Bottom-Up Parsing

Thanks to Charles E. Hughes

Reductions

- Top-down focuses on producing an input string from the start symbol
- Bottom-up focuses on reducing the string to the start symbol
- By definition, reduction is the reverse of production

Handle Pruning

- Bottom-up reverses a rightmost derivation since rightmost rewrites the leftmost non-terminal last
- Bottom-up must identify a **handle** of a sentential form (a string of terminals and non-terminals derived from the start symbol), where the handle is the substring that was replaced at the last step in a rightmost derivation leading to this sentential form.
- A handle must match the body (rhs) of some production
- Formally, if $S \Rightarrow_{rm}^{*} \alpha A \omega \Rightarrow_{rm} \alpha\beta\omega$ where $A \rightarrow \beta$ then β, in the position following α, is a handle of $\alpha\beta\omega$
- We would like handles to be unique, and they are so in unambiguous grammars
- **Handle pruning** is the process of reducing a sentential form to a deriving sentential form by reversing the last production

shift/reduce Parsing

- This involves a stack that holds the left part of a sentential for with the input holding the right part
- Initially the stack has a bottom of stack marker and the input is the entire string to be parsed, plus an end marker
  Stack = $ Input = w$
- Our goal is to consume the string and end up with the start symbol on stack
  Stack = $S Input = $

shift/reduce Process

- The process is one where we can either
  - Shift the next input symbol onto stack
  - Reduce “handle” on top of stack
  - Accept if successfully get to start symbol with all input consumed
  - Error is a syntax error is discovered

Conflicts in shift/reduce

- Handle pruning can encounter two types of conflicts
  - **reduce/reduce** is when there are two possible reductions and we cannot decide which to use
  - **shift/reduce** conflict is when we cannot decided whether to shift or reduce

Classic shift/reduce

stmt → **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| **other**

Stack = **$… if** *expr* **then** *stmt*

Input = **else … $**

Should we shift **else** into stack or reduce??

Can prefer shift over reduce, but that may not work as a general policy

Classic reduce/reduce

If have two types of expression lists preceded by an id. One is array reference using parentheses and other is a function call. Both can appear by themselves.

Relevant rules are:

stmt	$\rightarrow$	**id (** *p_list* **)**
	|	*expr*
p_list	$\rightarrow$	*p_list parm* | *parm*
e_list	$\rightarrow$	*e_list parm* | *expr*
expr	$\rightarrow$	**id (** *e_list* **) | id**
parm	$\rightarrow$	**id**

Stack = **$...id(id** Input = **, id)…$**

Is this first *expr* or a *parm*?

One solution is that we differentiate **procid** from **id** in symbol table and hence via lexical analysis. Then the third symbol in stack, not part of handle, determines the reduction. The key is context.

Our Goal

Find a useful subset of context free grammars that

1. Covers all or at least most unambiguous CF languages
2. Is easy to recognize
3. Avoids conflicts without severely limiting expressiveness
4. Is amenable to a fast parsing algorithm

LR Parsing

LR Parsing

LR(k) parsing.

left to right scanning

right-most derivation

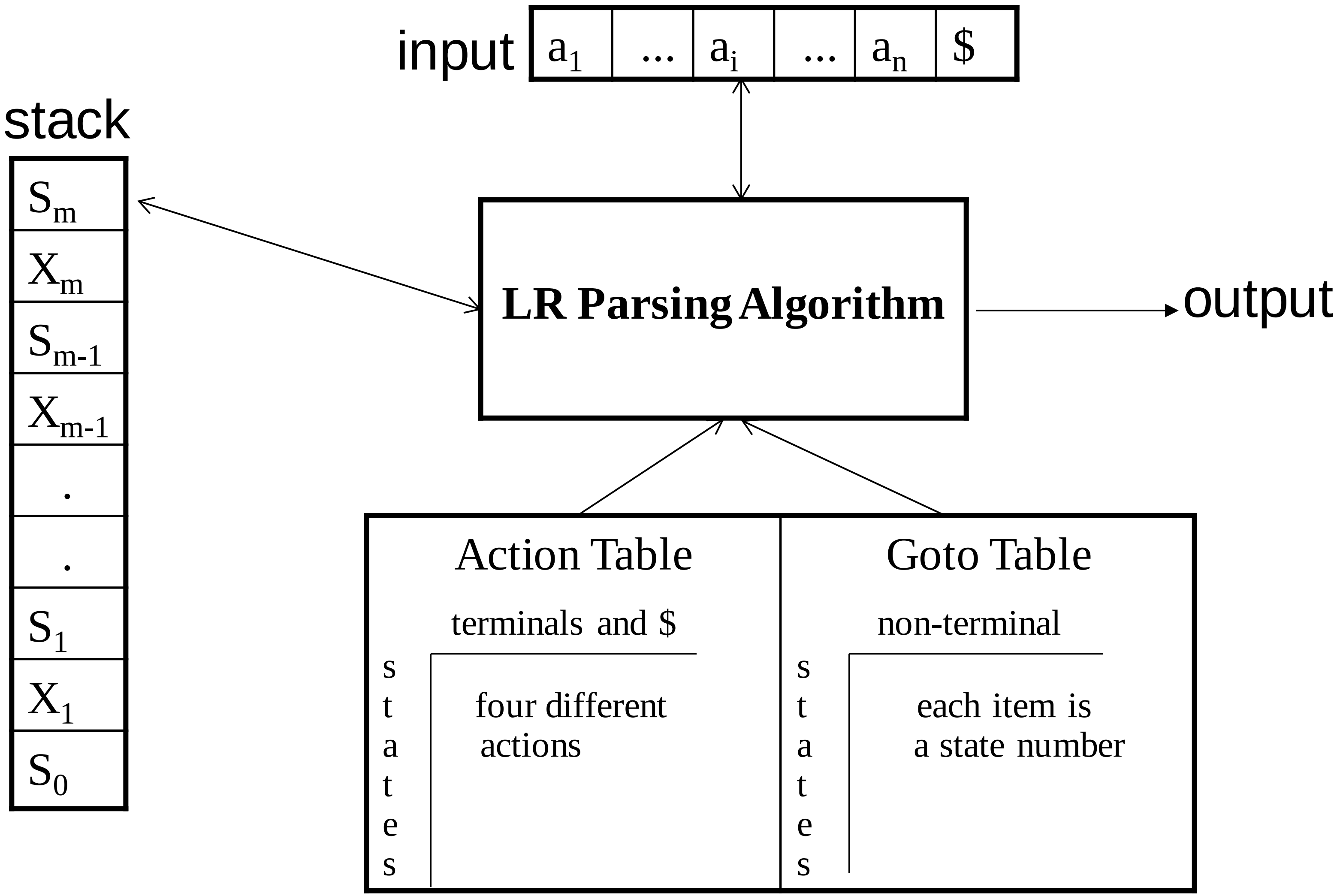
k lookahead

- LR is associated with bottom-up; LL with top-down
- LL(k), k>1, languages $\supset$ LL(k-1) languages
- LR(1) languages $\supset$ LL(k) languages, $k \geq 0$
- LR(k), k>1, languages = LR(1) languages
- However, LR(k), k>1, grammars $\supset$ LR(k-1) grammars
- LR grammars can find errors quickly, but they do not always have good context to recover

LR Parser Types

- SLR – simple LR parser
- LALR –look-head LR parser
- LR – most general LR parser
- SLR, LALR and LR are closely related
  - The parsing algorithm is the same
  - Their parsing tables are different

LR Parsing Algorithm



Configuration of LR Algorithm

- A configuration of a LR parsing is:

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

Stack — Rest of Input

- S_m and a_i decide 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 LR-Parser

1. **shift s** -- shifts the next input symbol onto the stack. Shift is performed only if **action[s_m,a_i] = sk**, where k is the new state. In this case

   (S_o X_1 S_1 ... X_m S_m, a_i a_{i+1} ... a_n \$) ➔ (S_o X_1 S_1 ... X_m S_m a_i k, a_{i+1} ... a_n \$)

2. **reduce A→β** (if **action[s_m,a_i] = rn** where n is a production number)
   - pop 2|β| items from the stack;
   - then push **A** and **k** where **k=goto[$s_{m-|\beta|}$,A]**

(S_o X_1 S_1 ... X_m S_m, a_i a_{i+1} ... a_n \$) ➔ (S_o X_1 S_1 ... $X_{m-|\beta|}$ $S_{m-|\beta|}$ A k, a_i ... a_n \$)

   - Output is the reducing production reduce A→β or the associated semantic action or both

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

Reduce Action

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

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

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

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

$$X_1 \ldots X_{m-j}\ A\ a_i \ldots a_n\ \$ \Rightarrow X_1 \ldots X_{m-j}\ Y_1 \ldots Y_j\ a_i\ a_{i+1} \ldots a_n\ \$$$

Expression Grammar

Example: Given the grammar:

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

Compute Follow.

	Follow
E	{ **)**, **+**, **$** }
T	{ **)** , *****, **+**, **$** }
F	{ **)** , ***** , **+**, **$** }

SLR Parsing Tables

- An **LR(0) item** of a grammar G is a production of G with a dot at some position of the right side.
- Ex: $A \rightarrow aBb$ LR(0) Items:
  - $A \rightarrow \blacksquare aBb$
  - $A \rightarrow a \blacksquare Bb$
  - $A \rightarrow aB \blacksquare b$
  - $A \rightarrow aBb \blacksquare$
- Sets of LR(0) items will be the states of action and goto tables 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 $\mathbf{A} \rightarrow \alpha \blacksquare \mathbf{B}\beta$ is in ***closure(I)*** and $\mathbf{B} \rightarrow \gamma$ is a production rule of G; then $\mathbf{B} \rightarrow \blacksquare \gamma$ will be in the ***closure(I)***. We will apply this rule until no more new LR(0) items can be added to ***closure(I)***.

Closure Example

E' → E	closure({E' → ▪E}) =
E → E+T	{ E' → ▪E kernel item
E → T	E → ▪E+T
T → T*F	E → ▪T
T → F	T → ▪T*F
F → (E)	T → ▪F
F → id	F → ▪(E)
	F → ▪id }

Closure Algorithm

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

Goto Function

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

If I is the set of items that are valid for some viable prefix γ, then goto(I,X) is the set of items that are valid for the viable prefix γX.

Example:

I ={ E' → ▪E, E → ▪E+T, E → ▪T,
T → ▪T*F, T → ▪F, F → ▪(E), F → ▪id }

goto(I,E) = { E' → E▪ , E → E▪ +T }

goto(I,T) = { E → T▪ , T → T▪ *F }

goto(I,F) = {T → F▪ }

goto(I,() = { F → (▪E), E → ▪E+T, E → ▪T, T → ▪T*F, T → ▪F,
F → ▪(E), F → ▪id }

goto(I,id) = { F → id▪ }

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***
- The goto function is a deterministic FSA (finite state automaton), DFA, on the sets in C.

Canonical LR(0) 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

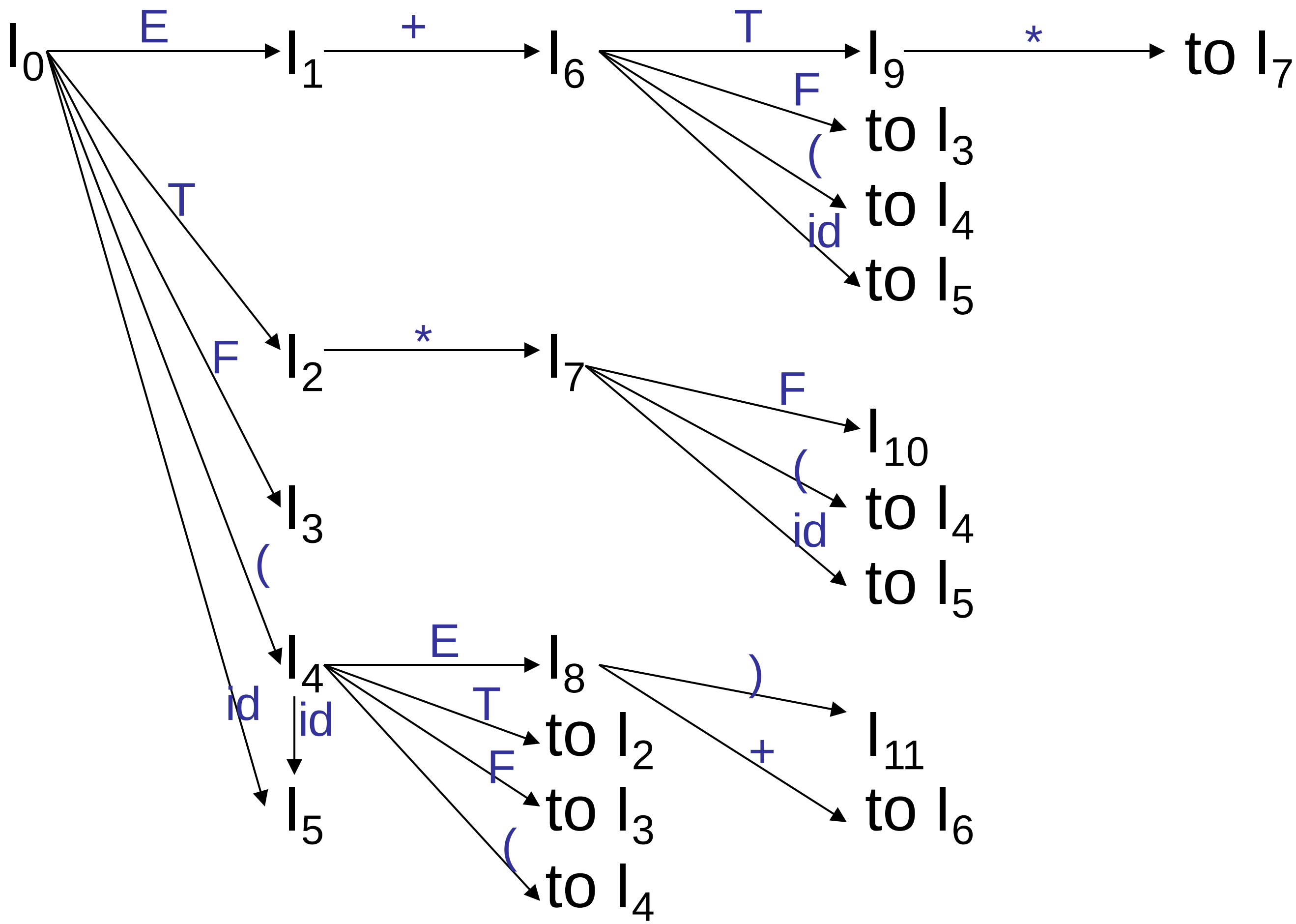
I_8: F → (E.)
E → E.+T

I_9: E → E+T.
T → T.*F

I_{10}: T → T*F.

I_{11}: F → (E).

DFA of Goto Function



Compute SLR Parsing Table

1. Construct the canonical collection of sets of LR(0) items for G'.

   C←{I_0,...,I_n}

2. Create the parsing action table as follows
   - If a is a terminal, **A→α.aβ** in I_i and goto(I_i,a)=I_j then action[i,a] is ***shift j***.
   - If **A→α.** is in I_i , then action[i,a] is ***reduce A→α*** for all a in **FOLLOW(A)** where **A≠S'**.
   - If **S'→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'→.S

(SLR) Parsing Tables

0) $E' \rightarrow E$
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 SLR-Parser

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	

SLR(1) Grammar

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

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 1

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
shift/reduce conflict

Action[2,=] = shift 6
Action[2,=] = reduce by R → L
[S ⇒L=R ⇒*R=R] so follow(R) contains =

Conflict Example2

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

$I_0: S' \rightarrow .S$

$S \rightarrow .AaAb$

$S \rightarrow .BbBa$

$A \rightarrow .$

$B \rightarrow .$

Problem

FOLLOW(A)={a,b}

FOLLOW(B)={a,b}

a → reduce by $A \rightarrow \varepsilon$

a → reduce by $B \rightarrow \varepsilon$

reduce/reduce conflict

b → reduce by $A \rightarrow \varepsilon$

b → reduce by $B \rightarrow \varepsilon$

reduce/reduce conflict

SLR Weakness

- In SLR method, state i makes a reduction by $A \rightarrow \alpha$ when the current token is **a**:
  - if $A \rightarrow \alpha.$ is in I_i and **a** is in 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 stack top. This means that making reduction in this case is not correct.

LR(1) Item

- To avoid some 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 an LR(1) item.
  - 1 refers to the length of the second component
  - The lookahead has no effect on an item of the form $[A \rightarrow \alpha.\beta,a]$, where β is not empty.
  - 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), and could be proper.

LR(1) Item (cont.)

- A state will contain $A \rightarrow \alpha\,\textbf{.}\,,a_1$ where $\{a_1,...,a_n\} \subseteq$ FOLLOW(A)

  ...

  $A \rightarrow \alpha\,\textbf{.}\,,a_n$
- When β is empty ($A \rightarrow \alpha.,a_1/a_2/ .. /a_n$), we do the reduction by $A\rightarrow\alpha$ only if the next input symbol is in the set $\{a_1,a_2, .. ,a_n\}$

  (not for any terminal in FOLLOW(A) as with SLR).

Canonical Collection

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

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.

Short Notation

- 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

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

Canonical LR(1) Collection

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

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

S → I_1: $S' \to S.$,\$

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

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

I_4: $S \to Aa.Ab$,\$
$A \to .$,b

A → I_6: $S \to AaA.b$,\$ — b → I_8: $S \to AaAb.$,\$

I_5: $S \to Bb.Ba$,\$
$B \to .$,a

B → I_7: $S \to BbB.a$,\$ — a → I_9: $S \to BbBa.$,\$

An Example

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

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)
: goto(I_3, c) = I_3
: goto(I_3, d) = I_4

I_4: goto(I_0, d) =
(C → d •, c/d)

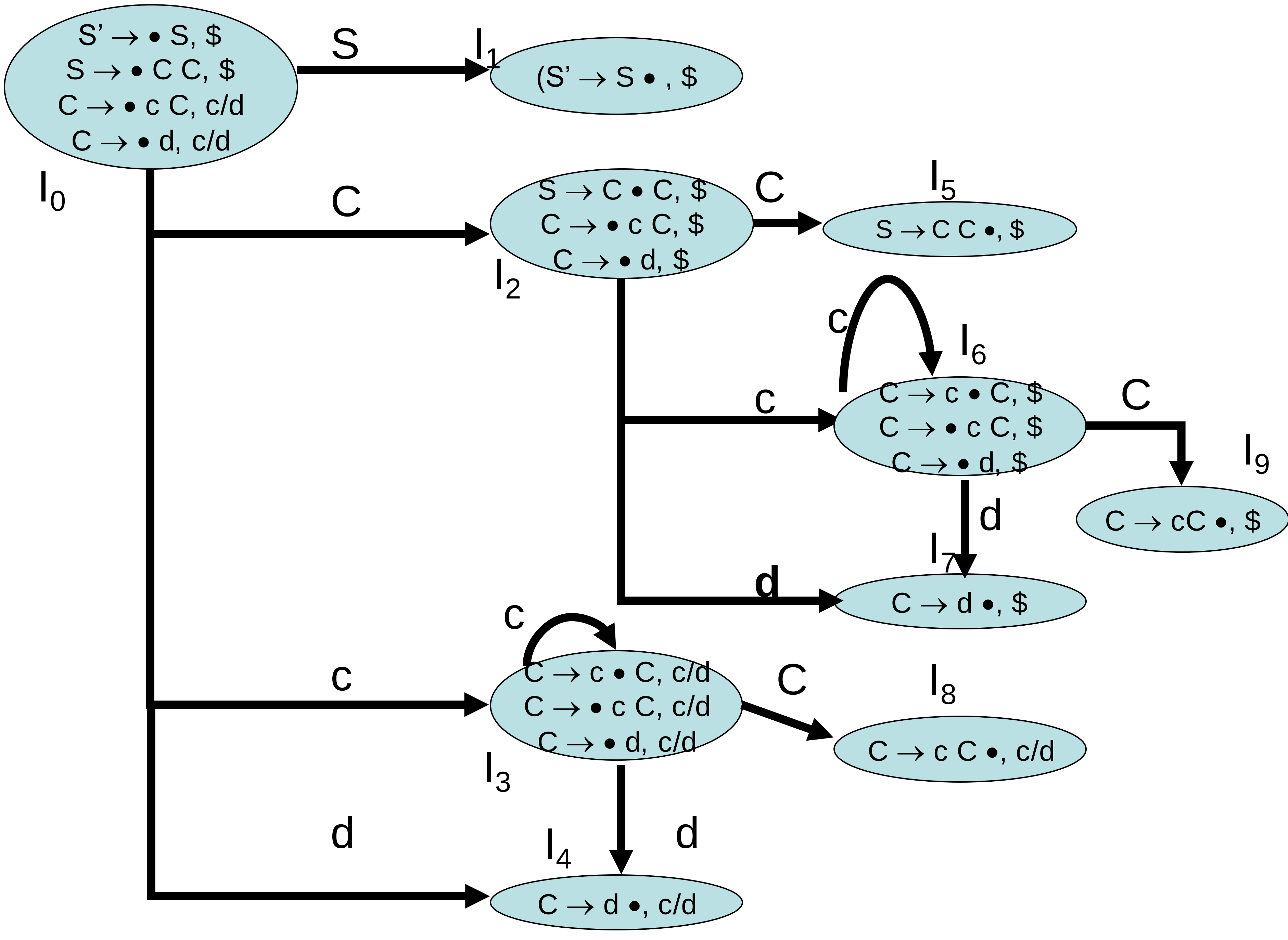
I_5: goto(I_2, C) =
(S → C C •, \$)

I_6: goto(I_2, c) =
(C → c • C, \$)
(C → • c C, \$)
(C → • d, \$)
: goto(I_6, c) = I_6
: goto(I_6, d) = I_7

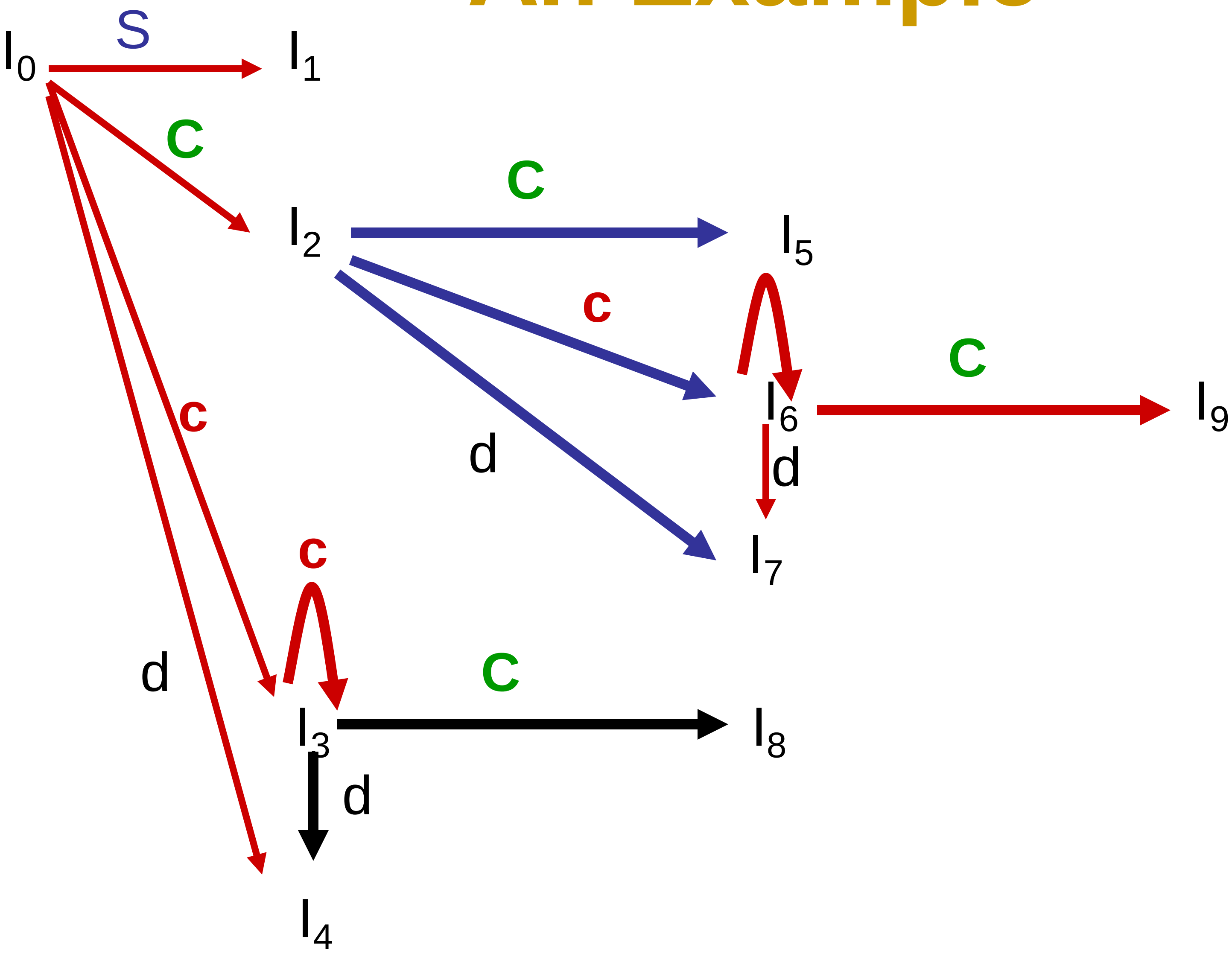
I_7: goto(I_2, d) =
(C → d •, \$)

I_8: goto(I_3, C) =
(C → c C •, c/d)

I_9: goto(I_6, C) =
(C → 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, $
C → • d, $
C
I9
C → cC •, $
d
I7
d
C → d •, $
c
c
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
I3
C
I8
C → c C •, c/d
d
I4
d
C → d •, c/d

An Example



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		

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

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

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, $\mathbf{A \rightarrow \alpha \blacksquare a\beta, b}$ in I_i and goto(I_i,**a**)=I_j then action[i,**a**] is ***shift j*.**
   - If $\mathbf{A \rightarrow \alpha ., a}$ is in I_i , then action[i,**a**] is ***reduce*** $\boldsymbol{A \rightarrow \alpha}$ where $\mathbf{A \neq S'}$.
   - If $\mathbf{S' \rightarrow S ., \$}$ is in I_i , then action[i,**$**] is ***accept***.
   - If any conflicting actions are 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 $\mathbf{S' \rightarrow .S, \$}$

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. ***Bison*** creates a LALR parser for the given grammar.

6. A state of an LALR parser will again be 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 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

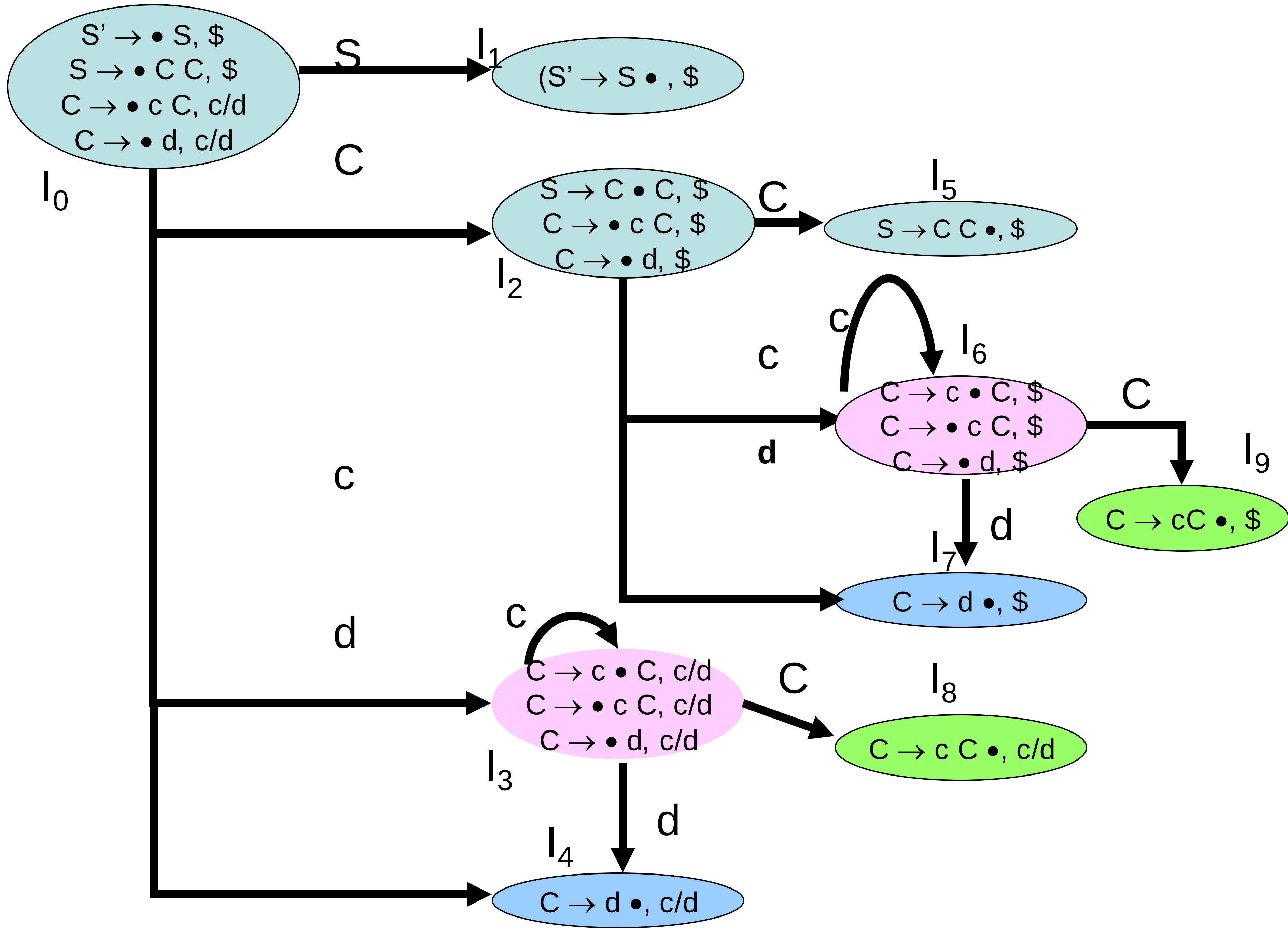
- We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.
- In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.

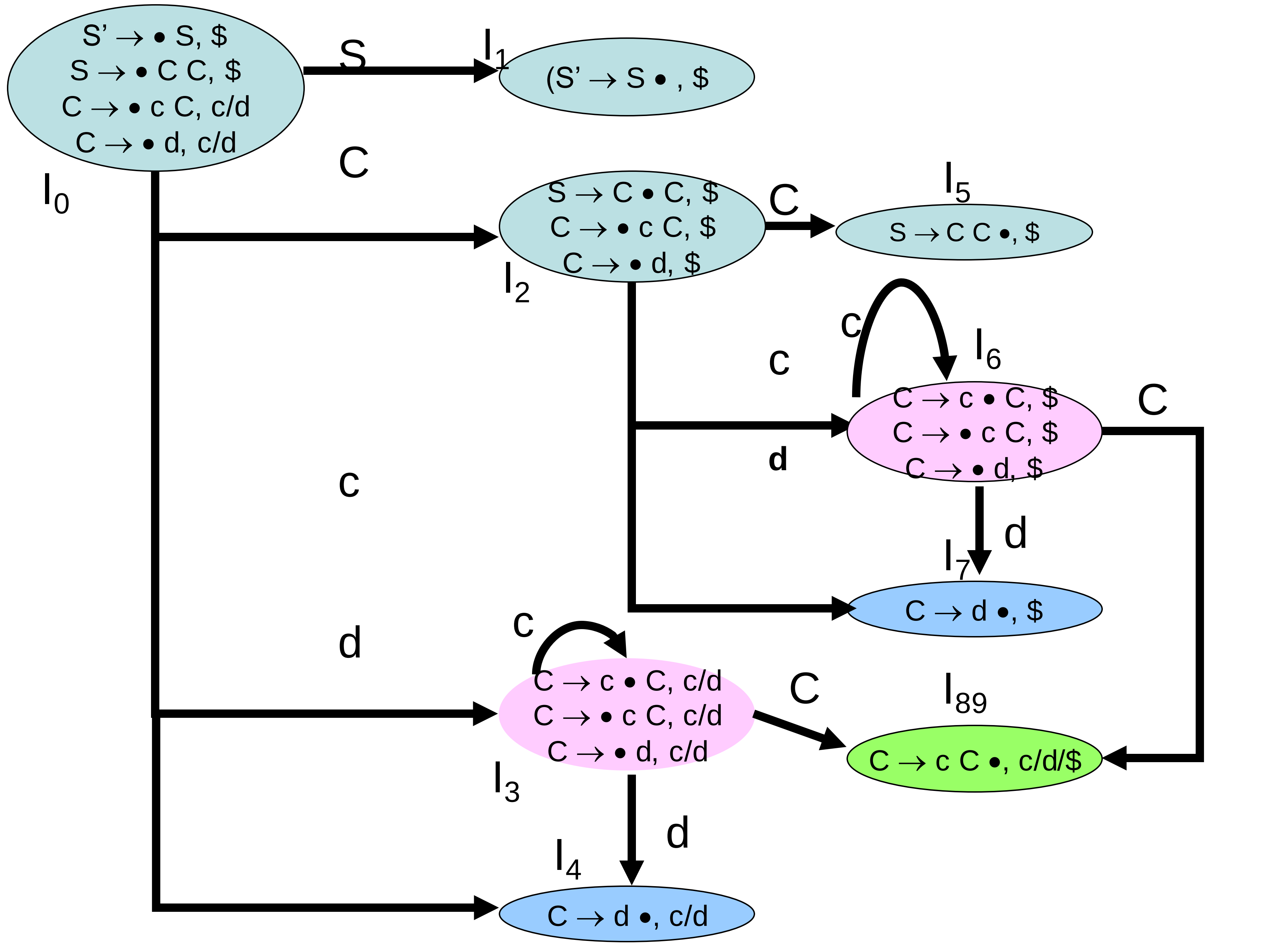
Creation of LALR Parsing Tables

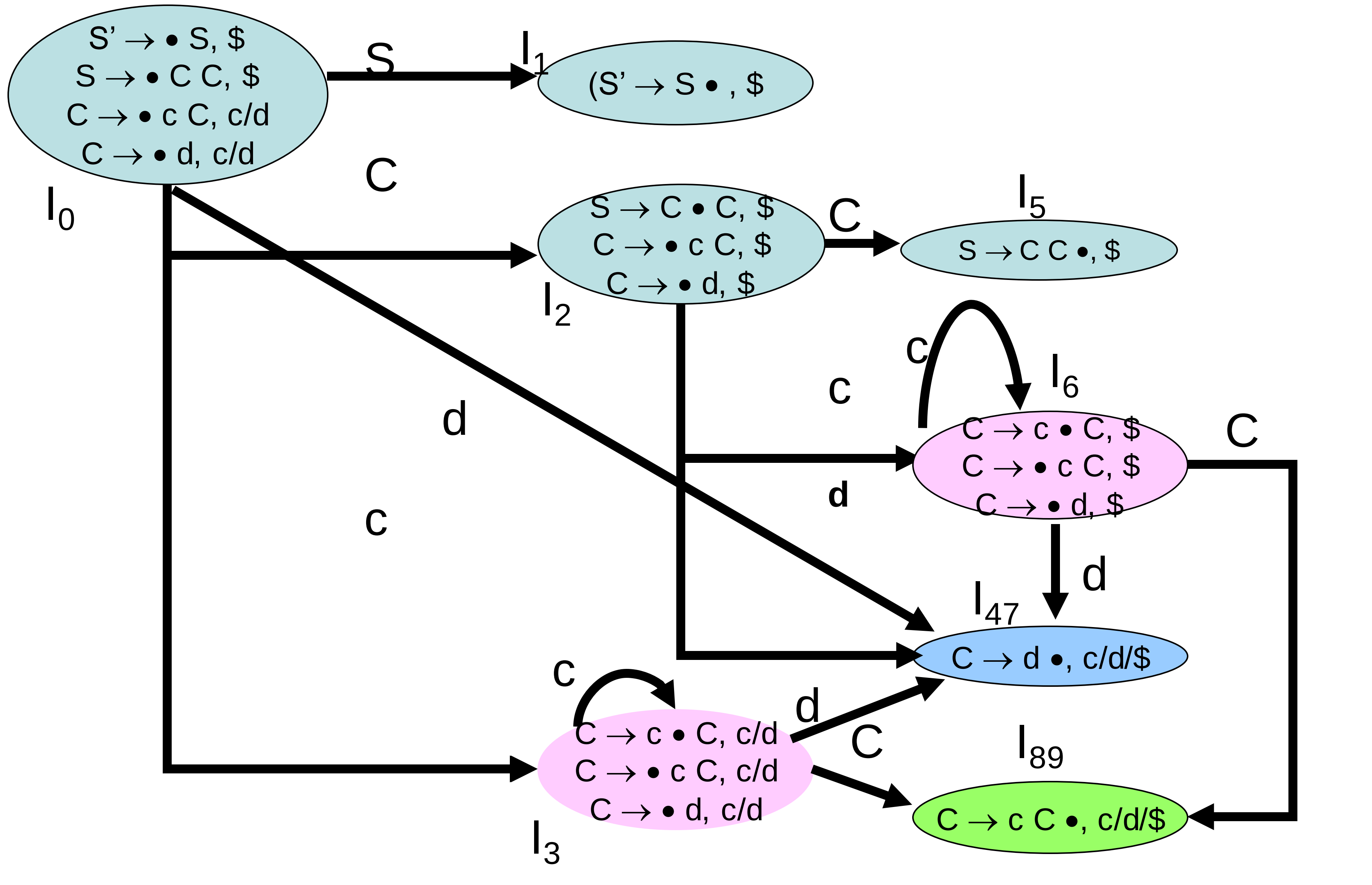
1. Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar.
2. For each core present; find all sets having that same core; replace those sets having same cores with a single set which is their union.

   $C=\{I_0,...,I_n\}$ ➔ $C'=\{J_0,...,J_m\}$ where $m \leq n$
3. Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.
   1. Note that: If $J=I_{i1} \cup ... \cup I_{ik}$ since $I_{i1},...,I_{ik}$ have same cores

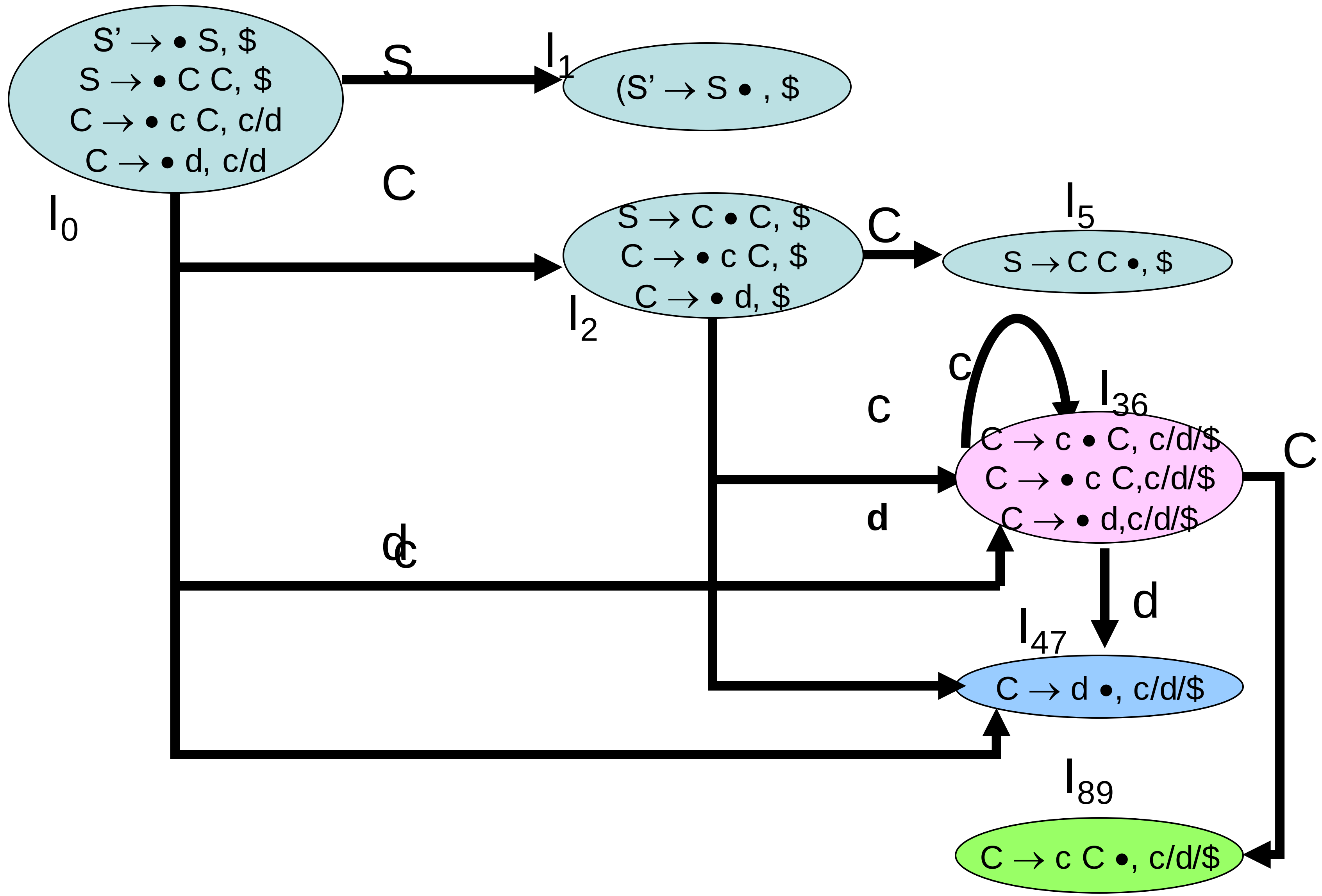
      ➔ cores of $goto(I_{i1},X),...,goto(I_{ik},X)$ must be same.
   2. So, $goto(J,X)=K$ where K is the union of all sets of items having same cores as $goto(I_{i1},X)$.
4. If no conflict is introduced, the grammar is LALR(1) grammar. (We may only introduce reduce/reduce conflicts; we cannot introduce a shift/reduce conflict)


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
I6
C → c • C, $
C → • c C, $
C → • d, $
d
C
I9
C → cC •, $
d
I7
C → d •, $
c
d
c
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
I3
C
I8
C → c C •, c/d
d
I4
C → d •, 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
I6
C → c • C, $
C → • c C, $
C → • d, $
d
C
d
I7
C → d •, $
c
d
c
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
I3
C
I89
C → c C •, c/d/$
d
I4
C → d •, 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
I6
C → c • C, $
C → • c C, $
C → • d, $
d
C
d
c
d
I47
C → d •, c/d/$
c
d
C
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
I3
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, c/d/$
C → • c C,c/d/$
C → • d,c/d/$
C
d
d
c
d
I47
C → d •, c/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		

Shift/Reduce Conflict

- We said 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:

$$A \rightarrow \alpha \centerdot ,a \qquad \text{and} \qquad B \rightarrow \beta \centerdot a\gamma,b$$

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

$$A \rightarrow \alpha \centerdot ,a \qquad \text{and} \qquad B \rightarrow \beta \centerdot a\gamma,c$$

But, this state also has 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.

$I_1 : A \rightarrow \alpha_{\blacksquare}, a$ $\qquad$ $I_2 : A \rightarrow \alpha_{\blacksquare}, b$

$B \rightarrow \beta_{\blacksquare}, b$ $\qquad$ $B \rightarrow \beta_{\blacksquare}, c$

$$\Downarrow$$

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

$B \rightarrow \beta_{\blacksquare}, b/c$

Canonical LALR(1)– Ex2

$S' \rightarrow S$
1) $S \rightarrow L=R$
2) $S \rightarrow R$
3) $L \rightarrow *R$
4) $L \rightarrow id$
5) $R \rightarrow L$

I_0: $S' \rightarrow \bullet S,\$$
$S \rightarrow \bullet L=R,\$$
$S \rightarrow \bullet R,\$$
$L \rightarrow \bullet *R,\$/=$
$L \rightarrow \bullet id,\$/=$
$R \rightarrow \bullet L,\$$

S: I_1: $S' \rightarrow S \bullet,\$$

L: I_2: $S \rightarrow L \bullet =R,\$$
$R \rightarrow L \bullet,\$$
→ to I_6

R: I_3: $S \rightarrow R \bullet,\$$

*: I_{411}: $L \rightarrow * \bullet R,\$/=$
$R \rightarrow \bullet L,\$/=$
$L \rightarrow \bullet *R,\$/=$
$L \rightarrow \bullet id,\$/=$

R: to I_{713}
L: to I_{810}
*: to I_{411}
id: to I_{512}

id: I_{512}: $L \rightarrow id \bullet,\$/=$

I_6: $S \rightarrow L= \bullet R,\$$
$R \rightarrow \bullet L,\$$
$L \rightarrow \bullet *R,\$$
$L \rightarrow \bullet id,\$$

R: to I_9
L: to I_{810}
*: to I_{411}
id: to I_{512}

I_9: $S \rightarrow L=R \bullet,\$$

I_{713}: $L \rightarrow *R \bullet,\$/=$

I_{810}: $R \rightarrow L \bullet,\$/=$

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– (for Ex2)

	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

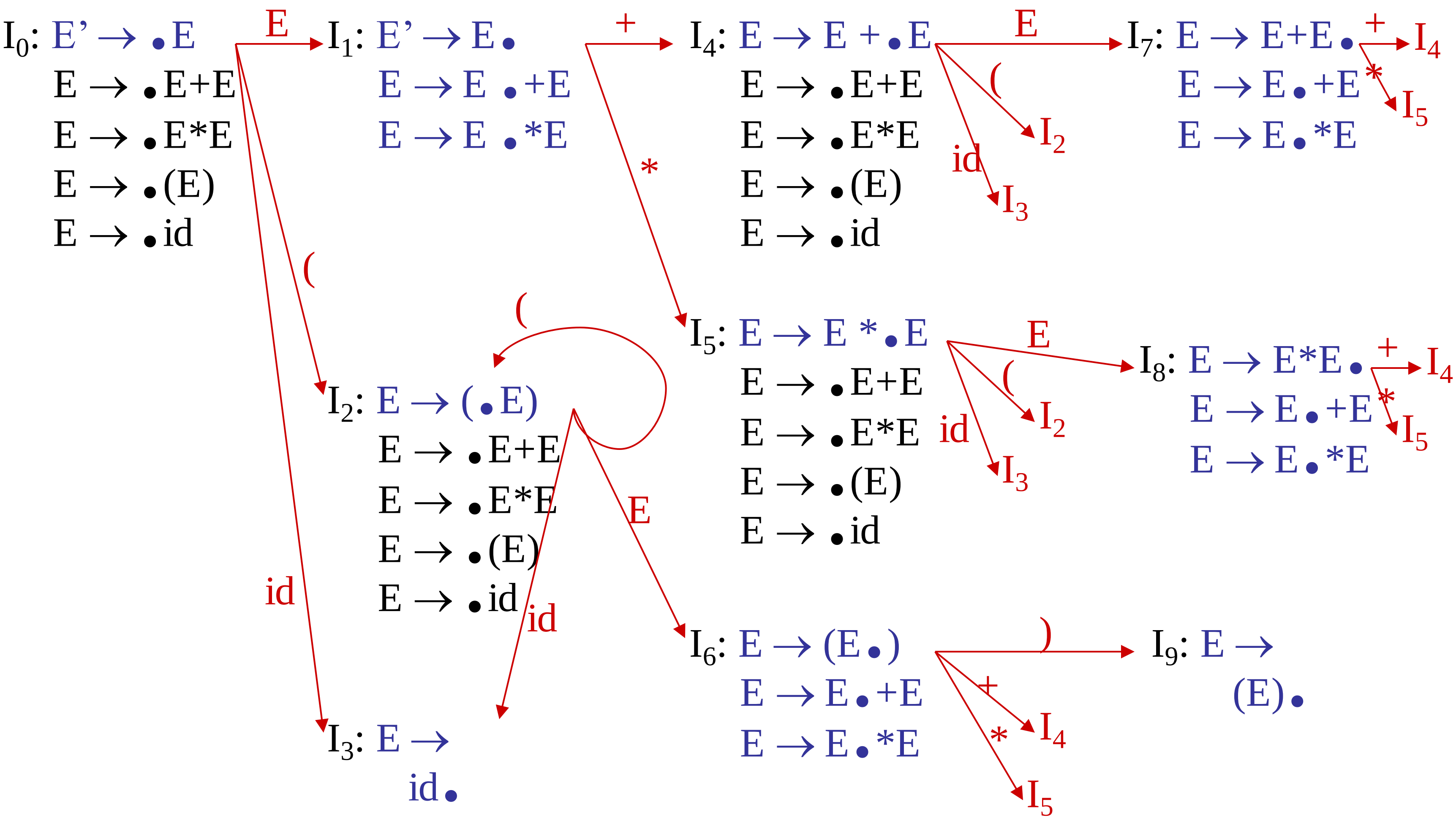
Using Ambiguous Grammars

- All grammars used in the construction of LR-parsing tables must be unambiguous.
- 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 use an ambiguous grammar?
  - Some of the ambiguous grammars are **more 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 for Ambiguous Grammar



SLR Tables for Amb 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 Tables for Amb 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 Tables for Amb 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

- 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 normal parsing.
- This nonterminal A is normally a basic programming block (there can be more than one choice for A).
  - stmt, expr, block, …

Phrase-Level Error Recovery

- 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

SLR Tables with Error Actions

	Action							Goto
	id	**+**	*****	**(**	**)**	**$**		**E**
0	s3	e1	e1	s2	e2	e1		1
1	e3	s4	s5	e3	e2	acc		
2	s3	e1	e1	s2	e2	e1		6
3	e1	r4	r4	e3	r4	r4		
4	s3	e1	e1	s2	e2	e1		7
5	s3	e1	e1	s2	e1	e1		8
6	e3	s4	s5	e3	s9	e4		
7	e3	r1	s5	e3	r1	r1		
8	e3	r2	r2	e3	r2	r2		
9	e3	r3	r3	e3	r3	r3		

Error Messages

- e1: Expected beginning of expression or subexpression (id or ‘(‘)
  - Fix: Shift id into stack and goto state 3 making believe we saw an id
  - If do this, message should be “expected operand”
- e2: Unbalanced right parenthesis
  - Fix: Ignore the ‘)’
- e3: Found start of subexpression when expecting continuation or end of current subexpression
  - Fix: ??
- e4: Found end of expression when expecting continuation (operator) or end of subexpression (‘)’)
  - Fix: ??