CS406: Compilers

Spring 2022

Week 5: Parsers – Bottom-up Parsing (background concepts), Bottom-up parsing (use of goto and action tables)

Concept: configuration / item

- Configuration or item has a form:

  $$A \rightarrow X_1 \ldots X_i \bullet X_{i+1} \ldots X_j$$

- Dot $\bullet$ can appear anywhere
- Represents a production part of which has been matched (what is to the left of Dot)
- LR parsers keep track of multiple (all) productions that can be potentially matched
  - We need a *configuration set*

Concept: configuration / item

- E.g. configuration set

```
stmt -> ID • := expr
stmt -> ID • : stmt
stmt -> ID •
```

Corresponding to productions:

```
stmt -> ID := expr
stmt -> ID : stmt
stmt -> ID
```

- Dot at the extreme left of RHS of a production denotes that production is predicted
- Dot at the extreme right of RHS of a production denotes that production is recognized
- if Dot precedes a Non-Terminal in a configuration set, more configurations need to be added to the set

Concept: closure

➢ For each configuration in the configuration set,

$A \ -> \ \alpha \bullet B\gamma$, where B is a non-terminal,

1 add configurations of the form:

$B \ -> \bullet \ \delta$

2 if the addition introduces a configuration with Dot behind a new non-Terminal N, add all configurations having the form N -> $\bullet$ ε

Repeat 2 when another new non-terminal is introduced and so on..

Concept: closure

Grammar

```
S -> E$
E -> E+T | T
T -> ID | (E)
```

- E.g. closure {S -> •E$}

  ⇩ Non-terminal

  S -> •E$

Concept: closure

Grammar

```
S -> E$
E -> E+T | T
T -> ID | (E)
```

- E.g. closure {S -> • E$}

Non-terminal

S -> •E$
E -> •E+T

Concept: closure

Grammar

```
S -> E$
E -> E+T | T
T -> ID | (E)
```

- E.g. closure {S -> • E$}

Non-terminal

```
S -> • E$
E -> • E+T
E -> • T
```

Concept: closure

- E.g. closure {S -> • E$}

⇩

```
S -> •E$
E -> •E+T
E -> •T
```

New Non-terminal

Grammar

```
S -> E$
E -> E+T | T
T -> ID | (E)
```

Concept: closure

- E.g. closure {S -> • E$}

⇩

```
S -> •E$
E -> •E+T
E -> •T
T -> •ID
```

New Non-terminal

Grammar

```
S -> E$
E -> E+T | T
T -> ID | (E)
```

Concept: closure

Grammar

```
S -> E$
E -> E+T | T
T -> ID | (E)
```

- E.g. closure {S -> • E$}

```
S ->•E$
E ->•E+T
E ->•T
T ->•ID
T ->•(E)
```

New Non-terminal

Concept: closure

Grammar

```
S -> E$
E -> E+T | T
T -> ID | (E)
```

- E.g. closure {S -> • E$}

$$\Downarrow$$

$$\left\{\begin{array}{l}
S \rightarrow \bullet E\$ \\
E \rightarrow \bullet E+T \\
E \rightarrow \bullet T \\
T \rightarrow \bullet ID \\
T \rightarrow \bullet (E)
\end{array}\right\}$$

Concept: successor

Grammar

```
S -> E$
E -> E+T | T
T -> ID | (E)
```

- E.g. successor ({S -> •E$}, E)

$$\left\{\begin{array}{l} \texttt{S -> •E\$} \\ \texttt{E -> •E+T} \\ \texttt{E -> •T} \\ \texttt{T -> •ID} \\ \texttt{T -> •(E)} \end{array}\right\} \xrightarrow{E} \left\{\begin{array}{l} \texttt{S -> E•\$} \\ \texttt{E -> E•+T} \end{array}\right\}$$

- Consider all symbols that are to the <u>immediate right of Dot</u> and compute respective successors
  - You must compute closure of successor before finalizing items in successor

Concept: CFSM

- Each configuration set becomes a state
- The symbol used as input for computing the successor becomes the transition
- Configuration-set finite state machine (CFSM)
  - The state diagram obtained after computing the chain of all successors (for all symbols) starting from the configuration involving the first production

Example: CFSM

Start with a configuration for the first production

P->• S

Grammar

P->S

S->x;S

S->e

Example: CFSM

Compute closure

P->• S ← Non-terminal

Grammar

P->S

S->x;S

S->e

Example: CFSM

Add item

```
P->•S
S->•x;S
```

Grammar

```
P->S
S->x;S
S->e
```

Example: CFSM

Add item

P->•S
S->•x;S
S->•e

Grammar

P->S
S->x;S
S->e

Example: CFSM

Grammar

P->S

S->x;S

S->e

No new non-terminal before Dot. This becomes a state in CFSM

P->• S

S->• x;S

S->• e

state 0

Example: CFSM

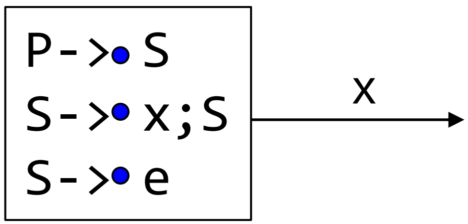
Grammar

P->S

S->x;S

S->e

Compute successor (of state 0) under symbol x



state 0

Consider items (in state 0), where x is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

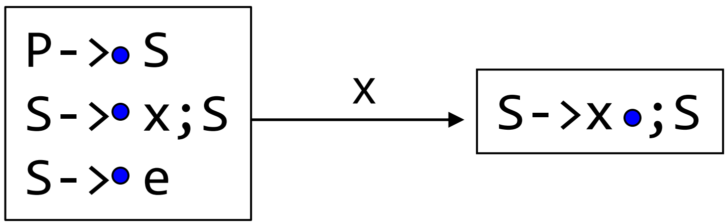
Grammar

P->S

S->x;S

S->e

Compute successor (of state 0) under symbol x



state 0

Consider items (in state 0), where x is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

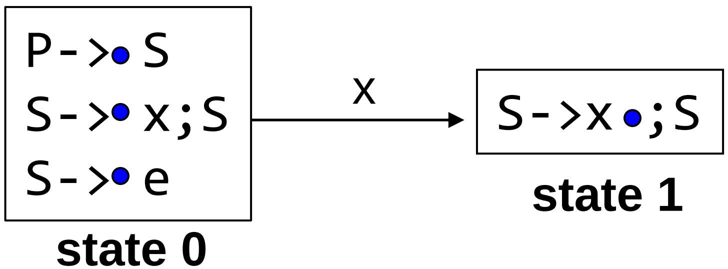
Grammar

P->S

S->x;S

S->e

Compute successor (of state 0) under symbol x

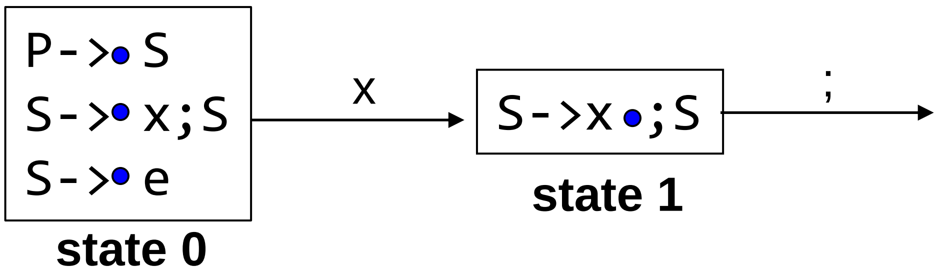


Consider items (in state 0), where x is to the immediate right of Dot. Advance Dot by one symbol.

No non-terminals immediately after Dot in the successor. So, no configurations get added. Successor becomes another state in CFSM.

Example: CFSM

Compute successor (of state 1) under symbol ;



Grammar

P->S

S->x;S

S->e

Consider items (in state 1), where ; is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

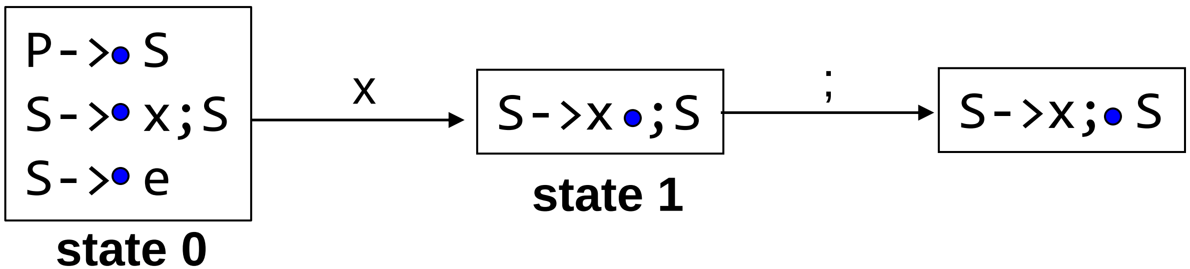
Grammar

P->S

S->x;S

S->e

Compute successor (of state 1) under symbol ;



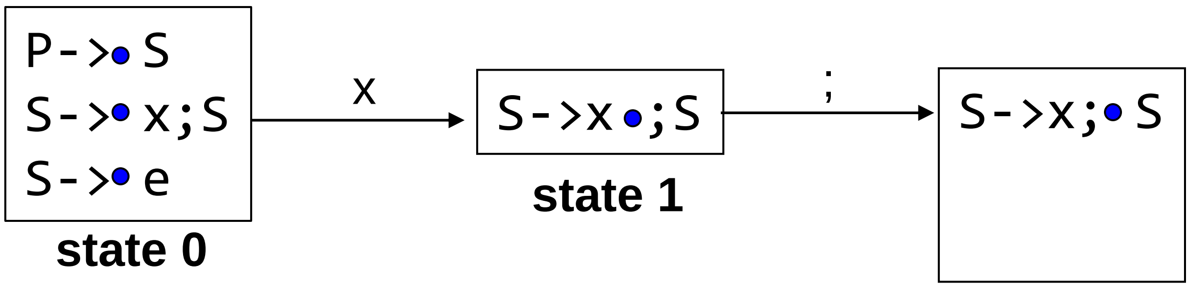
Consider items (in state 1), where ; is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

Grammar

P->S
S->x;S
S->e

Compute successor (of state 1) under symbol ;



Consider items (in state 1), where ; is to the immediate right of Dot. Advance Dot by one symbol.

There is a non-terminal immediately after Dot in the successor of state 1. So, add configurations.

Example: CFSM

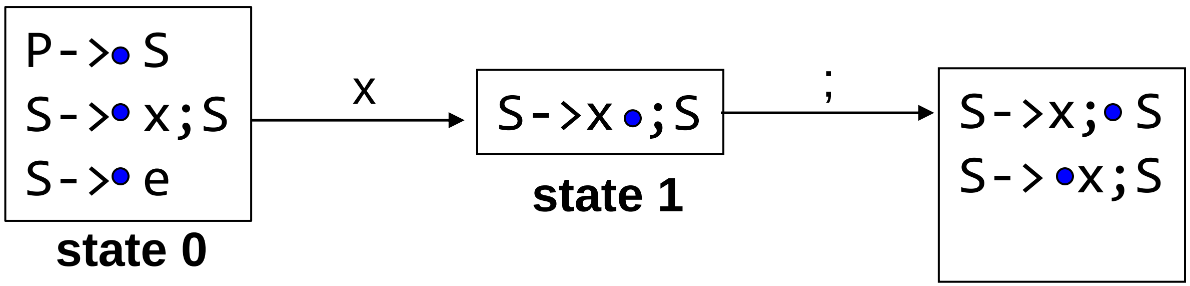
Grammar

P->S

S->x;S

S->e

Compute successor (of state 1) under symbol ;



Consider items (in state 1), where ; is to the immediate right of Dot. Advance Dot by one symbol.

There is a non-terminal immediately after Dot in the successor of state 1. So, add configurations.

Example: CFSM

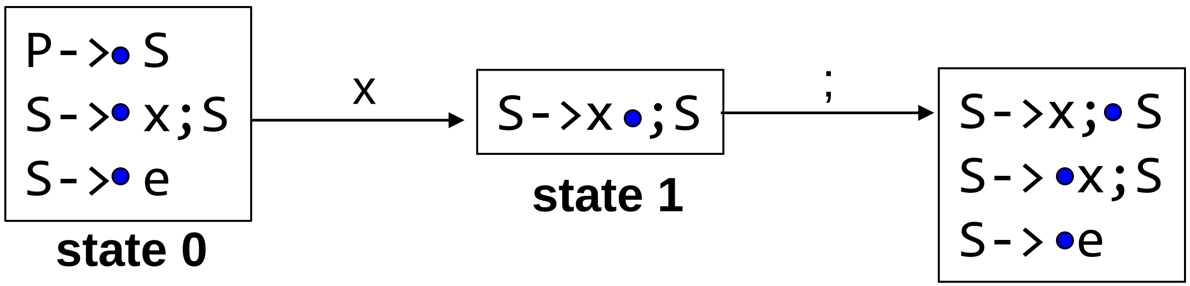
Grammar

P->S

S->x;S

S->e

Compute successor (of state 1) under symbol ;



Consider items (in state 1), where ; is to the immediate right of Dot. Advance Dot by one symbol.

There is a non-terminal immediately after Dot in the successor of state 1. So, add configurations.

Example: CFSM

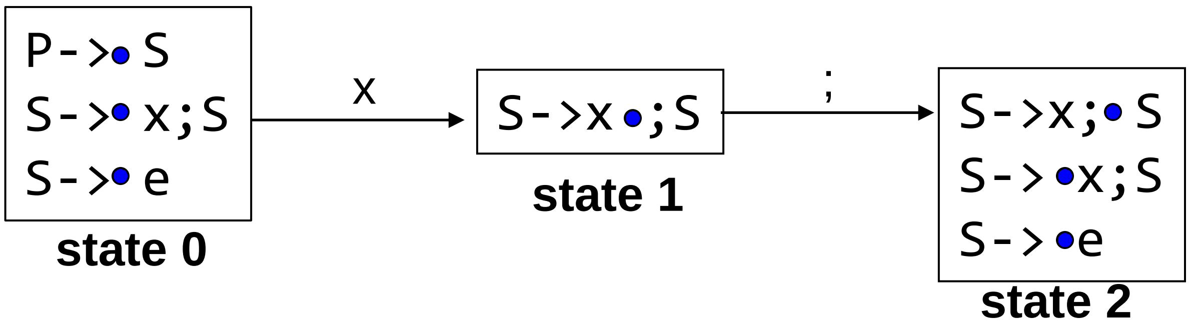
Grammar

P->S

S->x;S

S->e

Compute successor (of state 1) under symbol ;

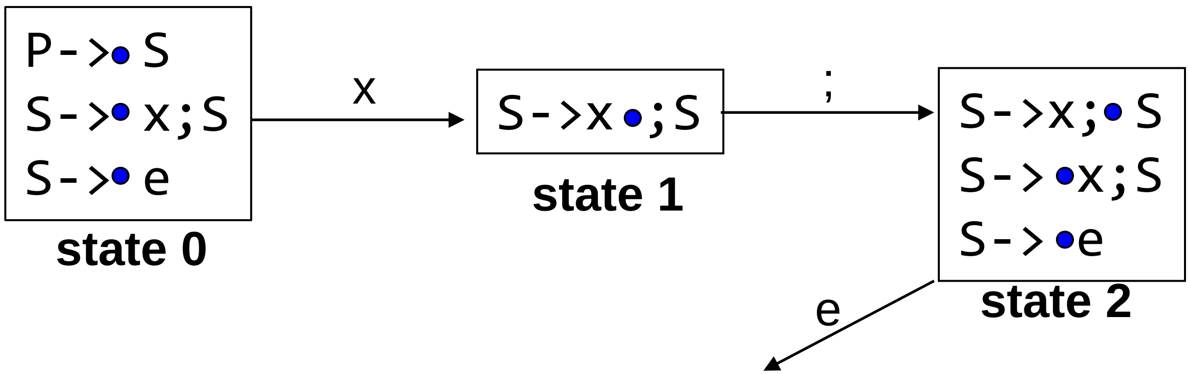


Consider items (in state 1), where ; is to the immediate right of Dot. Advance Dot by one symbol.

There is a non-terminal immediately after Dot in the successor of state 1. So, add configurations. No more items to be added. Becomes another state in CFSM.

Example: CFSM

Compute successor (of state 2) under symbol e



Grammar

P->S

S->x;S

S->e

Consider items (in state 2), where e is to the immediate right of Dot. Advance Dot by one symbol.

Example: CFSM

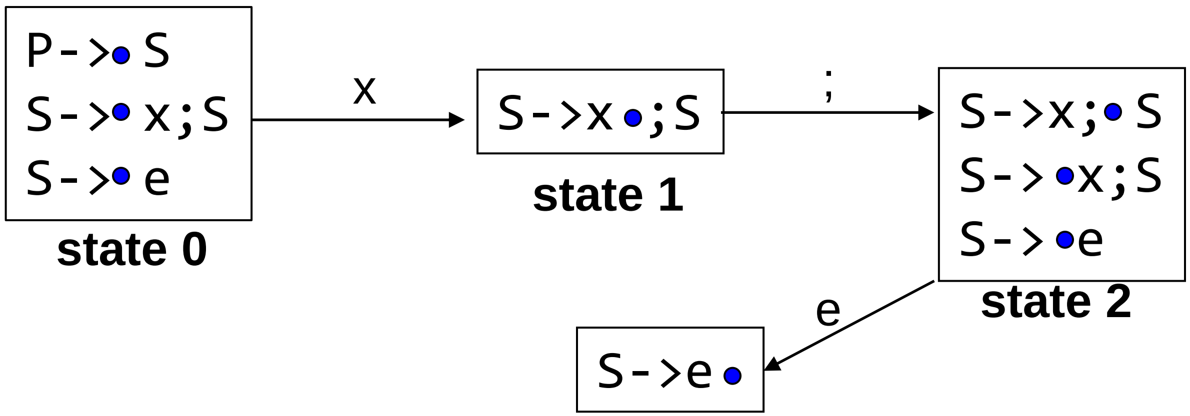
Grammar

P->S

S->x;S

S->e

Compute successor (of state 2) under symbol e



Consider items (in state 2), where e is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

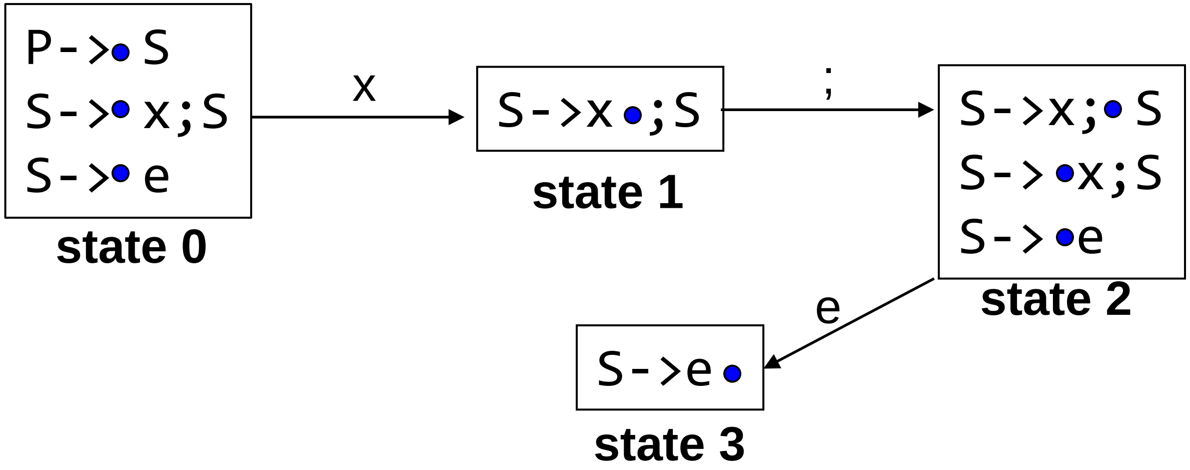
Grammar

P->S

S->x;S

S->e

Compute successor (of state 2) under symbol e



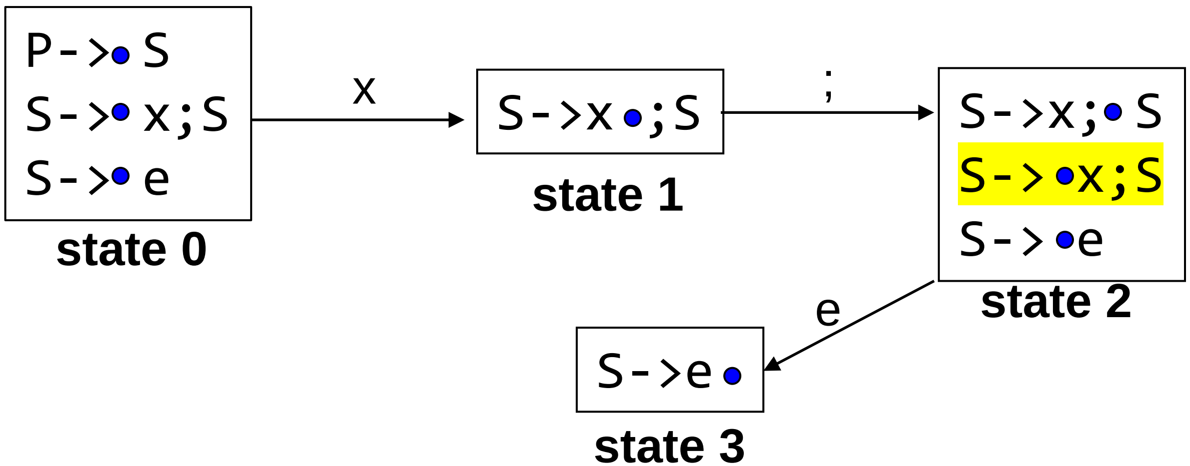
Consider items (in state 2), where e is to the immediate right of Dot. Advance Dot by one symbol. No more items to be added. Becomes another state in CFSM.

Example: CFSM

Grammar

```
P->S
S->x;S
S->e
```

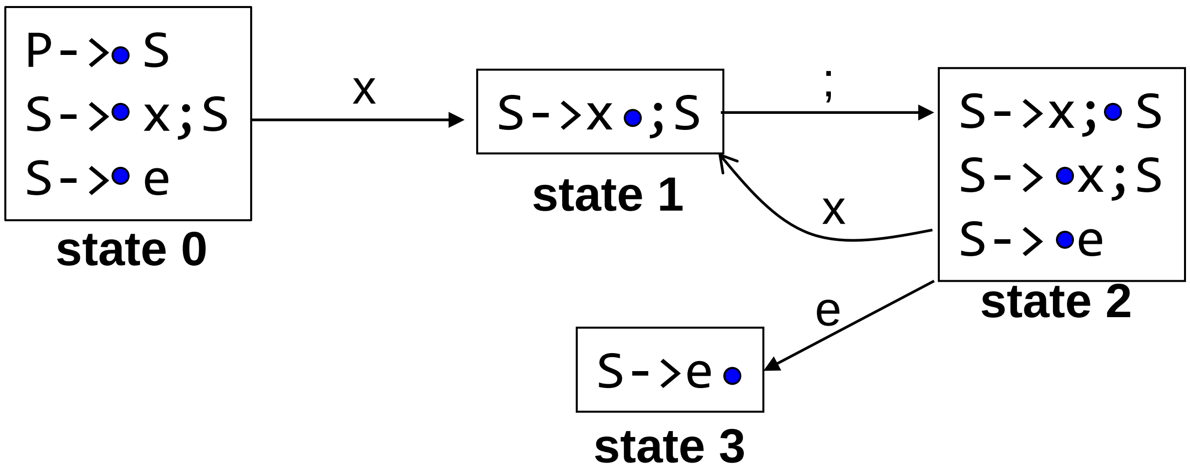
Compute successor (of state 2) under symbol x



Consider items (in state 2), where x is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

Compute successor (of state 2) under symbol x



Grammar

P->S

S->x;S

S->e

Consider items (in state 2), where x is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

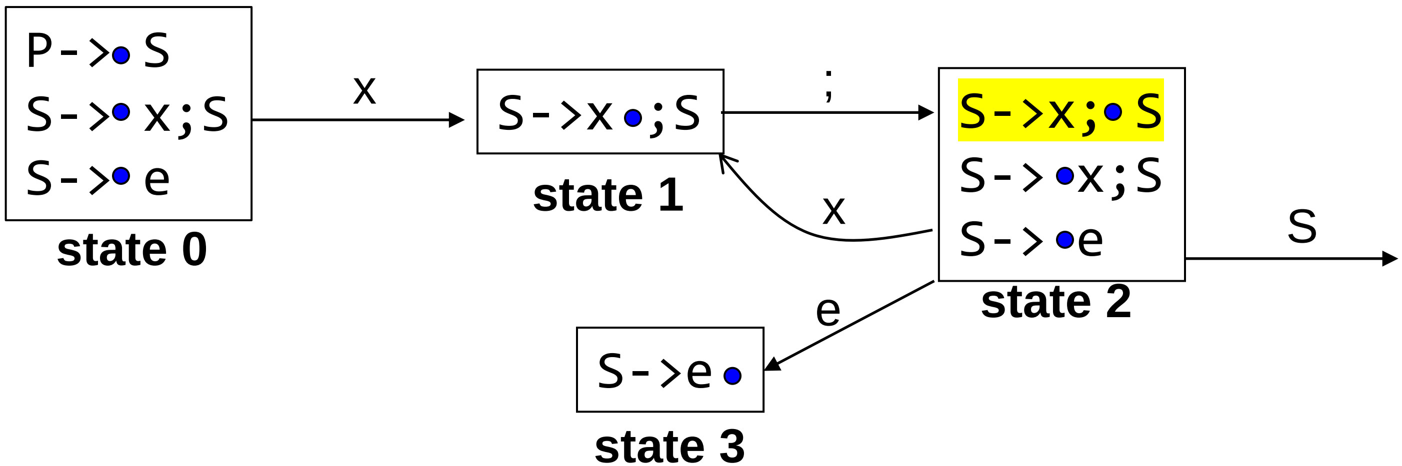
Grammar

P->S

S->x;S

S->e

Compute successor (of state 2) under symbol S



Consider items (in state 2), where S is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

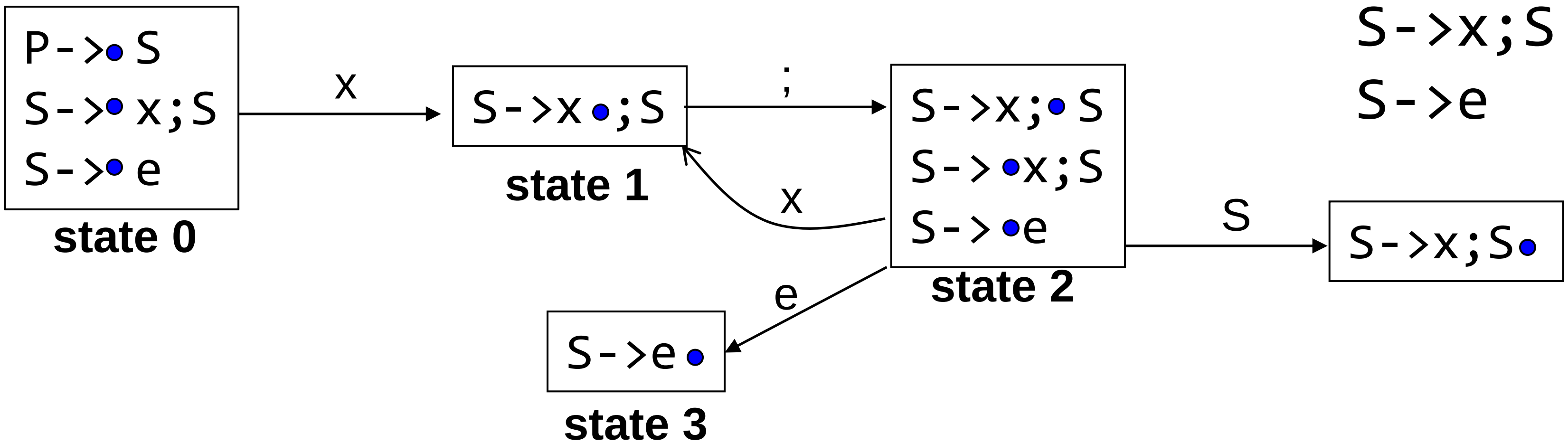
Grammar

P->S

S->x;S

S->e

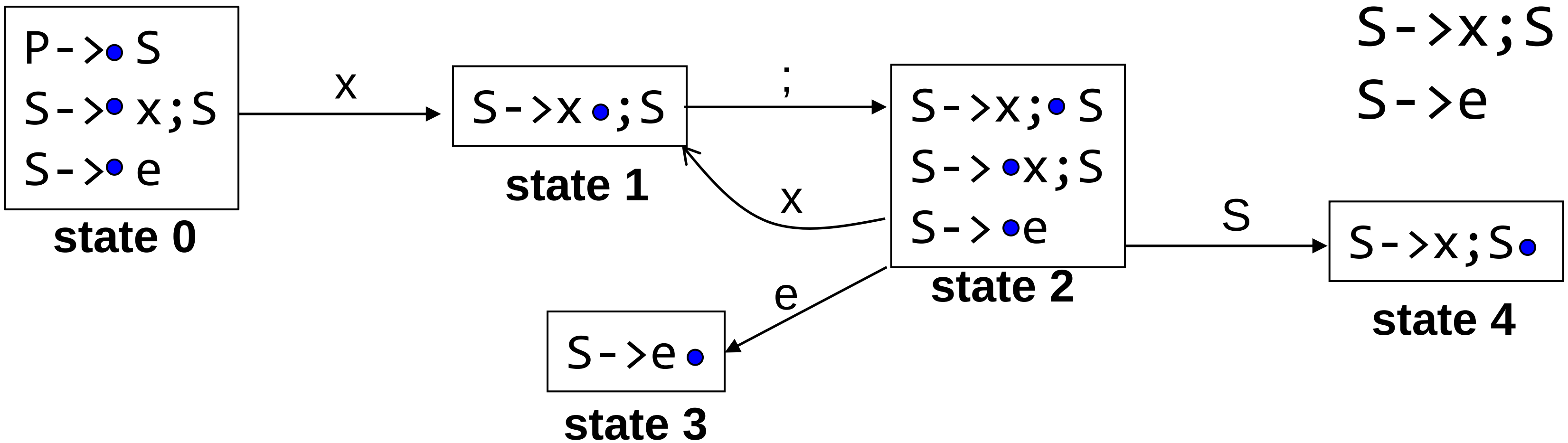
Compute successor (of state 2) under symbol S



Consider items (in state 2), where S is to the immediate right of Dot. Advance Dot by one symbol.

Example: CFSM

Compute successor (of state 2) under symbol S



Grammar

P->S

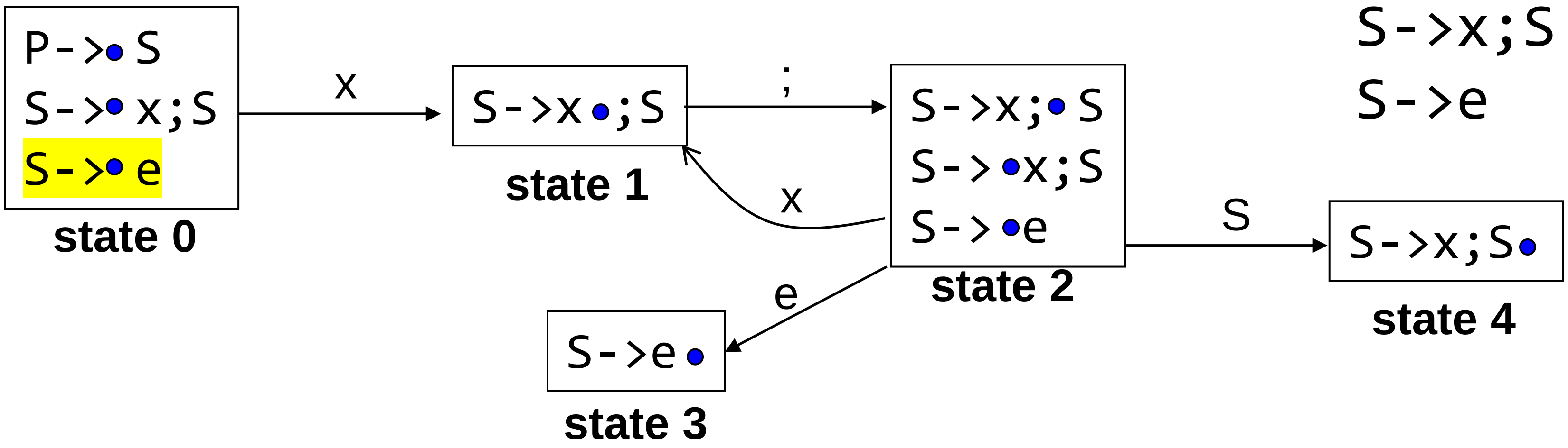
S->x;S

S->e

Consider items (in state 2), where S is to the immediate right of Dot. Advance Dot by one symbol. No more items to be added. Becomes another state in CFSM.

Example: CFSM

Compute successor (of state 0) under symbol e



Grammar

P->S
S->x;S
S->e

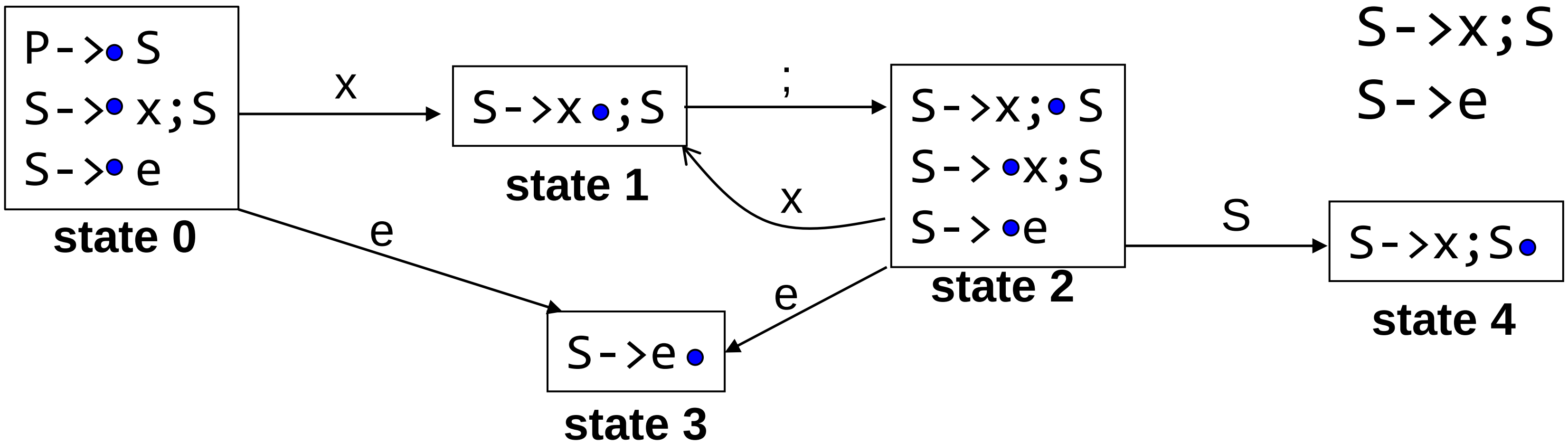
Consider items (in state 0), where e is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

Grammar

P->S
S->x;S
S->e

Compute successor (of state 0) under symbol e



Consider items (in state 0), where e is to the immediate right of Dot. Advance Dot by one symbol.

Example: CFSM

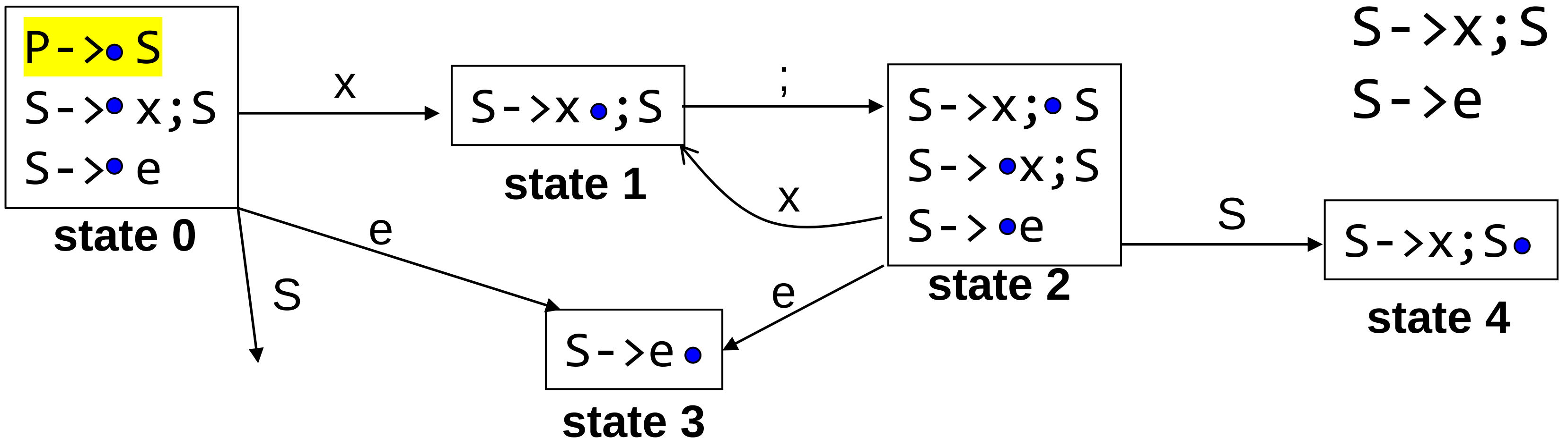
Grammar

P->S

S->x;S

S->e

Compute successor (of state 0) under symbol S



Consider items (in state 0), where S is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

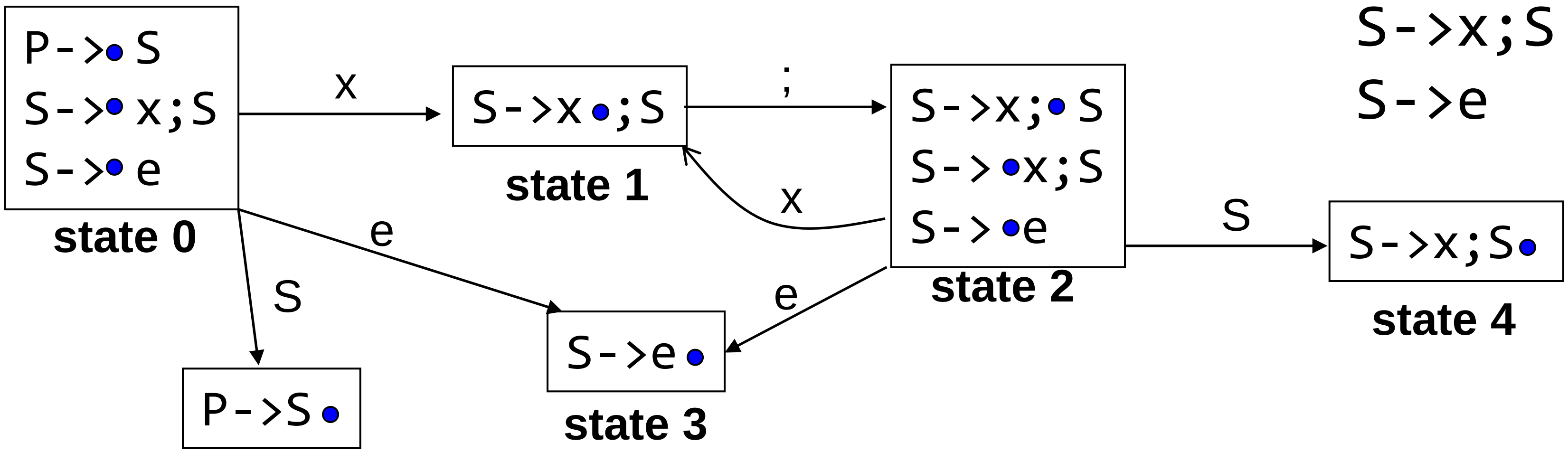
Grammar

P->S

S->x;S

S->e

Compute successor (of state 0) under symbol S



Consider items (in state 0), where S is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

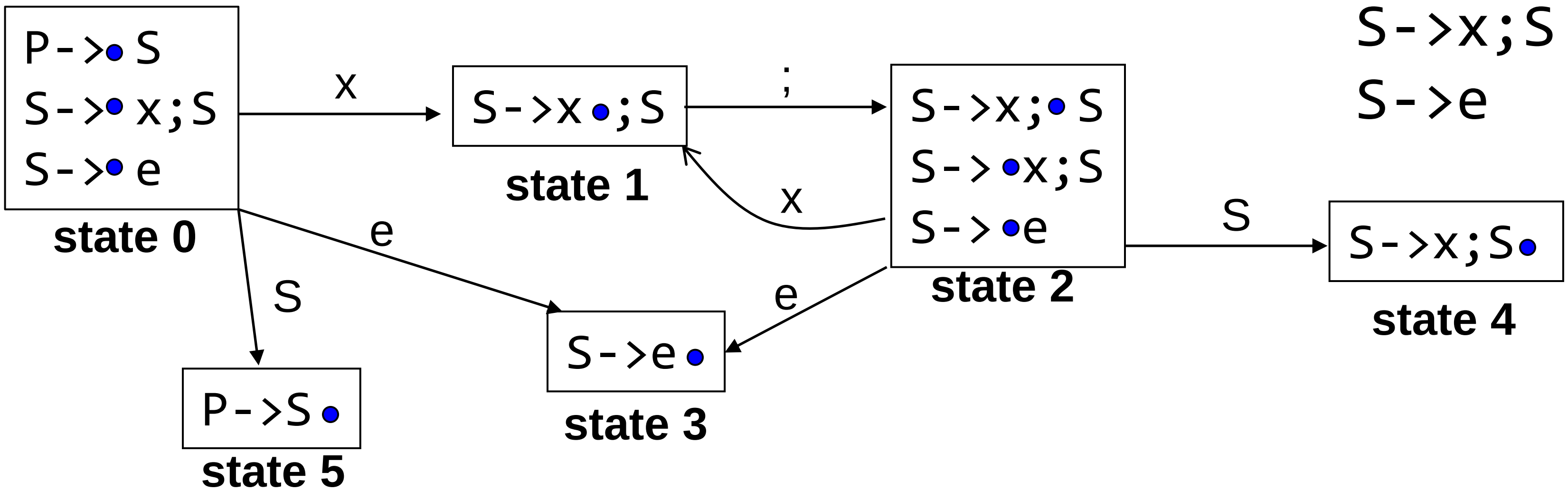
Grammar

P->S

S->x;S

S->e

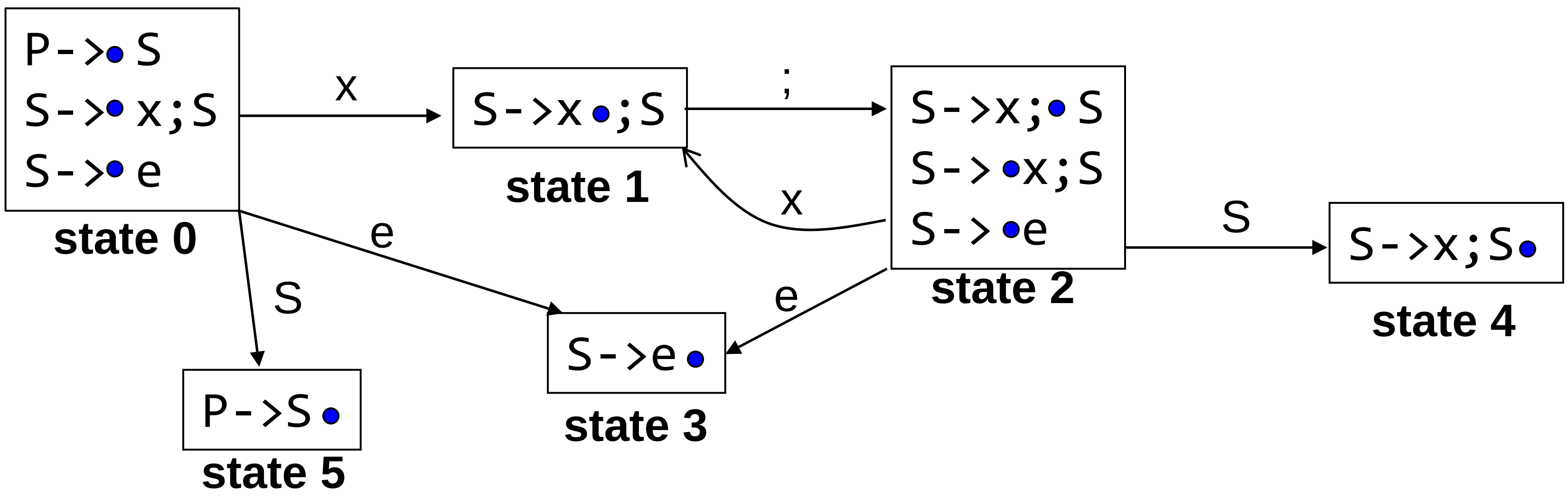
Compute successor (of state 0) under symbol S



Consider items (in state 0), where S is to the immediate right of Dot.
Advance Dot by one symbol. Cannot expand CFSM anymore.

Example: CFSM

- All states with Dot at extreme right become *reduce* states



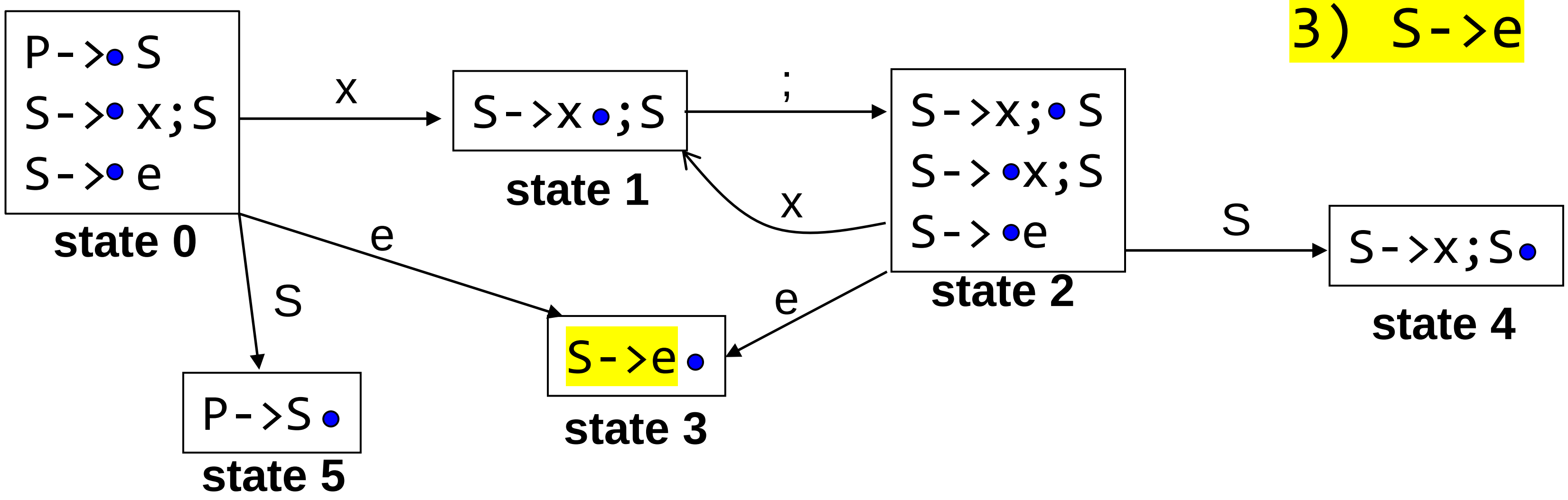
Example: CFSM

- All states with Dot at extreme right become *reduce* states

Grammar

1) P->S
2) S->x;S
3) S->e

Reduce 3



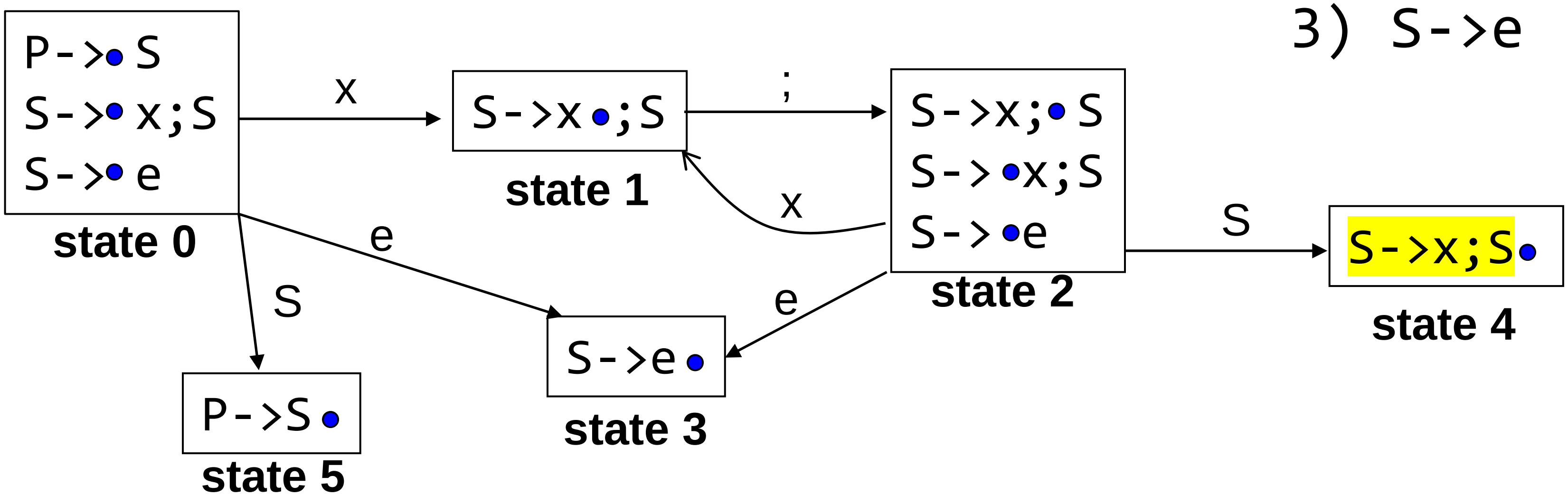
Example: CFSM

- All states with Dot at extreme right become *reduce* states

Grammar

1) P->S
2) S->x;S
3) S->e

Reduce 2



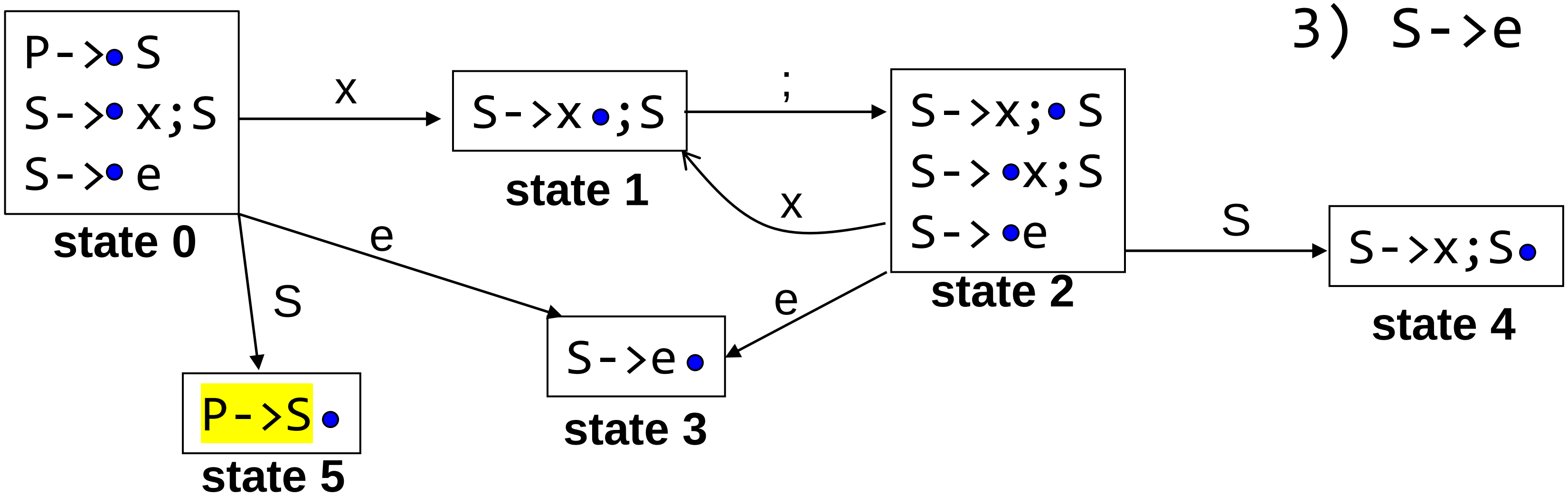
Example: CFSM

- All states with Dot at extreme right become *reduce* states

Accept

Grammar

1) P->S
2) S->x;S
3) S->e

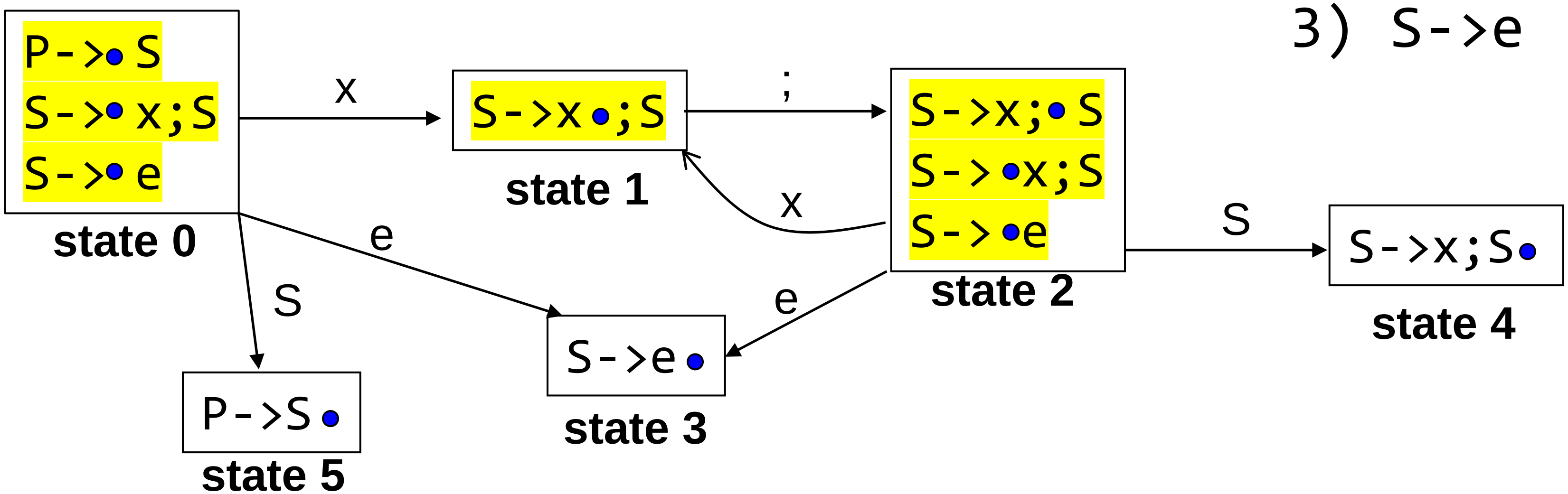


Example: CFSM

- Remaining states become *shift* states

Grammar

1) P->S
2) S->x;S
3) S->e



Conflicts

- What happens when a state has Dot at the extreme right for one item and in the middle for other items?

  Shift-reduce conflict

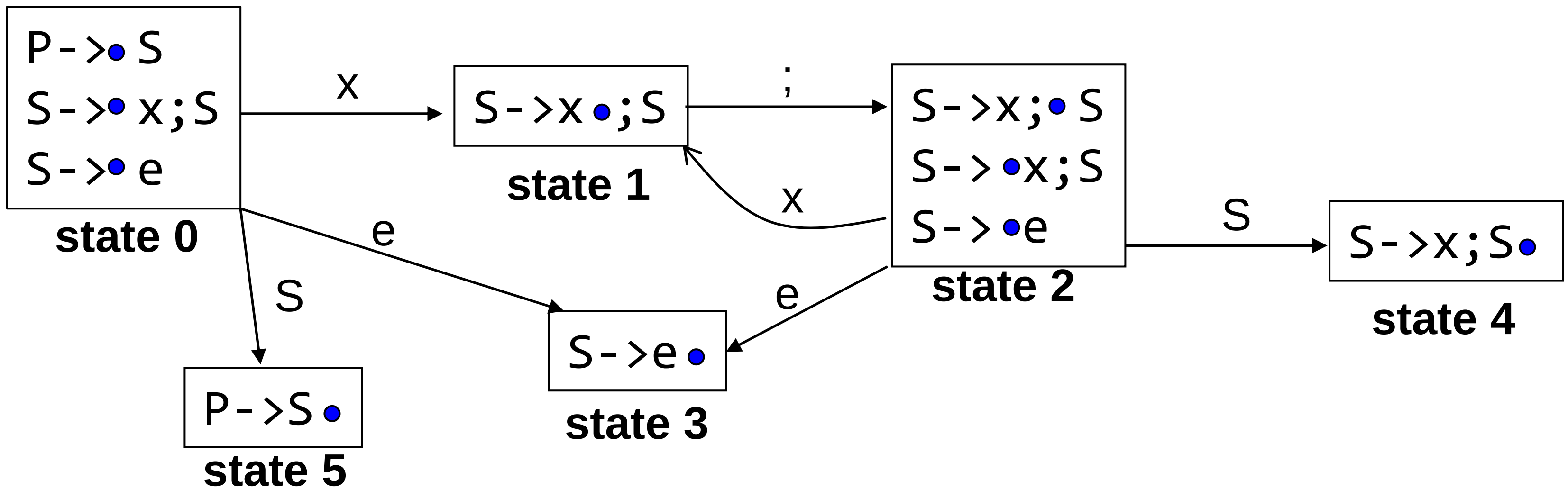
  Parser is unable to decide between shifting and reducing

- When Dot is at the extreme right for more than one items?

  Reduce-Reduce conflict

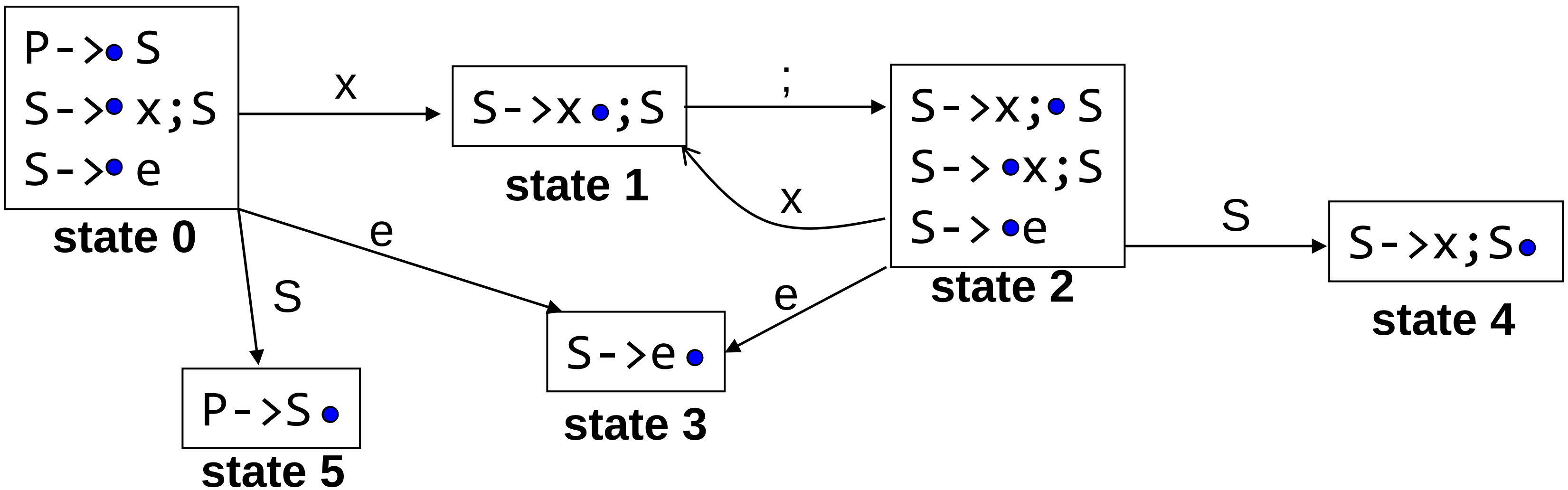
  Parser is unable to decide between which productions to choose for reducing

Example: goto table



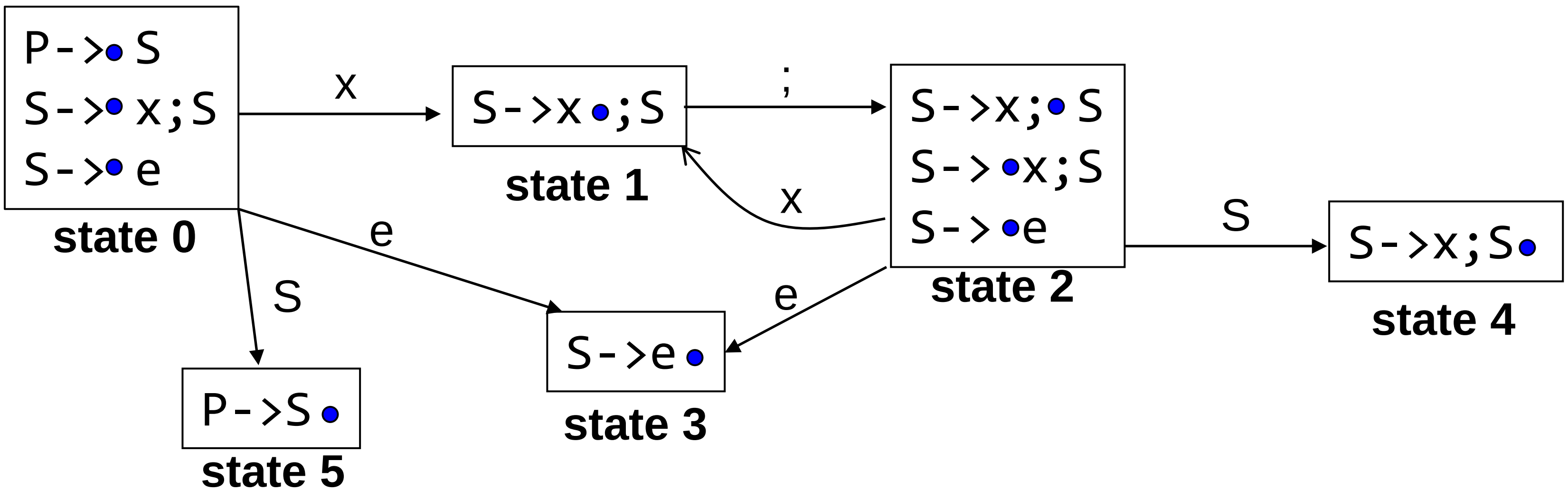
- construct transition table from CFSM.
  - Number of rows = number of states
  - Number of columns = number of symbols

Example: goto table



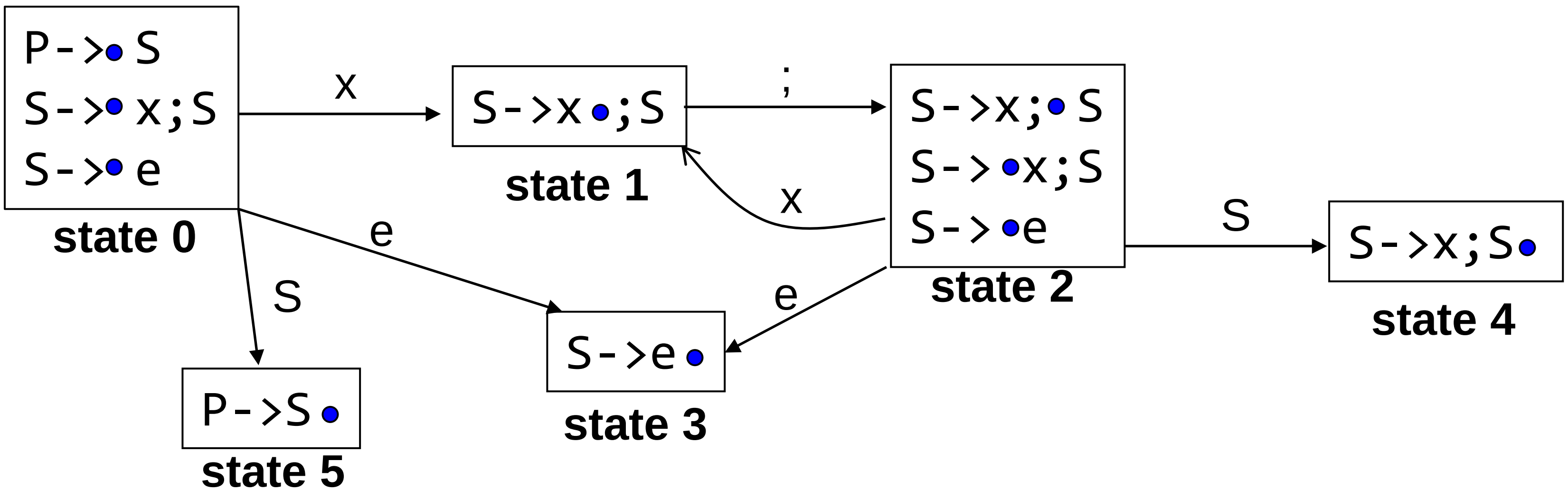
state	x	;	e	P	S
0	1		3		5
1		2			
2	1		3		4
3					
4					
5					

Example: action table



state	x
0	Shift
1	Shift
2	Shift
3	Reduce 3
4	Reduce 2
5	Accept

Example: action table



		Symbol					
		x	;	e	P	S	Action
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

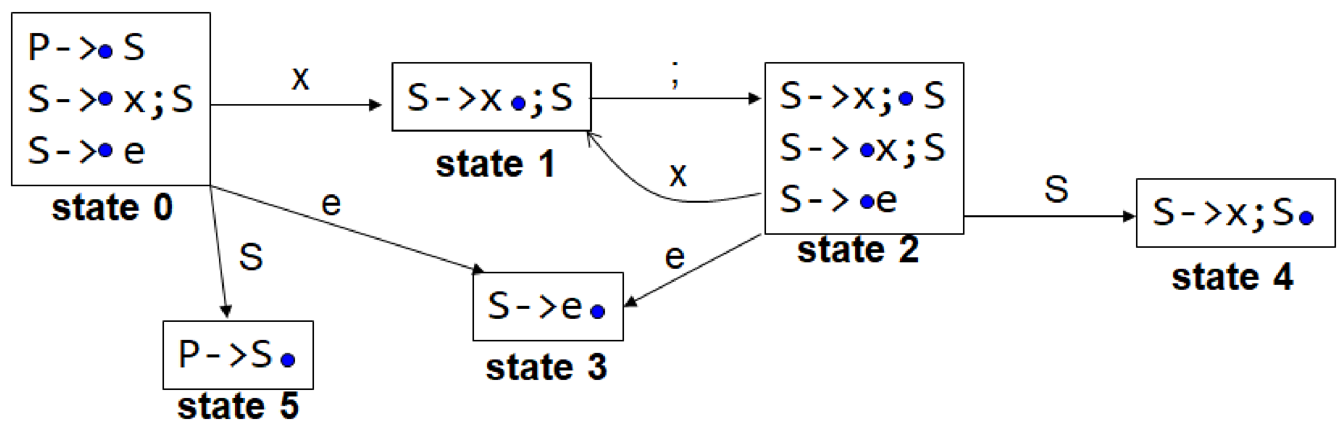
LR(0) Parsing

- Previous Example of LR Parsing was LR(0)
  - No (0) lookahead involved
  - Operate based on the parse stack state and with goto and action tables (How?)

LR(0) Parsing

- Assume: Parse stack contains α == saying that a e.g. prefix of `x;x` is seen in the input string

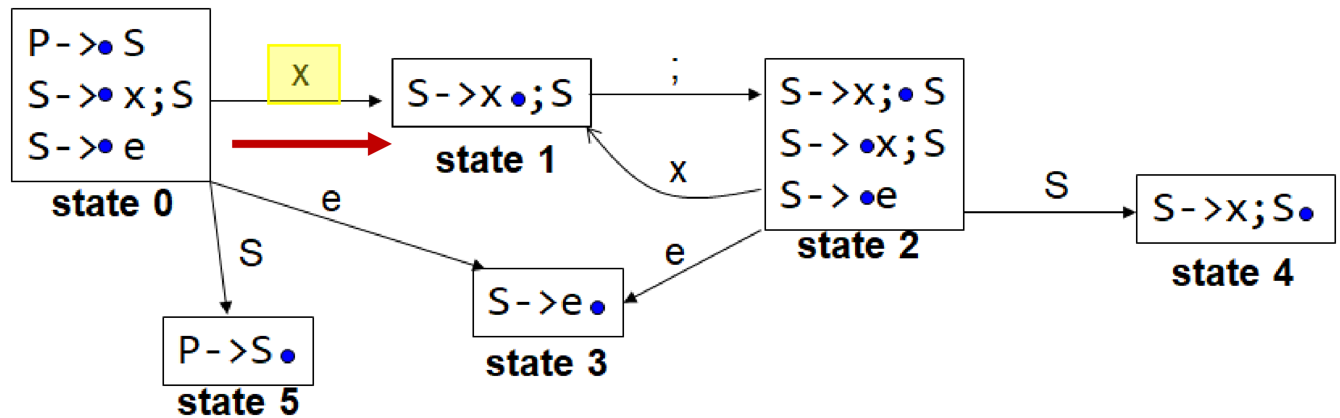
Parse Stack
0
0 1
0 1 2
0 1 2 1



LR(0) Parsing

- Assume: Parse stack contains α == saying that a prefix of `x;x` is seen in the input string

Parse Stack
0
0 1
0 1 2
0 1 2 1

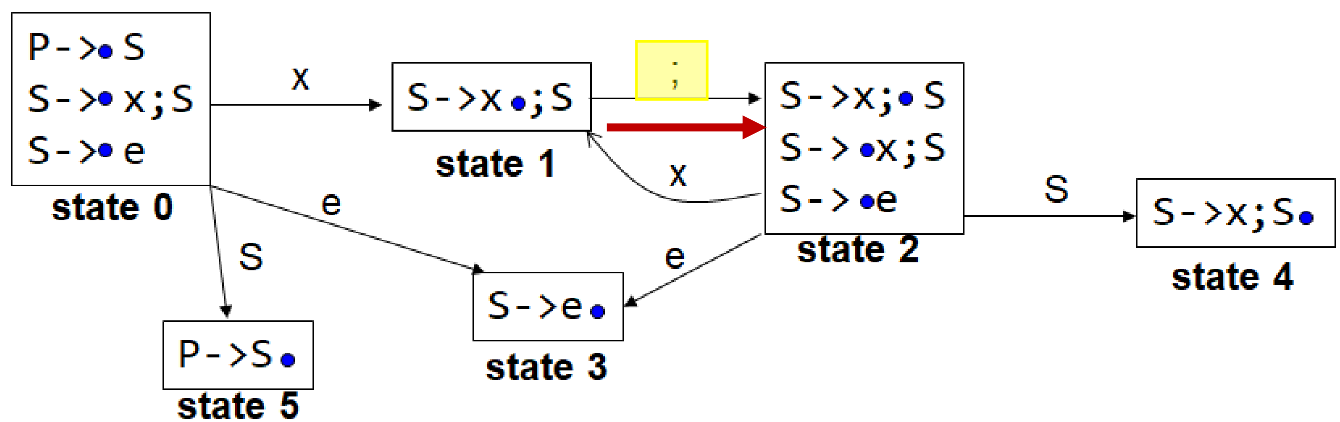


Go from state 0 to state 1 consuming x

LR(0) Parsing

- Assume: Parse stack contains α == saying that a prefix of `x;x` is seen in the input string

Parse Stack
0
0 1
0 1 2
0 1 2 1

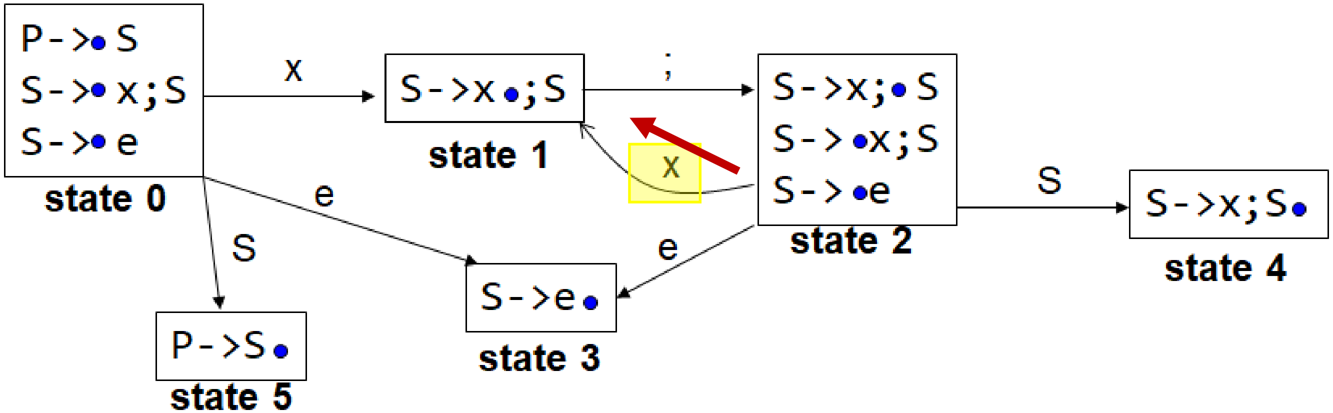


Go from state 1 to state 2 consuming ;

LR(0) Parsing

- Assume: Parse stack contains α == saying that a prefix of `x;x` is seen in the input string

Parse Stack
0
0 1
0 1 2
0 1 2 1



Go from state 2 to state 1 consuming x

LR(0) Parsing

- Assume: Parse stack contains α.

=> we are in some state s

LR(0) Parsing

- Assume: Parse stack contains α.

=> we are in some state s.

We reduce by $X \to \beta$ if state s contains $X \to \beta \bullet$

- Note: reduction is done based solely on the current state.

LR(0) Parsing

- Assume: Parse stack contains α.

=> we are in some state `s`.

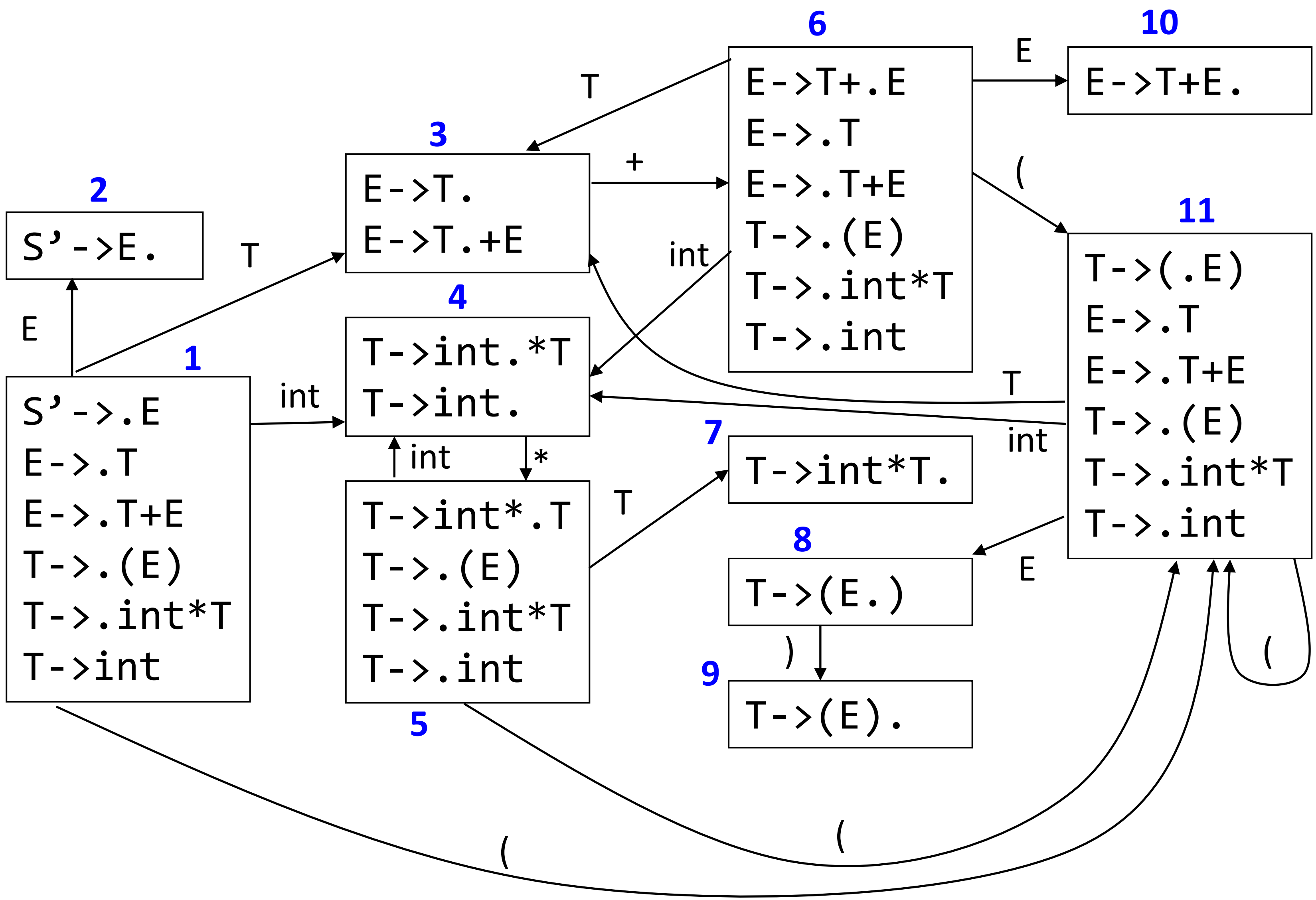
- Assume: Next input is t

We shift if s contains `X->β ● tω`

== `s` has a transition labelled `t`

LR(0) Parsing

- What if s contains $X \to \beta \bullet t\omega$ and $X \to \beta \bullet$?



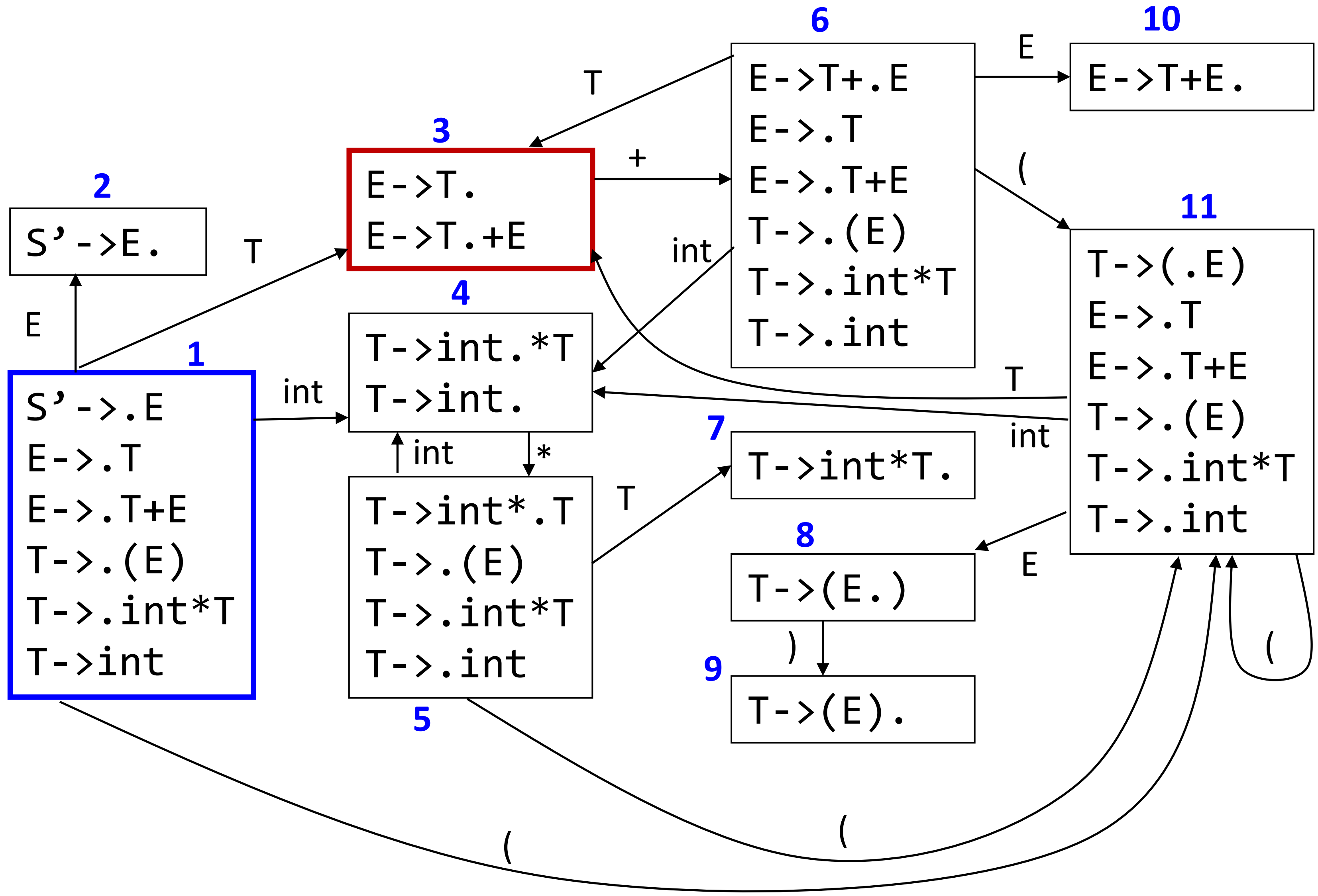
Conflicts or not?

SLR Parsing

- SLR Parsing improves the shift-reduce conflict states of LR(0):

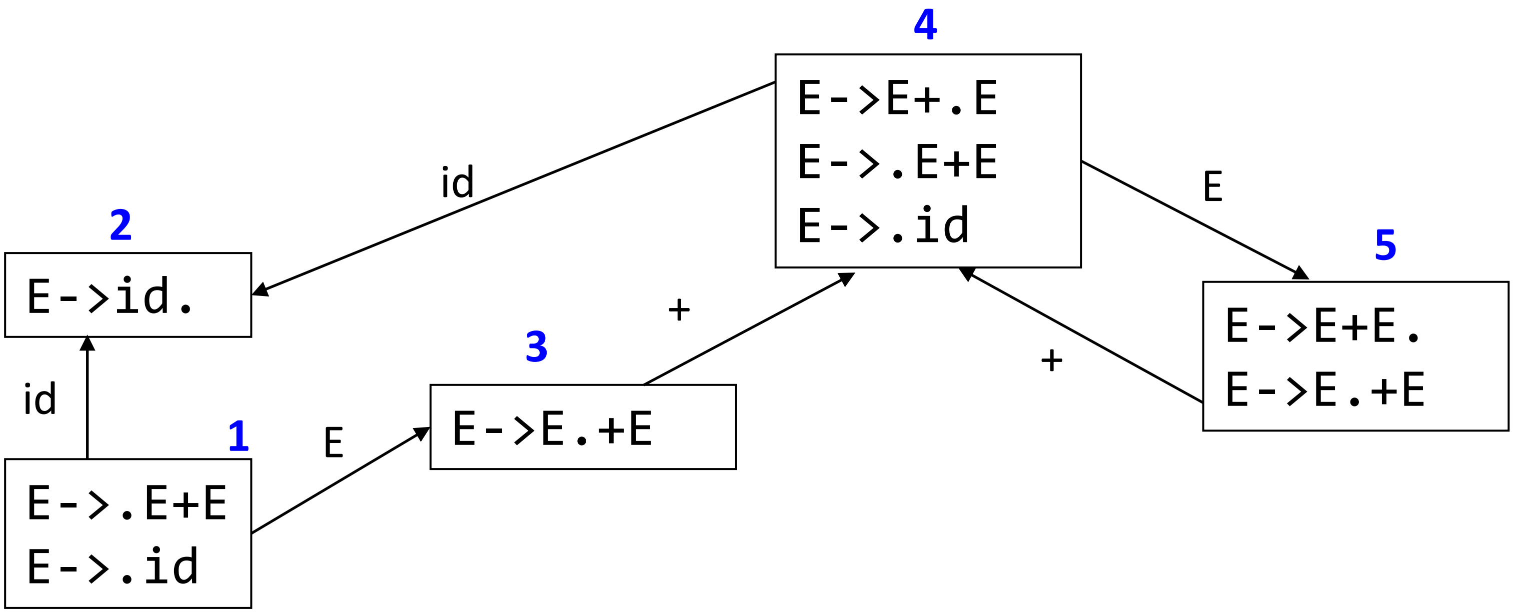
Reduce `X->β ●` `only if`

`t` ϵ Follow(X)



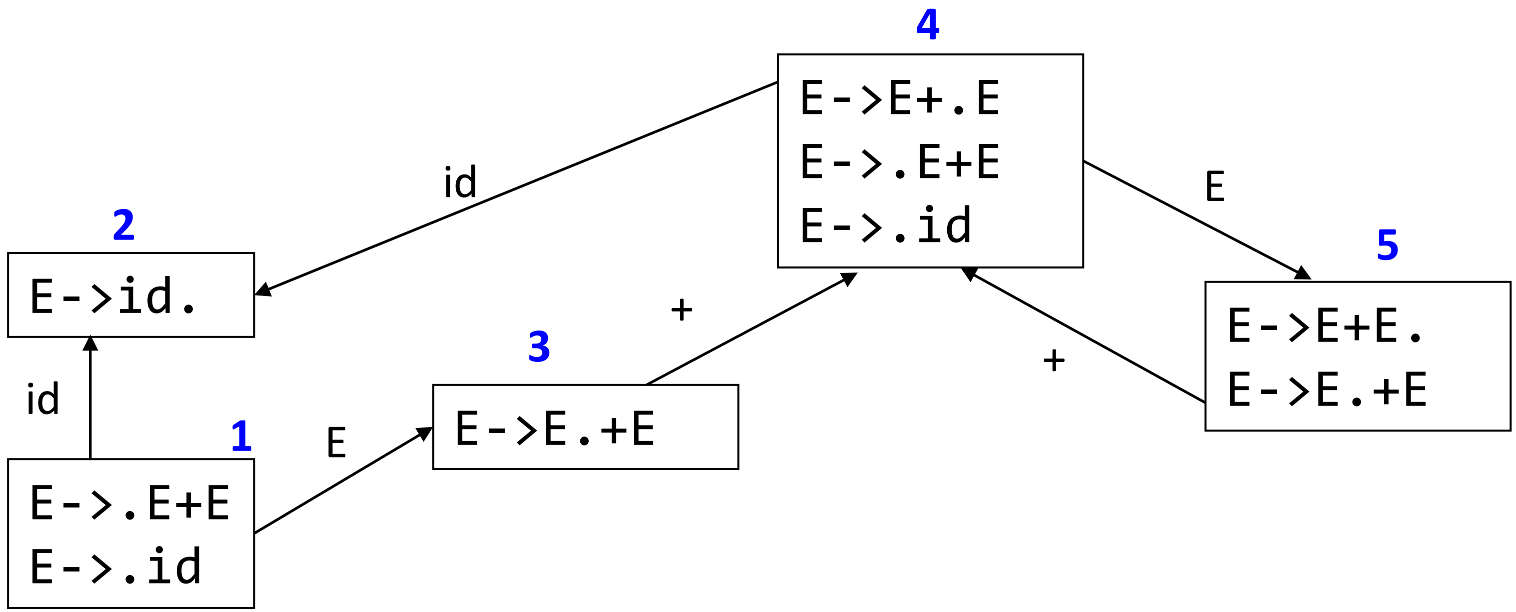
`Follow(E) = { $, ) }` => reduce by E->T. only if <u>next input</u> is $ or)

lookahead 1



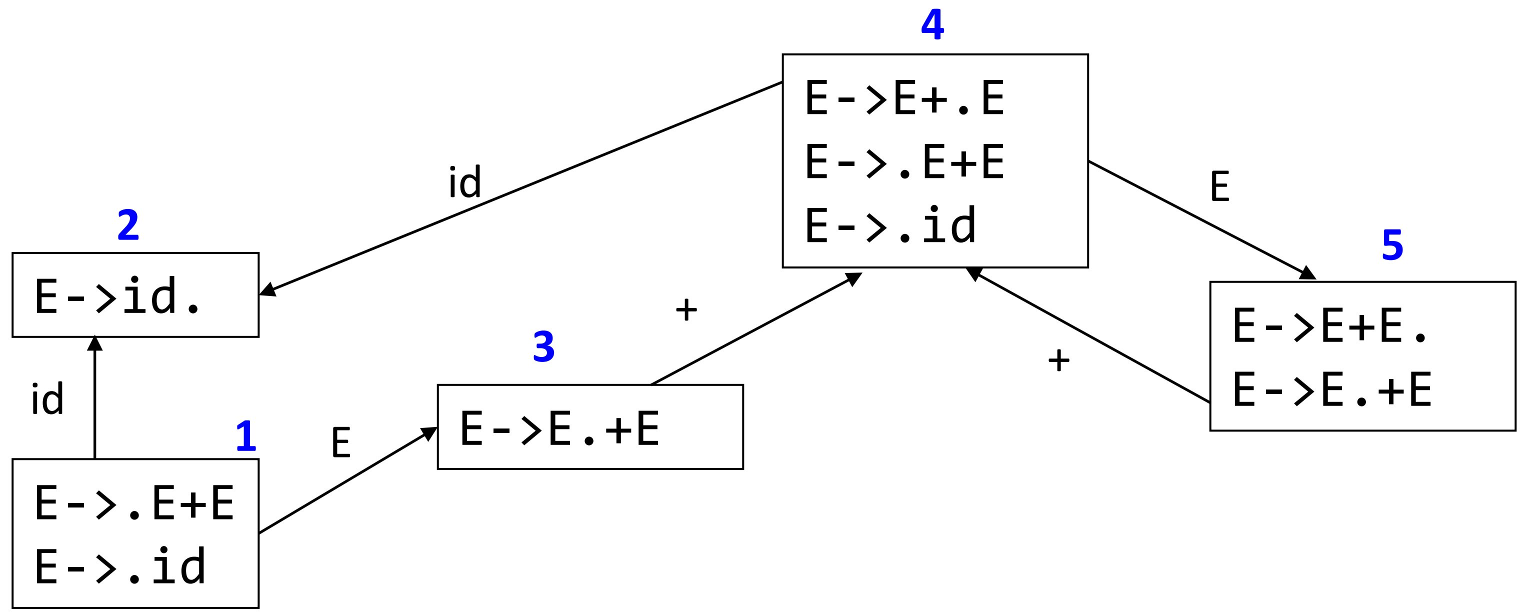
What about the grammar **E-> E + E | id** ?

LR(0)?



What about the grammar **E-> E + E | id** ?

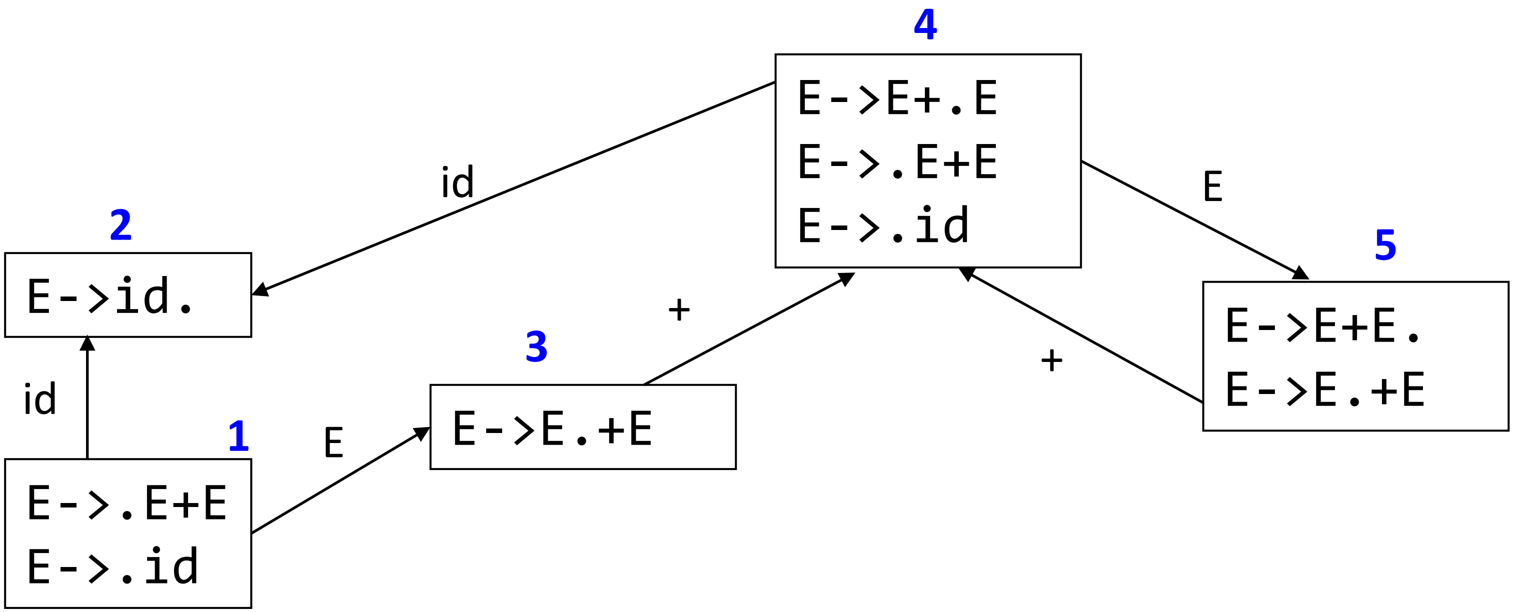
LR(0)? SLR(1)?



What about the grammar **E-> E + E | id** ?

LR(0)? SLR(1)?

Follow(E) = {+,$} => in state 5, reduce by E->E+E. only if <u>next input</u> is $ or +



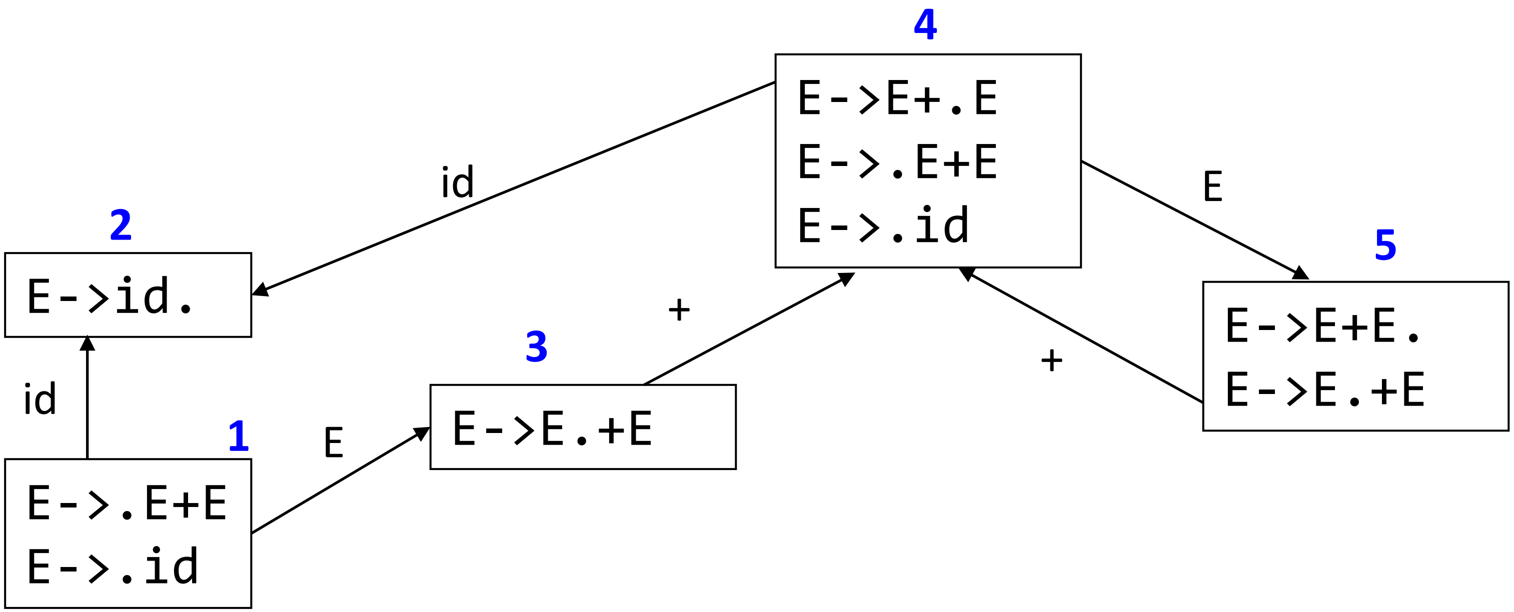
What about the grammar **E-> E + E | id** ?

LR(0)? SLR(1)?

Follow(E) = {+,$} => in state 5, <u>reduce by E->E+E. only if next input is $ or +</u>

But state 5 has E->E.+E (shift if next input is +)

Shift-reduce conflict!



What about the grammar **E-> E + E | id** ?

LR(0)? SLR(1)?

Follow(E) = {+,$} => in state 5, <u>reduce by E->E+E. only if next input is $ or +</u>

But state 5 has E->E.+E (shift if next input is +)
Shift-reduce conflict!

%left +

says reduce if the next input symbol is + i.e. prioritize rule E+E. over E.+E

LR(k) parsers

- LR(0) parsers
  - No lookahead
  - Predict which action to take by looking only at the symbols currently on the stack
- LR(k) parsers
  - Can look ahead *k* symbols
  - Most powerful class of deterministic bottom-up parsers
  - LR(1) and variants are the most common parsers

Top-down vs. Bottom-up parsers

- Top-down parsers expand the parse tree in *pre-order*
  - Identify parent nodes before the children
- Bottom-up parsers expand the parse tree in *post-order*
  - Identify children before the parents
- Notation:
  - LL(1): Top-down derivation with 1 symbol lookahead
  - LL(k): Top-down derivation with k symbols lookahead
  - LR(1): Bottom-up derivation with 1 symbol lookahead

slide courtesy: Milind Kulkarni