Bottom up parsing

- General idea
- LR(0)
- SLR
- LR(1)
- LALR
- To best exploit JavaCUP, should understand the theoretical basis (LR parsing);

Top-down vs Bottom-up

- Bottom-up more powerful than top-down;
  - Can process more powerful grammar than LL, will explain later.
- Bottom-up parsers are too hard to write by hand
  - but JavaCUP (and yacc) generates parser from spec;
- Bottom up parser uses right most derivation
  - Top down uses left most derivation;
- Less grammar translation is required, hence the grammar looks more natural;
- Intuition: bottom-up parse postpones decisions about which production rule to apply until it has more data than was available to top-down.
  - Will explain later

Bottom up parsing

- Start with a string of terminals;
- Build up from leaves of parse tree;
- Apply productions backwards;
- When reach start symbol & exhausted input, done;
- Shift-reduce is common bottom-up technique.
- Example:

  Grammar: S → *aABe*

  A → *Abc* | *b*

  B → *d*

  Reduce abbcde to S by four steps:

  abbcde

  aAbcde

  aAde

  aABe

  S

  – Notice the blue d should not be reduced into B in step 2.

  $S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abbcde$

  – How to get the right reduction steps?

Sentential form

- Sentential Form
  - Any string that can be derived from non-terminals.
  - Can consist of terminals and non terminals.
  - Example: E $\Rightarrow$ E+T$\Rightarrow$ E + id $\Rightarrow$ T+id $\Rightarrow$ id + id
  - Sentential forms: E+id, T+id, ...
  - Right sentential form: obtained by right most derivation
- Sentence
  - Sentential form with no non-terminals;
  - `id+id` is a sentence.

Handles

$S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abbcde$

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

- Informally, a handle of a sentential form is a substring that "can" be reduced.
  - Abc is a handle of the right sentential form aAbcde, because
    - A→Abc, and
    - after Abc is replaced by A, the resulting string aAde is still a right sentential form.
  - Is `d` a handle of `aAbcde`?
  - No. this is because aAbcBe is not a right sentential form.
- Formally, a handle of a right sentential form γ is a production A → β and a position in γ where the string β may be found and replaced by A.
  - If $S \Rightarrow^*_{rm} \alpha Aw \Rightarrow_{rm} \alpha\beta w$, then A →β in the position after α is a handle of αβw.
  - When the production A →β and the position are clear, we simply say "the substring β is a handle".

Handles in expression example

$E \rightarrow T + E \mid T$

$T \rightarrow int * T \mid int \mid (E)$

- Consider the string: int * int + int
- The rightmost derivation

$E \Rightarrow_{rm} T+E$

$\Rightarrow_{rm} T+T$

$\Rightarrow_{rm} T+int$

$\Rightarrow_{rm} int*T+int$

$\Rightarrow_{rm} int*int+int$

- For unambiguous grammar, there is exactly one handle for each right-sentential form.
- The question is, how to find the handle?
- Observation: The substring to the right of a handle contains only terminal symbols.

Shift-reduce parsing

- Break the input string into two parts: un-digested part and semi-digested part
  - Left part of input partly processed;
  - Right part completely unprocessed.

- Use stack to keep track of tokens seen so far and the rules already applied backwards (reductions)
- Shift next input token onto stack
- When stack top contains a “good” right-hand-side of a production, reduce by a rule;

- Important fact: Handle is always at the top of the stack.

Shift-reduce main loop

- *Shift*: If can't perform a reduction and there are tokens remaining in the unprocessed input, then transfer a token from the input onto the stack.
- *Reduce*: If we can find a rule A→α , and the contents of the stack are βα for some β (β may be empty), then reduce the stack to βA. The α is called a *handle*. Recognizing handles is key!
- *Accept*: S is at the top of the stack and input now empty, done
- *Error*: other cases.

Example 1

Grammar:
S –> E
E –> T | E + T
T –> id | (E)

Input string:
(id)

Parse Stack	Remaining input	Parser Action
	(id)$	Shift parenthesis onto stack
(	id)$	Shift id onto stack
(id	)$	Reduce: T→ id (pop RHS of production, push LHS, input unchanged)
(T	)$	Reduce: E → T
(E	)$	Shift right parenthesis
(E)	$	Reduce: T → (E)
T	$	Reduce: E → T
E	$	Reduce: S → E
S	$	Done: Accept

$$S \Rightarrow_{rm} E \Rightarrow_{rm} T \Rightarrow_{rm} (E) \Rightarrow_{rm} (T) \Rightarrow_{rm} (id)$$

Shift-Reduce Example 2

S → E
E → T | E + T
T → id | (E)

Input: (id + id)

Parse Stack	Remaining Input	Action
	(id + id) $	Shift (
(	id + id) $	Shift id
(id	+ id) $	Reduce T → id
(T	+ id) $	Reduce E → T
(E	+ id) $	Shift +
(E+	id) $	Shift id
(E+id	) $	Reduce T → id
(E+T	) $	Reduce E → E+T; (Ignore: E→T)
(E	) $	Shift)
(E)	$	Reduce T → (E)
T	$	Reduce E → T
E	$	Reduce S → E
S	$	Accept

(id ● +id)
(T ● +id)

(E +id ●)
(E+T ●)

(E) ●
T ●
E ●
S

Note that it is the reverse of the following rightmost derivation:

$S \Rightarrow_{rm} E \bullet \Rightarrow_{rm} T \bullet \Rightarrow_{rm} (E) \bullet \Rightarrow_{rm} (E+T \bullet) \Rightarrow_{rm} (E +id \bullet)$
$\Rightarrow_{rm} (T \bullet +id) \Rightarrow_{rm} (id \bullet +id)$

Conflicts during shift reduce parsing

- Reduce/reduce conflict

stack	input
... (E+T	)

  Which rule we should use, E → E+T or E → T ?

- Shift/reduce conflict
  - ifStat → if (E) S | if (E) S else S

stack	Input
... if (E) S	else ...

  - Both reduce and shift are applicable.
  - What we should do next, reduce or shift?

LR(K) parsing

- **L**eft-to-right, **R**ightmost derivation with k-token lookahead.
  - L - Left-to-right scanning of the input
  - R - Constructing rightmost derivation in reverse
  - k - number of input symbols to select a parser action
- Most general parsing technique for deterministic grammars.
  - Efficient, Table-based parsing
  - Parses by shift-reduce
  - Can massage grammars less than LL(1)
  - Can handle almost all programming language structures
  - LL $\subset$ LR $\subset$ CFG
- In general, not practical: tables too large (10^6 states for C++, Ada).
- Common subsets: SLR, LALR (1).

LR Parsing continued

- Data structures:
  - Stack of states {s}
  - Action table Action[s,a]; a ∈ T
  - Goto table Goto[s,X]; X ∈ N
- In LR parsing, push whole *states* on stack
  - Stack of states keeps track of what we've seen so far (*left context*): what we've shifted & reduced & by what rules.
- Use Action tables to decide shift vs reduce
- Use Goto table to move to new state

Main loop of LR parser

- Initial state S0 starts on top of stack;
- Given state *St* state on top of stack and the next input token *a*:
- If (Action[St, a] == shift Si)
  - Push new state Si onto stack
  - Call yylex to get next token
- If (Action[St, a] == reduce by Y → X1…Xn)
  - Pop off n states to find Su on top of stack
  - Push new state Sv = Goto[Su,Y] onto stack
- If (Action[St, a] == accept), done!
- If (Action[St, a] == error), can't continue to successful parse.

Example LR parse table

(1) E → E + T
(2) E → T
(3) T → (E)
(4) T → id

State on TOS	Action					Goto	
	id	**+**	**(**	**)**	**$**	**E**	**T**
0	S4		S3			S1	S2
1		S5			accept		
2	R2	R2	R2	R2	R2		
3	S4		S3			S6	S2
4	R4	R4	R4	R4	R4		
5	S4		S3				S8
6		S5		S7			
7	R3	R3	R3	R3	R3		
8	R1	R1	R1	R1	R1		

If (Action[St, a] == shift), Push new state Action[St, a] onto stack, Call yylex to get next token
If (Action[St, a] == reduce by Y → X1…Xn), Pop off n states to find Su on top of stack, Push new state Sv = Goto[Su,Y] onto stack

We explain how to construct this table later.

State on TOS	Action					Goto	
	id	+	(	)	$	E	T
0	S4		S3			S1	S2
1		S5			accept		
2	R2	R2	R2	R2	R2		
3	S4		S3			S6	S2
4	R4	R4	R4	R4	R4		
5	S4		S3				S8
6		S5		S7			
7	R3	R3	R3	R3	R3		
8	R1	R1	R1	R1	R1		

(1) E → E + T
(2) E → T
(3) T → (E)
(4) T → id

State stack	Remaining Input	Parser action
S0	id + (id)$	Shift S4 onto state stack, move ahead in input
S0S4	+ (id)$	Reduce 4) T → id, pop state stack, goto S2, input unchanged
S0S2	+ (id)$	Reduce 2) E → T, goto S1
S0S1	+ (id)$	Shift S5
S0S1S5	(id)$	Shift S3
S0S1S5S3	id)$	Shift S4 (saw another id)
S0S1S5S3S4	)$	Reduce 4) T → id, goto S2
S0S1S5S3S2	)$	Reduce 2) E → T, goto S6
S0S1S5S3S6	)$	Shift S7
S0S1S5S3S6S7	$	Reduce 3) T → (E), goto S8
S0S1S5S8	$	Reduce 1) E → E + T, goto S1 *
S0S1	$	Accept

Types of LR parsers

- LR (k)
- SLR (k) -- Simple LR
- LALR (k) – LookAhead LR
- k = # symbols lookahead
  - 0 or 1 in this class
  - Dragon book has general cases
- Start with simplest: LR(0) parser

LR (0) parser

- Advantages:
  - Simplest to understand,
  - Smallest tables
- Disadvantages
  - No lookahead, so too simple-minded for real parsers
- Good case to see how to build tables, though.
- We'll use LR(0) constructions in other LR(k) parsers
- Key to LR parsing is recognizing handles
  - Handle: sequence of symbols encoded in top stack states representing a right-hand-side of a rule we want to reduce by.

LR Tables

- Given grammar G, identify possible states for parser.
  - States encapsulate what we've seen and shifted and what are reduced so far
- Steps to construct LR table:
  - Construct states using LR(0) *configurations* (or *items*);
  - Figure out transitions between states

Configuration

- A configuration (or item) is a rule of G with a dot in the right-hand side.
- If rule A → XYZ in grammar, then the configs are

  A → •XYZ A → XY • Z

  A → X • YZ A → XYZ •
- Dot represents what parser has gotten in stack in recognizing the production.

  A → XYZ • means XYZ on stack. Reduce!

  A → X • YZ means X has been shifted. To continue parse, we must see a token that could begin a string derivable from Y.

- Notational convention:
  - X, Y, Z: symbol, either terminal or non-terminal
  - a, b, c : terminal
  - α, β , γ: sequence of terminals or non-terminals

Set of configurations

- A → X • YZ means X has been shifted. To continue parse, we must see a token that could begin a string derivable from Y.
- That is, we need to see a token in First(Y) (or in Follow(Y) if Y → ε)
- Formally, need to see a token t such that

  $Y \Rightarrow^* t\,\beta$ for some β
- Suppose Y → α | β also in G. Then these configs correspond to the same parse state:

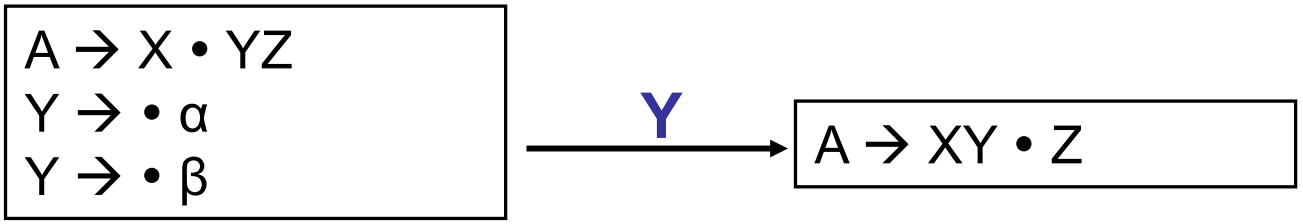
  A → X • YZ

  Y → • α

  Y → • β
- Since the above configurations represent the same state, we can:
  - Put them into a set together.
  - Add all other equivalent configurations to achieve closure. (algorithm later)
  - This set represents one parser state: the state the parser can be in while parsing a string.

Transitions between states

- Parser goes from one state to another based on symbols processed



- Model parse as a finite automaton!
- When a state (configuration set) has a dot at a end of an item, that is FA accept state
- Build LR(0) parser based on this FA

Constructing item sets & closure

- Starting Configuration:
  - Augment Grammar with symbol S'
  - Add production S' → S to grammar
  - Initial item set I_0 gets

    S' → • S
- Perform Closure on S' → • S

(That completes parser start state.)

- Compute Successor function to make next state (next item set)

Computing closure

- Closure(I)
  1. Initially every item in I is added to closure(I)
  2. If A → α ● B β is in closure(I)
     for all productions B → γ , add B → • γ
  3. Repeat step 2 until set gets no more additions.
- Example

Given the configuration set: { E → •E+T}

What is the closure of { E → •E+T}:

E → ● E + T by rule 1

E → ● T by rule 2

T → ● (E) by rule 2 and 3

T → ● id by rule 2 and 3

(1) E → E + T
(2) E → T
(3) T → (E)
(4) T → id

Building state transitions

- LR Tables need to know what state to goto after shift or reduce.
- Given Set C & symbol X, we define the set C' = *Successor* (C,X) as:

  For each config in C of the form Y → α ● X β,

  1. Add Y → α X ● β to C'
  2. Do closure on C'

- Informally, move by symbol X from one item set to another;
  - move ● to the right of X in all items where dot is before X;
  - remove all other items;
  - compute closure.

Successor example

- Given

  I= {E→ • E + T,
  E → • T,
  T → • (E),
  T → • id
  }

- What is successor(I, "(") ?
- move the • after "(" : T→(• E)
- compute the closure:

  T→ (• E)
  E→ • E + T
  E → • T
  T → • (E)
  T → • id

(1) E → E + T
(2) E → T
(3) T → (E)
(4) T → id

Construct the LR(0) table

- Construct F={I0, I1, I2, ..., In}
- State i is determined by Ii. The parsing actions for state i are:
  - if A→α ● is in Ii,
    - then set Action[i, a] to reduce A→α for all inputs (if A is not S')
  - If S'→S ● is in Ii,
    - then set action[i, $] to accept.
  - if A→α ● a β is in Ii and successor(Ii, a)=Ij,
    - then set action[I,j] to shift j. (a is a terminal)
- The goto transitions for state i are constructed for all non-terminals A using the rule:
  - if successor(Ii,A)=Ij, then goto[i, A]=j.
- All entries not defined by above rules are errors.
- The initial state I0 is the one constructed from S'→● S.

Steps of constructing LR(0) table

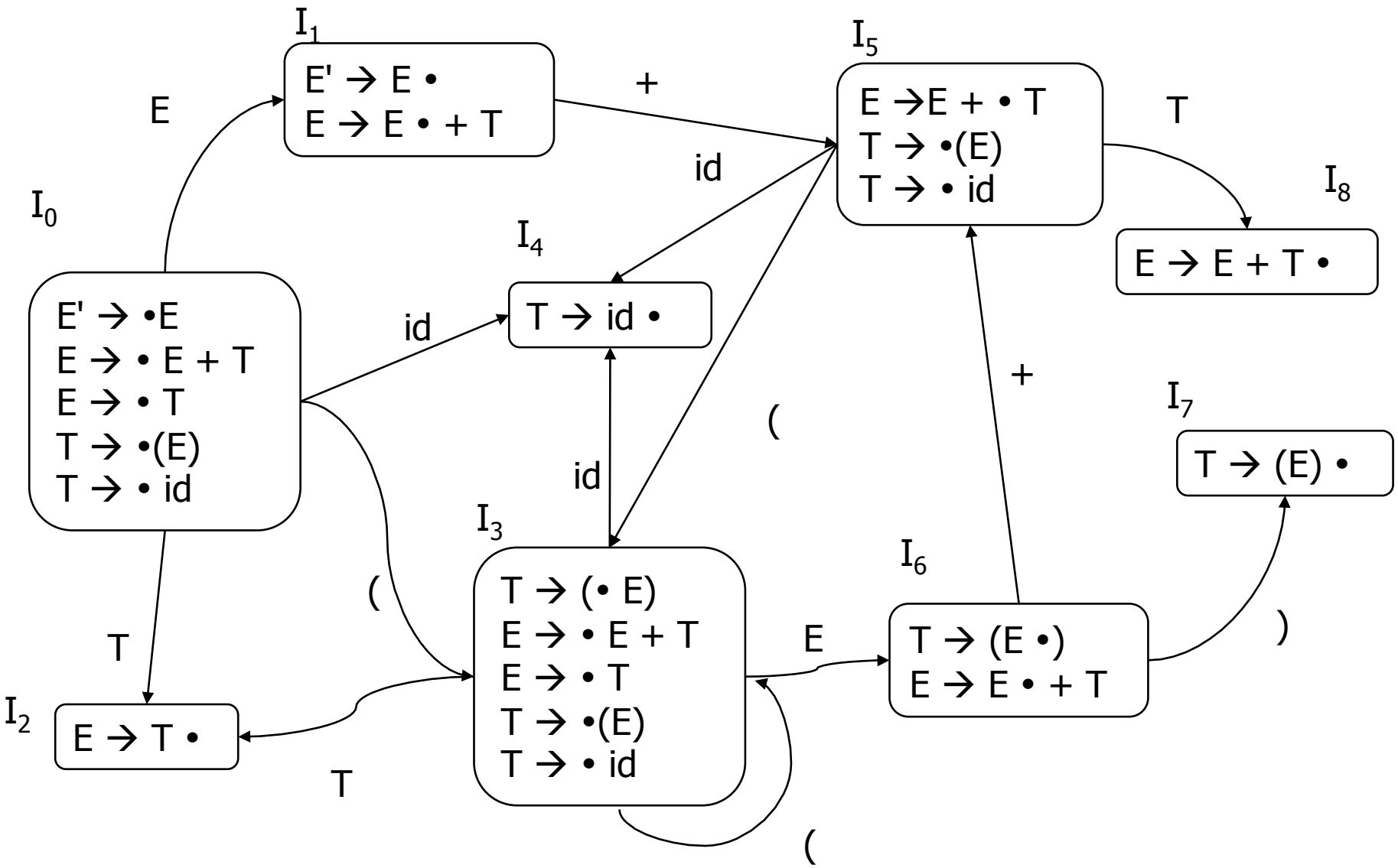
(0) E′ → E
(1) E → E + T
(2) E → T
(3) T → (E)
(4) T → id

1. Augment the grammar;
2. Draw the transition diagram;
    1. Compute the configuration set (item set/state);
    2. Compute the successor;
3. Fill in the Action table and Goto table.

Item sets example

Configuration set		Successor
I0:	E' → •E	I1
	E → •E+T	I1
	E → •T	I2
	T → •(E)	I3
	T → •id	I4
I1:	E' → E•	Accept (dot at end of E' rule)
	E → E•+T	I5
I2:	E → T•	Reduce 2 (dot at end)
I3:	T → (•E)	I6
	E → •E+T	I6
	E → •T	I2
	T → •(E)	I3
	T → •id	I4
I4:	T → id•	Reduce 4 (dot at end)
I5:	E → E+•T	I8
	T → •(E)	I3
	T → •id	I4
I6:	T → (E•)	I7
	E → E•+T	I5
I7:	T → (E)•	Reduce 3 (dot at end)
I8:	E → E+T•	Reduce 1 (dot at end)

Transition diagram



The parsing table

State on TOS	Action					Goto	
	id	**+**	**(**	**)**	**$**	**E**	**T**
0	S4		S3			1	2
1		S5			accept		
2	R2	R2	R2	R2	R2		
3	S4		S3			6	2
4	R4	R4	R4	R4	R4		
5	S4		S3				8
6		S5		S7			
7	R3	R3	R3	R3	R3		
8	R1	R1	R1	R1	R1		

Parsing an erroneous input

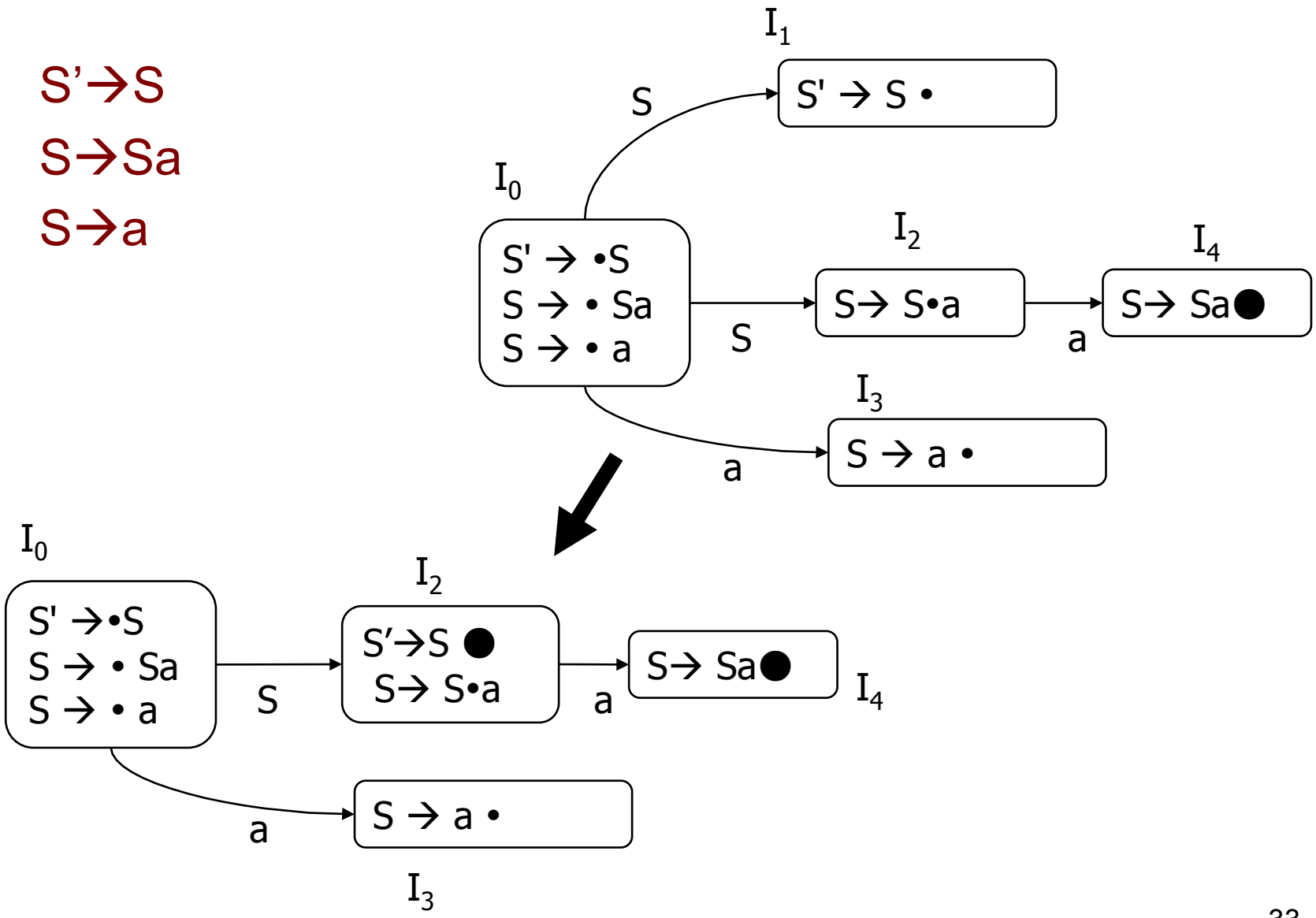
(0) E' → E
(1) E → E + T
(2) E → T
(3) T → (E)
(4) T → id

State stack	Input	Parser action
S0	id + +$	Shift S4
S0 S4	+ +$	Reduce 4) T → id, pop S4, Goto S2
S0 S2	+ +$	Reduce 2) E → T, pop S2, Goto S1
S0 S1	+ +$	Push S5
S0 S1 S5	+$	No Action [S5, +] Error!

State on TOS	Action					Goto	
	id	**+**	**(**	**)**	**$**	**E**	**T**
0	S4		S3			S1	S2
1		S5			accept		
2	R2	R2	R2	R2	R2		
3	S4		S3			S6	S2
4	R4	R4	R4	R4	R4		
5	S4		S3				S8
6		S5		S7			
7	R3	R3	R3	R3	R3		
8	R1	R1	R1	R1	R1		

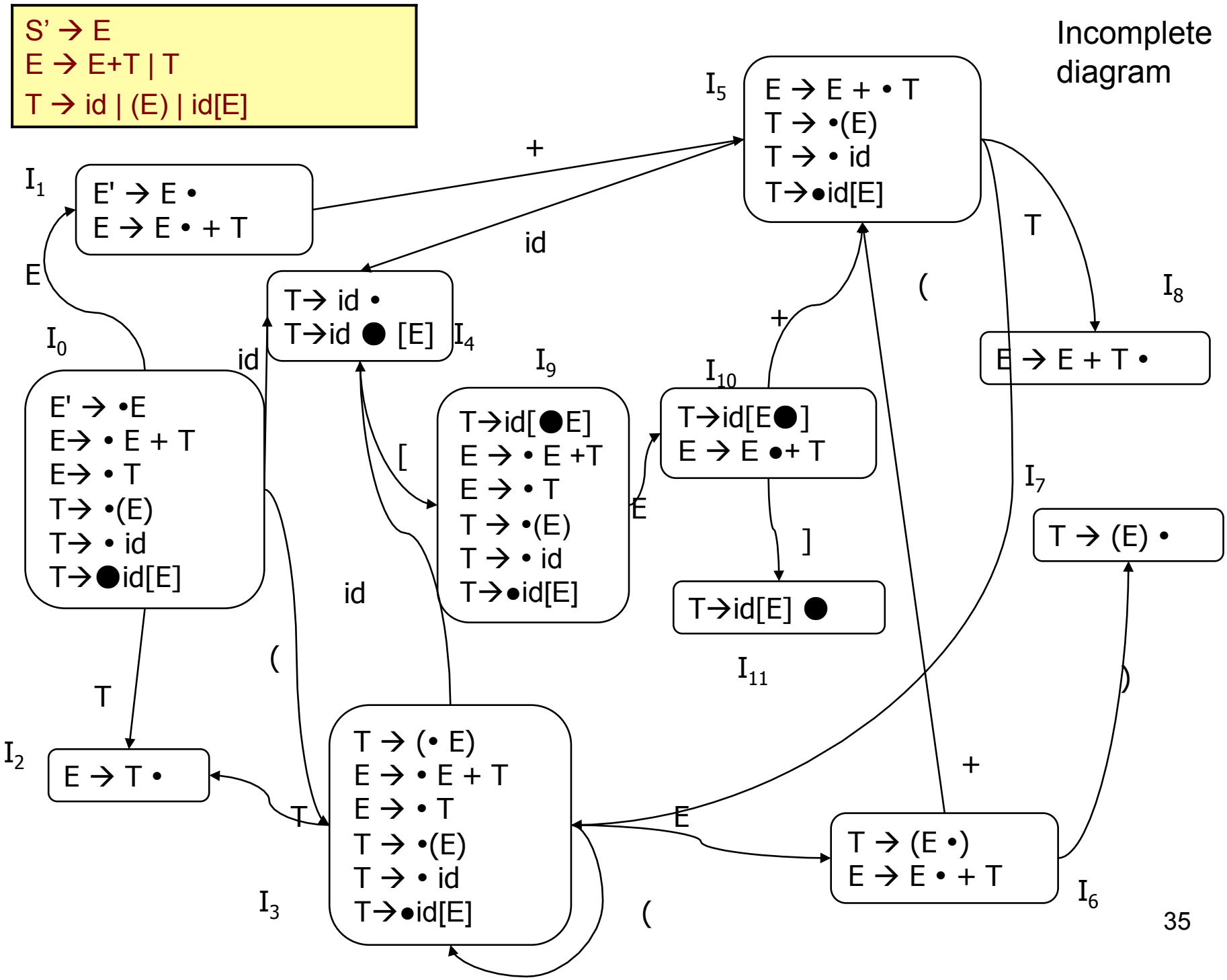
Subset construction and closure

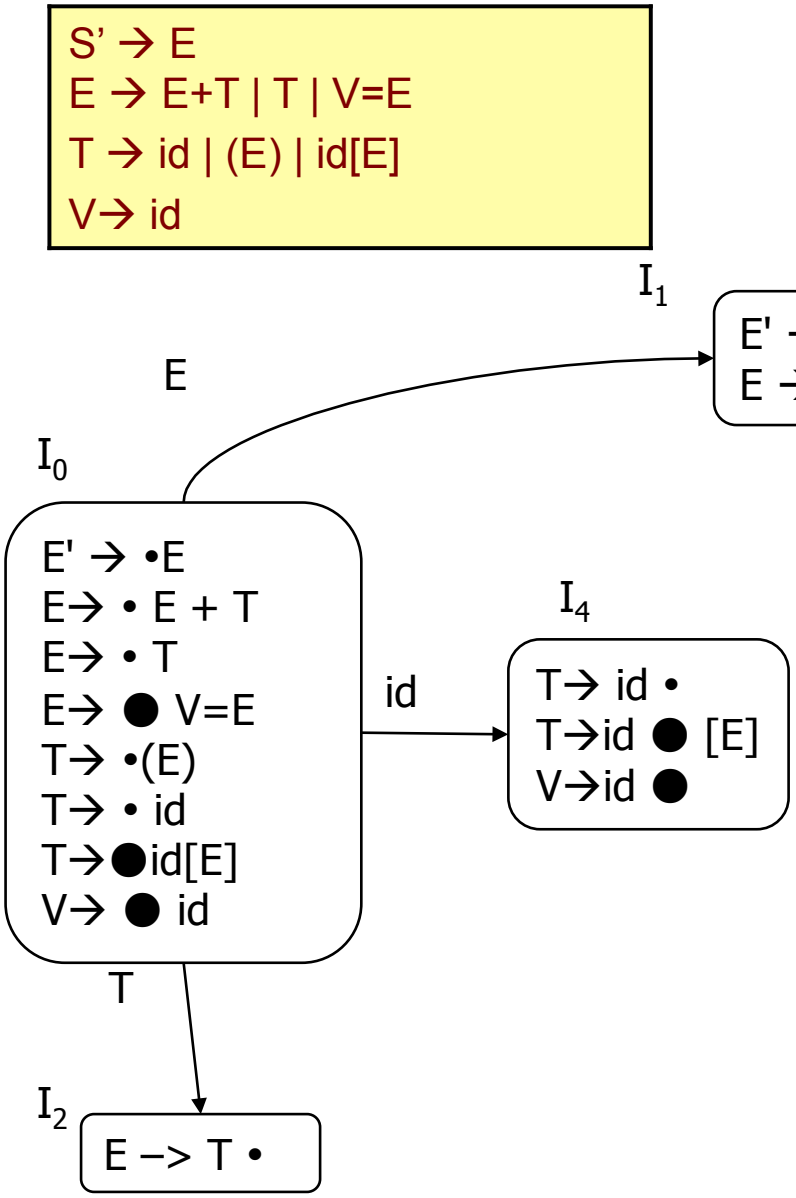
- S'→S
- S→Sa
- S→a



LR(0) grammar

- A grammar is LR(0) if the following two conditions hold:
  1. For any configuration set containing the item A→α●aβ, there is no complete item B→γ● in that set.
     - No shift/reduce conflict in any state
     - in table, for each state, either shift or reduce
  2. There is at most one complete item A→α● in each configuration set.
     - No reduce/reduce conflict
     - in table, for each state, use same reduction rule for every input symbol.
- Very few grammars meet the requirements to be LR(0).


S' → E
E → E+T | T
T → id | (E) | id[E]
Incomplete diagram
I0
E' → •E
E→ • E + T
E→ • T
T→ •(E)
T→ • id
T→●id[E]
I1
E' → E •
E → E • + T
I2
E → T •
I3
T → (• E)
E → • E + T
E → • T
T → •(E)
T → • id
T→●id[E]
I4
T→ id •
T→id ● [E]
I5
E → E + • T
T → •(E)
T → • id
T→●id[E]
I6
T → (E •)
E → E • + T
I7
T → (E) •
I8
E → E + T •
I9
T→id[●E]
E → • E +T
E → • T
T → •(E)
T → • id
T→●id[E]
I10
T→id[E●]
E → E ●+ T
I11
T→id[E] ●
E
T
id
(
+
[
]
)
35



Shift/reduce conflict:

T→ id •

T→ id • [E]

Reduce/reduce conflict:

T→ id •

V→ id •

SLR Parse table (incomplete)

State on TOS	Action							Goto	
	id	**+**	**(**	**)**	**$**	**[**	**]**	**E**	**T**
0	S4		S3					1	2
1		S5			accept				
2		R2		R2	R2		R2		
3	S4		S3					6	2
4		R4		R4	R4	S9	R4		
5	S4		S3						8
6		S5		S7					
7		R5		R5	R5		R5		
8		R1		R1	R1		R1		
9									
10									
11									

(0) S' → E
(1) E → E+T
(2) E→T
(3) E→ V=E
(4) T → id
(5) T→ (E)
(6) T→ id[E]
(7) V→id

LR(0) key points

- Start with augmented grammar.
- Generate *items* from productions.
  - Insert the Dot into all positions
- Generate item sets (or configurating sets) from items; they are our parser states.
- Generate state *transitions* from function successor (state, symbol).
- Build Action and Goto tables from states and transitions.
- Tables implement shift-reduce parser.
- View [states and transitions] as a finite automaton.

- An Item represents how far a parser is in recognizing part of one rule's RHS.
- An Item set combines various paths the parser might have taken so far, to diverge as more input is parsed.
- LR(0) grammars are easiest LR to understand, but too simple to use in real life parsing.

Simple LR(1) parsing: SLR

- LR(0)
  - One LR(0) state mustn't have both shift and reduce items, or two reduce items.
  - So any complete item (dot at end) must be in its own state; parser will always reduce when in this state.
- SLR
  - Peek ahead at input to see if reduction is appropriate.
  - Before reducing by rule A → XYZ, see if the next token is in Follow (A). Reduce *only* in that case. Otherwise, shift.

Construction for SLR tables

1. Construct F = {I_0 , I_1 , ... I_n }, the LR(0) item sets.
2. State i is Ii. The parsing actions for the state are:
   a) If A→α• is in Ii
      then set Action[i,a] to reduce A –> α for all a in Follow(A) (A is not S').
   b) If S' → S• is in Ii
      then set Action[i,$] to accept.
   c) If A→α•aβ is in Ii and successor(Ii , a) = Ij,
      then set Action[i,a] to shift j (a must be a terminal).
3. The goto transitions for state i are constructed for all non-terminals A using the rule:
   If successor(Ii, A) = Ij, then Goto [i, A] = j.
4. All entries not defined by rules 2 and 3 are errors.
5. The initial state is closure of set with item S'→ •S.

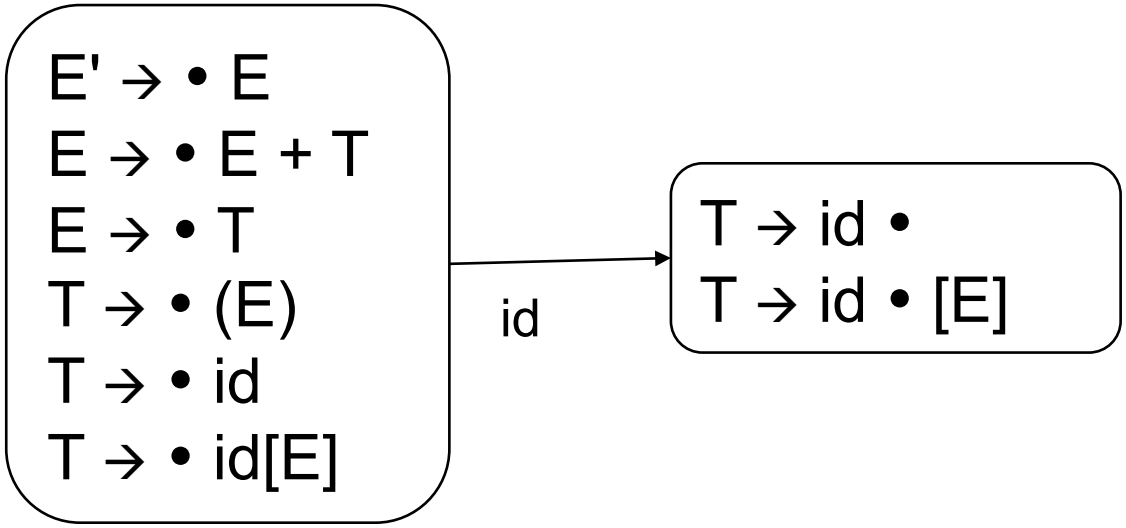
Properties of SLR

- Pickier rule about setting Action table is the only difference from LR(0) tables;
- If G is SLR it is unambiguous, but not vice versa;
- State can have both shift and reduce items, if Follow sets are disjoint.

SLR Example

E' →E
E →E + T | T
T →(E) | id
| id[E]

Item sets I_0 and successor (I_0, id):



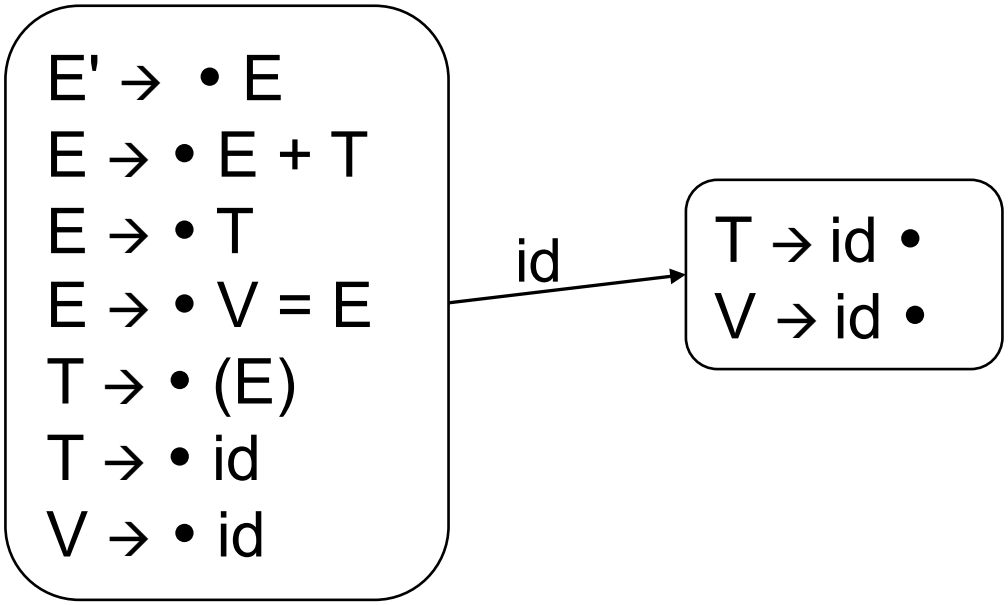
LR(0) parser sees both shift and reduce, but SLR parser consults Follow set:

Follow(T) = { +,),], $ } so

T → id • means reduce on + or) or] or $

T → id •[E] means shift otherwise (e.g. on [ )

SLR Example 2



E' → E
E → E + T | T
| V = E
T → (E) | id
V → id

Two complete LR(0) items, so reduce-reduce conflict in LR(0) grammar, but:

Follow(T) = { +,), $ }

Follow(V) = { = }

Disjoint, so no conflict. Separate Action entries in table.

SLR grammar

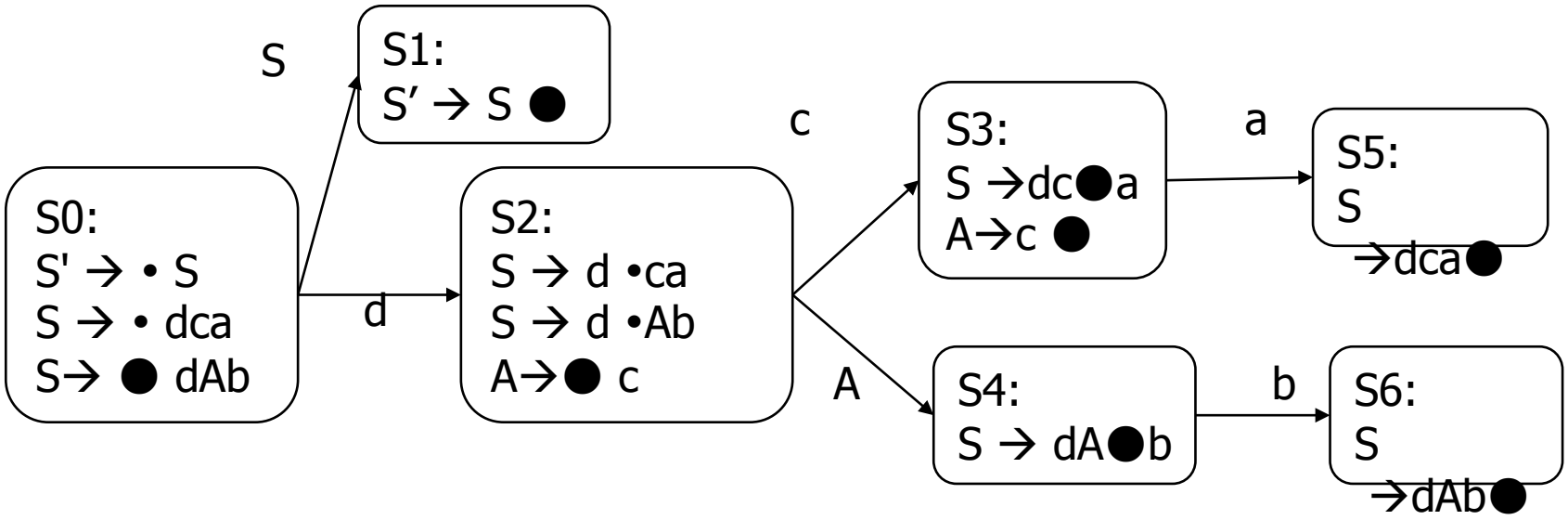
- A grammar is SLR if the following two conditions hold:
  - If items A → α • a β and B → γ • are in a state, then terminal a ∉ Follow(B).
    - no shift-reduce conflict on any state. This means the successor function for x from that set either shifts to a new state or reduces, but not both.
  - For any two complete items A → α • and B → β• in a state, the Follow sets must be disjoint. (Follow(A) ∩ Follow(B) is empty.)
    - no reduce-reduce conflict on any state. If more than one non-terminal could be reduced from this set, it must be possible to uniquely determine which using only one token of lookahead.
- Compare with LR(0) grammar:
  1. For any configuration set containing the item A→α●aβ, there is no complete item B→γ● in that set.
  2. There is at most one complete item A→α● in each configuration set.
- Note that LR(0) ⊂ SLR

SLR

1. S'→S
2. S→dca
3. S→dAb
4. A→c

In S3 there is reduce/shift conflict: It can be R4 or shift. By looking at the Follow set of A, the conflict is removed.

	Action					Goto	
	a	b	c	d	$	S	A
S0				S2		1	
S1					A		
S2			S3				4
S3	S5	R4					
S4		S6					
S5					R2		
S6					R3		



Parse trace

State stack	Input	Parser action
S0	dca$	Shift S2
S0 S2d	ca$	Shift S3
S0 S2d S3c	a$	shift S5
S0 S2d S3c S5a	$	Reduce 2
S0 S1S	$	Accept

Non-SLR example

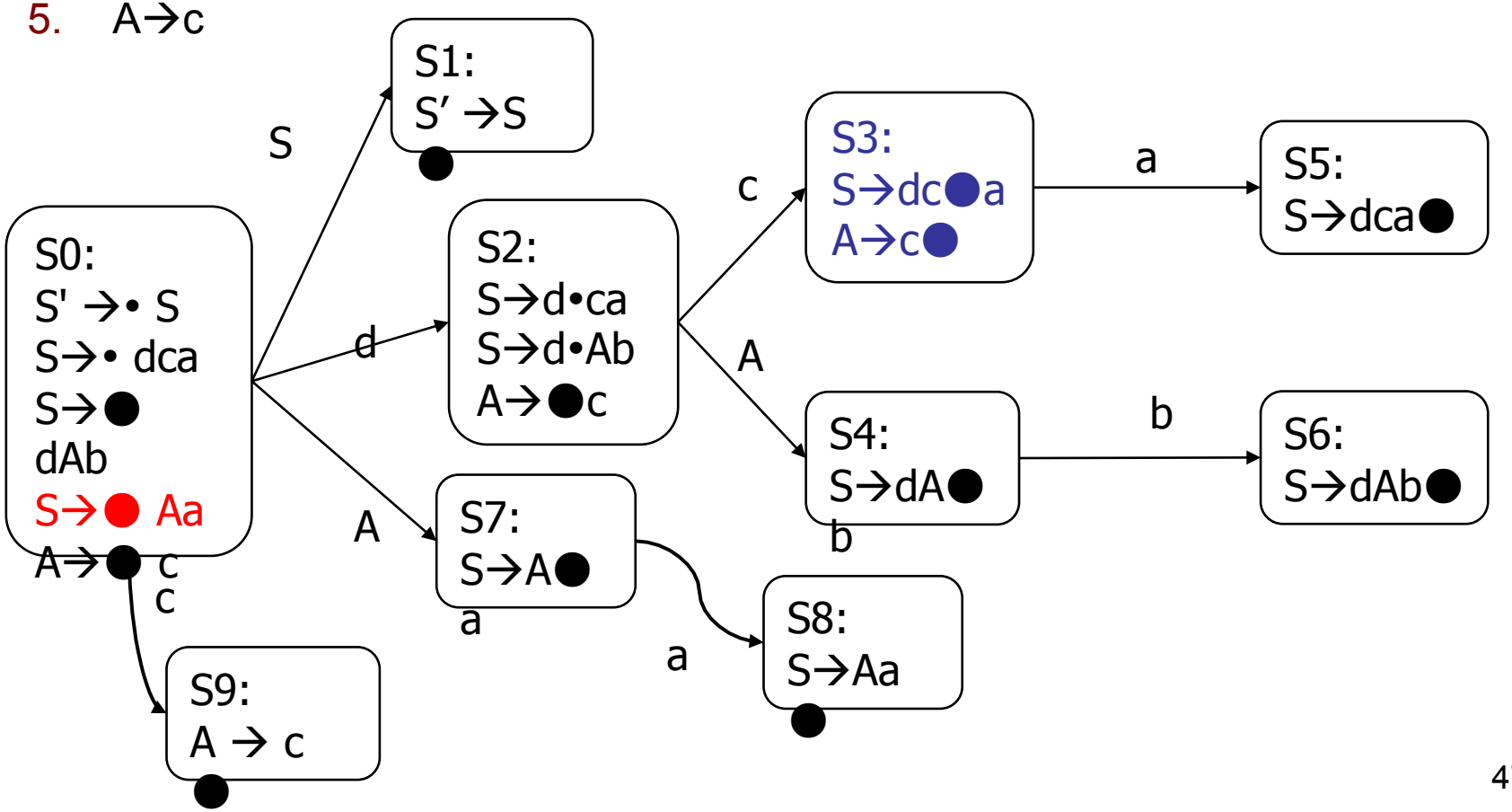
1. S'→S
2. S→dca
3. S→dAb
4. S→Aa
5. A→c

S3 has shift/reduce conflict.

By looking at Follow(A),

both *a* and *b* are in the follow set.

So under column a we still don't know whether to reduce or shift.



The conflict SLR parsing table

	Action					Goto	
	a	b	c	d	$	S	A
S0			S9	S2		1	7
S1					A		
S2			S3				4
S3	S5/R5	R5					
S4		S6					
S5					R2		
S6					R3		
S7	S8						
S8					R4		
S9	R5	R5					

Follow(A) = {a, b}

LR(1) parsing

- Make items carry more information.
- LR(1) item:

  A → X1...Xi • Xi+1...Xj , tok
- Terminal **tok** is the *lookahead.*
- Meaning:
  - have states for X1...Xi on stack already
  - expect to put states for Xi+1...Xj onto stack and then reduce, but
  - *only* if token following Xj is **tok**
  - tok can be $
- Split Follow(A) into separate cases
- Can cluster items notationally:

  [A→α•, a/b/c] means the three items: [A→α•, a] [A→α•, b] [A→α•, c]

  "Reduce α to A if next token is *a* or *b* or *c*"
- { a, b, c } ⊆ Follow(A)

LR(1) item sets

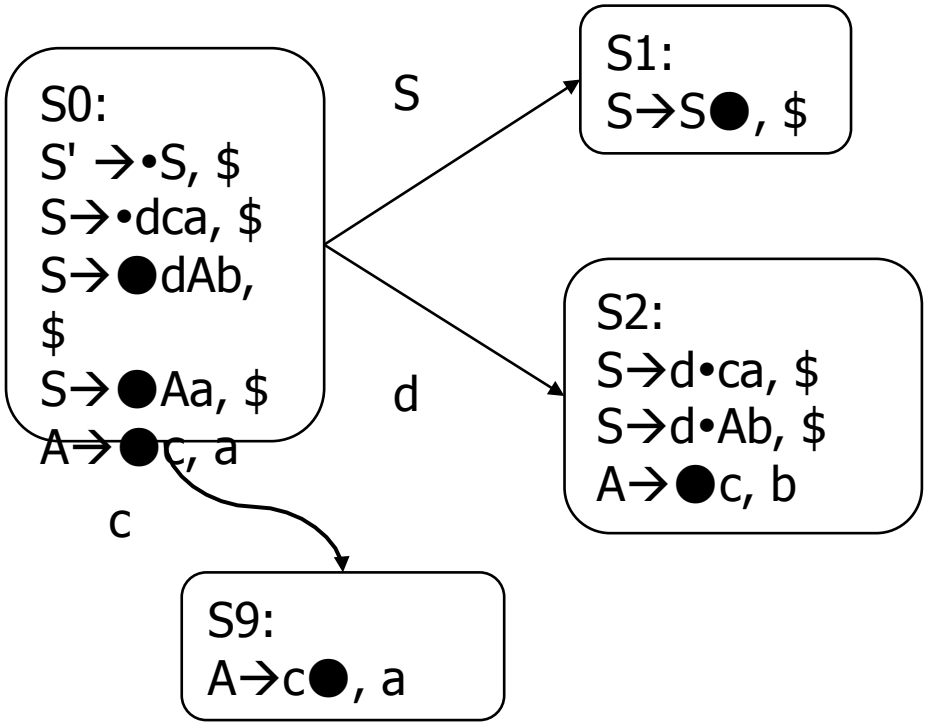
- More items and more item sets than SLR
- Closure: For each item [A→α•Bβ, a] in I,
  - for each production B → γ in G,
  - and for each terminal b in First(βa), add [B →• γ, b] to I

  (Add only items with the correct lookahead)
- Once we have a closed item set, use LR(1) successor function to compute transitions and next items.
- Example:
  - Initial item: [S'→●S, $]
  - What is the closure?
  - [S→●dca, $]
  - [S→●dAb, $]
  - [S→●Aa, $]
  - [A→●c, a]

- S'→S
- S→dca|dAb |Aa
- A→c

LR(1) successor function

- Given I an item set with [A→ α•Xβ, a],
  - Add [A →α X•β, a] to item set J.
  - successor(I,X) is the closure of set J.
  - Similar to successor function to LR(0), but we propagate the lookahead token for each item.
- Example



LR(1) tables

Action table entries:

- If [A→α●, a] ∈ Ii, then set Action[i,a] to reduce by rule A→α (A is not S').
- If [S'→S •, $] ∈ Ii then set Action[i,$] to accept.
- If [A →α●aβ, b] is in Ii and succ(Ii, a) = Ij, then set Action[i,a] to shift j. Here a is a terminal.

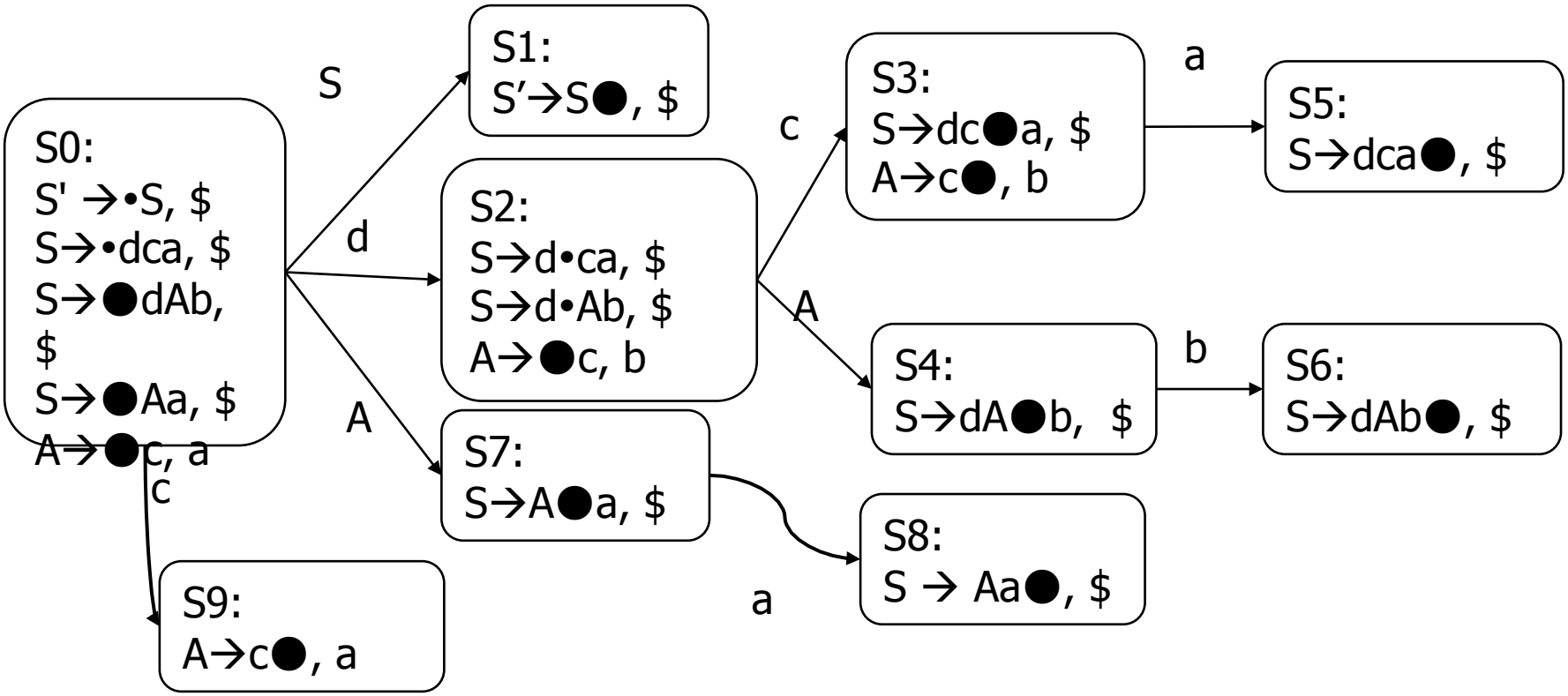
Goto entries:

For each state I & each non-terminal A :

- If succ(Ii, A) = Ij, then Goto [i, A] = j.

LR(1) diagram

1. S'→S
2. S→dca
3. S→dAb
4. S→Aa
5. A→c



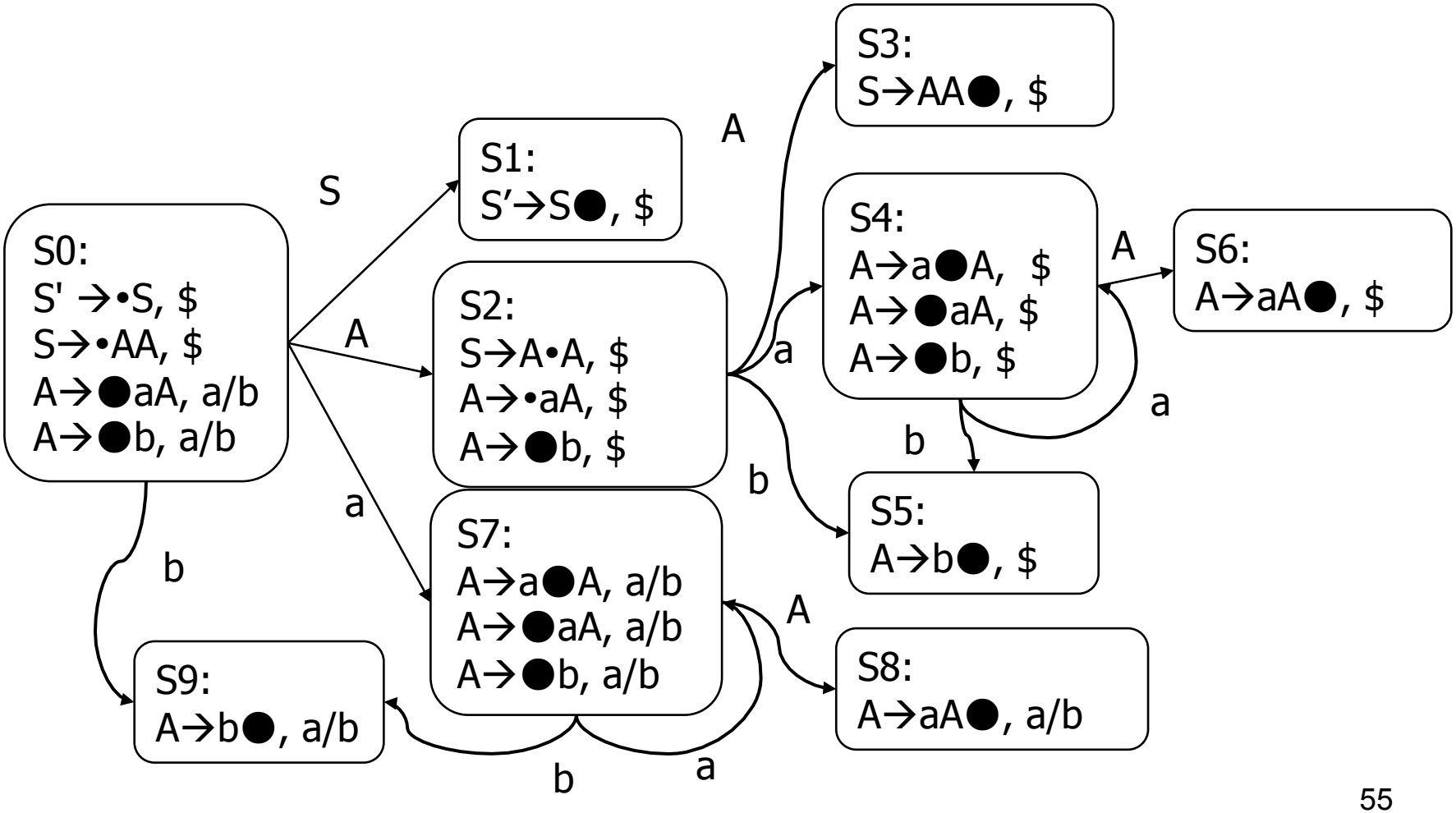
Create the LR(1) parse table

	Action					Goto	
	a	b	c	d	$	S	A
S0			S9	S2		1	7
S1					A		
S2			S3				4
S3	S5	R5					
S4		S6					
S5					R2		
S6					R3		
S7	S8						
S8					R4		
S9	R5						

Another LR(1) example

0) S' → S
1) S → AA
2) A → aA
3) A → b

- Create the transition diagram



Parse table

state	Action			Goto	
	a	b	$	S	A
S0	S7	S9		1	2
S1			Accept		
S2	S4	S5			3
S3			R1		
S4	S4	S5			6
S5			R3		
S6			R2		
S7	S7	S9			8
S8	R2	R2			
S9	R3	R3			

Parse trace

stack	remaining input	parse action
S0	baab$	S9
S0S9	aab$	R3 A→b
S0S2	aab	S4
S0S2S4	ab	S4
S0S2S4S4	b	S5
S0S2S4S4S5	$	R3 A→b
S0S2S4S4S6	$	R2 A→aA
S0S2S4S6	$	R2 A→aA
S0S2S3	$	R1 S→AA
S0S1	$	Accept

LR(1) grammar

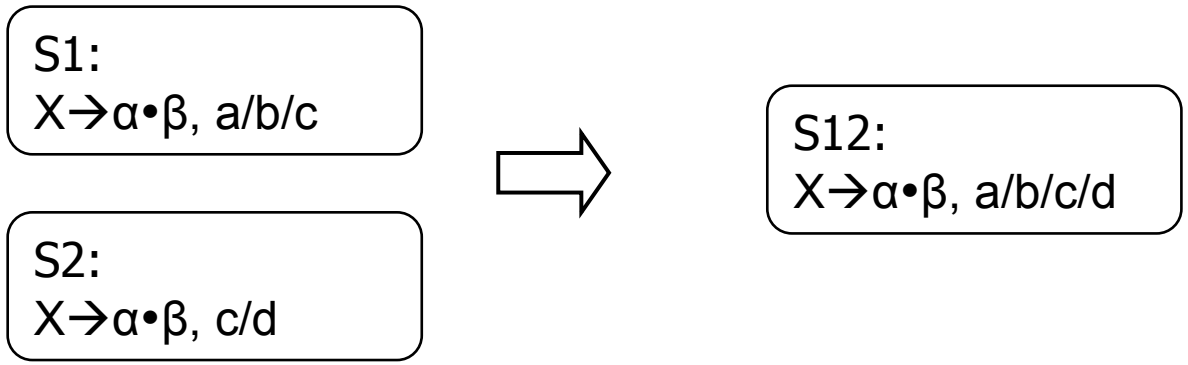
- A grammar is LR(1) if the following 2 conditions are satisfied for each configuration set:
  - For each item [A→α●aβ, b] in the set, there is no item in the set of the form [B→γ●, a]
    - In the action table, this translates to no shift/reduce conflict.
  - If there are two complete items [A→α●, a] and [B→β●, b] in the set, then a and b should be different.
    - In the action table, this translates to no reduce/reduce conflict
- Compare with the SLR grammar
  - For any item A→α•aβ in the set, with terminal a, there is no complete item B→γ• in that set with a in Follow(B).
  - For any two complete items A→α• and B→β• in the set, the Follow sets must be disjoint.
- Note that SLR(1) ⊂ LR(1)
- LR(0) ⊂ SLR(1) ⊂ LR(1)

LR(1) tables continued

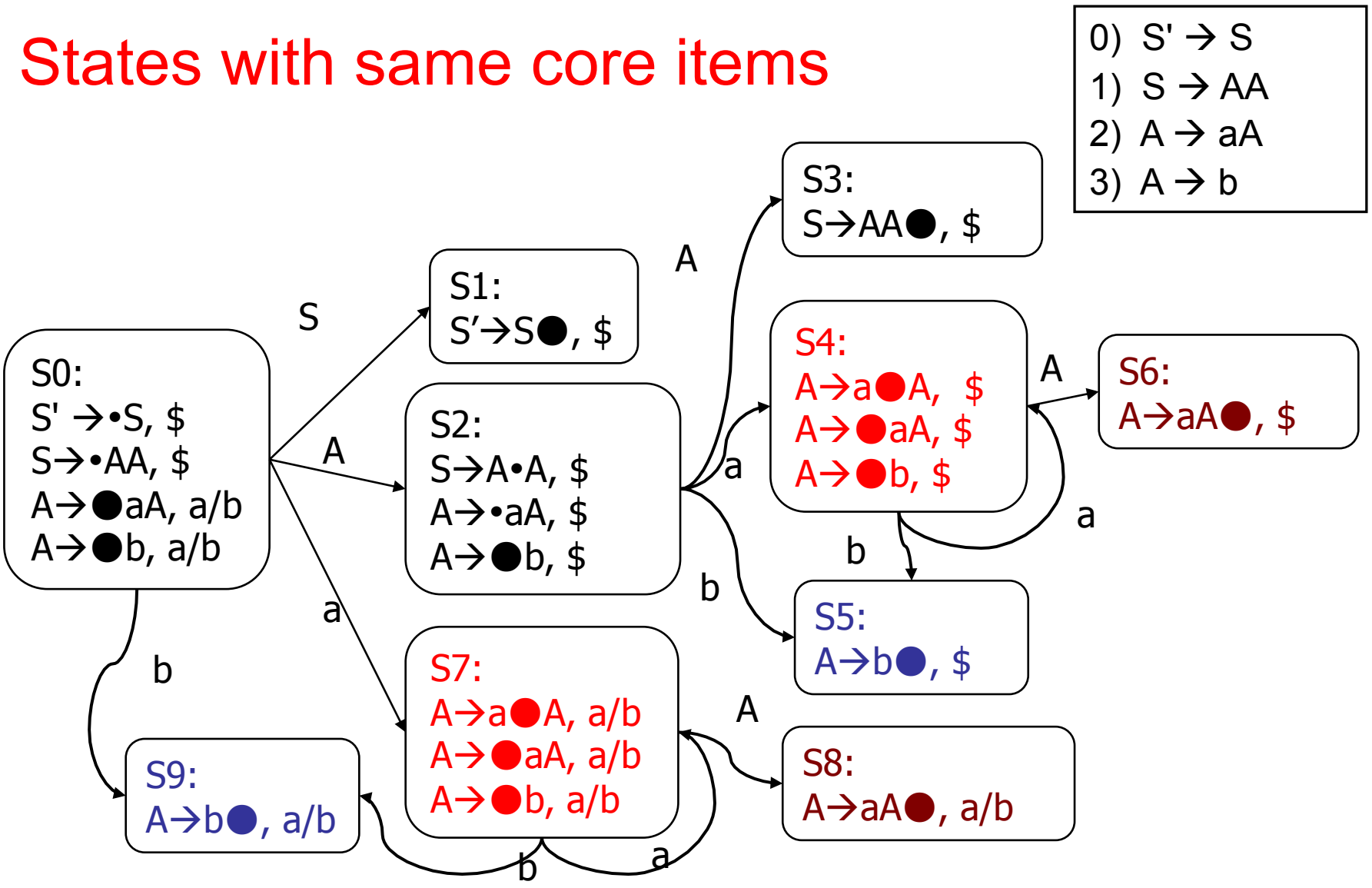
- LR(1) tables can get big – exponential in size of rules
- Can we keep the additional power we got from going SLR → LR without table explosion?
  - LALR!
- We split SLR(1) states to get LR(1) states, maybe too aggressively.
- Try to merge item sets that are *almost* identical.
- Tricky bit: Don't introduce shift-reduce or reduce-reduce conflicts.

LALR approach

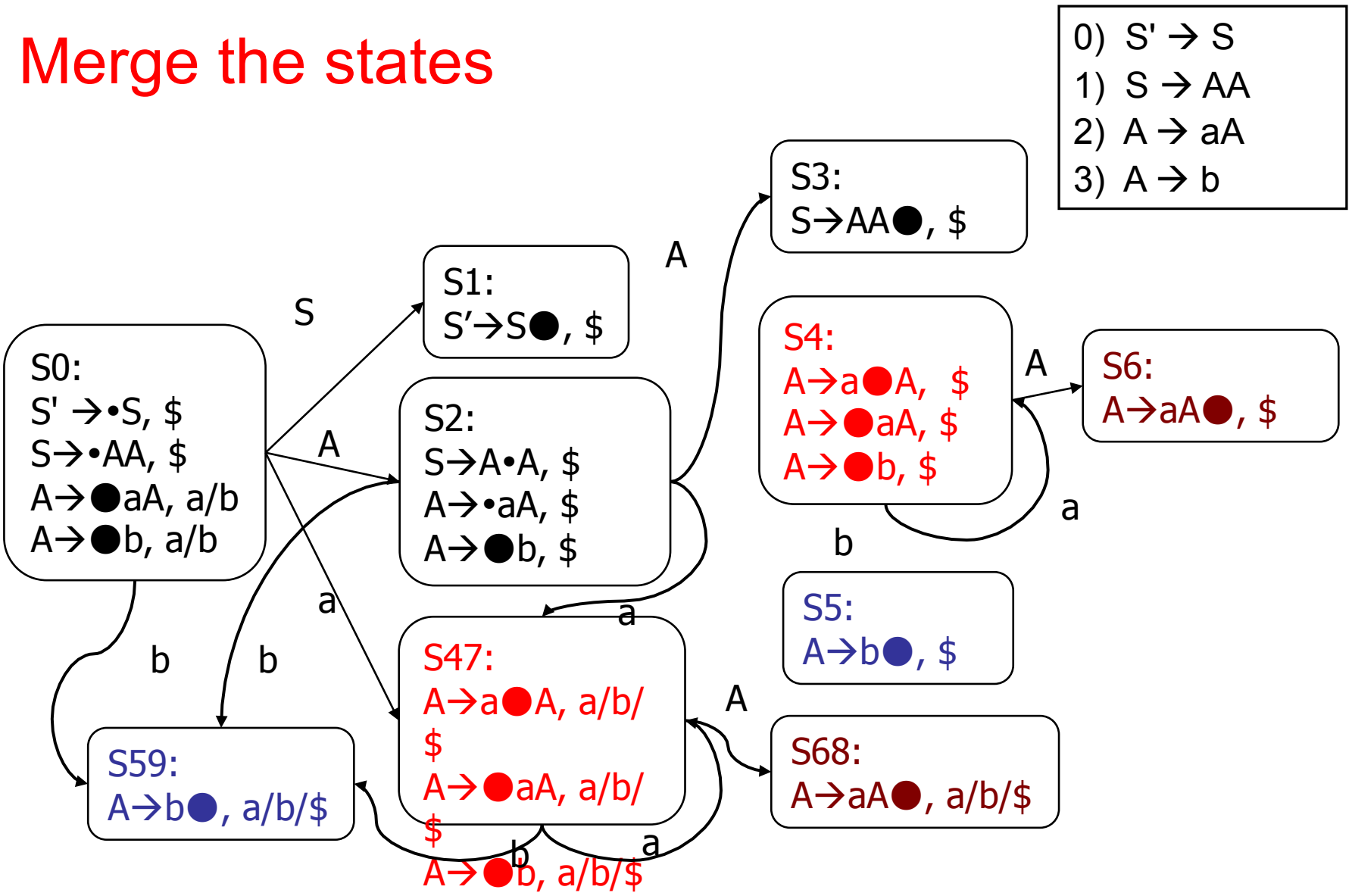
- Just say LALR (it's always 1 in practice)
- Given the numerous LR(1) states for grammar G, consider merging similar states, to get fewer states.
- Candidates for merging:
  - same core (LR(0) item)
  - only differences in lookaheads
- Example:



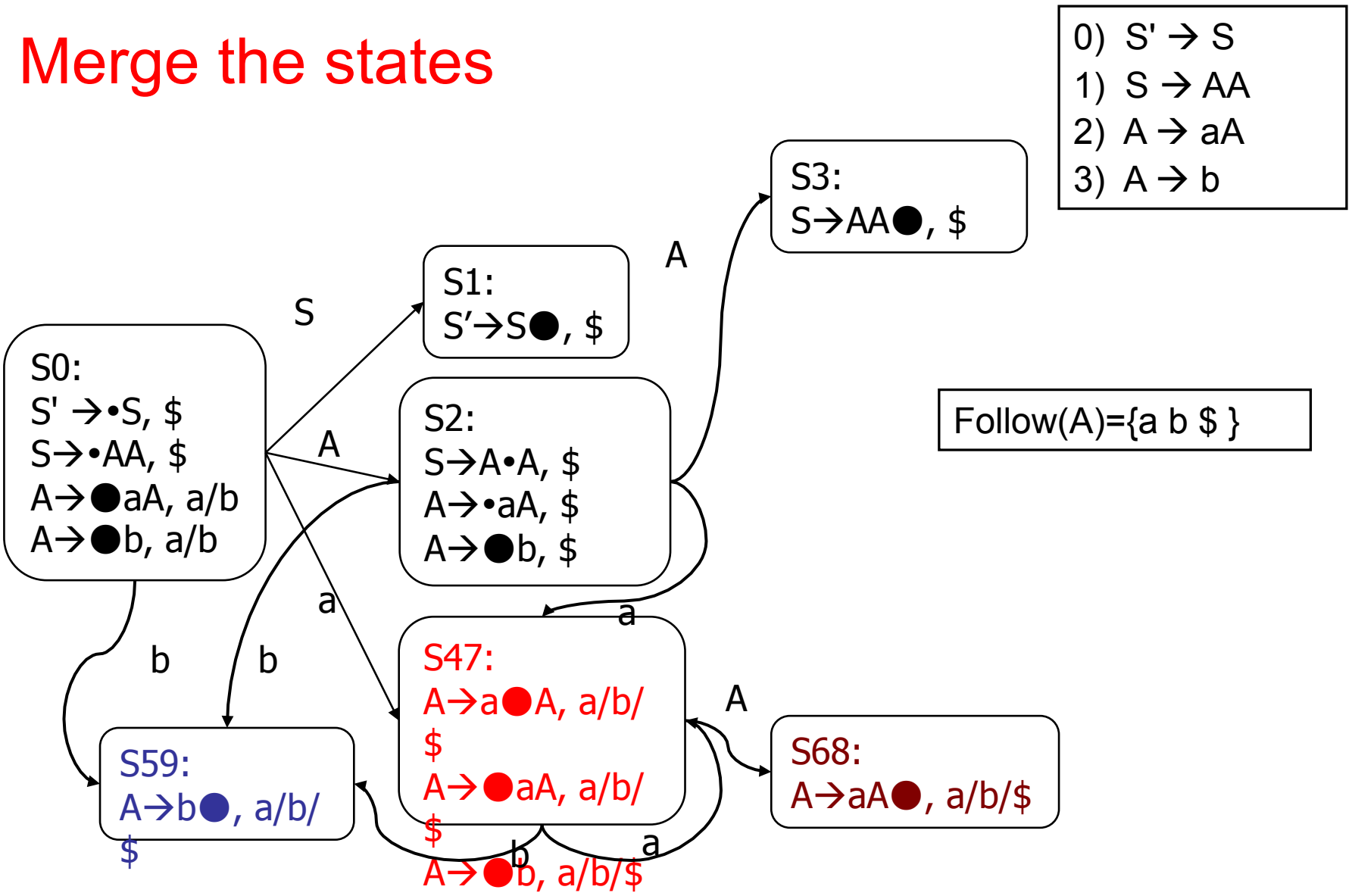
States with same core items



Merge the states



Merge the states

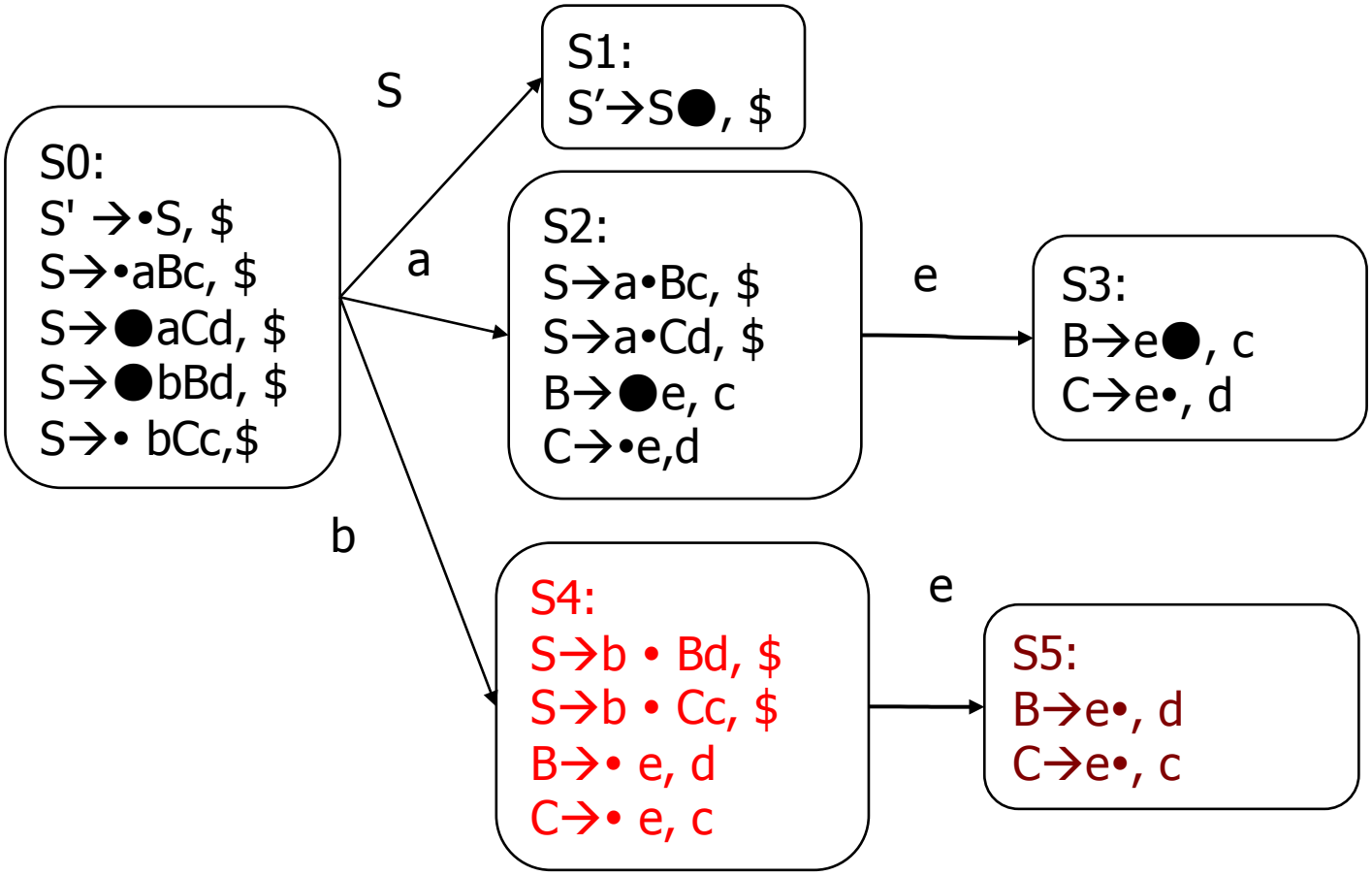


After the merge

- What happened when we merged?
  - Three fewer states
- Lookahead on items merged.
- In this case, lookahead in merged sets constitutes entire Follow set.
- So, we made SLR(1) grammar by merging.
- Result of merge usually not SLR(1).

conflict after merging

1) S →aBc|aCd|bBd|bCc
2) B→ e
3) C→ e



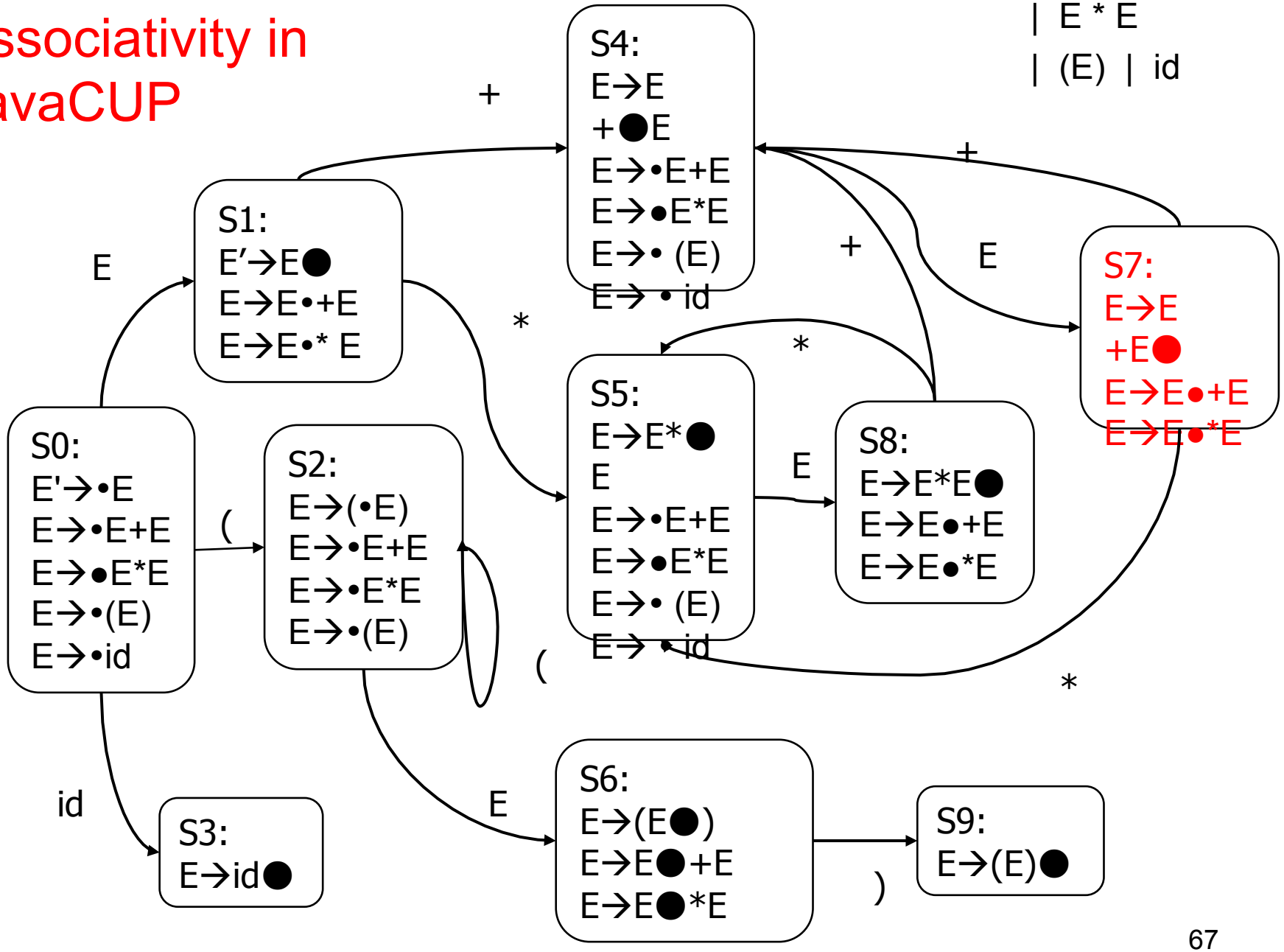
Practical consideration

- Ambiguity in LR grammars G: G produces multiple rightmost derivations. (i.e. can build two different parse trees for one input string.)
- Remember:

  $E \rightarrow E + E \mid E * E \mid (E) \mid id$
- We added terms and factors to force unambiguous parse with correct precedence and associativity
- What if we threw the grammar into an LR-grammar table-construction machine anyway?
- Conflicts = multiple action entries for one cell
  - We choose which entry to keep, toss others

Precedence and Associativity in JavaCUP

E → E + E
| E * E
| (E) | id

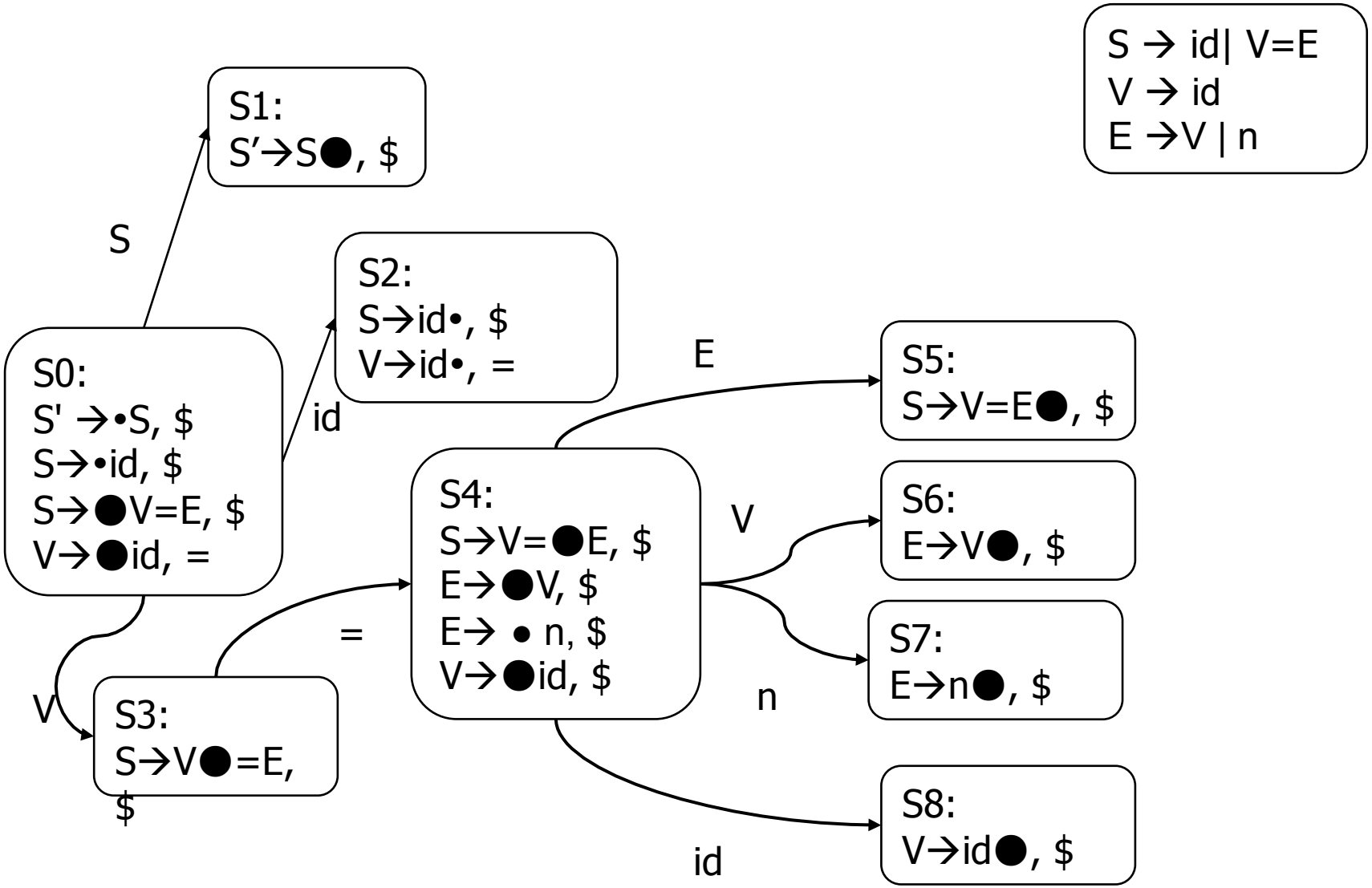


JavaCup grammar

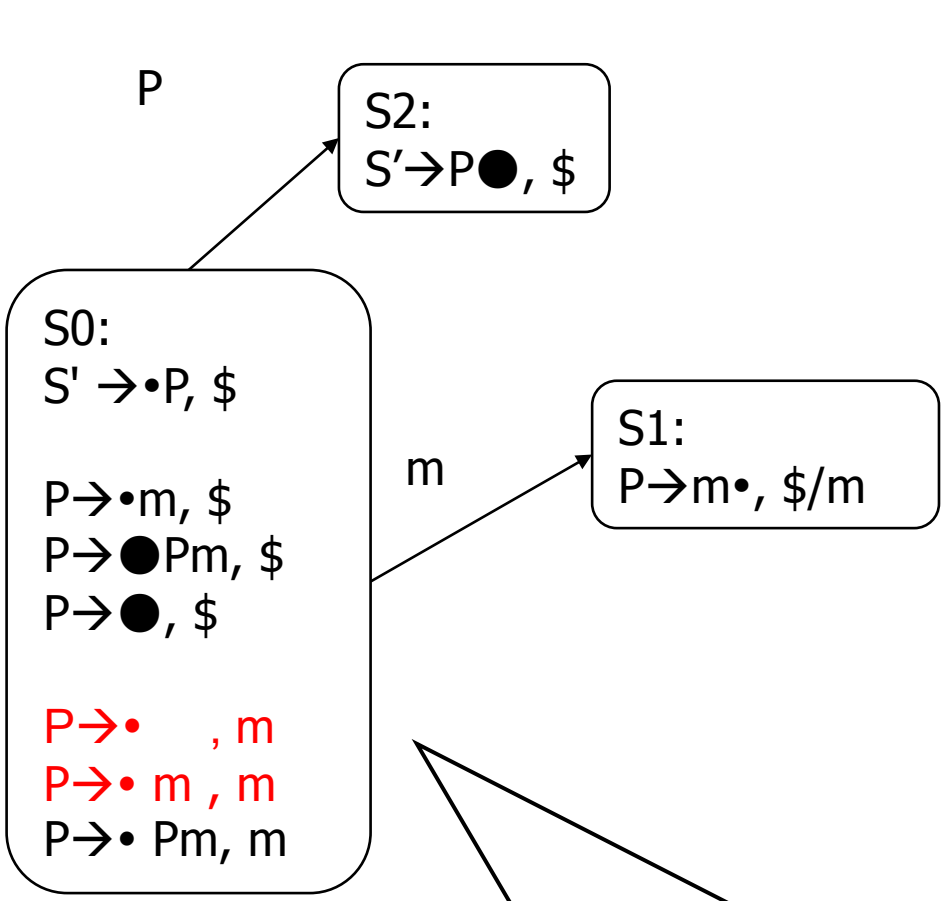
```
terminal PLUS, TIMES;
precedence left PLUS;
precedence left TIME;
E::=E PLUS E | E TIMES E |  ID
```

- What if the input is x+y+z?
  - When shifting + conflicts with reducing a production containing +, choose reduce
- What if the input is x+y*z?
- What if the input is x*y+z?

Transition diagram for assignment expr



Why are there conflicts in some rules in assignments?



Non LR(1) grammar
P→ m | Pm | ε

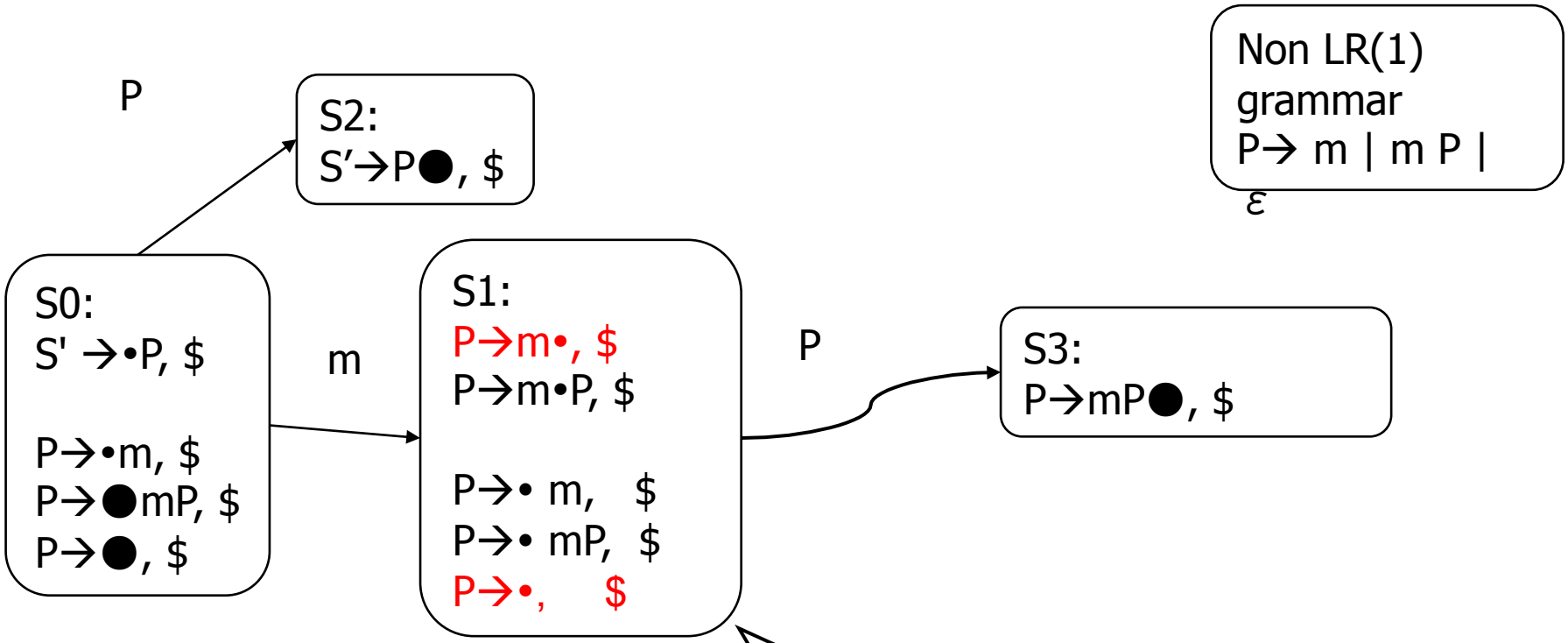
It is an ambiguous grammar. There are two rightmost/leftmost derivations for sentence m:

P➔Pm➔m

P➔m

*** Shift/Reduce conflict found in state #0
between P ::= (*)
and P ::= (*) m
under symbol m

a slightly changed grammar, still not LR



It is an ambiguous grammar. There are two parse trees for sentence m:

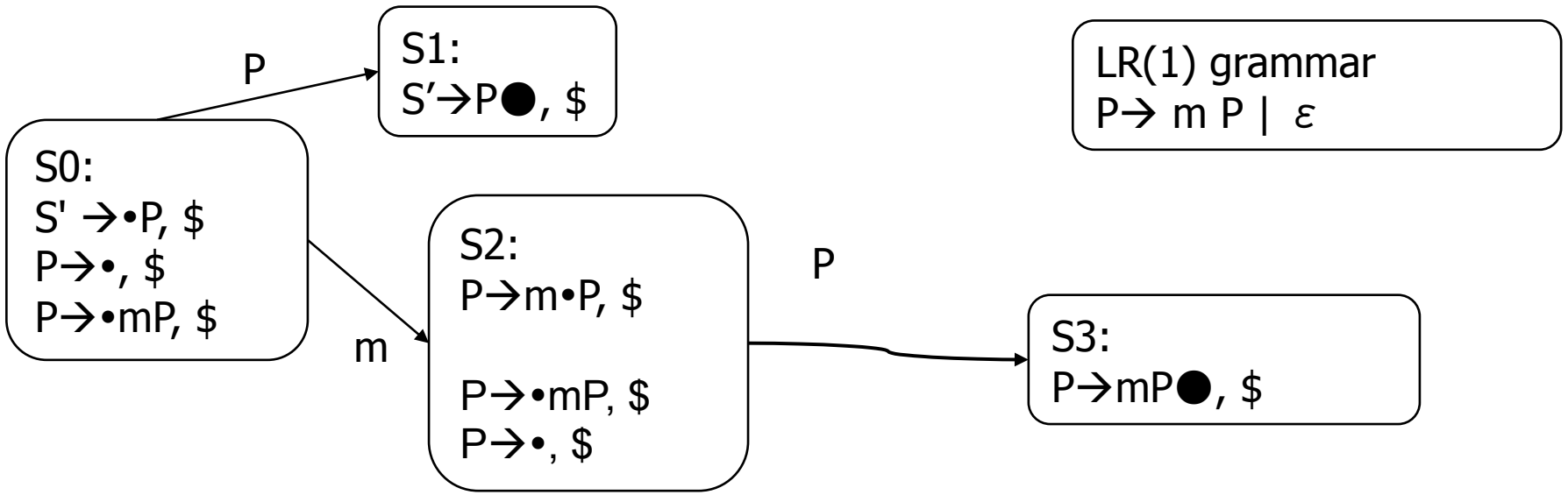
P➔mP ➔m

P➔m

Reduce/Reduce conflict found in state #1
between P ::= m (*)
and P ::= (*)
under symbols: {EOF}

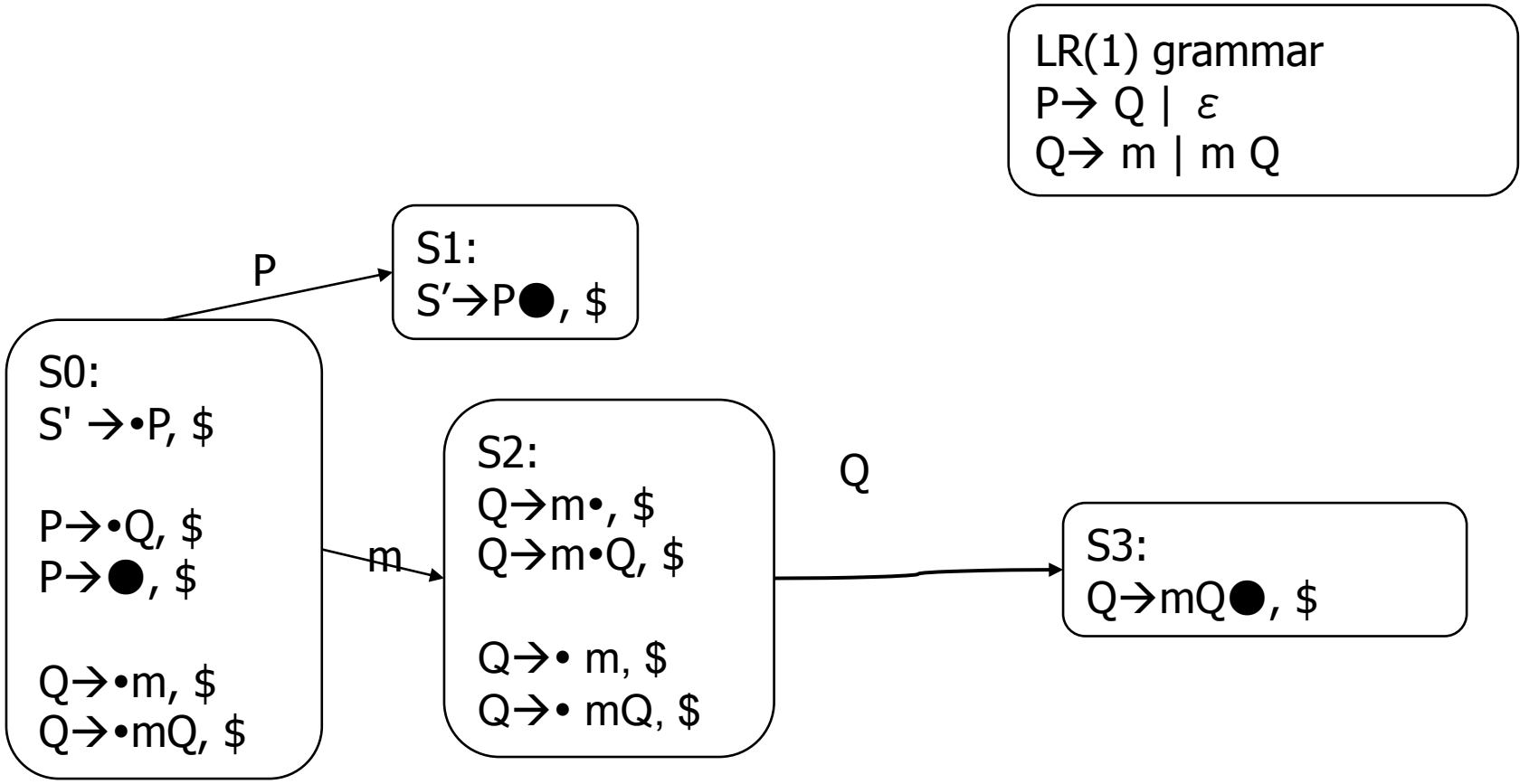
Produced from javacup

Modified LR(1) grammar



- Note that there are no conflicts

Another way of changing to LR(1) grammar

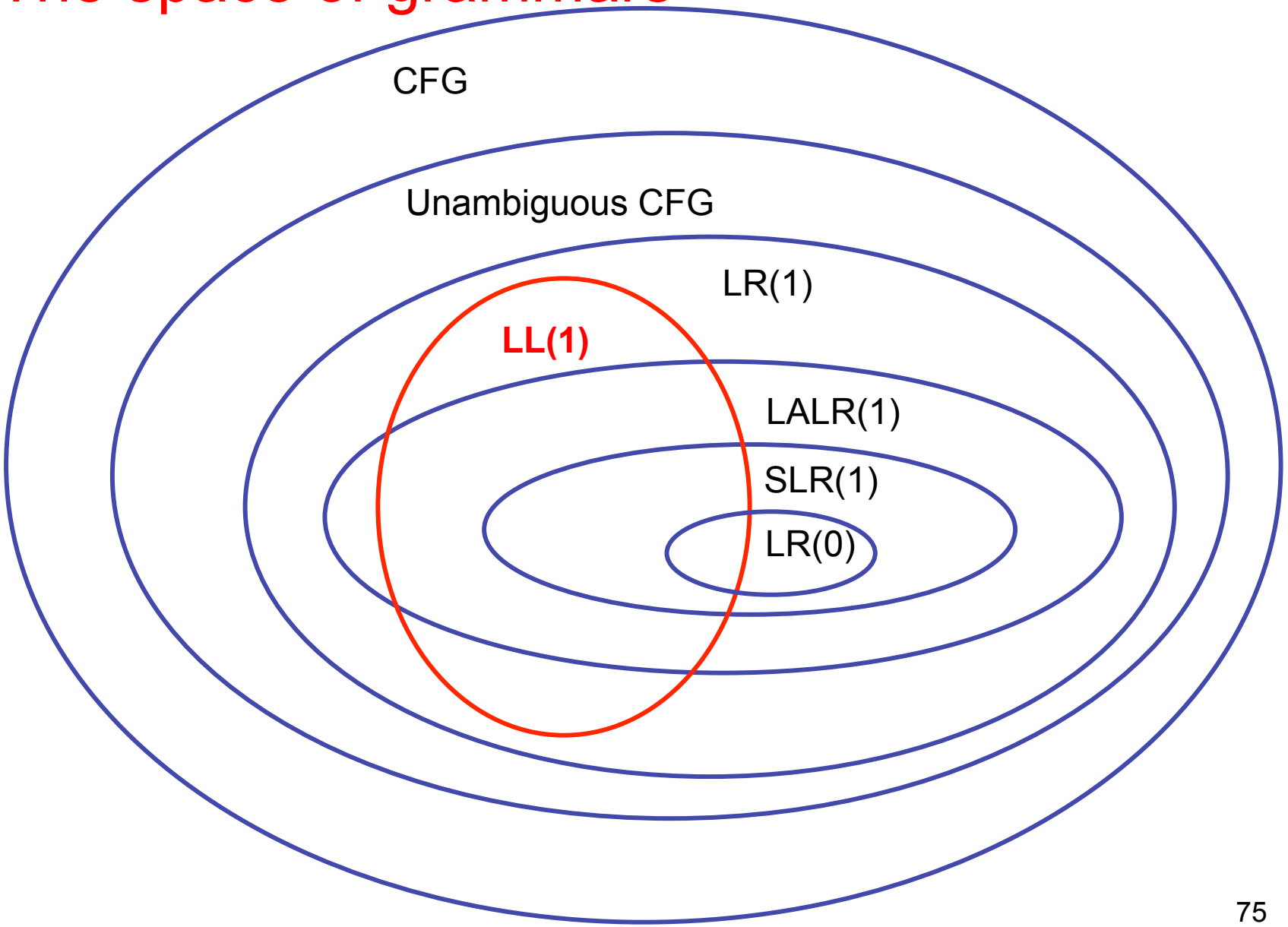


LR grammars: comparison

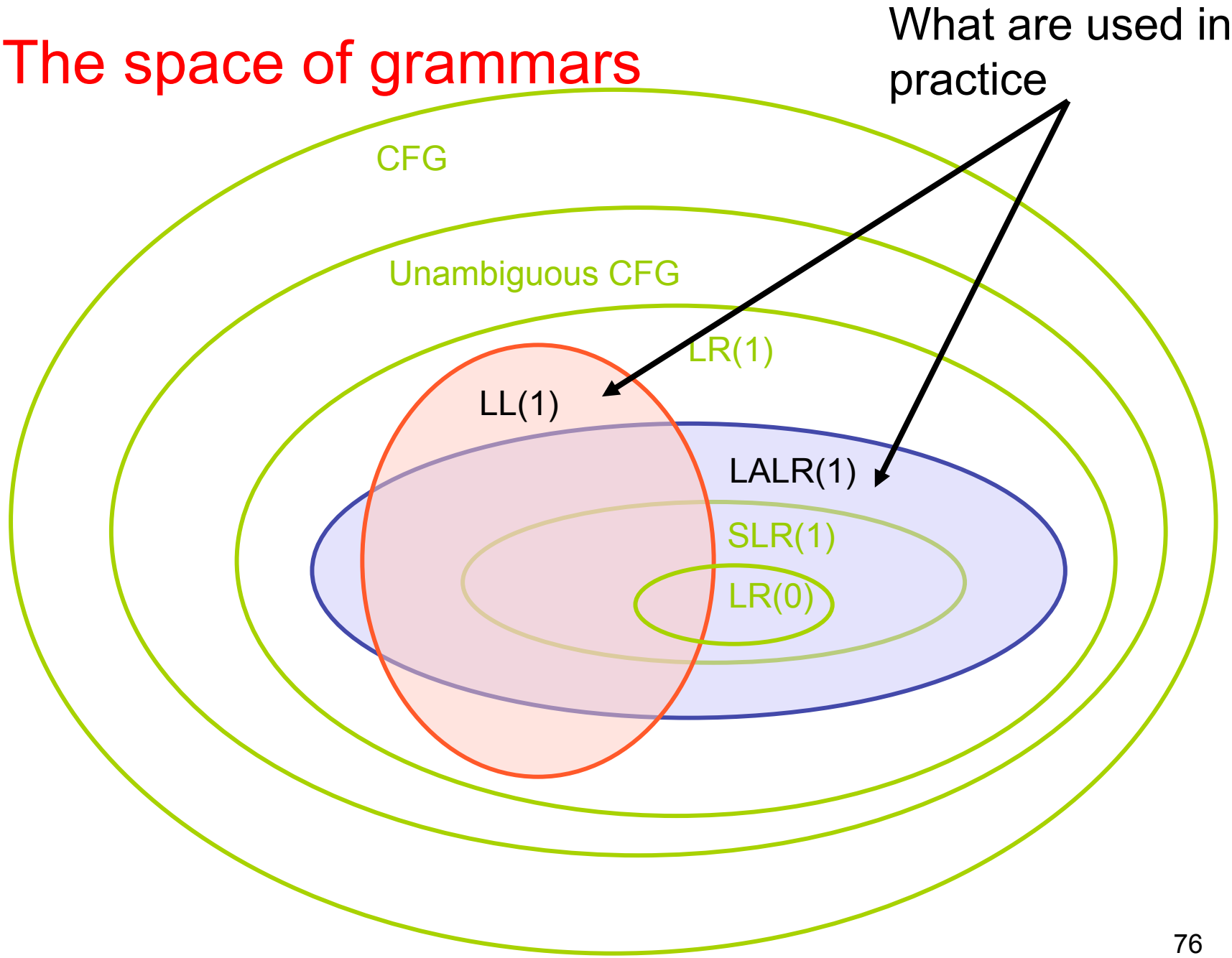
$$LR(0) \subset SLR(1) \subset LALR \subset LR(1) \subset CFG$$

	Advantages	Disadvantages
LR(0)	Smallest tables, easiest to build	Inadequate for many PL structures
SLR(1)	More inclusive, more information than LR(0)	Many useful grammars are not SLR (1)
LALR(1)	Same size tables as SLR, more langs, efficient to build	empirical, not mathematical
LR(1)	Most precise use of lookahead, most PL structures we want	Tables order of magnitude > SLR(1)

The space of grammars



The space of grammars

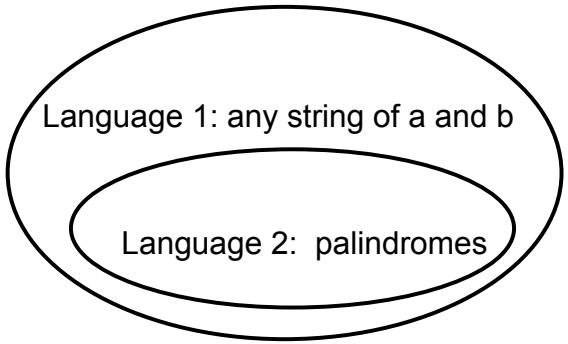


Verifying the language generated by a grammar

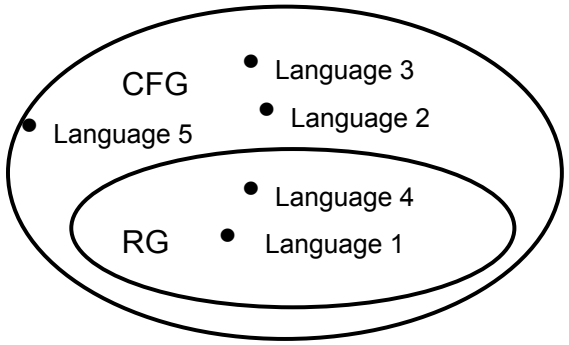
- To verify a grammar:
  - every string generated by G is in L
  - every string in L can be generated by G
- Example: S→(S)S|ε
  - the language is all the strings of balanced parenthesis, such as (), (()), ()(()())
- Proof part 1: every sentence derived from S is balanced.
  - basis: empty string is balanced.
  - induction: suppose that all derivations fewer than n steps produce balanced sentences, and consider a leftmost derivation of n steps. such a derivation must be of the form:

    $S \Rightarrow (S)S \Rightarrow^* (x)S \Rightarrow^* (x)y$
- Proof part 2: every balanced string can be derived from S
  - Basis: the empty string can be derived from S.
  - Induction: suppose that every balanced string of length less than 2n can be derived from S. Consider a balanced string w of length 2n. w must start with (. w can be written as (x)y, where x, y are balanced.

Hierarchy of grammars



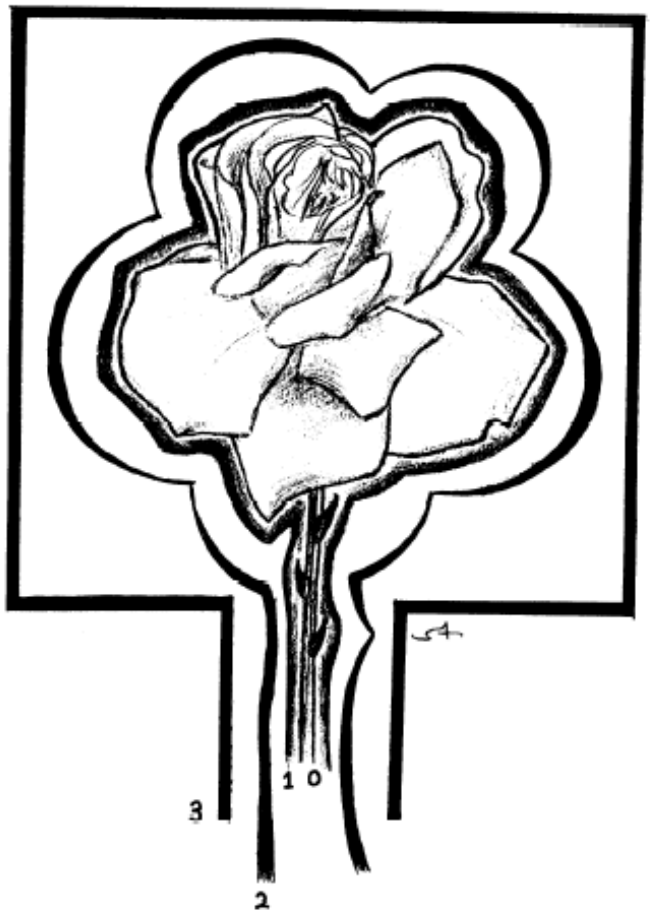
- CFG is more powerful than RE
- "Type n grammar is more powerful than type n+1 grammar"
- Example: Σ={a, b}
- The language of any string consists of a and b
  - A→aA|bA|ε
  - Can be describe by RE
- The language of palindromes consist of a and b
  - A→aAa | bAb |a|b|ε
  - Can be described by CFG, but not RE



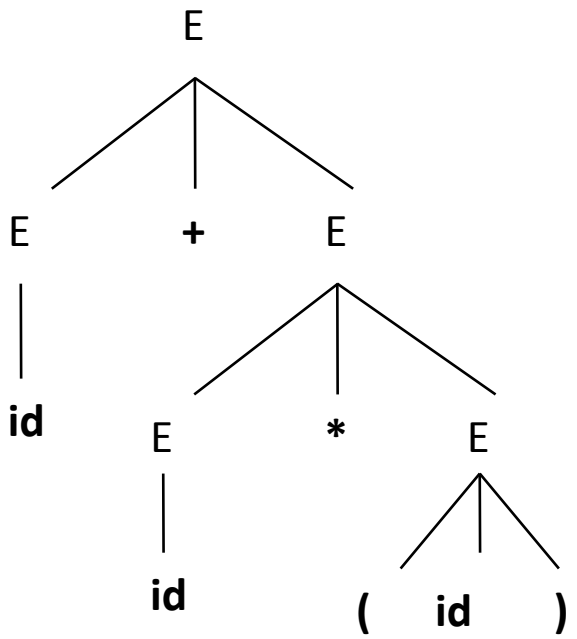
- When a grammar is more powerful, it is not that it can describe a larger language. Instead, the 'power' means the ability to restrict the set.
- More powerful grammar can define
  - more complicated boundary between correct and incorrect sentences.
  - Therefore, more different languages

Metaphoric comparison of grammars

- RE draw the rose use straight lines (ruler and T-square suffice)
- CFG approximate the outline by straight lines and circle segments (ruler and compasses)

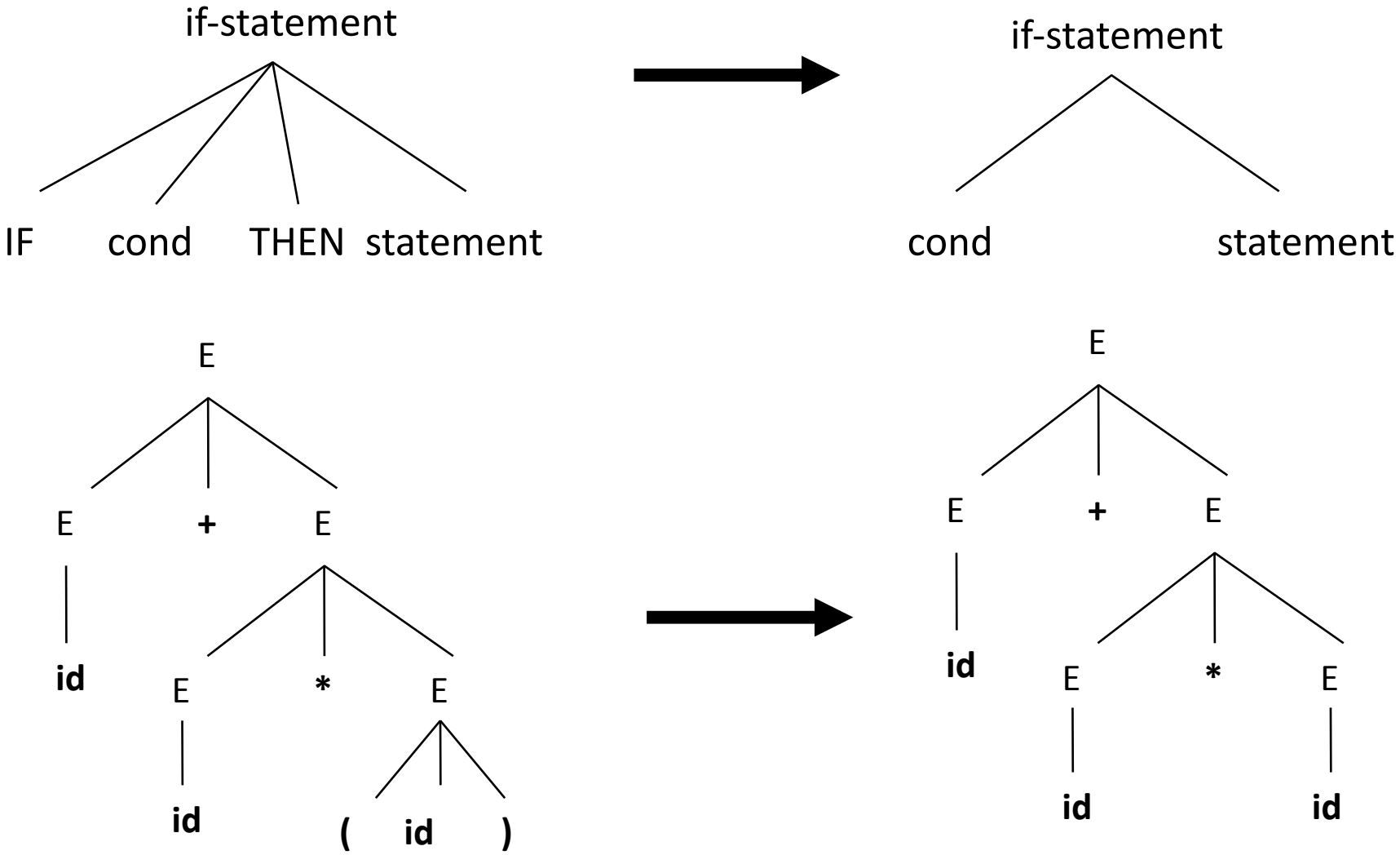


Abstract Syntax Tree--motivation



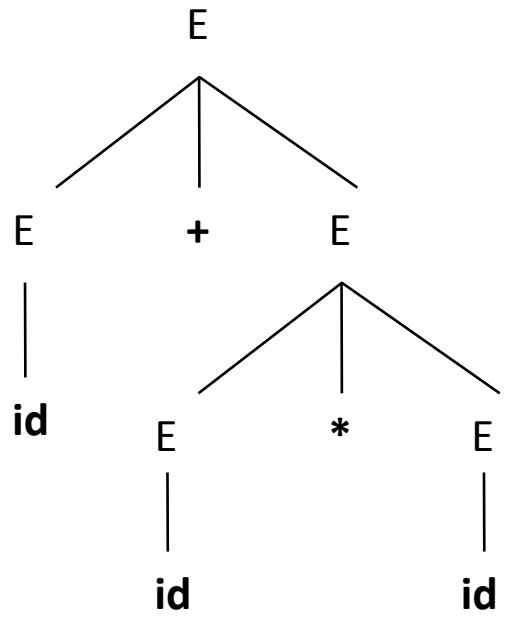
- The parse tree
  - contains too much detail
    - e.g. unnecessary terminals such as parentheses
  - depends heavily on the structure of the grammar
    - e.g. intermediate non-terminals
- Idea
  - strip the unnecessary parts of the tree, simplify it.
  - keep track only of important information
- AST
  - Conveys the syntactic structure of the program while providing abstraction.
  - Can be easily annotated with semantic information (attributes) such as type, numerical value, etc.
  - Can be used as intermediate representation.

AST vs. parse tree



Calc example

```
assignment ::= ID:e1 EQUAL expr:e2 {: RESULT = new Assignment(e1, e2); :} ;
expr ::= expr:e1 PLUS:e expr:e2 {: RESULT = new Expr(e1, e2, e); :}
    |expr:e1 MULTI:e expr:e2 {: RESULT = new Expr(e1, e2, e); :}
    | LPAREN expr:e RPAREN {: RESULT = e; :}
    | NUMBER:e {: RESULT= new Expr(e); :}
    | ID:e {: RESULT = new Expr(e); :}
```



Interpreter and translator example

```
expr      ::= expr:e1 PLUS expr:e2 {: RESULT = new Integer(e1.intValue()+ e2.intValue()); :}
            | expr:e1 MINUS expr:e2  {: RESULT = new Integer(e1.intValue()- e2.intValue());  :}
            | expr:e1 TIMES expr:e2  {: RESULT = new Integer(e1.intValue()* e2.intValue());  :}
            | expr:e1 DIVIDE expr:e2  {: RESULT = new Integer(e1.intValue()/ e2.intValue()); :}
            | LPAREN expr:e RPAREN {: RESULT = e;                          :}
            | NUMBER:e {: RESULT= e; :}    ;
```

- What is abstract is dependent on the application

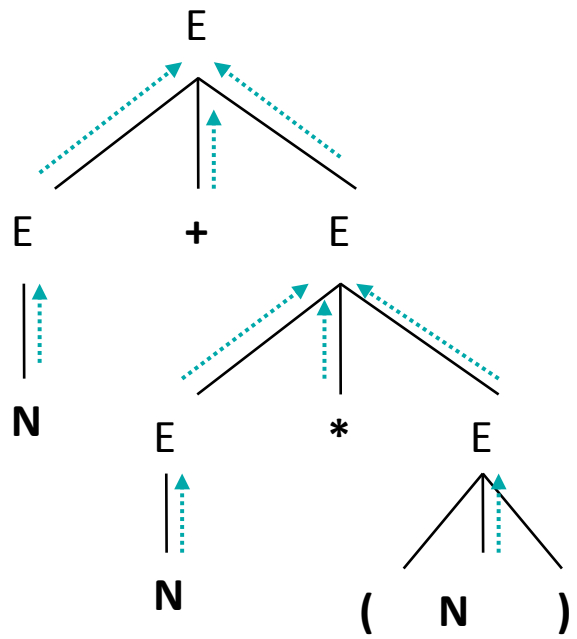
```
expr  ::= expr:e1 PLUS expr:e2   {: RESULT=e1+"+"+e2; :}
            | expr:e1 MINUS expr:e2  {: RESULT=e1+"-"+e2; :}
            | expr:e1 TIMES expr:e2  {: RESULT=e1+"*"+e2; :}
            | expr:e1 DIVIDE expr:e2  {: RESULT=e1+"/"+e2; :}
            | LPAREN expr:e RPAREN   {: RESULT="("+e+")"; :}
            | NUMBER:e  {: RESULT=e; :}
            | ID:e   {: RESULT=e; :}
            | fctCall:e   {: RESULT=e; :}
```

Attribute grammar

- Formal framework based on grammar and parse tree
- “attribute” the tree
  - Can add attributes (fields) to each node
  - augment grammar with rules defining attribute values
  - high-level specification, independent of evaluation scheme
  - Note: translation scheme has evaluation order
  - both inherited and synthesized attributes
- Attribute grammars are very general. Can be used for
  - infix to postfix translation of arithmetic expressions
  - type checking (context-sensitive analysis)
  - construction of intermediate representation (AST)
  - desk calculator (interpreter)
  - code generation (compiler)
- Another name for syntax directed translation

Dependencies among attributes

- values are computed from constants & other attributes
- synthesized attribute - value computed from children
  - attribute of left-hand side is computed from attributes in the right-hand side
  - bottom-up propagation
- inherited attribute - value computed from siblings & parent
  - attribute of symbol on right-hand is computed from attributes of left-hand side, or from attributes of other symbols on right-hand side
  - top-down propagation of information




```
expr    ::= expr:e1 PLUS expr:e2 {: RESULT = new Integer(e1.intValue()+ e2.intValue()); :}
          | expr:e1 MINUS expr:e2  {: RESULT = new Integer(e1.intValue()- e2.intValue());  :}
          | expr:e1 TIMES expr:e2  {: RESULT = new Integer(e1.intValue()* e2.intValue());  :}
          | expr:e1 DIVIDE expr:e2  {: RESULT = new Integer(e1.intValue()/ e2.intValue()); :}
          | LPAREN expr:e RPAREN {: RESULT = e;                          :}
          | NUMBER:e {: RESULT= e; :}    ;
```

Attributes

- For terminal: define some computable properties
  - e.g. the value of NUMBER
- For non-terminal (production): give computation rules for the properties of all sy
  - e.g. the value of a sum is the sum of the values of the operands
- The rule is local: only refers to symbols in the same production
- The evaluation of the attributes can require an arbitrary number of traversals of the AST: arbitrary context dependence (.e.g. the value of a declared constant is found in the constant declaration)
- Attribute definitions may be cyclic; checking whether an attribute grammar has cycles is decidable but potentially expensive
- In practice inherited attributes are handled by means of global data structures (symbol table)

```
expr     ::= expr:e1 PLUS expr:e2 {: RESULT = new Integer(e1.intValue()+ e2.intValue()); :}
           | expr:e1 MINUS expr:e2  {: RESULT = new Integer(e1.intValue()- e2.intValue());  :}
           | expr:e1 TIMES expr:e2  {: RESULT = new Integer(e1.intValue()* e2.intValue());  :}
           | expr:e1 DIVIDE expr:e2  {: RESULT = new Integer(e1.intValue()/ e2.intValue()); :}
           | LPAREN expr:e RPAREN {: RESULT = e;                          :}
           | NUMBER:e {: RESULT= e; :}    ;
```

Some examples of attributes

- For expressions: type
- For overloaded calls: candidate interpretations
- For identifiers: entity (defining_occurrence)
- For definitions: scope
- For data/function members: visibility (public, protected, private)
- …

Syntactic and semantic analysis

- Syntactic analysis generates a parse tree
  - Syntax analysis can not capture all the errors
  - Some rules are beyond context free grammar
    - e.g., a variable declaration needs to occur before the use of the variable
- Semantic analysis enforce context-dependent language rules that are not reflected in the BNF

- Semantic analysis adds semantic information to the parse tree/AST
  - e.g. determine types of all expressions
- General framework: compute attributes

Examples of semantic rules

- Variables
  - must be defined before being used
  - should not be defined multiple times
- Types
  - In an assignment stmt, the variable and the expression must have the same type
  - The test exprssion of an if statement must have boolean type
- Classes
  - can be defined only once
  - Inheritance relationship
  - Methods only defined once
- Reserved words can not be used as variable or function or class names
- Scope of variables etc.,
- Variable initialization
- …

Semantic analysis requirements are language dependent

Type checking

- One major category semantic analysis
- Steps
  - Type synthesis – assigning a type to each expression in the language
  - Type checking – making sure that these types are used in contexts where they are legal, catching type-related errors
- What is a type
  - Differs from language to language
  - A set of values and a set of operations on the values
  - A class is also a type
- Type checking
  - Ensures that operations are used with the correct types
- Why type checking
  - int x, y; y= x*x is fine
  - String x,y; y= x*x does not make sense
  - Should catch such type error
  - When to catch type error?