CS323: Compilers

Spring 2023

Week 4: Parsers - Top-Down Parsing (table-driven approach contd. and background concepts), Bottom-up parsing (use of goto and action tables)

Parsing using stack-based model

- How do we use the Parse Table constructed?

Top-Down Parsing - Example

`string: xacc$`

Stack	**Rem. Input**	**Action**
?	xacc$	?

What do you put on the stack?

Top-Down Parsing - Example

`string: xacc$`

Stack	**Rem. Input**	**Action**
?	xacc$	?

What do you put on the stack? – strings that you derive

Top-Down Parsing - Example

`string: xacc$`

Stack*	**Rem. Input**	**Action**
S	xacc$	?

Top-down parsing. So, start with S.

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

`string: xacc$`

Stack*	**Rem. Input**	**Action**
`S`	`xacc$`	?

Top-down parsing. So, start with S.

What action do you take when stack-top has symbol S and the string to be matched has terminal x in front?

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$

Top-down parsing. So, start with S.

What action do you take when stack-top has symbol S and the string to be matched has terminal x in front? – consult parse table

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

`string: xacc$`

Stack*	**Rem. Input**	**Action**
`S`	`xacc$`	`Predict(1) S->ABc$`
`ABc$`		

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	

What action do you take when stack-top has symbol A and the string to be matched has terminal x in front? – consult parse table

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA

What action do you take when stack-top has symbol A and the string to be matched has terminal x in front? – consult parse table

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$		

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

`string: xacc$`

Stack*	**Rem. Input**	**Action**
`S`	`xacc$`	`Predict(1) S->ABc$`
`ABc$`	`xacc$`	`Predict(2) A->xaA`
`xaABc$`	`xacc$`	**?**

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	?

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	Predict(4) A->c

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	Predict(4) A->c
cBc$		

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	Predict(4) A->c
cBc$	cc$	?

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	Predict(4) A->c
cBc$	cc$	match(c)

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	Predict(4) A->c
cBc$	cc$	match(c)
Bc$	c$	?

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

string: xacc$

Stack*	Rem. Input	Action
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	Predict(4) A->c
cBc$	cc$	match(c)
Bc$	c$	Predict(6) B->λ

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	Predict(4) A->c
cBc$	cc$	match(c)
Bc$	c$	Predict(6) B->λ
c$		

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	Predict(4) A->c
cBc$	cc$	match(c)
Bc$	c$	Predict(6) B->λ
c$	c$	?

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	Predict(4) A->c
cBc$	cc$	match(c)
Bc$	c$	Predict(6) B->λ
c$	c$	match(c)

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	Predict(4) A->c
cBc$	cc$	match(c)
Bc$	c$	Predict(6) B->λ
c$	c$	match(c)
$	$	Done!

* Stack top is on the left-side (first symbol) of the column

Identifying LL(1) Grammar

- What we saw was an example of LL(1) Grammar
  - Scan input **L**eft-to-right, produce **L**eft-most derivation with 1 symbol look-ahead

Identifying LL(1) Grammar

- What we saw was an example of LL(1) Grammar
  - Scan input **L**eft-to-right, produce **L**eft-most derivation with 1 symbol look-ahead
- Not all Grammars are LL(1)

A Grammar is LL(1) iff for a production A -> α | β, where

α and β are distinct:

1. For no terminal `a` do both α and β derive strings beginning with `a` (i.e. no common prefix)
2. At most one of α and β can derive an empty string
3. If $\beta \overset{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in `Follow(A)`. If $\alpha \overset{*}{\Rightarrow} \epsilon$, then β does not derive any string beginning with a terminal in `Follow(A)`

Example (Left Factoring)

- Consider

<stmt> → if <expr> then <stmt list> endif

<stmt> → if <expr> then <stmt list> else <stmt list> endif

- This is not LL(1) (why?)
- We can turn this in to

<stmt> → if <expr> then <stmt list> <if suffix>

<if suffix> → endif

<if suffix> → else <stmt list> endif

Example (Left Factoring)

- Consider

  <stmt> → if <expr> then <stmt list> endif

  <stmt> → if <expr> then <stmt list> else <stmt list> endif

- This is not LL(1) (why?)
- We can turn this in to

  <stmt> → if <expr> then <stmt list> <if suffix>

  <if suffix> → endif

  <if suffix> → else <stmt list> endif

Left Factoring

A -> α β | α μ

⇓

A -> α N
N -> β
N -> μ

Left recursion

- *Left recursion* is a problem for LL(1) parsers
  - LHS is also the first symbol of the RHS
- Consider:

  E → E + T

- What would happen with the stack-based algorithm?

Left recursion

- *Left recursion* is a problem for LL(1) parsers
  - LHS is also the first symbol of the RHS
- Consider:

  E → E + T

- What would happen with the stack-based algorithm?

  E
  E + T
  E + T + T
  E + T + T + T

Eliminating Left Recursion

A -> A α | β

⇓

A -> NT
N -> β
T -> αT
T -> λ

Eliminating Left Recursion

E -> E + T | T

E -> E1 Etail
E1 -> T
Etail -> + T Etail
Etail -> λ

LL(k) parsers

- Can look ahead more than one symbol at a time
  - *k*-symbol lookahead requires extending first and follow sets
  - 2-symbol lookahead can distinguish between more rules:

    A → ax | ay
- More lookahead leads to more powerful parsers
- What are the downsides?

LL(k)? - Example

1) S - > E
2) E -> (E+E)
3) E -> (E-E)
4) E -> x

string: ((x+x))$

Stack*	**Rem. Input**	**Action**
S	((x+x))$	Predict(1) S->E
E		Predict(2) or Predict(3)?

LL(1)

	(	**+**	**-**	**)**	**x**
S	1				1
E	2,3				4

LL(2)

	((	**+(**	**-(**	**)$**	**(x**
S	1				1
E	2,3				4

Are all grammars LL(k)?

- No! Consider the following grammar:

$$
\begin{aligned}
S &\rightarrow E \\
E &\rightarrow (E + E) \\
E &\rightarrow (E - E) \\
E &\rightarrow x
\end{aligned}
$$

- When parsing E, how do we know whether to use rule 2 or 3?
  - Potentially unbounded number of characters before the distinguishing '+' or '–' is found
  - No amount of lookahead will help!

In real languages?

- Consider the if-then-else problem
- `if x then y else z`
- Problem: else is optional
- `if a then if b then c else d`
  - Which `if` does the `else` belong to?
- This is analogous to a "bracket language": $[^i\ ]^j$ $(i \geq j)$

$S \rightarrow [\ S\ C$
$S \rightarrow \lambda$
$C \rightarrow\ ]$
$C \rightarrow \lambda$

[[ ] can be parsed: $SS\lambda C$ or $SSC\lambda$
(it's ambiguous!)

Solving the if-then-else problem

- The ambiguity exists at the language level. To fix, we need to define the semantics properly
  - "] matches nearest unmatched ["
  - This is the rule C uses for if-then-else
  - What if we try this?

S → [S
S → S1
S1 → [S1]
S1 → λ

This grammar is still not LL(1) (or LL(k) for any k!)

Two possible fixes

- If there is an ambiguity, prioritize one production over another
  - e.g., if C is on the stack, always match "]" before matching "λ"

S → [S C
S → λ
C →]
C → λ

- Another option: change the language!
  - e.g., all if-statements need to be closed with an endif

S → if S E
S → other
E → else S endif
E → endif

Parsing if-then-else

- What if we don’t want to change the language?
  - C does not require { } to delimit single-statement blocks
- To parse if-then-else, *we need to be able to look ahead at the entire rhs of a production* before deciding which production to use
  - In other words, we need to determine how many “]” to match before we start matching “[”s
- *LR parsers* can do this!

Bottom-up Parsing

- More general than top-down parsing
- Used in most parser-generator tools
- Need not have left-factored grammars (i.e. can have left recursion)
- E.g. can work with the bracket language

Bottom-up Parsing

- Reduce a string to start symbol by reverse ‘inverting’ productions

Bottom-up Parsing

- Reduce a string to start symbol by reverse ‘inverting’ productions

```
id * id + id
```

```
E -> T + E
E -> T
T -> id * T
T -> id
```

Bottom-up Parsing

- Reduce a string to start symbol by reverse 'inverting' productions

```
E -> T + E
E -> T
T -> id * T
T -> id
```

```
id * id + id
id * T + id
```

Bottom-up Parsing

- Reduce a string to start symbol by reverse ‘inverting’ productions

```
id * id + id
id * T + id
T + id
```

```
E -> T + E
E -> T
T -> id * T
T -> id
```

Bottom-up Parsing

- Reduce a string to start symbol by reverse ‘inverting’ productions

```
E -> T + E
E -> T
T -> id * T
T -> id
```

```
id * id + id
id * T + id
T + id
T + T
```

Bottom-up Parsing

- Reduce a string to start symbol by reverse ‘inverting’ productions

```
E -> T + E
E -> T
T -> id * T
T -> id
```

```
id * id + id
id * T + id
T + id
T + T
T + E
```

Bottom-up Parsing

- Reduce a string to start symbol by reverse ‘inverting’ productions

```
E -> T + E
E -> T
T -> id * T
T -> id
```

```
id * id + id
id * T + id
T + id
T + T
T + E
E
```

Bottom-up Parsing

- Reduce a string to start symbol by reverse ‘inverting’ productions

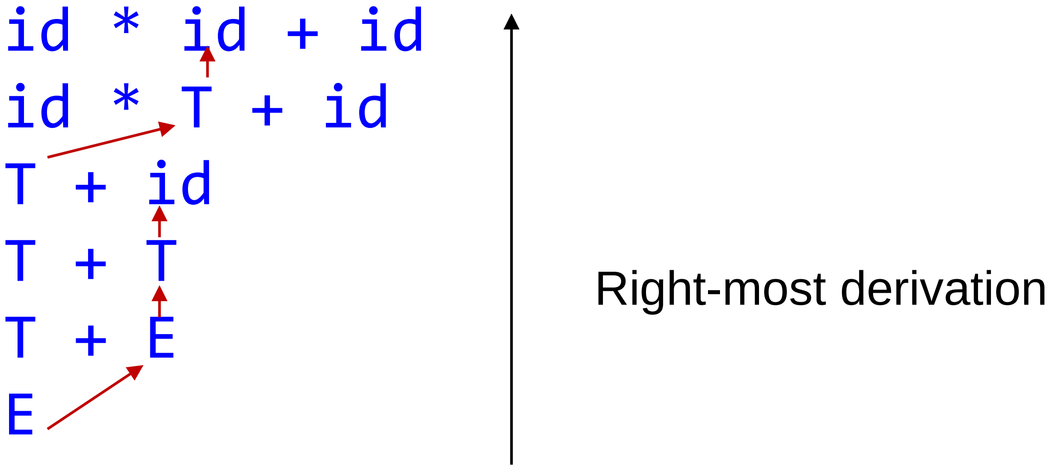
```
E -> T + E
E -> T
T -> id * T
T -> id
```

```
id * id + id
id * T + id
T + id
T + T
T + E
E
```

Bottom-up Parsing

- Reduce a string to start symbol by reverse ‘inverting’ productions

```
E -> T + E
E -> T
T -> id * T
T -> id
```



```
id * id + id
id * T + id
T + id
T + T
T + E
E
```

Right-most derivation

Bottom-up Parsing

- Scan the input left-to-right and shift tokens – put them on the stack.

```
| id * id + id

id | * id + id

id * | id + id

id * id | + id
```

```
E -> T + E
E -> T
T -> id * T
T -> id
```

Bottom-up Parsing

- Replace a set of symbols at the top of the stack that are RHS of a production. Put the LHS of the production on stack – Reduce

```
| id * id + id

id | * id + id

id * | id + id

id * id | + id
```

```
E -> T + E
E -> T
T -> id * T
T -> id
```

Bottom-up Parsing

- Did not discuss when and why a particular production was chosen

```
id * id + id
id * T + id
```

```
E -> T + E
E -> T
T -> id * T
T -> id
```

- *i.e. why replace the id highlighted in input string?*

LR Parsers

- Parser which does a Left-to-right, Right-most derivation
  - Rather than parse top-down, like LL parsers do, parse bottom-up, starting from leaves
- Basic idea: put tokens on a stack until an entire production is found
- Issues:
  - Recognizing the endpoint of a production
  - Finding the length of a production (RHS)
  - Finding the corresponding nonterminal (the LHS of the production)

Data structures

- At each state, given the next token,
  - A *goto table* defines the successor state
  - An *action table* defines whether to
    - *shift* – put the next state and token on the stack
    - *reduce* – an RHS is found; process the production
    - *terminate* – parsing is complete

Simple example

1. P → S
2. S → x ;S
3. S → e

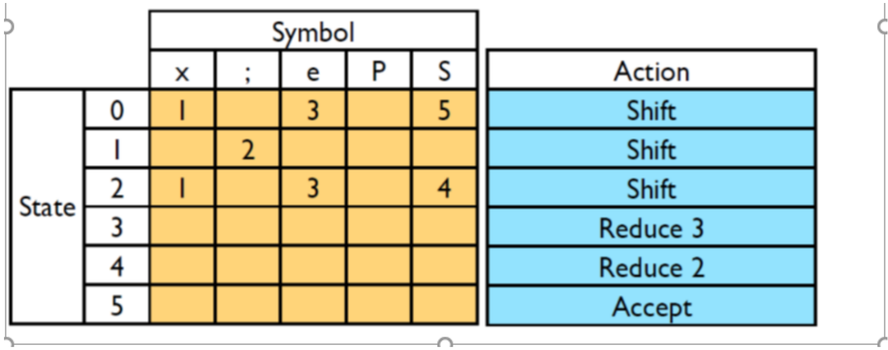
		Symbol					
		x	;	e	P	S	Action
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



		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

Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	?

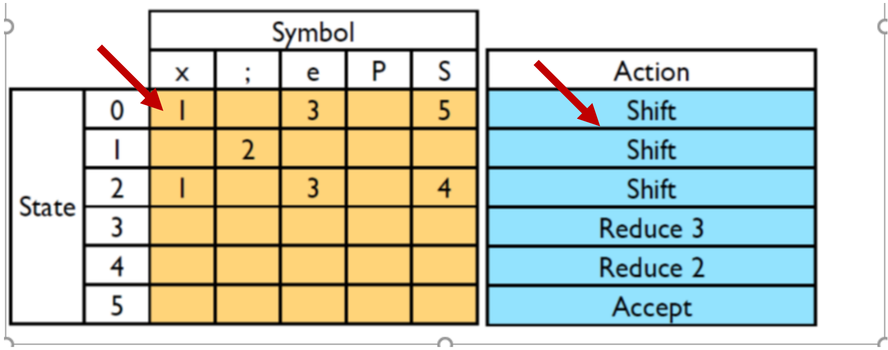
Start with state 0

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



		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

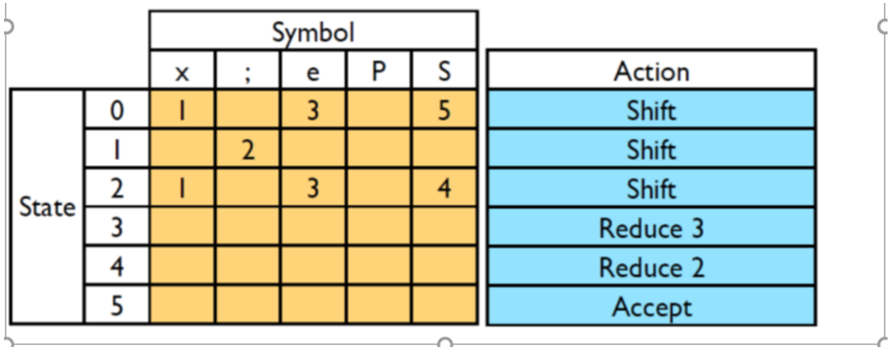
Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



		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

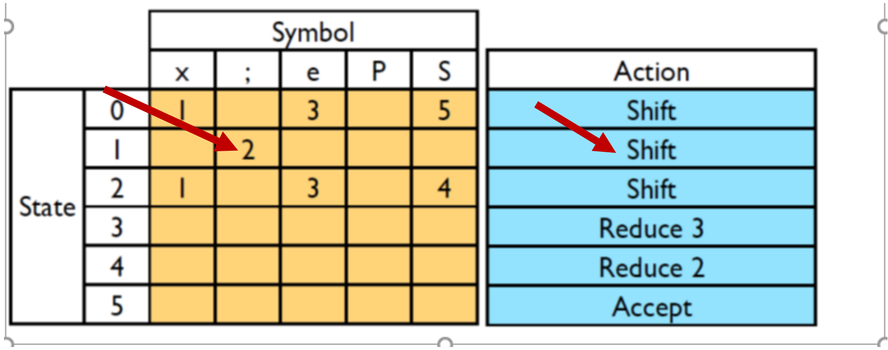
Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	?

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



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

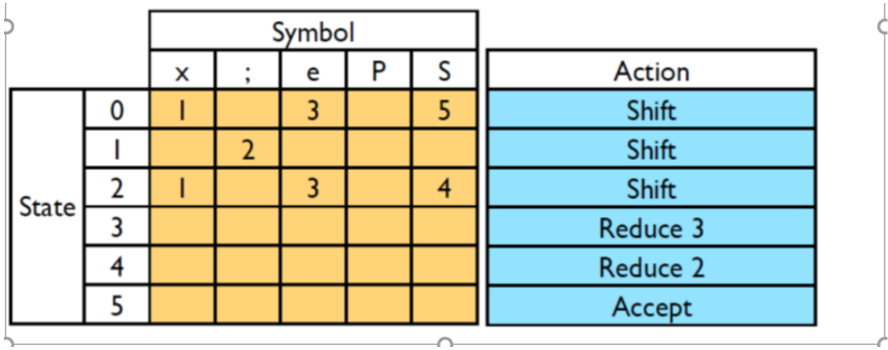
Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



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

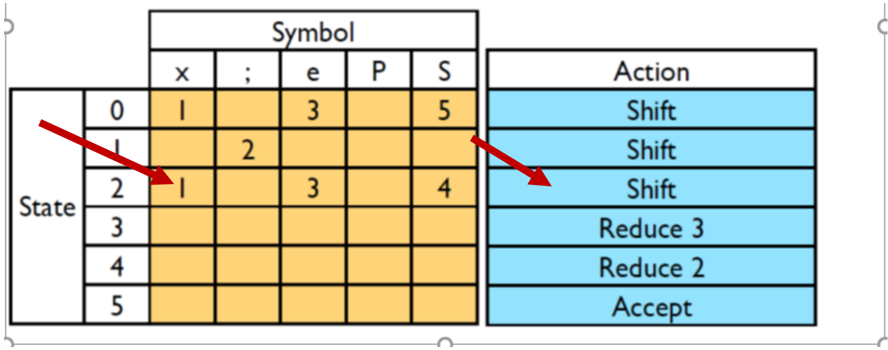
Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	**?**

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



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

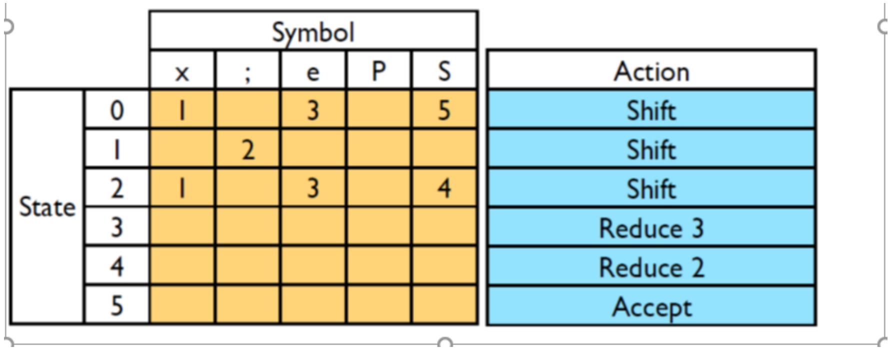
Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



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

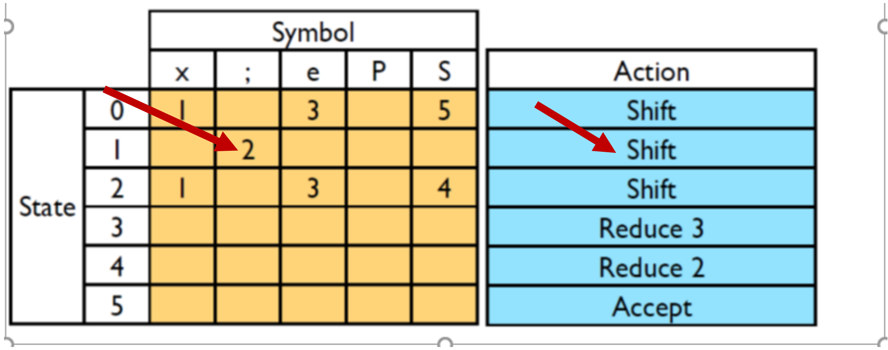
Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)
4	0 1 2 1	;e	?

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



		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

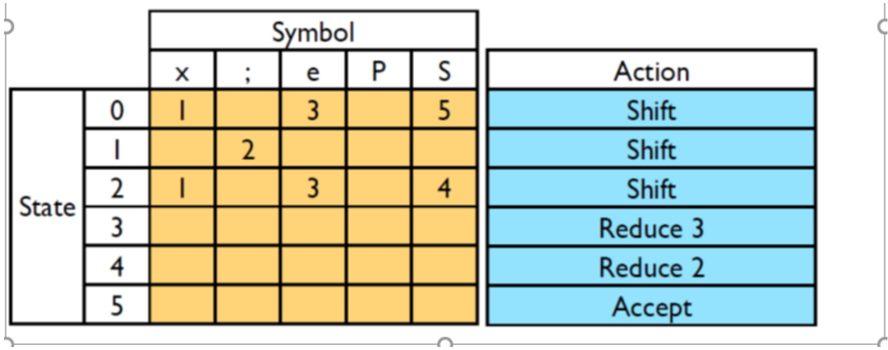
Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)
4	0 1 2 1	;e	Shift(2)

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



		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

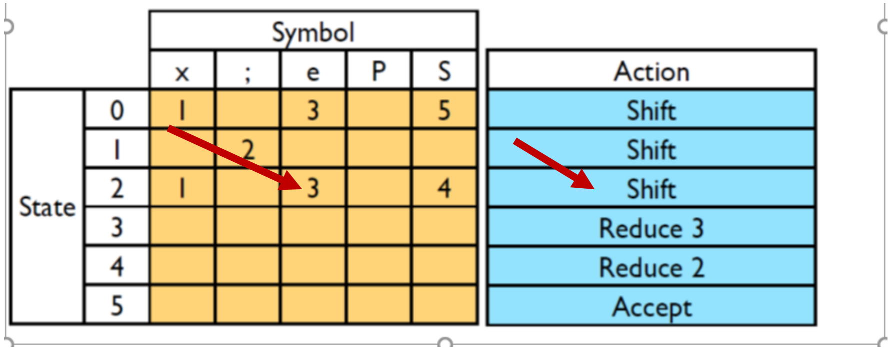
Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)
4	0 1 2 1	;e	Shift(2)
5	0 1 2 1 2	e	?

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



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

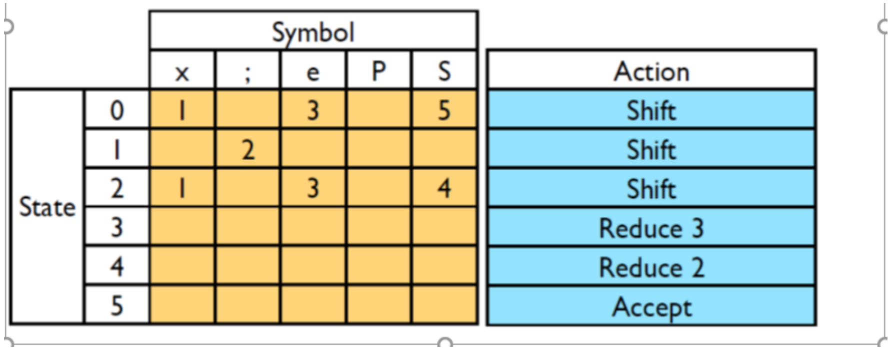
Step	Parse Stack	Rem. Input	Parser Action
1	0	`x;x;e`	Shift(1)
2	0 1	`;x;e`	Shift(2)
3	0 1 2	`x;e`	Shift(1)
4	0 1 2 1	`;e`	Shift(2)
5	0 1 2 1 2	`e`	Shift(3)

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



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

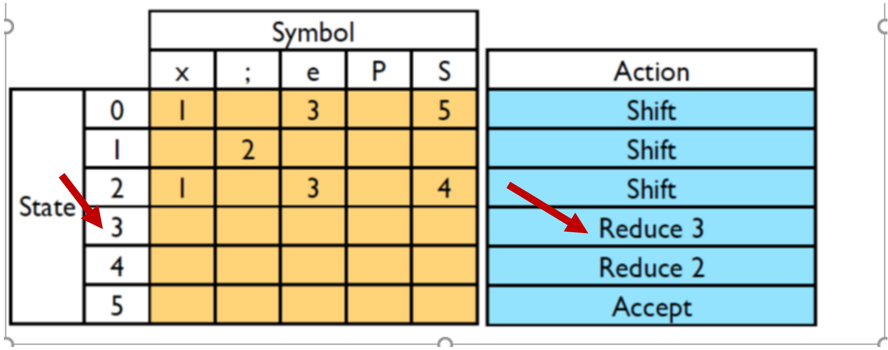
Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)
4	0 1 2 1	;e	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		?

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



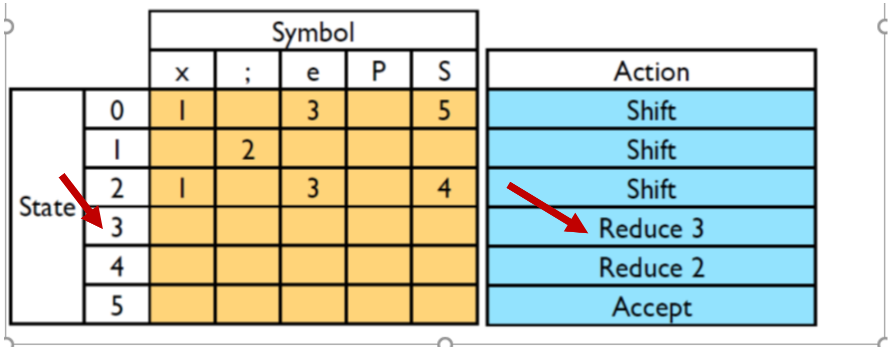
		Symbol					Action
		x	;	e	P	S	
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

Step	**Parse Stack**	**Rem. Input**	**Parser Action**
1	0	`x;x;e`	Shift(1)
2	0 1	`;x;e`	Shift(2)
3	0 1 2	`x;e`	Shift(1)
4	0 1 2 1	`;e`	Shift(2)
5	0 1 2 1 2	`e`	Shift(3)
6	0 1 2 1 2 3		Reduce 3

Example

I) P->S
II) S->x;S
III) S->e

Input string
x;x;e



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

Step	Parse Stack	Rem. Input	Parser Action
1	0	`x;x;e`	Shift(1)
2	0 1	`;x;e`	Shift(2)
3	0 1 2	`x;e`	Shift(1)
4	0 1 2 1	`;e`	Shift(2)
5	0 1 2 1 2	`e`	Shift(3)
6	0 1 2 1 2 3		Reduce 3
7	0 1 2 1 2		

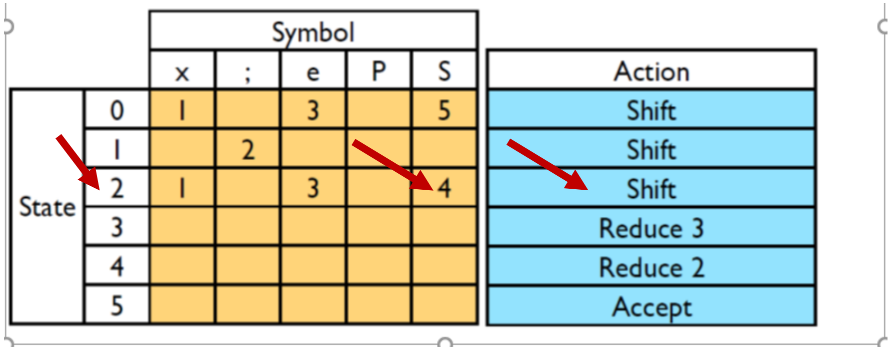
- *Look at rule III and pop 1 symbol of the stack because RHS of rule III has just 1 symbol*

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



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

Step	Parse Stack	Rem. Input	Parser Action
1	0	`x;x;e`	Shift(1)
2	0 1	`;x;e`	Shift(2)
3	0 1 2	`x;e`	Shift(1)
4	0 1 2 1	`;e`	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		Reduce 3
7	0 1 2 1 2		

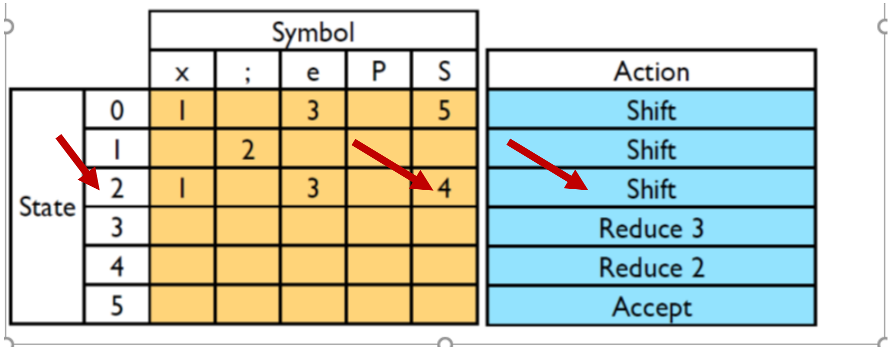
- *Now stack top has symbol 2 and LHS of rule III has S (imagine you saw S at input). Consult goto and action table.*

Example

I) P->S
II) S->x;S
III) S->e

Input string

`x;x;e`



<table>
<tr><td></td><td></td><td colspan="5">Symbol</td><td></td></tr>
<tr><td></td><td></td><td>x</td><td>;</td><td>e</td><td>P</td><td>S</td><td>Action</td></tr>
<tr><td>State</td><td>0</td><td>1</td><td></td><td>3</td><td></td><td>5</td><td>Shift</td></tr>
<tr><td></td><td>1</td><td></td><td>2</td><td></td><td></td><td></td><td>Shift</td></tr>
<tr><td></td><td>2</td><td>1</td><td></td><td>3</td><td></td><td>4</td><td>Shift</td></tr>
<tr><td></td><td>3</td><td></td><td></td><td></td><td></td><td></td><td>Reduce 3</td></tr>
<tr><td></td><td>4</td><td></td><td></td><td></td><td></td><td></td><td>Reduce 2</td></tr>
<tr><td></td><td>5</td><td></td><td></td><td></td><td></td><td></td><td>Accept</td></tr>
</table>

Step	**Parse Stack**	**Rem. Input**	**Parser Action**
1	0	`x;x;e`	Shift(1)
2	0 1	`;x;e`	Shift(2)
3	0 1 2	`x;e`	Shift(1)
4	0 1 2 1	`;e`	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		

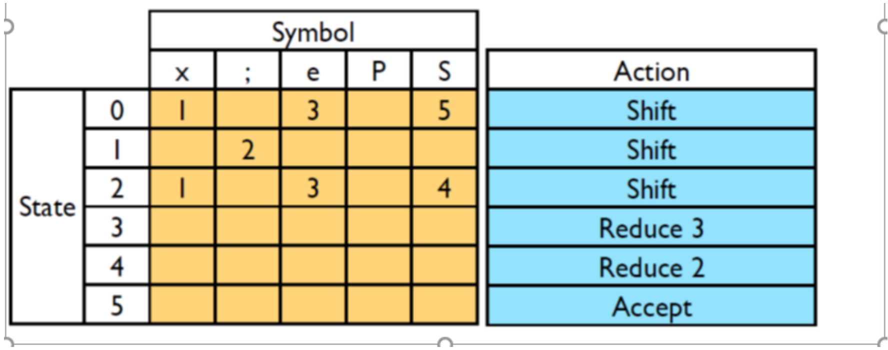
- *Now stack top has symbol 2 and LHS of rule III has S (imagine you saw S at input). Consult goto and action table. Shift(4)*

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



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

Step	Parse Stack	Rem. Input	Parser Action
1	0	`x;x;e`	Shift(1)
2	0 1	`;x;e`	Shift(2)
3	0 1 2	`x;e`	Shift(1)
4	0 1 2 1	`;e`	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		**?**

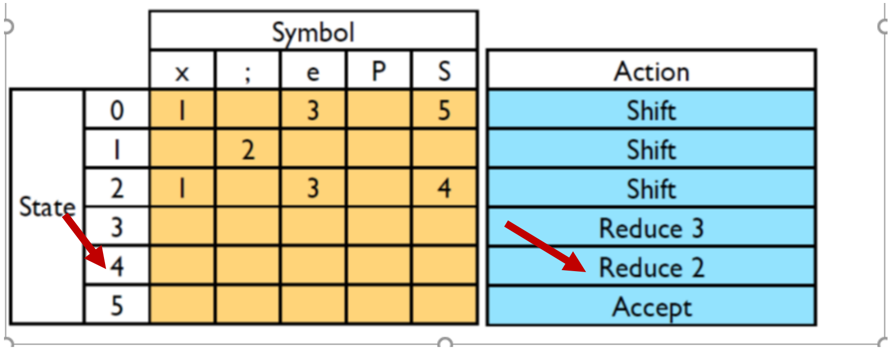
- *Now stack top has symbol 2 and LHS of rule III has S (imagine you saw S at input). Consult goto and action table. Shift(4)*

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



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

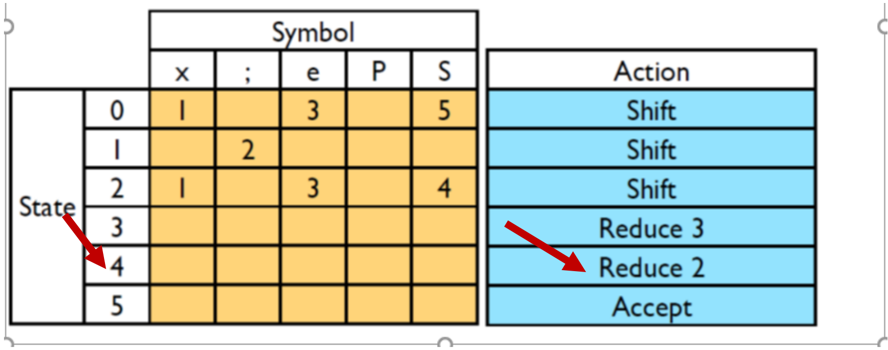
Step	Parse Stack	Rem. Input	Parser Action
1	0	`x;x;e`	Shift(1)
2	0 1	`;x;e`	Shift(2)
3	0 1 2	`x;e`	Shift(1)
4	0 1 2 1	`;e`	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		Reduce 2

Example

I) P->S
II) S->x;S
III) S->e

<u>Input string</u>

x;x;e



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

Step	Parse Stack	Rem. Input	Parser Action
1	0	`x;x;e`	Shift(1)
2	0 1	`;x;e`	Shift(2)
3	0 1 2	`x;e`	Shift(1)
4	0 1 2 1	`;e`	Shift(2)
5	0 1 2 1 2	`e`	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		Reduce 2
8	0 1 2		

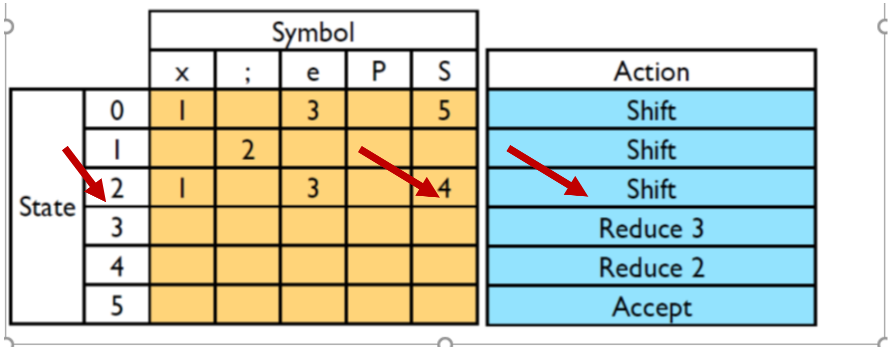
- *Look at rule II and pop 3 symbols of the stack because RHS of rule II has 3 symbols*

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



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

Step	Parse Stack	Rem. Input	Parser Action
1	0	`x;x;e`	Shift(1)
2	0 1	`;x;e`	Shift(2)
3	0 1 2	`x;e`	Shift(1)
4	0 1 2 1	`;e`	Shift(2)
5	0 1 2 1 2	`e`	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		Reduce 2
8	0 1 2		

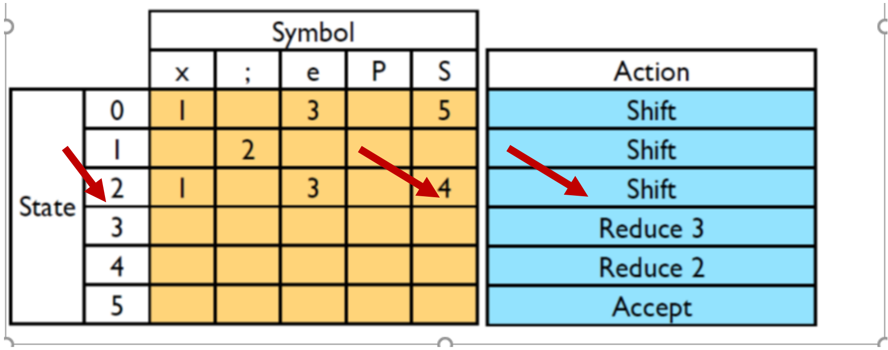
- *Now stack top has symbol 2 and LHS of rule II has S (imagine you saw S at input). Consult goto and action table.*

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



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

Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)
4	0 1 2 1	;e	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		Reduce 2 (shift(4))
8	0 1 2 4		

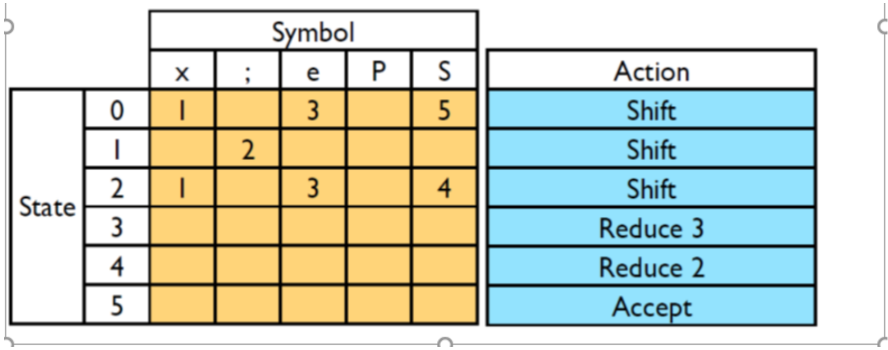
- *Now stack top has symbol 2 and LHS of rule II has S (imagine you saw S at input). Consult goto and action table. Shift(4)*

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



		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

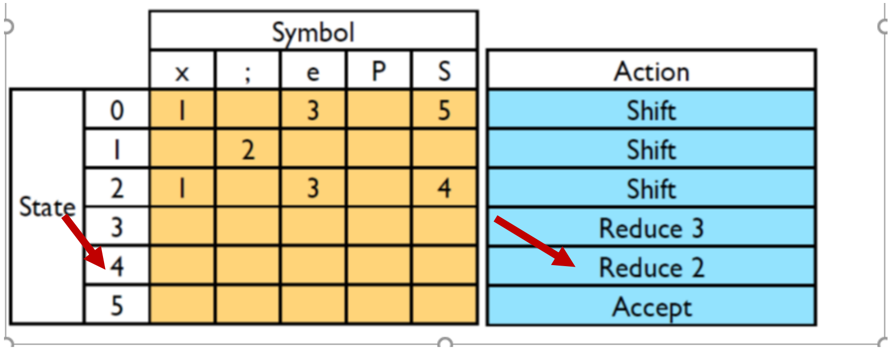
Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)
4	0 1 2 1	;e	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		Reduce 2 (shift(4))
8	0 1 2 4		?

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



		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

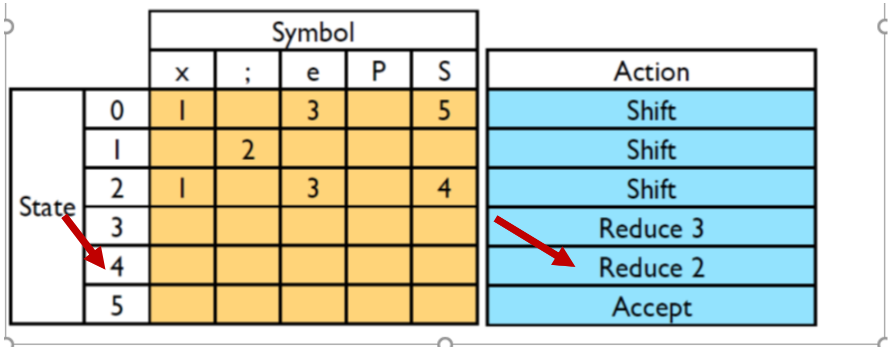
Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)
4	0 1 2 1	;e	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		Reduce 2 (shift(4))
8	0 1 2 4		Reduce 2

Example

I) P->S
II) S->x;S
III) S->e

<u>Input string</u>

x;x;e



		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

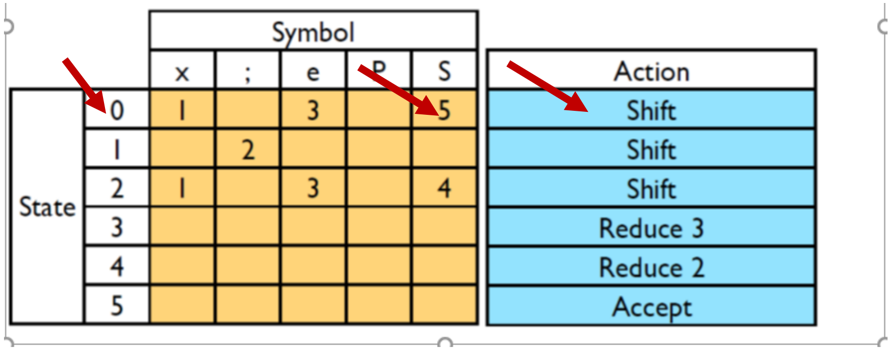
Step	Parse Stack	Rem. Input	Parser Action
1	0	`x;x;e`	Shift(1)
2	0 1	`;x;e`	Shift(2)
3	0 1 2	`x;e`	Shift(1)
4	0 1 2 1	`;e`	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		Reduce 2 (shift(4))
8	0 1 2 4		Reduce 2
9	0		

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



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

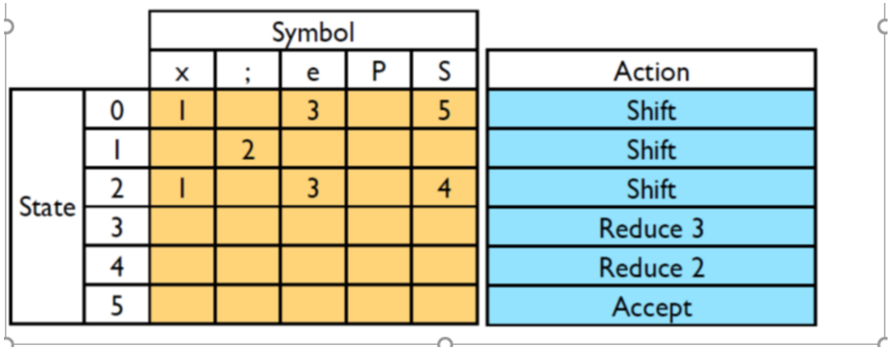
Step	Parse Stack	Rem. Input	Parser Action
1	0	`x;x;e`	Shift(1)
2	0 1	`;x;e`	Shift(2)
3	0 1 2	`x;e`	Shift(1)
4	0 1 2 1	`;e`	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		Reduce 2 (shift(4))
8	0 1 2 4		Reduce 2 (shift(5))
9	0 5		

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



		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

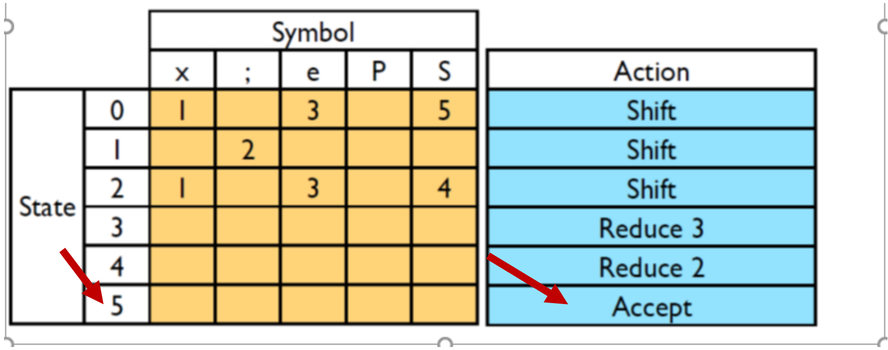
Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)
4	0 1 2 1	;e	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		Reduce 2 (shift(4))
8	0 1 2 4		Reduce 2 (shift(5))
9	0 5		?

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



		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

Step	Parse Stack	Rem. Input	Parser Action
1	0	`x;x;e`	Shift(1)
2	0 1	`;x;e`	Shift(2)
3	0 1 2	`x;e`	Shift(1)
4	0 1 2 1	`;e`	Shift(2)
5	0 1 2 1 2	`e`	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		Reduce 2 (shift(4))
8	0 1 2 4		Reduce 2 (shift(5))
9	0 5		Accept

means replace whatever is there in the stack with the start symbol

Example

I) P->S
II) S->x;S
III) S->e

Input string

|x;x;e

Initial scan pointer

Step	**Parser Action**
1	Shift(1)
2	Shift(2)
3	Shift(1)
4	Shift(2)
5	Shift(3)
6	Reduce 3 (shift(4))
7	Reduce 2 (shift(4))
8	Reduce 2 (shift(5))
9	Accept

Example

`I) P->S`
`II) S->x;S`
`III) S->e`

`Input string`

`|x;x;e`

Initial scan pointer

Step	Parser Action
1	Shift(1)
2	Shift(2)
3	Shift(1)
4	Shift(2)
5	Shift(3)
6	Reduce 3 (shift(4))
7	Reduce 2 (shift(4))
8	Reduce 2 (shift(5))
9	Accept

`x |; x ; e`

Example

I) P->S
II) S->x;S
III) S->e

Input string

|x;x;e

Initial scan pointer

Step	Parser Action
1	Shift(1)
2	Shift(2)
3	Shift(1)
4	Shift(2)
5	Shift(3)
6	Reduce 3 (shift(4))
7	Reduce 2 (shift(4))
8	Reduce 2 (shift(5))
9	Accept

x ;|x ; e

Example

I) P->S
II) S->x;S
III) S->e

Input string

|x;x;e

Initial scan pointer

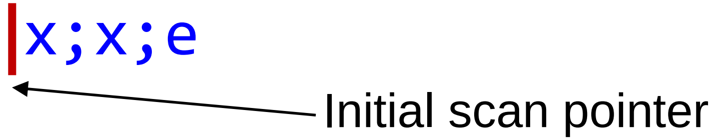
Step	Parser Action
1	Shift(1)
2	Shift(2)
3	Shift(1)
4	Shift(2)
5	Shift(3)
6	Reduce 3 (shift(4))
7	Reduce 2 (shift(4))
8	Reduce 2 (shift(5))
9	Accept

x ; x|; e

Example

I) P->S
II) S->x;S
III) S->e

<u>Input string</u>



`|x;x;e`

Initial scan pointer

Step	Parser Action
1	Shift(1)
2	Shift(2)
3	Shift(1)
4	Shift(2)
5	Shift(3)
6	Reduce 3 (shift(4))
7	Reduce 2 (shift(4))
8	Reduce 2 (shift(5))
9	Accept

`x ; x ;| e`

Example

I) P->S
II) S->x;S
III) S->e

Input string

|x;x;e

Initial scan pointer

Step	Parser Action
1	Shift(1)
2	Shift(2)
3	Shift(1)
4	Shift(2)
5	Shift(3)
6	Reduce 3 (shift(4))
7	Reduce 2 (shift(4))
8	Reduce 2 (shift(5))
9	Accept

x ; x ; e|

Example

I) P->S
II) S->x;S
III) S->e

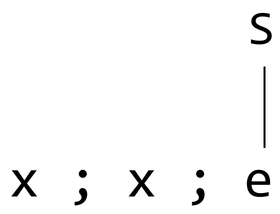
Input string

|x;x;e

Initial scan pointer

Step	Parser Action
1	Shift(1)
2	Shift(2)
3	Shift(1)
4	Shift(2)
5	Shift(3)
6	Reduce 3 (shift(4))
7	Reduce 2 (shift(4))
8	Reduce 2 (shift(5))
9	Accept

x ; x ; e|



Example

I) P->S
II) S->x;S
III) S->e

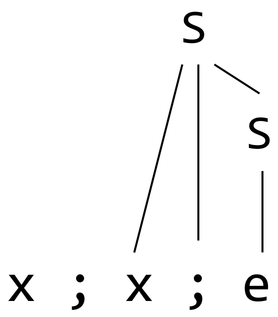
Input string

|x;x;e

Initial scan pointer

Step	Parser Action
1	Shift(1)
2	Shift(2)
3	Shift(1)
4	Shift(2)
5	Shift(3)
6	Reduce 3 (shift(4))
7	Reduce 2 (shift(4))
8	Reduce 2 (shift(5))
9	Accept

x ; x ; S|



Example

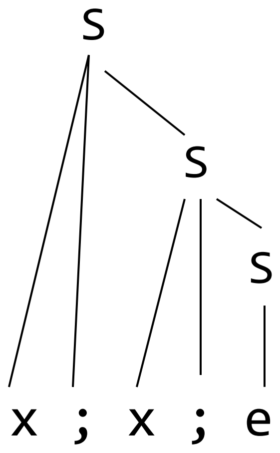
I) P->S
II) S->x;S
III) S->e

Input string

|x;x;e

Initial scan pointer

Step	Parser Action
1	Shift(1)
2	Shift(2)
3	Shift(1)
4	Shift(2)
5	Shift(3)
6	Reduce 3 (shift(4))
7	Reduce 2 (shift(4))
8	Reduce 2 (shift(5))
9	Accept



x ; S |

Example

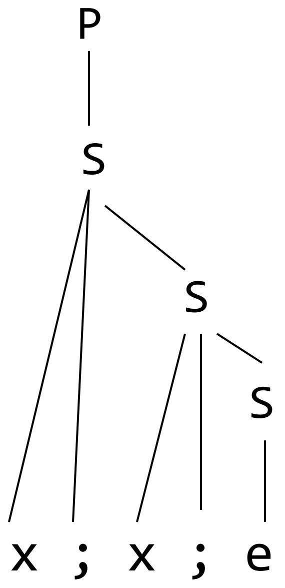
I) P->S
II) S->x;S
III) S->e

Input string

|x;x;e

Initial scan pointer

Step	Parser Action
1	Shift(1)
2	Shift(2)
3	Shift(1)
4	Shift(2)
5	Shift(3)
6	Reduce 3 (shift(4))
7	Reduce 2 (shift(4))
8	Reduce 2 (shift(5))
9	Accept



S |

Parsing using an LR(0) parser

- Basic idea: parser keeps track, simultaneously, of all possible productions that *could be matched* given what it's seen so far. When it sees a full production, match it.
- Maintain a *parse stack* that tells you what state you're in
  - Start in state 0
- In each state, look up in action table whether to:
  - *shift*: consume a token off the input; look for next state in goto table; push next state onto stack
  - *reduce*: match a production; pop off as many symbols from state stack as seen in production; look up where to go according to non-terminal we just matched; push next state onto stack
  - *accept*: terminate parse

Shift-Reduce Parsing

The LR parsing seen previously is an example of shift-reduce parsing

- When do we *shift* and when do we *reduce*?
  - *How do we construct 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 • 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 -> α•Bγ, where B is a non-terminal,

1 add configurations of the form:

  B -> • δ

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

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

Concept: closure

- E.g. closure {`S -> •E$`}

⇩ Non-terminal

`S -> •E$`

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$}

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

- E.g. closure {S ->• E$}

⇩

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

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
T -> •(E)
```

Grammar

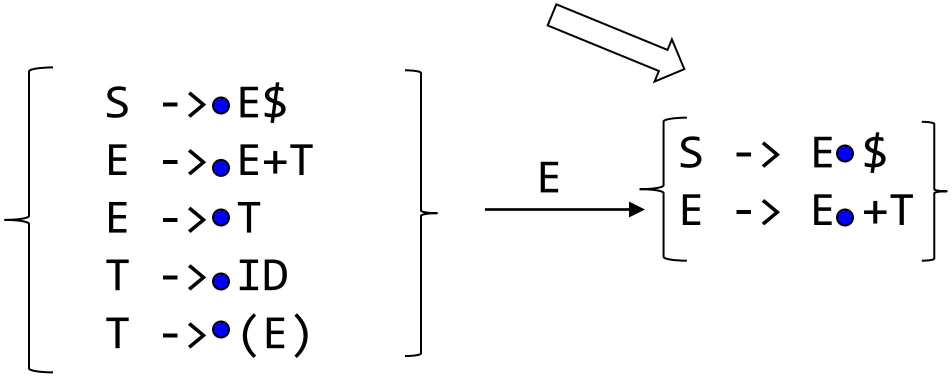
```
S -> E$
E -> E+T | T
T -> ID | (E)
```

Concept: successor

Grammar

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

- E.g. successor ({S -> ●E$}, E)


S -> ●E$
E -> ●E+T
E -> ●T
T -> ●ID
T -> ●(E)

E

S -> E●$
E -> E●+T

- Consider all symbols that are to the immediate right of Dot 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

Grammar

P->S

S->x;S

S->e

Add item

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

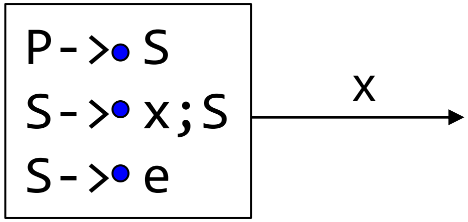
Grammar

P->S

S->x;S

S->e

Compute successor (of state 0) under symbol x



P->•S

S->•x;S

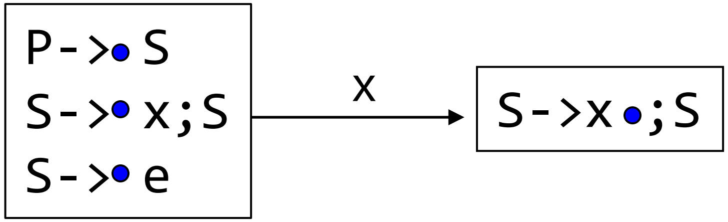
S->•e

state 0

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

Example: CFSM

Compute successor (of state 0) under symbol x



P->•S
S->•x;S
S->•e

state 0

x

S->x•;S

Grammar

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

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

Example: CFSM

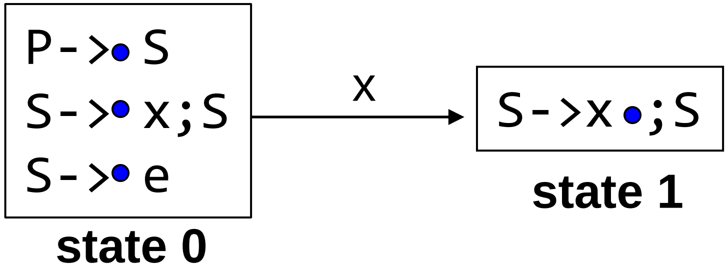
Grammar

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

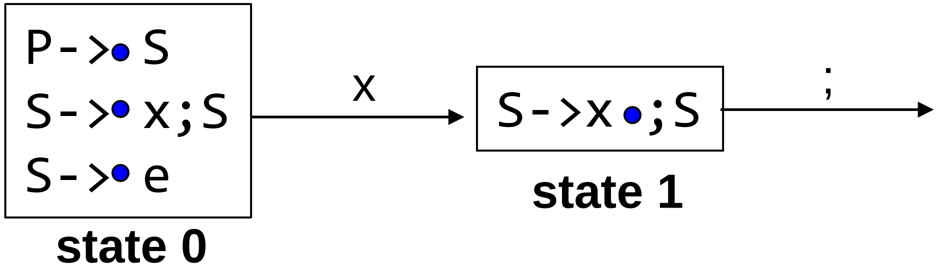
Grammar

P->S

S->x;S

S->e

Compute successor (of state 1) under symbol ;



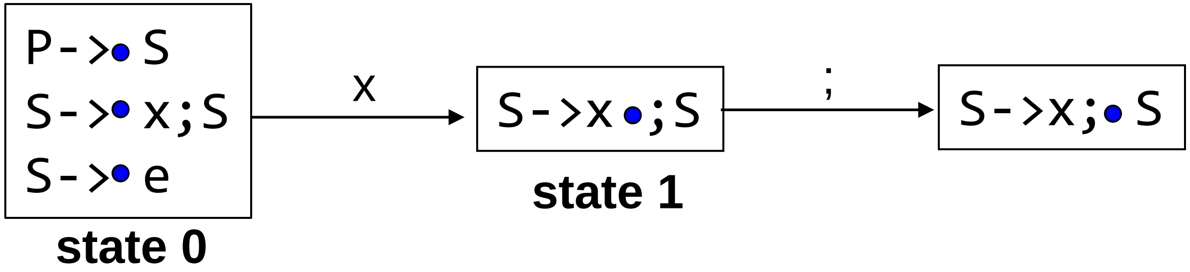
Consider items (in state 1), where ; is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

Grammar

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

Compute successor (of state 1) under symbol ;



```
P->•S
S->•x;S
S->•e
```

state 0

x

```
S->x•;S
```

state 1

;

```
S->x;•S
```

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

Example: CFSM

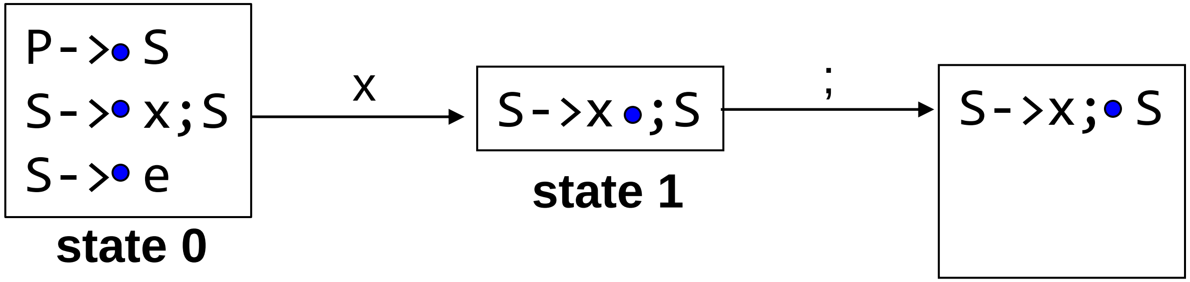
Grammar

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.

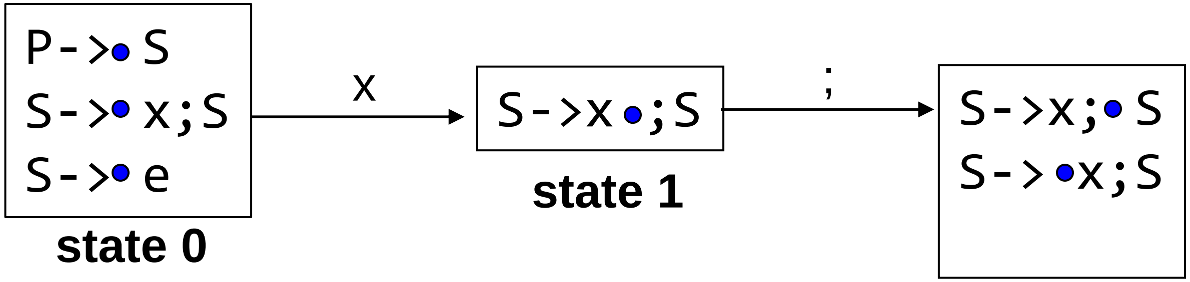
There is a non-terminal immediately after Dot in the successor of state 1. So, add configurations.

Example: CFSM

Grammar

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

Compute successor (of state 1) under symbol ;



`P->•S`
`S->•x;S`
`S->•e`

state 0

x

`S->x•;S`

state 1

;

`S->x;•S`
`S->•x;S`

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

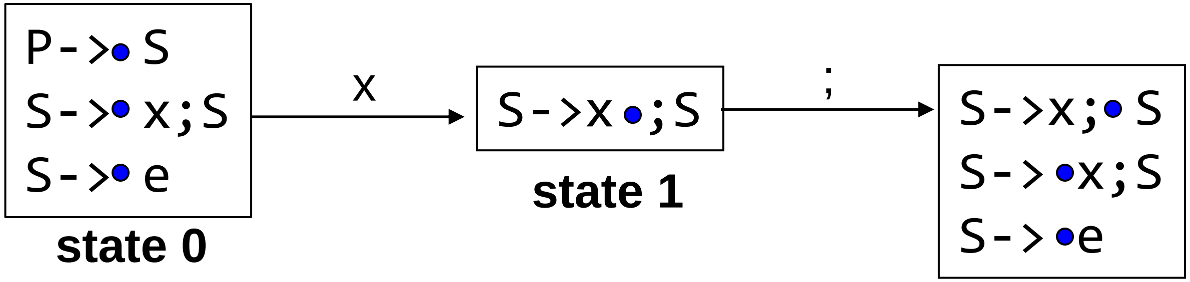
Grammar

P->S

S->x;S

S->e

Compute successor (of state 1) under symbol ;



P->• S
S->• x;S
S->• e
state 0

x

S->x •;S
state 1

;

S->x;• S
S-> •x;S
S-> •e

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

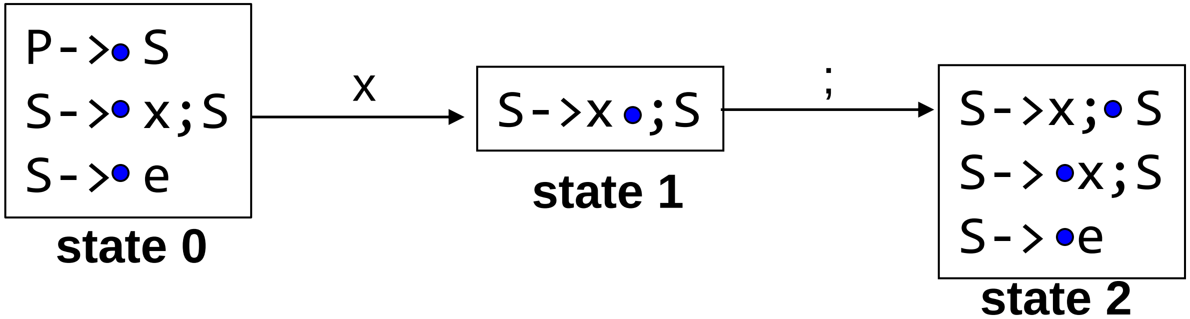
There is a non-terminal immediately after Dot in the successor of state 1. So, add configurations.

Example: CFSM

Grammar

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

Compute successor (of state 1) under symbol ;

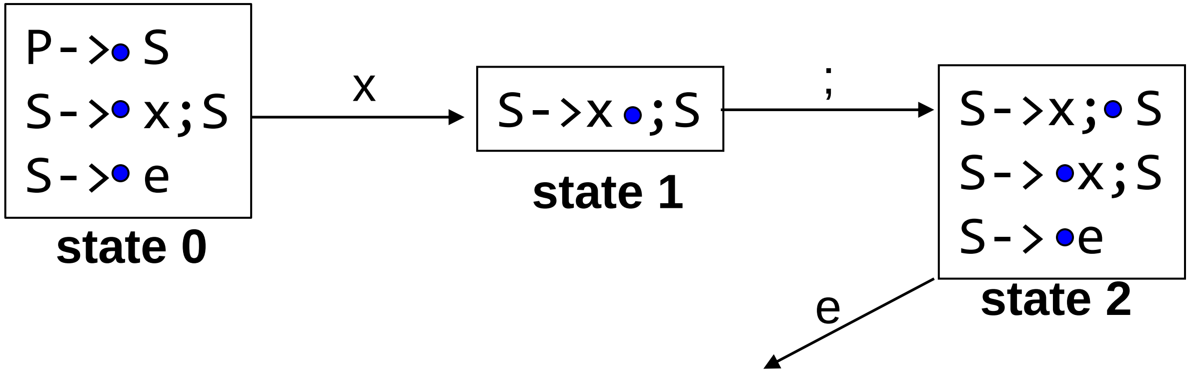


Consider items (in state 1), where ; is to the immediate right of Dot. Advance Dot by one symbol.

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



P->• S
S->• x;S
S->• e

state 0

S->x •;S

state 1

S->x;• S
S-> •x;S
S-> •e

state 2

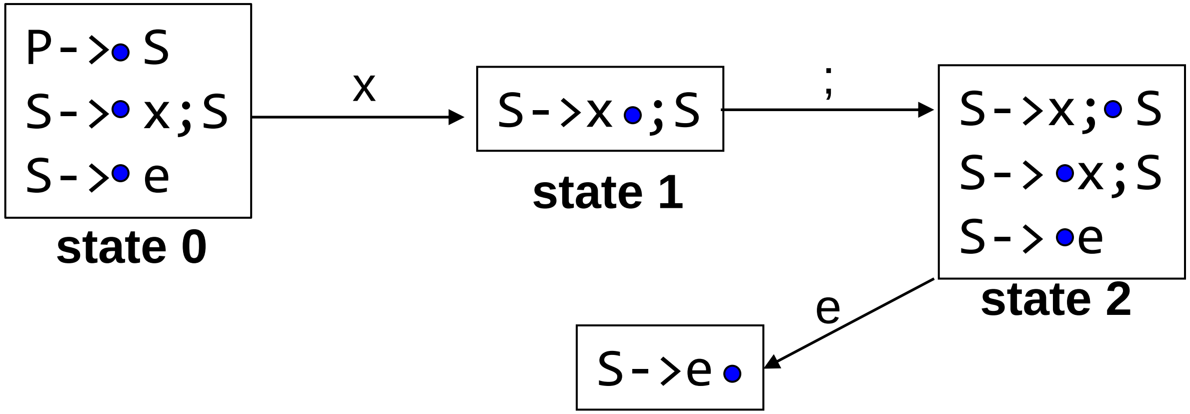
Grammar

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

Compute successor (of state 2) under symbol e



`P->•S`
`S->•x;S`
`S->•e`

state 0

x

`S->x•;S`

state 1

;

`S->x;•S`
`S->•x;S`
`S->•e`

state 2

e

`S->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

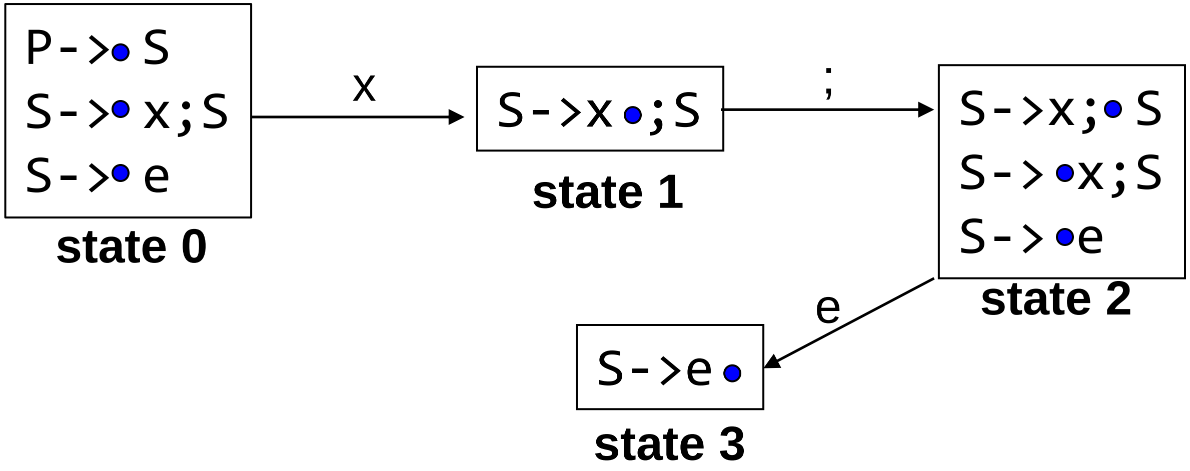
Grammar

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. No more items to be added. Becomes another state in CFSM.

Example: CFSM

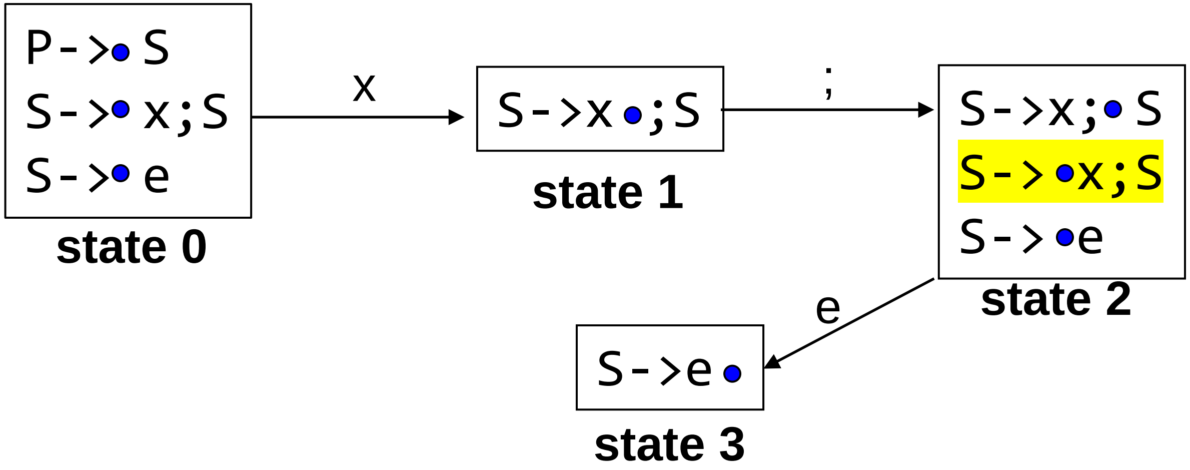
Grammar

P->S

S->x;S

S->e

Compute successor (of state 2) under symbol x



P->• S
S->• x;S
S->• e

state 0

x

S->x •;S

state 1

;

S->x;• S
S-> •x;S
S-> •e

state 2

e

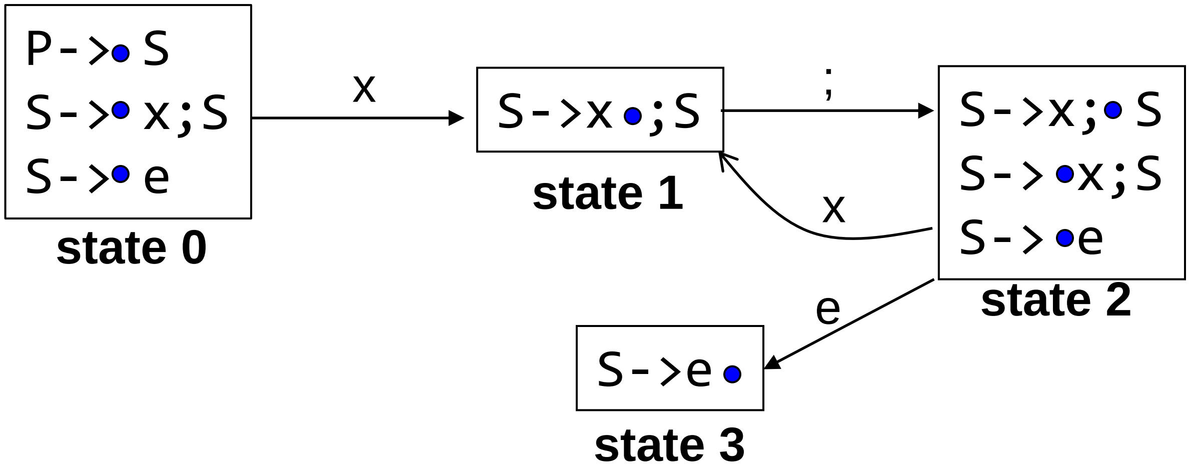
S->e •

state 3

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

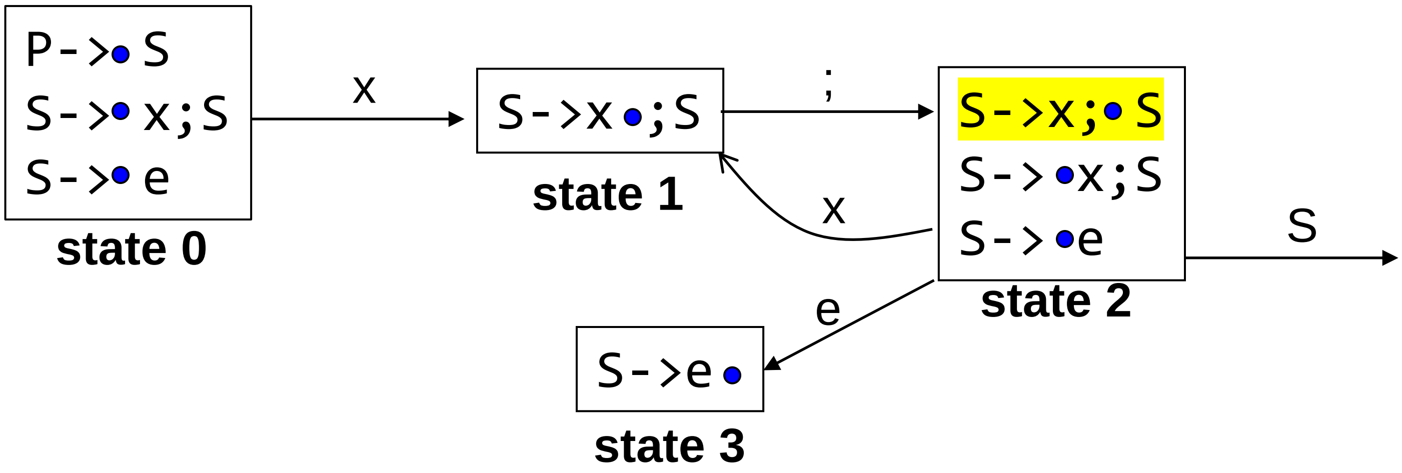
S->x;S

S->e

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

Example: CFSM

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.

Example: CFSM

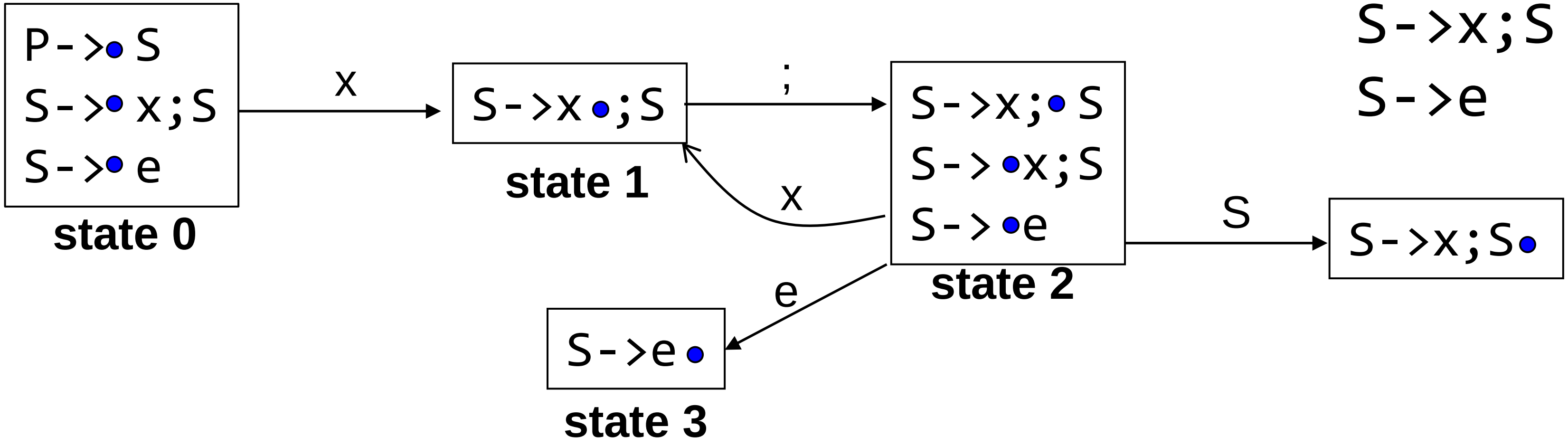
Grammar

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

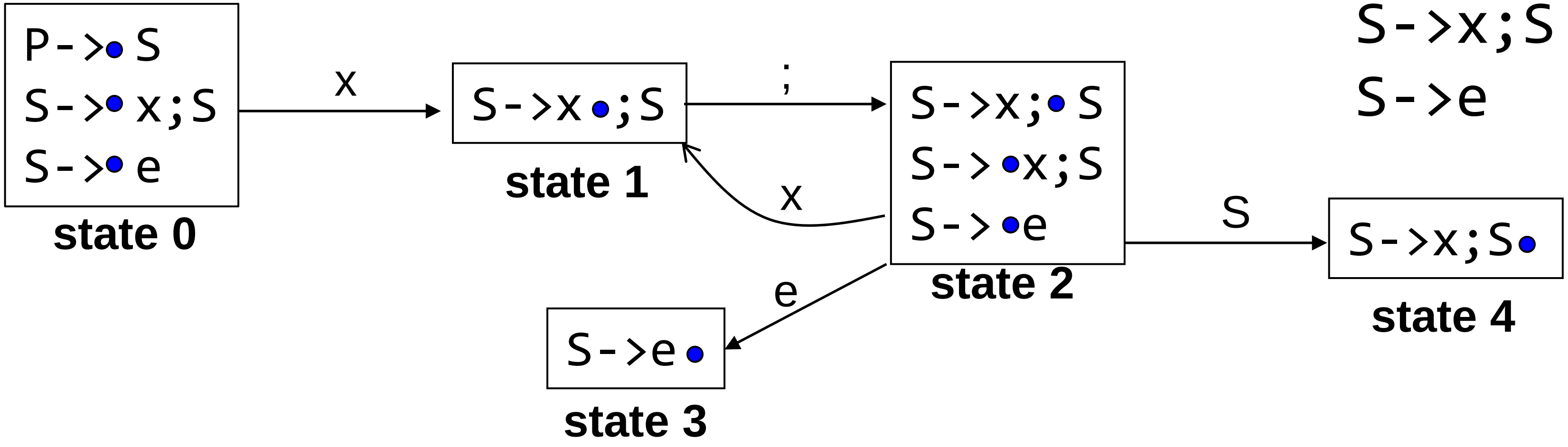
Grammar

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. No more items to be added. Becomes another state in CFSM.

Example: CFSM

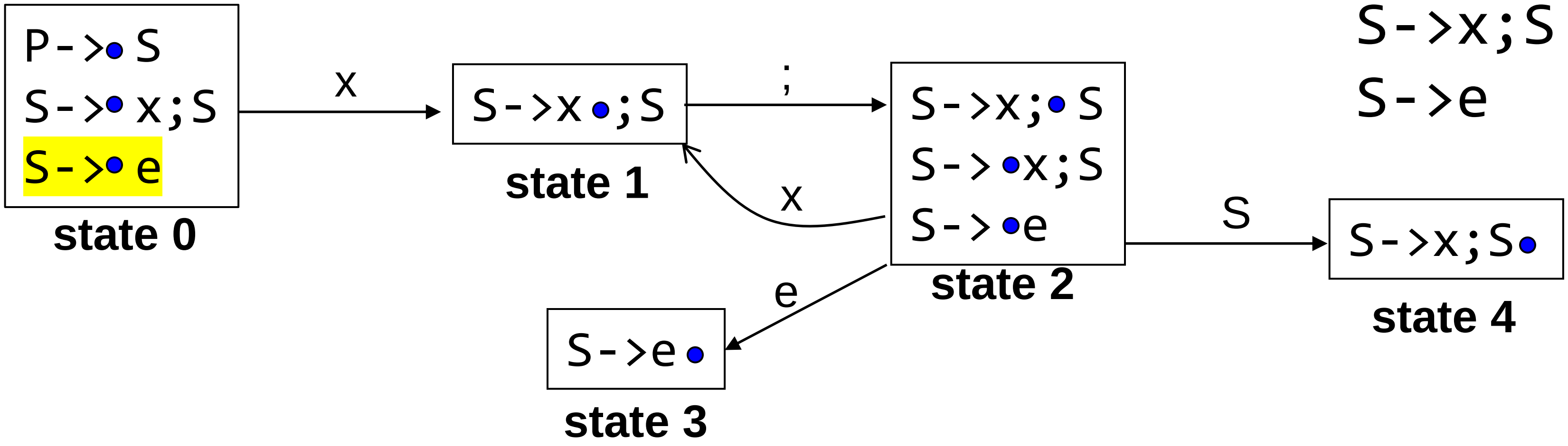
Grammar

P->S

S->x;S

S->e

Compute successor (of state 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

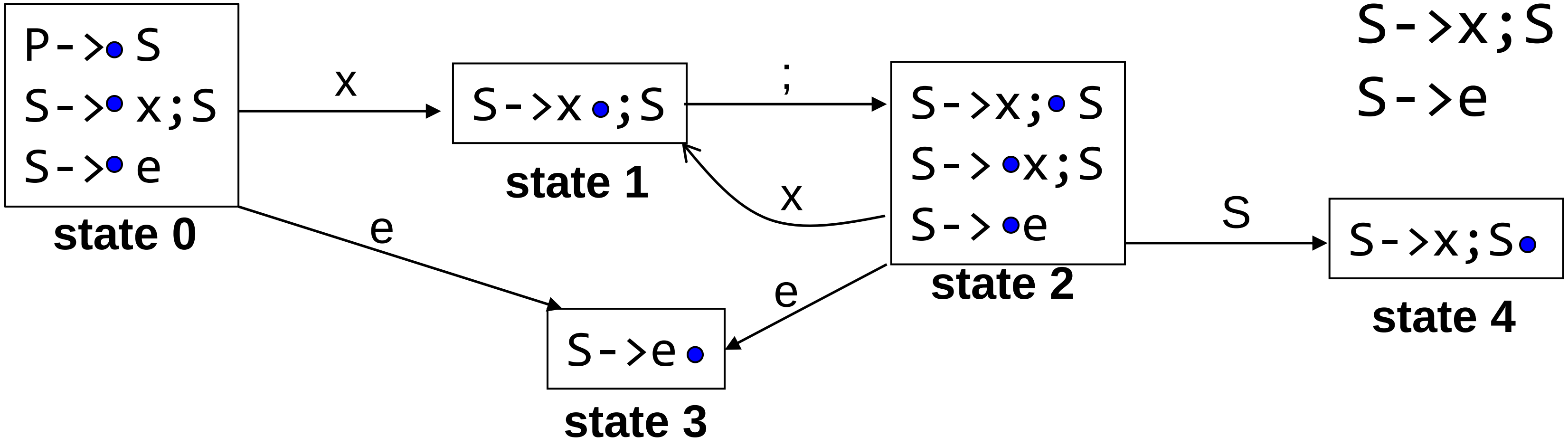
Grammar

P->S

S->x;S

S->e

Compute successor (of state 0) under symbol e



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

Example: CFSM

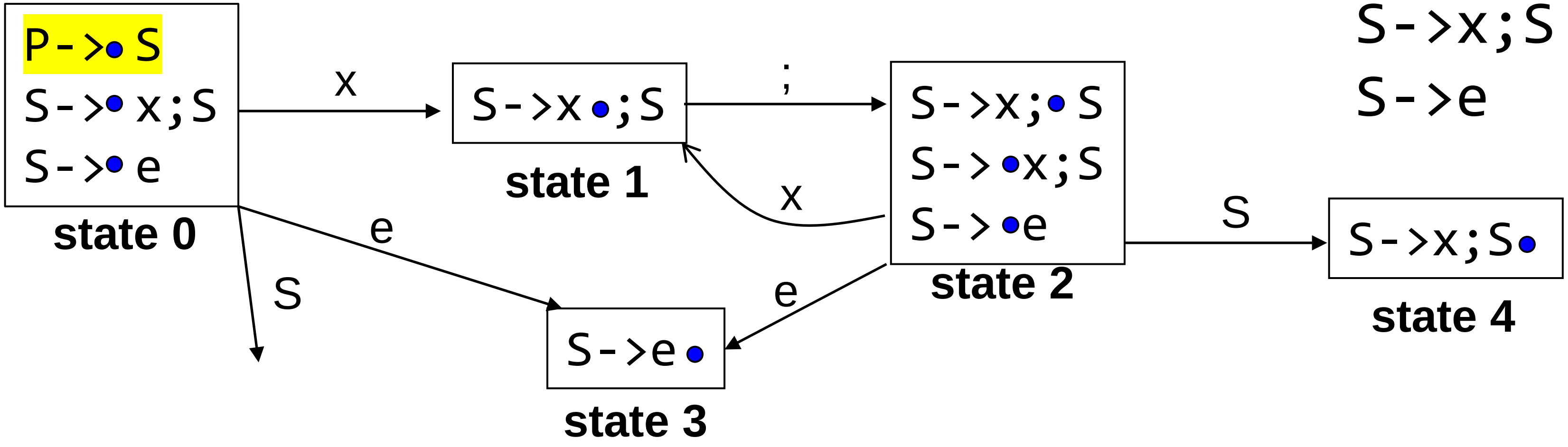
Grammar

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

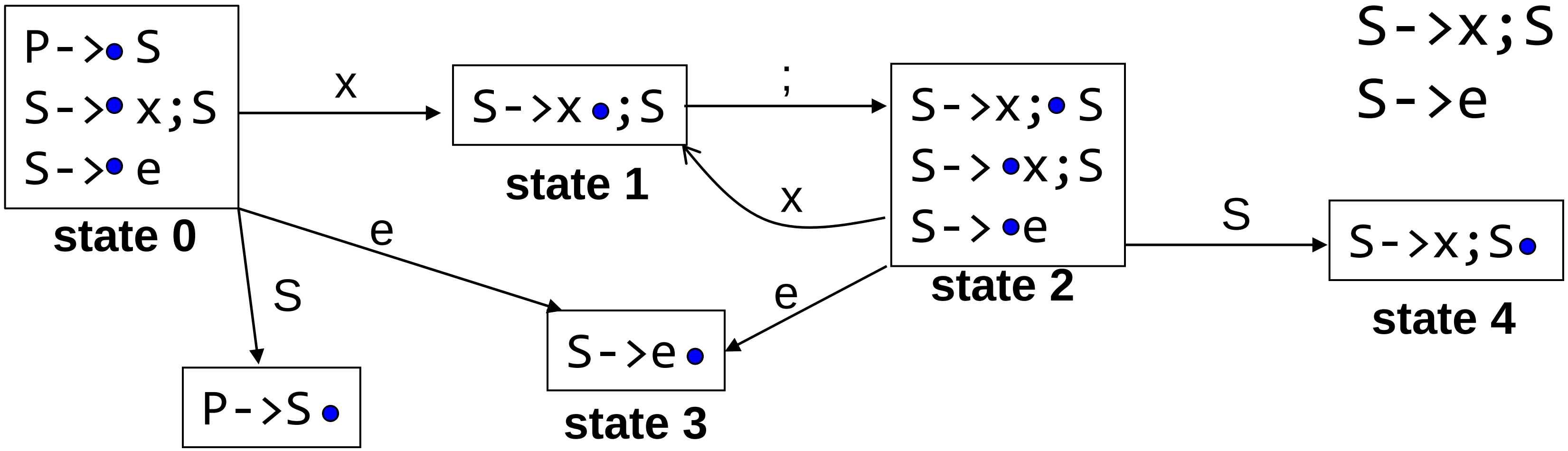
Grammar

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

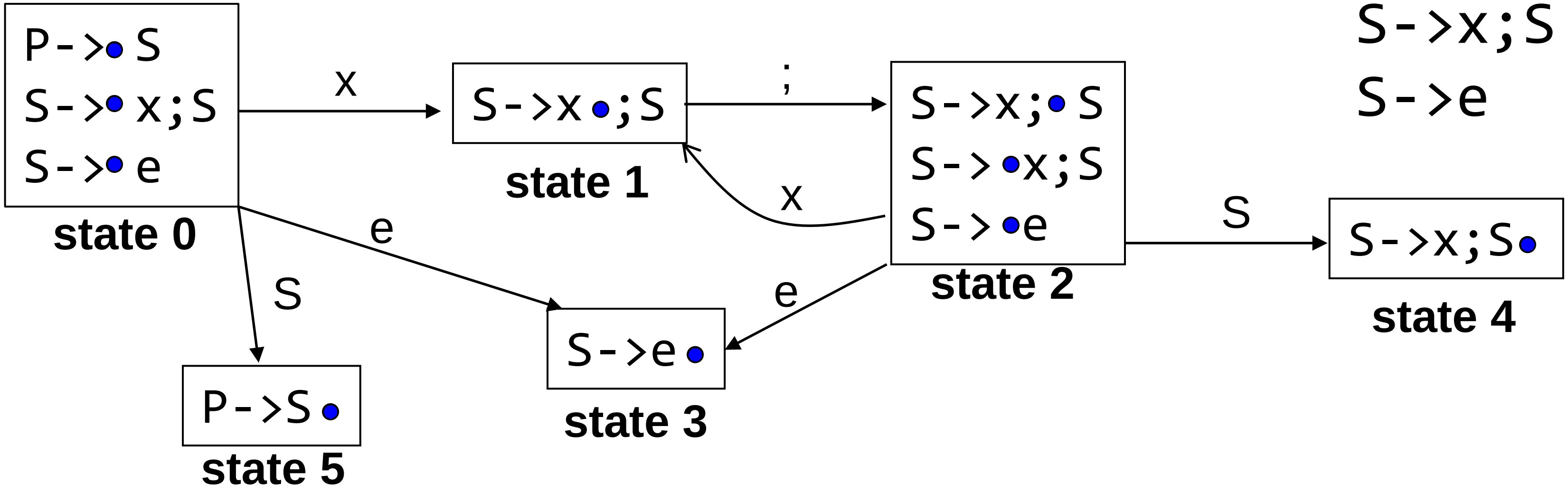
Grammar

P->S

S->x;S

S->e

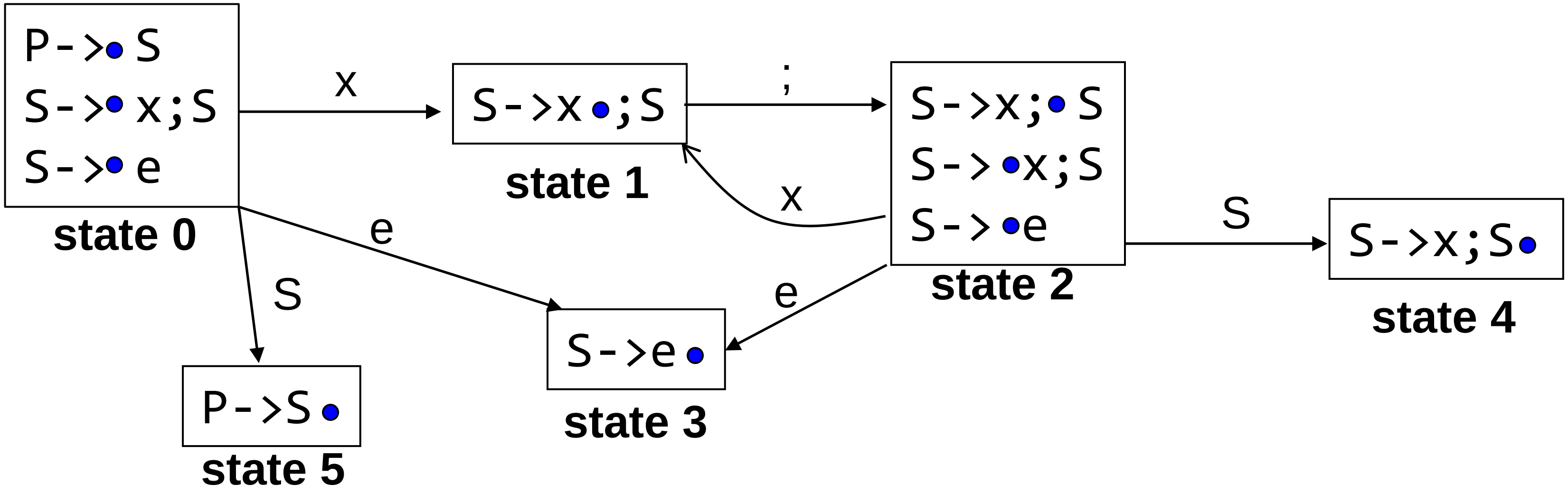
Compute successor (of state 0) under symbol S



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



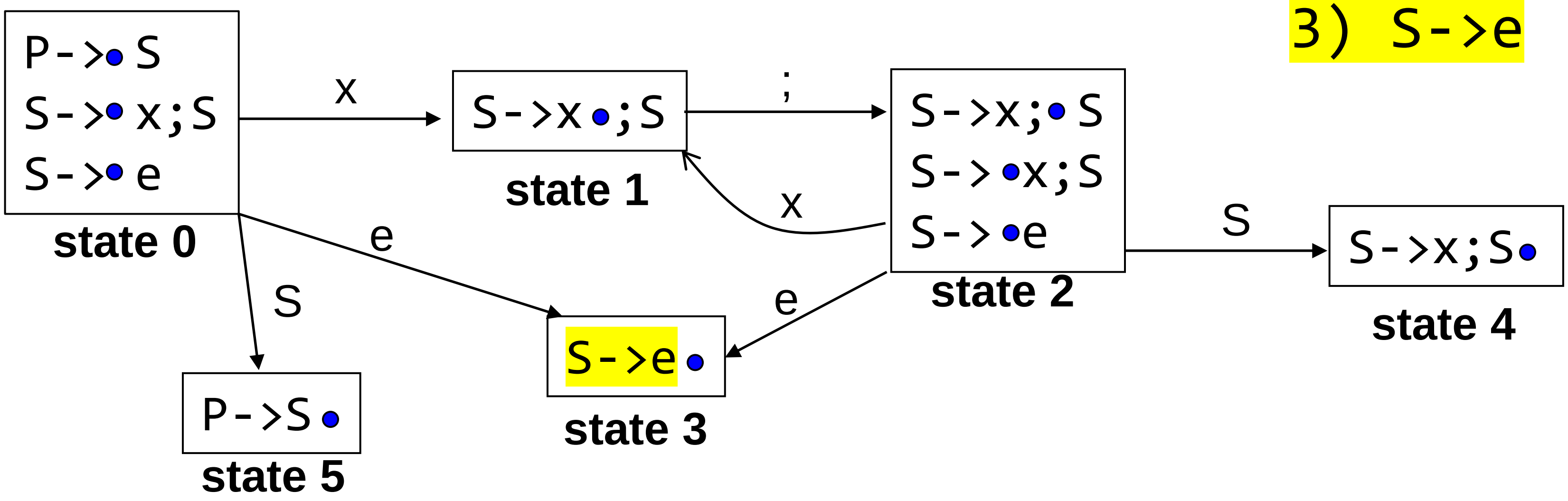
Example: CFSM

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

`Reduce 3`

Grammar

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



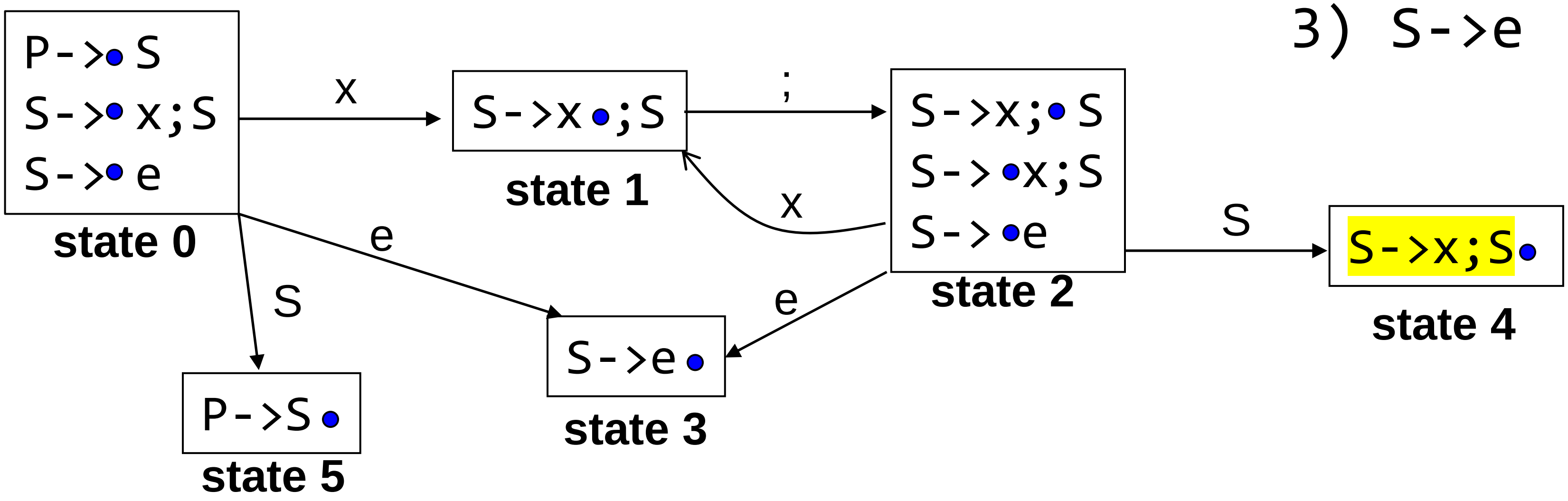
Example: CFSM

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

Reduce 2

Grammar

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



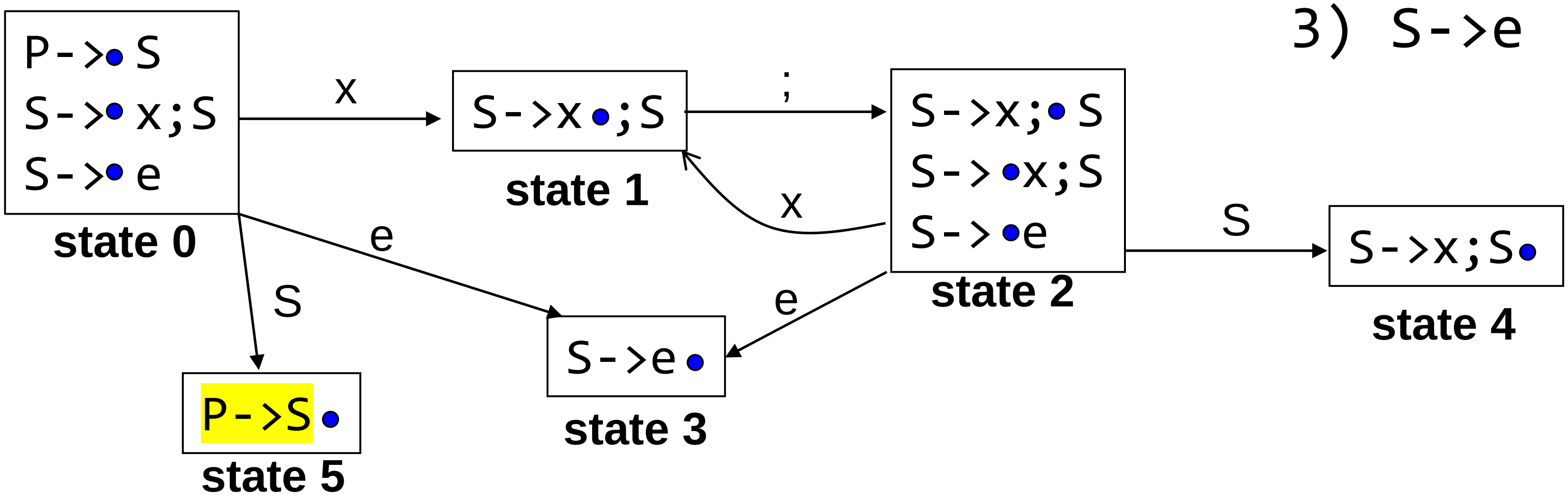
Example: CFSM

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

Grammar

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

Accept

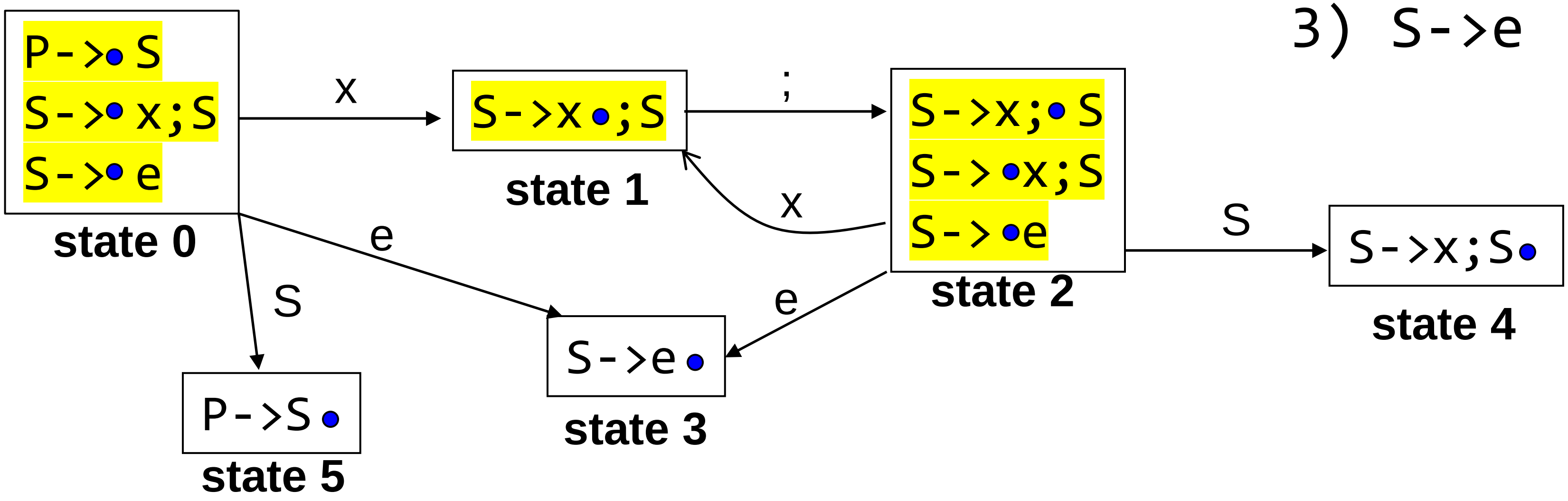


Example: CFSM

- 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

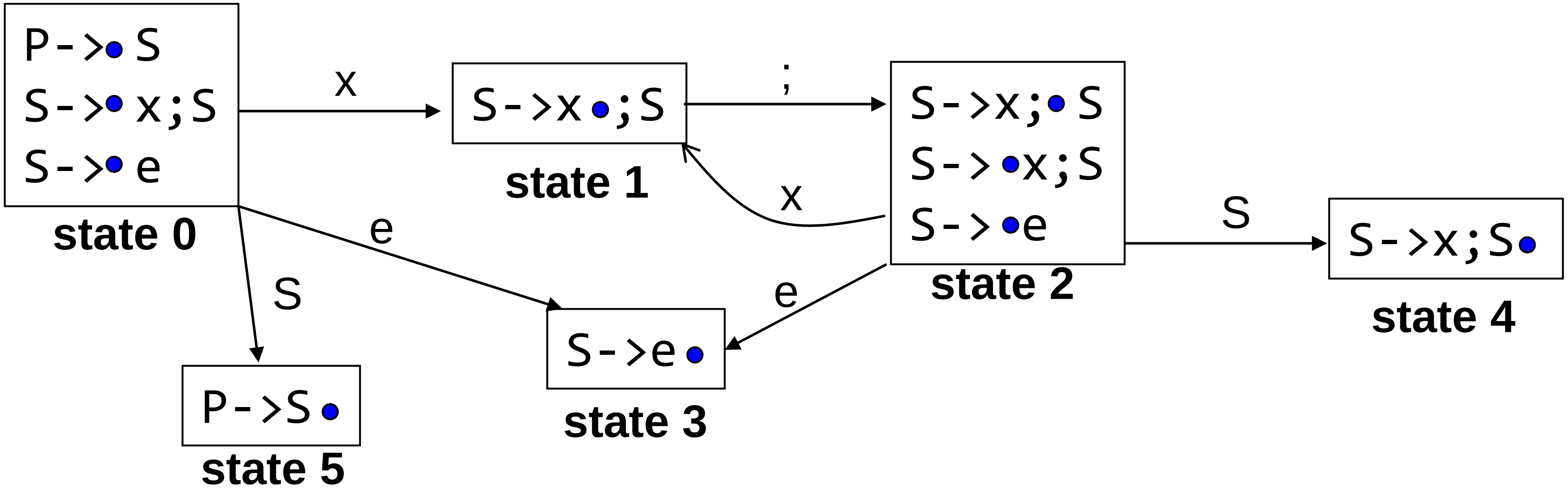
  Parser is unable to decide between shifting and reducing

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

  Reduce-Reduce conflict

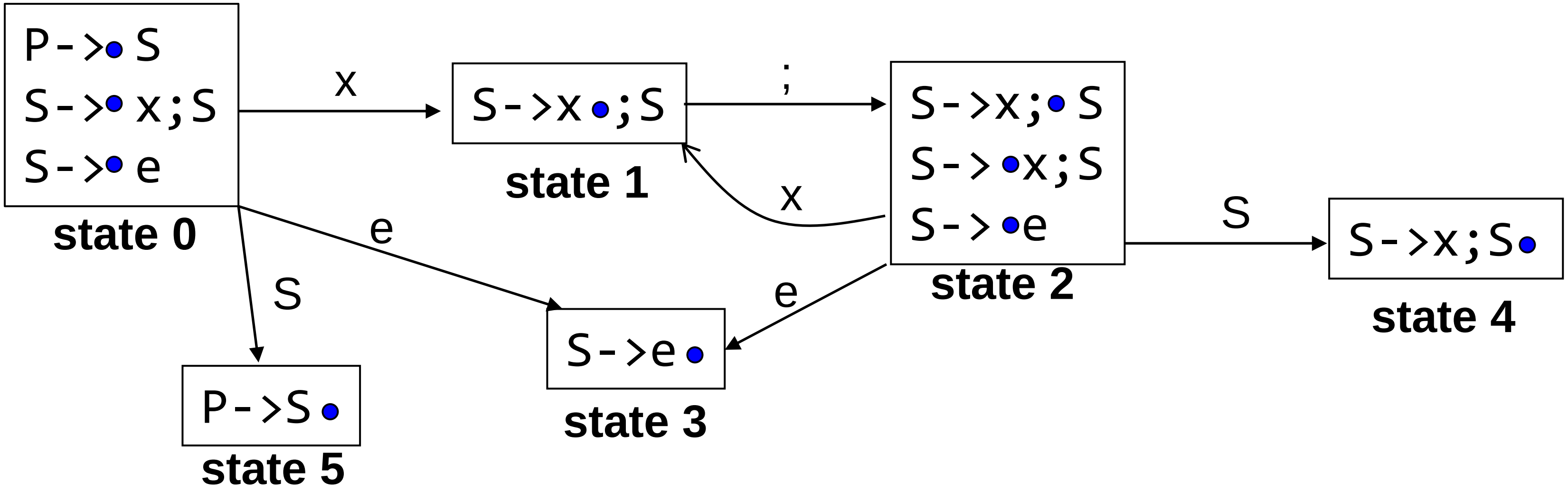
  Parser is unable to decide between which productions to choose for reducing

Example: goto table



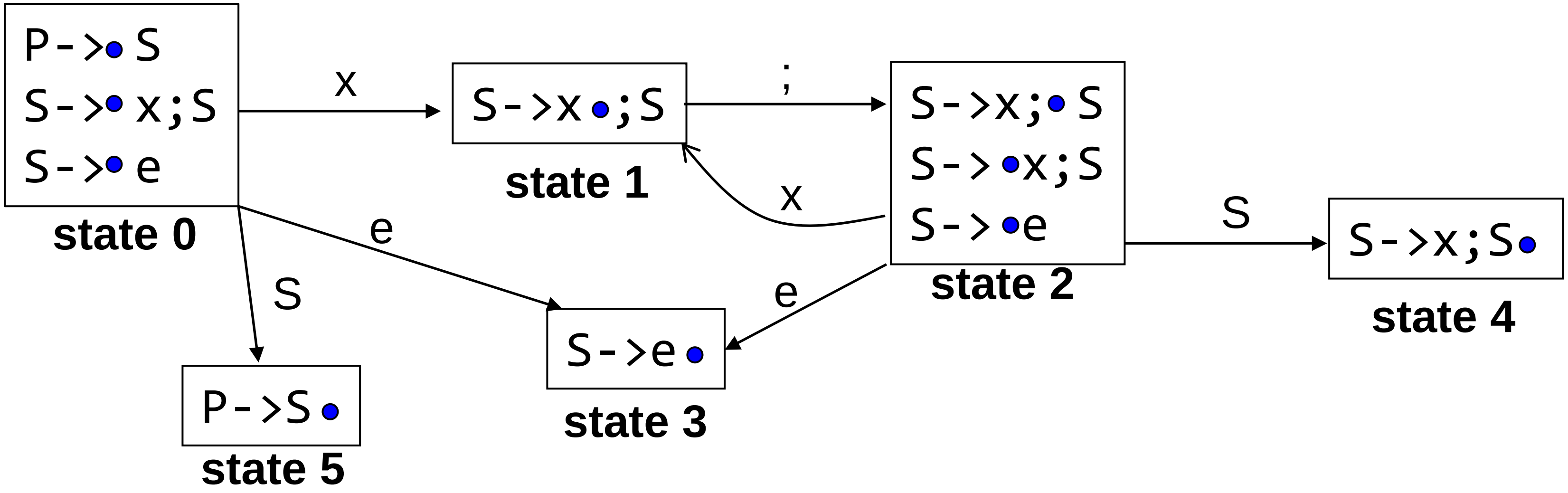
- construct transition table from CFSM.
  - Number of rows = number of states
  - Number of columns = number of symbols

Example: goto table



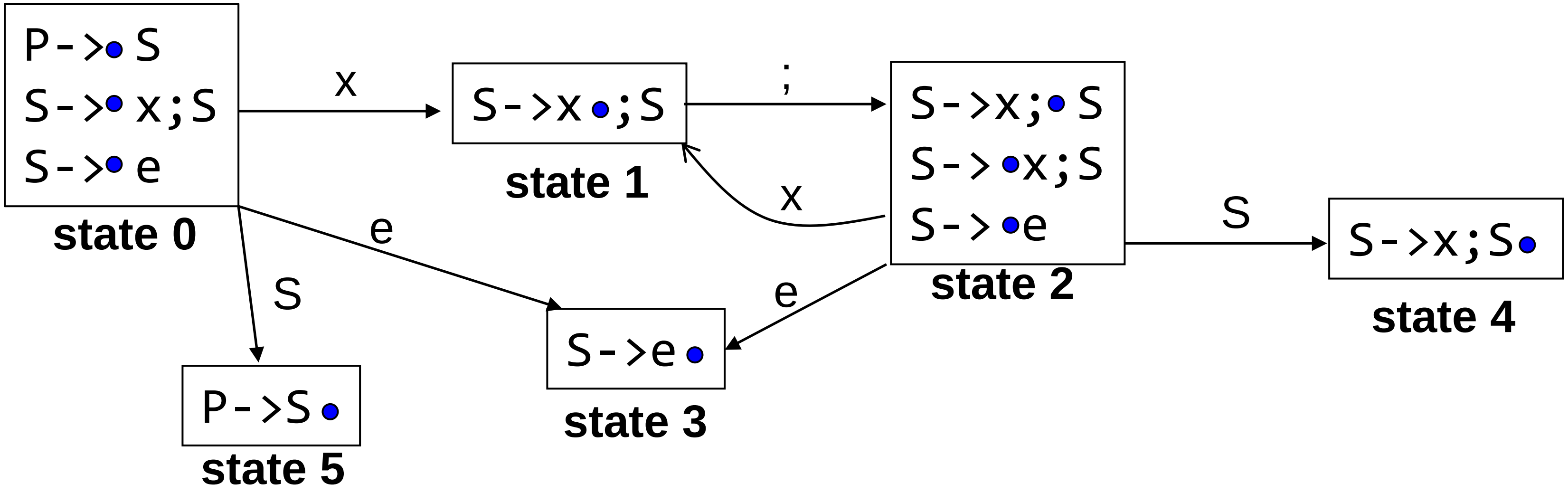
state	x	;	e	P	S
0	1		3		5
1		2			
2	1		3		4
3					
4					
5					

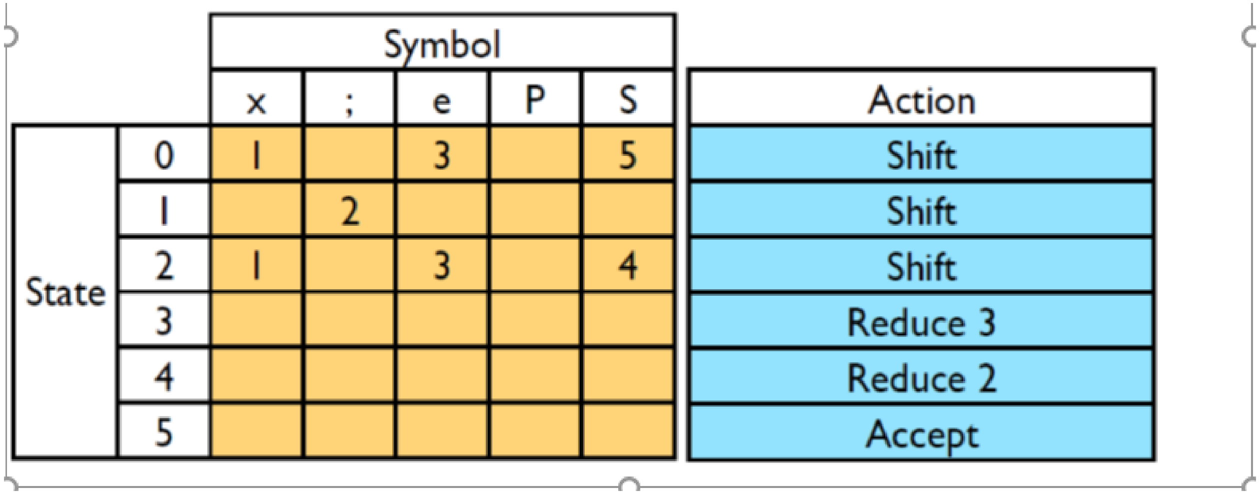
Example: action table



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

Example: action table





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

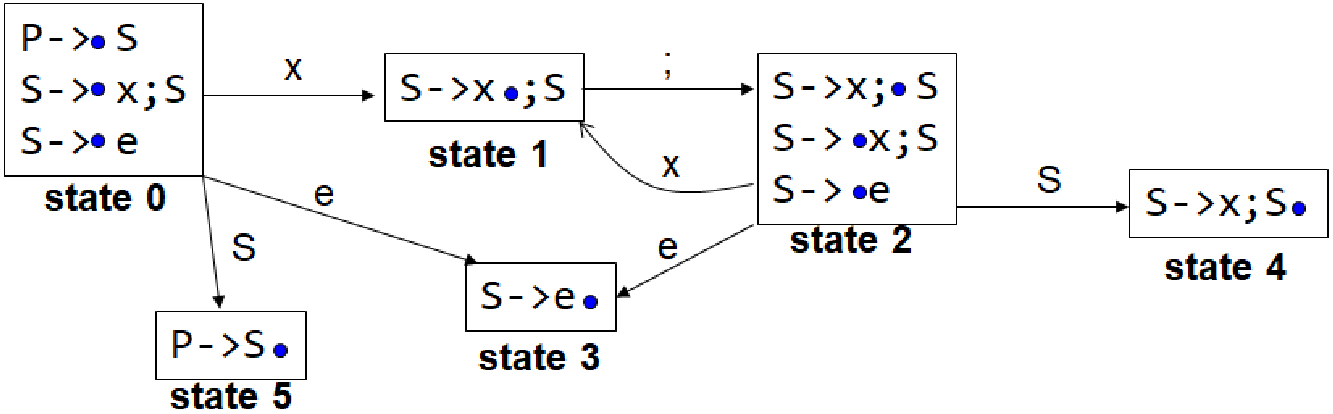
LR(0) Parsing

- 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

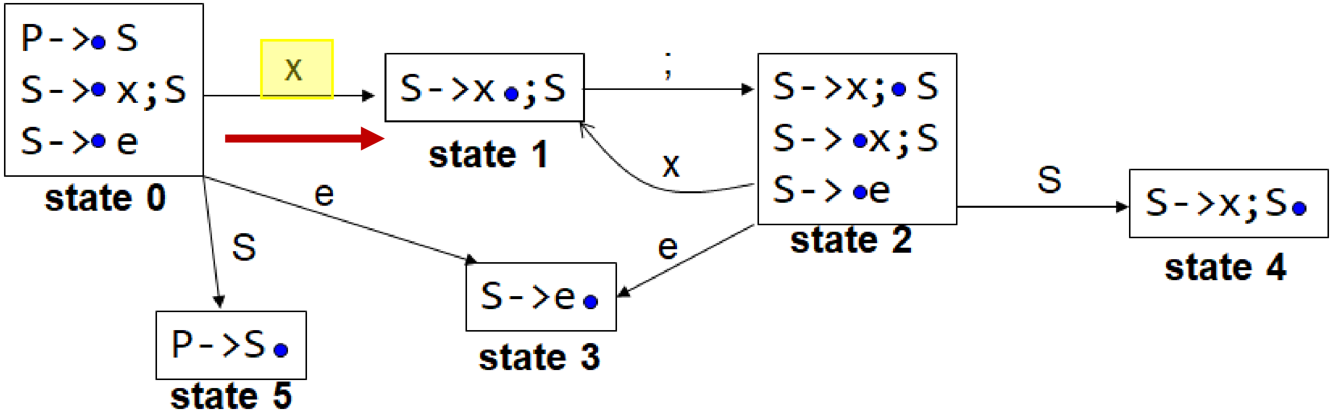
Parse Stack
0
0 1
0 1 2
0 1 2 1



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

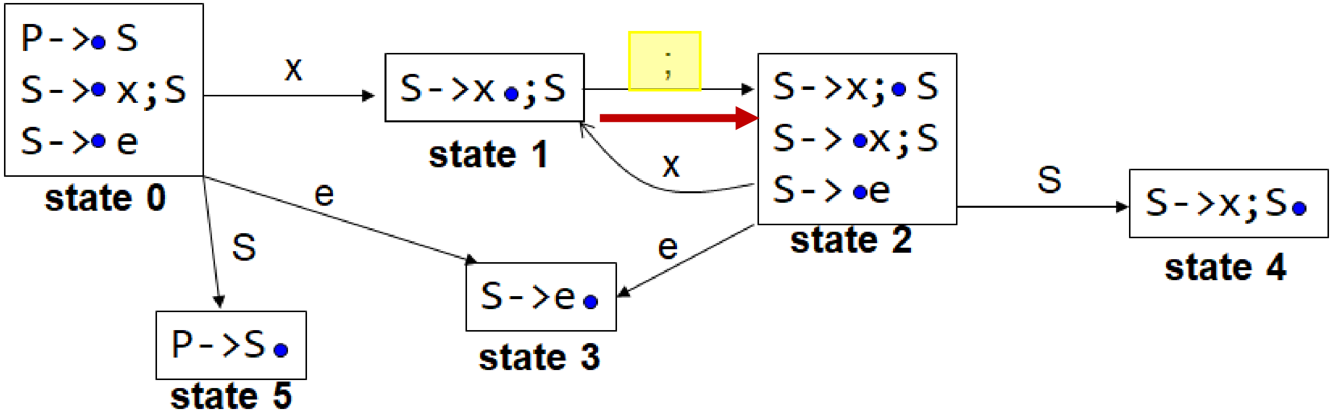


Go from state 0 to state 1 consuming x

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

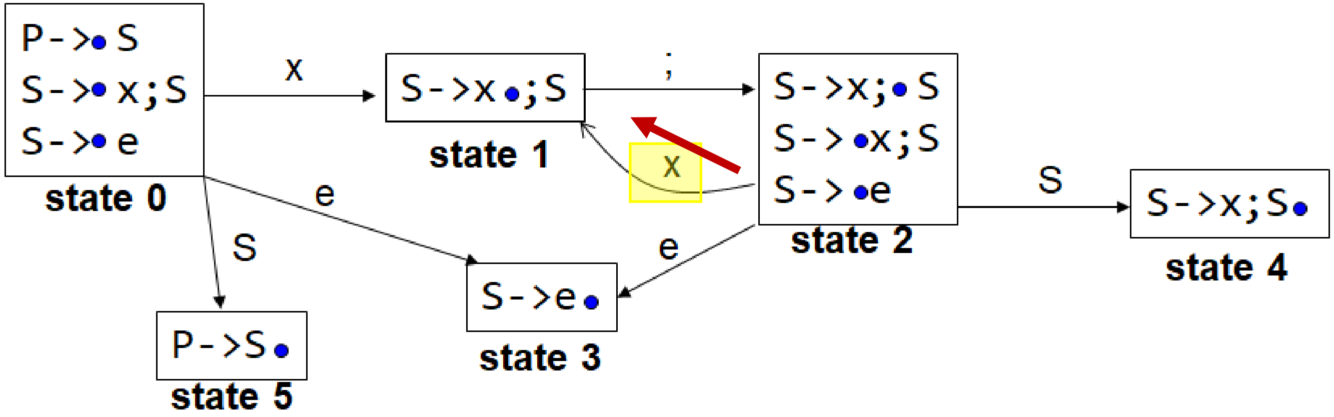


Go from state 1 to state 2 consuming ;

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->β if state s contains X->β●

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

LR(0) Parsing

- Assume: Parse stack contains α.

=> we are in some state `s`.

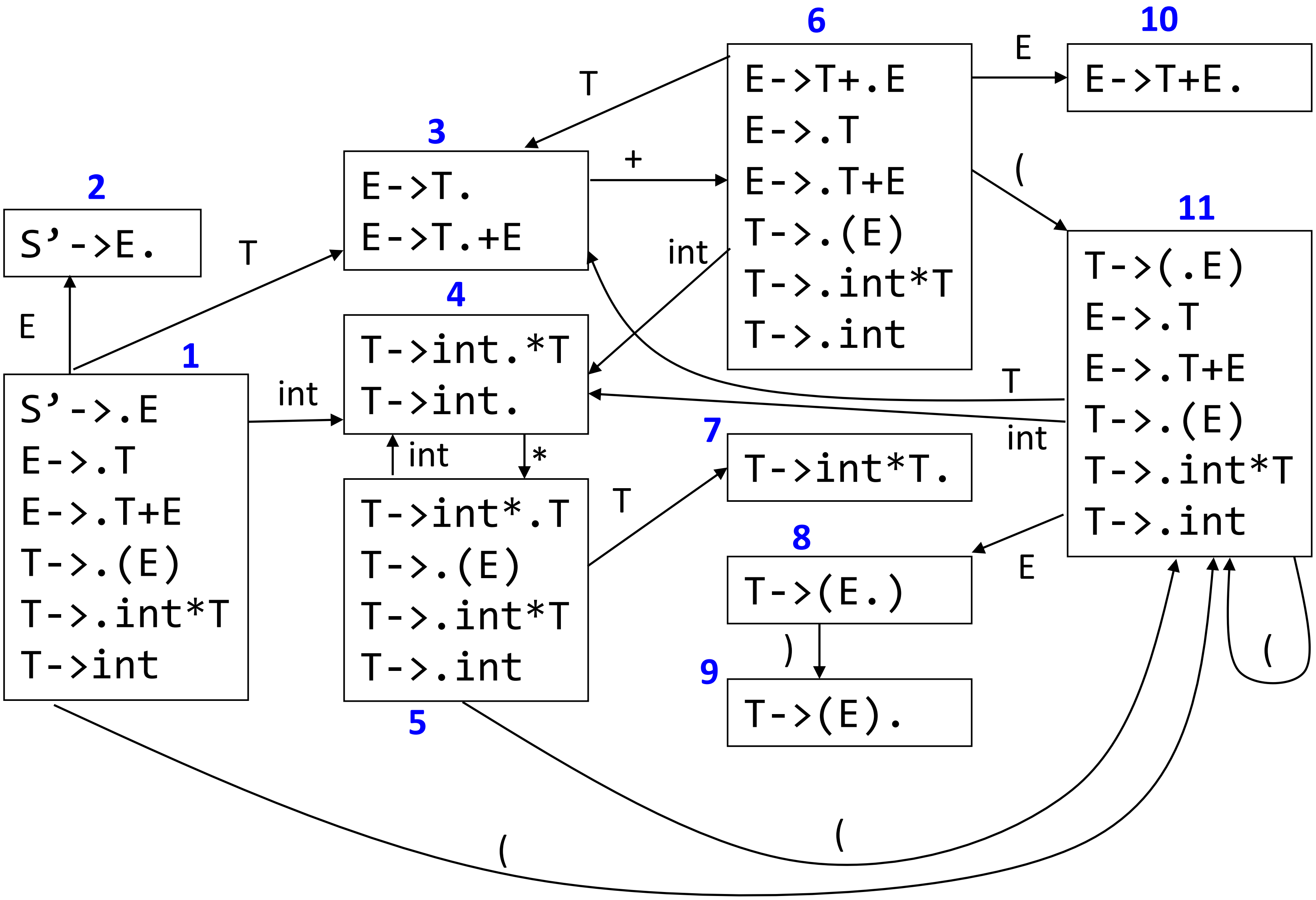
- Assume: Next input is t

We shift if s contains `X->β` ● `tω`

== s has a transition labelled `t`

LR(0) Parsing

- What if s contains `X->β ● tω` and `X->β●` ?



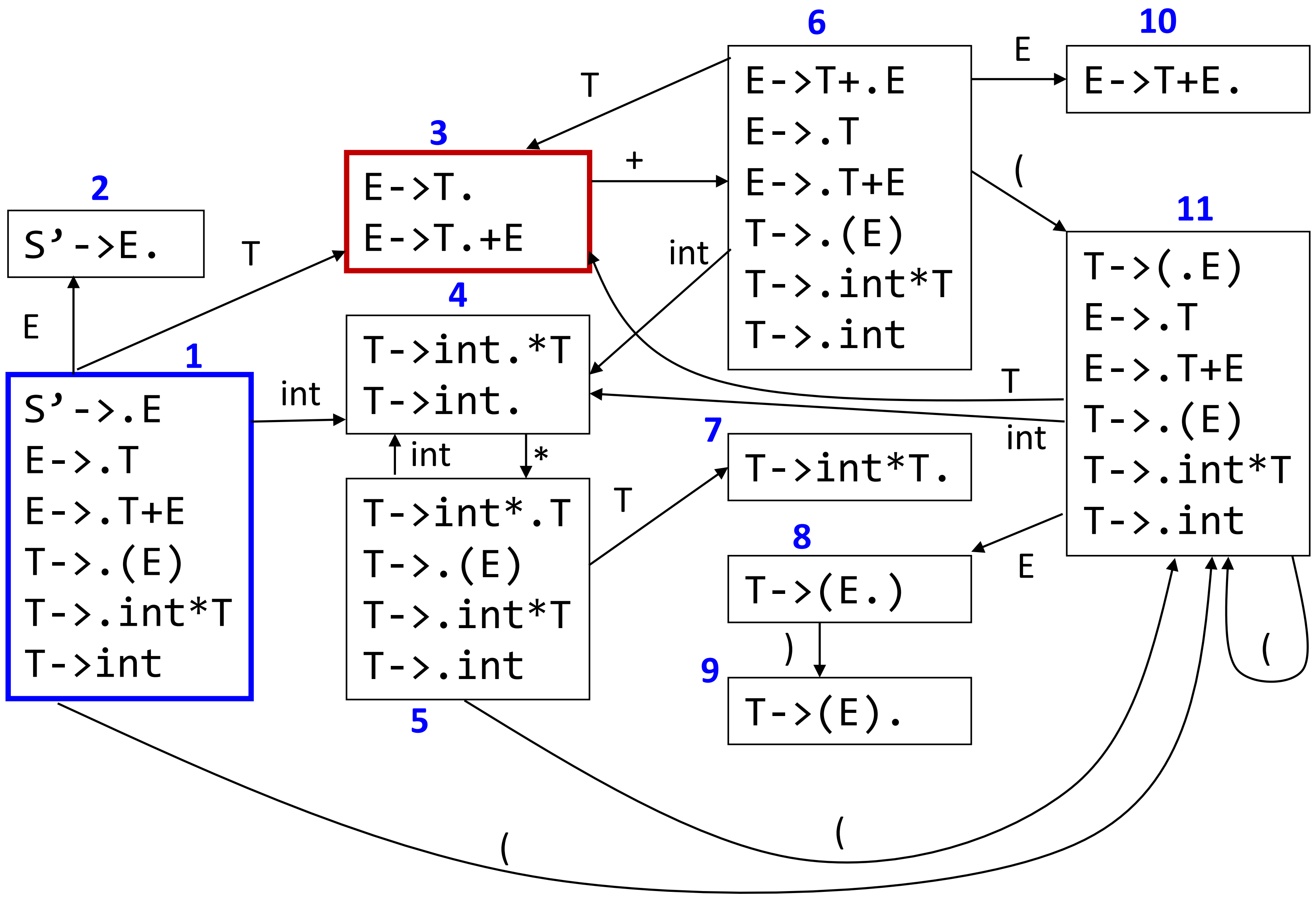
Conflicts or not?

SLR Parsing

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

Reduce `X->β` ● `only if`

`t` ϵ Follow(X)



1

S’->.E
E->.T
E->.T+E
T->.(E)
T->.int*T
T->int

2

S’->E.

3

E->T.
E->T.+E

4

T->int.*T
T->int.

5

T->int*.T
T->.(E)
T->.int*T
T->.int

6

E->T+.E
E->.T
E->.T+E
T->.(E)
T->.int*T
T->.int

7

T->int*T.

8

T->(E.)

9

T->(E).

10

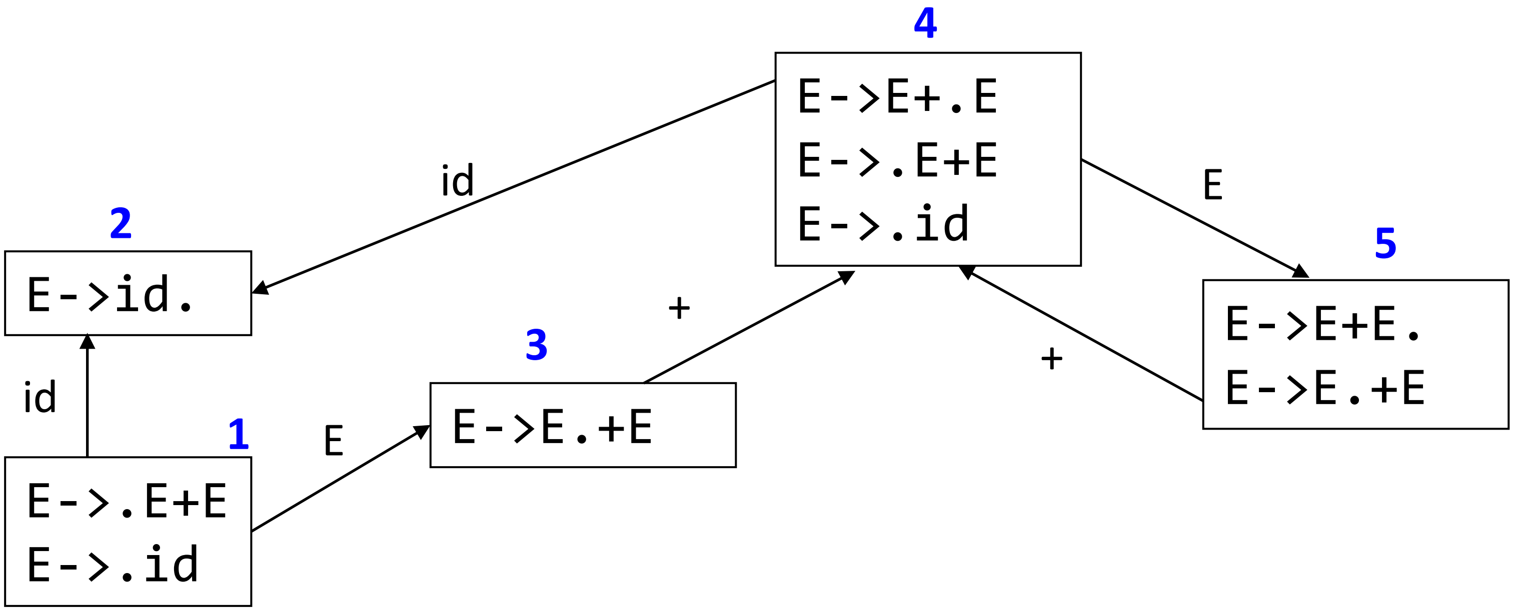
E->T+E.

11

T->(.E)
E->.T
E->.T+E
T->.(E)
T->.int*T
T->.int

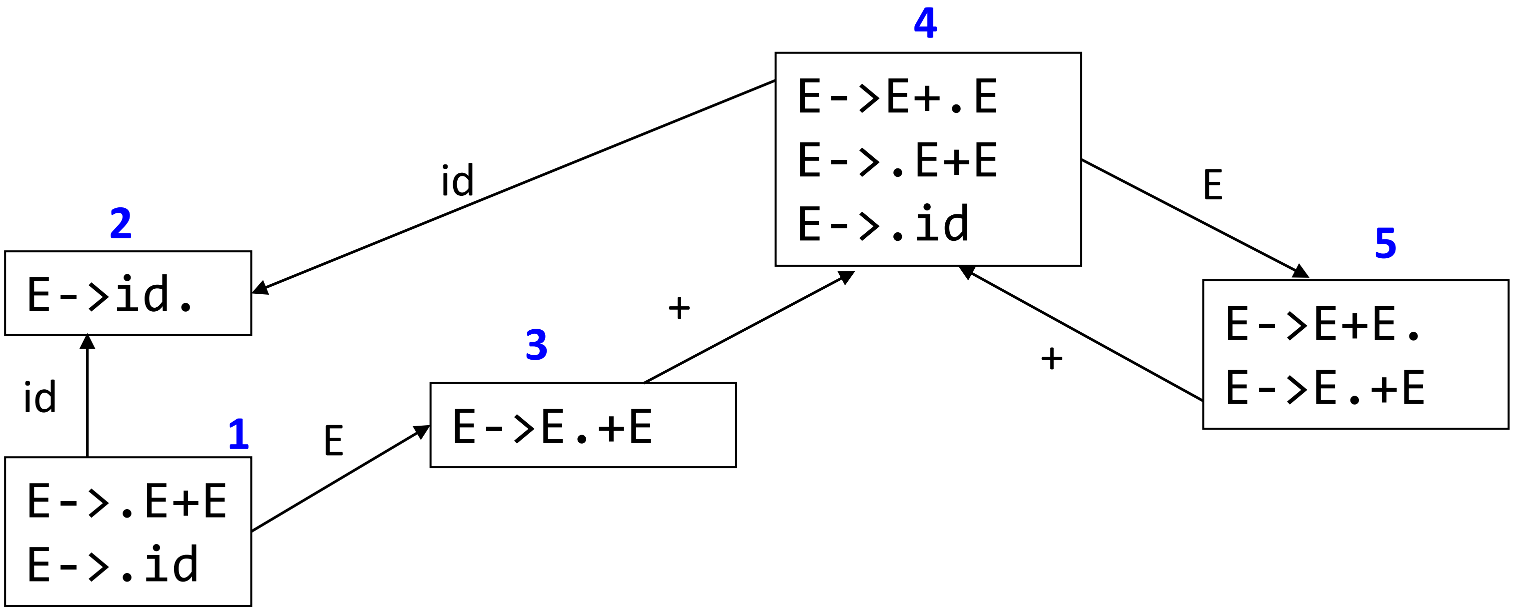
`Follow(E) = { $, ) }` => reduce by E->T. only if <u>next input</u> is $ or)

lookahead 1



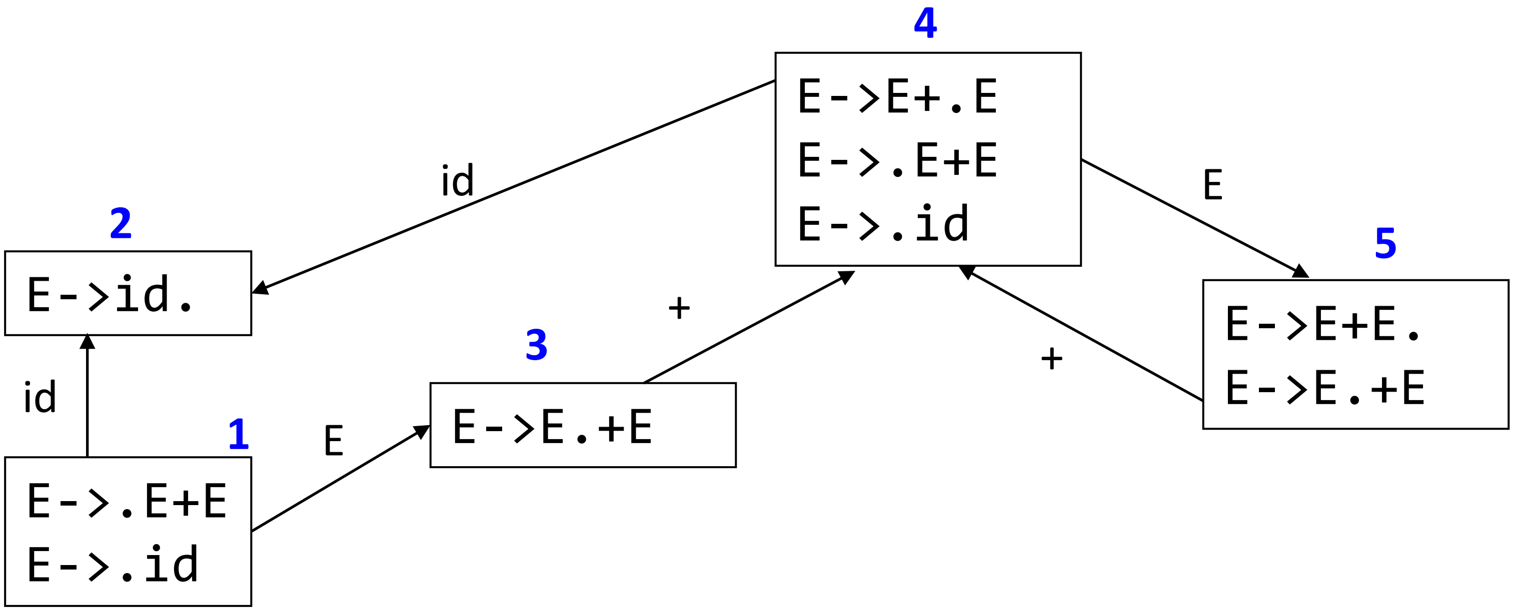
What about the grammar **`E-> E + E | id`** ?

`LR(0)?`



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

`LR(0)? SLR(1)?`



4

E->E+.E
E->.E+E
E->.id

2

E->id.

3

E->E.+E

5

E->E+E.
E->E.+E

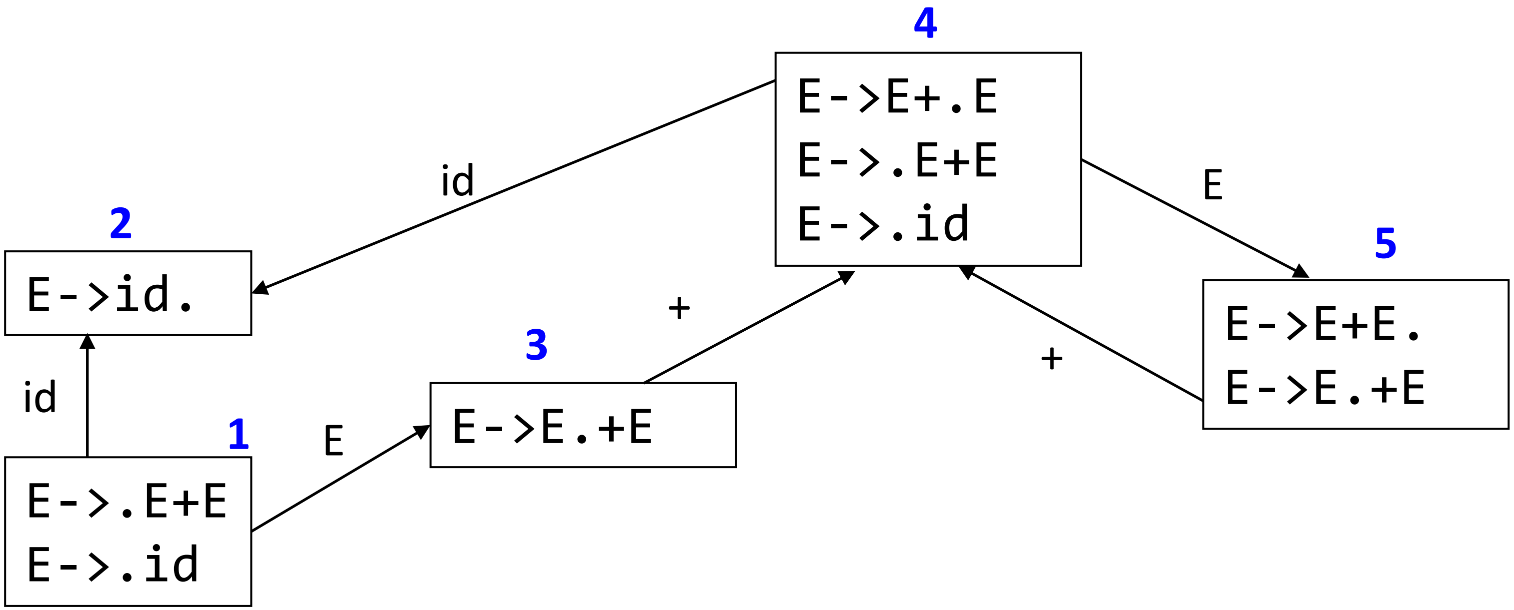
1

E->.E+E
E->.id

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

LR(0)? SLR(1)?

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



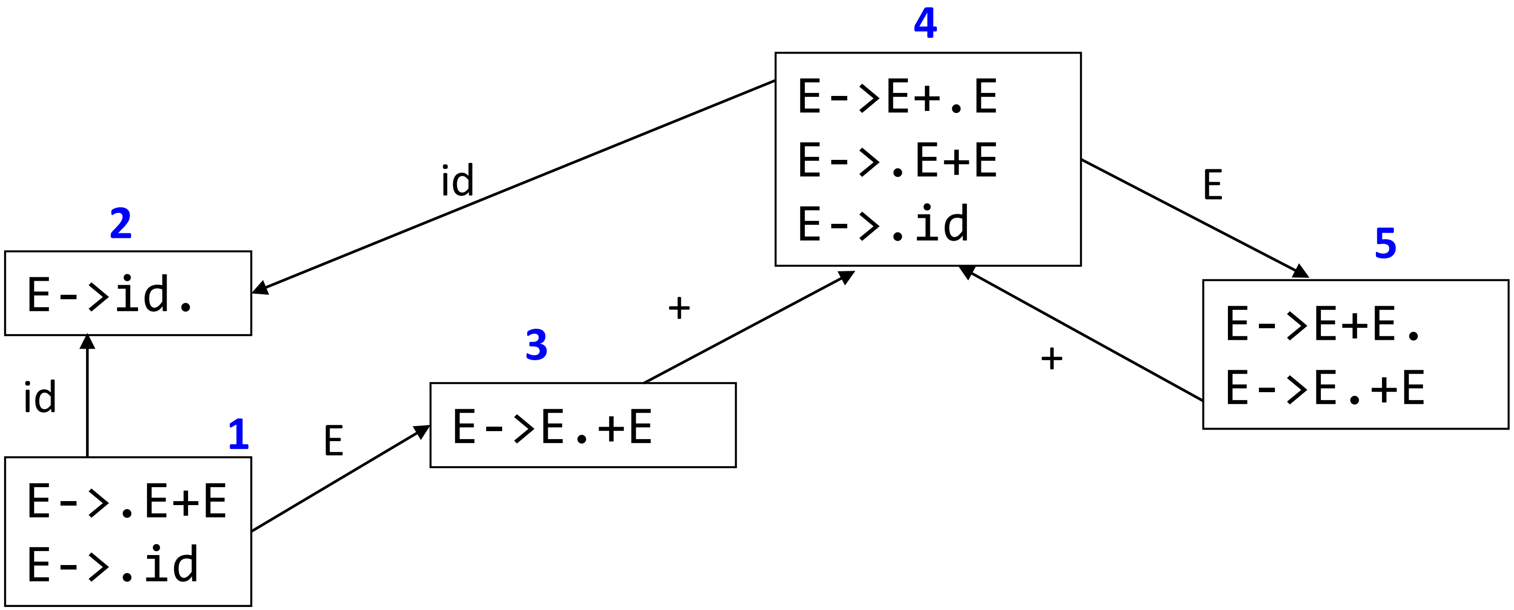
What about the grammar **E-> E + E | id** ?

LR(0)? SLR(1)?

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

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,` reduce by E->E+E. only if next input is $ or +

`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

Discussion: LR and LL Parsers

- LR Parsers:
  - For the next token, `t,` in input sequence, LR parsers try to answer: i) should I put this token on stack? or ii) should I replace a set of tokens that are at the top of a stack?

    In shift states (case i), if there is no transition out of that state for `t`, it is a syntax error.
- LL Parsers:
  - LL parsers ask the question: which rule should I use next based on the next input token `t`?. Only after expanding all non-terminals of the rule considered, they move on to consume the subsequent input tokens

Discussion: LR and LL Parsers

Grammar:
1: S -> F
2: S -> (S + F)
3: F -> a

input:
(a+)

Accepted or Not
accepted?

Parse Table (Top-Down)

Discussion: LR and LL Parsers

Goto and Action Table?

Grammar:
1: S -> F
2: S -> (S + F)
3: F -> a

input:
(a+)

Accepted or Not accepted?

Hand-Written Parser - FPE

- Fully parenthesized expression (FPE)
  - Expressions (algebraic notation) are the normal way we are used to seeing them. E.g. 2 + 3
  - *Fully-parenthesized* expressions are simpler versions: every binary operation is enclosed in parenthesis
    - E.g. 2 + 3 is written as (2+3)
    - E.g. (2 + (3 * 7))
    - We can ignore order-of-operations (PEMDAS rule) in FPEs.

FPE – definition

- Either a:
  1. A number (integer in our example) OR
  2. *Open parenthesis* ‘(‘ followed by
     FPE followed by
     an operator (‘+’, ‘-’, ‘*’, ‘/’) followed by
     FPE followed by
     closed parenthesis ‘)’

FPE – Notation

```
1. E -> INTLITERAL
2. E -> (E op E)
3. op -> ADD | SUB | MUL | DIV
```

Implementing a parser for FPE

1. One function defined for every non-terminal
   - `E, op`
2. One function defined for every production
   - `E1, E2`
3. One function defined for all terminals
   - `IsTerm`

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB | MUL | DIV
```

Implementing a parser for FPE

1. One function defined for every non-terminal
   - `E, op`
2. One function defined for every production
   - `E1, E2`
3. One function defined for all terminals
   - `IsTerm`

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB | MUL | DIV
```

Implementing a parser for FPE

This function checks if the next token returned by the scanner matches the expected token. Returns `true` *if match.* `false` *if no match.*

Assume that a scanner module has been provided. The scanner has one function, `GetNextToken`, that returns the next token in the sequence.

Can be any one of: INTLITERAL, LPAREN, RPAREN, ADD, SUB, MUL, DIV

```
bool IsTerm(Scanner* s, TOKEN tok) {
    return s->GetNextToken() == tok;
}
```

Implementing a parser for FPE

1. One function defined for every non-terminal
   - `E, op`
2. One function defined for every production
   - `E1, E2`
3. One function defined for all terminals
   - `IsTerm`

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB | MUL | DIV
```

Implementing a parser for FPE

This function implements production #1: E->INTLITERAL
Returns `true` *if the next token returned by the scanner is an INTLITERAL.* `false` *otherwise.*

```
bool E1(Scanner* s) {

   return IsTerm(s, INTLITERAL);

}
```

Implementing a parser for FPE

1. One function defined for every non-terminal
    - `E, op`
2. One function defined for every production
    - `E1, E2`
3. One function defined for all terminals
    - `IsTerm`

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB | MUL | DIV
```

Implementing a parser for FPE

This function implements production #2: `E->(E op E)`
Returns `true` *if the Boolean expression on line 2 returns* `true`*.* `false` *otherwise.*

```
1: bool E2(Scanner* s) {

2:  return IsTerm(s, LPAREN) &&
           E(s)              &&
           OP(s)             &&
           E(s)              &&
           IsTerm(s, RPAREN);

3: }
```

Implementing a parser for FPE

1. One function defined for every non-terminal
    - `E`, `op`
2. One function defined for every production
    - `E1`, `E2`
3. One function defined for all terminals
    - `IsTerm`

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB | MUL | DIV
```

Implementing a parser for FPE

This function implements production #3: `op->ADD|SUB|MUL|DIV`
Returns `true` *if the next token returned by the scanner is any one from* `ADD, SUB, MUL, DIV`*.* `false` *otherwise.*

```
bool OP(Scanner* s) {

   TOKEN tok = s->GetNextToken();

   if((tok == ADD) || (tok == SUB) || (tok ==
   MUL) || (tok == DIV))
      return true;

   return false;

}
```

Implementing a parser for FPE

1. One function defined for every non-terminal
   - `E, op`
2. One function defined for every production
   - `E1, E2`
3. One function defined for all terminals
   - `IsTerm`

```
1.E -> INTLITERAL
2.E -> (E op E)
3.op -> ADD | SUB | MUL | DIV
```

Implementing a parser for FPE

This function implements the routine for matching non-terminal E

Assume that GetCurTokenSequence returns a reference to the first token in a sequence of tokens maintained by the scanner

```
bool E(Scanner* s) {

   TOKEN* prevToken = s->GetCurTokenSequence();
   if(!E1(s)) {
      s->SetCurTokenSequence(prevToken);
      return E2(s);
   }
   return true;

}
```

Implementing a parser for FPE

This function implements the routine for matching non-terminal E

```
bool E(Scanner* s) {

    TOKEN* prevToken = s->GetCurTokenSequence();
    if(!E1(s)) {
        s->SetCurTokenSequence(prevToken);
        return E2(s);
    }
    return true;

}
```

//This line implements the check to see if the sequence of tokens match production #1: E->INTLITERAL.

Implementing a parser for FPE

This function implements the routine for matching non-terminal E

```
bool E(Scanner* s) {

   TOKEN* prevToken = s->GetCurTokenSequence();
   if(!E1(s)) {
       s->SetCurTokenSequence(prevToken);
       return E2(s);
   }
   return true;
```

//because E1(s) calls s->GetNextToken() internally, the reference to the sequence of tokens would have moved forward. This line restores the reference back to the first node in the sequence so that the scanner provides the correct sequence to the call E2 in next line

Implementing a parser for FPE

This function implements the routine for matching non-terminal E

```
bool E(Scanner* s) {

   TOKEN* prevToken = s->GetCurTokenSequence();
   if(!E1(s)) {
      s->SetCurTokenSequence(prevToken);
      return E2(s);
   }
   return true;
```

//This line implements the check to see if the sequence of tokens match production #2: E->(E op E)

Implementing a parser for FPE

```
IsTerm(Scanner* s, TOKEN tok) { return s->GetNextToken() == tok;}

bool E1(Scanner* s) {
     return IsTerm(s, INTLITERAL);
}

bool E2(Scanner* s) { return IsTerm(s, LPAREN) && E(s) && OP(s) && E(s) && IsTerm(s, RPAREN); }

bool OP(Scanner* s) {
     TOKEN tok = s->GetNextToken();
     if((tok == ADD) || (tok == SUB) || (tok == MUL) || (tok == DIV))
          return true;
     return false;
}

bool E(Scanner* s) {
     TOKEN* prevToken = s->GetCurTokenSequence();
     if(!E1(s)) {
          s->SetCurTokenSequence(prevToken);
          return E2(s);
     }
     return true;
}
```

Start the parser by invoking E().
Value returned tells if the expression is FPE or not.

Exercise

- What parsing technique does this parser use?