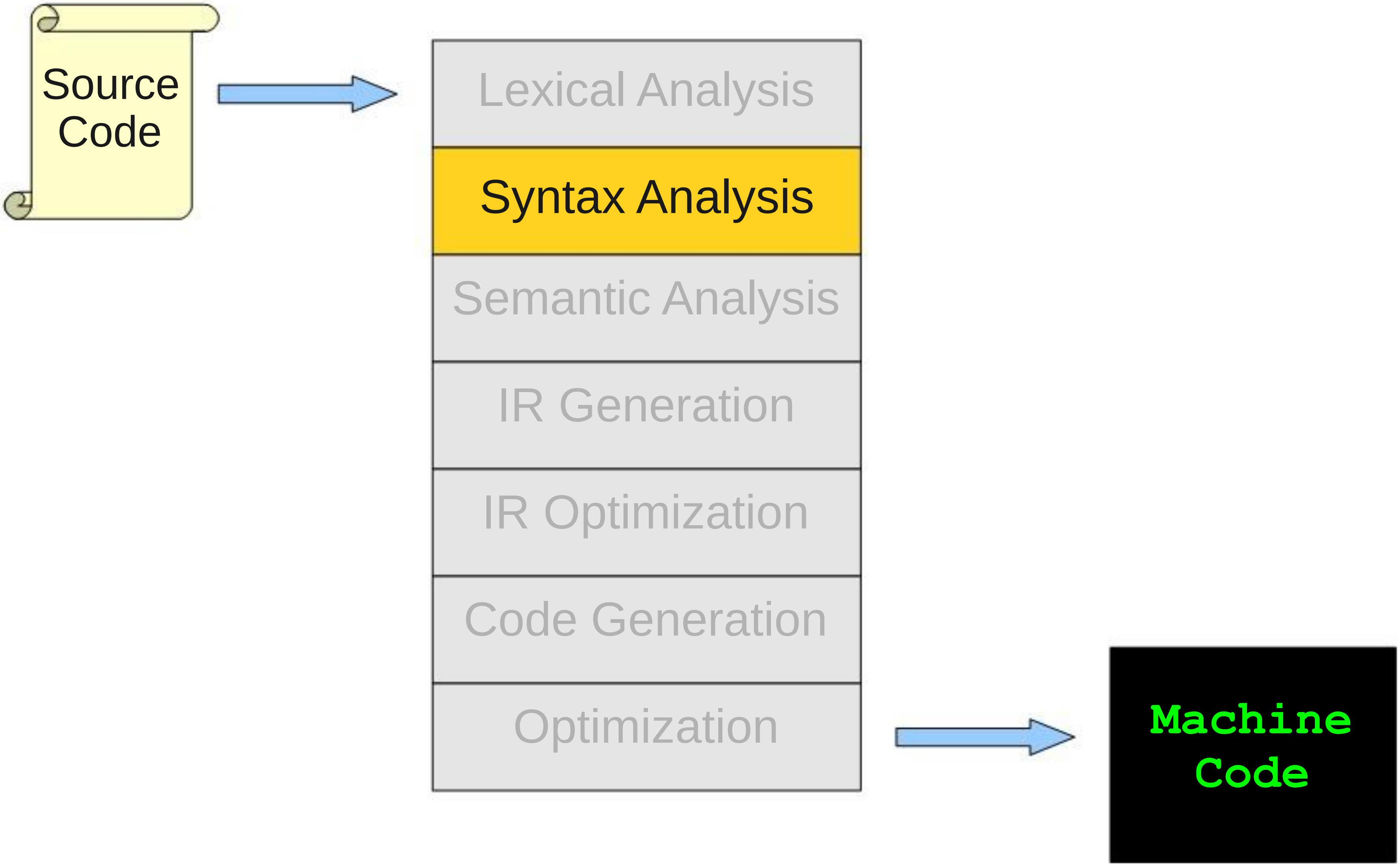
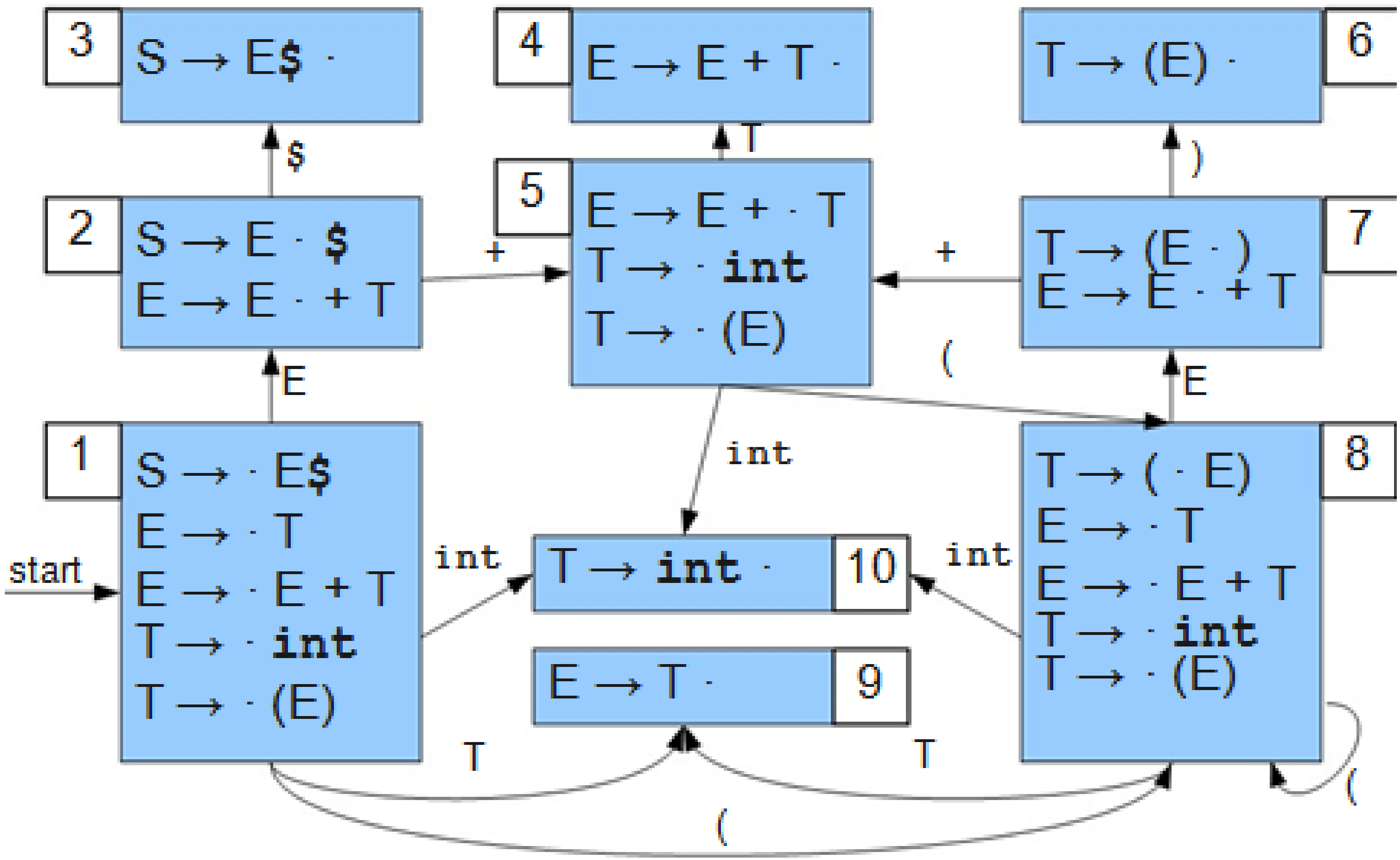
Bottom-Up Parsing II

Where We Are



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

Representing the Automaton as Table



LR(0) Tables

	`int`	`+`	`(`	`)`	`$`	S	E	T
1	10		8				2	9
2		5			3			
3								
4								
5	10		8					4
6								
7		5		6				
8	10		8				7	9
9								
10								

Goto Table

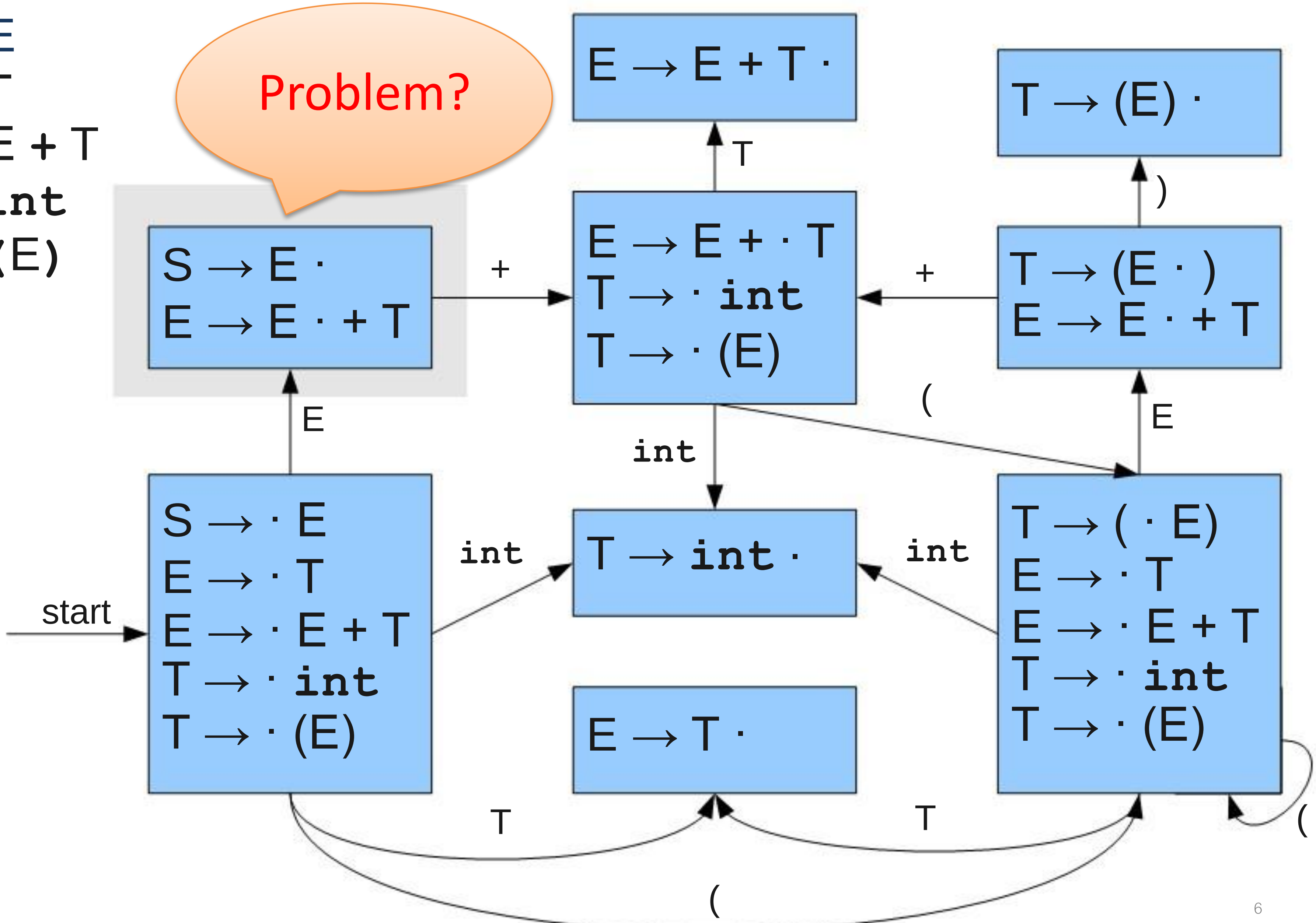
Action
Shift
Shift
Accept
Reduce E → E + T
Shift
Reduce T → (E)
Shift
Shift
Reduce E → T
Reduce T → `int`

Action Table

The Limits of LR(0)

A Non-LR(0) Grammar

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



LR Conflicts

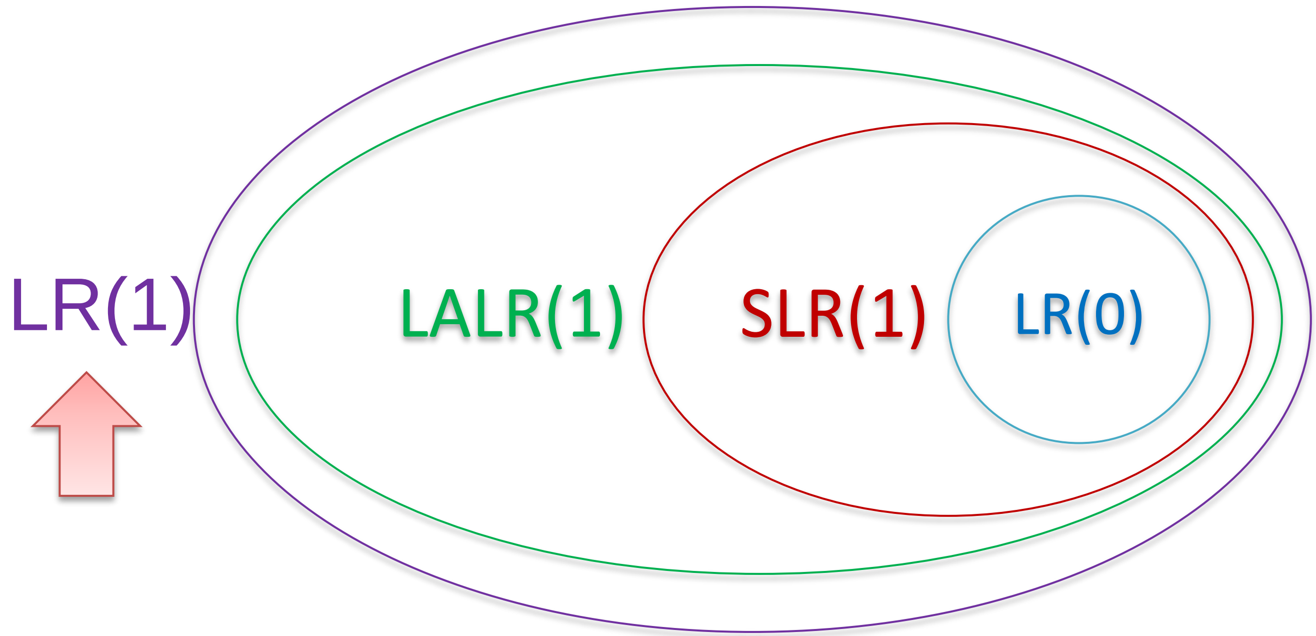
- A **shift/reduce conflict** is an error where a shift/reduce parser cannot tell whether to shift a token or perform a reduction.

- A **reduce/reduce conflict** is an error where a shift/reduce parser cannot tell which of many reductions to perform.

- A grammar whose handle-finding automaton contains a **shift/reduce** conflict or a **reduce/reduce** conflict is not LR(0).

- Having such conflicts indicates the possibility that the grammar is ambiguous.

Hierarchy of Shift/Reduce Parser



A Powerful Parser: LR(1)

- Bottom-up predictive parsing with
  - **L**: **L**eft-to-right scan
  - **R**: **R**ightmost derivation
  - (**1**): **One token lookahead**
- Substantially **more powerful** than the other methods we've covered so far.
- When deciding whether to shift or reduce, use look ahead to disambiguate.

Building Deterministic LR(1) Automata

Look ahead token

start

S → . E $

If the current token is E and the look ahead symbol is $, we will shift E on the top of stack.

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

Building Deterministic LR(1) Automata

Look ahead token

start

$S \rightarrow .\,E \quad \$$

$E \rightarrow .\,T \quad \$$

$E \rightarrow .\,E + T \quad \$$

in First of

Since E is a nonterminal symbol,
it can be further derived.

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

Building Deterministic LR(1) Automata

Look ahead token

start

$S \rightarrow .\,E \quad \$$

$E \rightarrow .\,T \quad \$$

$E \rightarrow .\,E + T \quad \$$

$E \rightarrow .\,T \quad +$

$E \rightarrow .\,E + T \quad +$

in First of

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

Building Deterministic LR(1) Automata

Look ahead token

start

S → . E	$
E → . T	$
E → . E + T	$
E → . T	+
E → . E + T	+
T → . int	$
T → . (E)	$

in First of

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

Building Deterministic LR(1) Automata

Look ahead token

start

S → . E	$
E → . T	$
E → . E + T	$
E → . T	+
E → . E + T	+
T → . int	$
T → . (E)	$
T → . int	+
T → . (E)	+

in First of

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

Building Deterministic LR(1) Automata

Look ahead token

start

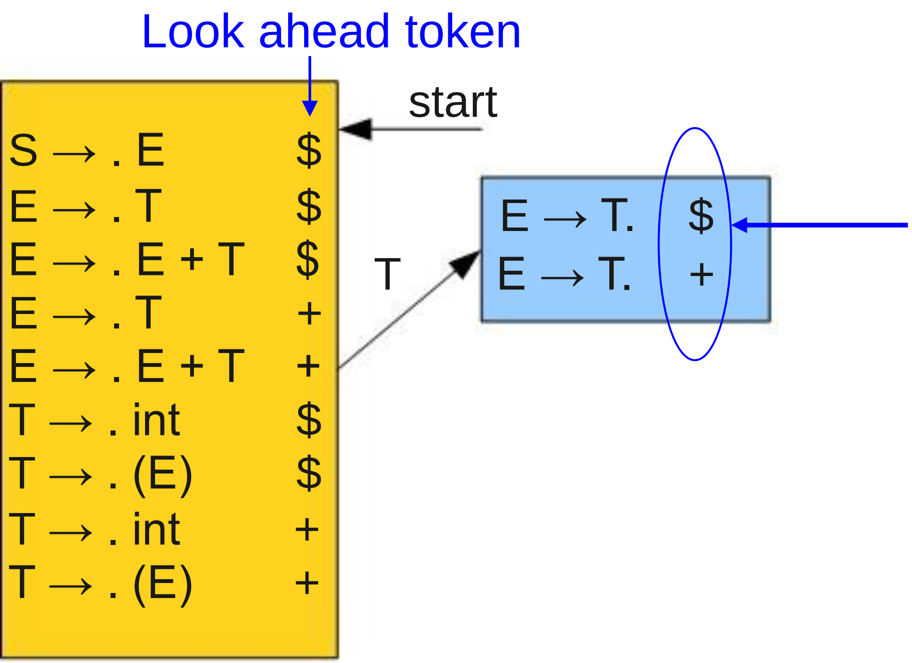
$S \rightarrow .\ E$	\$
$E \rightarrow .\ T$	\$
$E \rightarrow .\ E + T$	\$
$E \rightarrow .\ T$	+
$E \rightarrow .\ E + T$	+
$T \rightarrow .\ \text{int}$	\$
$T \rightarrow .\ (E)$	\$
$T \rightarrow .\ \text{int}$	+
$T \rightarrow .\ (E)$	+

No need to expand this production, since it will repeat the existing productions.

Dots are in front of terminal tokens, so we don't expand further.

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

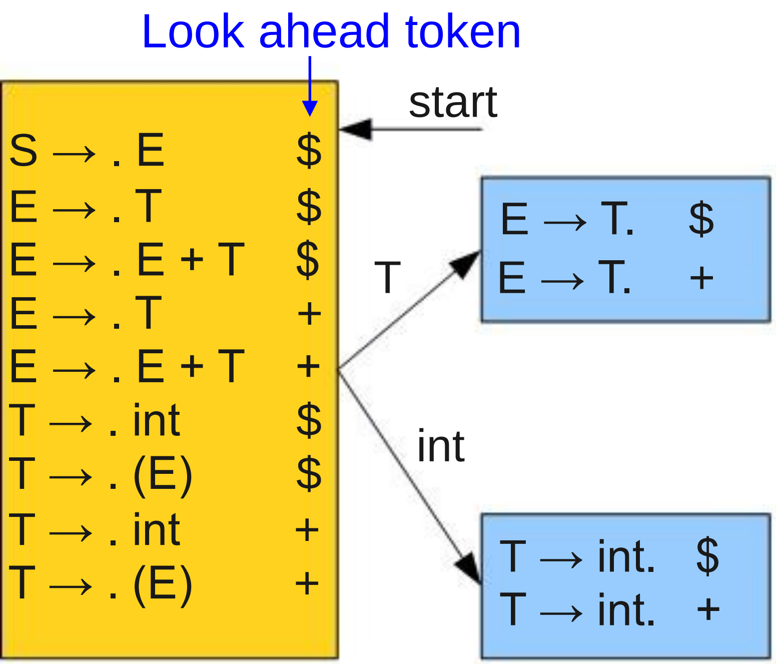
Building Deterministic LR(1) Automata



Look ahead symbols can make a distinction between two choices of reduction.

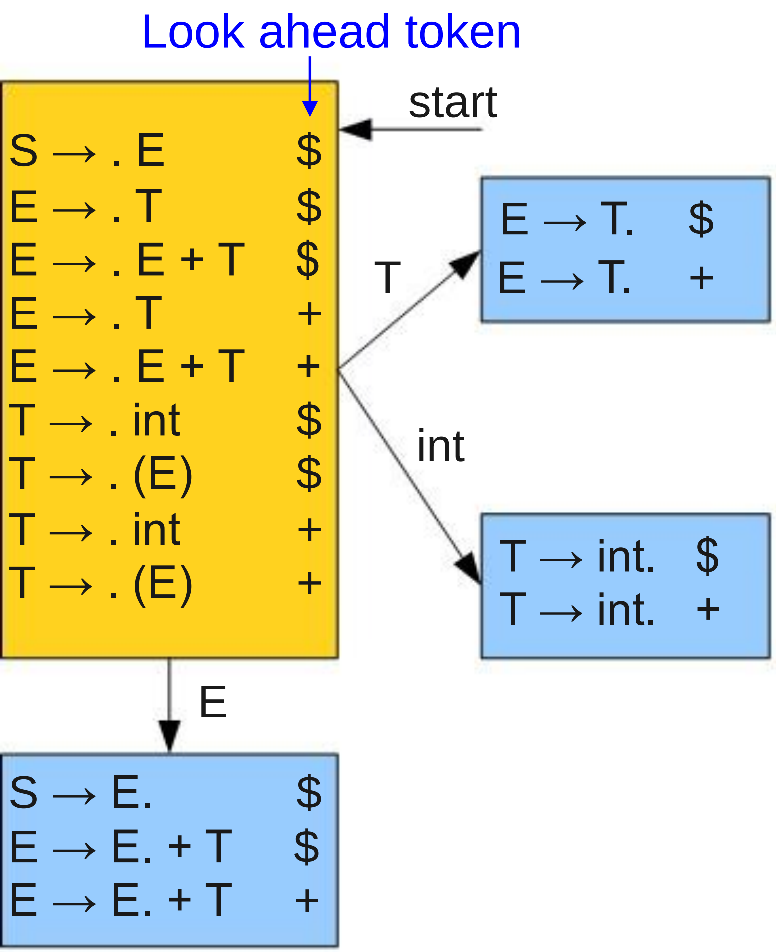
$S \rightarrow E$
$E \rightarrow T$
$E \rightarrow E$ **+** T
$T \rightarrow$ **int**
$T \rightarrow$ **(**E**)**

Building Deterministic LR(1) Automata



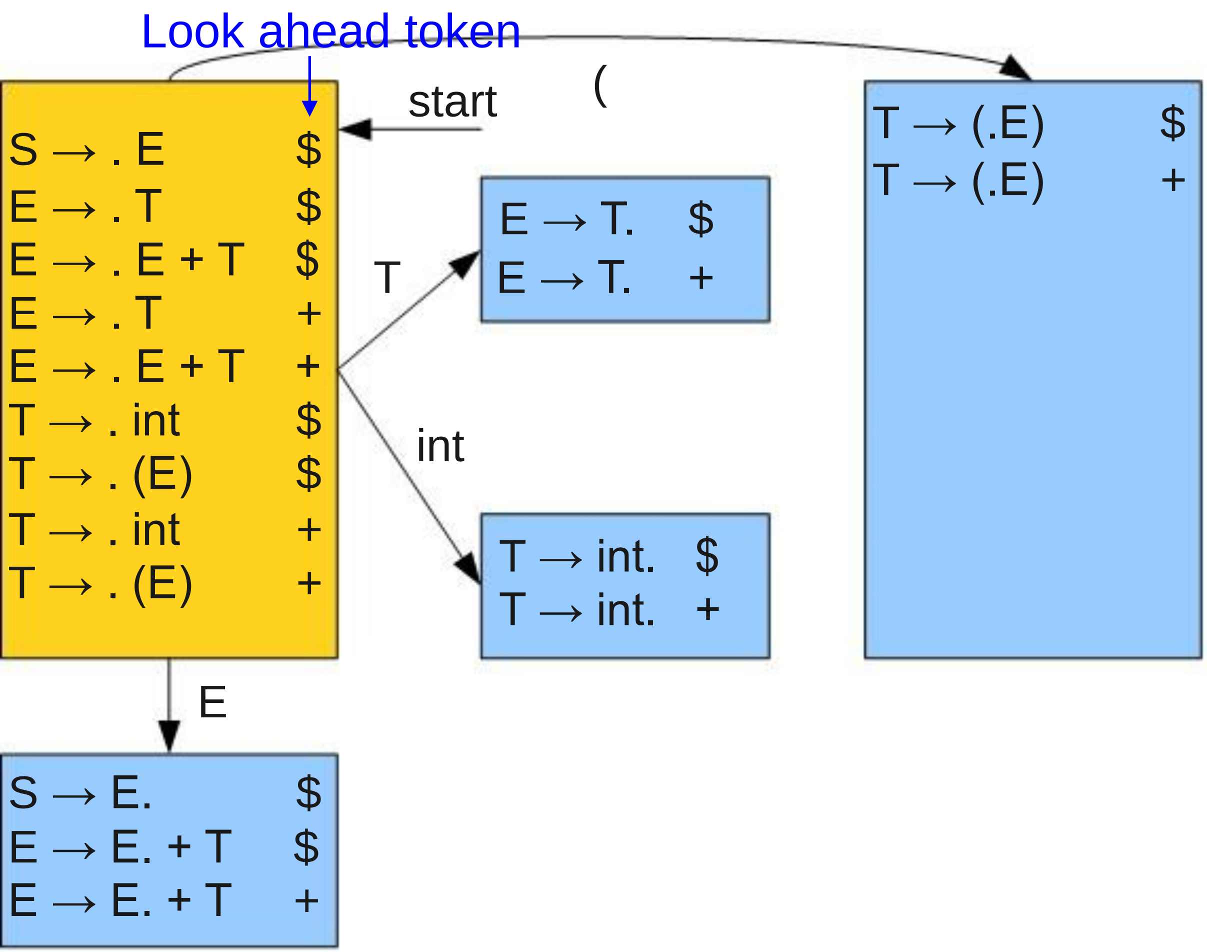
S → E
E → T
E → E **+** T
T → **int**
T → **(**E**)**

Building Deterministic LR(1) Automata



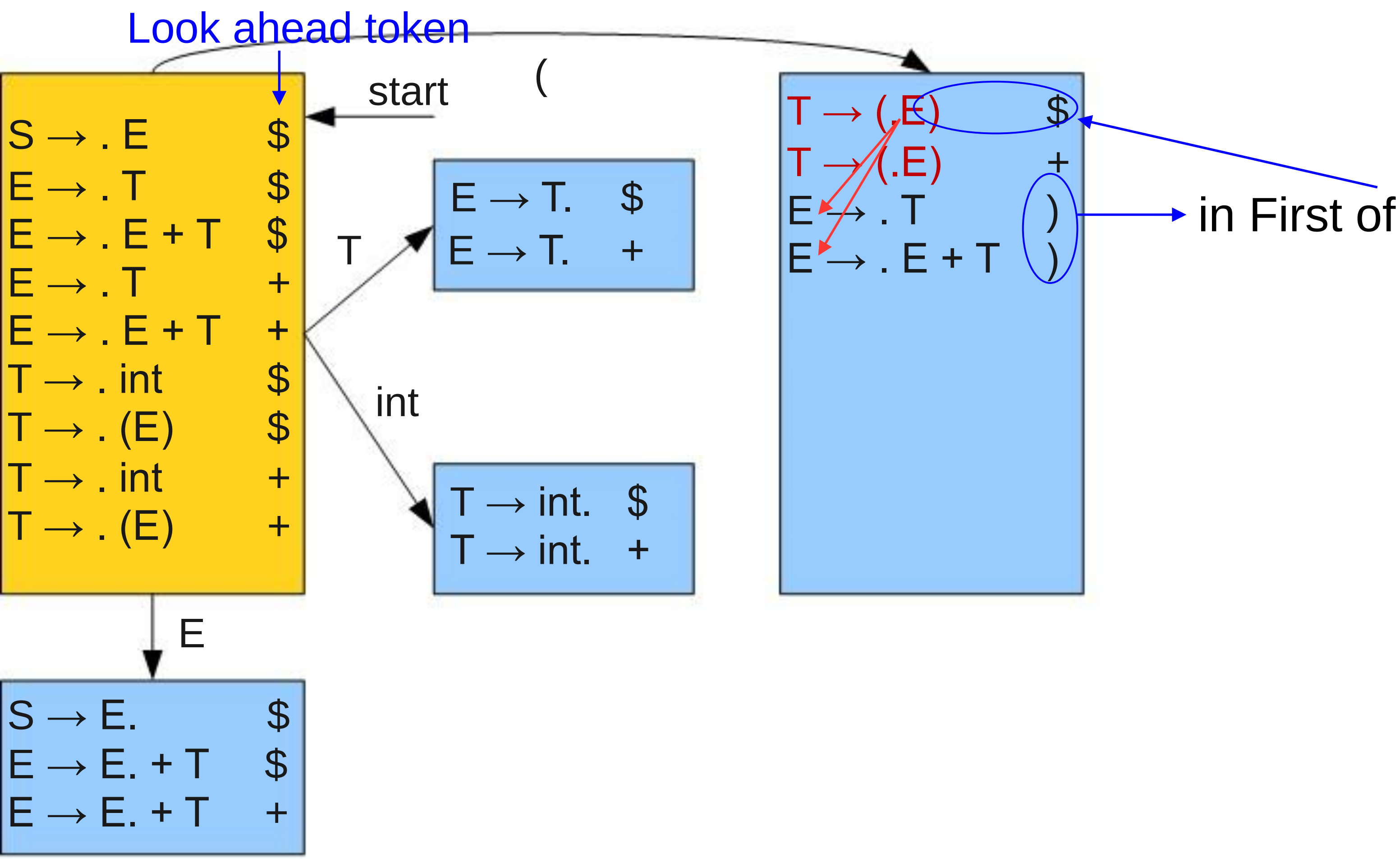
S → E
E → T
E → E **+** T
T → **int**
T → **(**E**)**

Building Deterministic LR(1) Automata



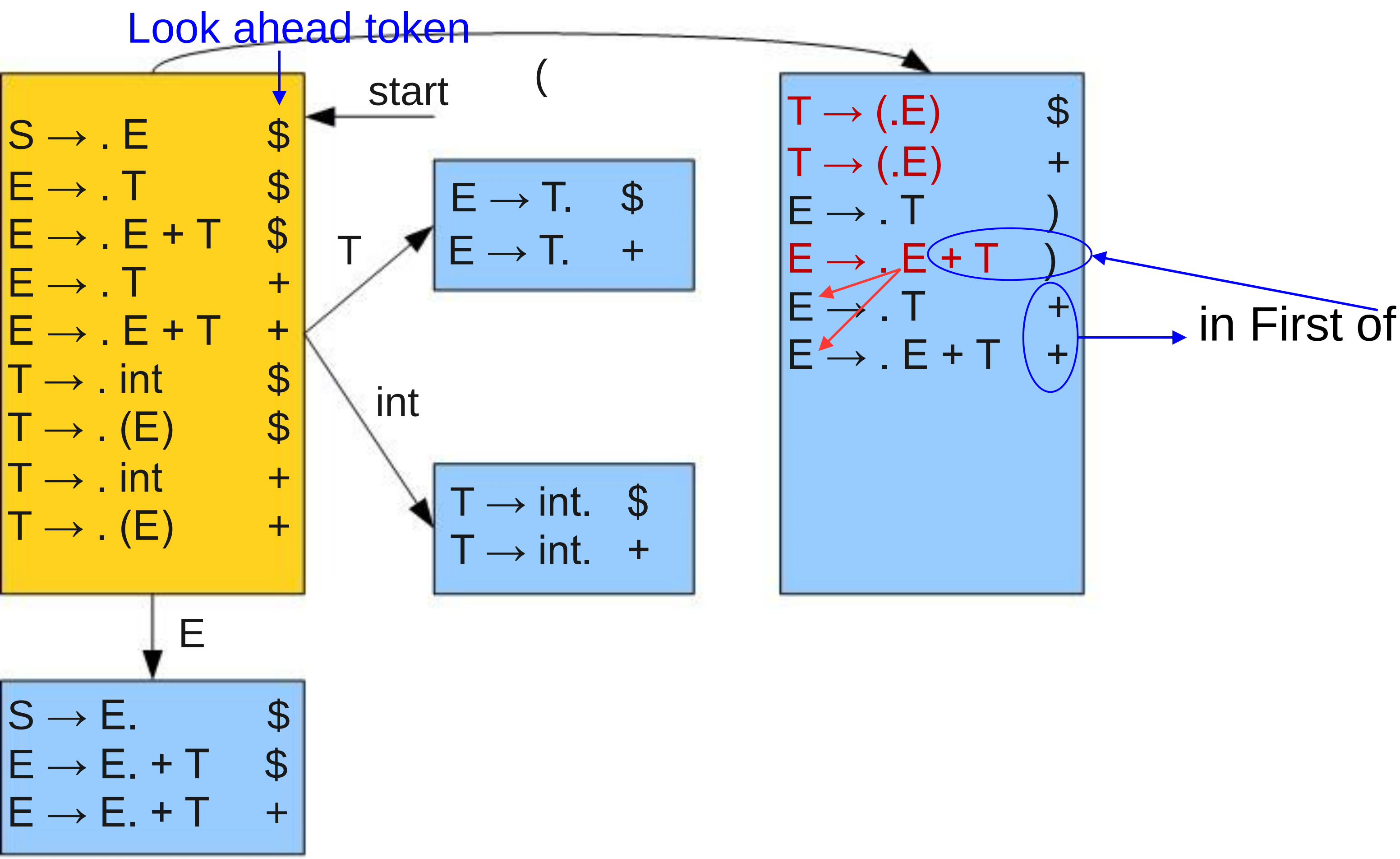
S → E
E → T
E → E **+** T
T → **int**
T → **(**E**)**

Building Deterministic LR(1) Automata



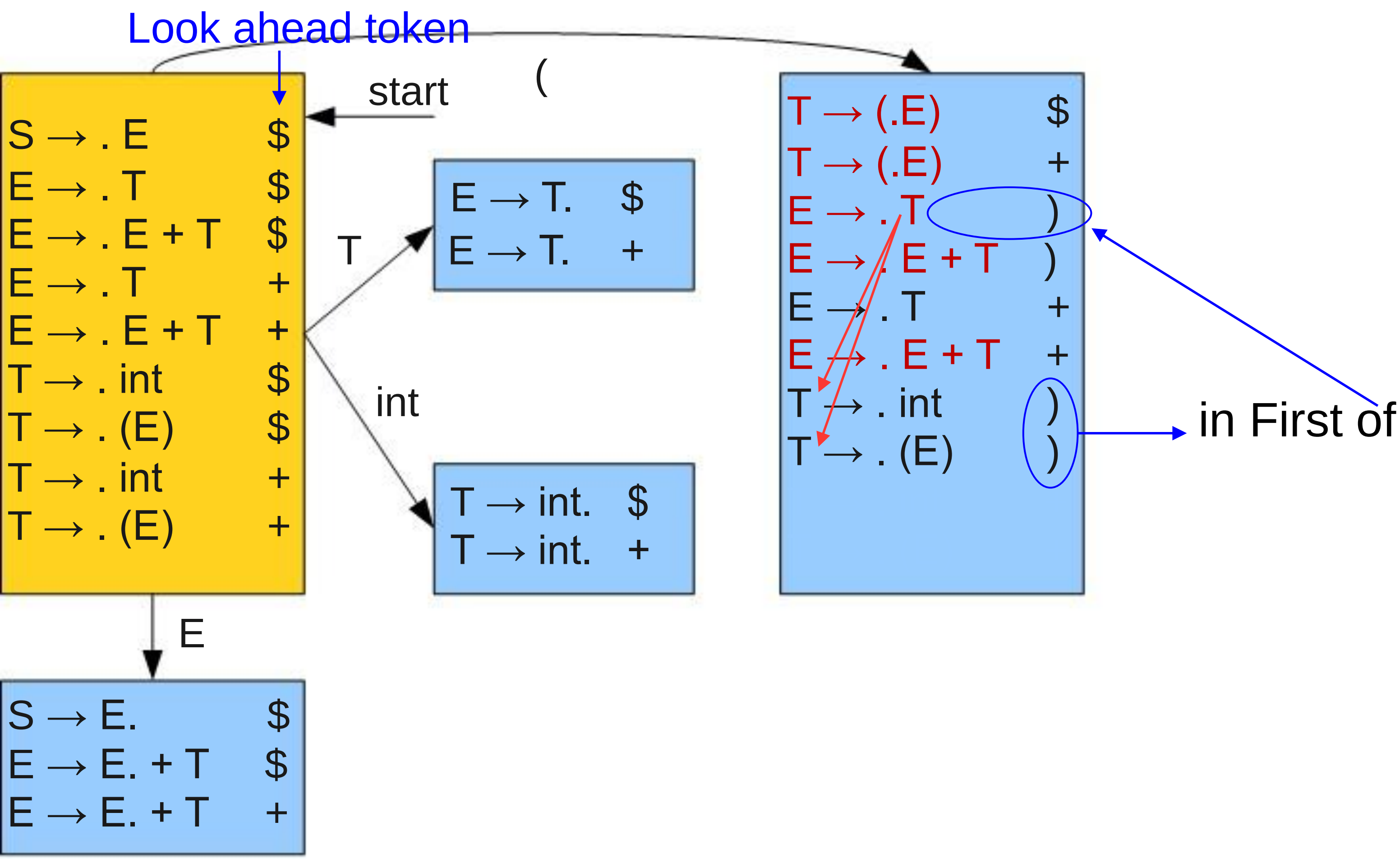
S → E
E → T
E → E **+** T
T → **int**
T → **(E)**

Building Deterministic LR(1) Automata



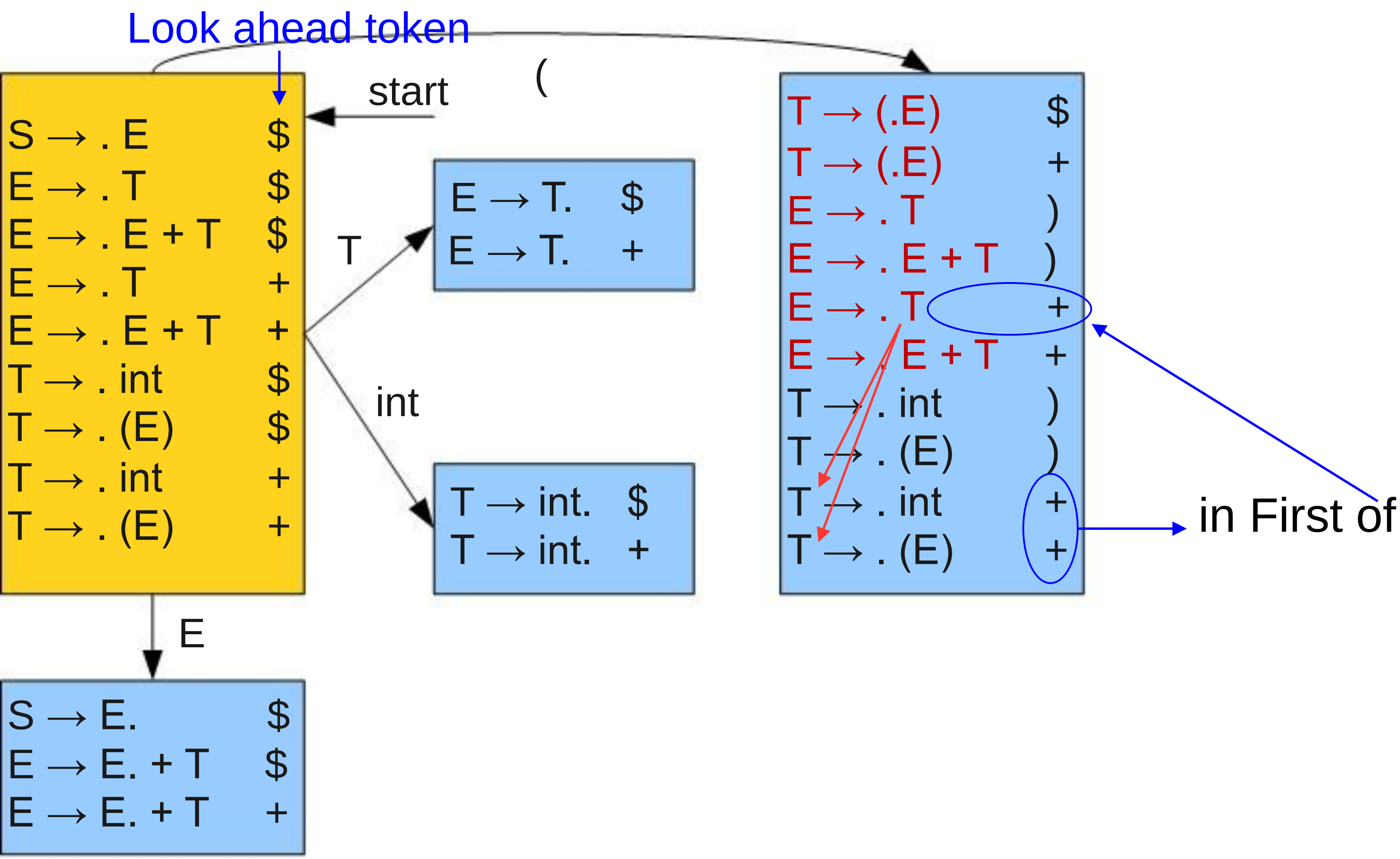
S → E
E → T
E → E **+** T
T → **int**
T → **(E)**

Building Deterministic LR(1) Automata



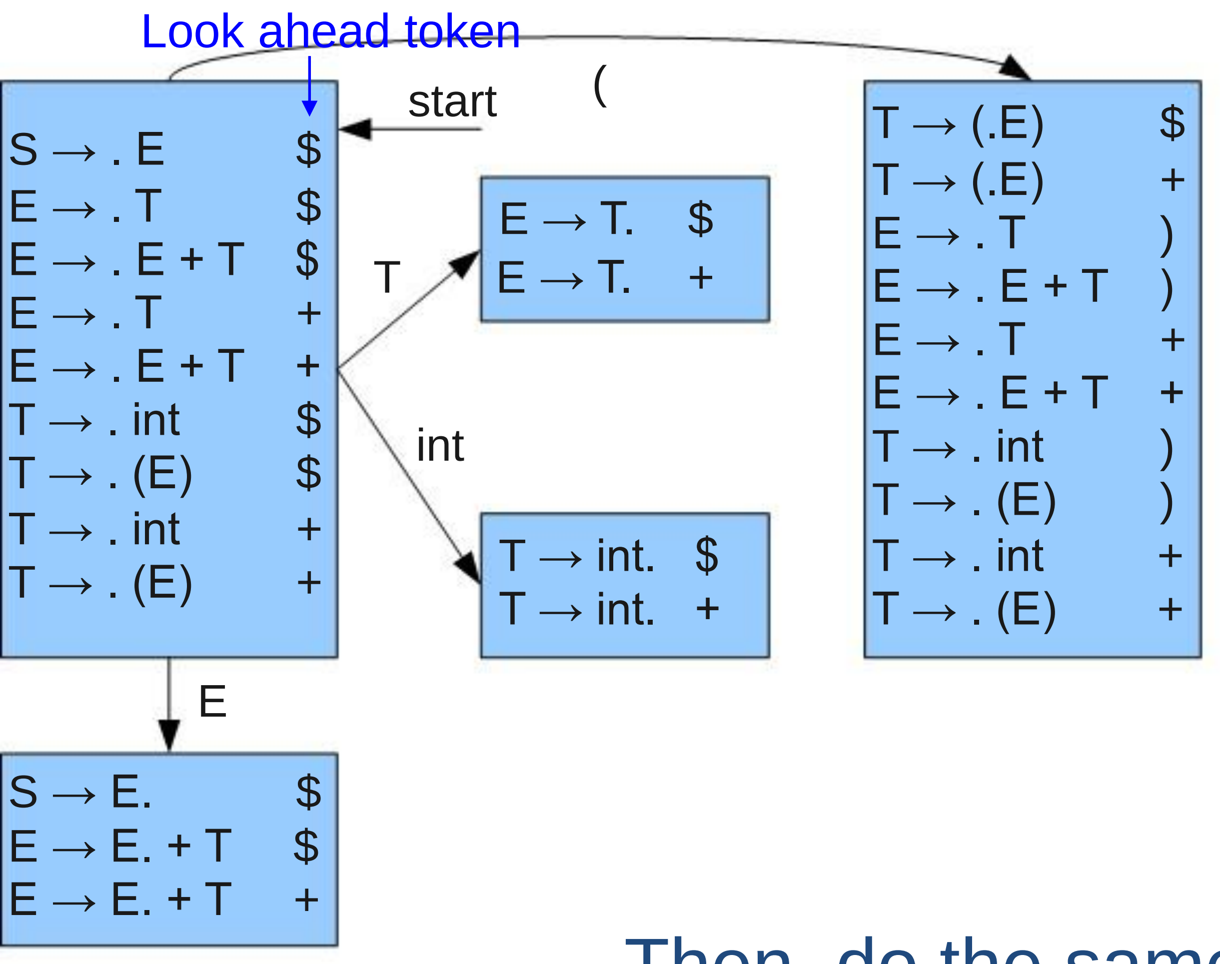
S → E
E → T
E → E **+** T
T → **int**
T → **(E)**

Building Deterministic LR(1) Automata



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

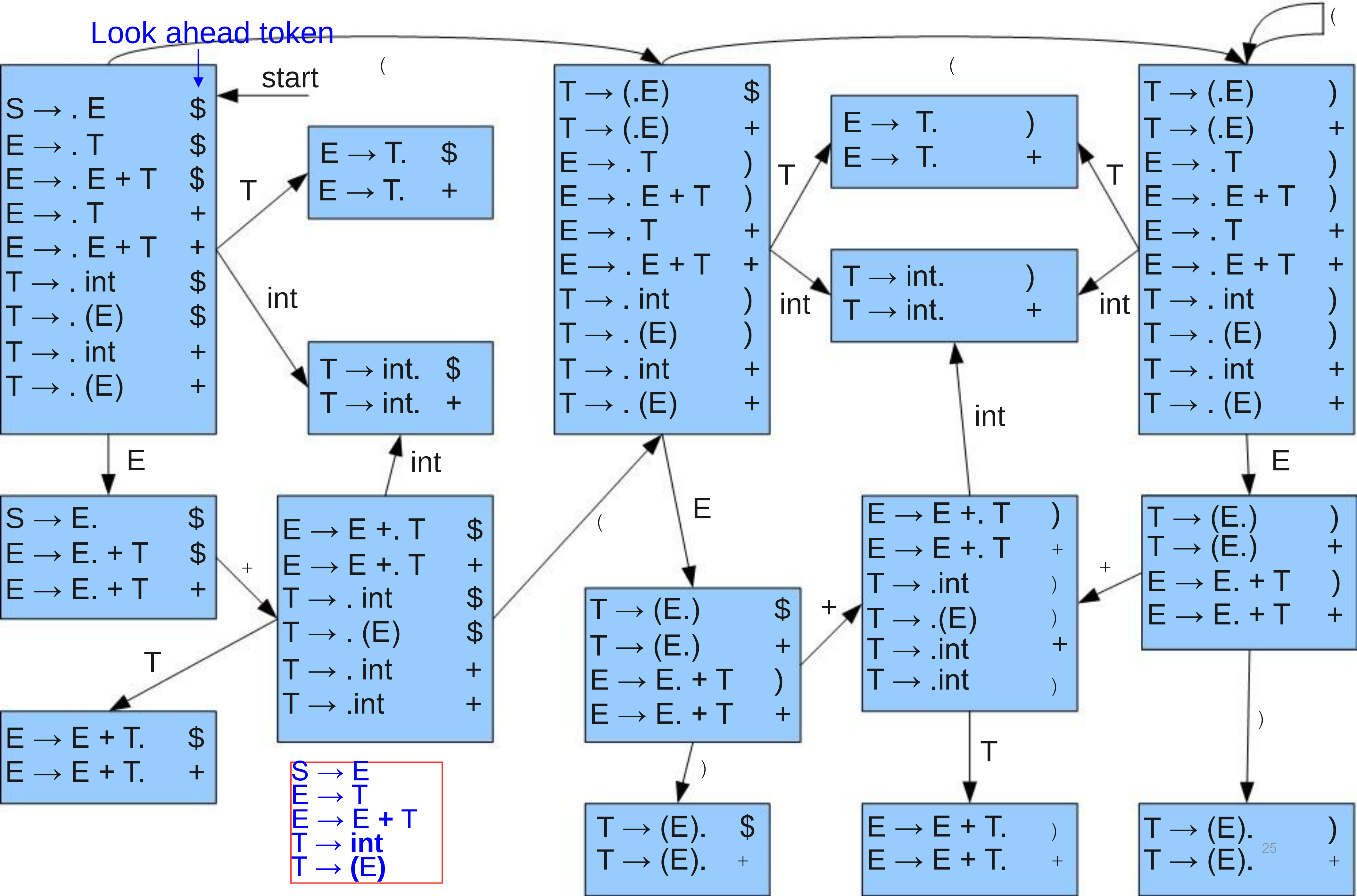
Building Deterministic LR(1) Automata



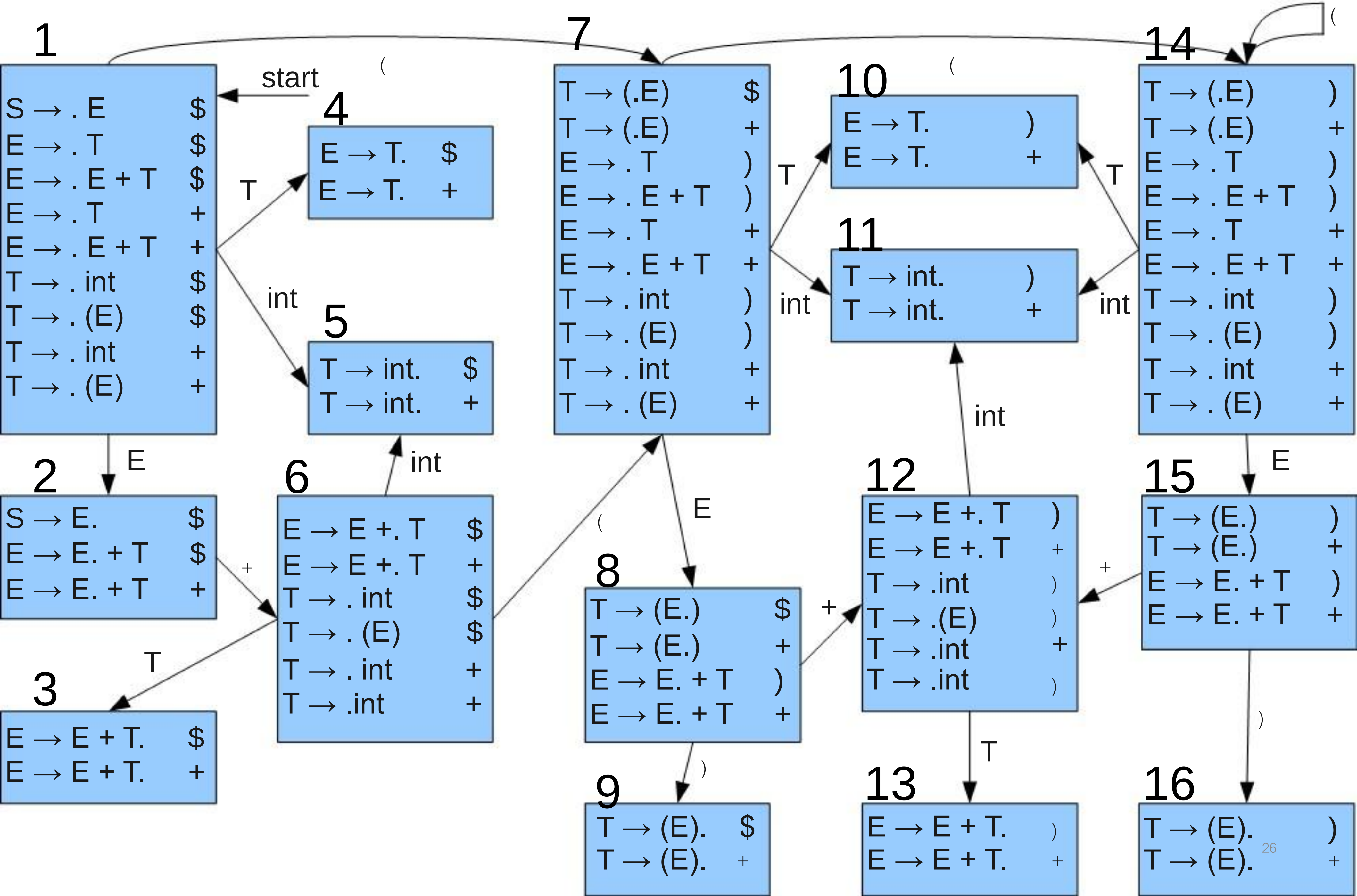
Then, do the same things with other states.

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

Building Deterministic LR(1) Automata



Representing LR(1) Automata as Table



$$S \rightarrow E \qquad (1)$$
$$E \rightarrow T \qquad (2)$$
$$E \rightarrow E + T \qquad (3)$$
$$T \rightarrow \texttt{int} \qquad (4)$$
$$T \rightarrow (E) \qquad (5)$$

	int	(	)	+	$	T	E
1	s5	s7				s4	s2
2				s6	ACCEPT		
3				r3	r3		
4				r2	r2		
5				r4	r4		
6	s5	s7				s3	
7	s10	s14				s10	s8
8			s9	s12			
9				r5	r5		
10			r2	r2			
11			r4	r4			
12	s11					s13	
13			r3	r3			
14	s11	s14				s10	s15
15			s16	s12			
16			r5	r5			

Notes:

s5: Shift to state 5

r5 : Reduce by using production #5

LR(1) Automata are Powerful

- LR(1) parsers are **extremely powerful**.
- Any LL(1) and LR(0) are the subsets of LR(1).
- Any **deterministic language** has an LR(1) grammar.

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.
- LR(1) tables for practical programming languages can have hundreds of thousands states !
- Consequently, LR(1) parsers are rarely used in practice.

!??!

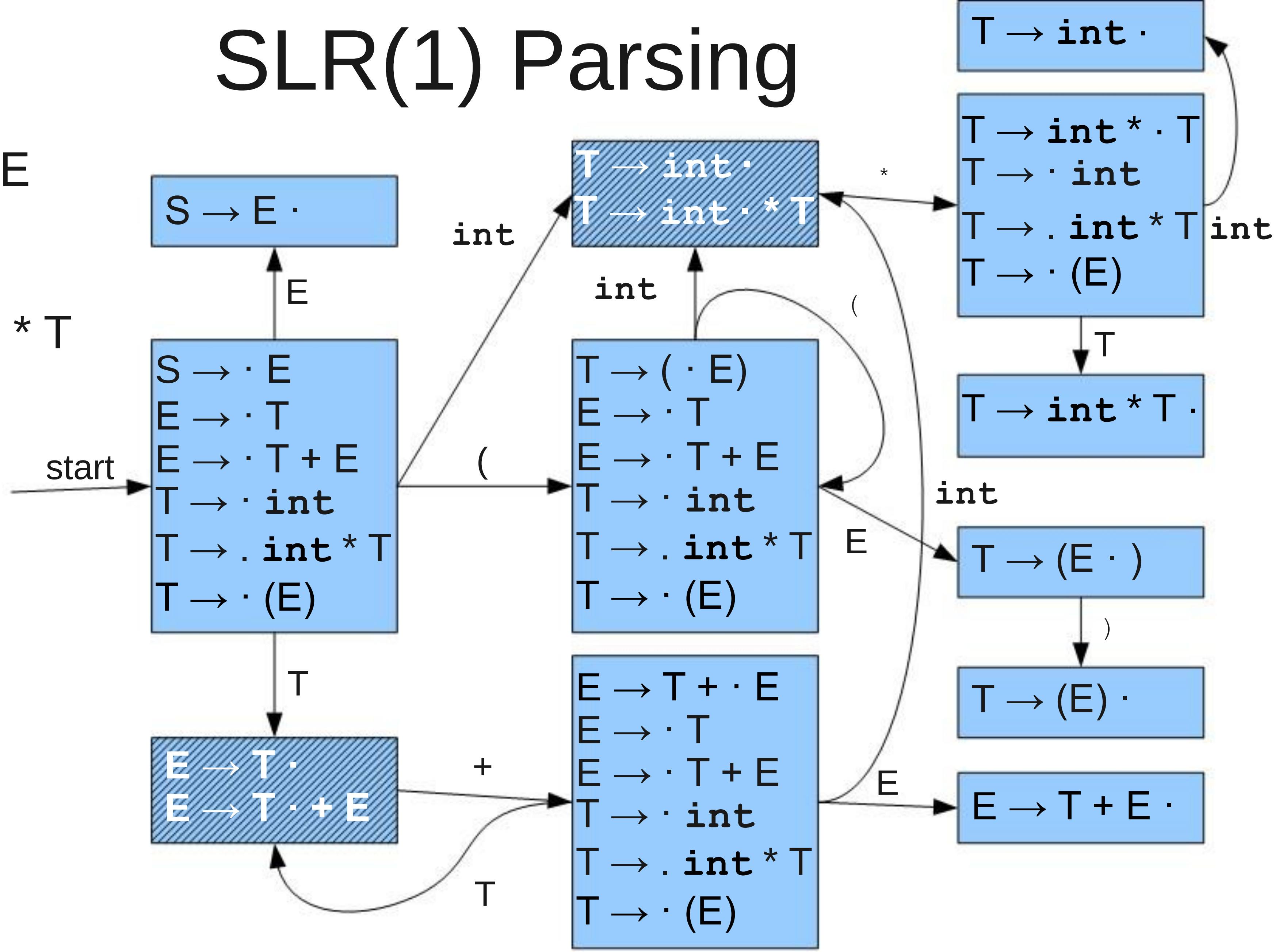
SLR(1)

- **Simple(1) LR**
- A compromising solution.
- Minor modification to LR(0) automaton that uses lookahead to avoid shift/reduce conflicts.
- Idea: Only reduce $A \rightarrow v$ if the next token t is in FOLLOW(A).
- Automaton identical to LR(0) automaton; only change is when we choose to reduce.

SLR(1) Parsing

S → E
E → T
E → T + E
T → `int`
T → (E)

T → `int` * T

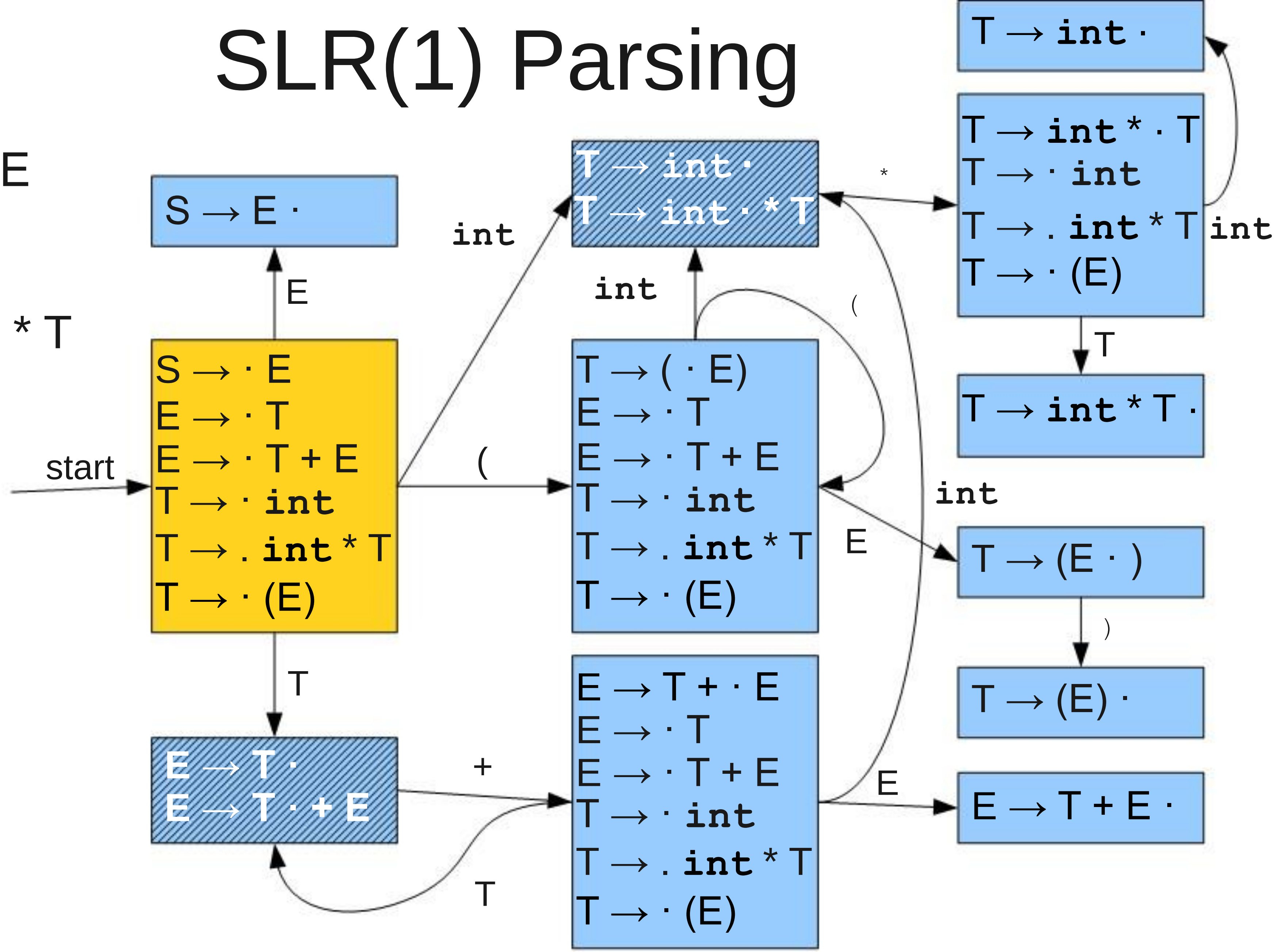


Shadowed states may encounter a **shift**/**reduce** conflict.

SLR(1) Parsing

$S \to E$
$E \to T$
$E \to T + E$
$T \to \texttt{int}$
$T \to (E)$

$T \to \texttt{int} * T$

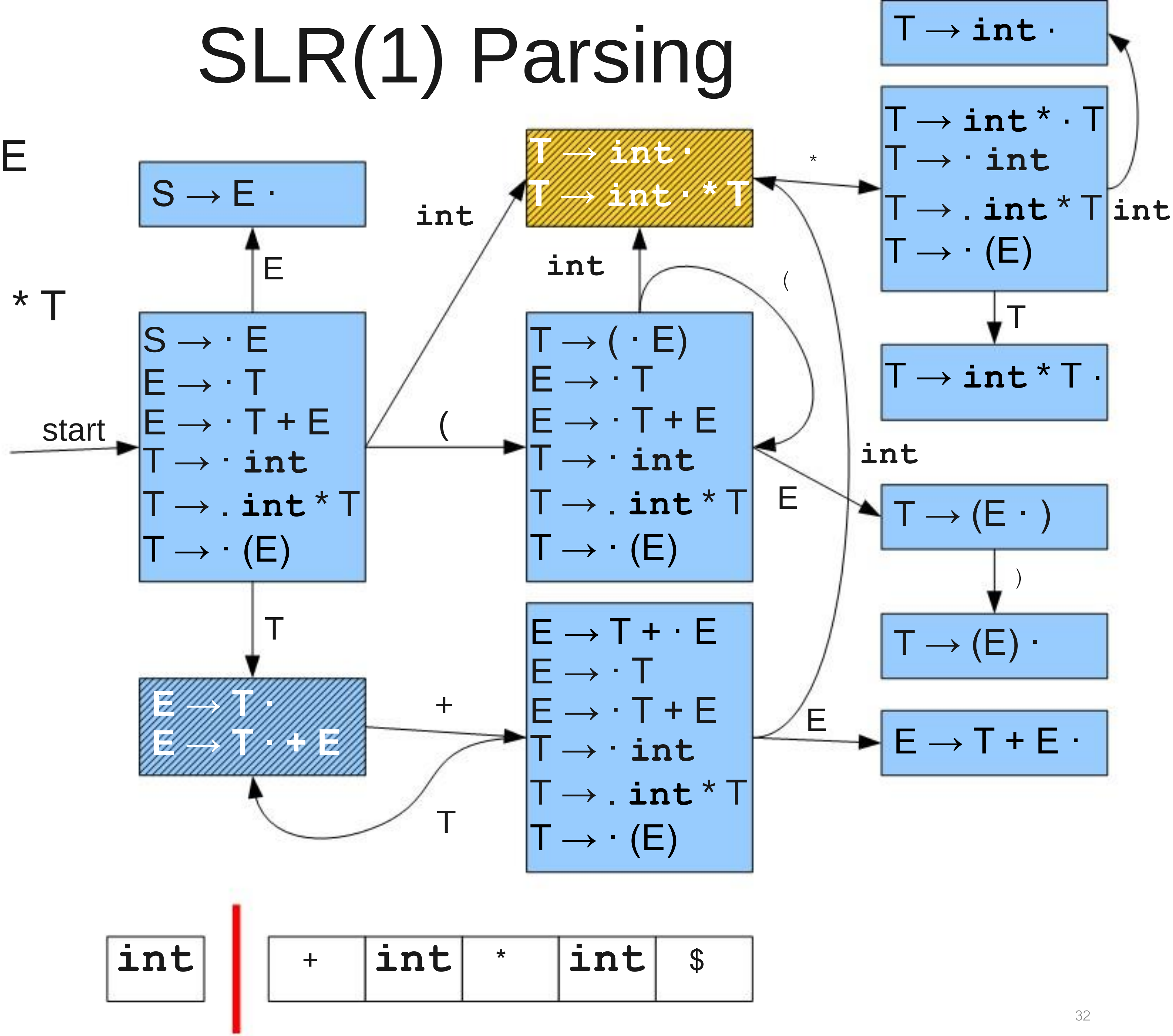


int	+	int	*	int	$

SLR(1) Parsing

$S \rightarrow E$
$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

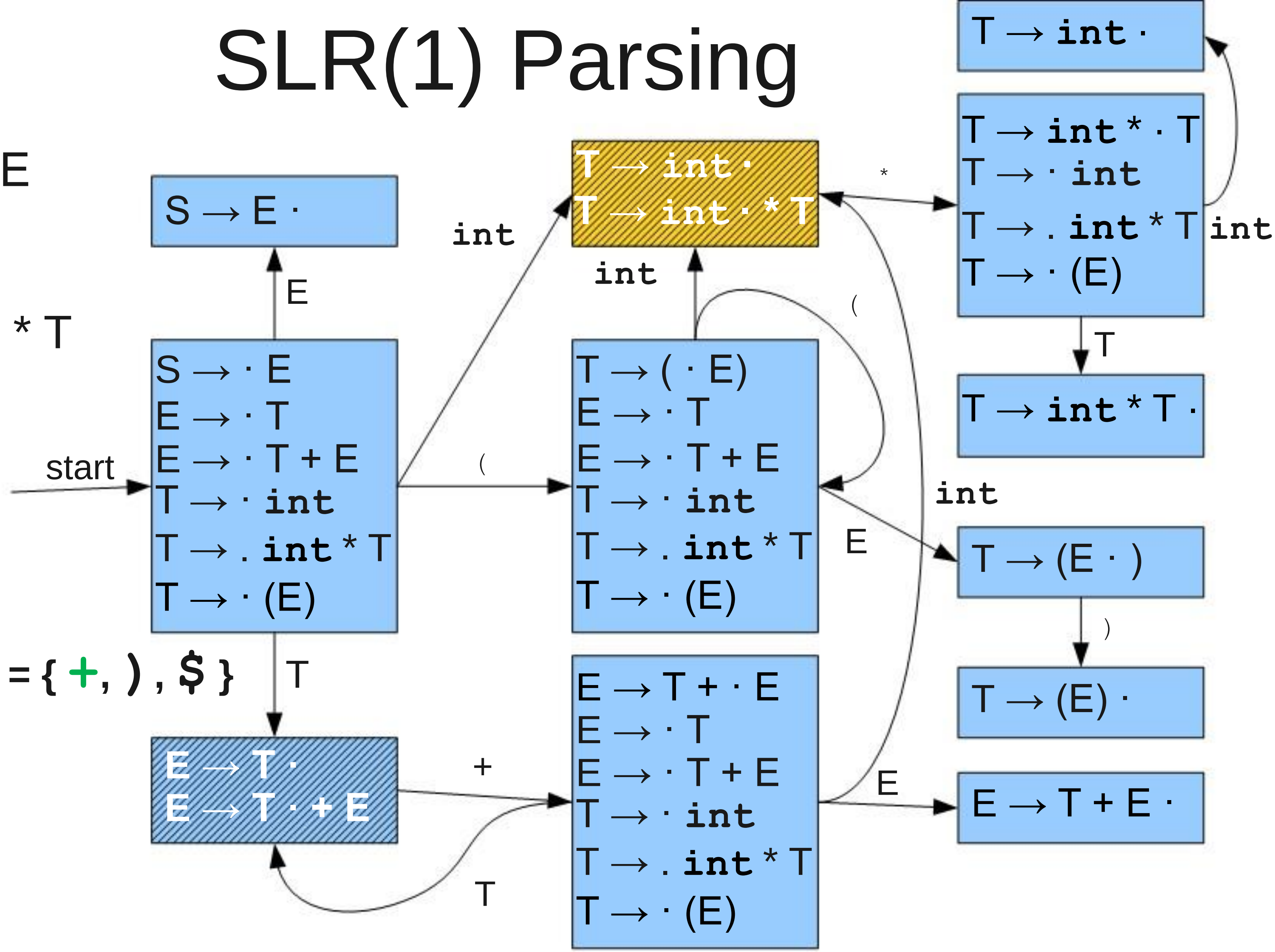
$T \rightarrow \texttt{int} * T$



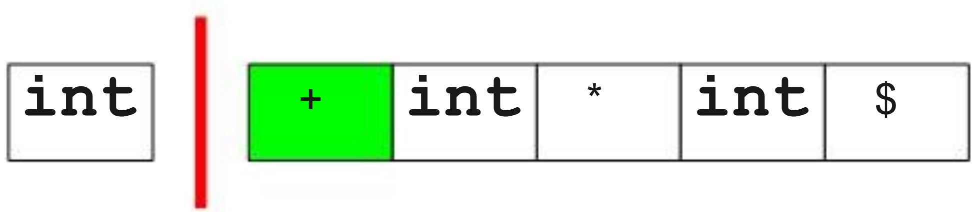
SLR(1) Parsing

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

T → **int** * T



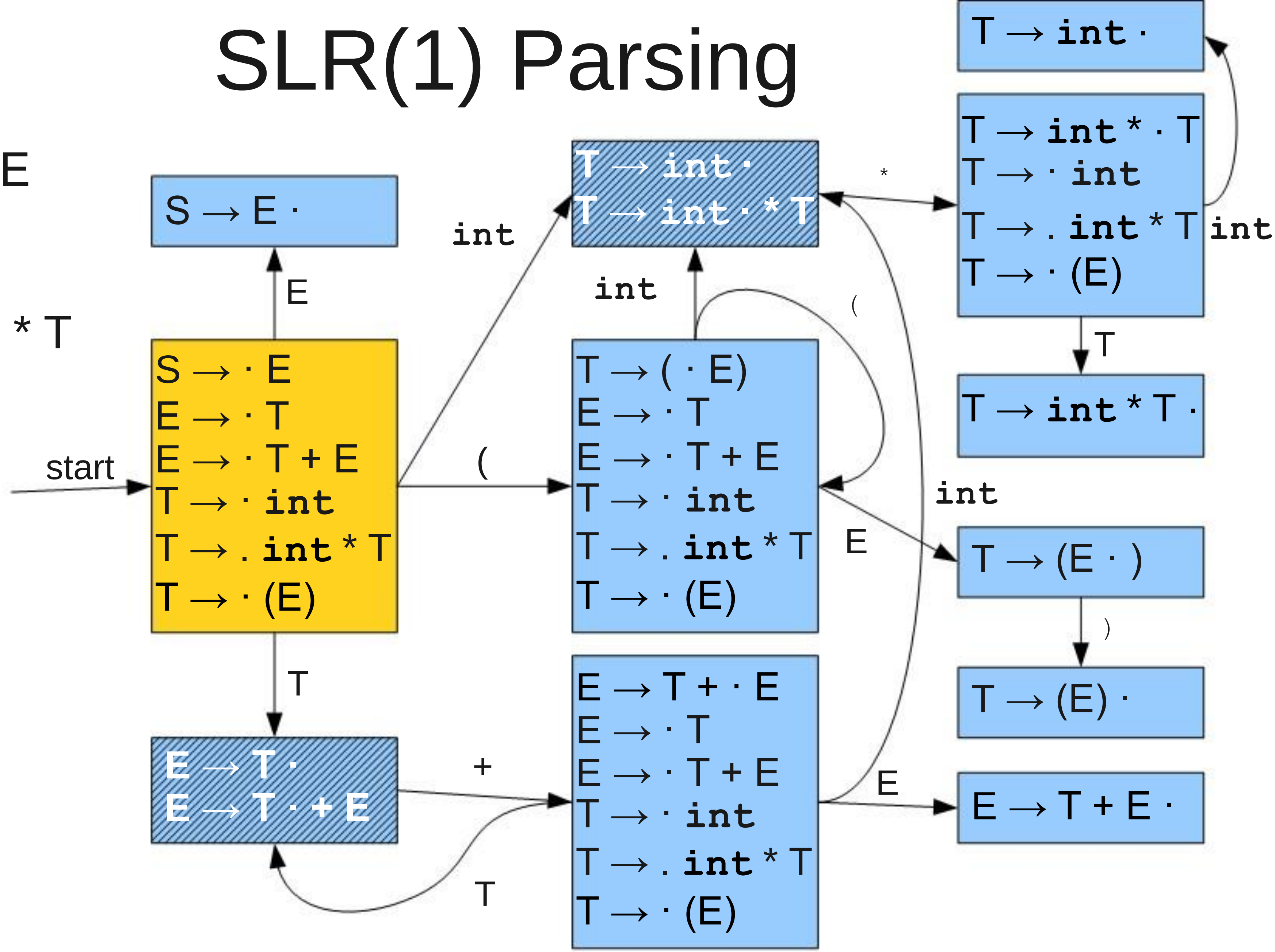
FOLLOW(T) = { +,) , $ }



SLR(1) Parsing

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

T → **int** * T

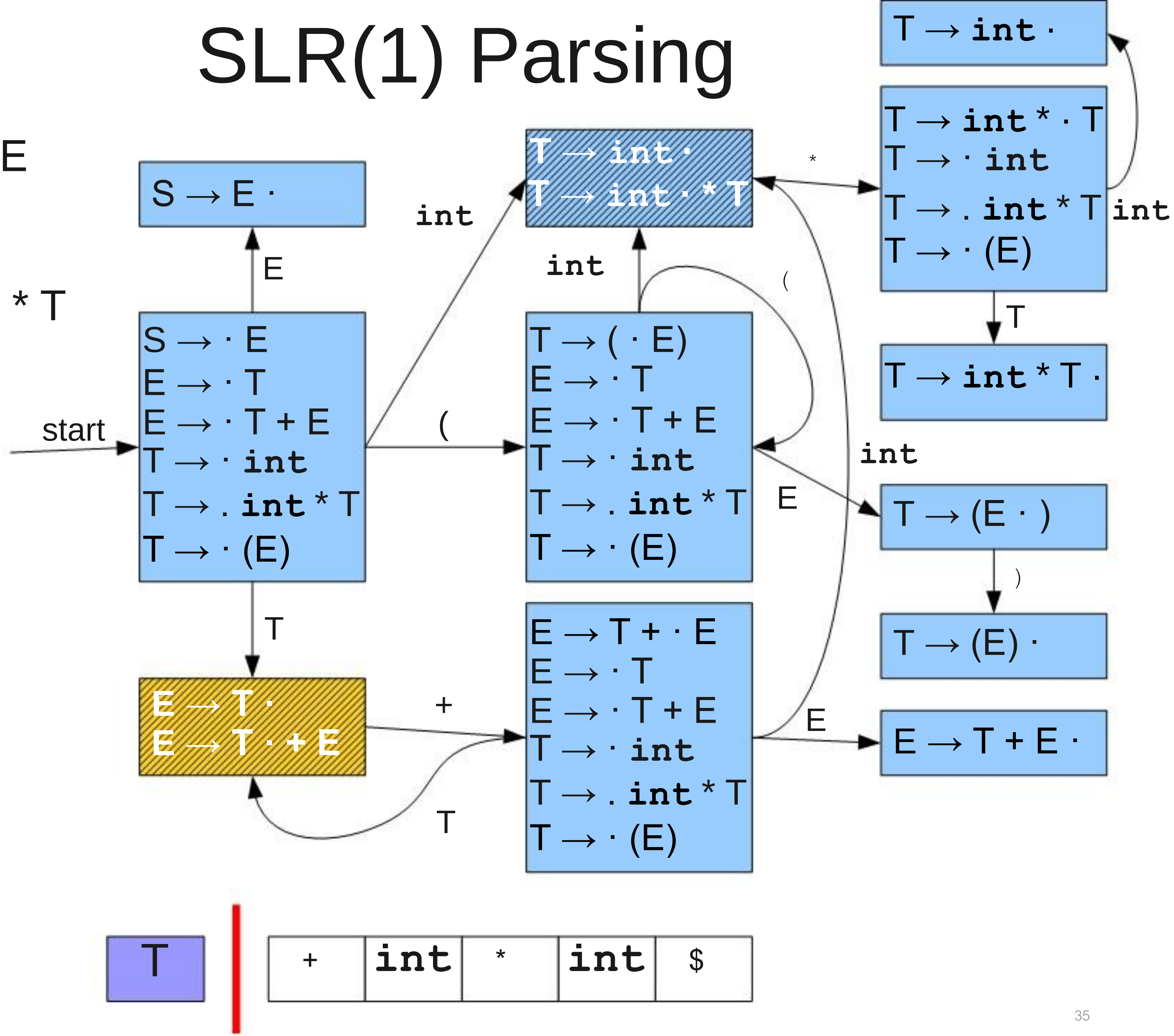


+	**int**	*	**int**	$

SLR(1) Parsing

$S \to E$
$E \to T$
$E \to T + E$
$T \to \texttt{int}$
$T \to (E)$

$T \to \texttt{int} * T$

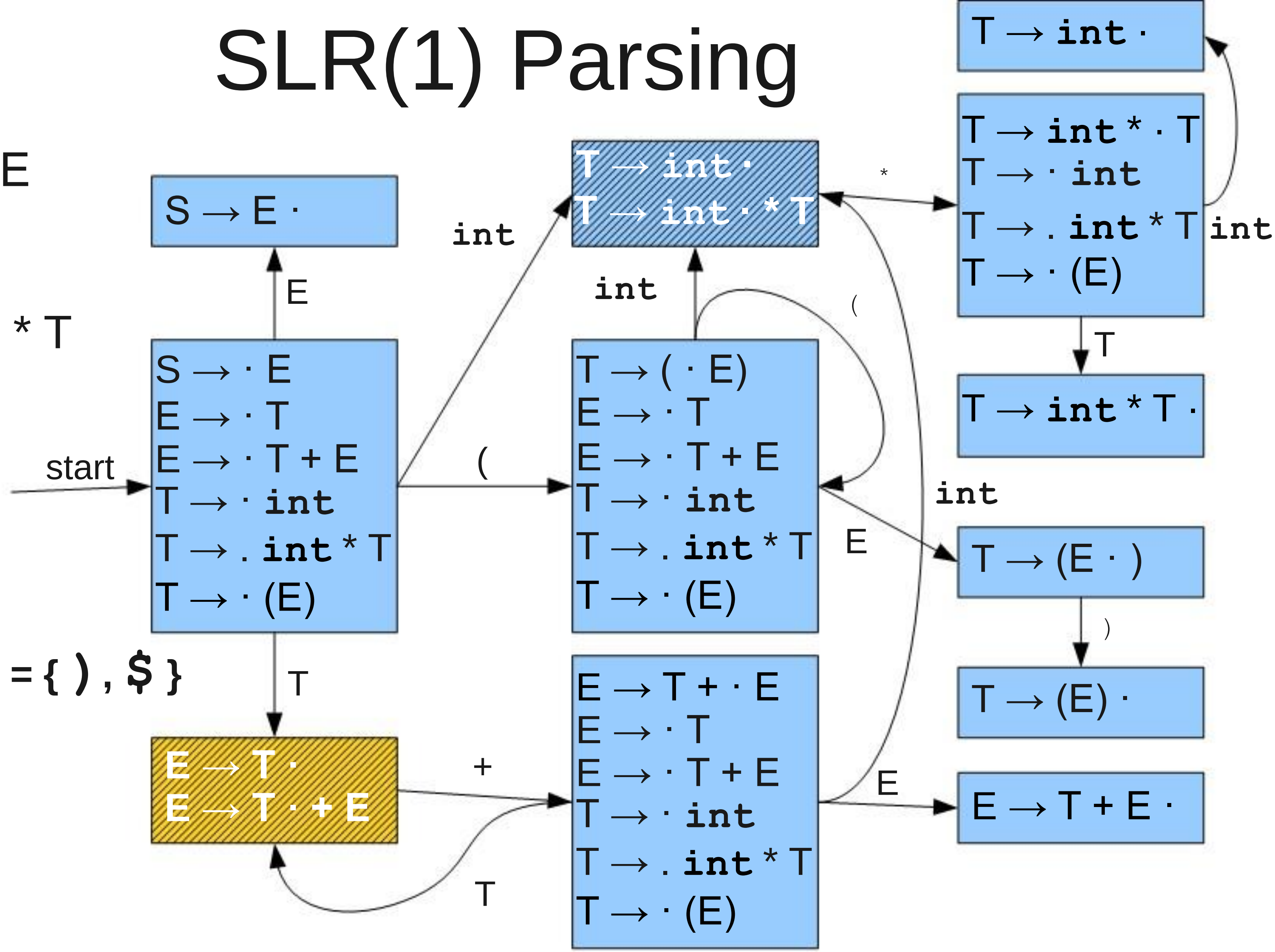


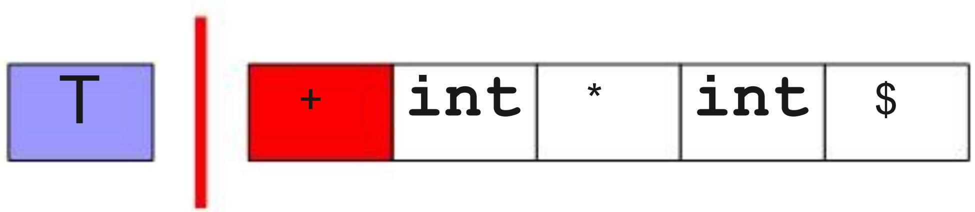
SLR(1) Parsing

$S \to E$
$E \to T$
$E \to T + E$
$T \to$ **int**
$T \to (E)$

$T \to$ **int** $* \; T$

FOLLOW(E) = {) , \$ }

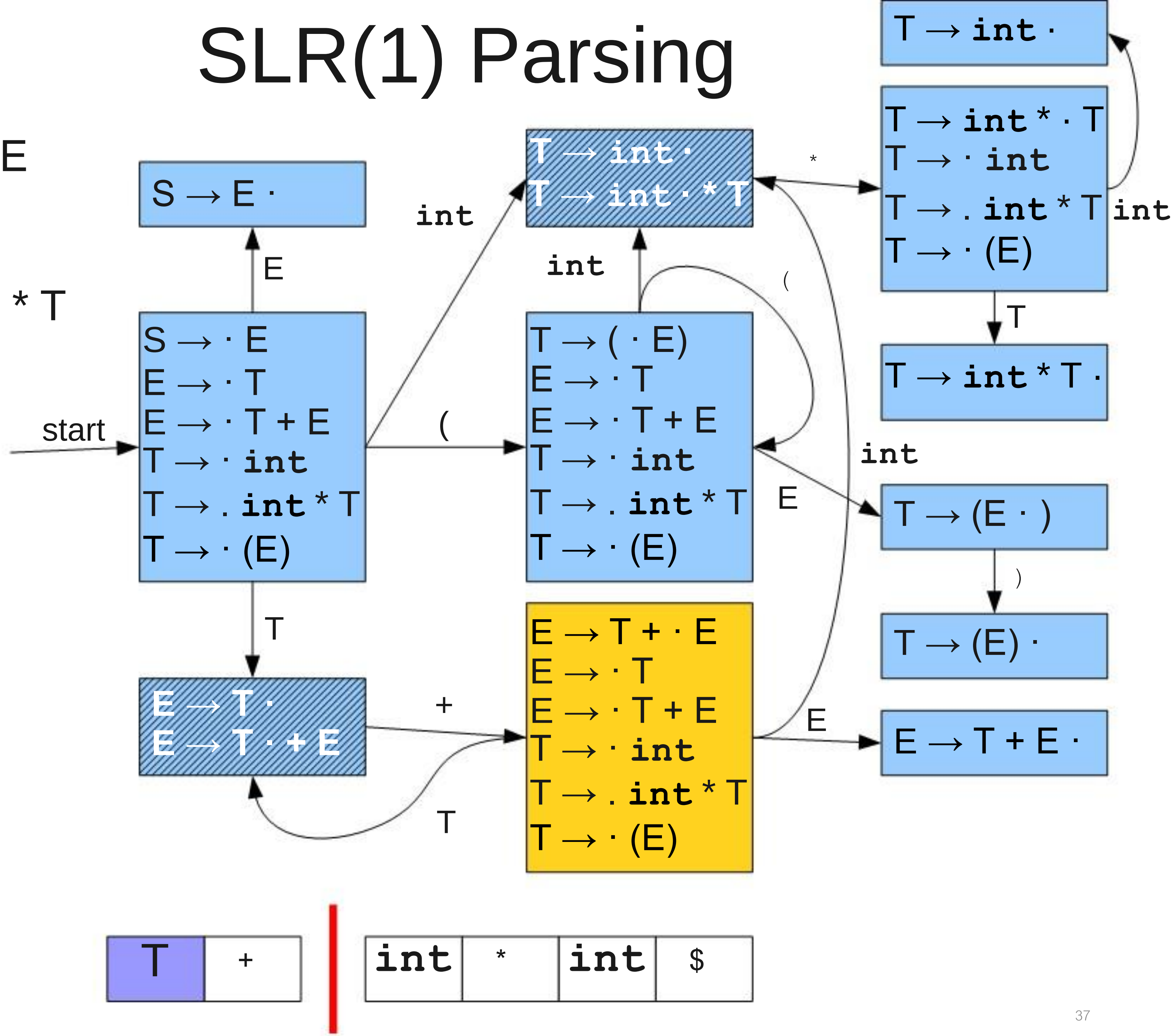




SLR(1) Parsing

$S \to E$
$E \to T$
$E \to T + E$
$T \to \texttt{int}$
$T \to (E)$

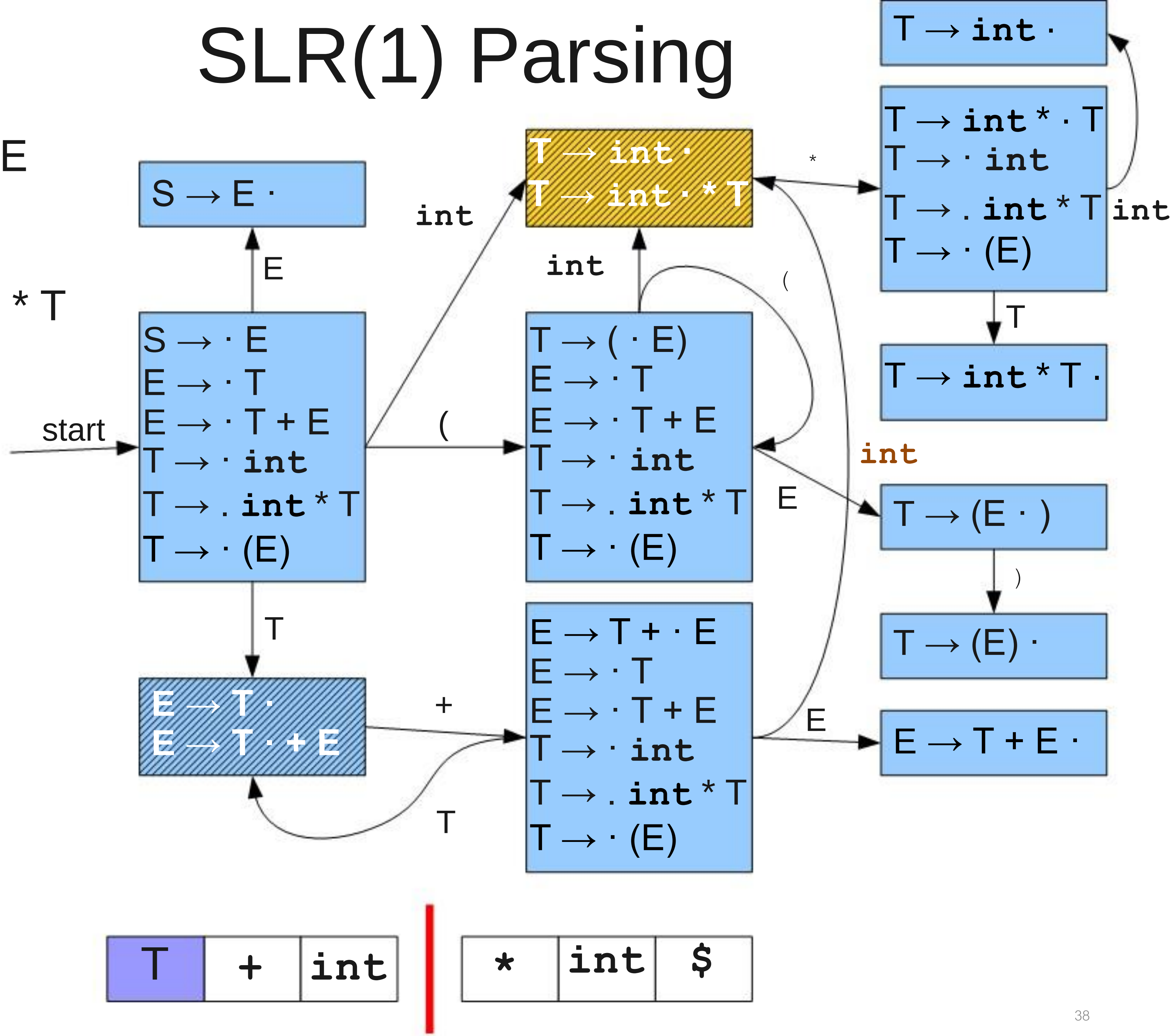
$T \to \texttt{int} * T$



SLR(1) Parsing

$S \to E$
$E \to T$
$E \to T + E$
$T \to$ `int`
$T \to (E)$

$T \to$ `int` $* T$

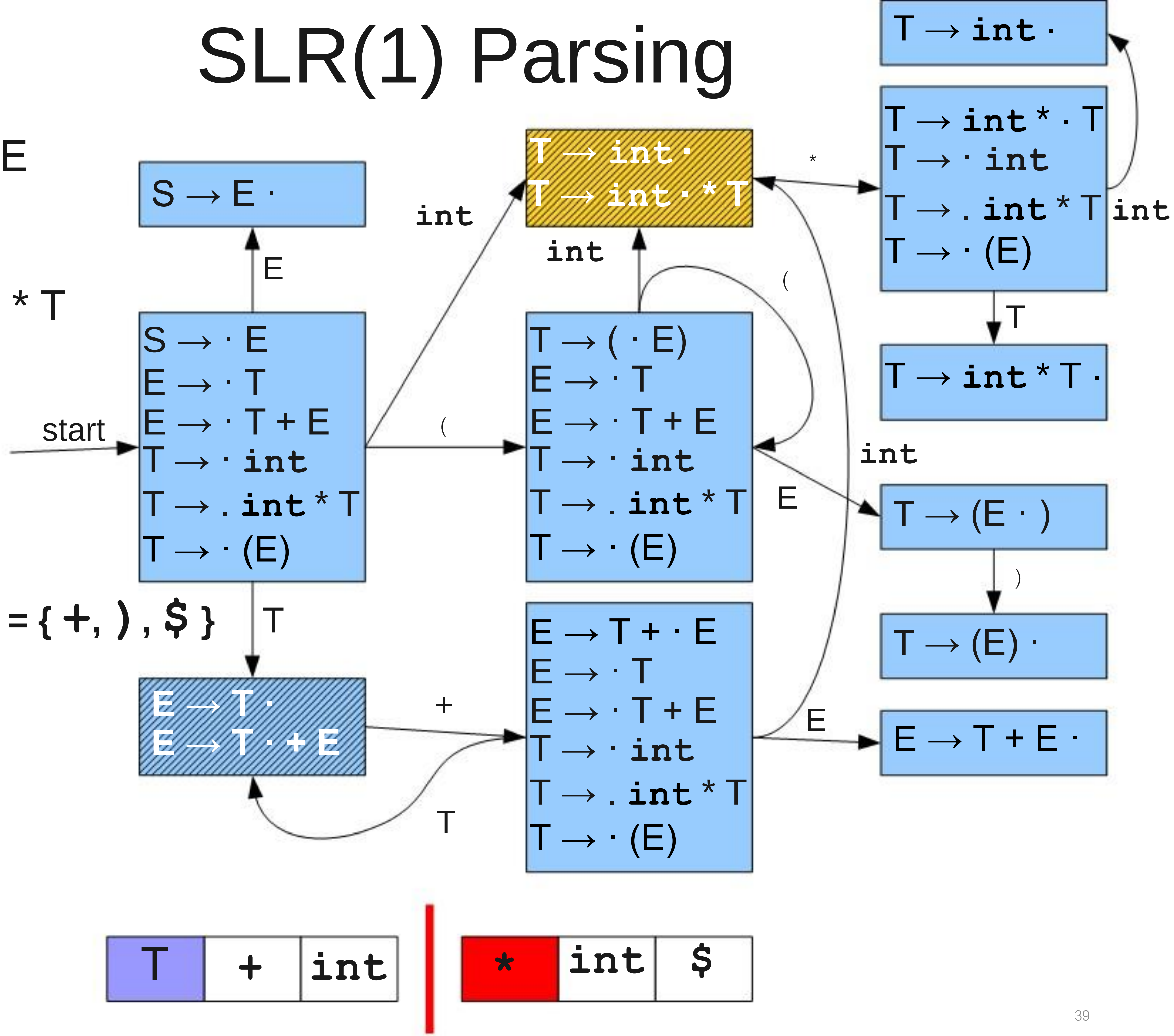


SLR(1) Parsing

$S \rightarrow E$
$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow \mathtt{int}$
$T \rightarrow (E)$

$T \rightarrow \mathtt{int} * T$

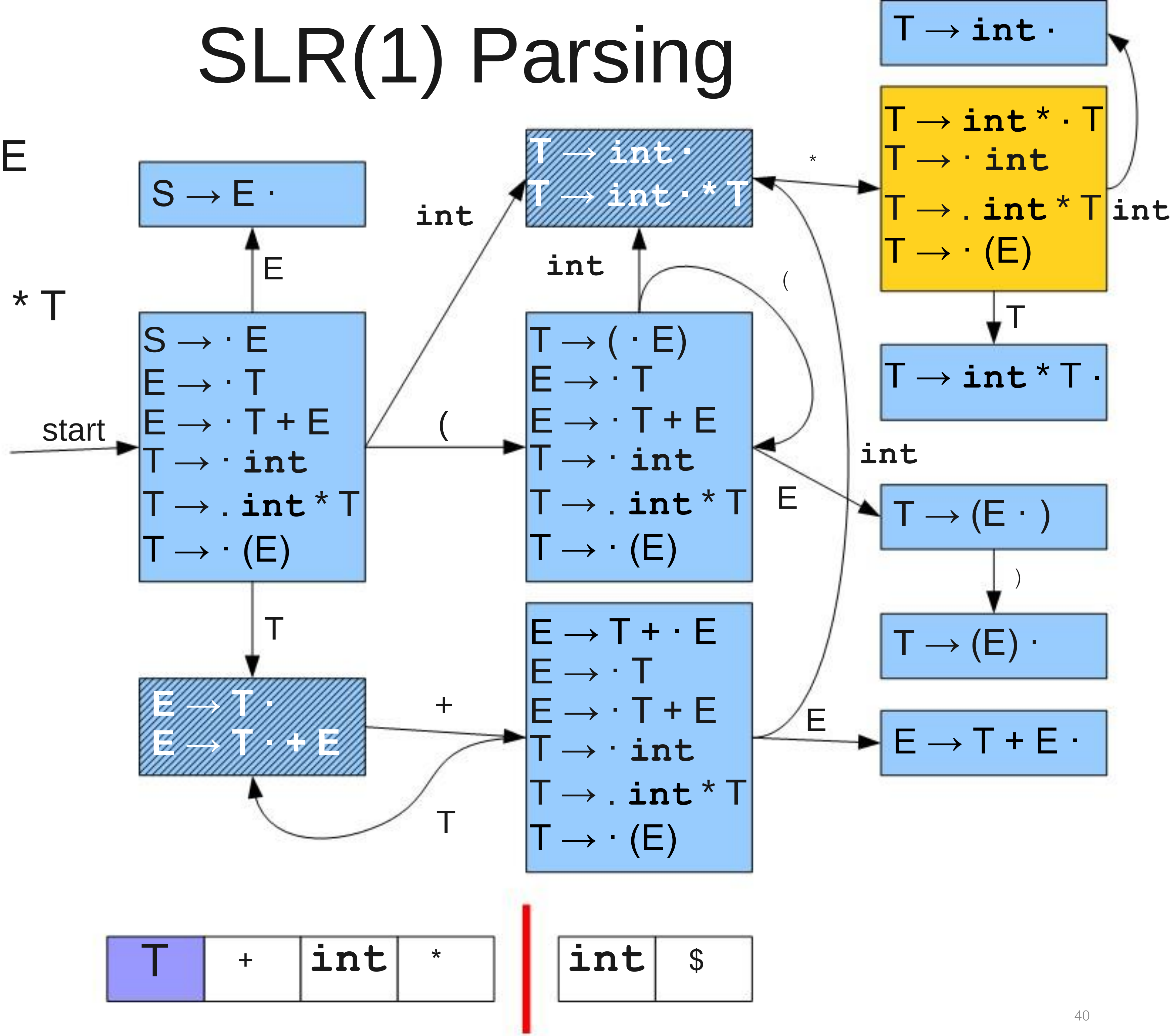
FOLLOW(T) = { +,) , $ }



SLR(1) Parsing

$S \rightarrow E$
$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

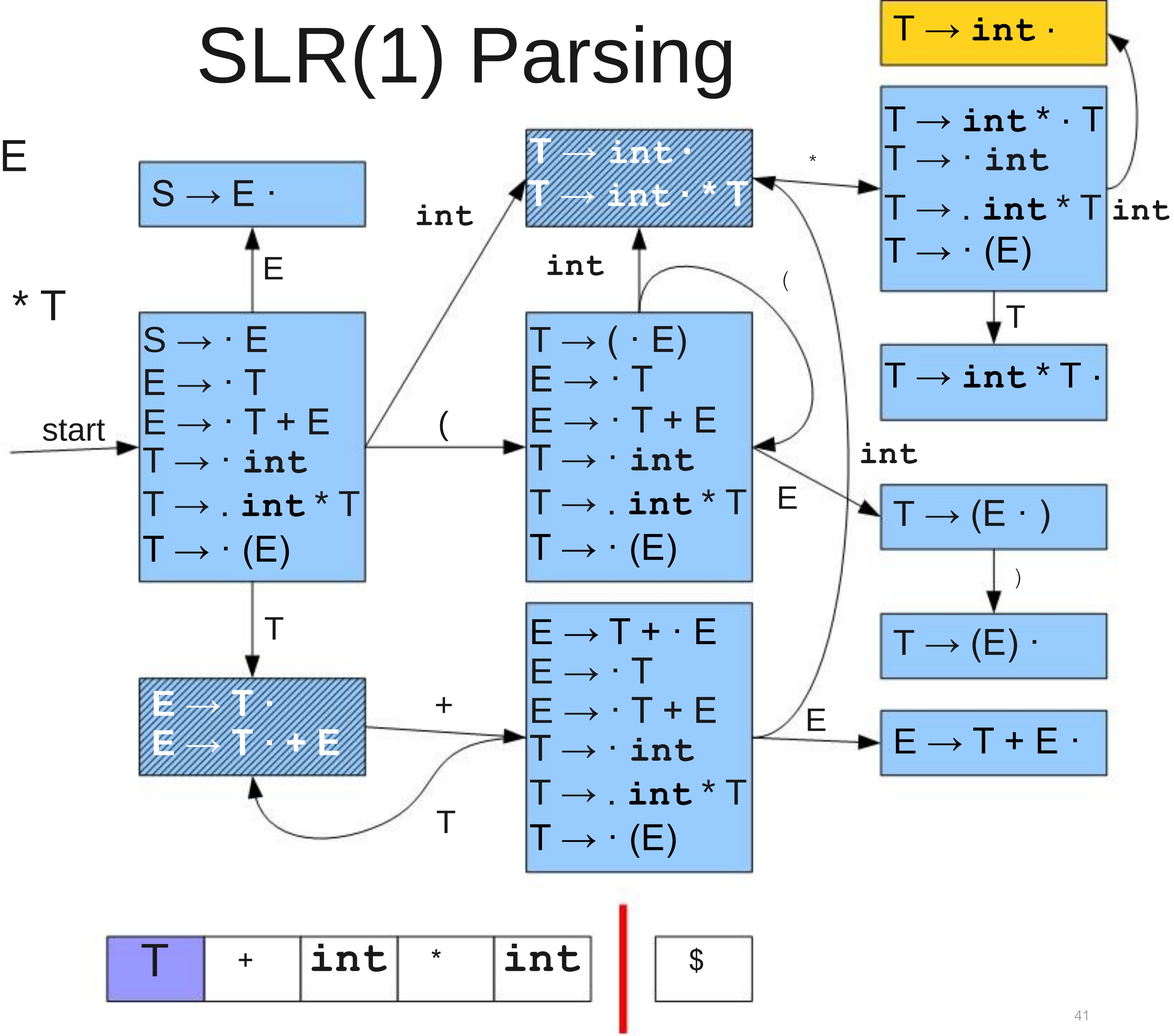
$T \rightarrow \texttt{int} * T$



SLR(1) Parsing

S → E
E → T
E → T + E
T → `int`
T → (E)

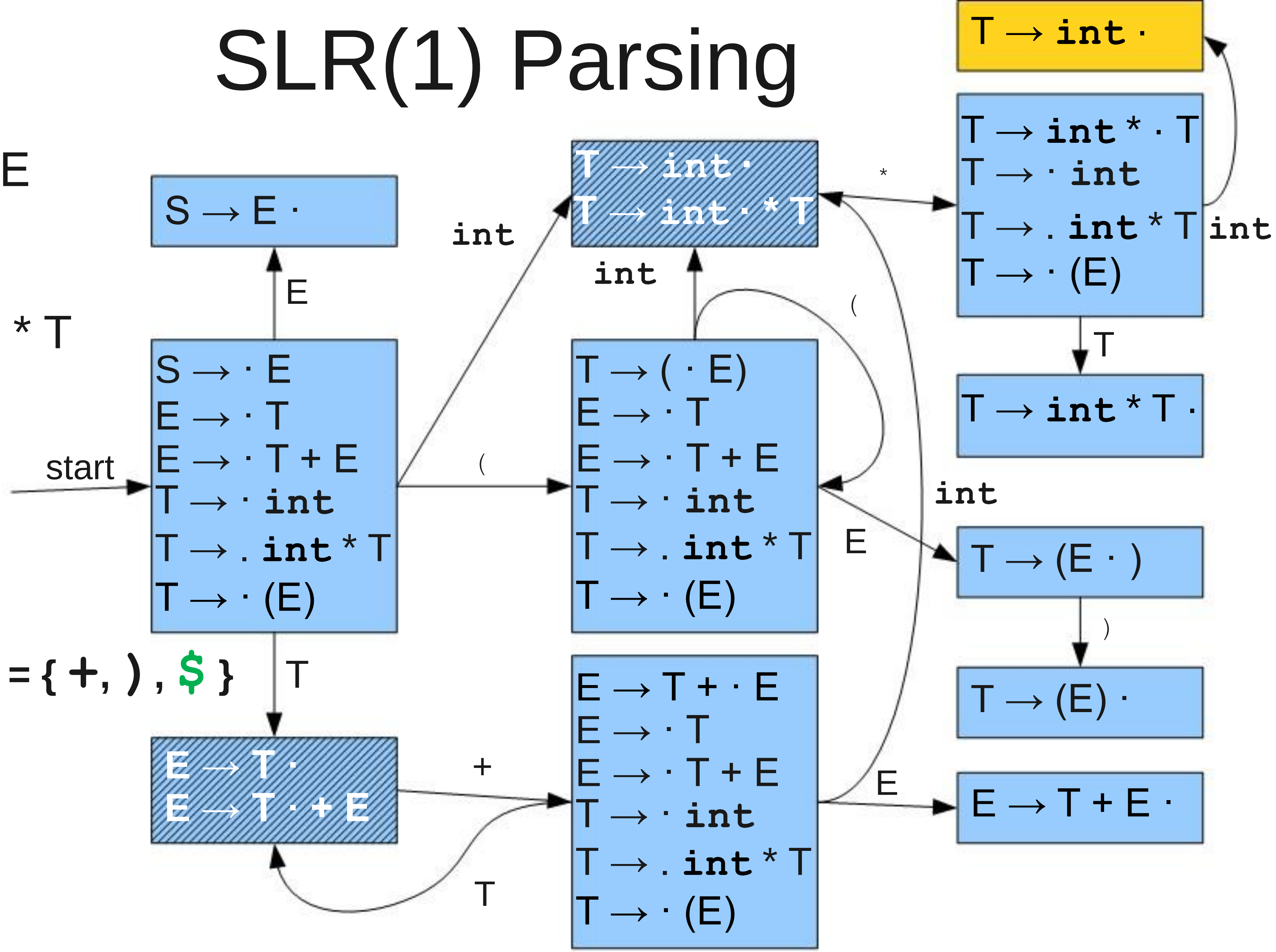
T → `int` * T



SLR(1) Parsing

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

T → **int** * T



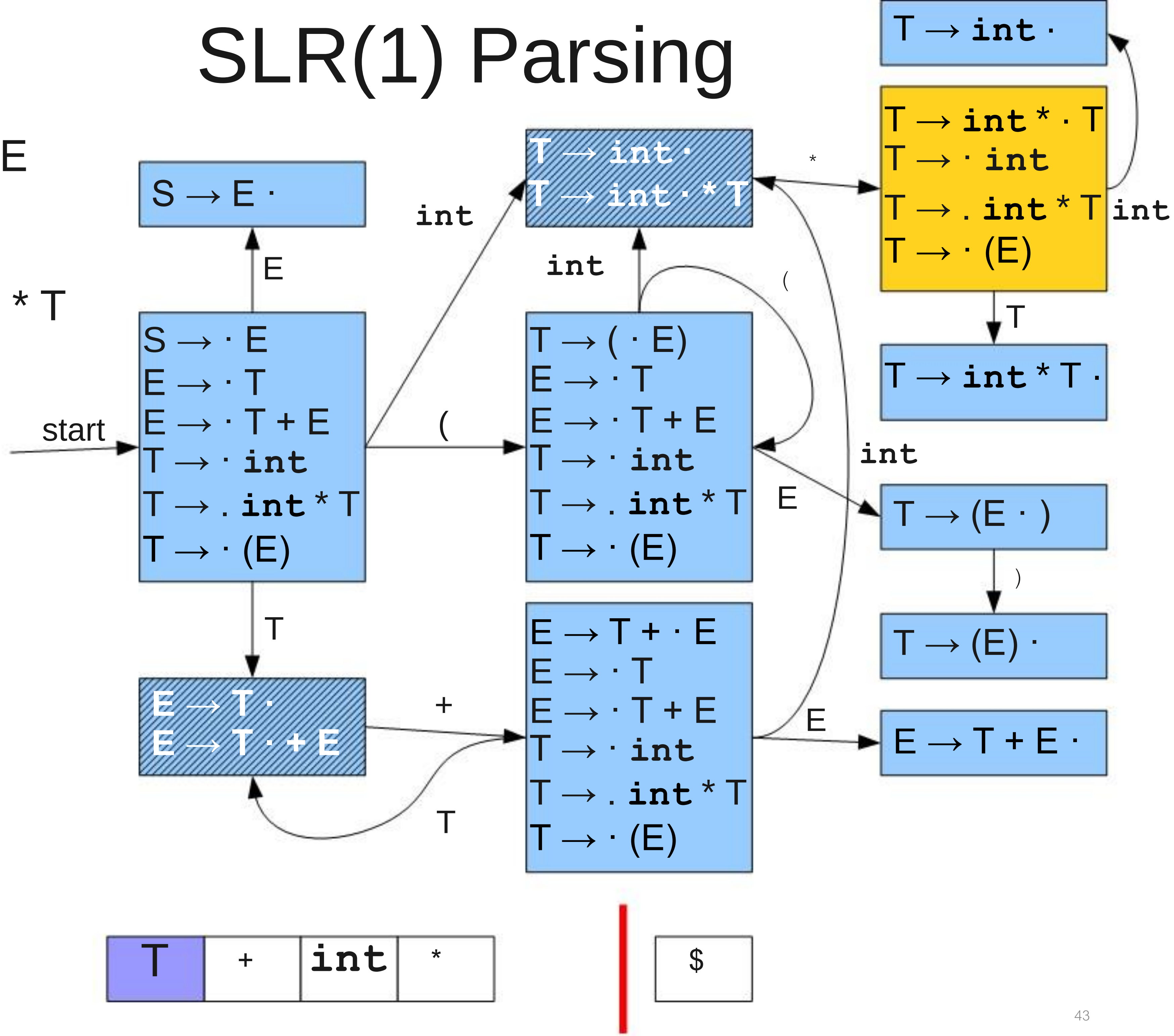
FOLLOW(T) = { +,) , $ }

T	+	int	*	int		$

SLR(1) Parsing

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

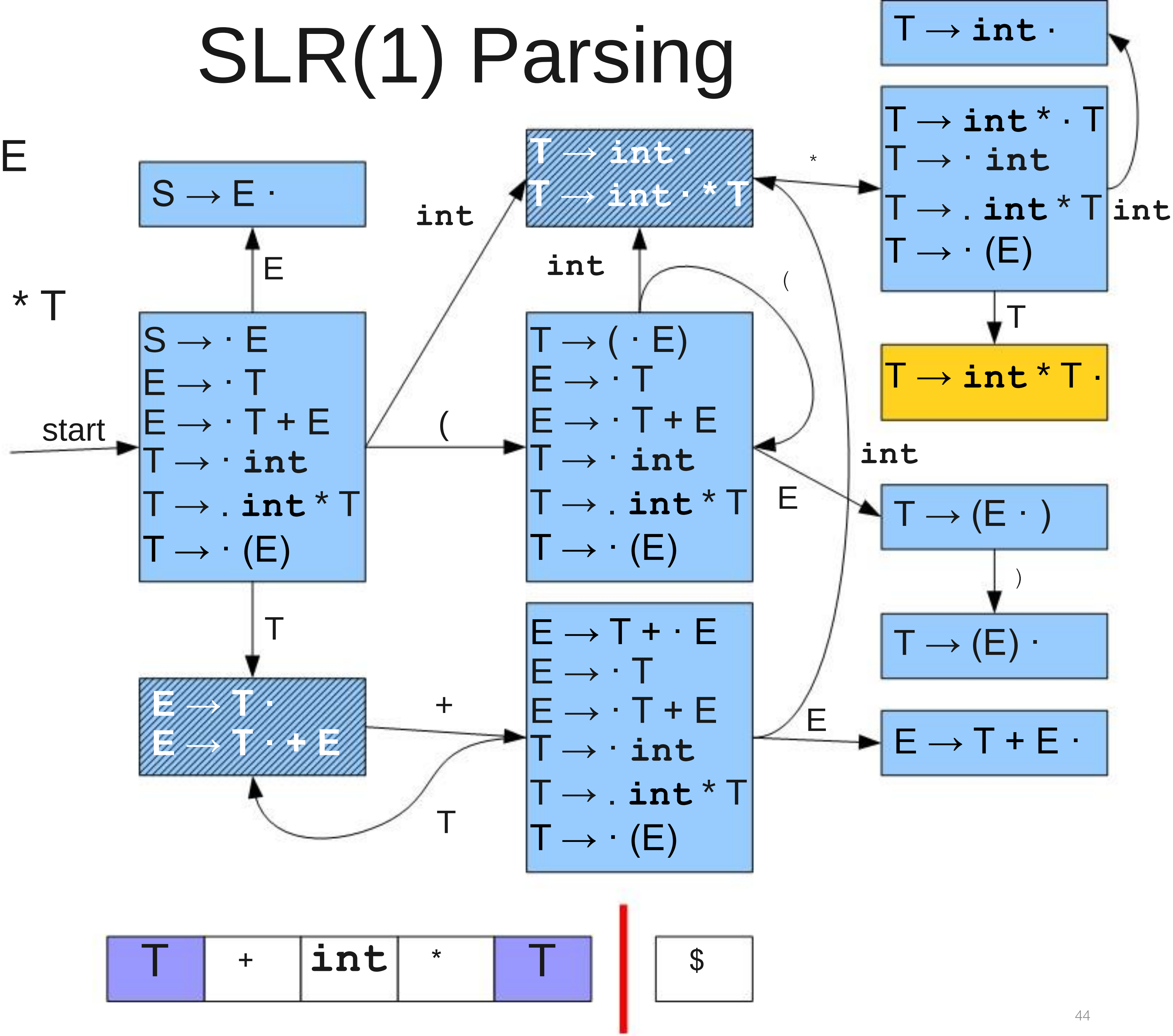
T → **int** * T



SLR(1) Parsing

$S \to E$
$E \to T$
$E \to T + E$
$T \to \texttt{int}$
$T \to (E)$

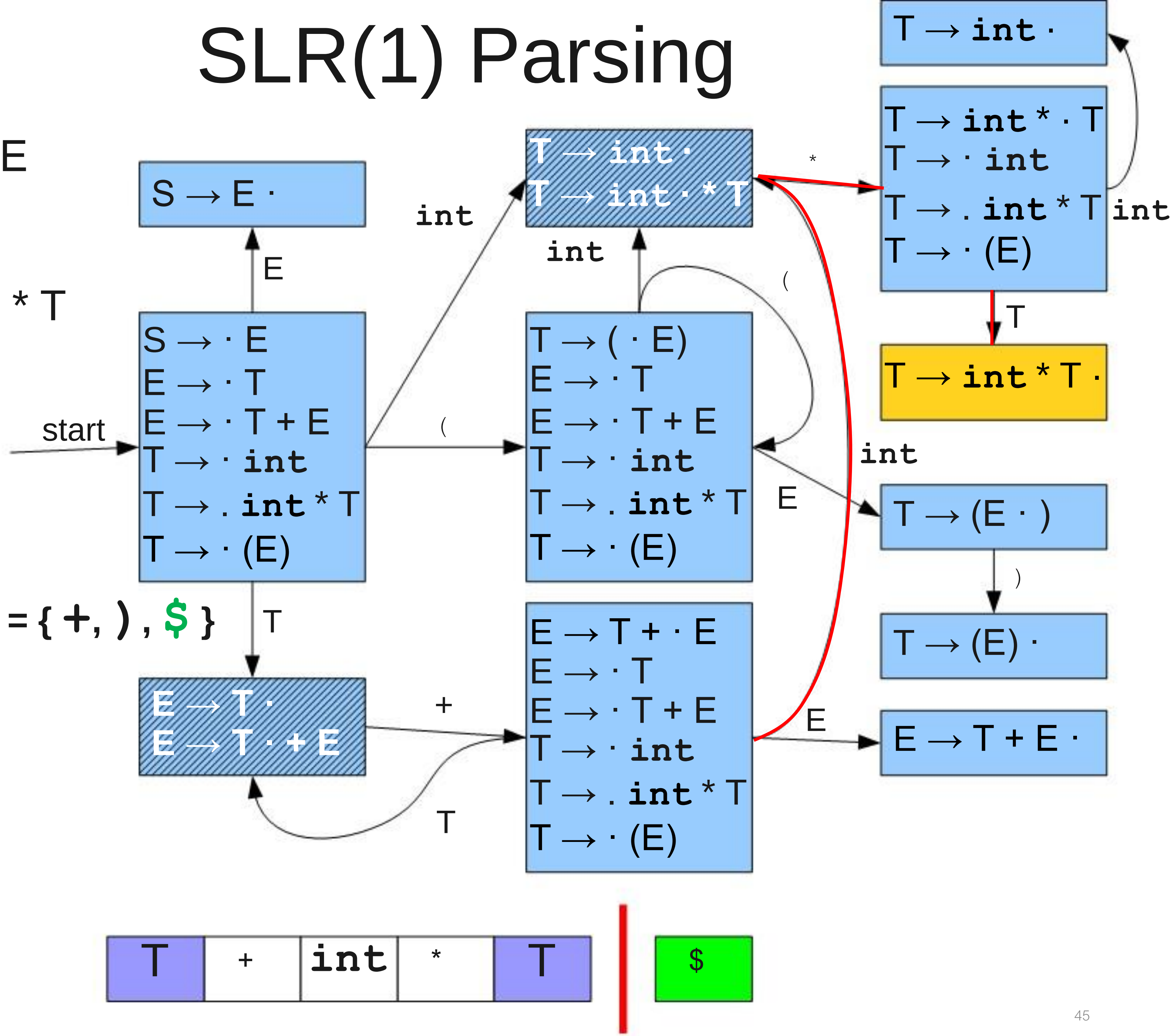
$T \to \texttt{int} * T$



SLR(1) Parsing

S → E
E → T
E → T + E
T → `int`
T → (E)

T → `int` * T

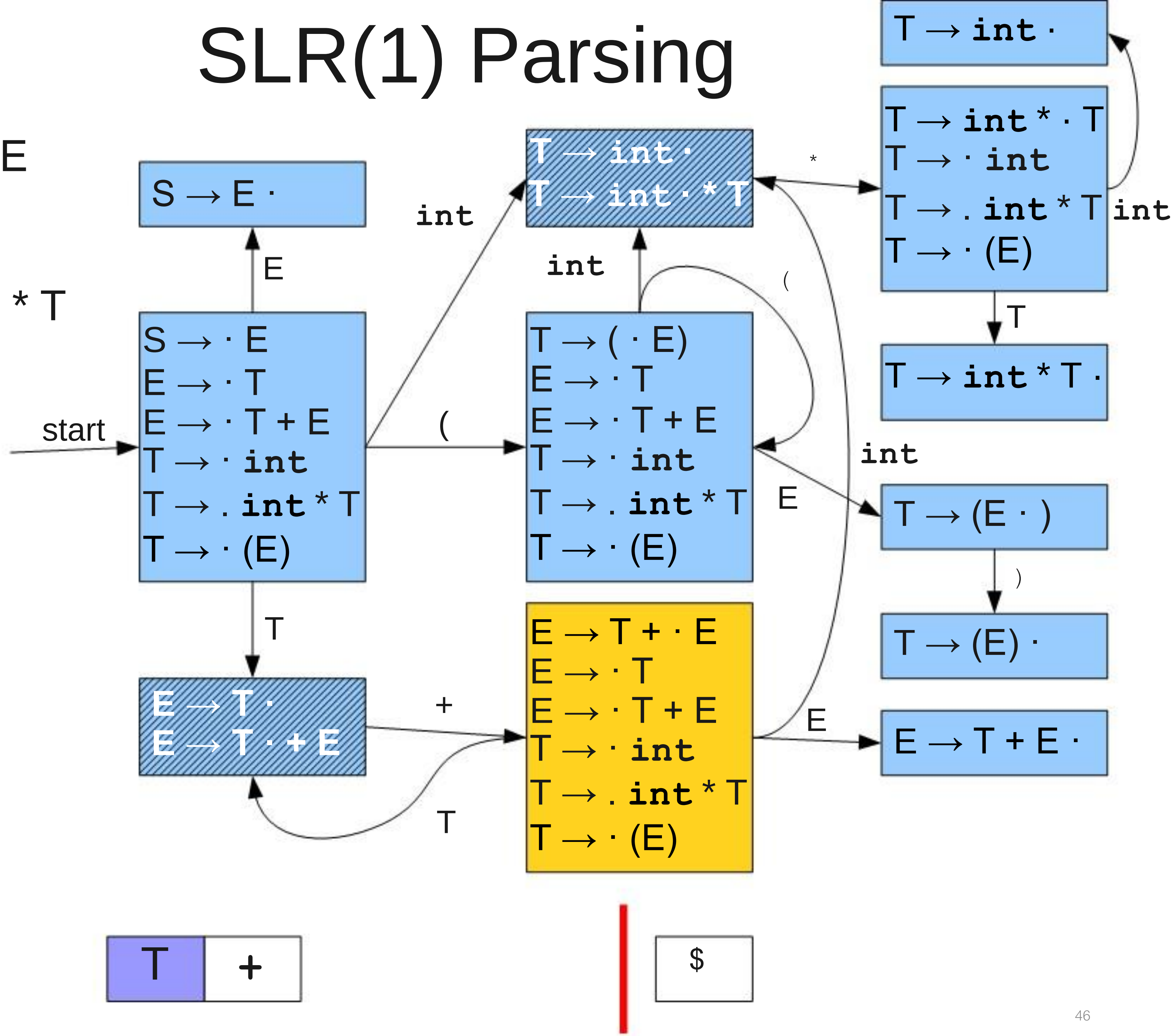


FOLLOW(T) = { +,) , $ }

SLR(1) Parsing

$S \rightarrow E$
$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

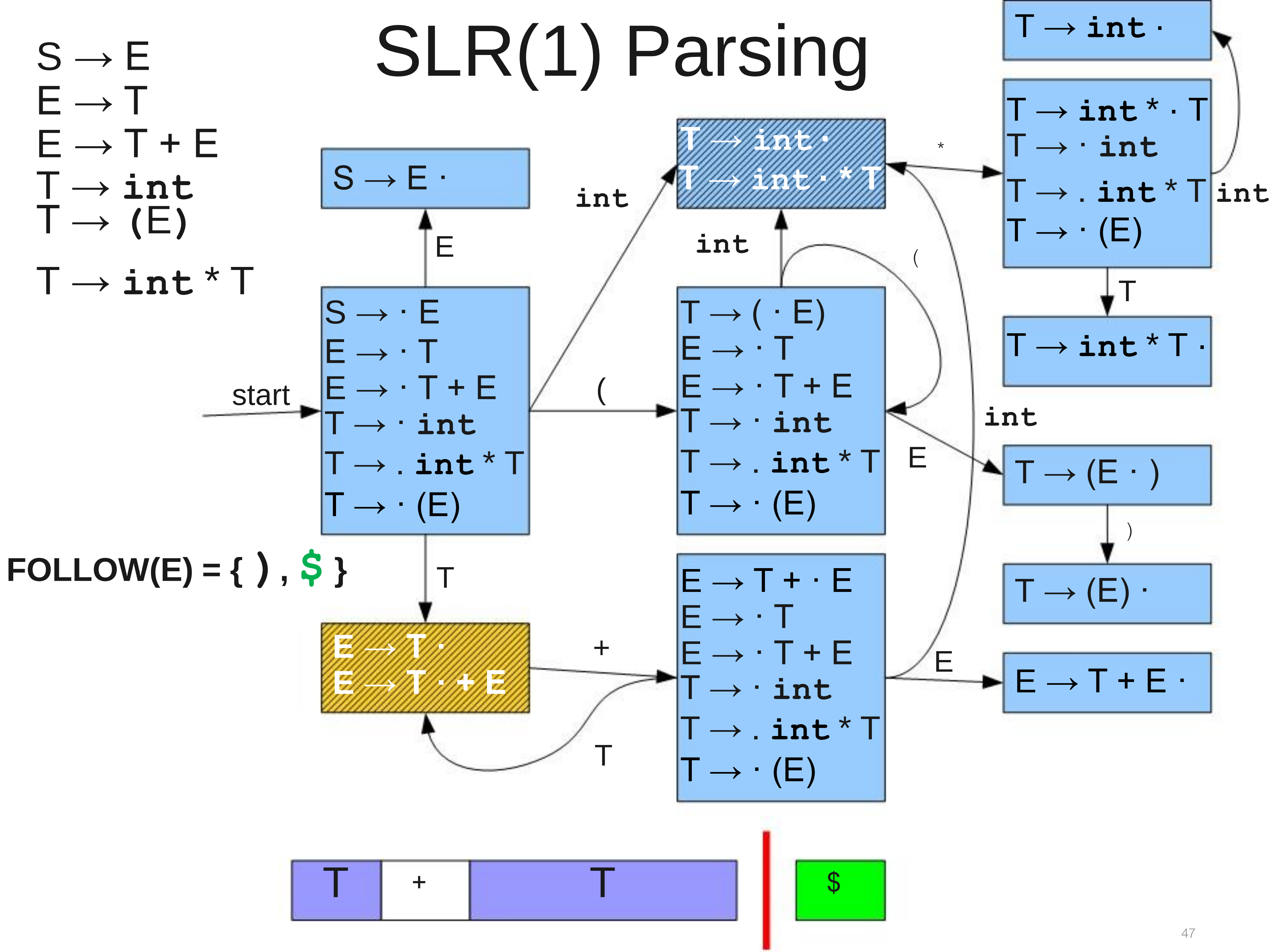
$T \rightarrow \texttt{int} * T$



SLR(1) Parsing

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

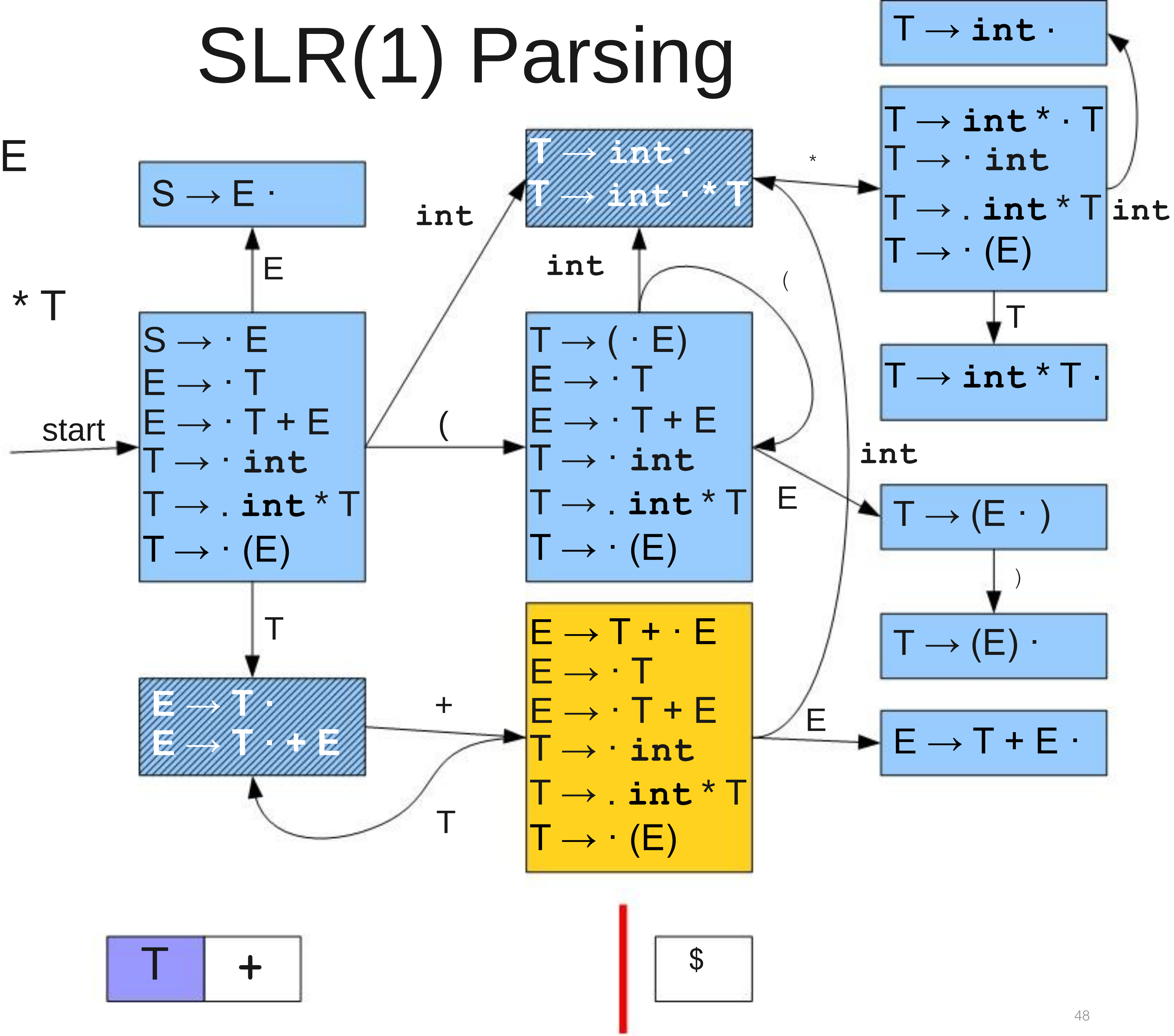
T → **int** * T



SLR(1) Parsing

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

$T \rightarrow$ **int** $* \; T$

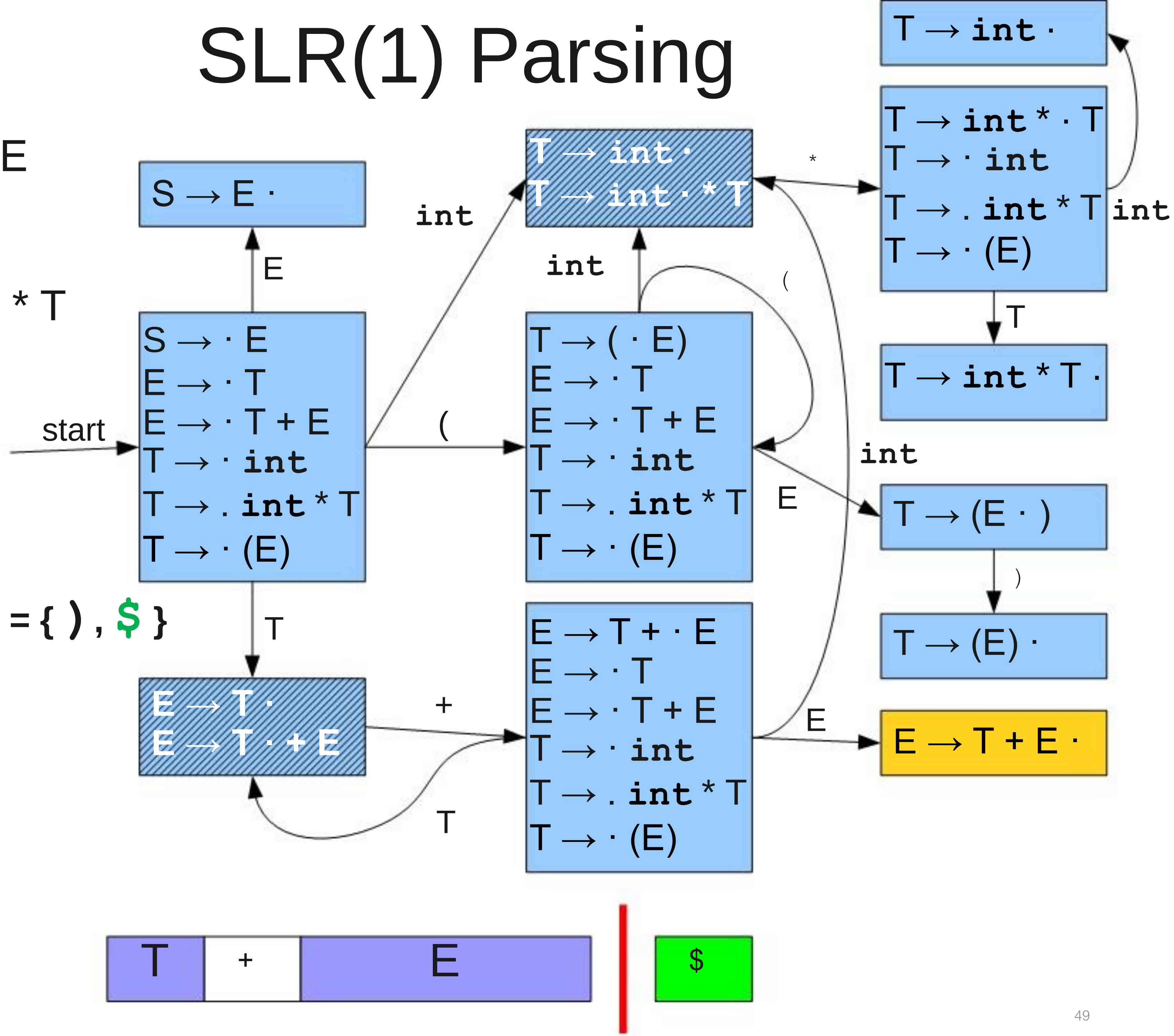


SLR(1) Parsing

$S \to E$
$E \to T$
$E \to T + E$
$T \to$ **int**
$T \to (E)$

$T \to$ **int** $* \; T$

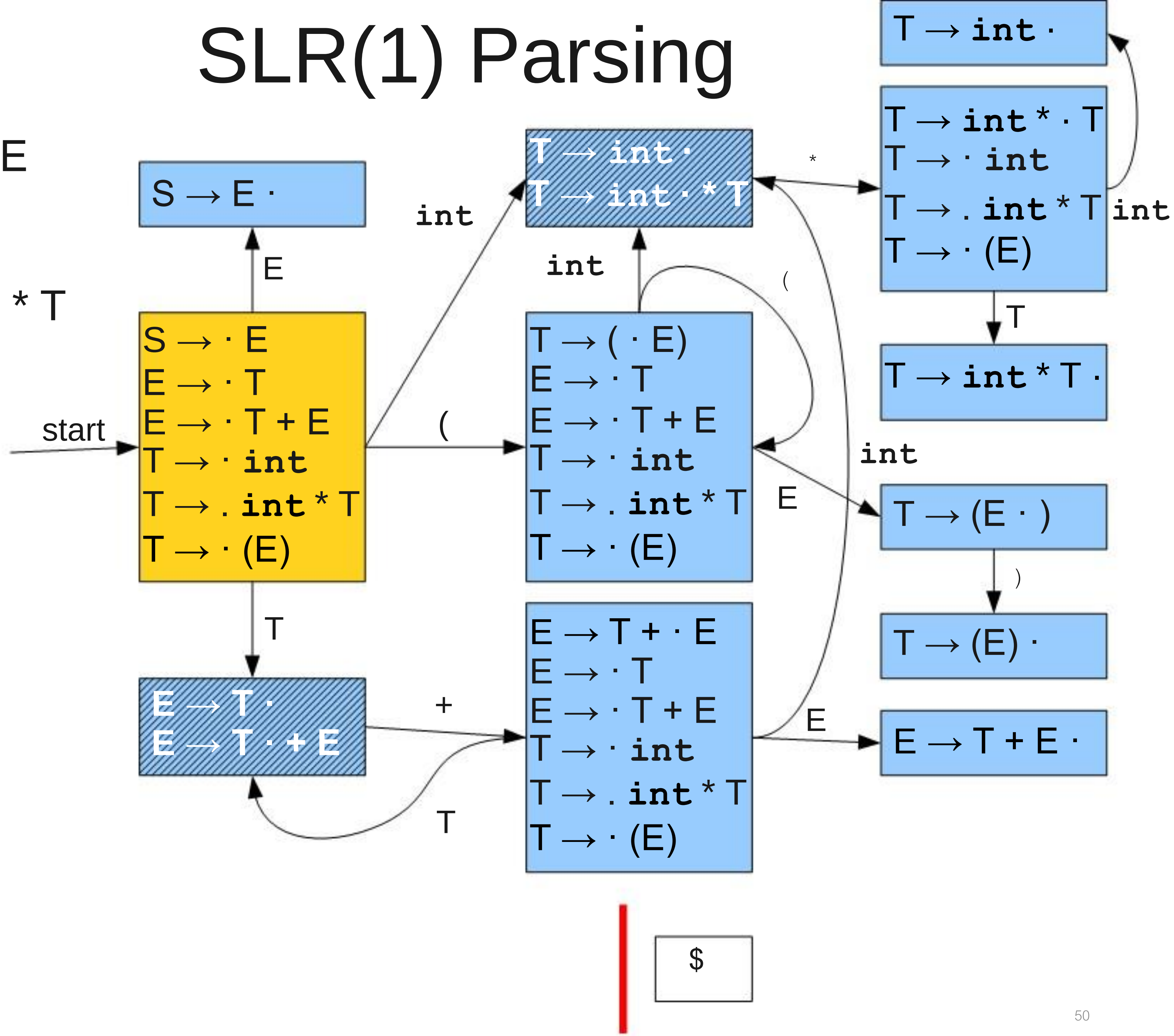
FOLLOW(E) = {) , $ }



SLR(1) Parsing

$S \to E$
$E \to T$
$E \to T + E$
$T \to$ **int**
$T \to (E)$

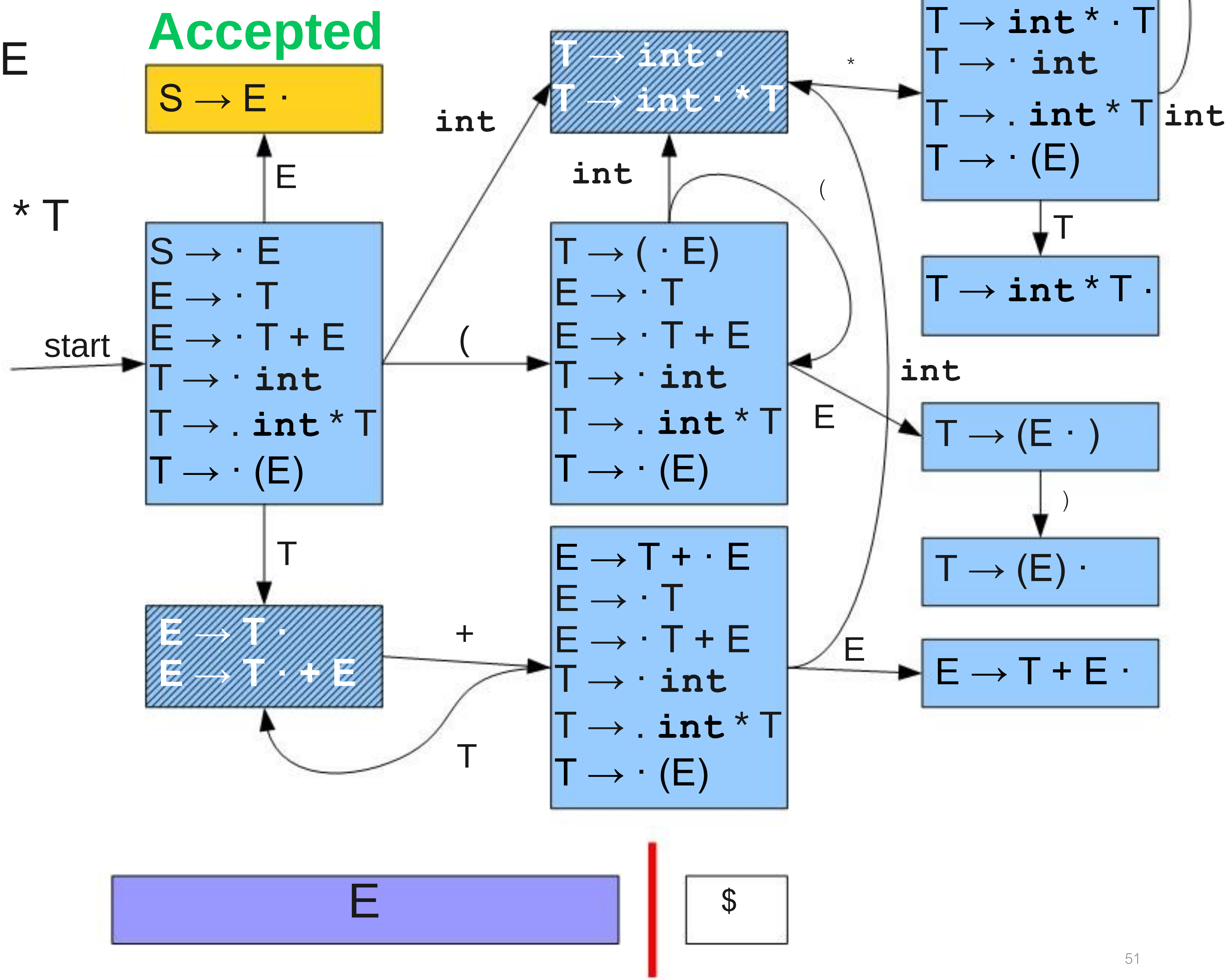
$T \to$ **int** $* \ T$



SLR(1) Parsing

$S \to E$
$E \to T$
$E \to T + E$
$T \to$ **int**
$T \to (E)$

$T \to$ **int** $* T$



Recap

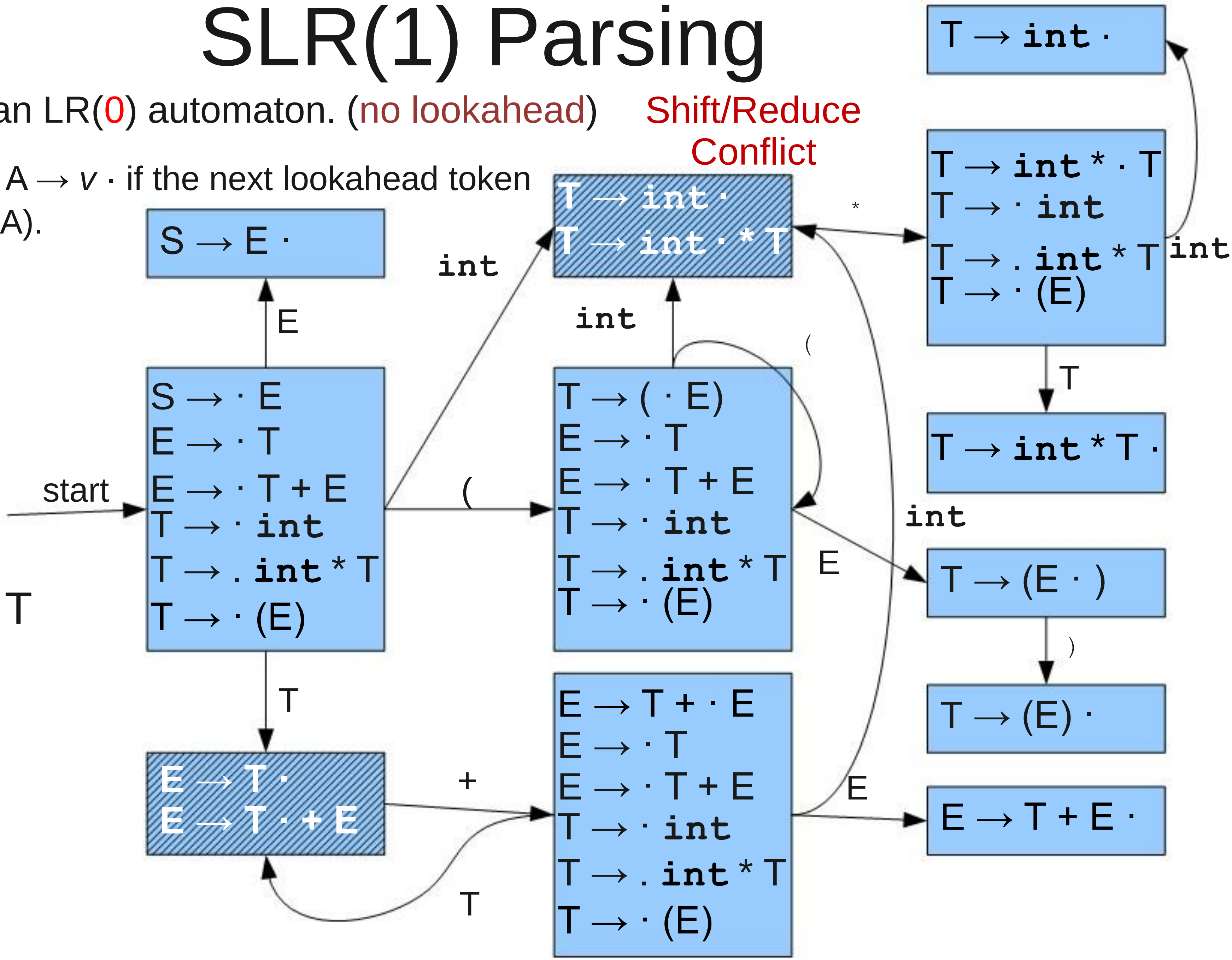
- LR(0) parsing only works on grammars when reduces are unambiguous.
- LR(1) parsing works on a large number of grammars, but requires too large a parse table.
- SLR(1) parsing augments LR(0) with one look ahead token.

Reduce $A \rightarrow v \cdot$ if the next lookahead token is in FOLLOW(A).

SLR(1) Parsing

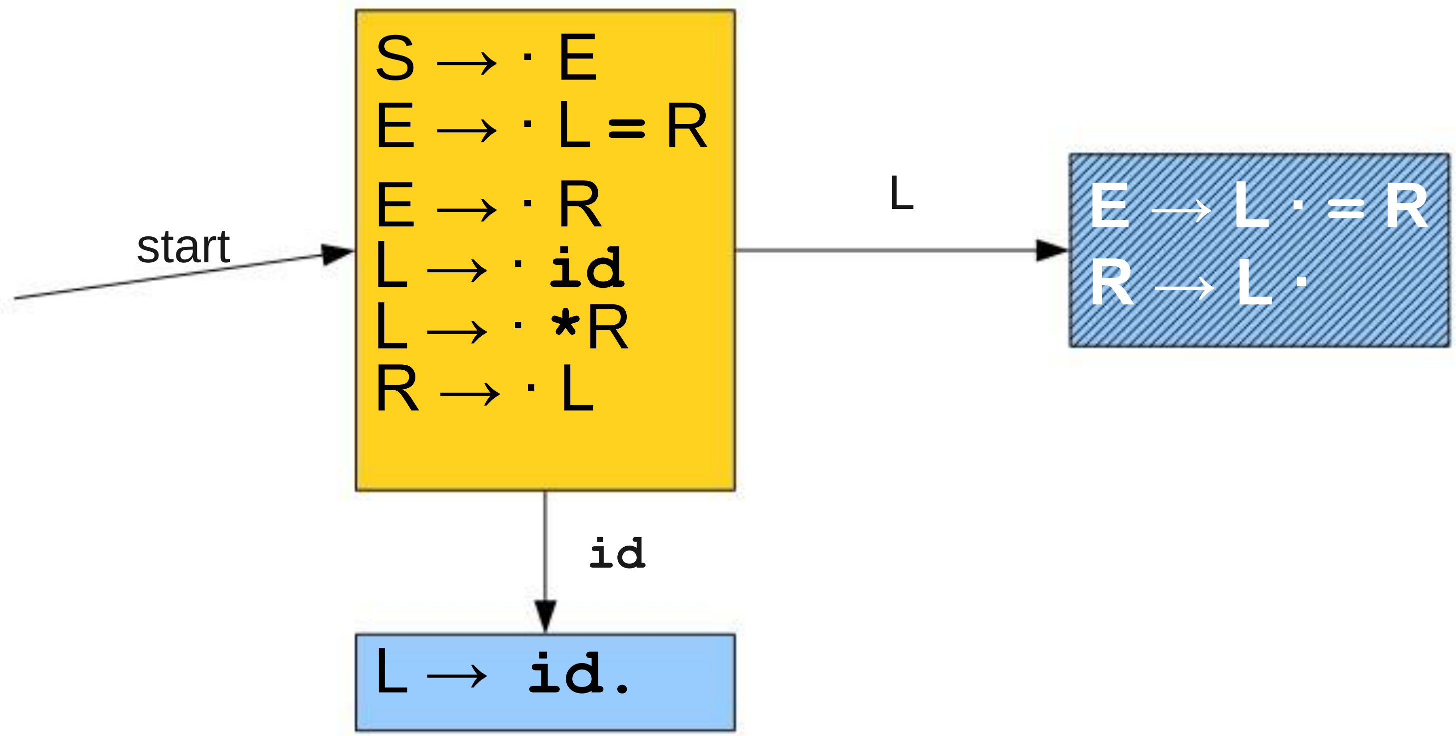
- Construct an LR(0) automaton. (no lookahead)
- Only reduce $A \to v \cdot$ if the next lookahead token is in FOLLOW(A).

$S \to E$
$E \to T$
$E \to T + E$
$T \to \texttt{int}$
$T \to (E)$
$T \to \texttt{int} * T$



The Limits of SLR(1)

$S \to E$
$E \to L = R$
$E \to R$
$L \to \textbf{id}$
$L \to *R$
$R \to L$



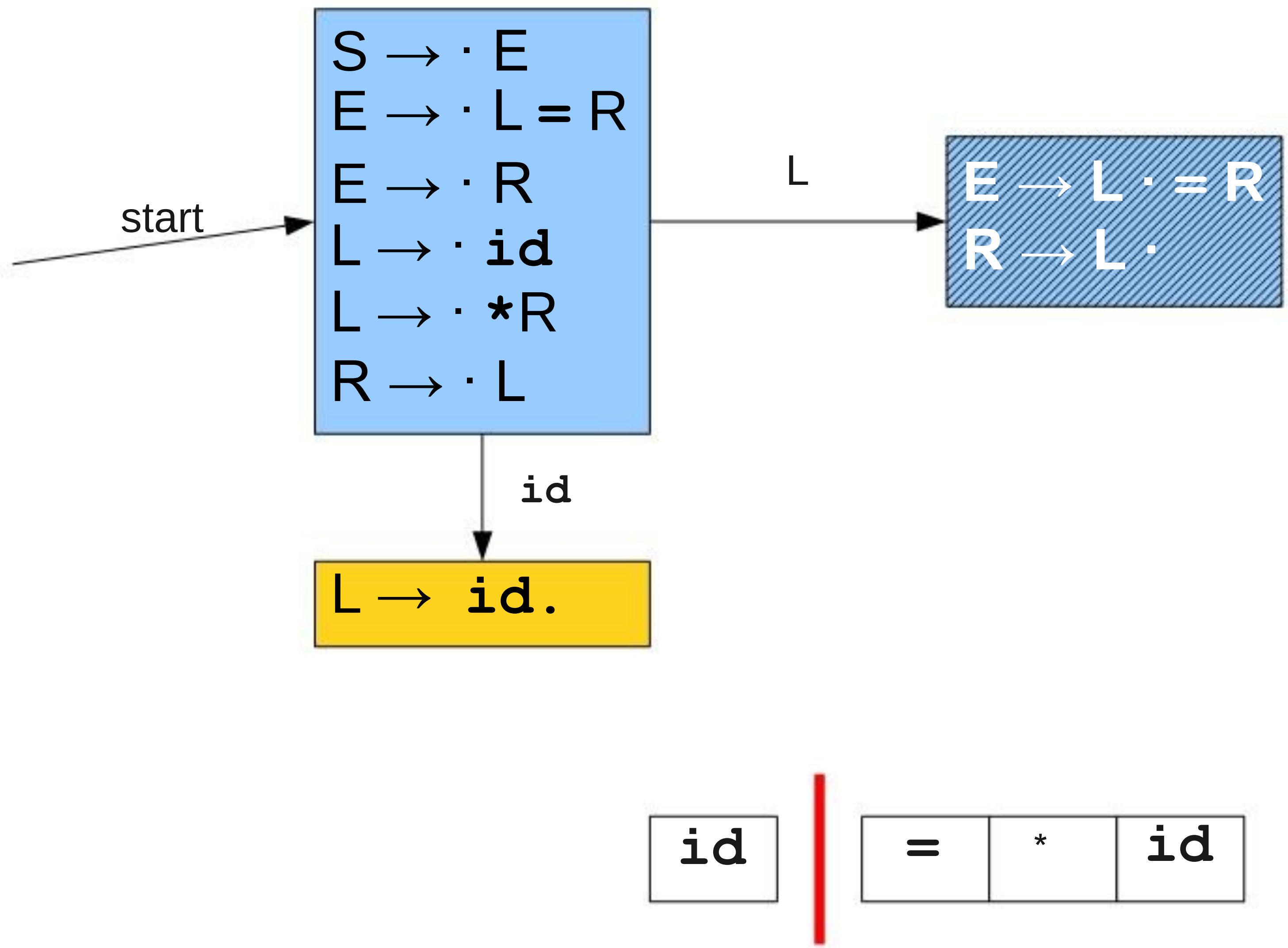
id	=	*	id

We still have a shift/reduce conflict!

Besides **shift**, We can also do action **reduce** since ‘=’ is in Follow(R)

A Lack of Context of SLR(1)

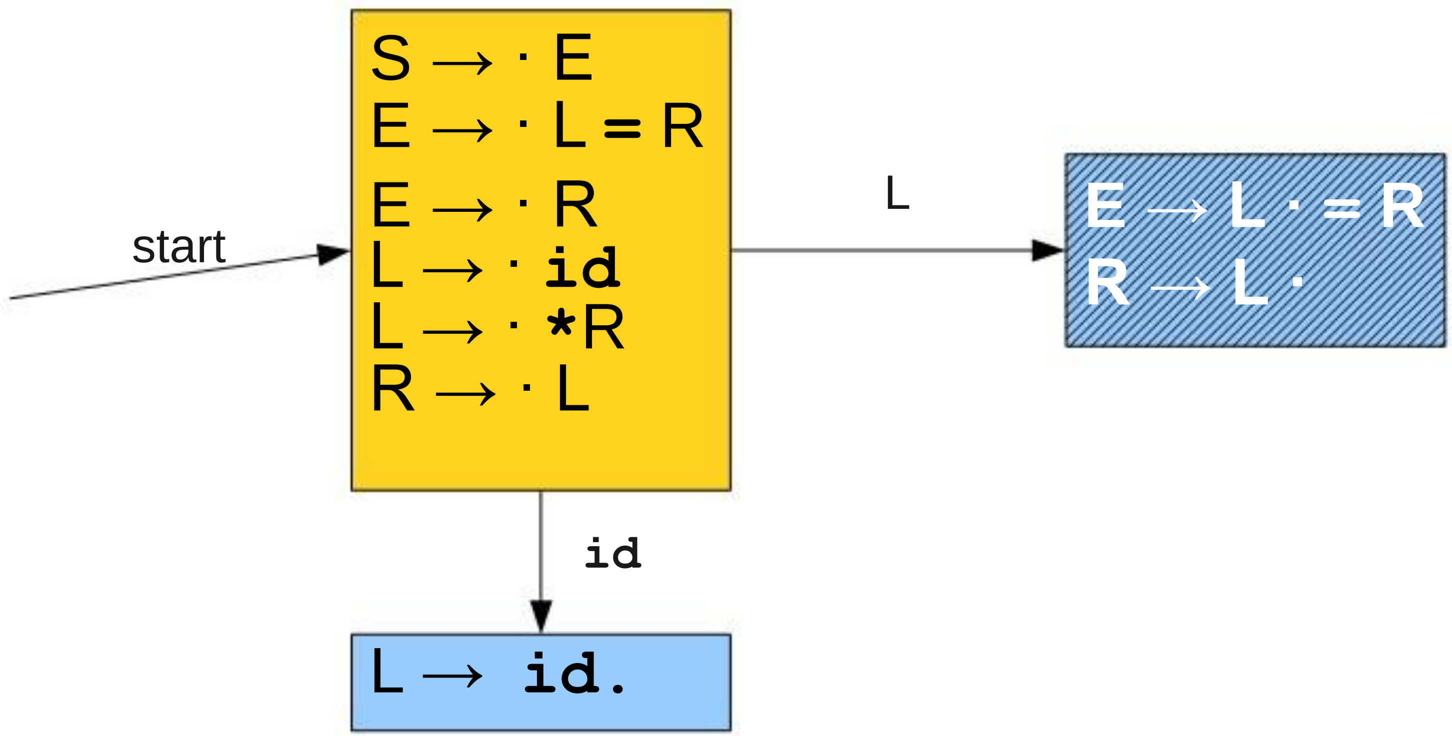
$S \rightarrow E$
$E \rightarrow L = R$
$E \rightarrow R$
$L \rightarrow$ **`id`**
$L \rightarrow$ **`*`**R
$R \rightarrow L$

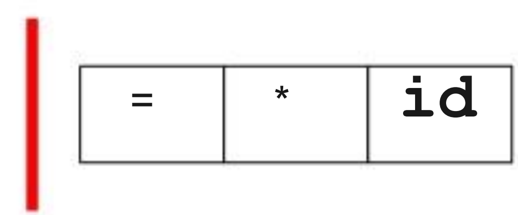


‘=’ is in Follow(L)

A Lack of Context of SLR(1)

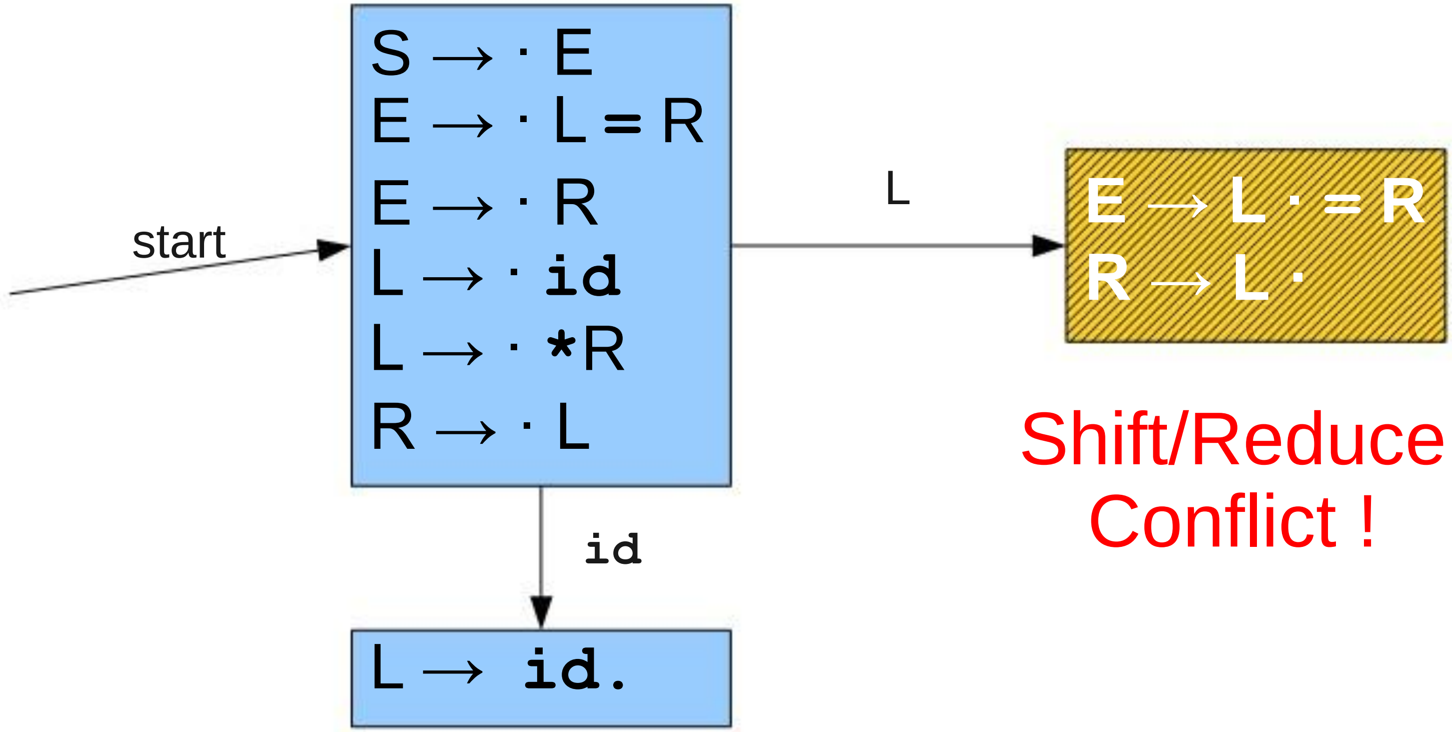
$S \rightarrow E$
$E \rightarrow L = R$
$E \rightarrow R$
$L \rightarrow \textbf{id}$
$L \rightarrow {*}R$
$R \rightarrow L$





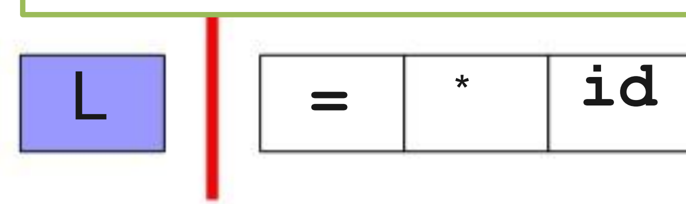
A Lack of Context of SLR(1)

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



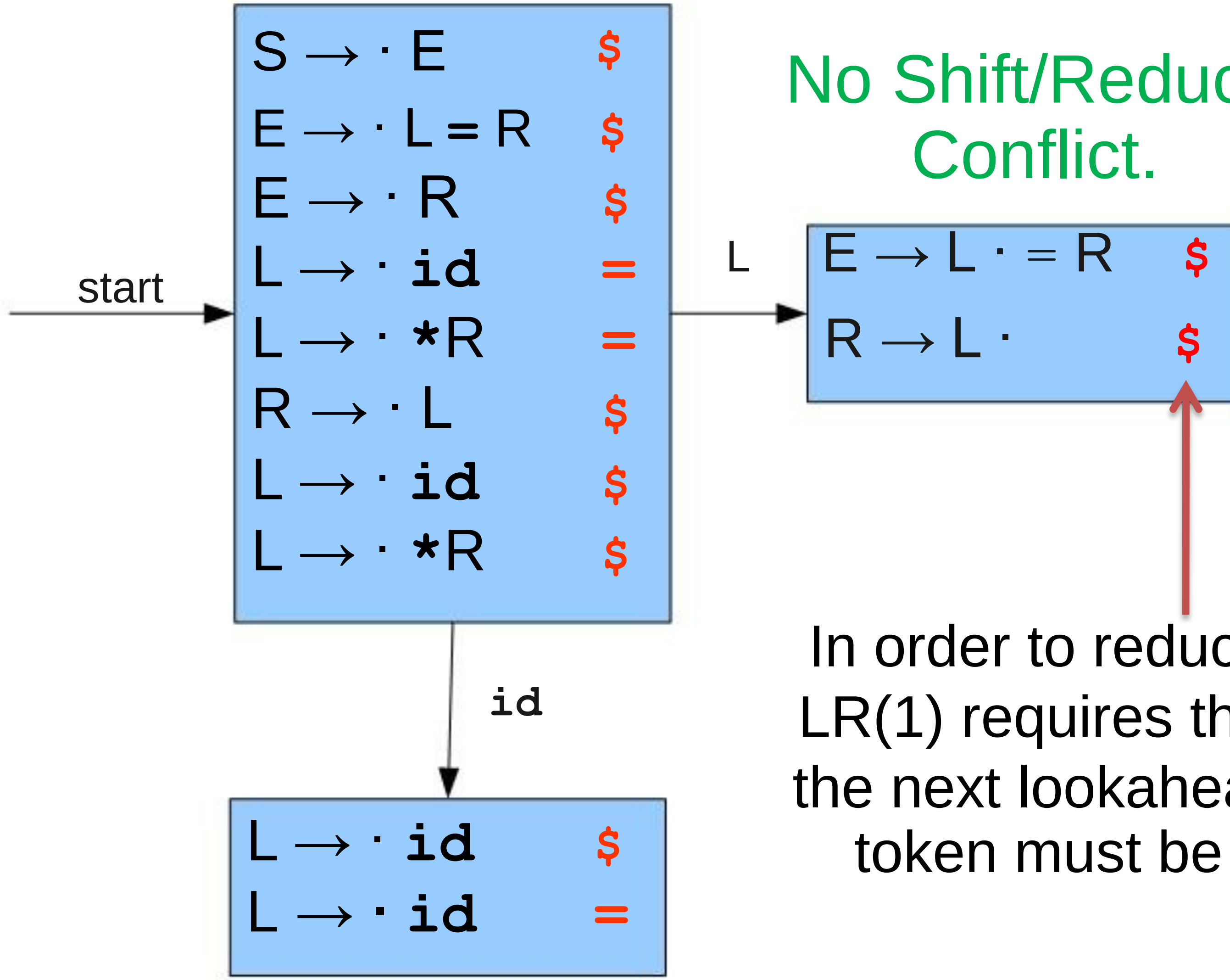
Shift/Reduce Conflict !

Follow(R) = Follow(E) U Follow(L)
= {$} U {=}



Recall LR(1) States

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



No Shift/Reduce Conflict.

In order to reduce, LR(1) requires that the next lookahead token must be $.

LR(1) and SLR(1)

- **SLR(1)** is weak because its lookahead information is **not precise**.
- **LR(1)** is impractical because its lookahead information makes the automaton **too big**.
- Can we retain the LR(1) automaton's **lookahead** information without all its states? In other words, can we combine states in LR(1) ?

Review of LR(1)

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

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

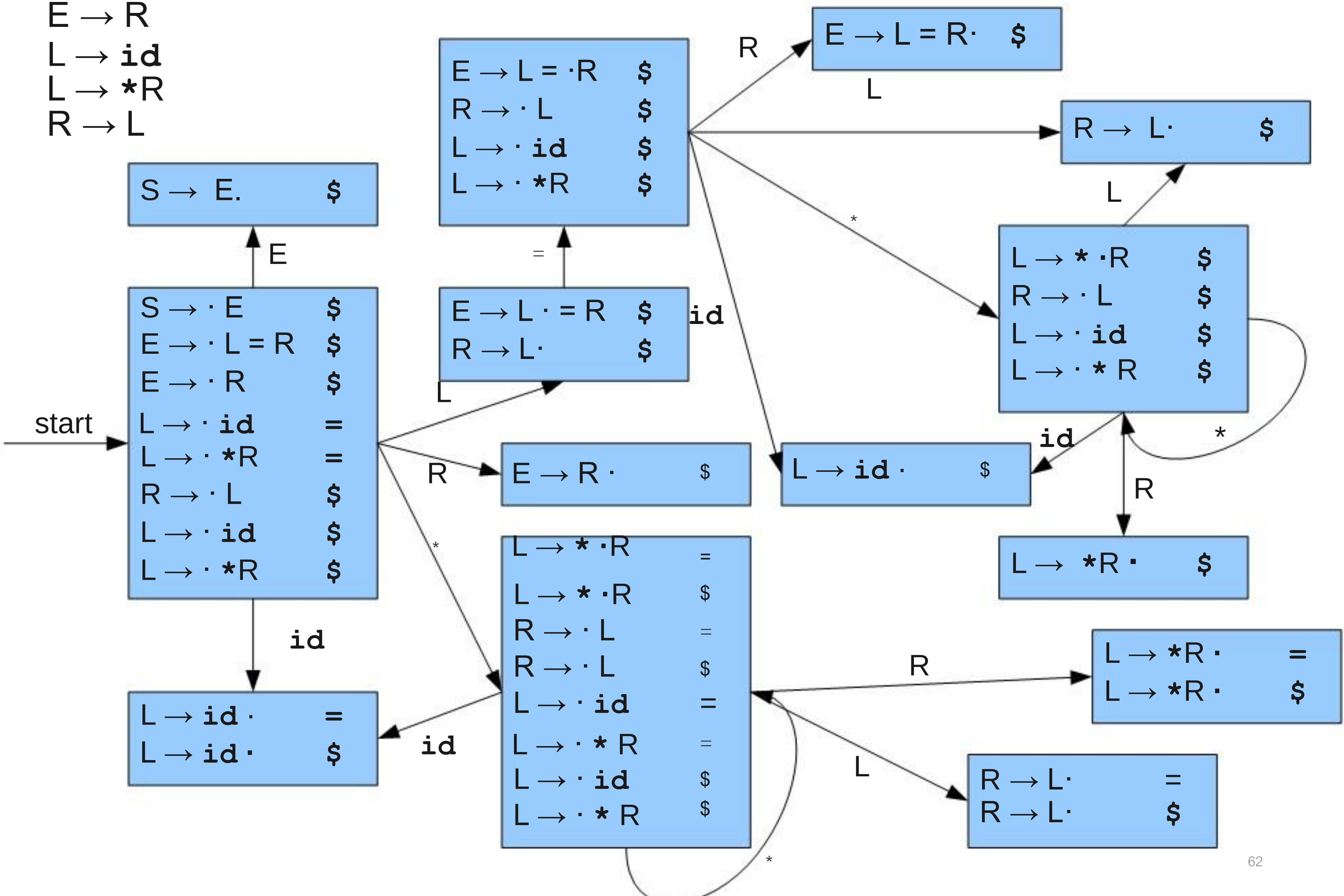
$T \to (\cdot E)$	)
$E \to \cdot E + T$	)
$E \to \cdot T$	)
$T \to \cdot$**int**	)
$T \to \cdot(E)$	)

A Surprisingly Powerful Idea

- 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**(1)
  - **Lookahead(1) + LR(0)**
- LALR(1) has almost the same size as LR(0) automaton.

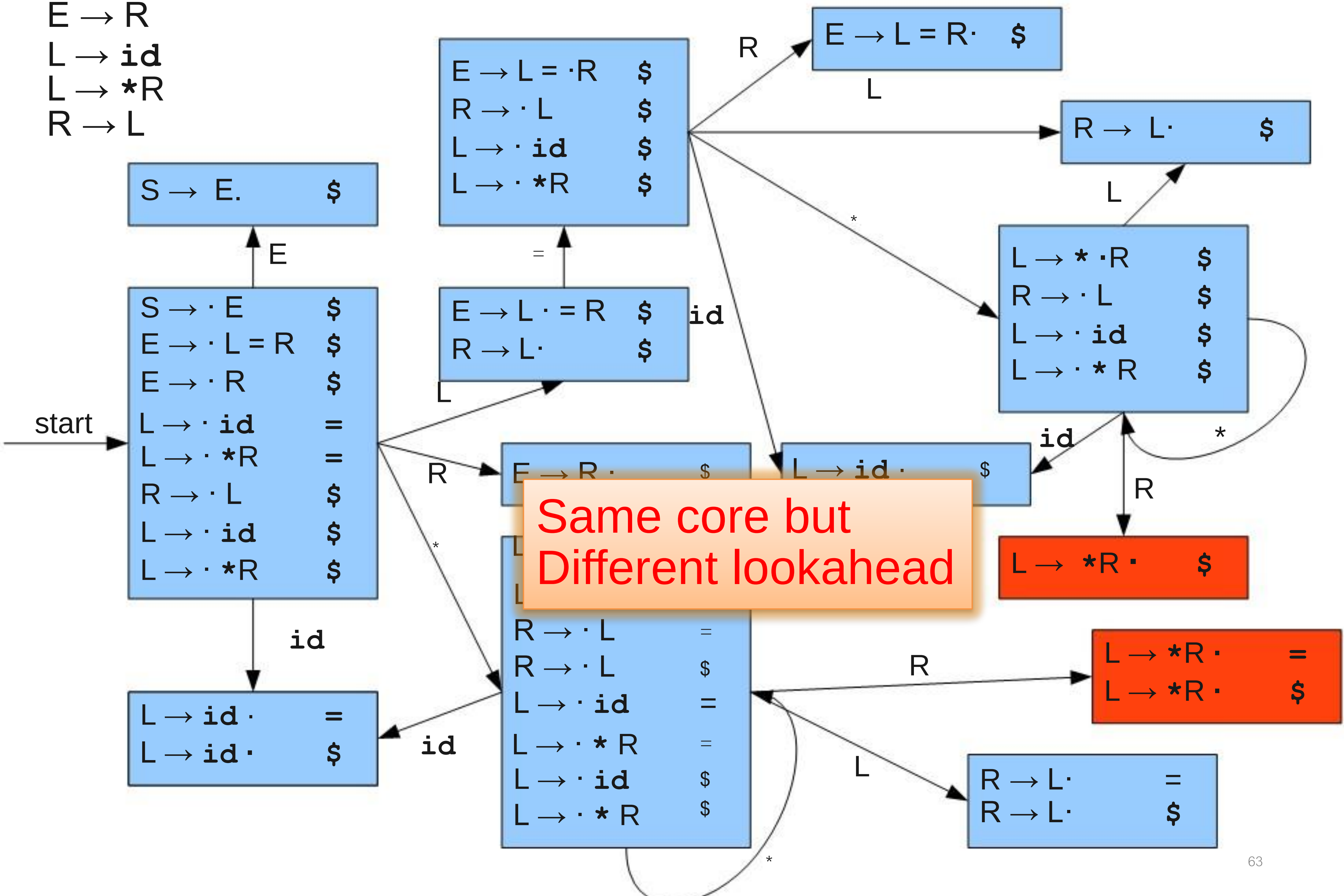
From LR(1) to LALR(1)

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



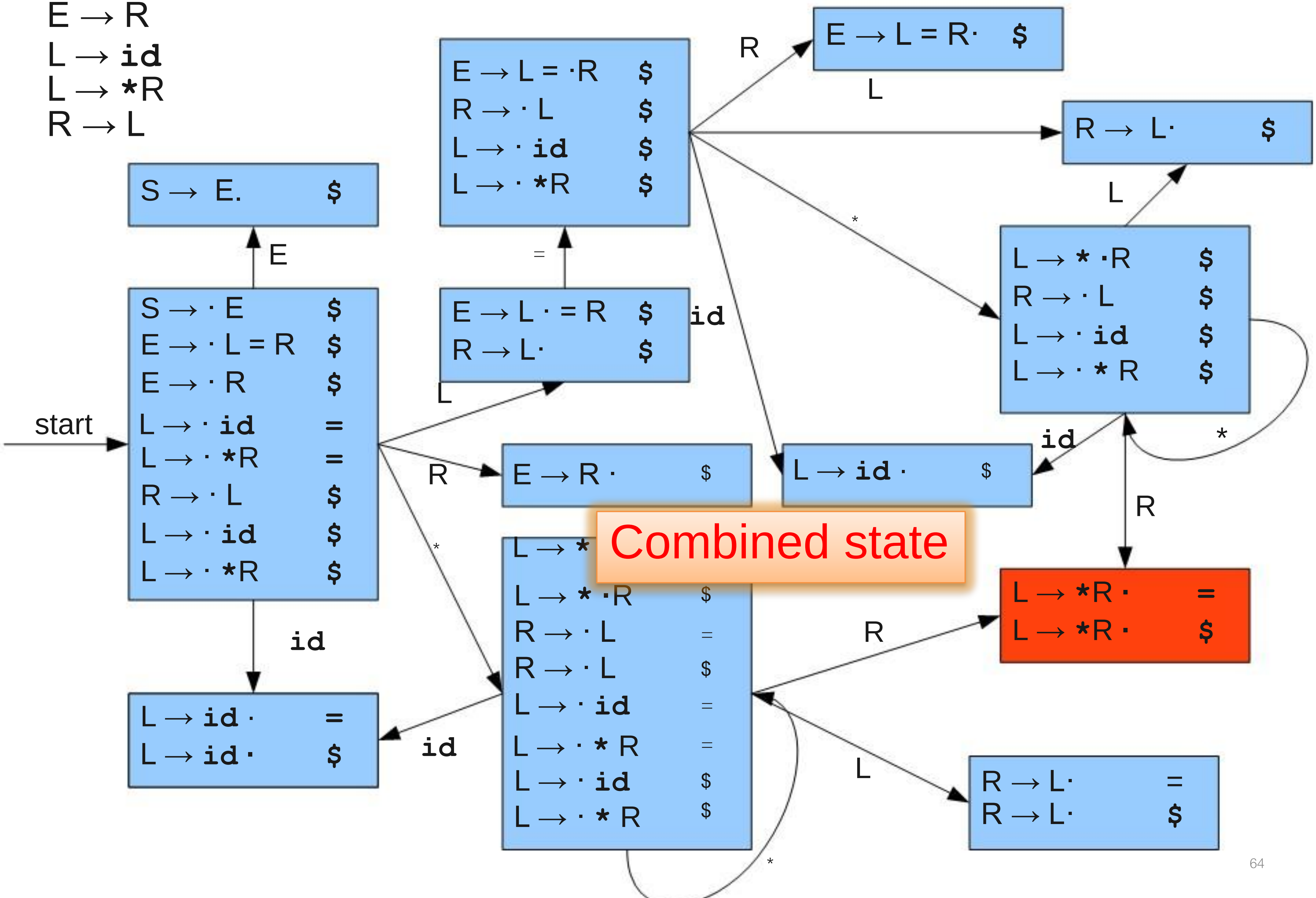
$S \to E$
$E \to L = R$
$E \to R$
$L \to \mathbf{id}$
$L \to *R$
$R \to L$

From LR(1) to LALR(1)



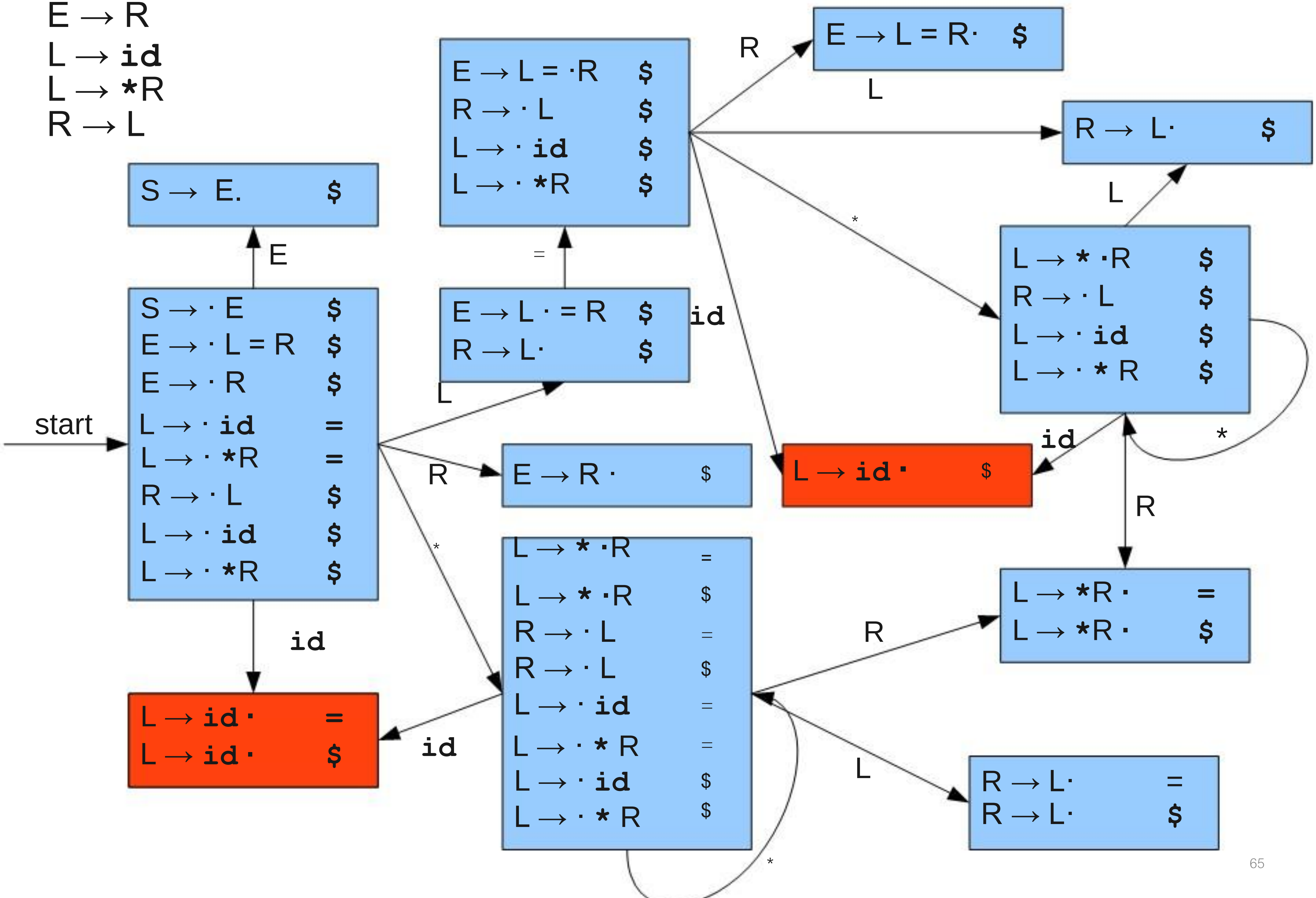
From LR(1) to LALR(1)

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



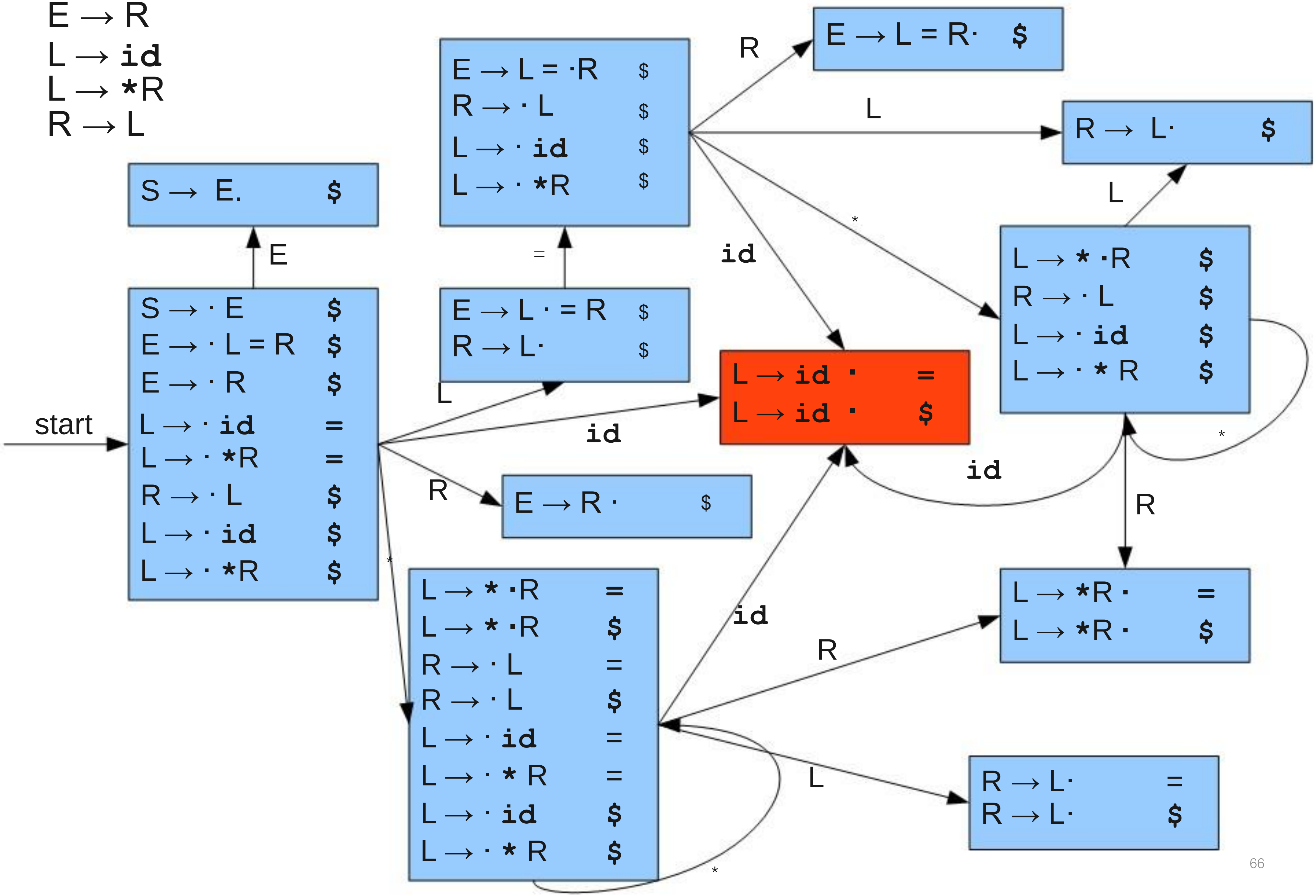
From LR(1) to LALR(1)

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



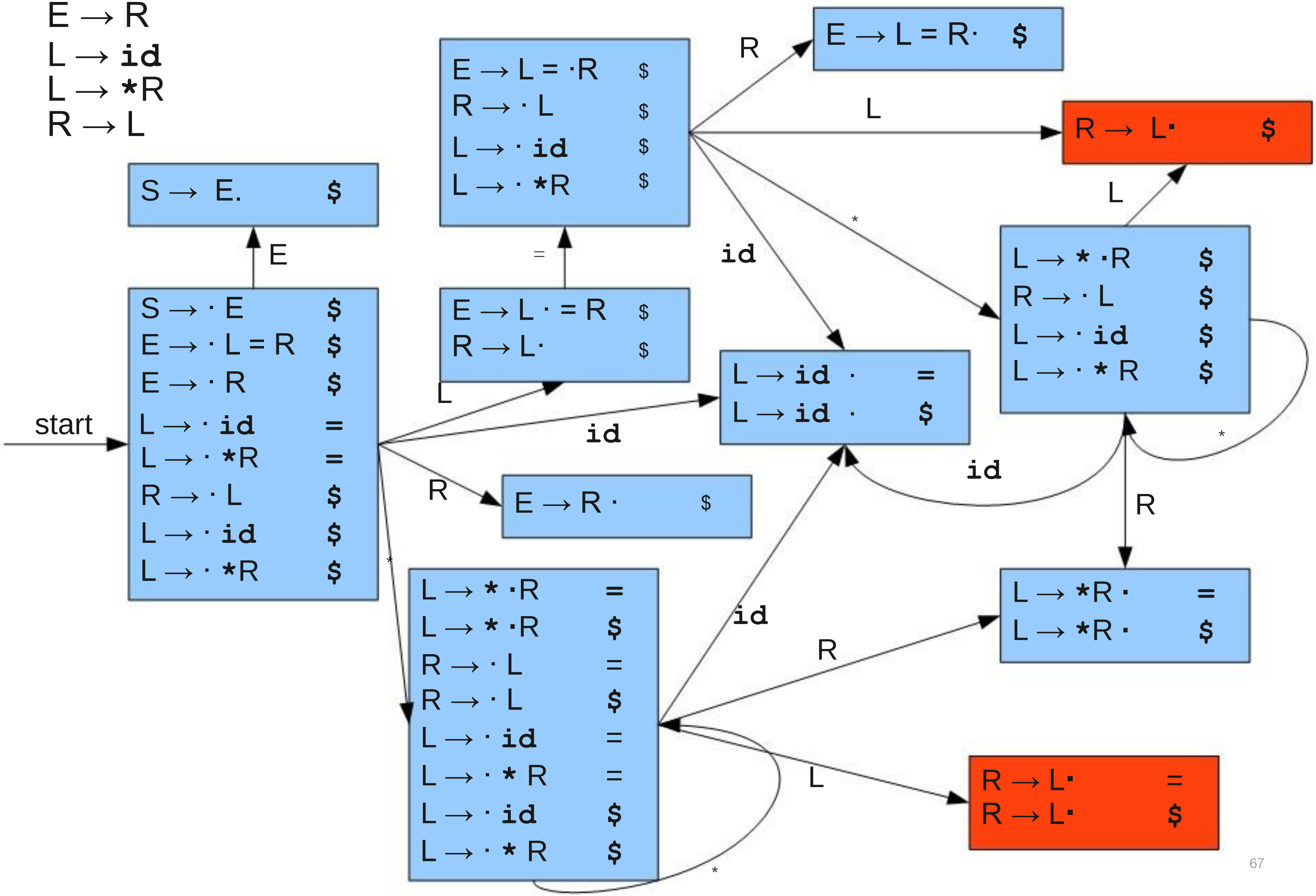
$S \to E$
$E \to L = R$
$E \to R$
$L \to$ **id**
$L \to$ ***R**
$R \to L$

From LR(1) to LALR(1)



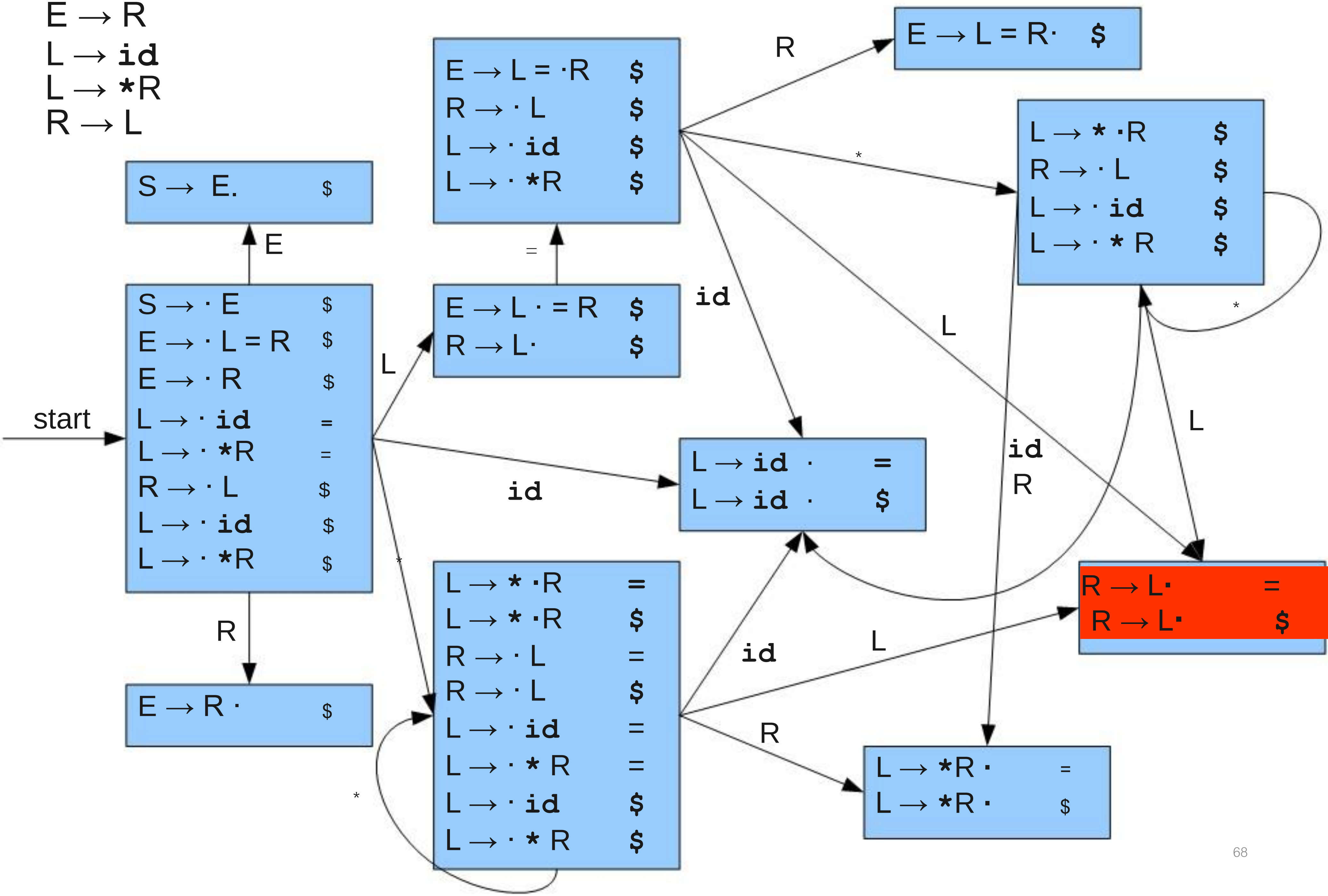
$S \rightarrow E$
$E \rightarrow L = R$
$E \rightarrow R$
$L \rightarrow \mathbf{id}$
$L \rightarrow {*}R$
$R \rightarrow L$

From LR(1) to LALR(1)



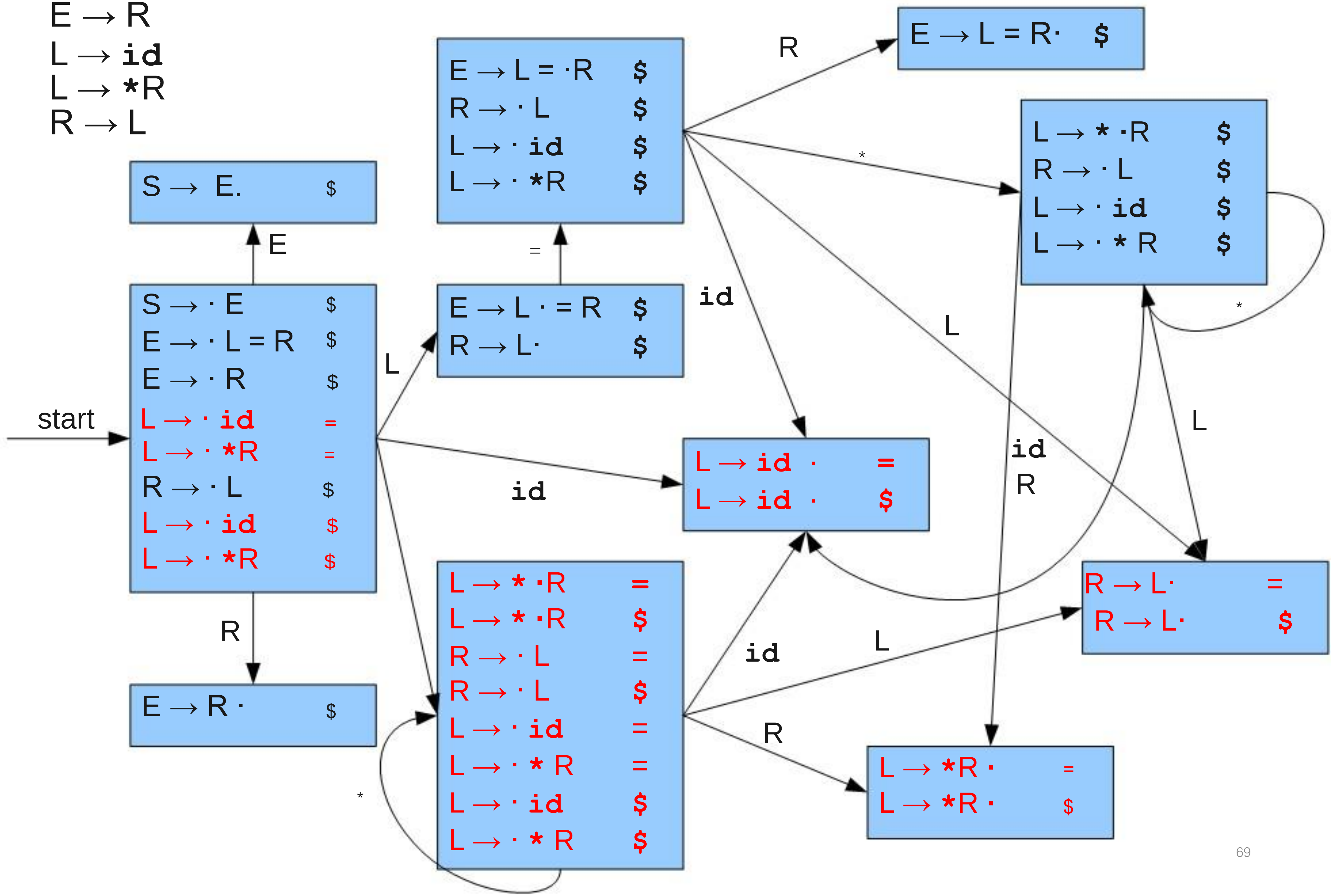
From LR(1) to LALR(1)

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



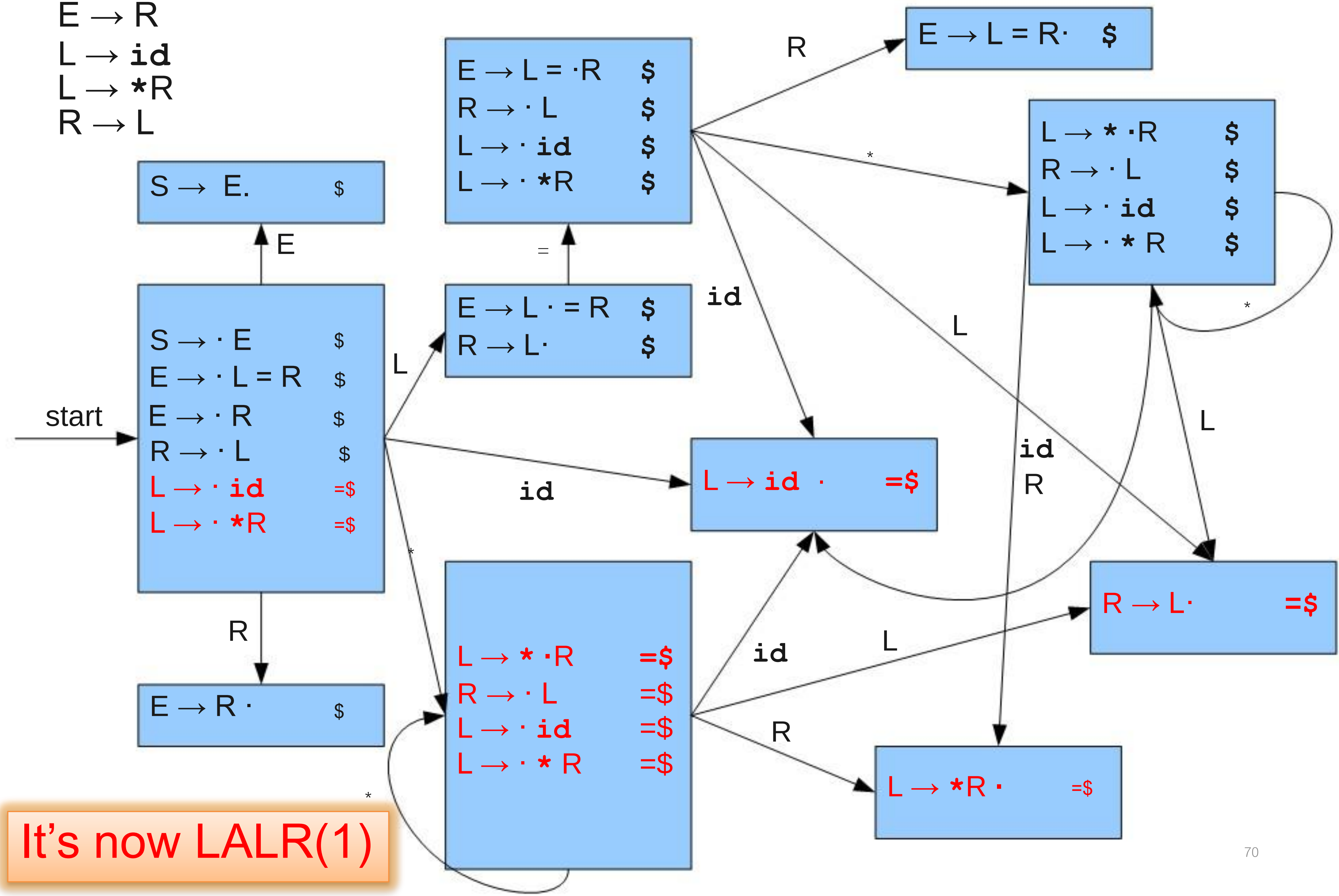
From LR(1) to LALR(1)

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



From LR(1) to LALR(1)

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

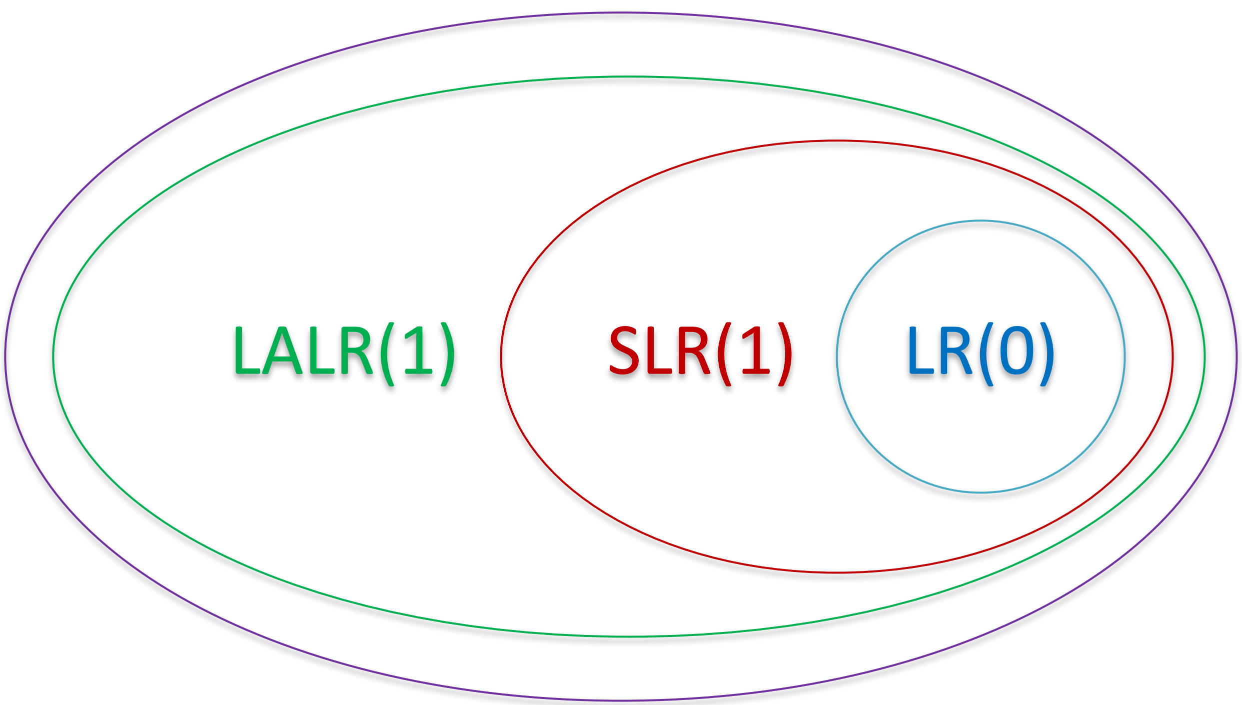


It’s now LALR(1)

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).

LR(1)



Next Time

- More intelligent lookaheads: LALR(1)