

CS620T: Compilers - Principles and Implementation

Autumn 2026

Week 3: Design and Implementation of
scanners and parsers contd..

Parsers – what do we need to know?

1. How do we define language constructs?
 - Context-free grammars
2. How do we write Parsers?
 - E.g. use a parser generator tool such as [Bison](#)
3. How do we determine: 1) valid strings in the language? 2) structure of program?
 - [LL Parsers, LR Parsers](#)



Next

Top-down Parsing

- Idea: we know sentence has to start with initial symbol
- Build up partial derivations by *predicting* what rules are used to expand non-terminals
- Often called *predictive parsers*
- If partial derivation has terminal characters, *match* them from the input stream

Top-down Parsing

- Also called recursive-descent parsing
- Equivalent to finding the left-derivation for an input string
 - Recall: expand the leftmost non-terminal in a parse tree
 - Expand the parse tree in pre-order i.e., identify parent nodes before children

Top-down Parsing

↑: next symbol to
be read

1: $S \rightarrow cAd$
2: $A \rightarrow ab$
3: | a

Step	Input string	Parse tree
1	cad ↑	S

String: cad

Start with S

Top-down Parsing

↑: next symbol to be read

1: $S \rightarrow cAd$

2: $A \rightarrow ab$

3: | a

Step	Input string	Parse tree
1	cad	S
2	↑ cad ↑	<pre>graph TD; S --> c; S --> A; S --> d; A --> a; A --> b; A --> a;</pre>

String: cad

Predict rule 1

Top-down Parsing

↑: next symbol to be read

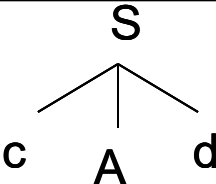
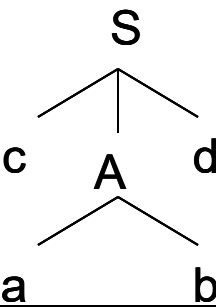
1: $S \rightarrow cAd$

2: $A \rightarrow ab$

3: | a

String: cad

Predict rule 2

Step	Input string	Parse tree
1	cad	S
2	↑ cad ↑	
3	cad ↑	

Top-down Parsing

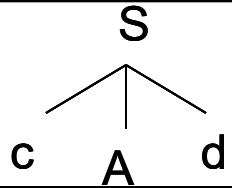
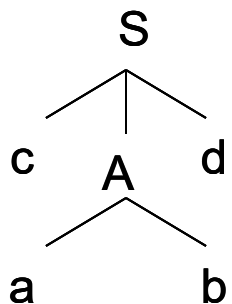
↑: next symbol to be read

1: $S \rightarrow cAd$

2: $A \rightarrow ab$

3: | a

String: cad

Step	Input string	Parse tree
1	cad	S
2	↑ cad	 <pre>graph TD; S1[S] --- c1[c]; S1 --- A1[A]; S1 --- d1[d];</pre>
3	cad ↑	 <pre>graph TD; S2[S] --- c2[c]; S2 --- A2[A]; S2 --- d2[d]; A2 --- a[a]; A2 --- b[b];</pre>

No more non terminals!

String doesn't match.

Backtrack.

Top-down Parsing

↑: next symbol to be read

1: $S \rightarrow cAd$

2: $A \rightarrow ab$

3: | a

Step	Input string	Parse tree
1	cad	S
2	↑ cad ↑	<pre>graph TD; S --> c; S --> A; S --> d; A --> a; A --> b;</pre>

String: cad

Top-down Parsing

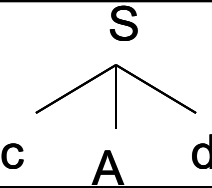
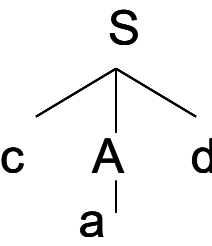
↑: next symbol to be read

1: $S \rightarrow cAd$

2: $A \rightarrow ab$

3: | a

String: cad

Step	Input string	Parse tree
1	cad	S
2	↑ cad ↑	
4	cad ↑	

Predict rule 3

Top-down Parsing – Table-driven Approach

1: $S \rightarrow F$

2: $S \rightarrow (S + F)$

3: $F \rightarrow a$

string: (a+a)

string': (a+a)\$

	(	)	a	+	\$
S	2	-	1	-	-
F	-	-	3	-	-

Assume that the table is given.

Top-down Parsing – Table-driven Approach

1: $S \rightarrow F$

2: $S \rightarrow (S + F)$

3: $F \rightarrow a$

string: (a+a)

string': (a+a)\$

	(	)	a	+	\$
S	2	-	1	-	-
F	-	-	3	-	-

Assume that the table is given.

- Table-driven (Parse Table) approach doesn't require backtracking

But how do we construct such a table?

Important Concepts: First Sets and Follow Sets

First and follow sets

- $\text{First}(\alpha)$: the set of terminals (and/or λ) that begin all strings that can be derived from α

- $\text{First}(A) = \{x, y, \lambda\}$

- $\text{First}(xA) = \{x\}$

- $\text{First}(AB) = \{x, y, b\}$

- $\text{Follow}(A)$: the set of terminals (and/or \$, but no λ s) that can appear immediately after A in some partial derivation

- $\text{Follow}(A) = \{b\}$

$$S \rightarrow A B \$$$

$$A \rightarrow x a A$$

$$A \rightarrow y a A$$

$$A \rightarrow \lambda$$

$$B \rightarrow b$$

First and follow sets

- $\text{First}(\alpha) = \{a \in V_t \mid \alpha \Rightarrow^* a\beta\} \cup \{\lambda \mid \text{if } \alpha \Rightarrow^* \lambda\}$
- $\text{Follow}(A) = \{a \in V_t \mid S \Rightarrow^+ \dots Aa \dots\} \cup \{\$ \mid \text{if } S \Rightarrow^+ \dots A \$\}$

S: start symbol

a: a terminal symbol

A: a non-terminal symbol

α, β : a string composed of terminals and non-terminals (typically, α is the RHS of a production

$\Rightarrow$: derived in 1 step

$\Rightarrow^*$: derived in 0 or more steps

$\Rightarrow^+$: derived in 1 or more steps

Computing first sets

- Terminal: $\text{First}(a) = \{a\}$
- Non-terminal: $\text{First}(A)$
 - Look at all productions for A
$$A \rightarrow X_1 X_2 \dots X_k$$
 - $\text{First}(A) \supseteq (\text{First}(X_1) - \lambda)$
 - If $\lambda \in \text{First}(X_1)$, $\text{First}(A) \supseteq (\text{First}(X_2) - \lambda)$
 - If λ is in $\text{First}(X_i)$ for all i , then $\lambda \in \text{First}(A)$
- Computing $\text{First}(\alpha)$: similar procedure to computing $\text{First}(A)$

Towards parser generators

- Key problem: as we read the source program, we need to decide what productions to use
- Step 1: find the tokens that can tell which production P (of the form $A \rightarrow X_1 X_2 \dots X_m$) applies

$\text{Predict}(P) =$

$$\begin{cases} \text{First}(X_1 \dots X_m) & \text{if } \lambda \notin \text{First}(X_1 \dots X_m) \\ (\text{First}(X_1 \dots X_m) - \lambda) \cup \text{Follow}(A) & \text{otherwise} \end{cases}$$

- If next token is in $\text{Predict}(P)$, then we should choose this production

First Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

$\text{first}(S) = \{ \text{?} \}$

Think of all possible strings derivable from S .
Get the **first terminal symbol** in those strings
or λ if S derives λ

First Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

$\text{first}(S) = \{ x, y, c \}$

First Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

$\text{first}(S) = \{ x, y, c \}$

$\text{first}(A) = \{ ? \}$

First Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

$\text{first}(S) = \{ x, y, c \}$

$\text{first}(A) = \{ x, y, c \}$

First Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

$\text{first}(S) = \{ x, y, c \}$
 $\text{first}(A) = \{ x, y, c \}$
 $\text{first}(B) = \{ \text{?} \}$

First Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

$\text{first}(S) = \{ x, y, c \}$
 $\text{first}(A) = \{ x, y, c \}$
 $\text{first}(B) = \{ b, \lambda \}$

Follow Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

$\text{follow}(S) = \{ \text{?} \}$

Think of all strings **possible in the language** having the form $..Sa..$. Get the **following terminal symbol** a after S in those strings or $\$$ if you get a string $..S\$$

Follow Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

$\text{follow}(S) = \{ \quad \}$

Follow Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

$\text{follow}(S) = \{ \quad \}$
 $\text{follow}(A) = \{ \text{?} \}$

Follow Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

$\text{follow}(S) = \{ \quad \}$

$\text{follow}(A) = \{ b, c \}$

e.g. $xaAbc\$$, $xaAc\$$

Follow Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

$\text{follow}(S) = \{ \quad \}$

$\text{follow}(A) = \{ b, c \}$ e.g. $xaAbc\$$, $xaAc\$$

What happens when you consider: $A \rightarrow xaA$ or $A \rightarrow yaA$?

Follow Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

$\text{follow}(S) = \{ \quad \}$

$\text{follow}(A) = \{ b, c \}$ e.g. $xaAbc\$$, $xaAc\$$

What happens when you consider: $A \rightarrow xaA$ or $A \rightarrow yaA$?

- You will get string of the form $A \Rightarrow^+ (xa)^+A$
- But we need strings of the form: $\dots Aa \dots$ or $\dots Ab \dots$ or $\dots Ac \dots$

Follow Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

$\text{follow}(S) = \{ \quad \}$
 $\text{follow}(A) = \{ b, c \}$
 $\text{follow}(B) = \{ ? \}$

Follow Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

$\text{follow}(S) = \{ \quad \}$
 $\text{follow}(A) = \{ b, c \}$
 $\text{follow}(B) = \{ c \}$

Predict Set - Example

1) $S \rightarrow ABc\$$

2) $A \rightarrow xaA$

3) $A \rightarrow yaA$

4) $A \rightarrow c$

5) $B \rightarrow b$

6) $B \rightarrow \lambda$

$\text{Predict}(P) =$

$$\begin{cases} \text{First}(X_1 \dots X_m) & \text{if } \lambda \notin \text{First}(X_1 \dots X_m) \\ (\text{First}(X_1 \dots X_m) - \lambda) \cup \text{Follow}(A) & \text{otherwise} \end{cases}$$

$\text{Predict}(1) = \{ ? \} = \text{First}(ABc\$) \text{ if } \lambda \notin \text{First}(ABc\$)$

Predict Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

	x	y	a	b	c	\$
S	1	1			1	
A						
B						

Predict (1) = { x, y, c }

Predict Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

	x	y	a	b	c	\$
S	1	1			1	
A						
B						

Predict (1) = { x, y, c }

Predict (2) = { ? } = First(xaA) if $\lambda \notin \text{First}(xaA)$

Predict Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

	x	y	a	b	c	\$
S	1	1			1	
A	2					
B						

Predict (1) = { x, y, c }

Predict (2) = { x }

Predict Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

	x	y	a	b	c	\$
S	1	1			1	
A	2					
B						

Predict (1) = { x, y, c }

Predict (2) = { x }

Predict (3) = { ? } = First(yaA) if $\lambda \notin \text{First}(yaA)$

Predict Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

	x	y	a	b	c	\$
S	1	1			1	
A	2	3				
B						

Predict (1) = { x, y, c }

Predict (2) = { x }

Predict (3) = { y }

Predict Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

	x	y	a	b	c	\$
S	1	1			1	
A	2	3				
B						

Predict (1) = { x, y, c }

Predict (2) = { x }

Predict (3) = { y }

Predict (4) = { ? } = First(c) if $\lambda \notin \text{First}(c)$

Predict Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

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

Predict (1) = { x, y, c }
Predict (2) = { x }
Predict (3) = { y }
Predict (4) = { c }

Predict Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

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

Predict (1) = { x, y, c }

Predict (2) = { x }

Predict (3) = { y }

Predict (4) = { c }

Predict (5) = { ? } = First(b) if $\lambda \notin \text{First}(b)$

Predict Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

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

Predict (1) = { x, y, c }
Predict (2) = { x }
Predict (3) = { y }
Predict (4) = { c }
Predict (5) = { b }

Predict Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

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

Predict (1) = { x, y, c }

Predict (2) = { x }

Predict (3) = { y }

Predict (4) = { c }

Predict (5) = { b }

Predict (6) = { ? } = First(λ) ?

Predict(P) =

$$\begin{cases} \text{First}(X_1 \dots X_m) & \text{if } \lambda \notin \text{First}(X_1 \dots X_m) \\ (\text{First}(X_1 \dots X_m) - \lambda) \cup \text{Follow}(A) & \text{otherwise} \end{cases}$$

Predict Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

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

Predict (1) = { x, y, c }

Predict (2) = { x }

Predict (3) = { y }

Predict (4) = { c }

Predict (5) = { b }

Predict (6) = { ? } = ~~First(λ)~~ ? Follow(B)

Predict(P) =

$$\begin{cases} \text{First}(X_1 \dots X_m) & \text{if } \lambda \notin \text{First}(X_1 \dots X_m) \\ (\text{First}(X_1 \dots X_m) - \lambda) \cup \text{Follow}(A) & \text{otherwise} \end{cases}$$

Predict Set - Example

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

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

Predict (1) = { x, y, c }

Predict (2) = { x }

Predict (3) = { y }

Predict (4) = { c }

Predict (5) = { b }

Predict (6) = { λ }

Computing Parse-Table

- 1) $S \rightarrow ABc\$$
- 2) $A \rightarrow xaA$
- 3) $A \rightarrow yaA$
- 4) $A \rightarrow c$
- 5) $B \rightarrow b$
- 6) $B \rightarrow \lambda$

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

$\text{first}(S) = \{x, y, c\}$
 $\text{first}(A) = \{x, y, c\}$
 $\text{first}(B) = \{b, \lambda\}$

$\text{follow}(S) = \{\}$
 $\text{follow}(A) = \{b, c\}$
 $\text{follow}(B) = \{c\}$

$P(1) = \{x, y, c\}$
 $P(2) = \{x\}$
 $P(3) = \{y\}$
 $P(4) = \{c\}$
 $P(5) = \{b\}$
 $P(6) = \{c\}$

Top-Down Parsing - Example

string: xacc\$

Stack

?

Rem. Input

xacc\$

Action

?

What do you put on the stack?

Top-Down Parsing - Example

string: xacc\$

Stack

?

Rem. Input

xacc\$

Action

?

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

Top-Down Parsing - Example

string: xacc\$

Stack*

S

Rem. Input

xacc\$

Action

?

Top-down parsing. So, start with S.

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?

Top-Down Parsing - Example

string: xacc\$

Stack*

S

Rem. Input

xacc\$

Action

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	

Top-Down Parsing - Example

string: xacc\$

Stack*

S
ABc\$

Rem. Input

xacc\$

Action

Predict(1) S → ABc\$

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

Top-Down Parsing - Example

string: xacc\$

Stack*

S
ABc\$

Rem. Input

xacc\$
xacc\$

Action

Predict(1) S->ABc\$

*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	

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	

Top-Down Parsing - Example

string: xacc\$

Stack*

S

ABc\$

xaABc\$

Rem. Input

xacc\$

xacc\$

Action

Predict(1) S → ABc\$

Predict(2) A → xaA

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

Top-Down Parsing - Example

string: xacc\$

Stack*	Rem. Input	Action
S	xacc\$	Predict(1) S → ABc\$
ABc\$	xacc\$	Predict(2) A → xaA
x aABc\$	x acc\$	?

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

Top-Down Parsing - Example

string: xacc\$

Stack*	Rem. Input	Action
S	xacc\$	Predict(1) S → ABc\$
ABc\$	xacc\$	Predict(2) A → xaA
x aABc\$	x acc\$	match(x)

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

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	

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	

Top-Down Parsing - Example

string: xacc\$

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

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

Top-Down Parsing - Example

string: xacc\$

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

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\$		

Top-Down Parsing - Example

string: xacc\$

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

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\$	?

Top-Down Parsing - Example

string: xacc\$

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

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
c Bc\$	c c\$	match(c)

Top-Down Parsing - Example

string: xacc\$

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

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\$	?

Top-Down Parsing - Example

string: xacc\$

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

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

Top-Down Parsing - Example

string: xacc\$

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

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)
B c\$	c\$	Predict(6) B → λ
c\$		

Top-Down Parsing - Example

string: xacc\$

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

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\$	?

Top-Down Parsing - Example

string: xacc\$

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

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)

Top-Down Parsing - Example

string: xacc\$

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

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!

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 $A \rightarrow \alpha \mid \beta$, 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 $\beta \xRightarrow{*} \epsilon$, then α does not derive any string beginning with a terminal in $\text{Follow}(A)$. If $\alpha \xRightarrow{*} \epsilon$, then β does not derive any string beginning with a terminal in $\text{Follow}(A)$

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`

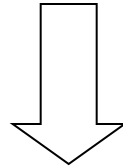
- This is not LL(1) (why?)
- We can turn this in to

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

`<if suffix> → endif`

`<if suffix> → else <stmt list> endif`

Left Factoring

$$A \rightarrow \alpha \beta \mid \alpha \mu$$

$$A \rightarrow \alpha N$$
$$N \rightarrow \beta$$
$$N \rightarrow \mu$$

Left recursion

- *Left recursion* is a problem for LL(1) parsers
 - LHS is also the first symbol of the RHS
- Consider:
$$E \rightarrow E + T$$
- 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:

$$E \rightarrow E + T$$

- What would happen with the stack-based algorithm?

E

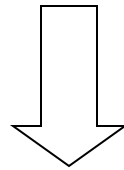
E + T

E + T + T

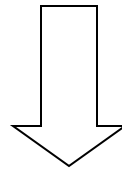
E + T + T + T

...

Eliminating Left Recursion

$$A \rightarrow A\alpha \mid \beta$$

$$A \rightarrow NT$$
$$N \rightarrow \beta$$
$$T \rightarrow \alpha T$$
$$T \rightarrow \lambda$$

Eliminating Left Recursion

$$E \rightarrow E + T \mid T$$

$$E \rightarrow E_1 \text{ Etail}$$
$$E_1 \rightarrow T$$
$$\text{Etail} \rightarrow + T \text{ Etail}$$
$$\text{Etail} \rightarrow \lambda$$

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:

$$A \rightarrow ax \mid ay$$

- More lookahead leads to more powerful parsers
- What are the downsides?

LL(k)? - Example

string: ((x+x))\$

- 1) $S \rightarrow E$
- 2) $E \rightarrow (E+E)$
- 3) $E \rightarrow (E-E)$
- 4) $E \rightarrow x$

Stack*	Rem. Input	Action
S	((x+x))\$	Predict(1) $S \rightarrow E$
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?

$$\begin{aligned} S &\rightarrow [S \\ S &\rightarrow SI \\ SI &\rightarrow [SI] \\ SI &\rightarrow \lambda \end{aligned}$$

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 ”

$$\begin{array}{lcl} S & \rightarrow & [S C \\ S & \rightarrow & \lambda \\ C & \rightarrow &] \\ C & \rightarrow & \lambda \end{array}$$

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

$$\begin{array}{lcl} S & \rightarrow & \text{if } S \text{ E} \\ S & \rightarrow & \text{other} \\ E & \rightarrow & \text{else } S \text{ endif} \\ E & \rightarrow & \text{endif} \end{array}$$

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!

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. $E \rightarrow \text{INTLITERAL}$

2. $E \rightarrow (E \text{ op } E)$

3. $\text{op} \rightarrow \text{ADD} \mid \text{SUB} \mid \text{MUL} \mid \text{DIV}$

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 $\rightarrow$ INTLITERAL

2.E $\rightarrow$ (E op E)

3.op $\rightarrow$ 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 $\rightarrow$ INTLITERAL

2.E $\rightarrow$ (E op E)

3.op $\rightarrow$ 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 $\rightarrow$ INTLITERAL

2.E $\rightarrow$ (E op E)

3.op $\rightarrow$ ADD | SUB | MUL | DIV

Implementing a parser for FPE

This function implements production #1: $E \rightarrow \text{INTLITERAL}$

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 $\rightarrow$ INTLITERAL

2.E $\rightarrow$ (E op E)

3.op $\rightarrow$ ADD | SUB | MUL | DIV

Implementing a parser for FPE

This function implements production #2: $E \rightarrow (E \text{ op } E)$

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 $\rightarrow$ INTLITERAL

2.E $\rightarrow$ (E op E)

3.op $\rightarrow$ ADD | SUB | MUL | DIV

Implementing a parser for FPE

This function implements production #3: $op \rightarrow \text{ADD} \mid \text{SUB} \mid \text{MUL} \mid \text{DIV}$

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 $\rightarrow$ INTLITERAL

2.E $\rightarrow$ (E op E)

3.op $\rightarrow$ ADD | SUB | MUL | DIV

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;  
}
```

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



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?

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

$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow id * T$

$T \rightarrow id$

Bottom-up Parsing

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

id * id + id
id * T + id

E $\rightarrow$ T + E
E $\rightarrow$ T
T $\rightarrow$ id * T
T $\rightarrow$ id

Bottom-up Parsing

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

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

E $\rightarrow$ T + E

E $\rightarrow$ T

T $\rightarrow$ id * T

T $\rightarrow$ id

Bottom-up Parsing

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

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

E $\rightarrow$ T + E
E $\rightarrow$ T
T $\rightarrow$ id * T
T $\rightarrow$ id

Bottom-up Parsing

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

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

E $\rightarrow$ T + E

E $\rightarrow$ T

T $\rightarrow$ id * T

T $\rightarrow$ id

Bottom-up Parsing

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

id * id + id

id * T + id

T + id

T + T

T + E

E

E $\rightarrow$ T + E

E $\rightarrow$ T

T $\rightarrow$ id * T

T $\rightarrow$ id

Bottom-up Parsing

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

id * id + id

id * T + id

T + id

T + T

T + E

E



E $\rightarrow$ T + E

E $\rightarrow$ T

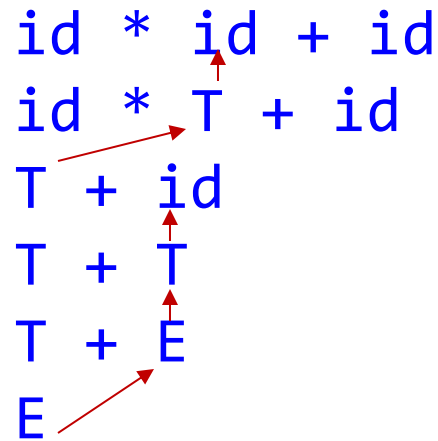
T $\rightarrow$ id * T

T $\rightarrow$ id

Bottom-up Parsing

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

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



Right-most derivation

$E \rightarrow T + E$
 $E \rightarrow T$
 $T \rightarrow id * T$
 $T \rightarrow id$

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

$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow id * T$

$T \rightarrow id$

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

$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow id * T$

$T \rightarrow id$

Bottom-up Parsing

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

id * id + id
id * T + id

E $\rightarrow$ T + E
E $\rightarrow$ T
T $\rightarrow$ id * T
T $\rightarrow$ 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. $P \rightarrow S$
2. $S \rightarrow x ; S$
3. $S \rightarrow e$

		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

Example

I) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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	$x;x;e$	?

Start with state 0

Example

I) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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)

Example

I) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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	$;x;e$	?

Example

I) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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)

Example

I) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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)

Example

I) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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	$;e$	?

Example

I) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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)

Example

I) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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	?

Example

I) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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)

Example

I) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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		?

Example

I) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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) $P \rightarrow S$

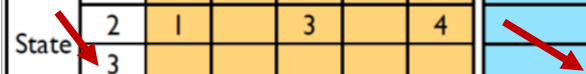
II) $S \rightarrow x;S$

III) $S \rightarrow e$ 

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) $P \rightarrow S$

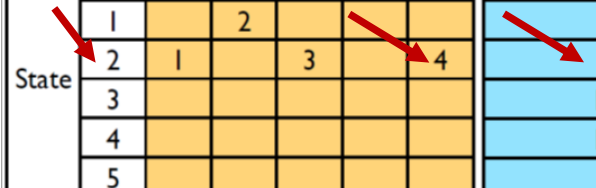
II) $S \rightarrow x;S$

III) $S \rightarrow e$ 

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) $P \rightarrow S$

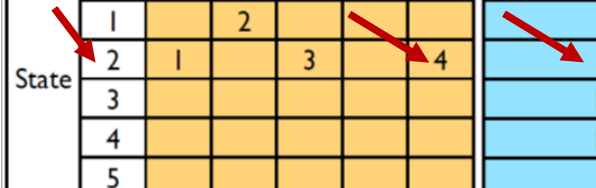
II) $S \rightarrow x;S$

III) $S \rightarrow e$ 

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) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

Input string

$x;x;e$

		Symbol					Action
State	0	x	;	e	P	S	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) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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
8	0 1 2		

CS620, IIT Dharwad

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

Example

I) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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
8	0 1 2		

CS620, IIT Dharwad

- 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) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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		

CS620, IIT Dharwad

- 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) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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		?

Example

I) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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		Reduce 2

Example

I) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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		Reduce 2
9	0		

Example

I) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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		Reduce 2 (shift(5))
9	0 5		

Example

I) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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		Reduce 2 (shift(5))
9	0 5		?

CS620, IIT Dharwad

Example

I) $P \rightarrow S$

II) $S \rightarrow x;S$

III) $S \rightarrow e$

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		Reduce 2 (shift(5))
9	0 5		Accept

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

Example

I) $P \rightarrow S$

Input string

II) $S \rightarrow x;S$

|x;x;e

III) $S \rightarrow e$

← 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) $P \rightarrow S$

Input string

II) $S \rightarrow x;S$

|x;x;e

III) $S \rightarrow e$

← 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) $P \rightarrow S$

Input string

II) $S \rightarrow x;S$

|x;x;e

III) $S \rightarrow e$

← 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) $P \rightarrow S$

Input string

II) $S \rightarrow x;S$

|x;x;e

III) $S \rightarrow e$

← 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) $P \rightarrow S$

Input string

II) $S \rightarrow x;S$

|x;x;e

III) $S \rightarrow e$

← 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) $P \rightarrow S$

Input string

II) $S \rightarrow x;S$

|x;x;e

III) $S \rightarrow e$

← 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) $P \rightarrow S$

Input string

II) $S \rightarrow x;S$

|x;x;e

III) $S \rightarrow e$

← 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|

$$\begin{array}{c} S \\ | \\ x ; x ; e \end{array}$$

Example

I) $P \rightarrow S$

Input string

II) $S \rightarrow x;S$

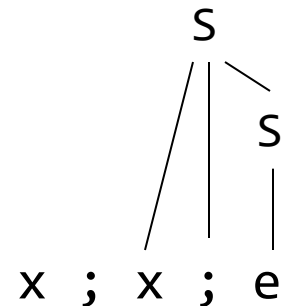
| $x;x;e$

III) $S \rightarrow e$

← 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) $P \rightarrow S$

Input string

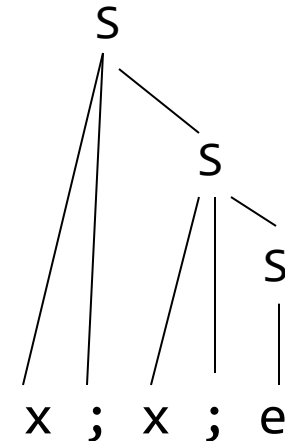
II) $S \rightarrow x;S$

| $x;x;e$

III) $S \rightarrow e$

← 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

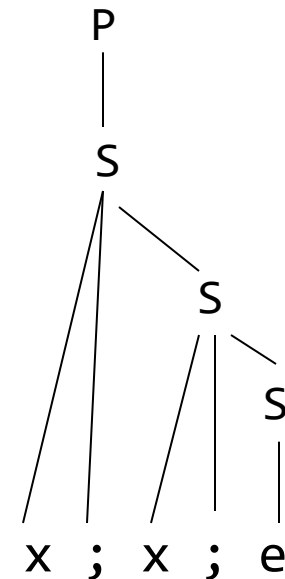


$x;S$ |
CS620, IIT Dharwad

Example

I) $P \rightarrow S$ Input string
 II) $S \rightarrow x;S$ | $x;x;e$
 III) $S \rightarrow e$ ← 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



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:

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

- 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,

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

1 add configurations of the form:

$B \rightarrow \bullet \delta$

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

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

Concept: closure

Grammar

$S \rightarrow E\$$
 $E \rightarrow E+T \mid T$
 $T \rightarrow ID \mid (E)$

➤ E.g. closure $\{S \rightarrow \bullet E\$ \}$


 $S \rightarrow \bullet E\$$

Concept: closure

➤ E.g. closure $\{S \rightarrow \bullet E \$\}$


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

Grammar

S $\rightarrow E \$$

E $\rightarrow E + T \mid T$

T $\rightarrow ID \mid (E)$

Concept: closure

➤ E.g. closure $\{S \rightarrow \bullet E \$\}$

↓
Non-terminal

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

Grammar

$S \rightarrow E \$$
 $E \rightarrow E + T \mid T$
 $T \rightarrow ID \mid (E)$

Concept: closure

➤ E.g. closure $\{S \rightarrow \bullet E \$\}$



$S \rightarrow \bullet E \$$

$E \rightarrow \bullet E + T$

$E \rightarrow \bullet T$

New Non-terminal

Grammar

$S \rightarrow E \$$

$E \rightarrow E + T \mid T$

$T \rightarrow ID \mid (E)$

Concept: closure

➤ E.g. closure $\{S \rightarrow \bullet E \$\}$



$S \rightarrow \bullet E \$$

$E \rightarrow \bullet E + T$

$E \rightarrow \bullet T$

$T \rightarrow \bullet ID$

New Non-terminal

Grammar

$S \rightarrow E \$$

$E \rightarrow E + T \mid T$

$T \rightarrow ID \mid (E)$

Concept: closure

➤ E.g. closure $\{S \rightarrow \bullet E \$\}$



$S \rightarrow \bullet E \$$

$E \rightarrow \bullet E + T$

$E \rightarrow \bullet T$

$T \rightarrow \bullet ID$

$T \rightarrow \bullet (E)$

New Non-terminal

Grammar

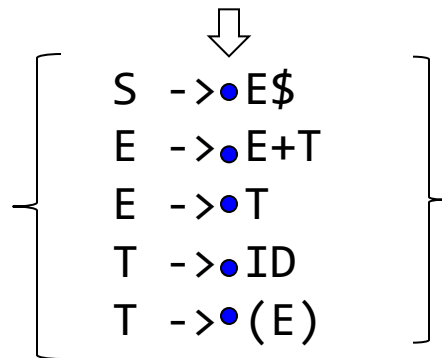
$S \rightarrow E \$$

$E \rightarrow E + T \mid T$

$T \rightarrow ID \mid (E)$

Concept: closure

➤ E.g. closure $\{S \rightarrow \bullet E\$ \}$



Grammar

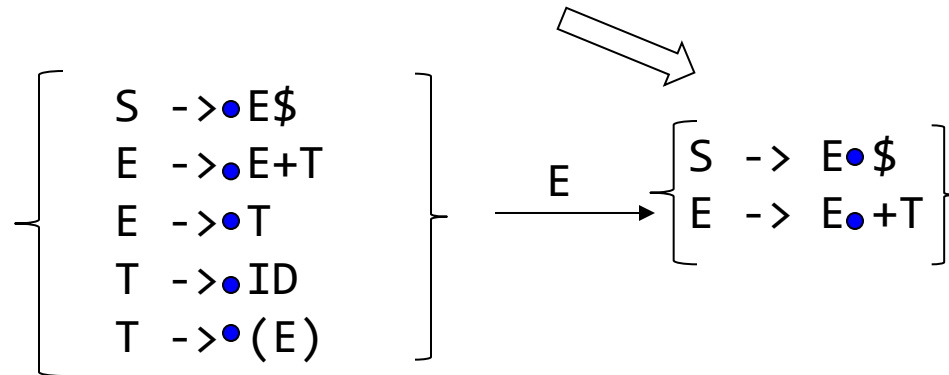
$S \rightarrow E\$$

$E \rightarrow E+T \mid T$

$T \rightarrow ID \mid (E)$

Concept: successor

➤ E.g. successor ($\{S \rightarrow \bullet E \$\}$, **E**)



Grammar

$S \rightarrow E \$$
 $E \rightarrow E + T \mid T$
 $T \rightarrow ID \mid (E)$

- 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

$P \rightarrow \bullet S$

Grammar

$P \rightarrow S$

$S \rightarrow x; S$

$S \rightarrow e$

Example: CFSM

Compute closure

$P \rightarrow \bullet S$ ← Non-terminal

Grammar

$P \rightarrow S$

$S \rightarrow x; S$

$S \rightarrow e$

Example: CFSM

Add item

$P \rightarrow \bullet S$

$S \rightarrow \bullet x; S$

Grammar

$P \rightarrow S$

$S \rightarrow x; S$

$S \rightarrow e$

Example: CFSM

Add item

$P \rightarrow \bullet S$

$S \rightarrow \bullet x; S$

$S \rightarrow \bullet e$

Grammar

$P \rightarrow S$

$S \rightarrow x; S$

$S \rightarrow e$

Example: CFSM

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

P - > • S
S - > • x ; S
S - > • e

state 0

Grammar

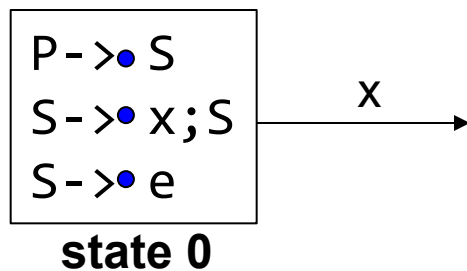
P - > S

S - > x ; S

S - > e

Example: CFSM

Compute successor (of state 0) under symbol x



Grammar

$P \rightarrow S$

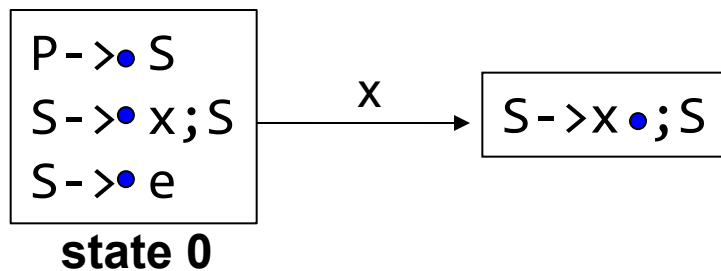
$S \rightarrow x ; S$

$S \rightarrow e$

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



Grammar

$P \rightarrow S$

$S \rightarrow x; S$

$S \rightarrow e$

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

Example: CFSM

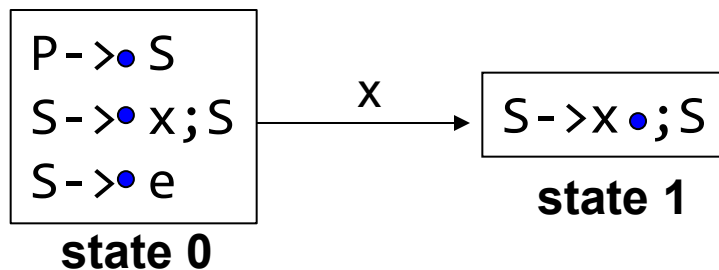
Grammar

$P \rightarrow S$

$S \rightarrow x; S$

$S \rightarrow e$

Compute successor (of state 0) under symbol x

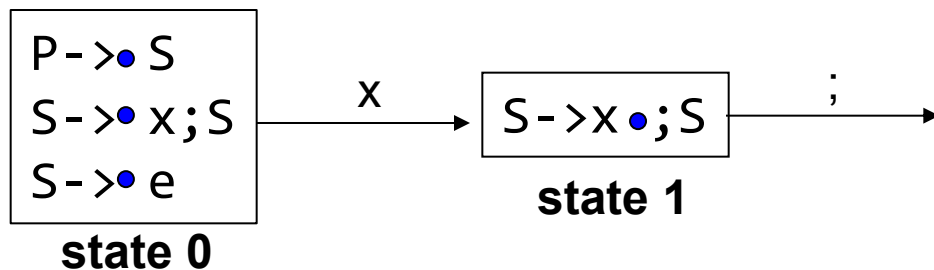


Consider items (in state 0), where x is to the immediate right of Dot.
Advance Dot by one symbol.

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

$P \rightarrow S$

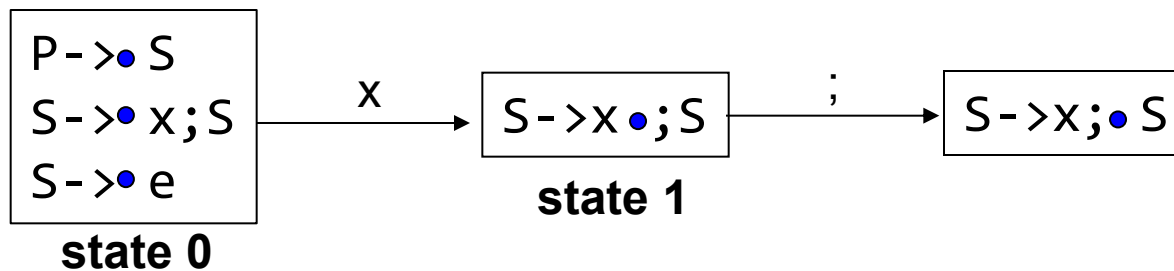
$S \rightarrow x; S$

$S \rightarrow e$

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

Example: CFSM

Compute successor (of state 1) under symbol ;



Grammar

$P \rightarrow S$

$S \rightarrow x; S$

$S \rightarrow e$

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

Example: CFSM

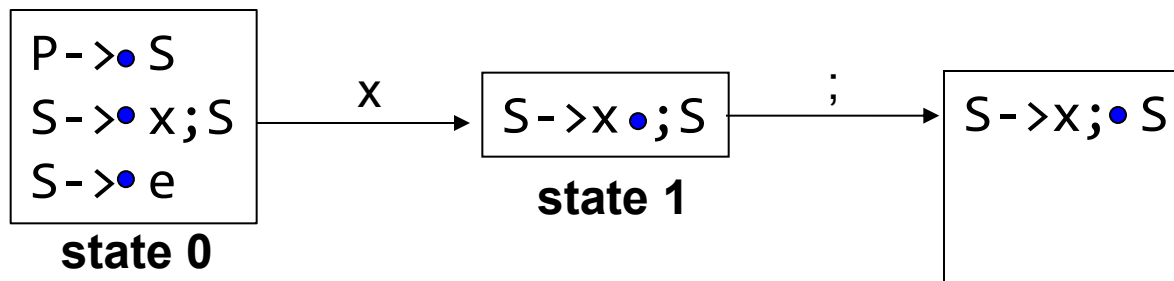
Grammar

$P \rightarrow S$

$S \rightarrow x; S$

$S \rightarrow e$

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

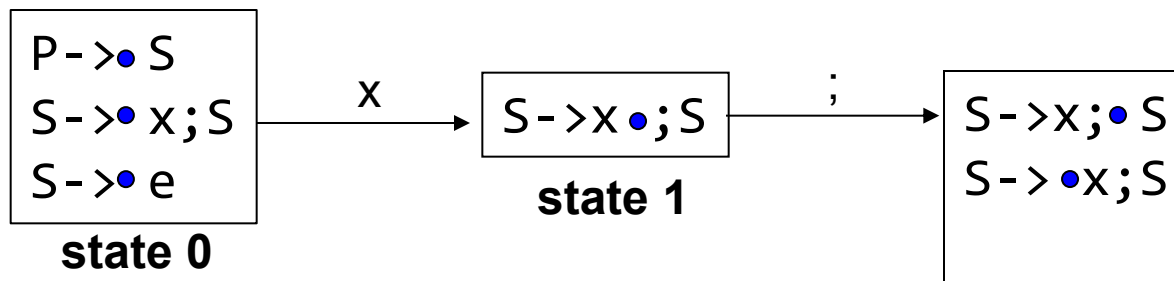
Grammar

$P \rightarrow S$

$S \rightarrow x;S$

$S \rightarrow e$

Compute successor (of state 1) under symbol ;

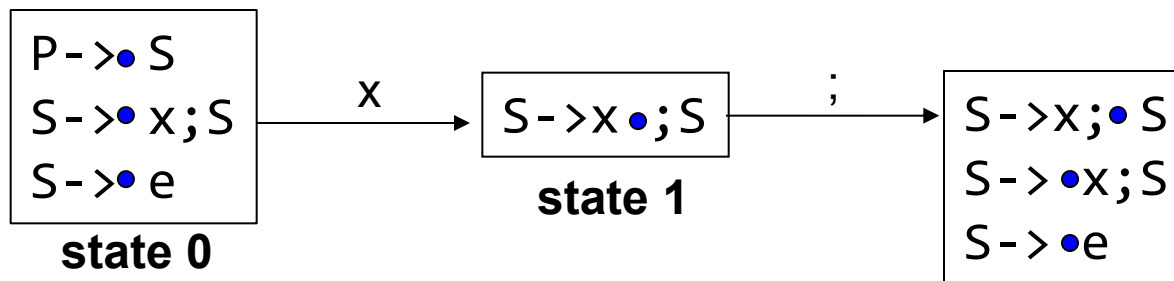


Consider items (in state 1), where ; is to the immediate right of Dot.
Advance Dot by one symbol.

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

Example: CFSM

Compute successor (of state 1) under symbol ;



Grammar

$P \rightarrow S$

$S \rightarrow x; S$

$S \rightarrow e$

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

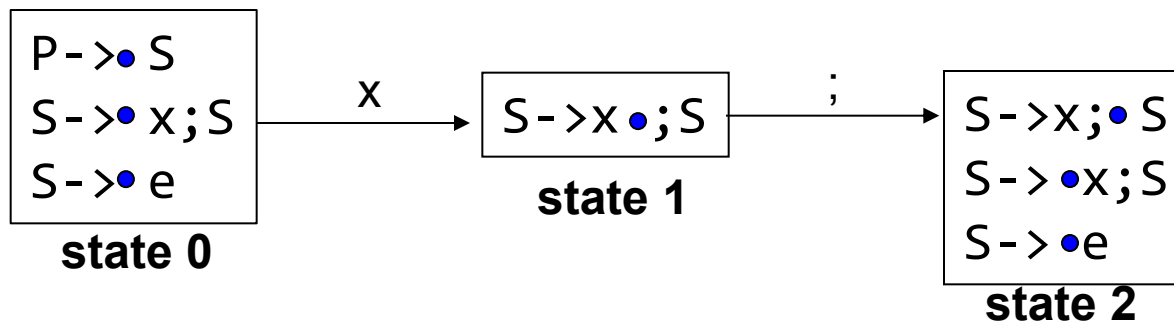
Grammar

$P \rightarrow S$

$S \rightarrow x; S$

$S \rightarrow e$

Compute successor (of state 1) under symbol ;

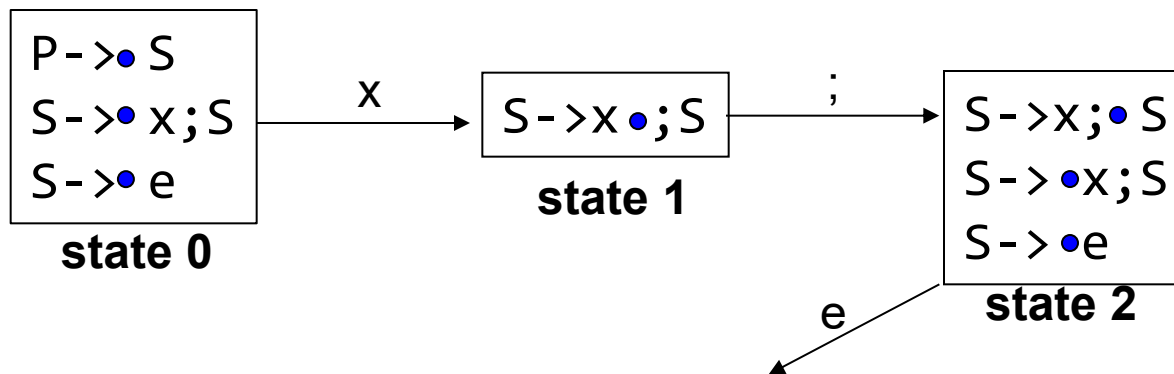


Consider items (in state 1), where ; is to the immediate right of Dot.
Advance Dot by one symbol.

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

$P \rightarrow S$

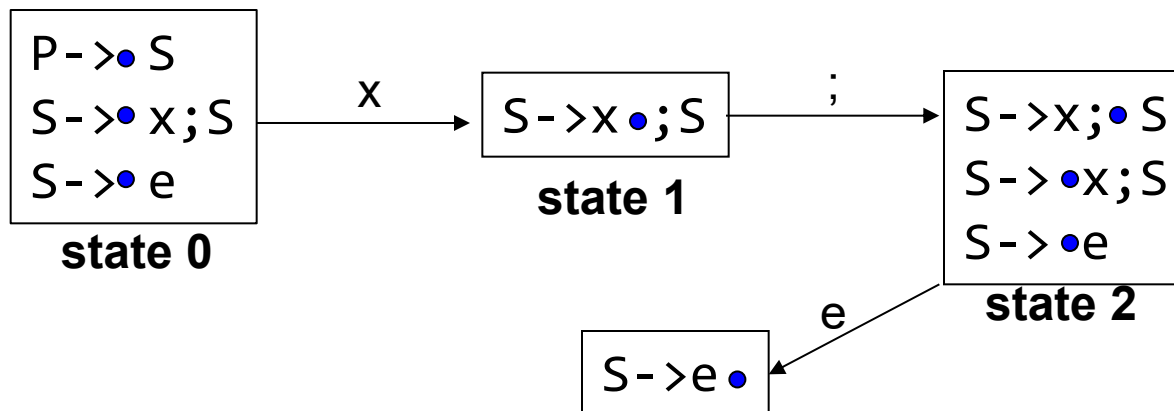
$S \rightarrow x; S$

$S \rightarrow e$

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



Grammar

$P \rightarrow S$

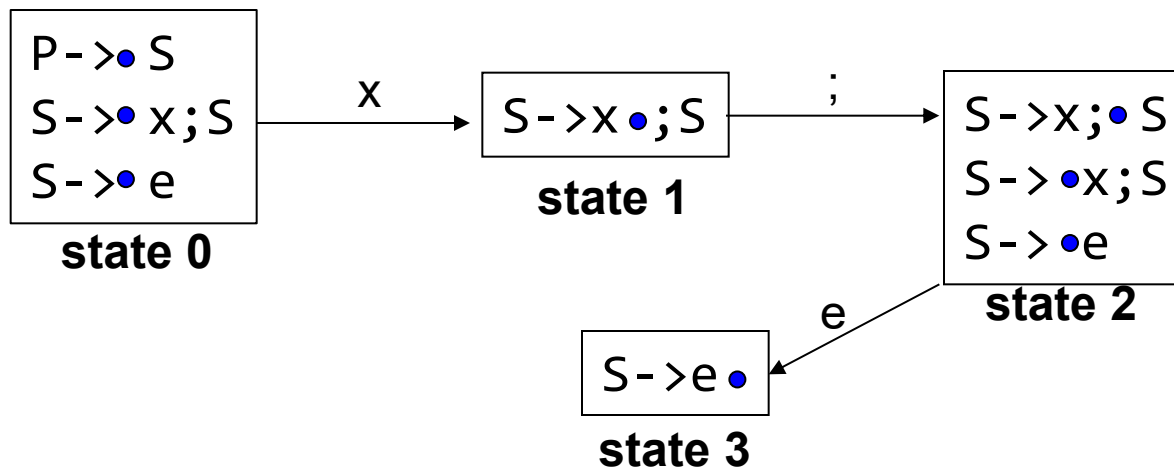
$S \rightarrow x; S$

$S \rightarrow e$

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



Grammar

$P \rightarrow S$

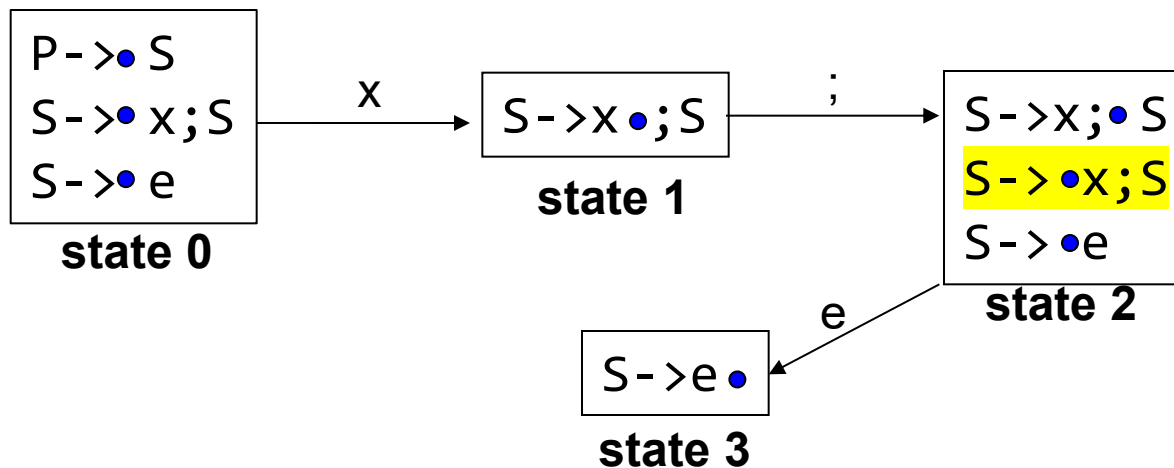
$S \rightarrow x; S$

$S \rightarrow e$

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

Compute successor (of state 2) under symbol x



Grammar

$P \rightarrow S$

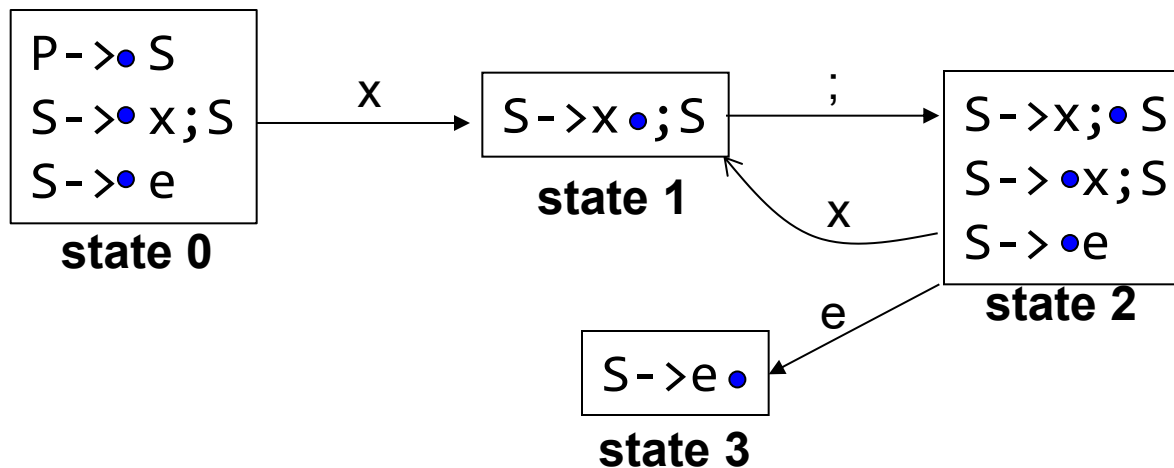
$S \rightarrow x; S$

$S \rightarrow e$

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 x



Grammar

$P \rightarrow S$

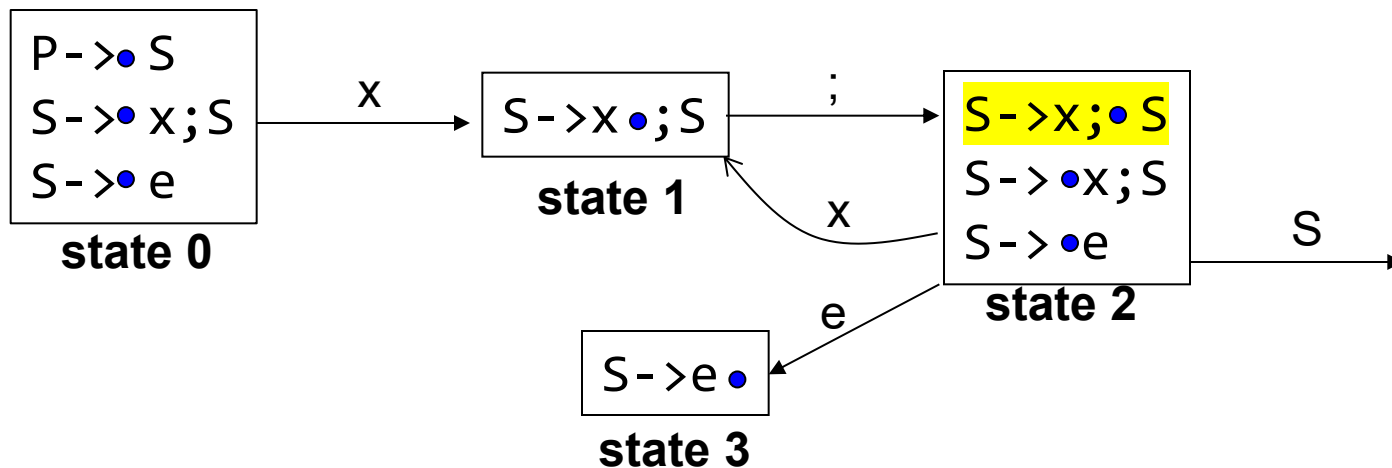
$S \rightarrow x;S$

$S \rightarrow e$

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

$P \rightarrow S$

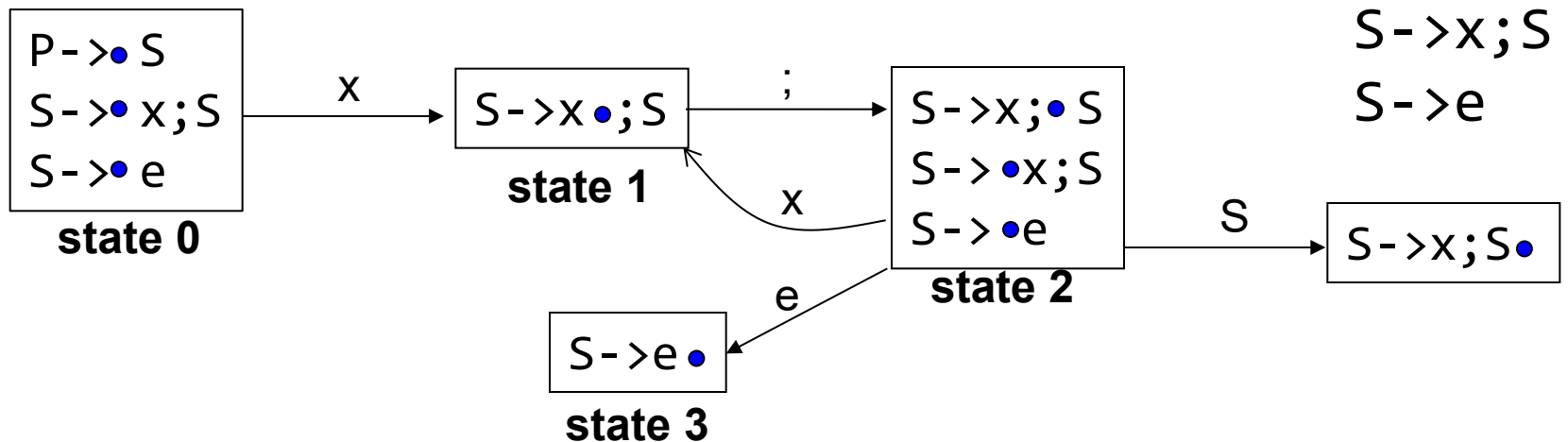
$S \rightarrow x; S$

$S \rightarrow e$

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

Example: CFSM

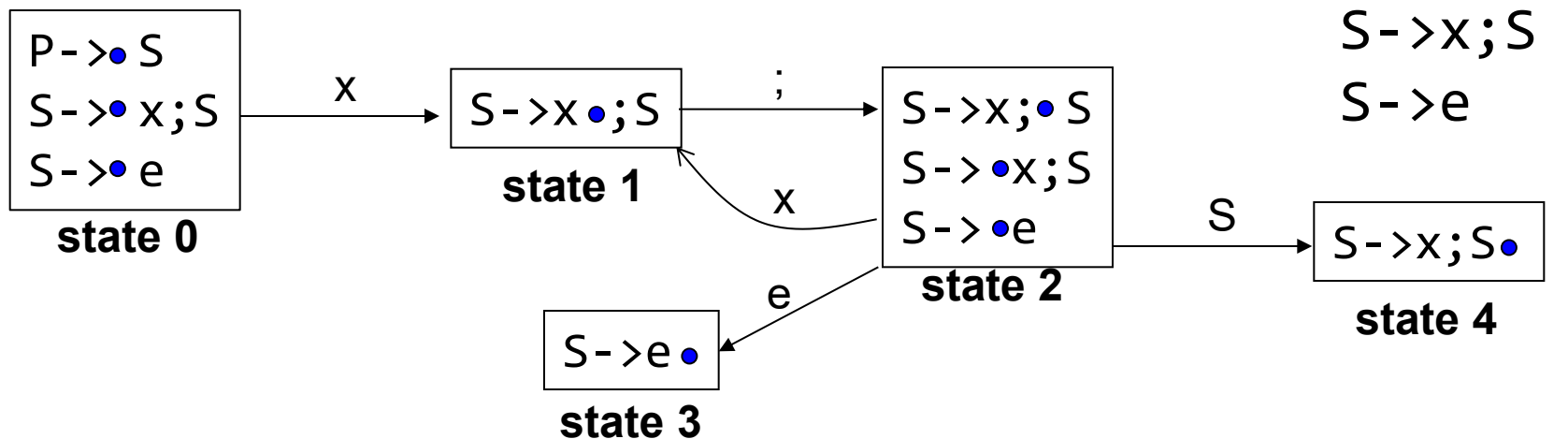
Compute successor (of state 2) under symbol S



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

Example: CFSM

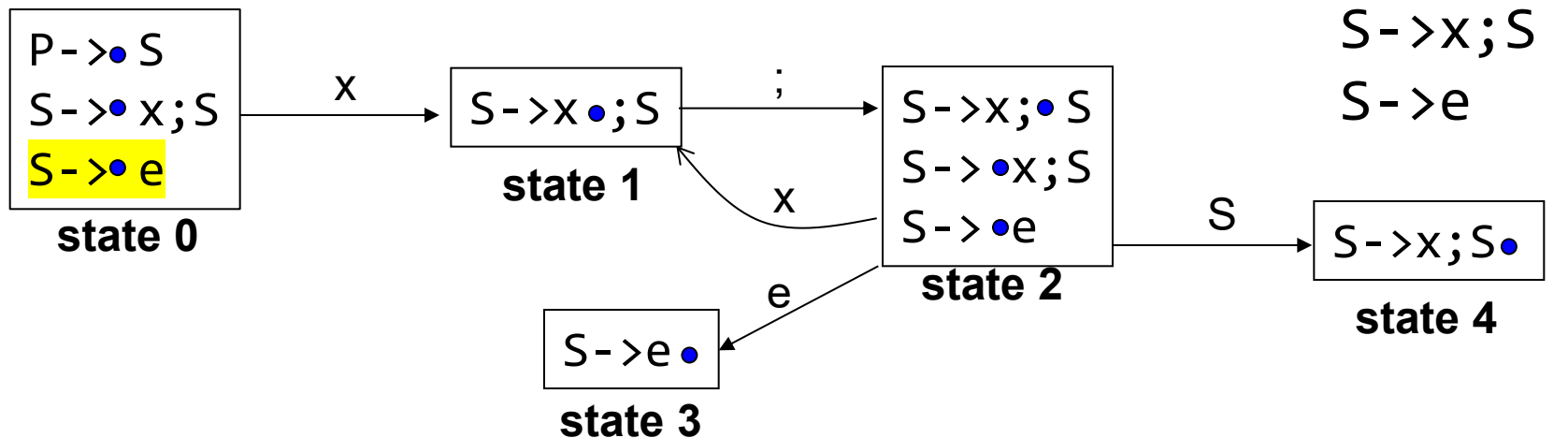
Compute successor (of state 2) under symbol S



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

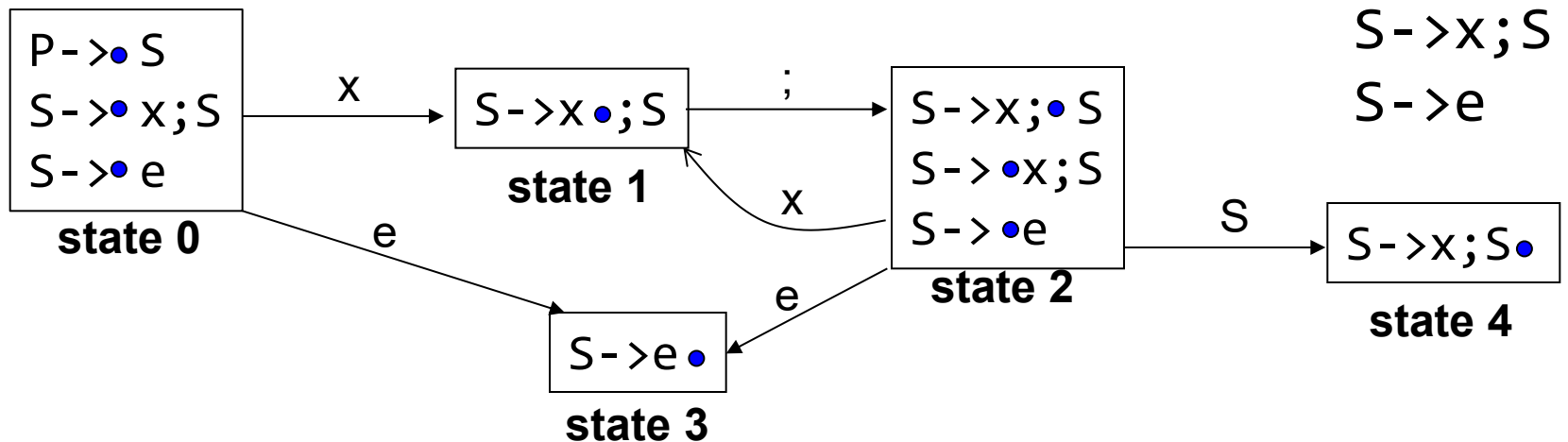
Compute successor (of state 0) under symbol e



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

Example: CFSM

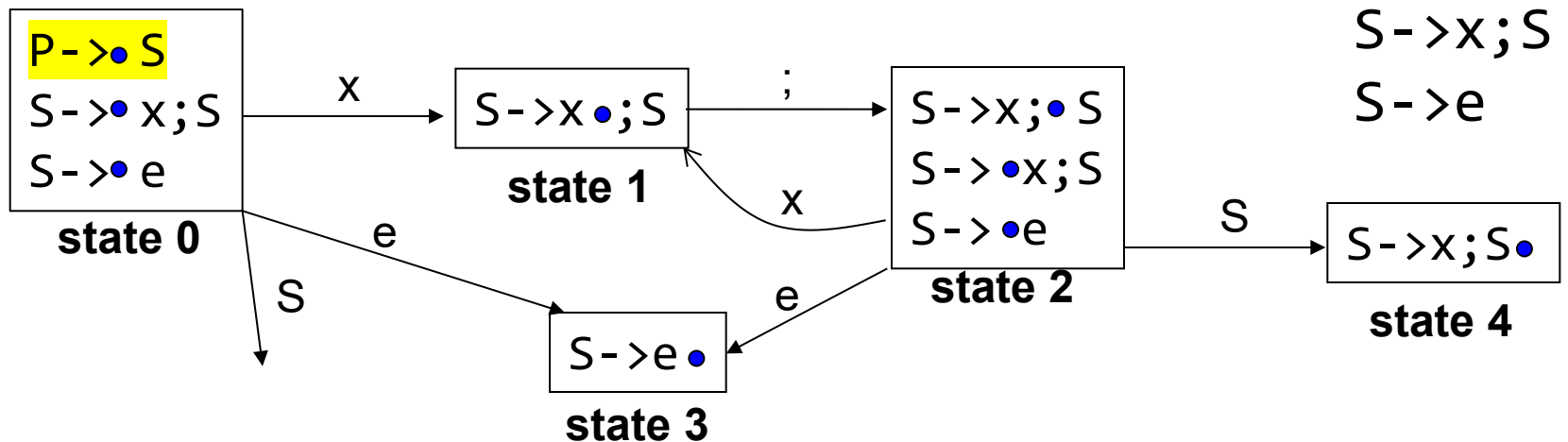
Compute successor (of state 0) under symbol e



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

Example: CFSM

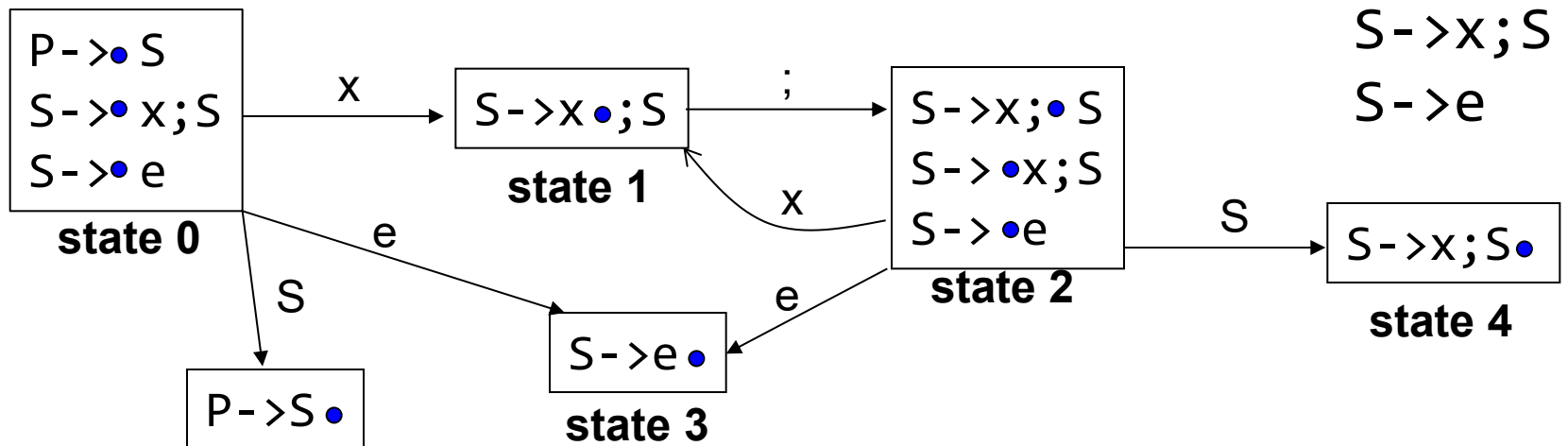
Compute successor (of state 0) under symbol S



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

Example: CFSM

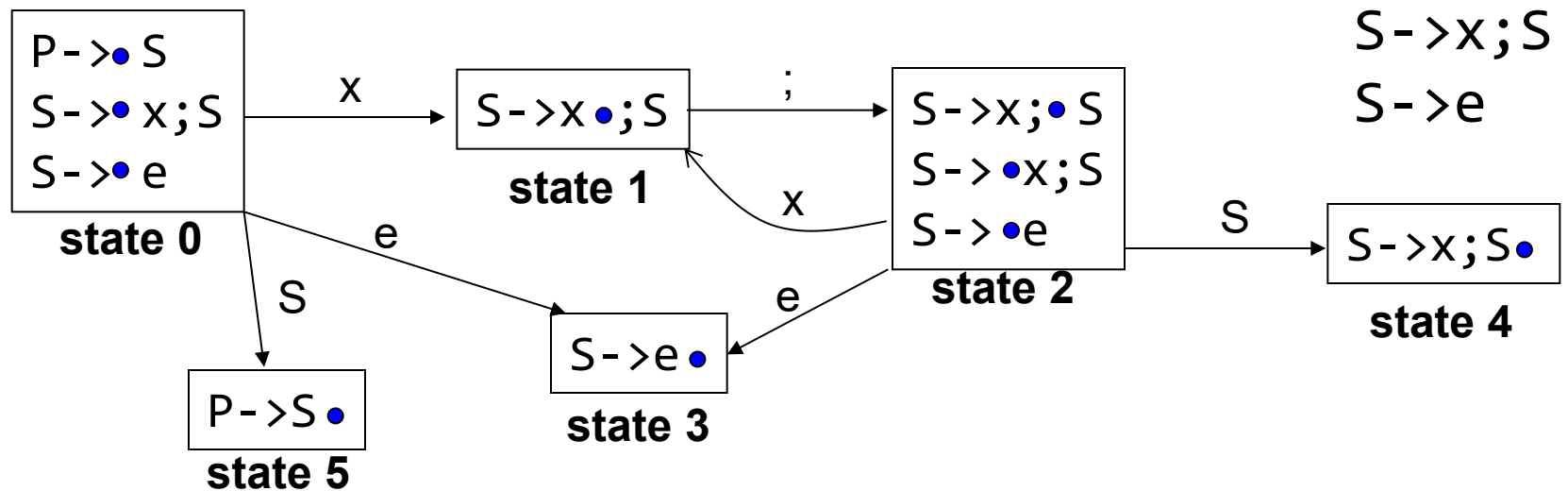
Compute successor (of state 0) under symbol S



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

Example: CFSM

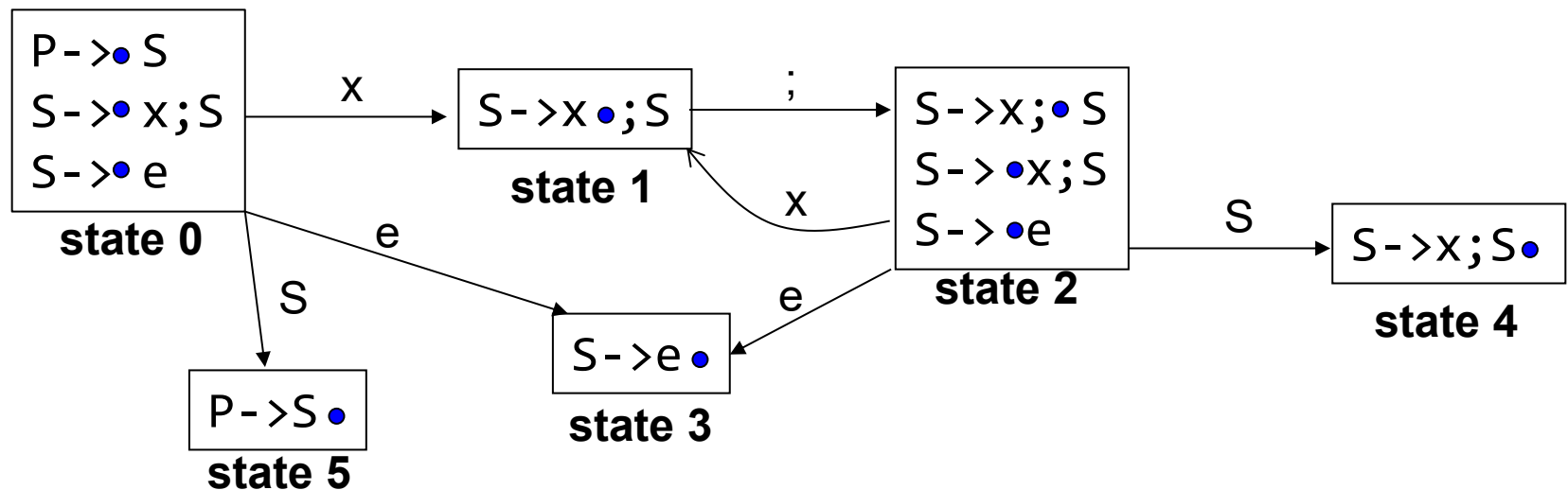
Compute successor (of state 0) under symbol S



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

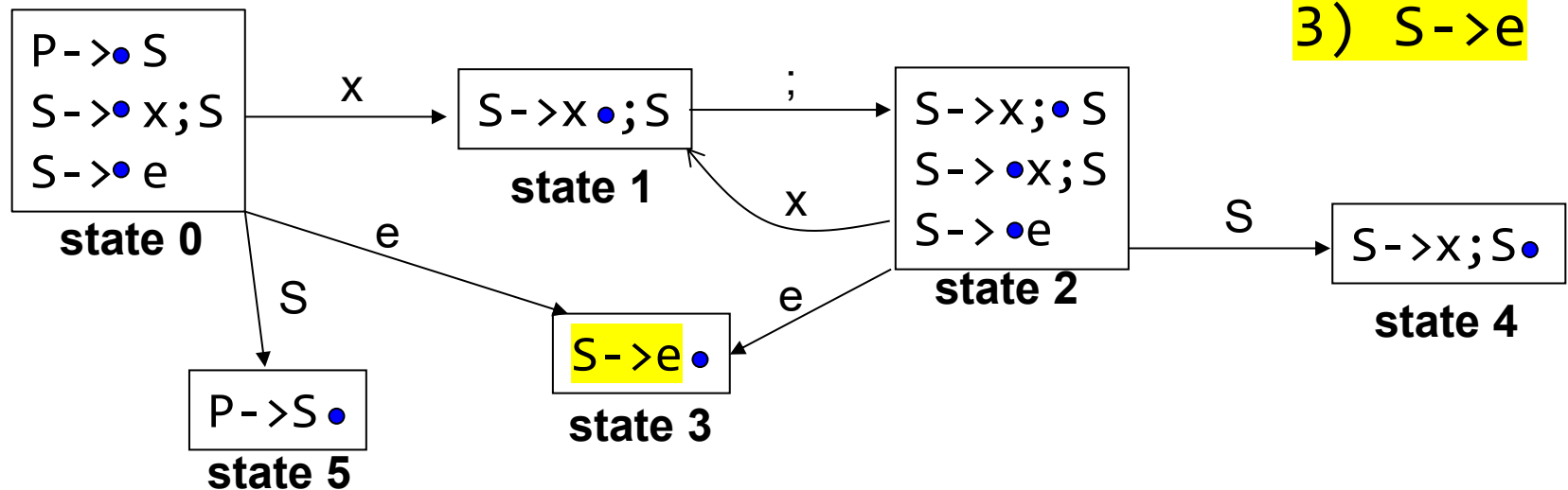
Grammar

1) $P \rightarrow S$

2) $S \rightarrow x;S$

3) $S \rightarrow e$

Reduce 3



Example: CFSM

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

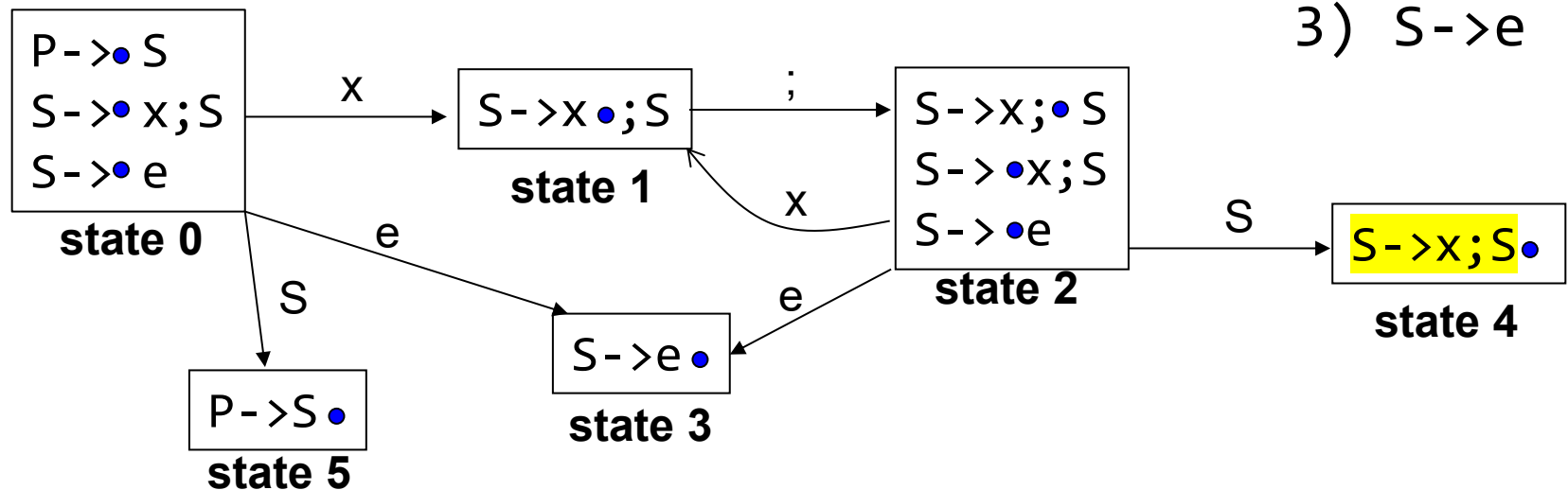
Reduce 2

Grammar

1) $P \rightarrow S$

2) $S \rightarrow x; S$

3) $S \rightarrow e$



Example: CFSM

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

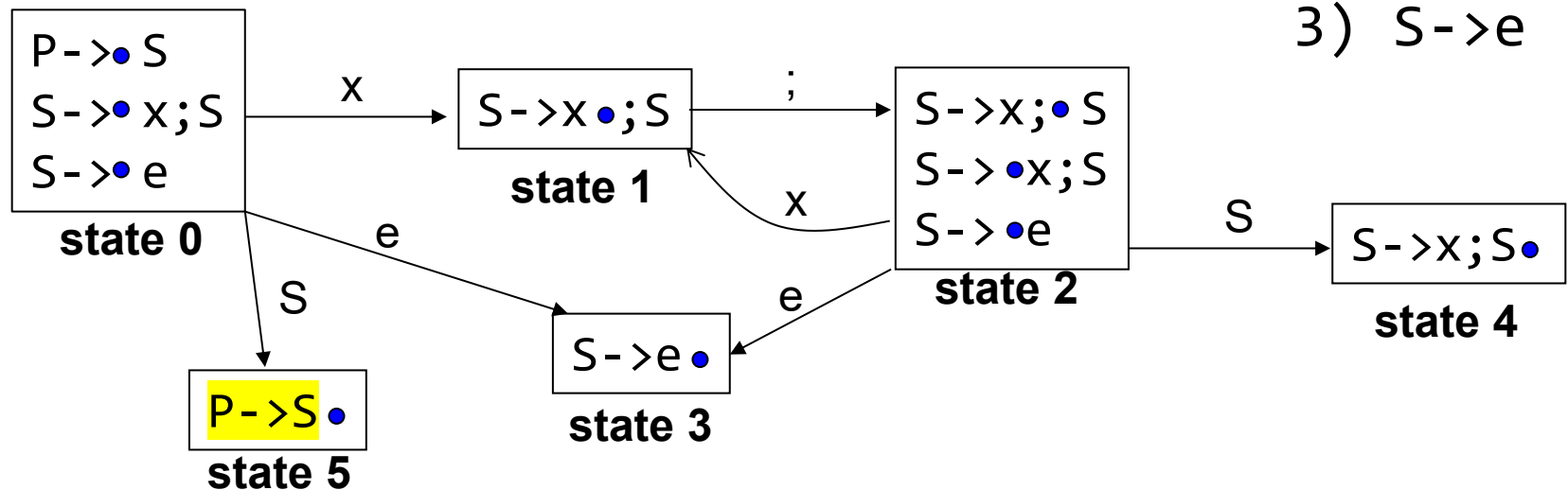
Accept

Grammar

1) $P \rightarrow S$

2) $S \rightarrow x; S$

3) $S \rightarrow e$

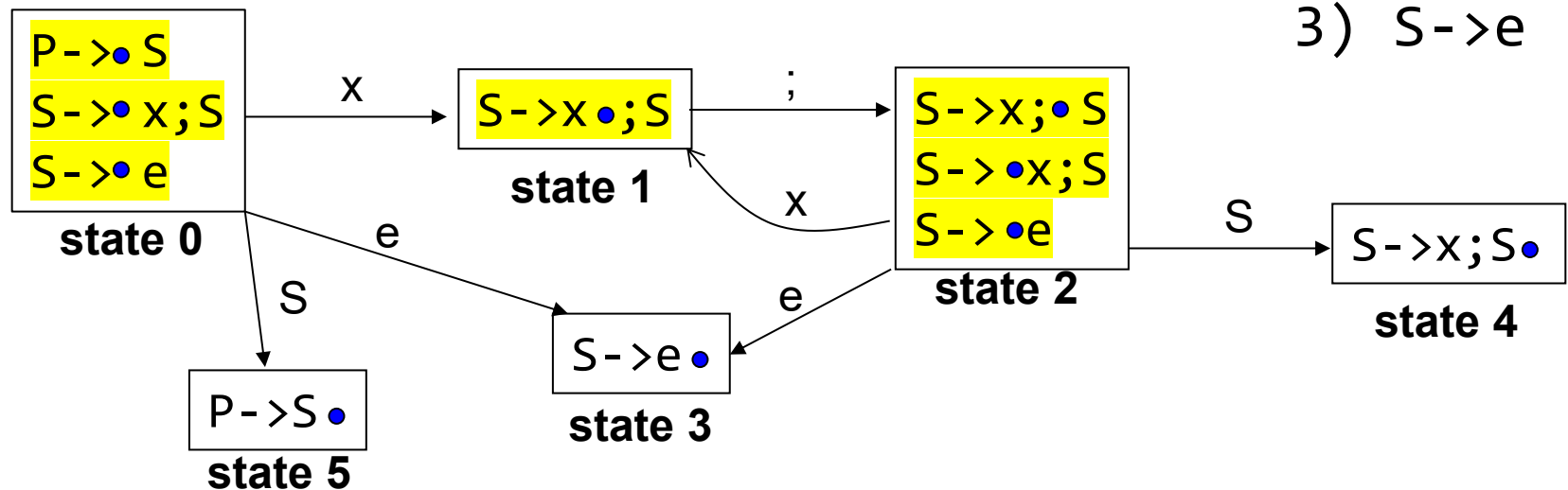


Example: CFSM

- Remaining states become *shift* states

Grammar

- 1) $P \rightarrow S$
- 2) $S \rightarrow x;S$
- 3) $S \rightarrow e$



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

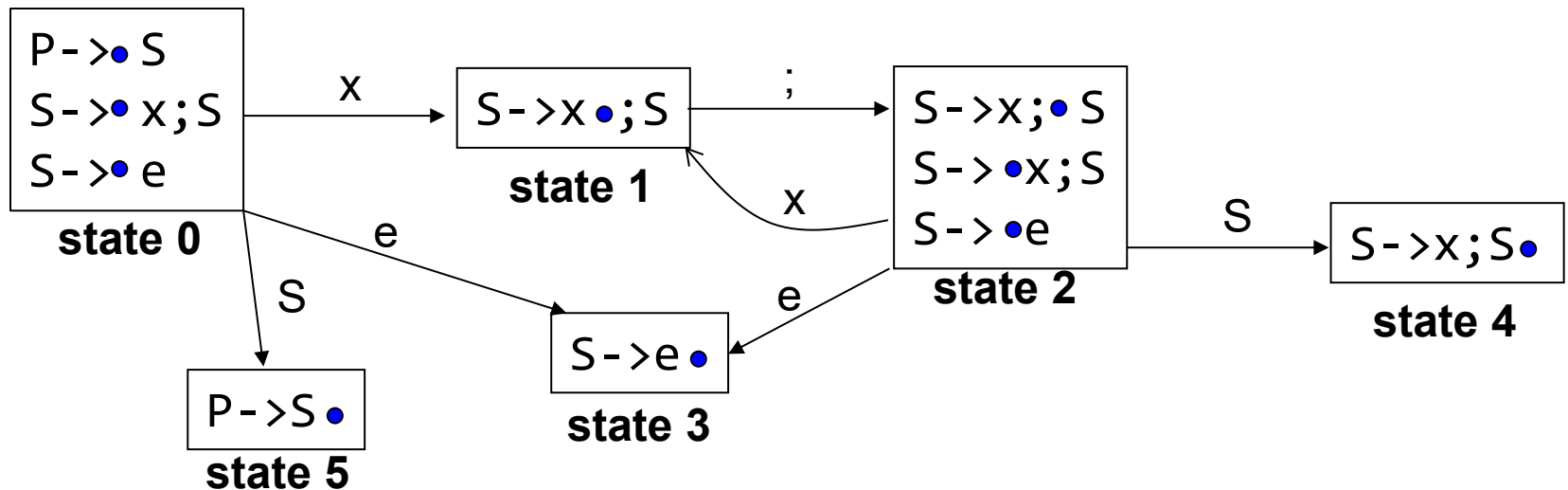
Parser is unable to decide between shifting and reducing

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

Reduce-Reduce conflict

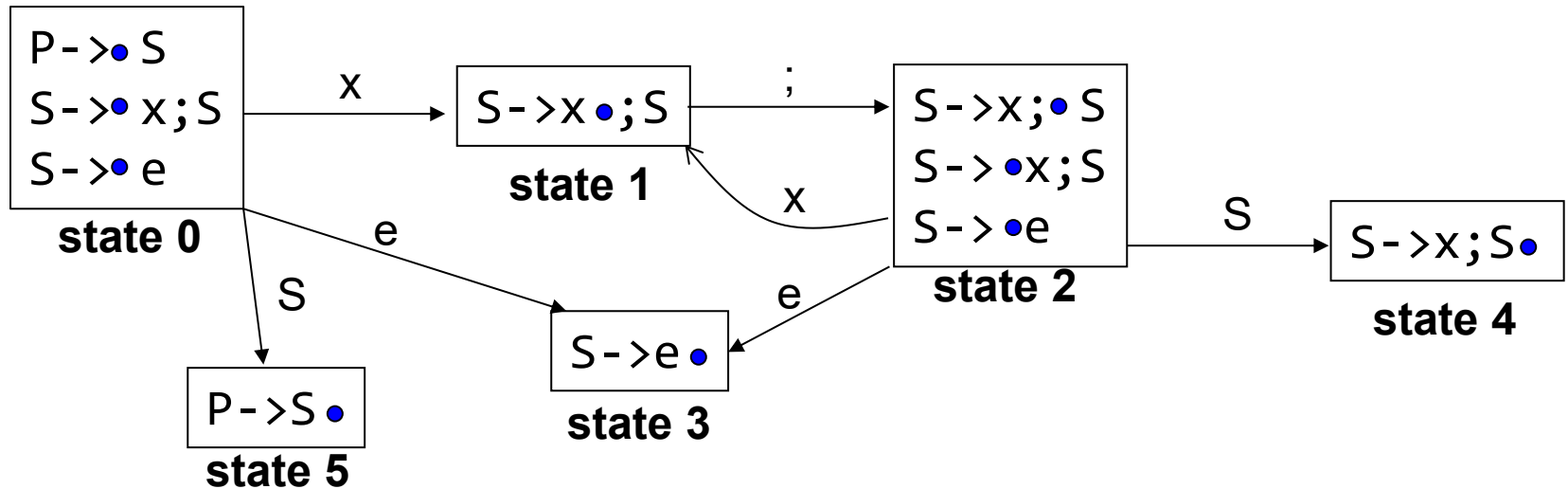
Parser is unable to decide between which productions to choose for reducing

Example: goto table



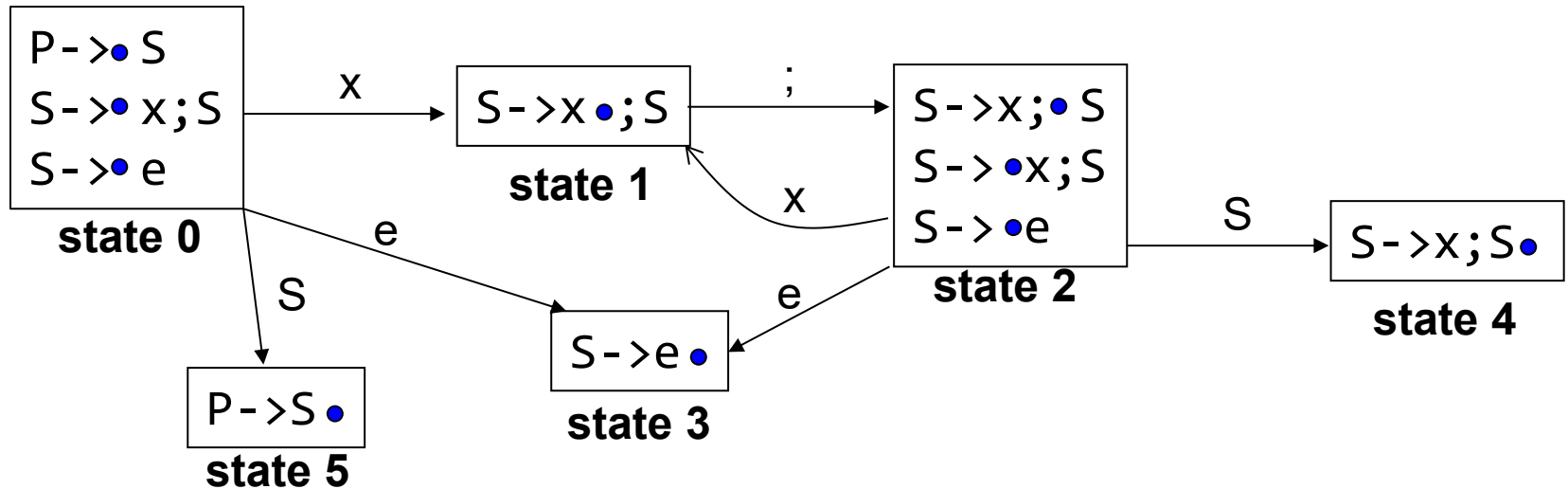
- construct transition table from CFSM.
 - Number of rows = number of states
 - Number of columns = number of symbols

Example: goto table



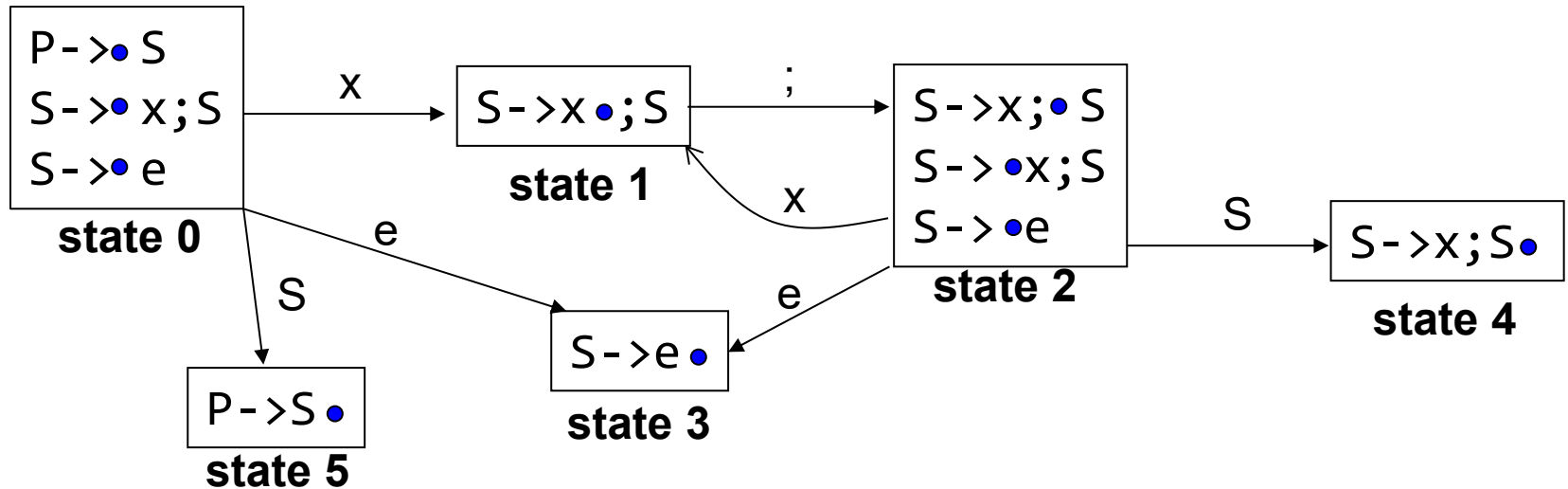
state	x	;	e	P	S
0	1		3		5
1		2			
2	1		3		4
3					
4					
5					

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

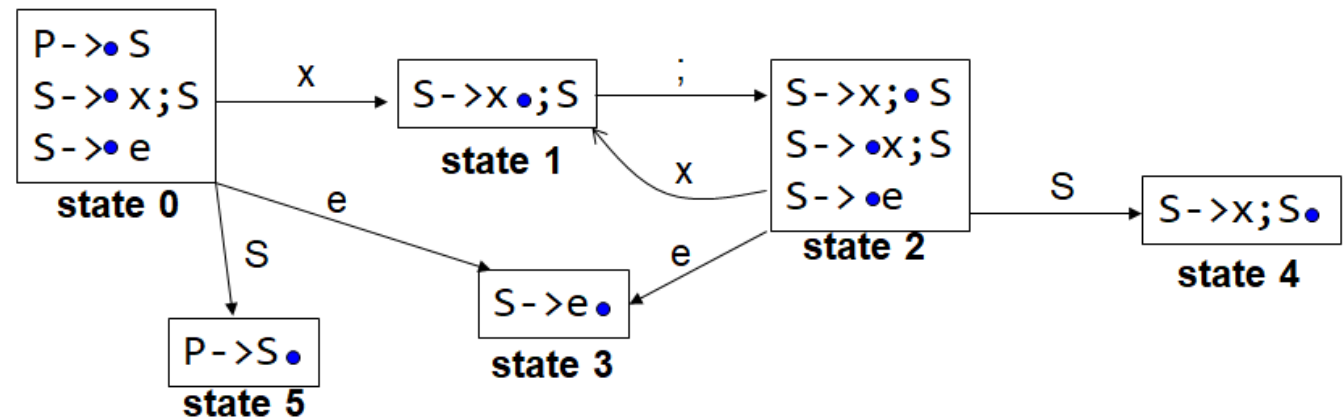
LR(0) Parsing

- 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 α e.g. prefix of **x;x** is seen in the input string

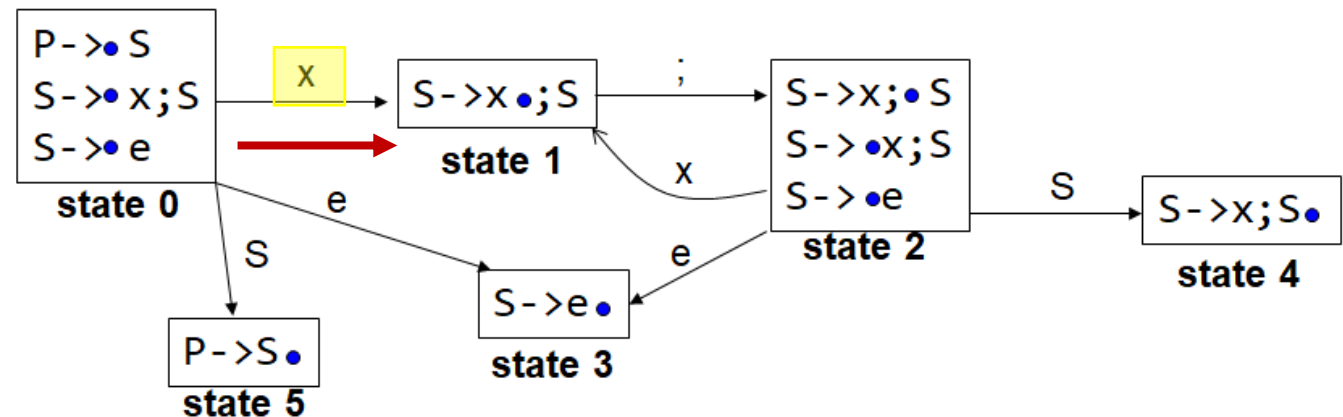
Parse Stack
0
0 1
0 1 2
0 1 2 1



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

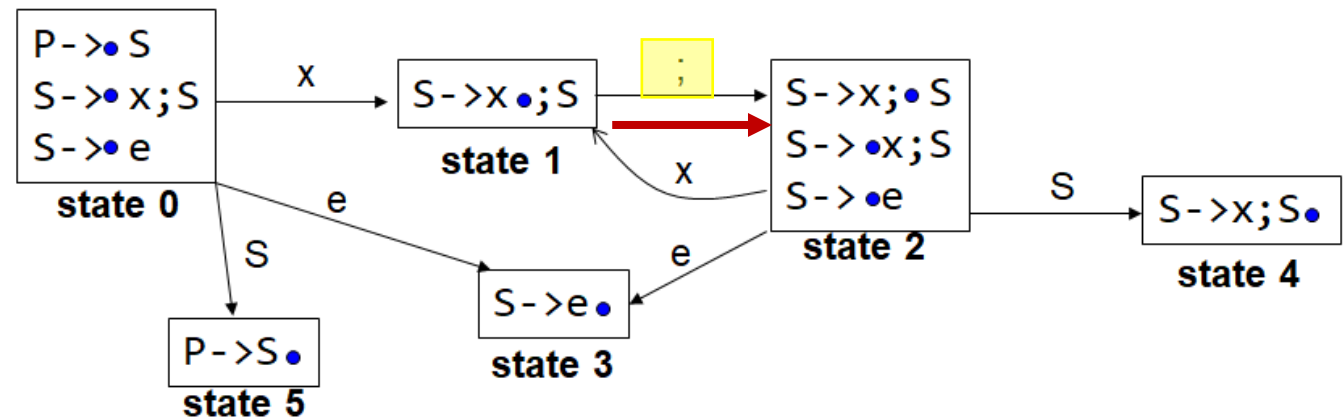


Go from state 0 to state 1 consuming x

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

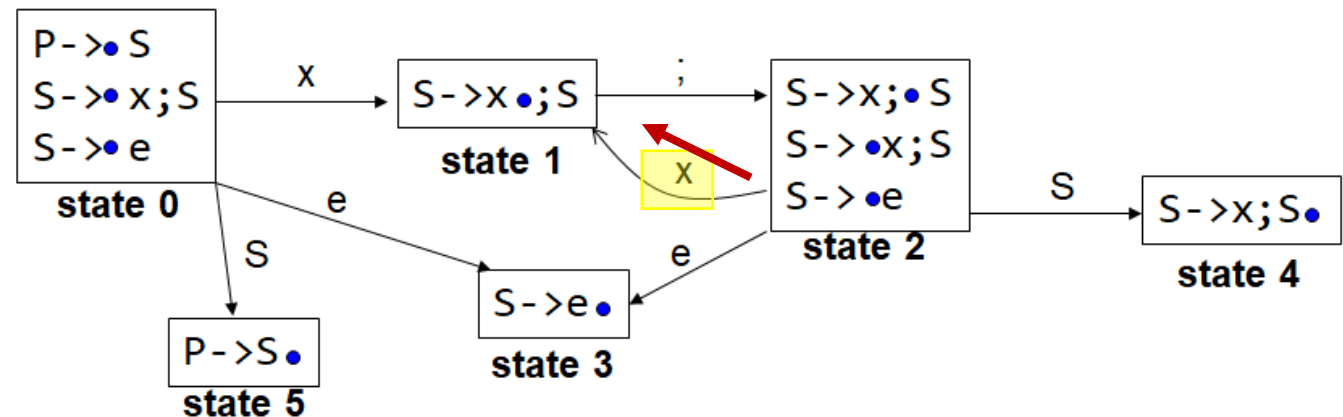


Go from state 1 to state 2 consuming ;

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 $X \rightarrow \beta$ if state s contains $X \rightarrow \beta \bullet$
- Note: reduction is done based solely on the current state.

LR(0) Parsing

- Assume: Parse stack contains α .

=> we are in some state s .

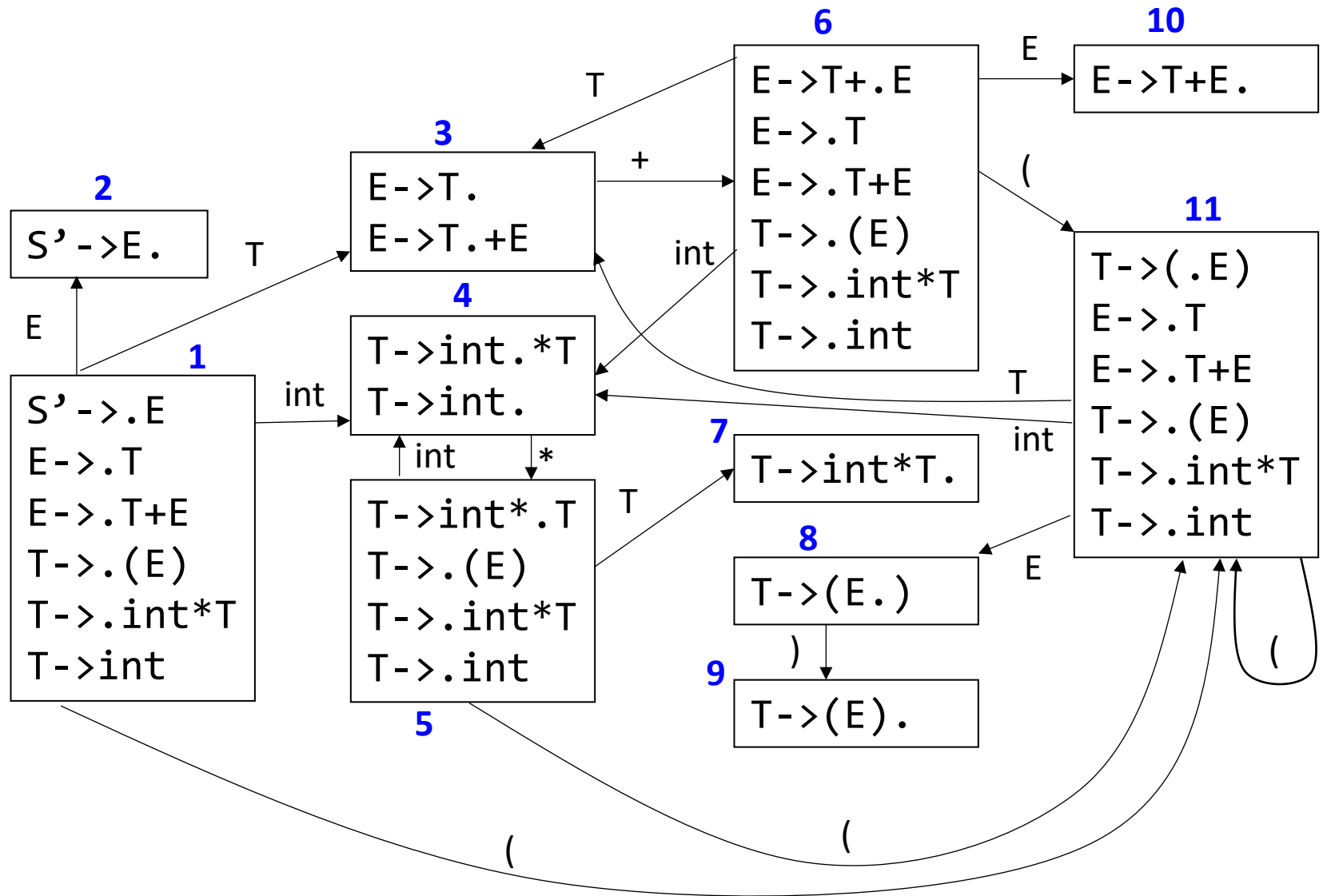
- Assume: Next input is t

We **shift** if s contains $X \rightarrow \beta \bullet t$

== s has a transition labelled t

LR(0) Parsing

- What if s contains $X \rightarrow \beta \bullet t w$ and $X \rightarrow \beta \bullet$?



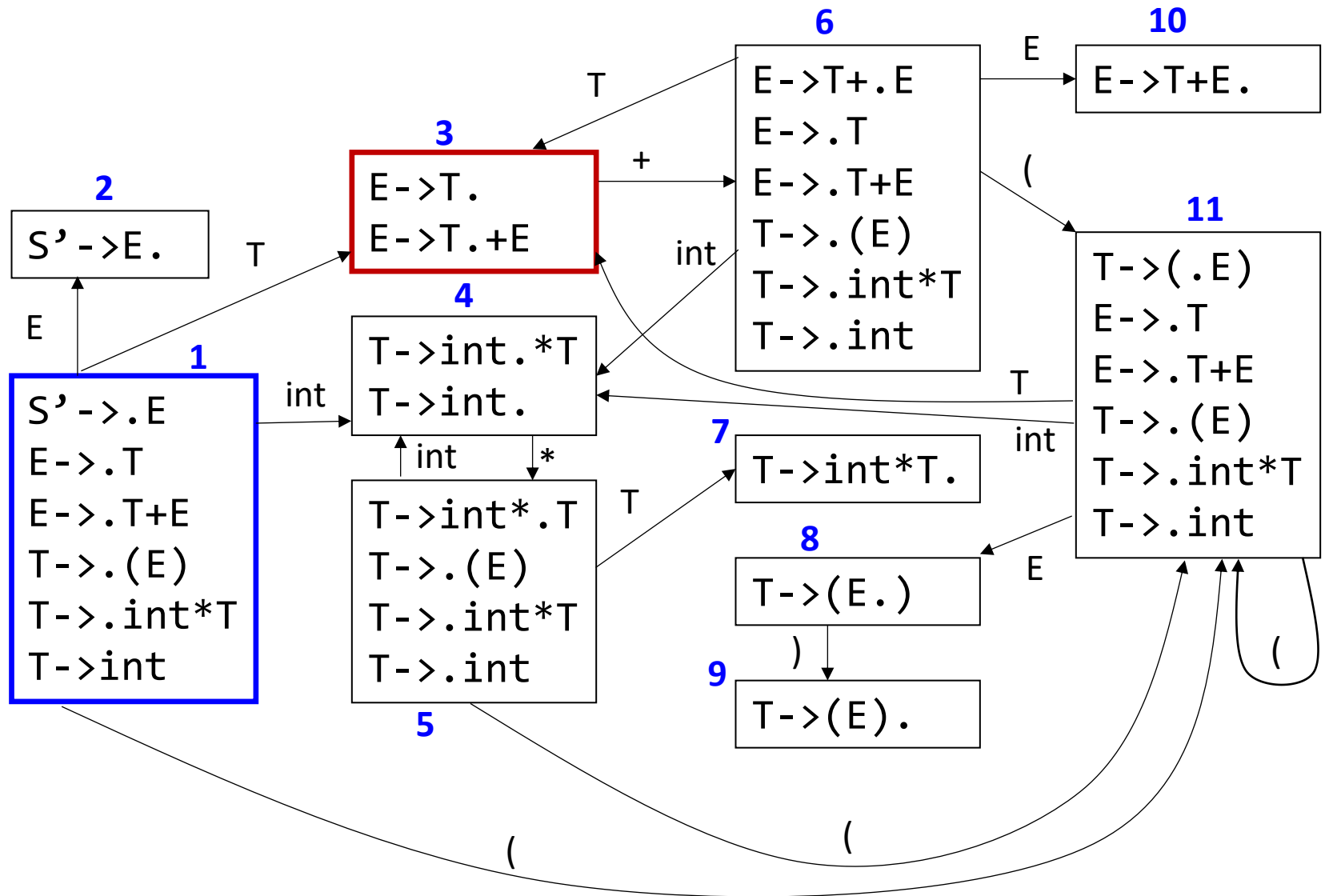
Conflicts or not?
CS620, IIT Dharwad

SLR Parsing

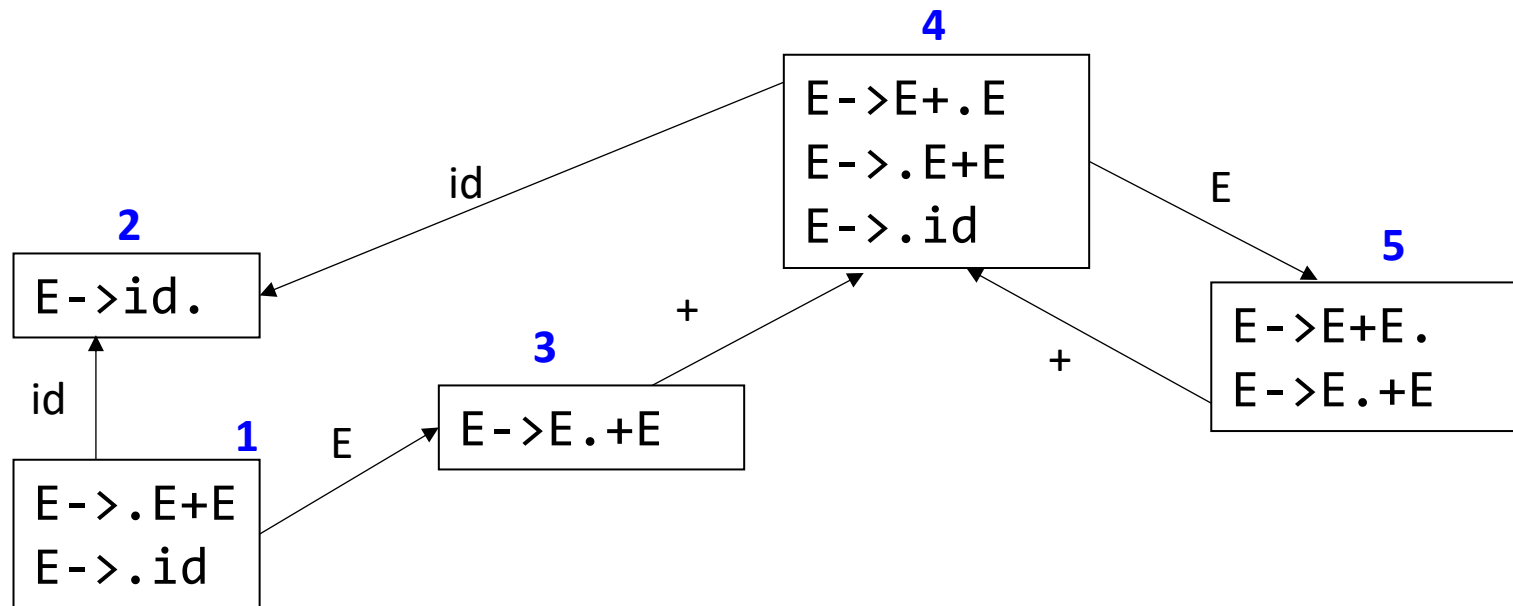
- SLR Parsing improves the shift-reduce conflict states of LR(0):

Reduce $X \rightarrow \beta \bullet$ only if

$t \in \text{Follow}(X)$

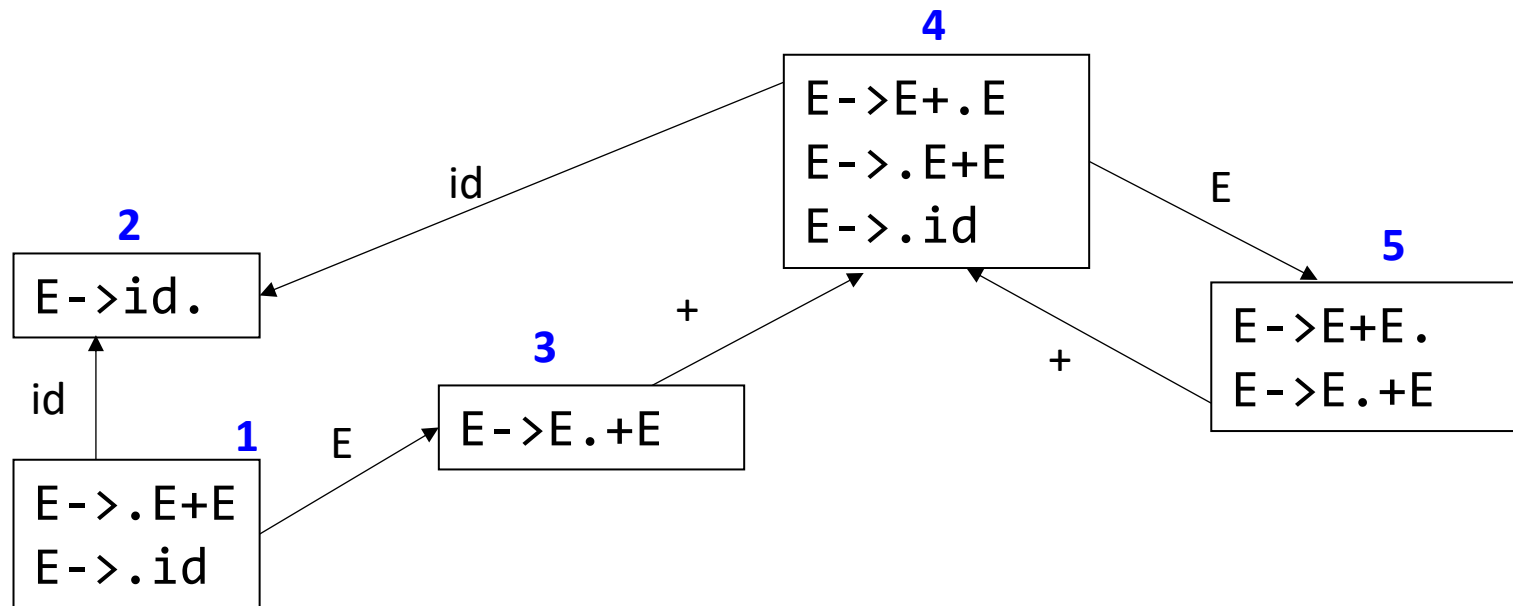


$\text{Follow}(E) = \{ \$,) \} \Rightarrow \text{reduce by } E \rightarrow T \text{ only if } \underline{\text{next input}} \text{ is } \$ \text{ or })$



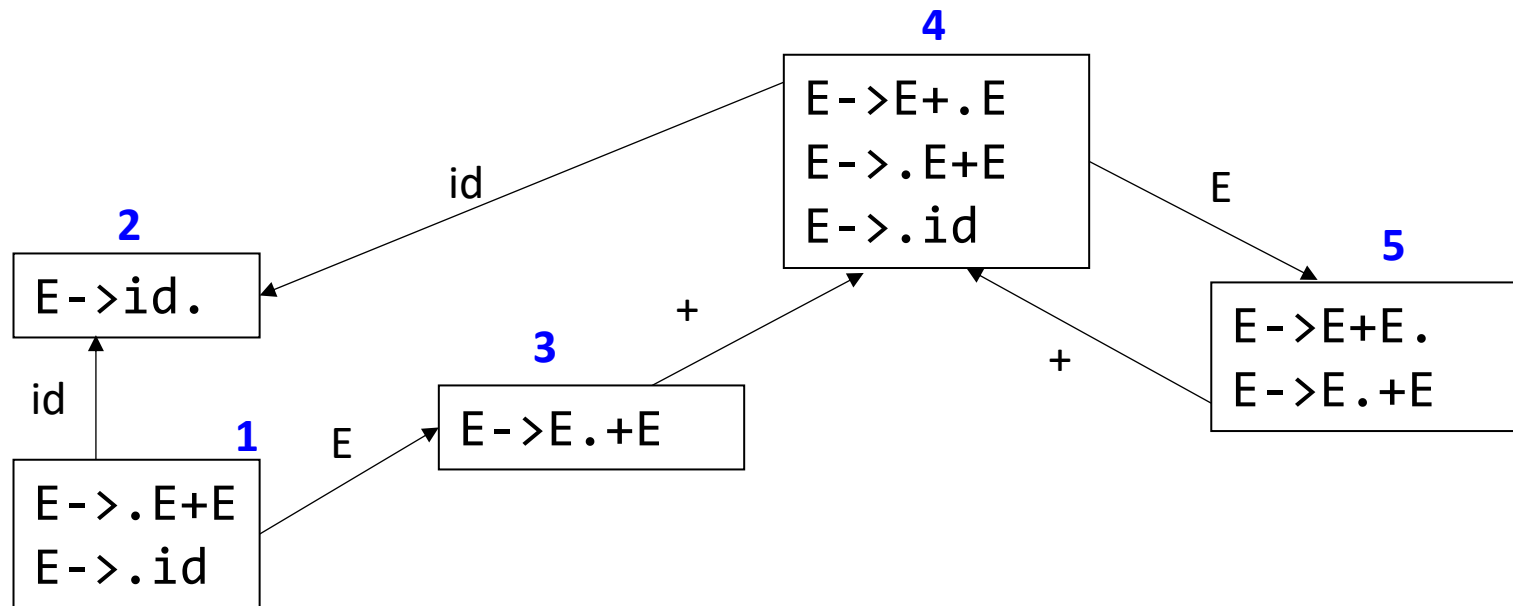
What about the grammar $E \rightarrow E + E \mid id$?

LR(0)?



What about the grammar $E \rightarrow E + E \mid id$?

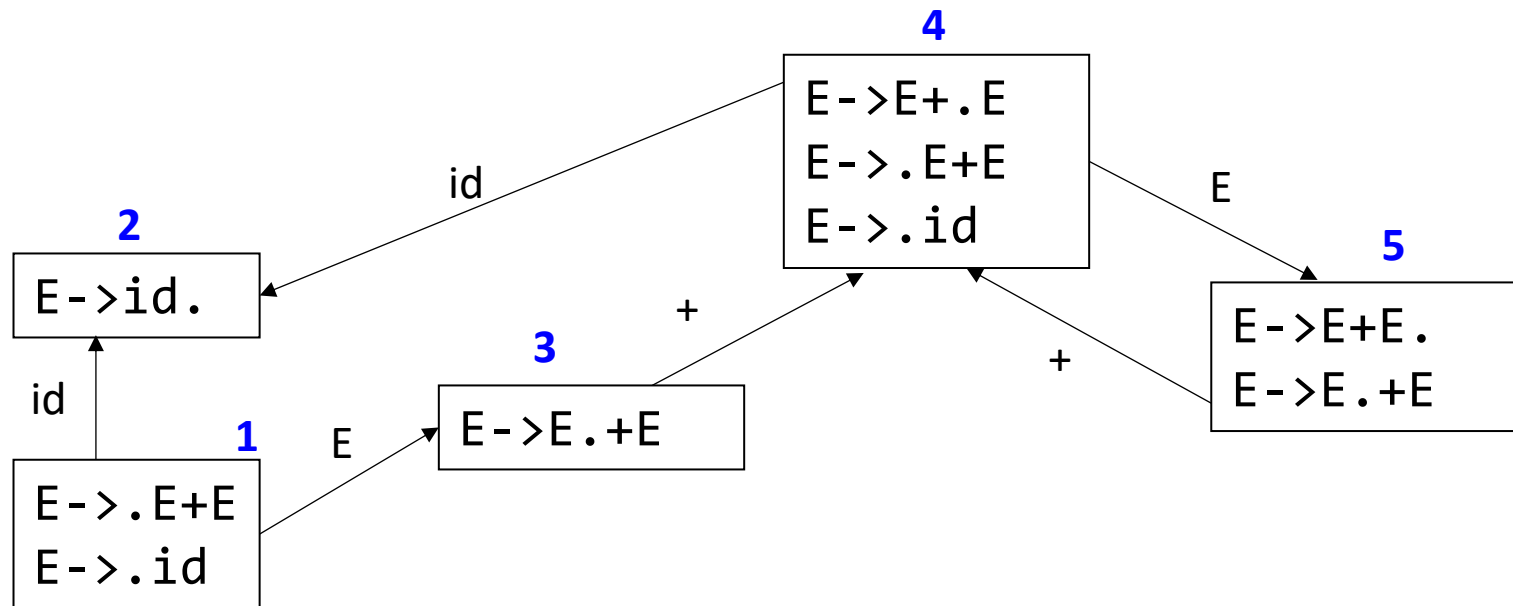
LR(0)? SLR(1)?



What about the grammar $E \rightarrow E + E \mid id$?

LR(0)? SLR(1)?

$\text{Follow}(E) = \{+, \$\} \Rightarrow$ in state 5, reduce by $E \rightarrow E + E$. only if next input is \$ or +

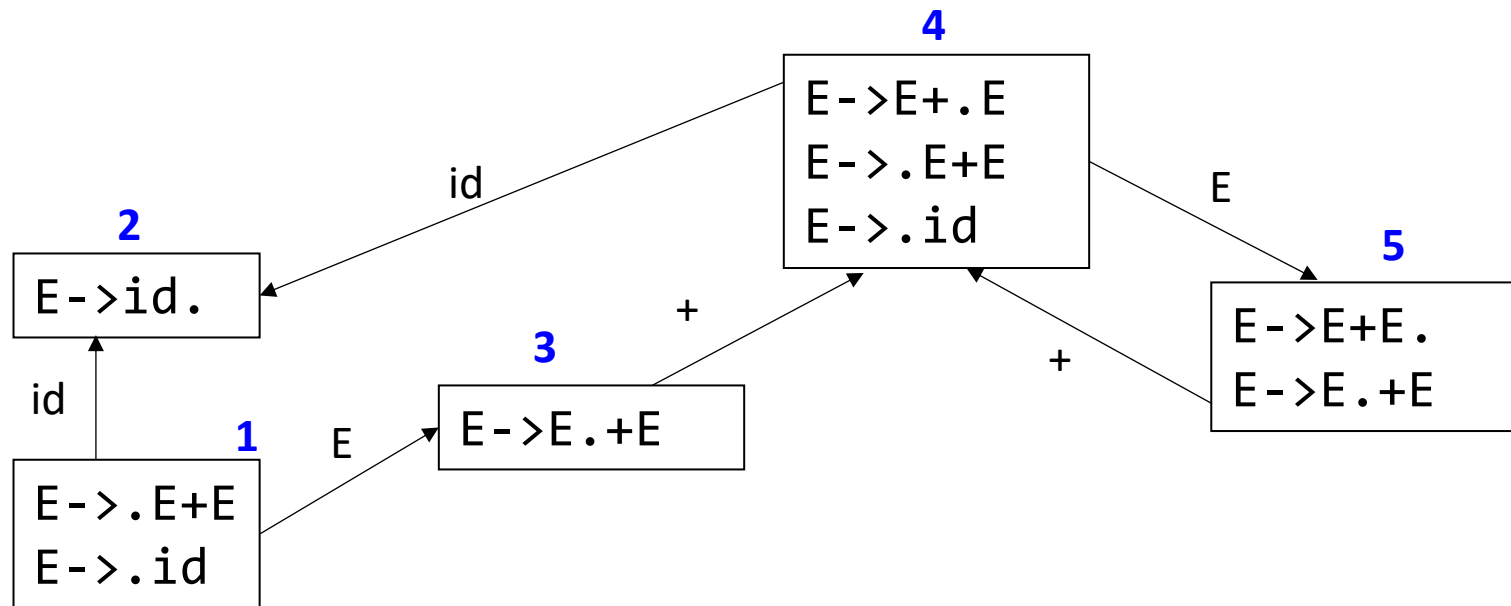


What about the grammar $E \rightarrow E + E \mid id$?

LR(0)? SLR(1)?

$\text{Follow}(E) = \{+, \$\} \Rightarrow$ in state 5, reduce by $E \rightarrow E+E$. only if next input is $\$$ or $+$

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



What about the grammar $E \rightarrow E + E \mid id$?

LR(0)? SLR(1)?

$\text{Follow}(E) = \{+, \$\} \Rightarrow$ in state 5, reduce by $E \rightarrow E+E$. only if next input is $\$$ or $+$

But state 5 has $E \rightarrow E.+E$ (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: $S \rightarrow F$

2: $S \rightarrow (S + F)$

3: $F \rightarrow a$

input:

(a+)

Accepted or Not
accepted?

Parse Table (Top-Down)

Discussion: LR and LL Parsers

Grammar:

1: $S \rightarrow F$

2: $S \rightarrow (S + F)$

3: $F \rightarrow a$

input:

(a+)

Accepted or Not
accepted?

Goto and Action Table?

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