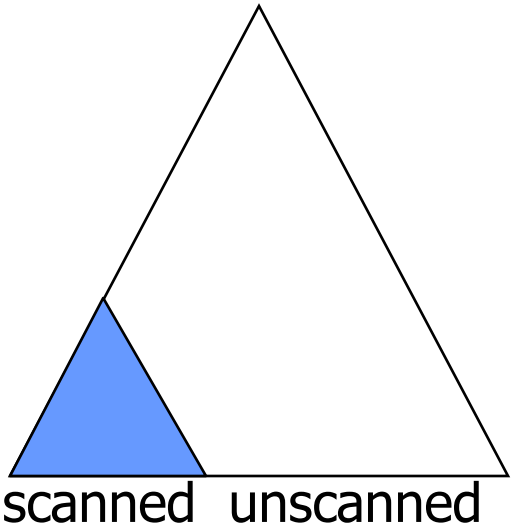
Bottom-up parsers

- Bottom-up parsers use the entire right-hand side of the production
- LR(k):
  - Left-to-right parse,
  - Rightmost derivation (in reverse),
  - k look ahead tokens



A Rightmost Derivation

1 S := Exp
2 Exp := Exp + Term
3 Exp := Exp - Term
4 Exp := Term
5 Term := Term * Factor
6 Term := Term / Factor
7 Term := Factor
8 Factor := **id**
9 Factor := **int**

input: 2+3*x

S

by 1 ⇒ Exp

by 2 ⇒ Exp + Term

by 5 ⇒ Exp + Term * Factor

by 8 ⇒ Exp + Term * $\mathbf{id}_x$

by 7 ⇒ Exp + Factor * $\mathbf{id}_x$

by 9 ⇒ Exp + $\mathbf{int}_3$ * $\mathbf{id}_x$

by 4 ⇒ Term + $\mathbf{int}_3$ * $\mathbf{id}_x$

by 7 ⇒ Factor + $\mathbf{int}_3$ * $\mathbf{id}_x$

by 9 ⇒ $\mathbf{int}_2$ + $\mathbf{int}_3$ * $\mathbf{id}_x$

A Rightmost Derivation In Reverse

int$_2$ + **int**$_3$ * **id**$_x$

Factor + **int**$_3$ * **id**$_x$

Term + **int** * **id**

Exp + [illegible]

Exp + Factor * **id**$_x$

Exp + Term * **id**$_x$

Exp + Term * Factor

Exp + Term

Exp

S

Lets keep track of where we are in the input.

A Rightmost Derivation In Reverse

int$_2$ + **int**$_3$ * **id**$_x$

Factor + **int**$_3$ * **id**$_x$

Term + **int**$_3$ * **id**$_x$

Exp + **int**$_3$ * **id**$_x$

Exp + Factor * **id**$_x$

Exp + Term * **id**$_x$

Exp + Term * Factor

Exp + Term

Exp

S

int$_2$ ● + **int**$_3$ * **id**$_x$

Factor ● + **int**$_3$ * **id**$_x$

Term ● + **int**$_3$ * **id**$_x$

Exp + **int**$_3$ ● * **id**$_x$

Exp + Factor ● * **id**$_x$

Exp + Term * **id**$_x$ ●

Exp + Term * Factor ●

Exp + Term ●

Exp ●

S ●

A Rightmost Derivation In Reverse

int$_2$ + **int**$_3$ * **id**$_x$

Factor + **int**$_3$ * **id**$_x$

Term + **int**$_3$ * **id**$_x$

Exp + **int**$_3$ * **id**$_x$

Exp + Factor * **id**$_x$

Exp + Term * **id**

Exp + Term *

Exp + Term

Exp

S

int$_2$ ● + **int**$_3$ * **id**$_x$

Factor ● + **int**$_3$ * **id**$_x$

Term ● + **int**$_3$ * **id**$_x$

Exp + **int**$_3$ ● * **id**$_x$

Exp + Factor ● * **id**$_x$

Exp + Term * **id**$_x$ ●

Factor ●

Exp ●

S ●

Lets format this differently,
<prefix of sentential form> input

A Rightmost Derivation In Reverse

	$\mathtt{int}_2 + \mathtt{int}_3 * \mathtt{id}_x \$$	shift 2
$\mathtt{int}_2$	$+ \mathtt{int}_3 * \mathtt{id}_x \$$	reduce by F → int
Factor		
Term		
Exp		
Exp +		
Exp + $\mathtt{int}_3$	$* \mathtt{id}_x \$$	
Exp + Factor	$* \mathtt{id}_x \$$	
Exp + Term	$* \mathtt{id}_x \$$	
Exp + Term *	$\mathtt{id}_x \$$	
Exp + Term * $\mathtt{id}_x$	$\$$	
Exp + Term * Factor	$\$$	
Exp + Term	$\$$	
Exp	$\$$	
S	$\$$	

When we reduce by a production: A → β,
β is on right side of sentential form.

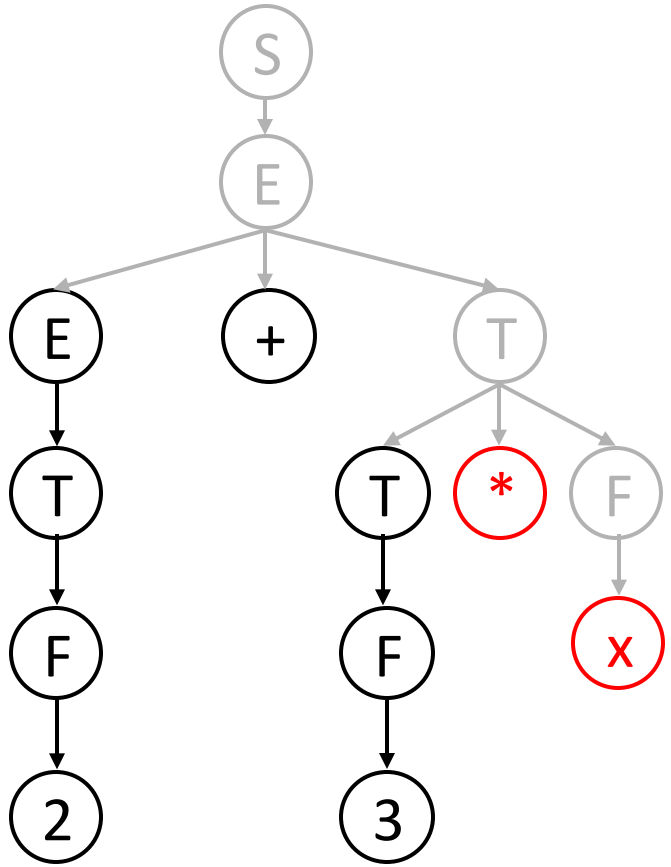
E.g., here β is 'int' and production is F → int

A Rightmost Derivation In Reverse

	$\mathbf{int}_2 + \mathbf{int}_3 * \mathbf{id}_x \$$
$\mathbf{int}_2$	$+ \mathbf{int}_3 * \mathbf{id}_x \$$
Factor	$+ \mathbf{int}_3 * \mathbf{id}_x \$$
Term	$+ \mathbf{int}_3 * \mathbf{id}_x \$$
Exp	$+ \mathbf{int}_3 * \mathbf{id}_x \$$
Exp +	$\mathbf{int}_3 * \mathbf{id}_x \$$
Exp + $\mathbf{int}_3$	$* \mathbf{id}_x \$$
Exp + Factor	$* \mathbf{id}_x \$$
Exp + Term	$* \mathbf{id}_x \$$

top of stack does not have a handle, so must shift.

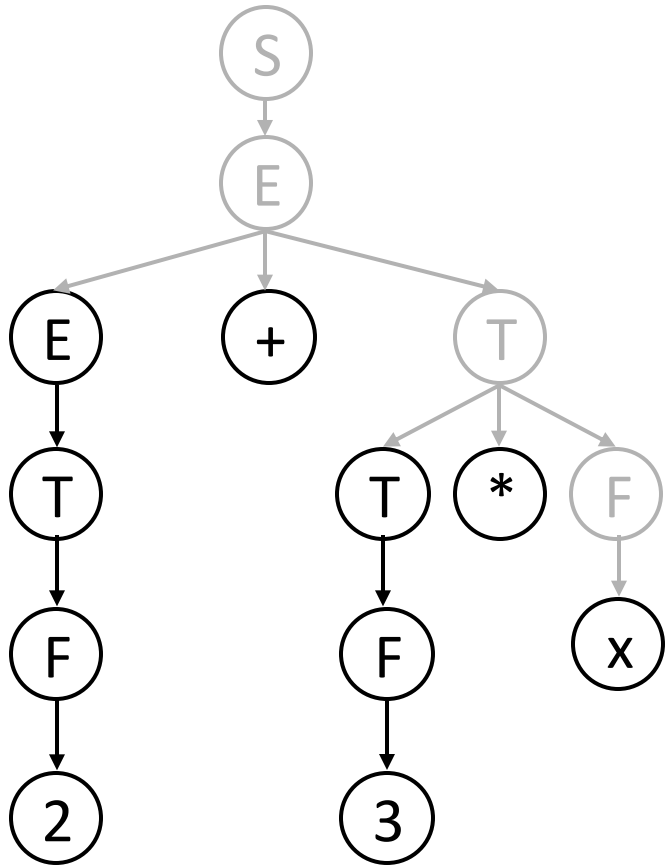
```
1   S := E
2   E := E + T
3   E := E - T
4   E := T
5   T := T * F
6   T := T / F
7   T := F
8   F := id
9   F := int
```



A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $
int$_2$	+ **int**$_3$ * **id**$_x$ $
Factor	+ **int**$_3$ * **id**$_x$ $
Term	+ **int**$_3$ * **id**$_x$ $
Exp	+ **int**$_3$ * **id**$_x$ $
Exp +	**int**$_3$ * **id**$_x$ $
Exp + **int**$_3$	* **id**$_x$ $
Exp + Factor	* **id**$_x$ $
Exp + Term	* **id**$_x$ $
Exp + Term *	**id**$_x$ $
Exp + Term * **id**$_x$	$
Exp + Term * Factor	$
Exp + Term	$
Exp	$
S	$

Now, x is a handle.

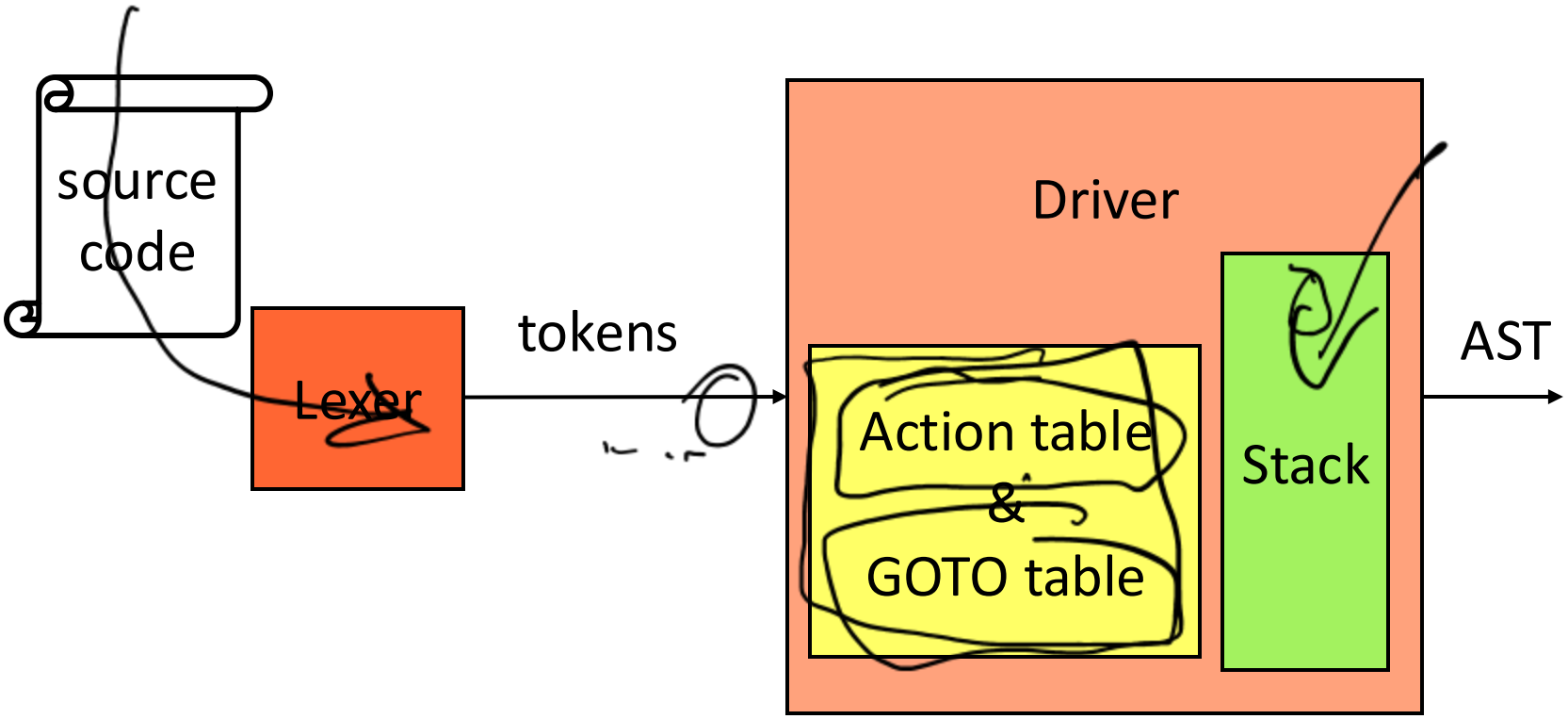


A Shift-Reduce Parser

- Stack holds the viable prefixes.
- input stream holds remaining source
- Four actions:
  - shift: push token from input stream onto stack
  - reduce: right-end of a handle (β of $A \rightarrow \beta$) is at top of stack, pop handle (β), push A
  - accept: success
  - error: syntax error discovered

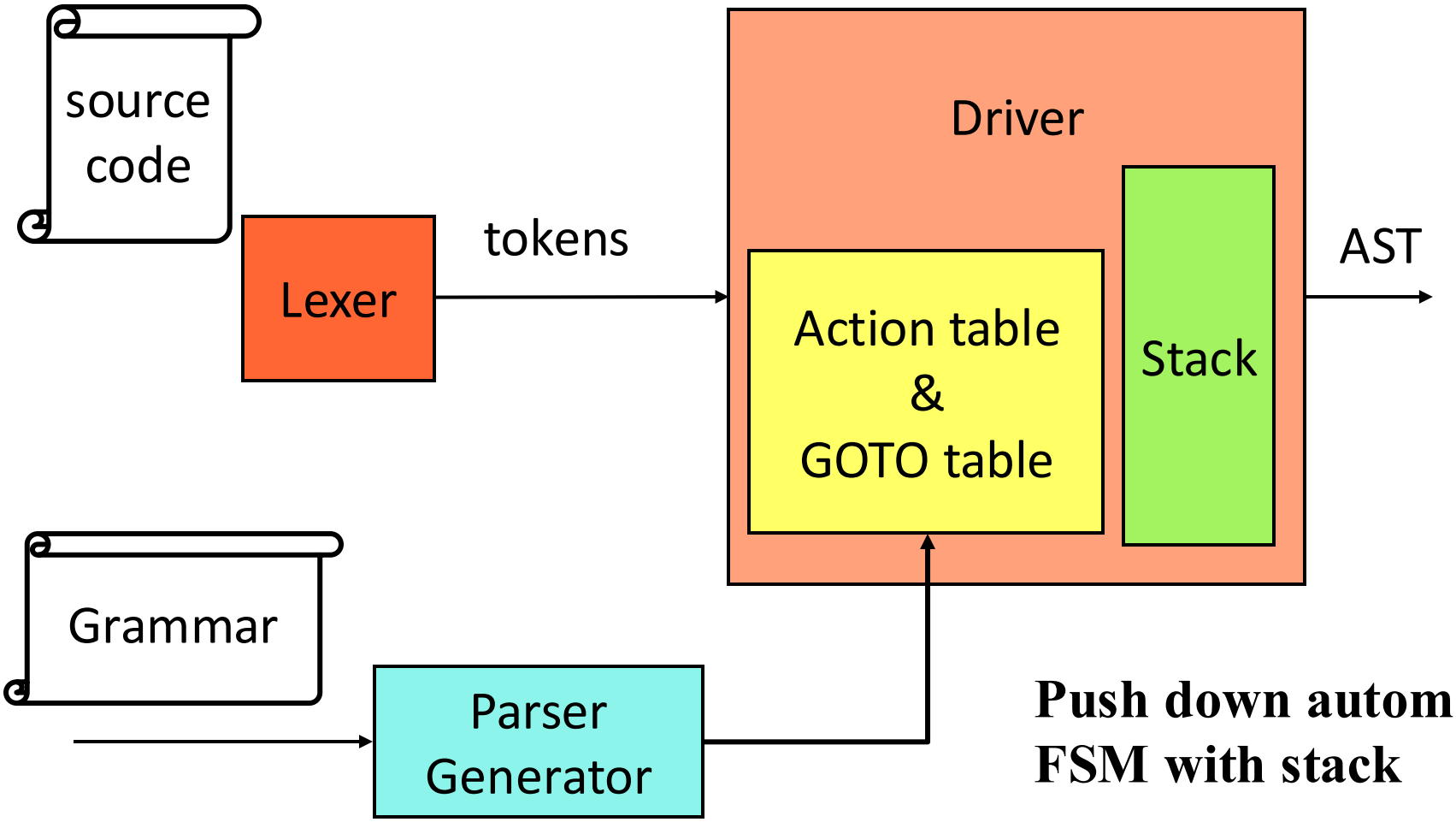
Key is recognizing handles efficiently

Table-driven LR(k) parsers



Push down automata:
FSM with stack

Table-driven LR(k) parsers



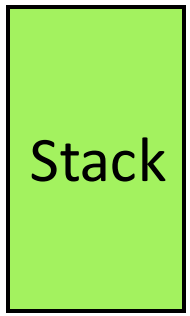
Push down automata: FSM with stack

Parser Loop

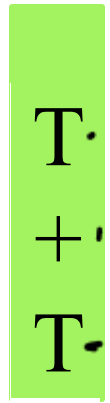
Driver

- Same code regardless of grammar
  - only tables change
- (Very) General Algorithm:
  - Based on table contents, top of stack, and current input character either
    - **shift**: pushes onto stack, reads next token
    - **reduce**: manipulate stack to simplify representation of already scanned input
    - **accept**: successfully scanned entire input
    - **error**: input not in language

Stack



- Represents the scanned input
- Contents?
  - Reduced nonterminals not enough
  - Must store previously seen *states*
    - the context of the current position
  - In fact, nonterminals unnecessary
    - include for readability



x + y• + z

Parser Tables

Action table
&
GOTO table

Action table

- given state *s* and **terminal** *a* tells parser loop what action (shift, reduce, accept, reject) to perform

Goto table

- used when performing reduction; given a state *s* and **nonterminal** *X* says what state to transition to

Parser Tables

Action table
&
GOTO table

s*N* push state *N* onto stack
r*R* reduce by rule *R*
g*N* goto state *N*
a accept
error

state	action			goto	
	ident	+	$	E	T
0	s3			**g1**	**g2**
1			**a**		
2		s4	r2		
3		r3	r3		
4	s3			**g5**	**g2**
5			r1		

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

Parser Loop Revisited

Driver

```
while(true)
   s = state on top of stack
   a = current input token
   if(action[s][a] == sN)                    shift
      push N
      read next input token
   else if(action[s][a] == rR)               reduce
      pop rhs of rule R from stack
      X = lhs of rule R
      N = state on top of stack
      push goto[N][X]
   else if(action[s][a] == a)                accept
      return success
   else                                      error
      return failure
```

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

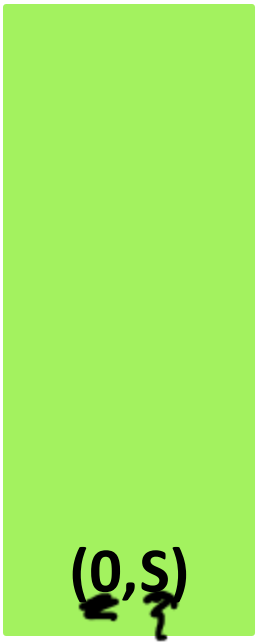
2 E → T

3 T → *identifier*

Current input token = **x**

State on top of the stack = **0**

x + y$



Stack

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **+**

State on top of the stack = **3**

x + y$

(3,x)

(0,S)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **+**

State on top of the stack = **3**

(3,x)

(0,S)

x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

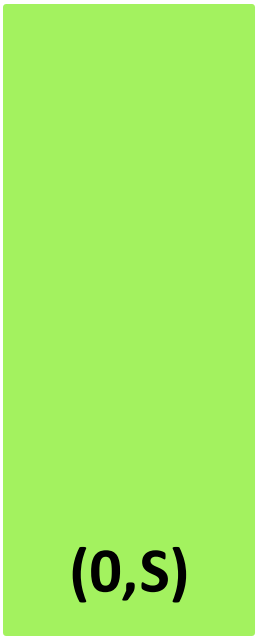
1 E → T + E

2 E → T

3 T → *identifier*

Current input token = +

State on top of the stack = 3



x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

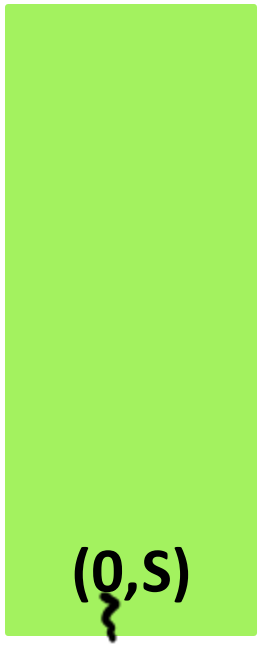
2 E → T

3 T → *identifier*

Current input token = +

State on top of the stack = 0

x + y$



Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **+**

State on top of the stack = **2**

x + y$

(2,T)

(0,S)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

Current input token = **+**
State on top of the stack = **2**

x + y$

(2,T)
(0,S)

Example

	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **y**

State on top of the stack = **4**

x + y$

(4,+)

(2,T)

(0,S)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **y**

State on top of the stack = **4**

x + y$

(4,+)
(2,T)
(0,S)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = $

State on top of the stack = 3

(3,y)

(4,+)

(2,T)

(0,S)

x + y$

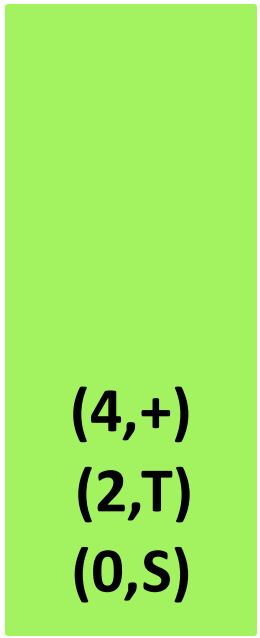
Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

Current input token = **$**
State on top of the stack = **3**

x + y$



(?,T)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

Current input token = **$**
State on top of the stack = **2**

x + y$

(2,T)
(4,+)
(2,T)
(0,S)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **$**

State on top of the stack = **2**

x + y$

(2,T)
(4,+)
(2,T)
(0,S)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = $

State on top of the stack = 2

x + y$

(?,E)

(4,+)

(2,T)

(0,S)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

Current input token = $
State on top of the stack = 5

(5,E)
(4,+)
(2,T)
(0,S)

x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **$**

State on top of the stack = **5**

x + y$

(5,E)
(4,+)
(2,T)
(0,S)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

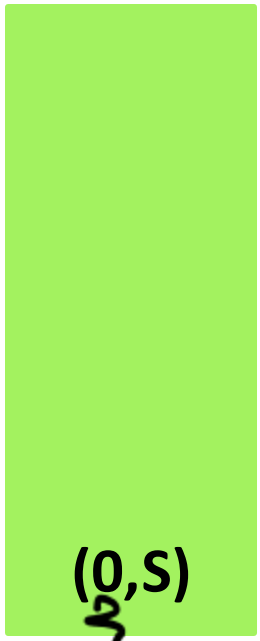
1 E → T + E

2 E → T

3 T → *identifier*

Current input token = $

State on top of the stack = 5



x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **$**

State on top of the stack = **1**

x + y$

(1,E)

(0,S)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

Accept!

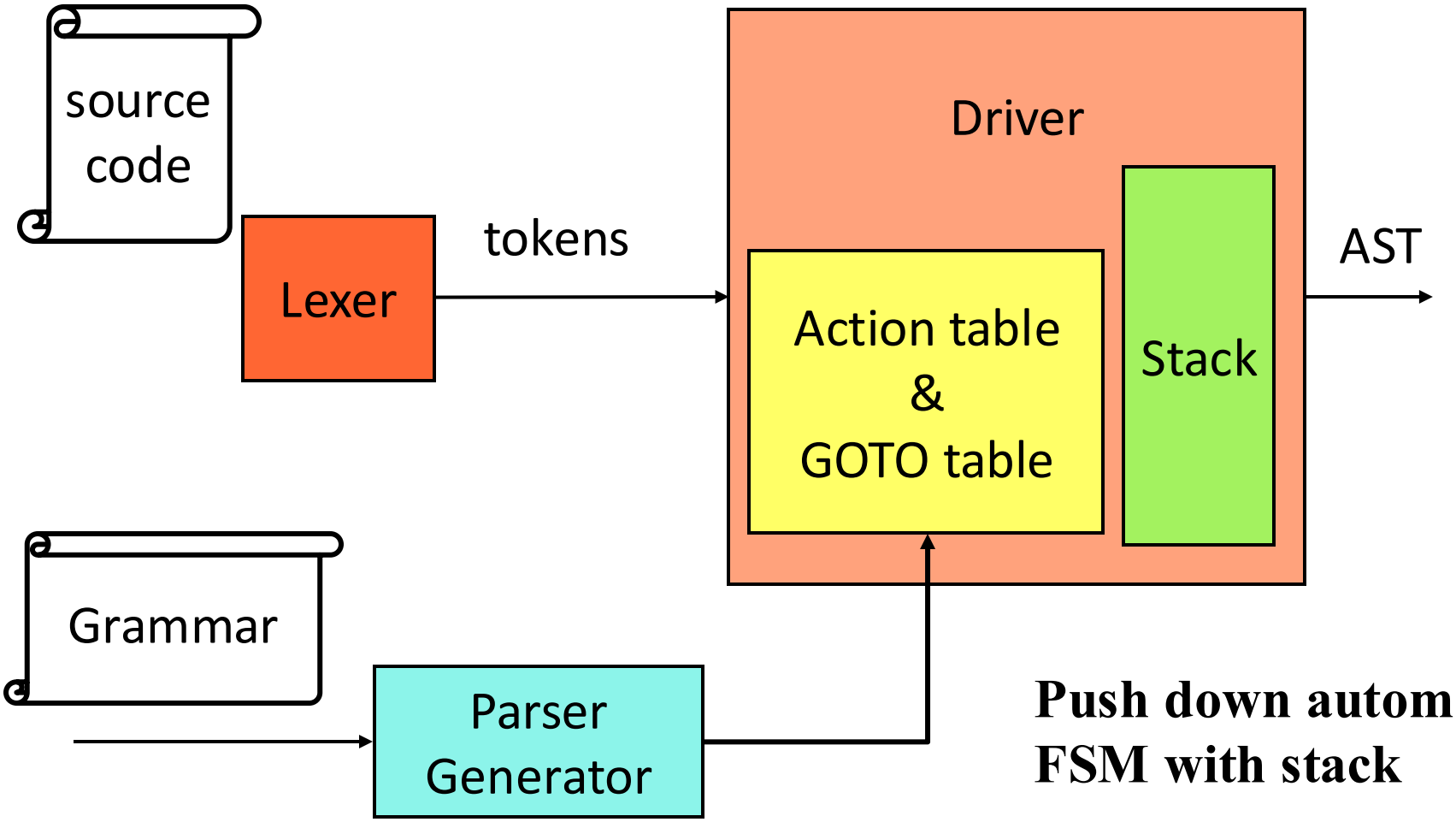
0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

Current input token = $
State on top of the stack = 1

x + y$

(1,E)
(0,S)

Table-driven LR(k) parsers



Push down automata: FSM with stack

The parser generator

Parser Generator

- Finds handles
- Creates the action and GOTO tables.
- Creates the states
  - Each state indicates how much of a handle we have seen
  - each state is a set of *items*

Items

- Items are used to identify handles.
- LR(k) items have the form:

  [production-with-dot, lookahead]
- For example, A → a X b has 4 LR(0) items
  - [A → • a X b]
  - [A → a • X b]
  - [A → a X • b]
  - [A → a X b •]

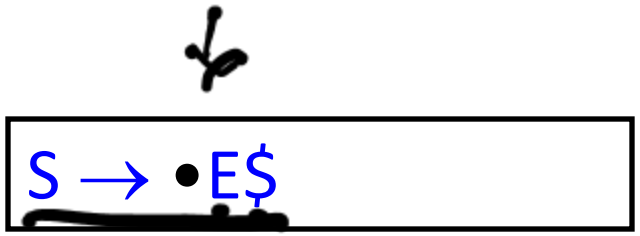
The • indicates how much of the handle we have recognized.

What LR(0) Items Mean

- [X $\rightarrow$ • α β γ]
  input is consistent with X $\rightarrow$ α β γ
- [X $\rightarrow$ α • β γ]
  input is consistent with X $\rightarrow$ α β γ and we have already recognized α
- [X $\rightarrow$ α β • γ]
  input is consistent with X $\rightarrow$ α β γ and we have already recognized α β
- [X $\rightarrow$ α β γ •]
  input is consistent with X $\rightarrow$ α β γ and we can reduce to X

Generating the States

- Start with start production.
- In this case, "S → E$"



0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

- Each state is consistent with what we have already shifted from the input and what is possible to reduce. So, what other items should be in this state?

Completing a state

- For each item in a state, add in all other consistent items.

$S \rightarrow \bullet E\$$
$E \rightarrow \bullet T + E$
$E \rightarrow \bullet T$
$T \rightarrow \bullet identifier$

0 $S \rightarrow E\$$
1 $E \rightarrow T + E$
2 $E \rightarrow T$
3 $T \rightarrow identifier$

- This is called, taking the closure of the state.

Closure*

```
closure(state)
   repeat
      foreach item A → a•Xb in state
         foreach production X → w
            state.add(X → •w)
   until state does not change
   return state
```

Intuitively:

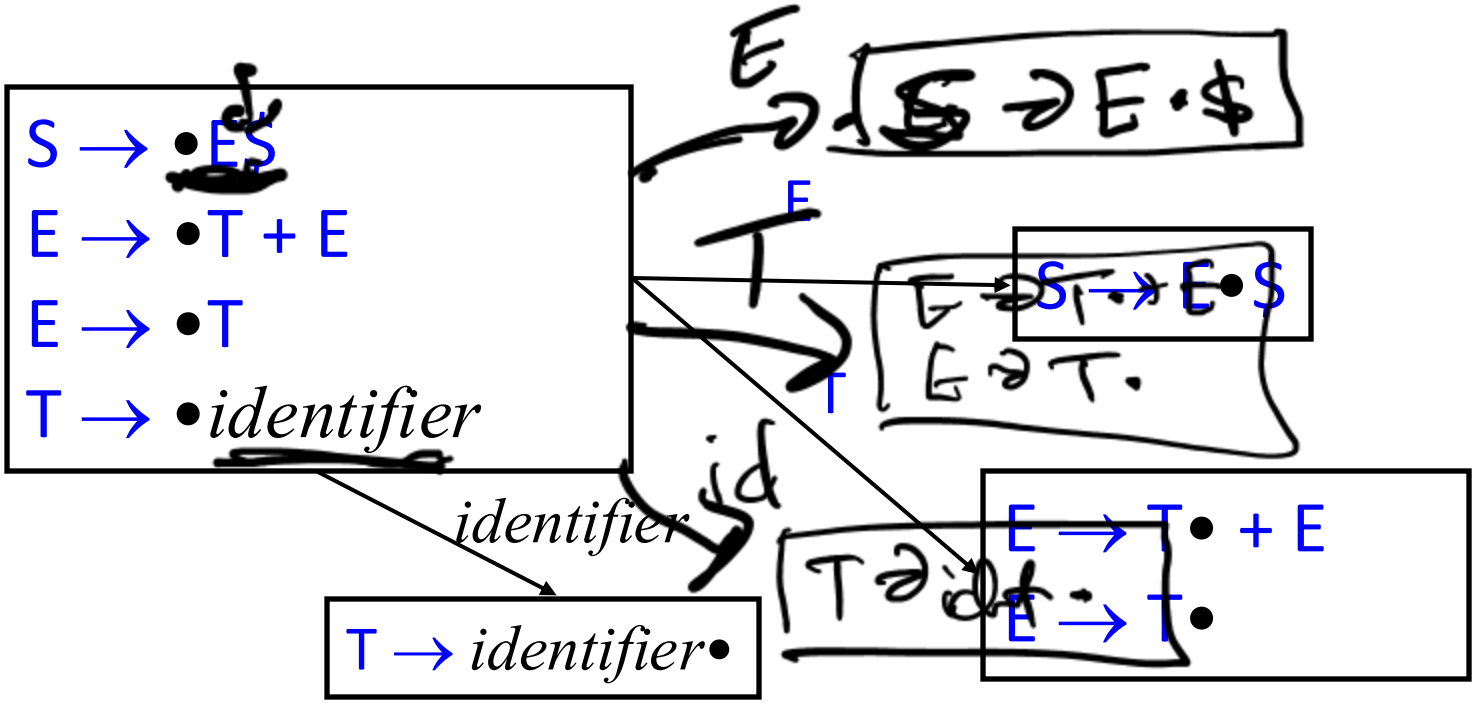
Given a set of items, add all production rules that could produce the nonterminal(s) at the current position in each item

*: for LR(0) items

What about the other states?

- How do we decide what the other states are?
- How do we decide what the transitions between states are?

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*



Next(state, sym)

- Next function determines what state to goto based on current state and symbol being recognized.
- For Non-terminal, this is used to determine the GOTO table.
- For terminal, this is used to determine the shift action.

Constructing states

```
initial_state = closure({start production})
state_set.add(initial_state)
state_queue.push(initial_state)

while(!state_queue.empty())
   s = state_queue.pop()
   foreach item A → a•Xb in s
      n = closure(next(s, X))
      if(!state_set.contains(n))
         state_set.add(n)
         state_queue.push(n)
```

A state is a set of LR(0) items

get "next" state

Closure*

closure({S → •E$}) =

S → •E$

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

*: for LR(0) items

Closure*

closure({S → •E$}) =

S → •E$
E → •T + E
E → •T
T → •*identifier*

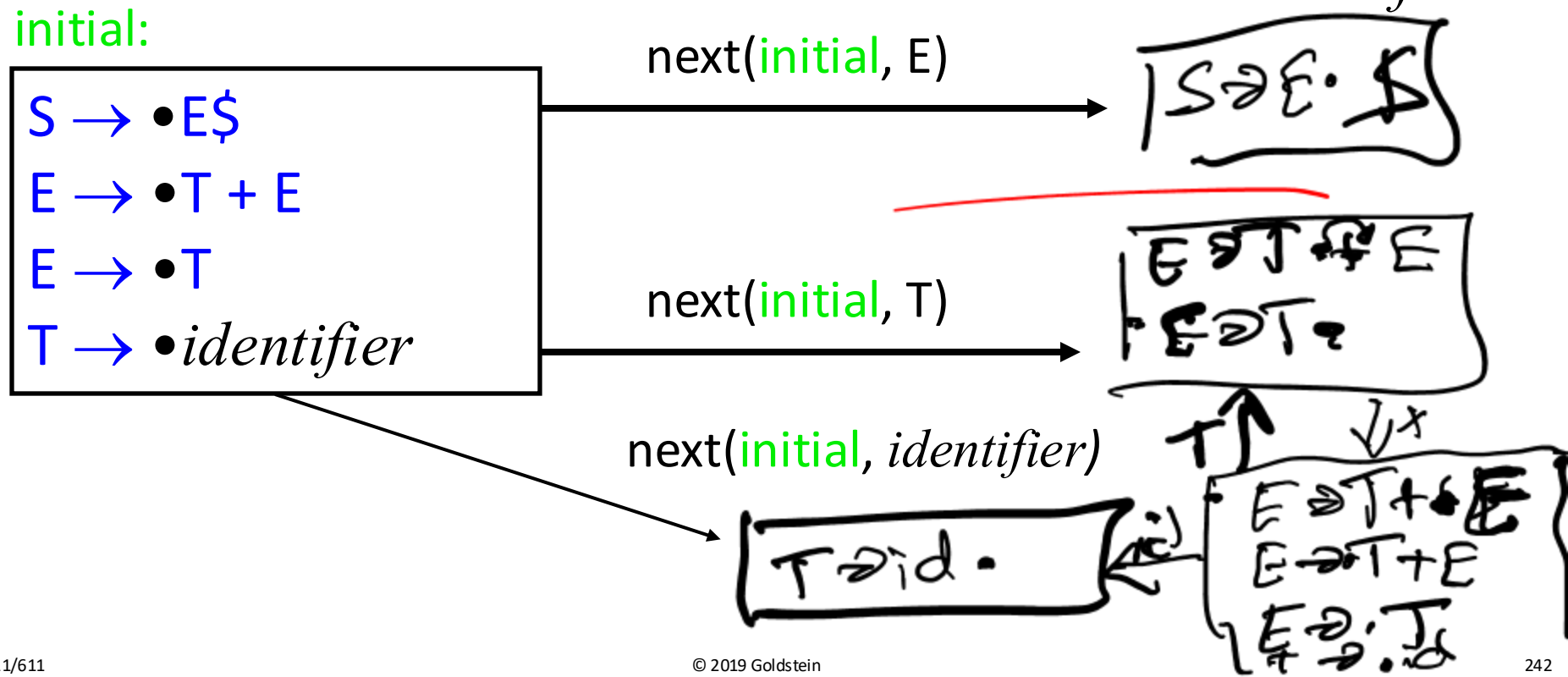
0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

*: for LR(0) items

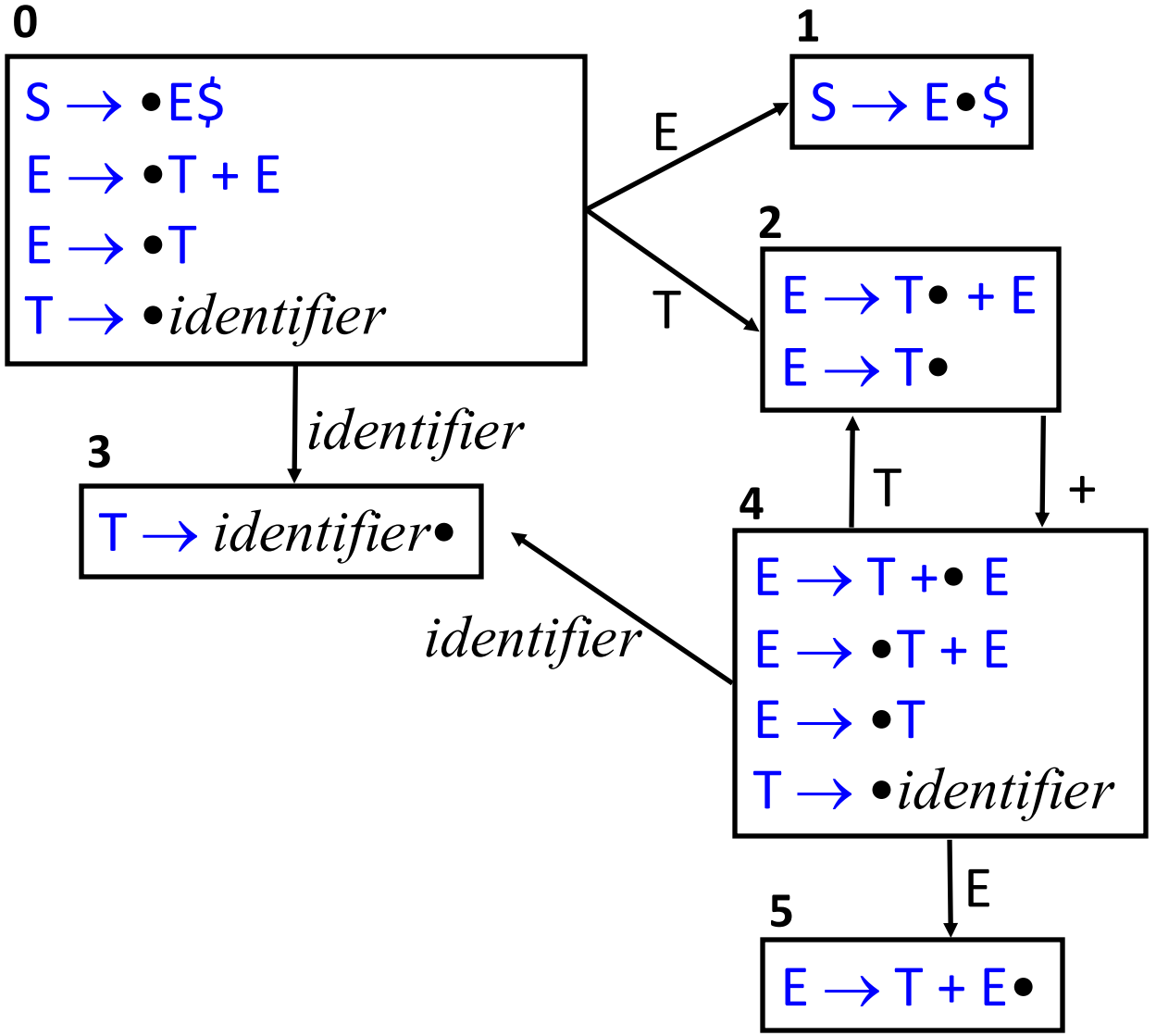
Next

```
next(state, X)
    ret = empty
    foreach item A → a•Xb in state
        ret.add(A → aX•b)
    return ret
```

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*



Example



0 S → E$

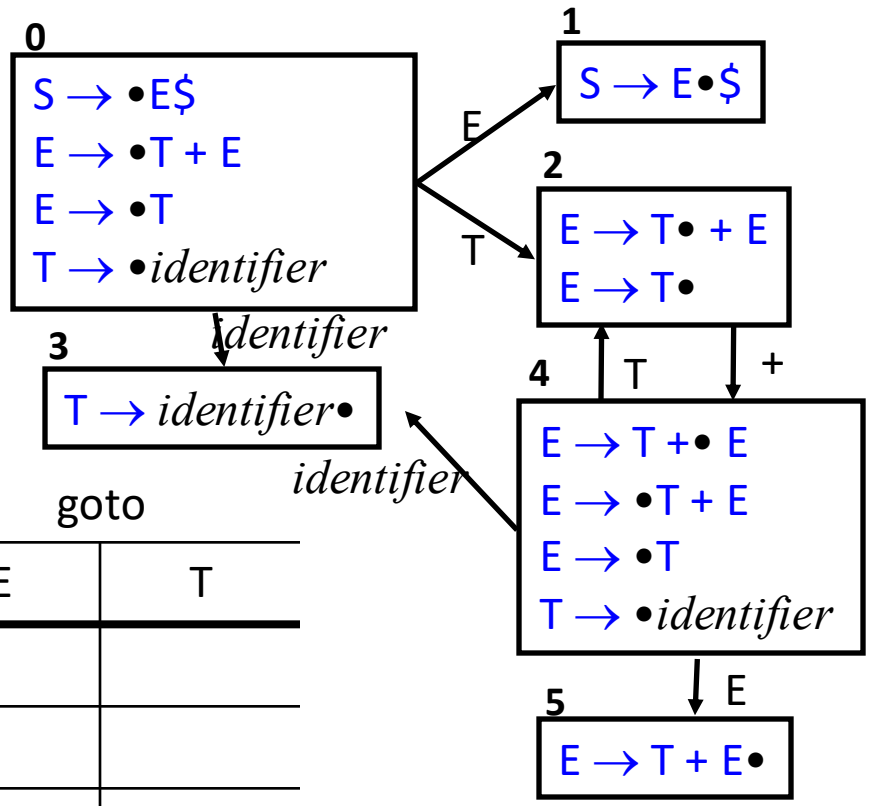
1 E → T + E

2 E → T

3 T → *identifier*

Parse Tables for LR(0) parser

What can we fill out?



	action			goto	
state	*ident*	+	$	E	T
0					
1					
2					
3					
4					
5					

0 S → E$

1 E → T + E

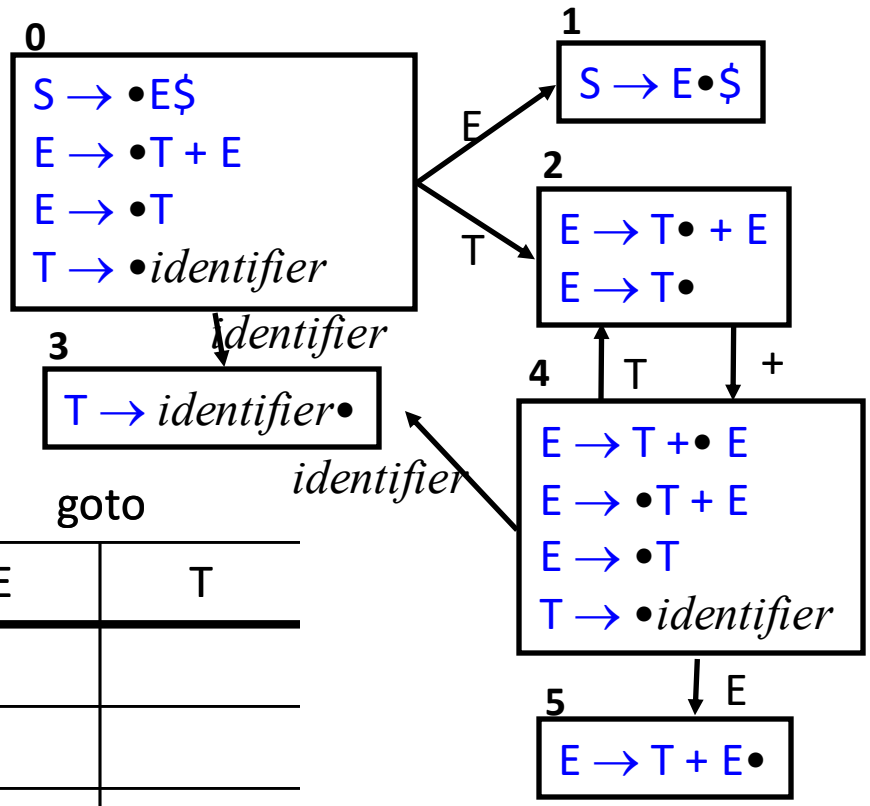
2 E → T

3 T → *identifier*

Parse Tables for LR(0) parser

shift

transition on terminal



state	action			goto	
	ident	+	$	E	T
0	s3				
1					
2		s4			
3					
4	s3				
5					

0 S → E$

1 E → T + E

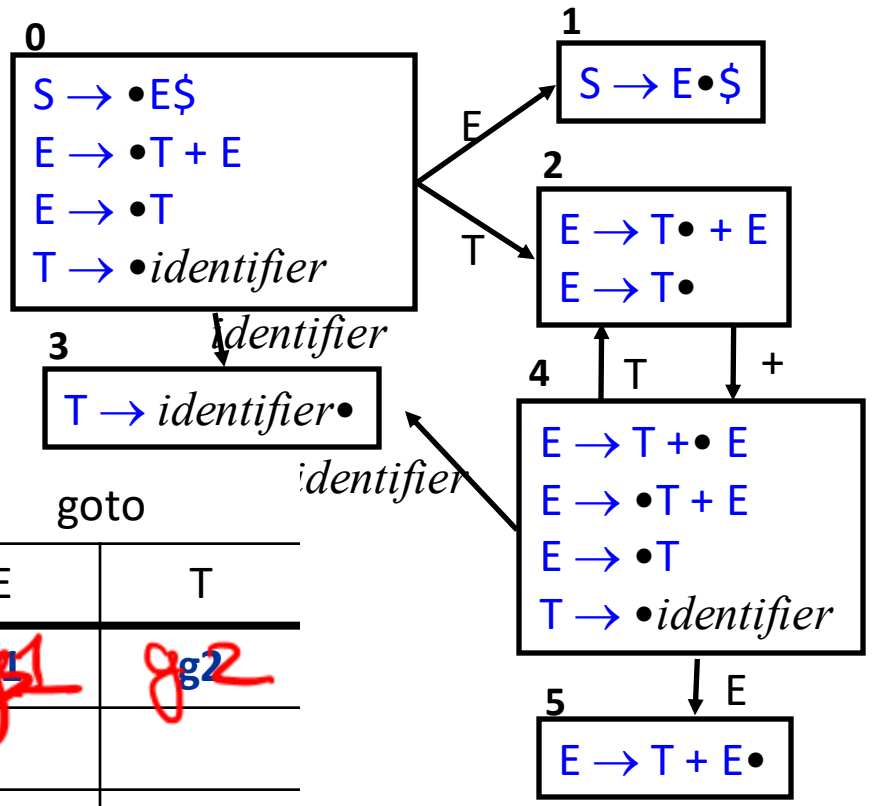
2 E → T

3 T → *identifier*

Parse Tables for LR(0) parser

goto

transition on nonterminal



state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1					
2		s4			
3					
4	s3			g5	g2
5					

0 S → E$

1 E → T + E

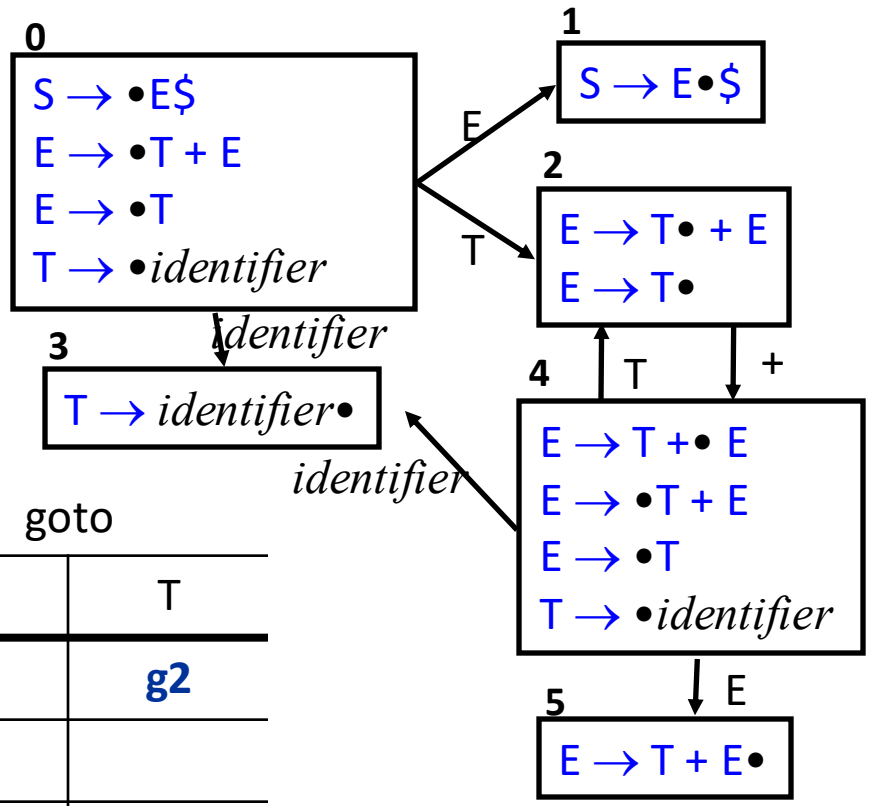
2 E → T

3 T → *identifier*

Parse Tables for LR(0) parser

accept

about to shift $



	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4			
3					
4	s3			g5	g2
5					

0 S → E$

1 E → T + E

2 E → T

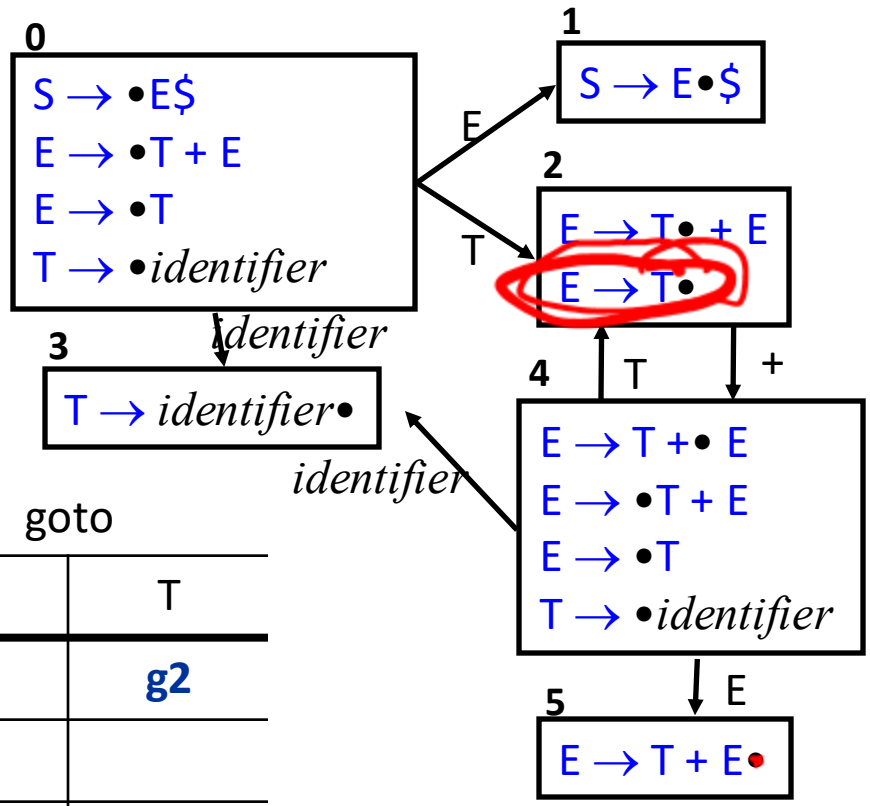
3 T → *identifier*

Parse Tables for LR(0) parser

reduce

item has dot at end

$A \rightarrow w\bullet$



	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2	R2	s4/R2	R2		
3	r3	r3	r3		
4	s3			g5	g2
5	r1	r1	r1		

0 $S \rightarrow E\$$

1 $E \rightarrow T + E$

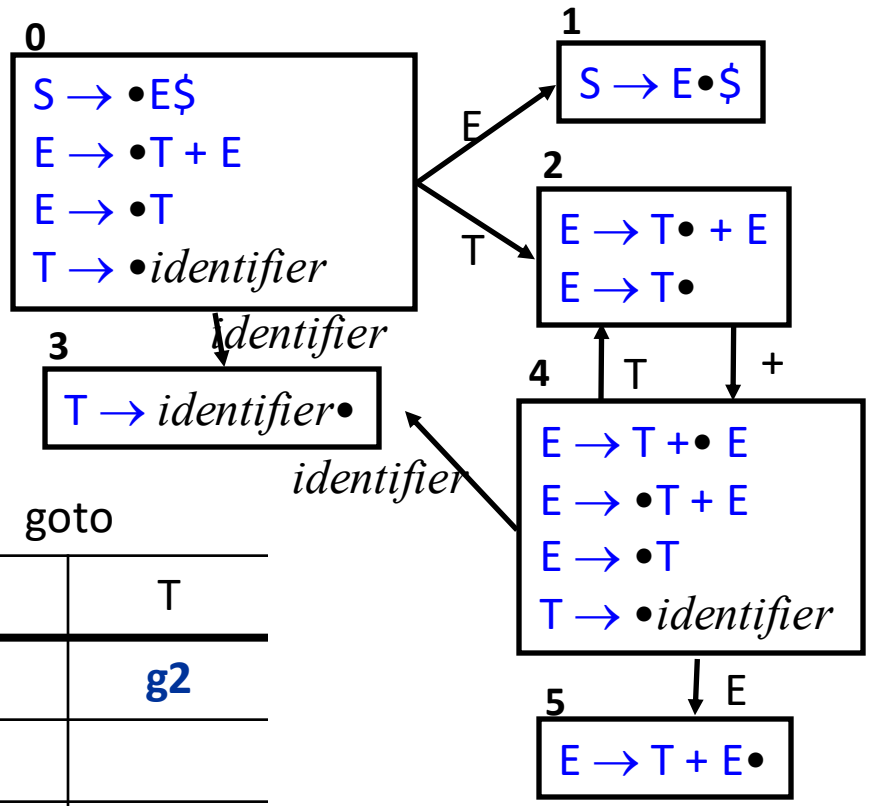
2 $E \rightarrow T$

3 $T \rightarrow identifier$

LR(0)

No lookahead

reduce state for *all* nonterminals



state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2	r2	r2/s4	r2		
3	r3	r3	r3		
4	s3			g5	g2
5	r1	r1	r1		

0 S → E$

1 E → T + E

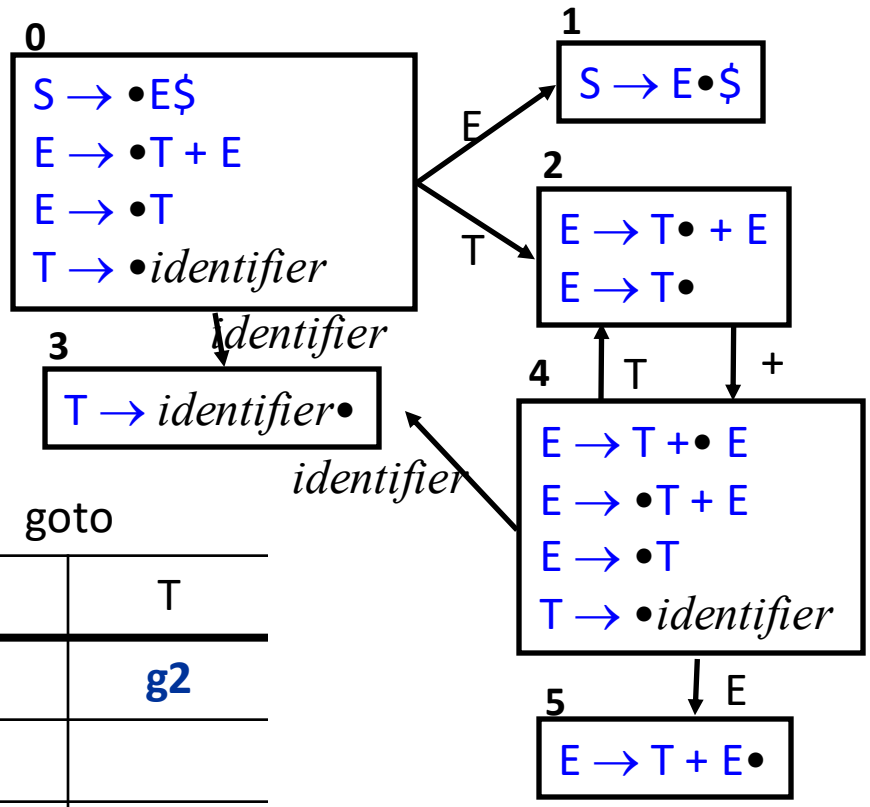
2 E → T

3 T → *identifier*

LR(0)

shift/reduce conflict

need to be pickier about when we reduce



	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2	r2	r2/s4	r2		
3	r3	r3	r3		
4	s3			g5	g2
5	r1	r1	r1		

0 S → E$

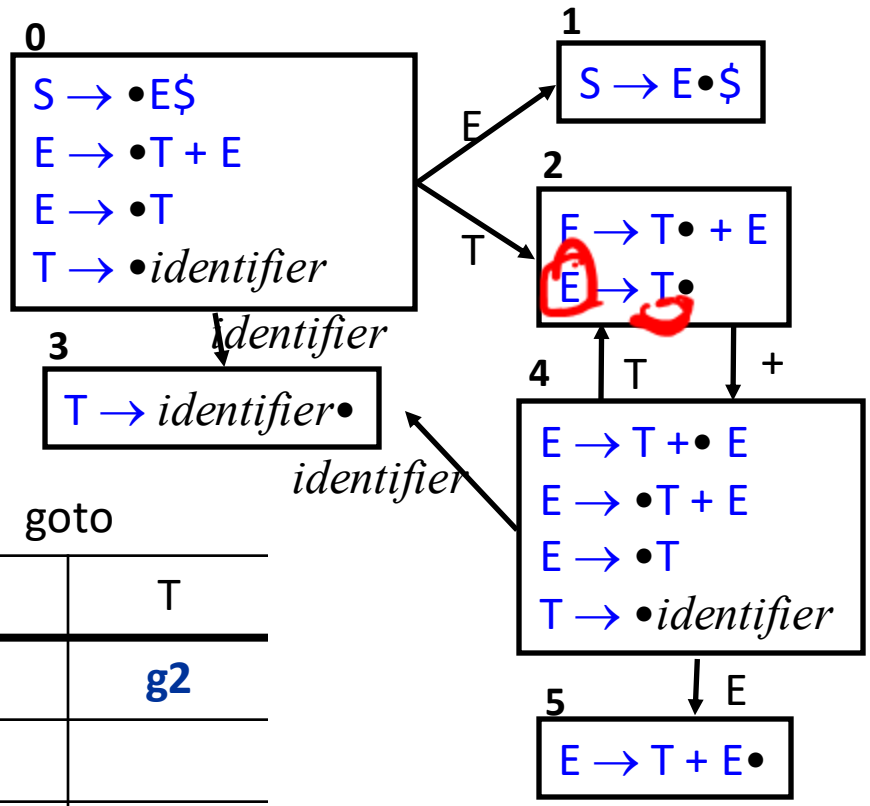
1 E → T + E

2 E → T

3 T → *identifier*

SLR - Simple LR

Only reduce in position (**s**,**a**) by rule R:A → *w* if **a** is in the *follow set* of A



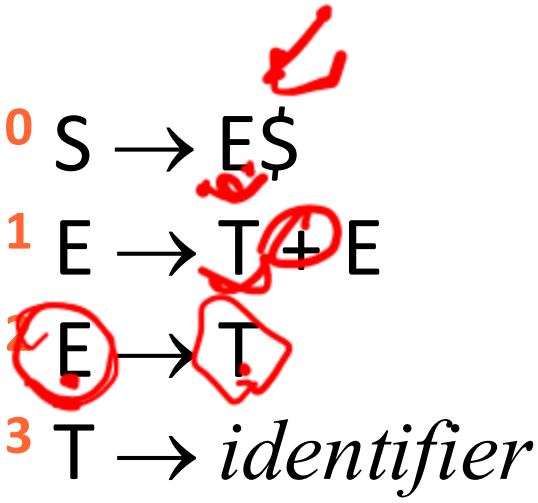
state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4			
3					
4	s3			g5	g2
5					

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

Reminder: Follow sets

follow(X)

set of terminals that can appear immediately after the nonterminal X in some sentential form



I.e., t ∈ FOLLOW(X) iff S ⇒* αXtβ for some α and β

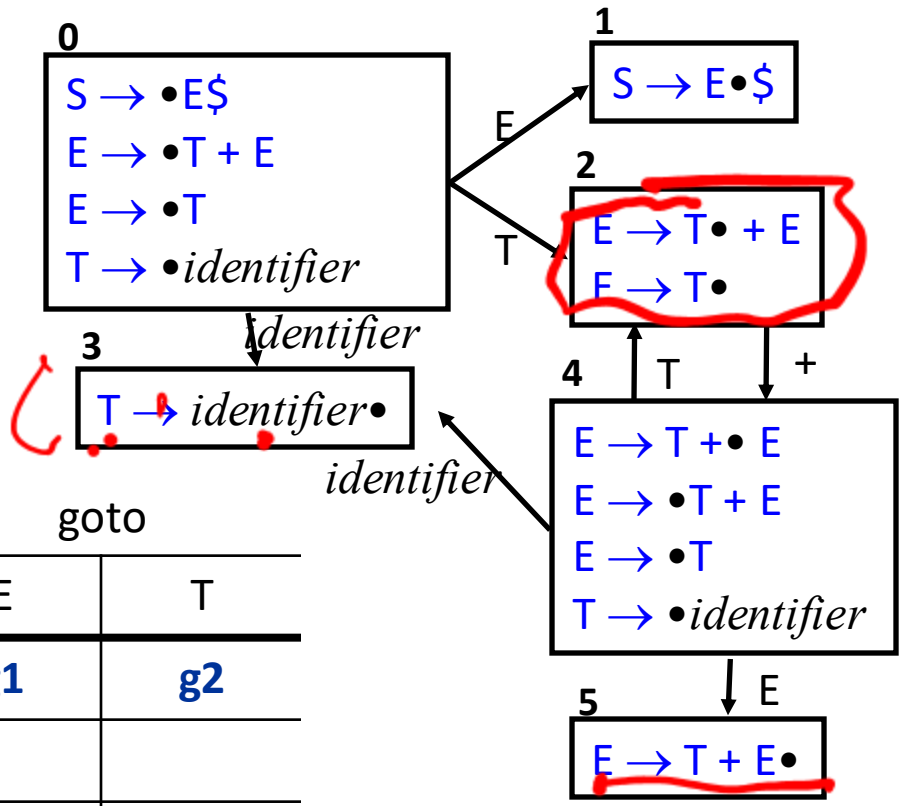
follow(E) = {$}

follow(T) = {+,$}

SLR - Reduce using follow sets

follow(E) = {$}

follow(T) = {+,$}



state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

SLR Limitations

- SLR uses LR(0) item sets
- Can remove some (but not all) shift/reduce conflicts using follow set
- Consider

0 S → E$

1 E → L = R

2 E → R

3 L → *id*

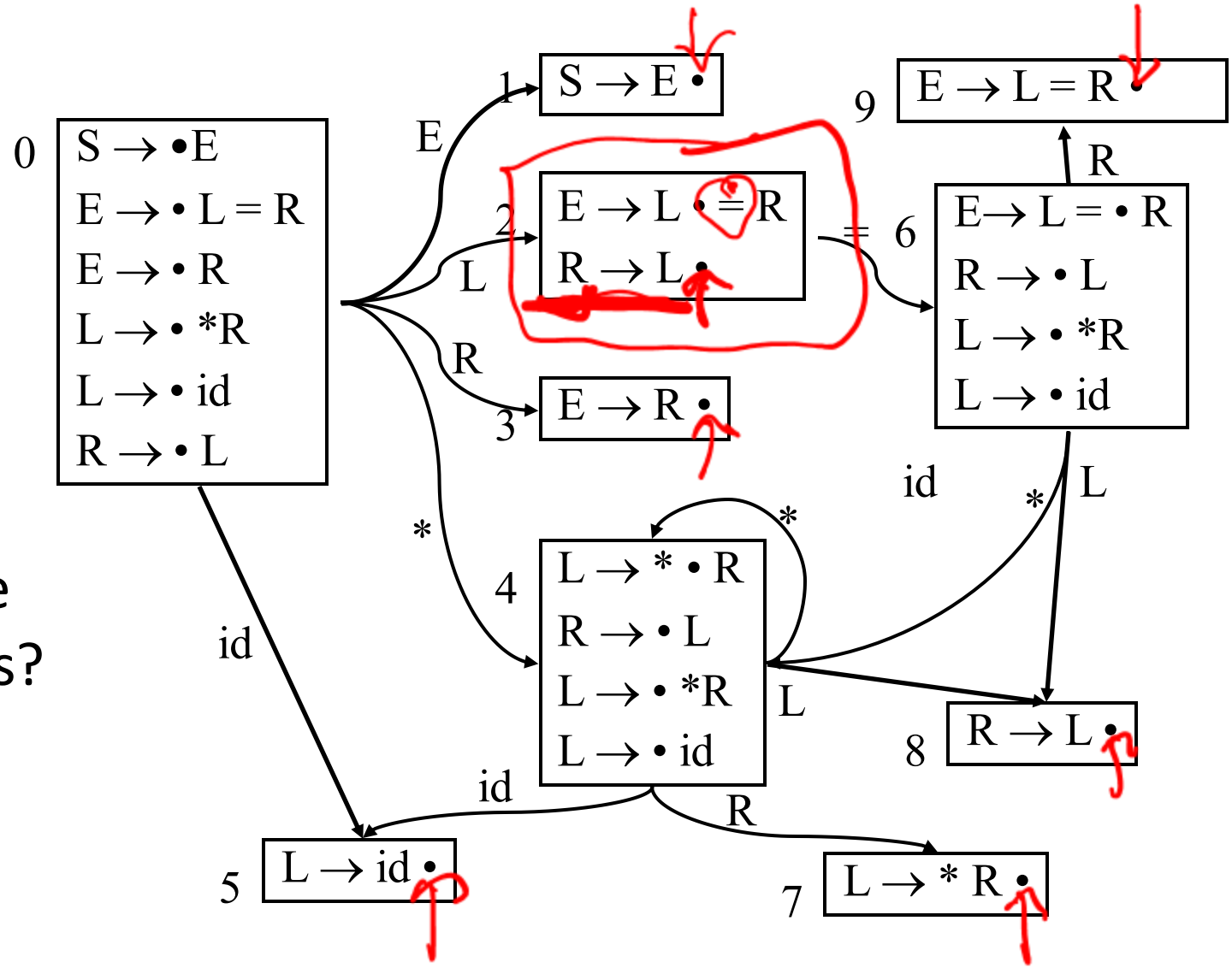
4 L → *R

5 R → L

Example

0 $S \rightarrow E\$$
1 $E \rightarrow L = R$
2 $E \rightarrow R$
3 $L \rightarrow id$
4 $L \rightarrow *R$
5 $R \rightarrow L$

What are the reduce states?

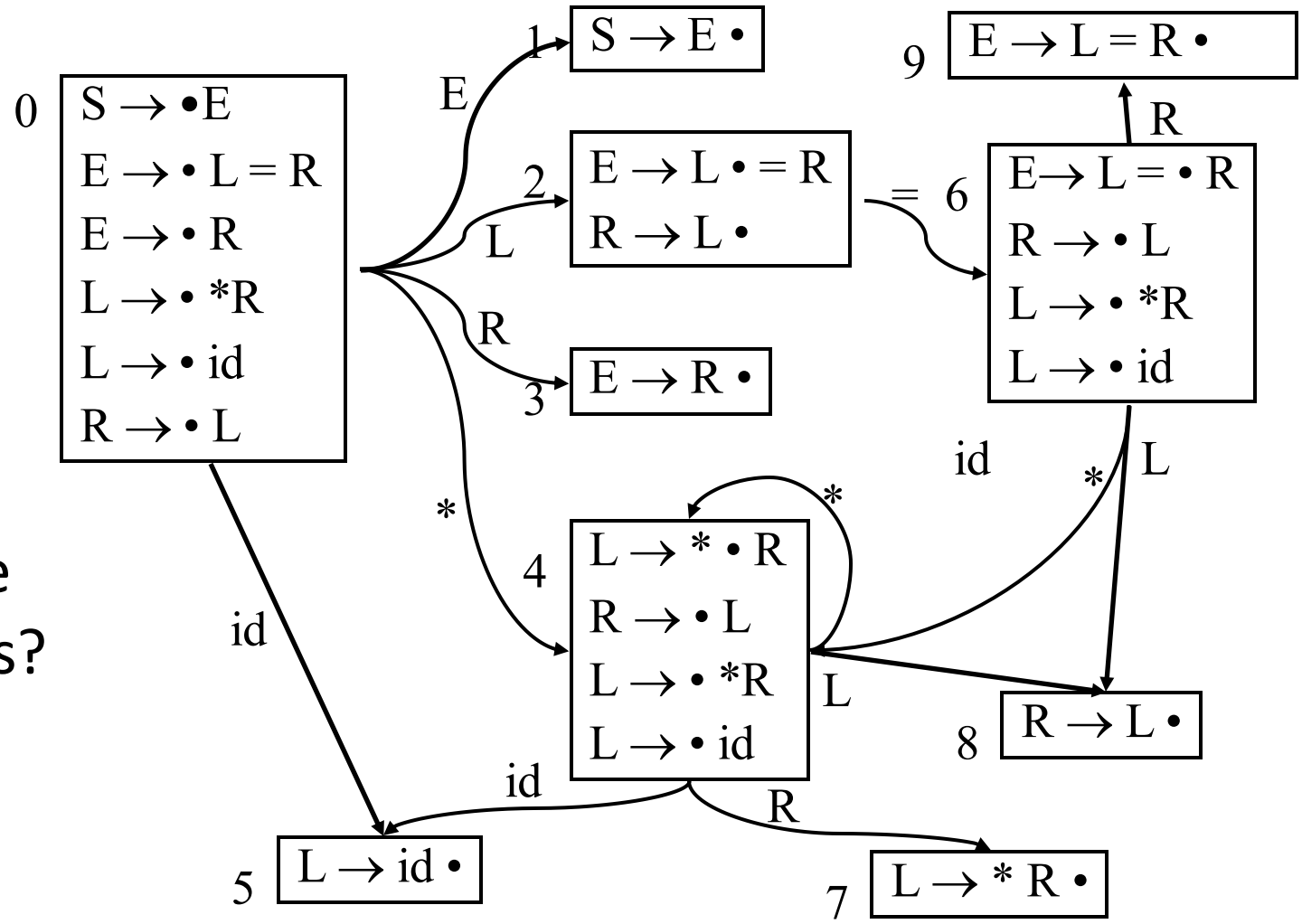


Example

0 S → E$
1 E → L = R
2 E → R
3 L → *id*
4 L → *R
5 R → L

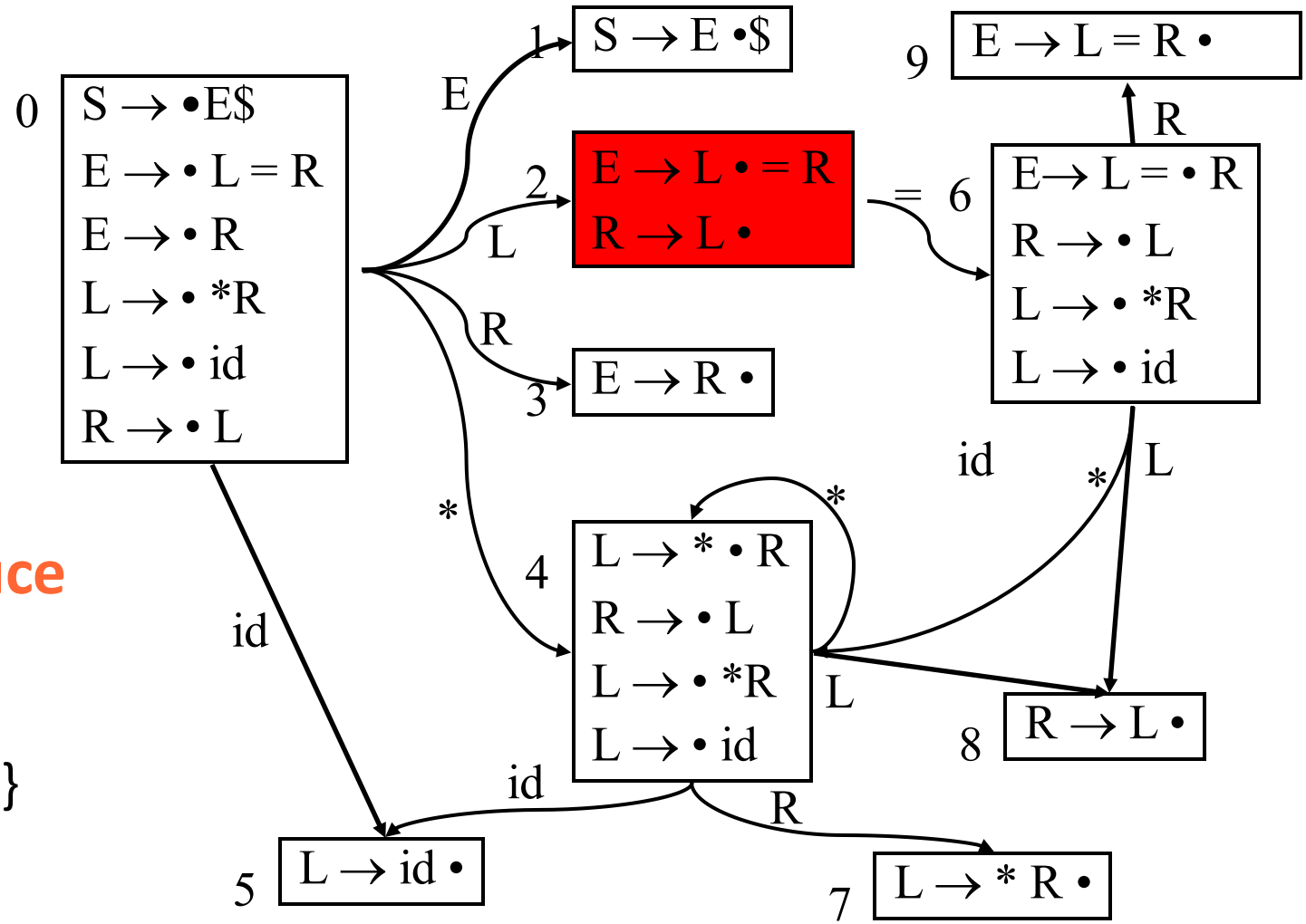
What are the reduce states?

1,2,3,5,7,8,9



Example

0 S → E$
1 E → L = R
2 E → R
3 L → *id*
4 L → *R
5 R → L



shift/reduce conflict

follow(R) = {=,$}

Problem with SLR

- Reduce on ALL terminals in FOLLOW set

S	→	L = R
	|	R
L	→	* R
	|	id
R	→	L

2 $S \rightarrow L \bullet = R$

$R \rightarrow L \bullet$

- FOLLOW(R) = FOLLOW(L)
- But, we should never reduce R → L on '=' I.e., R=… is not a viable prefix for a right sentential form
- Thus, there should be no reduction in state 2
- How can we solve this?

LR(1) Items

- An LR(1) item is an LR(0) item combined with a single terminal (the *lookahead*)
- $[X \rightarrow \alpha \bullet \beta, a]$ Means
  - α is at top of stack
  - Input string is derivable from βa
- In other words, when we reduce $X \rightarrow \alpha\beta$, a had better be the look ahead symbol.
- Or, Only put 'reduce by $X \rightarrow \alpha\beta$' in **`action`**`[s,a]`
- Can construct states as before, but have to modify closure

What LR(1) Items Mean

- [X $\rightarrow$ • α β γ, a]
  input is consistent with X $\rightarrow$ α β γ
- [X $\rightarrow$ α • β γ, a]
  input is consistent with X $\rightarrow$ α β γ and we have already recognized α
- [X $\rightarrow$ α β • γ, a]
  input is consistent with X $\rightarrow$ α β γ and we have already recognized α β
- [X $\rightarrow$ α β γ •, a]
  input is consistent with X $\rightarrow$ α β γ and if lookahead symbol is a, then we can reduce to X

LR(1) Closure

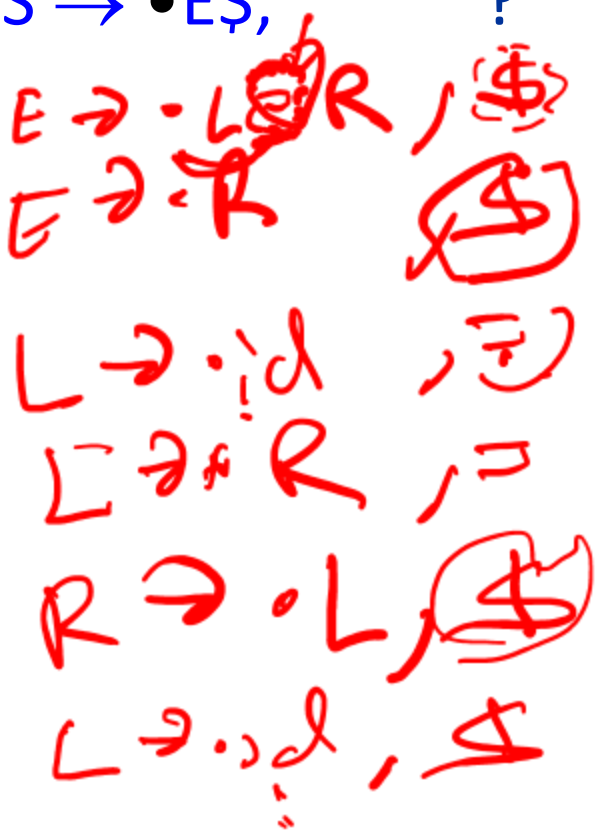
```
closure(state)
    repeat
       foreach item A → a•Xb, t in state
          foreach production X → w
            and each terminal t' in FIRST(bt)
             state.add(X → •w, t')
    until state does not change
    return state
```

```
foreach item A → a•Xb, t in state
    foreach production X → w
      foreach terminal t' in FIRST(bt)
          state.add(X → •w, t')
```

Closure

closure({S → •E$, ?}) =

S → •E$, ?



0 S → E$
1 E → L = R
2 E → R
3 L → *id*
4 L → *R
5 R → L

```
foreach item A → a•Xb, t in state
    foreach production X → w
      foreach terminal t' in FIRST(bt)
          state.add(X → •w, t')
```

Closure

closure({S → •E$, ?}) =

S → •E$, ?
E → •L = R, $
E → •R, $

0 S → E$
1 E → L = R
2 E → R
3 L → *id*
4 L → *R
5 R → L

```
foreach item A → a•Xb, t in state
    foreach production X → w
      foreach terminal t' in FIRST(bt)
          state.add(X → •w, t')
```

Closure

closure({S → •E$, ?}) =

S → •E$,	?
E → •L = R,	$
E → •R,	$
L → •*id*,	=
L → •*R,	=

0 S → E$
1 E → L = R
2 E → R
3 L → *id*
4 L → *R
5 R → L

```
foreach item A → a•Xb, t in state
    foreach production X → w
      foreach terminal t' in FIRST(bt)
          state.add(X → •w, t')
```

Closure

closure({S → •E$, ?}) =

S → •E$,	?
E → •L = R,	$
E → •R,	$
L → •*id*,	=
L → •*R,	=
R → •L,	$

0 S → E$
1 E → L = R
2 E → R
3 L → *id*
4 L → *R
5 R → L

```
foreach item A → a•Xb, t in state
    foreach production X → w
      foreach terminal t' in FIRST(bt)
          state.add(X → •w, t')
```

Closure

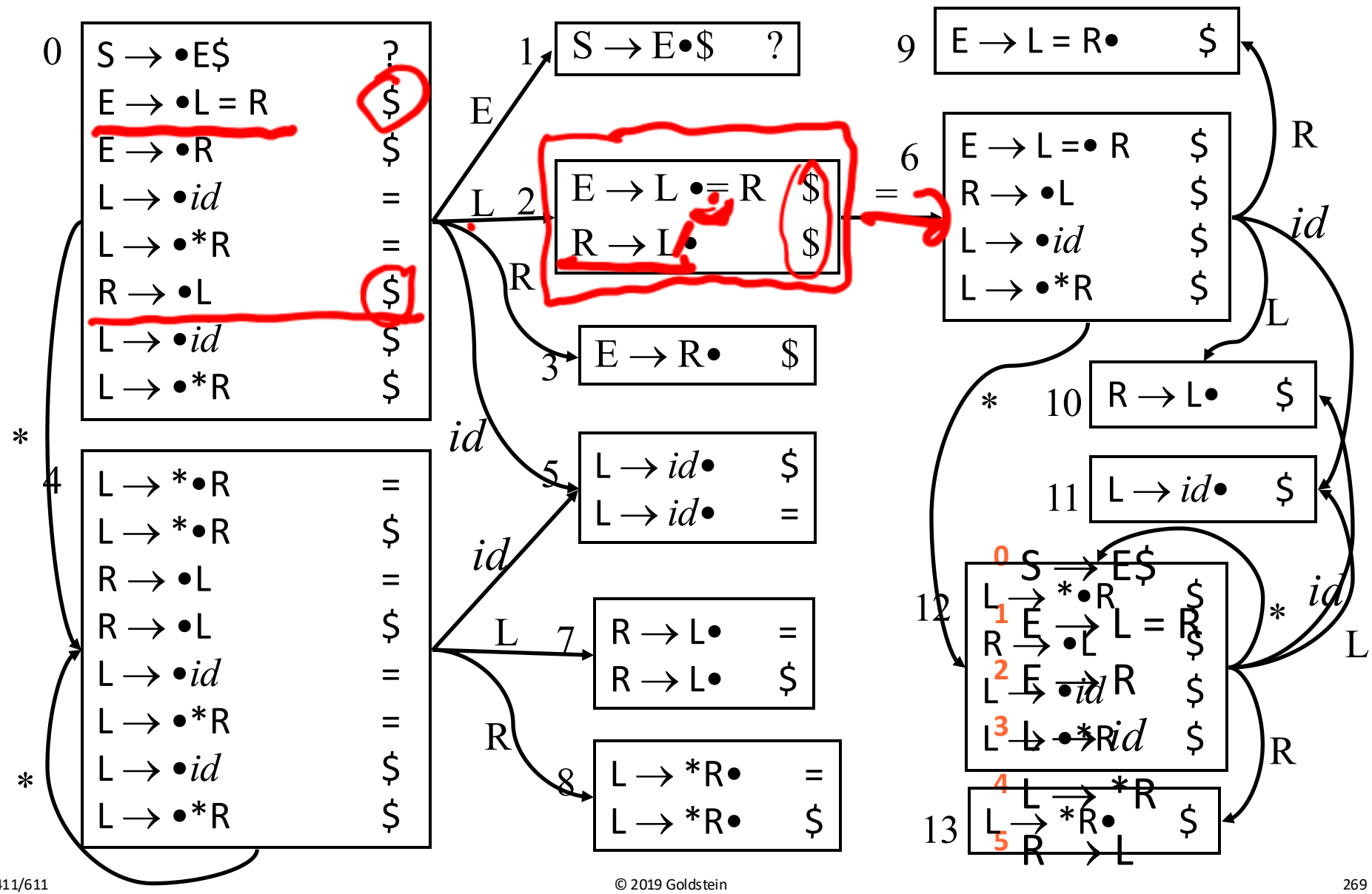
closure({S → •E$, ?}) =

S → •E$,	?
E → •L = R,	$
E → •R,	$
L → •*id*,	=
L → •*R,	=
R → •L,	$
L → •*id*,	$
L → •*R,	$

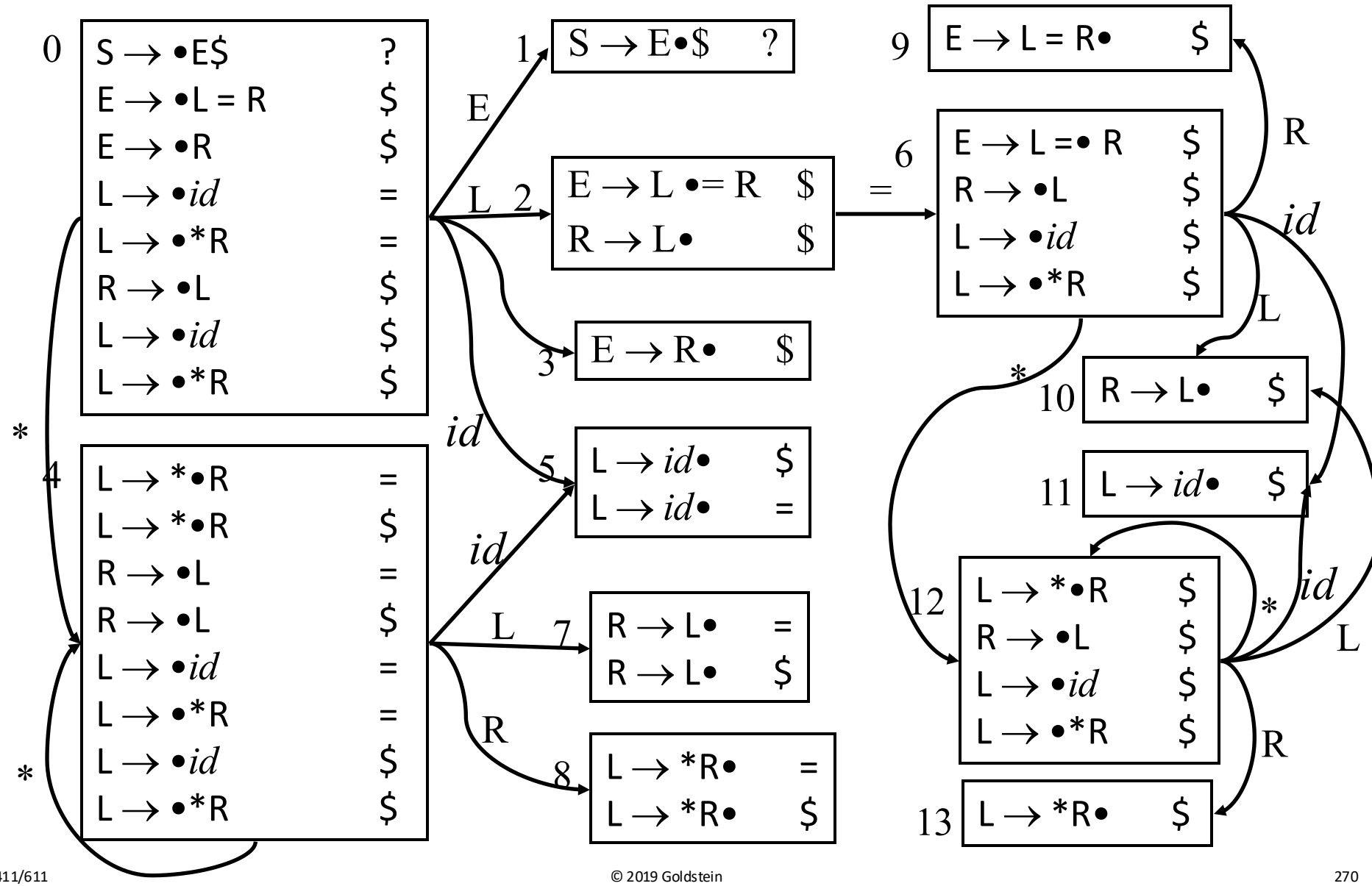
0 S → E$
1 E → L = R
2 E → R
3 L → *id*
4 L → *R
5 R → L

```
foreach item A → a•Xb, t in state
    foreach production X → w
      foreach terminal t' in FIRST(bt)
          state.add(X → •w, t')
```

LR(1) Example



LR(1) Example



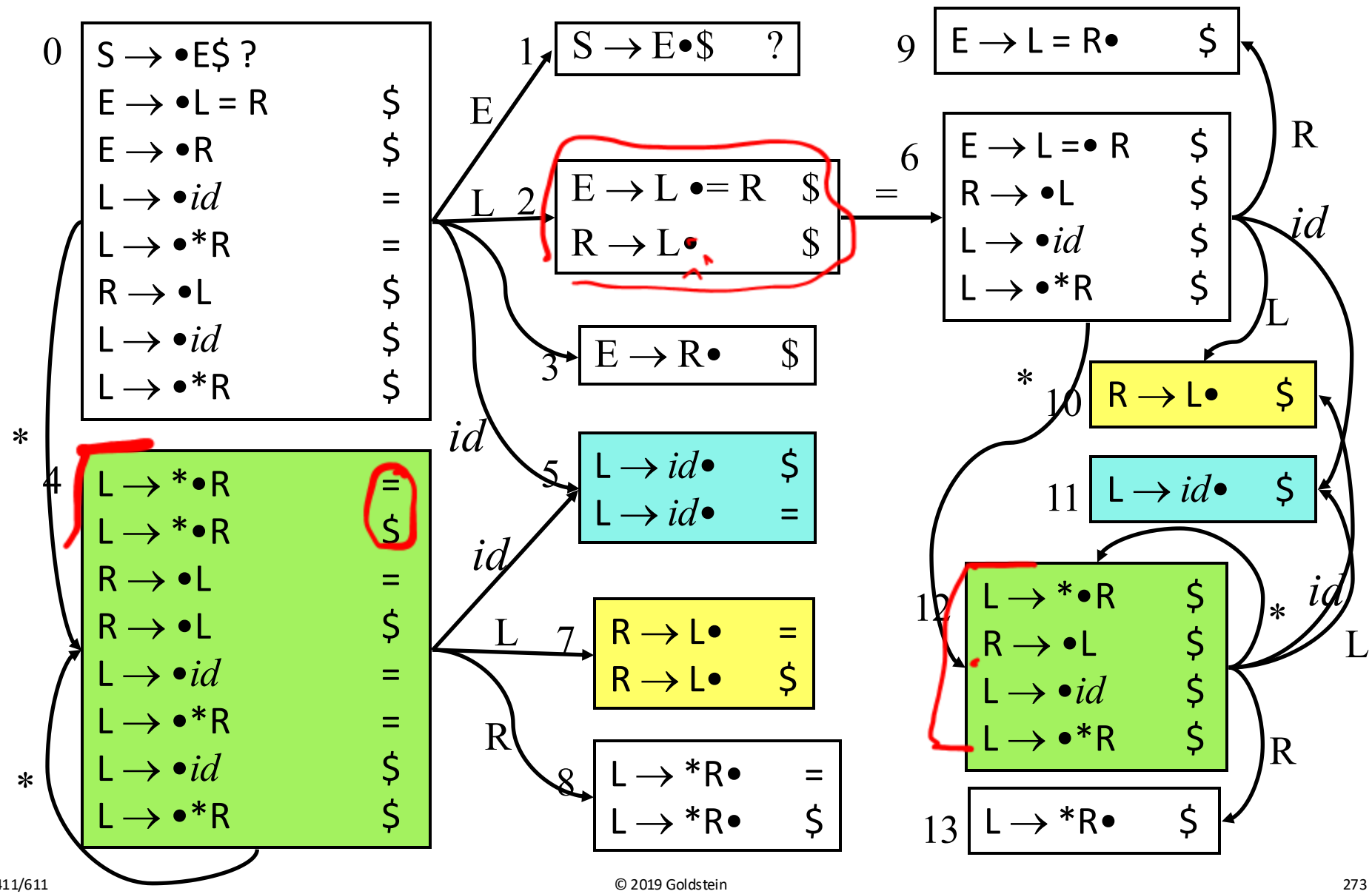
Parsing Table

- 14 states versus 10 LR(0) states
- In general, the number of states (and therefore size of the parsing table) is much larger with LR(1) items

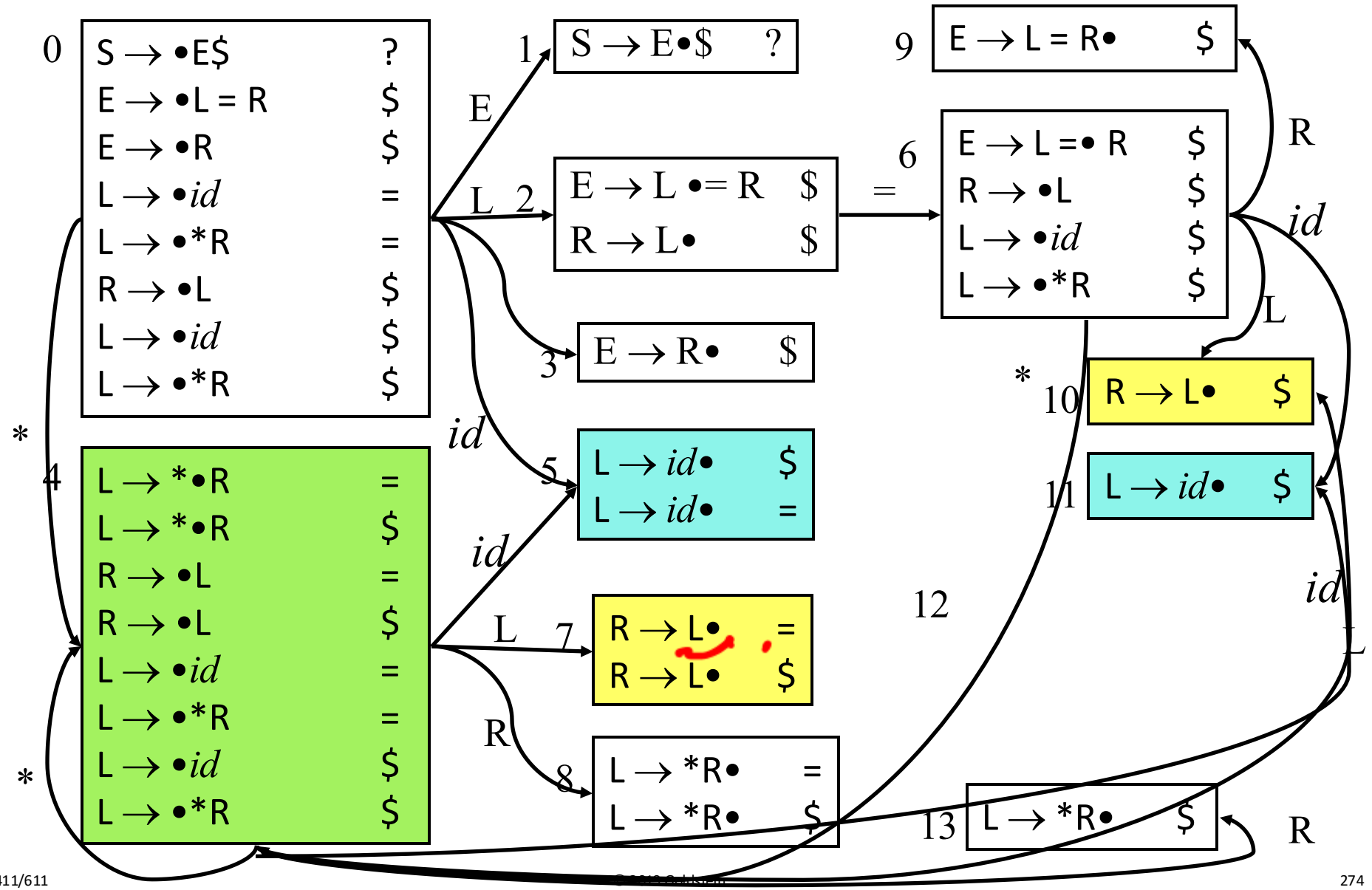
LALR: Lookahead LR

- More powerful than SLR
- Given LR(1) states, merge states that are identical except for lookaheads
- End up with same size table as SLR
- Can this introduce conflicts?

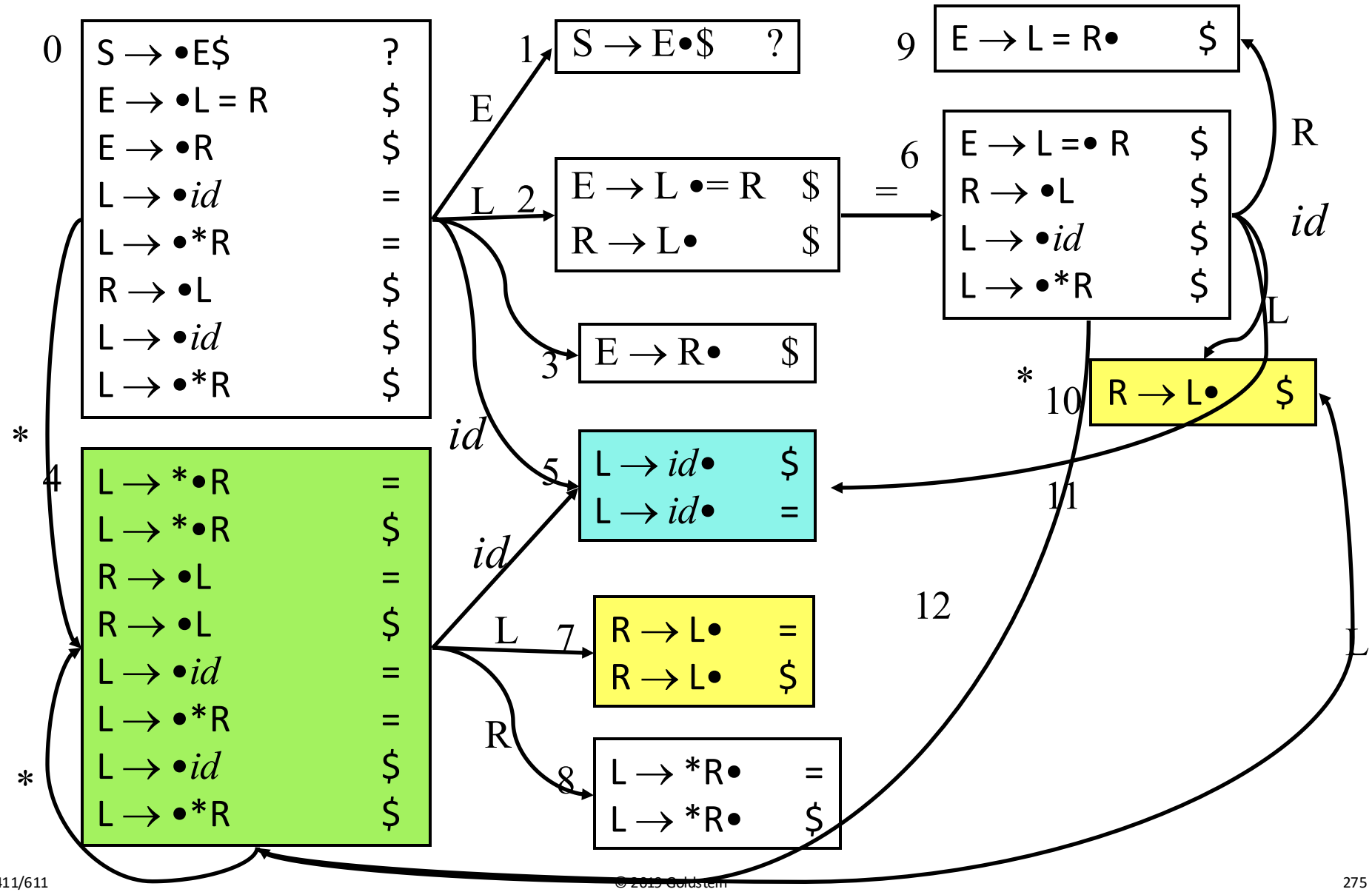
Merge-able states



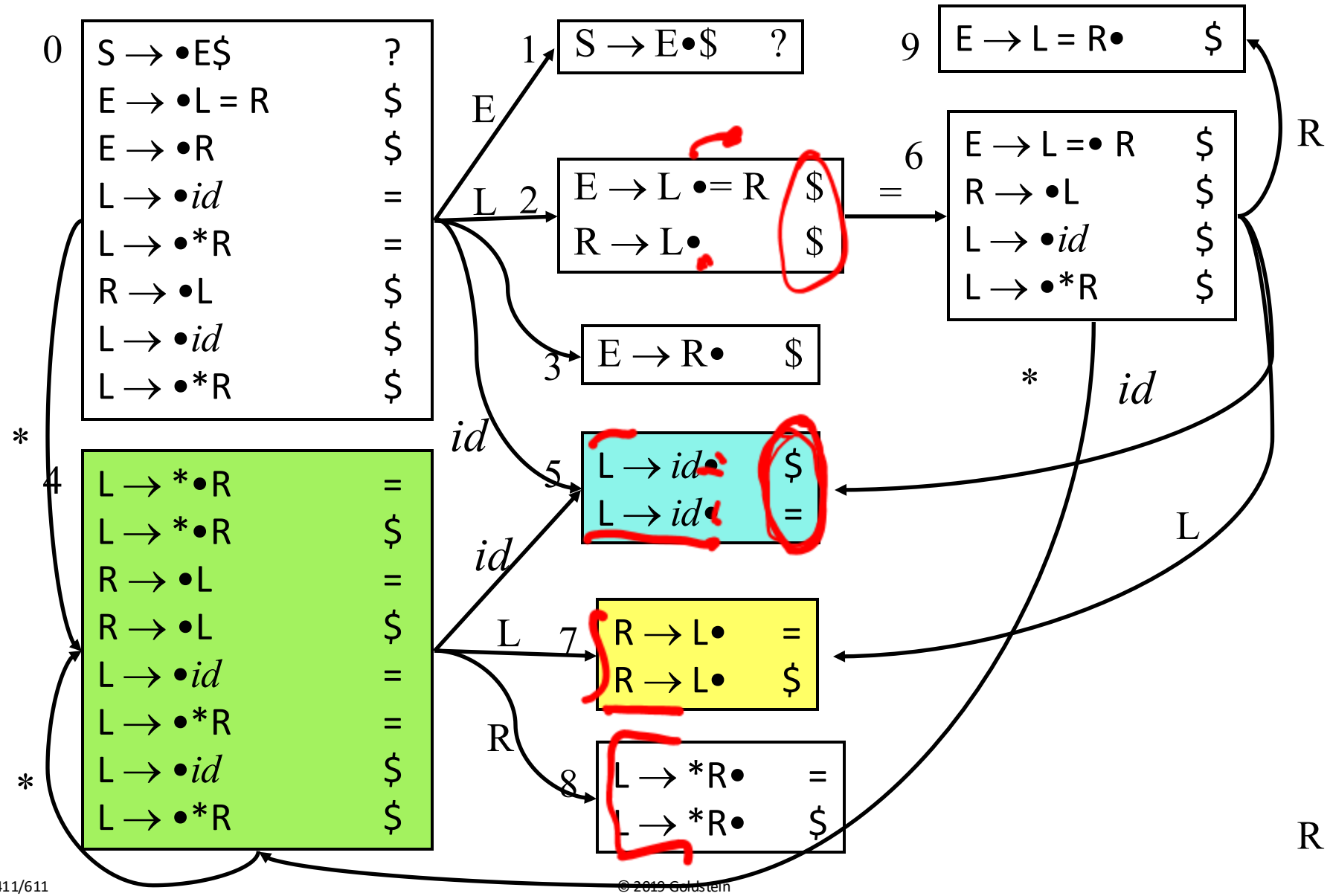
Merge-able states



Merge-able states



Merge-able states



LALR

- Can generate parse table without constructing LR(1) item sets
  - construct LR(0) item sets
  - compute *lookahead* sets
    - more precise than follow sets
- LALR is used by most parser generators (e.g., bison)

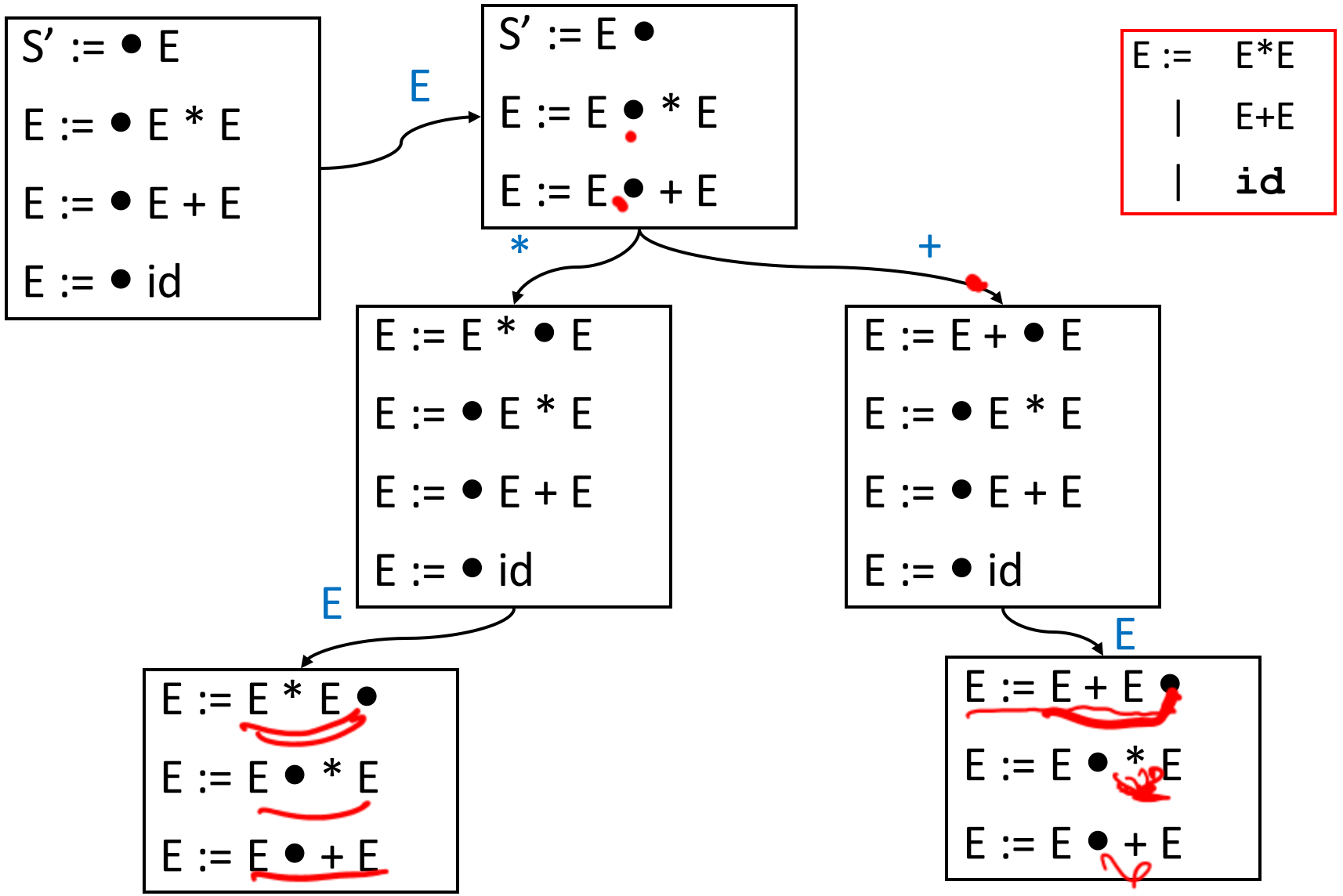
Recap

- LR(0) not very useful
- SLR uses follow sets to reduce
- LALR uses lookahead sets
- LR(1) uses full lookahead context

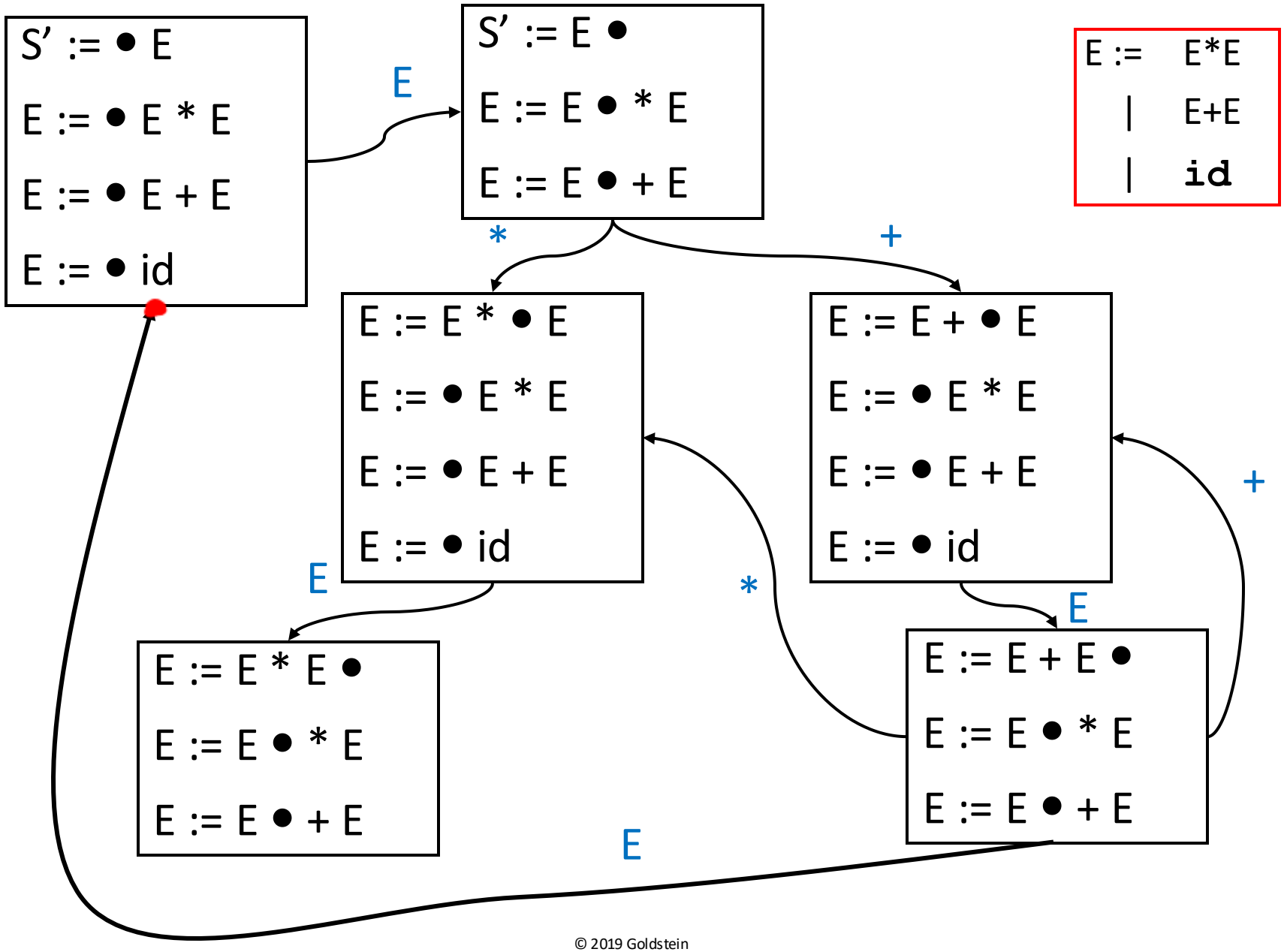
Power of shift-reduce parsers

- There are unambiguous grammars which which cannot be parsed with shift-reduce parsers.
- Such grammars can have
  - shift/reduce conflicts
  - reduce/reduce conflicts
- There grammars are not LR(k)
- But, we can often choose shift or reduce to recognize what want.

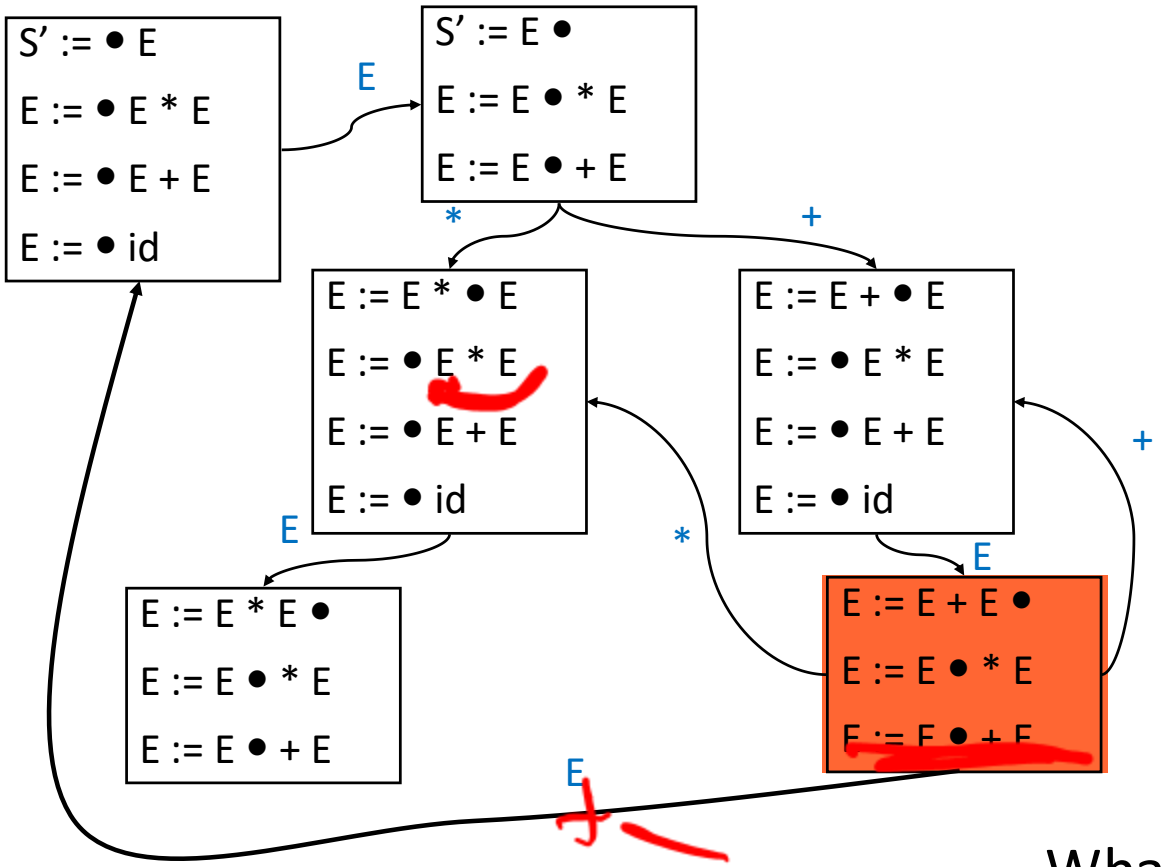
Expression Grammars & Precedence



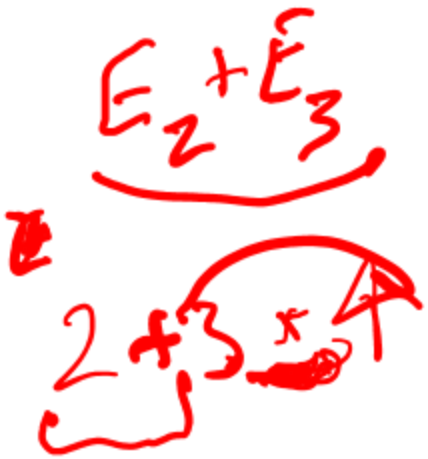
Expression Grammars & Precedence



Handling Ambiguity




```
E :=   E*E
   |   E+E
   |   id
```



What to do on + or *?

- shift
- reduce by E → E+E?

Bison

- Precedence and Associativity declarations
- Precedence derived from order of directivies: from lowest to highest
- Associativity from %left, %right, %nonassoc
- Can be attached to rules as well (This can solve the dangling if-else problem

Dangling Else

```
S := if E then S
   | if E then S else S
   | other
```

- We can be in the following state:

  … **if** E **then** S **else** … **$**

- What do we do?
  - shift the **else** (hoping to reduce by second rule)
  - reduce by first rule