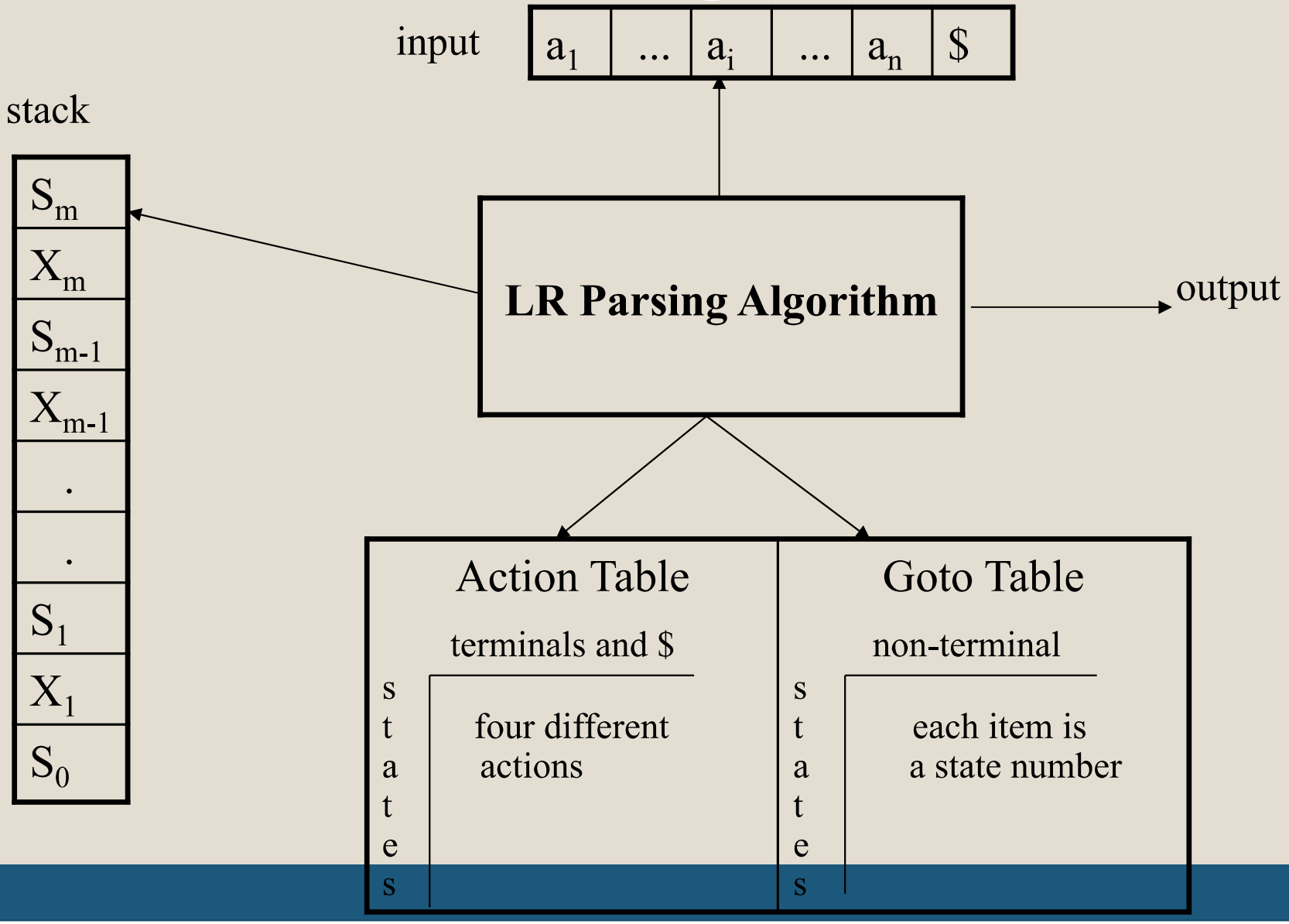
Outline

- LR Parsing algorithm
- LR(1) Parsing
- Constructing LR(1) DFA
- Constructing LR(1) Parsing table
- LALR(1) parsing
- Constructing LALR(1) DFA
- Constructing LALR(1) Parsing table
- Shift/reduce and reduce/reduce conflicts
- Ambiguous Grammars
- SLR(1) Parsing with Ambiguity
- Error Recovery in LR Parsing

LR Parsing Algorithm



LR(1) Parsing

- Substantially more powerful than the other methods we've covered so far.
- Tries to more intelligently find handles by using a lookahead token at each step.
- 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.
- LR(1) item **$A \rightarrow \alpha.\beta$, a**

LR(1) = LR(0) item + one look ahead terminal

LR(1) Parsing(cont.)

- LR(1) item $A \rightarrow \alpha.\beta\ ,\ a$
- 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)**).
- Note:

  the following LR(1) items like (a) can be written as (b) .

$S \rightarrow S\ .\ +\ E$	+
$S \rightarrow S\ .\ +\ E$	$
$S \rightarrow S\ +\ .\ E$	num

a

$S \rightarrow S\ .\ +\ E$	[+,$] or [+/$]
$S \rightarrow S\ +\ .\ E$	num

b

CLR(1) Parsing

- 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.
- **Rules for look-ahead sets:**

  1) The initial item set I_0 **contains only one token, the end marker ($).**

  2) other item set: closure(I)

  Given $A \rightarrow \alpha . B\beta , a$ 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)** .

- **Ex:**

$S \rightarrow AaAb$
$S \rightarrow BbBa$
$A \rightarrow \varepsilon$
$B \rightarrow \varepsilon$

$I_0 = \{[S' \rightarrow .S , \$],$
$[S \rightarrow .AaAb, \$],[S \rightarrow .BbBa, \$]$
$[A \rightarrow ., a] , [B \rightarrow . , b]$

CLR(1) Example1

- Consider the following grammar

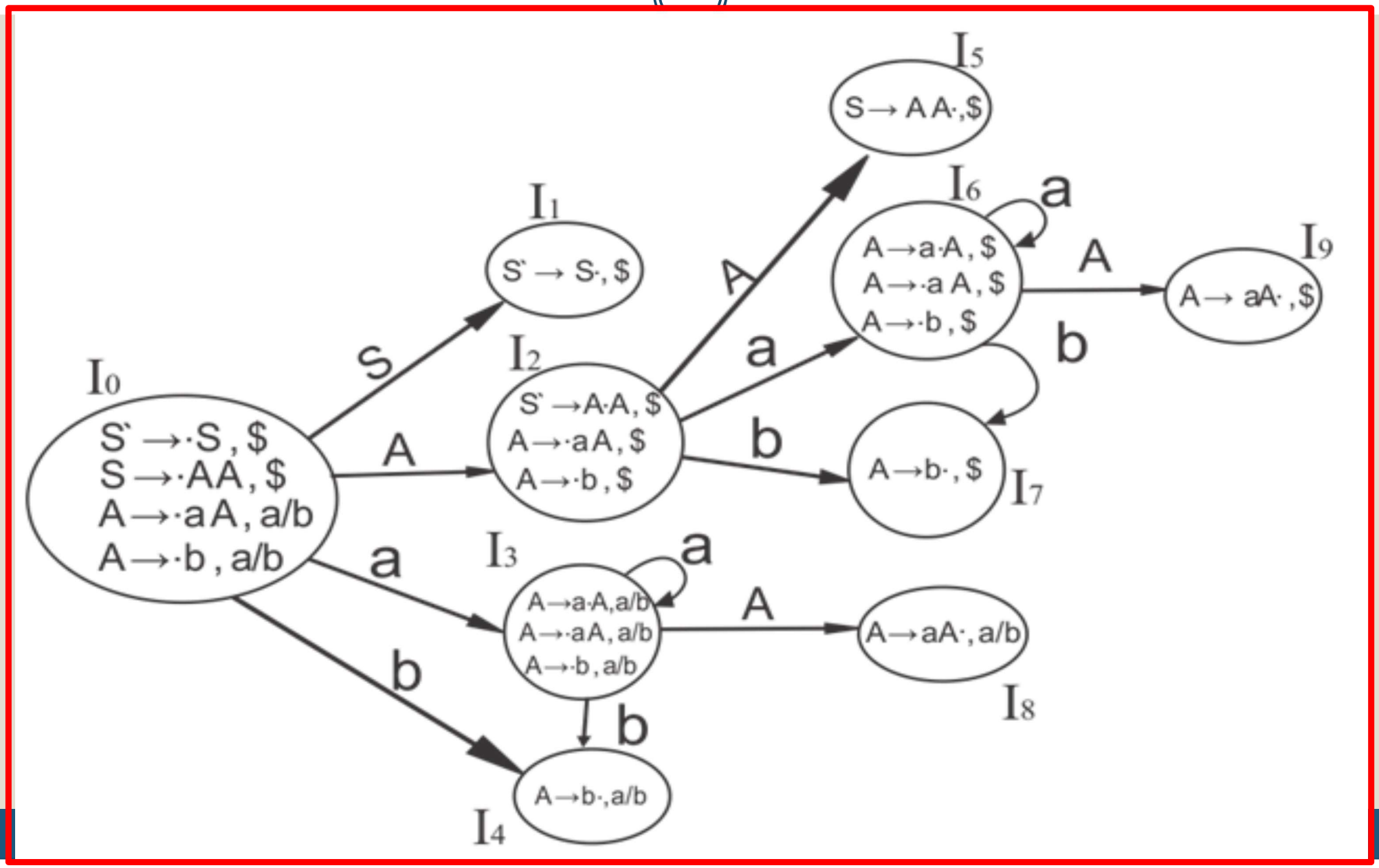
$$S \rightarrow AA$$
$$A \rightarrow aA$$
$$A \rightarrow b$$

First(A) = { a , b}
First(S) = { a , b}
Follow(S) = { $ }
Follow(A) = { a , b , $}

- Add Augment Production, insert '•' symbol at the first position for every production in G and also add the lookahead.
- Compute closure(I) where **I={[S' → .S, $]}**

$$S` \rightarrow \bullet S, \$$$
$$S \rightarrow \bullet AA, \$$$
$$A \rightarrow \bullet aA, a/b$$
$$A \rightarrow \bullet b, a/b$$

CLR(1) DFA


I0
S` →·S , $
S →·AA , $
A →·aA , a/b
A →·b , a/b
S
A
a
b
I1
S` → S·, $
I2
S` →A·A, $
A →·aA , $
A →·b , $
I3
A →a·A,a/b
A →·aA , a/b
A →·b , a/b
I4
A →b·,a/b
I5
S → AA·,$
I6
A →a·A , $
A →·a A , $
A →·b , $
I7
A →b·, $
I8
A →aA·, a/b
I9
A → aA· ,$

Construction of CLR(1) Parsing Tables

1. Construct **C→ {I_0,...,I_n}**, the canonical collection of sets of **LR(1) items** for G'.
2. **State (i)** is constructed from **I_i** . The parsing action table determined as follows:
   - If **[A→ α.aβ , b]** is in **I_i** and **goto(I_i,a)=I_j** then action[**i,a]** is **shift j**. a must be a terminal.
   - If **[A→α. , a]** is in **I_i** , then **action[i,a]** is **reduce A→α** where **A≠S'.**
   - If **[S'→S. , $]** is in **Ii** , then **action[i,$]** is **accept**.
   - If any **conflicting actions result** from the above rules, the grammar is **not LR(1).**
3. The Goto transitions are constructed for all **non-terminals A** using the rule: if **goto(I_i,A)=I_j** then **goto[i,A]=j**
4. All entries not defined by rule (2) and (3) are made **error**.
5. **Initial state** of the parser contains **[S'→.S,$]**

CLR(1) Parsing table

S → AA ... (1)

A → aA ....(2)

A → b ... (3)

	Action Table			Goto Table	
State	a	b	$	S	A
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		
8	R2	R2			
9			R2		

CLR(1) Example2

- Consider the following grammar

  S'→ S

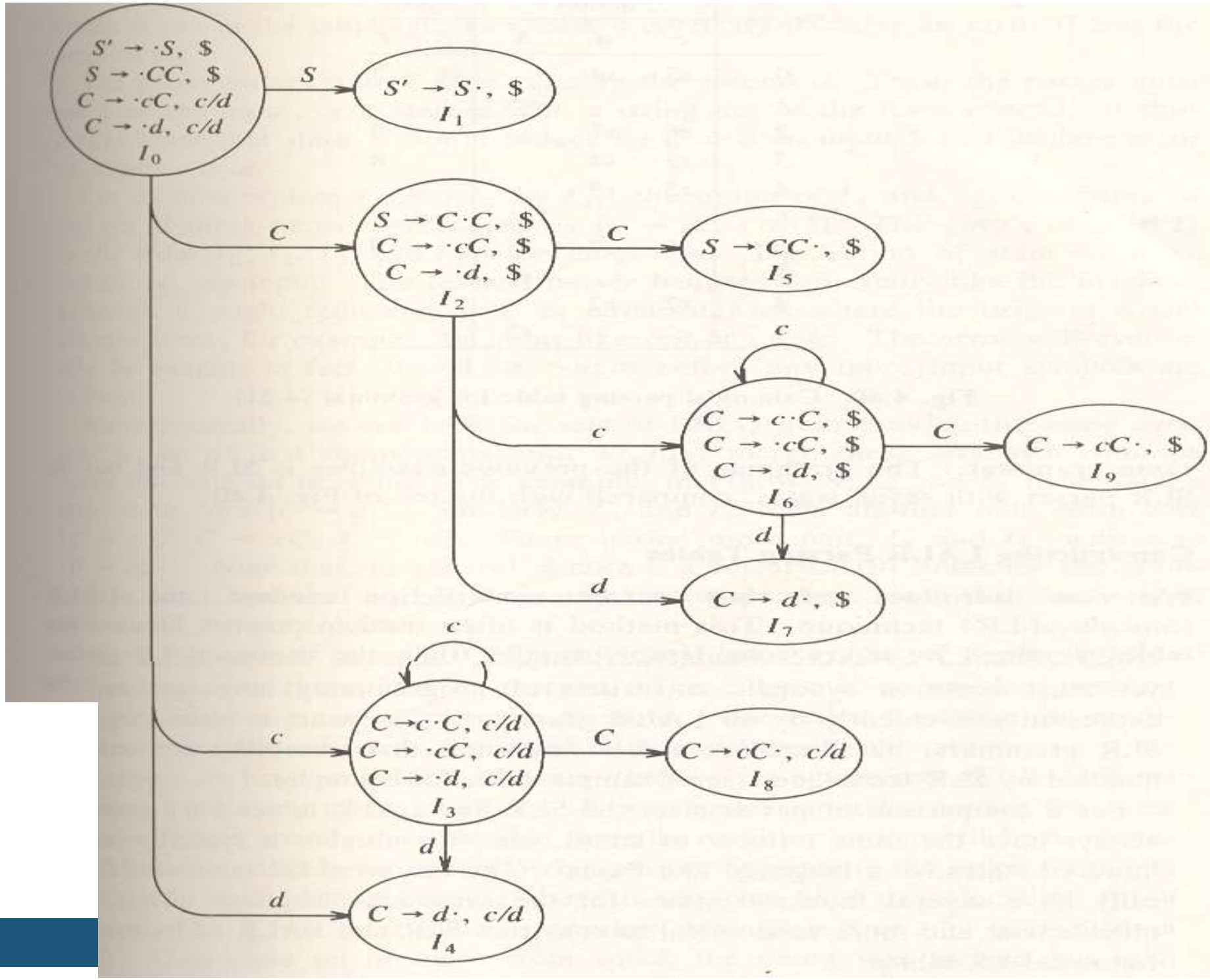
  S → CC

  C → cC | d

First(C) = {c, d}
First(S) = {c, d}
Follow(S) = {$}
Follow(C) = {c,d,$}

- Compute closure(I) where I={[S' → .S, $]}

S' → .S , $
S → .CC , $
C → .cC , c,d
C → .d , c,d


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

CLR(1) Parse table (Example2)

	Action Table			Goto Table	
State	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

1. **S → .CC , $**
2. **C → .cC , c,d**
3. **C → .d , c,d**

no shift/reduce or
no reduce/reduce conflict

so, it is a CLR(1) grammar

Home Work

- Construct an CLR(1) parsing DFA design and Parsing table for the following grammar :

S'→• S

S → • L=R

S → • R

L → • *R

L → • id

R → • L

LALR Parsing

- LALR stands for **LookAhead** LR technique.
- LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.
- The SLR and LALR tables for a grammar always have the same number of states.
- There are a few constructs that can not be easily handled by LALR parsers technique.
- A state of LALR parser will be again a set of CLR(1) items.

LALR Parsing

- Generally, we can look for sets of LR(1) items having the same core, that is, set of first components, we may merge these sets with common core into one set of items.
- Example: from previous figure Example(2) slide 12 .

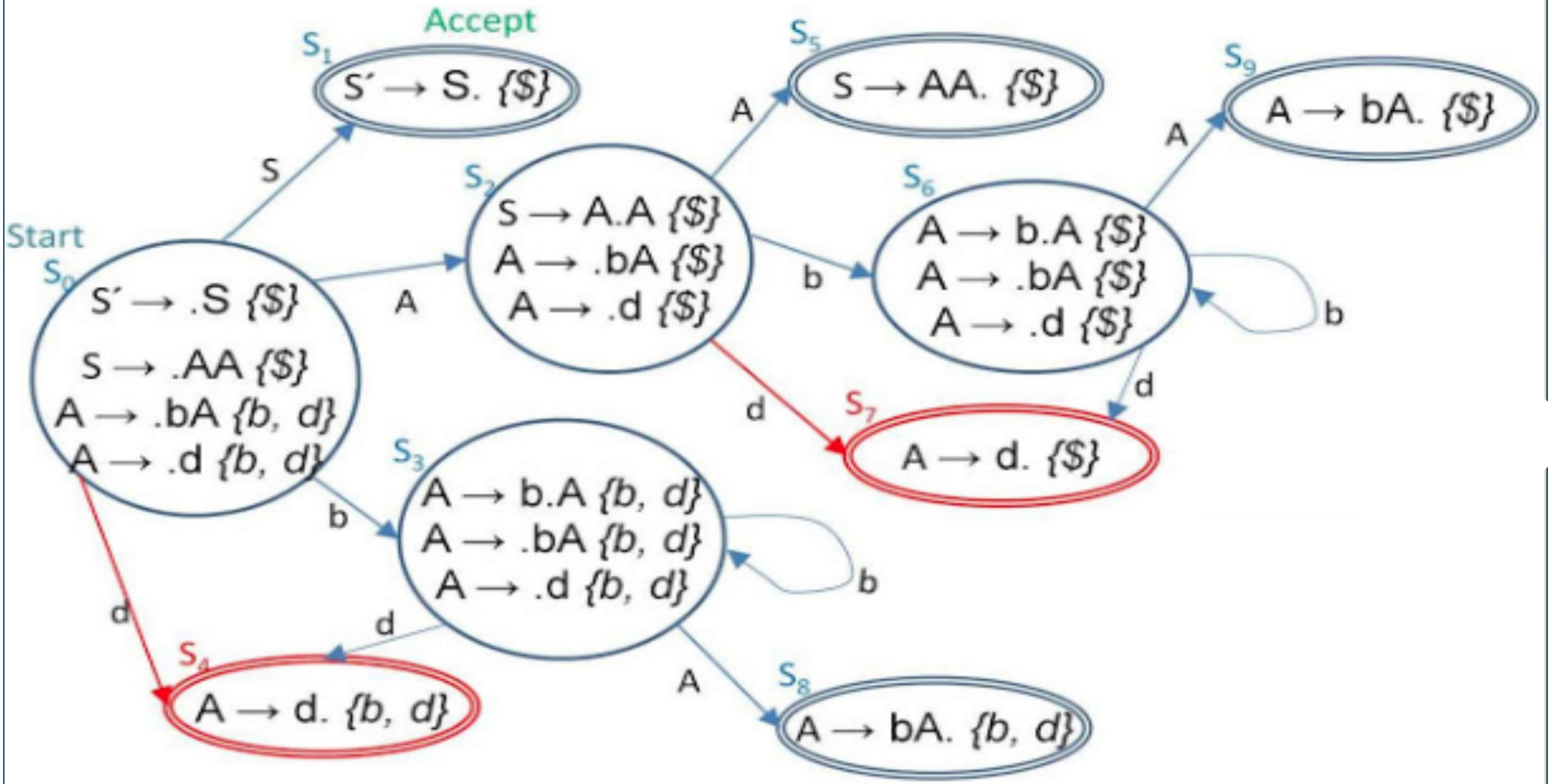
I_4: C → d., c/d
I_7: C → d., $

C → d. → Have the same core

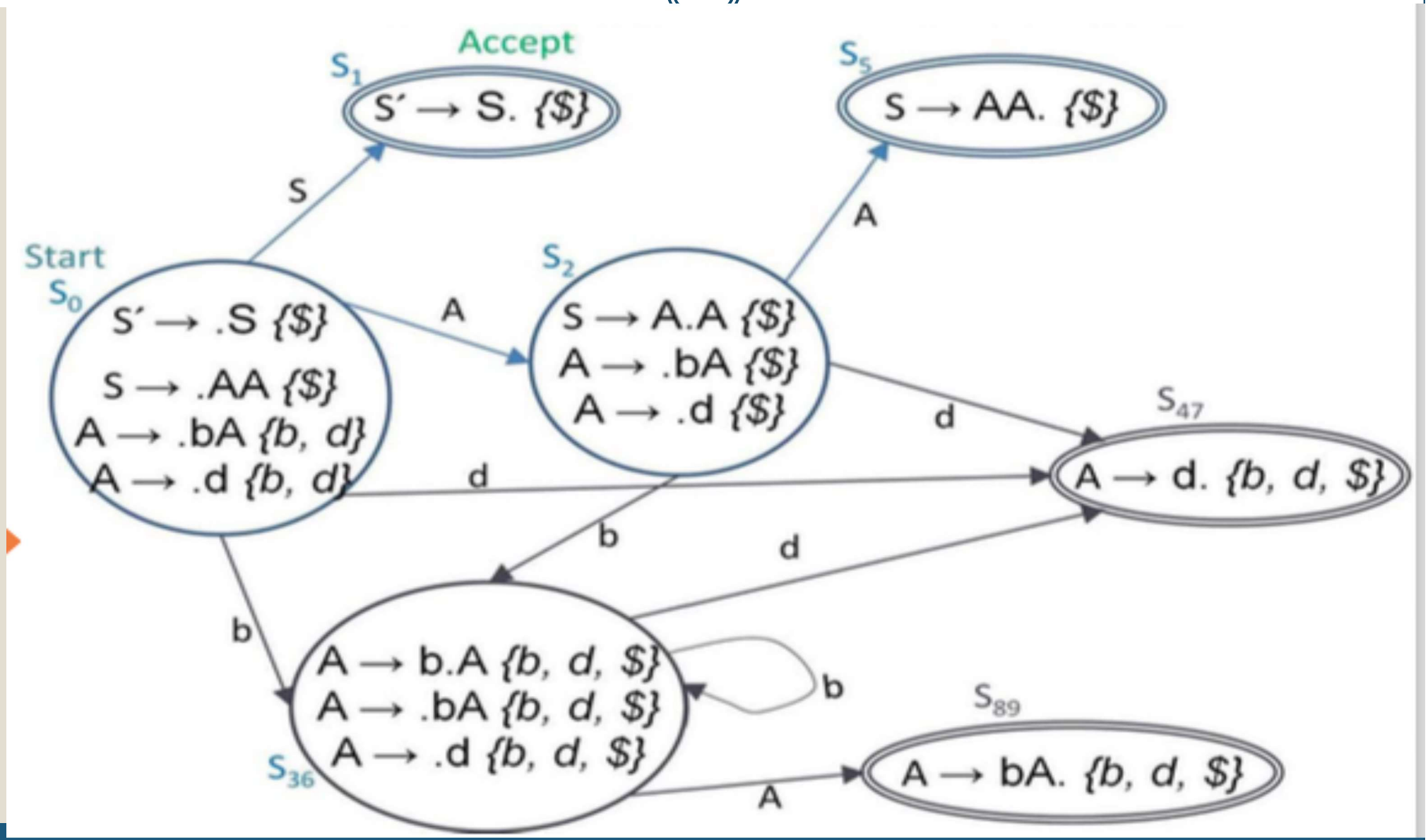
I_{47}: C → d., c/d/$

- We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.

CLR(1) Parsing



LALR(1) Parsing



LALR Parsing table

G:

1. S → AA

2. A → bA

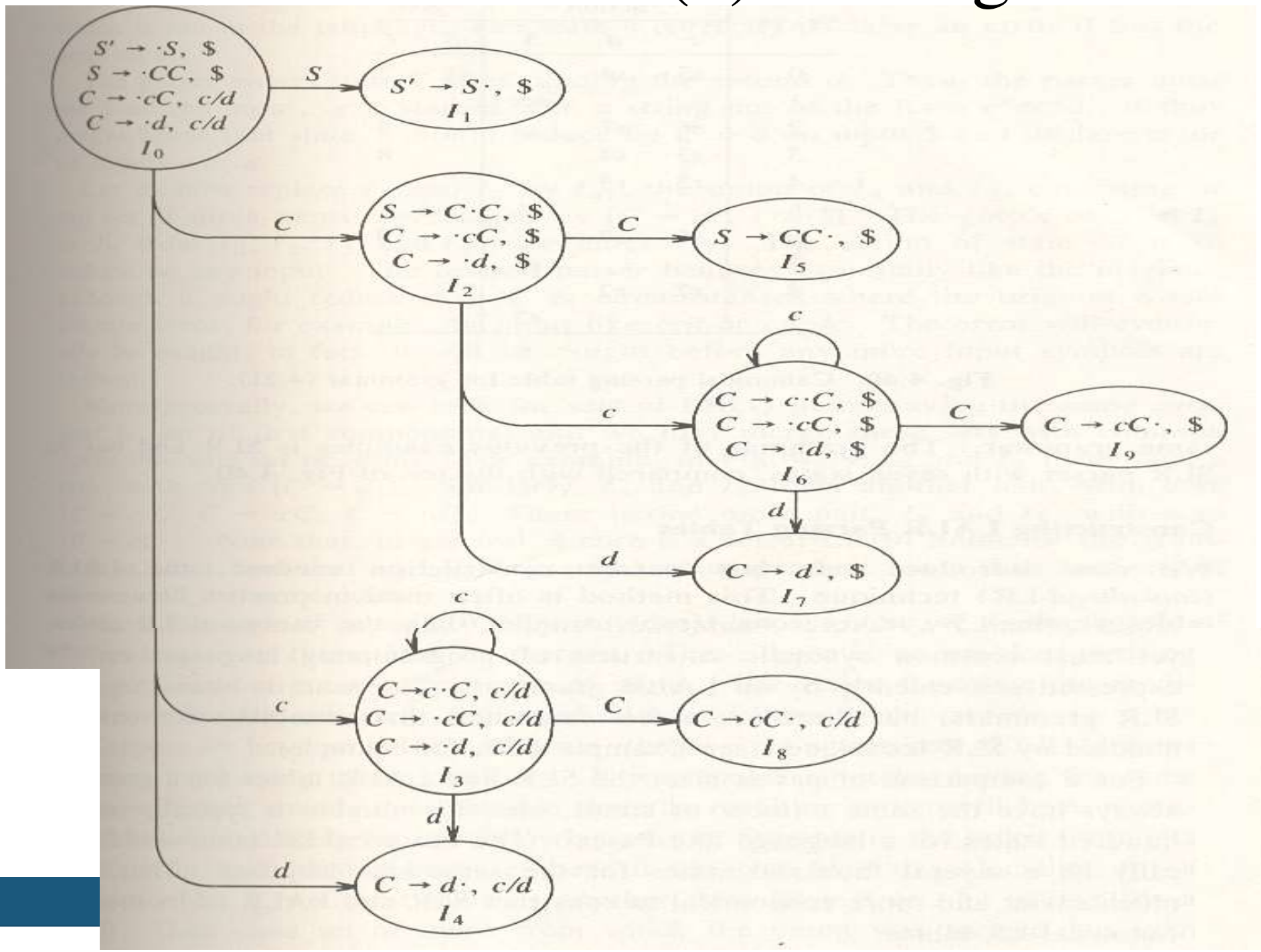
3. A → d

	Action Table			Goto Table	
State	b	d	$	S	A
0	s36	s47		1	2
1			accept		
2	s36	s47			5
36	s36	s47			89
47	R3	R3	R3		
5			R1		
89	R2	R2	R2		

Stack Implementation of LALR(1)

Stack	Input	Action
$0	bdd$	Shift
$0 b 36	dd$	Shift
$0 b 36 d 47	d$	Reduce(R3): **A→d**
$0 b 36 A 89	d$	Reduce(R2):**A→*bA***
$0 A 2	d$	Shift
$0 A 2 d 47	$	Reduce(R3): **A→d**
$0 A 2 A 5	$	Reduce(R1): **S→AA**
$0 S 1	$	Accept

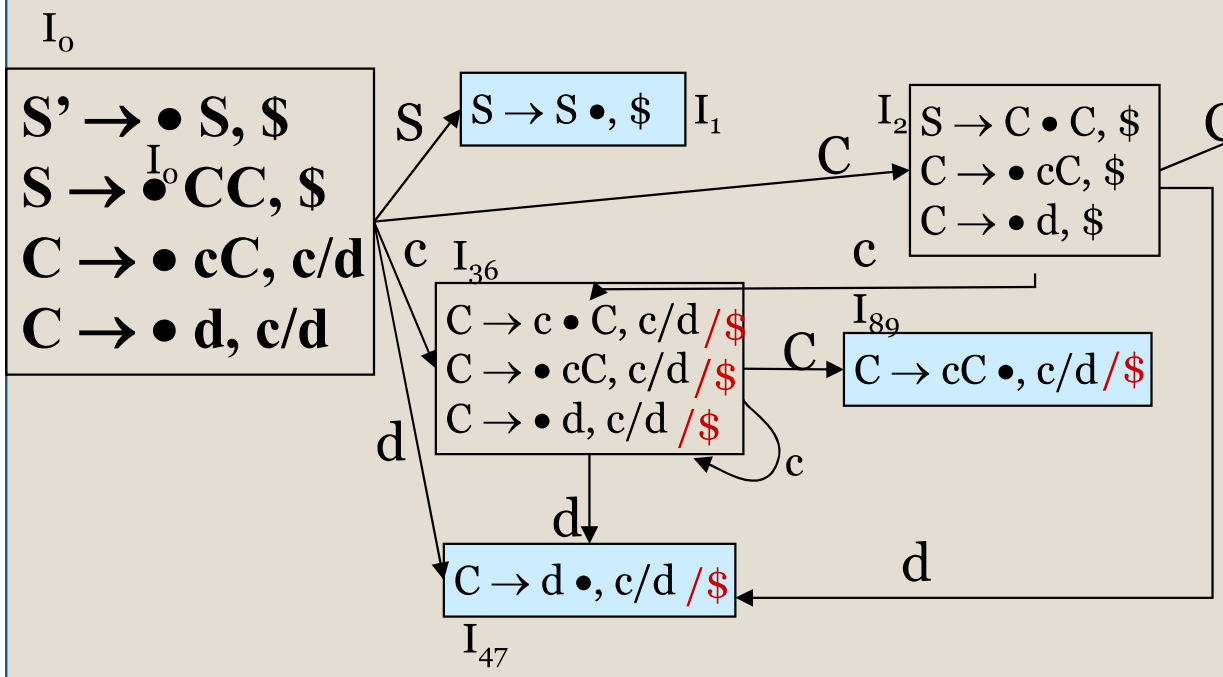
CLR (1) Parsing



LALR Parsing

G: 1. S → CC
2. C → cC
3. C → d

First(C) = {c, d}
First(S) = {c, d}
Follow(S) = {$}
Follow(C) = {c,d,$}



	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		

LALR Parsing

- Canonical LR(1) Parser  LALR Parser

  shrink no. of states

- The merging of states with common cores can never produce a **shift/reduce conflict**, because shift actions depend only on the core not the lookahead.
- However, that a merge will produce a **reduce/reduce conflict** (so the grammar is **NOT LALR**)

Conflicts in LALR(1) parsing

- LALR(1) parsers cannot introduce shift/reduce conflicts.
  - Such conflicts are caused when a lookahead is the same as a token on which we can shift. They depend on the core of the item. But we only merge states that had the same core to begin with. The only way for an LALR(1) parser to have a shift/reduce conflict is if one existed already in the LR(1) parser.
- LALR(1) parsers can introduce reduce/reduce conflicts.
  - Here's a situation when this might happen:

$A \rightarrow B \bullet, x$ $A \rightarrow C \bullet, y$	merges with	$A \rightarrow B \bullet, y$ $A \rightarrow C \bullet, x$	to give:	$A \rightarrow B \bullet, x/y$ $A \rightarrow C \bullet, x/y$

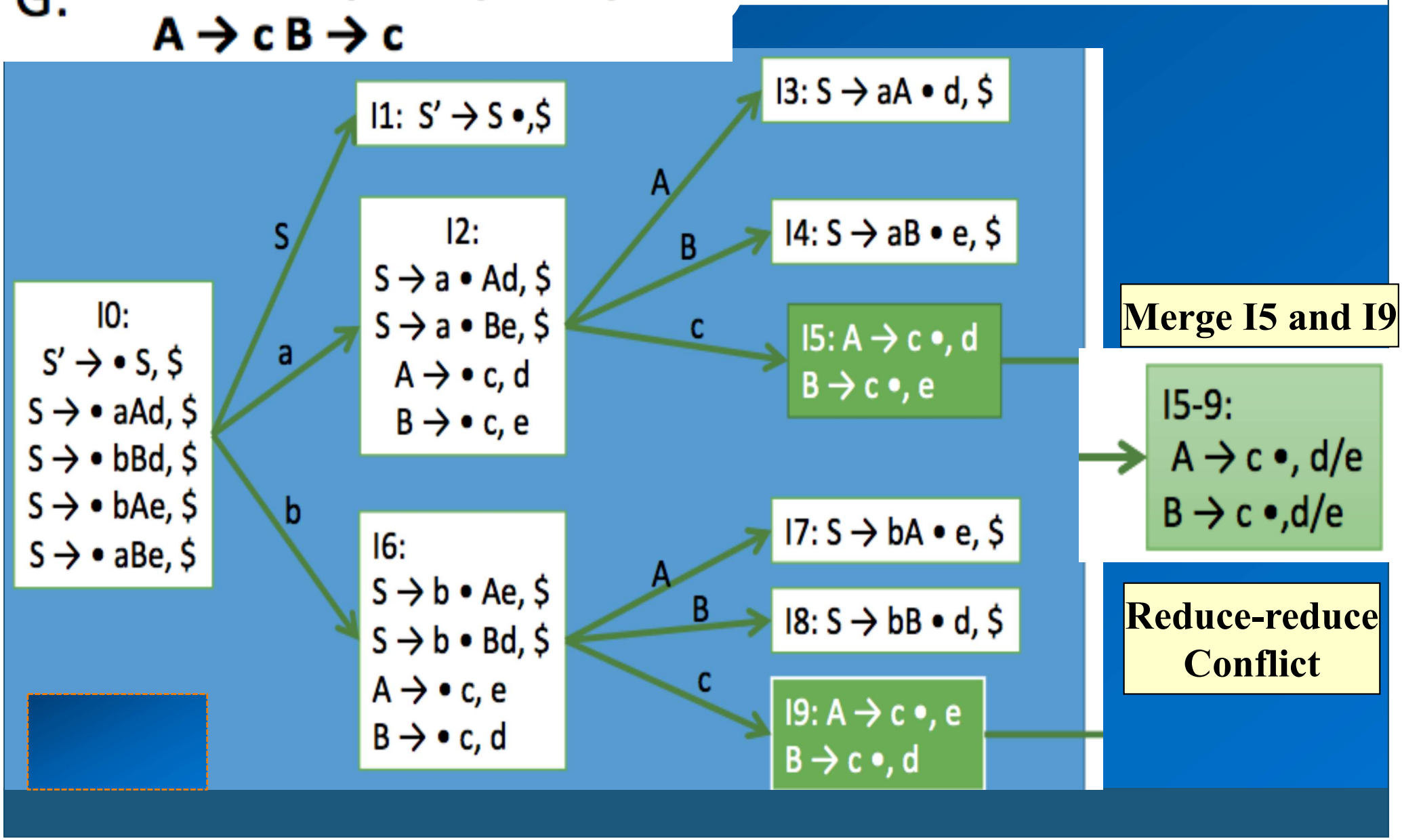
Shift-Reduce conflict

- **Cannot introduce shift-reduce conflict?**
  - **Assume**: two CLR states **I1, I2** are merged into an LALR state **I**
  - If conflict, **I** must have items
    - **$[A \rightarrow \alpha \bullet, a]$ and $[B \rightarrow \beta \bullet a\delta, b]$**
      - In fact, **α and β** have to be the same, otherwise, they won't come to the same state
    - If they are from different states, they are different core items, cannot be merged into **I**
    - If **I1** has **$[A \rightarrow \alpha \bullet, a]$ and $[B \rightarrow \alpha \bullet b\delta, c]$** and **I2** has

      $[A \rightarrow \alpha \bullet, d]$ and $[B \rightarrow \alpha \bullet b\delta, e]$
      - To have a conflict, we should have b = d or b = a, shift-reduce conflicts were there in I1 and I2 already!

Reduce-Reduce conflict

G: **S → aAd | bBd | bAe | aBe**
A → c B → c



Homework

- **Construct the**

  LR(0), SLR(1), CLR(1), and LALR(1) DFA Design , Action table and Goto table respectively for the following grammar:

$$S \rightarrow xSx$$

$$S \rightarrow x$$

Using Ambiguous Grammars

- It is a fact that every ambiguous grammar fails to be LR.
- However certain types of ambiguous are quite useful in the specification and implementation.
  - An ambiguous grammars provides a shorter, more natural specification than equivalent unambiguous grammar , for language constructs like expressions.
  - Uses in isolating commonly occurring syntactic constructs for special case optimization. By carefully adding new production to the grammar.
  - Usage of an ambiguous grammar may eliminate unnecessary reductions.
- **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.

SLR(1) Parsing with Ambiguity

1

S→ ·E
E→·E+E
E→·E*E
E → · int
E → · (E)

1. **S → E**
2. **E → E + E**
3. **E → E * E**
4. **E → (E)**
5. **E → int**

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

2

S→ E·
E → E ·+ E
E → E ·* E

3

E → E + ·E
E→·E+E
E→·E*E
E → · int
E → · (E)

4

E → E * ·E
E→·E+E
E→·E*E
E → · int
E → · (E)

5

E→E+E·
E→E·+E
E→E·*E

6

E→E*E·
E→E·+E
E→E·*E

7

E → (·E)
E→·E+E
E→·E*E
E → · int
E → · (E)

8

E → (E ·)
E → E ·+ E
E → E ·* E

9

E → (E) ·

10

E → int ·

	int	+	*	(	)	$	E
1	s10			s7			2
2		s3	s4			acc	
3	s10			s7			5
4	s10			s7			6
5		s3 r2	s4 r2		r2	r2	
6		s3 r3	s4 r3		r3	r3	
7	s10			s7			8
8		s3	s4		s9		
9		r4	r4		r4	r4	
10		r5	r5		r5	r5	

Resolving Conflicts with Precedence & associatively

- When choosing whether to reduce a rule containing t or shift the terminal r:
  - If t has higher priority, reduce.
  - If r has higher priority, shift.
  - If t and r have the same priority:
    - If t is left-associative, reduce.
    - If t is right-associative, shift.
    - If t is non-associative, error.

Resolving Conflicts with Precedence & associatively

- For example, in state 5 it must be the case that we have E + E on top of the stack, since one of the state 5 actions is r2.
  - If the lookahead is + we have a shift/reduce conflict. Here we want to reduce, to reflect the fact that + is left-associative.
  - If the lookahead is * we again have a shift/reduce conflict. In this case we should shift, since * has higher precedence than +.
- Among the state 6 actions is r3, so we must have E * E on top of the stack:
  - If the lookahead is + we have a shift/reduce conflict. Here we want to reduce, to reflect the precedence of * over +.
  - If the lookahead is * we again have a shift/reduce conflict. Here we want to reduce also, this time to reflect the fact that * is left-associative.

SLR(1) Parsing with Ambiguity

1. **S → E**
2. **E → E + E** (left-assoc, pri.(0))
3. **E → E * E** (left-assoc, pri.(1))
4. **E → (E)**
5. **E → int**

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

1
S→ ·E
E→·E+E
E→·E*E
E → · int
E → · (E)

2
S→ E·
E → E ·+ E
E → E ·* E

3
E → E + ·E
E→·E+E
E→·E*E
E → · int
E → · (E)

4
E → E * ·E
E→·E+E
E→·E*E
E → · int
E → · (E)

5
E→E+E·
E→E·+E
E→E·*E

6
E→E*E·
E→E·+E
E→E·*E

7
E → (·E)
E→·E+E
E→·E*E
E → · int
E → · (E)

8
E → (E ·)
E → E ·+ E
E → E ·* E

9
E → (E) ·

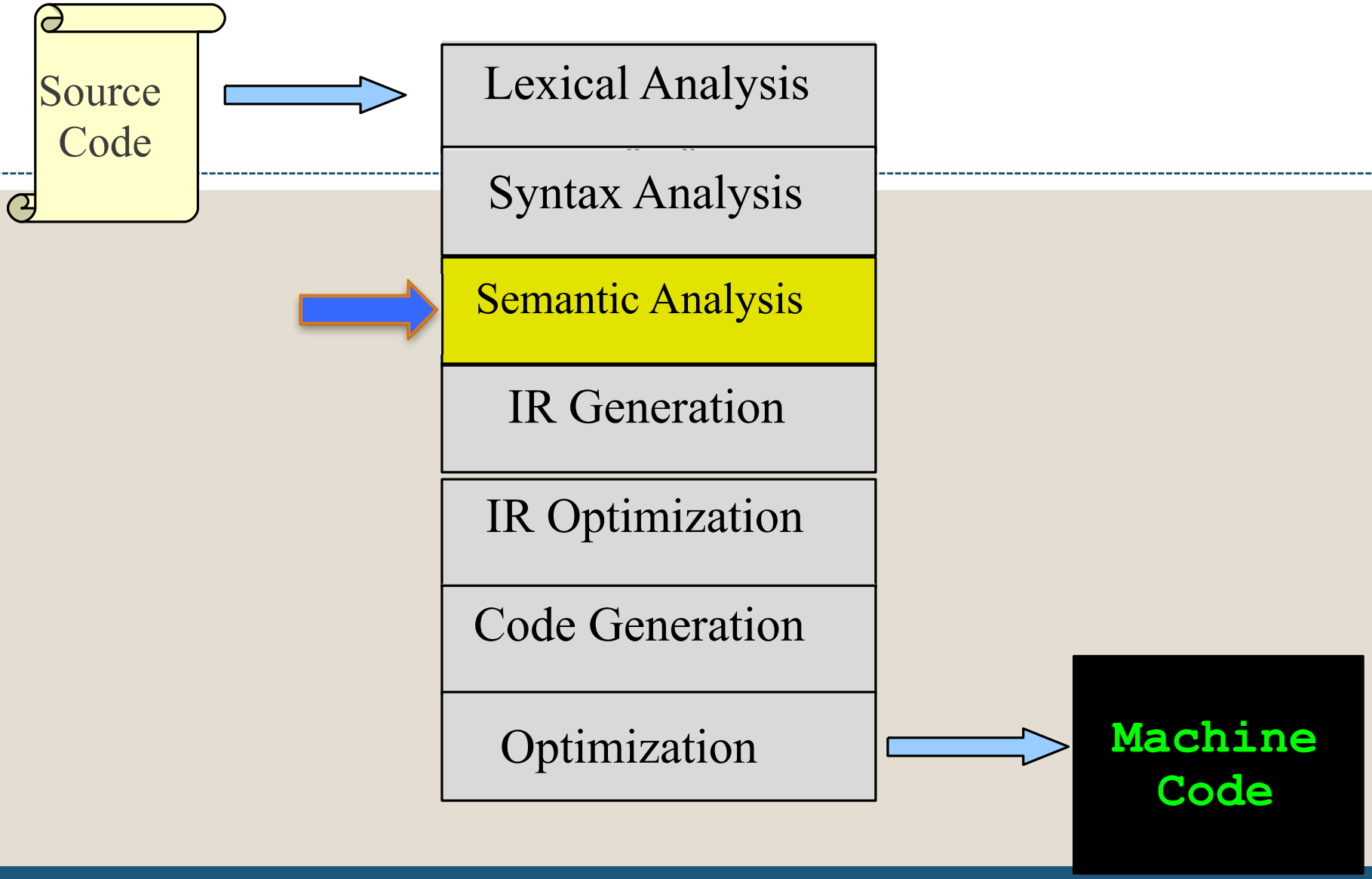
10
E → int ·

	int	+	*	(	)	$	E
1	s10			s7			2
2		s3	s4			acc	
3	s10			s7			5
4	s10			s7			6
5		r2	s4		r2	r2	
6		r3	r3		r3	r3	
7	s10			s7			8
8		s3	s4		s9		
9		r4	r4		r4	r4	
10		r5	r5		r5	r5	

Error Recovery in LR Parsing

- An LR parser will detect an error when it consults the parsing action table and finds an error entry. Or when an entry in the action table is found to be empty.
- 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 not make even a single reduction before announcing an error.
- The SLR and LALR parsers may make several reductions before announcing an error. But, they will never shift an erroneous input symbol onto the stack.
- The techniques are used to solve the problem such as :
  - Panic Mode Error Recovery in LR Parsing
  - Phrase-Level Error Recovery in LR Parsing

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Any Questions?