23Z602 COMPILER DESIGN
Unit-3
SYNTAX ANALYSIS

Dr.C.Kavitha
Associate Prof
Dept. of CSE
PSG College of Technology

Bottom Up Parsers:

Bottom Up Parsers: Shift Reduce Parser - LR Parser - LR (0) Item - Construction Of SLR Parsing Table - CLR Parser - LALR Parser.

Error Handling and Recovery in Syntax Analyzer

YACC Tool: Structure of YACC Program – Communication between LEX and YACC - Design of a Syntax Analyzer for a Sample Language.

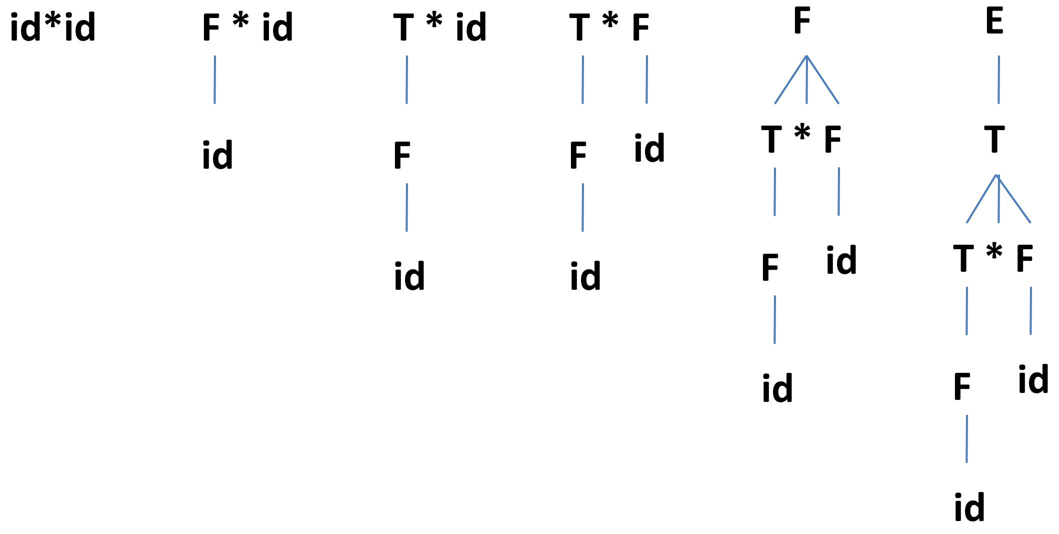
Outline

- **Bottom-up Parsing**
- **Parser generators**

Introduction

- Constructs parse tree for an input string beginning at the leaves (the bottom) and working towards the root (the top)
- Example: id*id

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



id*id

F * id
id

T * id
F
id

T * F
F id
id

F
T * F
F id
id

E
T
T * F
F id
id

Bottom-Up Parsing

- **Bottom-Up Parser : Constructs a parse tree for an input string beginning at the leaves(the bottom) and working up towards the root(the top)**
- **Bottom-up parsing is also known as *shift-reduce parsing* because its two main actions are shift and reduce.**
  - ❑ **At each shift action, the current symbol in the input string is pushed to a stack.**
  - ❑ **At each reduction step, the symbols at the top of the stack (this symbol sequence is the right side of a production) will replaced by the non-terminal at the left side of that production.**

Shift-reduce parser

- The general idea is to shift some symbols of input to the stack until a reduction can be applied
- At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of the production
- The key decisions during bottom-up parsing are about when to reduce and about what production to apply
- A reduction is a reverse of a step in a derivation
- The goal of a bottom-up parser is to construct a derivation in reverse:

  E=>T=>T*F=>T*id=>F*id=>id*id

Shift reduce parsing

- A stack is used to hold grammar symbols
- Handle always appear on top of the stack
- Initial configuration:

Stack	Input
$	w$

- Acceptance configuration

Stack	Input
$S	$

Shift-Reduce Parsing

- A shift-reduce parser tries to reduce the given input string into the starting symbol.

  a string $\underset{\text{reduced to}}{\square}$ the starting symbol

- At each reduction step, a substring of the input matching to the right side of a production rule is replaced by the non-terminal at the left side of that production rule.
- If the substring is chosen correctly, the right most derivation of that string is created in the reverse order.

  Rightmost Derivation: $S \overset{*}{\underset{rm}{\Rightarrow}} \omega$

  Shift-Reduce Parser finds: $\omega \underset{rm}{\Leftarrow} \dots \underset{rm}{\Leftarrow} S$

Handle pruning

- A handle is a substring of grammar symbols in a right-sentential form that matches a right-hand side of a production
- A Handle is a substring that matches the body of a production and whose reduction represents one step along the reverse of a rightmost derivation

Right sentential form	Handle	Reducing production
id*id	id	F->id
F*id	F	T->F
T*id	id	F->id
T*F	T*F	E->T*F

Handle Pruning

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \ldots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = \omega$$

- Start from γ_n, find a handle $A_n \rightarrow \beta_n$ in γ_n, and replace β_n in by A_n to get γ_{n-1}.
- Then find a handle $A_{n-1} \rightarrow \beta_{n-1}$ in γ_{n-1}, and replace β_{n-1} in by A_{n-1} to get γ_{n-2}.
- Repeat this, until we reach S.

A Shift-Reduce Parser

E ⟶ E+T | T
T ⟶ T*F | F
F ⟶ (E) | id

Right-Most Derivation of id+id*id
E ⇒ E+T ⇒ E+T*F ⇒ E+T*id ⇒ E+F*id
⇒ E+id*id ⇒ T+id*id ⇒ F+id*id ⇒ id+id*id

Right-Most Sentential form	HANDLE	Reducing Production
id+id*id	id	F→id
F+id*id	F	T→F
T+id*id	T	E→T
E+id*id	id	F→id
E+F*id	F	T→F
E+T*id	Id	F→id
E+T*F	T*F	T→T*F
E+T	E+T	E→E+T
E		

A Stack Implementation of a Shift-Reduce Parser

- There are four possible actions of a shift-parser action:

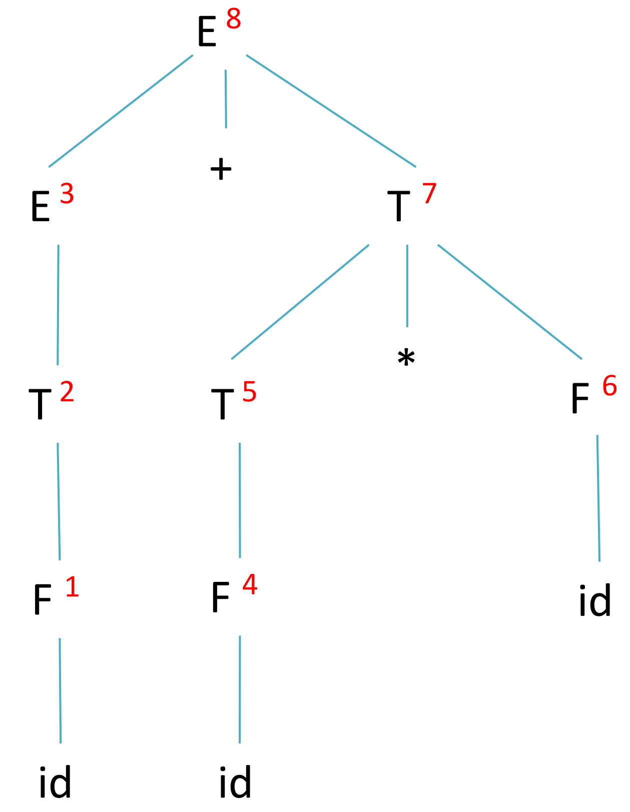
  1. Shift : The next input symbol is shifted onto the top of the stack.
  2. Reduce: Replace the handle on the top of the stack by the non-terminal.
  3. Accept: Successful completion of parsing.
  4. Error: Parser discovers a syntax error, and calls an error recovery routine.

- Initial stack just contains only the end-marker $.
- The end of the input string is marked by the end-marker $.

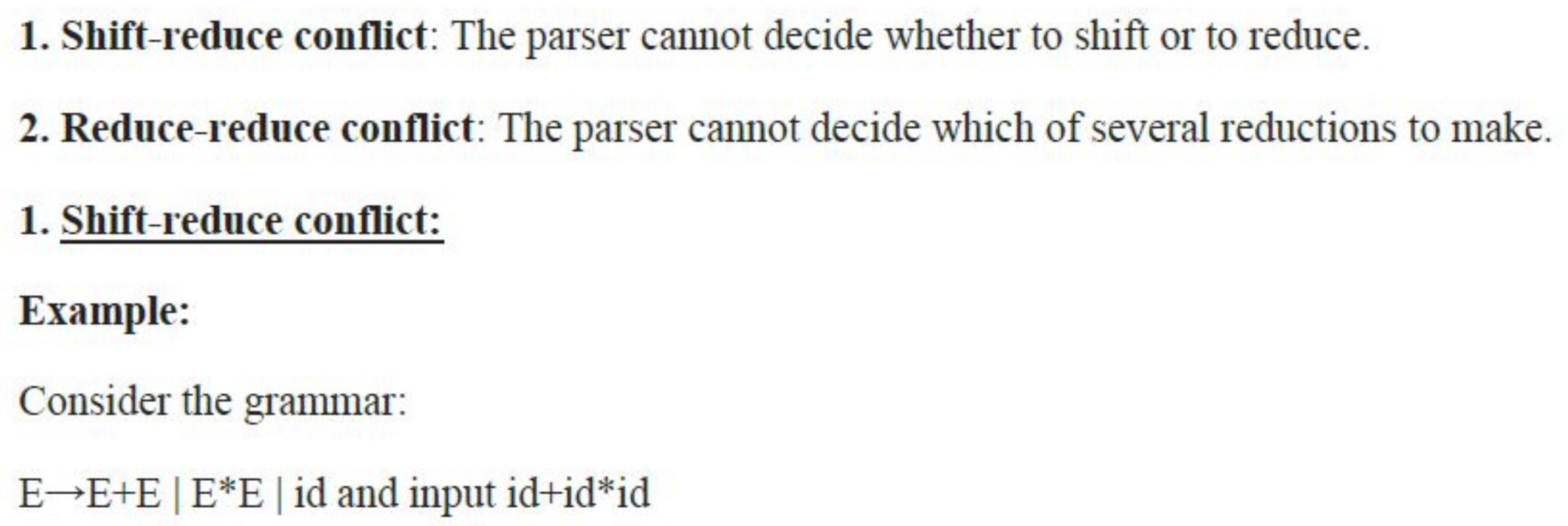
A Stack Implementation of A Shift-Reduce Parser

Stack	Input	Action
$	id+id*id$	shift
$id	+id*id$	Reduce by F→id
$F	+id*id$	Reduce by T→F
$T	+id*id$	Reduce by E→T
$E	+id*id$	Shift
$E+	Id*id$	Shift
$E+id	*id$	Reduce by F→id
$E+F	*id$	Reduce by T→F
$E+T	*id$	Shift
$E+T*	id$	Shift
$E+T*id	$	Reduce by F→id
$E+T*F	$	Reduce by T→T*F
$E+T	$	Reduce by E →E+T
$E	$	Accept

Parse Tree



Conflicts during shit reduce parsing



1. Shift-reduce conflict: The parser cannot decide whether to shift or to reduce.

2. Reduce-reduce conflict: The parser cannot decide which of several reductions to make.

1. Shift-reduce conflict:

Example:

Consider the grammar:

E→E+E | E*E | id and input id+id*id

Shift-Reduce Conflict

Grammar

$E \rightarrow E + E$

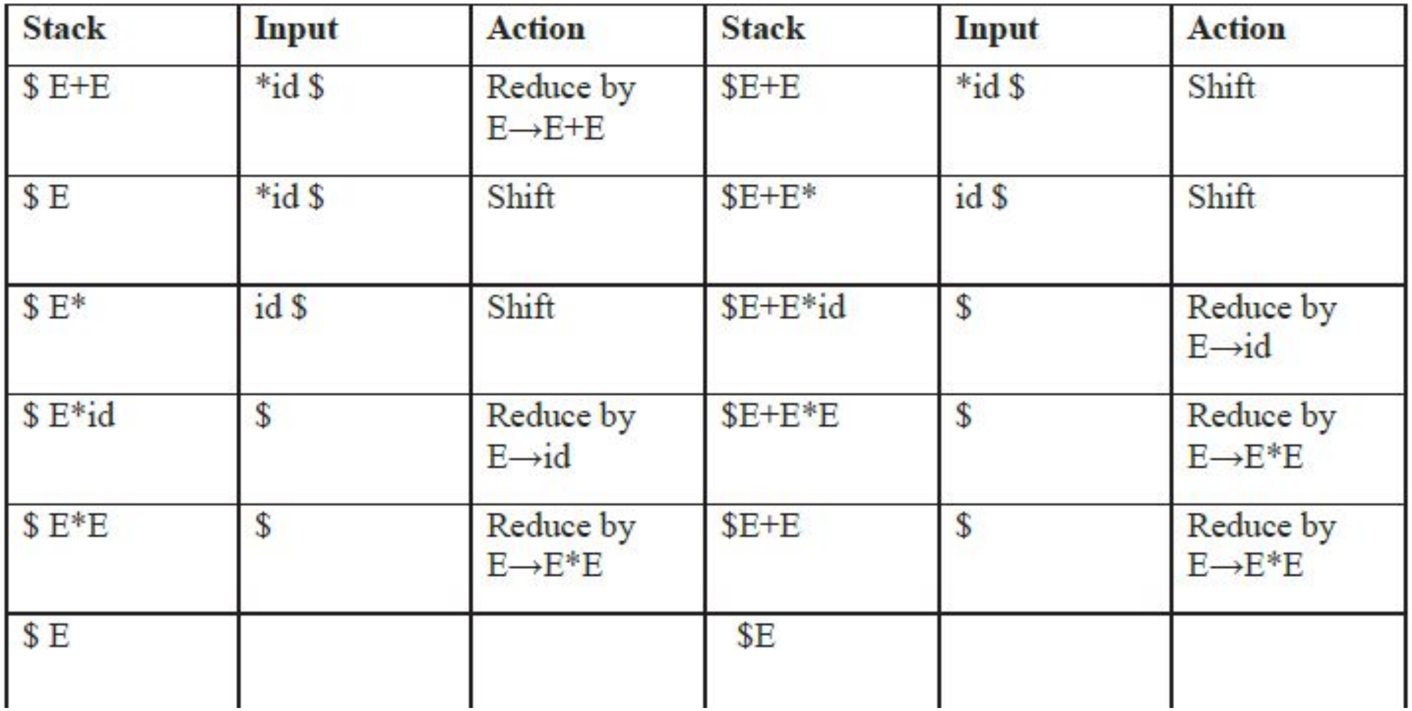
$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow \textbf{id}$

Stack	Input	Action
$\$$	id+id*id$	shift
$\$\underline{\textbf{id}}$	+id*id$	reduce $E \rightarrow \textbf{id}$
$\$E$	+id*id$	shift
$\$E+$	id*id$	shift
$\$E+\underline{\textbf{id}}$	*id$	reduce $E \rightarrow \textbf{id}$
$\$E+E$	*id$	**shift (or reduce?)**
$\$E+E*$	id$	shift
$\$E+E*\underline{\textbf{id}}$	$	reduce $E \rightarrow \textbf{id}$
$\$E+\underline{E*E}$	$	reduce $E \rightarrow E * E$
$\$\underline{E+E}$	$	reduce $E \rightarrow E + E$
$\$E$	$	accept

Shift-Reduce Conflict



Stack	Input	Action	Stack	Input	Action
$ E+E	*id $	Reduce by E→E+E	$E+E	*id $	Shift
$ E	*id $	Shift	$E+E*	id $	Shift
$ E*	id $	Shift	$E+E*id	$	Reduce by E→id
$ E*id	$	Reduce by E→id	$E+E*E	$	Reduce by E→E*E
$ E*E	$	Reduce by E→E*E	$E+E	$	Reduce by E→E*E
$ E			$E		

Shift-Reduce Conflict

Ambiguous grammar:
S → **if** *E* **then** *S*
| **if** *E* **then** *S* **else** *S*
| **other**

Stack	Input	Action
$…	…$	…
$…**if** *E* **then** *S*	**else**…$	shift or reduce?

Shift **else** to **if** *E* **then** *S* or
Reduce **if** *E* **then** *S*

Resolve in favor
of shift, so **else**
matches closest **if**

Reduce-Reduce Conflict

Grammar

$C \rightarrow A\ B$

$A \rightarrow \mathbf{a}$

$B \rightarrow \mathbf{a}$

Stack	Input	Action
$	**aa$**	shift
$\mathbf{\$}\underline{\mathbf{a}}$	**a$**	reduce $A \rightarrow \mathbf{a}$ $\underline{\text{or}}$ $B \rightarrow \mathbf{a}$?

Resolve in favor of reduce $A \rightarrow \mathbf{a}$, otherwise we're stuck!

2. Reduce-reduce conflict:

Consider the grammar:

M → R+R | R+c | R
R → c
and input c+c

Stack	Input	Action	Stack	Input	Action
$	c+c $	Shift	$	c+c $	Shift
$ c	+c $	Reduce by R→c	$ c	+c $	Reduce by R→c
$ R	+c $	Shift	$ R	+c $	Shift
$ R+	c $	Shift	$ R+	c $	Shift
$ R+c	$	Reduce by R→c	$ R+c	$	Reduce by M→R+c
$ R+R	$	Reduce by M→R+R	$ M	$	
$ M	$				

Practice problems

- Consider the following grammar-

  S → (L) | a

  L → L , S | S

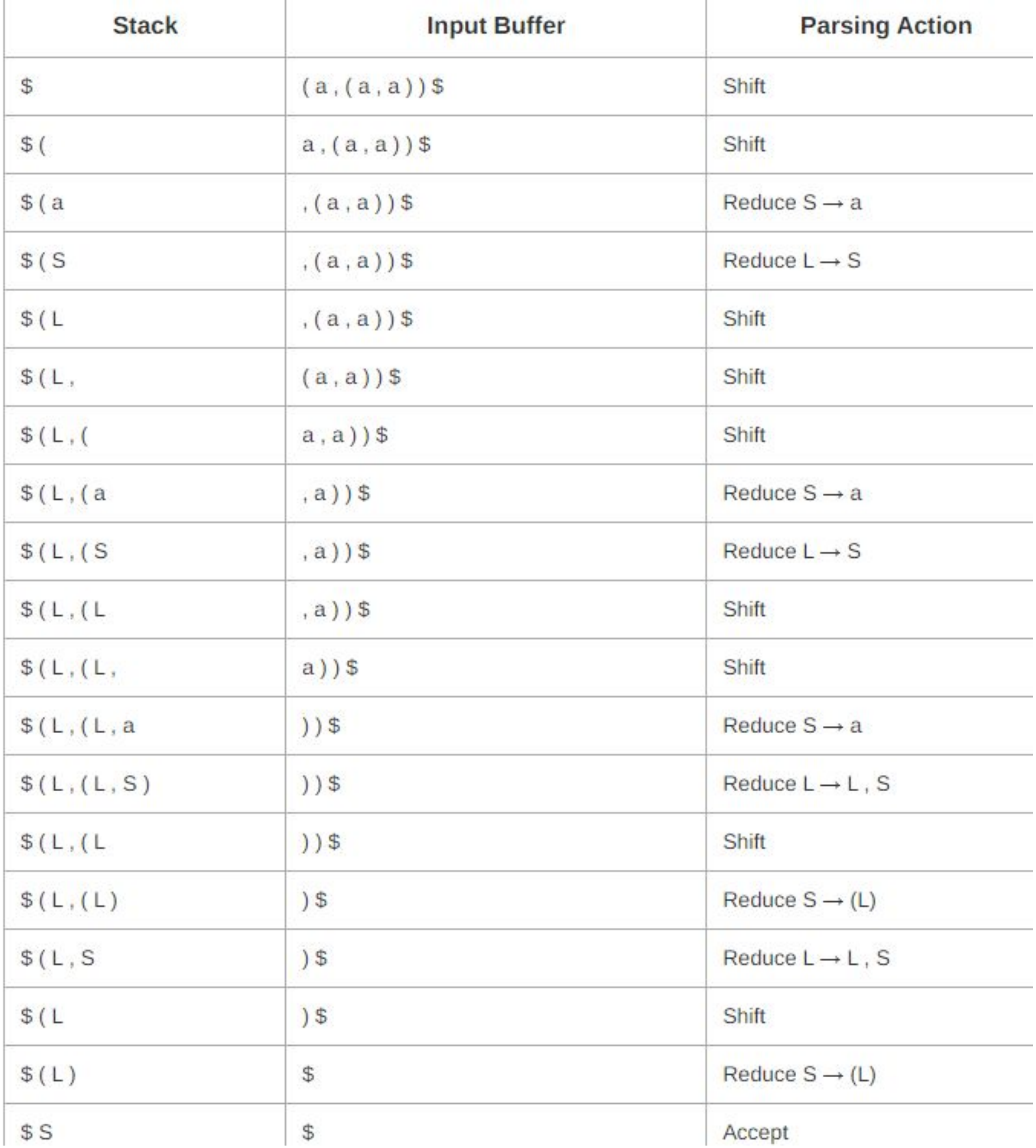
  Parse the input string **(a , (a , a))** using a shift-reduce parser.
- Consider the following grammar-

  S → T L;

  T → int | float

  L → L , id | id

  Parse the input string **int id , id ;** using a shift-reduce parser.



Stack	Input Buffer	Parsing Action
$	(a , (a , a)) $	Shift
$ (	a , (a , a)) $	Shift
$ (a	, (a , a)) $	Reduce S → a
$ (S	, (a , a)) $	Reduce L → S
$ (L	, (a , a)) $	Shift
$ (L ,	(a , a)) $	Shift
$ (L , (	a , a)) $	Shift
$ (L , (a	, a)) $	Reduce S → a
$ (L , (S	, a)) $	Reduce L → S
$ (L , (L	, a)) $	Shift
$ (L , (L ,	a)) $	Shift
$ (L , (L , a	)) $	Reduce S → a
$ (L , (L , S)	)) $	Reduce L → L , S
$ (L , (L	)) $	Shift
$ (L , (L)	) $	Reduce S → (L)
$ (L , S	) $	Reduce L → L , S
$ (L	) $	Shift
$ (L)	$	Reduce S → (L)
$ S	$	Accept

Stack	Input Buffer	Parsing Action
$	int id , id ; $	Shift
$ int	id , id ; $	Reduce T → int
$ T	id , id ; $	Shift
$ T id	, id ; $	Reduce L → id
$ T L	, id ; $	Shift
$ T L ,	id ; $	Shift
$ T L , id	; $	Reduce L → L , id
$ T L	; $	Shift
$ T L ;	$	Reduce S → T L
$ S	$	Accept

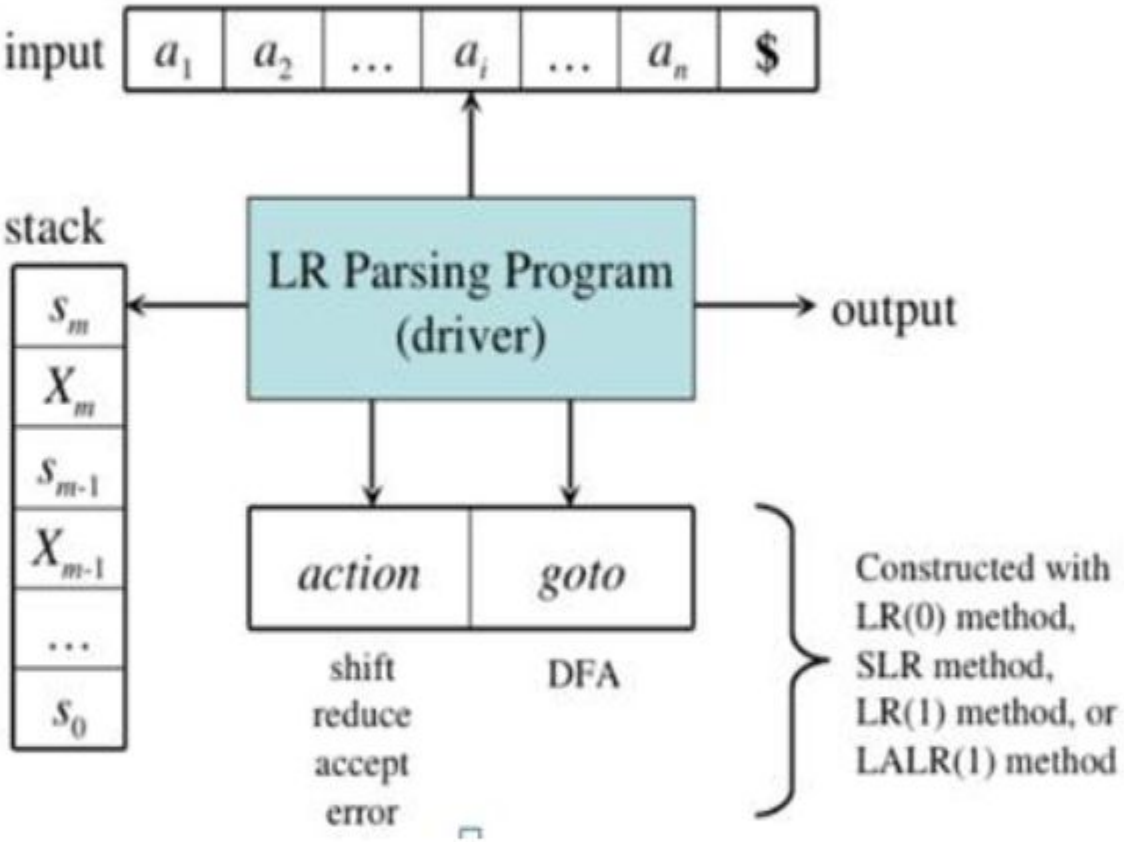
LR Parsing

- The most prevalent type of bottom-up parsers
- An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is called LR(k) parsing. The “L” is for left-to-right scanning of the input, the “R” for constructing a rightmost derivation in reverse, and the “k” for the number of input symbols of lookahead that are used in making parsing decisions. When (k) is omitted, it is assumed to be 1
- Why LR parsers?
  - Table driven
  - Can be constructed to recognize all programming language constructs
  - Most general non-backtracking shift-reduce parsing method
  - Can detect a syntactic error as soon as it is possible to do so
  - Class of grammars for which we can construct LR parsers are superset of those which we can construct LL parsers

Types of LR parsing method:

1. SLR- Simple LR
   - Easiest to implement, least powerful.
2. CLR- Canonical LR
   - Most powerful, most expensive.
3. LALR- Look -Ahead LR
   - Intermediate in size and cost between the other two methods

Model of an LR Parser



- It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (action and goto).
- The driver program is the same for all LR parser.
- The parsing table alone changes from one parser to another.
- The parsing program reads characters from an input buffer one at a time. The program uses a stack to store a string of the form s0X1s1X2s2…… Xmsm , where sm is on top. Each **Xi is a grammar symbol** and each **si is a symbol called a state**.

The parsing table consists of two parts : action and goto functions.

Action : The parsing program determines sm, the state currently on top of stack, and ai, the current input symbol. It then consults **action[sm,ai]** in the action table which can have one of **four values** :

1. shift s, where s is a state,
2. reduce by a grammar production A → β,
3. accept
4. error.

Goto : The function goto takes a state and grammar symbol as arguments and produces a state.

CONSTRUCTING SLR PARSING TABLE: To perform SLR parsing, take grammar as input and do the following:

1. Find LR(0) items.
2. Completing the closure.
3. Compute goto(I,X), where, I is set of items and X is grammar symbol.

LR(0) items:

An LR(0) item of a grammar G is a production of G with a dot at some position of the right side. For example, production $A \rightarrow XYZ$ yields the four items :

$A \rightarrow \bullet XYZ$

$A \rightarrow X \bullet YZ$

$A \rightarrow XY \bullet Z$

$A \rightarrow XYZ \bullet$

Closure operation: If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules:

1. Initially, every 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, then add the item $B \rightarrow . \gamma$ to I , if it is not already there. Apply this rule until no more new items can be added to closure(I).

Goto operation: Goto(I, X) is defined to be the closure of the set of all items $[A \rightarrow \alpha X \bullet \beta]$ such that $[A \rightarrow \alpha \bullet X \beta]$ is in I.

Steps to construct SLR parsing table for grammar G are:

1. Augment G and produce G'

   Augmented grammar:

   - G with addition of a production: S'->S
2. Construct the canonical collection of set of items C for G'
3. Construct the parsing action function **action and goto** using the algorithm that requires FOLLOW(A) for each non-terminal of grammar.

Closure algorithm

```
SetOfItems CLOSURE(I) {
    J=I;
    repeat
     for (each item A-> α.Bβ in J)
         for (each production B->γ of G)
             if (B->.γ is not in J)
                 add B->.γ to J;
    until no more items are added to J on one round;
    return J;
```

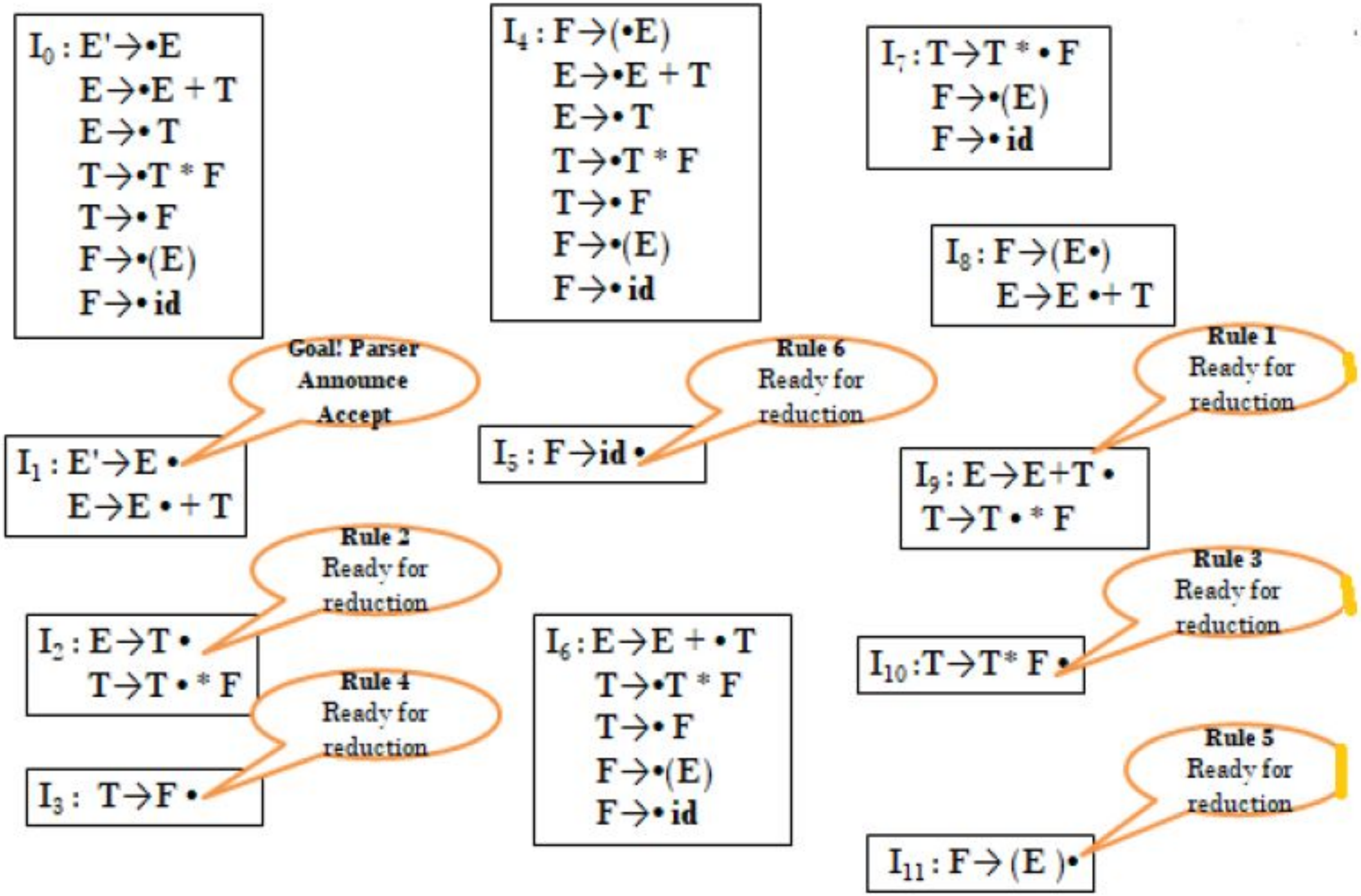
GOTO algorithm

```
SetOfItems  GOTO(I,X) {
  J=empty;
   if (A-> α.X β is in I)
    add CLOSURE(A-> αX. β ) to J;
   return J;
}
```

Canonical LR(0) items

```
Void items(G’) {
   C= CLOSURE({[S’->.S]});
   repeat
    for (each set of items I in C)
       for (each grammar symbol X)
          if (GOTO(I,X) is not empty and not in C)
         add GOTO(I,X) to C;
   until no new set of items are added to C on a
   round;
}
```

Canonical LR(0) collections



$I_0 : E' \rightarrow \bullet E$
$E \rightarrow \bullet E + T$
$E \rightarrow \bullet T$
$T \rightarrow \bullet T * F$
$T \rightarrow \bullet F$
$F \rightarrow \bullet (E)$
$F \rightarrow \bullet id$

$I_1 : E' \rightarrow E \bullet$
$E \rightarrow E \bullet + T$

Goal! Parser
Announce
Accept

$I_2 : E \rightarrow T \bullet$
$T \rightarrow T \bullet * F$

Rule 2
Ready for
reduction

$I_3 : T \rightarrow F \bullet$

Rule 4
Ready for
reduction

$I_4 : F \rightarrow (\bullet E)$
$E \rightarrow \bullet E + T$
$E \rightarrow \bullet T$
$T \rightarrow \bullet T * F$
$T \rightarrow \bullet F$
$F \rightarrow \bullet (E)$
$F \rightarrow \bullet id$

$I_5 : F \rightarrow id \bullet$

Rule 6
Ready for
reduction

$I_6 : E \rightarrow E + \bullet T$
$T \rightarrow \bullet T * F$
$T \rightarrow \bullet F$
$F \rightarrow \bullet (E)$
$F \rightarrow \bullet id$

$I_7 : T \rightarrow T * \bullet F$
$F \rightarrow \bullet (E)$
$F \rightarrow \bullet id$

$I_8 : F \rightarrow (E \bullet)$
$E \rightarrow E \bullet + T$

$I_9 : E \rightarrow E + T \bullet$
$T \rightarrow T \bullet * F$

Rule 1
Ready for
reduction

$I_{10} : T \rightarrow T * F \bullet$

Rule 3
Ready for
reduction

$I_{11} : F \rightarrow (E) \bullet$

Rule 5
Ready for
reduction

Constructing canonical LR(0) item sets

Example: Implement SLR Parser for the given grammar:

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

Step 1 : Convert given grammar into augmented grammar.

Augmented grammar:

E'⟶E
E⟶E + T
E⟶T
T⟶T * F
T⟶F
F⟶(E)
F⟶id

Step 2 : Find LR (0) items.

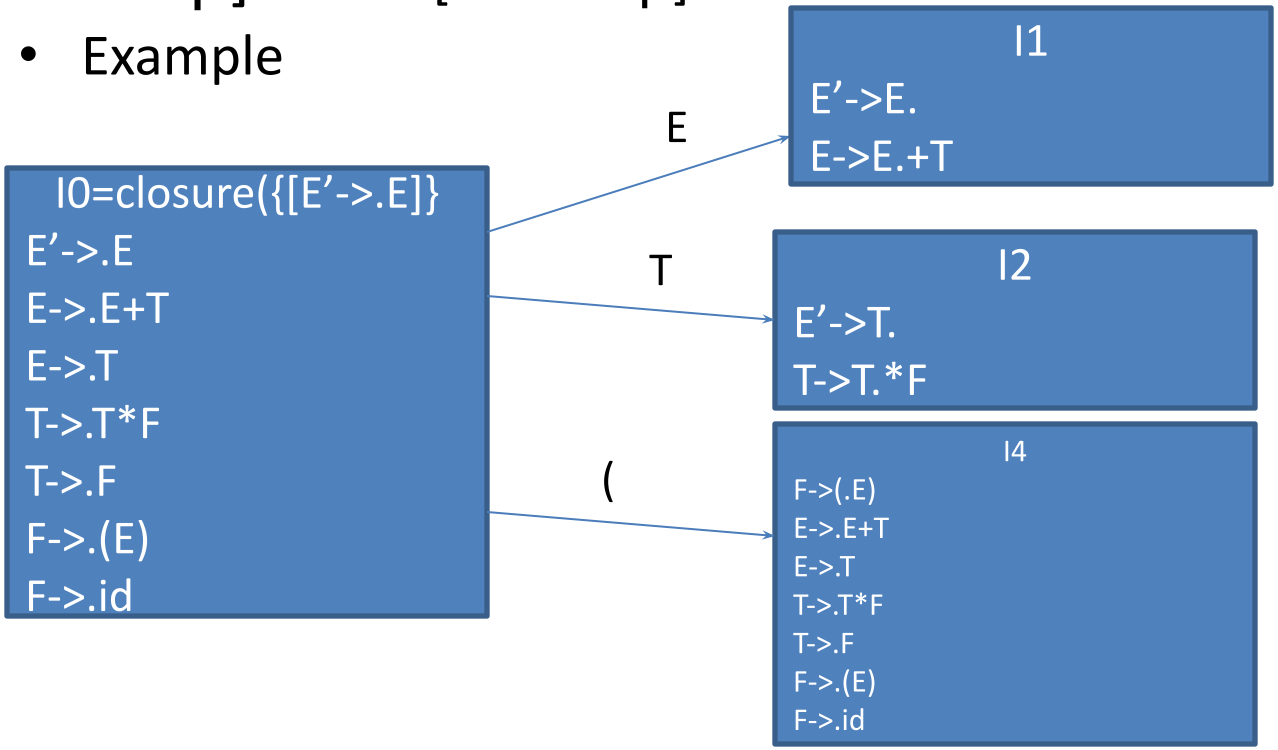
E’->E
E -> E + T | T
T -> T * F | F
F -> (E) | **id**

I0=closure({[E’->.E]}

E’->.E
E->.E+T
E->.T
T->.T*F
T->.F
F->.(E)
F->.id

Constructing canonical LR(0) item sets (cont.)

- Goto (I,X) where I is an item set and X is a grammar symbol is closure of set of all items [A-> αX. β] where [A-> α.X β] is in I
- Example



I0=closure({[E’->.E]}

E’->.E
E->.E+T
E->.T
T->.T*F
T->.F
F->.(E)
F->.id

I1

E’->E.
E->E.+T

I2

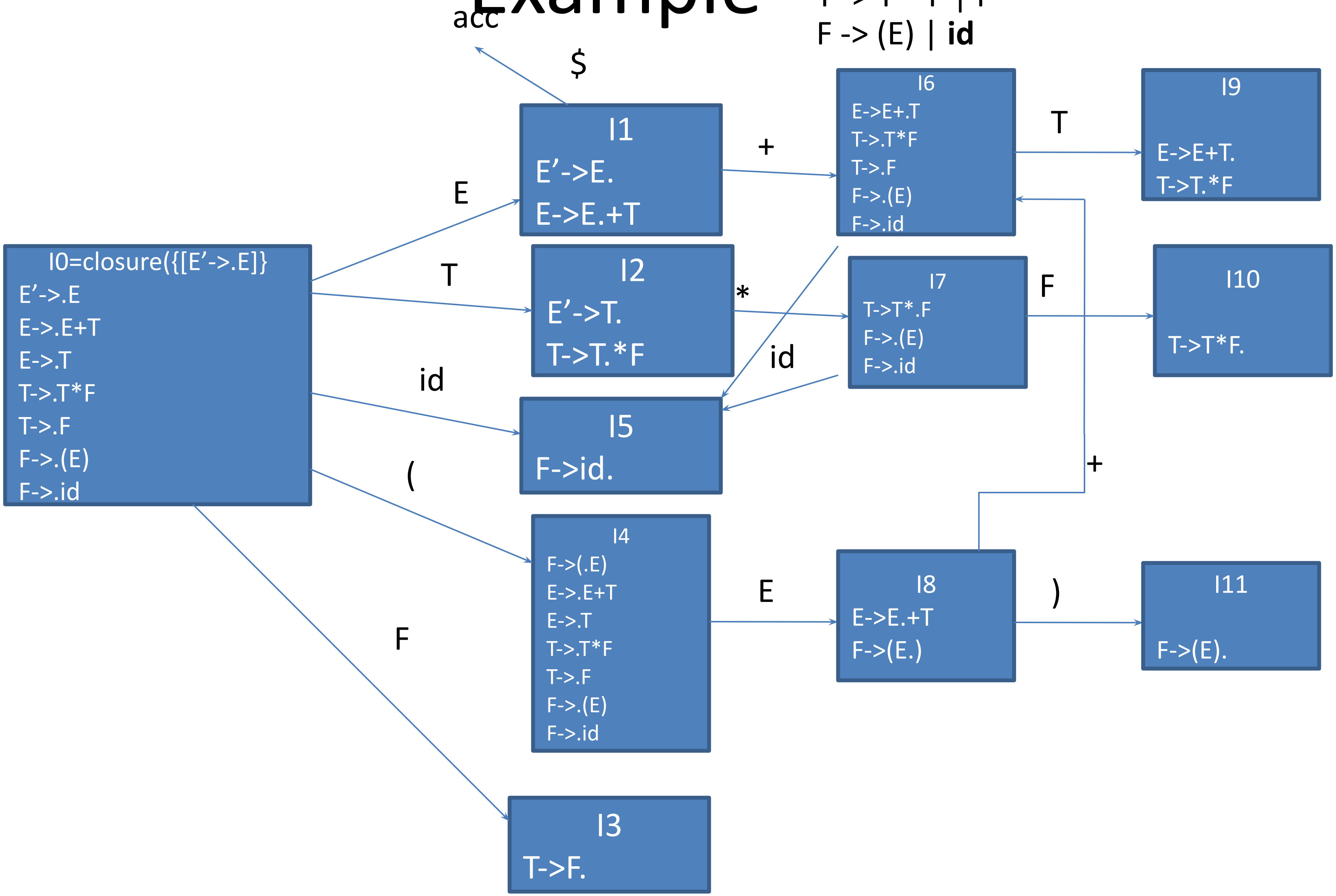
E’->T.
T->T.*F

I4

F->(.E)
E->.E+T
E->.T
T->.T*F
T->.F
F->.(E)
F->.id

Example

E’->E
E -> E + T | T
T -> T * F | F
F -> (E) | **id**



I0=closure({[E’->.E]}
E’->.E
E->.E+T
E->.T
T->.T*F
T->.F
F->.(E)
F->.id

I1
E’->E.
E->E.+T

I2
E’->T.
T->T.*F

I3
T->F.

I4
F->(.E)
E->.E+T
E->.T
T->.T*F
T->.F
F->.(E)
F->.id

I5
F->id.

I6
E->E+.T
T->.T*F
T->.F
F->.(E)
F->.id

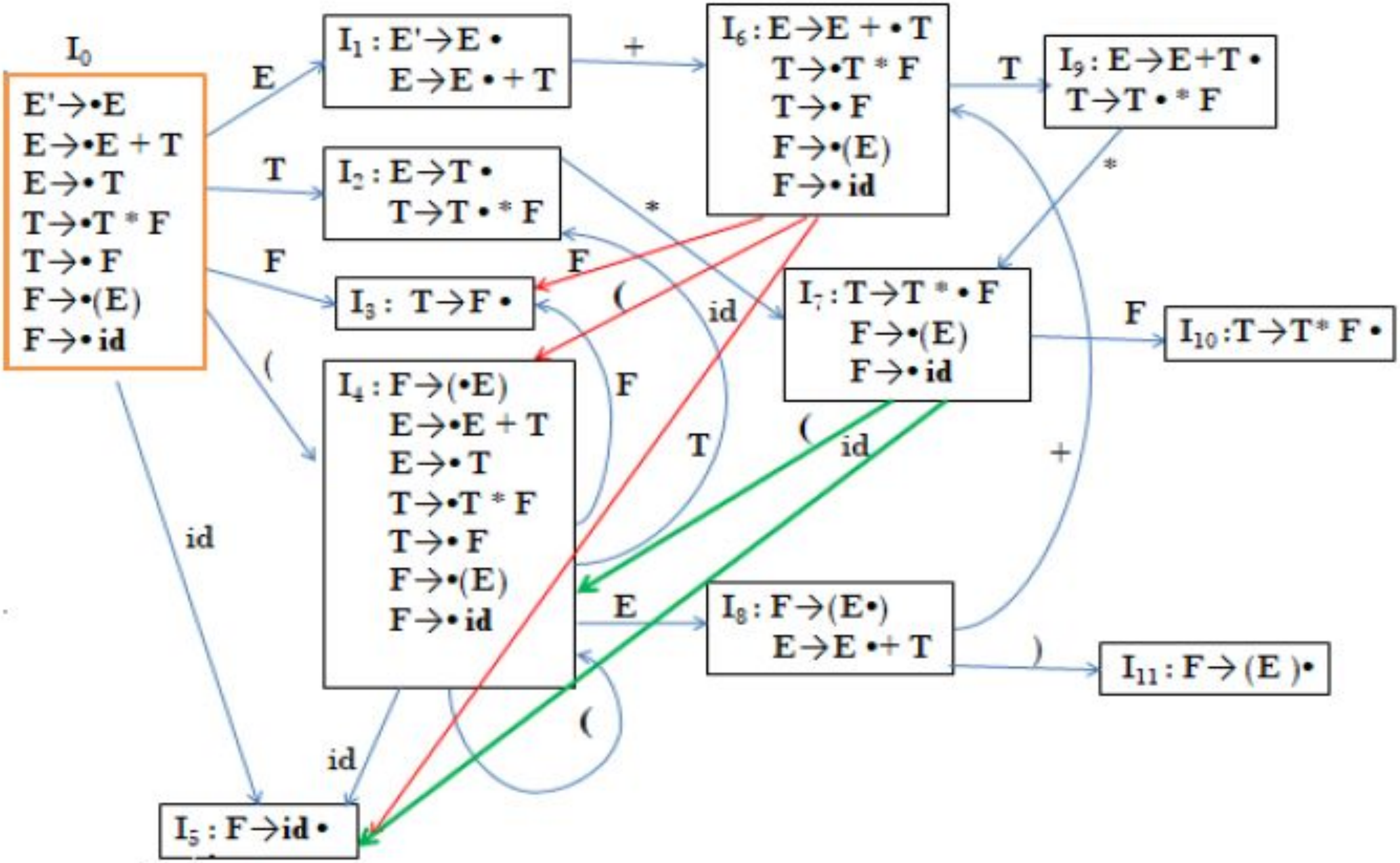
I7
T->T*.F
F->.(E)
F->.id

I8
E->E.+T
F->(E.)

I9
E->E+T.
T->T.*F

I10
T->T*F.

I11
F->(E).



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)•

Algorithm for construction of SLR parsing table:

Input : An augmented grammar G'

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

Method :

1. Construct C ={I0, I1, …. In}, the **collection of sets of LR(0) items for G'.**
2. State i is constructed from Ii. The parsing functions for state i are determined as follows:

   (a) If [A→α•aβ] is in Ii and **goto(Ii,a) = Ij**, then set **action[i,a] to "shift j".** Here 'a' must be terminal.

   (b) If[A→α•] is in Ii , then set **action[i,a]** to "**reduce A→α**" for all **'a'** in **FOLLOW(A).**

   (c) If [S'→S•] is in Ii, then set **action[i,$]** to "accept".

   If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

3. The goto transitions for state i are constructed for all **non-terminals A** using the rule: If **goto(Ii,A)= Ij**, then **goto[i,A] = j**.
4. All entries not defined by rules (2) and (3) are made **"error".**
5. The initial state of the parser is the one constructed from the set of items containing **[S'→•S]**.

3. Construction of Parsing table.

1. Computation of FOLLOW is required to fill the reduction action in the ACTION part of the table.

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

STATE	ACTON						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			

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

LR parsing algorithm

```
let ‘a’ be the first symbol of w$;
while(1) { /*repeat forever */
      let s be the state on top of the stack;
      if (ACTION[s,a] = shift t) {
        push t onto the stack;
        let a be the next input symbol;
      } else if (ACTION[s,a] = reduce A->β) {
        pop |β| symbols of the stack;
        let state t now be on top of the stack;
        push GOTO[t,A] onto the stack;
        output the production A->β;
      } else if (ACTION[s,a]=accept) break; /* parsing is done */
      else call error-recovery routine;
}
```

Step 4: Parse the given input.

(0) E’->E
(1) E -> E + T
(2) E-> T
(3) T -> T * F
(4) T-> F
(5) F -> (E)
(6) F**->id**

id*id+id$

STATE	ACTON						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			

Line	Stack	Symbols	Input	Action
(1)	0		id*id+id$	Shift to 5
(2)	05	id	*id+id$	Reduce by F->id
(3)	03	F	*id+id$	Reduce by T->F
(4)	02	T	*id+id$	Shift to 7
(5)	027	T*	id+id$	Shift to 5
(6)	0275	T*id	+id$	Reduce by F->id
(7)	02710	T*F	+id$	Reduce by T->T*F
(8)	02	T	+id$	Reduce by E->T
(9)	01	E	+id$	Shift
(10)	016	E+	id$	Shift
(11)	0165	E+id	$	Reduce by F->id
(12)	0163	E+F	$	Reduce by T->F
(13)	0169	E+T`	$	Reduce by E->E+T
(14)	01	E	$	accept

Conflict Example

S → L=R

S → R

L→ *R

L → id

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

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.

Problem

FOLLOW(R)={=,$}

= shift 6

reduce by R → L

shift/reduce conflict

Action[2,=] = shift 6

Action[2,=] = reduce by R → L

[S ⇒L=R ⇒*R=R] so follow(R) contains, =

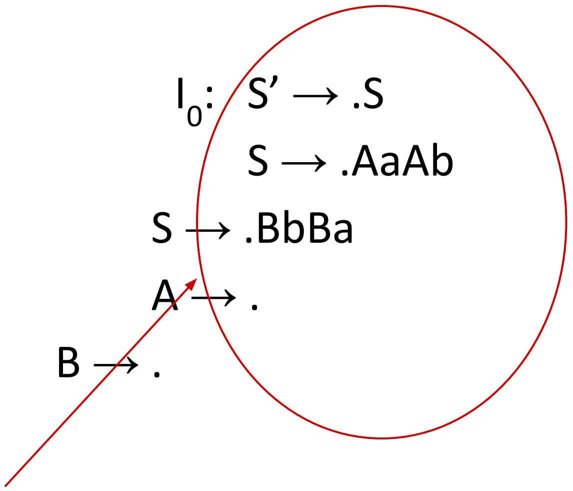
Conflict Example2

S → AaAb

S → BbBa

A → ε

B → ε



I_0: S’ → .S

S → .AaAb

S → .BbBa

A → .

B → .

Problem

FOLLOW(A)={a,b}

FOLLOW(B)={a,b}

areduce by A → ε

reduce by B → ε

reduce/reduce conflict

b reduce by A → ε

reduce by B → ε

reduce/reduce conflict

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 ∈.
  - 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 ($A \rightarrow \alpha.,a$), 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 $\{a_1,\ldots,a_n\} \subseteq$ 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→α**.**Bβ,a in closure(I) and B→γ is a production rule of G; then B→.γ,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

$A \rightarrow \alpha\mathbf{.}\beta,a_1$

…

$A \rightarrow \alpha\mathbf{.}\beta,a_n$

can be written as

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

Constructing LR(1) sets of items

```
SetOfItems Closure(I) {
      repeat
        for (each item [A->α.Bβ,a] in I)
                for (each production B->γ in G’)
                        for (each terminal b in First(βa))
                                add [B->.γ, b] to set I;
      until no more items are added to I;
      return I;
}

SetOfItems Goto(I,X) {
      initialize J to be the empty set;
      for (each item [A->α.Xβ,a] in I)
        add item [A->αX.β,a] to set J;
      return closure(J);
}

void items(G’){
      initialize C to Closure({[S’->.S,$]});
      repeat
        for (each set of items I in C)
                for (each grammar symbol X)
                        if (Goto(I,X) is not empty and not in C)
                                add Goto(I,X) to C;
      until no new sets of items are added to C;
}
```

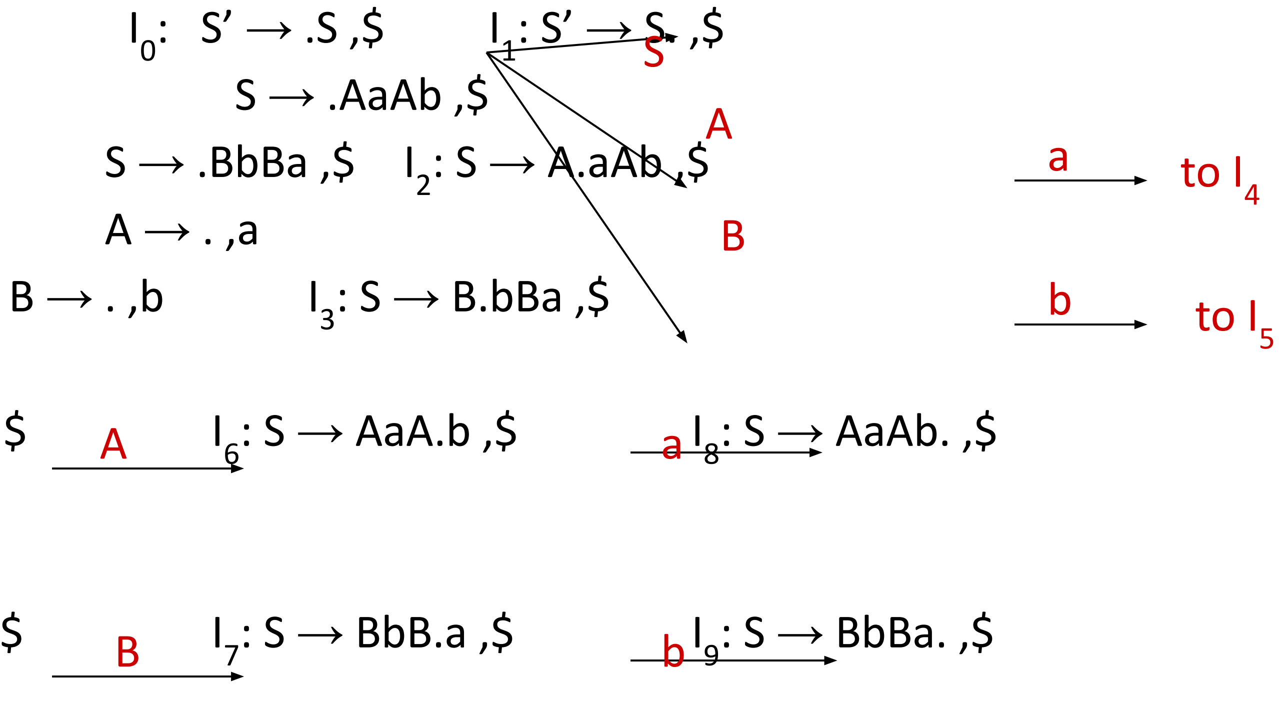
Canonical LR(1) Collection -- Example

S → AaAb

S → BbBa

A → ε

B → ε



I_0: S' → .S ,$
S → .AaAb ,$
S → .BbBa ,$
A → . ,a
B → . ,b

I_1: S' → S. ,$

I_2: S → A.aAb ,$ $\xrightarrow{a}$ to I_4

I_3: S → B.bBa ,$ $\xrightarrow{b}$ to I_5

I_4: S → Aa.Ab ,$
A → . ,b

$\xrightarrow{A}$ I_6: S → AaA.b ,$ $\xrightarrow{a}$ I_8: S → AaAb. ,$

I_5: S → Bb.Ba ,$
B → . ,a

$\xrightarrow{B}$ I_7: S → BbB.a ,$ $\xrightarrow{b}$ I_9: S → BbBa. ,$

An Example

0. $S' \rightarrow S$
1. $S \rightarrow C\ C$
2. $C \rightarrow c\ C$
3. $C \rightarrow d$

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

I_1: goto(I_0, S) = (S' → S · , \$)

I_2: goto(I_0, C) =
(S → C · C, \$)
(C → · c C, \$)
(C → · d, \$)

I_3: goto(I_0, c) =
(C → c · C, c/d)
(C → · c C, c/d)
(C → · d, c/d)

I_4: goto(I_0, d) =
(C → d ·, c/d)

I_5: goto(I_2, C) =
(S → C C ·, \$)

An Example

I_6: goto(I_2, c) =
(C → c · C, \$)
(C → · c C, \$)
(C → · d, \$)

I_7: goto(I_2, d) =
(C → d ·, \$)

I_8: goto(I_3, C) =
(C → c C ·, c/d)

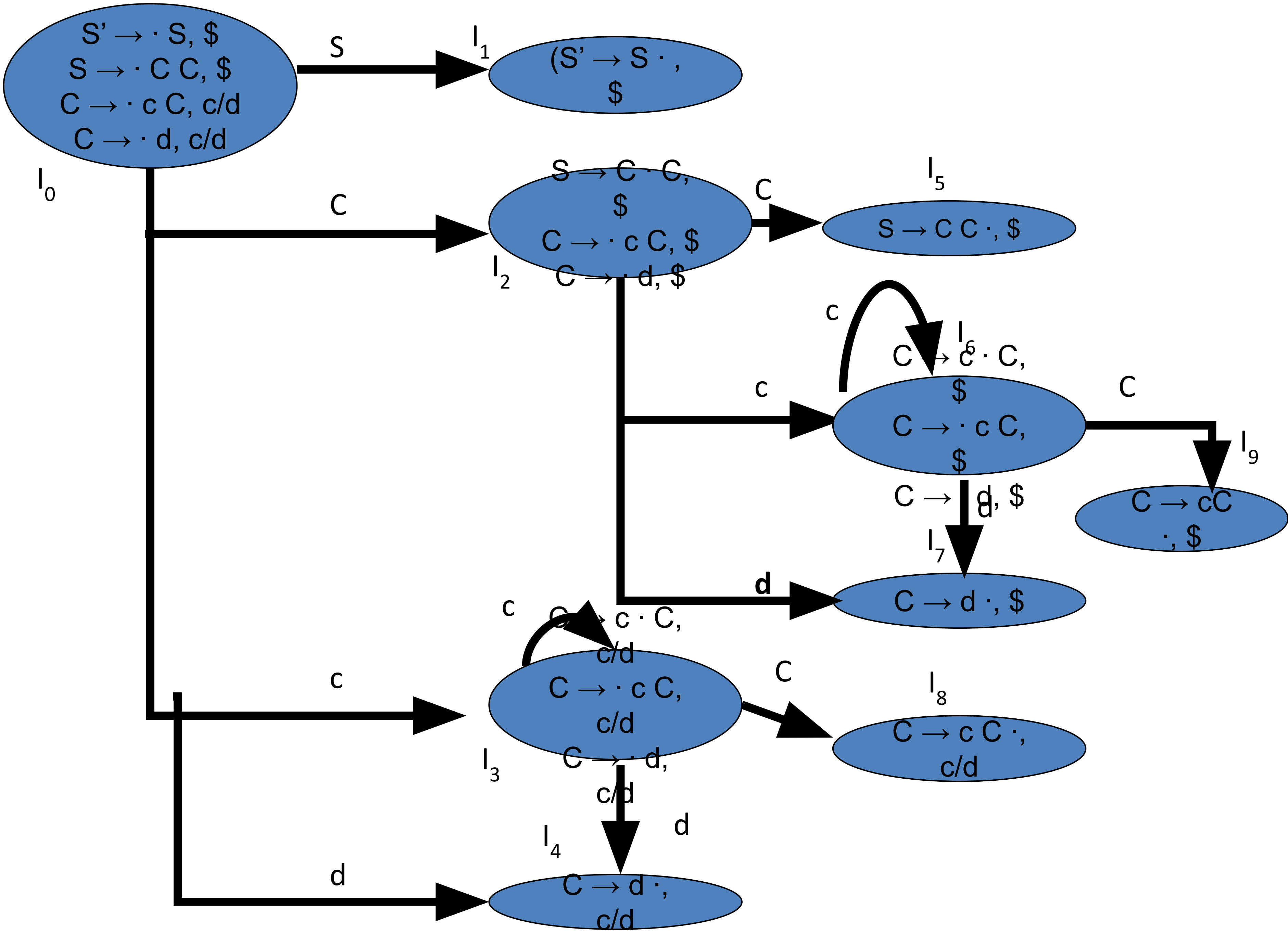
: goto(I_3, c) = I_3

: goto(I_3, d) = I_4

I_9: goto(I_6, C) =
(C → c C ·, \$)

: goto(I_6, c) = I_6

: goto(I_6, d) = I_7



S’ → · S, $
S → · C C, $
C → · c C, c/d
C → · d, c/d

(S’ → S · ,
$

S → C · C,
$
C → · c C, $
C → · d, $

S → C C ·, $

C → c · C,
$
C → · c C,
$
C → · d, $

C → cC
·, $

C → d ·, $

C → c · C,
c/d
C → · c C,
c/d
C → · d,
c/d

C → c C ·,
c/d

C → d ·,
c/d

Construction of LR(1) Parsing Tables

1. Construct the canonical collection of sets of LR(1) items for G'. $C \leftarrow \{I_0,...,I_n\}$
2. Create the parsing action table as follows
   - If a is a terminal, $A \rightarrow \alpha \bullet a\beta,b$ in I_i and $goto(I_i,a)=I_j$ then action[i,a] is ***shift j***.
   - If $A \rightarrow \alpha \bullet ,a$ is in I_i, then action[i,a] is ***reduce*** $\boldsymbol{A \rightarrow \alpha}$ where $A \neq S'$.
   - If $S' \rightarrow S \bullet ,\$$ 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		1	2
1			a		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

LALR Parsing Tables

1. **LALR** stands for **Lookahead LR.**
2. LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.
3. The number of states in SLR and LALR parsing tables for a grammar G are equal.
4. But LALR parsers recognize more grammars than SLR parsers.
5. ***yacc*** creates a LALR parser for the given grammar.
6. 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: $S \rightarrow L \bullet =R,\$$ □ $S \rightarrow L \bullet =R$ ← Core

$R \rightarrow L \bullet ,\$$ $R \rightarrow L \bullet$

- 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.

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

□ $L \rightarrow id \bullet ,\$$

$I_2{:}L \rightarrow id \bullet ,\$$ 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

INPUT: An Augmented Grammar G’.

OUTPUT: The LALR parsing table functions ACTION and GOTO for G’.

1. Construct $C=\{I_0,...,I_n\}$ the collection of sets of LR(1) items for G’.
2. 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.

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

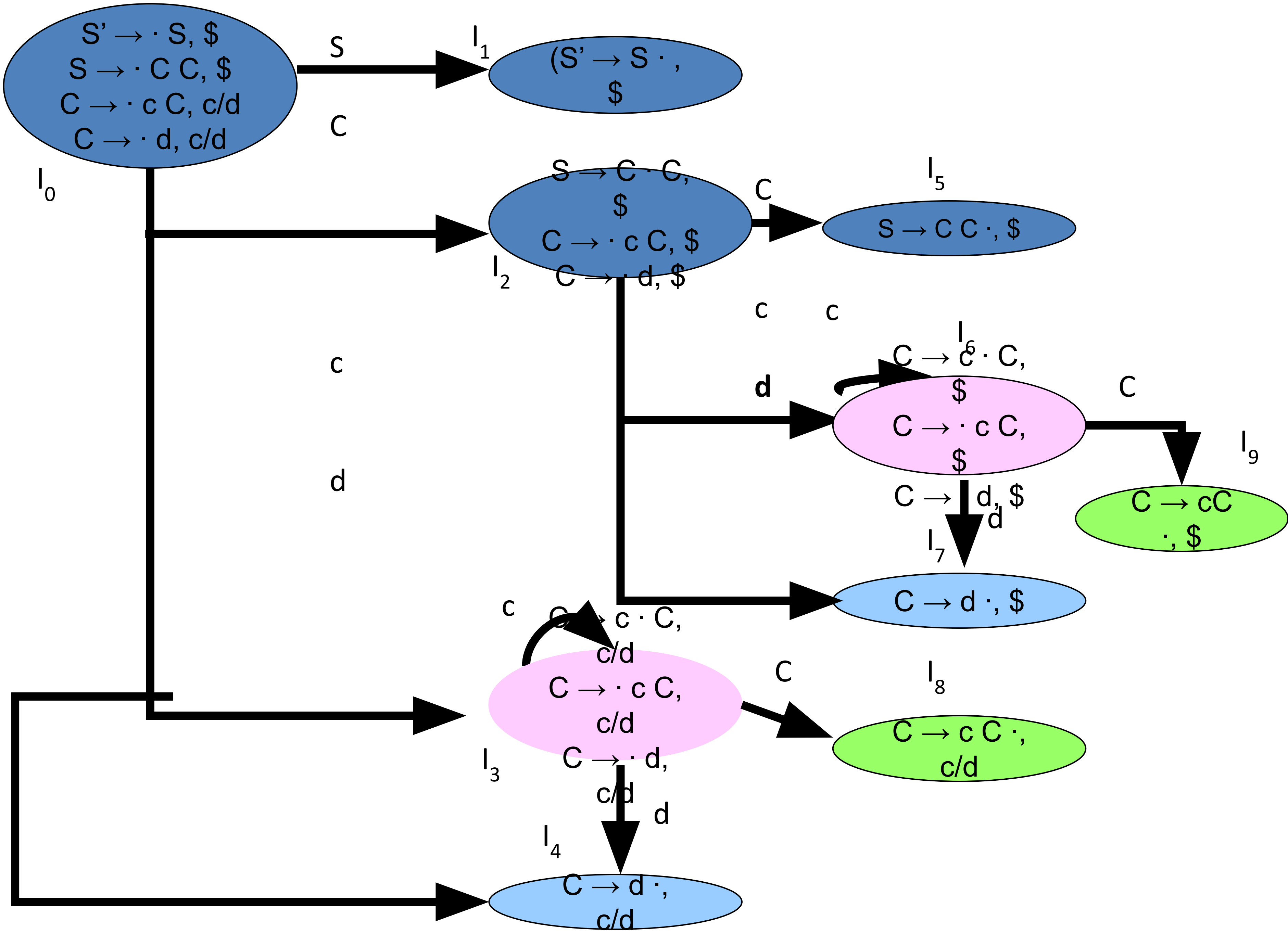
   $C'=\{J_1,...,J_m\}$ be the resulting sets of LR(1) items.
3. Create the parsing tables (**action and goto tables**) same as the construction of the parsing tables of LR(1) parser.

   If $\mathbf{J=I_1 \cup ... \cup I_k}$ 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 $goto(I_1,X)$.
4. 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)



I0

S’ → · S, $
S → · C C, $
C → · c C, c/d
C → · d, c/d

I1

(S’ → S · , $

I2

S → C · C, $
C → · c C, $
C → · d, $

I5

S → C C ·, $

I6

C → c · C, $
C → · c C, $
C → · d, $

I9

C → cC ·, $

I7

C → d ·, $

I3

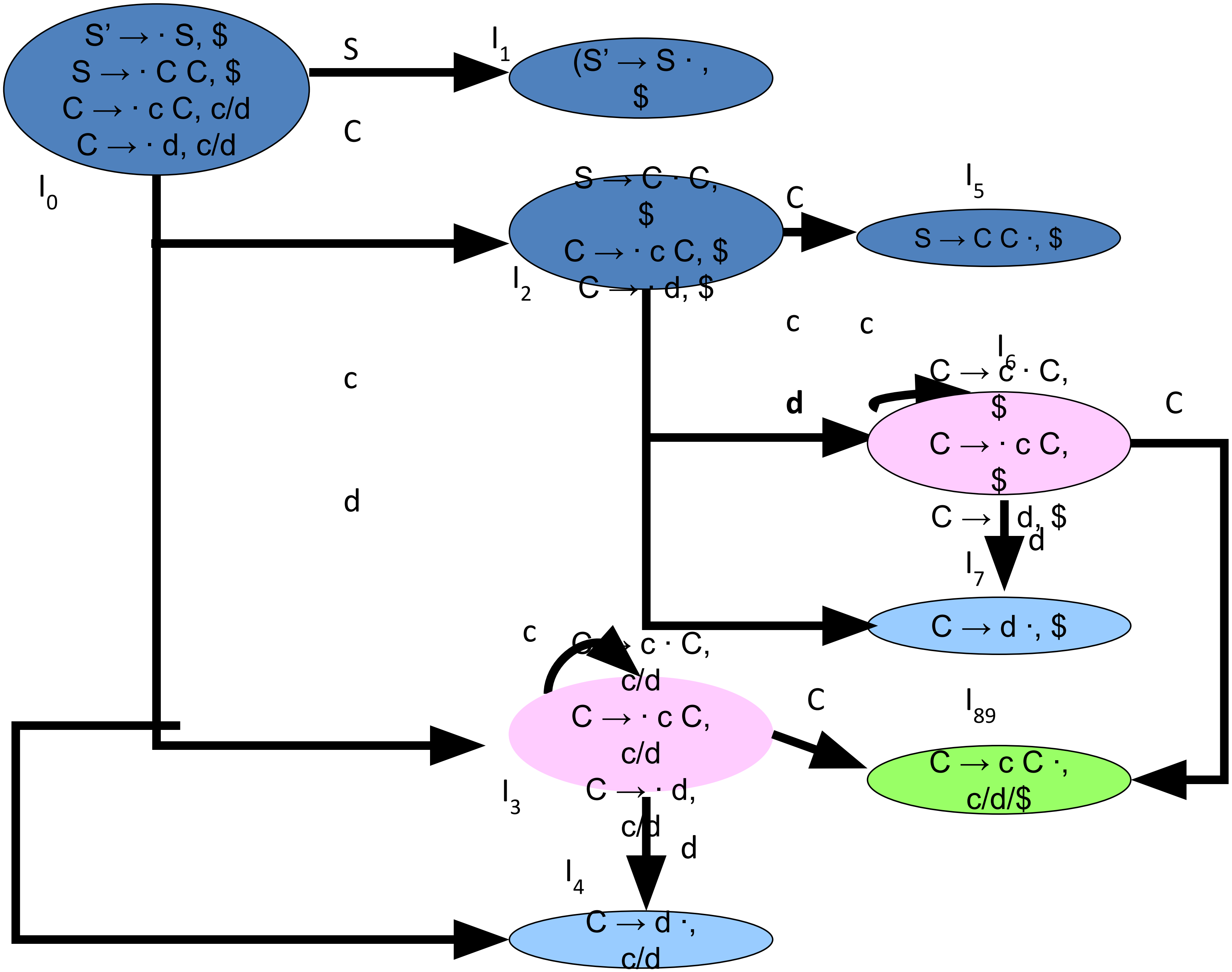
C → c · C, c/d
C → · c C, c/d
C → · d, c/d

I8

C → c C ·, c/d

I4

C → d ·, c/d



I_0

S’ → · S, $

S → · C C, $

C → · c C, c/d

C → · d, c/d

I_1

(S’ → S · , $

I_2

S → C · C, $

C → · c C, $

C → · d, $

I_5

S → C C ·, $

I_6

C → c · C, $

C → · c C, $

C → · d, $

I_7

C → d ·, $

I_3

C → c · C, c/d

C → · c C, c/d

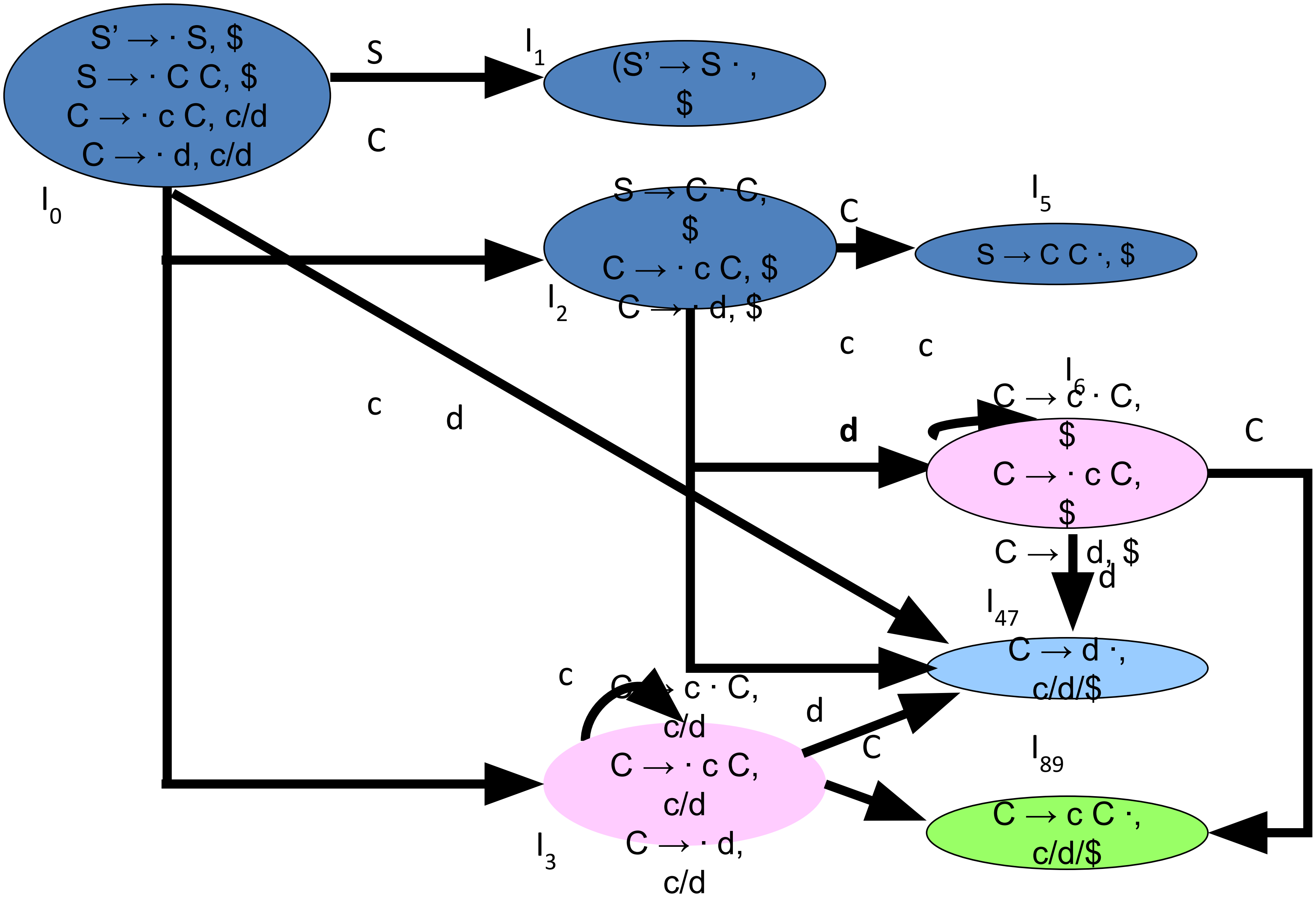
C → · d, c/d

I_{89}

C → c C ·, c/d/$

I_4

C → d ·, c/d



S' → · S, $
S → · C C, $
C → · c C, c/d
C → · d, c/d

(S' → S · ,
$

S → C · C,
$
C → · c C, $
C → · d, $

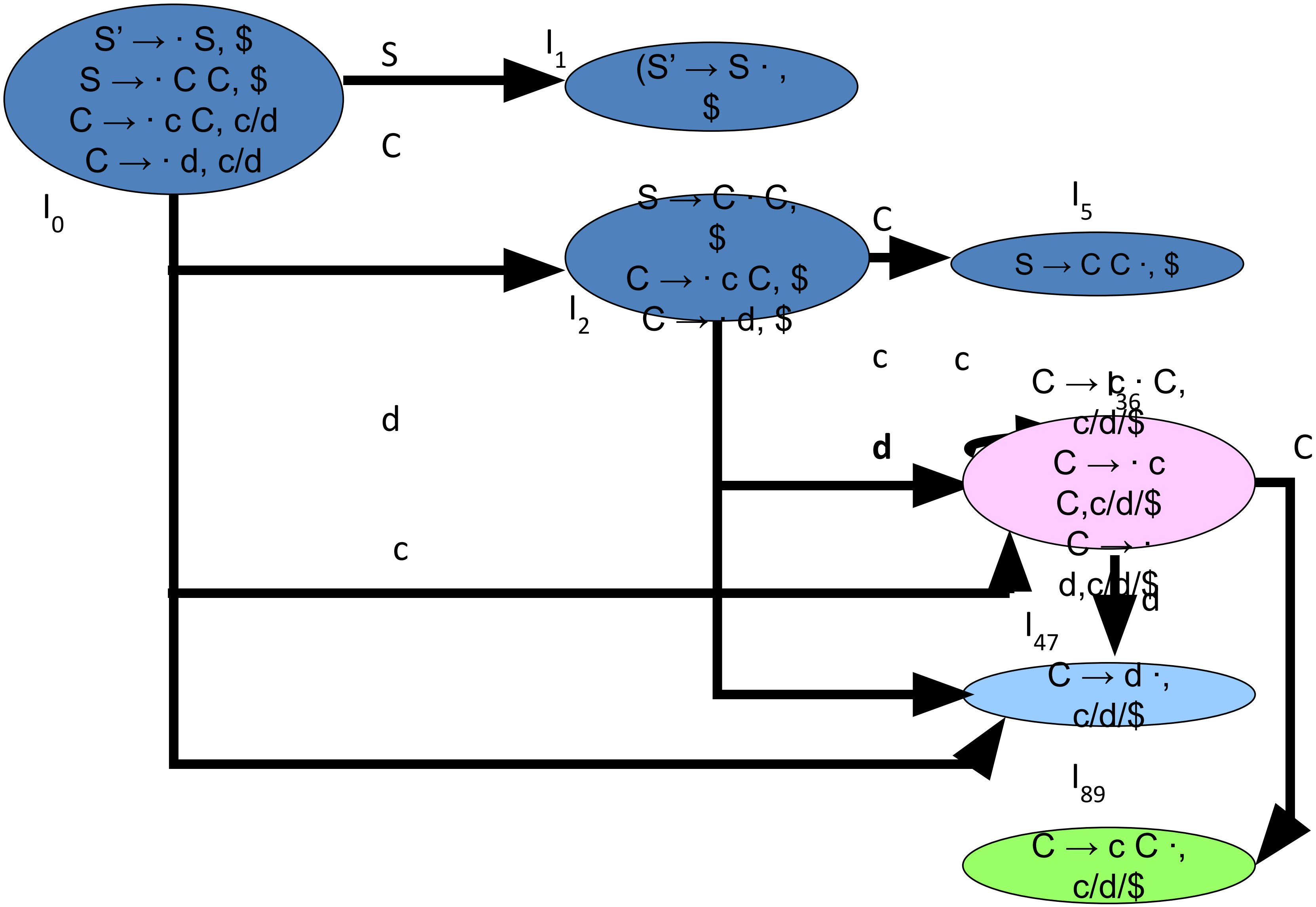
S → C C ·, $

C → c · C,
$
C → · c C,
$
C → · d, $

C → d ·,
c/d/$

C → c · C,
c/d
C → · c C,
c/d
C → · d,
c/d

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
c
C → c · C,
I36
c/d/$
C → · c
C,c/d/$
C → ·
d,c/d/$
d
d
C
d
c
I47
C → d ·,
c/d/$
I89
C → c C ·,
c/d/$

LALR Parse Table

START	Actions			goto	
	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		

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:

  $A \rightarrow \alpha \bullet ,a$ and $B \rightarrow \beta \bullet a\gamma ,b$
- This means that a state of the canonical LR(1) parser must have:

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

  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.

$$I_1 : \begin{array}{l} A \longrightarrow \alpha\bullet,a \\ B \longrightarrow \beta\bullet,b \end{array} \qquad\qquad I_2 : \begin{array}{l} A \longrightarrow \alpha\bullet,b \\ B \longrightarrow \beta\bullet,c \end{array}$$

$$\Downarrow$$

$$I_{12} : \begin{array}{l} A \longrightarrow \alpha\bullet,a/b \\ B \longrightarrow \beta\bullet,b/c \end{array}$$

□ reduce/reduce conflict

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 → E+T | T

E → E+E | E*E | (E) | id □ T → T*F | F

F → (E) | id

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 parser then stacks the 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

The End

Example

S’->S

S->CC

C->cC

C->d

Canonical LR(1) parsing table

- Method
  - Construct C={I0,I1, … , In}, the collection of LR(1) items for G’
  - State i is constructed from state Ii:
    - If [A->α.aβ, b] is in Ii and Goto(Ii,a)=Ij, then set ACTION[i,a] to “shift j”
    - If [A->α., a] is in Ii, then set ACTION[i,a] to “reduce A->α”
    - If {S’->.S,$] is in Ii, then set ACTION[I,$] to “Accept”
  - If any conflicts appears then we say that the grammar is not LR(1).
  - If GOTO(Ii,A) = Ij then GOTO[i,A]=j
  - All entries not defined by above rules are made “error”
  - The initial state of the parser is the one constructed from the set of items containing [S’->.S,$]

Example

S’->S

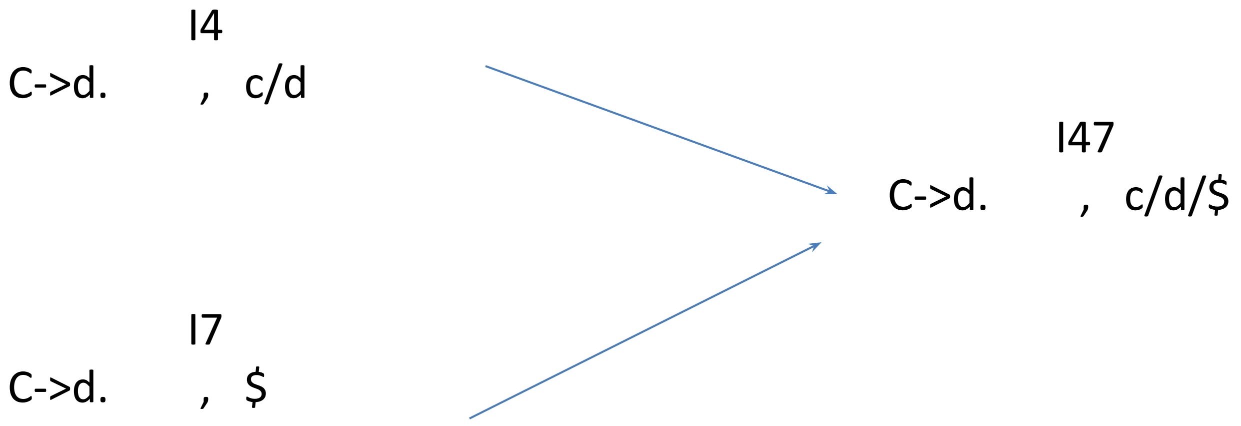
S->CC

C->cC

C->d

LALR Parsing Table

- For the previous example we had:



I4

C->d. , c/d

I47

C->d. , c/d/$

I7

C->d. , $

- State merges cant produce Shift-Reduce conflicts. Why?
- But it may produce reduce-reduce conflict

Example of RR conflict in state merging

S’->S

S -> aAd | bBd | aBe | bAe

A -> c

B -> c

An easy but space-consuming LALR table construction

- Method:
  1. Construct C={I0,I1,…,In} the collection of LR(1) items.
  2. For each core among the set of LR(1) items, find all sets having that core, and replace these sets by their union.
  3. Let C'={J0,J1,…,Jm} be the resulting sets. The parsing actions for state i, is constructed from Ji as before. If there is a conflict grammar is not LALR(1).
  4. If J is the union of one or more sets of LR(1) items, that is J = I1 UI2…IIk then the cores of Goto(I1,X), …, Goto(Ik,X) are the same and is a state like K, then we set Goto(J,X) =k.
- This method is not efficient, a more efficient one is discussed in the book

Compaction of LR parsing table

- Many rows of action tables are identical
  - Store those rows separately and have pointers to them from different states
  - Make lists of (terminal-symbol, action) for each state
  - Implement Goto table by having a link list for each nonterinal in the form (current state, next state)

Using ambiguous grammars

E->E+E

E->E*E

E->(E)

E->id

STATE	ACTON						GO TO
	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				
7		R1	S5		R1	R1	
8		R2	R2		R2	R2	
9		R3	R3		R3	R3	

I0: E’->.E
E->.E+E
E->.E*E
E->.(E)
E->.id

I1: E’->E.
E->E.+E
E->E.*E

I2: E->(.E)
E->.E+E
E->.E*E
E->.(E)
E->.id

I3: E->.id

I4: E->E+.E
E->.E+E
E->.E*E
E->.(E)
E->.id

I5: E->E*.E
E->(.E)
E->.E+E
E->.E*E
E->.(E)
E->.id

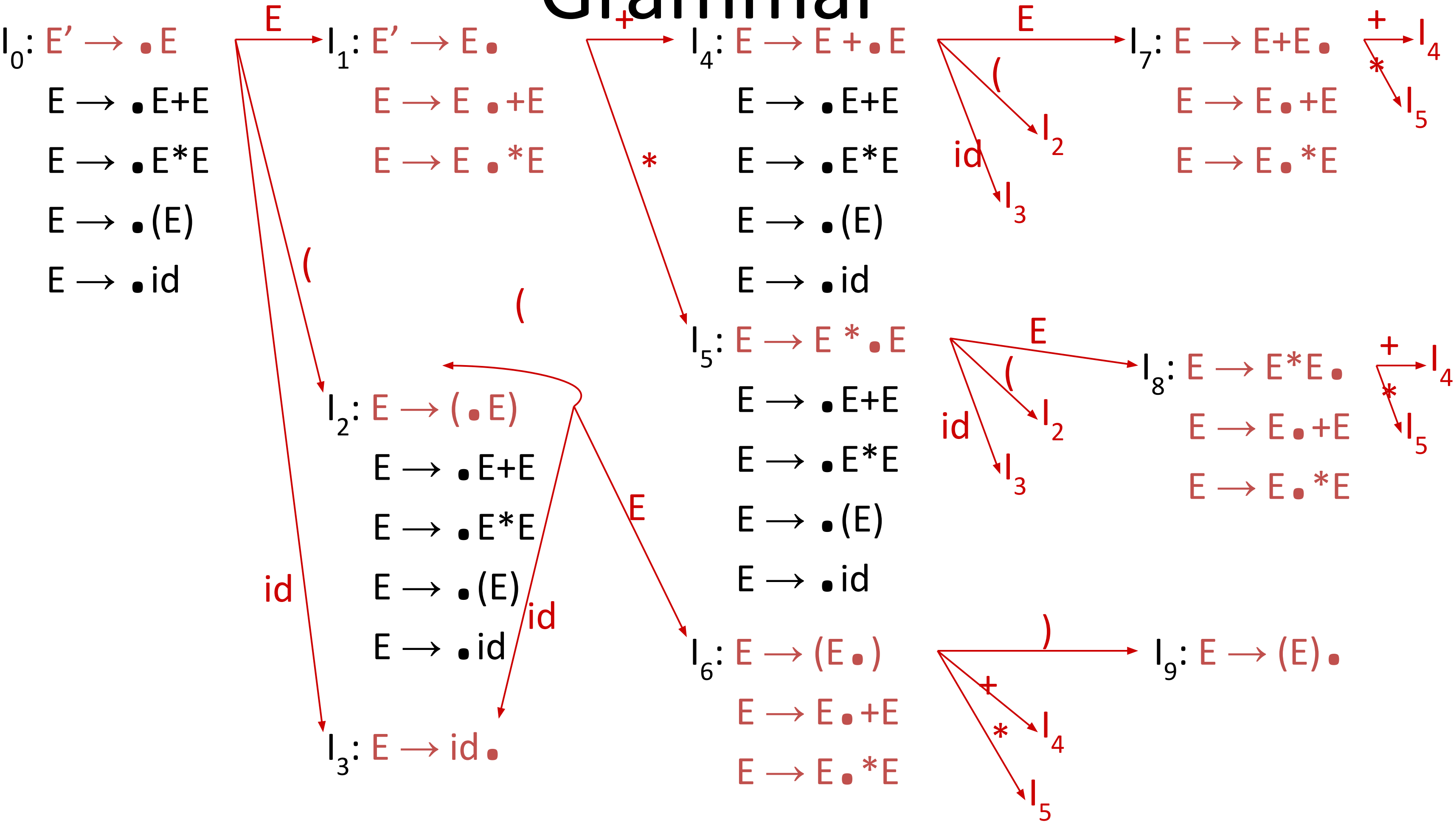
I6: E->(E.)
E->E.+E
E->E.*E

I7: E->E+E.
E->E.+E
E->E.*E

I8: E->E*E.
E->E.+E
E->E.*E

I9: E->(E).

Sets of LR(0) Items for Ambiguous Grammar



I_0: E' → •E
E → •E+E
E → •E*E
E → •(E)
E → •id

I_1: E' → E•
E → E •+E
E → E •*E

I_2: E → (•E)
E → •E+E
E → •E*E
E → •(E)
E → •id

I_3: E → id•

I_4: E → E +•E
E → •E+E
E → •E*E
E → •(E)
E → •id

I_5: E → E *•E
E → •E+E
E → •E*E
E → •(E)
E → •id

I_6: E → (E•)
E → E•+E
E → E•*E

I_7: E → E+E•
E → E•+E
E → E•*E

I_8: E → E*E•
E → E•+E
E → E•*E

I_9: E → (E)•

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 *.

$$I_0 \xrightarrow{E} I_1 \xrightarrow{*} I_5 \xrightarrow{E} I_8$$

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		