2CS701
Compiler Construction

Prof Monika Shah

Nirma University

Phase 2 : Syntax Analysis
Bottom-Up Parsing

Prof Monika Shah

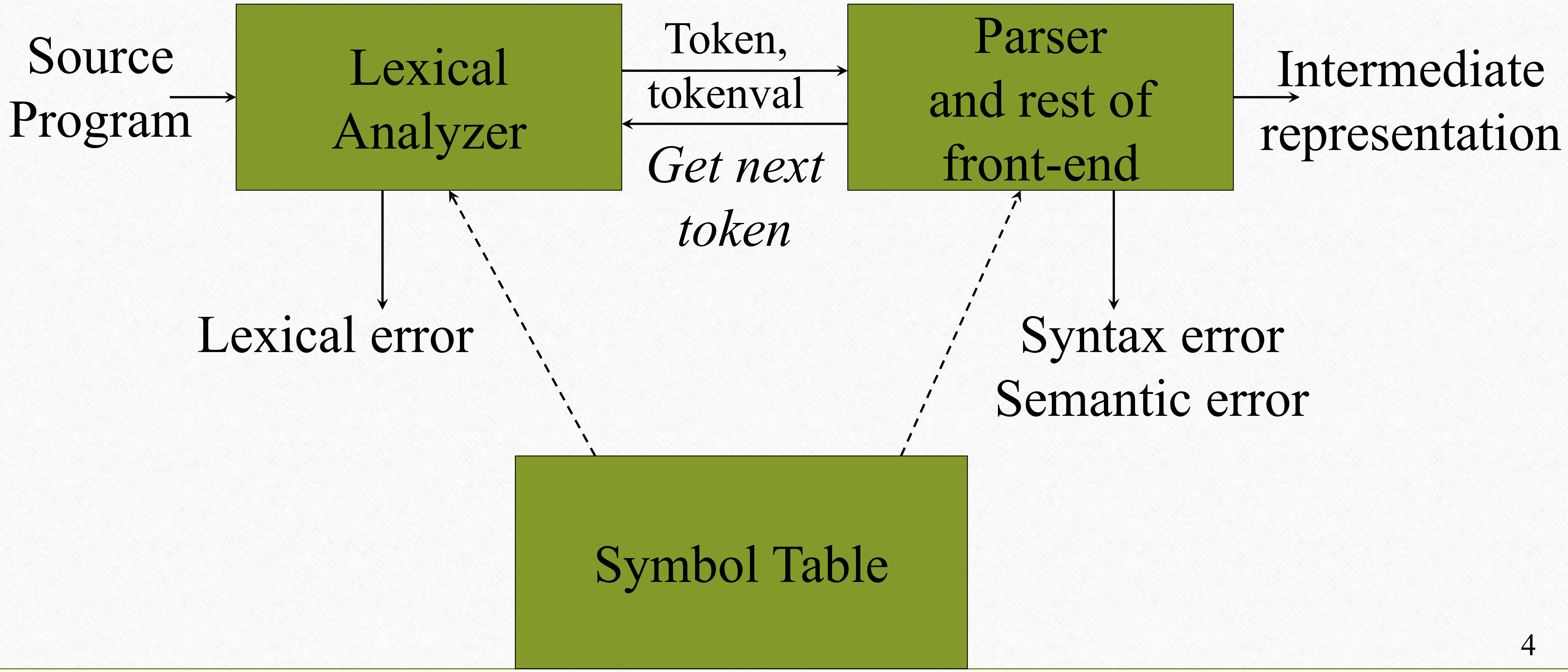
Nirma University

Ref : Ch.4 Compilers Principles, Techniques, and Tools by Alfred Aho, Ravi Sethi, and Jeffrey Ullman

Glimpse

- Role of the Syntax Analyzer (Parser)
- Type of Parsers
- Introduction to Bottom up Parsing
- Bottom Up Parsers
- Sentential Form
- Shift Reduce Parsing
- Conflicts in Shift Reduce parsing
- LR Parsing
- LR parsing table construction
- Other topics
  - Compaction of parsing table
  - Handling Ambiguity
  - Error recovery

Position of a Parser in the Compiler Model



Role of Syntax Analyzer

- Verify Syntax
- Report Syntax errors accurately with location
- Update Symbol Table
  - E.g. Update data type of variables
- Invoke Semantic Actions
  - Static checking i.e. Type checking of identifiers, expressions, functions
  - Syntax directed translation of source code to intermediate code generation

Bottom-Up Parsing

- A more powerful parsing technology
- LR grammars – more expressive than LL
  - Construct right-most derivation of program
  - Also work for Left-recursive grammars, virtually all programming languages are left-recursive
  - Easier to express syntax
- Shift-reduce parsers
  - Parsers for LR grammars
  - Automatic parser generators (yacc, bison)

Bottom-Up Parsing

- Right-most derivation – Backward
  - Start with the tokens
  - End with the start symbol
  - Match substring on RHS of production, replace by LHS

S → S + E | E
E → num | (S)

(1+2+(3+4))+5
← (E+2+(3+4)+5
← (S+2+(3+4))+5
← (S+E+(3+4))+5
← (S+(3+4))+5
← (S+(E+4))+5
← (S+(S+4))+5
← (S+(S+E))+5
← (S+(S))+5
← (S+E)+5
← (S)+5
← E+5
← S+E
← S

S → S + E | E

E → num | (S)

Top Down Parsing vs Bottom-Up Parsing

Left most derivation (Top to bottom)

← **S**
← **S**+E
← **E**+E
← (**S**)+E
← (**S**+E)+E
← (**S**+E+E)+E
← (**E**+E+E)+E
← (1+**E**+E)+E
← (1+2+**E**)+E
← (1+2+(**S**))+E
← (1+2+(**S**+E))+E
← (1+2+(**E**+E))+E
← (1+2+(3+**E**))+E
←(1+2+(3+4))+**E**
←(1+2+(3+4))+5

Rightmost derivation
Top-Down parsing in reverse order

← **S**
← S+**E**
← **S** + 5
← **E**+5
← (**S**)+5
← (S+**E**)+5
← (S+(**S**))+5
← (S+(S+**E**))+5
← (S+(**S**+4))+5
← (S+(**E**+4))+5
← (**S**+(3+4))+5
← (S+**E**+(3+4))+5
← (**S**+2+(3+4))+5
← (**E**+2+(3+4)+5
←(1+2+(3+4))+5

Bottom-Up Parsing

(1+2+(3+4))+5

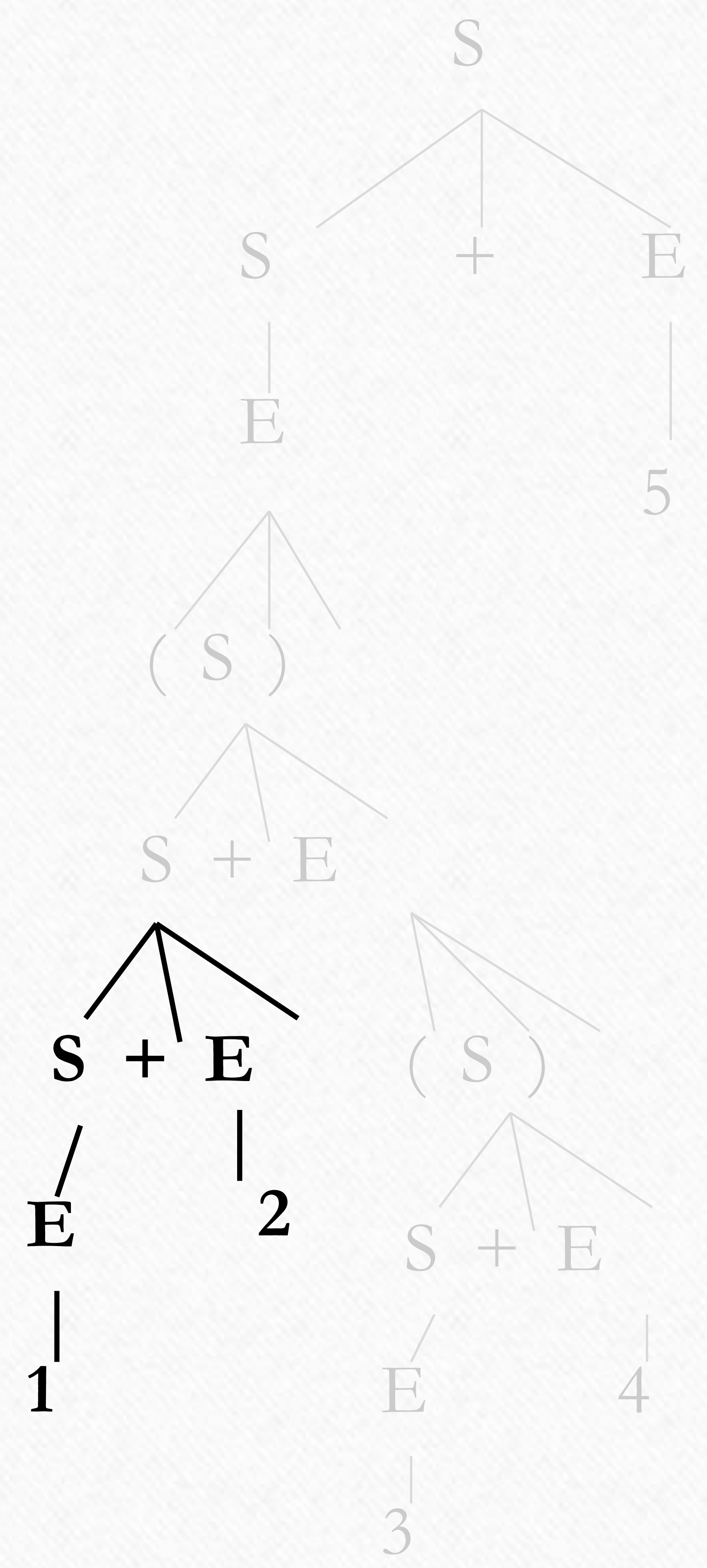
$\leftarrow$ (E+2+(3+4))+5

$\leftarrow$ (S+2+(3+4))+5

$\leftarrow$ (S+E+(3+4))+5

S $\rightarrow$ S + E | E
E $\rightarrow$ num | (S)

Advantage of bottom-up parsing: can postpone the selection of productions until more of the input is scanned

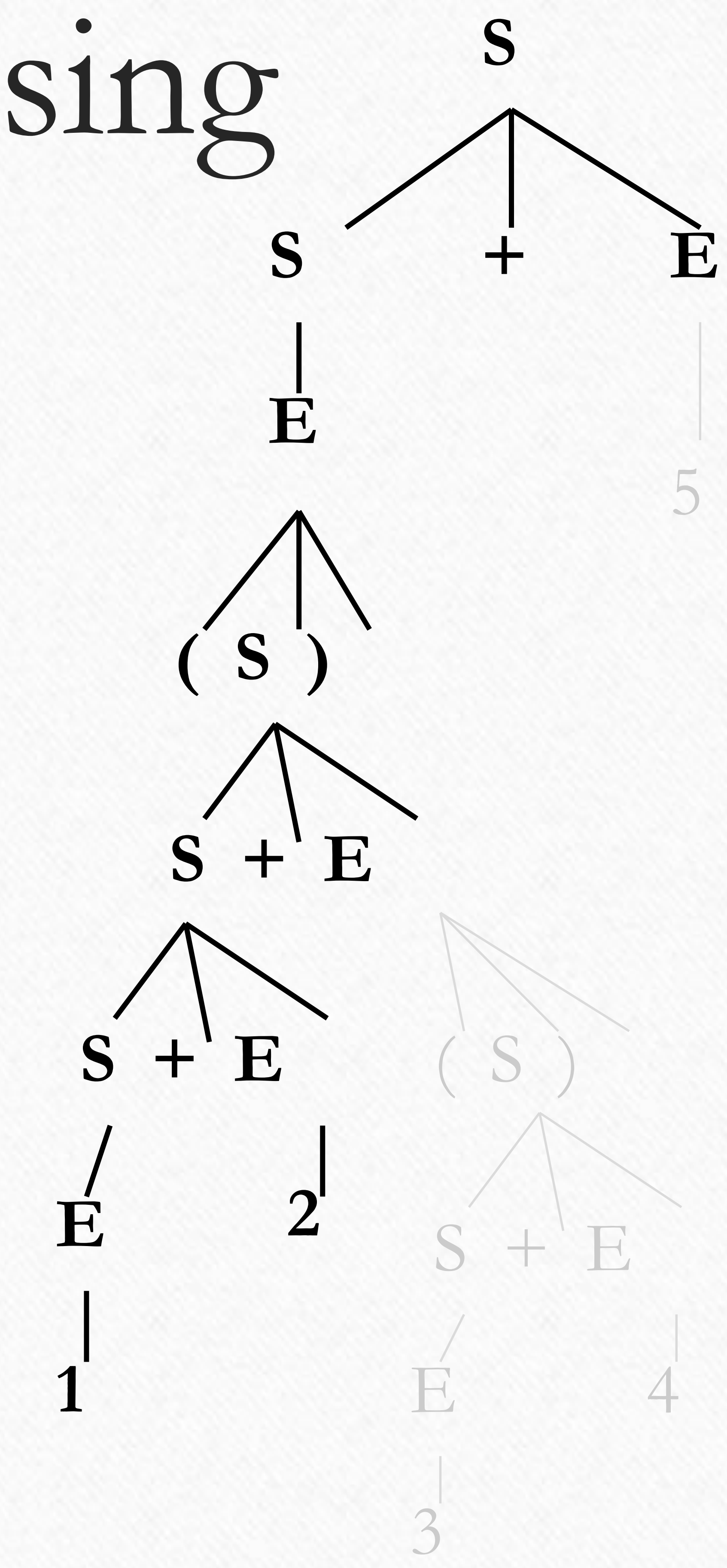


Top-Down Parsing

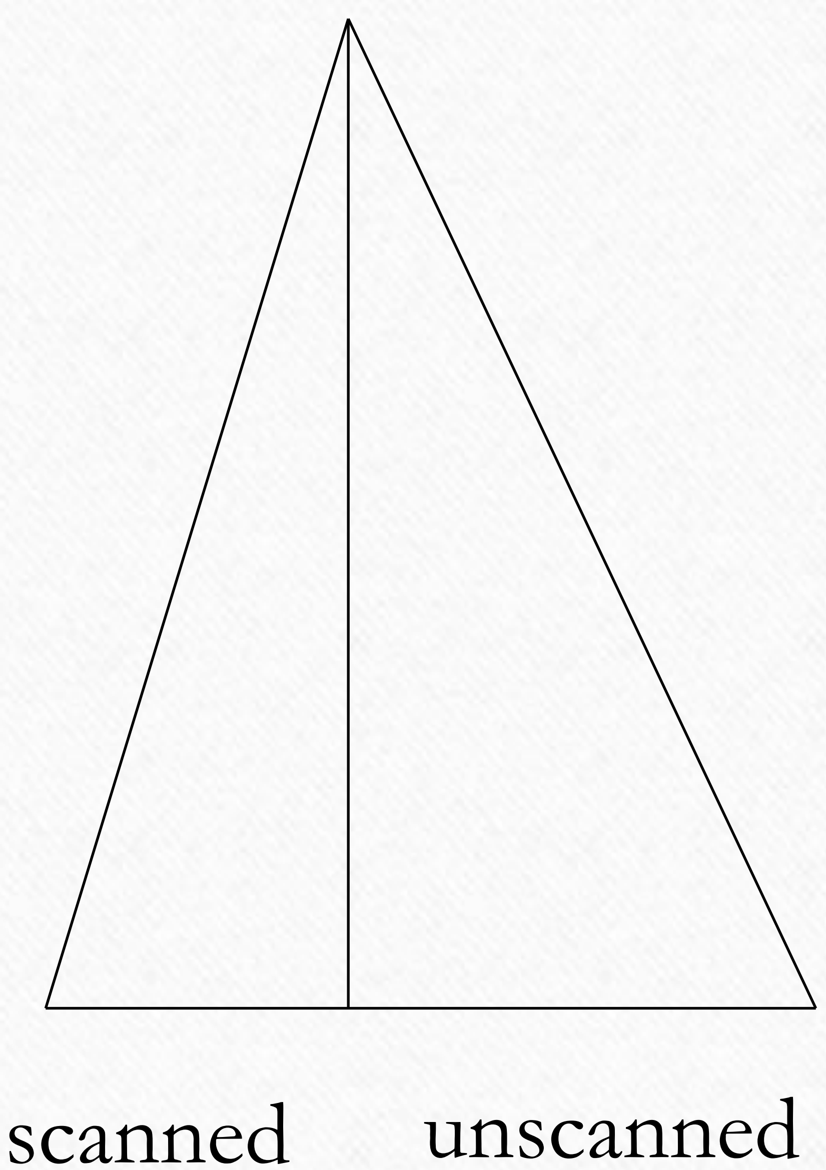
S → S+E → E+E → (S)+E → (S+E)+E
→ (S+E+E)+E → (E+E+E)+E
→ (1+E+E)+E → (1+2+E)+E …

S → S + E | E
E → num | (S)

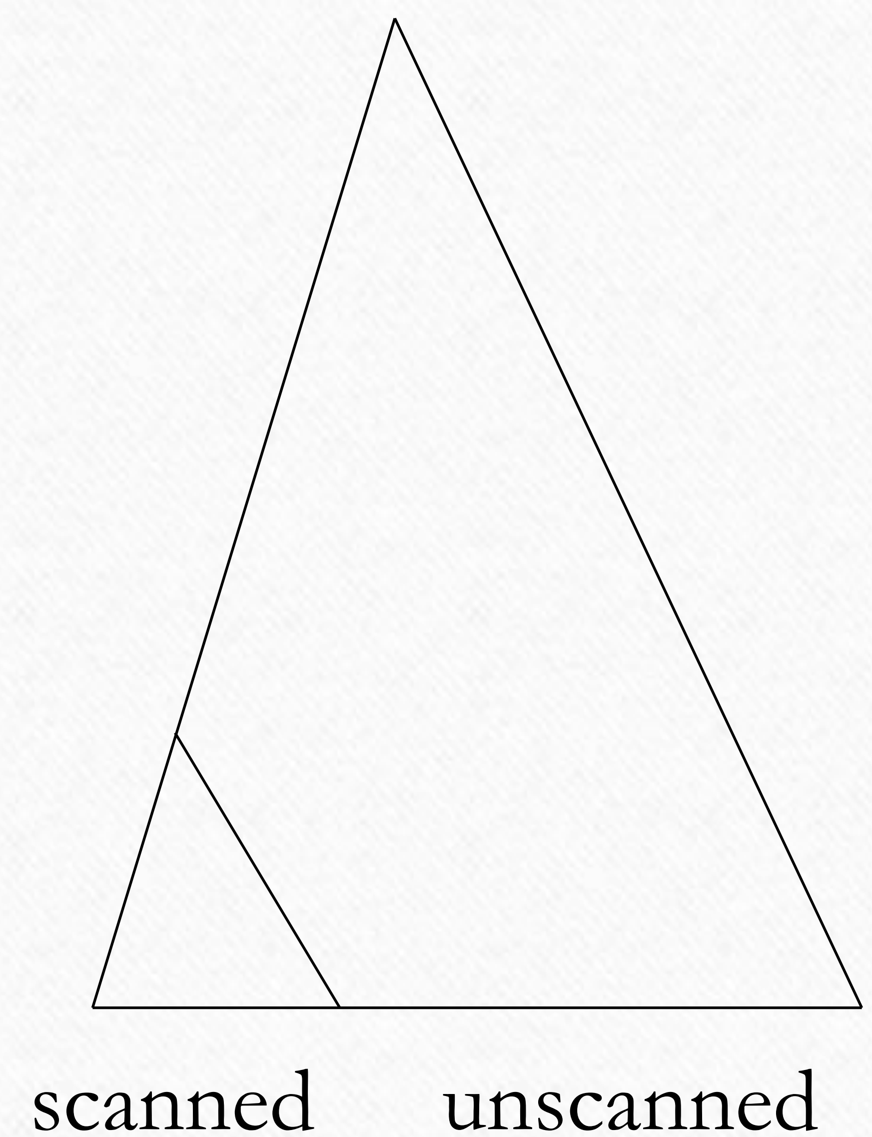
In left-most derivation, entire tree above token (2) has been expanded when encountered



Top-Down vs Bottom-Up



Top-down



Bottom-up

Bottom-up: Don't need to figure out as much of the parse tree for a given amount of input → More time to decide what rules to apply

Bottom-Up Parsing

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

Sentential Form

- For a grammar G, with start symbol S, any string α such that $S \Rightarrow^* \alpha$ is called a sentential form
- A left-sentential form is a sentential form that occurs in the leftmost derivation of some sentence.
- A right-sentential form is a sentential form that occurs in the rightmost derivation of some sentence.

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

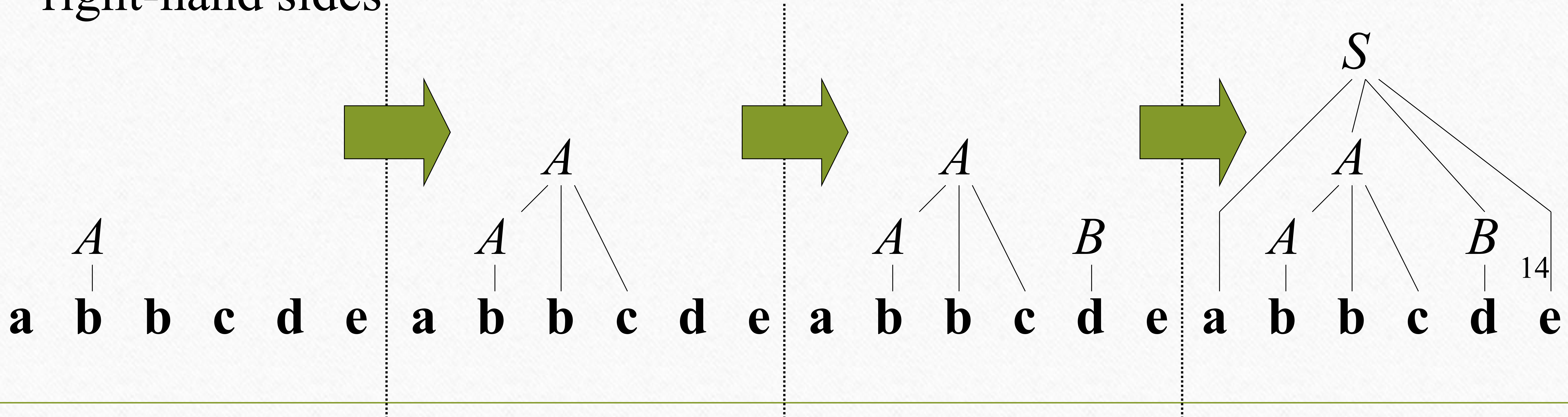
Shift-reduce corresponds to a rightmost derivation:

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

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

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

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



Handles

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

Grammar:

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

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

$B \rightarrow \mathbf{d}$

Sentential Form

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

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

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

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

S

Handle

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

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

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

… ?

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

Shift-Reduce Parsing

- Parsing actions: A sequence of shift and reduce operations
- Parser state: A stack of terminals and non-terminals (grows to the right)
- Current derivation step = stack + input

Derivation step	stack	Unconsumed input
(1+2+(3+4))+5 ←		(1+2+(3+4))+5
(E+2+(3+4))+5 ←	(E	+2+(3+4))+5
(S+2+(3+4))+5 ←	(S	+2+(3+4))+5
(S+E+(3+4))+5 ←	(S+E	+(3+4))+5
...		

Shift-Reduce Actions

stack	input	action
(	1+2+(3+4))+5	shift 1
(1	+2+(3+4))+5	

- Parsing is a sequence of shifts and reduces
- **Shift:** move look-ahead token to stack

stack	input	action
(S+E	+(3+4))+5	reduce S → S+ E
(S	+(3+4))+5	

- **Reduce:** Replace symbols β from top of stack with non-terminal symbol X corresponding to the production: X→ β (e.g., pop β, push X)

Stack Implementation of Shift-Reduce Parsing

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

Grammar:
$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow$ **id**

Find handles to reduce

How to resolve conflicts?

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

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

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

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

Shift-Reduce Parsing: Reduce-Reduce Conflicts

Grammar:

$C \to A\ B$

$A \to \mathbf{a}$

$B \to \mathbf{a}$

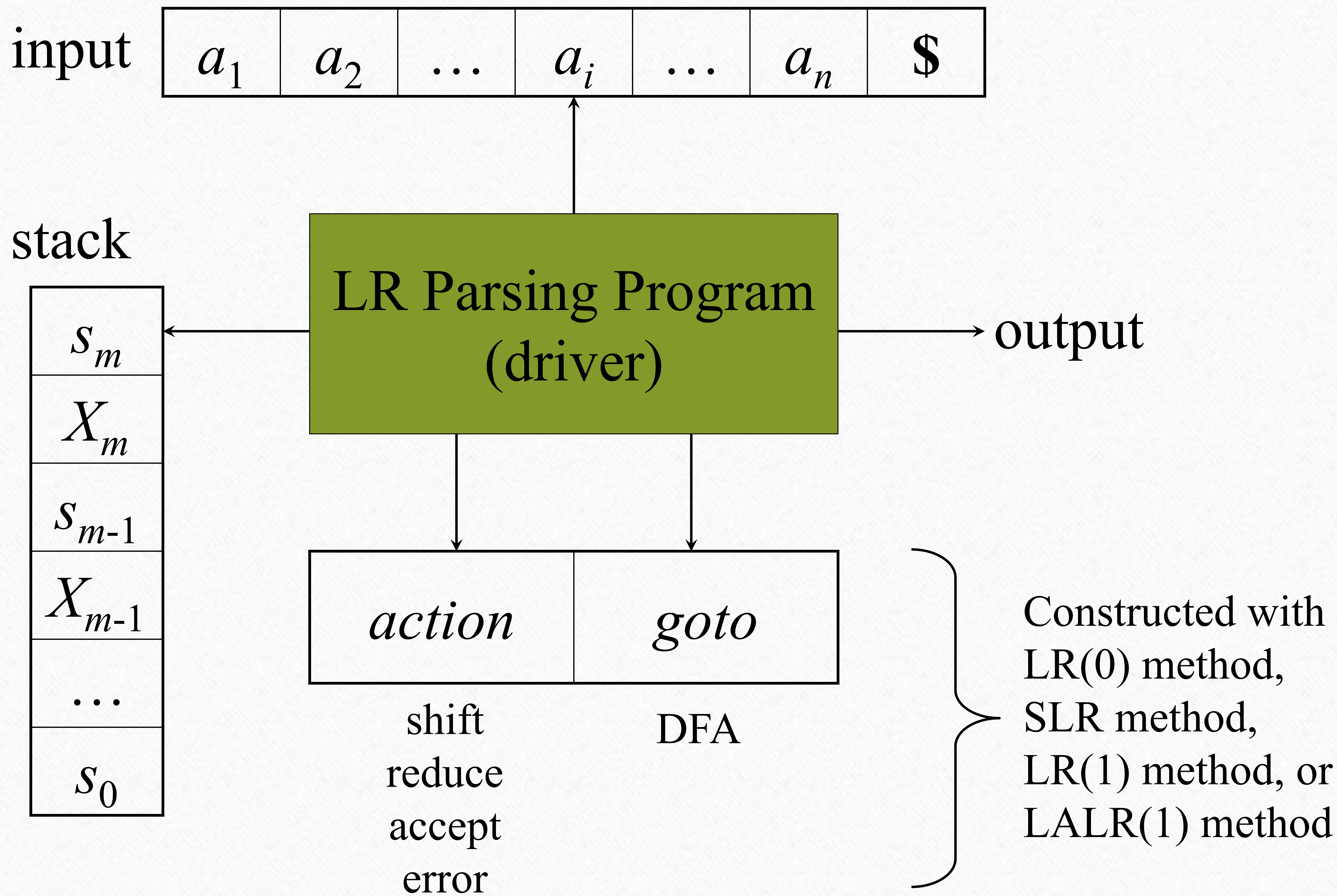
Stack	Input	Action
$	**aa$**	shift
$<ins>a</ins>	**a$**	reduce $A \to$ **a** <ins>or</ins> $B \to$ **a** ?

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

Parsing using LR(k) Parser

- **L**R(k) Input Scanning from Left to right.
- ➔ Suitable Data structure for Input : **Queue**
- L**R**(k) **R**ight Most Derivation in reverse order
- LR(**k**) Number of Look-ahead used to parse properly : **K**
- LR(K) Parsers
- LR(0) Parser
- SLR Parser
- LR(1) Parser
- LALR(1) Parser

Model of an LR Parser



LR Parsing (Driver)

Configuration (= LR parser state):

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

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 → 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 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 Parsing

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			

Grammar:

1. $E \rightarrow E + T$ 2. $E \rightarrow T$ 3. $T \rightarrow T * F$

4. $T \rightarrow F$ 5. $F \rightarrow (E)$ 6. $F \rightarrow \textbf{id}$

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

Construction of LR Parser

LR(0) Summary

- LR(0) state: set of LR(0) items
- LR(0) item: a production with a dot in RHS
- Compute LR(0) states and build DFA
  - Use closure operation to compute states
  - Use goto operation to compute transitions
- Build LR(0) parsing table from the DFA
- Use LR(0) parsing table to determine whether to shift or reduce

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
  $A \to X\ Y\ Z$
  has four items:
  $[A \to \bullet\ X\ Y\ Z]$
  $[A \to X \bullet Y\ Z]$
  $[A \to X\ Y \bullet Z]$
  $[A \to X\ Y\ Z \bullet]$
- Note that production $A \to \varepsilon$ has one item $[A \to \bullet]$

Constructing LR(0) 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 input
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 $[S' \rightarrow \bullet S]$

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

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

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)

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

$\{ [E' \to \bullet E] \}$

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

$\{ [E' \to \bullet E]$
$[E \to \bullet E + T]$
$[E \to \bullet T] \}$

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

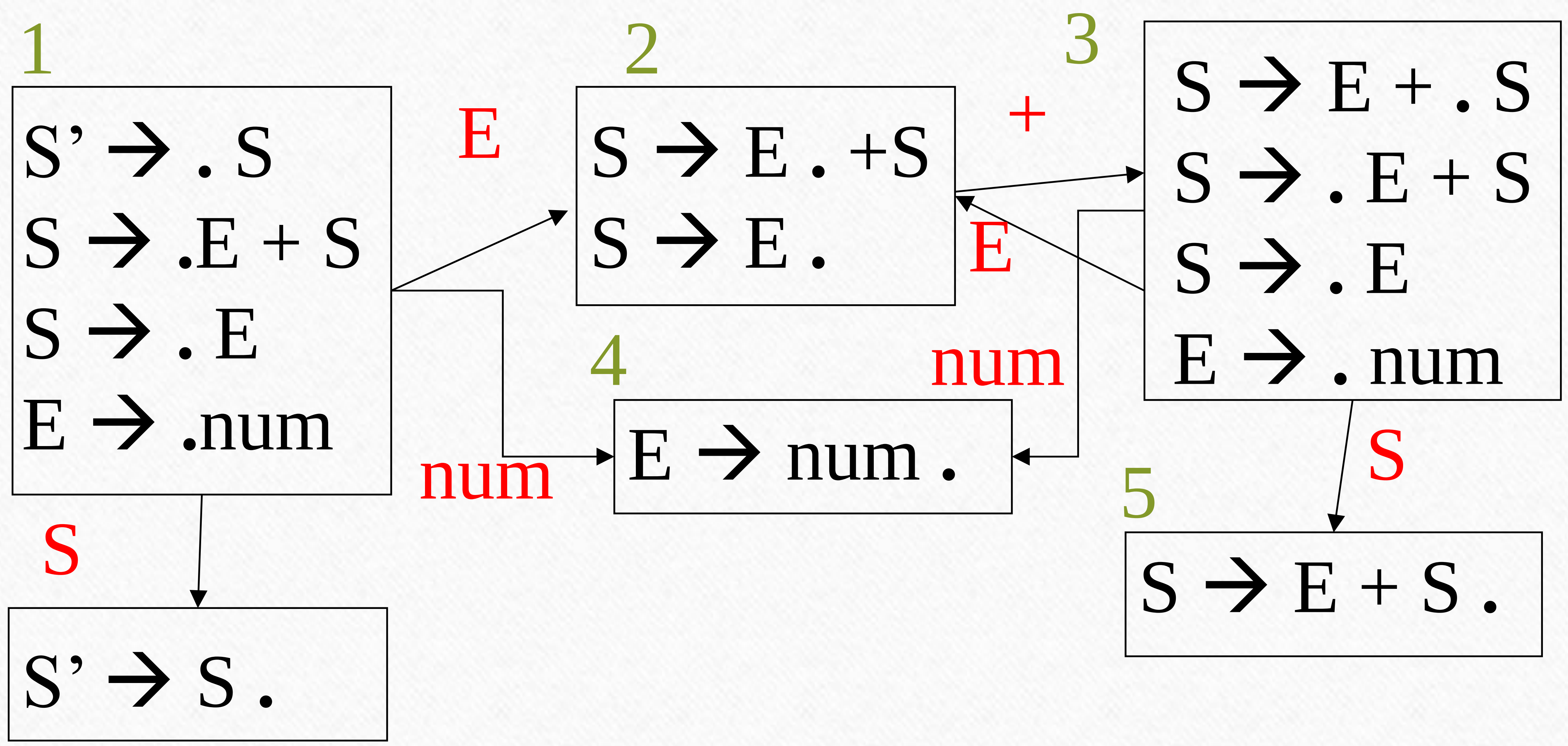
$\{ [E' \to \bullet E]$
$[E \to \bullet E + T]$
$[E \to \bullet T]$
$[T \to \bullet T * F]$
$[T \to \bullet F] \}$

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

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

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

LR(0) Parsing Table



<u>Grammar</u>
S → E + S | E
E → num

Shift or reduce in state 2?

	num	+	$	E	S
1	s4			g2	g6
2	S→E	s3/S→E	S→E		

Solve Conflict With Lookahead

- 3 popular techniques for employing lookahead of 1 symbol with bottom-up parsing
  - SLR – Simple LR
  - LALR – LookAhead LR
  - LR(1)
- Each as a different means of utilizing the lookahead
  - Results in different processing capabilities

LR(0) Limitations

- An LR(0) machine only works if states with reduce actions have a single reduce action
- With a more complex grammar, construction gives states with shift/reduce or reduce/reduce conflicts
- Need to use lookahead to choose

OK	shift/reduce	reduce/reduce
L → L , S .	L → L , S . S → S . , L	L → S , L . L → S .

A Non-LR(0) Grammar

- Grammar for addition of numbers
  - S → S + E | E
  - E → num
- Left-associative version is LR(0)
- Right-associative is not LR(0)
  - S → E + S | E
  - E → num

SLR Parsing

- SLR Parsing = Easy extension of LR(0)
  - For each reduction X → β, look at next symbol C
  - Apply reduction only if C is not in FOLLOW(X)
- SLR parsing table eliminates some conflicts
  - Same as LR(0) table except reduction rows
  - Adds reductions X → β only in the columns of symbols in FOLLOW(X)

Grammar
S → E + S | E
E → num

LR(0) Parse Table

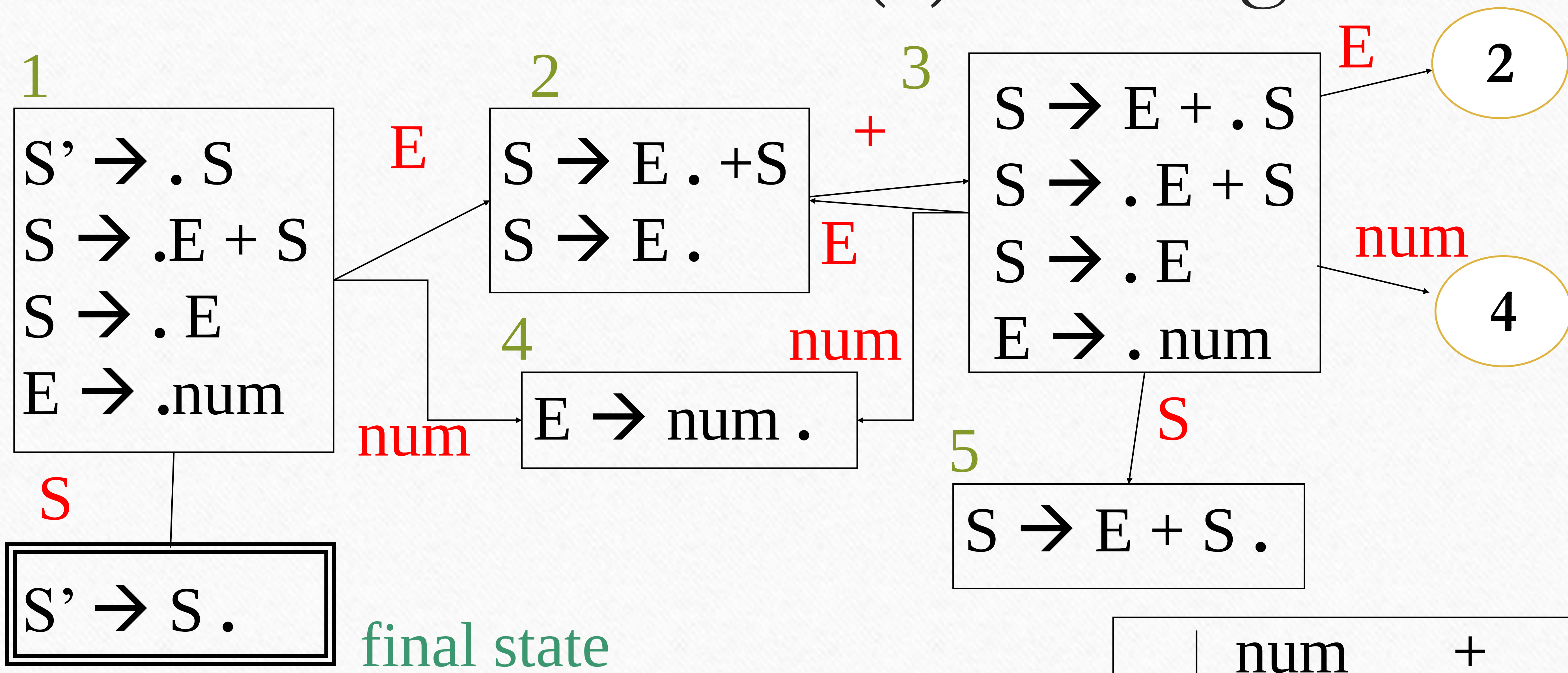
	num	+	$	E	S
1	s4			g2	g6
2	S→E	s3/S→E	S→E		

Example: FOLLOW(S) = {$}

	num	+	$	E	S
1	s4			g2	g6
2		s3	S→E		

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

LR(0) Parsing Table



Grammar
S → E + S | E
E → num

SLR Parse Table

	num	+	$	E	S
1	s4			g2	g6
2		s3	S→E		
3	s4			g2	g5
4		E→num	E→num		
5			S→E+S		
6			s7		
7			accept		

LR(0) Parse Table

	num	+	$	E	S
1	s4			g2	g6
2	S→E	s3/S→E	S→E		

SLR Parsing Table

Grammar
S → E + S | E
E → num

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

	num	+	$	E	S
1	s4			g2	g6
2		s3	S→E		
3	s4			g2	g5
4		E→num	E→num		
5			S→E+S		
6			s7		
7			accept		

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

Example SLR Parsing Table

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

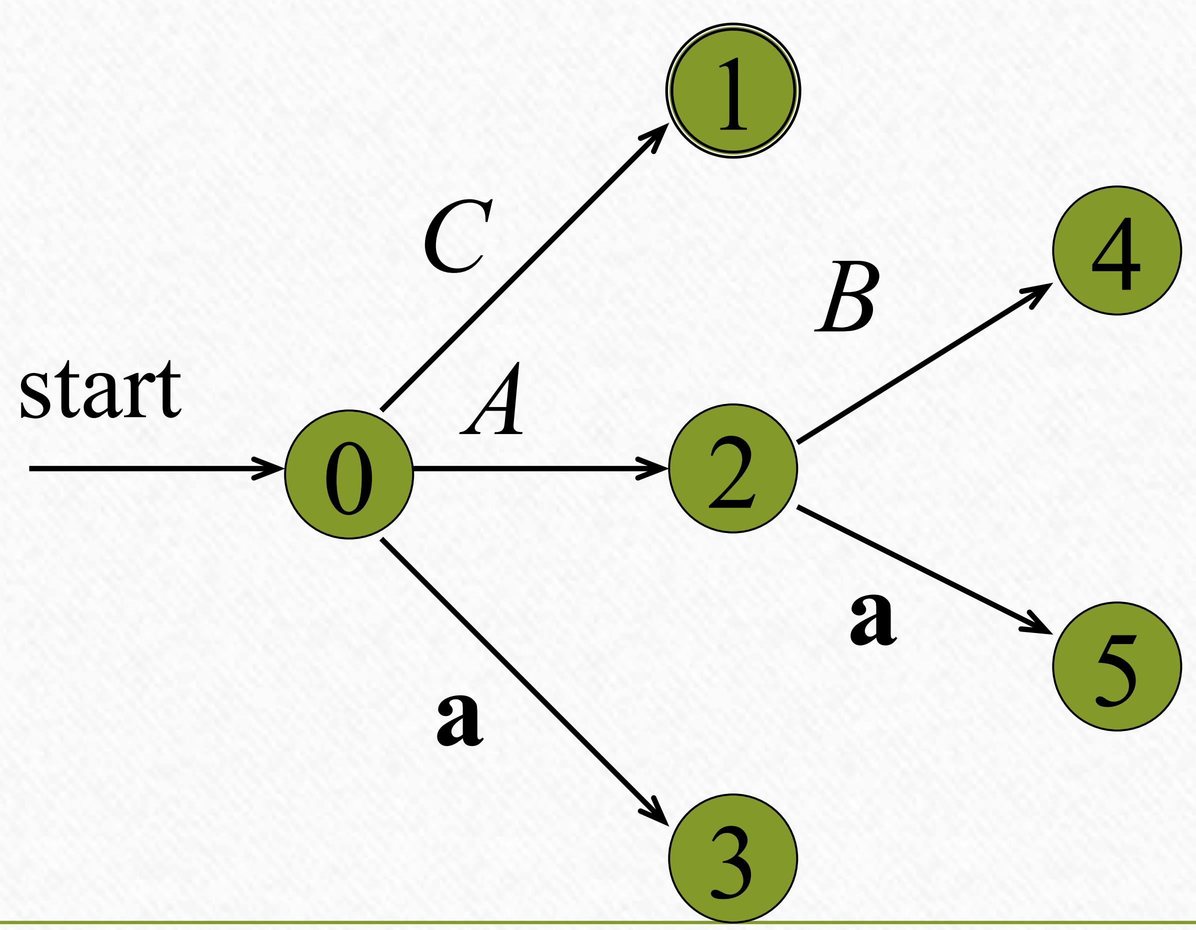
State I_1:
$C' \to C\bullet$

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



	a	**$**	*C*	*A*	*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. $B \to \mathbf{a}$

LR(0) vs SLR

- Difference in Parse Table construction
  - Reduce action for state having $X \rightarrow \beta\bullet$
  - **LR(0) :** For all terminal symbol
  - **SLR :** for all follow symbols of (X)
- SLR parsing more powerful than LR(0) parsing

Class Problem

Consider:

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow null$

$B \rightarrow null$

Is this grammar SLR ?

Is there any conflict ? Shift / reduce conflict ? Reduce/reduce conflict ?

Class Problem

Consider:

S → L = R
S → R
L → *R
L → ident
R → L

Think of L as l-value, R as r-value, and
* as a pointer dereference

When you create the states in the SLR(1) DFA,
2 of the states are the following:

S → L . = R R → L .	S → R .

Do you have any shift/reduce conflicts?

SLR and conflicts

- Every SLR grammar is unambiguous, but **not** every unambiguous grammar is SLR
- Consider for example the unambiguous grammar

$$S \rightarrow L = R \mid R$$
$$L \rightarrow * R \mid \mathbf{id}$$
$$R \rightarrow L$$

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

I_1:
$S' \rightarrow S\bullet$

I_2:
$S \rightarrow L\bullet=R$
$R \rightarrow L\bullet$

I_3:
$S \rightarrow R\bullet$

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

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

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

I_7:
$L \rightarrow *R\bullet$

I_8:
$R \rightarrow L\bullet$

I_9:
$S \rightarrow L=R\bullet$

$action[2,=]=s6$
$action[2,=]=r5$

no

Has no SLR parsing table

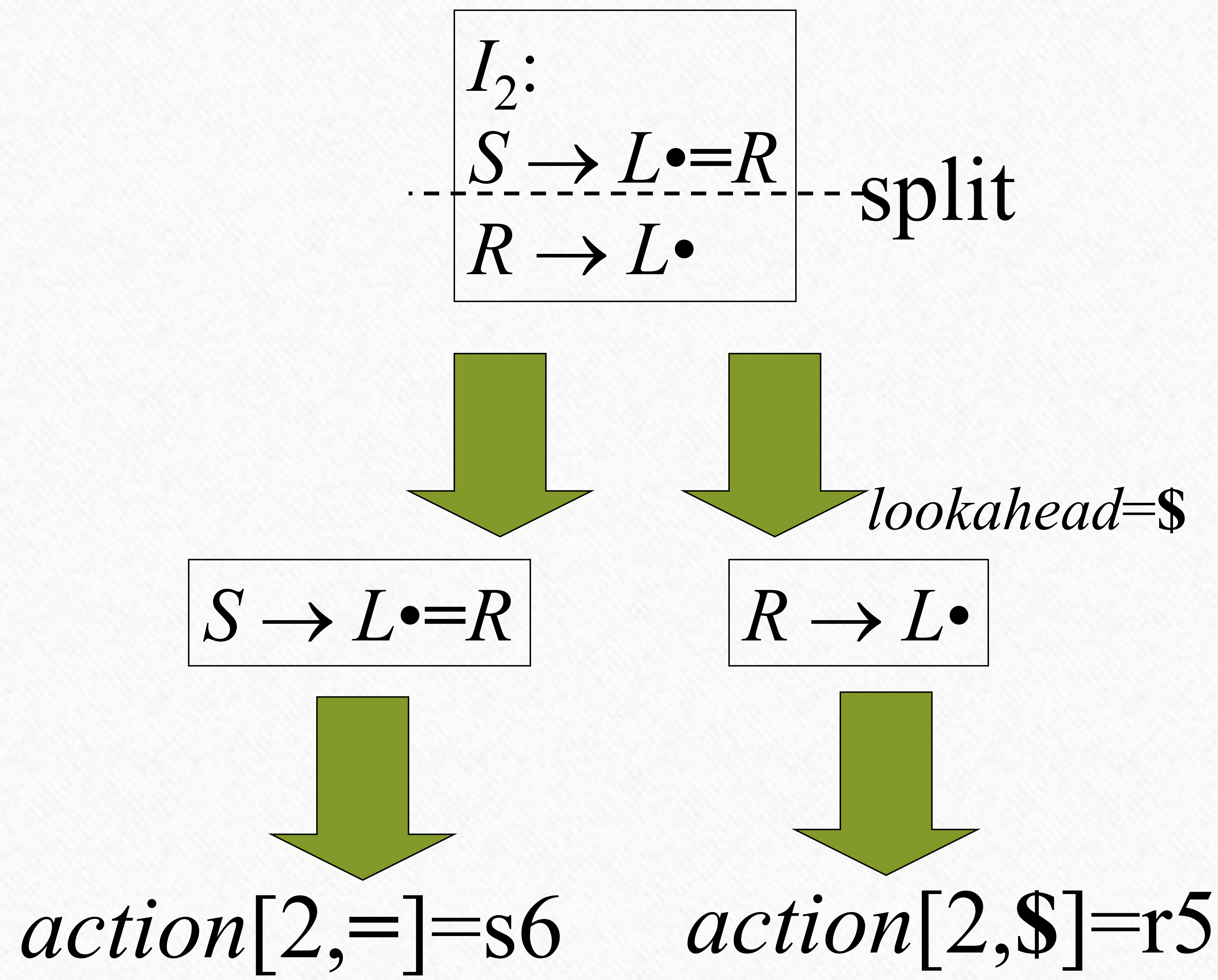
LR(1) Grammars

- 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. $S \to L = R$
  2. $\quad | \ R$
  3. $L \to * \ R$
  4. $\quad | \ \mathbf{id}$
  5. $R \to L$



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

The Closure Operation for LR(1) Items

1. Start with *closure*(I) = I
2. If $[A \to \alpha \bullet B\beta, a] \in$ *closure*(I) then for each production $B \to \gamma$ in the grammar and **each terminal $b \in$ FIRST(βa), add the item $[B \to \bullet\gamma, b]$** 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 $C = closure(\{[S' \rightarrow \bullet S, \$]\})$ (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(1) Items

- Unambiguous LR(1) grammar:

$$
\begin{aligned}
S &\rightarrow L = R \\
&\quad\mid R \\
L &\rightarrow * R \\
&\quad\mid \mathbf{id} \\
R &\rightarrow L
\end{aligned}
$$

- Augment with $S' \rightarrow S$
- LR(1) 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$(I_4,L)=I_8$
$[L \to \bullet *R,\ =/\$]$ 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$(I_6,R)=I_9$
$[R \to \bullet L,\ \$]$ goto$(I_6,L)=I_{10}$
$[L \to \bullet *R,\ \$]$ goto$(I_6,*)=I_{11}$
$[L \to \bullet \mathbf{id},\ \$]$ 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}: $[L \to *\bullet R,\ \$]$ goto$(I_{11},R)=I_{13}$
$[R \to \bullet L,\ \$]$ goto$(I_{11},L)=I_{10}$
$[L \to \bullet *R,\ \$]$ goto$(I_{11},*)=I_{11}$
$[L \to \bullet \mathbf{id},\ \$]$ 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	4
7			r4	r4			
8			r6	r6			
9				r2			
10				r6			
11	s12	s11				10	13
12				r5			
13				r4			

Summary of LR(0), SLR, LR(1) Parsing

- **Power :** LR(1) > SLR > LR(0)
- **States :** LR(0) = SLR
  - LR(1) > SLR

- More states => Space Complexity ↑

LALR(1) Grammars

- LR(1) parsing tables have many states
- LALR(1) parsing (Look-Ahead LR) combines LR(1) states 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

Example LALR(1) Grammar

- Unambiguous LR(1) grammar:

$$
\begin{aligned}
S &\to L \mathbf{=} R \\
&\mid R \\
L &\to \mathbf{*}\, R \\
&\mid \mathbf{id} \\
R &\to L
\end{aligned}
$$

- Augment with $S' \to S$
- LALR(1) items (next slide)

Constructing LALR(1) Parsing Tables

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

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

Shorthand for two items in the same set

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$(I_4,L)=I_8$
$[L \to \bullet{*}R,\ =/\$]$ 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$(I_6,R)=I_9$
$[R \to \bullet L,\ \$]$ goto$(I_6,L)=I_{10}$
$[L \to \bullet{*}R,\ \$]$ goto$(I_6,*)=I_{11}$
$[L \to \bullet\mathbf{id},\ \$]$ 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}: $[L \to {*}\bullet R,\ \$]$ goto$(I_{11},R)=I_{13}$
$[R \to \bullet L,\ \$]$ goto$(I_{11},L)=I_{10}$
$[L \to \bullet{*}R,\ \$]$ goto$(I_{11},*)=I_{11}$
$[L \to \bullet\mathbf{id},\ \$]$ goto$(I_{11},\mathbf{id})=I_{12}$

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

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

Example LALR(1) Parsing Table

	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			

	id	*	=	$	*S*	*L*	*R*
0	s512	s411			1	2	3
1				acc			
2			s6	r6			
3				r3			
411	s512	s411				810	713
512			r5	r5			
6	s512	s411				810	9
713			r4	r4			
810			r6	r6			
9				r2			

Example LALR(1) 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(1) if its LALR(1) parse table has no conflicts
  - LR(1) if its LR(1) parse table has no conflicts

Outline

- LR Parsing Summary
- Compaction of LR parsing table
- Deal with Ambiguous Grammar
- Error Recovery

LR Parsing Summary

	LR(0)	**SLR**	**LR(1)**	**LALR**
DFA Item	LR(0) item e.g. E→ E.+T	LR(0) item e.g. E→ E.+T	LR(1) item e.g. E→ E.+T, $	LR(1) item e.g. E→ E.+T, $
States	**LR(0) = SLR = LALR(1) < LR(1)**			
Power	**LR(0) < SLR < LALR < LR(1)**			
Shift action in Parse table	Shift input and Goto state for Terminal transition in DFA			
Goto action in parse table	Goto state for Non-Terminal transition in DFA			
Reduction action in parse table when entire production is parsed in DFA	For all terminal input set	For follow set Followset $\subseteq$ Inputset	For Look-ahead Look-ahead $\subseteq$ Followset	For Look-ahead
Accept and error entry in parse table	Same logic for all LR parsers			
Issue	Space Complexity			

Self Evaluation

- Show that the following grammar

  S → Aa | bAc | Bc | bBa

  A → d

  B → d

  is LR(1) but not LALR(1)
- Show that the following grammar

  S → Aa | bAc | dc | bda

  A → d

  Is LALR(1) but not SLR(1)
- Show that the following grammar

  S → AaAb | BbBa

  A → null

  B → null

  is LL(1) but not SLR(1)
- Check if the following grammar is SLR(1) or not

  S → E#

  E → T | E ; T

  T → Ta | null
- Check if the following grammar is LL(1) or not

  V → S#

  A → abS | c

  S → aAa | null

Compaction of LR parsing table

- Use technique of Sparse table compaction.

State	Symbol	Action
0	id	s5
	*	S4
	Any	error
1	$	Acc
	Any	Error
2	=	S6
	Any	r6
3	Any	r3
.		
.		
.		

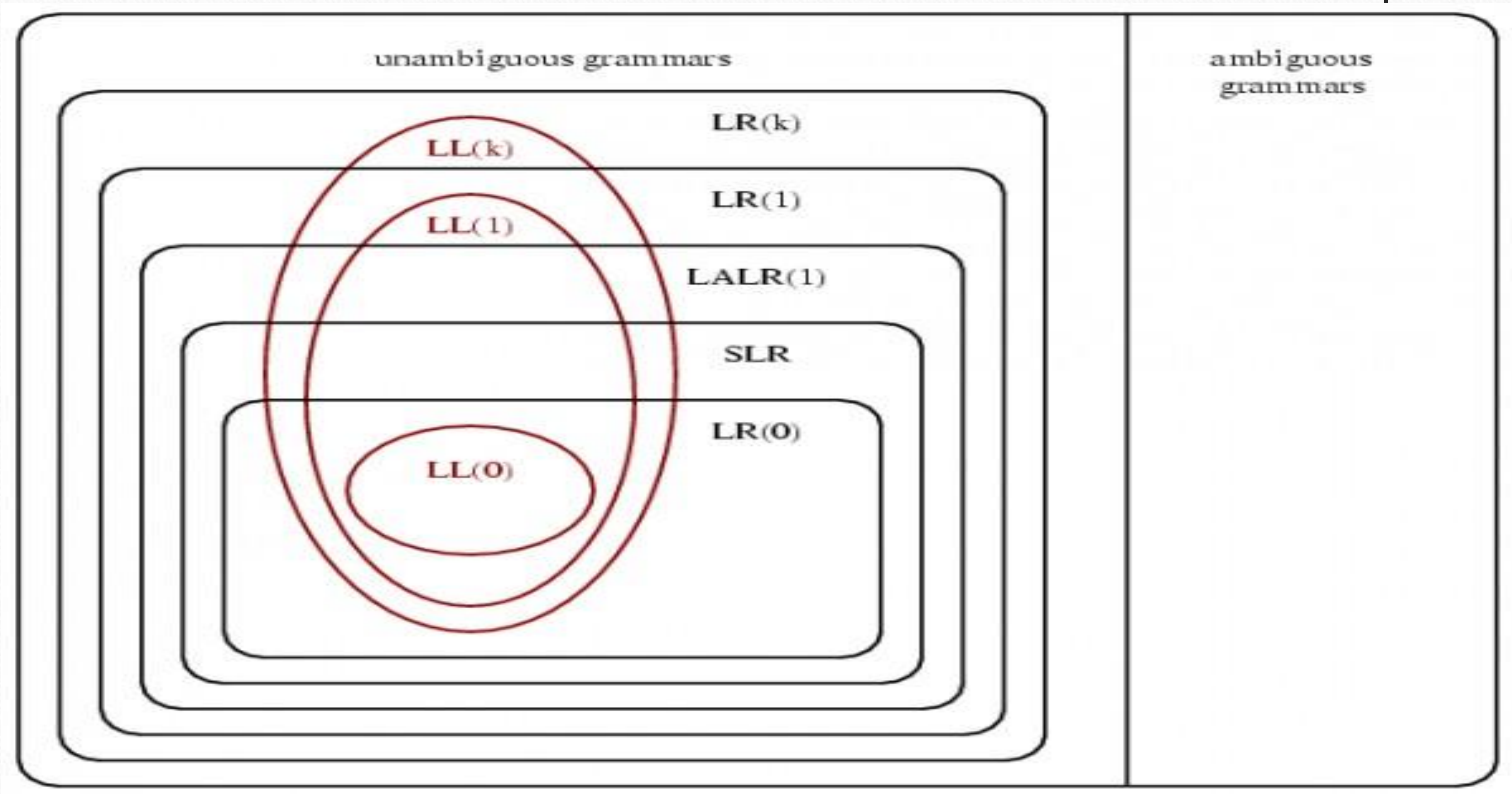
Shift-reduce table

Symbol	Current state	Next State
S	Any	1
L	0	2
	4	8
	Any	10
R	0	3
	4	7
	5	4
	11	13

Goto table

	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**	**4**
7			**r4**	**r4**			
8			**r6**	**r6**			
9				**r2**			
10				**r6**			
11	**s12**	**s11**				**10**	**13**
12				**r5**			
13				**r4**			

Determinant Context Free Grammar Relationships



LR(k) $\subseteq$ LR(k+1)
LL(k) $\subseteq$ LL(k+1)

LL(k) $\subseteq$ LR(k)
LR(0) $\subseteq$ SLR
LALR(1) $\subseteq$ LR(1)

Dealing with Ambiguous Grammars

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

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

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

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

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 $E \to E * E$

Other Ambiguity
Example: Dangling else

LR(1) parsing for Grammar:
Stmt → if expr then stmt else stmt
| if expr then stmt
| other

Prefer Shift over Reduce

I_0:
$S' \rightarrow \bullet S$,$
$S \rightarrow \bullet iSeS$, $
$S \rightarrow \bullet iS$, $
$S \rightarrow \bullet$a, $

I_1:
$S' \rightarrow S\bullet$,$

I_2:
$S \rightarrow$ i.SeS, $
S → i.S, $
S → .iSeS, e/$
S → .iS, e/$
S → . a, e/$

I_3:
$S \rightarrow$ a.,$

I_4:
$S \rightarrow$ iS.eS, $
S → iS., $

I_6:
$S \rightarrow$ a.,e/$

I_7:
$S \rightarrow$ i.SeS, e/$
S → i.S, e/$
S → .iSeS, e/$
S → .iS, e/$
S → . a e/$

I_8:
$S \rightarrow$ iS.eS, e/$
S → iS., e/$

Error Detection and Error Recovery in LR Parsing

Error Detection in LR Parsing

- Canonical LR parser :
  - Uses full LR(1) parse tables
  - **Never make a single reduction before recognizing the error** when a syntax error occurs on the input
- SLR and LALR :
  - It may still reduce when a syntax error occurs on the input
  - It 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*
  - There may be more than one choices of 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

Example : Panic mode error recovery in LR

Grammar : $S' \to S'\,S \mid S \quad S \to iSeS \mid iS \mid E\,; \quad E \to E + N \mid N$

Follow: S':{\$, i, N} **E**: {;, +}

State	i	+	;	N	e	$	S'	S	E
0	S3			S5			1	2	4
1	S3			S5		Acc		6	4
2	r2			r2		r2			
3	S3			S5				7	4
4		S8	S12						
5		r7	r7						
6	r1			r1		r1			
7	r4			r4	S9/r4	r4			
8				S10					
9	S3			S5				11	4
10		r6	r6						
11	r3			r3	r3	r3			
12	r5			r5	r5	r5			

Stack	Input	Remark
$0	N+i; i e N;	S5
$0N5	+i; i e N;	$E \to N$
$0E4	+i; i e N;	S8
$0E4+8	i; i e N;	**Syntax error: missing N, unexpected i (tokens after E +)** **Recovery : 1. Pop until Goto (until Nonterminal) ➔ pop +8** **2. Discard input until follow of E (here till ; that is i)**
$0E4	; i e N;	
$0E4;12	i e N;	$S \to E\,;$
$0S2	i e N;	$S' \to S$
$0S'1	i e N;	S3

Example : Panic mode error recovery in LR (cont…)

Grammar : S' → S' S | S   S → iSeS | iS | E ;   E → E + N | N

Follow: S':{$, i,e, N} **E:** {;, +}

State	i	+	;	N	e	$	S'	S	E
0	S3			S5			1	2	4
1	S3			S5		Acc		6	4
2	r2			r2		r2			
3	S3			S5				7	4
4		S8	S12						
5		r7	r7						
6	r1			r1		r1			
7	r4			r4	S9/r4	r4			
8				S10					
9	S3			S5				11	4
10		r6	r6						
11	r3			r3	r3	r3			
12	r5			r5	r5	r5			

Stack	Input	Remark
$0S'1i3	**e** N;$	**Syntax error : Missing statement S.** **(B'caz 3 has goto on S and E, and e is follow of S)** **Recovery: 1. Stack S**
$0S'1i3S7	**eN**;$	S9/r4 → prefer Shift
$0S'1i3S7e9	N;$	S3
$0S'1i3S7e9N5	;$	S5
$0S'1i3S7e9N5	;$	E → N ;
$0S'1i3S7e9E4	;$	S12
$0S'1i3S7e9E4;12	$	S->E;
$0S'1i3S7e9S11	$	S->iSeS
$0S'1S6	$	S'→S' S
$0S'1	$	Accept

Example : Phrase level error recovery in LR

Grammar :

S' → S' S | S

S → iSeS | iS | E ;

E → E + N | N

State	i	+	;	N	e	$	S'	S	E
0	S3	e1	e2	S5			1	2	4
1	S3			S5		Acc		6	4
2	r2			r2		r2			
3	S3			S5				7	4
4		S8	S12	e3					
5		r7	r7						
6	r1			r1		r1			
7	r4			r4	S9/r4	r4			
8				S10					
9	S3			S5				11	4
10		r6	r6						
11	r3			r3	r3	r3			
12	r5			r5	r5	r5			

Error	Description
e1	Missing operand, Inserted in input
e2	Extraneous symbol ; -> Deleted from input
e3	Missing operator, Inserted

Implement error routine(s) for every error entry (empty cell) in table

Example : Error recovery using error productions

Grammar :

S' → S' S | S | **error ; S**

S → iSeS | iS | E ;

E → E + N | N

State	i	+	;	N	e	$	S'	S	E
0	S3			S5			1	2	4
1	S3			S5		Acc		6	4
2	r2			r2		r2			
3	S3			S5				7	4
4		S8	S12						
5		r7	r7						
6	r1			r1		r1			
7	r4			r4	S9/r4	r4			
8				S10					
9	S3			S5				11	4
10		r6	r6						
11	r3			r3	r3	r3			
12	r5			r5	r5	r5			

Stack	Input	Remark
$0	N+e; i e N;	E → N
$0N5	**+e**; i e N;	
$0E4	+e; i e N;	**Syntax error** **Pop until state has error production** **Discard input '; ' until able to continue parsing**
$0E4+8	e**;** i e N;	
$0	i e N;	