CSL302: Compiler Design

Bottom Up Parsing

Vishwesh Jatala

Assistant Professor

Department of CSE

Indian Institute of Technology Bhilai

vishwesh@iitbhilai.ac.in

Acknowledgement

- Today's slides are modified from that of Stanford University:
  - *https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/*

Recap

- LR(0) only accepts languages where the handle can be found with no **right context**.
- SLR(1) is more powerful than LR(0) but is weaker than LR(1) as it has no contextual information.
- *LR(1): Substantially* more powerful than the other methods we've covered so far (more on that later).

  Tries to more intelligently find handles by using a lookahead token at each step.

Recap

- Each state in an LR(1) automaton is a combination of an LR(0) state and lookahead information.
- Two LR(1) items have the same **core** if they are identical except for lookahead.

$T \rightarrow (\cdot E)$	\$
$E \rightarrow \cdot E + T$	)
$E \rightarrow \cdot T$	)
$T \rightarrow \cdot$ **int**	)
$T \rightarrow \cdot (E)$	)

$T \rightarrow (\cdot E)$	)
$E \rightarrow \cdot E + T$	)
$E \rightarrow \cdot T$	)
$T \rightarrow \cdot$ **int**	)
$T \rightarrow \cdot (E)$	)

LR(1) Automata are **Huge**

- In a grammar with *n* terminals, could in theory be $O(2^n)$ times as large as the LR(0) automaton.
  - Replicate each state with all $O(2^n)$ possible lookaheads.
- LR(1) tables for practical programming languages can have hundreds of thousands or even *millions* of states.
- Consequently, LR(1) parsers are rarely used in practice.

What next?

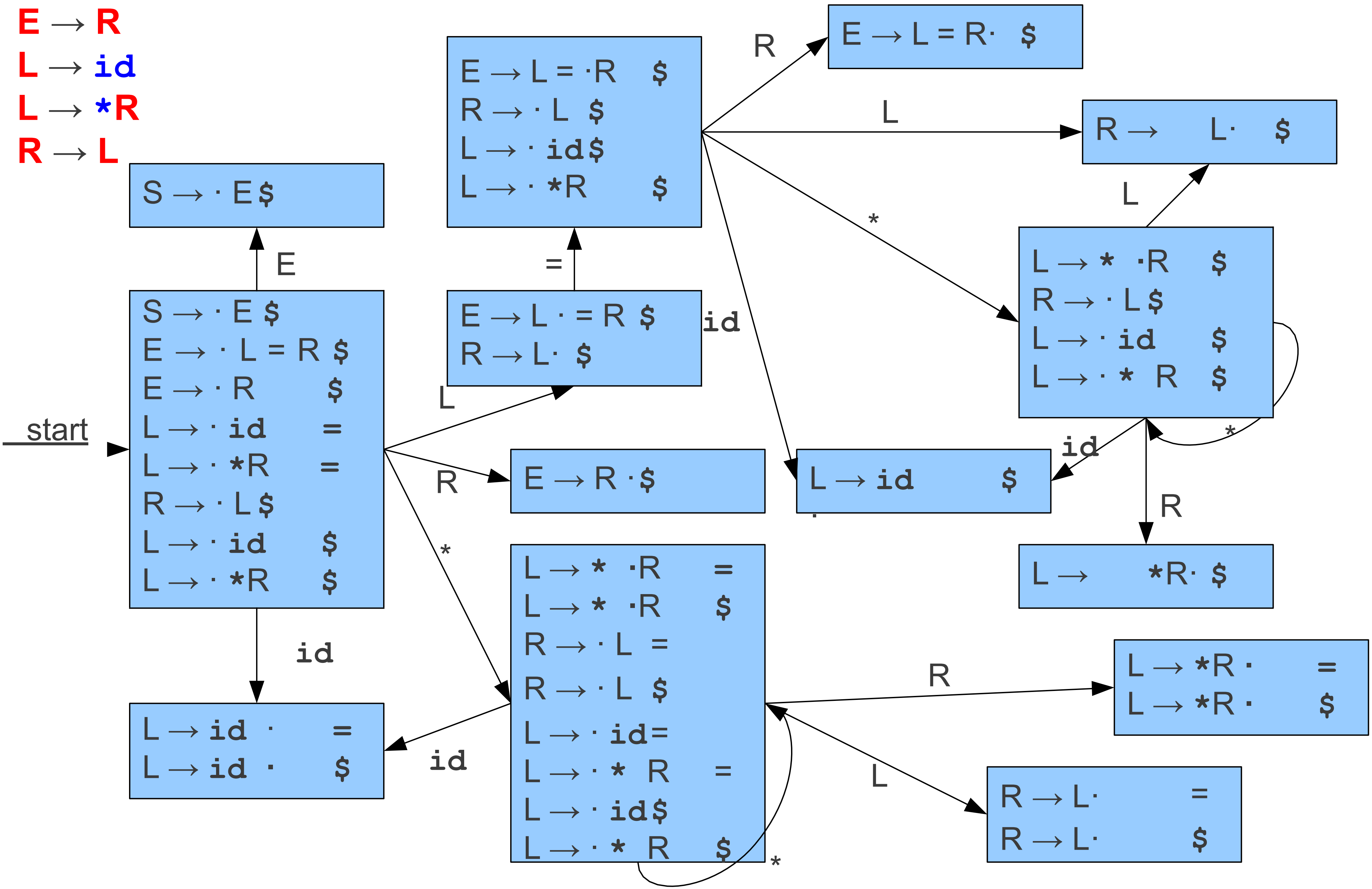
- In an LR(1) automaton, we have multiple states with the same core but different lookahead.
- What if we merge all these states together?
- This is called **LALR**
  - **Lookahead LR**

From LR(1) to LALR(1)

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

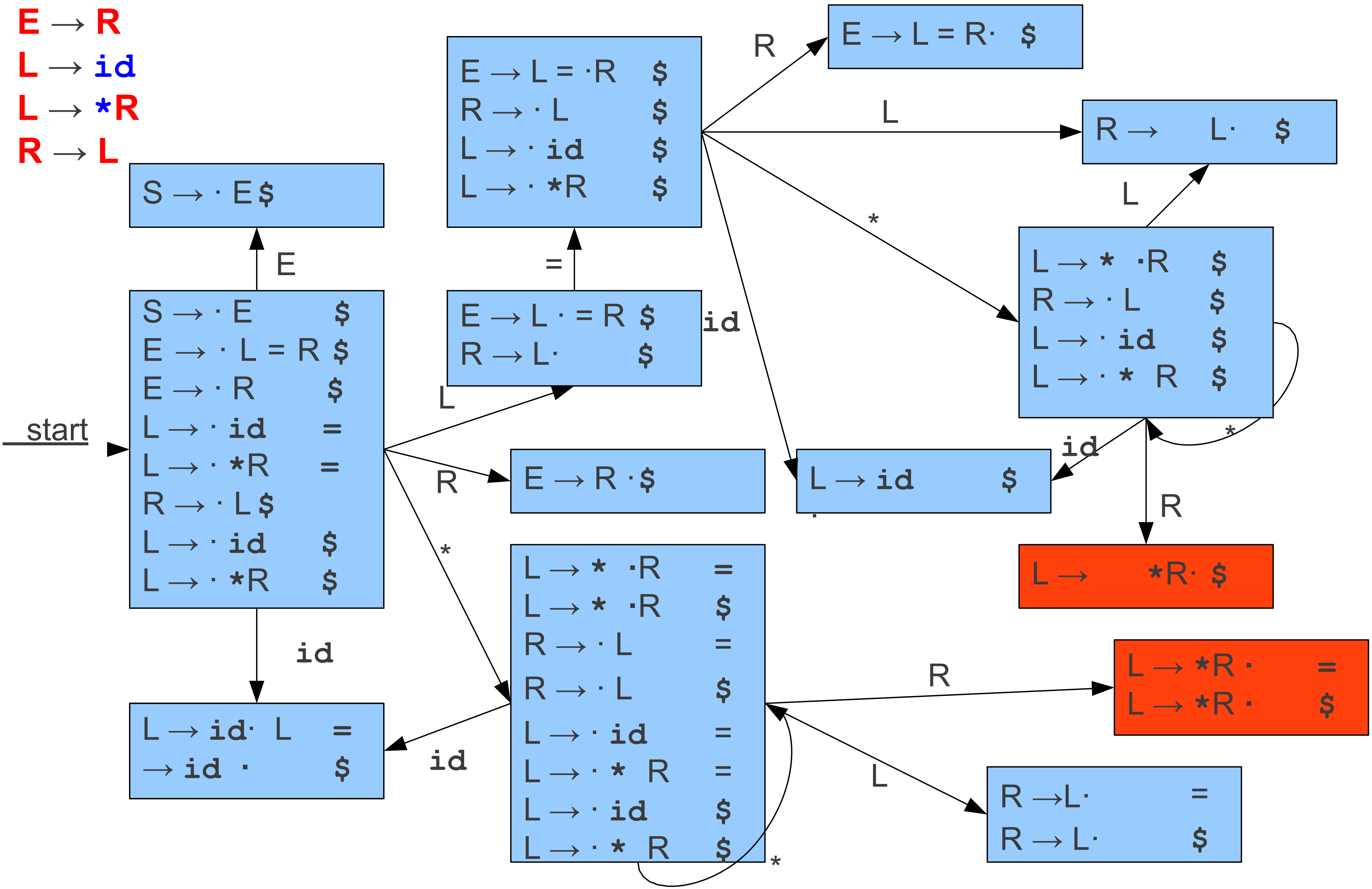
From LR(1) to LALR(1)

S → E
E → L = R
E → R
L → id
L → *R
R → L



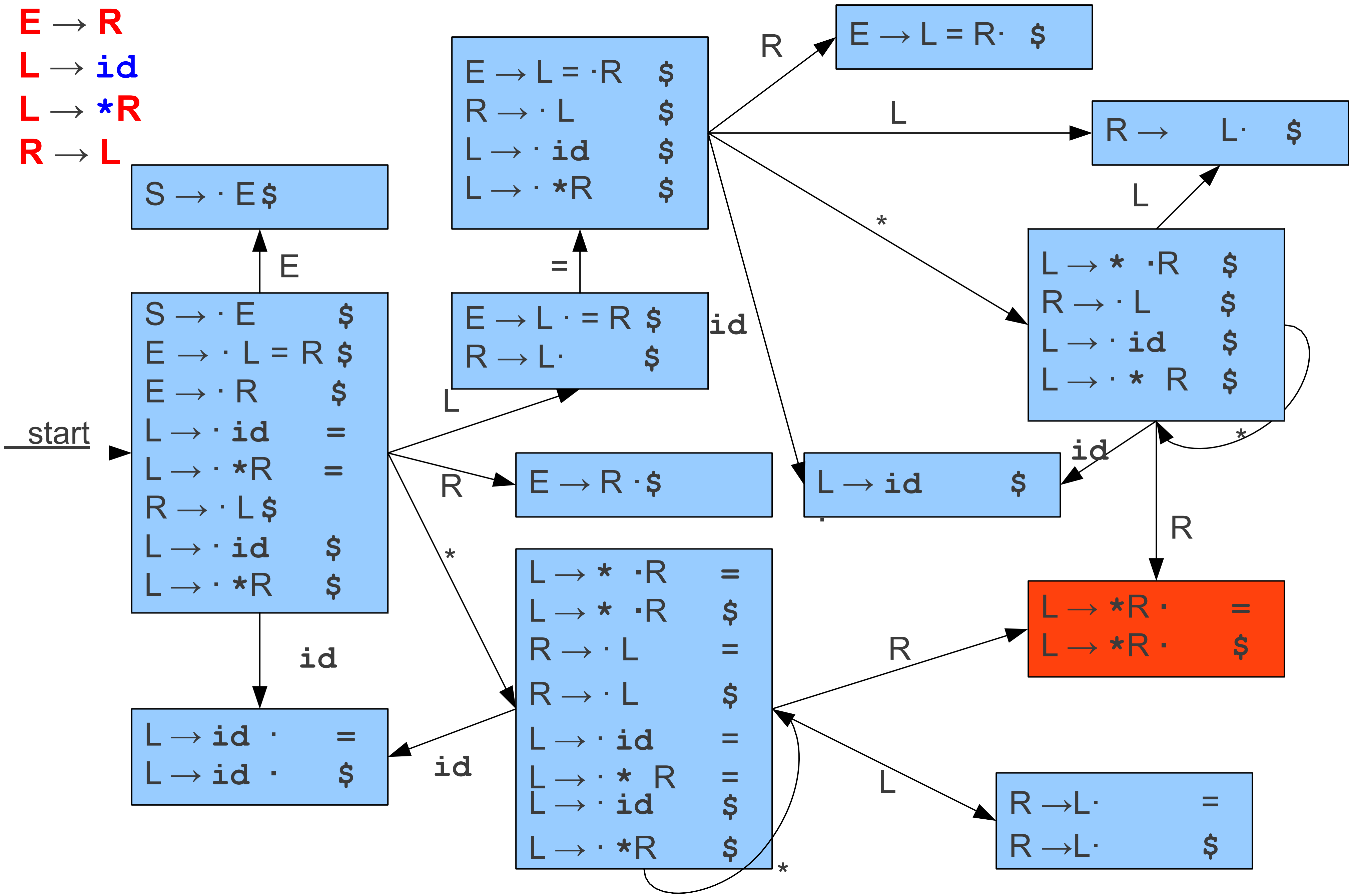
From LR(1) to LALR(1)

S → E
E → L = R
E → R
L → id
L → *R
R → L



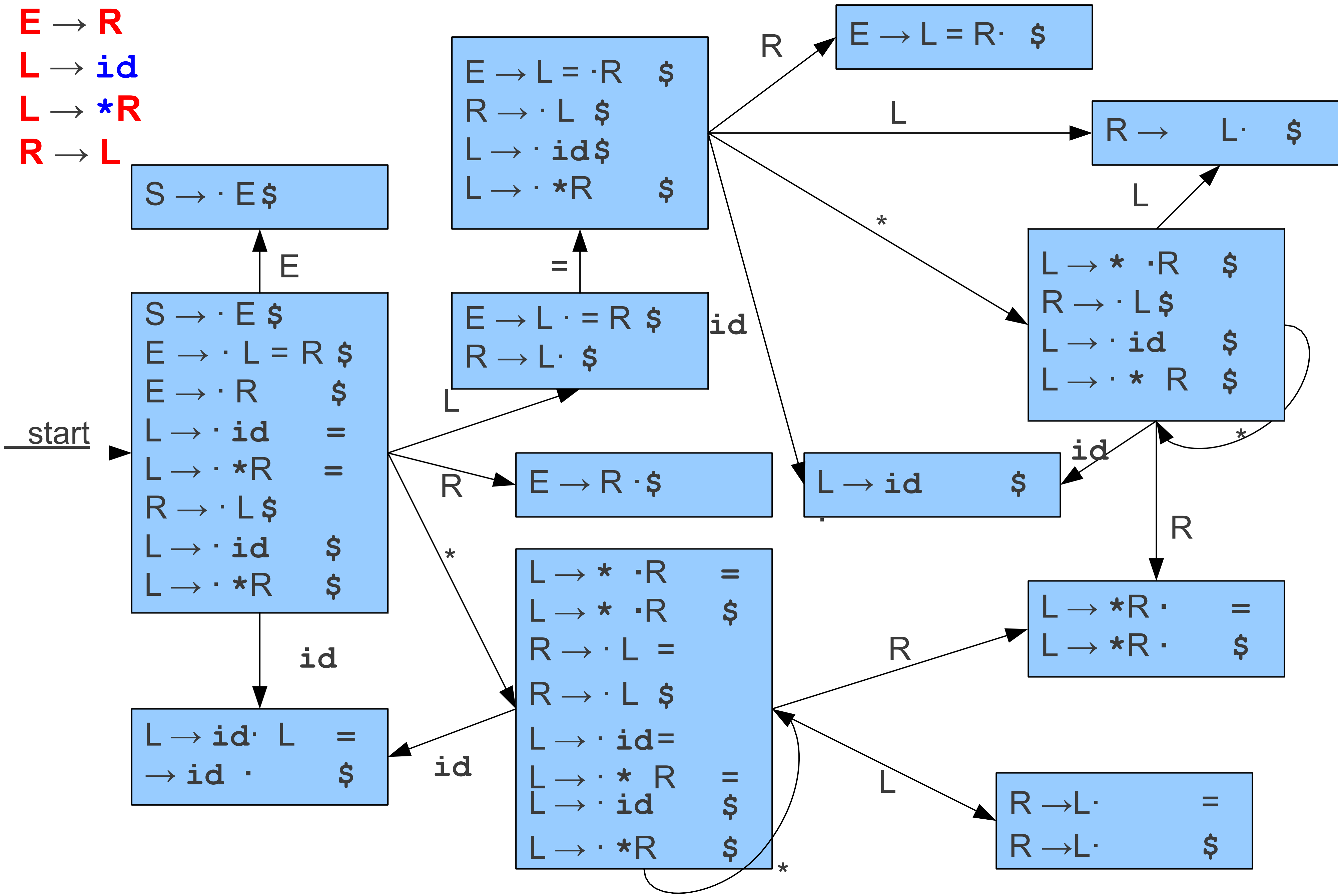
From LR(1) to LALR(1)

$S \rightarrow E$
$E \rightarrow L = R$
$E \rightarrow R$
$L \rightarrow \mathbf{id}$
$L \rightarrow *R$
$R \rightarrow L$



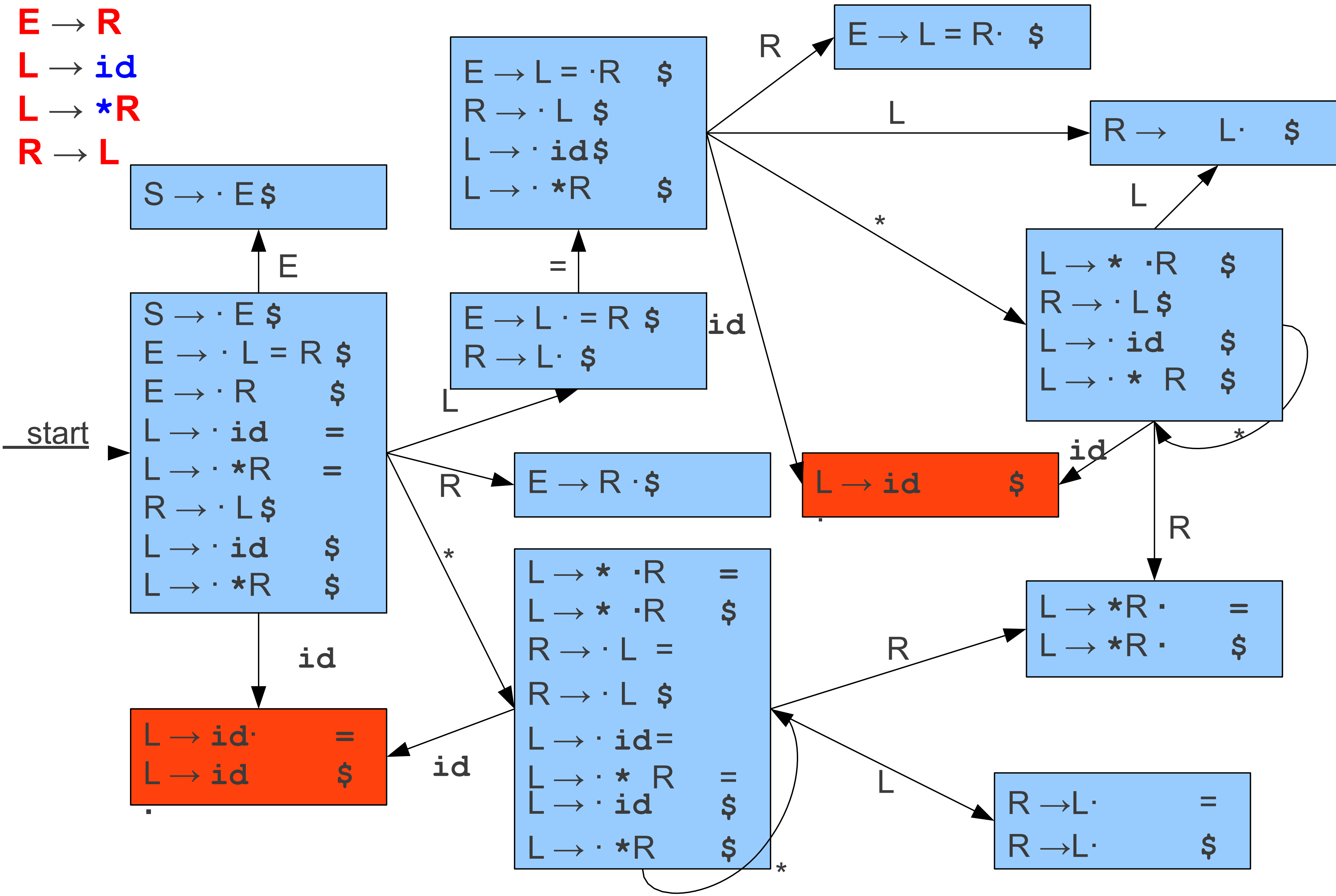
From LR(1) to LALR(1)

$S \to E$
$E \to L = R$
$E \to R$
$L \to id$
$L \to *R$
$R \to L$



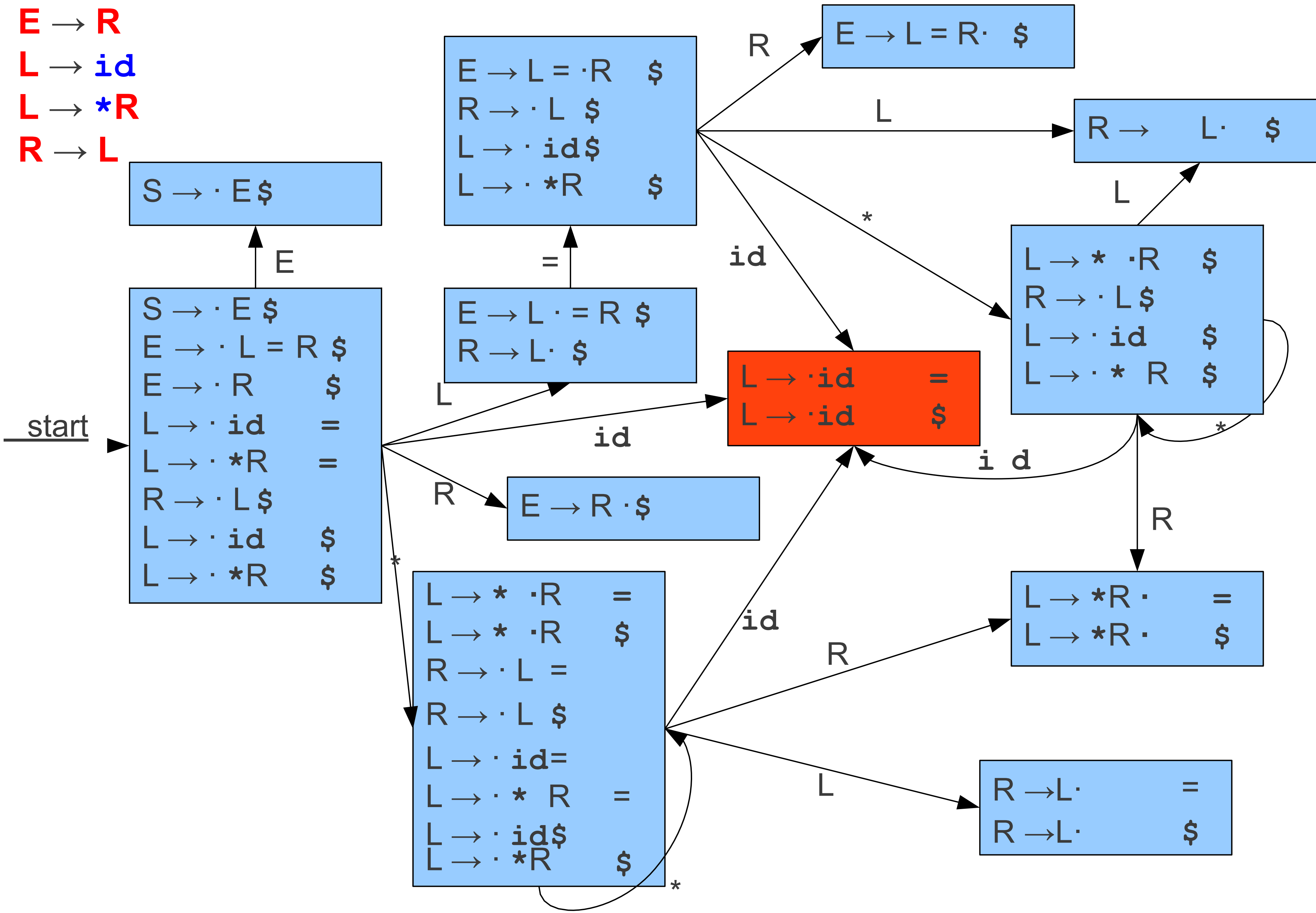
From LR(1) to LALR(1)

S → E
E → L = R
E → R
L → id
L → *R
R → L



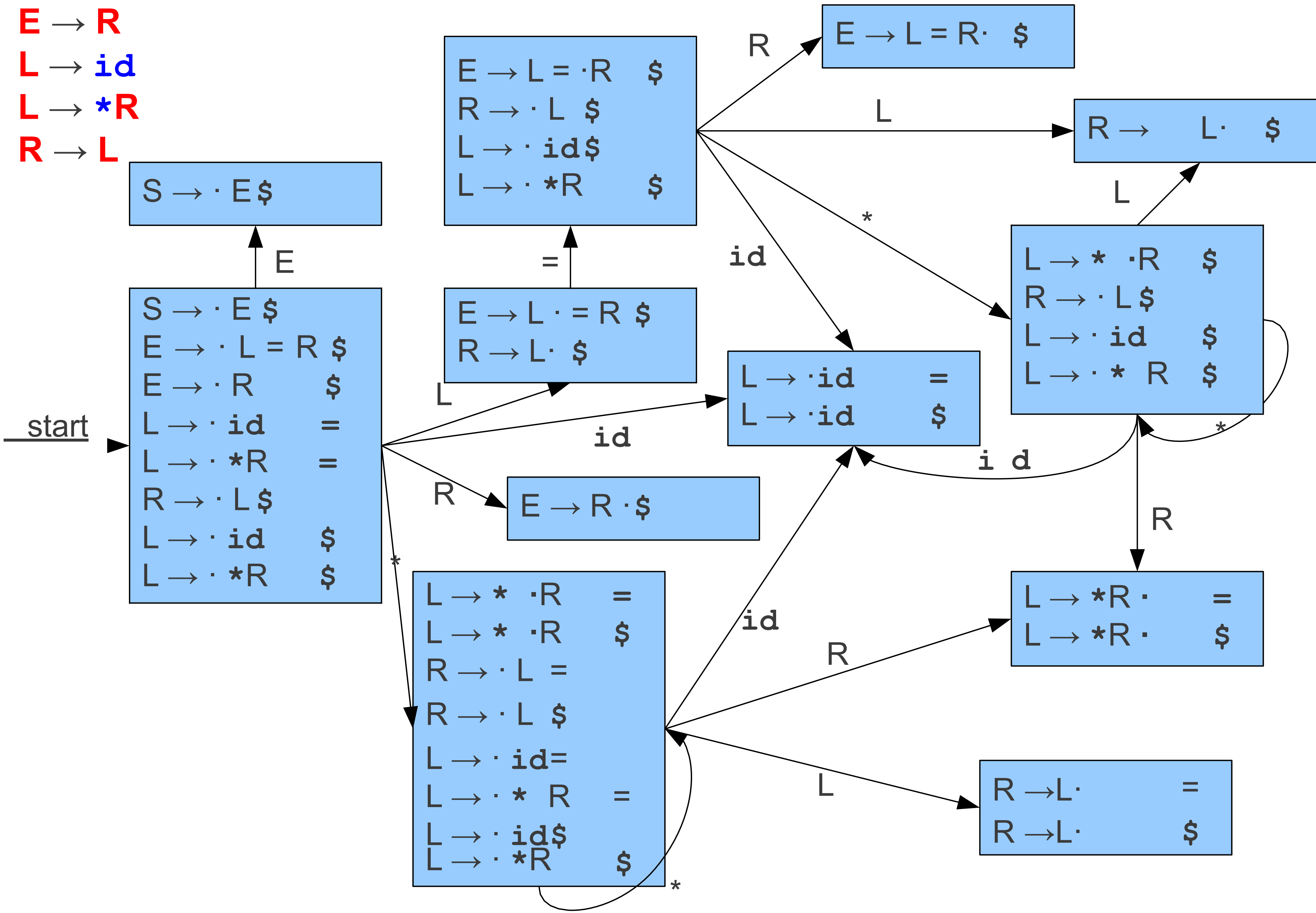
From LR(1) to LALR(1)

$S \to E$
$E \to L = R$
$E \to R$
$L \to id$
$L \to *R$
$R \to L$



From LR(1) to LALR(1)

$S \rightarrow E$
$E \rightarrow L = R$
$E \rightarrow R$
$L \rightarrow \texttt{id}$
$L \rightarrow *R$
$R \rightarrow L$



Another Example: LR(1) to LALR(1)

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow cC | d$

Exercise: Construct DFA for LR(1)

Figure source: Suggested course textbook

Another Example: LR(1) to LALR(1)

S' → S
S → CC
C → cC|d

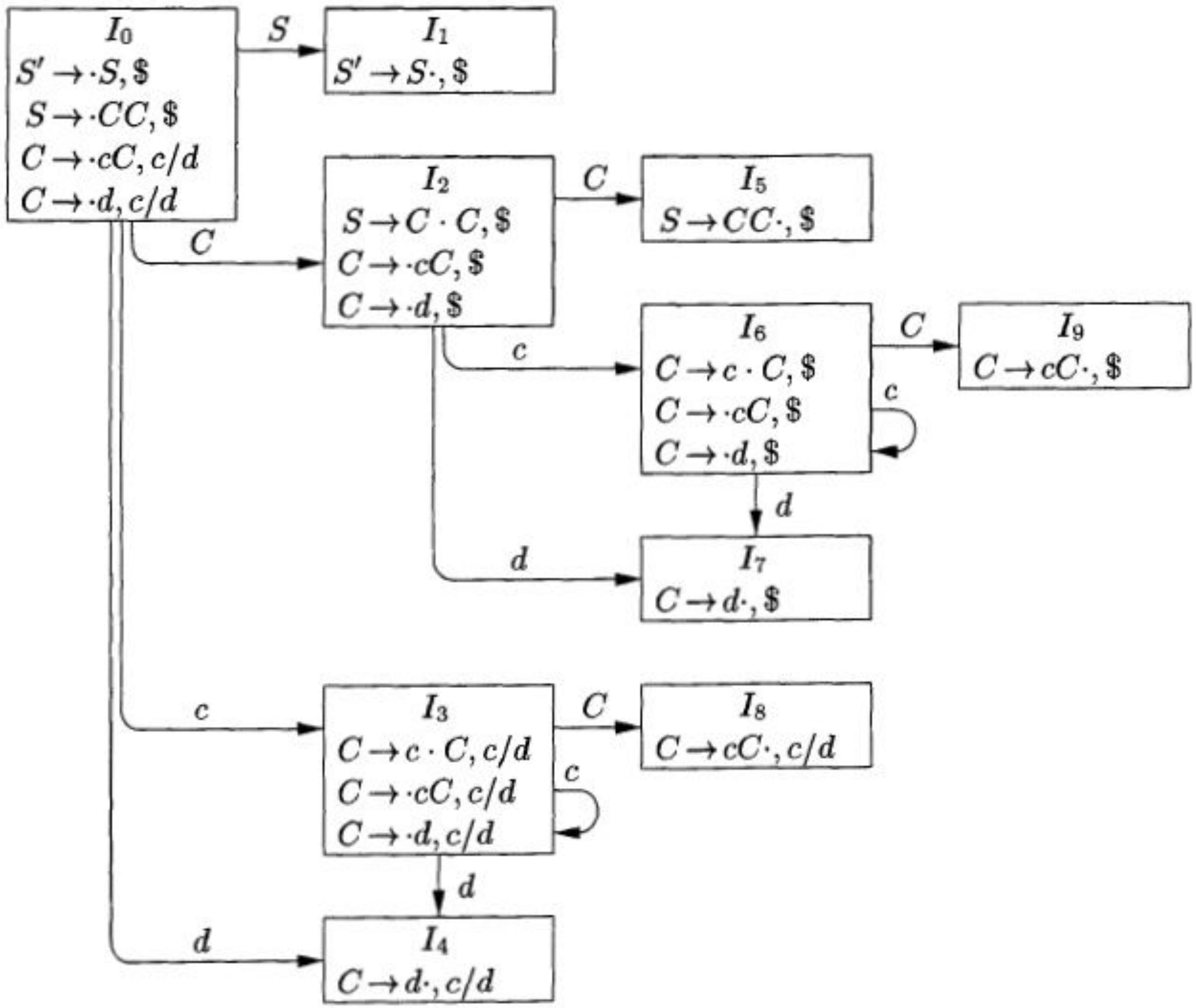


Figure source: Suggested course textbook

Another Example: LR(1) to LALR(1)

S' → S
S → CC
C →cC|d

STATE	ACTION			GOTO	
	c	*d*	$	*S*	*C*
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Figure source: Suggested course textbook

Another Example: LR(1) to LALR(1)

S' → S

S → CC

C → cC|d

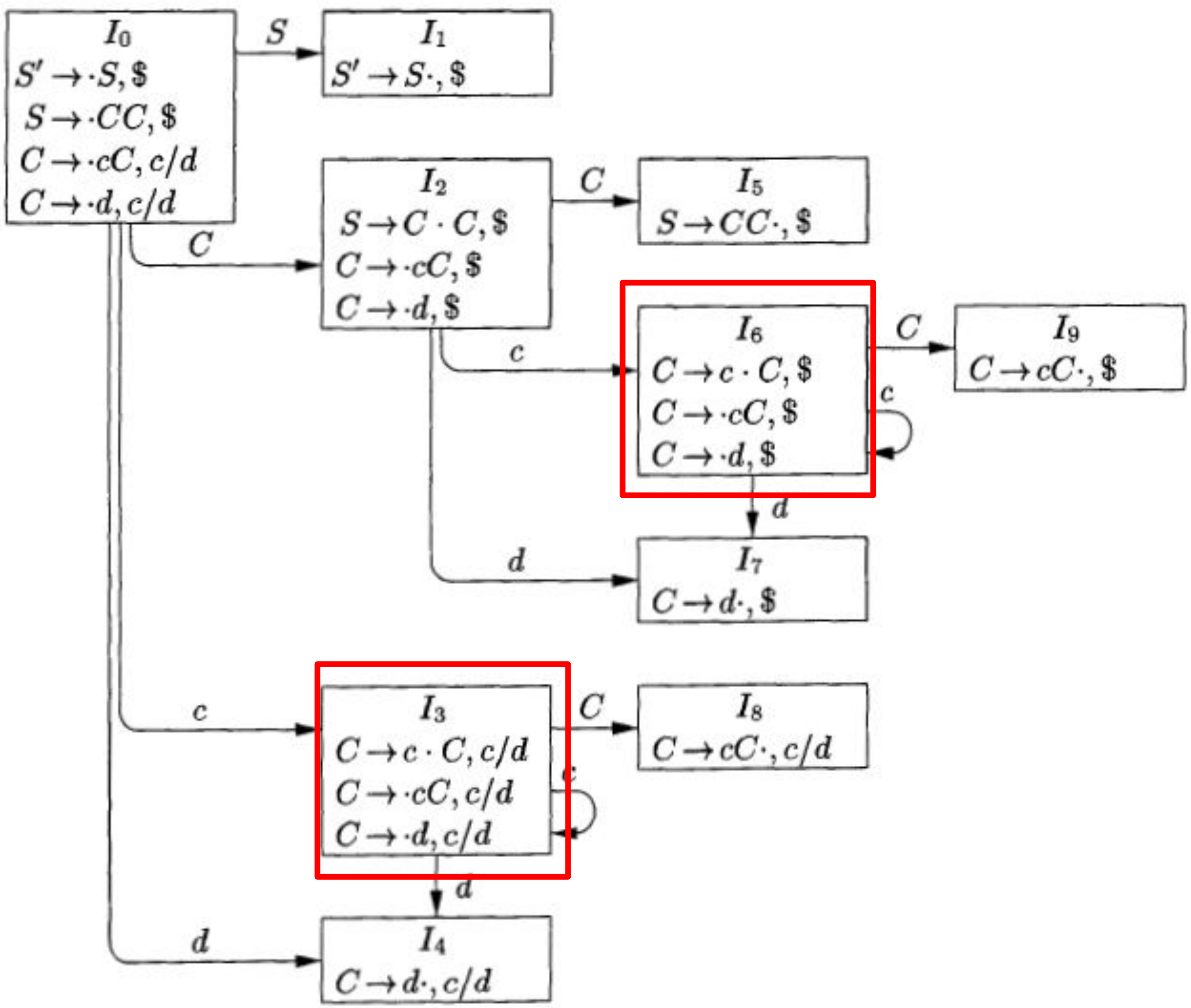


Figure source: Suggested course textbook

Another Example: LR(1) to LALR(1)

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow cC|d$

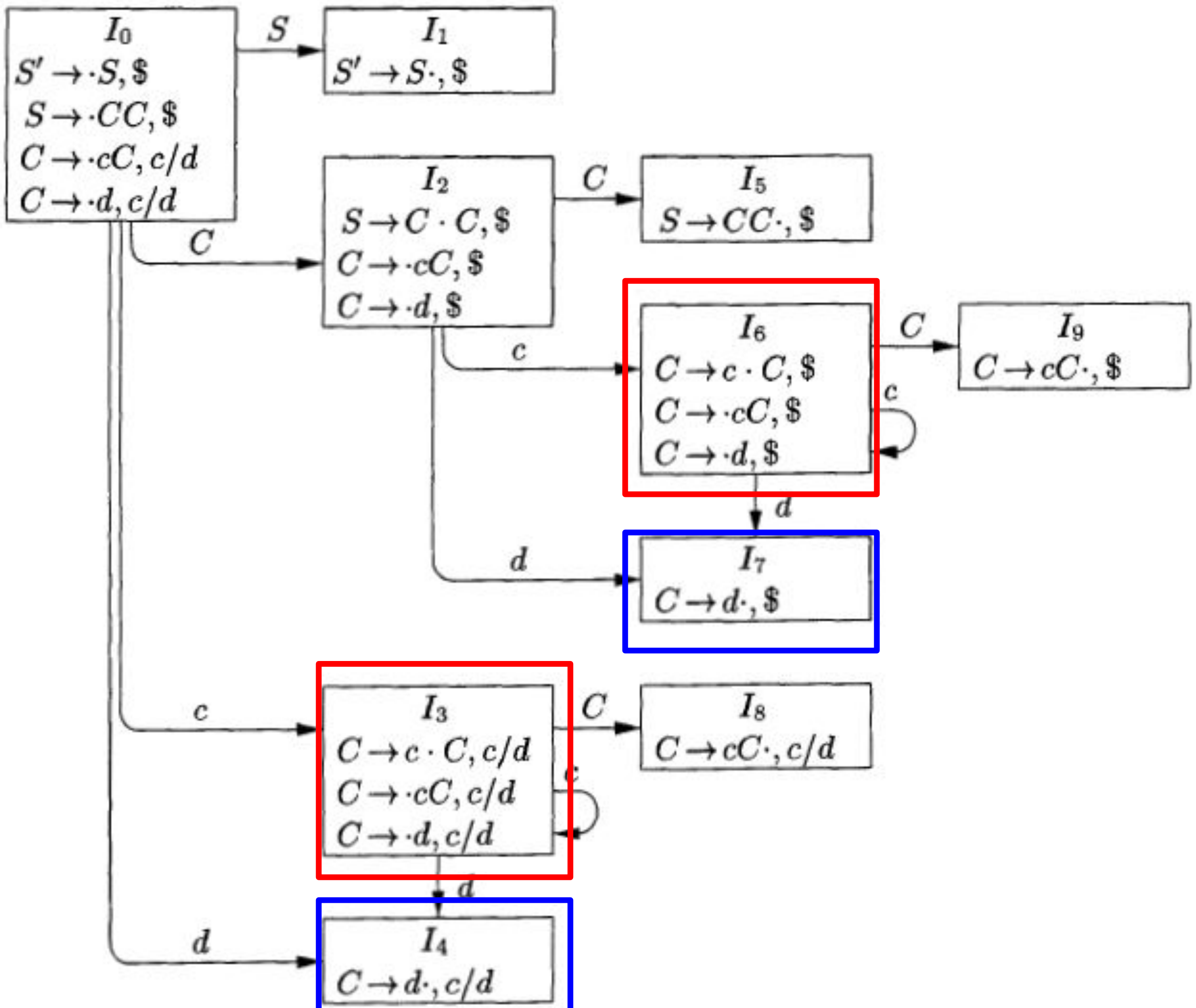


Figure source: Suggested course textbook

Another Example: LR(1) to LALR(1)

S' → S
S → CC
C → cC|d

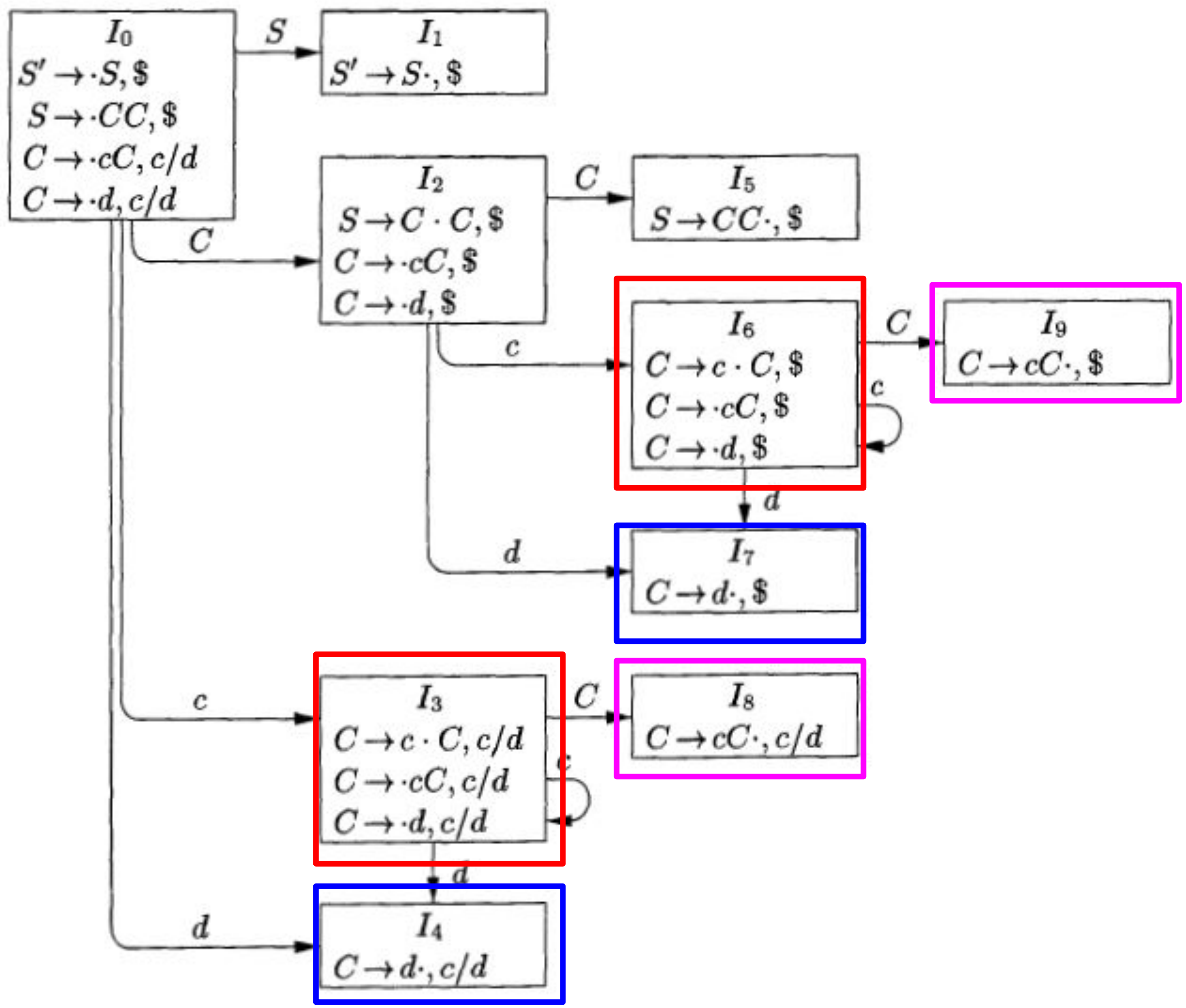


Figure source: Suggested course textbook

Another Example: LR(1) to LALR(1)

S’ → S
S → CC
C →cC|d

STATE	ACTION			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		

Figure source: Suggested course textbook

Advantages of LALR(1)

- Maintains context.
- Keeps automaton small.
  - Resulting automaton has same size as LR(0) automaton.

LALR(1) is Powerful

- Every LR(0) grammar is LALR(1).
- Every SLR(1) grammar is LALR(1)
- *Most* (but not all) LR(1) grammars are LALR(1).

LALR(1) isn't LR(1)

- Merging LR(1) states **can** introduce a reduce/reduce conflict.
- Often these conflicts appear without any good reason; this is one limitation of LALR(1).
- Merging LR(1) states cannot introduce a shift/reduce conflict.
- **Why?** Exercise

Summary of LALR(1)

- One of the most popular parsing algorithms in use today.
- Produced by the **`bison`** parser generator; rarely generated by hand.
- Can handle most, but not all, LR(1) languages.

Practical Concerns

Where Theory Meets Practice

- We've just covered six powerful parsing algorithms:
  - Leftmost DFS
  - LL(1)
  - LR(0)
  - SLR(1)
  - LALR(1)
  - LR(1)
- How do we make them work in practice?

Two Practical Concerns

- **Ambiguity**
  - Real grammars are often ambiguous.
  - Programmers are *terrible* at eliminating it.
  - How do you build a parser to try to combat it?
- **Error-handling**
  - How do you report errors intelligently?
  - How do you continue parsing after an error?

Ambiguity and Predictive Parsing

- The predictive parsers we have seen so far (LL(1), LR(0), SLR(1), LALR(1), LR(1)) only work on unambiguous grammars.
  - Intuitively: if grammar is ambiguous, cannot uniquely guess which production/reduction to use.
  - Formally proving this is somewhat involved.
- Most grammars for programming languages, unless cleverly written, are ambiguous.
- How can we handle this?

Parsing Ambiguous Grammars

- Consider this simple grammar for arithmetic expressions:

$$\mathbf{S} \rightarrow \mathbf{E}$$

$$\mathbf{E} \rightarrow \mathbf{E} \; \texttt{+} \; \mathbf{E}$$

$$\mathbf{E} \rightarrow \mathbf{E} \; \texttt{*} \; \mathbf{E}$$

$$\mathbf{E} \rightarrow \texttt{int}$$

$$\mathbf{E} \rightarrow \texttt{(}\mathbf{E}\texttt{)}$$

- This grammar is ambiguous.
  - e.g. Two trees for `int + int * int`
- What happens if we try parsing it?