Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-15/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 9

- Bottom-Up Parsing
- Shift-Reduce
- LR(0) automata and SLR parsing
- LR(1), LALR parsing

Shift-Reduce Parsing

Grammar:

$S \rightarrow$ **a** A B **e**

$A \rightarrow A$ **b c** | **b**

$B \rightarrow$ **d**

Reducing a sentence:

a <u>b</u> b c d e

a <u>*A* **b c</u> d e**

a *A* <u>d**</u> e**

<u>a *A* *B* e</u>

S

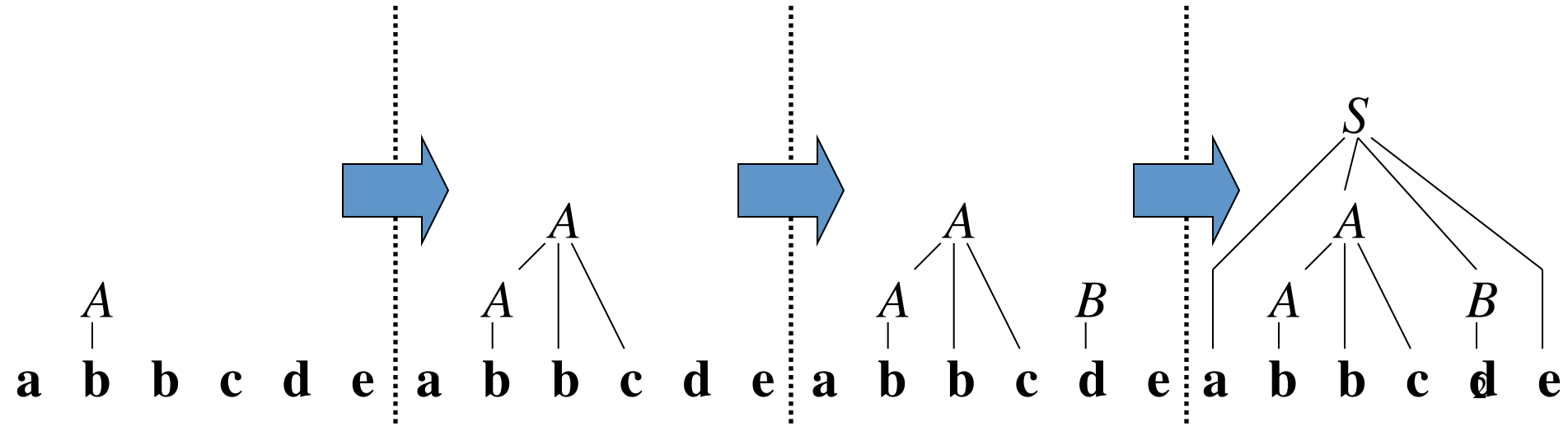
Shift-reduce corresponds to a rightmost derivation:

$S \Rightarrow_{rm}$ **a** A B **e**

$\Rightarrow_{rm}$ **a** A **d e**

$\Rightarrow_{rm}$ **a** A **b c d e**

$\Rightarrow_{rm}$ **a b b c d e**



Handles

- A *handle* in a *right-sentential form* α is a production A -> β and a position of β in α such that replacing β with A we get the previous right-sentential form in the rightmost derivation
- Note: if grammar is unambiguous, the handle is unique

Grammar:

$S \rightarrow$ **a** *A B* **e**

$A \rightarrow A$ **b c** | **b**

$B \rightarrow$ **d**

a b b c d e

a *A* **b c d e**

a *A* **d e**

a *A B* **e**

S

Handle

a b b c d e

a *A* **b c d e**

a *A A* **e**

... ?

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

Stack Implementation of Shift-Reduce Parsing

- Stack+input always is a right-sentential form
- Each step either shifts next char, or reduces handle on top of stack

Grammar:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow \mathbf{id}$

Stack	Input	Action
$	id+id*id$	shift
$id	+id*id$	reduce $E \rightarrow \mathbf{id}$
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce $E \rightarrow \mathbf{id}$
$E+E	*id$	shift (or reduce?)
$E+E*	id$	shift
$E+E*id	$	reduce $E \rightarrow \mathbf{id}$
$E+E*E	$	reduce $E \rightarrow E * E$
$E+E	$	reduce $E \rightarrow E + E$
$E	$	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 → **if** *E* **then** *S*
| **if** *E* **then** *S* **else** *S*
| **other**

Stack	Input	Action
$…	…**$**	…
$…**if** *E* **then** *S*	**else**…**$**	shift or reduce?

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

Shift-Reduce Parsing: Reduce-Reduce Conflicts

Grammar:

$C \rightarrow A\ B$

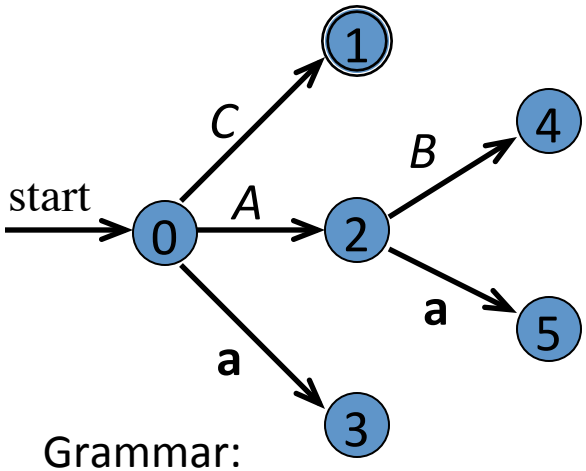
$A \rightarrow \mathbf{a}$

$B \rightarrow \mathbf{a}$

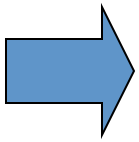
Stack	Input	Action
$	**aa$**	shift
$a	**a$**	reduce $A \rightarrow \mathbf{a}$ or $B \rightarrow \mathbf{a}$?

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

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



Grammar:
$S \rightarrow C$
$C \rightarrow A\ B$
$A \rightarrow \mathbf{a}$
$B \rightarrow \mathbf{a}$



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

$goto(I_0, C)$

State I_1:
$S \rightarrow C\bullet$

$goto(I_0, A)$

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

$goto(I_2, B)$

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

$goto(I_2, \mathbf{a})$

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

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

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

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

DFA for Shift/Reduce Decisions

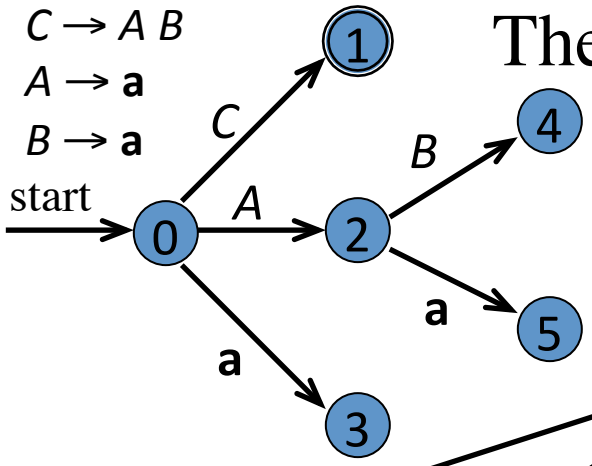
Grammar:

$S \rightarrow C$

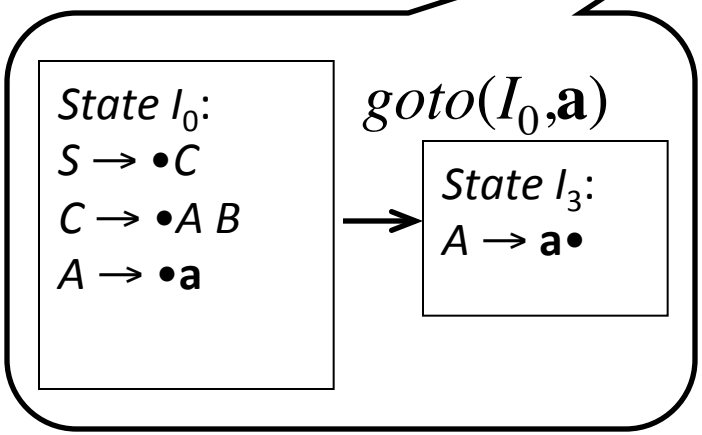
$C \rightarrow A\ B$

$A \rightarrow \mathbf{a}$

$B \rightarrow \mathbf{a}$



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



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* 1	$	accept ($S \rightarrow C$)

DFA for Shift/Reduce Decisions

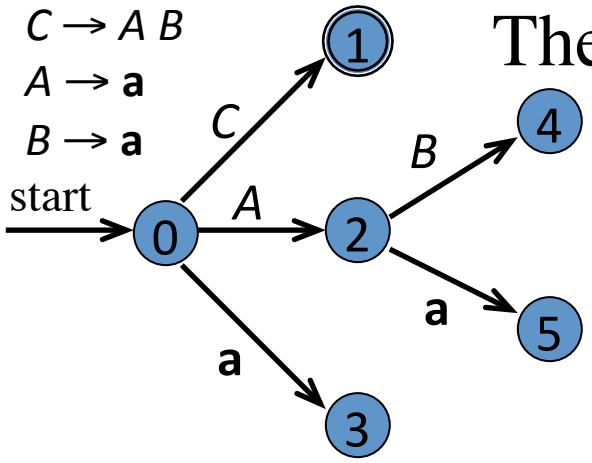
Grammar:

$S \rightarrow C$

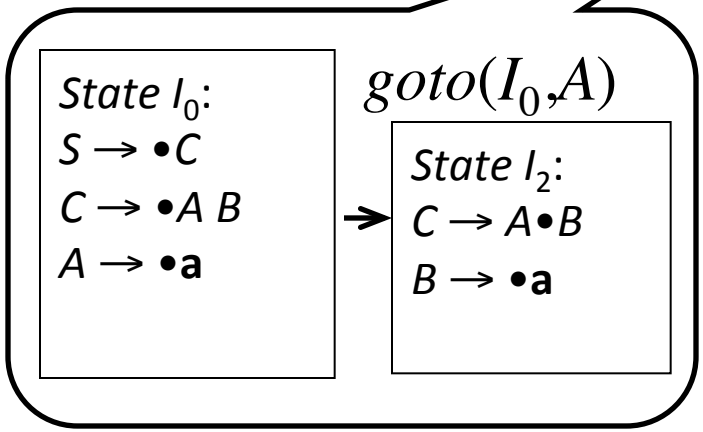
$C \rightarrow A\ B$

$A \rightarrow \mathbf{a}$

$B \rightarrow \mathbf{a}$



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



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* 1	$	accept ($S \rightarrow C$)

DFA for Shift/Reduce Decisions

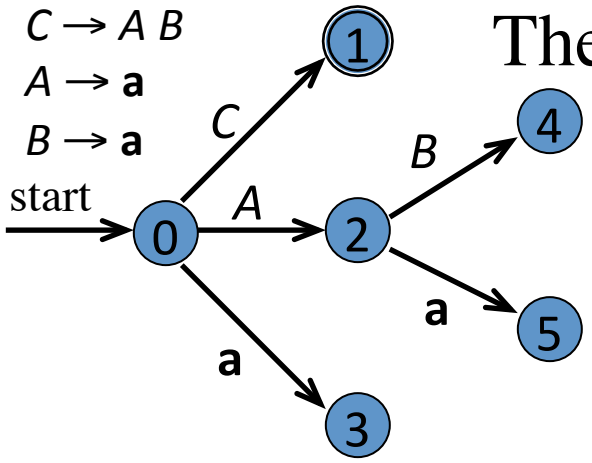
Grammar:

$S \rightarrow C$

$C \rightarrow A\ B$

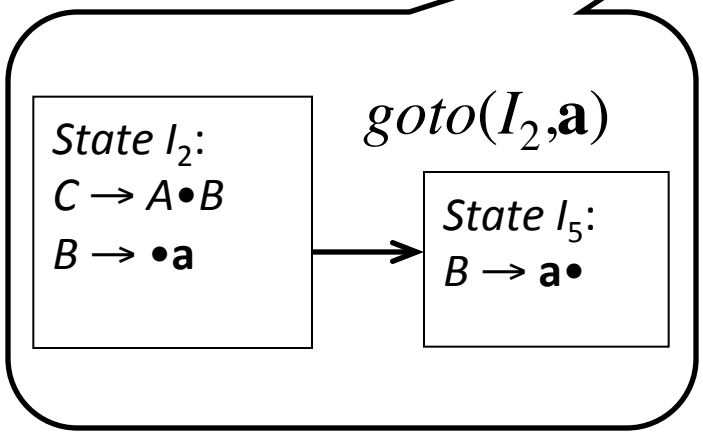
$A \rightarrow \mathbf{a}$

$B \rightarrow \mathbf{a}$



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

Stack	Input	Action
$ 0	**aa$**	start in state 0
$ 0	**aa$**	shift (and goto state 3)
$ 0 **a** 3	**a$**	reduce $A \rightarrow$ **a** (goto 2)
$ 0 *A* 2	**a$**	shift (goto 5)
$ 0 *A* 2 **a** 5	$	reduce $B \rightarrow$ **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

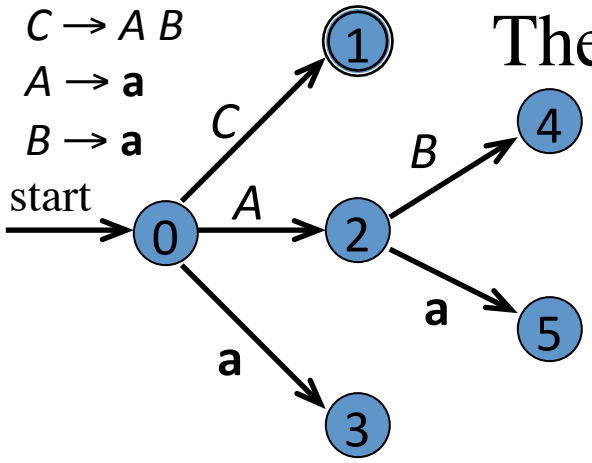
Grammar:

$S \rightarrow C$

$C \rightarrow A\ B$

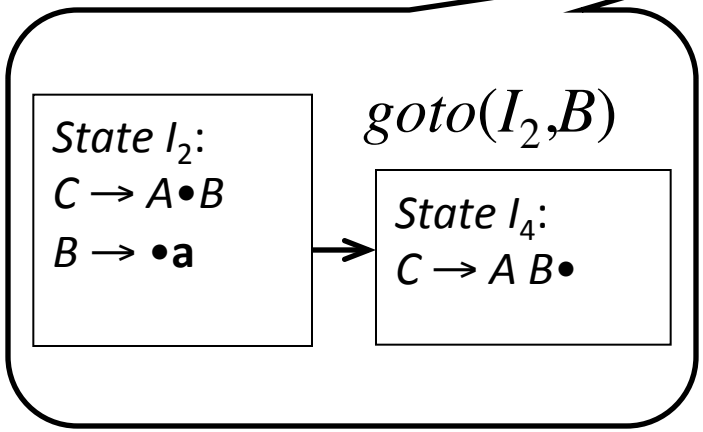
$A \rightarrow \mathbf{a}$

$B \rightarrow \mathbf{a}$



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

Stack	Input	Action
$ 0	**aa$**	start in state 0
$ 0	**aa$**	shift (and goto state 3)
$ 0 **a** 3	**a$**	reduce $A \rightarrow$ **a** (goto 2)
$ 0 *A* 2	**a$**	shift (goto 5)
$ 0 *A* 2 **a** 5	$	reduce $B \rightarrow$ **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

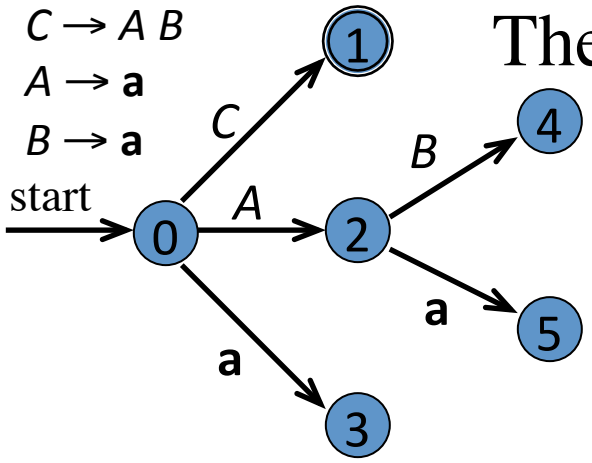
Grammar:

$S \rightarrow C$

$C \rightarrow A\ B$

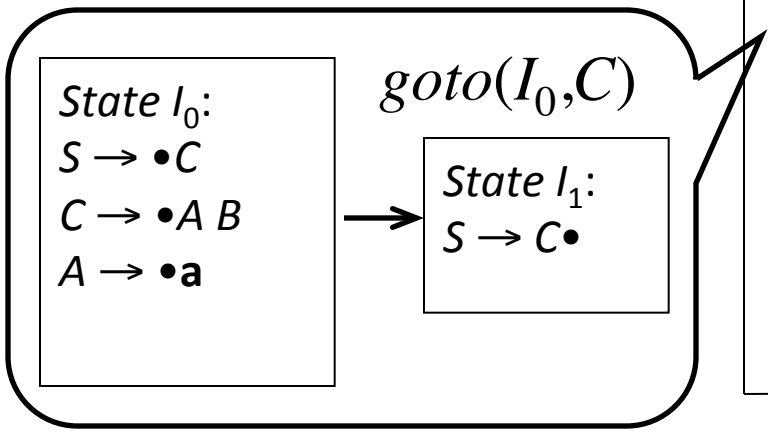
$A \rightarrow \mathbf{a}$

$B \rightarrow \mathbf{a}$



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

Stack	Input	Action
$ 0	aa$	start in state 0
$ 0	aa$	shift (and goto state 3)
$ 0 **a** 3	a$	reduce $A \rightarrow$ **a** (goto 2)
$ 0 *A* 2	a$	shift (goto 5)
$ 0 *A* 2 **a** 5	$	reduce $B \rightarrow$ **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

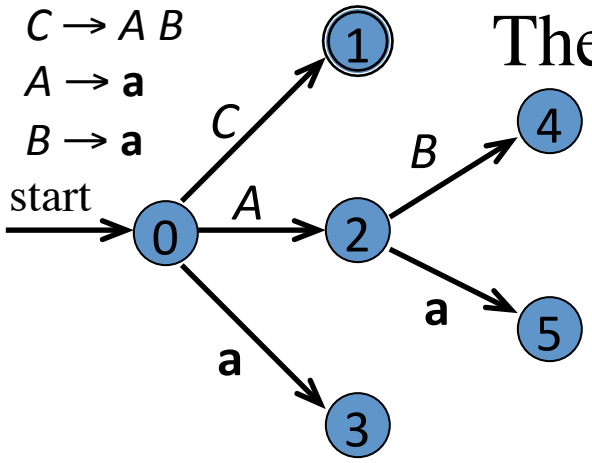
Grammar:

$S \rightarrow C$

$C \rightarrow A\ B$

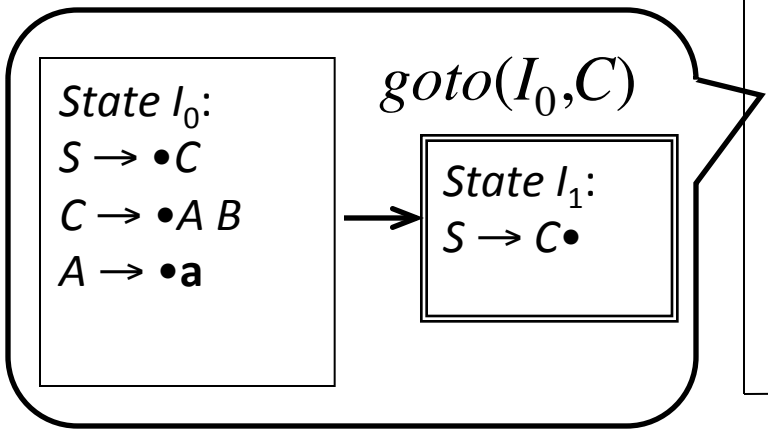
$A \rightarrow \mathbf{a}$

$B \rightarrow \mathbf{a}$

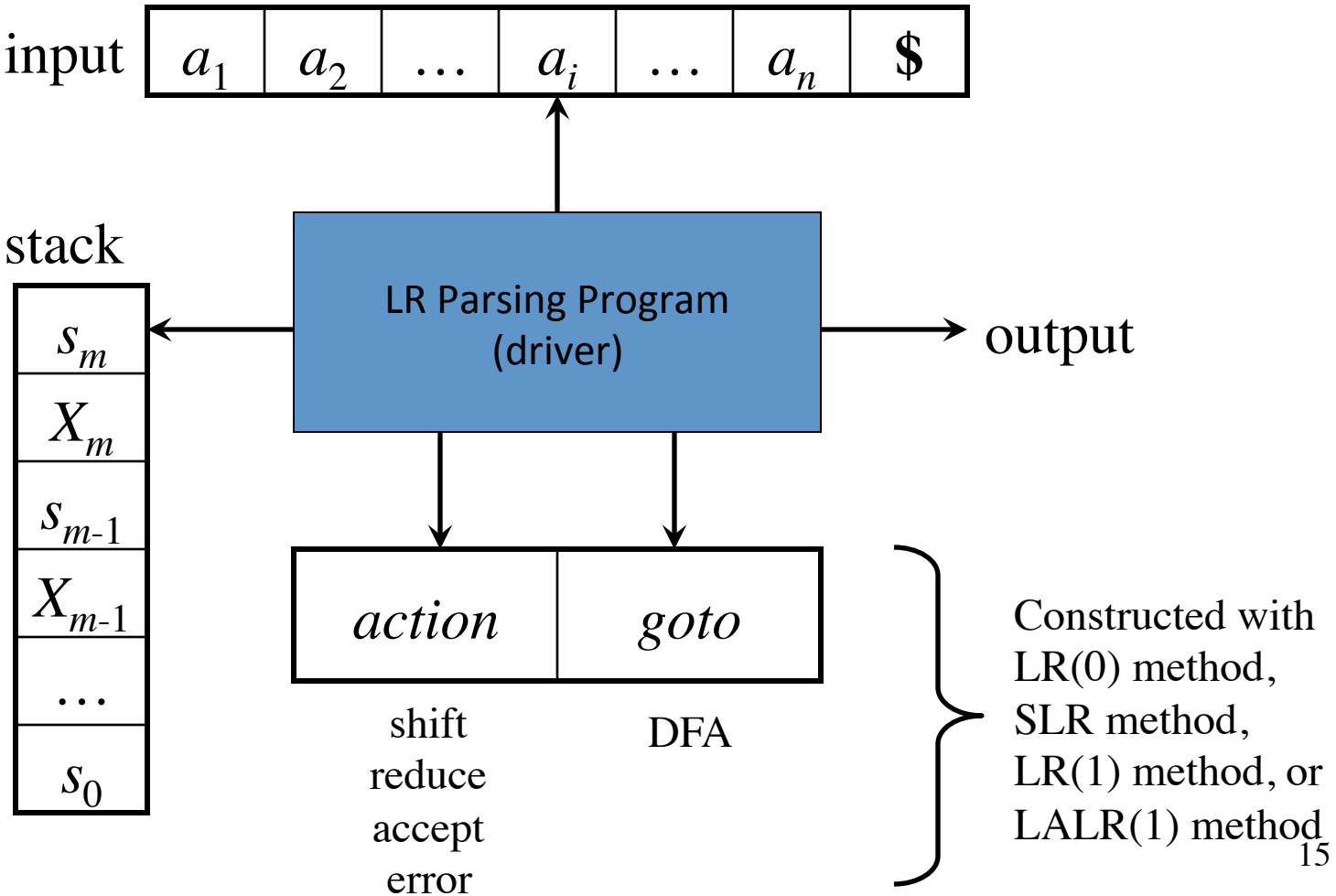


The states of the DFA are used to determine if a handle is on top of the stack

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



Model of an LR Parser



LR Parsing (Driver)

Configuration (= LR parser state):

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

Dropping the states, it is a right-sentential form

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

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

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

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

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

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

From LR(0) Automaton to LR(0) Parsing Table

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

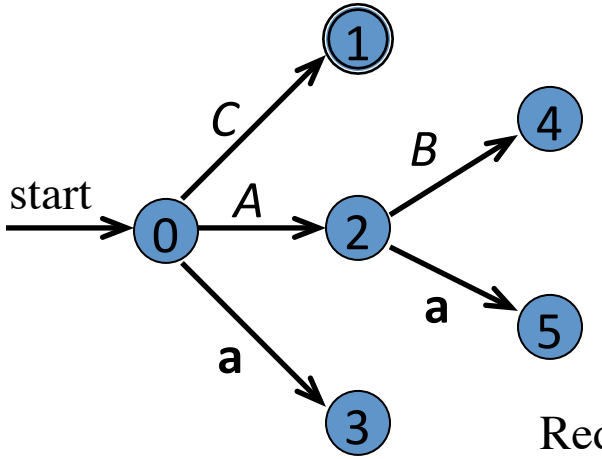
State I_1:
$C' \rightarrow C\bullet$

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

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

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

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



Shift & goto 3

Reduce by production #2

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

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

- Reduce actions are independent of the input symbol

Another Example LR Parse Table, constructed with the SLR algorithm

Grammar:

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

	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. $F \rightarrow \mathbf{id}$

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

Towards the LR(0) automaton: 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
- Intuition: position reached while parsing
- 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 $[A \rightarrow \bullet]$

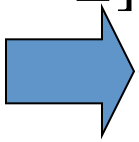
The Closure Operation for LR(0) Items

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

The Closure Operation (Example)

closure({[*E*' → •*E*]}) =

{ [*E*' → • *E*] }



{ [*E*' → • *E*]
[*E* → • *E* + *T*]
[*E* → • *T*] }

Add [*E*→•γ]

{ [*E*' → • *E*]
[*E* → • *E* + *T*]
[*E* → • *T*]
[*T* → • *T* * *F*]
[*T* → • *F*] }

Add [*T*→•γ]

{ [*E*' → • *E*]
[*E* → • *E* + *T*]
[*E* → • *T*]
[*T* → • *T* * *F*]
[*T* → • *F*]
[*F* → • **(** *E* **)**]
[*F* → • **id**] }

Add [*F*→•γ]

Grammar:

E → *E* **+** *T* | *T*

T → *T* ***** *F* | *F*

F → **(** *E* **)**

F → **id**

The Goto Operation for LR(0) Items

1. For each item $[A \rightarrow \alpha \bullet X \beta] \in I$, add the set of items *closure*($\{[A \rightarrow \alpha X \bullet \beta]\}$) 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 = { [$E' \rightarrow \bullet\ E$]
[$E \rightarrow \bullet\ E$ **+** T]
[$E \rightarrow \bullet\ T$]
[$T \rightarrow \bullet\ T$ ***** F]
[$T \rightarrow \bullet\ F$]
[$F \rightarrow \bullet$ **(** E **)**]
[$F \rightarrow \bullet$ **id**] }

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

Grammar:
$E \rightarrow E$ **+** $T \mid T$
$T \rightarrow T$ ***** $F \mid F$
$F \rightarrow$ **(** E **)**
$F \rightarrow$ **id**

The Goto Operation (Example 2)

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

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

Grammar:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E)$

$F \rightarrow \mathbf{id}$

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

Example Grammar and LR(0) Items

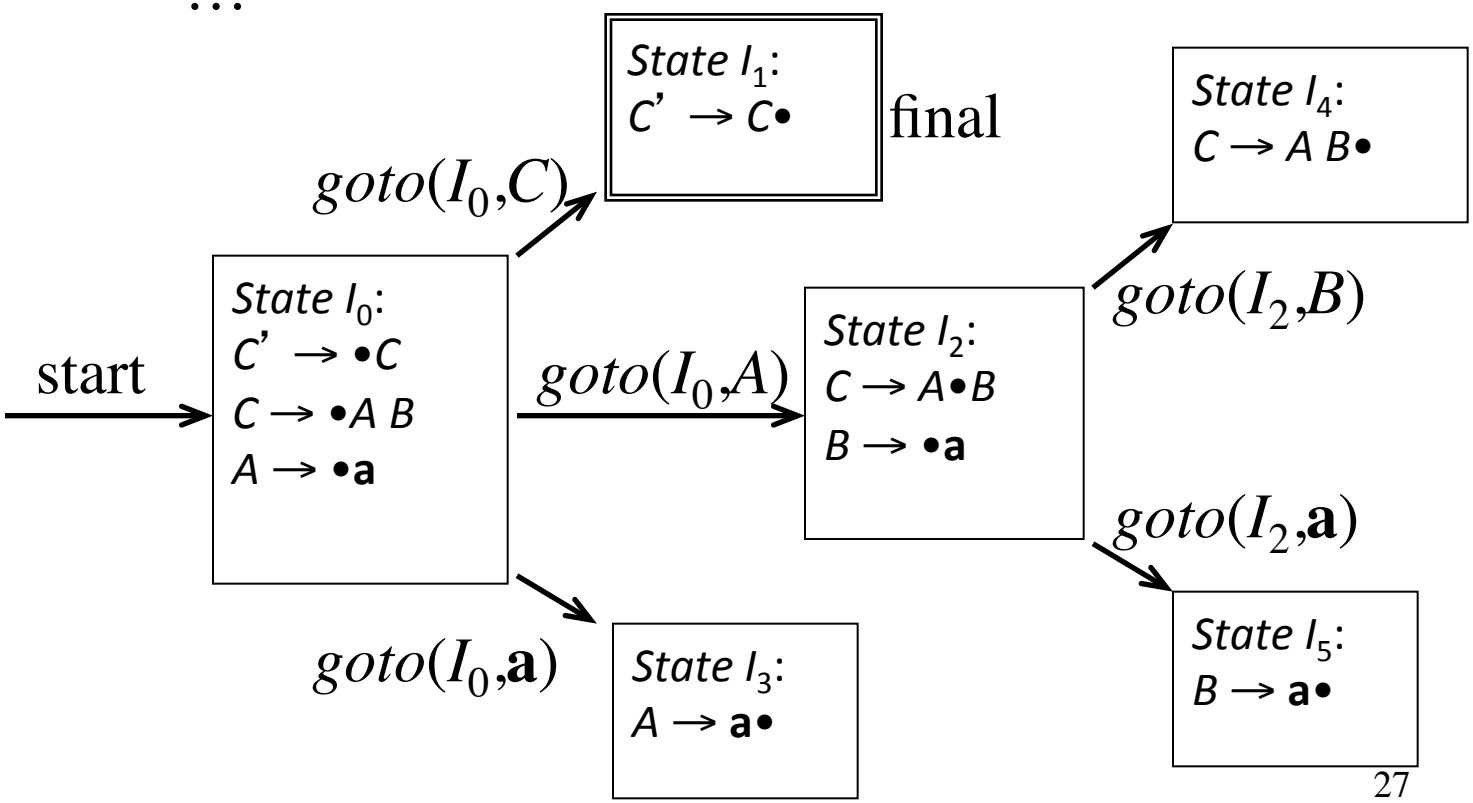
Augmented grammar:

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

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

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

…

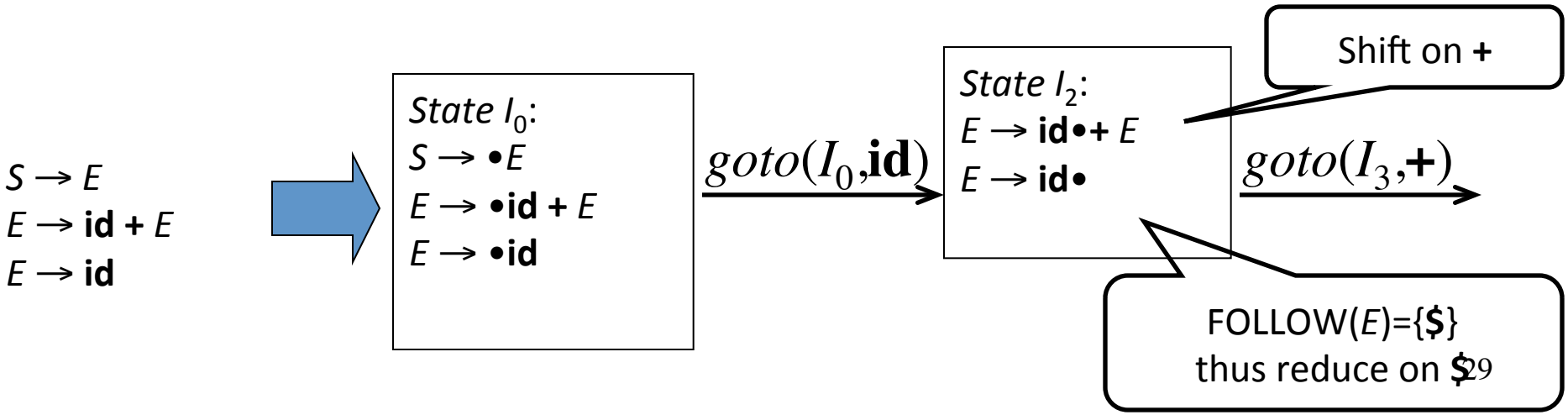


SLR Parsing

- 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. Use lookahead to solve certain conflicts.
- LR parser program uses the SLR parsing table to determine shift/reduce operations

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)

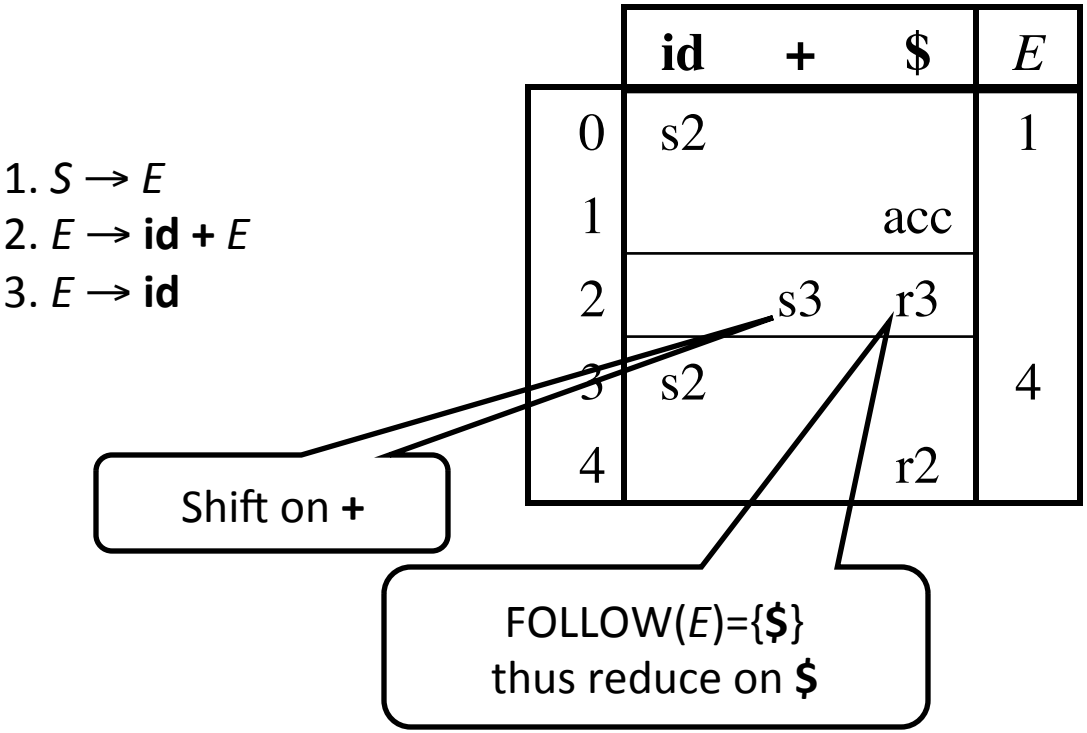


Constructing SLR Parsing Tables

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 $A\rightarrow\alpha$ for all $a \in$ 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 $[S' \rightarrow \bullet S]$

SLR Parsing Table

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



Example SLR Parsing Table

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

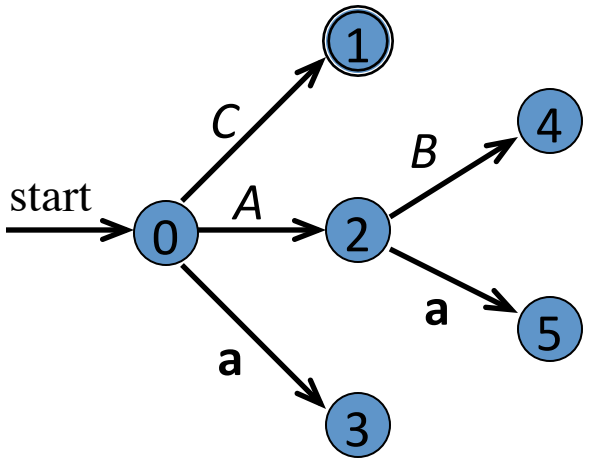
State I_1:
$C' \rightarrow C\bullet$

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

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

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

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



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

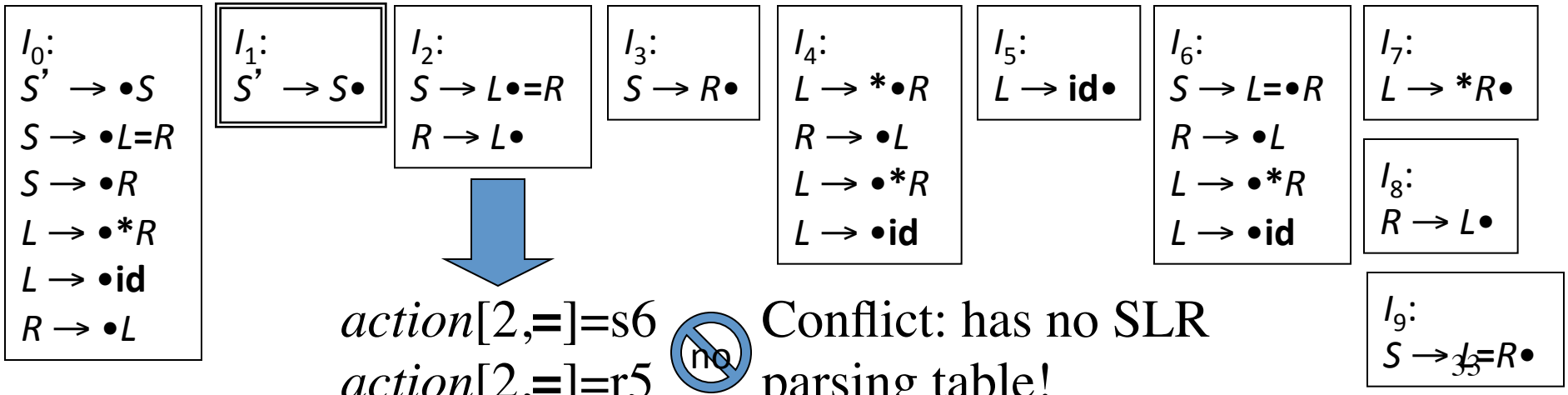
FOLLOW(*A*) = {**a**}
FOLLOW(*C*) = {**$**}
FOLLOW(*B*) = {**$**}

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

SLR, Ambiguity, and Conflicts

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

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



LR(1) Grammars

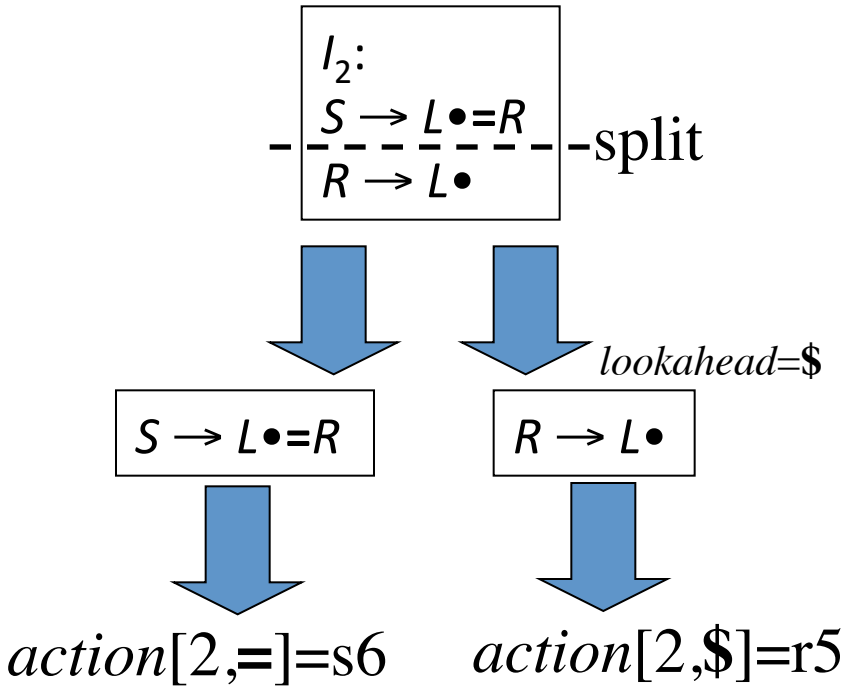
- SLR too simple
- LR(1) parsing uses lookahead to avoid unnecessary conflicts in parsing table
- LR(1) item = LR(0) item + lookahead

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

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

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. $L \rightarrow$ **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 βa
- For items of the form
  $[A \rightarrow \alpha \bullet, a]$
  the lookahead a is used to reduce $A \rightarrow \alpha$ only if the next symbol of the input is a. Clearly a is in FOLLOW(A), but not all of them must appear as lookahead.
- 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 $C = \mathit{closure}(\{[S' \rightarrow \bullet S, \$]\})$ (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 $\mathit{goto}(I,X) \notin C$ and $\mathit{goto}(I,X) \neq \emptyset$, add the set of items $\mathit{goto}(I,X)$ to C
4. Repeat 3 until no more sets can be added to C

Example Grammar and LR(1) Items

- Unambiguous LR(1) grammar:
  $S \rightarrow L = R$
  $S \rightarrow R$
  $L \rightarrow * R$
  $L \rightarrow \mathbf{id}$
  $R \rightarrow L$
- Augment with $S' \rightarrow S$
- LR(1) items (next slide)

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

I_1: $[S' \rightarrow S\bullet,$ **$**]

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

I_3: $[S \rightarrow R\bullet,$ **$**]

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

I_5: $[L \rightarrow$ **id**$\bullet,$ **=/$**]

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

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

I_8: $[R \rightarrow L\bullet,$ **=/$**]

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

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

Shorthand for two items in the same set

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

I_{12}: $[L \rightarrow$ **id**$\bullet,$ **$**]

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

Constructing Canonical LR(1) Parsing Tables

1. Augment the grammar with $S' \rightarrow S$
2. Construct the set $C=\{I_0,I_1,\ldots,I_n\}$ of LR(1) items
3. If $[A \rightarrow \alpha \bullet a \beta, b] \in I_i$ and $goto(I_i,a)=I_j$ then set $action[i,a]$=shift j
4. If $[A \rightarrow \alpha \bullet, a] \in I_i$ then set $action[i,a]$=reduce $A \rightarrow \alpha$ (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 LR(1) Parsing Table

Grammar:

1. S' → *S*
2. *S* → *L* = *R*
3. *S* → *R*
4. *L* → * *R*
5. *L* → **id**
6. *R* → *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

I_4: $[L \rightarrow *\bullet R,$ **=/$**$]$
$[R \rightarrow \bullet L,$ **=/$**$]$
$[L \rightarrow \bullet *R,$ **=/$**$]$
$[L \rightarrow \bullet$**id**$,$ **=/$**$]$

I_{11}: $[L \rightarrow *\bullet R,$ **$**$]$
$[R \rightarrow \bullet L,$ **$**$]$
$[L \rightarrow \bullet *R,$ **$**$]$
$[L \rightarrow \bullet$**id**$,$ **$**$]$

$[L \rightarrow *\bullet R,$ **=/$**$]$
$[R \rightarrow \bullet L,$ **=/$**$]$
$[L \rightarrow \bullet *R,$ **=/$**$]$
$[L \rightarrow \bullet$**id**$,$ **=/$**$]$

Example Grammar and LALR Parsing Table

- Unambiguous LR(1) grammar:

  $S \rightarrow L$ **=** R

  | R

  $L \rightarrow$ * R

  | **id**

  $R \rightarrow L$

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

I_0: [S' → •S, $] goto($I_0$,S)=$I_1$
[S → •L=R, $] goto($I_0$,L)=$I_2$
[S → •R, $] goto($I_0$,R)=$I_3$
[L → •*R, =/$] goto($I_0$,*)=$I_4$
[L → •**id**, =/$] goto($I_0$,**id**)=$I_5$
[R → •L, $] goto($I_0$,L)=$I_2$

I_1: [S' → S•, $]

I_2: [S → L•=R, $] goto($I_0$,=)=$I_6$
[R → L•, $]

I_3: [S → R•, $]

I_4: [L → *•R, =/$] goto($I_4$,R)=$I_7$
[R → •L, =/$] goto($I_4$,L)=$I_9$
[L → •*R, =/$] goto($I_4$,*)=$I_4$
[L → •**id**, =/$] goto($I_4$,**id**)=$I_5$

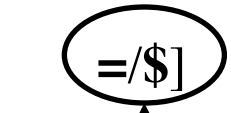
I_5: [L → **id**•, =/$]

I_6: [S → L=•R, $] goto($I_6$,R)=$I_8$
[R → •L, $] goto($I_6$,L)=$I_9$
[L → •*R, $] goto($I_6$,*)=$I_4$
[L → •**id**, $] goto($I_6$,**id**)=$I_5$

I_7: [L → *R•, =/$]

I_8: [S → L=R•, $]

I_9: [R → L•, =/$]



Shorthand
for two items

[R → L•, =]
[R → L•, $]

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
  - 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