Bottom Up Parsing

Bottom-Up Parsing

- LR methods (Left-to-right, Rightmost derivation)
  - SLR, Canonical LR, LALR
- Other special cases:
  - Shift-reduce parsing
  - Operator-precedence parsing

Operations:

Shift, Reduce, Error, Accept

$S \rightarrow \mathbf{a}\ A\ B\ \mathbf{e}$
$A \rightarrow A\ \mathbf{b}\ \mathbf{c}$
$A \rightarrow \mathbf{b}$
$B \rightarrow \mathbf{d}$

LL(1) parse
LtoR scan
LM derivation

LR
RM derivation
Sentential form

S	**reduce**
a *A* *B* **e**	**reduce, shift**
a *A* **d**	**reduce, shift**
a *A* **b c**	**shift, shift**
a *A* **b c d e**	**reduce**

a b b c d e

a *A* *A* **c d e**
a *A* *A* **c** *B* **e**

Shift- PUSH(char from String)
Reduce – POP (TOP/TOPs) replace with NT

Shift-Reduce Parsing

Grammar:

$S \rightarrow \mathbf{a}\ A\ B\ \mathbf{e}$

$A \rightarrow A\ \mathbf{b}\ \mathbf{c} \mid \mathbf{b}$

$B \rightarrow \mathbf{d}$

These match production's right-hand sides

Reducing a sentence:

$\mathbf{a}\ \underline{\mathbf{b}}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ \underline{A\ \mathbf{b}\ \mathbf{c}}\ \mathbf{d}\ \mathbf{e}$

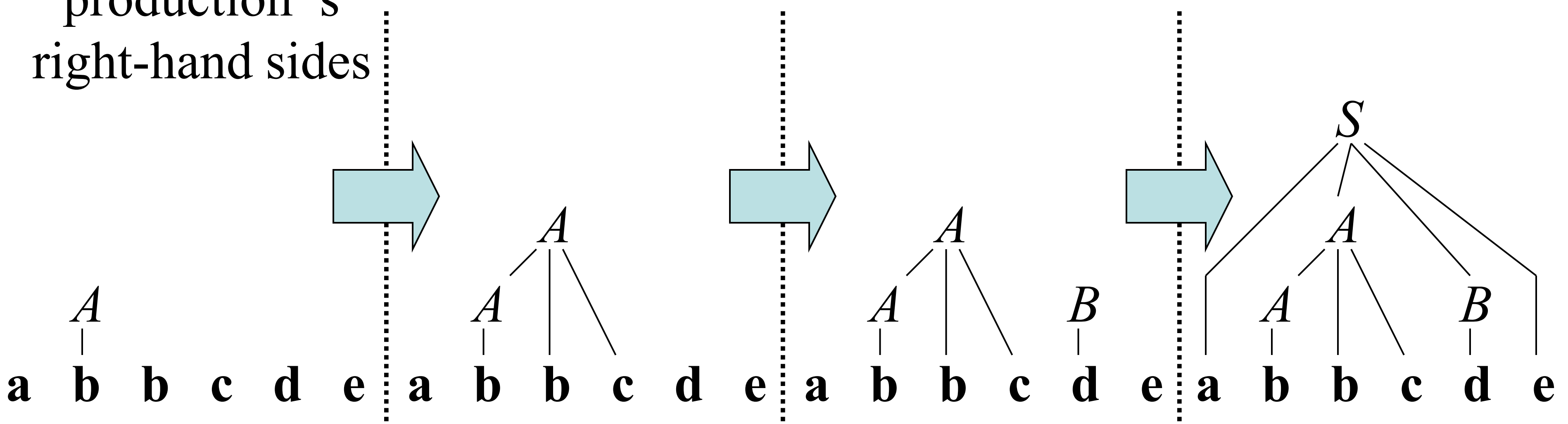
$\mathbf{a}\ A\ \underline{\mathbf{d}}\ \mathbf{e}$

$\underline{\mathbf{a}\ A\ B\ \mathbf{e}}\ .$

S

Shift-reduce corresponds to a rightmost derivation:

$$
\begin{aligned}
S &\Rightarrow_{rm} \mathbf{a}\ A\ B\ \mathbf{e} \\
&\Rightarrow_{rm} \mathbf{a}\ A\ \mathbf{d}\ \mathbf{e} \\
&\Rightarrow_{rm} \mathbf{a}\ A\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e} \\
&\Rightarrow_{rm} \mathbf{a}\ \mathbf{b}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}
\end{aligned}
$$



Handles

A *handle* is a substring of grammar symbols in a *right-sentential form* that matches a right-hand side of a production

Grammar:

$S \to \mathbf{a}\ A\ B\ \mathbf{e}$

$A \to A\ \mathbf{b}\ \mathbf{c} \mid \mathbf{b}$

$B \to \mathbf{d}$

$\mathbf{a}\ \underline{\mathbf{b}}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ \underline{A\ \mathbf{b}\ \mathbf{c}}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ A\ \underline{\mathbf{d}}\ \mathbf{e}$

$\underline{\mathbf{a}\ A\ B\ \mathbf{e}}$

S

Handle

$\mathbf{a}\ \underline{\mathbf{b}}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ A\ \underline{\mathbf{b}}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ A\ A\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

… ?

NOT a handle, because further reductions will fail (result is not a sentential form)

Operator Precedence

$a \gtrdot b$ a has higher precedence over b

$a \lessdot b$ a has lower precedence over b

$a \doteq b$ a has equal precedence over b

Note:

- id has highest precedence
- $ has lowest precedence
- Apply associativity in case of equal precedence

OP table

$E \rightarrow E+E \mid E*E \mid id$

lookahead

Stack[top]

	id	**+**	*****	**$**
id	Err	$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*****	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	Accept

$\$id+id*id\$$

$\$\lessdot id \gtrdot + \lessdot id \gtrdot * \lessdot id \gtrdot \$$

Basic Process

- Scan input string from left to right until $\gtrdot$
- Now scan backward from $\gtrdot$ until $\lessdot$
- String between $\lessdot \ldots \gtrdot$ is handle
- Reduce handle with head of the production
- Repeat until reaching start symbol

OP Algo

```
init STACK[top] to $
While do
   Let U be the stack[top]
   Let V be the next input symbol (lookahead)
   if U=V=$ then return ACCEPT
   if U⋖V or U≐V                           M[U][V]==2
     shift V onto STACK   //SHIFT
     advance input pointer(advance the lookahead)
   else if U⋗V                             M[U][V]==3
     do //REDUCE
       POP top of the stack, call it temp
     stop loop when M[stack[top]][temp]==2
   else
     error
end
```

STACK	INPUT	ACTION
`$`	`id + id * id $`	$\lessdot$
`$ id`	`+ id * id $`	$\gtrdot$
`$`	`+ id * id $`	$\lessdot$
`$ +`	`id * id $`	$\lessdot$
`$ + id`	`* id $`	$\gtrdot$
`$ +`	`* id $`	$\lessdot$
`$ + *`	`id $`	$\lessdot$
`$ + * id`	`$`	$\gtrdot$
`$ + *`	`$`	$\gtrdot$
`$ +`	`$`	$\gtrdot$
`$`	`$`	accept

	id	+	*	**$**
id	0	$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	1

M[stack[top]][V]

$\lessdot$ | $\doteq$ Shift

$\gtrdot$ Reduce

`id id + id $` ERR

ERR-0

Shift-2

Accept-1

Reduce-3

Precedence Relationship

- Need Two lists Firstop+ and Lastop+

- Firstop+:List of all terminals which can appear **first** in any body of production
- Lastop+: List of all terminals which can appear **last** in any body of production

Precedence Relationship

X→a…|Bc…|a|A   put a,B,c,A in Firstop(X)

Y→…u|…vW|u|P put u,v,W,P Lastop(Y)

Compute Firstop+ and Lastop+

- Replace each Non Terminal with its Firstop for Firstop+
- Same for Lastop+
- Drop all non terminals

Example

S→(L)|a

L→L,S|S

Firstop(S)={(a}   Lastop(S)={) a}

Firstop(L)={L , S}   Lastop(L)={, S}

Firstop+(S)={(a}   Lastop+(S)={) a}

Firstop+(L)={, (a}   Lastop+(L)={,) a}

$E \rightarrow E+T|T$

$T \rightarrow T*F|F$

$F \rightarrow (E)|id$

Firstop(E)={E + T} Lastop(E)={+ T}

Firstop(T)={T * F} Lastop(T)={* F}

Firstop(F)={(id} Lastop(F)={) id}

Firstop+(E)={+ * (id} Lastop+(E)={+ *) id}

Firstop+(T)={* (id} Lastop+(T)={*) id}

Firstop+(F)={(id} Lastop+(F)={) id}

Precedence Matrix

- Terminal **a** immediately precedes B in any production, put **a**$\lessdot\alpha$ where α is any terminal in Firstop+(B)

  A→ **a**B… then **a** $\lessdot$ Firstop+(B)
- Terminal **b** immediately follows C in any production, put $\beta\gtrdot$**b** where β is any terminal in Lastop+(C)

  A→…C**b**… then Lastop+(C) $\gtrdot$ **b**
- for aBc or ac occurs in any production then a $\lessdot$ c
- \$$\lessdot$Firstop+ lists
- Lastop+ lists $\gtrdot$ \$

Firstop+(E)={+,*,id,(}
Firstop+(T)={*,id,(}
Firstop+(F)={id,(}

Lastop+(E)={+, *, id,)}
Lastop+(T)={*, id,)}
Lastop+(F)={id,)}

E$\rightarrow$E+T|T
T$\rightarrow$T*F|F
F$\rightarrow$(E)|id

	id	**+**	*****	**(**	**)**	**$**
id		>	>		>	>
+	<	>	<	<	>	>
*****	<	>	>	<	>	>
(	<	<	<	<	<	
)		>	>		>	>
$	<	<	<	<		Acc

Example

S→(L)|a

L→L,S|S

Firstop+(S)={(a}   Lastop+(S)={) a}

Firstop+(L)={, (a}   Lastop+(L)={,) a}

(a,((a,a),(a,a)))

Stack Implementation of Shift-Reduce Parsing

Grammar:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow \mathbf{(}\ E\ \mathbf{)}$

$E \rightarrow \mathbf{id}$

Stack	Input	Action
$\$$	$\mathbf{id{+}id{*}id\$}$	shift
$\$\underline{\mathbf{id}}$	$\mathbf{{+}id{*}id\$}$	reduce $E \rightarrow \mathbf{id}$
$\$E$	$\mathbf{{+}id{*}id\$}$	shift
$\$E+$	$\mathbf{id{*}id\$}$	shift
$\$E+\underline{\mathbf{id}}$	$\mathbf{{*}id\$}$	reduce $E \rightarrow \mathbf{id}$
$\$E+E$	$\mathbf{{*}id\$}$	shift (or reduce?)
$\$E+E*$	$\mathbf{id\$}$	shift
$\$E+E*\underline{\mathbf{id}}$	$\mathbf{\$}$	reduce $E \rightarrow \mathbf{id}$
$\$E+\underline{E*E}$	$\mathbf{\$}$	reduce $E \rightarrow E * E$
$\$\underline{E+E}$	$\mathbf{\$}$	reduce $E \rightarrow E + E$
$\$E$	$\mathbf{\$}$	accept

How to resolve conflicts?

Found handles to reduce

Conflicts

- *Shift-reduce* and *reduce-reduce* conflicts are caused by
  - The limitations of the LR parsing method (even when the grammar is unambiguous)
  - Ambiguity of the grammar

Shift-Reduce Parsing: Shift-Reduce Conflicts

Ambiguous grammar:
$S \rightarrow$ **if** E **then** S
| **if** E **then** S **else** S
| **other**

Stack	Input	Action
$…	…**$**	…
$…<u>**if** *E* **then** *S*</u>	**else**…**$**	shift or reduce?
$…<u>**if** *E* **then** *S* **else**</u>		

Resolve in favor of shift, so **else** matches closest **if**

Shift-Reduce Parsing: Reduce-Reduce Conflicts

Stack	Input	Action
$	**aa$**	shift
$<u>a</u>	**a$**	reduce $A \rightarrow$ **a** <u>or</u> $B \rightarrow$ **a** ?

Grammar:

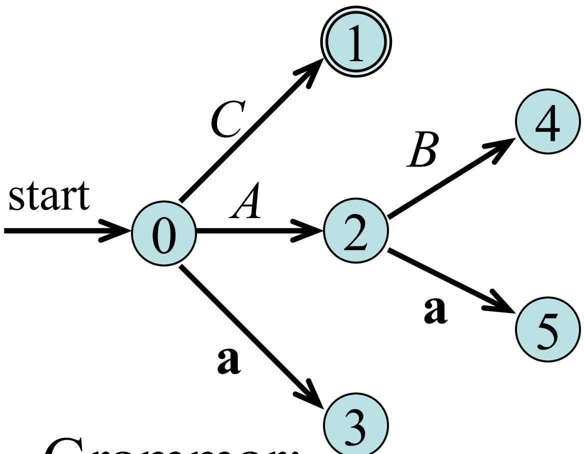
$C \rightarrow A\ B$

$A \rightarrow \mathbf{a}$

$B \rightarrow \mathbf{a}$

Resolve in favor of reducing $A \rightarrow$ **a**, otherwise we're stuck!

LR(k) Parsers: Use a DFA for Shift/Reduce Decisions



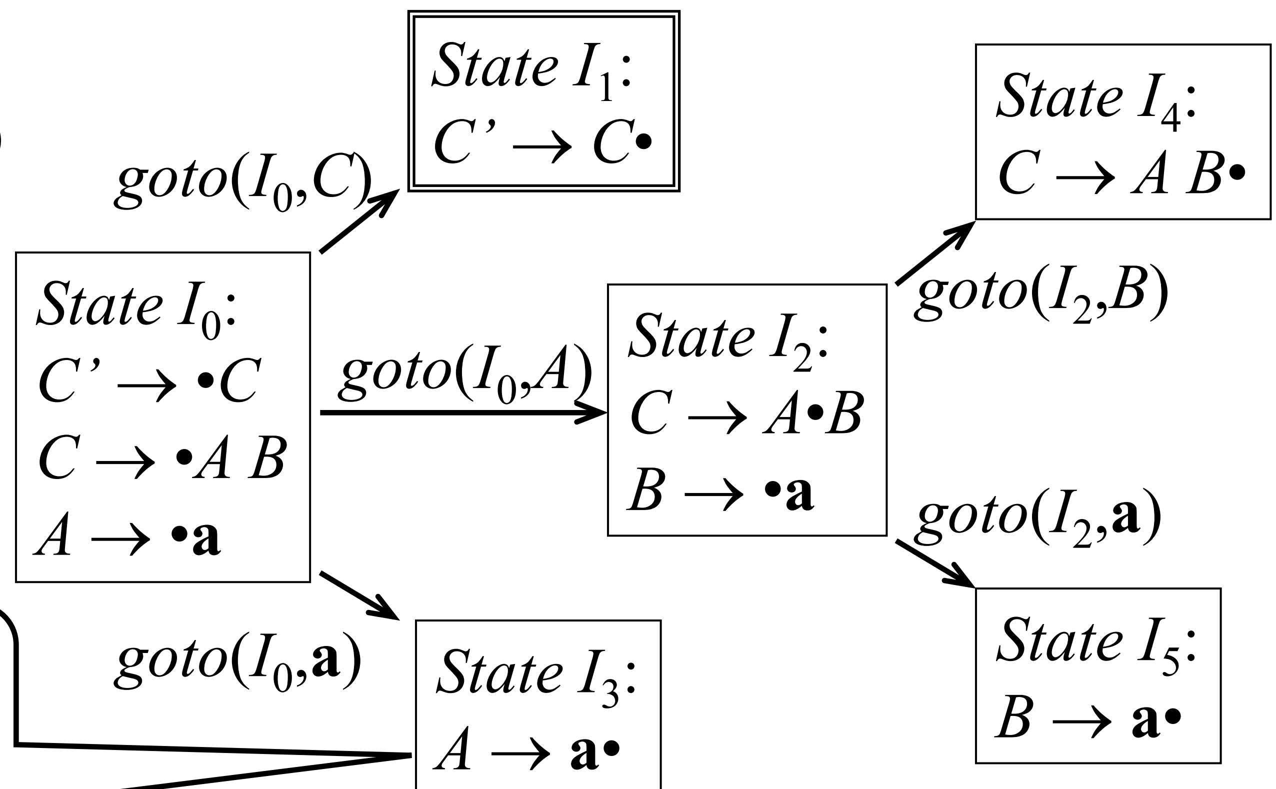
Grammar:

$C' \to C$

$C \to A\ B$

$A \to \mathbf{a}$

$B \to \mathbf{a}$



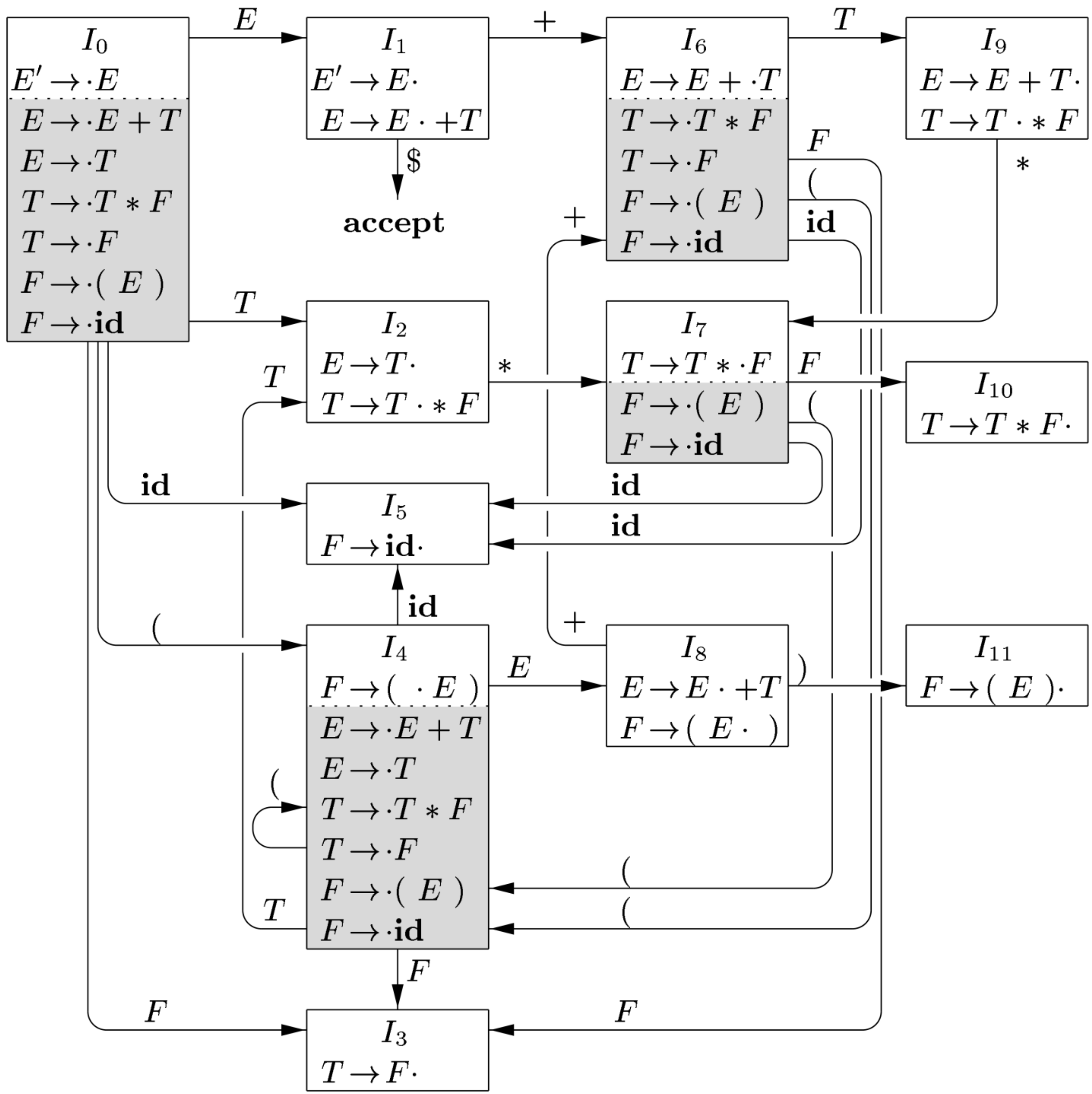
Can only reduce $A \to \mathbf{a}$ (not $B \to \mathbf{a}$)

$E' \rightarrow E$

$E \rightarrow E+T|T$

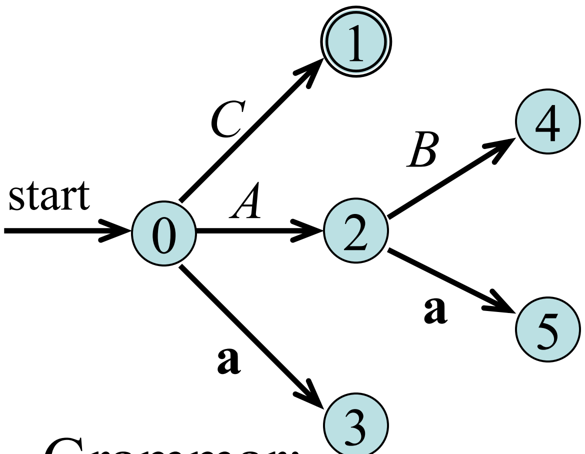
$T \rightarrow T*F|F$

$F \rightarrow (E)|id$



I_0
$E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_1
$E' \to E \cdot$
$E \to E \cdot + T$
accept
I_2
$E \to T \cdot$
$T \to T \cdot * F$
I_3
$T \to F \cdot$
I_4
$F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_5
$F \to \mathbf{id} \cdot$
I_6
$E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_7
$T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_8
$E \to E \cdot + T$
$F \to (E \cdot)$
I_9
$E \to E + T \cdot$
$T \to T \cdot * F$
I_{10}
$T \to T * F \cdot$
I_{11}
$F \to (E) \cdot$

LR(k) Parsers: Use a DFA for Shift/Reduce Decisions



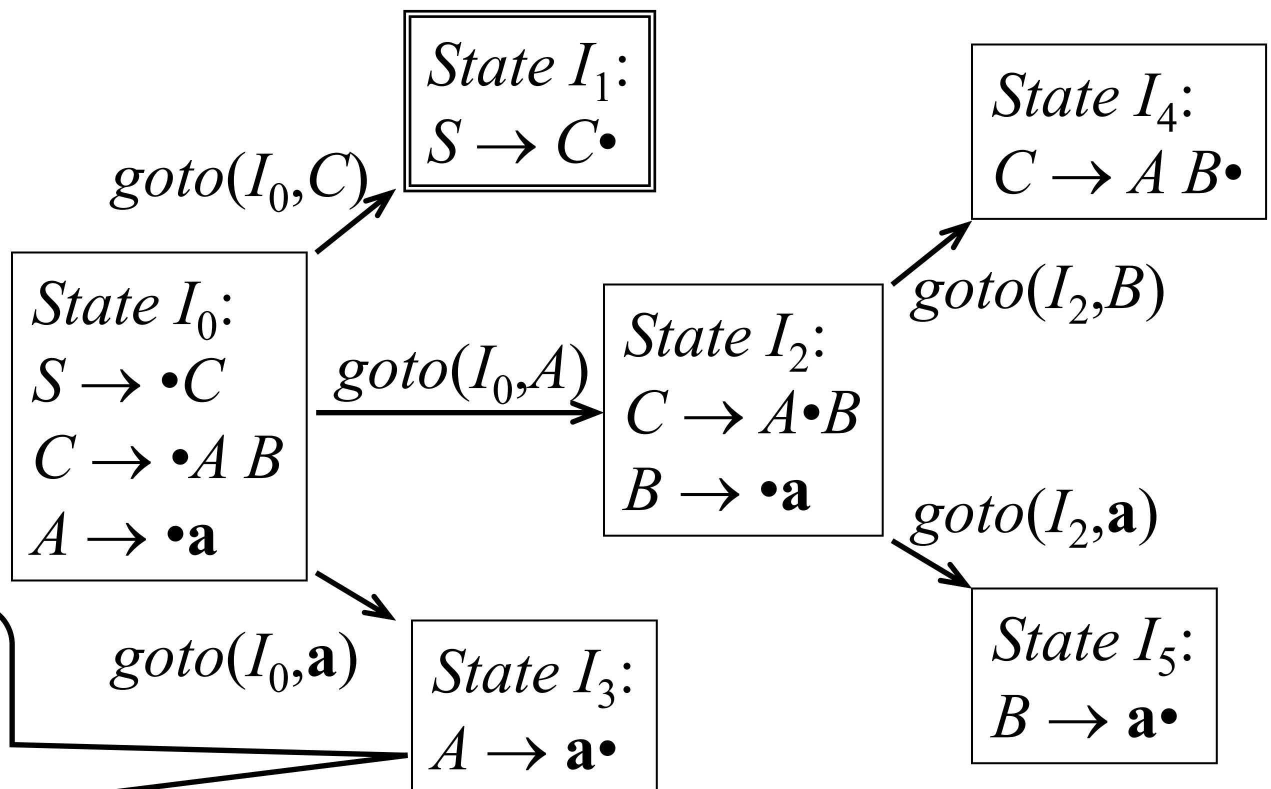
Grammar:

$S \rightarrow C$

$C \rightarrow A\ B$

$A \rightarrow \mathbf{a}$

$B \rightarrow \mathbf{a}$



Can only reduce $A \rightarrow \mathbf{a}$ (not $B \rightarrow \mathbf{a}$)

DFA for Shift/Reduce Decisions

The states of the DFA are used to determine if a handle is on top of the stack

Grammar:

1 $S \rightarrow C$

2 $C \rightarrow A\ B$

3 $A \rightarrow \mathbf{a}$

4 $B \rightarrow \mathbf{a}$

State I_0:
$S \rightarrow \bullet C$
$C \rightarrow \bullet A\ B$
$A \rightarrow \bullet\mathbf{a}$

$goto(I_0,\mathbf{a})$

State I_3:
$A \rightarrow \mathbf{a}\bullet$

Stack	Input	Action
\$ 0	**aa\$**	start in state 0
\$ $\underline{0}$	**$\underline{\text{a}}$a\$**	shift (and goto state 3)
\$ 0 **a** 3	**a\$**	reduce $A \rightarrow \mathbf{a}$ (goto 2)
\$ 0 A 2	**a\$**	shift (goto 5)
\$ 0 A 2 **a** 5	**\$**	reduce $B \rightarrow \mathbf{a}$ (goto 4)
\$ 0 A 2 B 4	**\$**	reduce $C \rightarrow AB$ (goto 1)
\$ 0 C 1	**\$**	accept ($S \rightarrow C$)

DFA for Shift/Reduce Decisions

The states of the DFA are used to determine if a handle is on top of the stack

Grammar:

$S \to C$

$C \to A\ B$

$A \to \mathbf{a}$

$B \to \mathbf{a}$

Stack	Input	Action
\$ 0	**aa\$**	start in state 0
\$ 0	**aa\$**	shift (and goto state 3)
\$ $\underline{0}$ $\underline{\mathbf{a}}$ 3	**a\$**	reduce $A \to \mathbf{a}$ (goto 2)
\$ 0 A 2	**a\$**	shift (goto 5)
\$ 0 A 2 **a** 5	**\$**	reduce $B \to \mathbf{a}$ (goto 4)
\$ 0 A 2 B 4	**\$**	reduce $C \to AB$ (goto 1)
\$ 0 C 1	**\$**	accept ($S \to C$)

State I_0:

$S \to \bullet C$

$C \to \bullet A\ B$

$A \to \bullet\mathbf{a}$

goto(I_0, A)

State I_2:

$C \to A \bullet B$

$B \to \bullet\mathbf{a}$

DFA for Shift/Reduce Decisions

The states of the DFA are used to determine if a handle is on top of the stack

Grammar:

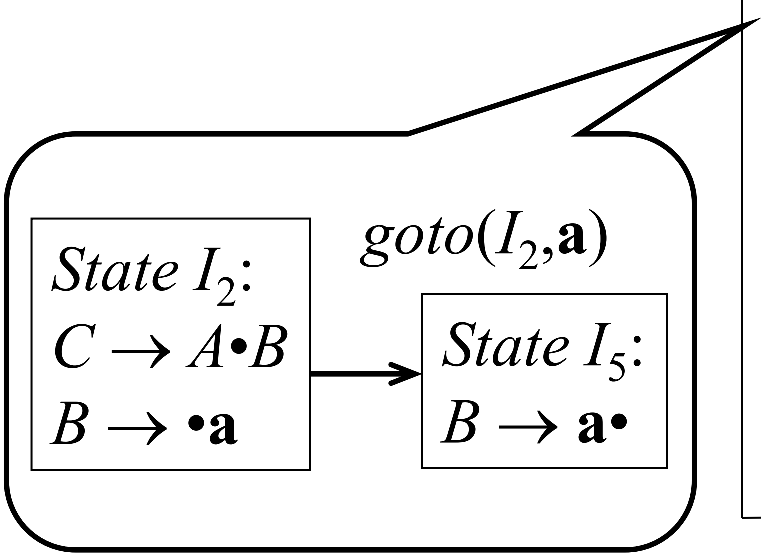
$S \to C$

$C \to A\ B$

$A \to \mathbf{a}$

$B \to \mathbf{a}$

Stack	Input	Action
\$ 0	**aa\$**	start in state 0
\$ 0	**aa\$**	shift (and goto state 3)
\$ 0 **a** 3	**a\$**	reduce $A \to \mathbf{a}$ (goto 2)
\$ 0 A $\underline{2}$	$\underline{\mathbf{a}}$**\$**	shift (goto 5)
\$ 0 A 2 **a** 5	**\$**	reduce $B \to \mathbf{a}$ (goto 4)
\$ 0 A 2 B 4	**\$**	reduce $C \to AB$ (goto 1)
\$ 0 C 1	**\$**	accept ($S \to C$)



DFA for Shift/Reduce Decisions

The states of the DFA are used to determine if a handle is on top of the stack

Grammar:

$S \to C$

$C \to A\ B$

$A \to \mathbf{a}$

$B \to \mathbf{a}$

Stack	Input	Action
\$ 0	**aa\$**	start in state 0
\$ 0	**aa\$**	shift (and goto state 3)
\$ 0 **a** 3	**a\$**	reduce $A \to \mathbf{a}$ (goto 2)
\$ 0 A 2	**a\$**	shift (goto 5)
\$ 0 A $\underline{2}$ $\underline{\mathbf{a}}$ 5	**\$**	reduce $B \to \mathbf{a}$ (goto 4)
\$ 0 A 2 B 4	**\$**	reduce $C \to AB$ (goto 1)
\$ 0 C 1	**\$**	accept ($S \to C$)

State I_2:

$C \to A \bullet B$

$B \to \bullet\mathbf{a}$

$goto(I_2, B)$

State I_4:

$C \to A\ B \bullet$

DFA for Shift/Reduce Decisions

The states of the DFA are used to determine if a handle is on top of the stack

Grammar:

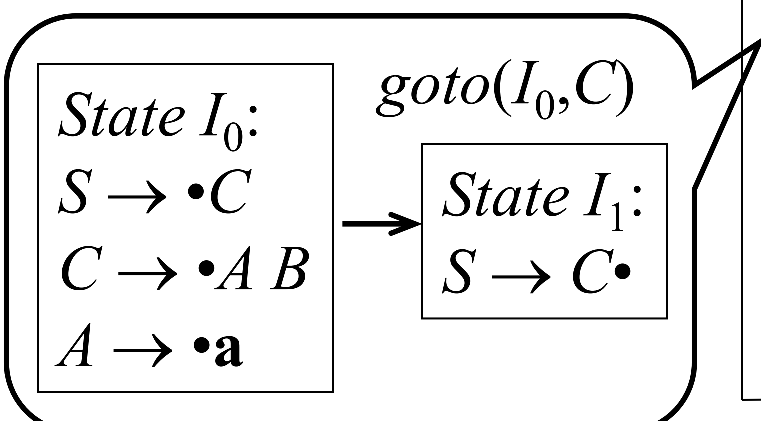
$S \rightarrow C$

$C \rightarrow A\ B$

$A \rightarrow \mathbf{a}$

$B \rightarrow \mathbf{a}$

Stack	Input	Action
\$ 0	**aa\$**	start in state 0
\$ 0	**aa\$**	shift (and goto state 3)
\$ 0 **a** 3	**a\$**	reduce $A \rightarrow \mathbf{a}$ (goto 2)
\$ 0 A 2	**a\$**	shift (goto 5)
\$ 0 A 2 **a** 5	**\$**	reduce $B \rightarrow \mathbf{a}$ (goto 4)
\$ $\underline{0}$ $\underline{A}$ 2 $\underline{B}$ 4	**\$**	reduce $C \rightarrow AB$ (goto 1)
\$ 0 C 1	**\$**	accept ($S \rightarrow C$)



DFA for Shift/Reduce Decisions

The states of the DFA are used to determine if a handle is on top of the stack

Grammar:

$S \rightarrow C$

$C \rightarrow A\ B$

$A \rightarrow \mathbf{a}$

$B \rightarrow \mathbf{a}$

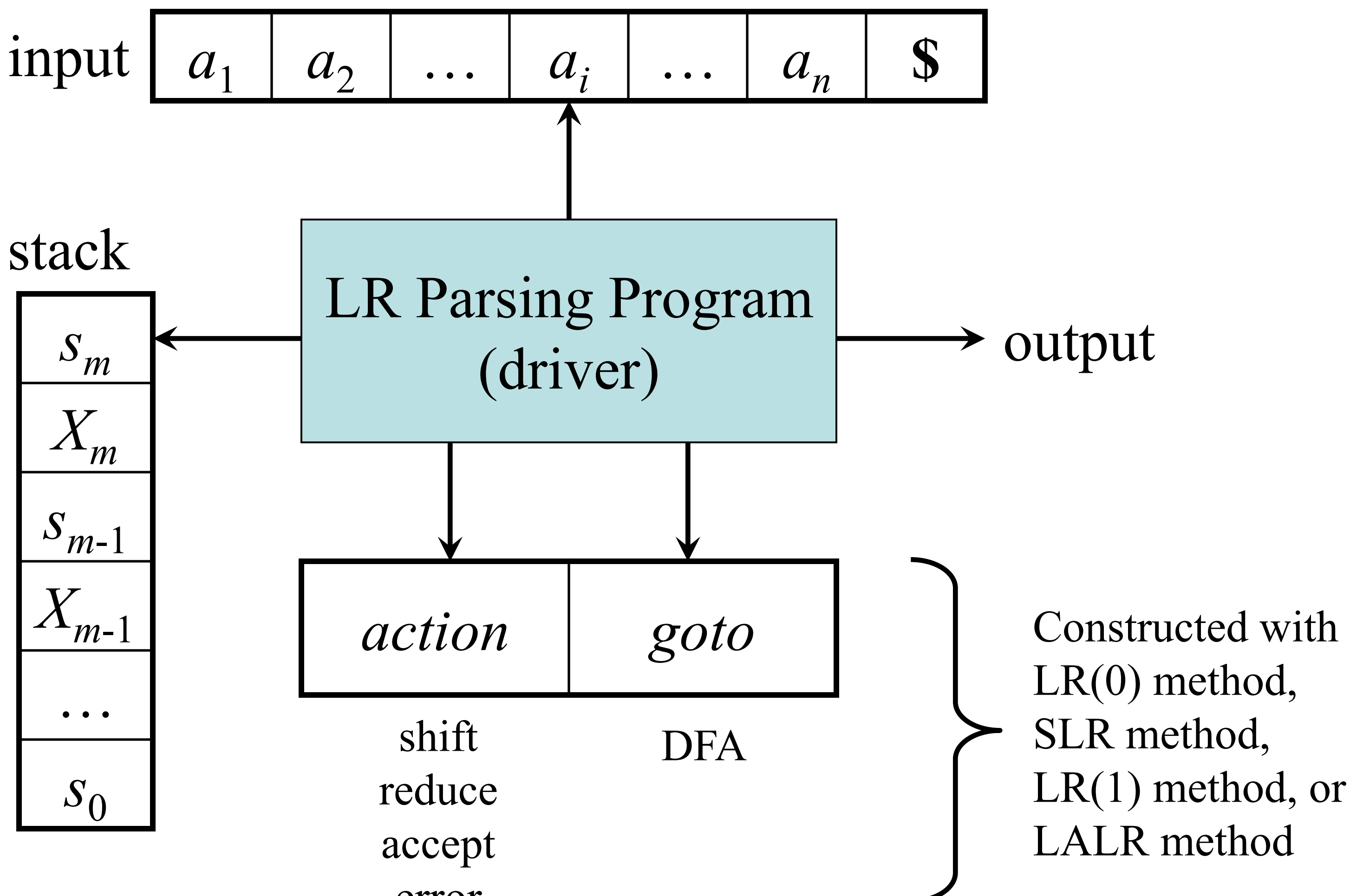
Stack	Input	Action
$ 0	**aa$**	start in state 0
$ 0	**aa$**	shift (and goto state 3)
$ 0 **a** 3	**a$**	reduce $A \rightarrow \mathbf{a}$ (goto 2)
$ 0 *A* 2	**a$**	shift (goto 5)
$ 0 *A* 2 **a** 5	**$**	reduce $B \rightarrow \mathbf{a}$ (goto 4)
$ 0 *A* 2 *B* 4	**$**	reduce $C \rightarrow AB$ (goto 1)
$ 0 *C* $\underline{1}$	$\underline{\mathbf{\$}}$	accept ($S \rightarrow C$)

State I_0:
$S \rightarrow \bullet C$
$C \rightarrow \bullet A\ B$
$A \rightarrow \bullet \mathbf{a}$

goto(I_0, C)

State I_1:
$S \rightarrow C \bullet$

Model of an LR Parser



LR Parsing (Driver)

Configuration (= LR parser state):

$$(\underbrace{s_0\, X_1\, s_1\, X_2\, s_2\, \ldots\, X_m\, s_m,}_{stack}\quad \underbrace{a_i\, a_{i+1}\, \ldots\, a_n\, \$}_{input})$$

If *action*[s_m,a_i] = shift s **then** push a_i, push s, and advance input:

$$(s_0\, X_1\, s_1\, X_2\, s_2\, \ldots\, X_m\, s_m\, a_i\, s,\quad a_{i+1}\, \ldots\, a_n\, \$)$$

If *action*[s_m,a_i] = reduce A $\rightarrow$ β and *goto*[s_{m-r},A] = s with r=|β| **then** pop $2r$ symbols, push A, and push s:

$$(s_0\, X_1\, s_1\, X_2\, s_2\, \ldots\, X_{m-r}\, s_{m-r}\, A\, s,\quad a_i\, a_{i+1}\, \ldots\, a_n\, \$)$$

If *action*[s_m,a_i] = accept **then** stop

If *action*[s_m,a_i] = error **then** attempt recovery

Example LR(0) Parsing Table

State I_0:
$C' \to \bullet C$
$C \to \bullet A\ B$
$A \to \bullet \mathbf{a}$

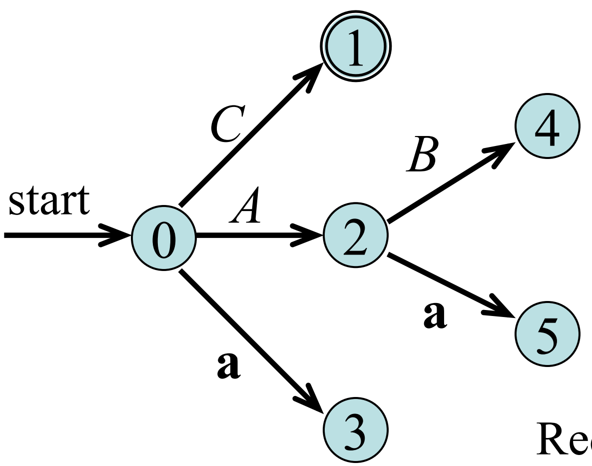
State I_1:
$C' \to C\bullet$

State I_2:
$C \to A\bullet B$
$B \to \bullet \mathbf{a}$

State I_3:
$A \to \mathbf{a}\bullet$

State I_4:
$C \to A\ B\bullet$

State I_5:
$B \to \mathbf{a}\bullet$

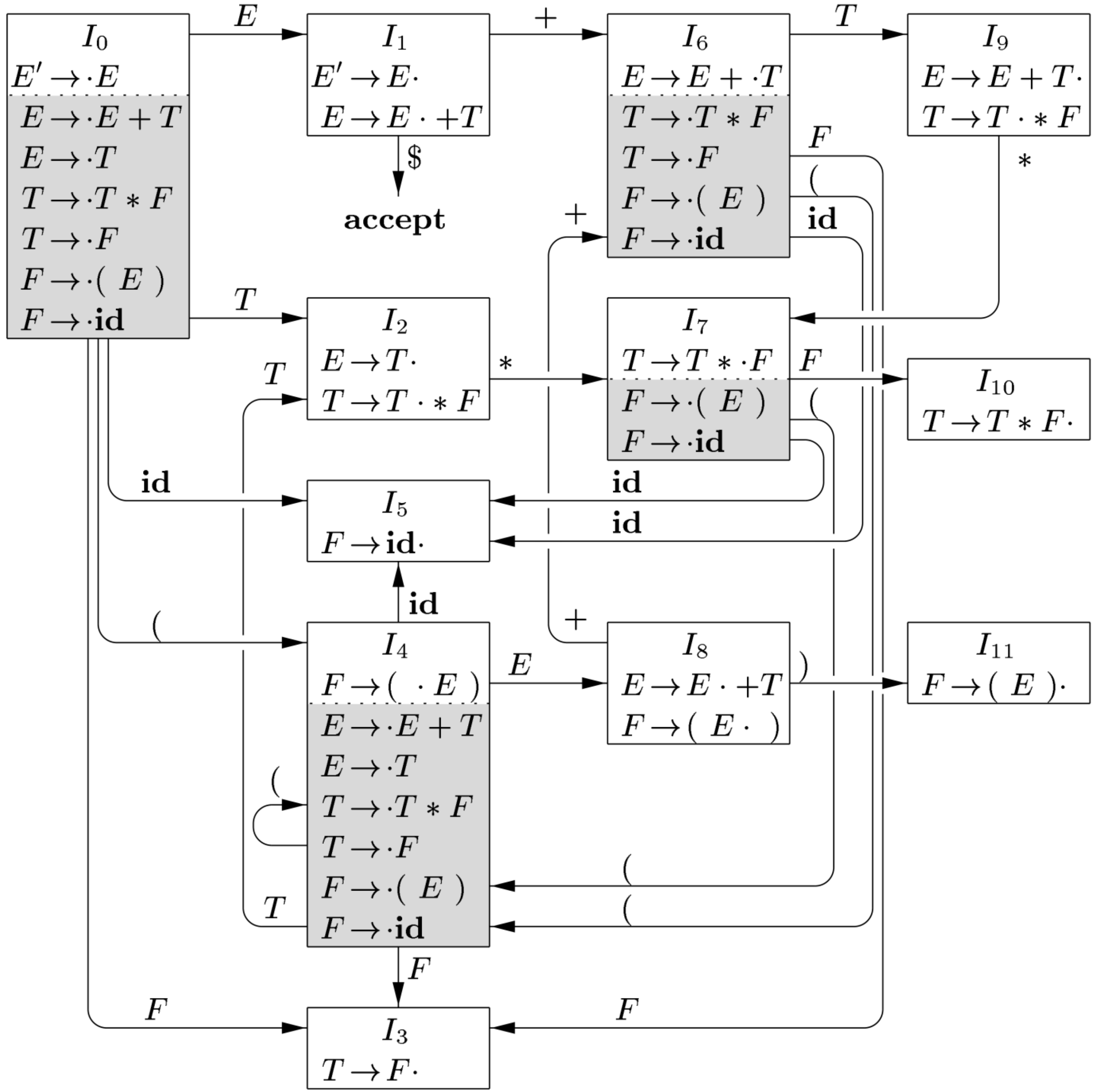


Shift & goto 3

	action		*goto*		
state	**a**	**$**	*C*	*A*	*B*
0	s3		1	2	
1		acc			
2	s5				4
3	r3	r3			
4	r2	r2			
5	r4	r4			

Reduce by production #2

Grammar:
1. $C' \to C$
2. $C \to A\ B$
3. $A \to \mathbf{a}$
4. $B \to \mathbf{a}$



I_0
$E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_1
$E' \to E \cdot$
$E \to E \cdot + T$
$
accept
I_2
$E \to T \cdot$
$T \to T \cdot * F$
I_3
$T \to F \cdot$
I_4
$F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_5
$F \to \mathbf{id} \cdot$
I_6
$E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_7
$T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_8
$E \to E \cdot + T$
$F \to (E \cdot)$
I_9
$E \to E + T \cdot$
$T \to T \cdot * F$
I_{10}
$T \to T * F \cdot$
I_{11}
$F \to (E) \cdot$

Another Example LR Parse Table

Grammar:

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow$ **(** E **)**
6. $F \rightarrow$ **id**

	action						*goto*		
state	**id**	**+**	*****	**(**	**)**	**$**	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Shift & goto 5

Reduce by production #1

Example LR Shift-Reduce Parsing

Grammar:

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow$ **(** E **)**
6. $F \rightarrow$ **id**

Stack	Input	Action
$ 0	**id*id+id$**	shift 5
$ 0 **id** 5	***id+id$**	reduce 6 goto 3
$ 0 *F* 3	***id+id$**	reduce 4 goto 2
$ 0 *T* 2	***id+id$**	shift 7
$ 0 *T* 2 * 7	**id+id$**	shift 5
$ 0 *T* 2 * 7 **id** 5	**+id$**	reduce 6 goto 10
$ 0 *T* 2 * 7 *F* 10	**+id$**	reduce 3 goto 2
$ 0 *T* 2	**+id$**	reduce 2 goto 1
$ 0 *E* 1	**+id$**	shift 6
$ 0 *E* 1 + 6	**id$**	shift 5
$ 0 *E* 1 + 6 **id** 5	**$**	reduce 6 goto 3
$ 0 *E* 1 + 6 *F* 3	**$**	reduce 4 goto 9
$ 0 *E* 1 + 6 *T* 9	**$**	reduce 1 goto 1
$ 0 *E* 1	**$**	accept

SLR Grammars

- SLR (Simple LR): SLR is a simple extension of LR(0) shift-reduce parsing
- SLR eliminates some conflicts by populating the parsing table with reductions $A \rightarrow \alpha$ on symbols in FOLLOW(A)

$S \rightarrow E$
$E \rightarrow \mathbf{id} + E$
$E \rightarrow \mathbf{id}$

State I_0:
$S \rightarrow \bullet E$
$E \rightarrow \bullet\mathbf{id} + E$
$E \rightarrow \bullet\mathbf{id}$

$goto(I_0,\mathbf{id})$

State I_2:
$E \rightarrow \mathbf{id}\bullet + E$
$E \rightarrow \mathbf{id}\bullet$

$goto(I_3,+)$

Shift on +

FOLLOW(E)={**$**}
thus reduce on **$**

SLR Parsing Table

- Reductions do not fill entire rows
- Otherwise the same as LR(0)

1. $S \to E$
2. $E \to \mathbf{id} + E$
3. $E \to \mathbf{id}$

	id	+	**$**	*E*
0	s2			1
1			acc	
2		s3	r3	
3	s2			4
4			r2	

Shift on +

FOLLOW(E)={**$**}
thus reduce on **$**

SLR Parsing

- An LR(0) state is a set of LR(0) items
- An LR(0) item is a production with a • (dot) in the right-hand side
- Build the LR(0) DFA by
  - *Closure operation* to construct LR(0) items
  - *Goto operation* to determine transitions
- Construct the SLR parsing table from the DFA
- LR parser program uses the SLR parsing table to determine shift/reduce operations

Constructing SLR Parsing Tables

1. Augment the grammar with $S' \to S$
2. Construct the set $C=\{I_0,I_1,\ldots,I_n\}$ of $LR(0)$ *items*
3. If $[A\to\alpha\bullet a\beta] \in I_i$ and $goto(I_i,a)=I_j$ then set $action[i,a]=$shift j
4. If $[A\to\alpha\bullet] \in I_i$ then set $action[i,a]=$reduce $\mathrm{A}\to\alpha$ for all $a \in$ FOLLOW(A) (apply only if $A\neq S'$)
5. If $[S' \to S\bullet]$ is in I_i then set $action[i,\$]=$accept
6. If $goto(I_i,A)=I_j$ then set $goto[i,A]=j$
7. Repeat 3-6 until no more entries added
8. The initial state i is the I_i holding item $[S' \to\bullet S]$

LR(0) Items of a Grammar

- An *LR*(0) *item* of a grammar G is a production of G with a • at some position of the right-hand side
- Thus, a production

  $A \rightarrow X\ Y\ Z$

  has four items:

  $[A \rightarrow \bullet\ X\ Y\ Z]$
  $[A \rightarrow X \bullet Y\ Z]$
  $[A \rightarrow X\ Y \bullet Z]$
  $[A \rightarrow X\ Y\ Z \bullet]$
- Note that production $A \rightarrow \varepsilon$ has one item $[A \rightarrow \bullet]$

Constructing the set of LR(0) Items of a Grammar

1. The grammar is augmented with a new start symbol S' and production $S' \rightarrow S$
2. Initially, set $C = closure(\{[S' \rightarrow \bullet S]\})$ (this is the start state of the DFA)
3. For each set of items $I \in C$ and each grammar symbol $X \in (N \cup T)$ such that $goto(I,X) \notin C$ and $goto(I,X) \neq \emptyset$, add the set of items $goto(I,X)$ to C
4. Repeat 3 until no more sets can be added to C

The Closure Operation for LR(0) Items

1. Start with *closure*(I) = I
2. If $[A \to \alpha \bullet B\beta] \in$ *closure*(I) then for each production $B \to \gamma$ in the grammar, add the item $[B \to \bullet\gamma]$ to I if not already in I
3. Repeat 2 until no new items can be added

The Closure Operation (Example)

$closure(\{[E' \to \bullet E]\}) =$

$\{ [E' \to \bullet E] \}$ ⇒ $\{ [E' \to \bullet E]$, $[E \to \bullet E + T]$, $[E \to \bullet T] \}$ ⇒ $\{ [E' \to \bullet E]$, $[E \to \bullet E + T]$, $[E \to \bullet T]$, $[T \to \bullet T * F]$, $[T \to \bullet F] \}$ ⇒ $\{ [E' \to \bullet E]$, $[E \to \bullet E + T]$, $[E \to \bullet T]$, $[T \to \bullet T * F]$, $[T \to \bullet F]$, $[F \to \bullet \mathbf{(}\ E\ \mathbf{)}]$, $[F \to \bullet\ \mathbf{id}] \}$

Add $[E\to\bullet\gamma]$

Add $[T\to\bullet\gamma]$

Add $[F\to\bullet\gamma]$

Grammar:

$E \to E + T \mid T$

$T \to T * F \mid F$

$F \to \mathbf{(}\ E\ \mathbf{)}$

$F \to \mathbf{id}$

The Goto Operation for LR(0) Items

1. For each item $[A \to \alpha \bullet X\beta] \in I$, add the set of items *closure*($\{[A \to \alpha X \bullet \beta]\}$) to *goto*($I$,$X$) if not already there
2. Repeat step 1 until no more items can be added to *goto*(I,X)
3. Intuitively, *goto*(I,X) is the set of items that are valid for the viable prefix γX when I is the set of items that are valid for γ

The Goto Operation (Example 1)

Suppose $I = \{\ [E' \to \bullet\ E]$
$[E \to \bullet\ E + T]$
$[E \to \bullet\ T]$
$[T \to \bullet\ T * F]$
$[T \to \bullet\ F]$
$[F \to \bullet\ \mathbf{(}\ E\ \mathbf{)}]$
$[F \to \bullet\ \mathbf{id}]\ \}$

Then $goto(I,E)$
$= closure(\{[E' \to E\ \bullet,\ E \to E\ \bullet + T]\})$
$= \{\ [E' \to E\ \bullet]$
$[E \to E\ \bullet + T]\ \}$

Grammar:
$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to \mathbf{(}\ E\ \mathbf{)}$
$F \to \mathbf{id}$

The Goto Operation (Example 2)

Suppose $I = \{\ [E' \to E\ \bullet],\ [E \to E\ \bullet + T]\ \}$

Then $goto(I,+) = closure(\{[E \to E + \bullet\ T]\}) = \{\ [E \to E + \bullet\ T]$
$[T \to \bullet\ T * F]$
$[T \to \bullet\ F]$
$[F \to \bullet\ \mathbf{(}\ E\ \mathbf{)}]$
$[F \to \bullet\ \mathbf{id}]\ \}$

Grammar:

$E \to E + T \mid T$

$T \to T * F \mid F$

$F \to \mathbf{(}\ E\ \mathbf{)}$

$F \to \mathbf{id}$

Example Grammar and LR(0) Items

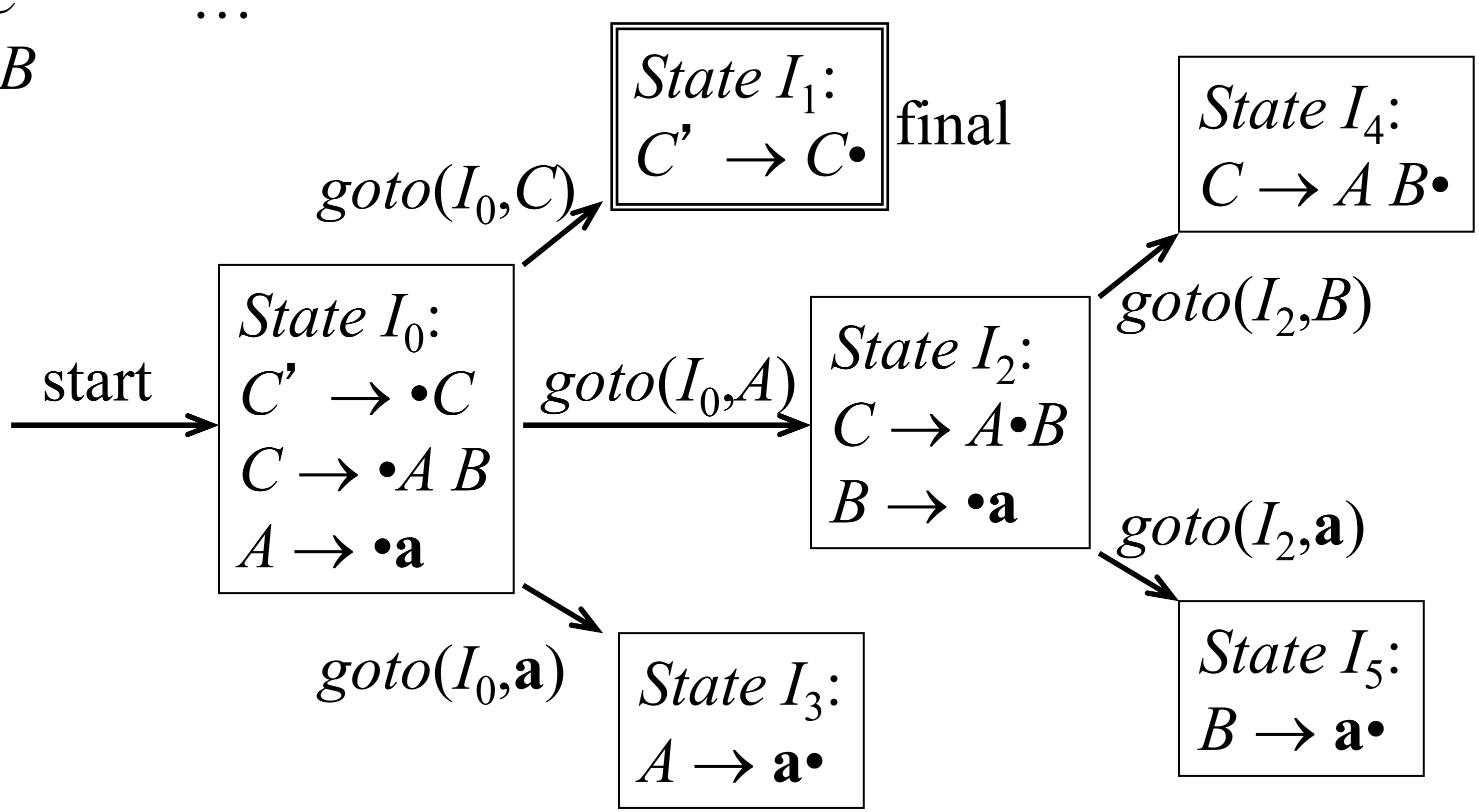
Augmented grammar:

1. $C' \to C$
2. $C \to A\ B$
3. $A \to \mathbf{a}$
4. $B \to \mathbf{a}$

$I_0 = closure(\{[C' \to \bullet C]\})$

$I_1 = \text{goto}(I_0,C) = closure(\{[C' \to C\bullet]\})$

…



Constructing SLR Parsing Tables

1. Augment the grammar with $S' \rightarrow S$
2. Construct the set $C=\{I_0,I_1,\ldots,I_n\}$ of $LR(0)$ *items*
3. If $[A\rightarrow\alpha\bullet a\beta] \in I_i$ and $goto(I_i,a)=I_j$ then set $action[i,a]$=shift j
4. If $[A\rightarrow\alpha\bullet] \in I_i$ then set $action[i,a]$=reduce $\mathrm{A}\rightarrow\alpha$ for all $a \in \mathrm{FOLLOW}(A)$ (apply only if $A\neq S'$)
5. If $[S' \rightarrow S\bullet]$ is in I_i then set $action[i,\$]$=accept
6. If $goto(I_i,A)=I_j$ then set $goto[i,A]=j$
7. Repeat 3-6 until no more entries added
8. The initial state i is the I_i holding item $[S' \rightarrow\bullet S]$

Example SLR Parsing Table

State I_0:
$C' \to \bullet C$
$C \to \bullet A\ B$
$A \to \bullet \mathbf{a}$

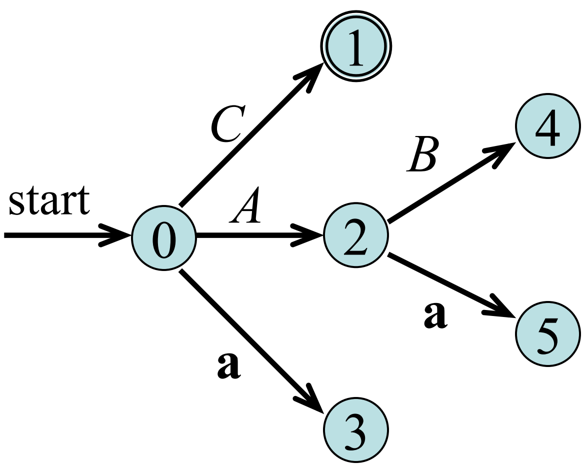
State I_1:
$C' \to C\bullet$

State I_2:
$C \to A\bullet B$
$B \to \bullet \mathbf{a}$

State I_3:
$A \to \mathbf{a}\bullet$

State I_4:
$C \to A\ B\bullet$

State I_5:
$B \to \mathbf{a}\bullet$



state	*action* **a**	*action* **$**	*goto* *C*	*goto* *A*	*goto* *B*
0	s3		1	2	
1		acc			
2	s5				4
3	r3				
4		r2			
5		r4			

Grammar:
1. $C' \to C$
2. $C \to A\ B$
3. $A \to \mathbf{a}$
4. $B \to \mathbf{a}$

FOLLOW(A) = {**a**}

FOLLOW(C) = {**$**}

FOLLOW(B) = {**$**}

SLR, Ambiguity, and Conflicts

- SLR grammars are unambiguous
- But **not** every unambiguous grammar is SLR
- Consider for example the unambiguous grammar

$$1.\ S \to L = R \qquad 2.\ S \to R$$
$$3.\ L \to *\,R \qquad 4.\ L \to \mathbf{id}$$
$$5.\ R \to L$$

I_0:
$S' \to \bullet S$
$S \to \bullet L{=}R$
$S \to \bullet R$
$L \to \bullet{*}R$
$L \to \bullet\mathbf{id}$
$R \to \bullet L$

I_1:
$S' \to S\bullet$

I_2:
$S \to L\bullet{=}R$
$R \to L\bullet$

I_3:
$S \to R\bullet$

I_4:
$L \to {*}\bullet R$
$R \to \bullet L$
$L \to \bullet{*}R$
$L \to \bullet\mathbf{id}$

I_5:
$L \to \mathbf{id}\bullet$

I_6:
$S \to L{=}\bullet R$
$R \to \bullet L$
$L \to \bullet{*}R$
$L \to \bullet\mathbf{id}$

I_7:
$L \to {*}R\bullet$

I_8:
$R \to L\bullet$

I_9:
$S \to L{=}R\bullet$

$action[2,=]=\mathrm{s}6$
$action[2,=]=\mathrm{r}5$

no

Conflict: has no SLR parsing table!

$S \rightarrow SS+|SS*|a$

SLR Parse Table??

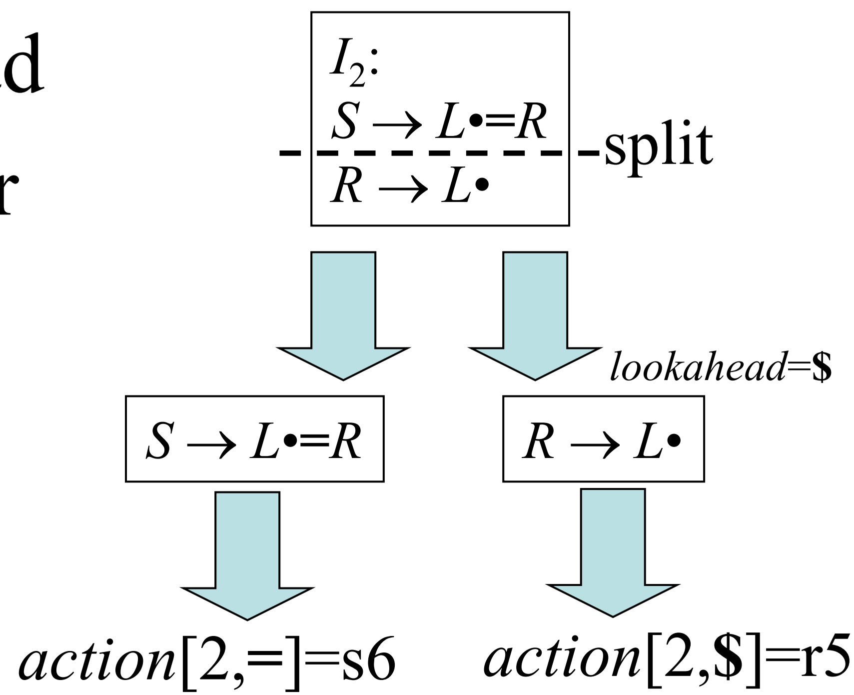
LR(1) Grammars

- SLR too simple
- LR(1) parsing uses lookahead to avoid unnecessary conflicts in parsing table
- LR(1) item = LR(0) item + lookahead

LR(0) item:	LR(1) item:
$[A\rightarrow\alpha\bullet\beta]$	$[A\rightarrow\alpha\bullet\beta,\ a]$

SLR Versus LR(1)

- Split the SLR states by adding LR(1) lookahead
- Unambiguous grammar
  1. $S \rightarrow L = R$
  2. $S \rightarrow R$
  3. $L \rightarrow * R$
  4. $L \rightarrow \textbf{id}$
  5. $R \rightarrow L$



Should not reduce on =, because no right-sentential form begins with R=

LR(1) Items

- An *LR*(1) *item*
  $[A\rightarrow\alpha\bullet\beta,\ a]$
  contains a *lookahead* terminal *a*, meaning α already on top of the stack, expect to parse β*a*
- For items of the form
  $[A\rightarrow\alpha\bullet,\ a]$
  the lookahead *a* is used to reduce $A\rightarrow\alpha$ only if the next lookahead of the input is *a*
- For items of the form
  $[A\rightarrow\alpha\bullet\beta,\ a]$
  with $\beta\neq\varepsilon$ the lookahead has no effect

The Closure Operation for LR(1) Items

1. Start with *closure*(I) = I
2. If $[A \rightarrow \alpha \bullet B\beta, a] \in$ *closure*(I) then for each production $B \rightarrow \gamma$ in the grammar and each terminal $b \in$ FIRST(βa), add the item $[B \rightarrow \bullet\gamma, b]$ to I if not already in I
3. Repeat 2 until no new items can be added

The Goto Operation for LR(1) Items

1. For each item $[A \rightarrow \alpha \bullet X\beta, a] \in I$, add the set of items *closure*$(\{[A \rightarrow \alpha X \bullet \beta, a]\})$ to *goto*(I,X) if not already there
2. Repeat step 1 until no more items can be added to *goto*(I,X)

Constructing the set of LR(1) Items of a Grammar

1. Augment the grammar with a new start symbol S' and production $S' \rightarrow S$
2. Initially, set $C = closure(\{[S' \rightarrow \bullet S, \mathbf{\$}]\})$ (this is the start state of the DFA)
3. For each set of items $I \in C$ and each grammar symbol $X \in (N \cup T)$ such that $goto(I,X) \notin C$ and $goto(I,X) \neq \varnothing$, add the set of items $goto(I,X)$ to C
4. Repeat 3 until no more sets can be added to C

Example Grammar and LR(1) Items

- Unambiguous LR(1) grammar:

  $S \rightarrow L = R$

  $S \rightarrow R$

  $L \rightarrow * R$

  $L \rightarrow \textbf{id}$

  $R \rightarrow L$

- Augment with $S' \rightarrow S$
- LR(1) items (next slide)

I_0: $[S' \to \bullet S,\ \$]$ $\text{goto}(I_0,S)=I_1$
$[S \to \bullet L{=}R,\ \$]$ $\text{goto}(I_0,L)=I_2$
$[S \to \bullet R,\ \$]$ $\text{goto}(I_0,R)=I_3$
$[L \to \bullet{*}R,\ {=}/\$]$ $\text{goto}(I_0,*)=I_4$
$[L \to \bullet\mathbf{id},\ {=}/\$]$ $\text{goto}(I_0,\mathbf{id})=I_5$
$[R \to \bullet L,\ \$]$ $\text{goto}(I_0,L)=I_2$

I_1: $[S' \to S\bullet,\ \$]$

I_2: $[S \to L\bullet{=}R,\ \$]$ $\text{goto}(I_0,=)=I_6$
$[R \to L\bullet,\ \$]$

I_3: $[S \to R\bullet,\ \$]$

I_4: $[L \to {*}\bullet R,\ {=}/\$]$ $\text{goto}(I_4,R)=I_7$
$[R \to \bullet L,\ {=}/\$]$ $\text{goto}(I_4,L)=I_8$
$[L \to \bullet{*}R,\ {=}/\$]$ $\text{goto}(I_4,*)=I_4$
$[L \to \bullet\mathbf{id},\ {=}/\$]$ $\text{goto}(I_4,\mathbf{id})=I_5$

I_5: $[L \to \mathbf{id}\bullet,\ {=}/\$]$

I_6: $[S \to L{=}\bullet R,\ \$]$ $\text{goto}(I_6,R)=I_9$
$[R \to \bullet L,\ \$]$ $\text{goto}(I_6,L)=I_{10}$
$[L \to \bullet{*}R,\ \$]$ $\text{goto}(I_6,*)=I_{11}$
$[L \to \bullet\mathbf{id},\ \$]$ $\text{goto}(I_6,\mathbf{id})=I_{12}$

I_7: $[L \to {*}R\bullet,\ {=}/\$]$

I_8: $[R \to L\bullet,\ {=}/\$]$

I_9: $[S \to L{=}R\bullet,\ \$]$

I_{10}: $[R \to L\bullet,\ \$]$

I_{11}: $[L \to {*}\bullet R,\ \$]$ $\text{goto}(I_{11},R)=I_{13}$
$[R \to \bullet L,\ \$]$ $\text{goto}(I_{11},L)=I_{10}$
$[L \to \bullet{*}R,\ \$]$ $\text{goto}(I_{11},*)=I_{11}$
$[L \to \bullet\mathbf{id},\ \$]$ $\text{goto}(I_{11},\mathbf{id})=I_{12}$

I_{12}: $[L \to \mathbf{id}\bullet,\ \$]$

I_{13}: $[L \to {*}R\bullet,\ \$]$

Constructing Canonical LR(1) Parsing Tables

1. Augment the grammar with $S' \to S$
2. Construct the set $C=\{I_0, I_1, \ldots, I_n\}$ of LR(1) items
3. If $[A \to \alpha \bullet a\beta,\ b] \in I_i$ and $goto(I_i, a)=I_j$ then set $action[i,a]$=shift j
4. If $[A \to \alpha \bullet,\ a] \in I_i$ then set $action[i,a]$=reduce $A \to \alpha$ (apply only if $A \neq S'$)
5. If $[S' \to S \bullet,\ \$]$ is in I_i then set $action[i,\$]$=accept
6. If $goto(I_i, A)=I_j$ then set $goto[i,A]=j$
7. Repeat 3-6 until no more entries added
8. The initial state i is the I_i holding item $[S' \to \bullet S, \$]$

Example LR(1) Parsing Table

Grammar:

1. $S' \to S$
2. $S \to L = R$
3. $S \to R$
4. $L \to$ ***** R
5. $L \to$ **id**
6. $R \to L$

	id	*****	**=**	**$**	*S*	*L*	*R*
0	s5	s4			1	2	3
1				acc			
2			s6	r6			
3				r3			
4	s5	s4				8	7
5			r5	r5			
6	s12	s11				10	9
7			r4	r4			
8			r6	r6			
9				r2			
10				r6			
11	s12	s11				10	13
12				r5			
13				r4			

LALR Parsing

- LR(1) parsing tables have many states
- LALR parsing (Look-Ahead LR) merges two or more LR(1) state into one state to reduce table size
- Less powerful than LR(1)
  - Will not introduce shift-reduce conflicts, because shifts do not use lookaheads
  - May introduce reduce-reduce conflicts, but seldom do so for grammars of programming languages

Constructing LALR Parsing Tables

1. Construct sets of LR(1) items
2. Combine LR(1) sets with sets of items that share the same first part

$$
\left.\begin{array}{lll}
I_4: & [L \to *\bullet R, & =/\$] \\
 & [R \to \bullet L, & =/\$] \\
 & [L \to \bullet * R, & =/\$] \\
 & [L \to \bullet \mathbf{id}, & =/\$] \\
 & & \\
I_{11}: & [L \to *\bullet R, & \$] \\
 & [R \to \bullet L, & \$] \\
 & [L \to \bullet * R, & \$] \\
 & [L \to \bullet \mathbf{id}, & \$]
\end{array}\right\}
\quad
\begin{array}{ll}
[L \to *\bullet R, & =/\$] \\
[R \to \bullet L, & =/\$] \\
[L \to \bullet * R, & =/\$] \\
[L \to \bullet \mathbf{id}, & =/\$]
\end{array}
$$

Shorthand for two items in the same set

Example Grammar and LALR Parsing Table

- Unambiguous LR(1) grammar:

$$S \to L = R$$
$$\quad \mid R$$
$$L \to * R$$
$$\quad \mid \textbf{id}$$
$$R \to L$$

- Augment with $S' \to S$
- LALR items (next slide)

I_0: $[S' \to \bullet S,\ \$]$ goto$(I_0,S)=I_1$
$[S \to \bullet L{=}R,\ \$]$ goto$(I_0,L)=I_2$
$[S \to \bullet R,\ \$]$ goto$(I_0,R)=I_3$
$[L \to \bullet{*}R,\ {=}/\$]$ goto$(I_0,*)=I_4$
$[L \to \bullet\mathbf{id},\ {=}/\$]$ goto$(I_0,\mathbf{id})=I_5$
$[R \to \bullet L,\ \$]$ goto$(I_0,L)=I_2$

I_1: $[S' \to S\bullet,\ \$]$

I_2: $[S \to L\bullet{=}R,\ \$]$ goto$(I_0,=)=I_6$
$[R \to L\bullet,\ \$]$

I_3: $[S \to R\bullet,\ \$]$

I_4: $[L \to {*}\bullet R,\ {=}/\$]$ goto$(I_4,R)=I_7$
$[R \to \bullet L,\ {=}/\$]$ goto$(I_4,L)=I_9$
$[L \to \bullet{*}R,\ {=}/\$]$ goto$(I_4,*)=I_4$
$[L \to \bullet\mathbf{id},\ {=}/\$]$ goto$(I_4,\mathbf{id})=I_5$

I_5: $[L \to \mathbf{id}\bullet,\ {=}/\$]$

I_6: $[S \to L{=}\bullet R,\ \$]$ goto$(I_6,R)=I_8$
$[R \to \bullet L,\ \$]$ goto$(I_6,L)=I_9$
$[L \to \bullet{*}R,\ \$]$ goto$(I_6,*)=I_4$
$[L \to \bullet\mathbf{id},\ \$]$ goto$(I_6,\mathbf{id})=I_5$

I_7: $[L \to {*}R\bullet,\ {=}/\$]$

I_8: $[S \to L{=}R\bullet,\ \$]$

I_9: $[R \to L\bullet,\ {=}/\$]$

Shorthand for two items

$[R \to L\bullet,\ =]$
$[R \to L\bullet,\ \$]$

Example LALR Parsing Table

Grammar:

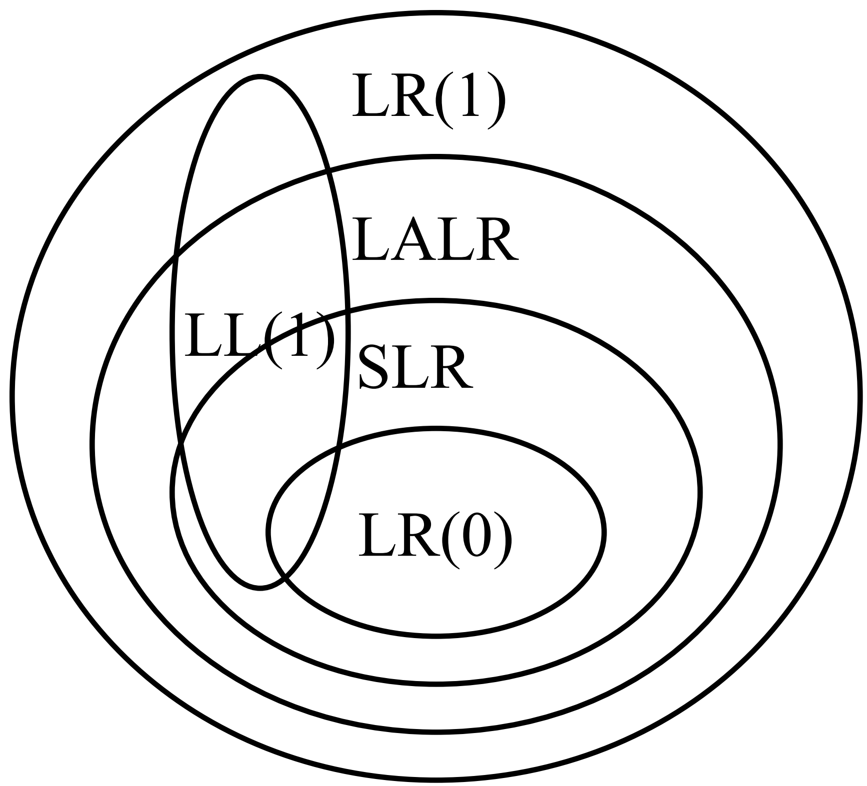
1. $S' \rightarrow S$
2. $S \rightarrow L = R$
3. $S \rightarrow R$
4. $L \rightarrow$ ***** R
5. $L \rightarrow$ **id**
6. $R \rightarrow L$

	id	*****	**=**	**$**	*S*	*L*	*R*
0	s5	s4			1	2	3
1				acc			
2			s6	r6			
3				r3			
4	s5	s4				9	7
5			r5	r5			
6	s5	s4				9	8
7			r4	r4			
8				r2			
9			r6	r6			

LL, SLR, LR, LALR Summary

- LL parse tables computed using FIRST/FOLLOW
  - Nonterminals × terminals → productions
  - Computed using FIRST/FOLLOW
- LR parsing tables computed using closure/goto
  - LR states × terminals → shift/reduce actions
  - LR states × nonterminals → goto state transitions
- A grammar is
  - LL(1) if its LL(1) parse table has no conflicts
  - SLR if its SLR parse table has no conflicts
  - LALR if its LALR parse table has no conflicts
  - LR(1) if its LR(1) parse table has no conflicts

LL, SLR, LR, LALR Grammars



Dealing with Ambiguous Grammars

1. $S' \to E$
2. $E \to E + E$
3. $E \to$ **id**

	id	+	$	E
0	s2			1
1		s3	acc	
2		r3	r3	
3	s2			4
4		s3/r2	r2	

stack	input
$ 0	**id+id+id$**
…	…
$ 0 E 1 + 3 E 4	**+id$**

Shift/reduce conflict:
action[4,+] = shift 4
action[4,+] = reduce $E \to E + E$

When shifting on +:
yields right associativity
id+(id+id)

When reducing on +:
yields left associativity
(id+id)+id

Using Associativity and Precedence to Resolve Conflicts

- Left-associative operators: reduce
- Right-associative operators: shift
- Operator of higher precedence on stack: reduce
- Operator of lower precedence on stack: shift

$S' \to E$
$E \to E + E$
$E \to E * E$
$E \to \mathbf{id}$

stack	input	
$ 0	**id*id+id$**	
…	…	
$ 0 E 1 * 3 E 5	**+id$**	reduce $E \to E * E$

Error Detection in LR Parsing

- Canonical LR parser uses full LR(1) parse tables and will never make a single reduction before recognizing the error when a syntax error occurs on the input
- SLR and LALR may still reduce when a syntax error occurs on the input, but will never shift the erroneous input symbol

Error Recovery in LR Parsing

- Panic mode
  - Pop until state with a goto on a nonterminal *A* is found, (where *A* represents a major programming construct), push *A*
  - Discard input symbols until one is found in the FOLLOW set of *A*
- Phrase-level recovery
  - Implement error routines for every error entry in table
- Error productions
  - Pop until state has error production, then shift on stack
  - Discard input until symbol is encountered that allows parsing to continue