CS406: Compilers

Spring 2021

Week 5: Parsers

First Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

`first (S) = { ? }`

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 -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

first (S) = { x, y, c }
```

First Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

first (S) = { x, y, c }
first (A) = {  ?  }
```

First Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

first (S) = { x, y, c }
first (A) = { x, y, c }
```

First Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

first (S) = { x, y, c }
first (A) = { x, y, c }
first (B) = {  ?  }
```

First Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

first (S) = { x, y, c }
first (A) = { x, y, c }
first (B) = { b, λ }
```

Follow Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

`follow (S) = { ? }`

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 -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

follow (S) = {  }
```

Follow Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

follow (S) = {   }
follow (A) = { ? }
```

Follow Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

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

Follow Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

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

What happens when you consider: `A -> xaA or A -> yaA ?`

Follow Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

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

What happens when you consider: `A -> xaA or A -> yaA ?`

- You will get string of the form `A=>+ (xa)+A`
- But we need strings of the form: `..Aa..` or `..Ab.` or `..Ac..`

Follow Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

follow (S) = {   }
follow (A) = { b, c }
follow (B) = { ? }
```

Follow Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

```
follow (S) = {   }
follow (A) = { b, c }
follow (B) = { c }
```

Predict Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

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

Predict (1) = { ? } = First(ABc$) if λ ∉ First(ABc$)

Predict Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

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

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

Predict Set - Example

1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

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

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

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

Predict Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

	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 -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

	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 $\lambda \notin$ First(yaA)

Predict Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

	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 -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

	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 $\lambda \notin$ First(c)

Predict Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

	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 -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

	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 λ ∉ First(b)

Predict Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

	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 -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

	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) = { ? }

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

= First(λ) ?

Predict Set - Example

1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

	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) = { ? }

$$\mathrm{Predict}(P) = \begin{cases} \mathrm{First}(X_1 \ldots X_m) & \text{if } \lambda \notin \mathrm{First}(X_1 \ldots X_m) \\ \boxed{(\mathrm{First}(X_1 \ldots X_m) - \lambda) \cup \mathrm{Follow}(A) \quad \text{otherwise}} \end{cases}$$

= ~~First(λ)~~ ? Follow(B)

Predict Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

	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) = { c }
```

Computing Parse-Table

1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

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

first (S) = {x, y, c}
first (A) = {x, y, c}
first(B) = {b, λ}

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

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

Parsing using parse table and a stack-based model (non-recursive)

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 $\beta \stackrel{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in `Follow(A)`. If $\alpha \stackrel{*}{\Rightarrow} \epsilon$, 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

$$\Downarrow$$

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:

  $E \rightarrow E + T$

- 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

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

$$\Downarrow$$

$A \rightarrow NT$

$N \rightarrow \beta$

$T \rightarrow \alpha T$

$T \rightarrow \lambda$

Eliminating Left Recursion

E -> E + 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?

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!

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

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": $[^i\,]^j$ $(i \geq j)$

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

[[ ] can be parsed: SSλC or SSCλ
(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?

$$
\begin{aligned}
S &\rightarrow [\ S \\
S &\rightarrow S1 \\
S1 &\rightarrow [\ S1\ ] \\
S1 &\rightarrow \lambda
\end{aligned}
$$

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"

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

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

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

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

- <u>Reduce</u> 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

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

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

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

```
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
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

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

Example

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

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

<u>Input string</u>

x;x;e

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

<u>Input string</u>

x;x;e

State	Symbol: 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)

Example

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

<u>Input string</u>

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	?

Example

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

<u>Input string</u>

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					
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	?

Example

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

<u>Input string</u>

x;x;e

State	Symbol					Action
	x	;	e	P	S	
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	Symbol					Action
	x	;	e	P	S	
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

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	?

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)

Example

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

<u>Input string</u>

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		?

Example

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

<u>Input string</u>

x;x;e

State	Symbol 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

Example

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

<u>Input string</u>

`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
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

<u>Input string</u>

x;x;e

State	Symbol: 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
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

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		

- *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
```

<u>Input string</u>

x;x;e

State	Symbol 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		?

- *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

<u>Input string</u>

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

Example

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

<u>Input string</u>

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
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

<u>Input string</u>

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

<u>Input string</u>

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 (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

State	Symbol					Action
	x	;	e	P	S	
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

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
9	0		

Example

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

<u>Input string</u>

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 (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
```

<u>Input string</u>

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

<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 (shift(4))
8	0 1 2 4		Reduce 2 (shift(5))
9	0 5		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

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 ; 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|

S

x ; x ; e

Example

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

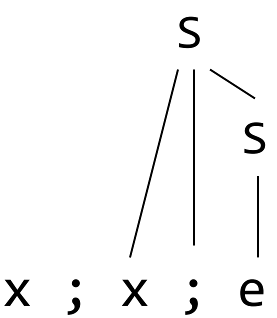
Input string

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
```

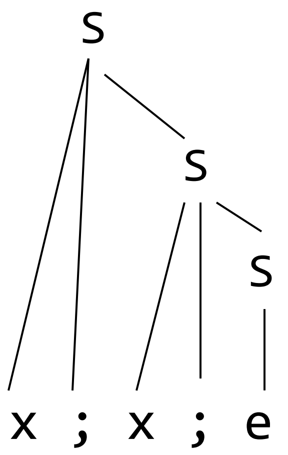
Input string

|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

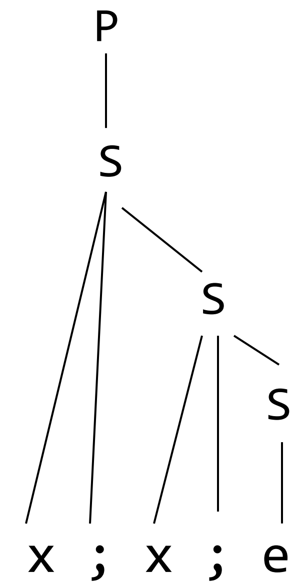
Input string

|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 |



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 $\bullet$ 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 $N \rightarrow \bullet \varepsilon$

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

- E.g. closure {S ->• E$}

Non-terminal

```
S ->•E$
E ->•E+T
```

Grammar

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

Concept: closure

- E.g. closure {S ->• E$}

⇩

Non-terminal

```
S ->•E$
E ->•E+T
E ->•T
```

Grammar

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

Concept: closure

- E.g. closure {`S ->• E$`}

⇩

```
S ->•E$
E ->•E+T
E ->•T
```

New Non-terminal

Grammar

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

Concept: closure

- E.g. closure {`S ->• E$`}

⇩

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

New Non-terminal

Grammar

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

Concept: closure

Grammar

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

- E.g. closure {`S ->• E$`}

⇩

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

New Non-terminal

Concept: closure

- E.g. closure {S ->• E$}

$$\Downarrow$$

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

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) } $\xrightarrow{E}$ { S -> E•$, E -> E•+T }

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

Concept: CFSM

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

Example: CFSM

Grammar

P->S

S->x;S

S->e

Start with a configuration for the first production

P->• S

Example: CFSM

Compute closure

P->• S ← Non-terminal

Grammar

P->S

S->x;S

S->e

Example: CFSM

Add item

```
P->•S
S->•x;S
```

Grammar

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

Example: CFSM

Add item

```
P->•S
S->•x;S
S->•e
```

Grammar

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

Example: CFSM

Grammar

P->S

S->x;S

S->e

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

P->• S
S->• x;S
S->• e

state 0

Example: CFSM

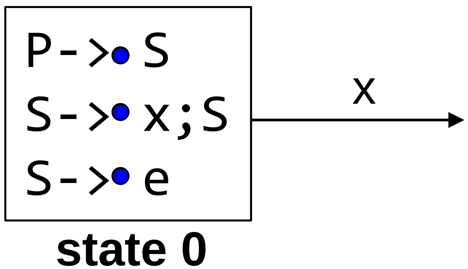
Grammar

P->S

S->x;S

S->e

Compute successor (of state 0) under symbol x



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

Example: CFSM

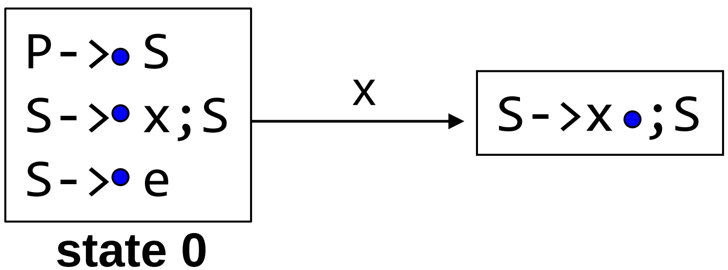
Grammar

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.

Example: CFSM

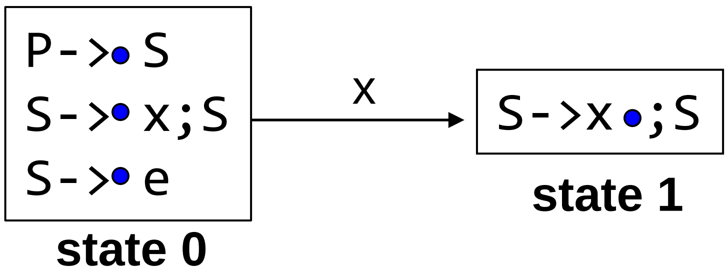
Grammar

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

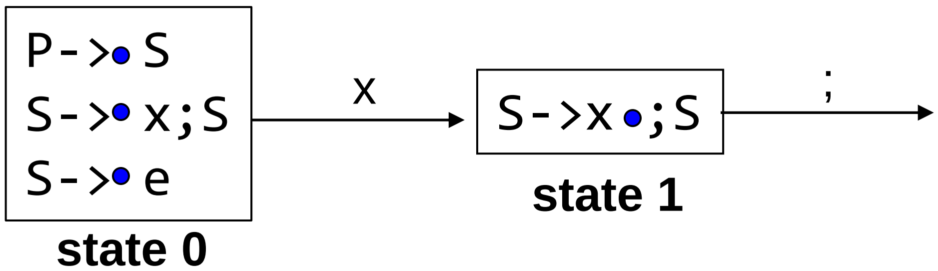
Grammar

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

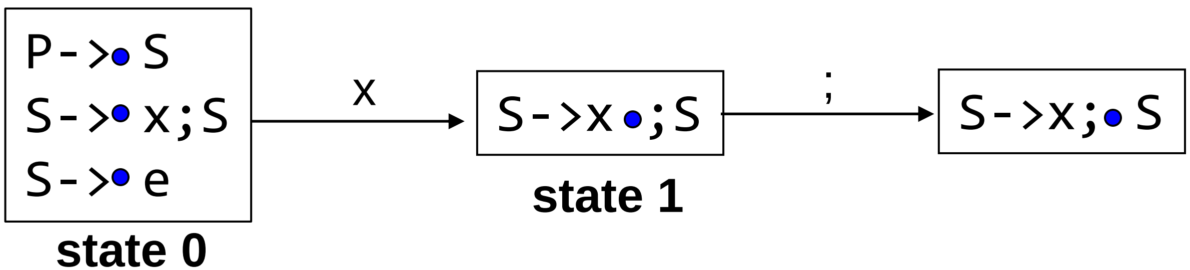
Grammar

P->S

S->x;S

S->e

Compute successor (of state 1) under symbol ;



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

Example: CFSM

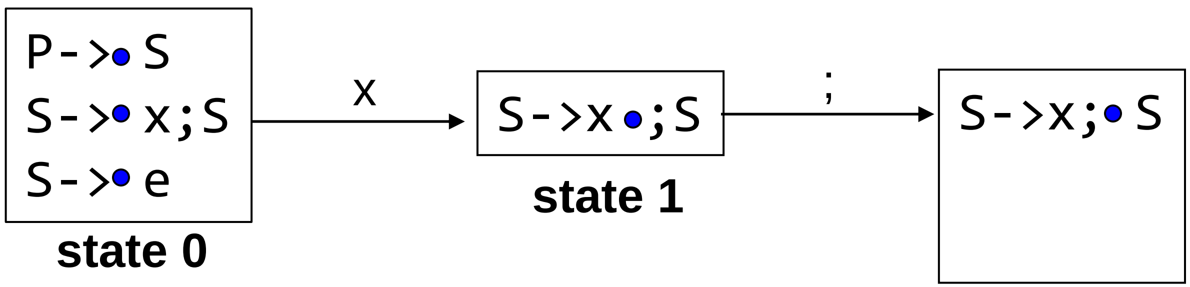
Grammar

P->S

S->x;S

S->e

Compute successor (of state 1) under symbol ;



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

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

Example: CFSM

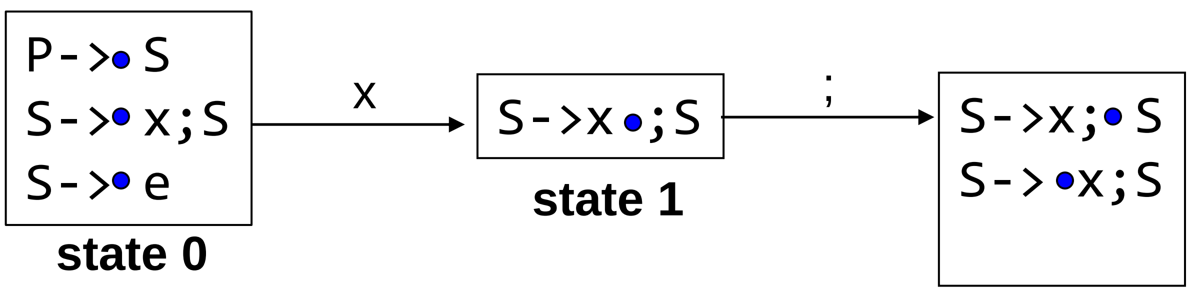
Grammar

P->S

S->x;S

S->e

Compute successor (of state 1) under symbol ;



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

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

Example: CFSM

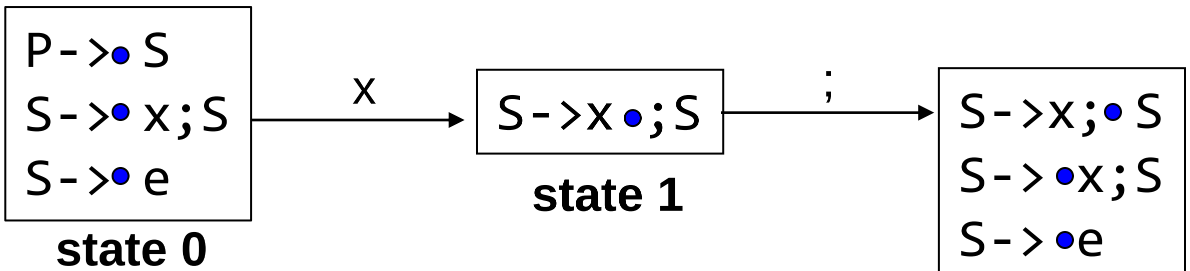
Grammar

P->S

S->x;S

S->e

Compute successor (of state 1) under symbol ;



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

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

Example: CFSM

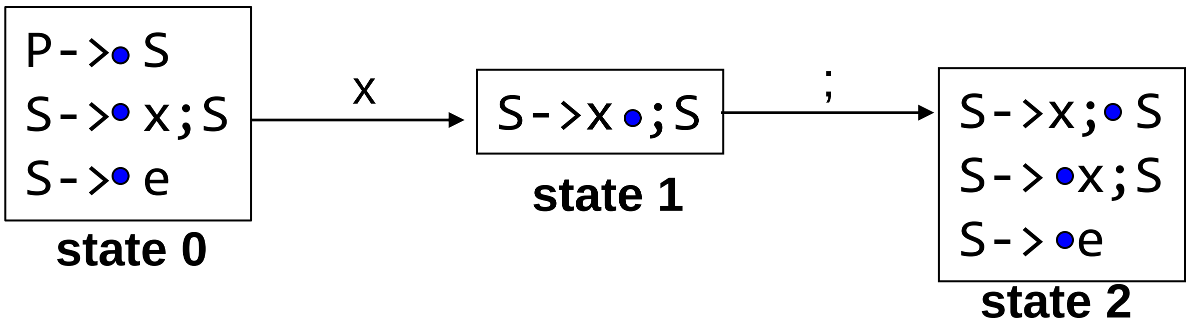
Grammar

P->S

S->x;S

S->e

Compute successor (of state 1) under symbol ;



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

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

Example: CFSM

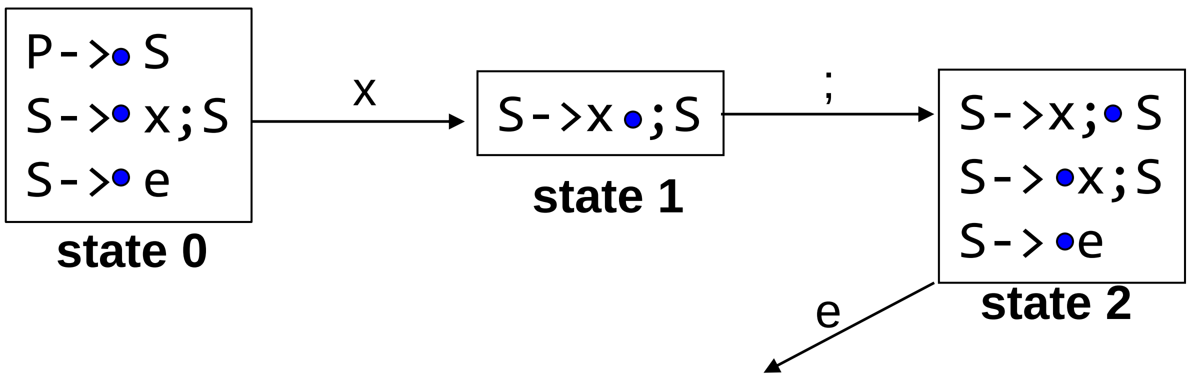
Grammar

P->S

S->x;S

S->e

Compute successor (of state 2) under symbol e



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

Example: CFSM

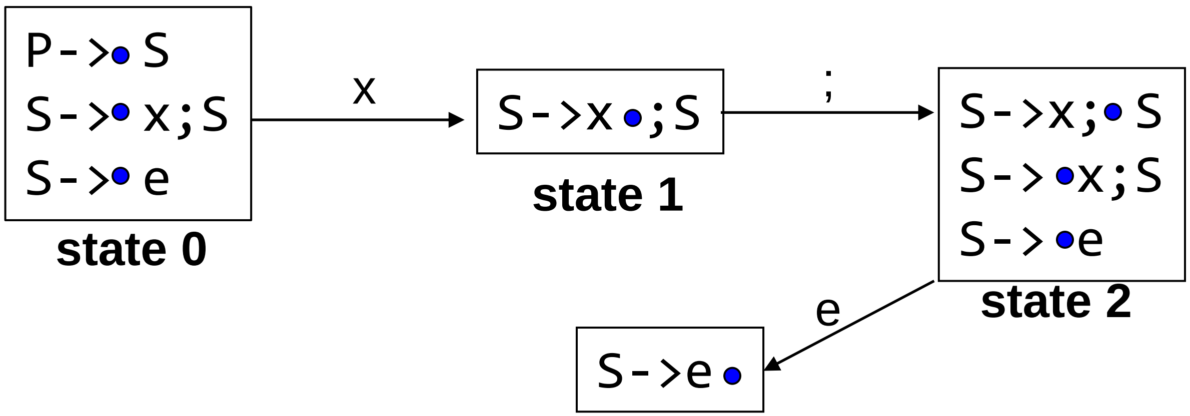
Grammar

P->S

S->x;S

S->e

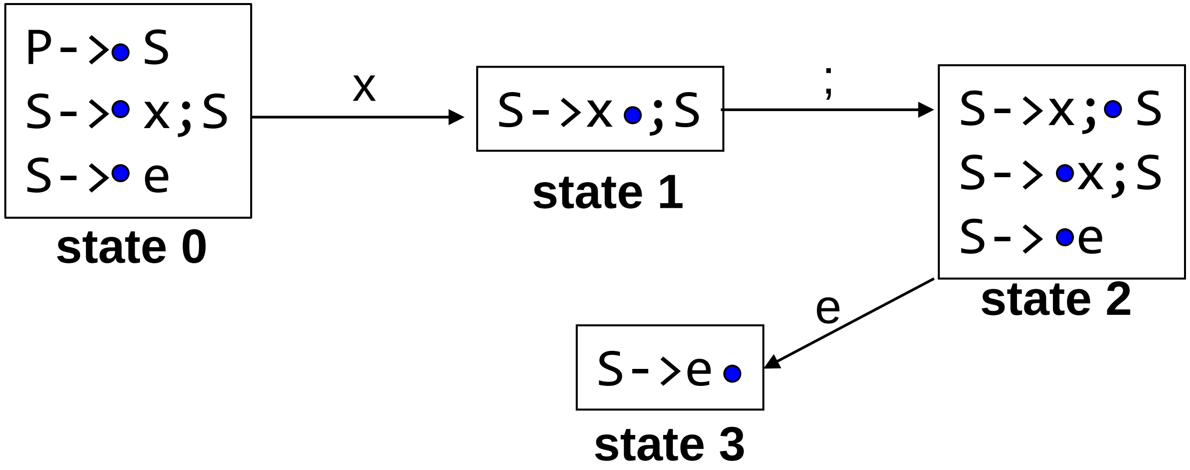
Compute successor (of state 2) under symbol e



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

Example: CFSM

Compute successor (of state 2) under symbol e



Grammar

P->S

S->x;S

S->e

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

Example: CFSM

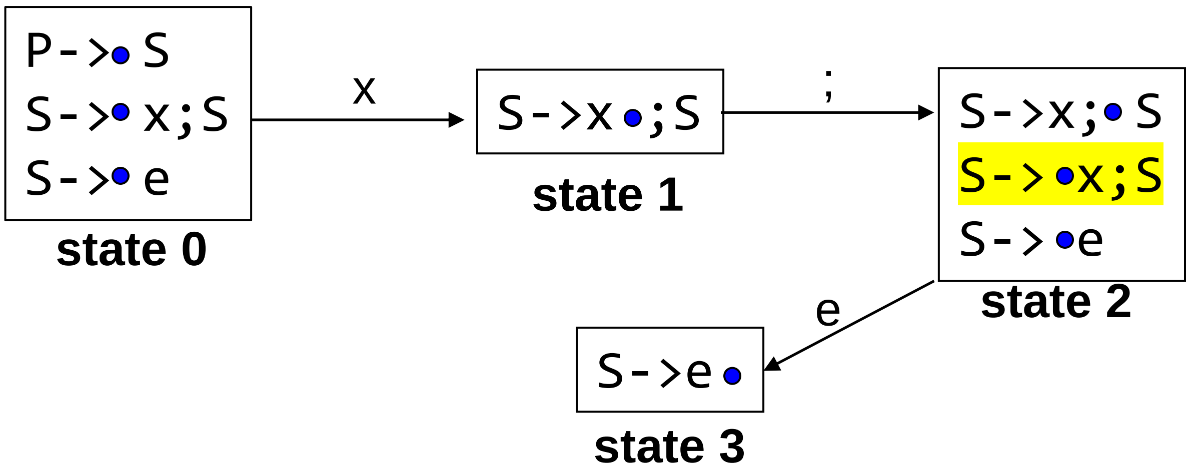
Grammar

P->S

S->x;S

S->e

Compute successor (of state 2) under symbol x



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

Example: CFSM

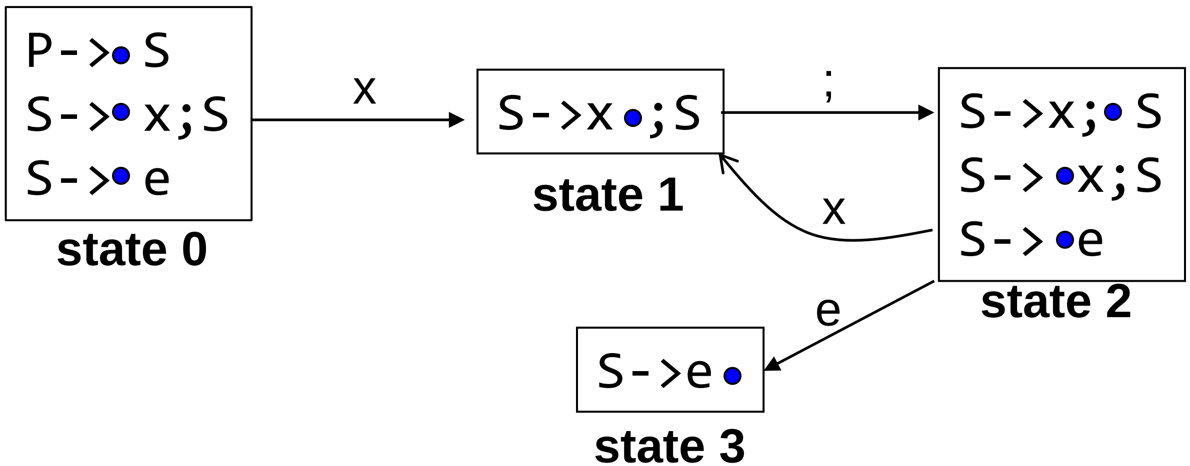
Grammar

P->S

S->x;S

S->e

Compute successor (of state 2) under symbol x



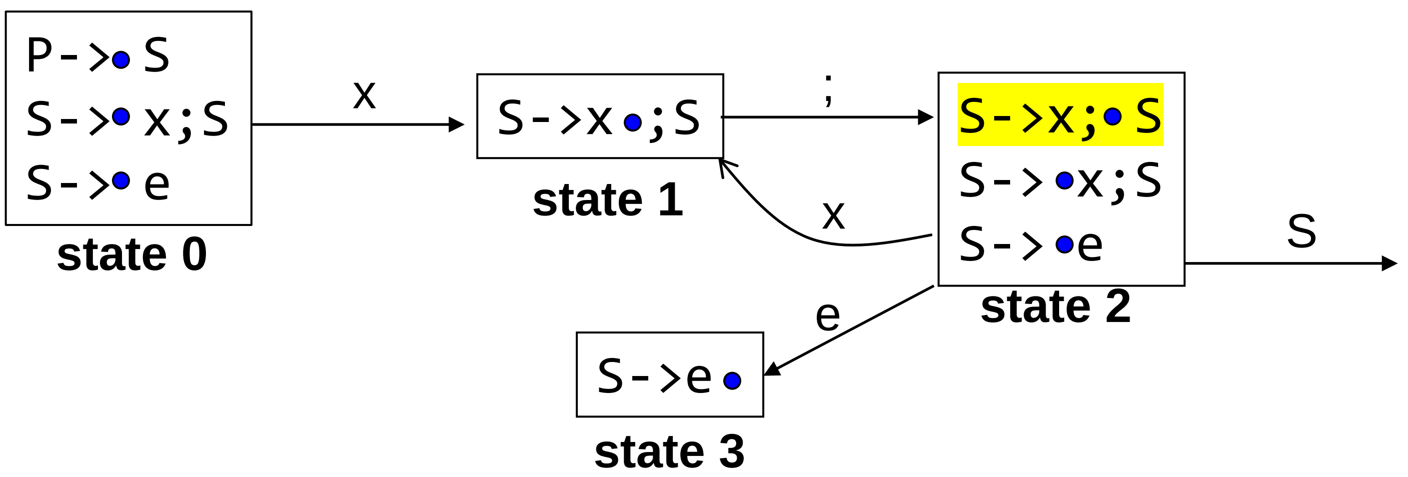
Consider items (in state 2), where x is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

Grammar

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

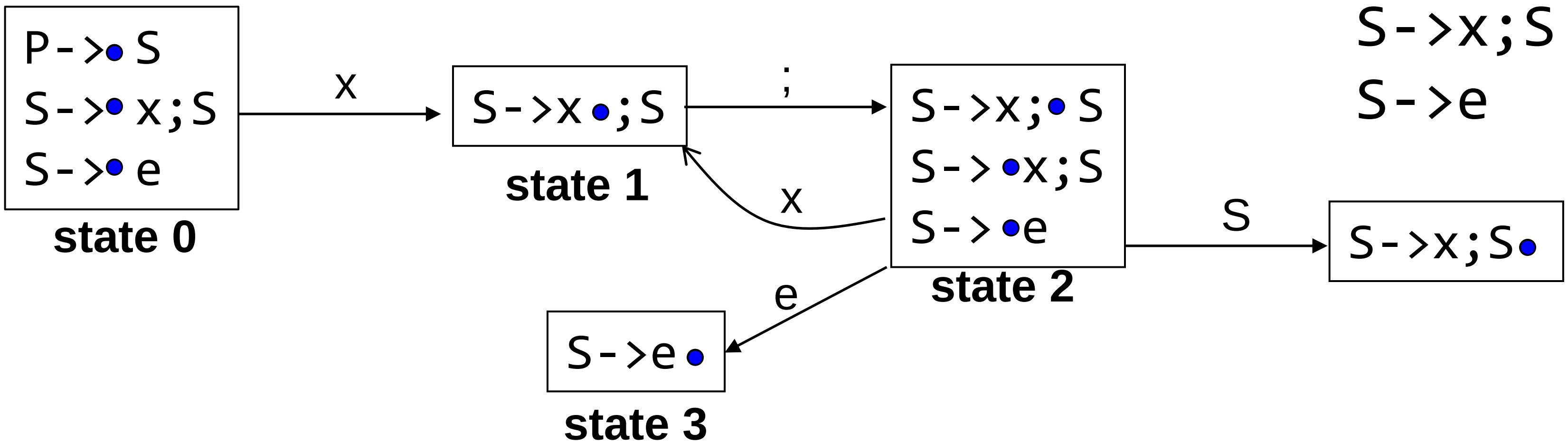
Compute successor (of state 2) under symbol S



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

Example: CFSM

Compute successor (of state 2) under symbol S



Grammar

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

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

Example: CFSM

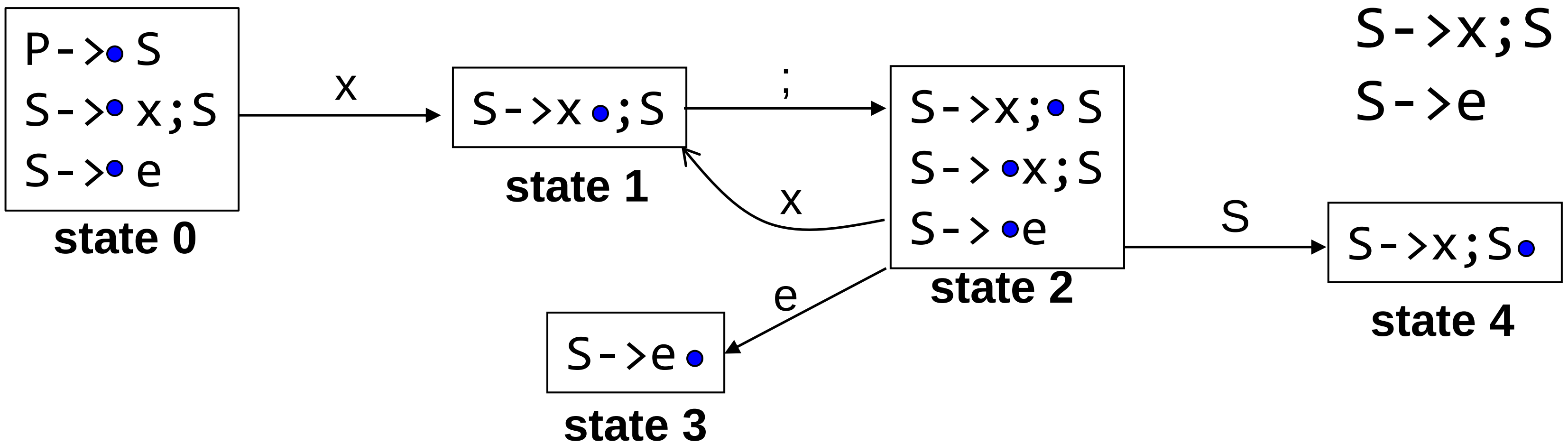
Grammar

P->S

S->x;S

S->e

Compute successor (of state 2) under symbol S



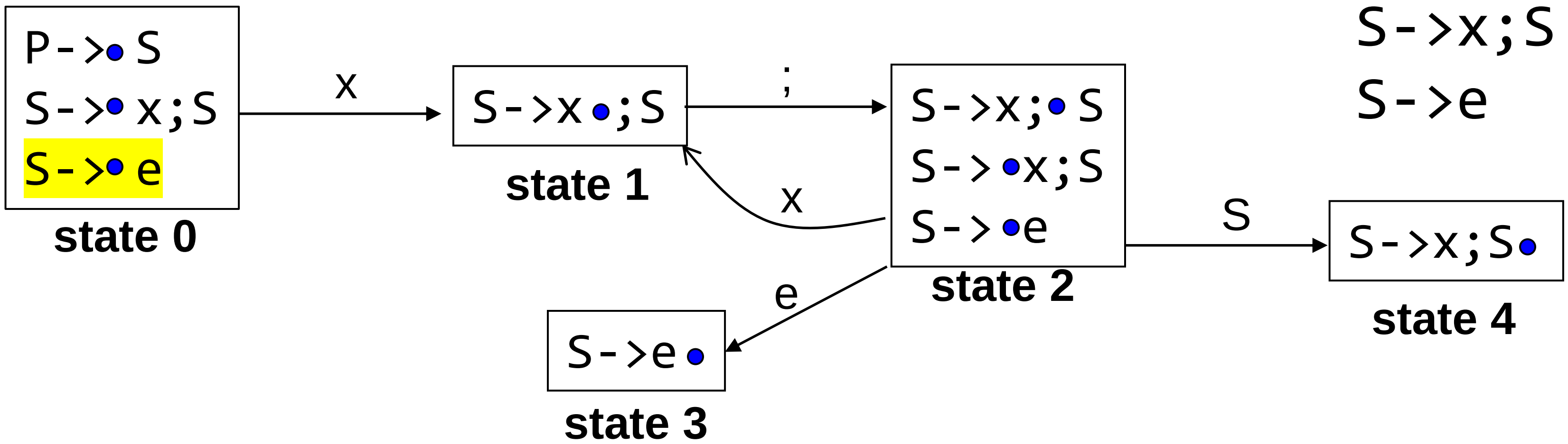
Consider items (in state 2), where S is to the immediate right of Dot. Advance Dot by one symbol. No more items to be added. Becomes another state in CFSM.

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

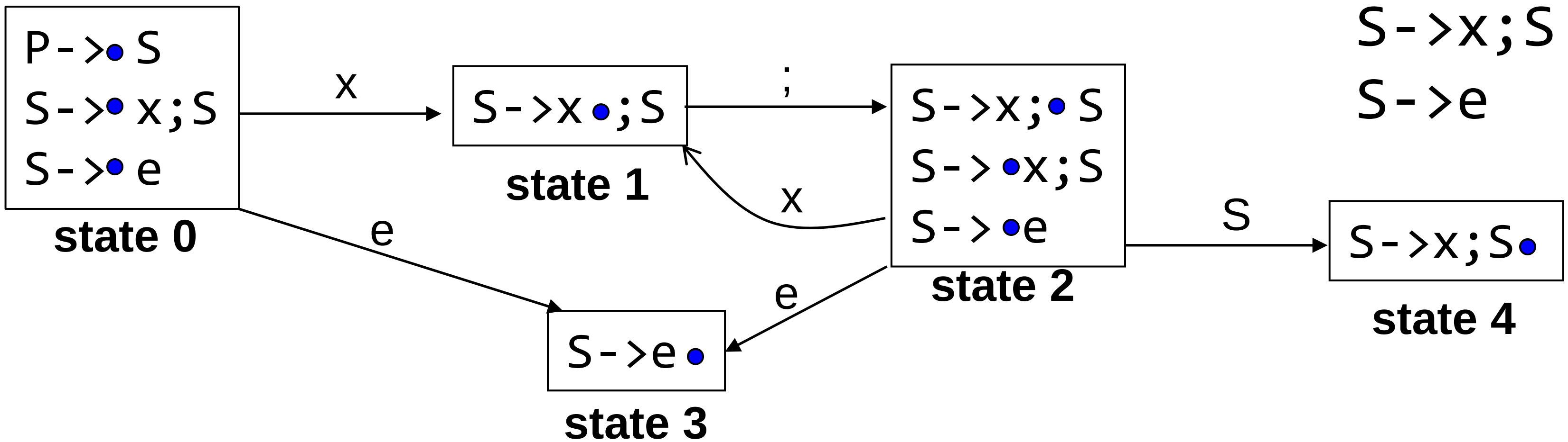
Grammar

P->S

S->x;S

S->e

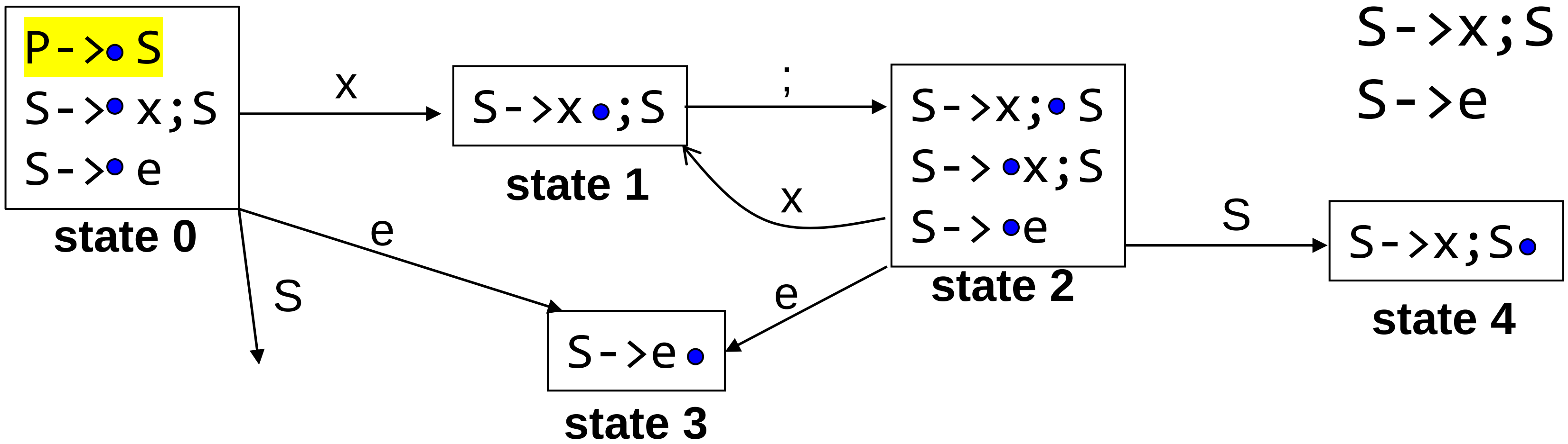
Compute successor (of state 0) under symbol e



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

Example: CFSM

Compute successor (of state 0) under symbol S



Grammar

P->S

S->x;S

S->e

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

Example: CFSM

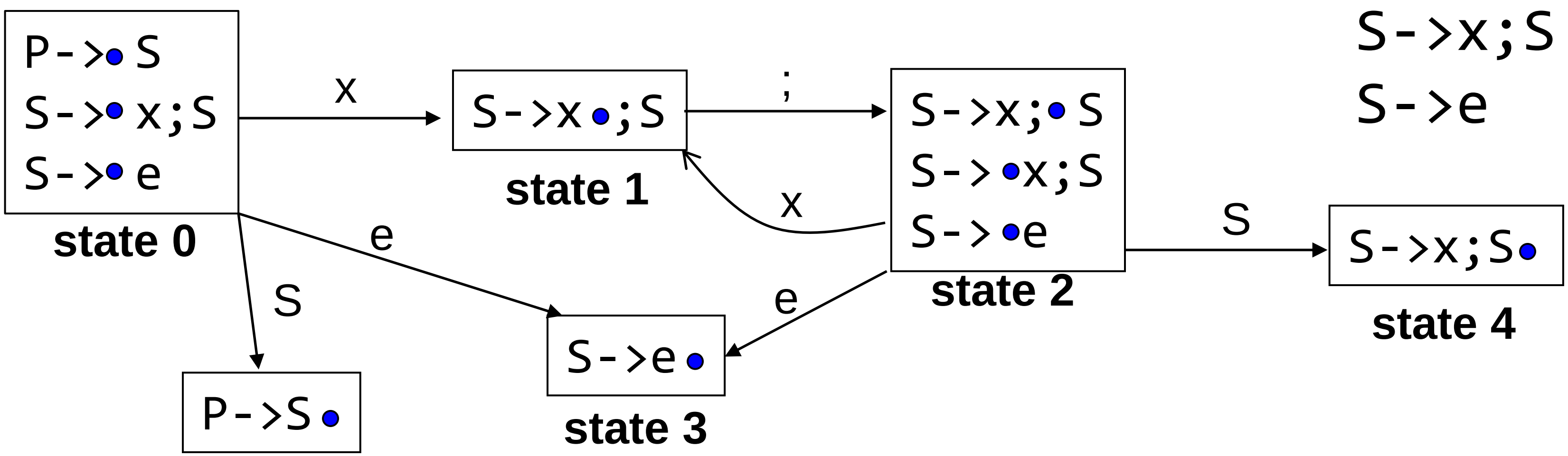
Grammar

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

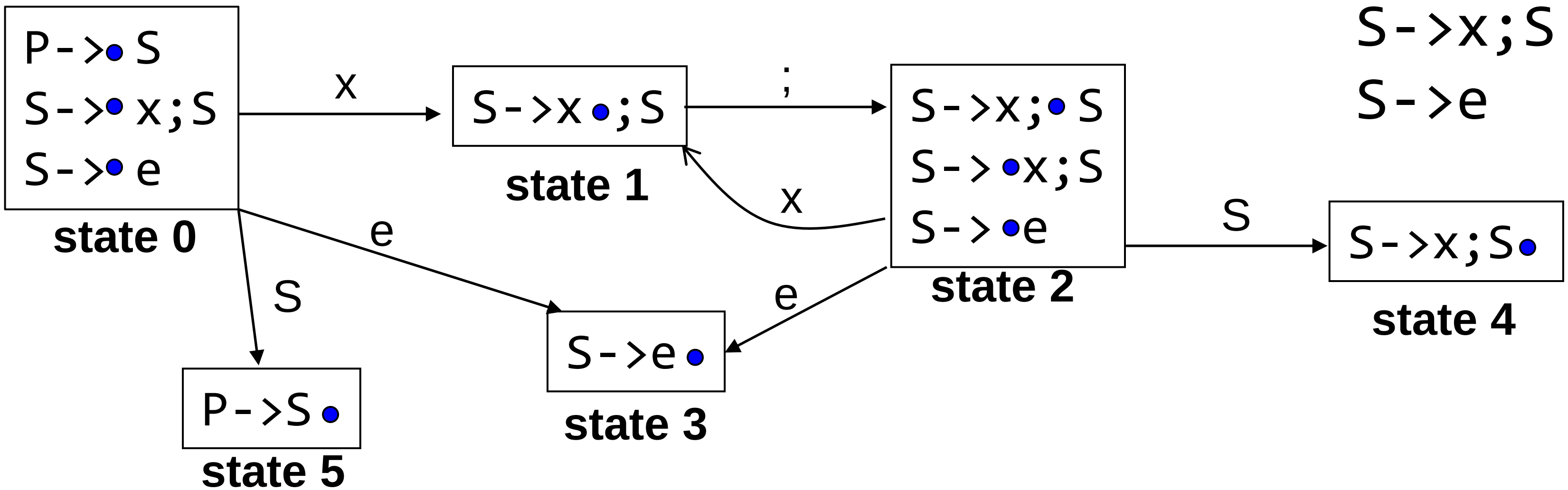
Grammar

P->S

S->x;S

S->e

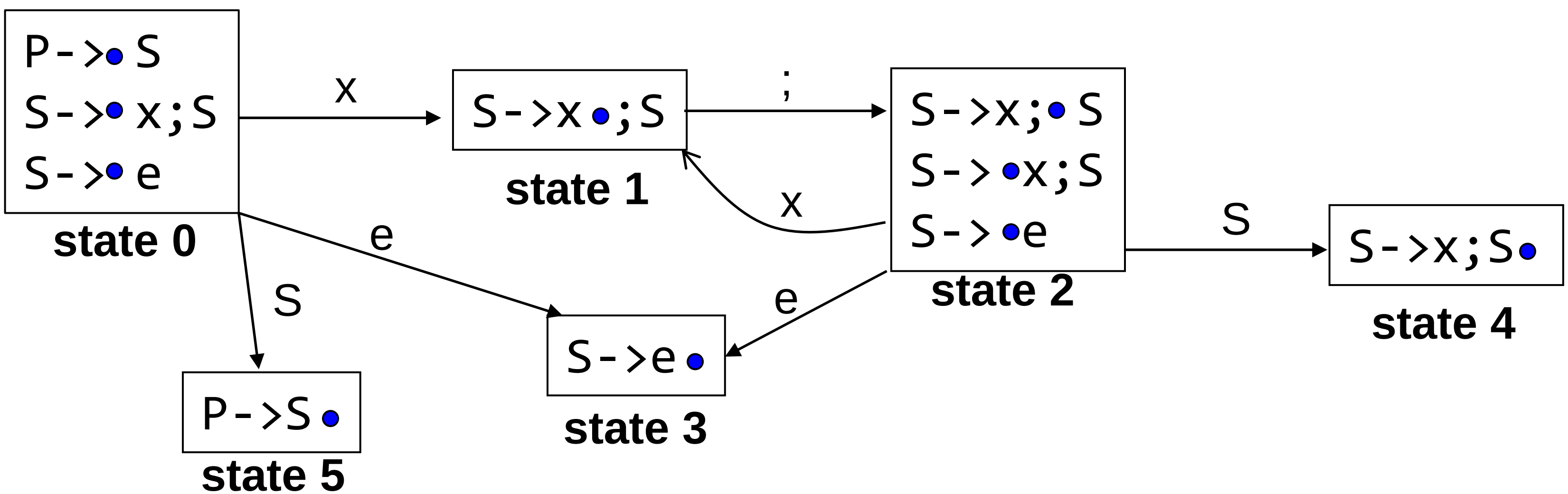
Compute successor (of state 0) under symbol S



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



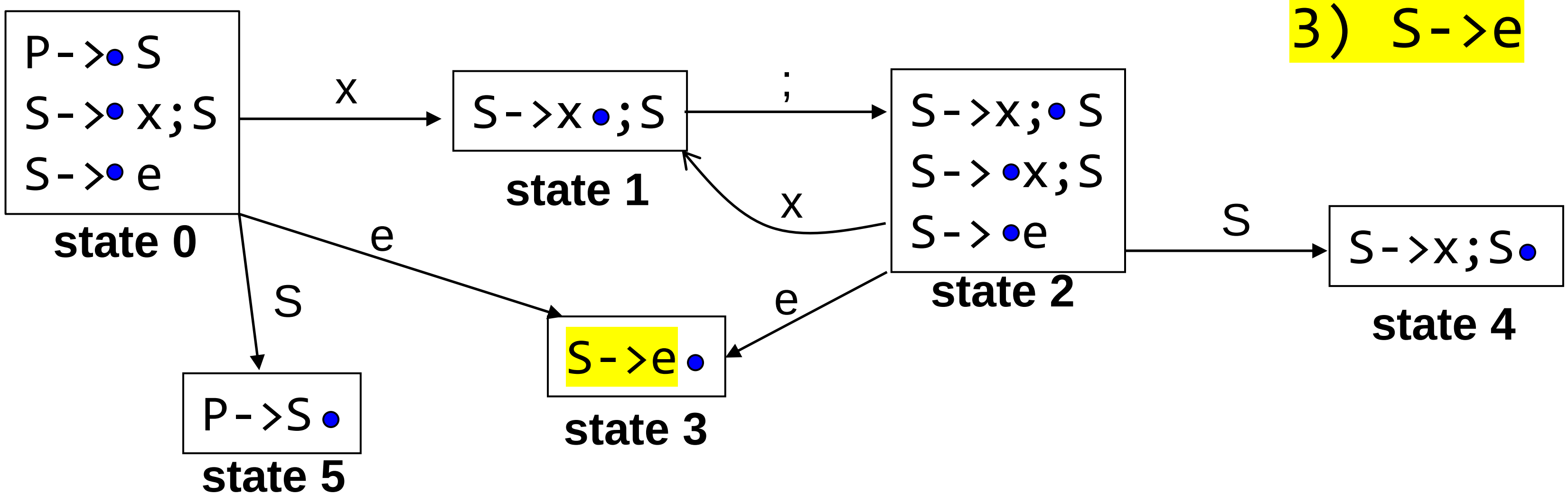
Example: CFSM

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

Reduce 3

Grammar

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



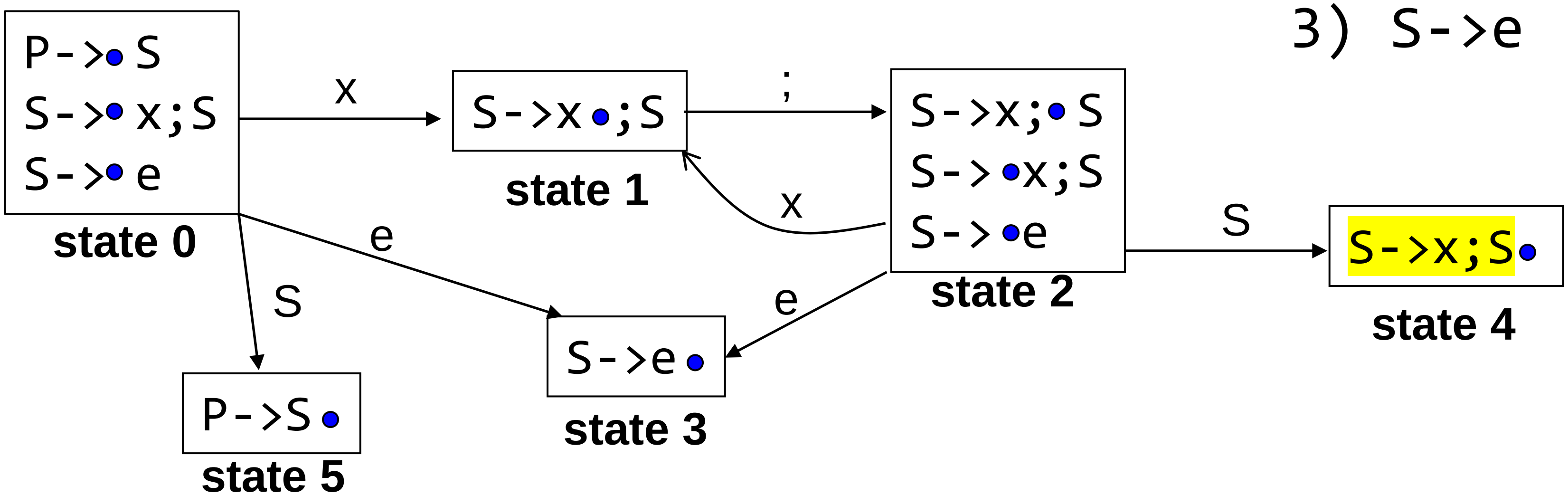
Example: CFSM

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

Reduce 2

Grammar

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



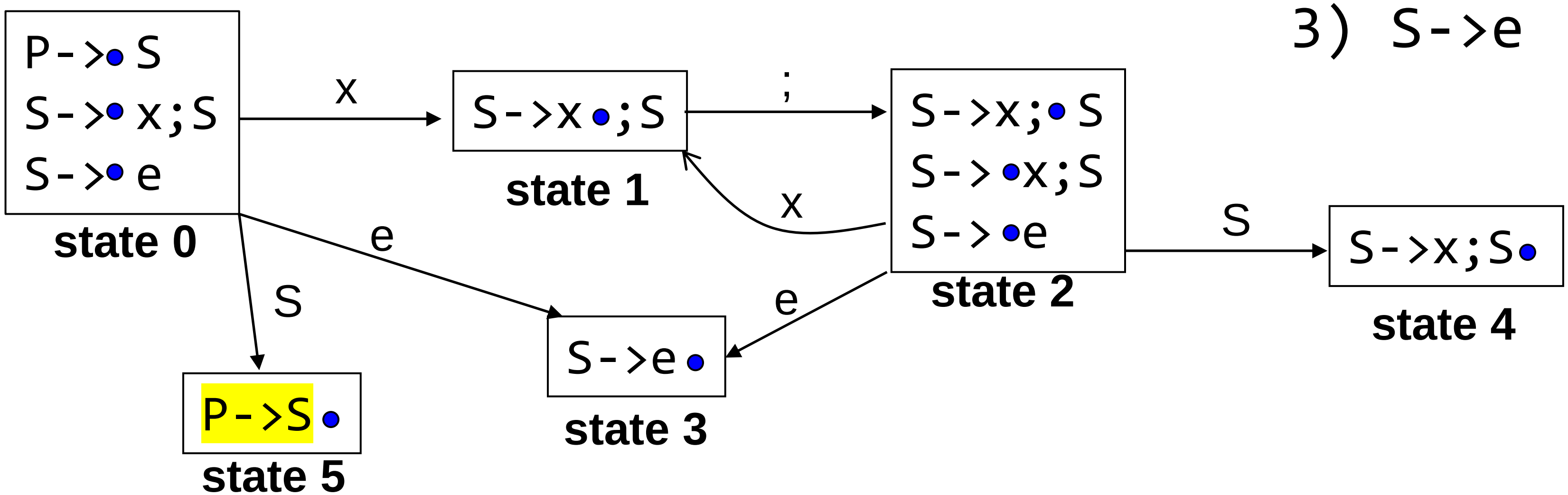
Example: CFSM

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

Accept

Grammar

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

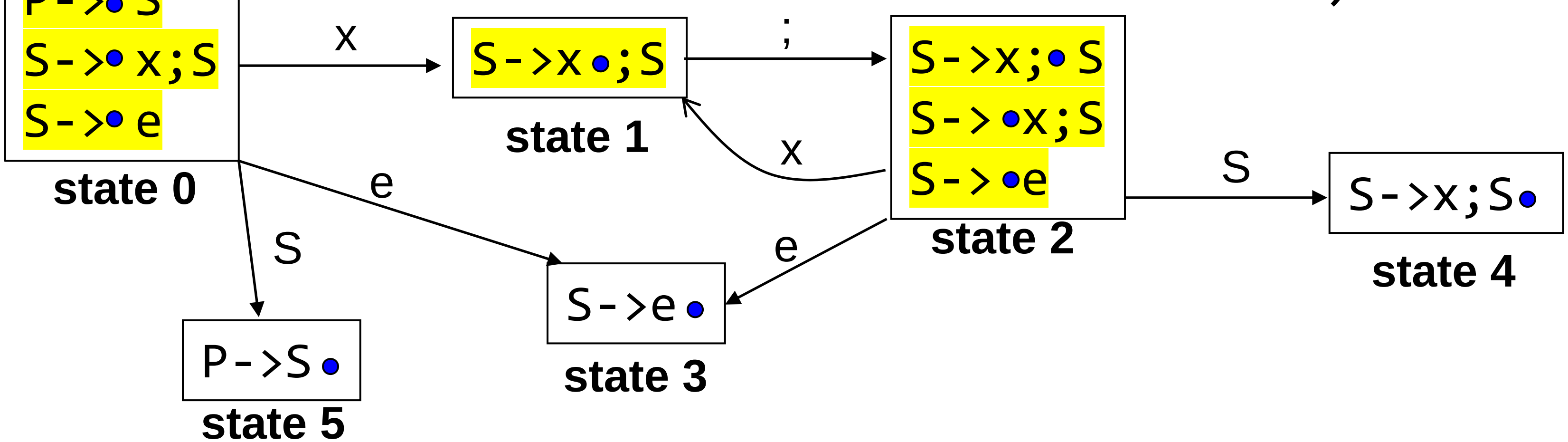


Example: CFSM

- Remaining states become *shift* states

Grammar

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



Conflicts

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

  Shift-reduce conflict

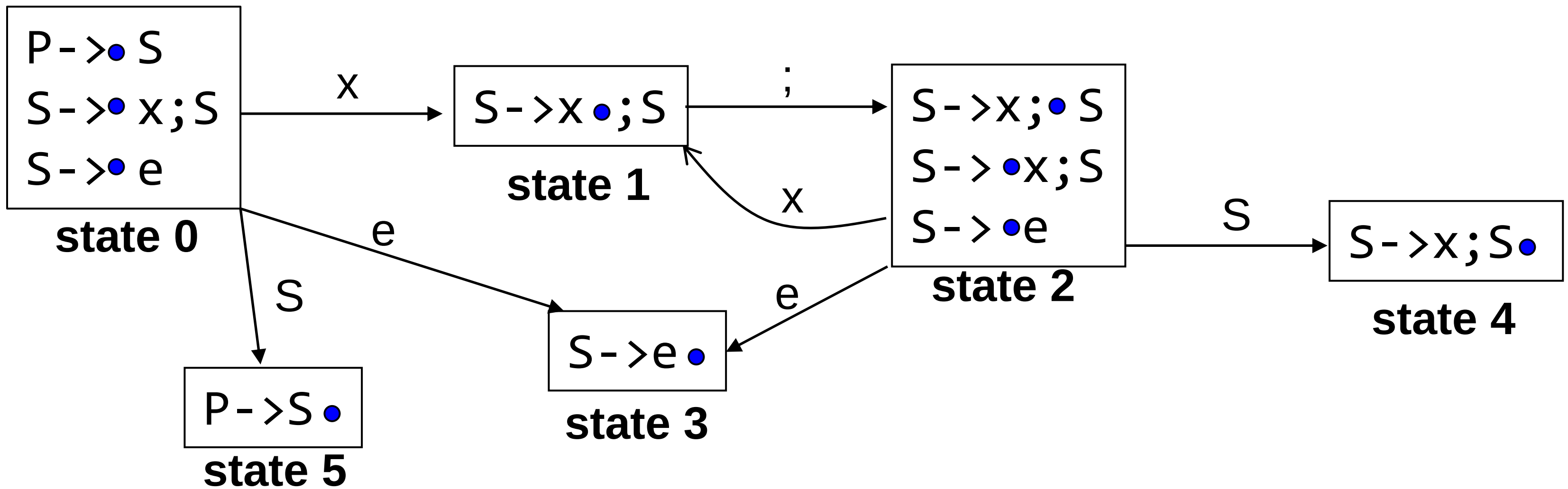
  Parser is unable to decide between shifting and reducing

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

  Reduce-Reduce conflict

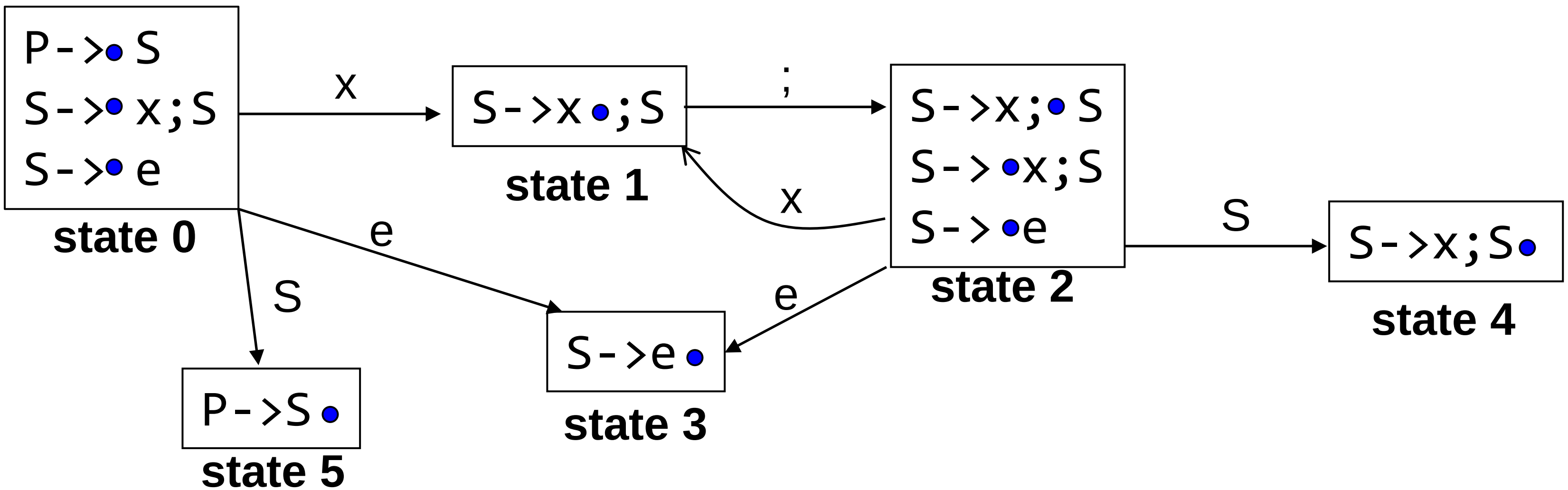
  Parser is unable to decide between which productions to choose for reducing

Example: goto table



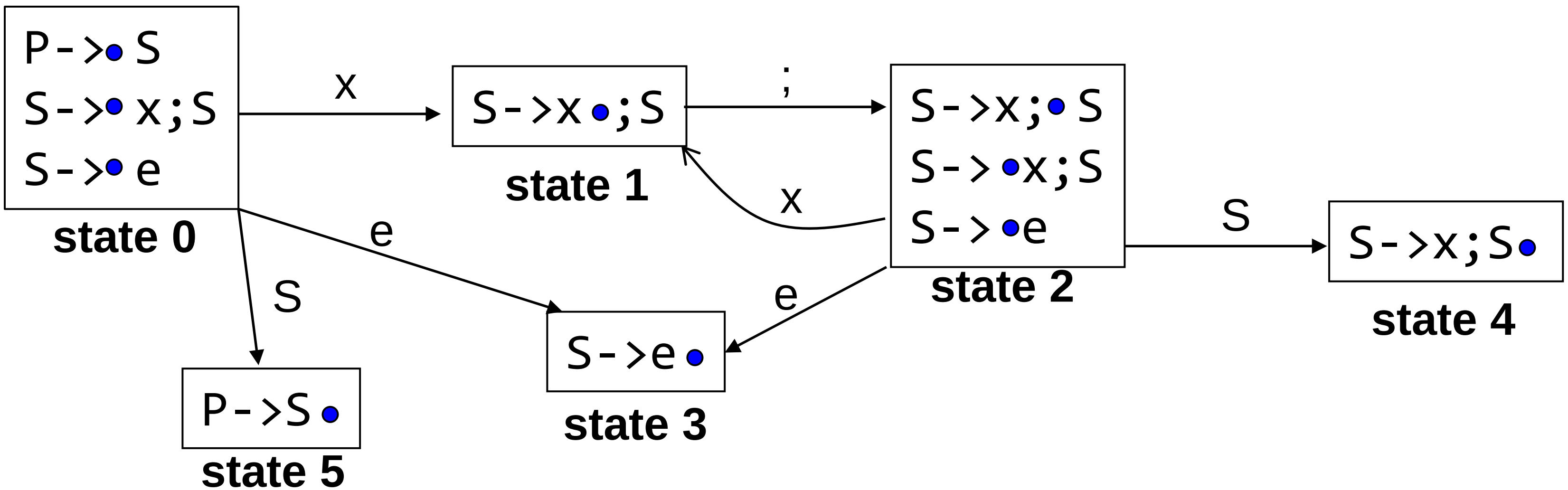
- construct transition table from CFSM.
  - Number of rows = number of states
  - Number of columns = number of symbols

Example: goto table



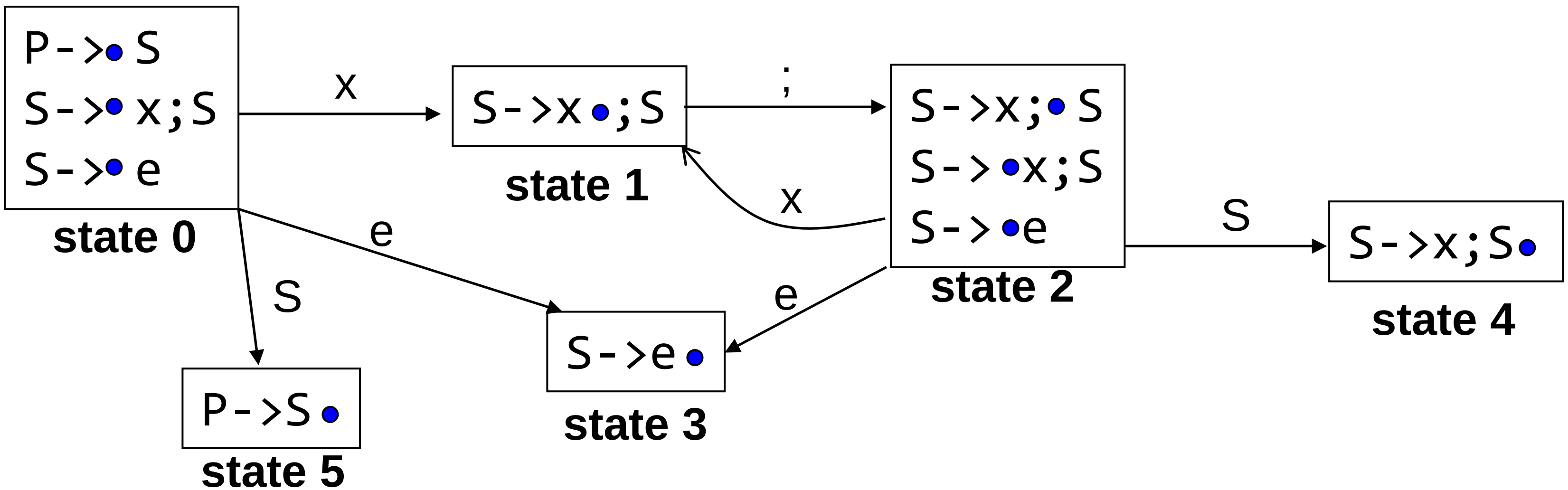
state	x	;	e	P	S
0	1		3		5
1		2			
2	1		3		4
3					
4					
5					

Example: action table



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

Example: action table



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

Example

- Example of LR(0) parsing
  - No lookahead involved
  - Operate based on the parse stack state and with goto and action tables

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

Suggested Reading

- Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D.Ullman: Compilers: Principles, Techniques, and Tools, 2/E, AddisonWesley 2007
  - Chapter 2 (2.4), Chapter 4
- Fisher and LeBlanc: Crafting a Compiler with C
  - Chapter 4, Chapter 5, and Chapter 6