Syntactic Analysis (Bottom-up Parsing)

Dr. P K Singh

LR Parsers

The most powerful shift-reduce parsing (yet efficient) is:

LR(k) parsing.

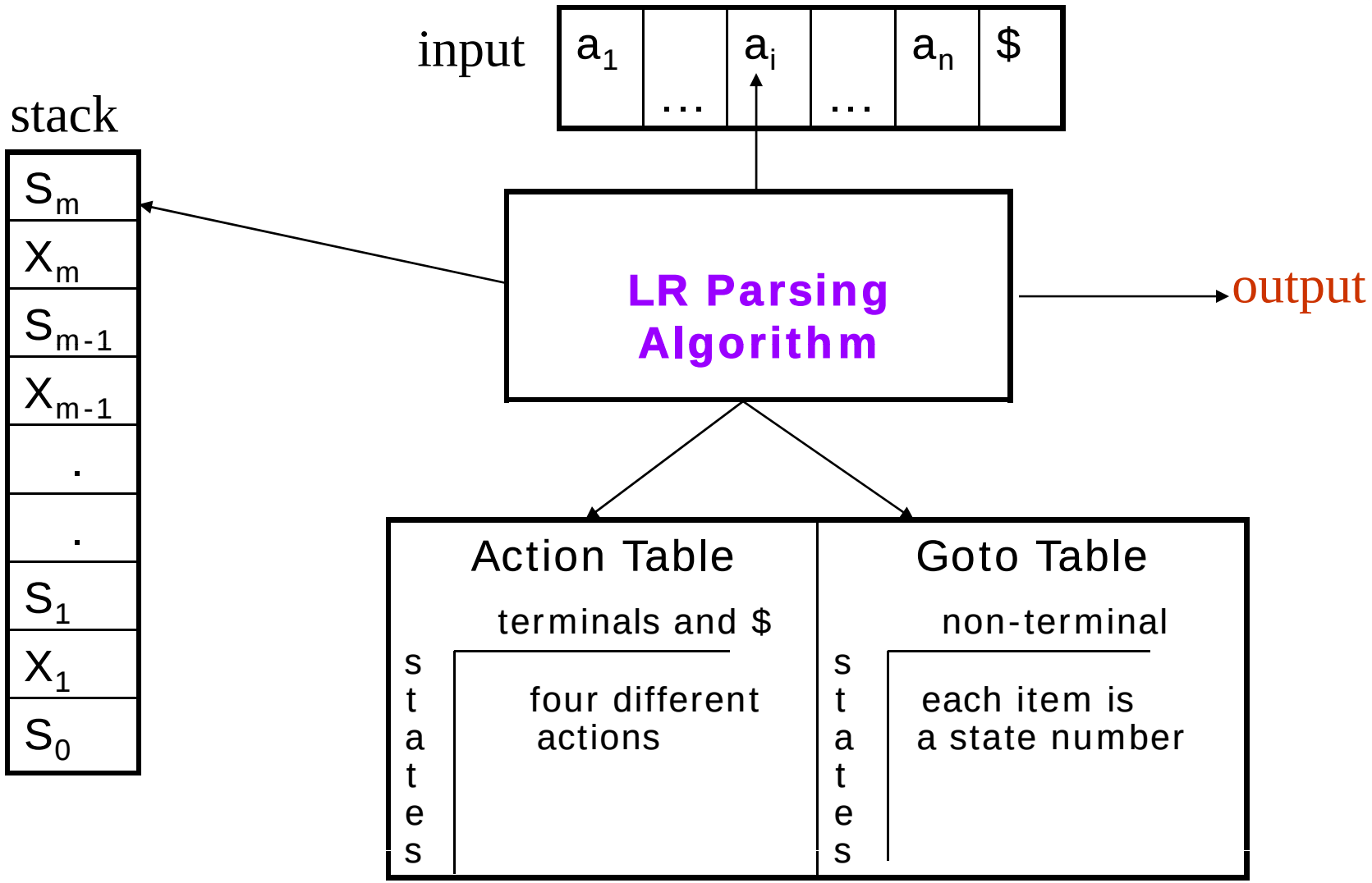
- LR parsing is attractive because:
  - LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
  - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

    LL(1)-Grammars $\subset$ LR(1)-Grammars
  - An LR-parser can detect a syntactic error as soon as it is possible to do so a left-to-right scan of the input.

LR Parsers

- **LR-Parsers**
  - covers wide range of grammars.
  - SLR – simple LR parser
  - LR – most general LR parser
  - LALR – intermediate LR parser (look-head LR parser)
  - SLR, LR and LALR work same (they used the same algorithm), only their parsing tables are different.

LR Parsing


input
a_1
...
a_i
...
a_n
$
stack
S_m
X_m
S_{m-1}
X_{m-1}
.
.
S_1
X_1
S_0
LR Parsing Algorithm
output
Action Table
terminals and $
states
four different actions
Goto Table
non-terminal
states
each item is a state number

A Configuration of LR Parsing Algorithm

- A configuration of a LR parsing is:

$$(S_o X_1 S_1 \ldots X_m S_m, \; a_i a_{i+1} \ldots a_n \$)$$

Stack — Rest of Input

- S_m and a_i decides the parser action by consulting the parsing action table. (*Initial Stack* contains just S_o)

- A configuration of a LR parsing represents the right sentential form:

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

Actions of A LR-Parser

1. **shift s** -- shifts the next input symbol and the state **s** onto the stack

   $(S_o X_1 S_1 \ldots X_m S_m, a_i a_{i+1} \ldots a_n \$) \rightarrow (S_o X_1 S_1 \ldots X_m S_m a_i s, a_{i+1} \ldots a_n \$)$

2. **reduce $A \rightarrow \beta$** (or **rn** where n is a production number)
   - pop $2|\beta|$ (=r) items from the stack;
   - then push **A** and **s** where **$s = goto[s_{m-r}, A]$**

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

   - Output is the reducing production reduce $A \rightarrow \beta$

3. **Accept** – Parsing successfully completed
4. **Error** -- Parser detected an error (an empty entry in the action table)

Reduce Action

- pop $2|\beta|$ $(=r)$ items from the stack; let us assume that $\beta = Y_1 Y_2 \ldots Y_r$
- then push **A** and **s** where **$s = goto[s_{m-r}, A]$**

$(S_o X_1 S_1 \ldots X_{m-r} S_{m-r} Y_1 S_{m-r} \ldots Y_r S_m, a_i a_{i+1} \ldots a_n \$)$

➔ $(S_o X_1 S_1 \ldots X_{m-r} S_{m-r} A s, a_i \ldots a_n \$)$

- In fact, $Y_1 Y_2 \ldots Y_r$ is a handle.

$X_1 \ldots X_{m-r} A a_i \ldots a_n \$ \Rightarrow X_1 \ldots X_m Y_1 \ldots Y_r a_i a_{i+1} \ldots a_n \$$

(SLR) Parsing Tables for Expression Grammar

1) $E \rightarrow E+T$
2) $E \rightarrow T$
3) $T \rightarrow T*F$
4) $T \rightarrow F$
5) $F \rightarrow (E)$
6) $F \rightarrow id$

Action Table | Goto Table

state	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				

Actions of A (S)LR-Parser -- Example

stack	input	action	output
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	accept	

Constructing SLR Parsing Tables

LR(0) Item

- An **LR(0) item** of a grammar G is a production of G a dot at the some position of the right side.
- Ex: $A \rightarrow aBb$ Possible LR(0) Items: (four different possibility)
  - $A \rightarrow \blacksquare aBb$
  - $A \rightarrow a \blacksquare Bb$
  - $A \rightarrow aB \blacksquare b$
  - $A \rightarrow aBb \blacksquare$
- Sets of LR(0) items will be the states of action and goto table of the SLR parser.
- A collection of sets of LR(0) items (**the canonical LR(0) collection**) is the basis for constructing SLR parsers.
- *Augmented Grammar*:

  G' is G with a new production rule $S' \rightarrow S$ where S' is the new starting symbol.

The Closure Operation

- 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 the two rules:

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

  2. If $A \rightarrow \alpha.B\beta$ is in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow .\gamma$ will be in the closure(I).
     We will apply this rule until no more new LR(0) items can be added to closure(I).

The Closure Operation -- Example

$E' \rightarrow E$

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

closure({$E' \rightarrow \cdot E$}) =

{ $E' \rightarrow \cdot E$ ⟵ kernel items

$E \rightarrow \cdot E+T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T*F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id$ }

Goto Operation

- If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  - If $A \rightarrow \alpha \blacksquare X\beta$ in I

    then every item in **closure({$A \rightarrow \alpha X \blacksquare \beta$})** will be in goto(I,X).

Example:

I = { $E' \rightarrow \blacksquare E$, $E \rightarrow \blacksquare E+T$, $E \rightarrow \blacksquare T$,
$T \rightarrow \blacksquare T*F$, $T \rightarrow \blacksquare F$,
$F \rightarrow \blacksquare (E)$, $F \rightarrow \blacksquare id$ }

goto(I,E) = { $E' \rightarrow E \blacksquare$, $E \rightarrow E \blacksquare +T$ }

goto(I,T) = { $E \rightarrow T \blacksquare$, $T \rightarrow T \blacksquare *F$ }

goto(I,F) = {$T \rightarrow F \blacksquare$ }

goto(I,() = { $F \rightarrow (\blacksquare E)$, $E \rightarrow \blacksquare E+T$, $E \rightarrow \blacksquare T$, $T \rightarrow \blacksquare T*F$, $T \rightarrow \blacksquare F$,
$F \rightarrow \blacksquare (E)$, $F \rightarrow \blacksquare id$ }

goto(I,id) = { $F \rightarrow id \blacksquare$ }

Construction of The Canonical LR(0) Collection

- To create the SLR parsing tables for a grammar G, we will create the canonical LR(0) collection of the grammar G'.
- ***Algorithm***:

  C is { closure({S'→ ▪ S}) }

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

    for each I in ***C*** and each grammar symbol X

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

        add goto(I,X) to ***C***

- goto function is a DFA on the sets in C.

The Canonical LR(0) Collection -- Example

I_0: E' → .E
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_1: E' → E
E → E.+T

I_2: E → T.
T → T.*F

I_3: T → F.

. I_4: F → (.E)
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_5: F → id.

I_6: E → E+.T
T → .T*F
T → .F
F → .(E)
F → .id

I_7: T → T*.F
F → .(E)
F → .id

.

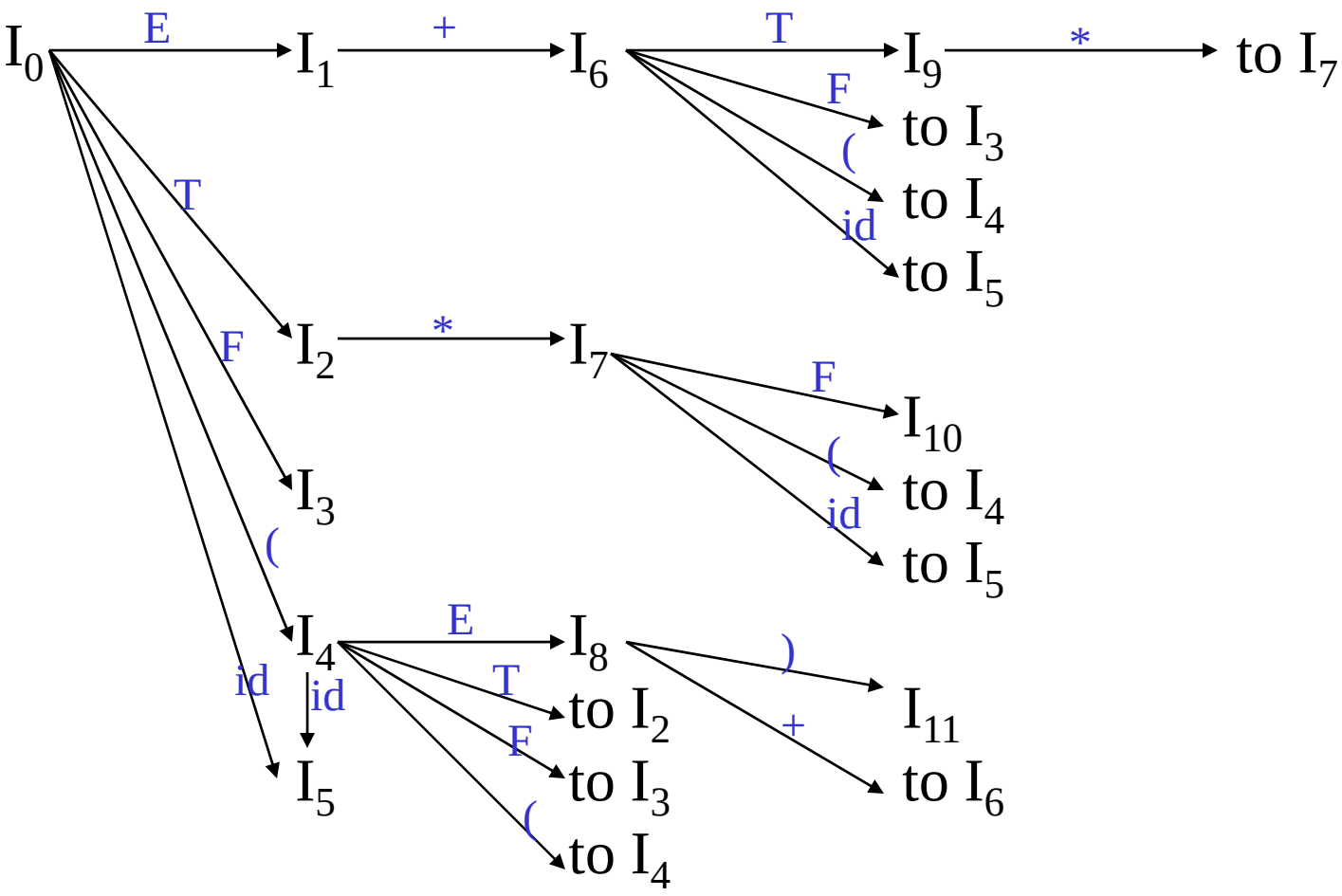
I_8: F → (E.)
E → E.+T

I_9: E → E+T.
T → T.*F

I_{10}: T → T*F

I_{11}: F → (E).

Transition Diagram (DFA) of Goto Function



Constructing SLR Parsing Table
(of an augumented grammar G')

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

2. Create the parsing action table as follows
   - If a is a terminal, $A \rightarrow \alpha.a\beta$ in I_i and $goto(I_i,a)=I_j$ then action[i,a] is ***shift j.***
   - If $A \rightarrow \alpha.$ is in I_i, then action[i,a] is ***reduce*** $A \rightarrow \alpha$ for all a in FOLLOW(A) where $A \neq S'$.
   - If $S' \rightarrow S.$ is in I_i, then action[i,$] is ***accept.***
   - If any conflicting actions generated by these rules, the grammar is not SLR(1).

3. Create the parsing goto table
   - for all non-terminals A, if $goto(I_i,A)=I_j$ then goto[i,A]=j

4. All entries not defined by (2) and (3) are errors.

5. Initial state of the parser contains $S' \rightarrow .S$

Parsing Tables of Expression Grammar

	Action Table						Goto Table		
state	**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			

SLR(1) Grammar

- An LR parser using SLR(1) parsing tables for a grammar G is called as the SLR(1) parser for G.
- If a grammar G has an SLR(1) parsing table, it is called SLR(1) grammar (or SLR grammar in short).
- Every SLR grammar is unambiguous, but every unambiguous grammar is not a SLR grammar.

shift/reduce and reduce/reduce conflicts

- If a state does not know whether it will make a shift operation or reduction for a terminal, we say that there is a **shift/reduce conflict**.
- If a state does not know whether it will make a reduction operation using the production rule i or j for a terminal, we say that there is a **reduce/reduce conflict**.
- If the SLR parsing table of a grammar G has a conflict, we say that that grammar is not SLR grammar.

Conflict Example (Shift-Reduce)

$S \rightarrow L=R$
$S \rightarrow R$
$L \rightarrow *R$
$L \rightarrow id$
$R \rightarrow L$

I_0: $S' \rightarrow .S$
$S \rightarrow .L=R$
$S \rightarrow .R$
$L \rightarrow .*R$
$L \rightarrow .id$
$R \rightarrow .L$

I_1: $S' \rightarrow S.$

I_2: $S \rightarrow L.=R$
$R \rightarrow L.$

I_3: $S \rightarrow R.$

I_4: $L \rightarrow *.R$
$R \rightarrow .L$
$L \rightarrow .*R$
$L \rightarrow .id$

I_5: $L \rightarrow id.$

I_6: $S \rightarrow L=.R$
$R \rightarrow .L$
$L \rightarrow .*R$
$L \rightarrow .id$

I_7: $L \rightarrow *R.$

I_8: $R \rightarrow L.$

I_9: $S \rightarrow L=R.$

Problem

FOLLOW(R) = { =, $ }

= → shift 6

→ reduce by $R \rightarrow L$

shift/reduce conflict

Action[2,=] = shift 6

Action[2,=] = reduce by $R \rightarrow L$

[$S \Rightarrow L=R \Rightarrow *R=R$] so follow(R) contains, =

Conflict Example2 (Reduce-Reduce)

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

I_0: $S' \rightarrow .S$

$S \rightarrow .AaAb$

$S \rightarrow .BbBa$

$A \rightarrow .$

$B \rightarrow .$

Problem

$FOLLOW(A) = \{a, b\}$

$FOLLOW(B) = \{a, b\}$

a → reduce by $A \rightarrow \varepsilon$

reduce by $B \rightarrow \varepsilon$

reduce/reduce conflict

b → reduce by $A \rightarrow \varepsilon$

reduce by $B \rightarrow \varepsilon$

reduce/reduce conflict

Constructing Canonical LR(1) Parsing Tables

- In SLR method, the state i makes a reduction by $A \rightarrow \alpha$ when the current token is a:
  - if the $A \rightarrow \alpha.$ in the I_i and a is FOLLOW(A)

- In some situations, βA cannot be followed by the terminal a in a right-sentential form when $\beta\alpha$ and the state i are on the top stack. This means that making reduction in this case is not correct.

$S \rightarrow AaAb$	$S \Rightarrow AaAb \Rightarrow Aab \Rightarrow ab$	$S \Rightarrow BbBa \Rightarrow Bba \Rightarrow ba$
$S \rightarrow BbBa$		
$A \rightarrow \varepsilon$	$Aab \Rightarrow \varepsilon ab$	$Bba \Rightarrow \varepsilon ba$
$B \rightarrow \varepsilon$	$AaAb \Rightarrow Aa \varepsilon b$	$BbBa \Rightarrow Bb \varepsilon a$

LR(1) Item

- To avoid some of invalid reductions, the states need to carry more information.
- Extra information is put into a state by including a terminal symbol as a second component in an item.
- A LR(1) item is:

$A \rightarrow \alpha.\beta,a$ where **a** is the look-head of the LR(1) item

(**a** is a terminal or end-marker.)

- Such an object is called LR(1) item.
  - 1 refers to the length of the second component
  - The lookahead has no effect in an item of the form $[A \rightarrow \alpha.\beta,a]$, where β is not $\in$.
  - But an item of the form $[A \rightarrow \alpha.,a]$ calls for a reduction by $A \rightarrow \alpha$ only if the next input symbol is a.
  - The set of such a's will be a subset of FOLLOW(A), but it could be a proper subset.

LR(1) Item (cont.)

- When β (in the LR(1) item $A \rightarrow \alpha.\beta,a$) is not empty, the look-head does not have any affect.
- When β is empty ($A \rightarrow \alpha.,a$), we do the reduction by $A \rightarrow \alpha$ only if the next input symbol is **a** (not for any terminal in FOLLOW(A)).
- A state will contain $A \rightarrow \alpha.,a_1$ where $\{a_1,...,a_n\} \subseteq$ FOLLOW(A)

...

$A \rightarrow \alpha.,a_n$

Canonical Collection of Sets of LR(1) Items

- The construction of the canonical collection of the sets of LR(1) items are similar to the construction of the canonical collection of the sets of LR(0) items, except that *closure* and *goto* operations work a little bit different.

closure(I) is: (where I is a set of LR(1) items)

- every LR(1) item in I is in closure(I)
- if $A \rightarrow \alpha . B\beta, a$ in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow .\gamma, b$ will be in the closure(I) for each terminal b in FIRST(βa) .

goto operation

- If I is a set of LR(1) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  - If $A \rightarrow \alpha.X\beta,a$ in I
    then every item in **closure({$A \rightarrow \alpha X.\beta,a$})** will be in goto(I,X).

Construction of The Canonical LR(1) Collection

- ***Algorithm***:

  C is { closure({S'→.S,$}) }

  repeat the followings until no more set of LR(1) items can be added to ***C***.

    for each I in ***C*** and each grammar symbol X

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

        add goto(I,X) to ***C***

- goto function is a DFA on the sets in C.

A Short Notation for The Sets of LR(1) Items

- A set of LR(1) items containing the following items

$$A \rightarrow \alpha . \beta, a_1$$

$$\ldots$$

$$A \rightarrow \alpha . \beta, a_n$$

can be written as

$$A \rightarrow \alpha . \beta, \ a_1/a_2/\ldots/a_n$$

An Example

1. S' → S
2. S → C C
3. C → c C
4. C → d

I_0: closure({(S' → • S, $)}) =
(S' → • S, $)
(S → • C C, $)
(C → • c C, c/d)
(C → • d, c/d)

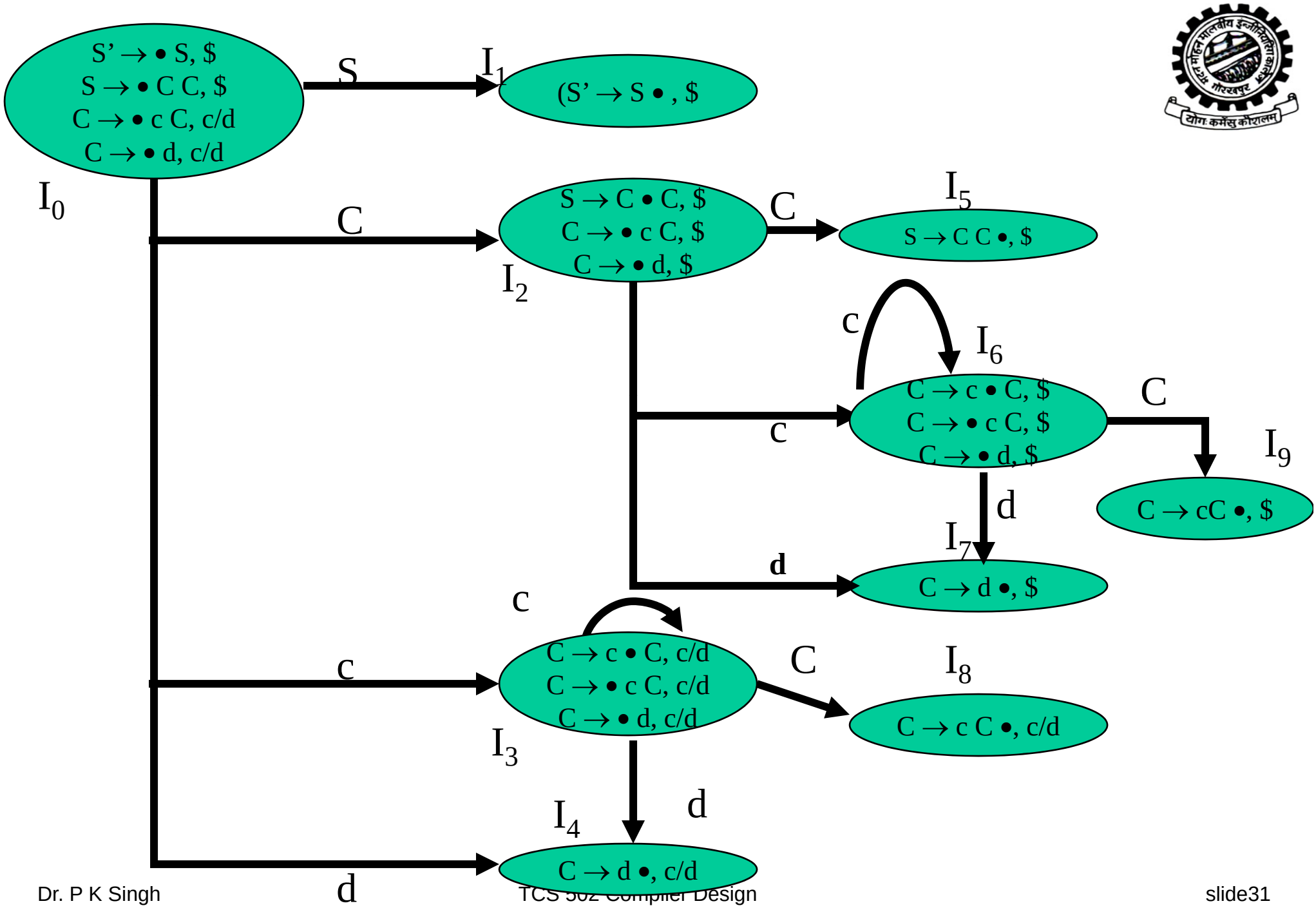
I_1: goto(I_1, S) = (S' → S • , $)

I_2: goto(I_1, C) =
(S → C • C, $)
(C → • c C, $)
(C → • d, $)

I_3: goto(I_1, c) =
(C → c • C, c/d)
(C → • c C, c/d)
(C → • d, c/d)

I_4: goto(I_1, d) =
(C → d •, c/d)

I_5: goto(I_3, C) =
(S → C C •, $)


S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I0
S
I1
(S' → S • , $
C
S → C • C, $
C → • c C, $
C → • d, $
I2
C
I5
S → C C •, $
c
I6
C → c • C, $
C → • c C, $
C → • d, $
c
C
I9
C → cC •, $
d
I7
C → d •, $
d
c
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
I3
c
C
I8
C → c C •, c/d
d
I4
C → d •, c/d
d

An Example

I_6: goto(I_3, c) =
(C → c • C, $)
(C → • c C, $)
(C → • d, $)

I_7: goto(I_3, d) =
(C → d •, $)

I_8: goto(I_4, C) =
(C → c C •, c/d)

: goto(I_4, c) = I_4

: goto(I_4, d) = I_5

I_9: goto(I_7, c) =
(C → c C •, $)

: goto(I_7, c) = I_7

: goto(I_7, d) = I_8

Construction of LR(1) Parsing Tables

1. Construct the canonical collection of sets of LR(1) items for G'.
   $C \leftarrow \{I_0, \ldots, I_n\}$
2. Create the parsing action table as follows
   - If a is a terminal, $A \rightarrow \alpha \cdot a\beta, b$ in I_i and $goto(I_i, a) = I_j$ then action[i,a] is ***shift j***.
   - If $A \rightarrow \alpha \cdot, a$ is in I_i, then action[i,a] is ***reduce*** $A \rightarrow \alpha$ where $A \neq S'$.
   - If $S' \rightarrow S \cdot, \$$ is in I_i, then action[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not LR(1).
3. Create the parsing goto table
   - for all non-terminals A, if $goto(I_i, A) = I_j$ then goto[i,A]=j
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser contains $S' \rightarrow .S, \$$

An Example

	c	d	$	S	C
0	s3	s4		g1	g2
1			a		
2	s6	s7			g5
3	s3	s4			g8
4	r3	r3			
5			r1		
6	s6	s7			g9
7			r3		
8	r2	r2			
9			r2		

The Core of LR(1) Items

- The core of a set of LR(1) Items is the set of their first components (i.e., LR(0) items)
- The core of the set of LR(1) items

$$\{ (C \rightarrow c \bullet C, c/d),$$
$$(C \rightarrow \bullet c\ C, c/d),$$
$$(C \rightarrow \bullet d, c/d) \}$$

is $\{ C \rightarrow c \bullet C,$
$C \rightarrow \bullet c\ C,$
$C \rightarrow \bullet d \}$

Canonical LR(1) Collection -- Example

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

I_0: $S' \rightarrow .S$,$

$S \rightarrow .AaAb$,$

$S \rightarrow .BbBa$,$

$A \rightarrow .$,a

$B \rightarrow .$,b

S → I_1: $S' \rightarrow S.$,$

A → I_2: $S \rightarrow A.aAb$,$ —a→ to I_4

B → I_3: $S \rightarrow B.bBa$,$ —b→ to I_5

I_4: $S \rightarrow Aa.Ab$,$ —A→ I_6: $S \rightarrow AaA.b$,$ —a→ I_8: $S \rightarrow AaAb.$,$

$A \rightarrow .$,b

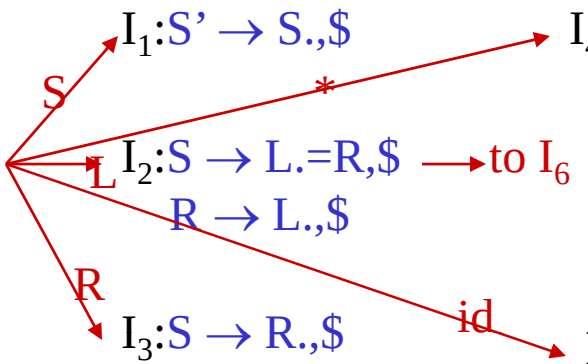
I_5: $S \rightarrow Bb.Ba$,$ —B→ I_7: $S \rightarrow BbB.a$,$ —b→ I_9: $S \rightarrow BbBa.$,$

$B \rightarrow .$,a

Canonical LR(1) Collection – Example2

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

I_0: S' → .S,$
S → .L=R,$
S → .R,$
L → .*R,$/=
L → .id,$/=
R → .L,$



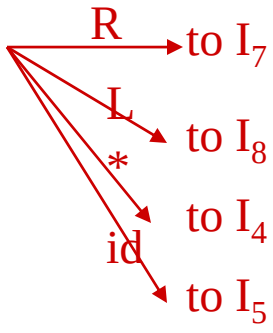
I_1: S' → S.,$

I_2: S → L.=R,$ → to I_6
R → L.,$

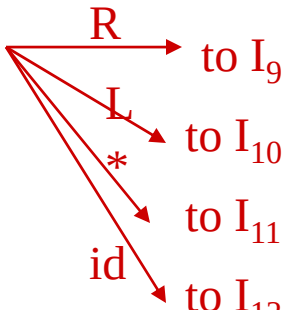
I_3: S → R.,$

I_4: L → *.R,$/=
R → .L,$/=
L→ .*R,$/=
L → .id,$/=

I_5: L → id.,$/=



I_6: S → L=.R,$
R → .L,$
L → .*R,$
L → .id,$



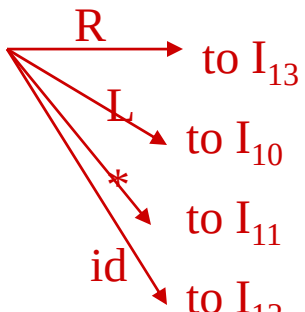
I_7: L → *R.,$/=

I_8: R → L.,$/=

I_9: S → L=R.,$

I_{10}: R → L.,$

I_{11}: L → *.R,$
R → .L,$
L→ .*R,$
L → .id,$



I_{12}: L → id.,$

I_{13}: L → *R.,$

I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}

LR(1) Parsing Tables – (for Example2)

	id	*	=	$		S	L	R
0	s5	s4				1	2	3
1				acc				
2			s6	r5				
3				r2				
4	s5	s4					8	7
5			r4	r4				
6	s12	s11					10	9
7			r3	r3				
8			r5	r5				
9				r1				
10				r5				
11	s12	s11					10	13
12				r4				
13				r3				

no shift/reduce or
no reduce/reduce conflict

so, it is a LR(1) grammar

LALR Parsing Tables

- **LALR** stands for **LookAhead LR.**
- LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.
- The number of states in SLR and LALR parsing tables for a grammar G are equal.
- But LALR parsers recognize more grammars than SLR parsers.
- ***yacc*** creates a LALR parser for the given grammar.
- A state of LALR parser will be again a set of LR(1) items.

Creating LALR Parsing Tables

Canonical LR(1) Parser ➔ LALR Parser

shrink # of states

- This shrink process may introduce a **reduce/reduce** conflict in the resulting LALR parser (so the grammar is NOT LALR)
- But, this shrink process does not produce a **shift/reduce** conflict.

The Core of A Set of LR(1) Items

- The core of a set of LR(1) items is the set of its first component.

Ex: $S \rightarrow L \blacksquare =R,\$$ ➔ $S \rightarrow L \blacksquare =R$ ← Core

$R \rightarrow L \blacksquare ,\$$ $R \rightarrow L \blacksquare$

- We will find the states (sets of LR(1) items) in a canonical LR(1) parser with same cores. Then we will merge them as a single state.

I_1: $L \rightarrow id \blacksquare ,=$ A new state: I_{12}: $L \rightarrow id \blacksquare ,=$

➔ $L \rightarrow id \blacksquare ,\$$

I_2: $L \rightarrow id \blacksquare ,\$$ have same core, merge them

- We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.
- In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.

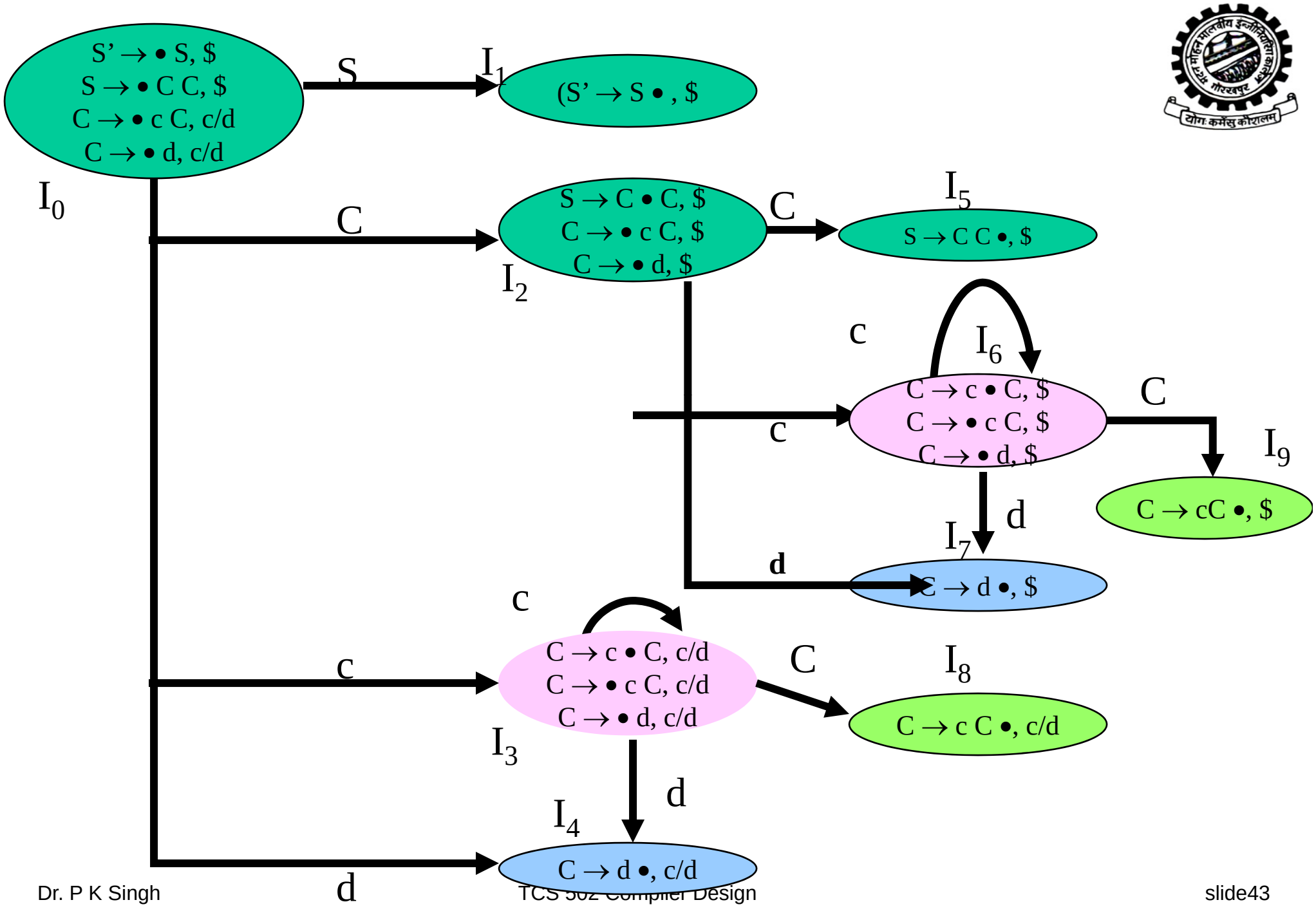
Creation of LALR Parsing Tables

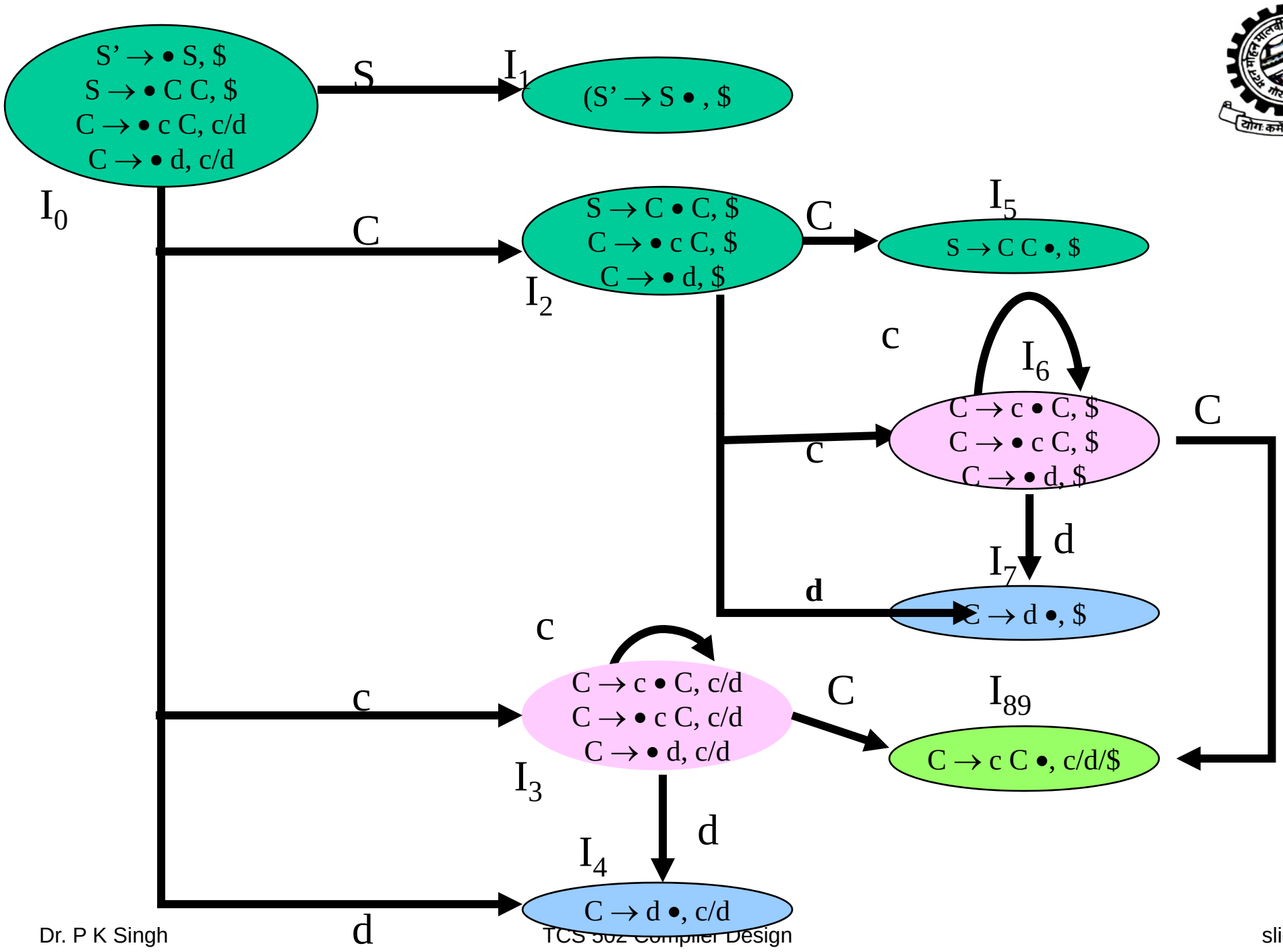
- Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar.
- For each core present; find all sets having that same core; replace those sets having same cores with a single set which is their union.

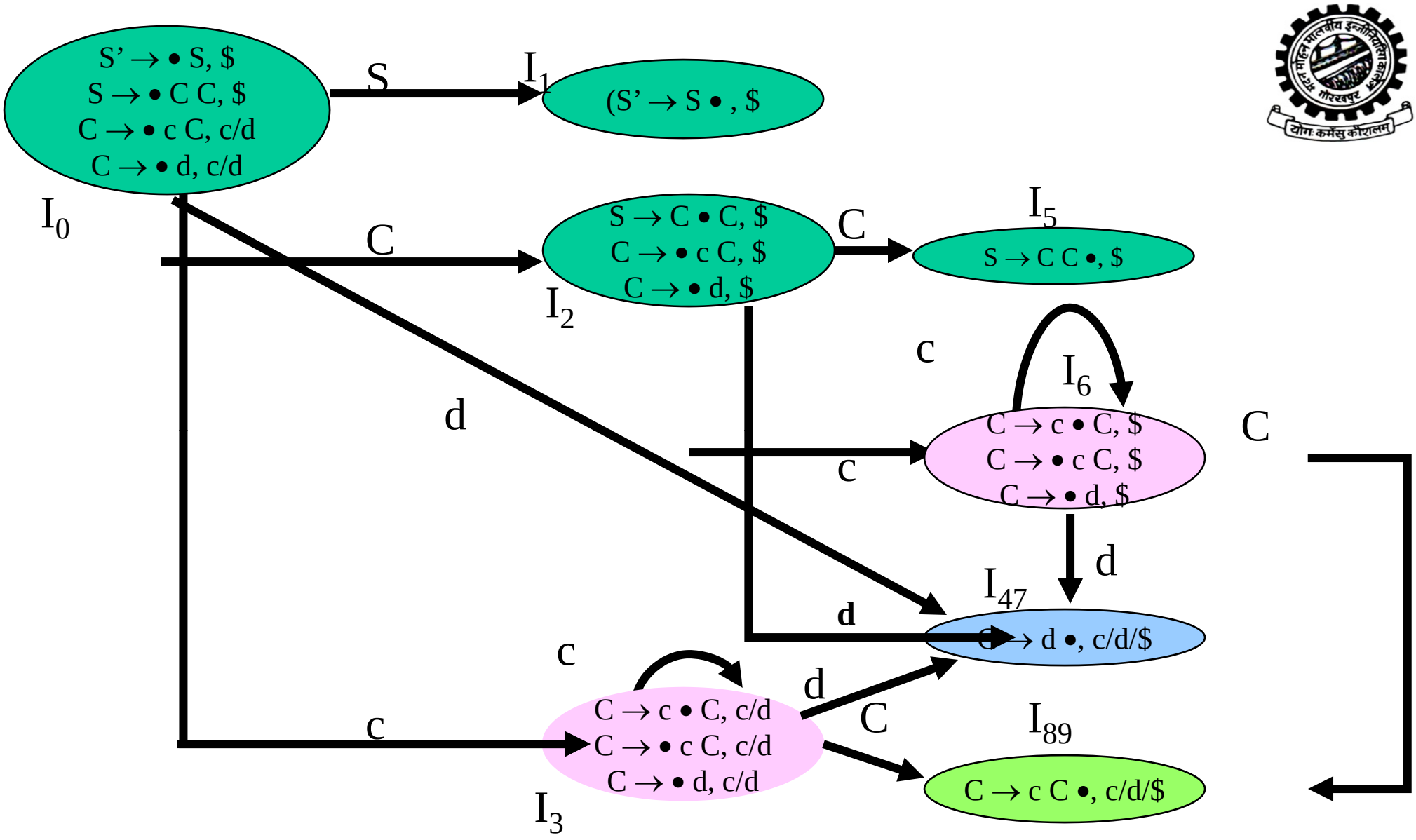
$$C=\{I_0,...,I_n\} \rightarrow C'=\{J_1,...,J_m\} \quad \text{where } m \leq n$$

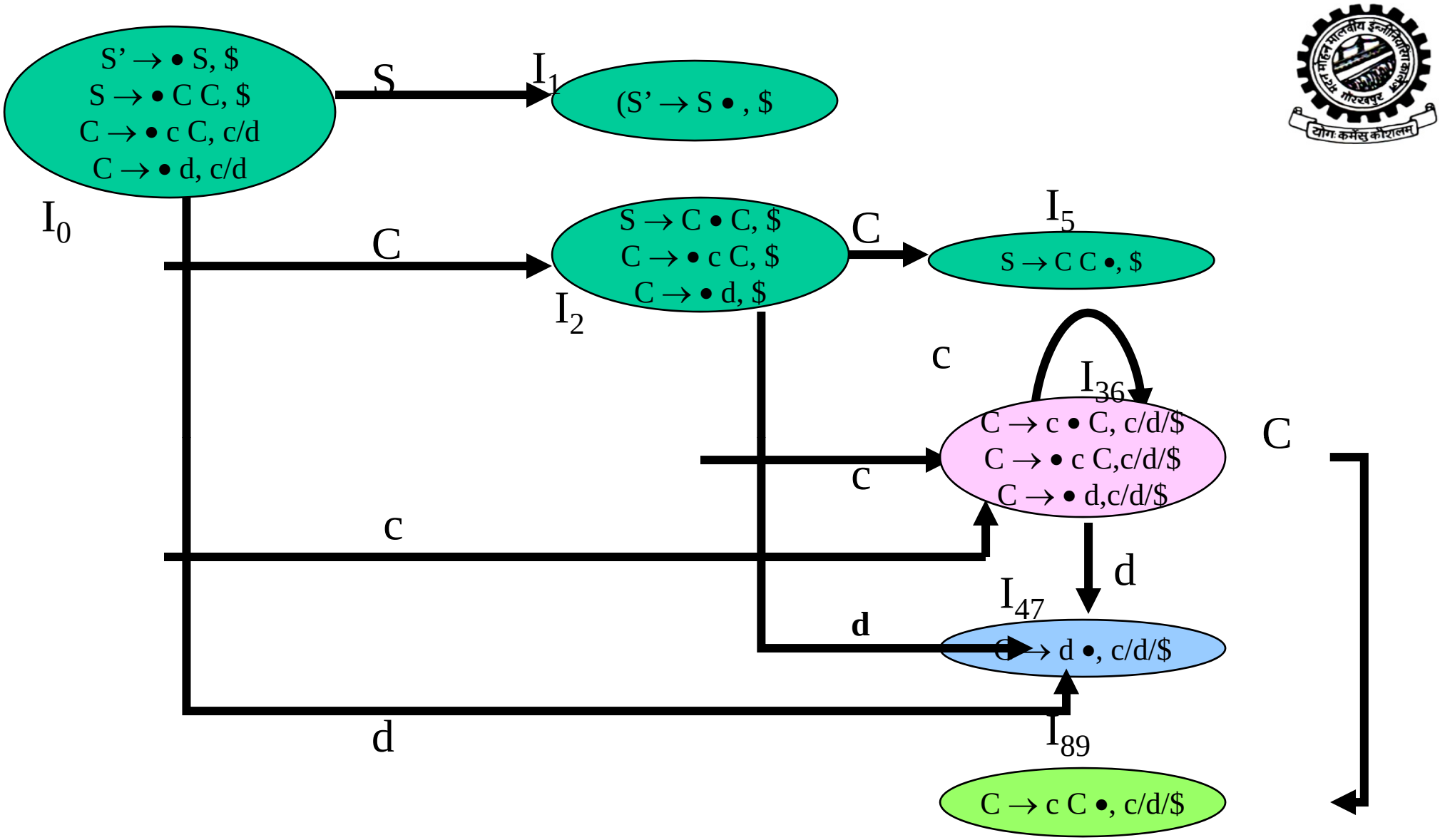
- Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.
  - ✓ Note that: If $J=I_1 \cup ... \cup I_k$ since $I_1,...,I_k$ have same cores

    ➔ cores of $goto(I_1,X),...,goto(I_2,X)$ must be same.
  - ✓ So, $goto(J,X)=K$ where K is the union of all sets of items having same cores as $goto(I_1,X)$.
- If no conflict is introduced, the grammar is LALR(1) grammar.
  (We may only introduce reduce/reduce conflicts; we cannot introduce a shift/reduce conflict)


S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I0
S
I1
(S' → S • , $
C
S → C • C, $
C → • c C, $
C → • d, $
I2
C
I5
S → C C •, $
c
I6
C → c • C, $
C → • c C, $
C → • d, $
c
C
I9
C → cC •, $
d
I7
C → d •, $
d
c
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
I3
c
C
I8
C → c C •, c/d
d
I4
C → d •, c/d
d


S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I0
S
I1
(S' → S • , $
C
S → C • C, $
C → • c C, $
C → • d, $
I2
C
I5
S → C C •, $
c
I6
C → c • C, $
C → • c C, $
C → • d, $
c
C
d
I7
C → d •, $
d
c
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
I3
c
C
I89
C → c C •, c/d/$
d
I4
C → d •, c/d
d


S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I0
S
I1
(S' → S • , $
C
S → C • C, $
C → • c C, $
C → • d, $
I2
C
I5
S → C C •, $
c
I6
C → c • C, $
C → • c C, $
C → • d, $
d
c
C
d
I47
C → d •, c/d/$
d
c
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
I3
c
d
C
I89
C → c C •, c/d/$


S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I0
S
I1
(S' → S • , $
C
S → C • C, $
C → • c C, $
C → • d, $
I2
C
I5
S → C C •, $
c
I36
C → c • C, c/d/$
C → • c C,c/d/$
C → • d,c/d/$
C
c
c
d
I47
d
C → d •, c/d/$
d
I89
C → c C •, c/d/$

LALR Parse Table

	c	d	$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Creation of LALR Parsing Tables

- Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar.
- Find each core; find all sets having that same core; replace those sets having same cores with a single set which is their union.

  $C=\{I_0,\ldots,I_n\}$ ➔ $C'=\{J_1,\ldots,J_m\}$ where $m \leq n$

- Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.
  - Note that: If $J=I_1 \cup \ldots \cup I_k$ since $I_1,\ldots,I_k$ have same cores

    ➔ cores of $goto(I_1,X),\ldots,goto(I_2,X)$ must be same.
  - So, $goto(J,X)=K$ where K is the union of all sets of items having same cores as $goto(I_1,X)$.

- If no conflict is introduced, the grammar is LALR(1) grammar. (We may only introduce reduce/reduce conflicts; we cannot introduce a shift/reduce conflict)

Shift/Reduce Conflict

- We say that we cannot introduce a shift/reduce conflict during the shrink process for the creation of the states of a LALR parser.
- Assume that we can introduce a shift/reduce conflict. In this case, a state of LALR parser must have:

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

- This means that a state of the canonical LR(1) parser must have:

  $A \rightarrow \alpha.,a$ and $B \rightarrow \beta.a\gamma,c$

  But, this state has also a shift/reduce conflict. i.e. The original canonical LR(1) parser has a conflict.

  (Reason for this, the shift operation does not depend on lookaheads)

Reduce/Reduce Conflict

- But, we may introduce a reduce/reduce conflict during the shrink process for the creation of the states of a LALR parser.

I_1 : $A \rightarrow \alpha \blacksquare , a$

$B \rightarrow \beta \blacksquare , b$

I_2: $A \rightarrow \alpha \blacksquare , b$

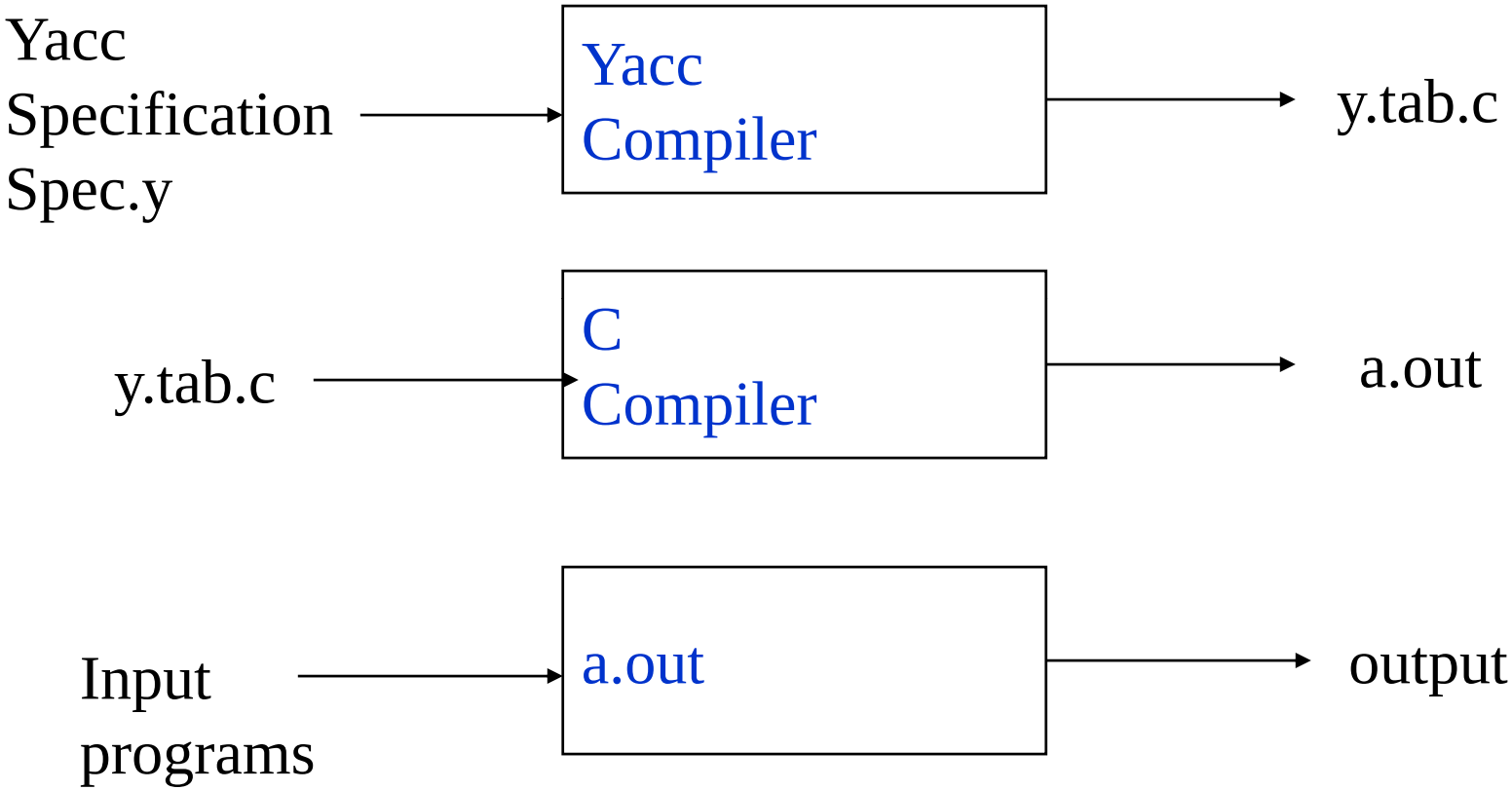
$B \rightarrow \beta \blacksquare , c$

$\Downarrow$

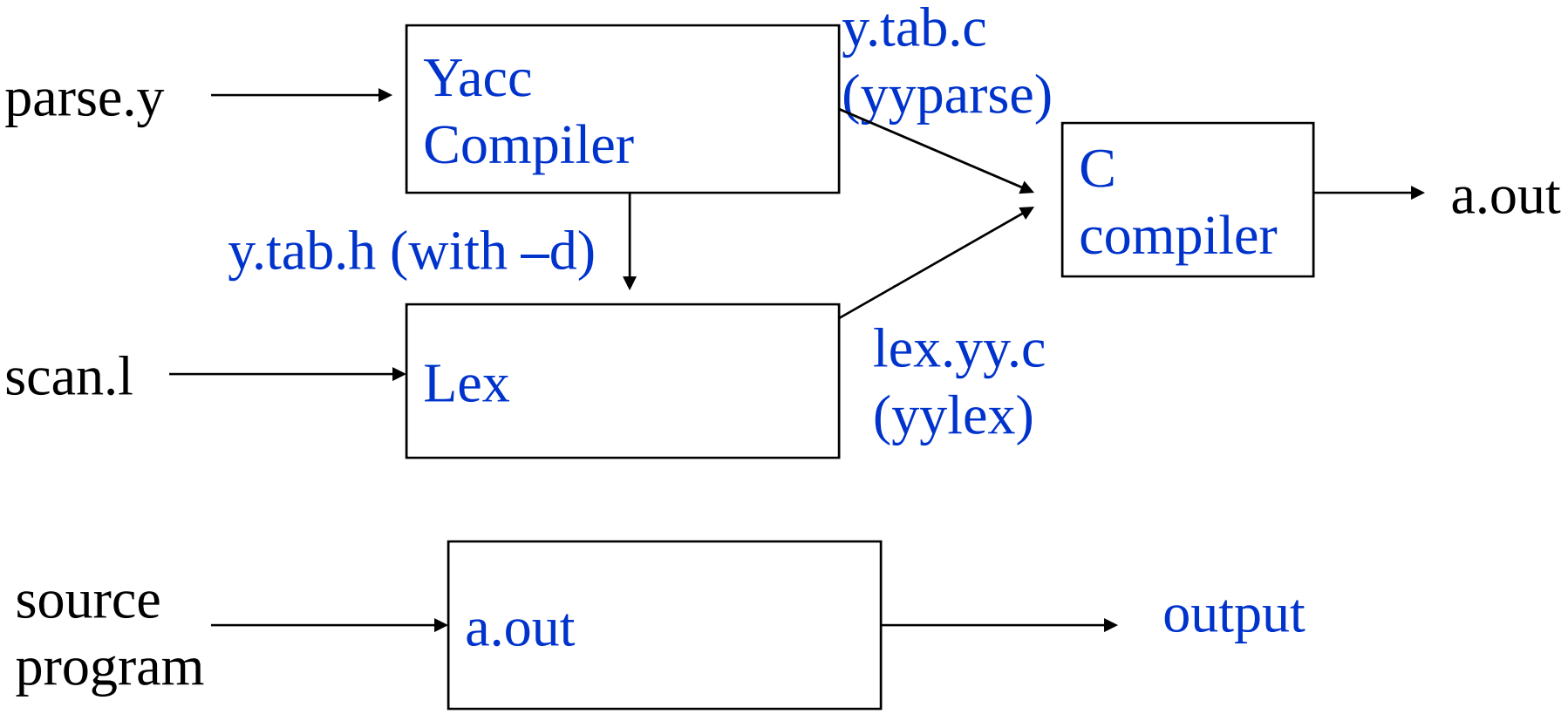
I_{12}: $A \rightarrow \alpha \blacksquare , a/b$ ➔ reduce/reduce conflict

$B \rightarrow \beta \blacksquare , b/c$

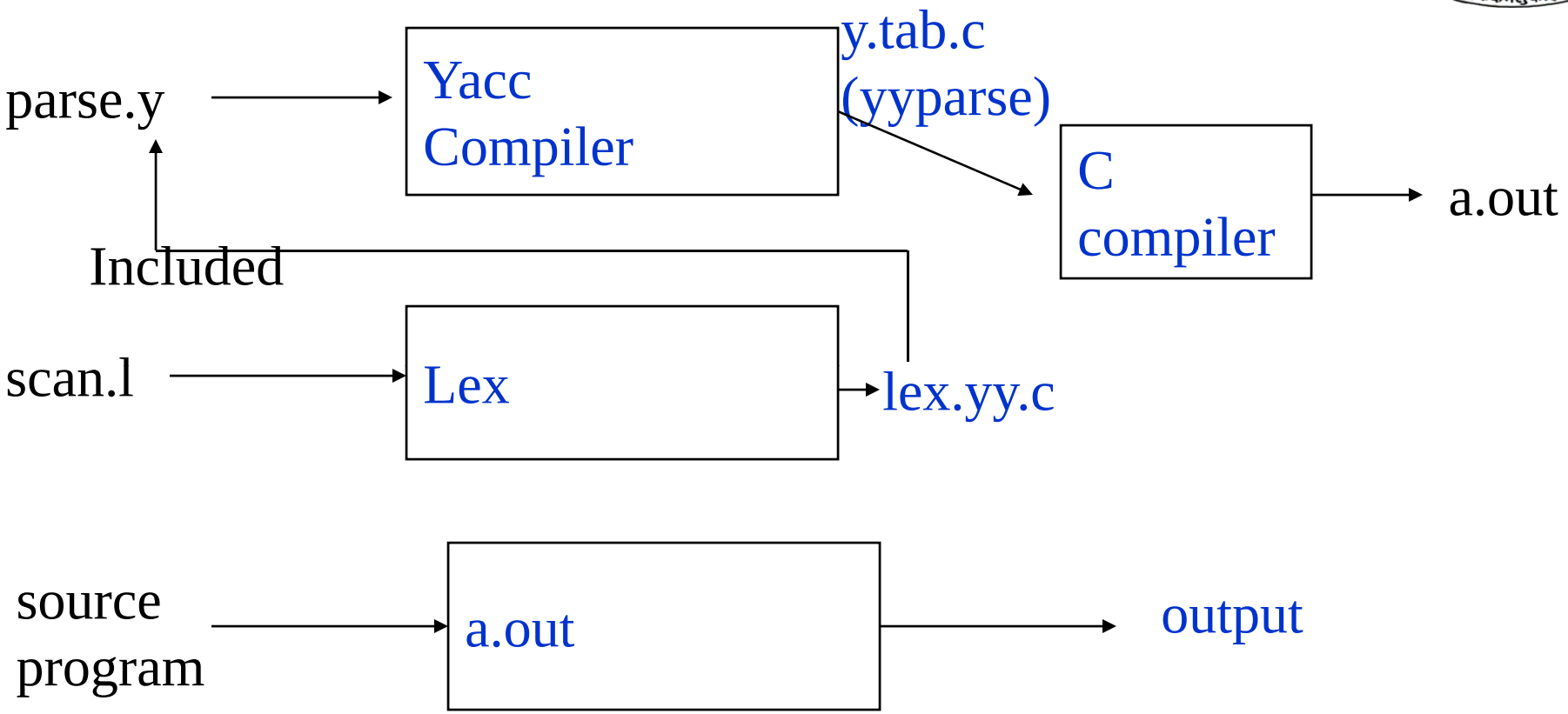
Parser Construction with YACC



Working with Lex



Working with Lex



Yacc format Overall structure

A yacc program consists of three parts:

Declarations

%% *<- Part separator*

translation rules

%%

User functions

Declarations

- As with Lex, you can include C statements in the declarations section (for example #include statements, and declarations of temporary variables that will be used in the user-routines). These should be surrounded by *%{ and %}.*
- But more importantly, *you can declare the grammar tokens, for example:*

```
%token DIGIT

%token OPERATOR

etc..
```

Translation rules

- Translation rules have the format

  Grammar Production Rule1 {semantic action1}

  Grammar Production Rule2 {semantic action2}

  ...

- For example, for the grammar rule E -> Digit OP Digit

  E : DIGIT OP DIGIT { printf("Expression!\n"); }

- The semantic value associated with a token is denoted by $X, where X is the position of the token in the Expression

  For example, $1 is the first digit's value, and $3 is the third's

  $$ is the resulting value for the non-terminal on the left of the expression

User Functions

The third part can be used to provide auxiliary user functions that the translation rules use; these will be simply copied along to the generated code

```
%%
void CreateARMHeader(void)
{
        printf("For example, I can write to a file an ARM
        program header \n");
}
```

Yacc examples

We will construct a simple calculator for evaluating arithmetic expressions

The grammar we need:

E -> E + T | T
T -> T * F | F
F -> (E) | digit

```
%token DIGIT
%%
Line : Expr '\n'          {printf("%d\n",$1); return(1);}
;
Expr : Expr '+' Term   {$$ = $1 + $3;}
| Term
;
Term : Term '*' Factor {$$ = $1 * $3}
| Factor
;
Factor : '(' expr ')'      {$$ = $2;}
: DIGIT
;
```

Yacc examples

```
%{
#include <ctype.h>
%}
%token DIGIT
%%
Line        : Expr '\n'      {printf("%d\n",$1); return(1);}
            ;
Expr        : Expr '+' Term              {$$ = $1 + $3;}
            | Term
            ;
Term        : Term '*' Factor            {$$ = $1 * $3}
            | Factor
            ;
Factor      : '(' expr ')'   {$$ = $2;}
            : DIGIT
            ;
%%
yylex() {
            int c;
            c= getchar();
            if(isdigit(c)){
                        yylval=c-'0';
                        return DIGIT;
            }
            return c;
}
```

Yacc examples(ambiguous Grammar)

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double
%}
%token NUM
%left '+' '-'
%left '*' '/'
%%
line        : line Expr '\n'        {printf("\n\t Answer = %g\n",$2); }
            | line '\n'
            |
            ;
Expr        : Expr '+' Expr         {$$ = $1 + $3;}
            | Expr '-' Expr         {$$ = $1 - $3;}
            | Expr '/' Expr         {$$ = $1 / $3;}
            | Expr '*' Expr         {$$ = $1 * $3;}
            |'(' expr ')'           {$$ = $2;}
            |NUM
            ;
%%
yylex() {
            int c;
            while((c= getchar()==' ');
            if((c=='.'||(isdigit(c))){
                        ungetc(c,stdin);
                        scanf("%lf",&yylval);
                        return NUM;
            }
            return c;
}
```

yylex() (Through Lex) calc.l

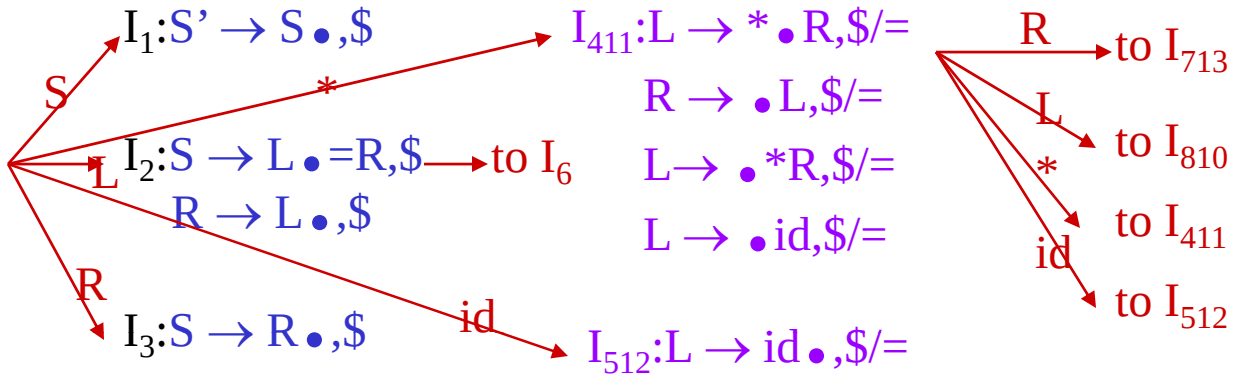
```
NUM         [0-9]+\.?| [0-9]*\.[0-9]+
%%
[ ]
{NUM}       {sscanf(yytext,"%lf", &yylval);
               return NUM;}
\n|.        {return yytext[0];}
```

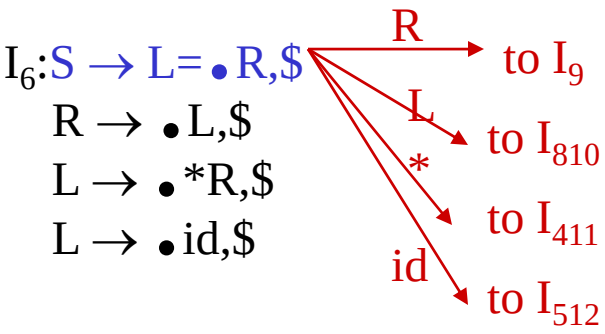
Yacc examples(with Lex)

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double
%}
%token NUM
%left '+' '-'
%left '*' '/'
%%
line    : line Expr '\n'    {printf("\n\t Answer = %g\n",$2); }
        | line '\n'
        |
        ;
Expr    : Expr '+' Expr     {$$ = $1 + $3;}
        | Expr '-' Expr     {$$ = $1 - $3;}
        | Expr '/' Expr             {$$ = $1 / $3;}
        | Expr '*' Expr     {$$ = $1 * $3;}
        |'(' expr ')'               {$$ = $2;}
        |NUM
        ;
%%
#include "lex.yy.c"
```

Canonical LALR(1) Collection – Example2

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





I9:S → L=R•,$

I713:L → *R•,$/=

I810: R → L•,$/=

Same Cores

I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}

LALR(1) Parsing Tables – (for Example2)

	id	*	=	$	S	L	R
0	s5	s4			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s12	s11				10	9
7			r3	r3			
8			r5	r5			
9				r1			

no shift/reduce or
no reduce/reduce conflict

so, it is a LALR(1) grammar

Using Ambiguous Grammars

- All grammars used in the construction of LR-parsing tables must be un-ambiguous.
- Can we create LR-parsing tables for ambiguous grammars ?
  - Yes, but they will have conflicts.
  - We can resolve these conflicts in favor of one of them to disambiguate the grammar.
  - At the end, we will have again an unambiguous grammar.
- Why we want to use an ambiguous grammar?
  - Some of the ambiguous grammars are **much natural**, and a corresponding unambiguous grammar can be very complex.
  - Usage of an ambiguous grammar may **eliminate unnecessary reductions**.
- Ex.

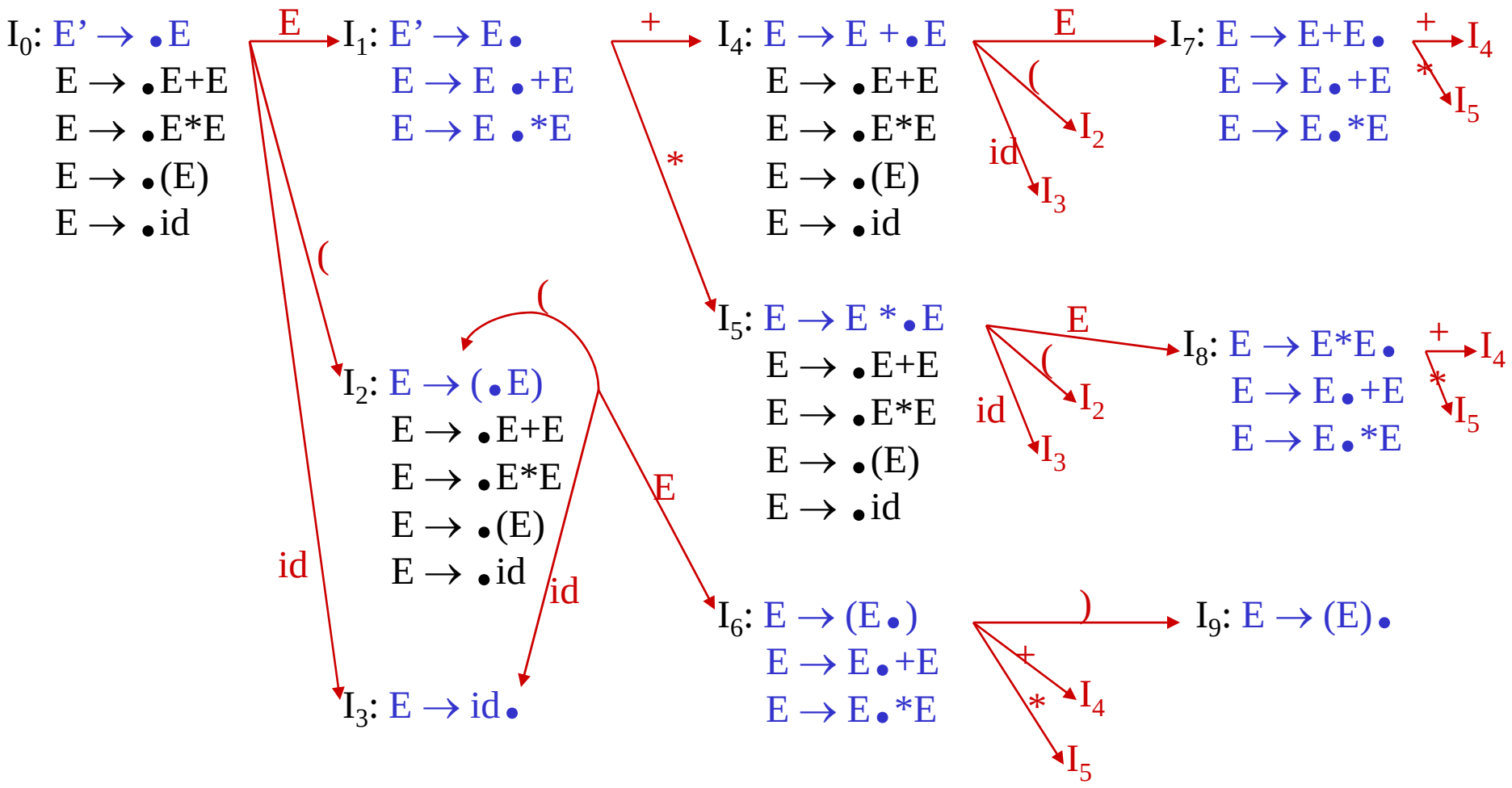
$E \rightarrow E+E \mid E*E \mid (E) \mid id$ ➔

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Sets of LR(0) Items for Ambiguous Grammar



SLR-Parsing Tables for Ambiguous Grammar

FOLLOW(E) = { $, +, *,) }

State I_7 has shift/reduce conflicts for symbols + and *.

$$I_0 \xrightarrow{E} I_1 \xrightarrow{+} I_4 \xrightarrow{E} I_7$$

when current token is +

shift ➔ + is right-associative

reduce ➔ + is left-associative

when current token is *

shift ➔ * has higher precedence than +

reduce ➔ + has higher precedence than *

SLR-Parsing Tables for Ambiguous Grammar

FOLLOW(E) = { $, +, *,) }

State I_8 has shift/reduce conflicts for symbols + and *.

$$I_0 \xrightarrow{E} I_1 \xrightarrow{*} I_5 \xrightarrow{E} I_7$$

when current token is *

shift ➔ * is right-associative

reduce ➔ * is left-associative

when current token is +

shift ➔ + has higher precedence than *

reduce ➔ * has higher precedence than +

SLR-Parsing Tables for Ambiguous Grammar

	Action							Goto
	id	**+**	*****	**(**	**)**	**$**		**E**
0	s3			s2				1
1		s4	s5			acc		
2	s3			s2				6
3		r4	r4		r4	r4		
4	s3			s2				7
5	s3			s2				8
6		s4	s5		s9			
7		r1	s5		r1	r1		
8		r2	r2		r2	r2		
9		r3	r3		r3	r3		

Error Recovery in LR Parsing

- An LR parser will detect an error when it consults the parsing action table and finds an error entry. All empty entries in the action table are error entries.
- Errors are never detected by consulting the goto table.
- An LR parser will announce error as soon as there is no valid continuation for the scanned portion of the input.
- A canonical LR parser (LR(1) parser) will never make even a single reduction before announcing an error.
- The SLR and LALR parsers may make several reductions before announcing an error.
- But, all LR parsers (LR(1), LALR and SLR parsers) will never shift an erroneous input symbol onto the stack.

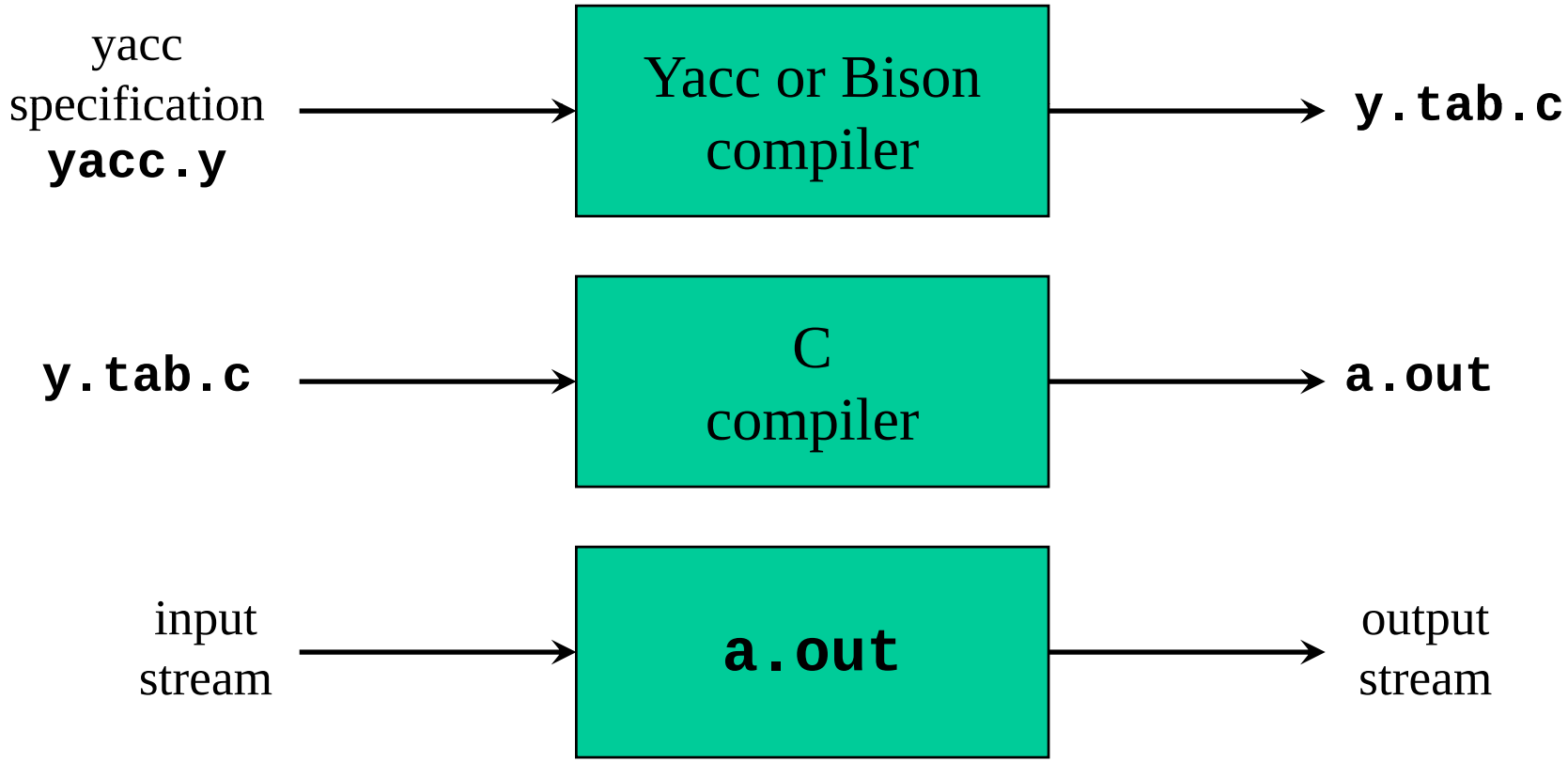
Panic Mode Error Recovery in LR Parsing

- Scan down the stack until a state **s** with a goto on a particular nonterminal **A** is found. (Get rid of everything from the stack before this state s).
- Discard zero or more input symbols until a symbol **a** is found that can legitimately follow A.
  - The symbol a is simply in FOLLOW(A), but this may not work for all situations.
- The parser stacks the nonterminal **A** and the state **goto[s,A]**, and it resumes the normal parsing.
- This nonterminal A is normally is a basic programming block (there can be more than one choice for A).
  - stmt, expr, block, ...

Phrase-Level Error Recovery in LR Parsing

- Each empty entry in the action table is marked with a specific error routine.
- An error routine reflects the error that the user most likely will make in that case.
- An error routine inserts the symbols into the stack or the input (or it deletes the symbols from the stack and the input, or it can do both insertion and deletion).
  - missing operand
  - unbalanced right parenthesis

Creating an LALR(1) Parser with Yacc/Bison



Yacc Specification

- A *yacc specification* consists of three parts:
  yacc declarations, and C declarations within **%{ %}**
  %%
  translation rules
  %%
  user-defined auxiliary procedures
- The *translation rules* are productions with actions:
  $production_1$ { $semantic\ action_1$ }
  $production_2$ { $semantic\ action_2$ }
  ...
  $production_n$ { $semantic\ action_n$ }

Writing a Grammar in Yacc

- Productions in Yacc are of the form
  Nonterminal **:** tokens/nonterminals { *action* }
  | tokens/nonterminals { *action* }
  ...
  ;
- Tokens that are single characters can be used directly within productions, e.g. **'+'**
- Named tokens must be declared first in the declaration part using
  %token *TokenName*

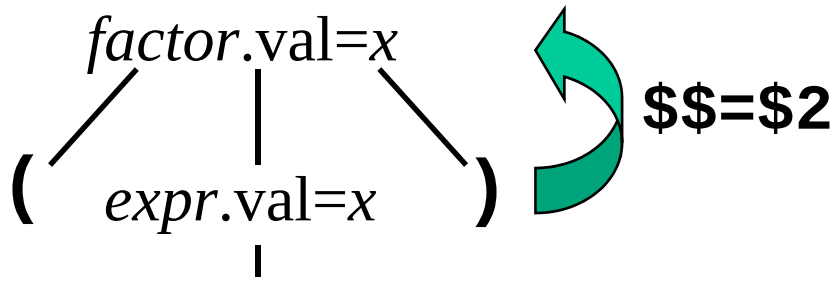
Synthesized Attributes

- Semantic actions may refer to values of the *synthesized attributes* of terminals and nonterminals in a production:

  $X : Y_1\ Y_2\ Y_3\ \ldots\ Y_n \qquad \{\ action\ \}$

  - **$$** refers to the value of the attribute of X
  - **$***i*** refers to the value of the attribute of Y_i
- For example

  `factor : '(' expr ')' { $$=$2; }`



Example 1

```
%{ #include <ctype.h> %}
%token DIGIT
%%
line    : expr '\n'             { printf("%d\n", $1); }
        ;
expr    : expr '+' term         { $$ = $1 + $3; }
        | term                  { $$ = $1; }
        ;
term    : term '*' factor       { $$ = $1 * $3; }
        | factor                { $$ = $1; }
        ;
factor  : '(' expr ')'          { $$ = $2; }
        | DIGIT                 { $$ = $1; }
        ;
%%
int yylex()
{ int c = getchar();
  if (isdigit(c))
  { yylval = c-'0';
    return DIGIT;
  }
  return c;
}
```

Also results in definition of **`#define DIGIT xxx`**

Attribute of **`factor`** (child)

Attribute of **`term`** (parent)

Attribute of token (stored in **`yylval`**)

Example of a very crude lexical analyzer invoked by the parser

Dealing With Ambiguous Grammars

- By defining operator precedence levels and left/right associativity of the operators, we can specify ambiguous grammars in Yacc, such as $E \rightarrow E$ **+** E | E **-** E | E ***** E | E **/** E | **(** E **)** | **-** E | **num**
- To define precedence levels and associativity in Yacc's declaration part:

```
%left '+' '-'
%left '*' '/'
%right UMINUS
```

Example 2

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
lines   : lines expr '\n'       { printf("%g\n", $2); }
        | lines '\n'
        | /* empty */
        ;
expr    : expr '+' expr         { $$ = $1 + $3; }
        | expr '-' expr         { $$ = $1 - $3; }
        | expr '*' expr         { $$ = $1 * $3; }
        | expr '/' expr         { $$ = $1 / $3; }
        | '(' expr ')'          { $$ = $2; }
        | '-' expr %prec UMINUS { $$ = -$2; }
        | NUMBER
        ;
%%
```

Double type for attributes and **`yylval`**

Example 2 (cont'd)

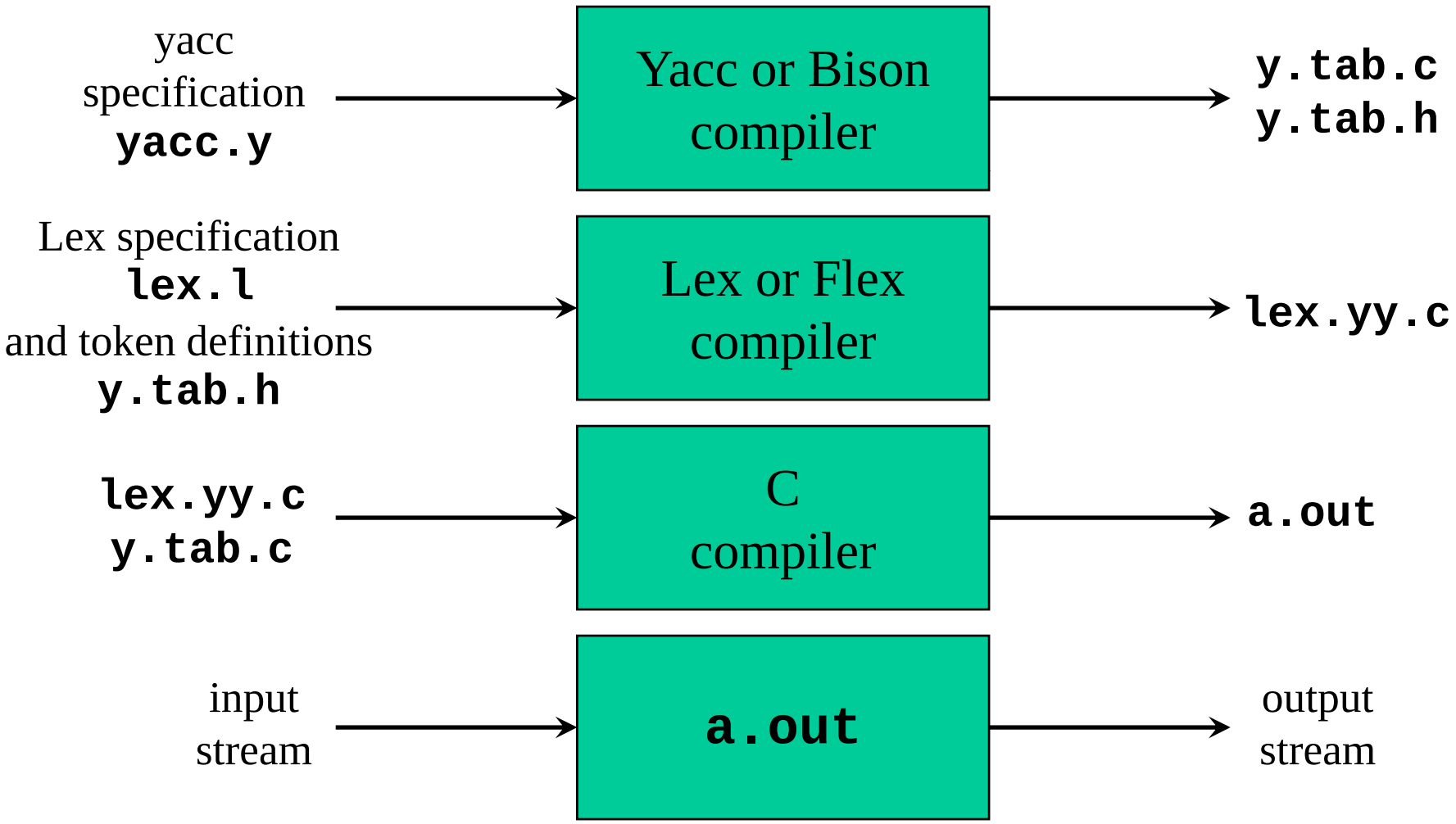
```
%%
int yylex()
{ int c;
  while ((c = getchar()) == ' ')
    ;
  if ((c == '.') || isdigit(c))
  { ungetc(c, stdin);
    scanf("%lf", &yylval);
    return NUMBER;
  }
  return c;
}
int main()
{ if (yyparse() != 0)
    fprintf(stderr, "Abnormal exit\n");
  return 0;
}
int yyerror(char *s)
{ fprintf(stderr, "Error: %s\n", s);
}
```

Crude lexical analyzer for fp doubles and arithmetic operators

Run the parser

Invoked by parser to report parse errors

Combining Lex/Flex with Yacc/Bison



Lex Specification for Example 2

```
%option noyywrap
%{
#include "y.tab.h"

extern double yylval;
%}
number  [0-9]+\.?|[0-9]*\.[0-9]+
%%
[ ]             { /* skip blanks */ }
{number}        { sscanf(yytext, "%lf", &yylval);
                  return NUMBER;
                }
\n|.            { return yytext[0]; }
```

Generated by Yacc, contains
#define NUMBER xxx

Defined in **y.tab.c**

```
yacc -d example2.y
lex example2.l
gcc y.tab.c lex.yy.c
./a.out
```

```
bison -d -y example2.y
flex example2.l
gcc y.tab.c lex.yy.c
./a.out
```