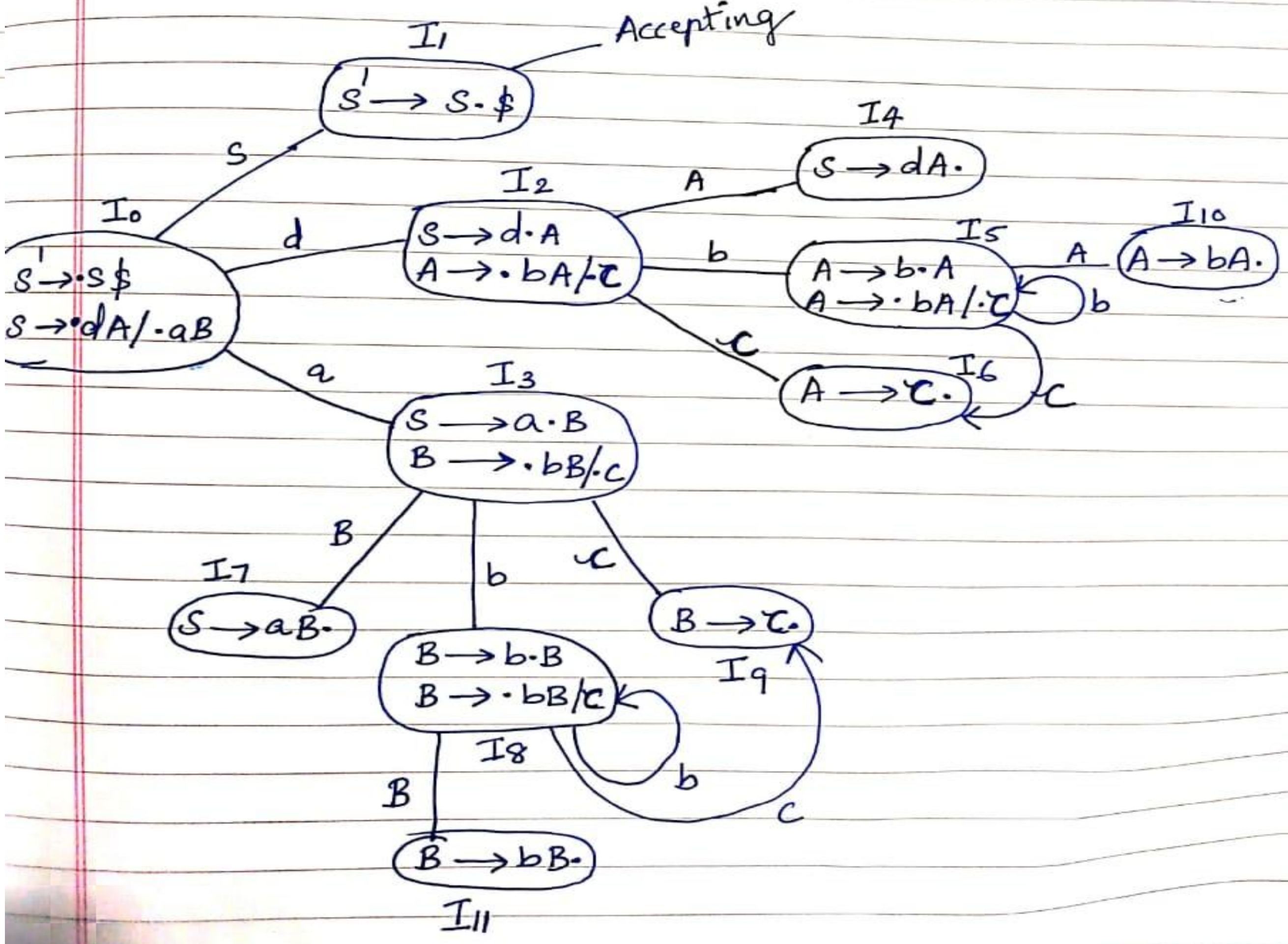
<u>Augmented grammar :</u>

$S' \to S\$$ —— (r_1)
$S \to dA$ —— (r_2)
$\quad / aB$ —— (r_3)
$A \to bA$ —— (r_4)
$\quad / c$ —— (r_5)
$B \to bB$ —— (r_6)
$\quad / c$ —— (r_7)

Canonical collection of LR(0) items

$/c$ —— (r_7)

✓ <u>Canonical collection of LR(0) items</u>



Check whether the given grammar is suitable for SLR(1) parsing

Grammar

$S \rightarrow Aa$ — r_1

$S \rightarrow bAc$ — r_2

$S \rightarrow Bc$ — r_3

$S \rightarrow bBa$ — r_4

$A \rightarrow d$ — r_5

$B \rightarrow d$ — r_6

Augmented grammar

$S' \rightarrow S\$$

$S \rightarrow Aa$

$S \rightarrow bAc$

$S \rightarrow Bc$

$S \rightarrow bBa$

$A \rightarrow d$

$B \rightarrow \cdot d$

Canonical collection of LR(0) items

closure operation

$S' \to \cdot S\$$

$S \to \cdot Aa$

$S \to \cdot bAc$

$S \to \cdot Bc$

$S \to \cdot bBa$

$A \to \cdot d$

$B \to \cdot d$

- Goto $(I_0, S) = I_1$

  $S' \to S \cdot \$$

- Goto $(I_0, A) = I_2$

  $S \to A \cdot a$

- Goto $(I_0, b) = I_3$

  $S \to b \cdot Ac$ $A \to \cdot d$

  ~~$S \to bAc$~~ $B \to \cdot d$

  $S \to b \cdot Ba$

- Goto $(I_0, B) = I_4$

  $S \to B \cdot c$

- Goto $(I_0, d) = I_5$

  $A \to d \cdot$

  $B \to d \cdot$

~~Goto (I_1, a)~~

- Goto $(I_2, a) = I_6$

  $S \to Aa \cdot$

- Goto $(I_3, A) = I_7$

  $S \to bA \cdot c$

- Goto $(I_3, B) = I_8$

  $S \to bB \cdot a$

- Goto $(I_3, d) = I_5$ //Loop

- Goto $(I_4, c) = I_9$

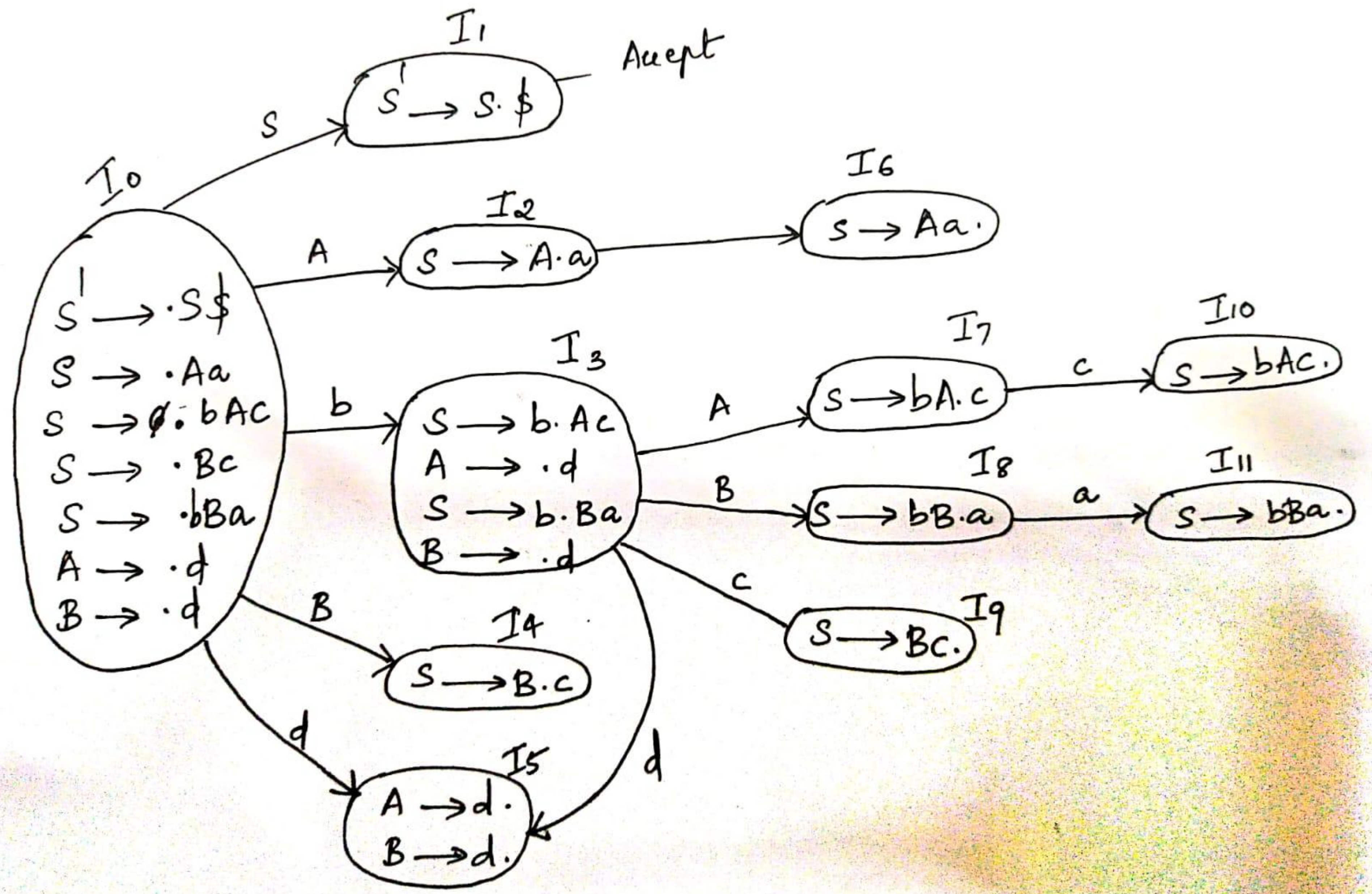
  $S \to Bc \cdot$

- Goto $(I_7, c) = I_{10}$

  $S \to bAc \cdot$

- Goto $(I_8, a) = I_{11}$

  $S \to bBa \cdot$



I1
S' → S.$
Accept
I0
S' → .S$
S → .Aa
S → .bAC
S → .Bc
S → .bBa
A → .d
B → .d
S
A
b
B
d
I2
S → A.a
I6
S → Aa.
I3
S → b.Ac
A → .d
S → b.Ba
B → .d
I7
S → bA.c
I10
S → bAC.
c
A
I8
S → bB.a
I11
S → bBa.
a
B
c
I9
S → Bc.
I4
S → B.c
I5
A → d.
B → d.
d

SLR parse Table

State	Action					Goto		
	a	b	c	d	$	S	A	B
0		S3		S5			2	4
1					Acc			
2	S6							
3				S5			7	8
4			S9					
5	r5/r6		r5/r6					
6					r1			
7			S10					
8	S11							
9					r3			
10					r2			
11					r4			

follow (S) = { $ }
follow (A) = { a, c }
follow (B) = { a, c }

Hence the grammar is not suitable for SLR(1) parsing.

- A grammar is not SLR(1) if there **exists a shift-reduce or reduce-reduce conflict in the SLR(1) parsing table**, meaning that **a reduce action cannot be uniquely determined using only the FOLLOW set of the corresponding non-terminal**.

Question:: Construct SLR parsing table for the following grammar. Check if the grammar is SLR or not. Justify your answer.

E->T +E |T

F-> id

Question:: Check whether the given grammar is LR(0) or not

S→L = R | R

L→ * R | id

R→ L

Question:: Check whether the following grammar LR(0) or not

S → AaAb | BbBa

A → €

B → €

Example 4.39. Every SLR(1) grammar is unambiguous, but there are many unambiguous grammars that are not SLR(1). Consider the grammar with productions

$$
\begin{aligned}
S &\rightarrow L = R \\
S &\rightarrow R \\
L &\rightarrow * R \\
L &\rightarrow \mathbf{id} \\
R &\rightarrow L
\end{aligned}
\tag{4.20}
$$

Write the canonical collection of LR(0) items and create SLR parsing table

Grammar (4.20) is not ambiguous. This shift/reduce conflict arises from the fact that the SLR parser construction method is not powerful enough to remember enough left context to decide what action the parser should take on input = having seen a string reducible to L. The canonical and LALR methods, to be discussed next, will succeed on a larger collection of grammars, including grammar (4.20). It should be pointed out, however, that there are unambiguous grammars for which every LR parser construction method will produce a parsing action table with parsing action conflicts. Fortunately, such grammars can generally be avoided in programming language applications. □

CLR and LALR Parsing

CLR Parsing LR(1) canonical items

Grammar

r1, r2, r3, r4

$S \rightarrow aAd \,/\, bBd \,/\, aBe \,/\, bAe$

$A \rightarrow c$ — r5

$B \rightarrow c$ — r6

Augmented Grammar

$S' \rightarrow S\$$

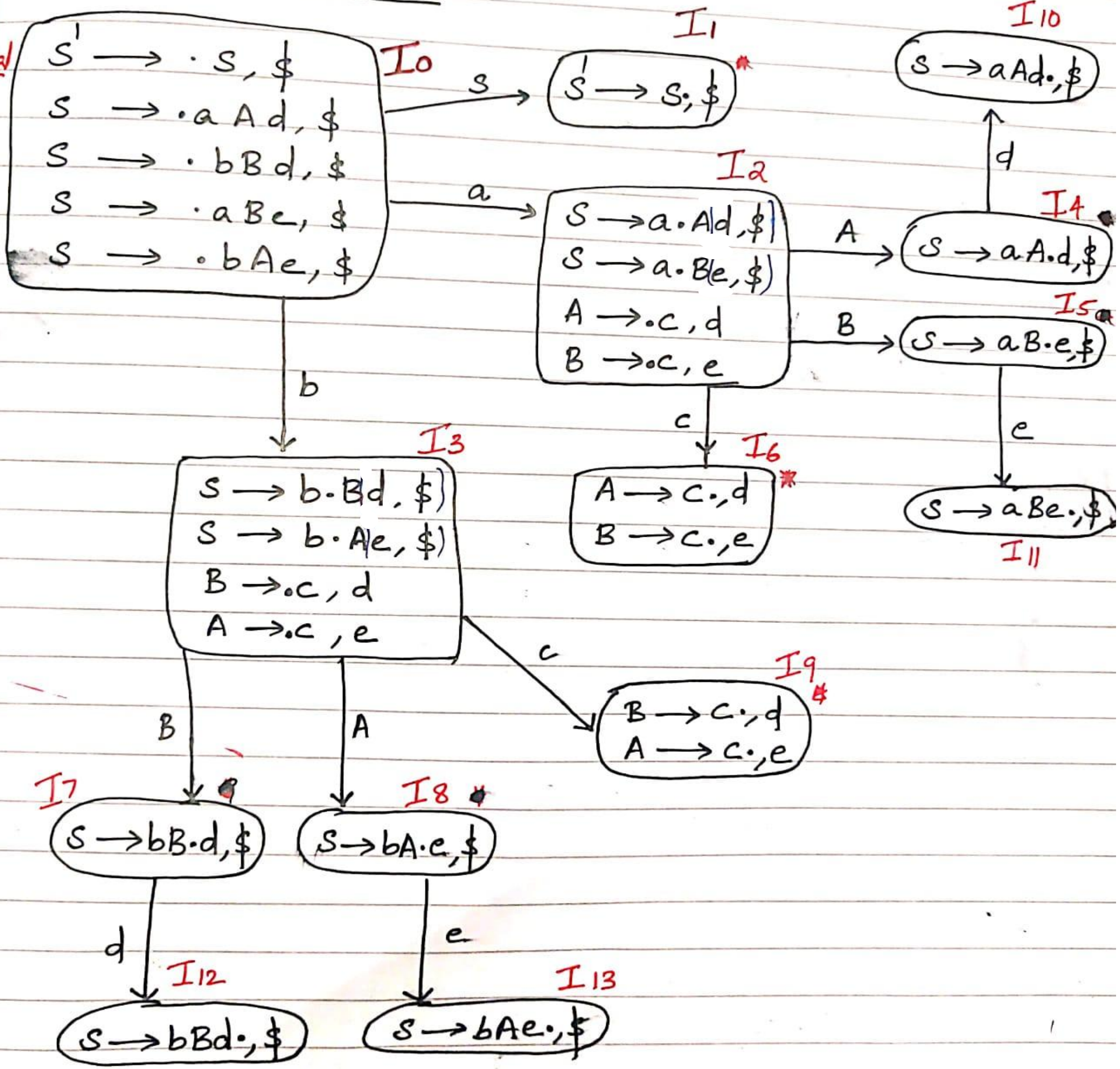
$S \rightarrow aAd \,/\, bBd \,/\, aBe \,/\, bAe$

$A \rightarrow c$

$B \rightarrow c$

I10

LR(1) Canonical items



CLR (1) parsing Table construction

State	Action						Goto		
	a	b	c	d	e	$	S	A	B
0	S2	S3					1		
1						Acc			
2								4	5
3			S9					8	7
4				S10					
5					S11				
6				r5	r6				
7				S12					
8					S13				
9				r6	r5				
10						r1			
11						r3			
12						r2			
13						r4			

Q.(II) $S \rightarrow (S)/a$.

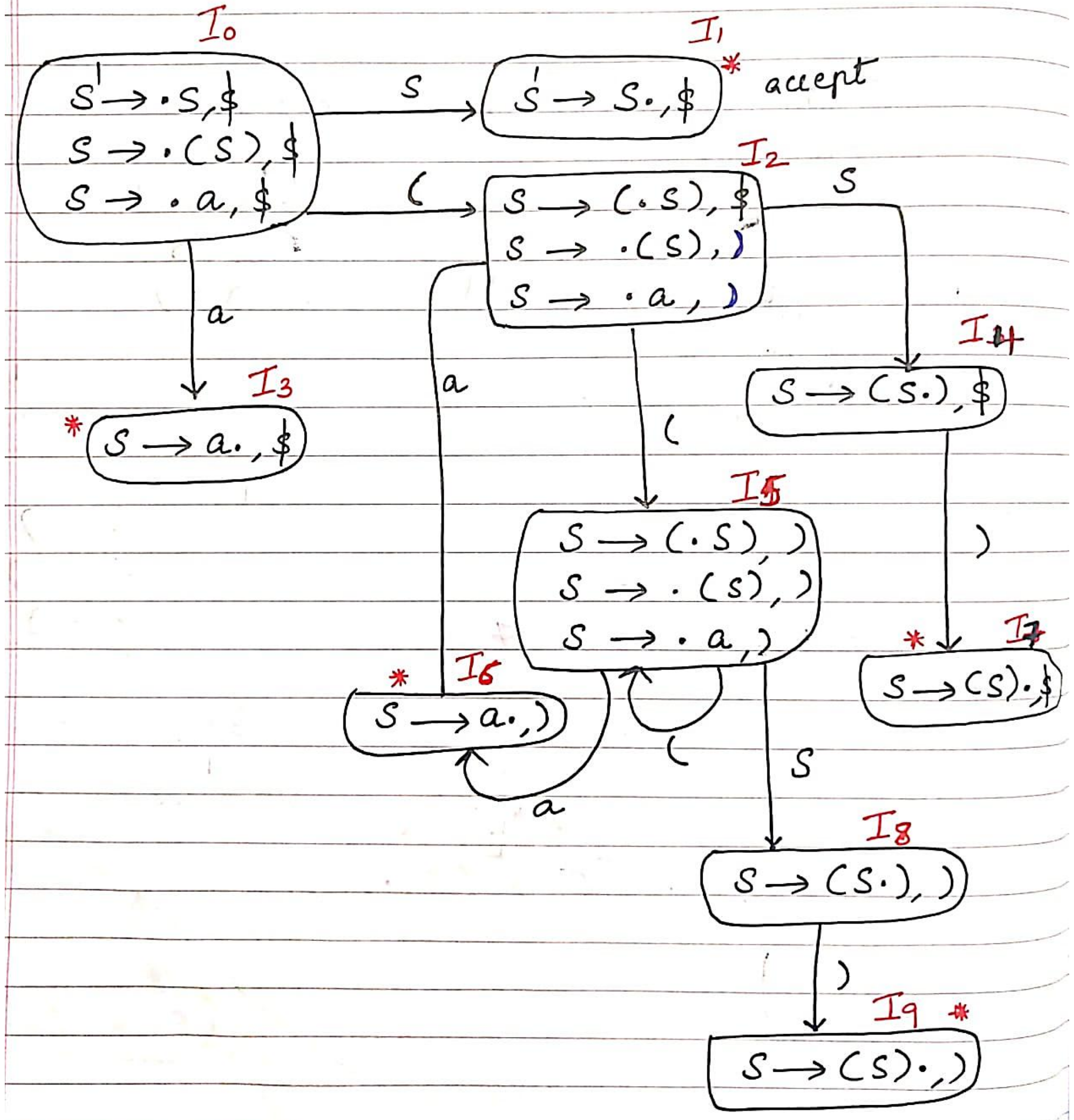
Augmented grammar

$$S' \rightarrow S\,\$$$
$$S \rightarrow (S) \;/\; a$$

r1 r2

Canonical collection of LR(1) items

Canonical collection of LR(1) items



CLR parsing Table

State	Action				Goto
	(	)	a	$	S
0	S2		S3		1
1				acc	
2	S5		S6		4
3				r2	
4		S7			
5	S5		S6		8
6		r2			
7				r1	
8		S9			
9		r1			

LALR parsing Table.

State	Action				Goto
	(	)	a	$	S
0	S25		S36		1
1				acc	
25	S25		S36		48
36		r2		r2	
48		S79			
79		r1		r1	

LR Parsing Algorithm

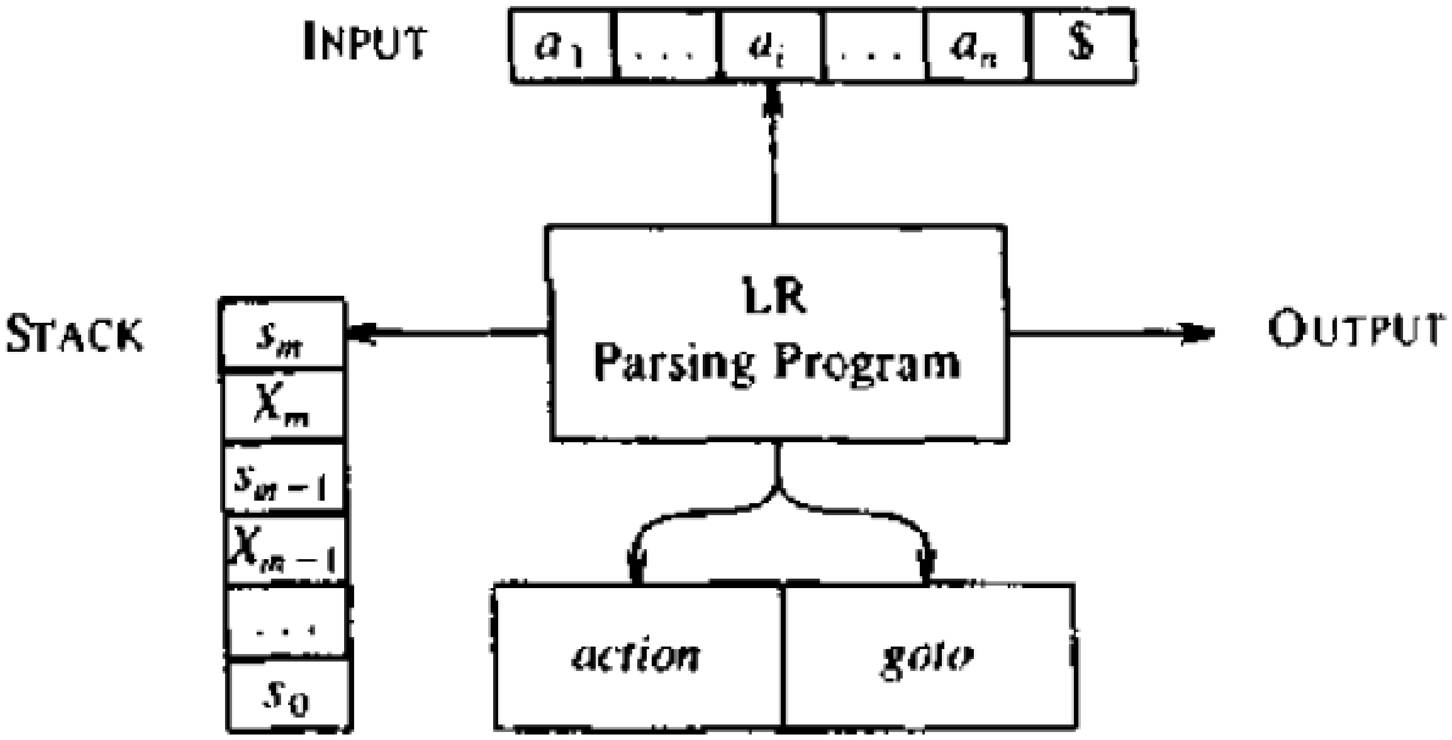


Fig. 4.29. Model of an LR parser.

- It consists of
  - ✓ Input
  - ✓ Output
  - ✓ Stack
  - ✓ Driver program
  - ✓ Parsing Table (Has two parts: action, goto)
- The driver program is the same for all LR parsers; only the parser table changes from one parser to another.
- The parser program reads characters from an input buffer one at a time.
- The program uses a stack to store a string of the form $s_0X_1s_1X_2S_2\ldots\ldots X_ms_m$, where s_m is on the top.
  - ➢ **X_i → grammar symbol**
  - ➢ **S_i → state symbol**

- Each state symbol summarizes the information contained in the stack below it, and the **combination of the state symbol on top of the stack and the current input symbol are used to index the parsing table and determine shift reduce parsing decision**.
- Parsing table consists of two parts, parsing action function action and a goto function goto.

The program driving LR parser behaves as follows:

1. It determines s_m the state currently on top of the stack, and a_i the current input symbol.
2. It then consults action[s_m,a_i] the paring action table entry for state s_m and input a_i, which can one of four values

a. Shift s, where s is a state
b. Reduce by a grammar production $A \rightarrow B$
c. Accept
d. Error

- The function **goto** takes a state and grammar symbol as arguments and produces a state.
- Goto function of a parsing table constructed from a grammar G is the transition function of a deterministic finite automata that recognizes the viable prefixes of G

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

$$(s_0 X_1 s_1 X_2 s_2 \cdots X_m s_m, \; a_i a_{i+1} \cdots a_n \$)$$

This configuration represents the right-sentential form

$$X_1 X_2 \cdots X_m a_i a_{i+1} \cdots a_n$$

in essentially the same way as a shift-reduce parser would; only the presence of states on the stack is new.

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:

1. If $action[s_m, a_i]$ = shift s, the parser executes a shift move, entering the configuration

$$(s_0\, X_1\, s_1\, X_2\, s_2 \cdots X_m\, s_m\, a_i\, s, \quad a_{i+1} \cdots a_n\$)$$

Here the parser has shifted both the current input symbol a_i and the next state s, which is given in $action[s_m, a_i]$, onto the stack; a_{i+1} becomes the current input symbol.

2. If $action[s_m, a_i]$ = reduce $A \to \beta$, then the parser executes a reduce move, entering the configuration

$$(s_0 X_1 s_1 X_2 s_2 \cdots X_{m-r} s_{m-r} A s, \quad a_i a_{i+1} \cdots a_n \$)$$

where $s = goto[s_{m-r}, A]$ and r is the length of β, the right side of the production. Here the parser first popped $2r$ symbols off the stack (r state symbols and r grammar symbols), exposing state s_{m-r}. The parser then pushed both A, the left side of the production, and s, the entry for $goto[s_{m-r}, A]$, onto the stack. The current input symbol is not changed in a reduce move. For the LR parsers we shall construct, $X_{m-r+1} \cdots X_m$, the sequence of grammar symbols popped off the stack, will always match β, the right side of the reducing production.

3. If $action[s_m, a_i]$ = accept, parsing is completed.

4. 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 w and an LR parsing table with functions *action* and *goto* for a grammar G.

Output. If w is in $L(G)$, a bottom-up parse for w; otherwise, an error indication.

Method. Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the program in Fig. 4.30 until an accept or error action is encountered. □

Algorithm

```
set ip to point to the first symbol of w$;
repeat forever begin
    let s be the state on top of the stack and
        a the symbol pointed to by ip;
    if action[s, a] = shift s' then begin
        push a then s' on top of the stack;
        advance ip to the next input symbol
    end
    else if action[s, a] = reduce A -> β then begin
        pop 2*|β| symbols off the stack;
        let s' be the state now on top of the stack;
        output the production A -> β
    end
    else if action[s, a] = accept then
        return
    else error()
end
```

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $F \to (E)$
(6) $F \to \mathbf{id}$

1. s*i* means shift and stack state *i*,
2. r*j* means reduce by production numbered *j*,
3. acc means accept,
4. blank means error.

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

Fig. 4.31. Parsing table for expression grammar.

	Stack	Input	Action
(1)	0	**id** * **id** + **id** $	shift
(2)	0 **id** 5	* **id** + **id** $	reduce by $F \to$ **id**
(3)	0 F 3	* **id** + **id** $	reduce by $T \to F$
(4)	0 T 2	* **id** + **id** $	shift
(5)	0 T 2 * 7	**id** + **id** $	shift
(6)	0 T 2 * 7 **id** 5	+ **id** $	reduce by $F \to$ **id**
(7)	0 T 2 * 7 F 10	+ **id** $	reduce by $T \to T*F$
(8)	0 T 2	+ **id** $	reduce by $E \to T$
(9)	0 E 1	+ **id** $	shift
(10)	0 E 1 + 6	**id** $	shift
(11)	0 E 1 + 6 **id** 5	$	reduce by $F \to$ **id**
(12)	0 E 1 + 6 F 3	$	reduce by $T \to F$
(13)	0 E 1 + 6 T 9	$	$E \to E+T$
(14)	0 E 1	$	accept

Fig. 4.32. Moves of LR parser on **id** * **id** + **id**.

SLR(1) v/s CLR(1) v/s LALR(1)

Feature	SLR(1) Parser (Simple LR)	CLR(1) Parser (Canonical LR)	LALR(1) Parser (Look-Ahead LR)
Parsing Table Size	Smallest (fewer states)	Largest (most states)	Medium (states merged to reduce size)
Grammar Handling	Limited (only simple grammars)	Most powerful (handles almost all grammars)	Nearly as powerful as CLR but compact

Difference conti..

Basis for Decisions	Uses FOLLOW sets for reductions	Uses look-ahead symbols to make precise decisions	Uses merged look-ahead symbols, similar to CLR but optimized
Conflicts (Shift-Reduce, Reduce-Reduce)	More conflicts due to reliance on FOLLOW sets	Least conflicts because of look-ahead symbols	May introduce reduce-reduce conflicts when merging states
Error Detection	Delayed (errors detected later)	Delayed (similar to SLR)	Similar to CLR, not always immediate

Difference conti..

Time and Space Complexity	Low (fast but limited)	High (slow due to large tables but powerful)	Medium (optimized for efficiency)
Ease of Implementation	Easiest (simplest to build)	Most complex (large tables make it harder)	Easier than CLR but slightly more complex than SLR
Used In	Simple parsers and educational tools	Strong theoretical compilers (not widely used in practice)	Most real-world compilers (YACC, Bison, etc.)

LR Grammar

- A grammar for which we can construct a parsing table is said to be an LR grammar.
- In order for a grammar to be LR it is sufficient that a left-to-right, shift-reduce parser be able to recognize handles when they appear on top of the stack.

<u>Characteristics of LR Grammars</u>

- A grammar is LR(k) if:
    1. It can be parsed from Left to right (L).
    2. It constructs a Rightmost derivation (R) in reverse.
    3. It uses k tokens of lookahead to decide parsing actions.

- The most common variant is LR(1), which uses 1 token lookahead.
- LALR(1) is commonly used because it balances power and efficiency.
- If a grammar is CLR(1), it is always LALR(1) and SLR(1).
- Compilers prefer LALR(1) over CLR(1) because of smaller parsing tables

Exercise Problems:

1. Construct operator precedence table for the following grammar
2. E->E+E|E*E|id.
3. Construct SLR parsing table for the following grammar. Check if the grammar is SLR or not. Justify your answer.

   E->T +E |T

   F-> id
4. Construct canonical LR(0) collection of items for the grammar below.

   S➔ L=R/R

   L ➔ *R/id

   R ➔ L
5. Also identify a shift reduce conflict in the LR(0) collection constructed above.
6. List all the LR(0) items for the grammar S→AS|b , A→SA|a