

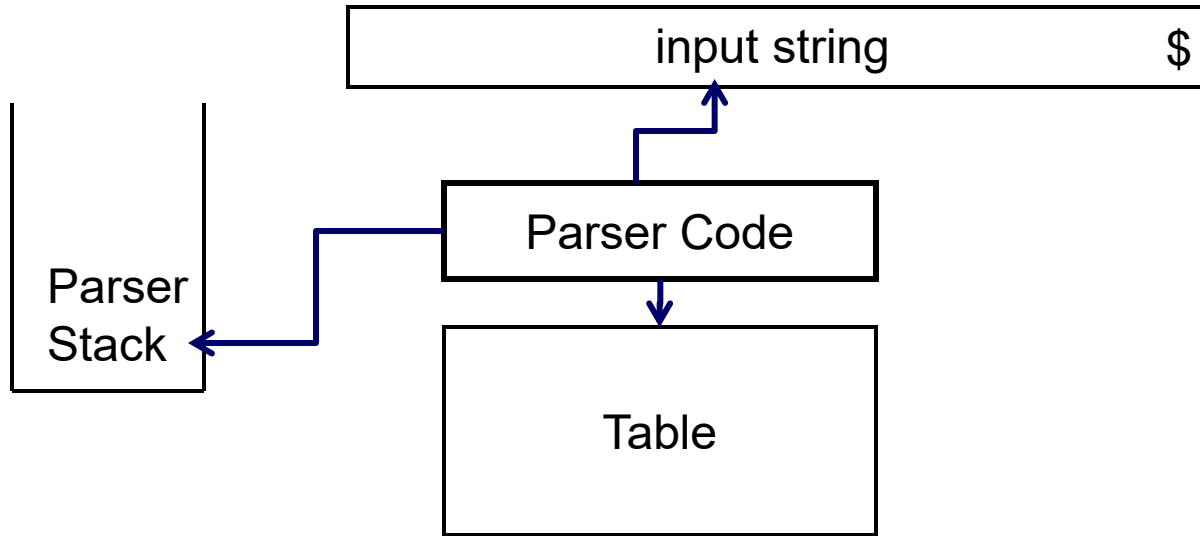
# Bottom Up Parsing

PITT CS 1622

# Bottom Up Parsing

- ❑ More powerful than top down
  - Don't need left factored grammars
  - Can handle left recursion
  - Can parse a larger set of grammars (and languages)
  
- ❑ Begins at leaves and works to the top
  - In reverse order of rightmost derivation  
(In effect, builds tree from left to right)
  
- ❑ Also known as Shift-Reduce parsing
  - Involves two types of operations: shift and reduce

# Parser Implementation

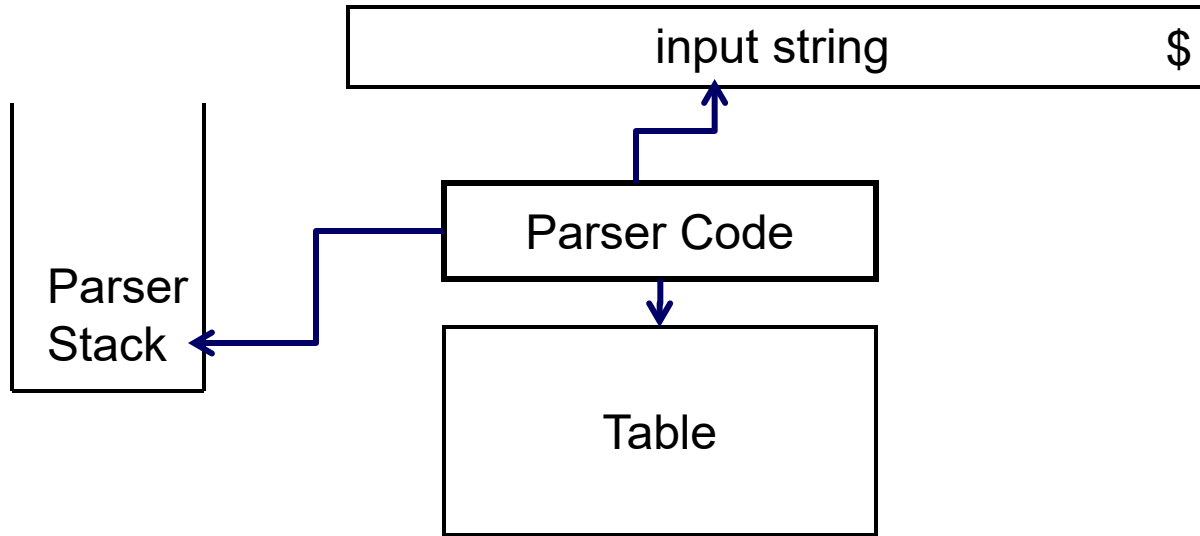


Parser Stack – holds consumed portion of derivation string

Table – “actions” performed based on rules of grammar, and current state of stack and input string

Parser Code – next action based on (**current token, stack top**)

# Parser Implementation



## Actions

1. **Shift** – consume input symbol and push symbol onto the stack
2. **Reduce** – pop RHS at stack top and push LHS of a production rule, reducing stack contents
3. **Accept** – success (when reduced to start symbol and input at \$)
4. **Error**

# Bottom-Up compared to Top-Down

❑ Conceptual difference in how the stack works:

	Stack Content At Input Start	Stack Content At Input End	Stack Represents	Key Operations
<b>Top-Down</b>	Start Symbol	Nothing	Unconsumed input	Match / Expand
<b>Bottom-up</b>	Nothing	Start Symbol	Consumed input	Shift / Reduce

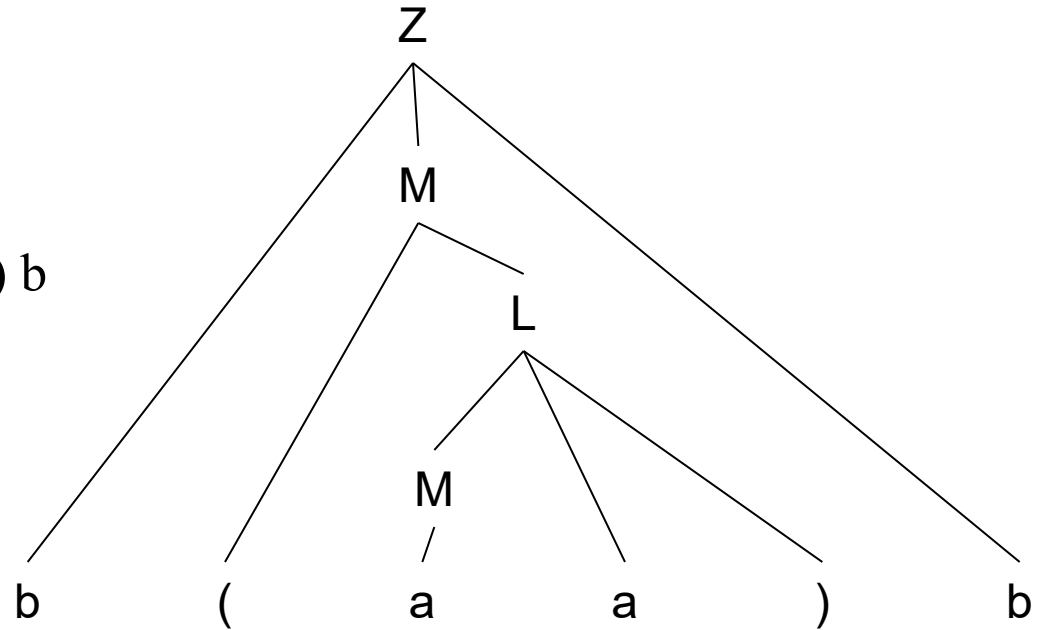
- ❑ But both use a stack to parse languages with nested structures
- Not surprising since CFGs are parsed using Pushdown Automata!

$Z \rightarrow b M b$

$M \rightarrow ( L \mid a$

$L \rightarrow M a \mid )$

Considering string:  $w = b ( a a ) b$



The rightmost derivation of this parse tree:

$Z \Rightarrow b M b \Rightarrow b ( L b \Rightarrow b ( M a ) b \Rightarrow b ( a a ) b$

Bottom up parsing involves finding “handles” (RHSs) to reduce

$b ( a a ) b \Rightarrow b ( M a ) b \Rightarrow b ( L b \Rightarrow b M b \Rightarrow Z$

$$Z \rightarrow b M b$$

$$M \rightarrow ( L \mid a$$

$$L \rightarrow M a ) \mid )$$

String  
 $b ( a a ) \$$

Stack	Input	Action
\$	b ( a a ) b \$	shift
\$ b	( a a ) b \$	shift
\$ b (	a a ) b \$	shift
\$ b ( a	a ) b \$	reduce
\$ b ( M	a ) b \$	shift
\$ b ( M a	) b \$	shift
\$ b ( M a )	b \$	reduce
\$ b ( L	b \$	reduce
\$ b M	b \$	shift
\$ b M b	\$	reduce
\$ Z	\$	accept

# Sentential Form and Handle

- ❑ **Sentential form:** Any string derivable from the start symbol
- ❑ **Handle:** RHS of a production rule that, when reduced to LHS in a sentential form, will lead to another sentential form
  
- ❑ **Definition:**
  - Let  $\alpha\beta w$  be a sentential form where
    - $\alpha, \beta$  is a string of terminals and non-terminals
    - $w$  is a string of terminals
    - $X \rightarrow \beta$  is a production rule
  - Then  $\beta$  is a handle of  $\alpha\beta w$  if
    - $S \Rightarrow^* \alpha X w \Rightarrow \alpha\beta w$  by a rightmost derivation
  - Handles formalize the intuition “ $\beta$  should be reduced to  $X$  for a successful parse”, but does not really say how to find them



# Single Pass Left-to-Right Scan

- Note in the formulation of a handle  $S \Rightarrow^* \alpha X w \Rightarrow \alpha \beta w$ 
  - $\alpha$  is a string of terminals and non-terminals
  - $w$  is a string of only terminals
  
- Why is this so?
  - Let's assume  $w$  contained a non-terminal  $Y$
  - $S \Rightarrow^* \alpha X w_1 Y w_2 \Rightarrow \alpha \beta w_1 Y w_2$ , where  $w = w_1 Y w_2$
  - Above is not a sentential form in a rightmost derivation.
  - Contradiction!
- What are the implications?
  - $\alpha \beta$  is consumed input and  $w$  is unconsumed input
  - Reduction only happens at the frontier of consumed input
  - Amenable to single pass left-to-right scan

# Handle Always Occurs at Top of Stack

## □ Grammar

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

Sentential form	Handle	Production
$id_1 + id_2 * id_3$	$id_1$	$E \rightarrow id$
$E + id_2 * id_3$	$id_2$	$E \rightarrow id$
$E + E * id_3$	$id_3$	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
$E$		

## □ # indicates top of stack (at the frontier of reduction where the handle is)

Left of # : stack contents, Right of # : unconsumed input string

$$\begin{aligned}
 id_1 \# + id_2 * id_3 &\Rightarrow E \# + id_2 * id_3 \Rightarrow E + \# id_2 * id_3 \Rightarrow E + id_2 \# * id_3 \\
 &\Rightarrow E + E \# * id_3 \Rightarrow E + E * id_3 \# \Rightarrow E + E * E \# \Rightarrow E + E \# \Rightarrow E
 \end{aligned}$$

## □ Stack works because reducing $X \rightarrow \beta$ involves popping recently pushed symbols

# Handle Always Occurs at Top of Stack

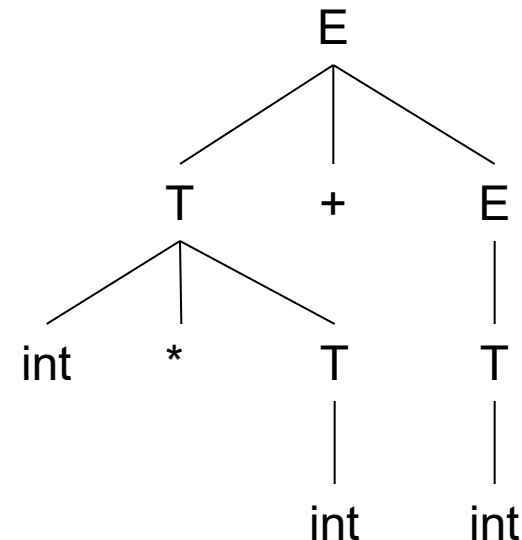
- Consider our usual grammar

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

$$T \rightarrow \text{int} * T \mid \text{int} \mid ( E )$$

Consider the string:  $\text{int} * \text{int} + \text{int}$

<i>sentential form</i>	<i>production</i>
$\text{int} * \text{int} \# + \text{int}$	$T \rightarrow \text{int}$
$\text{int} * T \# + \text{int}$	$T \rightarrow \text{int} * T$
$T + \text{int} \#$	$T \rightarrow \text{int}$
$T + T \#$	$E \rightarrow T$
$T + E \#$	$E \rightarrow T + E$
$E \#$	



- Reduction of a handle always happens at the top of the stack
- Makes life easier for parser (no need to access middle of stack)

# Ambiguous Grammars

❑ Conflicts arise with ambiguous grammars

- Just like LL parsing, bottom up parsing tries to predict the correct action, but if there are multiple correct actions, conflicts arise

❑ Example:

- Consider the ambiguous grammar

$$E \rightarrow E * E \mid E + E \mid ( E ) \mid \text{int}$$

Sentential form	Actions	Sentential form	Actions
int * int + int	shift	int * int + int	shift
...	...	...	...
E * E # + int	<b>reduce <math>E \rightarrow E * E</math></b>	E * E # + int	<b>shift</b>
E # + int	shift	E * E + # int	shift
E + # int	shift	E * E + int #	reduce $E \rightarrow \text{int}$
E + int #	reduce $E \rightarrow \text{int}$	E * E + E #	reduce $E \rightarrow E + E$
E + E #	reduce $E \rightarrow E + E$	E * E #	reduce $E \rightarrow E * E$
E #		E #	

# Ambiguity

- ❑ Previous shift-reduce conflict occurred because of ambiguity
  - Due to lack of precedence between  $+$  and  $*$  in the grammar
  - Ambiguity shows up as “conflicts” in the parsing table  
(More than one action in parse table, just like for LL parsers)
- ❑ Shift-reduce conflict also occurs with input “int + int + int”
  - Due to ambiguous associativity of  $*$  and  $+$
- ❑ Can always rewrite to encode precedence and associativity
  - But can sometimes result in convoluted grammars
  - Tools have other means to encode precedence and association
    - %left ‘+’ ‘-’
    - %left ‘\*’ ‘/’

# Properties of Bottom Up Parsing

- ❑ Handles always appear at the top of the stack
  - Never in middle of stack
  - Justifies use of stack in shift – reduce parsing
- ❑ Results in an easily generalized shift – reduce strategy
  - If there is no handle at the top of the stack, shift
  - If there is a handle, reduce to the non-terminal
  - Easy to automate the synthesis of the parser using a table
- ❑ Can have conflicts
  - If it is legal to either shift or reduce then there is a shift-reduce conflict.
  - If there are two legal reductions, then there is a reduce-reduce conflict.
  - Most often occur because of ambiguous grammars
    - In rare cases, because of non-ambiguous grammars not amenable to parser

# Types of Bottom Up Parsers

- ❑ Types of bottom up parsers
  - Simple precedence parsers
  - Operator precedence parsers
  - LR family parsers
  - ...
  
- ❑ In this course, we will only discuss LR family parsers
  - Most automated tools generate either LL or LR parsers
  - Precedence parsers are weaker siblings of LR parsers

# LR Parsers are more powerful than LL

## ❑ LR family of parsers

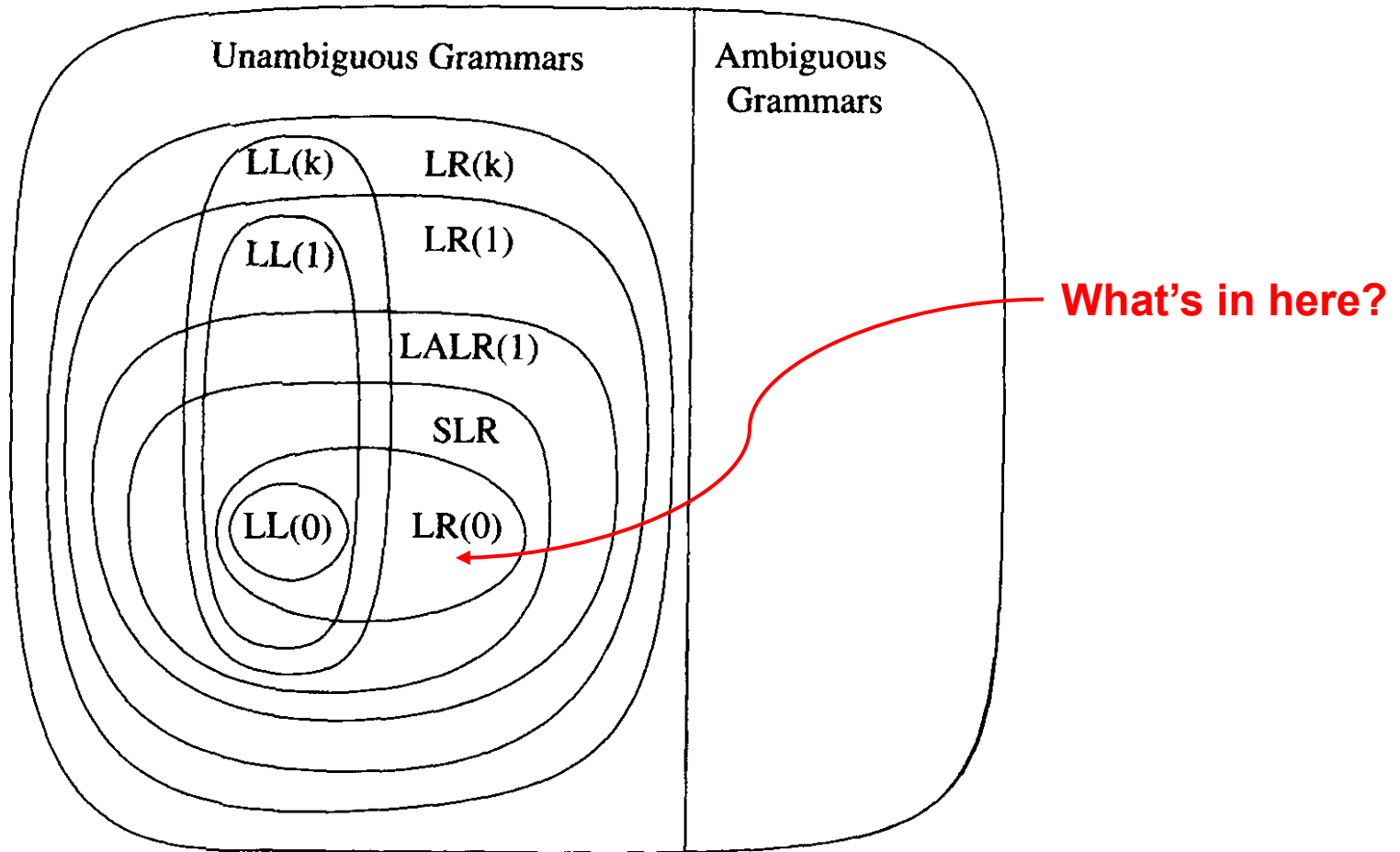
- LR(k)    L – left to right scan  
            R – rightmost derivation in reverse  
            k elements of look ahead

## ❑ Pros in comparison to LL(k)

1. More powerful than LL(k)
  - Handles more grammars: no left recursion removal, no left factoring needed
  - Handles more languages:  $LL(k) \subset LR(k)$
2. As efficient as LL(k)
  - Linear in time and space to length of input (same as LL(k))
3. As convenient as LL(k)
  - Can generate automatically from grammar – YACC, Bison



# A Hierarchy of Grammar Classes



# LR Parsers are harder to deal with

## ❑ Cons in comparison to LL(k)

1. More complex in structure compared to LL(k)
  - Structure of parser looks nothing like grammar
  - Parse conflicts are hard to understand and debug
2. Harder to emit informative error messages and recover from errors
  - LR is a bottom-up while LL is a top-down parser
  - When parse error occurs,  
LR: Knows only of currently reduced non-terminal  
LL: Knows how upper levels of tree look like and context of error
  - LL can emit smart error messages referring to context of error
  - LL can perform better error recovery according to context

---

# Implementation

## -- LR Parsing

---

Pitt, CS 1622

# Viable Prefix

□ Definition:  $\alpha$  is a viable prefix if

- There is a  $w$  where  $\alpha w$  is a rightmost sentential form, where  $w$  is the unconsumed input string
- In other words, if there is a  $w$  where  $\alpha \# w$  is a configuration of a shift-reduce parser

$b ( a \# a ) b \Rightarrow b ( M \# a ) b \Rightarrow b ( L \# b \Rightarrow b M \# b \Rightarrow Z \#$

□ If contents of parse stack is a viable prefix, that means the parser is on the right track (at least for the consumed input)

□ Shift-reduce parsing is the process of massaging the contents of the parse stack from viable prefix to viable prefix

- Error if neither shifting or reducing results in a viable prefix

# Massaging into a Viable Prefix

□ How do you know what results in a viable prefix?

➤ Example grammar

$S \rightarrow a B S \mid b$

$B \rightarrow b$

➤ Example shift and reduce on:  $a \# b b$

Shift:  $a \# b b \Rightarrow a b \# b$       How do you know shifting is the answer?

Reduce:  $a b \# b \Rightarrow a B \# b$       Should I apply  $B \rightarrow b$  (and not  $S \rightarrow b$ )?

□ You need to keep track of where you are on the RHS of rules

➤ In example

Shift:  $a \# b b \Rightarrow a b \# b$

$S \rightarrow a \# B S, B \rightarrow \# b \Rightarrow S \rightarrow a \# B S, B \rightarrow b \#$

Reduce:  $a b \# b \Rightarrow a B \# b$

$S \rightarrow a \# B S, B \rightarrow b \# \Rightarrow S \rightarrow a B \# S, S \rightarrow \# a B S, S \rightarrow \# b$

# LR(0) Item Notation

□ LR(0) Item: a production + a dot on the RHS

- Dot indicates extent of production already seen
- In example grammar

Items for production  $S \rightarrow a B S$

$S \rightarrow \cdot a B S$

$S \rightarrow a \cdot B S$

$S \rightarrow a B \cdot S$

$S \rightarrow a B S \cdot$

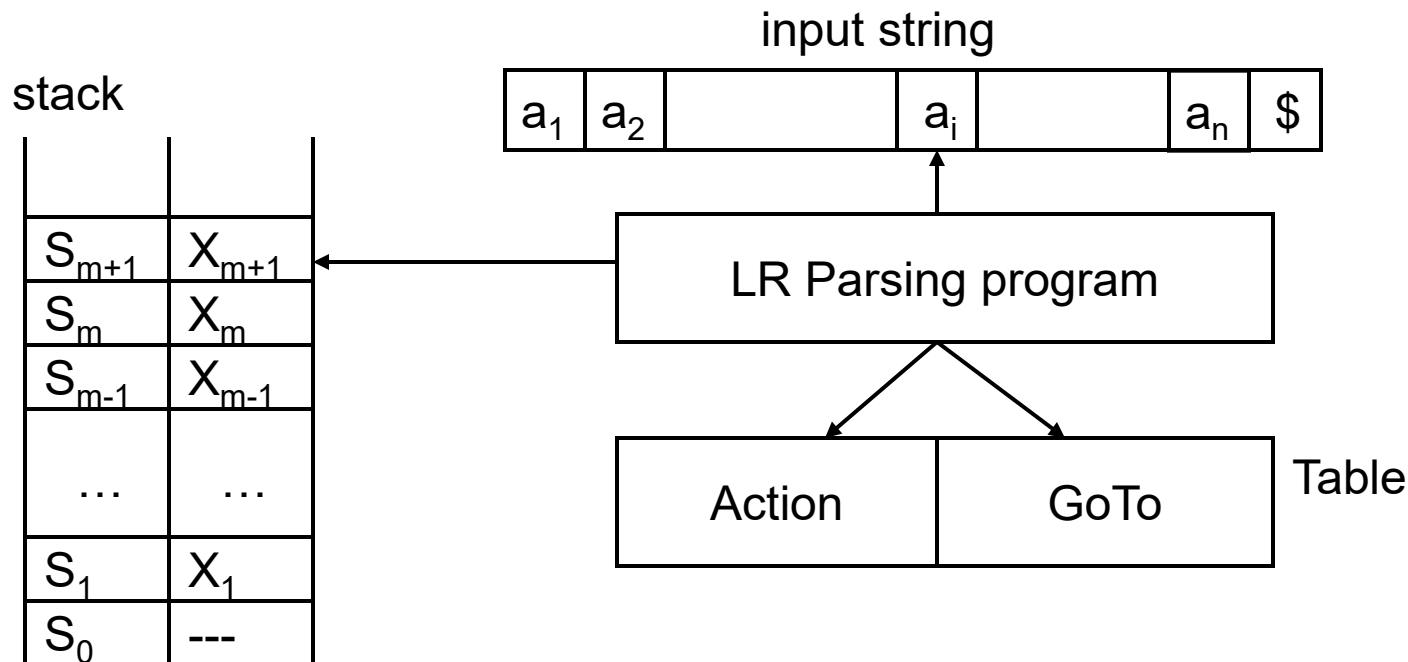
□ Items denote the idea of the viable prefix. E.g.

- $S \rightarrow \cdot a B S$  : to be a viable prefix, terminal 'a' needs to be shifted
- $S \rightarrow a \cdot B S$  : to be a viable prefix, a set of terminals need to be shifted and reduced to non-terminal 'B'

# States in the LR Parser

- ❑ State of LL parser is simply tuple (stack top symbol, input)
- ❑ State of LR parser is more complex
  - Must keep track of where we are in the RHS production rules
  - State is denoted by a set of LR(0) items
- ❑ Why a *set* of LR(0) items?
  - There may be multiple candidate RHSs for the prefix. E.g.  
For grammar  $S \rightarrow a b \mid a c$ ,  
 $S \rightarrow a . b$  and  $S \rightarrow a . c$  would be items in the same set
  - Even for one RHS, you may need multiple items expressing the same position, if the following symbol is a non-terminal. E.g.  
For grammar  $S \rightarrow a B, B \rightarrow b$   
 $S \rightarrow a . B$  and  $B \rightarrow . b$  would be items in the same set
- ❑ LR parsers keep track of states alongside symbols in stack

# Parser Implementation in More Detail



- Each grammar symbol  $X_i$  is associated with a state  $S_i$
- Contents of stack ( $X_1X_2\dots X_m$ ) is a viable prefix
- Contents of stack + input ( $X_1X_2\dots X_ma_1\dots a_n$ ) is a right sentential form
  - If the input string is a member of the language
- Uses **state** at the top of stack and current input to index into parsing table to determine whether to shift or reduce



# Parser Actions

$S_m$	$X_m$
$S_{m-1}$	$X_{m-1}$
...	...
$S_1$	$X_1$
$S_0$	---

...	$X_{m+1}$	...	\$
-----	-----------	-----	----

$S_m : A \rightarrow X_m \cdot X_{m+1}$   
 $S_{m-1} : A \rightarrow \cdot X_m X_{m+1}$   
 $B \rightarrow X_{m-1} \cdot A$

Shift

$S_{m+1}$	$X_{m+1}$
$S_m$	$X_m$
$S_{m-1}$	$X_{m-1}$
...	...
$S_1$	$X_1$
$S_0$	---

...	$X_{m+1}$	...	\$
-----	-----------	-----	----

$S_{m+1} : A \rightarrow X_m X_{m+1} \cdot$   
 $S_m : A \rightarrow X_m \cdot X_{m+1}$   
 $S_{m-1} : A \rightarrow \cdot X_m X_{m+1}$   
 $B \rightarrow X_{m-1} \cdot A$

Reduce(1)

$S_{m+1} : A \rightarrow X_m X_{m+1} \cdot$   
 $S_m : A \rightarrow X_m \cdot X_{m+1}$   
 $S_{m-1} : A \rightarrow \cdot X_m X_{m+1}$   
 $B \rightarrow X_{m-1} \cdot A$

...	$X_{m+2}$	...	\$
-----	-----------	-----	----

<del><math>S_{m+1}</math></del>	<del><math>X_{m+1}</math></del>
<del><math>S_m</math></del>	<del><math>X_m</math></del>
$S_{m-1}$	$X_{m-1}$
...	...
$S_1$	$X_1$
$S_0$	---

(2)

	$A$
$S_{m-1}$	$X_{m-1}$
...	...
$S_1$	$X_1$
$S_0$	---

GOTO

$S_A : B \rightarrow X_{m-1} A \cdot$   
 $S_{m-1} : A \rightarrow \cdot X_m X_{m+1}$   
 $B \rightarrow X_{m-1} \cdot A$

$S_A$	$A$
$S_{m-1}$	$X_{m-1}$
...	...
$S_1$	$X_1$
$S_0$	---

# Parser Actions

□ Assume configuration =  $S_0X_1S_1X_2S_2\ldots X_mS_m\#a_ia_{i+1}\ldots a_n\$$

□ Actions can be one of:

1. Shift input  $a_i$  and push new state **S**
  - New configuration =  $S_0X_1S_1X_2S_2\ldots X_mS_m a_i S \# a_{i+1}\ldots\$$
  - Where **Action**  $[S_m, a_i] = s[S]$
2. Reduce using Rule **R** ( $A \rightarrow \beta$ ) and push new state **S**
  - Let  $k = |\beta|$ , pop  $2*k$  symbols and push A
  - New configuration =  $S_0X_1S_1\ldots X_{m-k}S_{m-k}A S \# a_i a_{i+1}\ldots\$$
  - Where **Action**  $[S_m, a_i] = r[R]$  and **GoTo**  $[S_{m-k}, A] = [S]$
3. Accept – parsing is complete (**Action**  $[S_m, a_i] = \text{accept}$ )
4. Error – report and stop (**Action**  $[S_m, a_i] = \text{error}$ )

# Parse Table: Action and Goto

□ Action  $[S_m, a_i]$  can be one of:

- **s[S]: shift** input symbol  $a_i$  and push **state S**  
(One item in  $S_m$  must be of the form  $A \rightarrow \alpha \cdot a_i \beta$ )
- **r[R]: reduce** using **rule R** on seeing input symbol  $a_i$   
(One item in  $S_m$  must be  $R: A \rightarrow \alpha \cdot$ , where  $a_i \in \text{Follow}(A)$ )
  - Use  $\text{GoTo}[S_{m-|\alpha|}, A]$  to figure out state to push with  $A$
- **Accept** (One item in  $S_m$  must be  $S' \rightarrow S \cdot$  where  $S$  is the original start symbol, and  $a_i$  must be  $\$$ )
- **Error** (Cannot shift, reduce, accept on symbol  $a_i$  in state  $S_m$ )

□  $\text{GoTo}[S_m, X_i]$  is **[S]**:

- Next **state** to push when pushing nonterminal  $X_i$  from a reduction  
(At least one item in  $S_m$  must be of the form  $A \rightarrow \alpha \cdot X_i \beta$ )
- Similar to shifting input except now we are “shifting” a nonterminal

## □ Grammar

1.  $S \rightarrow E$
2.  $E \rightarrow E + T$
3.  $E \rightarrow T$
4.  $T \rightarrow \text{id}$
5.  $T \rightarrow (E)$

Non-terminal	Follow
S	\$
E	+ ) \$
T	+ ) \$

## ACTION

	+	id	(	)	\$
S0		s3	s4		
S1	s7				accept
S2	r3			r3	r3
S3	r4			r4	r4
S4		s3	s4		
S5	s7			s6	
S6	r5			r5	r5
S7		s3	s4		
S8	r2			r2	r2

## GOTO

	E	T	S
S0	1	2	
S1			
S2			
S3			
S4	5	2	
S5			
S6			
S7		8	
S8			

## □ Grammar

1.  $S \rightarrow E$
2.  $E \rightarrow E + T$
3.  $E \rightarrow T$
4.  $T \rightarrow \text{id}$
5.  $T \rightarrow (E)$

Non-terminal	Follow
S	\$
E	+ ) \$
T	+ ) \$

## ACTION

	+	id	(	)	\$
S0		s3	s4		
S1	s7				accept
S2	r3			r3	r3
S3	r4			r4	r4
S4		s3	s4		
S5	s7			s6	
S6	r5			r5	r5
S7		s3	s4		
S8	r2			r2	r2

## GOTO

	E	T	S
S0	1	2	
S1			
S2			
S3			
S4	5	2	
S5			
S6			
S7		8	
S8			

# Parse Table in Action

## □ Example input string

id + id + id

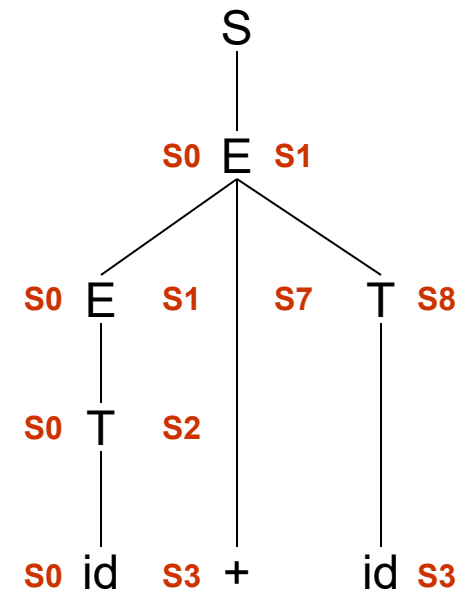
## □ Parser actions

Stack	Input	Actions
S0	id + id + id \$	Action[S0, id] (s3): Shift “id”, Push S3
S0 id S3	+ id + id \$	Action[S3, +] (r4): Reduce rule 4 (T→id) GoTo[S0, T] (2): Push S2
S0 T S2	+ id + id \$	Action[S2, +] (r3): Reduce rule 3 (E→T) GoTo[S0, E] (1): Push S1
S0 E S1	+ id + id \$	Action[S1, +] (s6): Shift “+”, Push S7
S0 E S2 + S7	id + id \$	Action[S7, id] (s3): Shift “id”, Push S3
...	...	...
S0 E S2 + S7 T S8	+ id \$	Action[S8, +] (r1): Reduce rule 2 (E→E+T) GoTo[S0, E] (1): Push S1
...	...	...

# Power Added to DFA by Stack

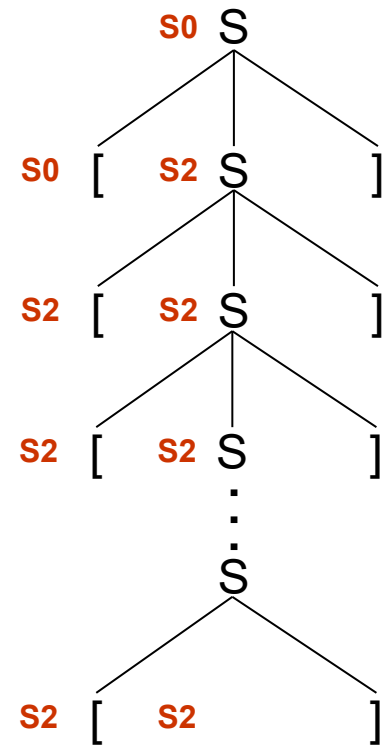
- ❑ LR parser is basically DFA+Stack (Pushdown Automaton)
- ❑ DFA: can only remember one state ( “dot” in current rule )
- ❑ DFA + Stack: remembers current state and all past states ( “dots” in rules higher up in the tree waiting for next symbol )

Stack	Input	Action
S0	id + id \$	s3
S0 id S3	+ id \$	r4, goto[S0, T]
S0 T S2	+ id \$	r3, goto[S0, E]
S0 E S1	+ id \$	s7
S0 E S2 + S7	id \$	s3
S0 E S2 + S7 id S3	\$	r4, goto[S7, T]
S0 E S2 + S7 T S8	\$	r2, goto[S0, E]
S0 E S1	\$	Accept



# Power Added to DFA by Stack

- ❑ Remember the following CFG for the language  $\{ [^i ]^i \mid i \geq 1 \}$ ?  
 $S \rightarrow [ S ] \mid [ ]$
- ❑ Regular grammars (or DFAs) could not recognize language because the state machine had to “count”
- ❑ LR parser stack counts number of **[ symbols**
- ❑ Q: Is this language LL(1)?
  - Yes. After left-factoring.  
 $S \rightarrow [ S', S' \rightarrow S ] \mid [ ]$
  - LL parser stack counts number of **] symbols**
  - Same pushdown automaton but different usage





# LR Parse Table Construction

- ❑ Must be able to decide on action from:
  - State at the top of stack
  - Next  $k$  input symbols (In practice,  $k = 1$  is often sufficient)
  
- ❑ To construct LR parse table from grammar
  1. Build deterministic finite automaton (DFA) using LR(0) items
  2. Express DFA using Action and GoTo tables
  
- ❑ State: Where we are currently in the structure of the grammar
  - Expressed as a set of LR(0) items
  - Each item expresses position in the RHS of a rule using a dot

# Construction of LR States

1. Create augmented grammar  $G'$  for  $G$ 
  - Given  $G: S \rightarrow \alpha \mid \beta$ , create  $G': S' \rightarrow S \mid S \rightarrow \alpha \mid \beta$
  - Creates a single rule  $S' \rightarrow S$  that when reduced, signals acceptance
2. Create first state by performing a *closure* on initial item  $S' \rightarrow \cdot S$ 
  - **Closure(I)**: computes set of items expressing the same position as  $I$
  - $\text{Closure}(\{S' \rightarrow \cdot S\}) = \{S' \rightarrow \cdot S, S \rightarrow \cdot \alpha, S \rightarrow \cdot \beta\}$
3. Create additional states by performing a *goto* on each symbol
  - **Goto(I, X)**: creates state that can be reached by advancing  $X$
  - If  $\alpha$  was single symbol, the following new state would be created:  
 $\text{Goto}(\{S' \rightarrow \cdot S, S \rightarrow \cdot \alpha, S \rightarrow \cdot \beta\}, \alpha) =$   
 $\text{Closure}(\{S \rightarrow \alpha \cdot\}) = \{S \rightarrow \alpha \cdot\}$
4. Repeatedly perform gotos until there are no more states to add

# Closure Function

□ Closure(I) where I is a set of items

- Returns the state (set of items) that express the same position as I
- Items in I are called **kernel items**
- Rest of items in closure(I) are called **non-kernel items**

□ Let N be a non-terminal

- If dot is in front of N, then add each production for that N and put dot at the beginning of the RHS
  - $A \rightarrow \alpha . B \beta$  is in I ; we expect to see a string derived from B
  - $B \rightarrow . \gamma$  is added to the closure, where  $B \rightarrow \gamma$  is a production
  - Apply rule until nothing is added

➤ Given

$$\begin{aligned} S &\rightarrow E \\ E &\rightarrow E + T \\ E &\rightarrow T \\ T &\rightarrow \text{id} \mid ( E ) \end{aligned}$$

$\text{Closure}(\{ S \rightarrow . E \}) = \{ S \rightarrow . E, E \rightarrow . E + T, E \rightarrow . T, T \rightarrow . \text{id}, T \rightarrow . ( E ) \}$

# Kernel and Non-kernel Items

## □ Two kinds of items

### ➤ Kernel items

- Items that act as “**seed**” items when creating a state
- What items act as seed items when states are created?
  - Initial state:  $S' \rightarrow \cdot S$
  - Additional state: from  $\text{goto}(I, X)$  so has  $X$  at left of dot
- Besides  $S' \rightarrow \cdot S$ , all kernel items have **dot in the middle of RHS**

### ➤ Non-kernel items

- Items added during the **closure** of kernel items
- All non-kernel items have **dot at the beginning of RHS**

# Goto Function

- ❑ Goto (I, X) where I is a set of items and X is a symbol
  - Returns state (set of items) that can be reached by advancing X
  - For each  $A \rightarrow \alpha . X \beta$  in I,  
Closure( $A \rightarrow \alpha X . \beta$ ) is added to goto(I, X)
  - X can be a terminal or non-terminal
    - Terminal if obtained from input string by shifting
    - Non-terminal if obtained from reduction
  - Example
    - $\text{Goto}(\{T \rightarrow . ( E )\}, ( ) = \text{closure}(\{T \rightarrow ( . E )\})$
- ❑ Ensures every symbol consumption results in a viable prefix

# Construction of DFA

- ❑ Algorithm to compute set C (set of all states in DFA)

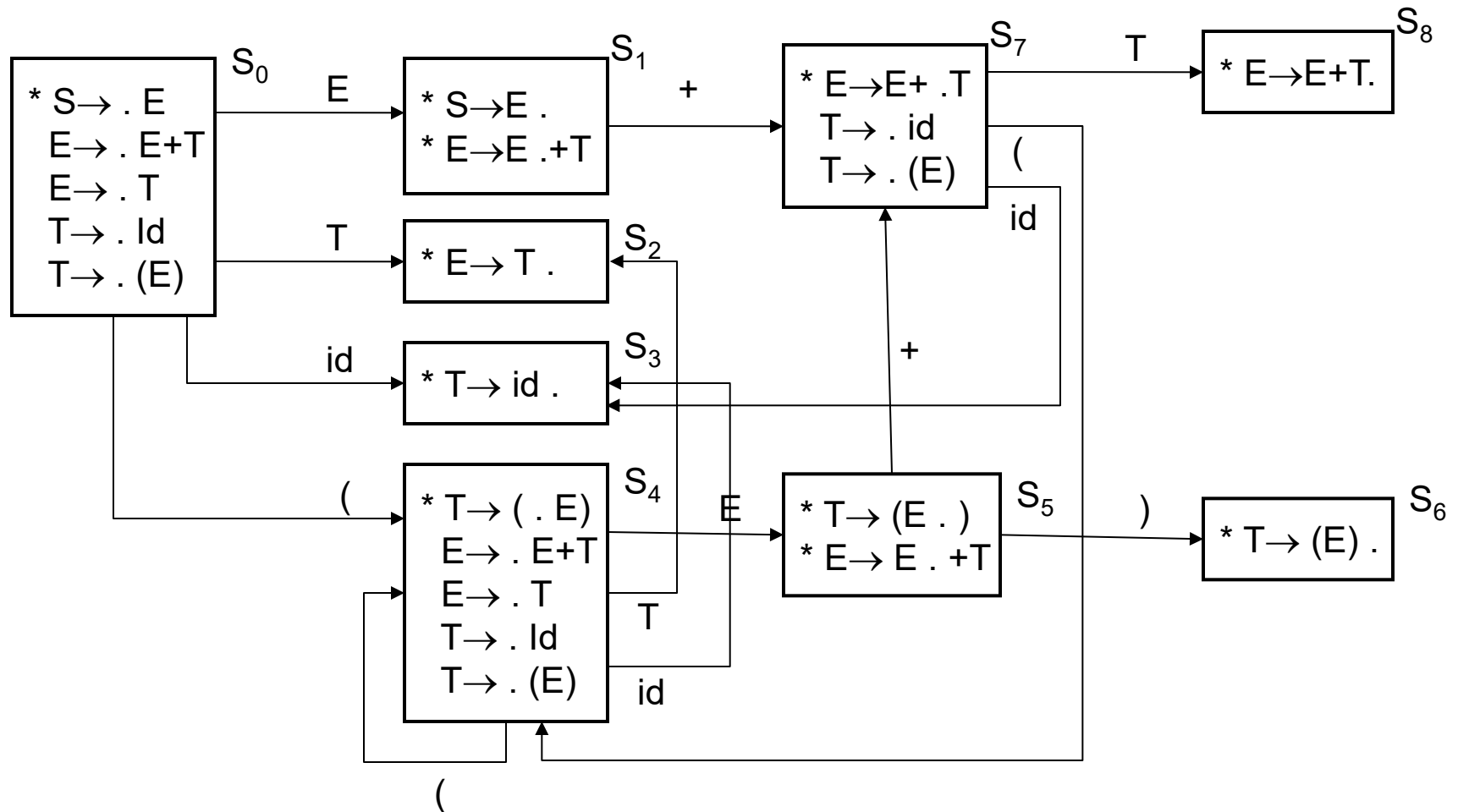
```
void constructDFA (G') {  
    C = {closure({S' → . S})} // Add initial state to C  
    repeat  
        for (each state I in C)  
            for (each grammar symbol X)  
                if (goto(I, X) is not empty and not in C)  
                    add goto(I, X) to C  
    until no new states are added to C  
}
```

- ❑ Add transitions from I to goto(I, X) on symbol X

□ Example:  $S \rightarrow E$   
 $E \rightarrow E + T \mid T$   
 $T \rightarrow \text{id} \mid ( E )$

- $S_0 = \text{closure}(\{S \rightarrow . E\})$   
 $= \{S \rightarrow . E, E \rightarrow . E + T, E \rightarrow . T, T \rightarrow . \text{id}, T \rightarrow . ( E )\}$
- $\text{goto}(S_0, E) = \text{closure}(\{S \rightarrow E ., E \rightarrow E . + T\})$   
 $S_1 = \{S \rightarrow E ., E \rightarrow E . + T\}$
- $\text{goto}(S_0, T) = \text{closure}(\{E \rightarrow T .\})$   
 $S_2 = \{E \rightarrow T .\}$
- $\text{goto}(S_0, \text{id}) = \text{closure}(\{T \rightarrow \text{id} .\})$   
 $S_3 = \{T \rightarrow \text{id} .\}$
- .....
- $S_8 = \dots$

□ DFA for the previous grammar  
 ( \* are closures applied to kernel items )





# Building Parse Table from DFA

- ACTION [state, terminal symbol]
- GOTO [state, non-terminal symbol]
- Filling in the ACTION and GOTO cells
  1. If  $[A \rightarrow \alpha \bullet a \beta]$  is in  $S_i$  and  $\text{goto}(S_i, a) = S_j$ , where “a” is a terminal  
then  $\text{ACTION}[S_i, a] = \text{shift } j$
  2. If  $[A \rightarrow \alpha \bullet A \beta]$  is in  $S_i$  and  $\text{goto}(S_i, A) = S_j$ , where “A” is a non-terminal  
then  $\text{GOTO}[S_i, A] = S_j$
  3. If  $[A \rightarrow \alpha \bullet]$  is in  $S_i$   
then  $\text{ACTION}[S_i, a] = \text{reduce } A \rightarrow \alpha$  for all  $a \in \text{Follow}(A)$
  4. If  $[S' \rightarrow S_0 \bullet]$  is in  $S_i$   
then  $\text{ACTION}[S_i, \$] = \text{accept}$
- Two potential prediction conflicts
  - Reduce-reduce conflict: when an ACTION cell has two 3s
  - Shift-reduce conflict: when an ACTION cell has both 1 and 3
  - ☞ More lookahead in  $\text{Follow}(A)$  may improve prediction accuracy

## □ Grammar

1.  $S \rightarrow E$
2.  $E \rightarrow E + T$
3.  $E \rightarrow T$
4.  $T \rightarrow \text{id}$
5.  $T \rightarrow (E)$

Non-terminal	Follow
S	\$
E	+ ) \$
T	+ ) \$

## ACTION

	+	id	(	)	\$
S0		s3	s4		
S1	s7				accept
S2	r3			r3	r3
S3	r4			r4	r4
S4		s3	s4		
S5	s7			s6	
S6	r5			r5	r5
S7		s3	s4		
S8	r2			r2	r2

## GOTO

	E	T	S
S0	1	2	
S1			
S2			
S3			
S4	5	2	
S5			
S6			
S7		8	
S8			

# Types of LR Parsers

- ❑ SLR – simple LR (what we saw so far was SLR(1))
  - Small parse table
  - Not as powerful
- ❑ Canonical LR
  - Much larger parse table
  - More powerful (can parse more grammars)
- ❑ LALR
  - Look ahead LR
  - In between the 2 previous ones in power and overhead

Overall parsing algorithm is the same – table is different

# Conflict due to not enough lookahead

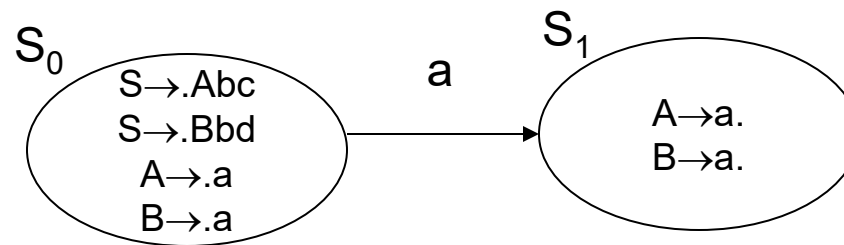
- Consider the grammar G

$S \rightarrow A b c \mid B b d$

$A \rightarrow a$

$b \in \text{Follow}(A)$  and also  $b \in \text{Follow}(B)$

$B \rightarrow a$



- What is reduced when “a b” is seen? reduce to A or B?
  - Reduce-reduce conflict
- G is not SLR(1) but SLR(2)
  - We need 2 symbols of look ahead to look past b:
    - b c – reduce to A
    - b d – reduce to B
  - Possible to extend SLR(1) to k symbols of look ahead – allows larger class of CFGs to be parsed

# SLR(k)

□ Extend SLR(1) definition to SLR(k) as follows

let  $\alpha, \beta \in V^*$

- $\text{First}_k(\alpha) = \{ x \in V_T^* \mid (\alpha \Rightarrow *x\beta \text{ where } |x|=k) \text{ or } (\alpha \Rightarrow *x \text{ where } |x| \leq k) \}$ 
  - all k-symbol terminal prefixes of strings derivable from  $\alpha$
- $\text{Follow}_k(B) = \{ w \in V_T^* \mid S \Rightarrow *\alpha B \gamma \text{ and } w \in \text{First}_k(\gamma) \}$ 
  - all k symbol terminal strings that can follow B in some derivation

# SLR(k) Parse Table

Let  $S$  be a state and lookahead  $b \in V_T^*$  such that  $|b| \leq k$

1. If  $A \rightarrow \alpha. \in S$  and  $b \in \text{Follow}_k(A)$  then
  - $\text{Action}(S, b)$  – reduce using production  $A \rightarrow \alpha$ ,
2. If  $D \rightarrow \alpha.a \gamma \in S$  and  $a \in V_T$  and  $b \in \text{First}_k(a \gamma \text{ Follow}_k(D))$ 
  - $\text{Action}(S, b) = \text{shift “a” and push state goto}(S, a)$

For  $k = 1$ , this definition reduces to SLR(1)

Reduce: Trivially true

Shift:  $\text{First}_1(a \gamma \text{ Follow}_1(D)) = \{a\}$

# $SLR(k-1) \subset SLR(k)$

□ Consider

$$S \rightarrow A b^{k-1} c \mid B b^{k-1} d$$

$$A \rightarrow a$$

$$B \rightarrow a$$

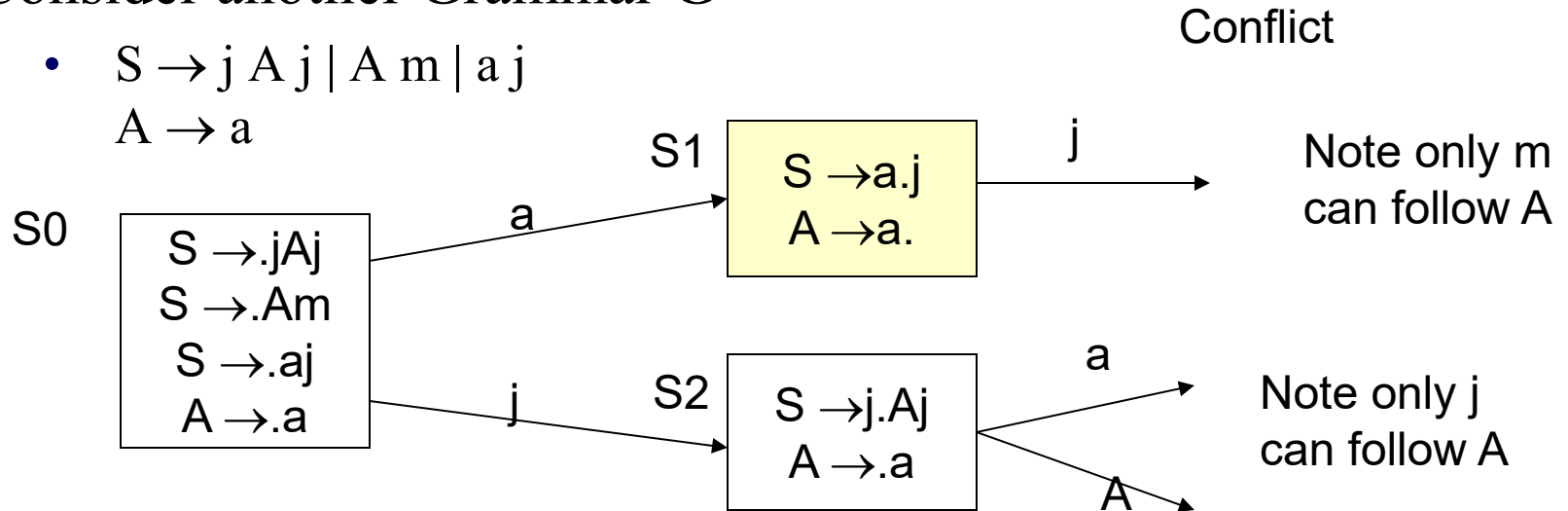
$SLR(k)$  not  $SLR(k-1)$

- cannot decide what to reduce,
- reduce  $a$  to  $A$  or  $B$  depends the next  $k$  symbols  
 $b^{k-1} c$  or  $b^{k-1} d$

# Non-SLR(k) for any k

□ Consider another Grammar G

- $S \rightarrow j A j \mid A m \mid a j$   
 $A \rightarrow a$



$\text{Follow}(A) = \{j, m\}$

State S1:  $[A \rightarrow a.]$  – reduce using this production (on j or m)

$[S \rightarrow a.j]$  – shift j  $\rightarrow$  shift-reduce conflict  $\rightarrow$  not SLR(1)

? SLR(k) for some  $k > 1$ ?

For reducing  $A \rightarrow a.$ :  $\text{Follow}_k(A) = \text{First}_k(j\text{Follow}_k(S)) + \text{First}_k(m\text{Follow}_k(S)) = \{j\$, m\$, \}$ ,

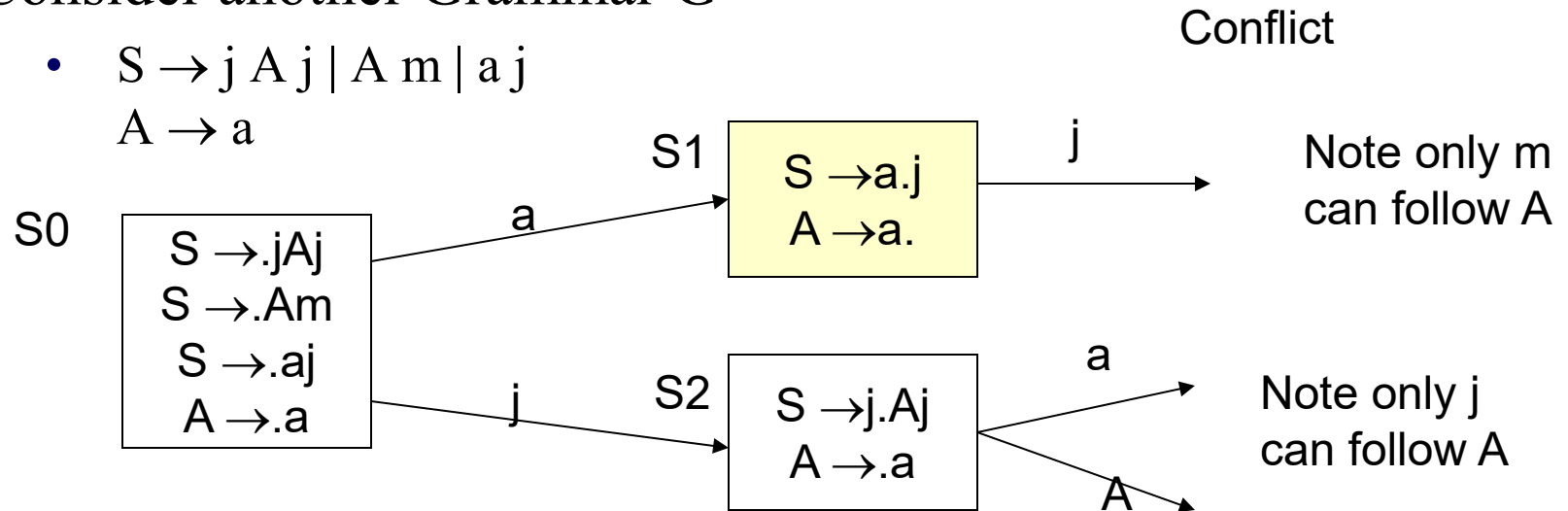
For shifting  $S \rightarrow a.j$ :  $\text{First}_k(j\text{Follow}_k(S)) = \{j\$, \}$  so not SLR(k) for any k !!!



# Non-SLR(k) for any k

## □ Consider another Grammar G

- $S \rightarrow j A j \mid A m \mid a j$   
 $A \rightarrow a$



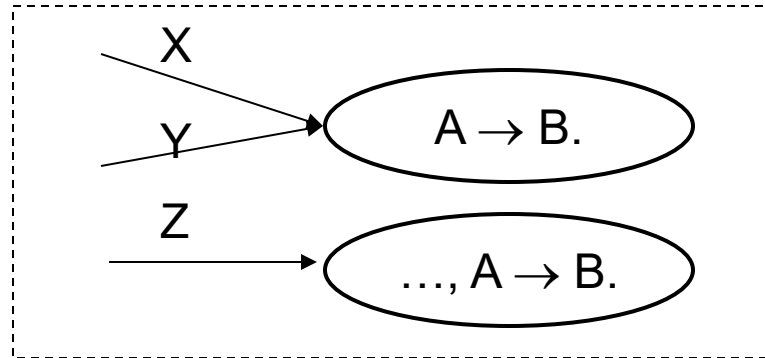
## □ Problem: $\text{Follow}(A) = \{j, m\}$ is too imprecise

- In S1, we should reduce A only when lookahead is  $\{m\}$
- The fact that  $\{j\}$  can follow A in another context is irrelevant

## □ Canonical LR:

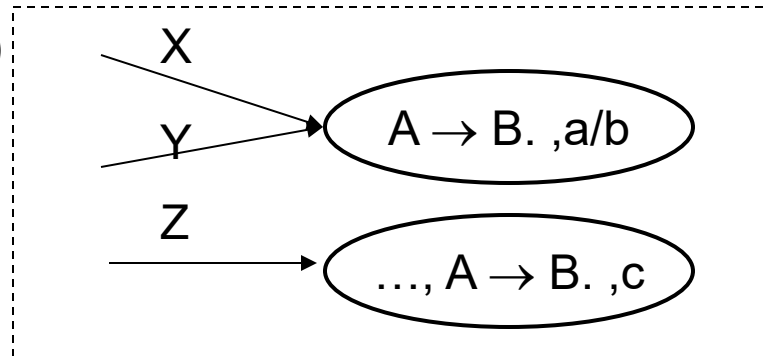
- Encode appropriate lookahead for the reduction of each LR item
- Appropriate lookahead is the follow set in the given context

SLR(1)



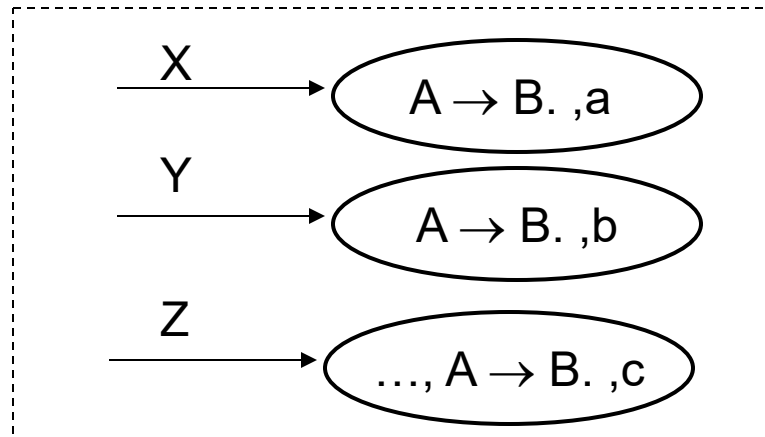
$\text{Follow}(A) = \{a, b, c\}$

LALR(1)



state merging

LR(1)



state splitting

# Constructing Canonical LR

## □ LR(1) item: LR item with one lookahead

- $[A \rightarrow \alpha.\beta, \mathbf{a}]$  where  $A \rightarrow \alpha\beta$  is a production and  $\mathbf{a}$  is a terminal or  $\$$ 
  - Meaning: Only terminal  $\mathbf{a}$  can follow  $A$  in this context
  - When we reach  $[A \rightarrow \alpha\beta., \mathbf{a}]$ , reduce  $A$  only if lookahead matches terminal  $\mathbf{a}$
- $[A \rightarrow \alpha.\beta, \mathbf{a/b}]$ : means both  $\mathbf{a}$  and  $\mathbf{b}$  can follow  $A$  in this context
  - $\{a, b\} \subseteq \text{Follow}(A)$ , a more precise version of the follow set

## □ LR(k) item: LR item with k lookahead

- $[A \rightarrow \alpha.\beta, \mathbf{a/b}]$ :  $a, b \in V_T^*$  such that  $|a| \leq k, |b| \leq k$

# Constructing Canonical LR

❑ Essentially the same as LR(0) items only adding lookahead

➤ Modify closure and goto function

❑ Changes for **closure** function

➤ Initialize lookahead:

If  $[A \rightarrow \alpha.B\beta, a]$  and  $B \rightarrow \delta$ ,

then  $[B \rightarrow \cdot \delta, c] \in \text{closure}([A \rightarrow \alpha.B\beta, a])$ , where  $c \in \text{First}(\beta a)$

❑ Changes for **goto** function

➤ Carry over lookahead:

if  $[A \rightarrow \alpha.X\beta, a] \in I$ , then  $\text{goto}(I, X) = [A \rightarrow \alpha X \cdot \beta, a]$

# Example

□ Grammar

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow eC \mid d$

□ S0:  $\text{closure}(S' \rightarrow .S, \$)$

$[S' \rightarrow .S, \$]$

$[S \rightarrow .CC, \$]$

$\text{first}(\epsilon \$) = \{\$ \}$

$[C \rightarrow .eC, e/d]$

$\text{first}(C \$) = \{e, d\}$

$[C \rightarrow .d, e/d]$

$\text{first}(C \$) = \{e, d\}$

□ S1:  $\text{goto}(S0, S) = \text{closure}(S' \rightarrow S., \$)$

$[S' \rightarrow S., \$]$

□ S2:  $\text{goto}(S0, C) = \text{closure}(S \rightarrow C.C, \$)$

$[S \rightarrow C.C, \$]$

$[C \rightarrow .eC, \$]$

$\text{first}(\epsilon \$) = \{\$ \}$

$[C \rightarrow .d, \$]$

$\text{first}(\epsilon \$) = \{\$ \}$

- S3:  $\text{goto}(S0, e) = \text{closure}(C \rightarrow e.C, e/d)$

$[C \rightarrow e.C, e/d]$

$[C \rightarrow .eC, e/d] \quad \text{first}(\epsilon e/d) = \{e, d\}$

$[C \rightarrow .d, e/d] \quad \text{first}(\epsilon e/d) = \{e, d\}$
- S4:  $\text{goto}(S0, d) = \text{closure}(C \rightarrow d., e/d)$

$[C \rightarrow d., e/d]$
- S5:  $\text{goto}(S2, C) = \text{closure}(S \rightarrow CC., \$)$

$[S \rightarrow CC., \$]$
- S6:  $\text{goto}(S2, e) = \text{closure}(C \rightarrow e.C, \$)$

$[C \rightarrow e.C, \$]$

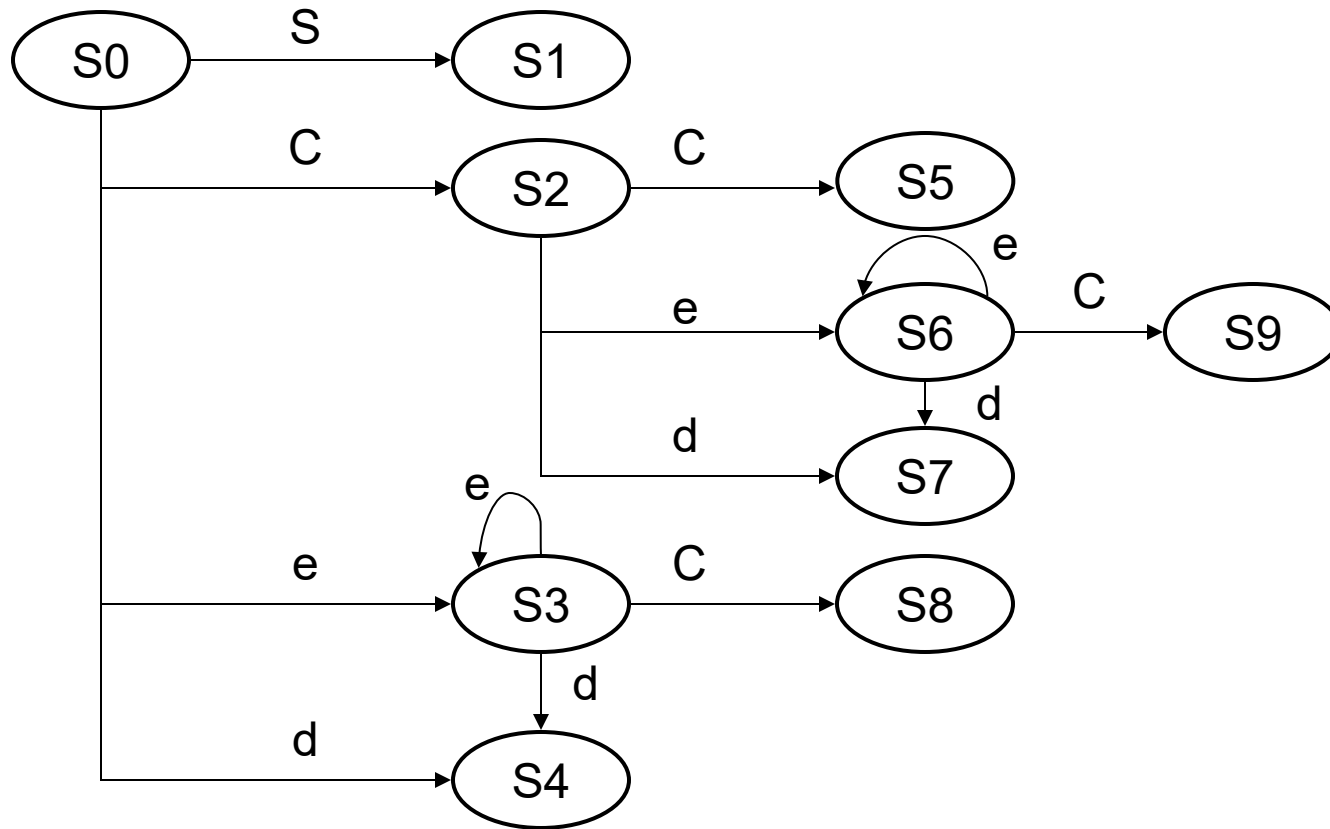
$[C \rightarrow .eC, \$] \quad \text{first}(\epsilon \$) = \{\$ \}$

$[C \rightarrow .d, \$] \quad \text{first}(\epsilon \$) = \{\$ \}$
- S7:  $\text{goto}(S2, d) = \text{closure}(C \rightarrow d., \$)$

$[C \rightarrow d., \$]$
- S8:  $\text{goto}(S3, C) = \text{closure}(C \rightarrow eC., e/d)$

$[C \rightarrow eC., e/d]$
- S9:  $\text{goto}(S6, C) = \text{closure}(C \rightarrow eC., \$)$

$[C \rightarrow eC., \$]$



Note S3, S6 are same except for lookahead (also true for S4, S7 and S8, S9)  
In SLR(1) – one state represents both

# Constructing Canonical LR Parse Table

- ❑ Shifting: same as before
- ❑ Reducing:
  - Don't use follow set (too coarse grain)
  - Reduce only if input matches lookahead for item
- ❑ Action and GOTO
  1. if  $[A \rightarrow \alpha \bullet a \beta, b] \in S_i$  and  $\text{goto}(S_i, a) = S_j$ ,  
Action[I,a] = s[Sj] – shift and goto state j if input matches a  
*Note: same as SLR*
  2. if  $[A \rightarrow \alpha \bullet, a] \in S_i$   
Action[I,a] = r[R] – reduce R:  $A \rightarrow \alpha$  if input matches a  
*Note: for SLR, reduced if input matches Follow(A)*

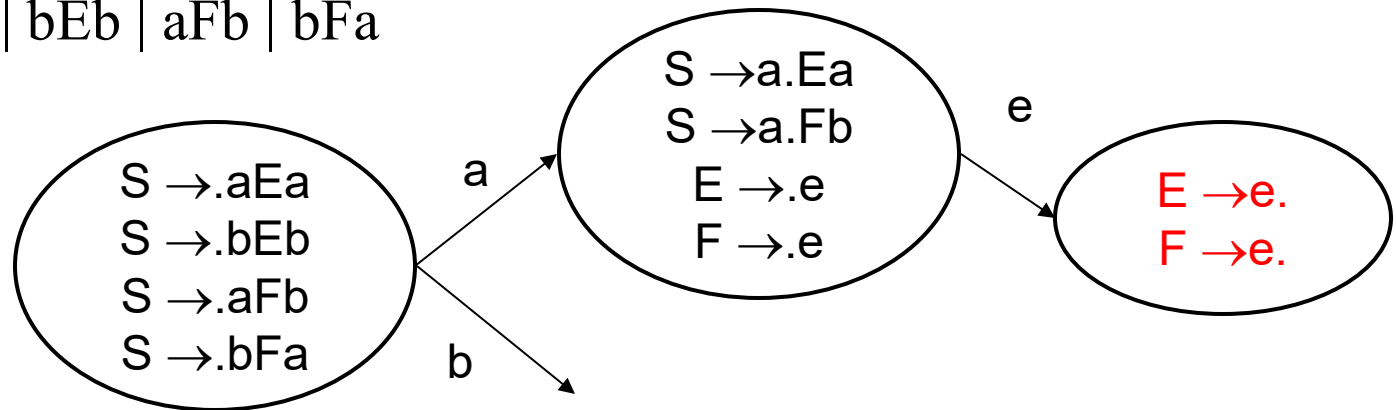


## □ Revisit SLR and LR

➤  $S \rightarrow aEa \mid bEb \mid aFb \mid bFa$

$E \rightarrow e$

$F \rightarrow e$



□ Not SLR(1): reduce/reduce conflict.

➤  $\text{Follow}(E) = \text{Follow}(F) = \{a, b\}$

□ LR(1): no conflict because state is split to account for context

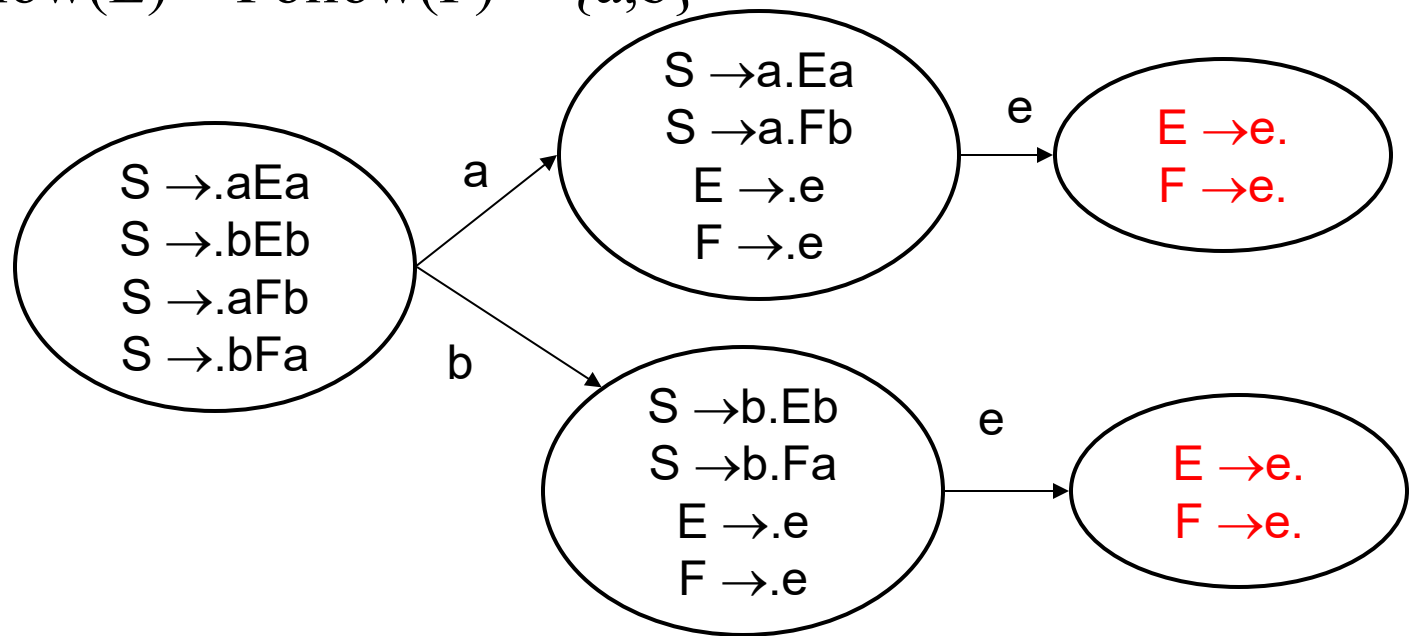
➤  $\text{Follow}(E) = \{a\}$  only if preceded by a

➤  $\text{Follow}(E) = \{b\}$  only if preceded by b

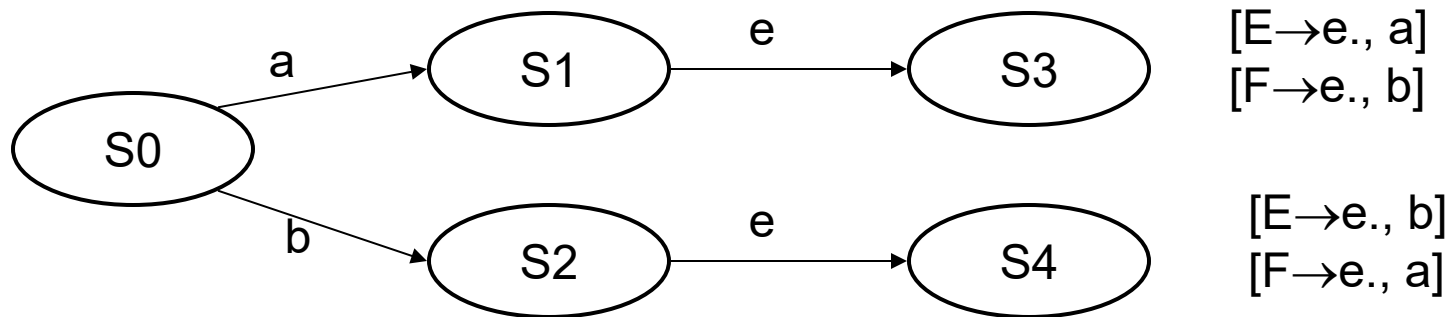
➤  $\text{Follow}(F) = \{a\}$  only if preceded by b

➤  $\text{Follow}(E) = \{b\}$  only if preceded by a

❑ SLR:  $\text{Follow}(E) = \text{Follow}(F) = \{a, b\}$



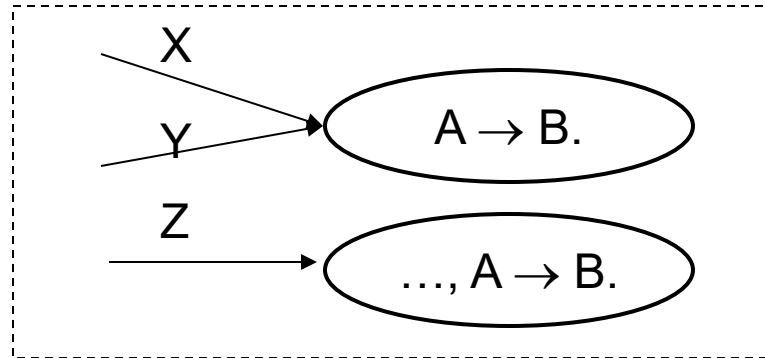
❑ LR: Follow sets more precise



# SLR(1) and LR(1)

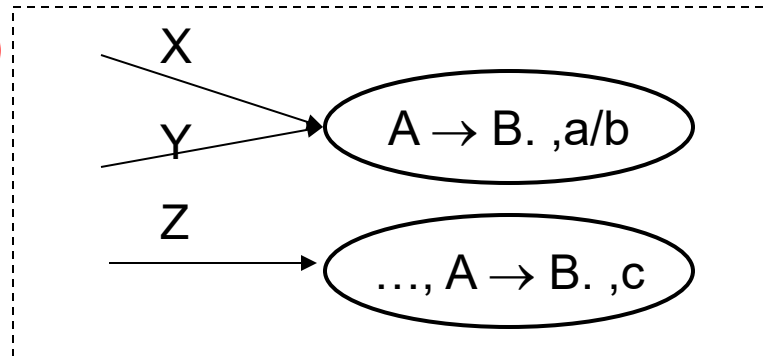
- ❑ LR(1) more powerful than SLR(1) – can parse more grammars
- ❑ But LR(1) may end up with many more states than SLR(1)
  - One LR(0) item may split up to many LR(1) items  
(Potentially as many as the powerset of the entire alphabet)
- ❑ LALR(1) – compromise between LR(1) and SLR(1)
  - Constructed by merging LR(1) states with the same core
    - Ends up with same number of states as SLR(1)
    - But items still retain some lookahead info – still better than SLR(1)
  - Used in practice because most programming language syntactic structures can be represented by LALR (not true for SLR)

SLR(1)



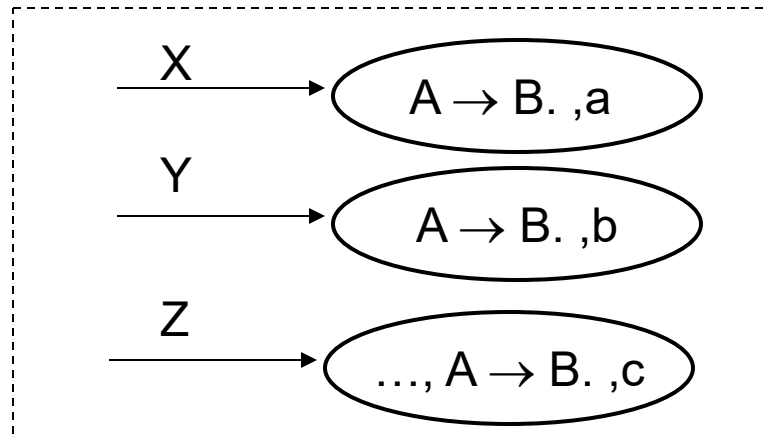
$\text{Follow}(A) = \{a, b, c\}$

LALR(1)

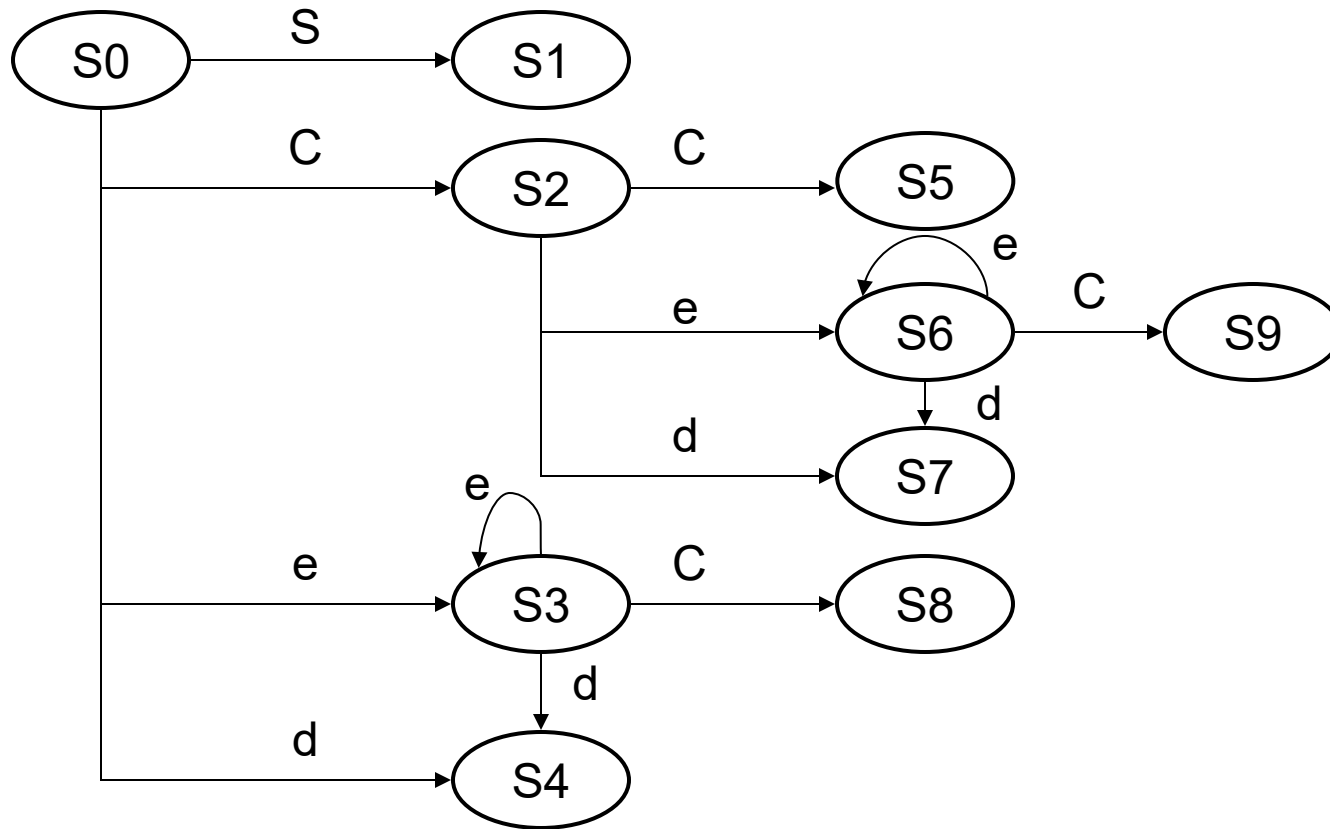


state merging

LR(1)



state splitting



Note S3 / S6, S4 / S7, S8 / S9 have same core (same except for lookahead).  
In an SLR(1) parser, one state represents both states.

# Example

□ Grammar

$$S' \rightarrow S$$
$$S \rightarrow CC$$
$$C \rightarrow eC \mid d$$

S3: goto(S0,e)=closure( $C \rightarrow e.C$ , e/d)

[ $C \rightarrow e.C$ , e/d]

[ $C \rightarrow .eC$ , e/d]

[ $C \rightarrow .d$ , e/d]

S4: goto(S0, d)=closure( $C \rightarrow d.$ , e/d)

[ $C \rightarrow d.$ , e/d]

S8: goto(S3, C)=closure( $C \rightarrow eC.$ , e/d)

[ $C \rightarrow eC.$ , e/d]

S6: goto(S2,e)=closure( $C \rightarrow e.C$ , \$)

[ $C \rightarrow e.C$ , \$]

[ $C \rightarrow .eC$ , \$]

[ $C \rightarrow .d$ , \$]

S7: goto(S2, d)=closure( $C \rightarrow d.$ , \$)

[ $C \rightarrow d.$ , \$]

S9: goto(S6,C)=closure( $C \rightarrow eC.$ , \$)

[ $C \rightarrow eC.$ , \$]

Note S3 / S6, S4 / S7, S8 / S9 have same core (same except for lookahead).  
In an SLR(1) parser, one state represents both states.

# Merging states

## □ Can merge S3 and S6

S3: goto(S0,e)=closure( $C \rightarrow e.C$ , e/d)

$[C \rightarrow e.C, e/d]$

$[C \rightarrow .eC, e/d]$

$[C \rightarrow .d, e/d]$

S6: goto(S2,e)=closure( $C \rightarrow e.C$ , \$)

$[C \rightarrow e.C, \$]$

$[C \rightarrow .eC, \$]$

$[C \rightarrow .d, \$]$

S36:  $[C \rightarrow e.C, e/d/\$]$

$[C \rightarrow .eC, e/d/\$]$

$[C \rightarrow .d, e/d/\$]$

## □ Similarly

- S47:  $[C \rightarrow d., e/d/\$]$
- S89:  $[C \rightarrow eC., e/d/\$]$

# Effect of Merging: Introduces conflicts

## 1. Merging of states can introduce conflicts

- cannot introduce shift-reduce conflicts
- can introduce reduce-reduce conflicts

### □ Shift-reduce conflicts

Suppose  $S_{ij}$ :  $[A \rightarrow \alpha., \textcolor{red}{a}/b/c]$  reduce on input  $a$   
 $[B \rightarrow \beta.\textcolor{red}{a}\delta, x/y/z]$  shift on input  $a$   
formed by merging  $S_i$  and  $S_j$

Then,  $S_i$ :  $[A \rightarrow \alpha., \text{lookahead}_i]$   $S_j$ :  $[A \rightarrow \alpha., \text{lookahead}_j]$   
 $[B \rightarrow \beta.\textcolor{red}{a}\delta, \dots]$   $[B \rightarrow \beta.\textcolor{red}{a}\delta, \dots]$

And, either  $\textcolor{red}{a} \in \text{lookahead}_i$  or  $\textcolor{red}{a} \in \text{lookahead}_j$

☞ Conflict existed in the first place!



# A reduce-reduce conflict due to merging

$S \rightarrow aEa \mid bEb \mid aFb \mid bFa$

$E \rightarrow e$

$F \rightarrow e$

S3:  $[E \rightarrow e., a]$

$[F \rightarrow e., b]$

S4:  $[E \rightarrow e., b]$

$[F \rightarrow e., a]$

After merging S34:  $[E \rightarrow e., a/b]$

$[F \rightarrow e., a/b]$

- Both reductions are applied on lookahead a and b,  
i.e. reduce-reduce conflict

# Effect of Merging: Delays error detection

## 2. Detection of errors may be delayed

- On error, LALR parsers will not perform shifts beyond an LR parser but may perform more reductions before finding error

- Example:  $S' \rightarrow S$                        $S \rightarrow CC$

$C \rightarrow eC \mid d$

and input string eed\$

- Canonical LR: Parse Stack  $S0 \ e \ S3 \ e \ S3 \ d \ S4$

State  $S4$  on  $\$$  input = error  $S4: \{C \rightarrow d., e/d\}$

- LALR:

stack:  $S0 \ e \ S_{36} \ e \ S_{36} \ d \ S_{47}$      $\rightarrow$  state  $S_{47}$  input  $\$$ , reduce  $C \rightarrow d$

stack:  $S0 \ e \ S_{36} \ e \ S_{36} \ C \ S_{89}$      $\rightarrow$  reduce  $C \rightarrow eC$

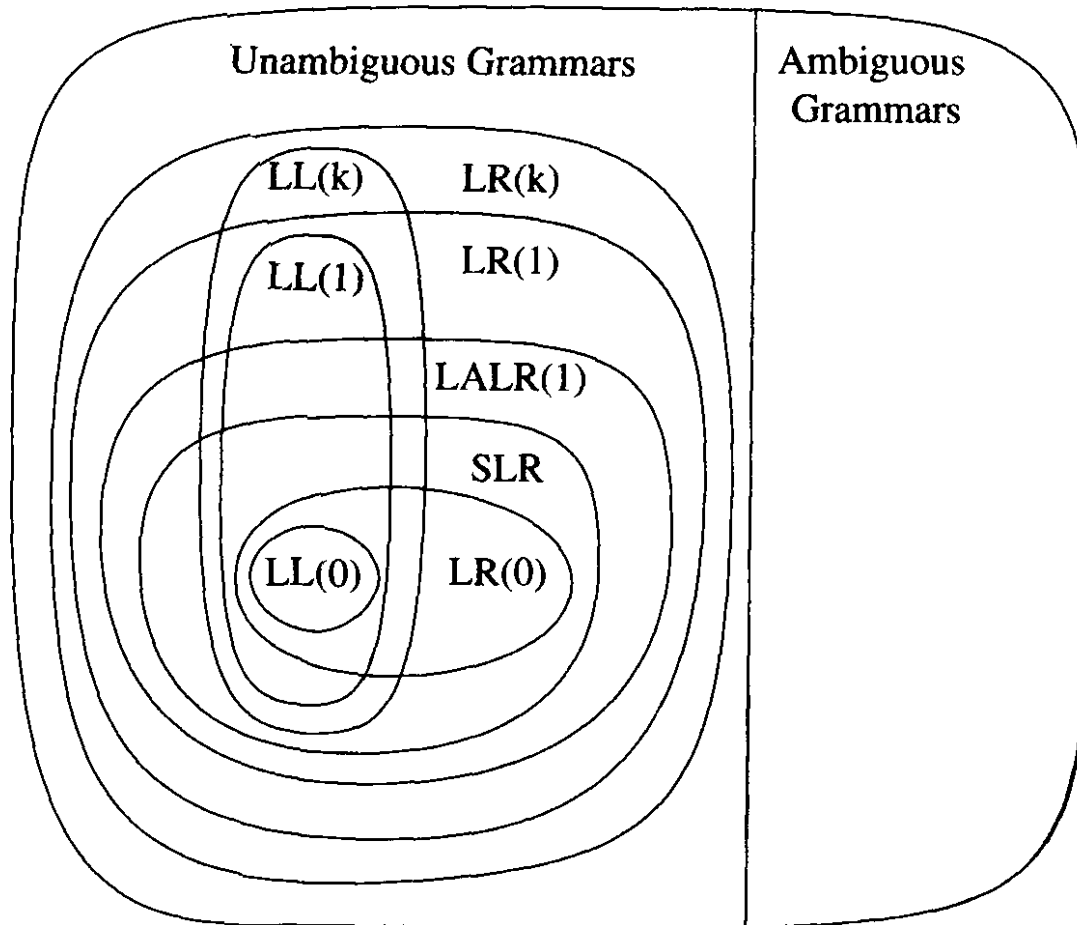
stack:  $S0 \ e \ S_{36} \ C \ S_{89}$              $\rightarrow$  reduce  $C \rightarrow eC$

stack:  $S0 \ C \ S2$                        $\rightarrow$  state  $S2$  on input  $\$$ , error

# Error Recovery

- ❑ To uncover multiple errors, parser must be able to recover from errors.
- ❑ Simple error recovery (by discarding offending code sequence)
  1. Decide on non-terminal A: candidate for discarding
    - Typically, an expression, statement, or block of code
  2. Continue to scan down the stack until a state S with a goto on a particular non-terminal A is found
  3. Discard input tokens until a token 'a' is found that can follow A
    - E.g. if A is a statement, then 'a' would be ';'
  4. Push state Goto[a,A] on stack and continue parsing

# A Hierarchy of Grammar Classes



# $LALR(k) \subset LR(k)$

- ❑  $LR(k)$  is strictly more powerful compared to  $LALR(k)$ 
  - $LALR$  merges states, which can introduce conflicts
- ❑ Unlike  $LL$  and  $LR$ , no formal definition on what is  $LALR$ 
  - Definition by construction: if  $LALR$  parser has no conflicts
  - Conflicts due to state merging are hard to define formally (Hence, they are unpredictable and hard to reason with)
- ❑ Nonetheless,  $LALR(1)$  has become popular
  - YACC, Bison, etc.
  - Most programming languages have an  $LALR(1)$  grammar
  - Reduce-reduce conflicts due to state merging are rare (conflicts are mostly due to ambiguity)

# SLR(k) $\subset$ LALR(k)

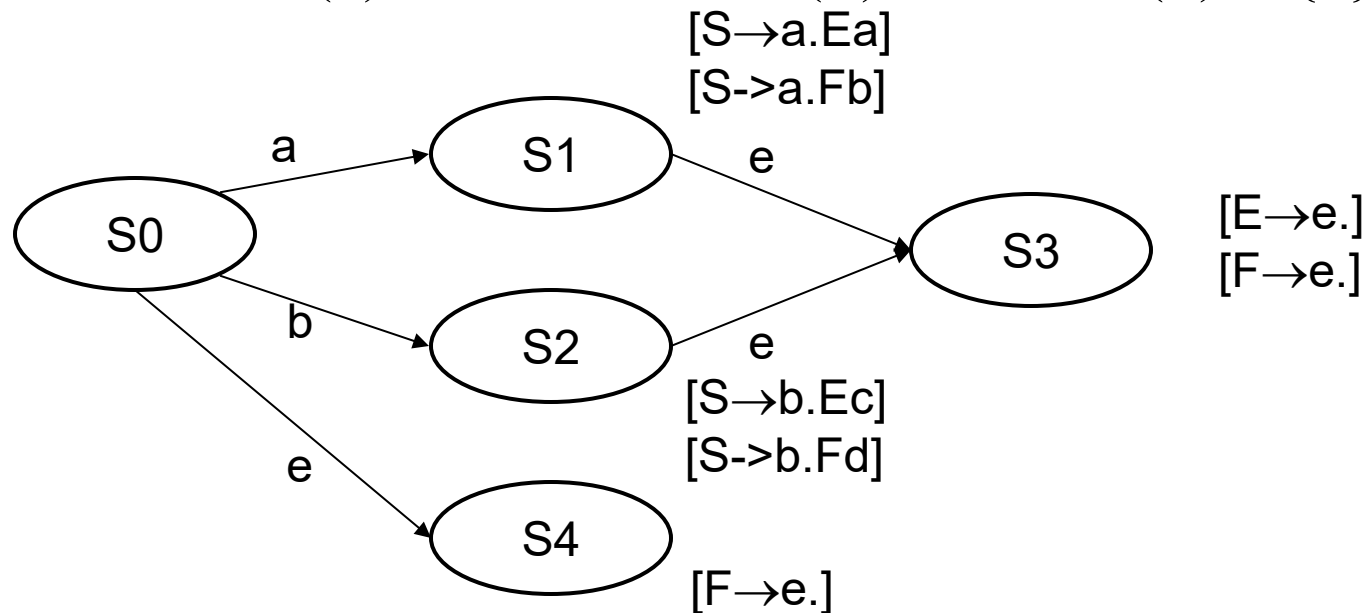
□ Let's consider this Grammar G:

➤  $S \rightarrow aEa \mid aFb \mid bEc \mid bFd \mid Fa$

$E \rightarrow e$

$F \rightarrow e$

□ It is non-SLR(1) because  $\text{Follow}(E) \cap \text{Follow}(F) = \{a\}$



# SLR(k) $\subset$ LALR(k)

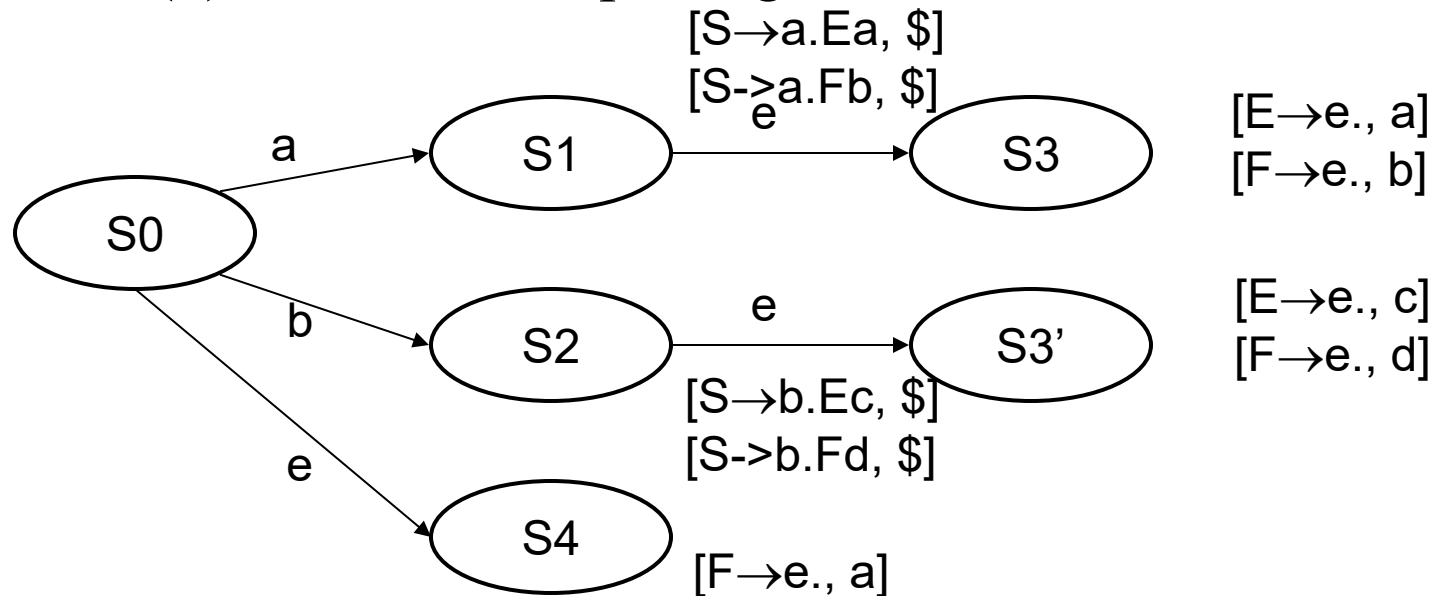
□ Let's consider this Grammar G:

➤  $S \rightarrow aEa \mid aFb \mid bEc \mid bFd \mid Fa$

$E \rightarrow e$

$F \rightarrow e$

□ But LR(1) thanks to S3 splitting to S3 and S3'



# SLR(k) $\subset$ LALR(k)

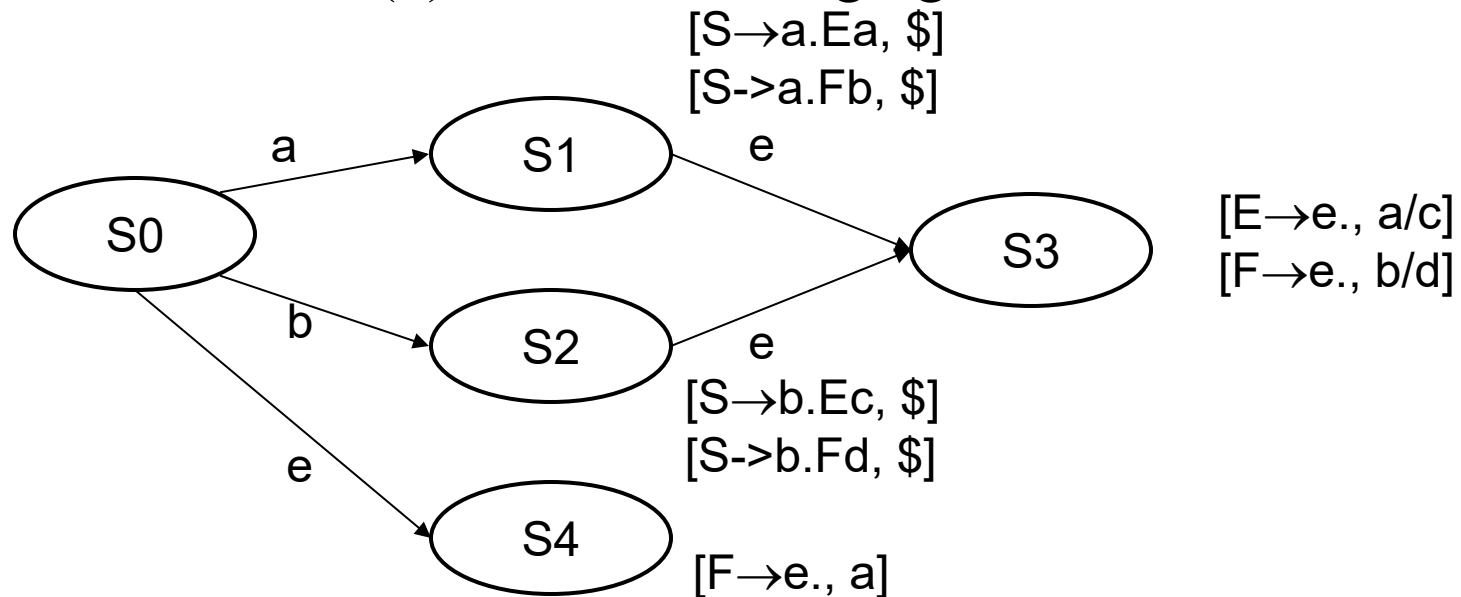
□ Let's consider this Grammar G:

➤  $S \rightarrow aEa \mid aFb \mid bEc \mid bFd \mid Fa$

$E \rightarrow e$

$F \rightarrow e$

□ And also LALR(1) even after merging back S3' and S3





# $LL(k) \subset LR(k)$

- ❑ LL(k) parser, each expansion  $A \rightarrow \alpha$  is decided on the basis of
  - Current non-terminal at the top of the stack
    - Which LHS to produce
  - k terminals of lookahead at *beginning* of RHS
    - Must guess which RHS by peeking at first few terminals of RHS
- ❑ LR(k) parser, each reduction  $A \rightarrow \alpha \bullet$  is decided on the basis of
  - RHS at the top of the stack
    - Can postpone choice of RHS until entire RHS is seen
    - Common left factor is okay – waits until entire RHS is seen anyway
    - Left recursion is okay – does not impede forming RHS for reduction
  - k terminals of lookahead *beyond* RHS
    - Can decide on RHS after looking at entire RHS plus lookahead

# LL(k) $\neq$ SLR(k)

- ❑ Neither is strictly more powerful than the other
- ❑ Advantage of SLR: can delay decision until entire RHS seen
  - LL must decide RHS with a few symbols of lookahead
- ❑ Disadvantage of SLR: lookahead applied out of context
  - Consider grammar:  $S \rightarrow Bb \mid Cc \mid aBc$ ,  $B \rightarrow \varepsilon$ ,  $C \rightarrow \varepsilon$
  - Initial state  $S_0 = \{S \rightarrow . Bb \mid . Cc \mid . aBc, B \rightarrow ., C \rightarrow .\}$
  - For SLR(1), reduce-reduce conflict on  $B \rightarrow .$  and  $C \rightarrow .$ 
    - $\text{Follow}(B) = \{b, \textcolor{red}{c}\}$  and  $\text{Follow}(C) = \{\textcolor{red}{c}\}$
  - For LL(1), no conflict
    - $\text{First}(Bb) = \{b\}$ ,  $\text{First}(Cc) = \{c\}$ ,  $\text{First}(aBC) = \{a\}$
- ❑ For the same reason, LL  $\neq$  LALR

$$LL(0) \subset LR(0) \equiv LALR(0) \equiv SLR(0)$$

□  $LR(0) \equiv LALR(0) \equiv SLR(0)$

- $LR(0) \equiv LALR(0) \equiv SLR(0)$  since lookahead is meaningless.
- If a state has a reduce item, there can be no other items.  
(If there is, it will result in a conflict with the reduce action.)
- This makes grammars very restrictive and unusable.

□  $LL(0) \subset LR(0)$

- $LL(0)$  can only have one RHS per non-terminal to avoid conflict.
- $LR(0)$  can still have multiple RHSs per non-terminal.
- E.g.  $S \rightarrow a \mid b$  is not  $LL(0)$  but is  $LR(0)$ .

# $L(\text{GLR}) \equiv L(\text{CFG})$

□ GLR: Generalized LR parser where  $L(\text{GLR}) \equiv L(\text{CFG})$

➤ “Parsing Techniques. A Practical Guide.” by Grune et al. (2008)

<https://link.springer.com/book/10.1007/978-0-387-68954-8>

➤ An LR family parser that does the following on a conflict

1. Fork the parse stack and follow each action separately
2. If forked parse stack results in a parse error, discard it

➤ Uses any LR table (e.g. SLR, LALR, Canonical LR)

➤ GNU Bison: an implementation of GLR

<https://www.gnu.org/software/bison/>

# $L(\text{GLL}) \equiv L(\text{CFG})$

❑ Is there a generalized LL parser that can parse all CFGs?

➤ Recall, LL parsers have trouble with left-recursion

❑ GLL: Generalize LL parser

➤ “GLL Parsing” by Scott et al. (2010)

<https://www.sciencedirect.com/science/article/pii/S1571066110001209>

➤ How does it deal with left-recursion?

- Idea similar to GLR: fork stack on every conflict due to left-recursion (And try out all numbers of left-recursion until parse is successful)
- Difference is, you can potentially end up with many more forked stacks
- Developed “Graph Structured Stack” to minimize stack memory

➤ GoGLL: an implementation of GLL

<https://github.com/goccmack/gogll>

---

# Using Automatic Tools

## -- YACC

---

Pitt, CS 1622

# Using a Parser Generator

- ❑ YACC is an LALR(1) parser generator
  - YACC: Yet Another Compiler-Compiler
- ❑ YACC constructs an LALR(1) table and reports an error when a table entry is multiply defined
  - A shift and a reduce – reports shift/reduce conflict
  - Multiple reduces – reports reduce/reduce conflict
  - Most conflicts are due to ambiguous grammars
  - Must resolve conflicts
    - By specifying associativity or precedence rules
    - By modifying the grammar
    - YACC outputs detail about where the conflict occurred (by default, in the file “y.output”)

# Shift/Reduce Conflicts

- Typically due to ambiguities in the grammar
- Classic example: the dangling else

$S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \mid \text{OTHER}$

will have DFA state containing

$[S \rightarrow \text{if } E \text{ then } S. , \text{else}]$

$[S \rightarrow \text{if } E \text{ then } S. \text{ else } S , \text{else}]$

so on ‘else’ we can shift or reduce

- Default (YACC, bison, etc.) behavior is to shift
  - Default behavior is the correct one in this case
  - Better not to rely on this and remove ambiguity



# More Shift/Reduce Conflicts

□ Consider the ambiguous grammar

$$E \rightarrow E + E \mid E * E \mid \text{int}$$

we will have the states containing

$$\begin{array}{ll} [E \rightarrow E * . E, +/*] & [E \rightarrow E * E . , +/*] \\ [E \rightarrow . E + E, +/*] & \xRightarrow{E} [E \rightarrow E . + E, +/*] \end{array}$$

...

...

Again we have a shift/reduce conflict on input +

- In this case, we need to reduce (\* is higher than +)
- Easy (better) solution: declare precedence rules for \* and +
- Hard solution: rewrite grammar to be unambiguous

# More Shift/Reduce Conflicts

## □ Declaring precedence and associativity in YACC

%left '+' '-'

%left '\*' '/'

### ➤ Interpretation:

- +, -, \*, / are left associative
- +, - have lower precedence compared to \*, /  
(associativity declarations are in the order of increasing precedence)
- Precedence of a candidate rule for reduction is the precedence of the last terminal in that rule (e.g. For 'E → E+E .' , level is same as '+')

### ➤ Resolve shift/reduce conflict with a shift if:

- No precedence declared for either rule or terminal
- Input terminal has higher precedence than the rule
- The precedence levels are the same and right associative

# Use Precedence to Solve S/R Conflict

$$\begin{array}{ccc} [E \rightarrow E^* \cdot E, +/*] & & [E \rightarrow E^* E \cdot, +/*] \\ [E \rightarrow \cdot E + E, +/*] & \xRightarrow{E} & [E \rightarrow E \cdot + E, +/*] \\ \dots & & \dots \end{array}$$

- we will choose reduce because precedence of rule  $E \rightarrow E^* E$  is higher than that of terminal  $+$

$$\begin{array}{ccc} [E \rightarrow E + \cdot E, +/*] & & [E \rightarrow E + E \cdot, +/*] \\ [E \rightarrow \cdot E + E, +/*] & \xRightarrow{E} & [E \rightarrow E \cdot + E, +/*] \\ \dots & & \dots \end{array}$$

- we will choose reduce because  $E \rightarrow E + E$  and  $+$  have the same precedence and  $+$  is left-associative

## ❑ Back to our dangling else example

$[S \rightarrow \text{if } E \text{ then } S. , \text{ else}]$

$[S \rightarrow \text{if } E \text{ then } S. \text{ else } S, \text{ else}]$

- Can also eliminate conflict by precedence declarations:  
%nonassoc 'then'  
%nonassoc 'else'
- Perhaps less intuitive compared to arithmetic precedence
- Use precedence only if it enhances readability of code

# Reduce/Reduce Conflicts

- Usually due to ambiguity in the grammar

- Example: a sequence of identifiers

$$S \rightarrow \varepsilon \mid \text{id} \mid \text{id } S$$

There are two rightmost derivations for the string 'id'

$$S \Rightarrow \text{id}$$

$$S \Rightarrow \text{id } S \Rightarrow \text{id}$$

How does this ambiguous grammar confuse the parser?

# Reduce/Reduce Conflicts

□ Consider the states

$[S' \rightarrow .S, \$]$

$[S \rightarrow ., \$]$

$[S \rightarrow .id, \$]$

$[S \rightarrow .id S, \$]$

$\xRightarrow{id}$

$[S \rightarrow id. , \$]$

$[S \rightarrow id.S, \$]$

$[S \rightarrow ., \$]$

$[S \rightarrow .id, \$]$

$[S \rightarrow .id S, \$]$

Reduce/reduce conflict on input “id\$”

$S' \Rightarrow S \Rightarrow id$

$S' \Rightarrow S \Rightarrow id S \Rightarrow id$

Remove ambiguity by rewriting the grammar:  $S \rightarrow \epsilon \mid id S$

# Semantic Actions

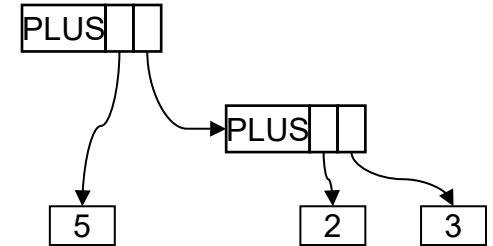
- ❑ Semantic actions are implemented for LR parsing
  - keep attributes on the semantic stack – parallel to the parse stack
    - on shift  $a$ , push attribute for  $a$  on semantic stack
    - on reduce  $X \rightarrow \alpha$ 
      - pop attributes for  $\alpha$
      - compute attribute for  $X$  based on attributes for  $\alpha$
      - push it on the semantic stack
- ❑ Creating an AST
  - Bottom up
  - Create leaf node from attribute values of token(s) in RHS
  - Create internal node from subtree(s) passed on from RHS

# Performing Semantic Actions

❑ Example 1: attribute is value of expression

$$E \rightarrow T + E \qquad \{\$ \$ = \$1 + \$2;\}$$
$$|T\rangle = \frac{1}{\sqrt{2}}(|\uparrow\downarrow\rangle + |\downarrow\uparrow\rangle)$$
$$T \rightarrow \text{int} * T \qquad \{\$ \$ = \$1 * \$2;\}$$
$$| \text{int} \quad \{ \$\$ = \$1; \}$$

consider the parsing of the string  $3 * 5 + 8$



❑ Example 2: attribute is AST for expression

$$E \rightarrow \text{int} \quad \{\$ \$ = \text{mkleaf}(\$1);\}$$

E+E	{ \$\$ = mktree(plus, \$1, \$2); }
-----	------------------------------------

(E)  $\{\$ = \$1;\}$

- a bottom-up evaluation of the ast attribute:

```
E.ast = mktree(plus, mkleaf(5),
               mktree(plus, mkleaf(2), mkleaf(3) ) )
```