



编译原理

Complier Principles

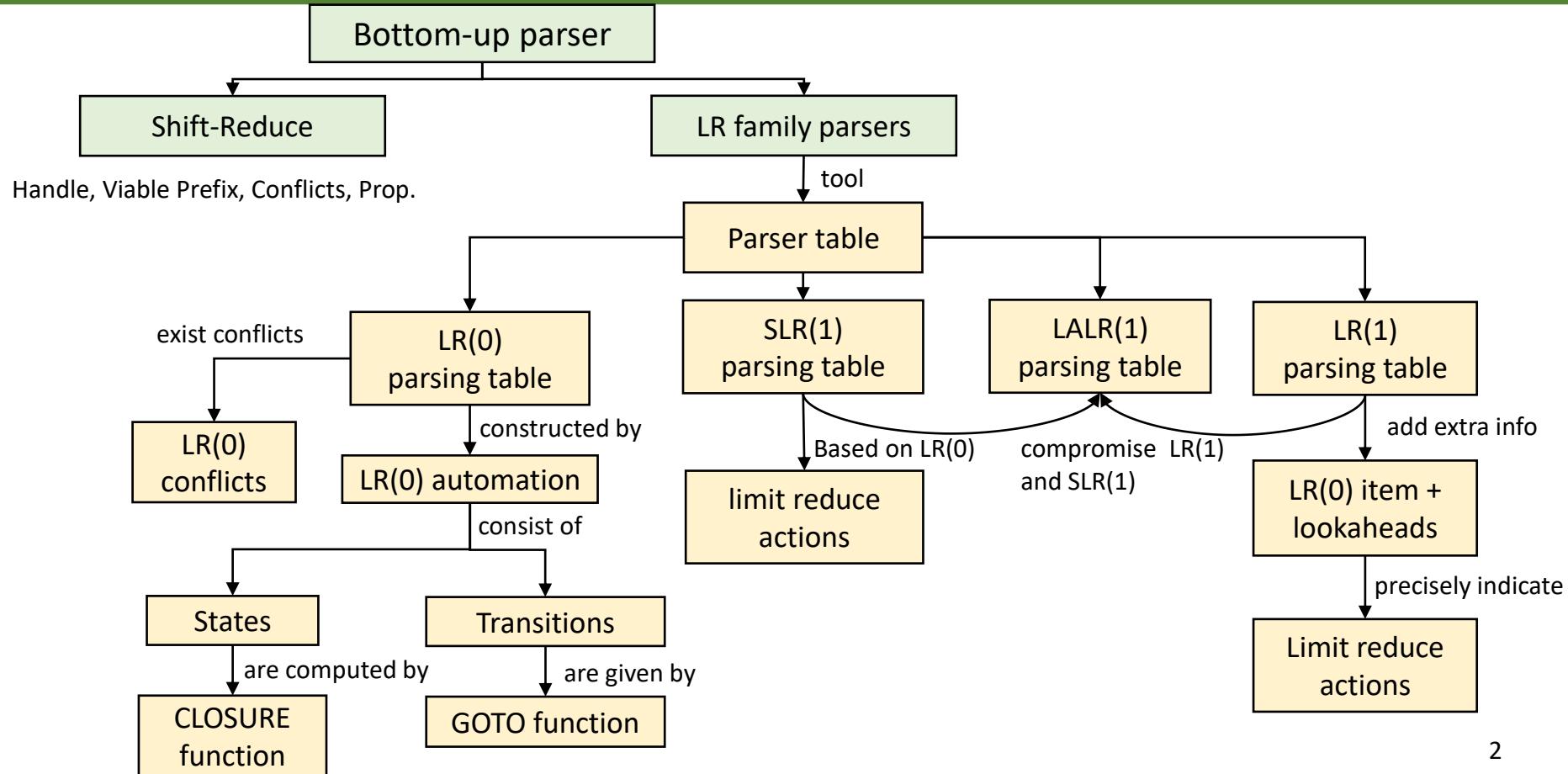
Lecture 5

Syntax Analysis: Bottom-Up

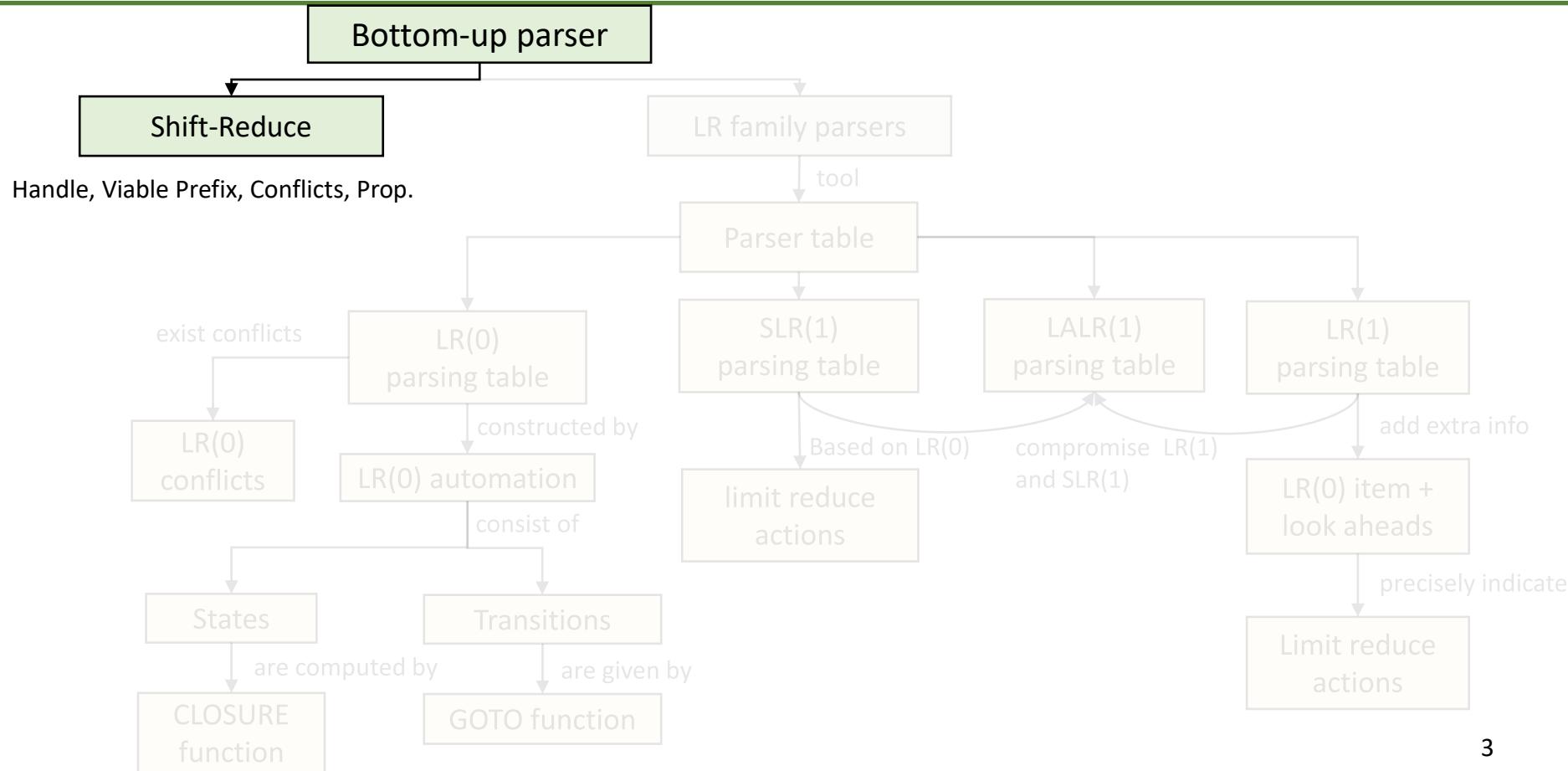
赵帅

计算机学院
中山大学

Contents



Contents



Bottom-up parsing[自下而上分析]



中山大學
SUN YAT-SEN UNIVERSITY

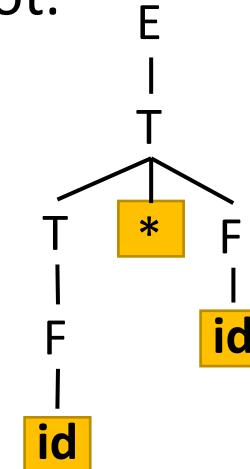
- Bottom-up parsing [自下而上分析]

- ◆ Start from the **input string**, and gradually **reduce**[规约] it (the inverse process of the rightmost derivation[最右推导的逆过程]) until it is reduced to the **start symbol** of the grammar.

- ◆ Construct a syntax tree from the leaves to the root.

- ◆ More powerful than top down:

- Don't need left factoring.
 - Can handle left recursion.

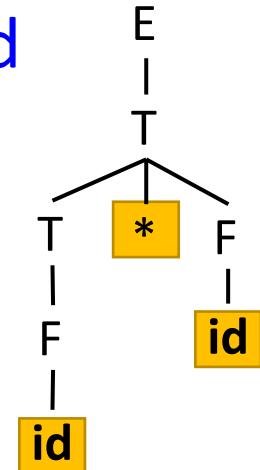


Bottom-up parsing [自下而上分析]

- Bottom-up parsing [自下而上分析]
 - ◆ Parser code structure nothing like grammar
 - Very difficult to implement manually
 - Automated tools exist to convert to code

- $G(E): E \rightarrow E+T \mid T; T \rightarrow T^*F \mid F; F \rightarrow (E) \mid \text{id}, \quad \text{id} * \text{id}$

- ◆ The rightmost derivation:
 - $E \Rightarrow T \Rightarrow T^* F \Rightarrow T^* \text{id} \Rightarrow F^* \text{id} \Rightarrow \text{id} * \text{id}$
 - ◆ The leftmost reduction:
 - ◆ $\text{id} * \text{id} \Rightarrow F^* \text{id} \Rightarrow T^* \text{id} \Rightarrow T^* F \Rightarrow T \Rightarrow E$



Bottom-up (Revisit)



中山大學
SUN YAT-SEN UNIVERSITY

- **Shift-Reduce** parser:
 - ◆ Shift[移入], Reduce[规约], Accept[接受], Error[报错]

Bottom-up: Shift-Reduce [移入-归约]



- **Shift-reduce parsing** is a form of bottom-up parsing in which a stack[栈] holds grammar symbols and an input buffer holds the input string.
- **Reduction**[规约]: At each reduction step, a specific substring matching *the body of a production* is **replaced**[替换] by the non-terminal at *the head of that production*.
- **Shift**[移入]: During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack.

Bottom-up: Shift-Reduce [移入-归约]



- There are **four actions** that a shift-reduce parser can make:
 - ◆ **Shift[移入]**: Shift the next input symbol onto the top of the stack.
 - ◆ **Reduce[规约]**: The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
 - ◆ **Accept[接受]**: Announce successful completion of parsing.
 - ◆ **Error[报错]**: Discover a syntax error and call an error recovery routine.

Bottom-up: Shift-Reduce [移入-归约]



中山大學
SUN YAT-SEN UNIVERSITY

Example:

- $G(S): S \rightarrow aAcBe;$
- $A \rightarrow b;$
- $A \rightarrow Ab;$
- $B \rightarrow d.$

Input String:

- abbcde\$

Stack	Input	Action
\$	abbcde\$	Shift
\$a	bbcde\$	Shift
\$ab	bcde\$	Reduce A \rightarrow b
\$aA	bcde\$	Shift
\$aAb	cde\$	Reduce A \rightarrow Ab
\$aA	cde\$	Shift
\$aAc	de\$	Shift
\$aAc	e\$	Reduce B \rightarrow d
\$aAcB	e\$	Shift
\$aAcBe	\$	Reduce S \rightarrow aAcBe
\$S	\$	ACCEPT

Key Issue [一个关键问题]

- The key decisions during bottom-up parsing are about (1) when to **shift** or **reduce** and (2) what **production** to apply, as the parse proceeds.
 - ◆ when to shift or reduce?
 - ◆ what production to apply?

Stack	Input	Action
\$	abbcde\$	Shift
\$a	bbcde\$	Shift
\$ab	bcde\$	Reduce A → b
\$aA	bcde\$	Shift
\$aAb	cde\$	Reduce A → Ab

Reduce A → b??

$G(S): S \rightarrow aAcBe;$
 $A \rightarrow b;$
 $A \rightarrow Ab;$
 $B \rightarrow d.$

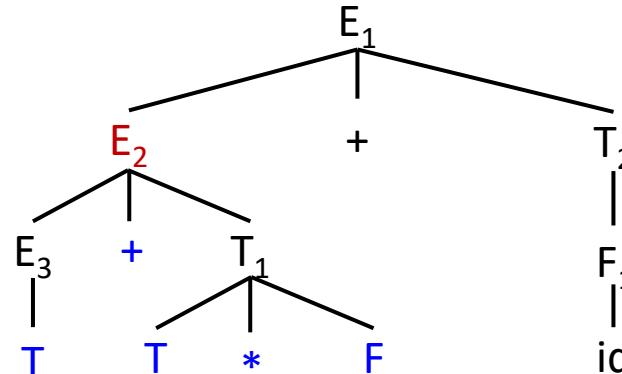
- **Right-sentential form** [最右句型]: a sentential form that occurs in the rightmost derivation[规范推导].

$G(E)$:

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * id \Rightarrow F * id \Rightarrow id * id$$

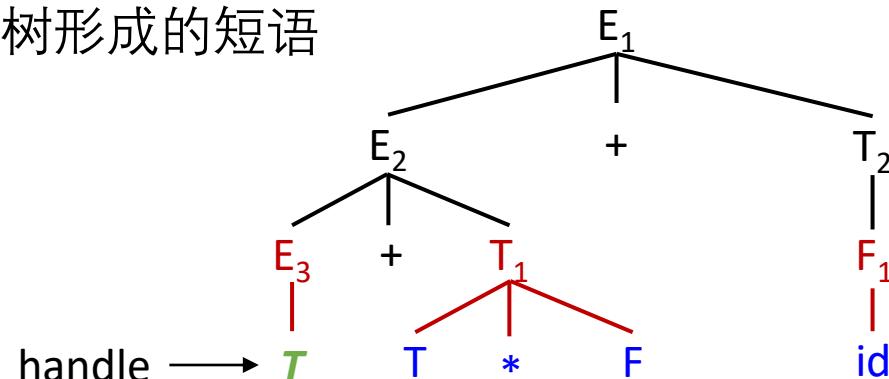
- **Phrase**[短语]: If $\alpha_1 A \alpha_2$ is a right sentential form of $G(S)$, and $S \xrightarrow{*} \alpha_1 A \alpha_2 \xrightarrow{+} \alpha_1 \beta \alpha_2$, then β is a phrase of $\alpha_1 A \alpha_2$.

- ◆ 一个句型的语法树中任意叶子结点所组成的符号串都是该句型的短语



Handle [句柄]

- **Simple Phrase**[直接短语]: If $\alpha_1 A \alpha_2$ is a right sentential form of $G(S)$, and $S \xrightarrow{*} \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$, then β is a **simple phrase** of $\alpha_1 A \alpha_2$.
 - ◆ 只有父子两代的子树形成的短语
- **Handle**[句柄]: The **leftmost simple phrase** of a sentential form.
 - ◆ 语法树中最左那棵只有父子两代的子树形成的短语
 - ◆ 一个句型中只有一个句柄
- We only want to reduce at handles
 - ◆ How to find it?



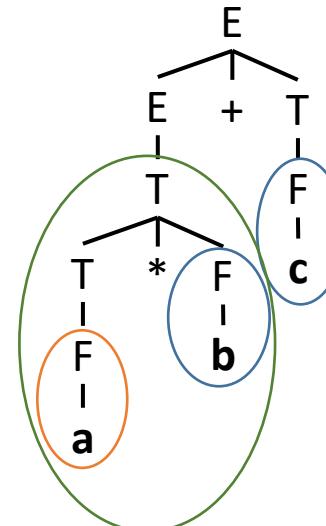
Example

- $G(E)$: $E \rightarrow T \mid E+T; \quad T \rightarrow F \mid T^*F; \quad F \rightarrow (E) \mid a \mid b \mid c$
- For sentential form $a * b + c$
 - **Phrase**: a^*b+c, a^*b, a, b, c
 - **Simple Phrase**: a, b, c
 - **Handle**: a

短语: 一个句型的语法树中任意树叶结点组成的短语

直接短语: 只有父子两代的子树形成的短语

句柄: 语法树中最左那棵只有父子两代的子树形成的短语



One More Example

- Gramar G(E):

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

$$T \rightarrow T^* F \mid F$$

$$F \rightarrow (E) \mid id$$

- Sentential form: $T + T^* F + id$

- Phrase:

- $E_1: T + T^* F + id$

- $E_2: T + T^* F$

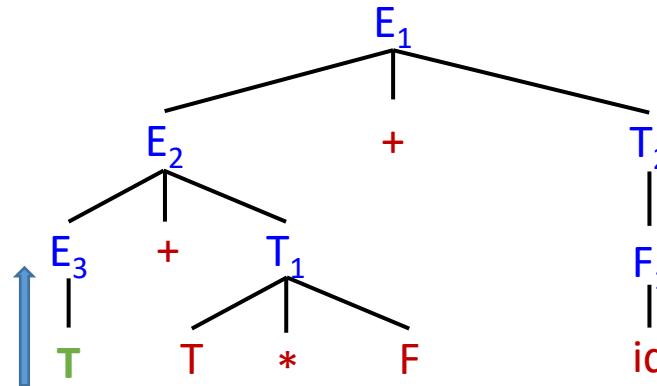
- $E_3: T$

- $T_1: T^* F$

- $T_2, F_1: id$

Handle

Simple Phrase



One More Example

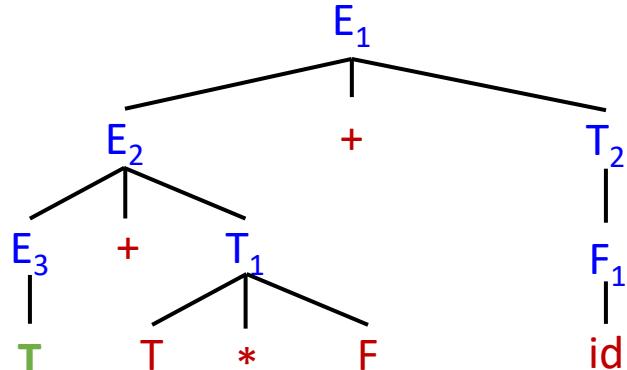
- Gramar G(E): $E \rightarrow E+T \mid T \quad T \rightarrow T^*F \mid F \quad F \rightarrow (E) \mid id$
- Sentential form: $T + T^*F + id$
- Phrase: $E_1: T+T^*F+id \quad E_2: T+T^*F \quad E_3: T \quad T_1: T^*F \quad T_2, F_1: id$

Handle Simple Phrase

Shift/Reduce:

Stack:

Input: $T + T^*F + id$ (Shift)



One More Example

- Gramar G(E): $E \rightarrow E+T \mid T \quad T \rightarrow T^*F \mid F \quad F \rightarrow (E) \mid id$
- Sentential form: $T + T^*F + id$
- Phrase: $E_1: T+T^*F+id \quad E_2: T+T^*F \quad E_3: T \quad T_1: T^*F \quad T_2, F_1: id$

Handle Simple Phrase

Shift/Reduce:

Stack:

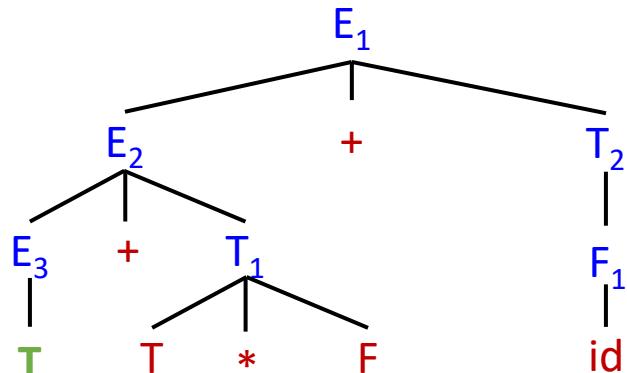
Stack: T

Input: T + T * F + id

(Shift)

Input: + T * F + id

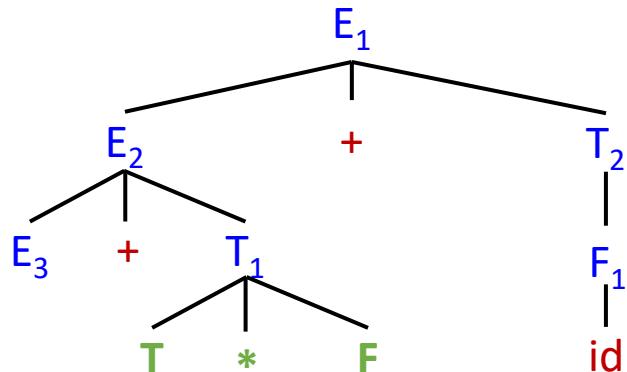
(Reduce $E_3 \rightarrow T$)



One More Example

- Gramar G(E): $E \rightarrow E+T \mid T \quad T \rightarrow T^*F \mid F \quad F \rightarrow (E) \mid id$
- Sentential form: $T + T^*F + id$
- Phrase: $E_1: T+T^*F+id \quad E_2: T+T^*F \quad E_3: T \quad T_1: T^*F \quad T_2, F_1: id$

Handle Simple Phrase



Shift/Reduce:

Stack:

Stack: T

Stack: E_3

Input: $T + T^*F + id$

(Shift)

Input: $+ T^*F + id$

(Reduce $E_3 \rightarrow T$)

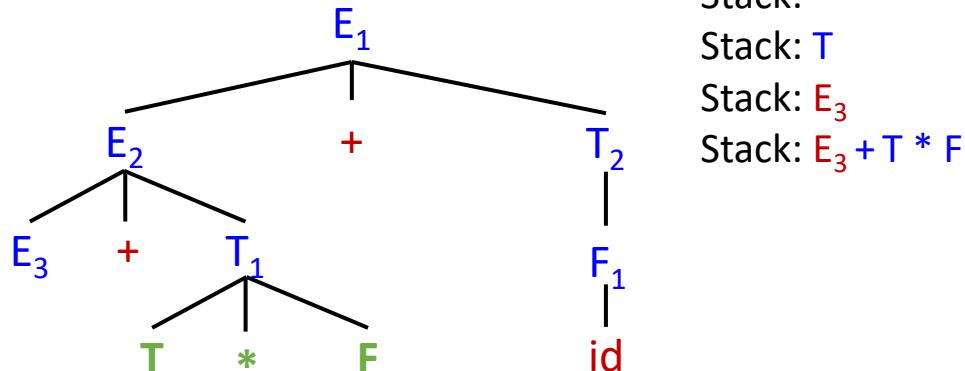
Input: $+ T^*F + id$

(Shift x4)

One More Example

- Gramar G(E): $E \rightarrow E+T \mid T \quad T \rightarrow T^*F \mid F \quad F \rightarrow (E) \mid id$
- Sentential form: $T + T^*F + id$
- Phrase: $E_1: T+T^*F+id \quad E_2: T+T^*F \quad E_3: T \quad T_1: T^*F \quad T_2, F_1: id$

Handle Simple Phrase



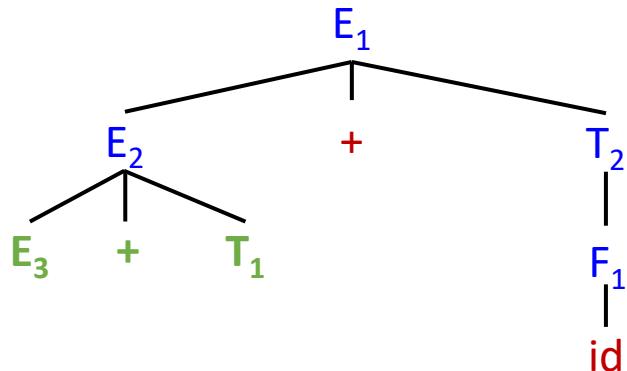
Shift/Reduce:

Stack:	Input: $T + T^*F + id$	(Shift)
Stack: T	Input: $+ T^*F + id$	(Reduce $E_3 \rightarrow T$)
Stack: E_3	Input: $+ T^*F + id$	(Shift x4)
Stack: $E_3 + T^*F$	Input: $+ id$	(Reduce $T_1 \rightarrow T^*F$)

One More Example

- Gramar G(E): $E \rightarrow E+T \mid T \quad T \rightarrow T^*F \mid F \quad F \rightarrow (E) \mid id$
- Sentential form: $T + T^*F + id$
- Phrase: $E_1: T+T^*F+id \quad E_2: T+T^*F \quad E_3: T \quad T_1: T^*F \quad T_2, F_1: id$

Handle Simple Phrase

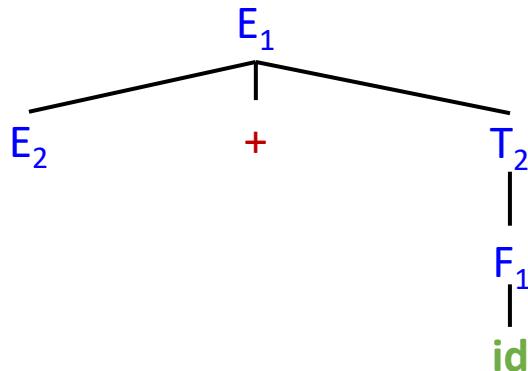


Shift/Reduce:

Stack:	Input: $T + T^*F + id$	(Shift)
Stack: T	Input: $+ T^*F + id$	(Reduce $E_3 \rightarrow T$)
Stack: E_3	Input: $+ T^*F + id$	(Shift x4)
Stack: $E_3 + T^*F$	Input: $+ id$	(Reduce $T_1 \rightarrow T^*F$)
Stack: $E_3 + T_1$	Input: $+ id$	(Reduce $E_2 \rightarrow E_3 + T_1$)

One More Example

- Gramar G(E): $E \rightarrow E+T \mid T \quad T \rightarrow T^*F \mid F \quad F \rightarrow (E) \mid id$
- Sentential form: $T + T^*F + id$
- Phrase: $E_1: T+T^*F+id \quad E_2: T+T^*F \quad E_3: T \quad T_1: T^*F \quad T_2, F_1: id$



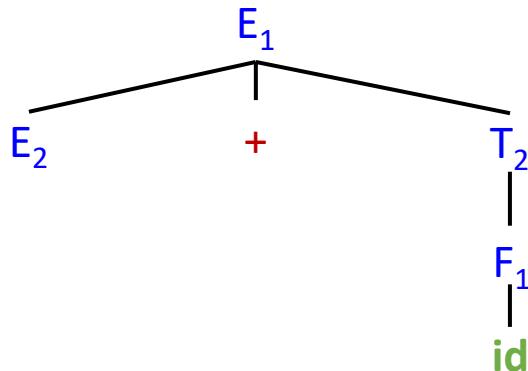
Handle Simple Phrase

Shift/Reduce:

Stack:	Input: $T + T^*F + id$	(Shift)
Stack: T	Input: $+ T^*F + id$	(Reduce $E_3 \rightarrow T$)
Stack: E_3	Input: $+ T^*F + id$	(Shift x4)
Stack: $E_3 + T^*F$	Input: $+ id$	(Reduce $T_1 \rightarrow T^*F$)
Stack: $E_3 + T_1$	Input: $+ id$	(Reduce $E_2 \rightarrow E_3 + T_1$)
Stack: E_2	Input: $+ id$	(Shift x2)

One More Example

- Gramar G(E): $E \rightarrow E+T \mid T \quad T \rightarrow T^*F \mid F \quad F \rightarrow (E) \mid id$
- Sentential form: $T + T^*F + id$
- Phrase: $E_1: T+T^*F+id \quad E_2: T+T^*F \quad E_3: T \quad T_1: T^*F \quad T_2, F_1: id$

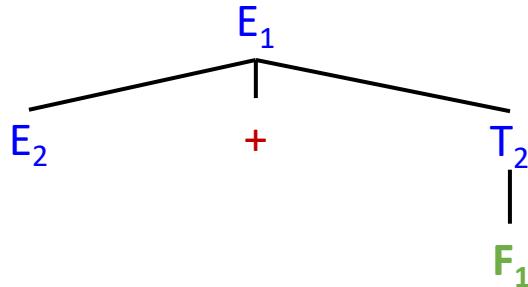


Shift/Reduce:

Stack:	Input: $T + T^*F + id$	(Shift)
Stack: T	Input: $+ T^*F + id$	(Reduce $E_3 \rightarrow T$)
Stack: E_3	Input: $+ T^*F + id$	(Shift x4)
Stack: $E_3 + T^*F$	Input: $+ id$	(Reduce $T_1 \rightarrow T^*F$)
Stack: $E_3 + T_1$	Input: $+ id$	(Reduce $E_2 \rightarrow E_3 + T_1$)
Stack: E_2	Input: $+ id$	(Shift x2)
Stack: $E_2 + id$	Input:	(Reduce $F_1 \rightarrow id$)

One More Example

- Gramar G(E): $E \rightarrow E+T \mid T \quad T \rightarrow T^*F \mid F \quad F \rightarrow (E) \mid id$
- Sentential form: $T + T^*F + id$
- Phrase: $E_1: T+T^*F+id \quad E_2: T+T^*F \quad E_3: T \quad T_1: T^*F \quad T_2, F_1: id$



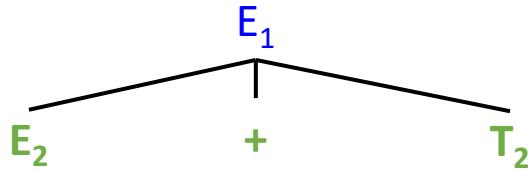
Handle Simple Phrase

Shift/Reduce:

Stack:	Input: $T + T^*F + id$	(Shift)
Stack: T	Input: $+ T^*F + id$	(Reduce $E_3 \rightarrow T$)
Stack: E_3	Input: $+ T^*F + id$	(Shift x4)
Stack: $E_3 + T^*F$	Input: $+ id$	(Reduce $T_1 \rightarrow T^*F$)
Stack: $E_3 + T_1$	Input: $+ id$	(Reduce $E_2 \rightarrow E_3 + T_1$)
Stack: E_2	Input: $+ id$	(Shift x2)
Stack: $E_2 + id$	Input:	(Reduce $F_1 \rightarrow id$)
Stack: $E_2 + F_1$	Input:	(Reduce $T_2 \rightarrow F_1$)

One More Example

- Gramar G(E): $E \rightarrow E+T \mid T \quad T \rightarrow T^*F \mid F \quad F \rightarrow (E) \mid id$
- Sentential form: $T + T^*F + id$
- Phrase: $E_1: T+T^*F+id \quad E_2: T+T^*F \quad E_3: T \quad T_1: T^*F \quad T_2, F_1: id$



Shift/Reduce:

Stack:	Input: $T + T^*F + id$	(Shift)
Stack: T	Input: $+ T^*F + id$	(Reduce $E_3 \rightarrow T$)
Stack: E_3	Input: $+ T^*F + id$	(Shift x4)
Stack: $E_3 + T^*F$	Input: $+ id$	(Reduce $T_1 \rightarrow T^*F$)
Stack: $E_3 + T_1$	Input: $+ id$	(Reduce $E_2 \rightarrow E_3 + T_1$)
Stack: E_2	Input: $+ id$	(Shift x2)
Stack: $E_2 + id$	Input:	(Reduce $F_1 \rightarrow id$)
Stack: $E_2 + F_1$	Input:	(Reduce $T_2 \rightarrow F_1$)
Stack: $E_2 + T_2$	Input:	(Reduce $E_1 \rightarrow E_2 + T_2$)

One More Example

- Gramar G(E): $E \rightarrow E+T \mid T \quad T \rightarrow T^*F \mid F \quad F \rightarrow (E) \mid id$
- Sentential form: $T + T^*F + id$
- Phrase: $E_1: T+T^*F+id \quad E_2: T+T^*F \quad E_3: T \quad T_1: T^*F \quad T_2, F_1: id$

Shift/Reduce:		
Stack: E_1	Input: $T + T^*F + id$	(Shift)
Stack: T	Input: $+ T^*F + id$	(Reduce $E_3 \rightarrow T$)
Stack: E_3	Input: $+ T^*F + id$	(Shift x4)
Stack: $E_3 + T^*F$	Input: $+ id$	(Reduce $T_1 \rightarrow T^*F$)
Stack: $E_3 + T_1$	Input: $+ id$	(Reduce $E_2 \rightarrow E_3 + T_1$)
Stack: E_2	Input: $+ id$	(Shift x2)
Stack: $E_2 + id$	Input:	(Reduce $F_1 \rightarrow id$)
Stack: $E_2 + F_1$	Input:	(Reduce $T_2 \rightarrow F_1$)
Stack: $E_2 + T_2$	Input:	(Reduce $E_1 \rightarrow E_2 + T_2$)
Stack: E_1	Input:	(Accept)

Example

- $G(S): S \rightarrow aAcBe; A \rightarrow b; A \rightarrow Ab; B \rightarrow d.$

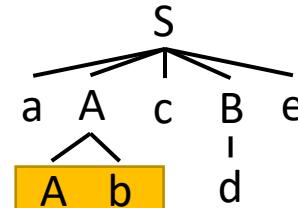
Stack	Input	Action
\$	abbcde\$	Shift
\$a	bbcde\$	Shift
\$ab	bcde\$	Reduce $A \rightarrow b$
\$aA	bcde\$	Shift
\$a A	cde\$	Reduce $A \rightarrow Ab$

Reduce $A \rightarrow Ab?$

OR

Reduce $A \rightarrow b?$

Ab is Handle, so we choose $A \rightarrow Ab$ to reduce.



Handle Always Occurs at Stack Top



- Does handle appear **outside the stack?**
 - ◆ It can, but handle will eventually be shifted in, placing it at top of stack.
- Why does handle not appear in the **middle of the stack?**
 - ◆ Parser eagerly reduces **when handle is at top of stack.**
- 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

Viable Prefix [活前缀]



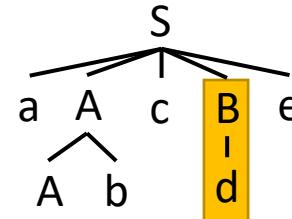
- A **viable prefix**[活前缀/可行前缀] is a prefix of a right-sentential form that does **not pass the end of the rightmost handle** of that sentential form.
 - ◆ Stack content is always a viable prefix, guaranteeing the shift / reduce is **on the right track**.
- **Example 1:** $S \Rightarrow aE b \Rightarrow aaE b$, the handle is **aE**, so the **viable prefix** is **a**, **aa**, **aaE**, but not **aaEb**.

Stack	Input
\$...	Handle...\$
\$...H	andle...\$
\$...Ha	ndle...\$
\$...Han	dle...\$

Viable Prefix [活前缀]

- Example 2: $G(S)$: $S \rightarrow aAcBe; A \rightarrow b; A \rightarrow Ab; B \rightarrow d.$
- “**aAcde**” is a right sentential form, it is split between the stack and the input buffer, **aA**, **aAc**, **aAcd** are all viable prefix of **aAcde**.

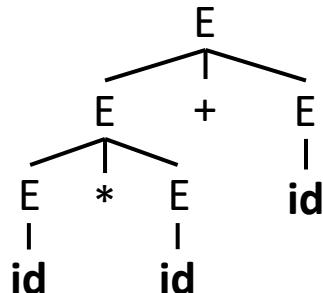
Stack	Input	Action
\$aA	cde\$	Shift
\$aAc	de\$	Shift
\$aAcd	e\$	Reduce $B \rightarrow d$



Conflicts[冲突]



- Conflicts arise with ambiguous grammars.
- Consider $G(E)$:
- $E \rightarrow E^*E \mid E+E \mid (E) \mid id$,
- sentential form: **id * id + id**

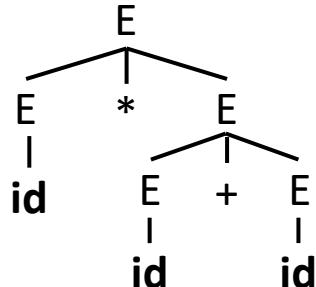


Step	Stack	Input	Action
0	\$	id*id+id\$	Shift
1	\$id	*id+id\$	Reduce E → id
2	\$E	*id+id\$	Shift
3	\$E*	id+id\$	Shift
4	\$E*id	+id\$	Reduce E → id
5	\$E*E	+id\$	Reduce E → E*E
6	\$E	+id\$	Shift
7	\$E+	id\$	Shift
8	\$E+id	\$	Reduce E → id
9	\$E+E	\$	Reduce E → E+E
10	\$E	\$	ACCEPT

Conflicts[冲突]



- Conflicts arise with ambiguous grammars.
- Consider $G(E)$:
- $E \rightarrow E^*E \mid E+E \mid (E) \mid id$,
- sentential form: **id * id + id**



Step	Stack	Input	Action
0	\$	id*id+id\$	Shift
1	\$id	*id+id\$	Reduce $E \rightarrow id$
2	\$E	*id+id\$	Shift
3	\$E*	id+id\$	Shift
4	\$E*id	+id\$	Reduce $E \rightarrow id$
5	\$E*E	+id\$	Shift (Reduce $E \rightarrow E^*E$)
6	\$E*E+id	\$	Reduce $E \rightarrow id$
7	\$E*E+E	\$	Reduce $E \rightarrow E+E$
8	\$E*E	\$	Reduce $E \rightarrow E^*E$
9	\$E	\$	ACCEPT

Conflicts[冲突]



- Conflicts arise with ambiguous grammars.
 - Grammar: $E \rightarrow E^*E \mid E+E \mid (E) \mid id$
 - sentential form: $id * id + id$
- In the steps marked blue, both shift or reduce by $E \rightarrow E^*E$ is okay since the **precedence** of + and * is not specified in the grammar, same problem with associativity.
- As usual, we should remove conflicts due to ambiguity:
 - ◆ Rewrite grammar/parser to encode precedence and associativity.
 - Operator Precedence Parsers (OPP), see Section 4.6 in Dragon Book, 1st Edition
 - ◆ Get rid of remaining ambiguity (e.g., if-then-else)

5	\$E^*E	+id\$	Shift (Reduce E → E^*E)
---	--------	-------	----------------------------

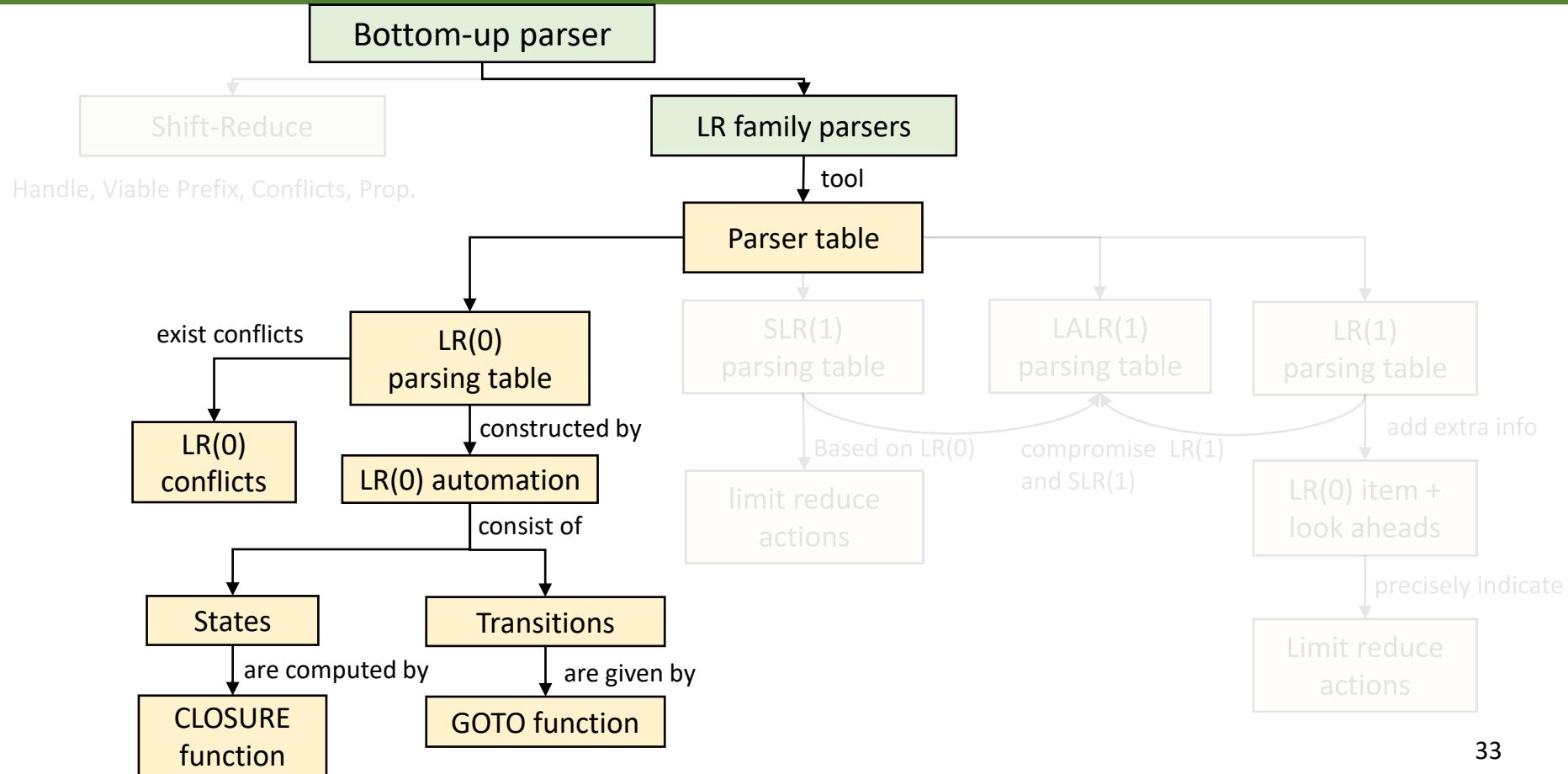
Properties of Bottom-up Parsing



中山大學
SUN YAT-SEN UNIVERSITY

- 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
- Types of conflicts
 - ◆ If it is possible to either shift or reduce then there is a **shift-reduce conflict**.
 - ◆ If there are **two possible 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.

Contents



Types of Bottom-Up Parsers

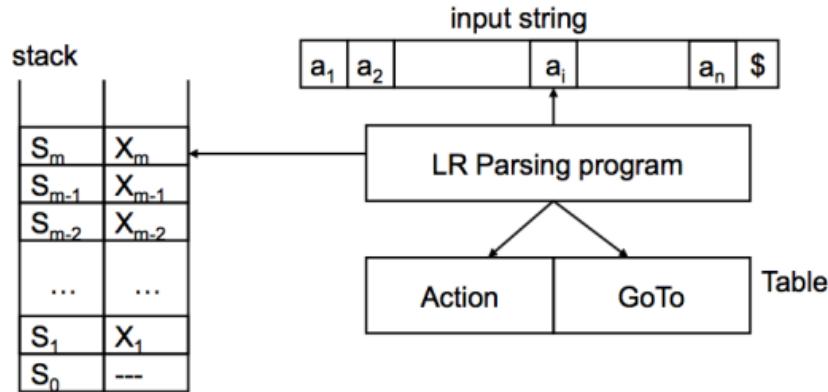
- Different types of bottom-up parsers:
 - ◆ Simple precedence parsers
 - ◆ Operator precedence parsers
 - ◆ Recursive ascent parsers
 - ◆ **LR family parsers**
 - ◆ others...
- Here, we will mainly focus on the **LR family parsers**, which are a type of **bottom-up** parser that analyzes deterministic context-free languages in linear time.



LR(k) Parser

- **LR(k)** : member of LR family of parsers
 - ◆ "L" stands for **left-to-right** scanning of the input.
 - ◆ "R" stands for constructing a **rightmost derivation in reverse**.
 - ◆ " k " is the number of input symbols of the **lookahead** used to make number of parsing decision.
- Comparison with LL(k) parser
 - ◆ **Efficient** as LL(k): Linear in time and space to length of input (same as LL(k)).
 - ◆ **Convenient** as LL(k): Can generate automatically from grammar – YACC, Bison.
 - ◆ More **complex** than LL(k): Harder to debug parser when grammar causes conflicting predictions.
 - ◆ More **powerful** than LL(k): Handles more grammars: no left recursion removal, left factoring needed.

LR Parser

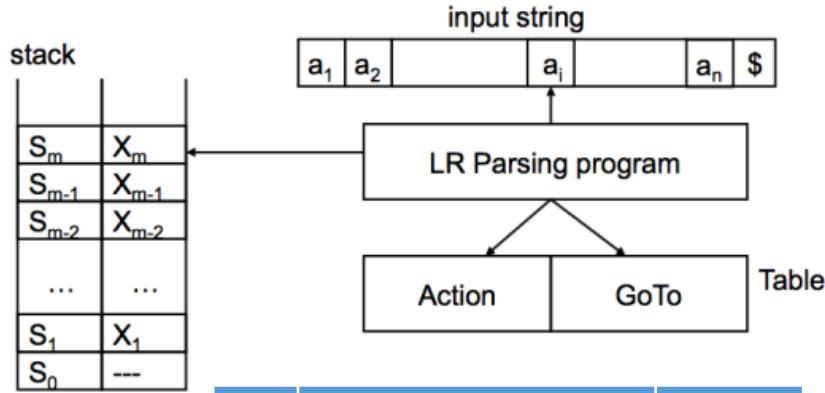


- The stack holds a sequence of **states**, $S_0 S_1 \dots S_m$ (S_m is the top)
 - ◆ States are to **track where we are** in a parse.
 - ◆ Each grammar symbol X_i is associated with a state S_i
- Contents of stack + input ($X_1 X_2 \dots X_m a_i \dots a_n$) is a right sentential form
 - ◆ If the input string is a member of the language
- Uses $[S_i, a_i]$ to index into parsing table to determine action

Bottom-up (Revisit)



- **LR(0) Parser:**



State	Action					Goto	
	*	+	0	1	\$	E	B
0			s1	s2		3	4
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	s5	s6			acc		
4	r3	r3	r3	r3	r3		
5			s1	s2			7
6			s1	s2			8
7	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2		

Parse Table

- The LR-parsing algorithm must decide **when to shift** and **when to reduce** (and in the latter case, by which production).
- It does so by consulting two tables: **ACTION** and **GOTO**.
- The basic algorithm is the same for all LR parsers, what changes are the tables ACTION and GOTO.

State	Action					Goto	
	Terminals					Non-Terminals	
0							
1							
...							

Action table[动作表]

- The **action table** is indexed by a **state** of the parser and a **terminal** and contains three types of actions.
 - ◆ Shift, Reduce, Accept, Error
- Action[S, a]** tells the parser what to do when the state on top of the stack is **S** and terminal **a** is the next input token.

(1) $E \rightarrow E * B$
(2) $E \rightarrow E + B$
(3) $E \rightarrow B$
(4) $B \rightarrow 0$
(5) $B \rightarrow 1$

State (s)	Action				
	*	+	0	1	\$
0			s1	s2	
1	r4	r4	r4	r4	r4
2	r5	r5	r5	r5	r5
3	s5	s6			acc
4	r3	r3	r3	r3	r3
5			s1	s2	
6			s1	s2	
7	r1	r1	r1	r1	r1
8	r2	r2	r2	r2	r2

Possible Actions[可能的动作]



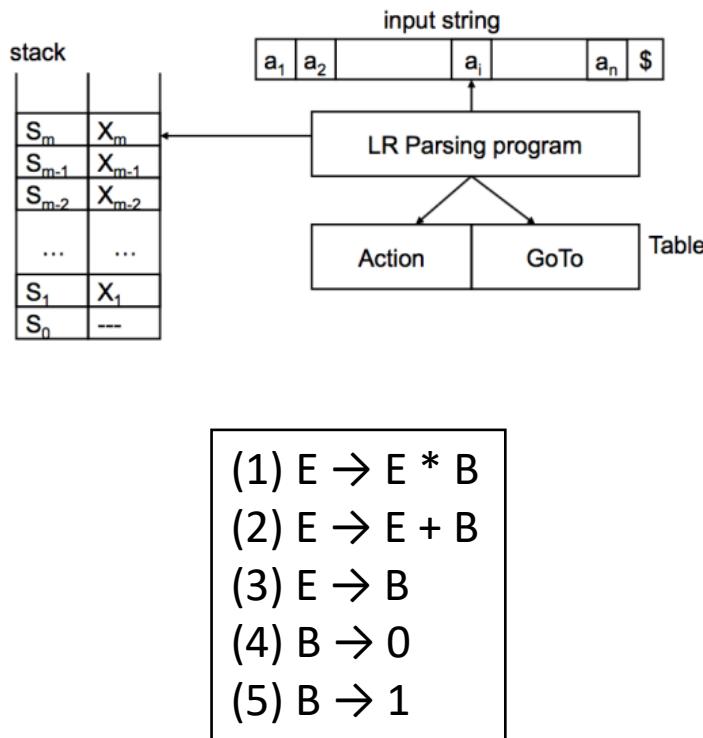
- ACTION[S_m, a_i] has four possible actions:
 - Shift (s_x): the handle is not completely loaded in the stack
 1. Shift the next input symbol onto the top of the stack
 2. Add State x to the stack
 - Reduce (r_y): the handle is at the top of the stack
 1. Pop the handle and all its associated states.
 2. Reduce the handle by the yth production (say A → a)
 3. Push A onto the stack
 4. Push state S_j by GOTO table, S_j = GOTO[S_i, A] and S_i is the current state.
 - Accept
 - ◆ ACTION[S_i, a_i] = ACC, then parsing is complete and successfully
 - Error
 - ◆ ACTION[S_i, a_i] = <empty>, then report error and stop

Goto Table[跳转表]

- The **goto table** is indexed by a **state** of the parser and a **nonterminal**
 - Define the **next state** of the parser if it has recognized a certain nonterminal.
 - This table is important to find out the next state after every **reduction**.
- GOTO[S_i, A]** indicates the **new state** to place on the top of the stack after a reduction is performed
- After a reduction action is performed:
 - a non-terminal **A** is pushed onto the stack **without an associated state**.
 - The current state at the top is **S_i**
 - Goto table tells which **state** should be pushed together with A by **GOTO[S_i, A]**.

State	Goto	
	E	B
0	3	4
1		
2		
3		
4		
5		7
6		8
7		
8		

Parse Table



State	Action					Goto	
	*	+	0	1	\$	E	B
0			s1	s2		3	4
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	s5	s6			acc		
4	r3	r3	r3	r3	r3		
5			s1	s2			7
6			s1	s2			8
7	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2		

Parser Actions



Initial

Stack:

S_0					
\$					

Input: $a_1a_2\dots a_n\$$

General

Stack:

S_0	S_1	S_2	...	S_m	
\$	X_1	X_2	...	X_m	

Input: $a_i a_{i+1} \dots a_n \$$

- If $\text{ACTION}[s_m, a_i] = s_x$, then do **shift**:

- ◆ Shift a_i on stack, which is removed from the input
- ◆ Enters state x , i.e., pushes state x on stack

Shift

S_0	S_1	S_2	...	S_m	s_x
\$	X_1	X_2	...	X_m	a_i

$a_{i+1} \dots a_n \$$

Parser Actions

- If $\text{ACTION}[s_m, a_i] = r_x$, then do **reduce**:

- ◆ Pops k (states, symbols) from stack
- ◆ Pushes A on stack
- ◆ $\text{GOTO}[s_{m-k}, A] = S_j$, then

General	S_0	S_1	S_2	...	S_m	
	\$	X_1	X_2	...	X_m	
Pop	S_0	S_1	S_2	...	S_{m-k}	
	\$	X_1	X_2	...	X_{m-k}	
Push & GOTO	S_0	S_1	S_2	...	S_{m-k}	S_j
	\$	X_1	X_2	...	X_{m-k}	A

x^{th} production: $A \rightarrow X_{m-k+1} \dots X_m$

$a_i a_{i+1} \dots a_n \$$

$a_i a_{i+1} \dots a_n \$$

$a_i a_{i+1} \dots a_n \$$

LR Parsing Program



- Input: string w and parse table with ACTION/GOTO
- Output: reduction steps of w , or error
- Initial: s_0 on the stack, $w\$$ in the input buffer

```
while (1) {
    let s and a be the state and symbol on top of the stack;
    if (ACTION[s,a] =  $S_t$ ) {
        push (t,a) onto the stack;
    } else if (ACTION[s,a] =  $r_k$ ) { /* A ->  $\beta$  */
        pop | $\beta$ | (states, symbols) off the stack;
        push (GOTO[s,A],A) onto the stack;
    } else if (ACTION[s,a] = acc)
        break;
    else
        call error-recovery routine;
}
```

Example: Parse Table

- This example of LR parsing uses the following small grammar **with goal symbol E**:

$$(1) E \rightarrow E * B$$

$$(2) E \rightarrow E + B$$

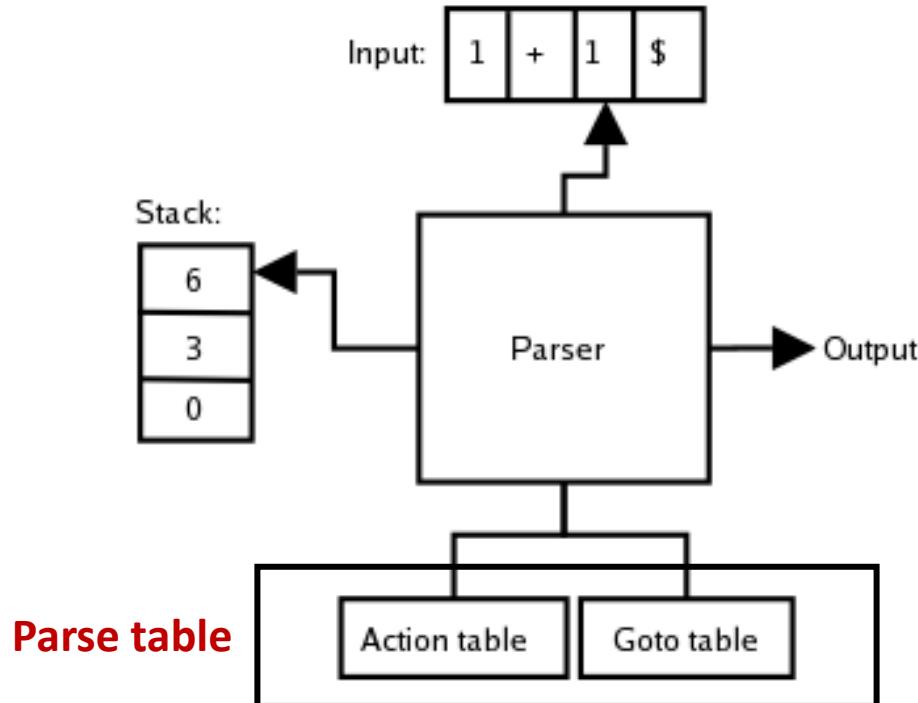
$$(3) E \rightarrow B$$

$$(4) B \rightarrow 0$$

$$(5) B \rightarrow 1$$

- to parse the following input:

1 + 1



source: https://en.wikipedia.org/wiki/LR_parser#Table_construction

Example: Parse Table

- Grammar:

$$\begin{array}{l}
 (1) E \rightarrow E * B \\
 (2) E \rightarrow E + B \\
 (3) E \rightarrow B \\
 (4) B \rightarrow 0 \\
 (5) B \rightarrow 1
 \end{array}$$

Input:

$1 + 1$

- Table entry:

- ◆ s_i : shifts the input symbol and moves to state i (i.e., push state on stack)
- ◆ r_j : reduce by production numbered j
- ◆ acc : accept
- ◆ $empty$: error

State	Action						Goto	
	*	+	0	1	\$	E	B	
0			s1	s2		3	4	
1	r4	r4	r4	r4	r4			
2	r5	r5	r5	r5	r5			
3	s5	s6				acc		
4	r3	r3	r3	r3	r3			
5			s1	s2				7
6			s1	s2				8
7	r1	r1	r1	r1	r1			
8	r2	r2	r2	r2	r2			

Example: Parsing steps

- (1) $E \rightarrow E * B$
- (2) $E \rightarrow E + B$
- (3) $E \rightarrow B$
- (4) $B \rightarrow 0$
- (5) $B \rightarrow 1$

Stack	input	ACTION	GOTO
<u>0</u>	\$	<u>1+1\$</u>	s2

State	Action						Goto	
	*	+	0	1	\$	E	B	
0			s1	s2		3	4	
1	r4	r4	r4	r4	r4			
2	r5	r5	r5	r5	r5			
3	s5	s6			acc			
4	r3	r3	r3	r3	r3			
5			s1	s2				7
6			s1	s2				8
7	r1	r1	r1	r1	r1			
8	r2	r2	r2	r2	r2			

Example: Parsing steps

- (1) $E \rightarrow E * B$
- (2) $E \rightarrow E + B$
- (3) $E \rightarrow B$
- (4) $B \rightarrow 0$
- (5) $B \rightarrow 1$

Stack		input	ACTION	GOTO
0	\$	1+1\$	s2	
0 <u>2</u>	\$1	<u>+1\$</u>	r5	4

State	Action						Goto	
	*	+	0	1	\$	E	B	
0			s1	s2		3	4	
1	r4	r4	r4	r4	r4			
2	r5	r5	r5	r5	r5			
3	s5	s6			acc			
4	r3	r3	r3	r3	r3			
5			s1	s2				7
6			s1	s2				8
7	r1	r1	r1	r1	r1			
8	r2	r2	r2	r2	r2			

Example: Parsing steps

- (1) $E \rightarrow E * B$
- (2) $E \rightarrow E + B$
- (3) $E \rightarrow B$
- (4) $B \rightarrow 0$
- (5) $B \rightarrow 1$

Stack		input	ACTION	GOTO
0	\$	1+1\$	s2	
02	\$1	+1\$	r5	4
0	\$	+1\$		

State	Action						Goto	
	*	+	0	1	\$	E		B
0			s1	s2		3	4	
1	r4	r4	r4	r4	r4			
2	r5	r5	r5	r5	r5			
3	s5	s6			acc			
4	r3	r3	r3	r3	r3			
5			s1	s2			7	
6			s1	s2				8
7	r1	r1	r1	r1	r1			
8	r2	r2	r2	r2	r2			

Example: Parsing steps

- (1) $E \rightarrow E * B$
- (2) $E \rightarrow E + B$
- (3) $E \rightarrow B$
- (4) $B \rightarrow 0$
- (5) $B \rightarrow 1$

Stack		input	ACTION	GOTO
0	\$	1+1\$	s2	
02	\$1	+1\$	r5	4
0 <u>4</u>	\$B	<u>+1\$</u>	r3	3

State	Action						Goto	
	*	+	0	1	\$	E	B	
0			s1	s2		3	4	
1	r4	r4	r4	r4	r4			
2	r5	r5	r5	r5	r5			
3	s5	s6			acc			
4	r3	r3	r3	r3	r3			
5			s1	s2				7
6			s1	s2				8
7	r1	r1	r1	r1	r1			
8	r2	r2	r2	r2	r2			

Example: Parsing steps

- (1) $E \rightarrow E * B$
- (2) $E \rightarrow E + B$
- (3) $E \rightarrow B$
- (4) $B \rightarrow 0$
- (5) $B \rightarrow 1$

Stack		input	ACTION	GOTO
0	\$	1+1\$	s2	
02	\$1	+1\$	r5	4
04	\$B	+1\$	r3	3
0	\$	+1\$		

State	Action						Goto	
	*	+	0	1	\$	E	B	
0			s1	s2		3	4	
1	r4	r4	r4	r4	r4			
2	r5	r5	r5	r5	r5			
3	s5	s6				acc		
4	r3	r3	r3	r3	r3			
5			s1	s2				7
6			s1	s2				8
7	r1	r1	r1	r1	r1			
8	r2	r2	r2	r2	r2			

Example: Parsing steps

- (1) $E \rightarrow E * B$
- (2) $E \rightarrow E + B$
- (3) $E \rightarrow B$
- (4) $B \rightarrow 0$
- (5) $B \rightarrow 1$

Stack		input	ACTION	GOTO
0	\$	1+1\$	s2	
02	\$1	+1\$	r5	4
04	\$B	+1\$	r3	3
03	\$E	<u>+1\$</u>	s6	

State	Action						Goto	
	*	+	0	1	\$	E	B	
0			s1	s2		3	4	
1	r4	r4	r4	r4	r4			
2	r5	r5	r5	r5	r5			
3	s5	s6			acc			
4	r3	r3	r3	r3	r3			
5			s1	s2				7
6			s1	s2				8
7	r1	r1	r1	r1	r1			
8	r2	r2	r2	r2	r2			



Example: Parsing steps

- (1) $E \rightarrow E * B$
- (2) $E \rightarrow E + B$
- (3) $E \rightarrow B$
- (4) $B \rightarrow 0$
- (5) $B \rightarrow 1$

Stack		input	ACTION	GOTO
0	\$	1+1\$	s2	
02	\$1	+1\$	r5	4
04	\$B	+1\$	r3	3
03	\$E	+1\$	s6	
03 <u>6</u>	\$E+	<u>1\$</u>	s2	

State	Action						Goto	
	*	+	0	1	\$	E	B	
0			s1	s2		3	4	
1	r4	r4	r4	r4	r4			
2	r5	r5	r5	r5	r5			
3	s5	s6			acc			
4	r3	r3	r3	r3	r3			
5			s1	s2				7
6			s1	s2				8
7	r1	r1	r1	r1	r1			
8	r2	r2	r2	r2	r2			



Example: Parsing steps

- (1) $E \rightarrow E * B$
- (2) $E \rightarrow E + B$
- (3) $E \rightarrow B$
- (4) $B \rightarrow 0$
- (5) $B \rightarrow 1$

Stack		input	ACTION	GOTO
0	\$	1+1\$	s2	
02	\$1	+1\$	r5	4
04	\$B	+1\$	r3	3
03	\$E	+1\$	s6	
036	\$E+	1\$	s2	
036 <u>2</u>	\$E+1	\$	r5	8

State	Action						Goto	
	*	+	0	1	\$	E	B	
0			s1	s2		3	4	
1	r4	r4	r4	r4	r4			
2	r5	r5	r5	r5	r5			
3	s5	s6			acc			
4	r3	r3	r3	r3	r3			
5			s1	s2			7	
6			s1	s2				8
7	r1	r1	r1	r1	r1			
8	r2	r2	r2	r2	r2			



Example: Parsing steps

- (1) $E \rightarrow E * B$
- (2) $E \rightarrow E + B$
- (3) $E \rightarrow B$
- (4) $B \rightarrow 0$
- (5) $B \rightarrow 1$

Stack		input	ACTION	GOTO
0	\$	1+1\$	s2	
02	\$1	+1\$	r5	4
04	\$B	+1\$	r3	3
03	\$E	+1\$	s6	
036	\$E+	1\$	s2	
0362	\$E+1	\$	r5	8
036	\$E+	\$		

State	Action						Goto	
	*	+	0	1	\$	E	B	
0			s1	s2		3	4	
1	r4	r4	r4	r4	r4			
2	r5	r5	r5	r5	r5			
3	s5	s6			acc			
4	r3	r3	r3	r3	r3			
5			s1	s2			7	
6			s1	s2				8
7	r1	r1	r1	r1	r1			
8	r2	r2	r2	r2	r2			



Example: Parsing steps

- (1) $E \rightarrow E * B$
- (2) $E \rightarrow E + B$
- (3) $E \rightarrow B$
- (4) $B \rightarrow 0$
- (5) $B \rightarrow 1$

Stack		input	ACTION	GOTO
0	\$	1+1\$	s2	
02	\$1	+1\$	r5	4
04	\$B	+1\$	r3	3
03	\$E	+1\$	s6	
036	\$E+	1\$	s2	
0362	\$E+1	\$	r5	8
036 <u>8</u>	\$E+B	<u>\$</u>	r2	3

State	Action						Goto	
	*	+	0	1	\$	E	B	
0			s1	s2		3	4	
1	r4	r4	r4	r4	r4			
2	r5	r5	r5	r5	r5			
3	s5	s6			acc			
4	r3	r3	r3	r3	r3			
5			s1	s2			7	
6			s1	s2				8
7	r1	r1	r1	r1	r1			
8	r2	r2	r2	r2	r2			



Example: Parsing steps

- (1) $E \rightarrow E * B$
- (2) $E \rightarrow E + B$
- (3) $E \rightarrow B$
- (4) $B \rightarrow 0$
- (5) $B \rightarrow 1$

Stack		input	ACTION	GOTO
0	\$	1+1\$	s2	
02	\$1	+1\$	r5	4
04	\$B	+1\$	r3	3
03	\$E	+1\$	s6	
036	\$E+	1\$	s2	
0362	\$E+1	\$	r5	8
0368	\$E+B	\$	r2	3
0	\$	\$		

State	Action						Goto	
	*	+	0	1	\$	E	B	
0			s1	s2		3	4	
1	r4	r4	r4	r4	r4			
2	r5	r5	r5	r5	r5			
3	s5	s6			acc			
4	r3	r3	r3	r3	r3			
5			s1	s2			7	
6			s1	s2				8
7	r1	r1	r1	r1	r1			
8	r2	r2	r2	r2	r2			



Example: Parsing steps

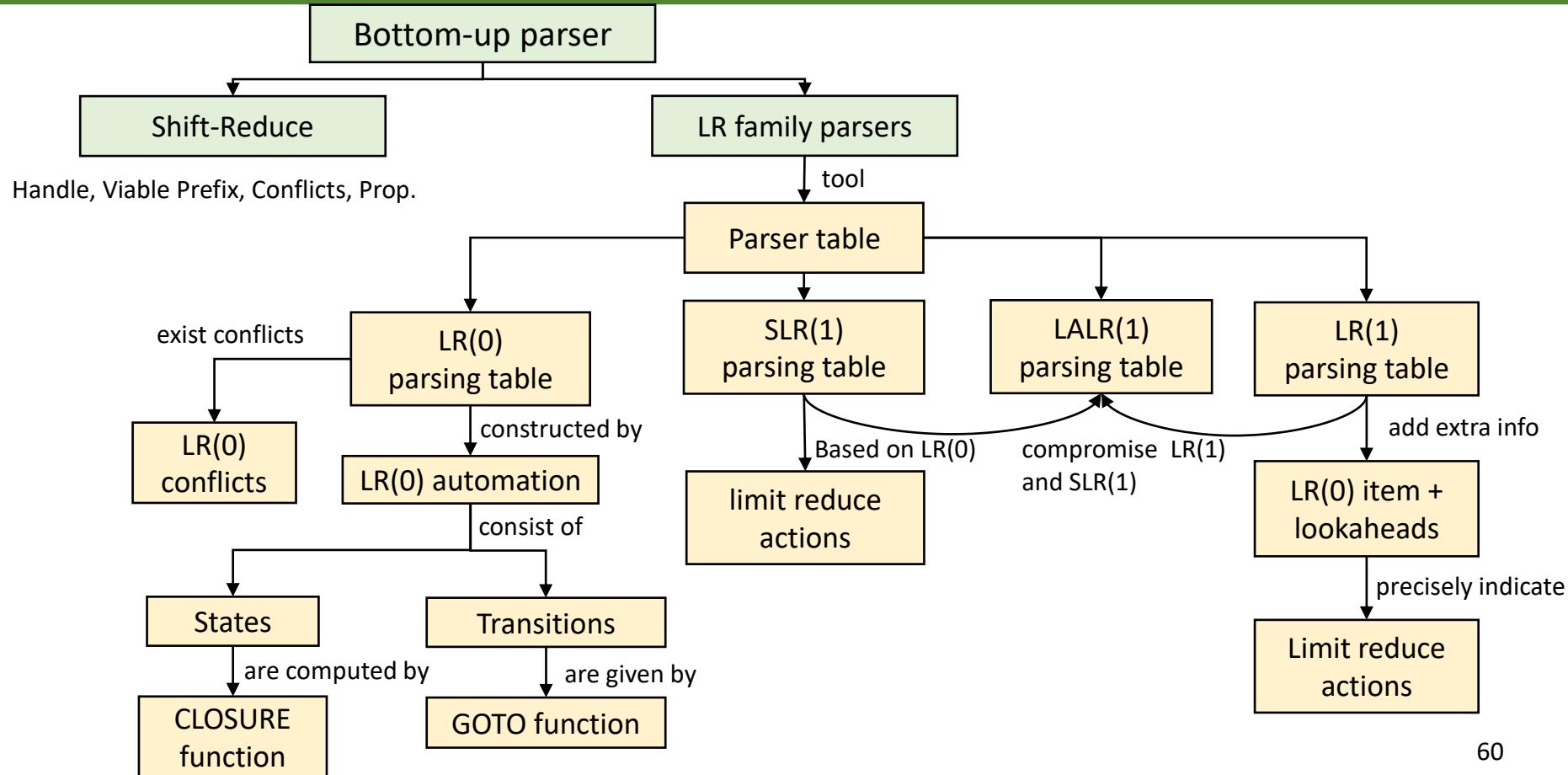
- (1) $E \rightarrow E * B$
- (2) $E \rightarrow E + B$
- (3) $E \rightarrow B$
- (4) $B \rightarrow 0$
- (5) $B \rightarrow 1$

Stack		input	ACTION	GOTO
0	\$	1+1\$	s2	
02	\$1	+1\$	r5	4
04	\$B	+1\$	r3	3
03	\$E	+1\$	s6	
036	\$E+	1\$	s2	
0362	\$E+1	\$	r5	8
0368	\$E+B	\$	r2	3
03	\$E	\$	acc	

State	ACTION						GOTO	
	*	+	0	1	\$	E	B	
0			s1	s2		3	4	
1	r4	r4	r4	r4	r4			
2	r5	r5	r5	r5	r5			
3	s5	s6			acc			
4	r3	r3	r3	r3	r3			
5			s1	s2			7	
6			s1	s2				8
7	r1	r1	r1	r1	r1			
8	r2	r2	r2	r2	r2			

How to build this table?

Revisit



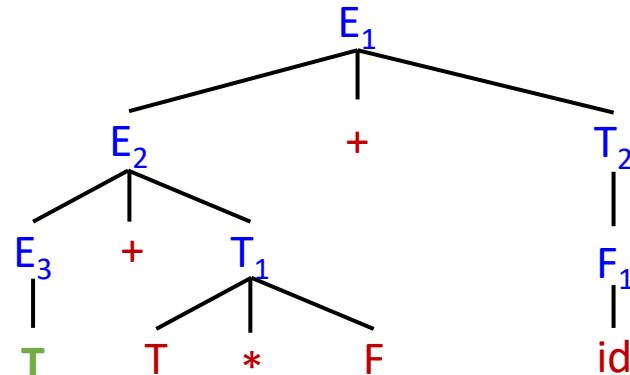
Bottom-up (Revisit)

- **Shift-Reduce** parser:

- From the input to the start symbol, rightmost derivation in reverse.
 - ◆ Shift[移入], Reduce[规约], Accept[接受], Error[报错]
 - ◆ When to shift/reduce and What production to apply

- ◆ Handle tells. But how to find it?

- ◆ Right-sentential form[最右句型], Phrase[短语],
- ◆ Simple Phrase[直接短语], Handle[句柄], Viable Prefix[活前缀]



- ◆ Conflicts

- ◆ Shift-reduce, reduce-reduce
- ◆ LR(0) cannot resolve conflicts unless we write the grammar.

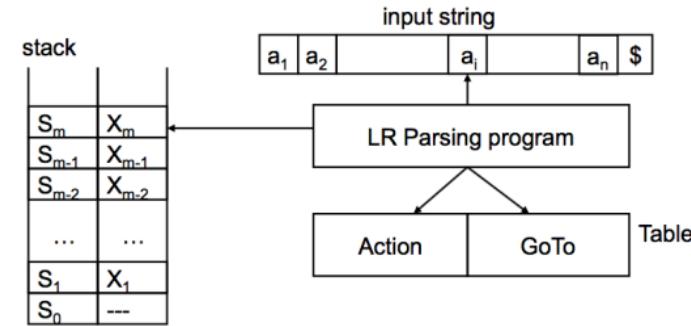
Bottom-up (Revisit)

- **LR(0) parser:**

- The stack holds a sequence of **states**, $S_0 S_1 \dots S_m$ (S_m is the top)
- stack + input ($X_1 X_2 \dots X_m a_i \dots a_n$) is a right sentential form
- Driven by an **Action&Goto** parser table

- Uses $[S_i, a_i]$ to index into parsing table to determine action:

- **ACTION** $[S_m, a_i]$:
 - **Shift** (s_x): Shift the **next input** symbol with state x
 - **Reduce** (r_y): reduce the top $|\beta|$ symbols by the y^{th} production
- **GOTO** $[S_i, A]$:
 - Indexed after each reduction
 - shows the new state to push based on S_i (state on the top after the reduction) and A (new non-terminal being pushed)



State	Action					Goto	
	*	+	0	1	\$	E	B
0			s1	s2		3	4
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	s5	s6			acc		
4	r3	r3	r3	r3	r3		
5			s1	s2			7
6			s1	s2			8
7	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2		

Construct Parse Table

- Construct parsing table:
 - identify the **possible states** and arrange the **transitions** among them.
- Different Parsing tables
 - **LR(0)** – Too weak (no lookahead)
 - Simplest LR parsing, only considers stack to decide shift/reduce
 - **SLR(1)** – Simple LR, 1 token lookahead
 - lookahead from first/follow rules derived from LR(0)
 - Keeps table as small as LR(0)
 - **LALR(1)** – Most common, 1 token lookahead
 - fancier lookahead analysis using the same LR(0) automaton as SLR(1)
 - **LR(1)** – 1 token lookahead, complex algorithms and big tables
 - **LR(k)** – k tokens lookahead, even bigger tables

State	Action						Goto	
	*	+	0	1	\$	E		
0			s1	s2		3	4	
1	r4	r4	r4	r4	r4			
2	r5	r5	r5	r5	r5	r5		
3	s5	s6				ac		
4	r3	r3	r3	r3	r3	r3		
5			s1	s2				7
6			s1	s2				8
7	r1	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2	r2		

State in LR Parsing[状态]



- How does a shift-reduce parser know **when to shift and when to reduce**? [Parser怎么知道何时移进, 何时规约]
 - ◆ Using the figure below as an example, with stack content **\$T** and next input symbol *****, how does the parser know that **T** is not a handle, and we need to shift instead of reducing T to E? [对于LR(0) parser, 其实就是要让编译器能找到handle]
- An LR parser makes shift-reduce decisions by **maintaining states** to keep track of where we are in a parse [状态追踪]
 - ◆ States represent sets of “**items**”

Grammar

- $E \rightarrow T$
- $T \rightarrow F \mid T * F$
- $F \rightarrow id$

STACK	INPUT	ACTION
\$	$id_1 * id_2 \$$	shift
$\$ id_1$	$* id_2 \$$	reduce by $F \rightarrow id$
$\$ F$	$* id_2 \$$	reduce by $T \rightarrow F$
$\\$ T$	$* id_2 \$$	shift
$\$ T *$	$id_2 \$$	shift
$\$ T * id_2$	\$	reduce by $F \rightarrow id$
$\$ T * F$	\$	reduce by $T \rightarrow T * F$
$\$ T$	\$	reduce by $E \rightarrow T$
$\$ E$	\$	accept

- The parsing table is based on the notion of **LR(0) items** (simply called **items** here) which are **grammar rules with a special dot added somewhere in the right-hand side**.
- For example, the rule $A \rightarrow XYZ$ has the following four items:
 - ◆ $A \rightarrow \bullet XYZ$
 - ◆ $A \rightarrow X \bullet YZ$
 - ◆ $A \rightarrow XY \bullet Z$
 - ◆ $A \rightarrow XYZ \bullet$
- Rules of the form $A \rightarrow \epsilon$ have only a single item $A \rightarrow \bullet$

The meaning of Items



- $A \rightarrow \bullet XYZ$
 - ◆ Indicates that we **hope to see** a string derivable from **XYZ** next on the input
- $A \rightarrow X \bullet YZ$
 - ◆ Indicates that we have just **seen** on the input a string derivable from **X** and that we **hope next to see** a string derivable from **YZ**
- $A \rightarrow XY \bullet Z$
- $A \rightarrow XYZ \bullet$
 - ◆ Indicates that we have **seen the body** **XYZ** and that it may be time to **reduce** **XYZ** to **A**

- Example:
 - ◆ Suppose we are currently in this position: $A \rightarrow X \bullet YZ$
 - ◆ We have just recognized X and expect the upcoming input to contain a sequence derivable from YZ (say, $Y \rightarrow u|w$)
 - ◆ Y is further derivable from either u or w :
 - $A \rightarrow X \bullet YZ$
 - $Y \rightarrow \bullet u$
 - $Y \rightarrow \bullet w$
 - ◆ The above items can be placed into a set, called as **configuration set** [配置集] of the LR parser
- Parsing tables have **one state** corresponding to each set
 - ◆ The states can be modeled as a **finite automaton**, where we move from one state to another via transitions marked with a symbol of the CFG

Augmented Grammar[增广文法]



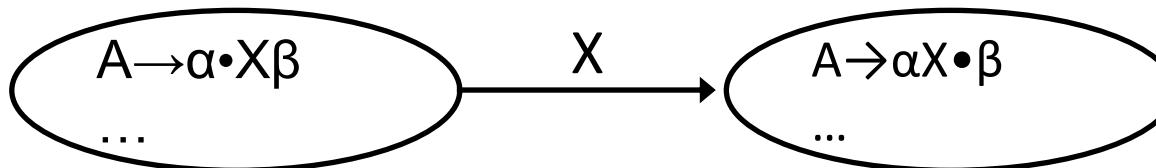
- We want to start with **an item with a dot before the start symbol $S (\bullet S)$** and move to **an item with a dot after $S (S \bullet)$**
 - ◆ Represents shifting and reducing an entire sentence of the grammar[完成了整个句子的移进规约]
 - ◆ Thus, we need S to appear on the right side of a production
- Modify the grammar by adding the production
 - ◆ $S' \rightarrow \bullet S$ becomes the **first item in the start state of the FA**

Grammar	Augmented Grammar
(0) $E \rightarrow E^* B$ (1) $E \rightarrow E + B$ (2) $E \rightarrow B$ (3) $B \rightarrow 0$ (4) $B \rightarrow 1$	\Rightarrow (0) $S \rightarrow E$ (1) $E \rightarrow E^* B$ (2) $E \rightarrow E + B$ (3) $E \rightarrow B$ (4) $B \rightarrow 0$ (5) $B \rightarrow 1$

Construct the LR(0) Automaton

- Each **FA state** corresponding to **a set of items**
- How to construct a state? -- **closure operation**
 - ◆ Closure: the action of adding equivalent items to a set
 - ◆ Example: $S' \rightarrow \cdot S$ $S \rightarrow \cdot BB$ $B \rightarrow \cdot aB$ $B \rightarrow \cdot b$
- How to construct transition form one state to another? -- **goto operation**
 - ◆ A transition on symbol **X** from state **i** to state **j**

(0) $S' \rightarrow S$
(1) $S \rightarrow BB$
(2) $B \rightarrow aB$
(3) $B \rightarrow b$

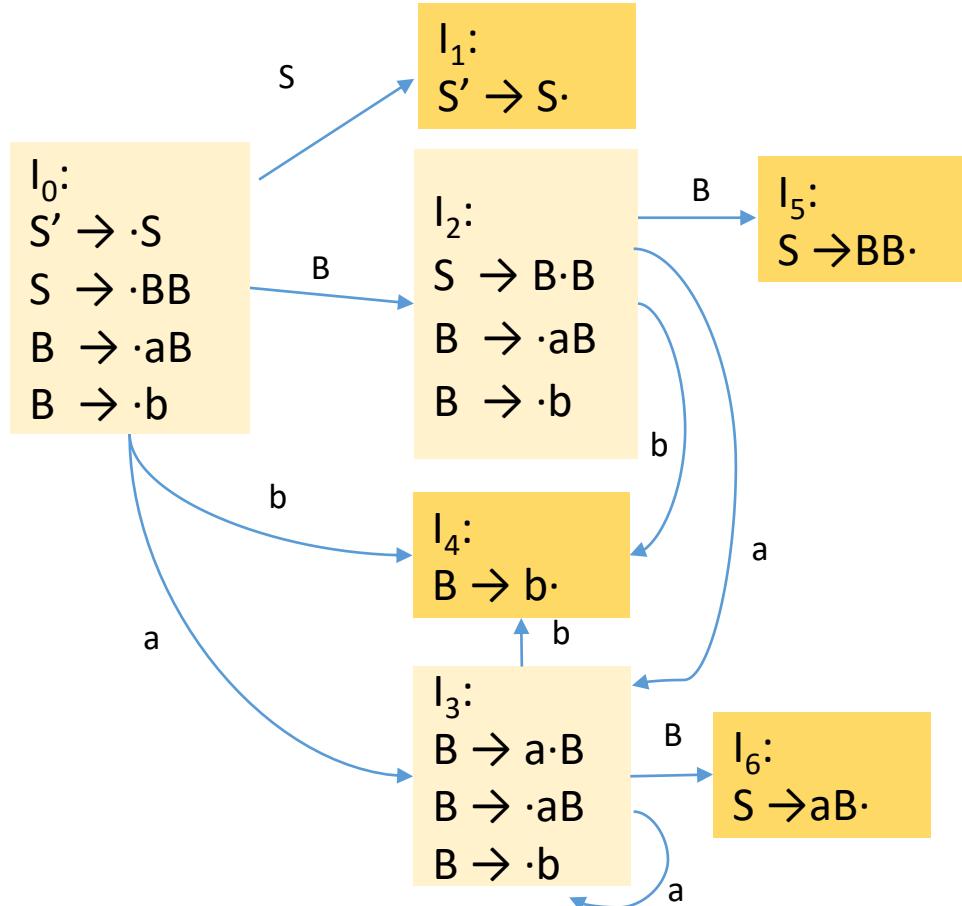


Example



Grammar

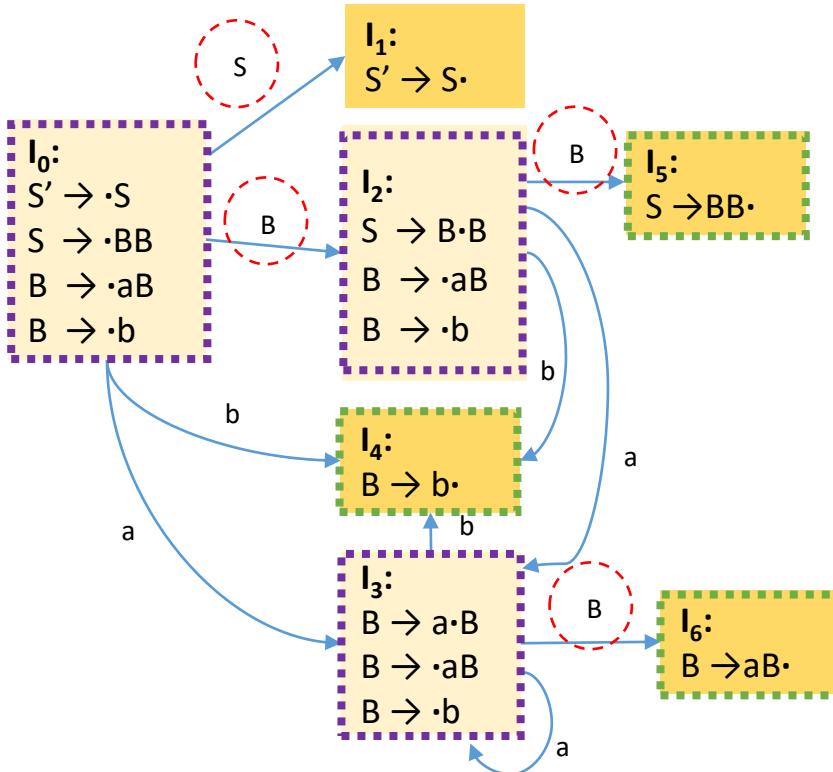
- (0) $S' \rightarrow S$
- (1) $S \rightarrow BB$
- (2) $B \rightarrow aB$
- (3) $B \rightarrow b$



Example (cont.)

Grammar

- (0) $S' \rightarrow S$
- (1) $S \rightarrow BB$
- (2) $B \rightarrow aB$
- (3) $B \rightarrow b$



State	ACTION			GOTO	
	a	b	\$	S	B
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

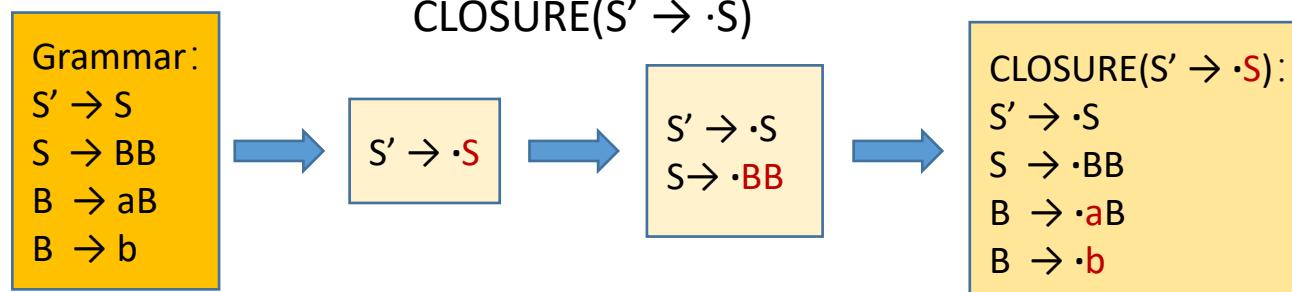
“state j ” refers to the state of the set of items I_j

How to construct this DFA?

CLOSURE() [闭包]

- **Closure of item sets:** if I is a set of items for a grammar G , then $\text{CLOSURE}(I)$ is the set of items constructed from I by the two rules:
 1. Initially, add every item in I to $\text{CLOSURE}(I)$
 2. For any item $A \rightarrow \alpha \cdot B\beta$ in $\text{CLOSURE}(I)$, if $B \rightarrow \gamma$ is a production, and $B \rightarrow \cdot\gamma$ is not already there , then add item $B \rightarrow \cdot\gamma$ to $\text{CLOSURE}(I)$

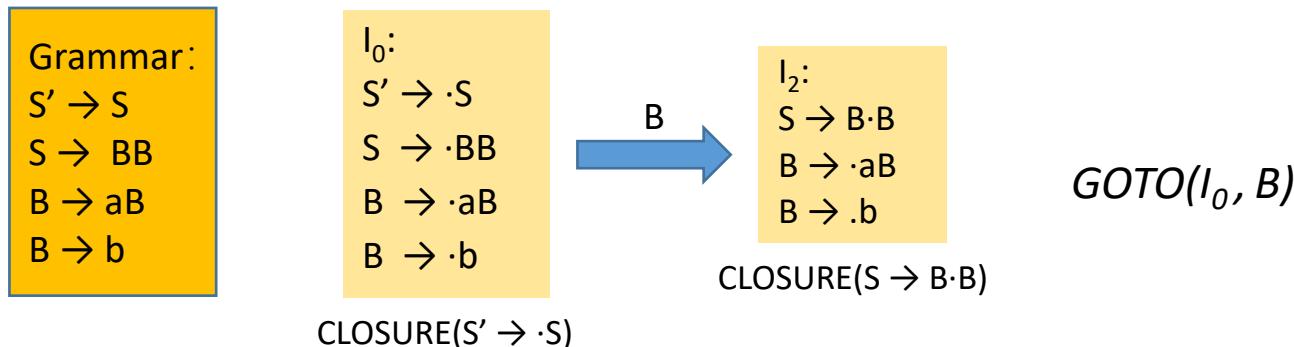
◆ Apply the above rules until no more new items can be added to $\text{CLOSURE}(I)$



Note: the location of the point •

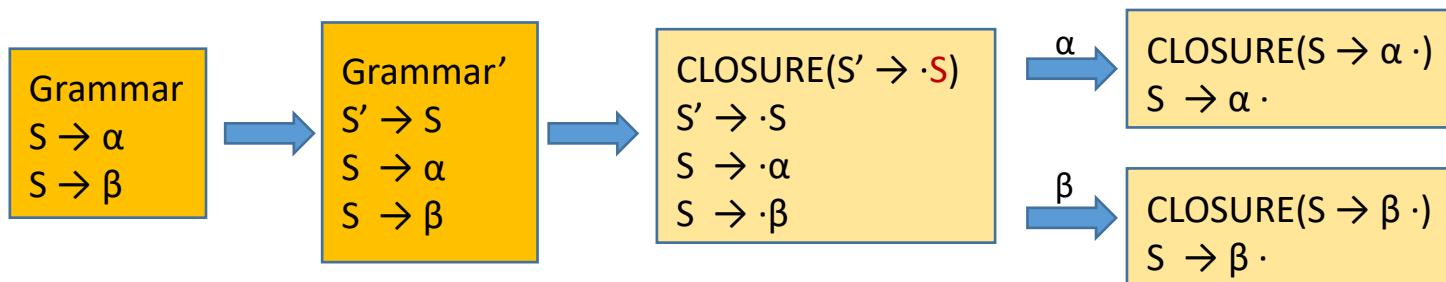
GOTO() [跳转]

- **GOTO(I, X)**: returns state (set of items) that can be reached by advancing I by X
 - ◆ I is a set of items and X is a grammar symbol
 - ◆ Return the closure of the set of all items $[A \rightarrow \alpha X \cdot \beta]$ when $[A \rightarrow \alpha \cdot X \beta]$ is in I
 - ◆ Used to define the transitions in the LR(0) automaton
 - The states of the automaton correspond to sets of items, and $\text{GOTO}(I, X)$ specifies the transition from the state for I under input X



Construct LR(0) States [构造LR(0)状态集]

- Create **augmented grammar** G' for G : [增广文法]
 - ◆ Given $G: S \rightarrow \alpha \mid \beta$, create $G': S' \rightarrow S, S \rightarrow \alpha \mid \beta$
 - ◆ Creates a single rule $S' \rightarrow S$ that when reduced, signals acceptance
- Create 1st state by performing a **closure** on initial item $S' \rightarrow \cdot S$ [初始状态]
 - ◆ $\text{Closure}(\{S' \rightarrow \cdot S\}) = \{S' \rightarrow \cdot S, S \rightarrow \cdot \alpha, S \rightarrow \cdot \beta\}$
- Create additional states by performing a **GOTO** on each symbol [添加状态]
 - ◆ $\text{Goto}(\{S' \rightarrow \cdot S, S \rightarrow \cdot \alpha, S \rightarrow \cdot \beta\}, \alpha) = \text{Closure}(\{S \rightarrow \alpha \cdot\}) = \{S \rightarrow \alpha \cdot\}$
- Repeatedly perform gotos until there are no more states to add [重复操作]



Construct DFA [构造DFA]



- Compute canonical LR(0) collection [规范LR(0)项集族, C], i.e., set of all states in DFA [DFA中的所有状态的集合]
 - ◆ A collection of sets of LR(0) items that provides the basis for constructing a DFA that is used to make parsing decisions
 - ◆ Each state of the automaton represents a set of items in the C, such an automaton is called an LR(0) automaton
- All new states are added through GOTO(I, X) [DFA中的所有边的集合]
 - ◆ State transitions are done on symbol X

```
void itemSet (G') {  
    C = { CLOSURE({[S' → ·S]}) };  
    while (no new states are added to C)  
        for (each state I in C)  
            for (each grammar symbol X)  
                if (GOTO(I, X) is not empty && is not in C)  
                    add GOTO (I, X) to C;  
    }  
}
```

LR(0) Automaton[自动机]



- **The LR(0) automaton**: a shift action means the transition of the current state to a new state
 - ◆ State: a set of items in **C**
 - Start state S_0 : **CLOSURE**($\{[S' \rightarrow \cdot S]\}$)
 - State S_j refers to the state corresponding to the set of items I_j
 - ◆ States are computed by the **CLOSURE** function
 - ◆ Transitions are given by the **GOTO** function
- How does the automaton help with shift-reduce decisions?
 - ◆ Suppose we are in a state j , and next input is a
 - ◆ Then, we choose shift when state j has a transition on a
 - ◆ Otherwise, we choose to reduce
 - The items in state j tell us which production to use
 - If there are not such item , the input a is illegal

The example

[建立第一个项]

$$S_0 = \text{Closure}(\{S' \rightarrow \cdot S\})$$

$$= \{S' \rightarrow \cdot S, S \rightarrow \cdot BB, B \rightarrow \cdot aB, B \rightarrow \cdot b\} \quad (I_0)$$

[通过跳转建立后续项]

$$\text{goto}(S_0, B) = \text{closure}(\{S \rightarrow B \cdot B\})$$

$$S_2 = \{S \rightarrow B \cdot B, B \rightarrow \cdot aB, B \rightarrow \cdot b\} \quad (I_2)$$

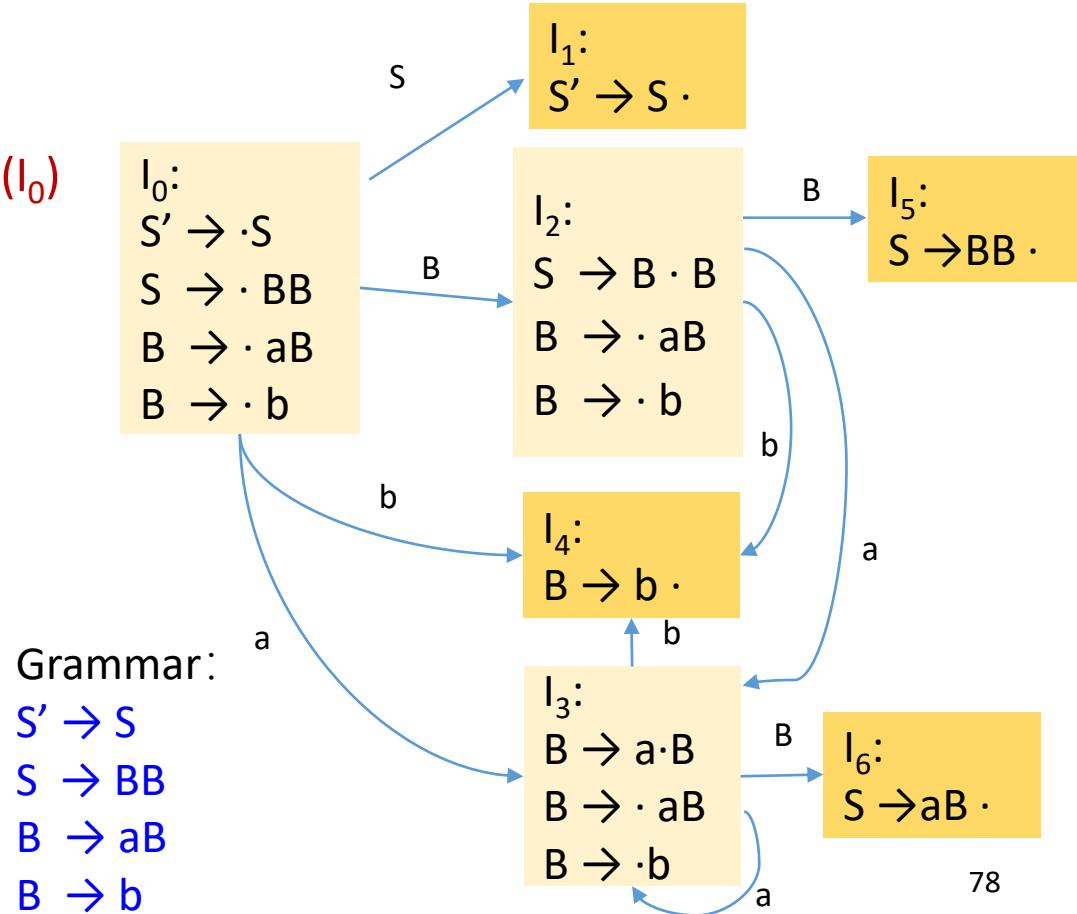
$$\text{goto}(S_0, a) = \text{closure}(\{B \rightarrow a \cdot B\})$$

$$S_3 = \{B \rightarrow a \cdot B, B \rightarrow \cdot aB, B \rightarrow \cdot b\} \quad (I_3)$$

$$\text{goto}(S_0, b) = \text{closure}(\{B \rightarrow b \cdot\})$$

$$S_4 = \{B \rightarrow b \cdot\} \quad (I_4)$$

...





Build Parse Table from DFA

- **ACTION** [S_i, a]:

- ◆ If $[A \rightarrow \alpha \cdot a\beta]$ is in S_i and $\text{goto}(S_i, a) = S_j$, where “ a ” is a terminal, then $\text{ACTION}[S_i, a] = S_j$ (shift j)
- ◆ If $[A \rightarrow \alpha \cdot]$ is in S_i and $A \rightarrow \alpha$ is j^{th} rule, then $\text{ACTION}[S_i, a] = r_j$ (reduce j)
- ◆ If $[S' \rightarrow S \cdot]$ is in S_i , then $\text{ACTION}[S_i, \$] = \text{accept}(\text{acc})$
- ◆ If no conflicts among ‘shift’ and ‘reduce’ (the first two ‘if’s above), then this parser is able to parse the given grammar

- **GOTO** [S_i, A]:

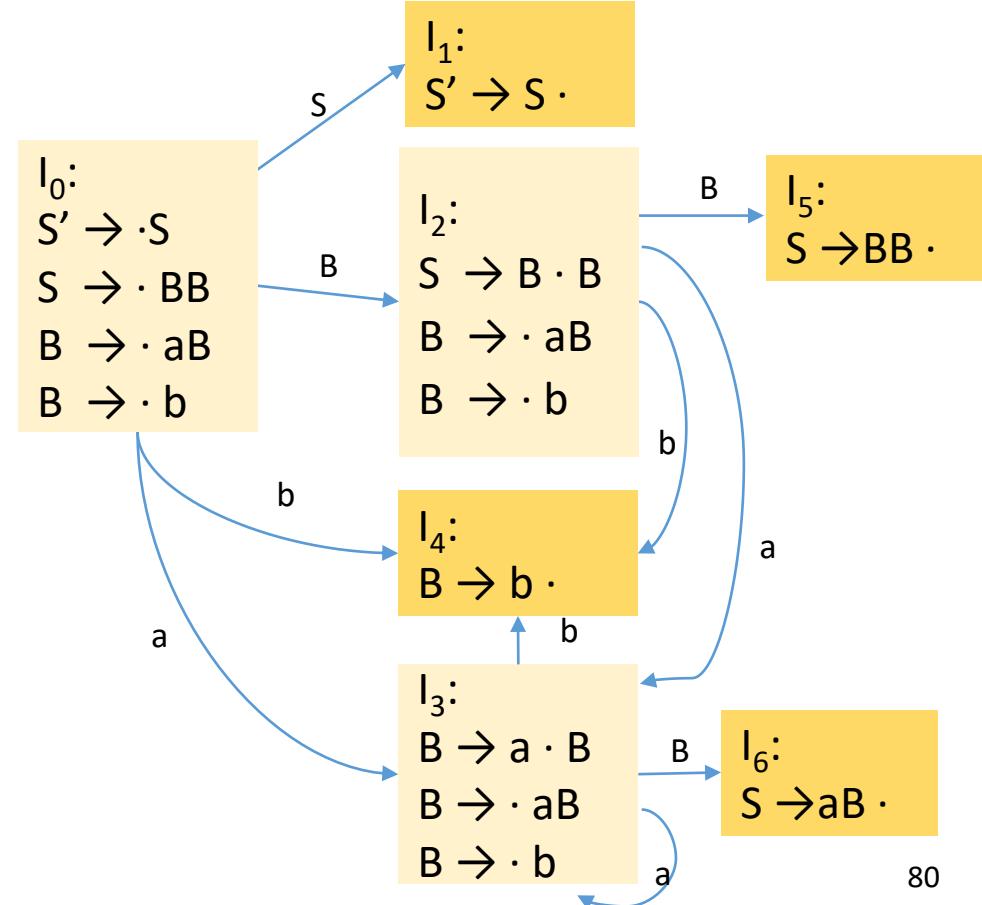
- ◆ if $\text{goto}(S_i, A) = S_j$ then $\text{GOTO}[S_i, A] = j$

The example

Grammar:

- (0) $S' \rightarrow S$
- (1) $S \rightarrow BB$
- (2) $B \rightarrow aB$
- (3) $B \rightarrow b$

State	ACTION			GOTO	
	a	b	\$	S	B
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		



LR(0) Parsing



- Construct LR(0) automaton from the Grammar
- Assumptions:
 - ◆ Input buffer contains α
 - ◆ Next input is t
 - ◆ After input α , the DFA is in state S
- Reduce if
 - ◆ S contains item $X \rightarrow \alpha\cdot$
- Shift if
 - ◆ S contains item $X \rightarrow \alpha\cdot t\omega$
 - ◆ Equivalent to saying S has a transition labeled t

LR(0) Parsing(cont.)

- The parser must be able to determine what action to take in each state **without looking at any further input symbols**
 - ◆ By only considering what the parsing stack contains so far
 - ◆ This is the ‘0’ in the parser name
- In an LR(0) table, each state must **only shift or reduce**
 - ◆ Thus, an LR(0) state can only have **exactly one reduce item** and **cannot have both shift and reduce items**
 - ◆ If the grammar contains the production $A \rightarrow \epsilon$, then the item $A \rightarrow \cdot\epsilon$ will create a **shift-reduce conflict** if there is any other non-null production for A
 - ϵ -rules are fairly common in programming language grammars

LR(0) Conflicts



- LR(0) has a **reduce-reduce conflict** if:

- ◆ Any state has two reduce items:
 - ◆ $X \rightarrow \beta \cdot$ and $Y \rightarrow \omega \cdot$

- LR(0) has a **shift-reduce conflict** if:

- ◆ Any state has a reduce item and a shift item:
 - ◆ $X \rightarrow \beta \cdot$ and $Y \rightarrow \omega \cdot t$

Grammar:

$$\begin{aligned} E' &\rightarrow E \\ E &\rightarrow E+T \mid T \mid V=E \\ T &\rightarrow (E) \mid i \mid i[E] \\ V &\rightarrow i \end{aligned}$$


$I_0:$

$$\begin{aligned} E' &\rightarrow \cdot E \\ E &\rightarrow \cdot T \\ E &\rightarrow \cdot E+T \\ E &\rightarrow \cdot V=E \\ T &\rightarrow \cdot(E) \\ T &\rightarrow \cdot i \\ T &\rightarrow \cdot i[E] \\ V &\rightarrow \cdot i \end{aligned}$$


$I_1:$

$$\begin{aligned} T &\rightarrow i \cdot \\ T &\rightarrow i \cdot [E] \\ V &\rightarrow i \cdot \end{aligned}$$

LR(0) Limitations[局限性]



- LR(0) conflicts are generally caused by **reduce** actions

- ◆ Shift-**Reduce** conflict

$$A \rightarrow \alpha \cdot \alpha \beta$$

- ◆ Reduce-Reduce conflict

$$B \rightarrow r \cdot$$

$$A \rightarrow \beta \cdot$$

$$B \rightarrow r \cdot$$

- ◆ If the item is complete ($A \rightarrow \alpha \cdot$), the parser must choose to reduce[项目形式完整就归约]
 - Is this always appropriate?
 - The next upcoming token may tell us something different
- ◆ What tokens may tell the reduction is not appropriate?
 - Perhaps **Follow(A)** could be useful here

LR(0) Parsing

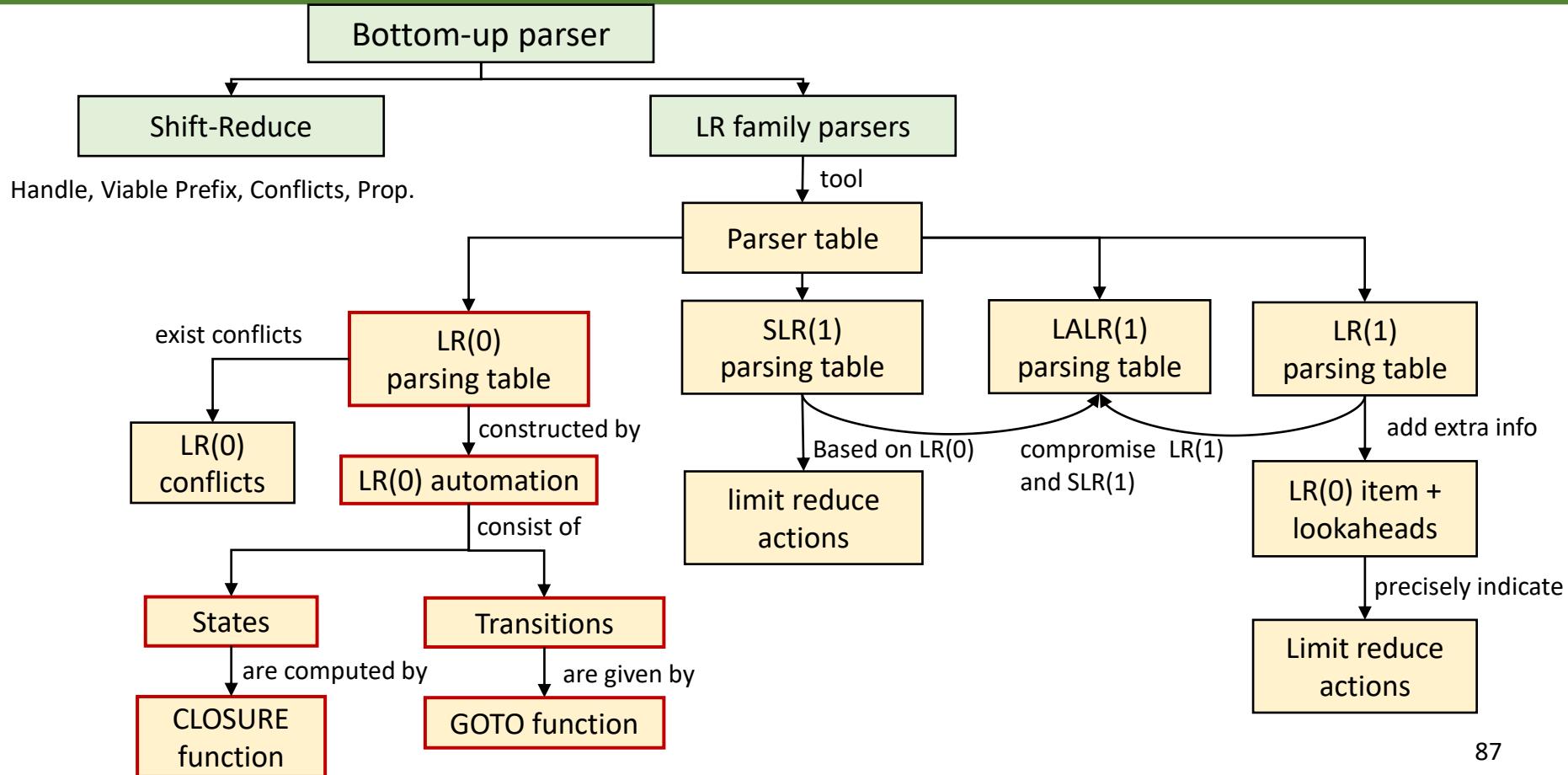


- Construct LR(0) automaton from the Grammar
- Assumptions:
 - ◆ Input buffer contains α
 - ◆ Next input is t
 - ◆ After input α , the DFA is in state S
- Reduce if
 - ◆ S contains item $X \rightarrow \alpha\cdot$
- Shift if
 - ◆ S contains item $X \rightarrow \alpha\cdot t\omega$
 - ◆ Equivalent to saying S has a transition labeled t

LR(0) Parsing(cont.)

- The parser must be able to determine what action to take in each state **without looking at any further input symbols**
 - ◆ By only considering what the parsing stack contains so far
 - ◆ This is the ‘0’ in the parser name
- In an LR(0) table, each state must **only shift or reduce**
 - ◆ Thus, an LR(0) state can only have **exactly one reduce item** and **cannot have both shift and reduce items**
 - ◆ If the grammar contains the production $A \rightarrow \epsilon$, then the item $A \rightarrow \cdot\epsilon$ will create a **shift-reduce conflict** if there is any other non-null production for A
 - ϵ -rules are fairly common in programming language grammars

Revisit



Bottom-up (Revisit)

- Construction of **LR(0) table**

- Tell the parser when to shift/reduce
- Answers which production to apply for a reduction

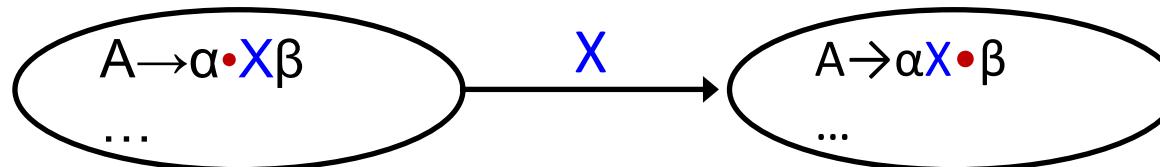
- **States and Items**

- **States**: a set of items indicating where we are
- **Items**: productions with a dot

- ◆ $A \rightarrow X \bullet YZ$ (Looking forward to Y)
- ◆ $A \rightarrow XYZ \bullet$ (all symbols ready, can go for the reduction)

- Parsing tables have **one state** corresponding to a set of items (a.k.a the configuration set)

- The **states and their transitions** can be implemented by **finite automata**.

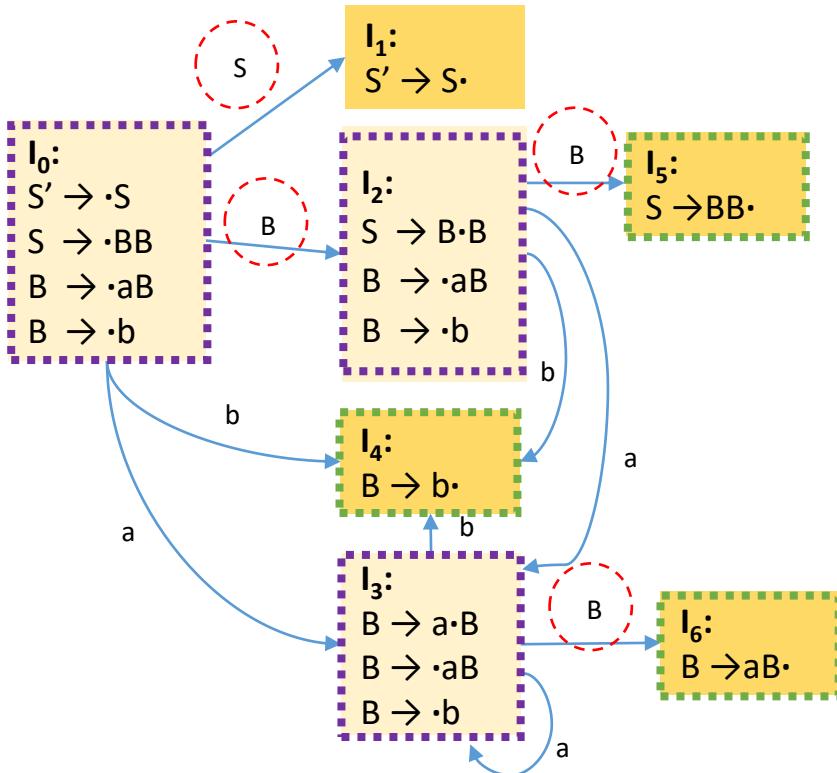


Stat e	Action						Goto	
	*	+	0	1	\$	E	B	
0			s1	s2		3	4	
1	r4	r4	r4	r4	r4			
2	r5	r5	r5	r5	r5			
3	s5	s6				acc		
4	r3	r3	r3	r3	r3			
5			s1	s2				7
6			s1	s2				8
7	r1	r1	r1	r1	r1			
8	r2	r2	r2	r2	r2			

Example (Revisit)

Grammar

- (0) $S' \rightarrow S$
- (1) $S \rightarrow BB$
- (2) $B \rightarrow aB$
- (3) $B \rightarrow b$



State	ACTION			GOTO	
	a	b	\$	S	B
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

“state j ” refers to the state of the set of items I_j

Bottom-up (Revisit)

- We want to see $S \bullet$ from $\bullet S$
- The construction of **LR(0)** automaton
 - Augmented grammar G' for G
 - Add $S' \rightarrow S$
 - Building states and transitions: **closure** and **goto**
- **CLOSURE:**

Grammar:
 $S' \rightarrow S$
 $S \rightarrow BB$
 $B \rightarrow aB$
 $B \rightarrow b$

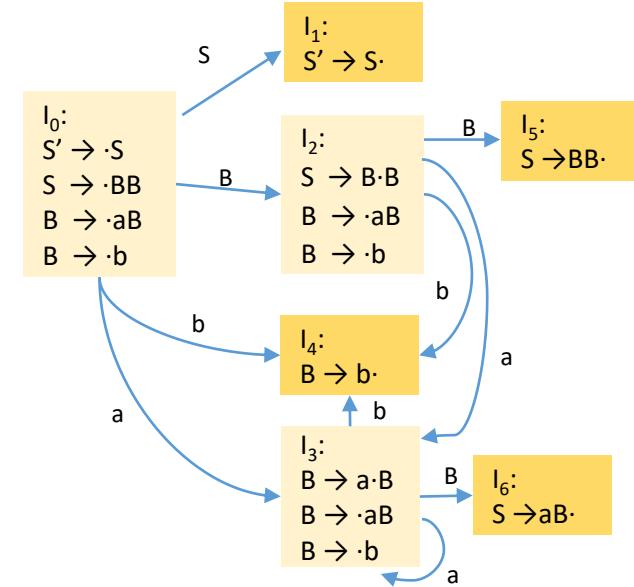
CLOSURE($S' \rightarrow \cdot S$)

$S' \rightarrow \cdot S$

$S' \rightarrow \cdot S$
 $S \rightarrow \cdot BB$

CLOSURE($S' \rightarrow \cdot S$):
 $S' \rightarrow \cdot S$
 $S \rightarrow \cdot BB$
 $B \rightarrow \cdot aB$
 $B \rightarrow \cdot b$

Note: the location of the point \cdot



Bottom-up (Revisit)

- We want to see $S \bullet$ from $\bullet S$
- The construction of **LR(0)** automaton
 - Augmented grammar G' for G
 - Add $S' \rightarrow S$
 - Building states and transitions: **closure** and **goto**
 - **GOTO:**

Grammar:
 $S' \rightarrow S$
 $S \rightarrow BB$
 $B \rightarrow aB$
 $B \rightarrow b$

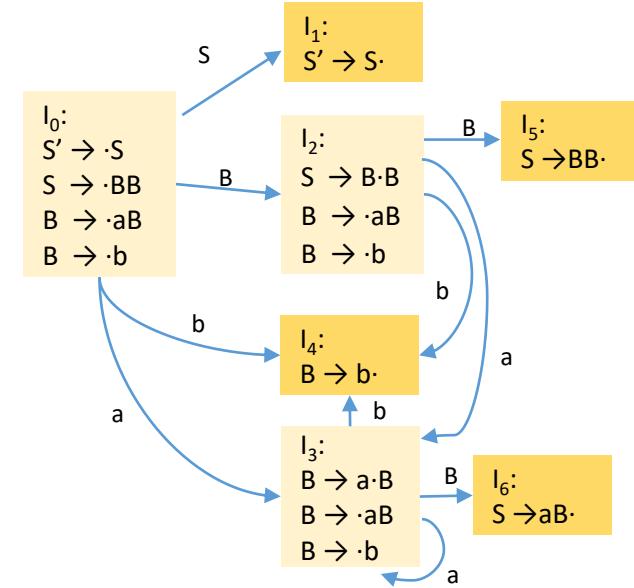
$I_0:$
 $S' \rightarrow \cdot S$
 $S \rightarrow \cdot BB$
 $B \rightarrow \cdot aB$
 $B \rightarrow \cdot b$

$GOTO(I_0, B)$

\xrightarrow{B}

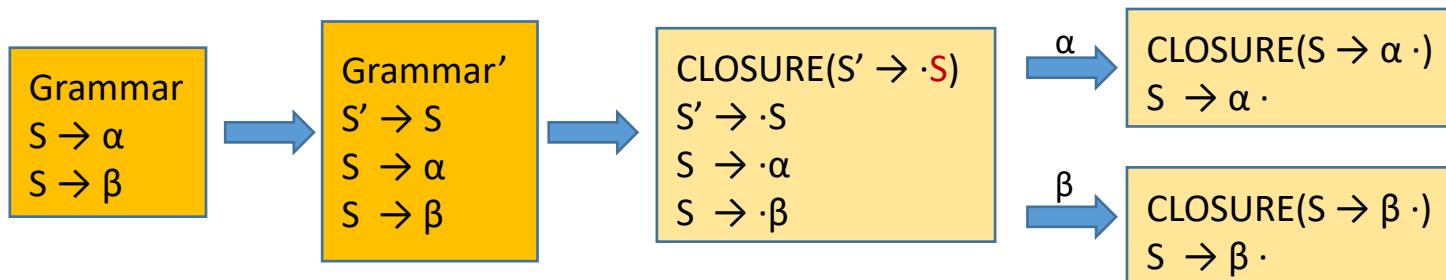
$I_2:$
 $S \rightarrow B \cdot B$
 $B \rightarrow \cdot aB$
 $B \rightarrow \cdot b$

CLOSURE($S \rightarrow B \cdot B$)



Construct LR(0) States (Revisit)

- Create **augmented** grammar G' for G : [增广文法]
 - ◆ Given $G: S \rightarrow \alpha \mid \beta$, create $G': S' \rightarrow S, S \rightarrow \alpha \mid \beta$
 - ◆ Creates a single rule $S' \rightarrow S$ that when reduced, signals acceptance
- Create 1st state by performing a **closure** on initial item $S' \rightarrow \cdot S$ [初始状态]
 - ◆ $\text{Closure}(\{S' \rightarrow \cdot S\}) = \{S' \rightarrow \cdot S, S \rightarrow \cdot \alpha, S \rightarrow \cdot \beta\}$
- Create additional states by performing a **GOTO** on each symbol [添加状态]
 - ◆ $\text{Goto}(\{S' \rightarrow \cdot S, S \rightarrow \cdot \alpha, S \rightarrow \cdot \beta\}, \alpha) = \text{Closure}(\{S \rightarrow \alpha \cdot\}) = \{S \rightarrow \alpha \cdot\}$
- Repeatedly perform gotos until there are no more states to add [重复操作]



The example (Revisit)

[建立第一个项]

$$S_0 = \text{Closure}(\{S' \rightarrow \cdot S\})$$

$$= \{S' \rightarrow \cdot S, S \rightarrow \cdot BB, B \rightarrow \cdot aB, B \rightarrow \cdot b\} \quad (I_0)$$

[通过跳转建立后续项]

$$\text{goto}(S_0, B) = \text{closure}(\{S \rightarrow B \cdot B\})$$

$$S_2 = \{S \rightarrow B \cdot B, B \rightarrow \cdot aB, B \rightarrow \cdot b\} \quad (I_2)$$

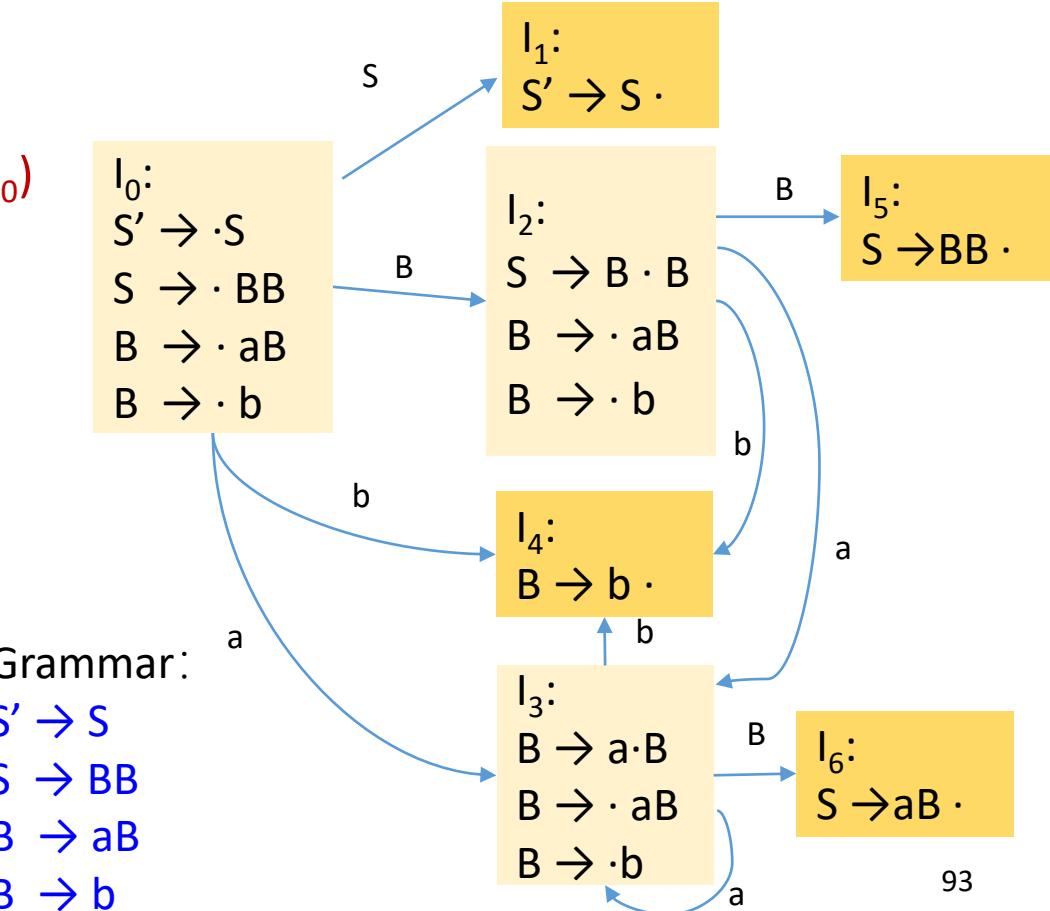
$$\text{goto}(S_0, a) = \text{closure}(\{B \rightarrow a \cdot B\})$$

$$S_3 = \{B \rightarrow a \cdot B, B \rightarrow \cdot aB, B \rightarrow \cdot b\} \quad (I_3)$$

$$\text{goto}(S_0, b) = \text{closure}(\{B \rightarrow b \cdot\})$$

$$S_4 = \{B \rightarrow b \cdot\} \quad (I_4)$$

...

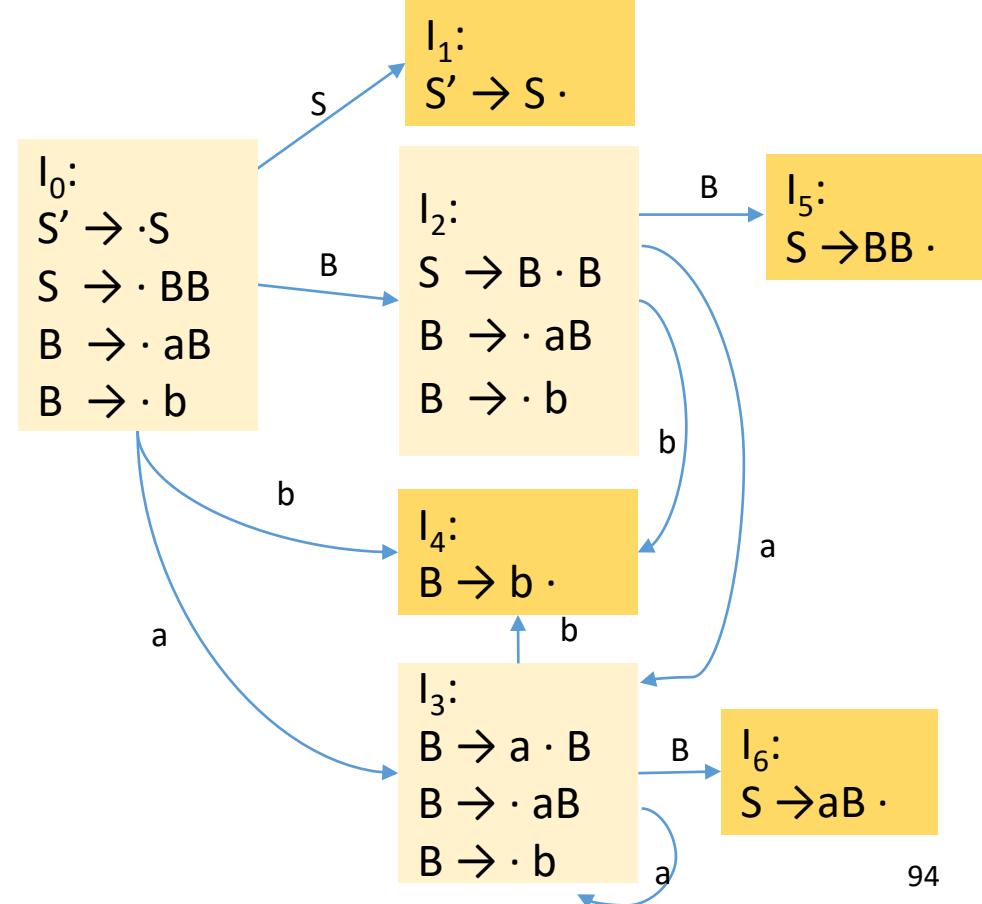


The example (Revisit)

Grammar:

- (0) $S' \rightarrow S$
- (1) $S \rightarrow BB$
- (2) $B \rightarrow aB$
- (3) $B \rightarrow b$

State	ACTION			GOTO	
	a	b	\$	S	B
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		



LR(0) Conflicts



- LR(0) has a **reduce-reduce conflict** if:

- ◆ Any state has two reduce items:
 - ◆ $X \rightarrow \beta \cdot$ and $Y \rightarrow \omega \cdot$

- LR(0) has a **shift-reduce conflict** if:

- ◆ Any state has a reduce item and a shift item:
 - ◆ $X \rightarrow \beta \cdot$ and $Y \rightarrow \omega \cdot t$

Grammar:

$$\begin{aligned} E' &\rightarrow E \\ E &\rightarrow E+T \mid T \mid V=E \\ T &\rightarrow (E) \mid i \mid i[E] \\ V &\rightarrow i \end{aligned}$$


$I_0:$

$$\begin{aligned} E' &\rightarrow \cdot E \\ E &\rightarrow \cdot T \\ E &\rightarrow \cdot E+T \\ E &\rightarrow \cdot V=E \\ T &\rightarrow \cdot(E) \\ T &\rightarrow \cdot i \\ T &\rightarrow \cdot i[E] \\ V &\rightarrow \cdot i \end{aligned}$$


$I_1:$

$$\begin{aligned} T &\rightarrow i \cdot \\ T &\rightarrow i \cdot [E] \\ V &\rightarrow i \cdot \end{aligned}$$

LR(0) Limitations[局限性]



- LR(0) conflicts are generally caused by **reduce** actions

- ◆ Shift-**Reduce** conflict

$$A \rightarrow \alpha \cdot \alpha \beta$$

- ◆ Reduce-Reduce conflict

$$B \rightarrow r \cdot$$

$$A \rightarrow \beta \cdot$$

$$B \rightarrow r \cdot$$

- ◆ If the item is complete ($A \rightarrow \alpha \cdot$), the parser must choose to reduce[项目形式完整就归约]

- Is this always appropriate?

- The next upcoming token may tell us something different

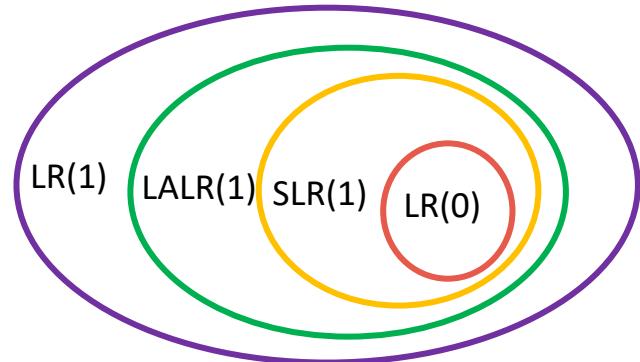
- ◆ What tokens may tell the reduction is not appropriate?

- Perhaps **Follow(A)** could be useful here

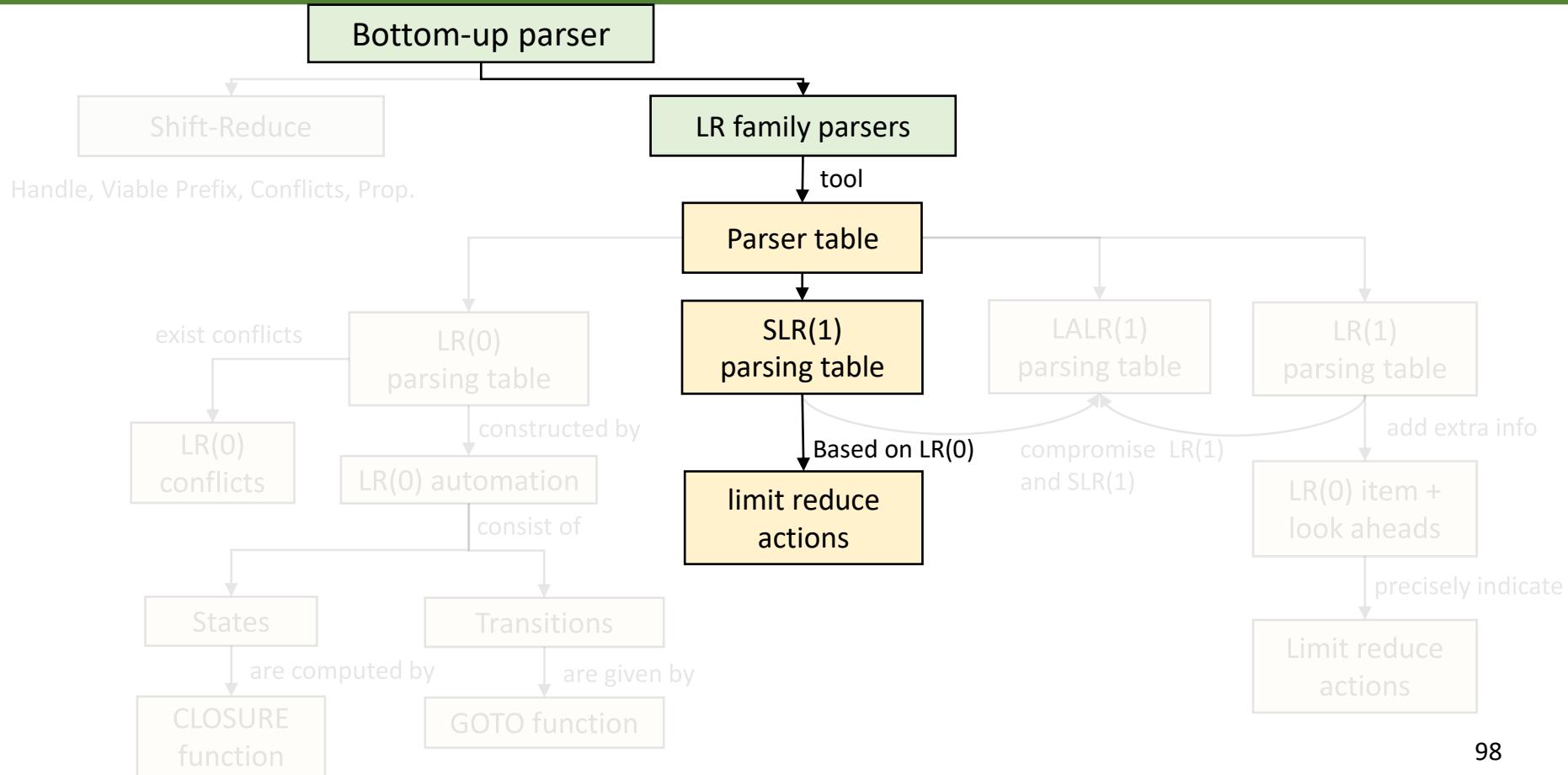
- [如果我们知道Follow(A)没有next input a, 我们就不能用 $A \rightarrow \beta$ 规约, 不然肯定会报错, 因为没有能推导出 Aa 的产生式]

LR(0) Summary

- LR(0) is the **simplest** LR parsing
 - ◆ Table-driven **shift-reduce** parser
 - Action table[s, **a**] + Goto table[s, **X**]
 - ◆ Weakest, not used much in practice
 - ◆ Parses without using any lookahead
- Adding just one token of lookahead vastly increases the parsing power [考慮展望]
 - ◆ **SLR(1)**: simple LR(1), use **FOLLOW** [用FOLLOW判断规约]
 - ◆ **LR(1)**: use stack info + dedicated symbols [比FOLLOW更精细]
 - ◆ **LALR(1)**: balance SLR(1) and LR(1)[折衷]



Contents



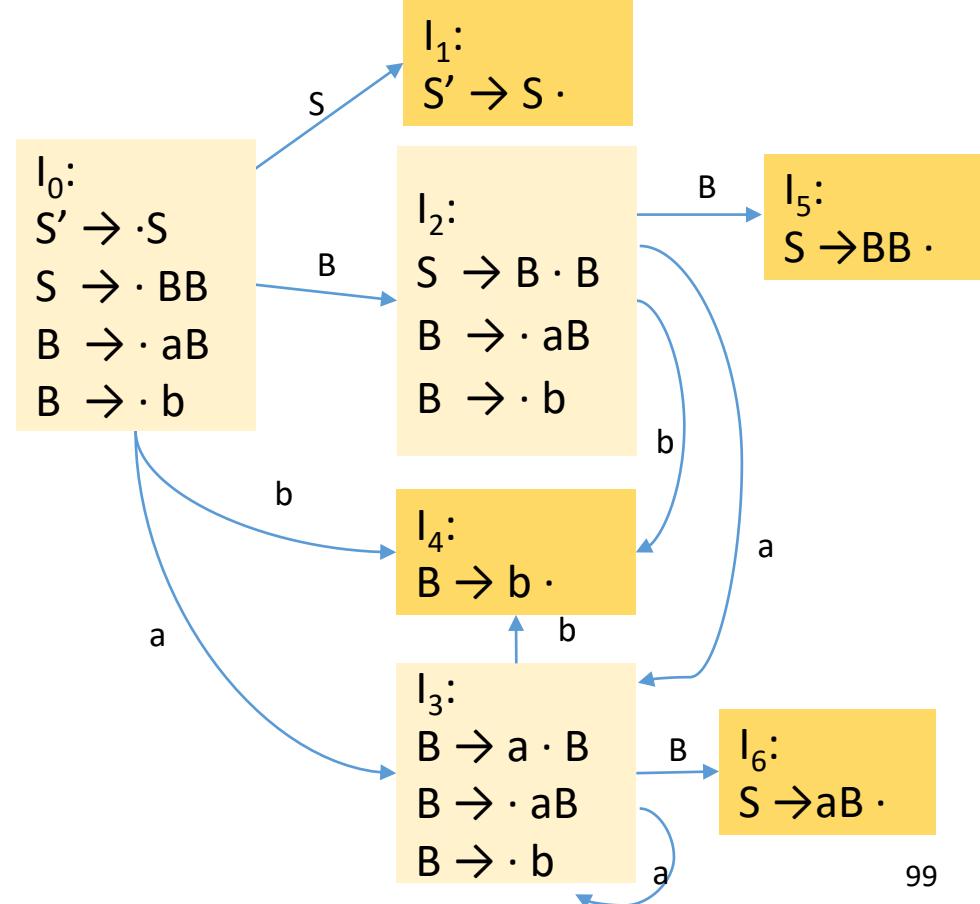
LR(0) Example (revisit)



Grammar:

- (0) $S' \rightarrow S$
- (1) $S \rightarrow BB$
- (2) $B \rightarrow aB$
- (3) $B \rightarrow b$

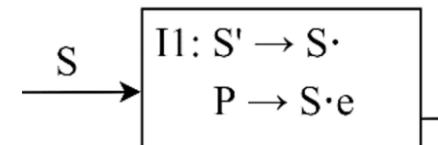
State	ACTION			GOTO	
	a	b	\$	S	B
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		



SLR(1) Parsing

- SLR (Simple LR)

- ◆ Use the same LR(0) configuring sets and have the same table structure and parser operation
- ◆ Allow both **shift** and **reduce** items in the same state as well as **multiple reduce** items.
- ◆ The SLR(1) parser will be able to determine which action to take as long as **the follow sets are disjoint**.
- ◆ The difference comes in assigning table actions
 - Use one token of lookahead to help arbitrate among the conflicts
 - Reduce only if the **next input token is a member of the FOLLOW set** of the nonterminal being reduced to [下一token在FOLLOW集才归约]



Example

- Suppose *id* is the first token of the input
 - S_1 : the set has a **shift-reduce** conflict and a **reduce-reduce** conflict
 - $\text{Follow}(T) = \{ +,),], \$ \}$, $\text{Follow}(V) = \{ = \}$
 - If input *t* in $\text{Follow}(T)$, it will **reduce to T**
 - If input *t* in $\text{Follow}(V)$, it will **reduce to V**
 - Else the input [will **shift**

$\text{FOLLOW}(A) = \{a \mid S \xrightarrow{*} \dots Aa\dots, a \in V_T\}$,
including \$[结束标记]

Grammar:
 $E' \rightarrow E$
 $E \rightarrow E+T \mid T \mid V=E$
 $T \rightarrow (E) \mid id \mid id[E]$
 $V \rightarrow id$



$I_0:$
 $E' \rightarrow \cdot E$
 $E \rightarrow \cdot T$
 $E \rightarrow \cdot E+T$
 $E \rightarrow \cdot V=E$
 $T \rightarrow \cdot (E)$
 $T \rightarrow \cdot id$
 $T \rightarrow \cdot id[E]$
 $V \rightarrow \cdot id$



$I_1:$
 $T \rightarrow id \cdot$
 $T \rightarrow id \cdot [E]$
 $V \rightarrow id \cdot$

- A grammar is SLR(1) if the following two conditions hold for each configuring set:
 1. For any item $A \rightarrow u \cdot xv$ in the set, with terminal x , there is no complete item $B \rightarrow w \cdot$ in that set with x in $\text{Follow}(B)$
 - In the tables, this translates **no shift-reduce conflict** on any state
 2. For any two complete items $A \rightarrow u \cdot$ and $B \rightarrow v \cdot$ in the set, the follow sets must be disjoint, i.e., $\text{Follow}(A) \cap \text{Follow}(B)$ is empty
 - This translates to **no reduce-reduce conflict** on any state
 - If more than one nonterminal could be reduced from this set, it must be possible to uniquely determine which using only one token of lookahead

SLR(1) Advantages [SLR(1)优势]



- SLR(1) v.s. LR(0)
 - ◆ Adding just one token of lookahead and using the Follow set **greatly expands the class of grammars** that can be parsed without conflict
- SLR(1) is a simple improvement over LR(0)
 - ◆ LR(0) easily gets shift-reduce and reduce-reduce conflicts
 - Always reduce on a completed item (might be too ambitious)
 - ◆ SLR(1) uses the same configuration sets (i.e., states), same table structure and parser operation
 - ◆ But SLR(1) reduces only if the next input token is in Follow set
 - i.e., different table actions from LR(0)
- SLR(1) is capable to determine which action to take as long as the **Follow sets are disjoint**
 - ◆ So, one state can have **both shift and reduce items** with multiple reduce items

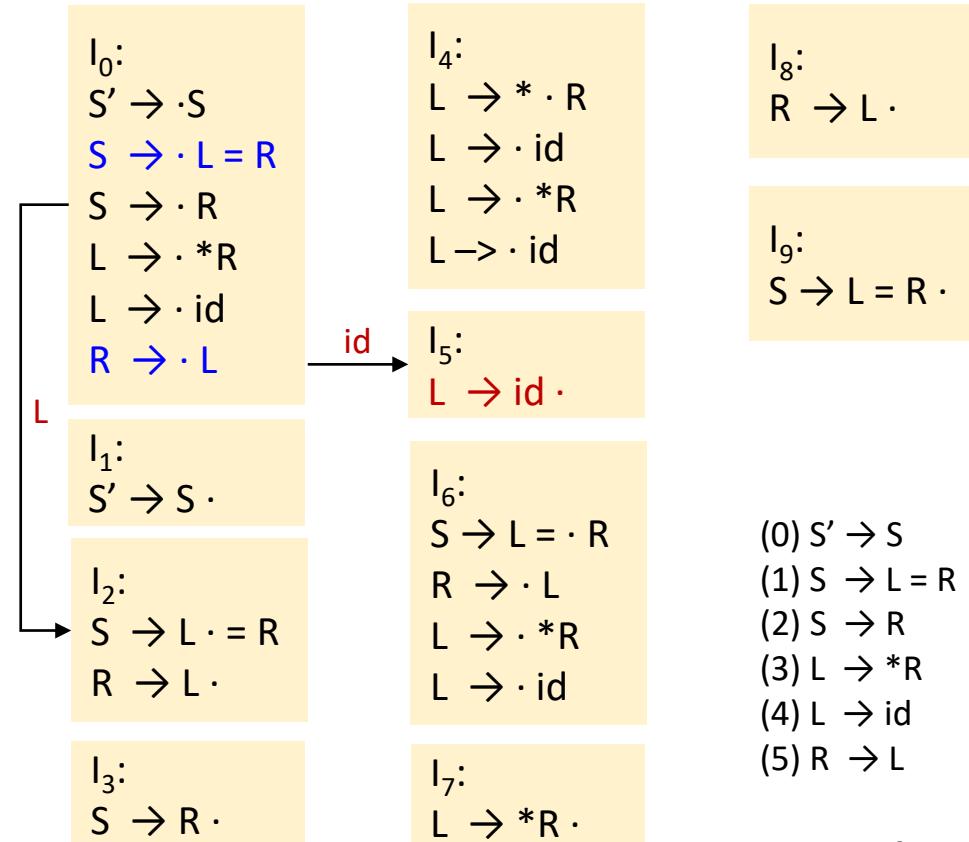
SLR(1) Limitations [SLR(1)局限性]



- When we have a complete configuration (i.e., dot at the end) such as $X \rightarrow u\cdot$, we know that it is reducible
 - ◆ We allow such a reduction as long as **the next symbol is in Follow(X)**.
- However, it may be that we should not reduce for every symbol in $\text{Follow}(X)$, because the **symbols below u** on the stack **preclude**[排除] u being a handle for reduction in this case
- In other words, SLR(1) states **only tell** us about **the sequence on top** of the stack, not what is below it on the stack

SLR(1) Limitation Example

- For input string : $\text{id} = \text{id}$
 - Initially, at S_0 , push id
 - Move to S_5 , after shifting id to stack (S_5 is also pushed to stack)
 - Reduce, and back to S_0 , and further GOTO S_2
 - S_5 has a completed item, and next input = is in Follow(L)
 - S_5 and id are popped from stack , and L is pushed onto stack
 - $\text{GOTO}(S_0, L) = S_2$

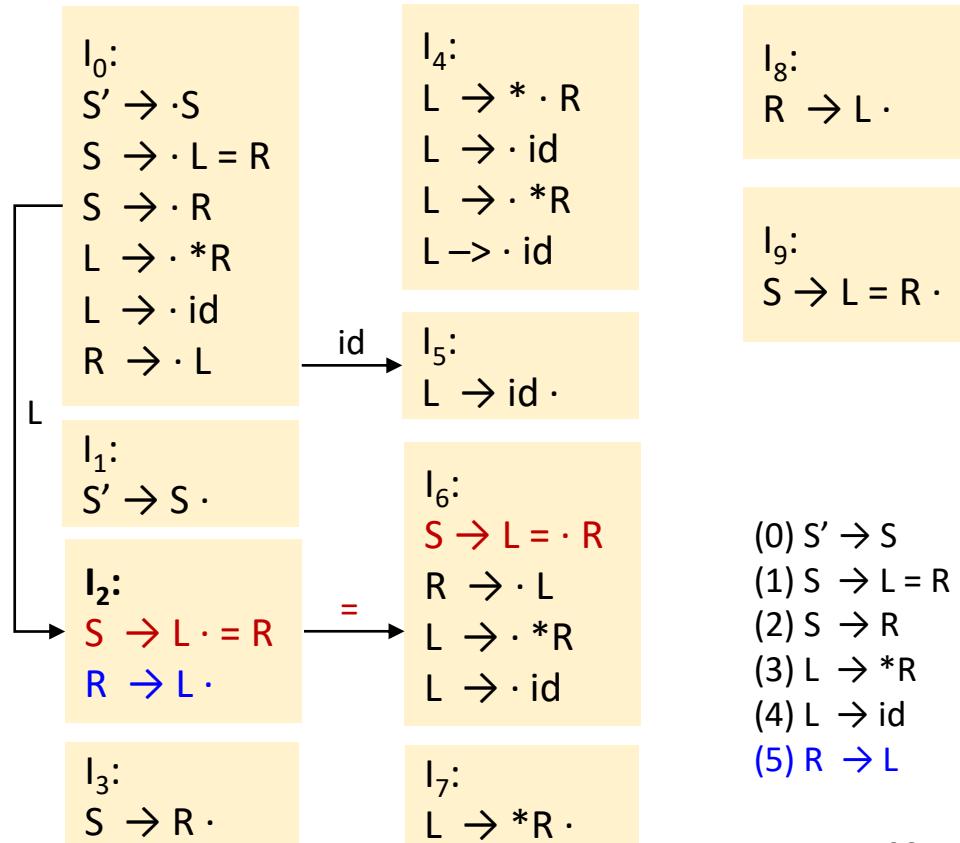


SLR(1) Limitation Example (Cont.)

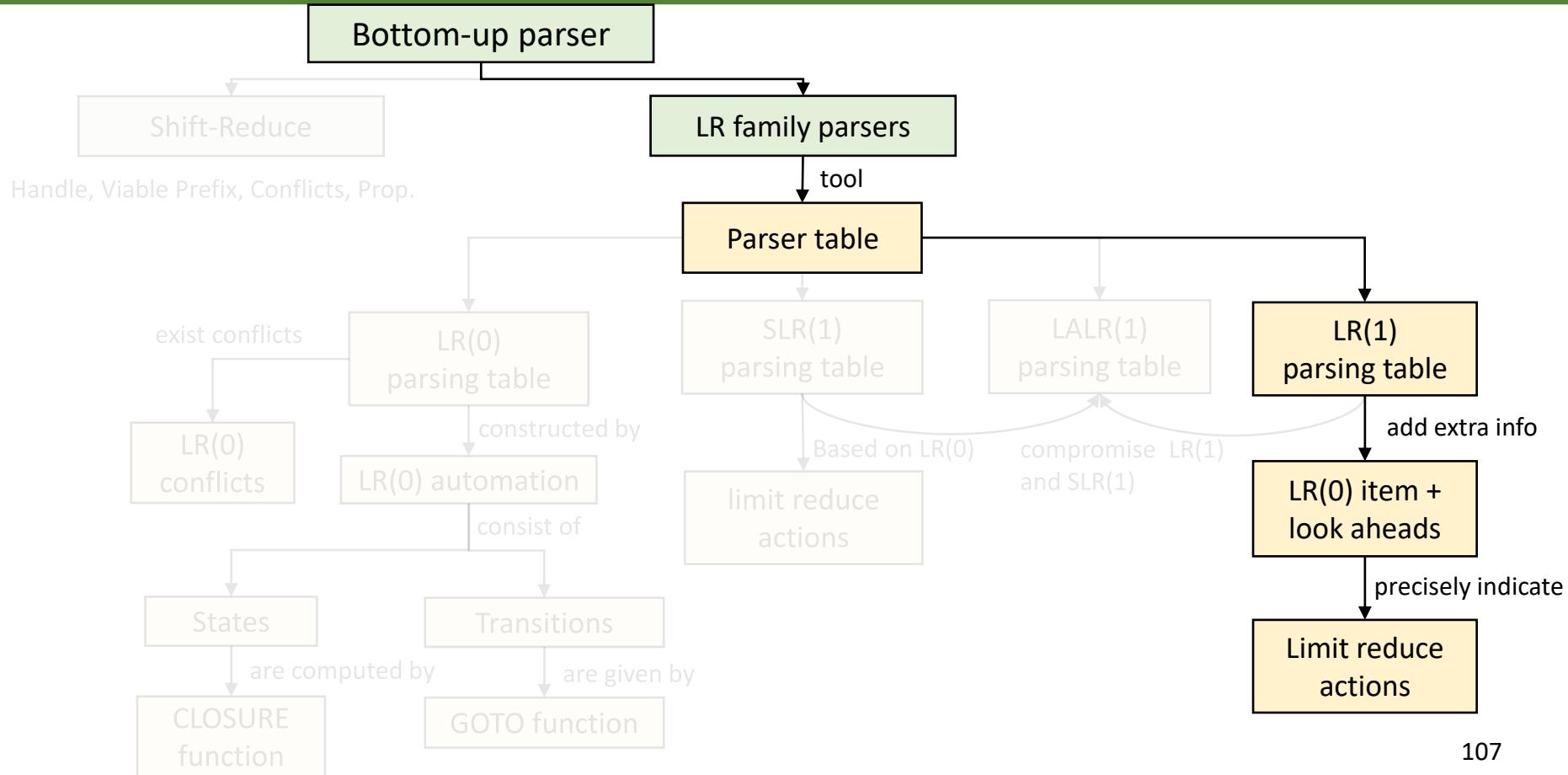


中山大學
SUN YAT-SEN UNIVERSITY

- Choices upon seeing $=$ coming up in the input:
 - ◆ Action[2, $=$] = s6 :
 - Move on to find the rest of assignment
 - ◆ Action[2, $=$] = r5 :
 - $= \in \text{Follow}(R)$
 - Leads to a wrong direction!
 - This is a shift-reduce conflict
 - ◆ Should only reduce when we have seen * or $=$, i.e., when the next input is $\$$.
 - ◆ SLR parser fails to remember enough info



Contents



Improving SLR(1)



- We don't need to see additional symbols beyond the first token in the input, we have already seen the info that allows us to determine the correct choice [lookahead展望信息已足够]
- Retain a little more of the **left context** that brought us here
 - ◆ Divide an SLR(1) state into separate states to differentiate the possible means by which that sequence has appeared on the stack [额外使用栈内stack信息, FOLLOW是input buffer的信息]
- Just using the Follow set is not discriminating enough as the guide for when to reduce [FOLLOW集不够]
 - ◆ For the example, the Follow set contains symbols that can follow R in any position within a valid sentence
 - ◆ But it does not precisely indicate which symbols follow R at this particular point in a derivation

LR(1) Parsing



- LR parsing adds the required extra info into the state
 - ◆ By redefining items to include a **terminal symbol** as an added component
[让项目中包含终结符]
- General form of **LR(1) items**: $A \rightarrow X_1 \dots X_i \bullet X_{i+1} \dots X_j, a$
 - ◆ We have states $X_1 \dots X_i$ on the stack
 - ◆ suppose $X_{i+1} \dots X_j$ are on the stack, we can perform reduction when:
 - the next input symbol is a
 - a is called the lookahead of the configuration
- The lookahead **only** works with **completed items**[完成项]
 - ◆ $A \rightarrow X_1 \dots X_j \bullet, a$
 - ◆ only reduce when next symbol is a (a is either a terminal or $\$$)
 - ◆ Multi lookahead symbols: $A \rightarrow u \bullet, a/b/c$



LR(1) Parsing(cont.)

- When to reduce?
 - ◆ LR(0): if the configuration set has a completed item (i.e., dot at the end)
 - ◆ SLR(1): only if the next input token is in the Follow set
 - ◆ LR(1): only if the next input token is exactly a (terminal or \$)
 - ◆ Trend: **more and more precise**
- **LR(1) items: LR(0) item + lookahead terminals**
 - ◆ In many cases, LR(1) differs only in their lookahead components
 - ◆ The extra lookahead terminals allow to make parsing decisions beyond the SLR(1) capability, but with a big price [代价]
 - More distinguished items and thus more sets
 - Greatly **increases Goto and Action table size**

$S' \rightarrow \cdot S$	$S' \rightarrow \cdot S, \$$
LR(0)	LR(1)

LR(1) Construction



• Configuration sets

- ◆ Sets construction are essentially the same with SLR, but differing on **Closure()** and **Goto()**
 - Because of the lookahead symbol

• Closure()

- ◆ Given each item $[A \rightarrow u \cdot Bv, a]$ in I , for each production rule $B \rightarrow w$ in G' , add $[B \rightarrow \cdot w, b]$ to I :

- $b \in \text{First}(va)$ && $[B \rightarrow \cdot w, b]$ is not in I
- v can be nullable

- ◆ Lookahead is the **First(va)**, which are the symbols that can follow B

- (0) $S' \rightarrow S$
- (1) $S \rightarrow XX$
- (2) $X \rightarrow aX$
- (3) $X \rightarrow b$

$S' \rightarrow \cdot S, \$$



$I_0:$
 $S' \rightarrow \cdot S, \$$
 $S \rightarrow \cdot XX, \text{First}(\epsilon \$)$
 $X \rightarrow \cdot aX, \text{First}(X \$)$
 $X \rightarrow \cdot b, \text{First}(X \$)$



$I_0:$
 $S' \rightarrow \cdot S, \$$
 $S \rightarrow \cdot XX, \$$
 $X \rightarrow \cdot aX, a/b$
 $X \rightarrow \cdot b, a/b$

LR(1) Construction (cont.)

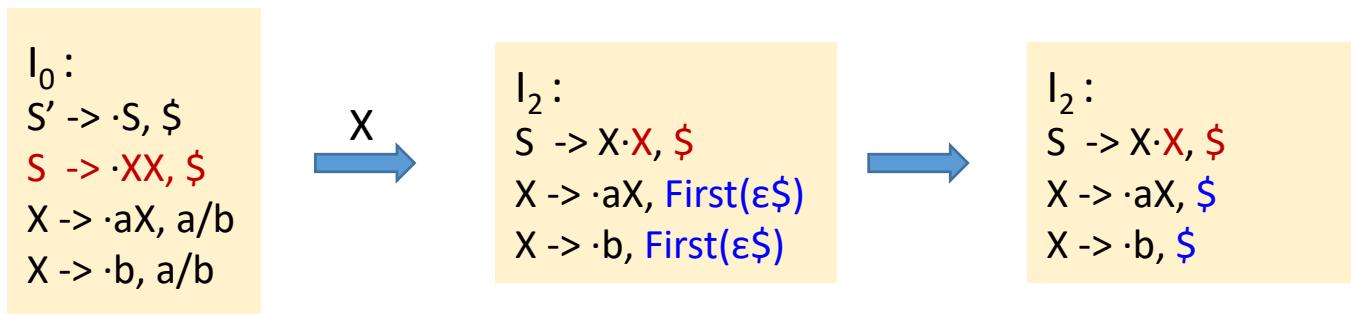


- **Goto(I, X)**

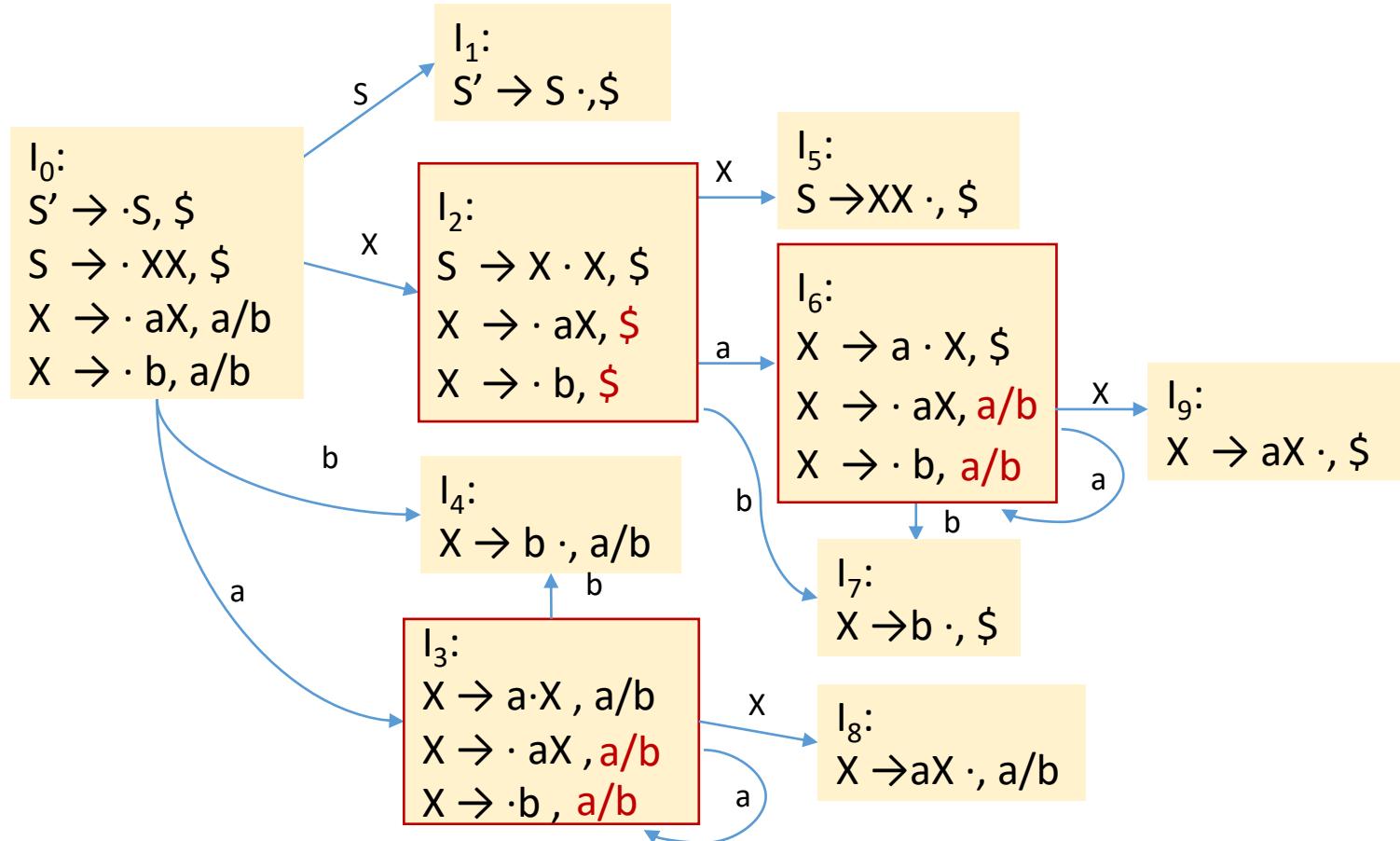
- ◆ For item $[A \rightarrow u \cdot X v, a]$ in I , $\text{Goto}(I, X) = \text{Closure}([A \rightarrow u X \cdot v, a])$
- ◆ Basically the same Goto function as defined for LR(0)
 - But have to **propagate the lookahead** [传递] when computing the transitions

- **Overall steps**

- ◆ Start from the initial set Closure ($[S' \rightarrow \cdot S, \$]$)
- ◆ Construct configuration sets following $\text{Goto}(I, X)$
- ◆ Repeat until no new sets can be added



Example



LR(1) Parse Table[解析表]



中山大學
SUN YAT-SEN UNIVERSITY

- Shift[移进]
 - ◆ Same as LR(0) and SLR(1)
 - ◆ Don't care the lookahead symbols
- Reduce[归约]
 - ◆ Don't use Follow set (that's SLR(1), too coarse-grain[粗粒度])
 - ◆ Reduce only if input matches lookahead for item
- ACTION and GOTO[表格]
 - ◆ If $[A \rightarrow \alpha \cdot a\beta, b] \in S_i$ and $\text{goto}(S_i, a) = S_j$, $\text{Action}[i, a] = S_j$
 - Shift a and goto state j
 - Same as SLR(1)
 - ◆ If $[A \rightarrow \alpha \cdot, a] \in S_i$ and next input is a , $\text{Action}[i, a] = r_k$
 - Reduce using k^{th} production ($A \rightarrow \alpha$) if input matches a
 - For SLR, reduced if the next input is in $\text{Follow}(A)$

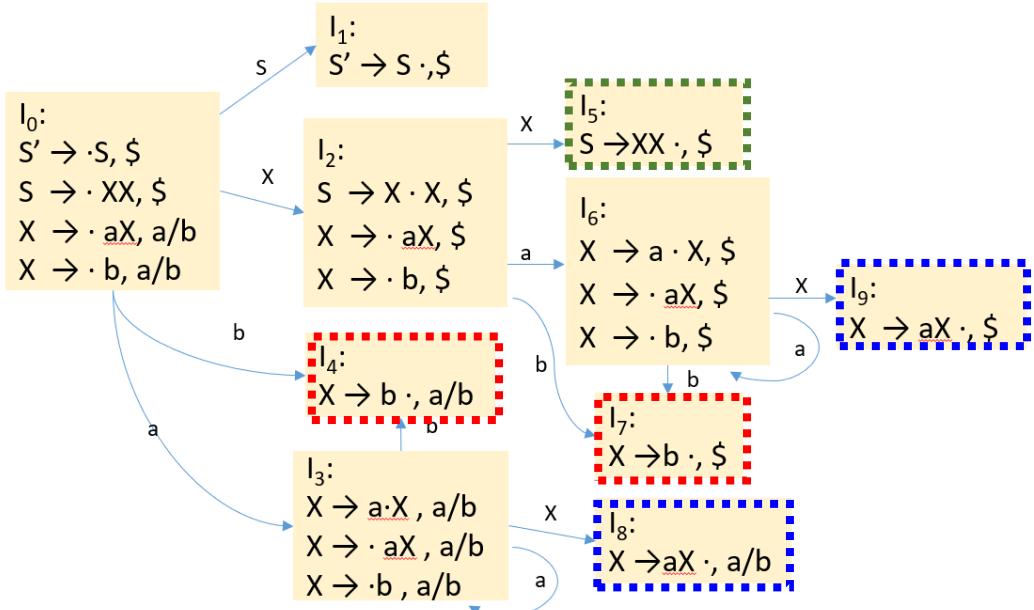
Example

(0) $S' \rightarrow S$

(1) $S \rightarrow XX$

(2) $X \rightarrow aX$

(3) $X \rightarrow b$

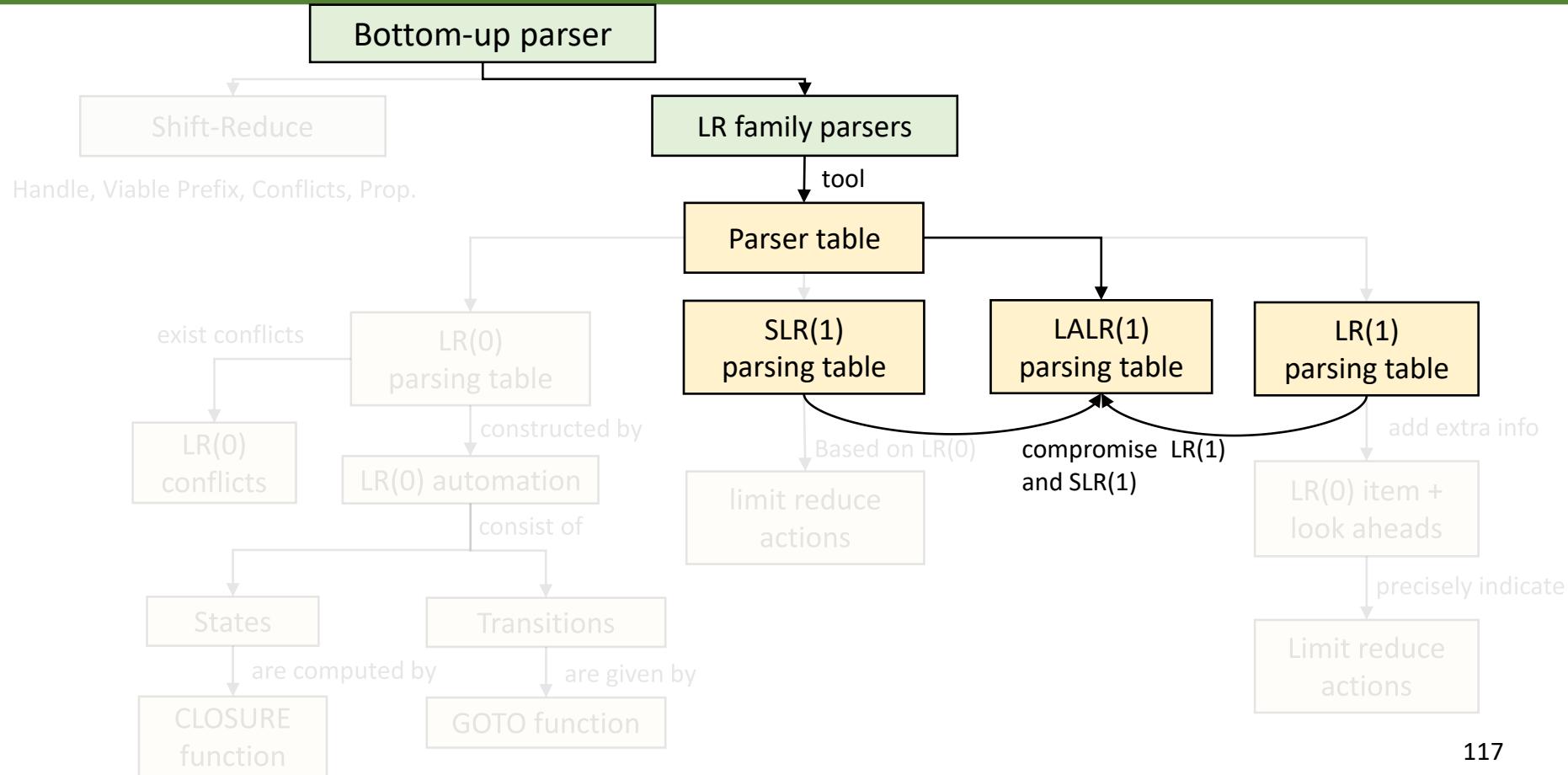


Stat	Action			Goto	
	a	b	\$	S	X
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

LR(1) Grammars

- Every SLR(1) grammar is LR(1), but the LR(1) parser may have more states than SLR(1) parser
 - ◆ LR(1) parser splits states based on differing lookaheads, thus it may avoid conflicts that would otherwise result in using the full Follow set
- A grammar is LR(1) if the following two conditions hold for each configurating set
 - ◆ (1) For any item $[A \rightarrow u \cdot xv, a]$ in the set, with terminal x , there is no item in the set of form $[B \rightarrow v \cdot, x]$
 - In the table, this translates no shift-reduce conflict on any state
 - ◆ (2) The lookaheads for all complete items within the set must be disjoint, e.g. set cannot have both $[A \rightarrow u \cdot, a]$ and $[B \rightarrow v \cdot, a]$
 - This translates to no reduce-reduce conflict on any state

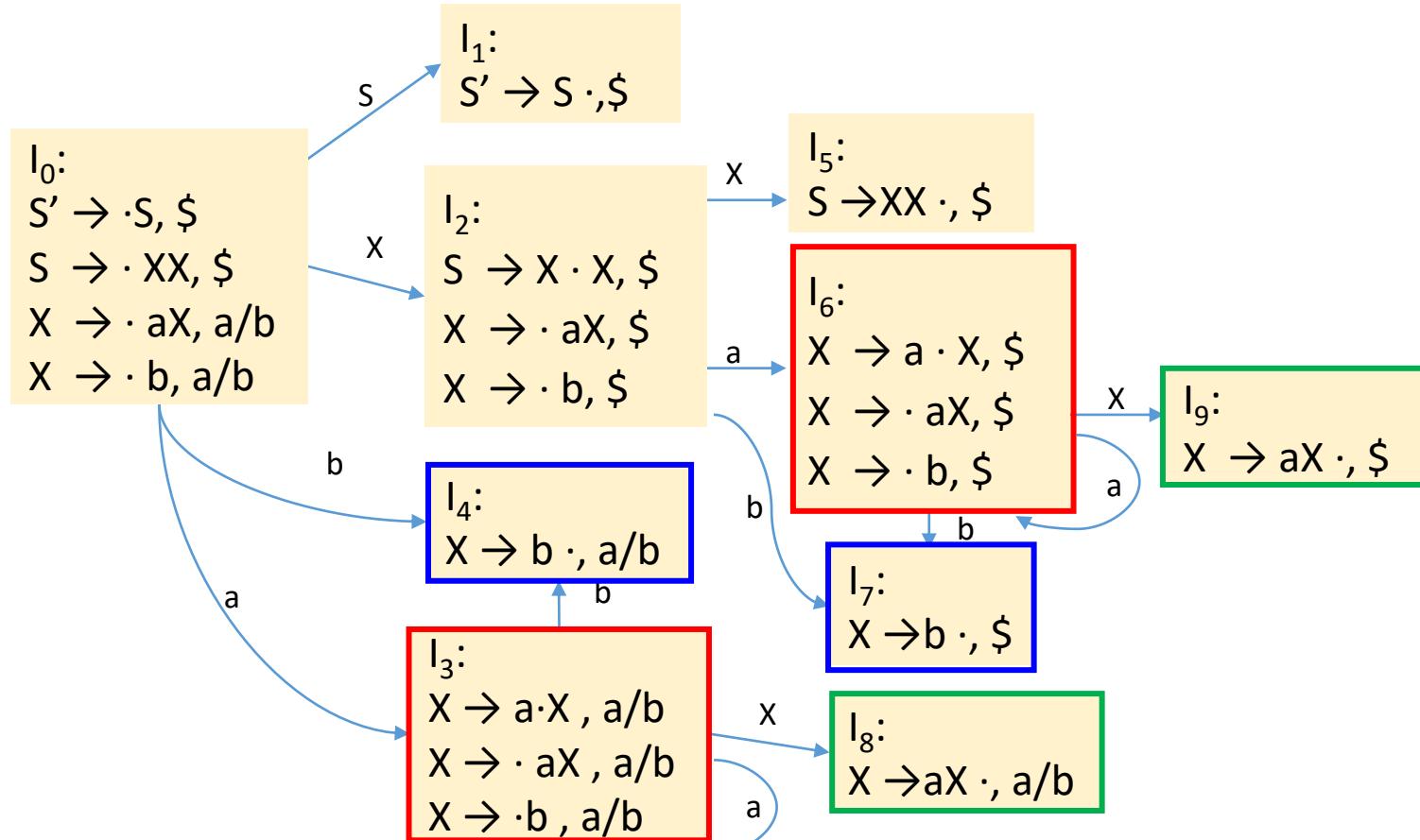
Contents



LALR(1) Parser

- What's the **drawbacks of LR(1)?**
 - ◆ With state splitting, the LR(1) parser can have many **more states** than SLR(1) or LR(0) parser
 - One **LR(0)** item may split up to many LR(1) items
 - As many as all possible lookaheads
 - In theory can lead to an exponential increase in #states
- **LALR (lookahead LR) – compromise LR(1) and SLR(1)[折衷]**
 - ◆ Reduce the number of states in LR(1) parser by **merging similar states**
 - Reduces the #states to the same as SLR(1), but still retains the power of LR(1) lookahead
 - ◆ **Similar states:** have **same number of items**, the **core of each item is identical**, and they differ only in their lookahead sets

Example



State Merging[状态合并]

- Merge states with the same core
 - Core: LR(1) items minus the lookahead (i.e., LR(0) items)
 - All items are identical except lookahead

$I_3:$
 $X \rightarrow a \cdot X, a/b$
 $X \rightarrow .aX, a/b$
 $X \rightarrow \cdot b, a/b$

$I_6:$
 $X \rightarrow a \cdot X, \$$
 $X \rightarrow .aX, \$$
 $X \rightarrow \cdot b, \$$



$I_{36}:$
 $X \rightarrow a \cdot X, a/b/\$$
 $X \rightarrow .aX, a/b/\$$
 $X \rightarrow \cdot b, a/b/\$$

$I_4:$
 $X \rightarrow b \cdot, a/b$

$I_7:$
 $X \rightarrow b \cdot, \$$



$I_{47}:$
 $X \rightarrow b \cdot, a/b/\$$

$I_8:$
 $X \rightarrow aX \cdot, a/b$

$I_9:$
 $X \rightarrow aX \cdot, \$$



$I_{89}:$
 $X \rightarrow aX \cdot, a/b/\$$

State Merging (cont.)

Stat e	Action			Goto	
	a	b	\$	s	x
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

LR(1)

Stat e	Action			Goto	
	a	b	\$	s	x
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Grammar

- (0) $S' \rightarrow S$
- (1) $S \rightarrow XX$
- (2) $X \rightarrow aX$
- (3) $X \rightarrow b$

LALR(1)

States 3 and 6: State 36
 States 4 and 7: State 47
 States 8 and 9: State 89

Merge Effects[合并效果]



1. Merging of states can introduce **conflicts**[引入归约-归约冲突]
 - ◆ Can introduce **reduce-reduce** (r-r) conflicts
 - LR was introduced to split the Follow set on reduce action
 - Merging reverts the splitting
 - ◆ **Cannot** introduce **shift-reduce** (s-r) conflicts
 - i.e., a s-r conflict cannot exist in a merged set unless the conflict was already present in one of the original LR(1) sets
2. Detection of errors may be delayed[推迟错误识别]
 - ◆ On error, LALR parsers will not perform shifts beyond an LR parser, but may perform **more reductions before finding error**

Merge Conflict: Shift-Reduce



- Shift-reduce conflicts are **not** introduced by merging
- Suppose:

S_{ij} contains: $[A \rightarrow \alpha \cdot, a]$ reduce on input a

$[B \rightarrow \beta \cdot a \gamma, b]$ shift on input a

Formed by merging S_i and S_j [注: S_{ij} 并不一定只有这两个items]

- Then:
 - ◆ Cores must be the same for S_i and S_j , and thus one of them must contain $[A \rightarrow \alpha \cdot, a]$
 - ◆ and it must have an item $[B \rightarrow \beta \cdot a \gamma, c]$ for some random c
 - This state has the same shift/reduce conflict on a , i.e., the grammar was not LR(1)
 - ◆ Shift-reduce conflicts were already present in either S_i and S_j (or both) and not newly introduced by merging

Merge Conflict: Reduce-Reduce

- Reduce-reduce conflicts can be introduced by merging
- In this case, we say the grammar is not LALR(1)

$S' \rightarrow S$

$S \rightarrow aBc \mid bCc \mid aCd \mid bBd$

$B \rightarrow e$

$C \rightarrow e$

I₀: $S' \rightarrow \bullet S, \$$
 $S \rightarrow \bullet aBc, \$$
 $S \rightarrow \bullet bCc, \$$
 $S \rightarrow \bullet aCd, \$$
 $S \rightarrow \bullet bBd, \$$

I₁: $S' \rightarrow S\bullet, \$$

I₂: $S \rightarrow a\bullet Bc, \$$
 $S \rightarrow a\bullet Cd, \$$
 $B \rightarrow \bullet e, c$
 $C \rightarrow \bullet e, d$

I₆₉:

$C \rightarrow e\cdot, c/d$
 $B \rightarrow e\cdot, d/c$

I₃: $S \rightarrow b\bullet Cc, \$$
 $S \rightarrow b\bullet Bd, \$$
 $C \rightarrow \bullet e, c$
 $B \rightarrow \bullet e, d$

I₄: $S \rightarrow aB\bullet c, \$$

I₅: $S \rightarrow aC\bullet d, \$$

I₆: $B \rightarrow e\bullet, c$
 $C \rightarrow e\bullet, d$

I₇: $S \rightarrow bC\bullet c, \$$

Reduce to **B or C** when next token is c or d

I₈: $S \rightarrow bB\bullet d, \$$

I₉: $B \rightarrow e\bullet, d$
 $C \rightarrow e\bullet, c$

I₁₀: $S \rightarrow aBc\bullet, \$$

I₁₁: $S \rightarrow aCd\bullet, \$$

I₁₂: $S \rightarrow bCc\bullet, \$$

I₁₃: $S \rightarrow bBd\bullet, \$$

1. Merging of states can introduce **conflicts**[引入归约-归约冲突]
 - ◆ Can introduce reduce-reduce (r-r) conflicts
 - LR was introduced to split the Follow set on reduce action
 - Merging reverts the splitting
 - ◆ Cannot introduce shift-reduce (s-r) conflicts
 - i.e., a s-r conflict cannot exist in a merged set unless the conflict was already present in one of the original LR(1) sets
2. Detection of errors may be delayed[推迟错误识别]
 - ◆ On error, LALR parsers will not perform shifts beyond an LR parser, but may perform more reductions before finding error

Example: Error Delay



State	Action			Goto	
	a	b	\$	S	X
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Grammar
(0) $S' \rightarrow S$
(1) $S \rightarrow XX$
(2) $X \rightarrow aX$
(3) $X \rightarrow b$

Input
aab\$

State
Symbol

State
Symbol

State
Symbol

State
Symbol

aab\$ 33

abs s3

54

 \$

Example: Error Delay(cont.)



Grammar

(0) $S' \rightarrow S$ (1) $S \rightarrow XX$ (2) $X \rightarrow aX$ (3) $X \rightarrow b$

Input aab\$

State	Action			Goto	
	a	b	\$	S	X
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

State	S_0			
Symbol	\$			aab\$
State	$S_0 S_{36}$			
Symbol	\$ a			ab\$
State	$S_0 S_{36} S_{36}$			
Symbol	\$ a a			b\$
State	$S_0 S_{36} S_{36} S_{47}$			
Symbol	\$ a a b			\$
State	$S_0 S_{36} S_{36} S_{89}$			
Symbol	\$ a a X			\$
State	$S_0 S_{36} S_{89}$			
Symbol	\$ a X			\$
State	$S_0 S_2$			
Symbol	\$ X			\$

127

LALR Table Construction [解析表构建]



- LALR(1) parsing table is built from the configuration sets in the same way as LR(1) [同样方法构建的项目集]
 - ◆ The **lookaheads** determine where to place reduce actions
 - ◆ If there are no mergable states, the LALR(1) table will be identical to the LR(1) table and we gain nothing [退化为LR(1)]
 - ◆ Usually, there will be states that can be merged and the LALR table will thus have **fewer rows** than LR(1)
- LALR(1) table have the **same #states (rows)** with **SLR(1)** and **LR(0)**, but have **fewer reduce actions** [同等数目的状态,但更少的归约动作]
 - ◆ Some reductions are not valid if we are more precise about the lookahead
 - ◆ Some conflicts in SLR(1) and LR(0) are avoided by LALR(1)
 - ◆ For C language: SLR/LALR - 100s states, LR - 1000s states

LALR Table Construction (Cont.)

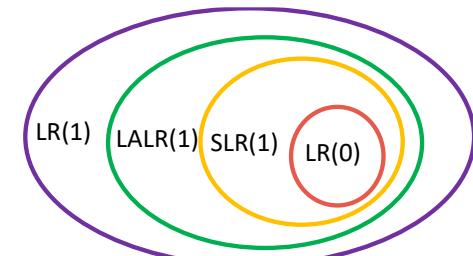


- Brute force[暴力方法]
 - ◆ Construct LR(1) states, then merge states with same core
 - ◆ If no conflicts, you have a LALR parser
 - ◆ Inefficient: building LR(1) items are expensive in time and space
 - We need a better solution
- Efficient way[高效方式]
 - ◆ Avoid initial construction of LR(1) states
 - ◆ Merge states on-the-fly (step-by-step merging)
 - States are created as in LR(1)
 - On state creation, immediately merge if there is an opportunity

LALR(1) Grammars

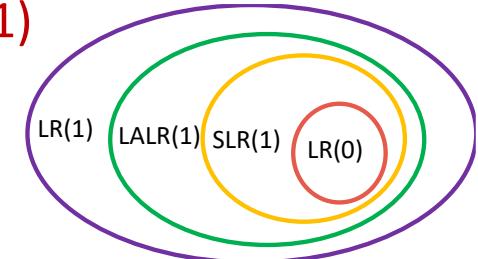


- For a grammar, if the LALR(1) parse table has no conflicts, then we say the grammar is LALR(1)
 - ◆ No formal definition of a set of rules
- LALR(1) is a **subset of LR(1)** and a **superset of SLR(1)**
 - ◆ A SLR(1) grammar is definitely LALR(1)
 - ◆ A LR(1) grammar may or may not be LALR(1)
 - Depends on whether merging introduces conflicts
 - ◆ A non-SLR(1) grammar may be LALR(1)
 - Depends on whether the more precise lookaheads resolve the SLR(1) conflicts
 - ◆ Most used variant of the LR family



LALR Summary

- LALR(1) provides a good **balancing** between LR(1) and SLR(1)
 - ◆ Range of grammars supported: $\text{LR} > \text{LALR} > \text{SLR}$
 - ◆ Number of states in the table: $\text{LR} > \text{LALR} = \text{SLR}$
- If a grammar **G** is LR but **is not** SLR
 - ◆ cannot simply rely on the Follow set to resolve conflicts
 - Need more precise decision on reduction -- **lookahead**
 - ◆ LR grammars have a large parsing table with many states, we can **merge** some of the states, i.e., LALR
 - No Confliction after merge --> G is LALR
 - The confliction in SLR is resolved.
 - LALR have the same number of states with SLR, but less reduction actions
 - If no states can be merged, then $\text{LALR}=\text{LR}$ for G
- If a grammar G is SLR, G is also LR and LALR. Therefore, we don't need to split the states using specific lookahead, that is, $\text{LALR}=\text{SLR}$



LL vs. LR Parsing (LL < LR)

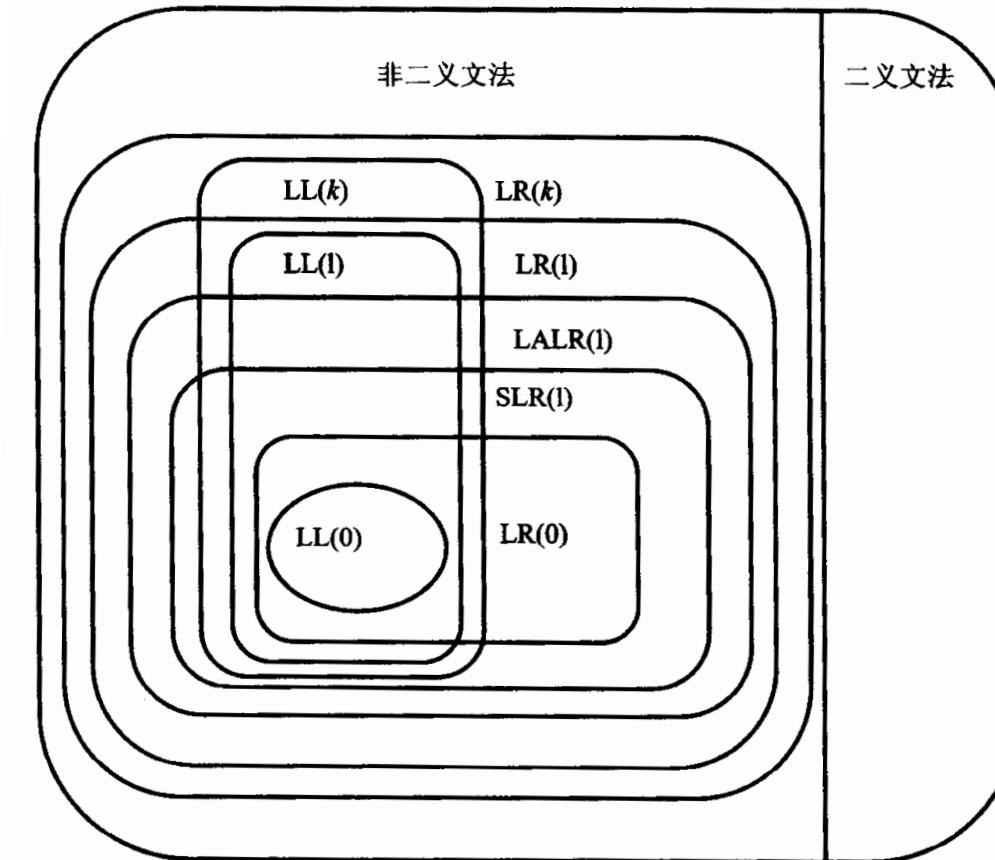


- LL(k) parser, each expansion $A \rightarrow \alpha$ is decided based on
 - ◆ Current non-terminal at the top of the stack[依赖LHS]
 - Which LHS to produce
 - ◆ k terminals of lookahead at **beginning** of RHS[展望RHS]
 - Must **guess** which RHS by peeking at **first few terminals** of RHS
- LR(k) parser,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

Hierarchy of Grammars[文法层级]

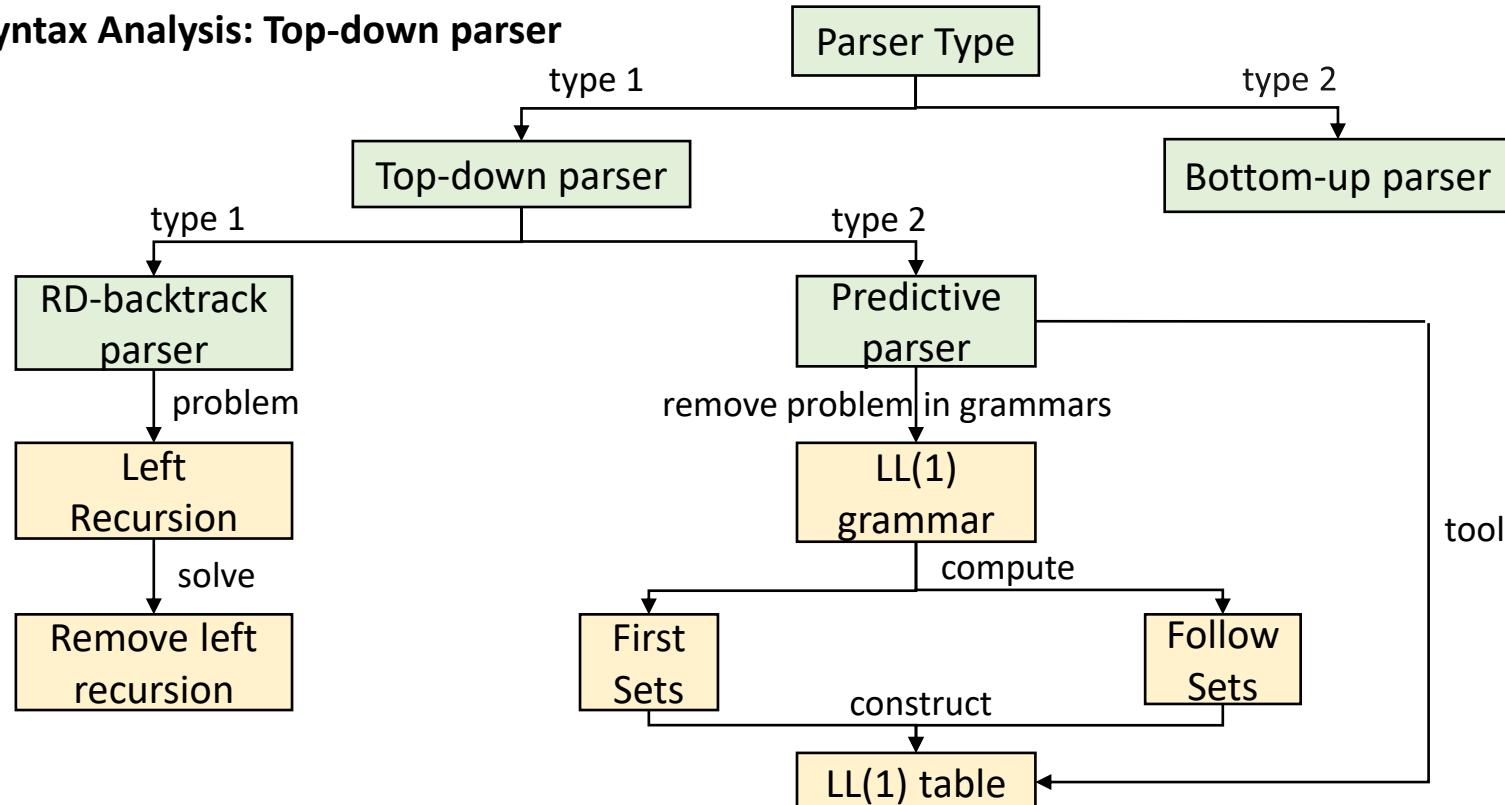


中山大學
SUN YAT-SEN UNIVERSITY

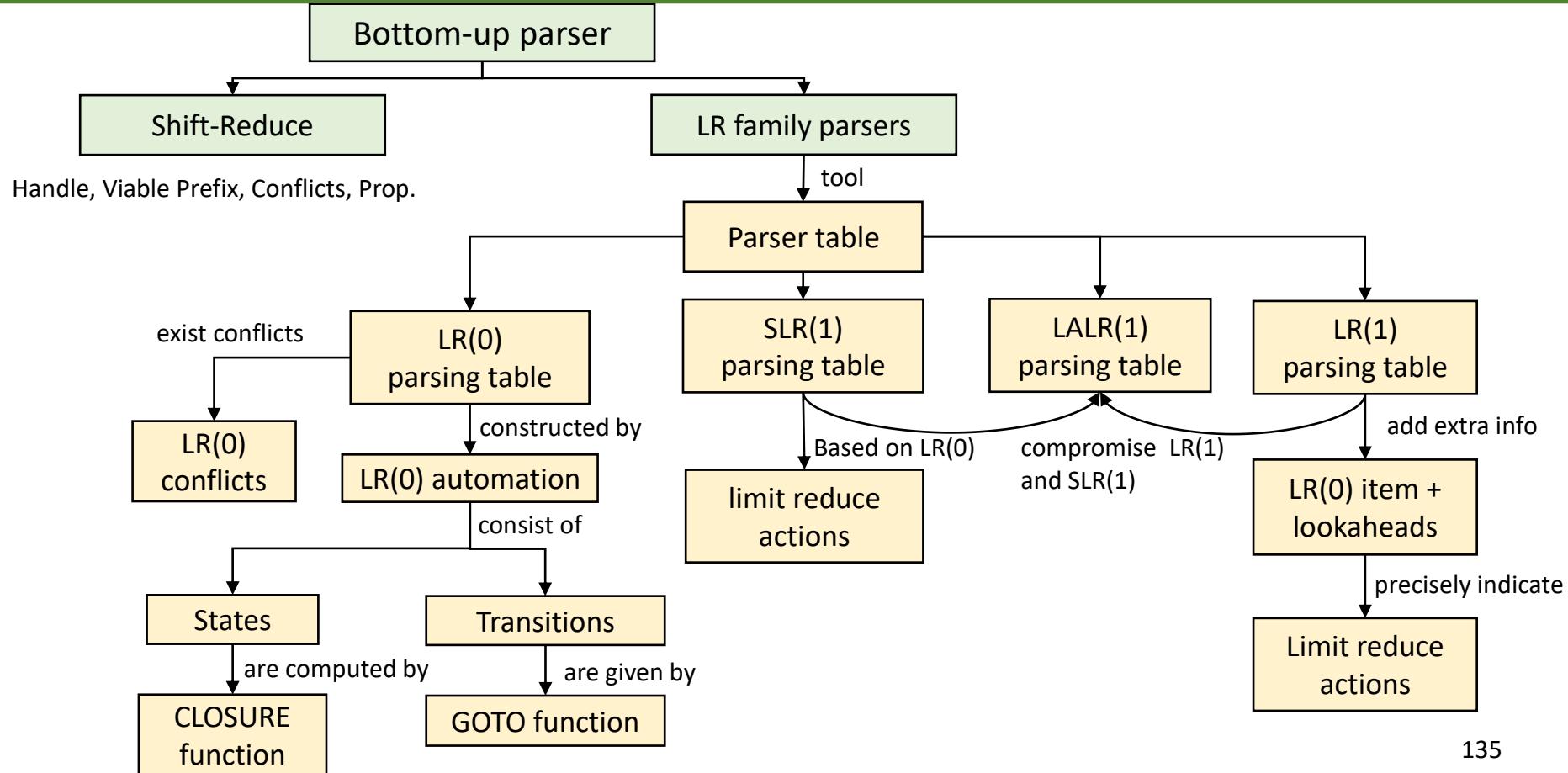


Mind Map[思维导图]

Syntax Analysis: Top-down parser



Summary



Summary of Syntax analysis



中山大學
SUN YAT-SEN UNIVERSITY

- Syntax analysis – the second phase of compilation
 - ◆ Input: a sequence of token provided by the Lexical Analysis
 - ◆ Output: a parse tree or abstract syntax tree (AST)
- Syntax specification
 - RE/FA is not powerful enough (e.g., $a^n b^n$, where $n > 0$)
 - So Grammar is needed, especially for context-free grammar (CFG)
- Grammar G: $G = (V_T, V_N, S, \delta)$
 - ◆ V_T : terminal symbols[终结符] = tokens from Lexical Analysis, leaves of the parse tree
 - ◆ V_N : non-terminal symbols[非终结符], internal nodes of the parse tree
 - ◆ S : start symbol[开始符号]
 - ◆ δ : set of productions[产生式]: LHS -> RHS

Summary of Syntax analysis



- Derivation
 - ◆ The application of the productions (from LHS to RHS)
 - starts from the start symbol S to the input string
- Reduce
 - ◆ The reverse of derivation (from RHS to LHS)
 - starts from the input string to the start symbol S
- Parse tree
 - ◆ Graphical representation of Derivation/Reduce without the order of productions
- Ambiguous grammar
 - ◆ A sentence has more than one parse (leftmost or rightmost) trees
 - ◆ Ambiguity can be resolved by defining precedence and associativity for the grammar

Summary of Syntax analysis

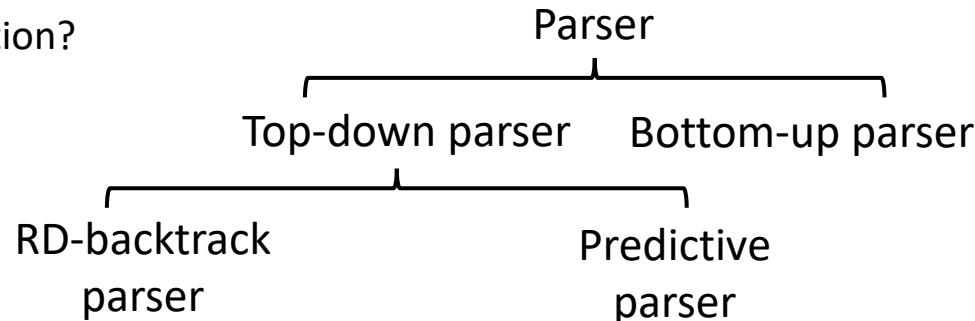
- Syntax analysis takes a sentence as the input, and outputs the parse tree or AST of the sentence

◆ **Top-down**: from the root to the leaves, in each step:

- Which non-terminal to replace?
- Which production?

◆ **Bottom-up**: from leaves to the root

- Shift or reduce?
- If reduce, then which production?





Summary of Syntax analysis

- Top-Down Parser [自顶向下分析]
 - RDP [递归下降分析]
 - **left recursion** [左递归] -- Remove Left Recursion [消除直接/间接左递归]
 - **Backtracking** [回溯] -- **left factoring** [提取左公因子]
 - Predictive Parsing [预测分析]
 - ◆ Four components: input buffer, **stack**, **parse table**, parser driver
 - ◆ Action (**expand** or **match**) based on <stack top, current token>
 - ◆ Definition of LL(1)/LL(k) [$\text{LL}(1)$ / $\text{LL}(k)$ 文法的定义]
 - ◆ **FIRST & FOLLOW** [终结首符集 & 后继终结符号集]
 - ◆ **LL(1) Parse Table** [$\text{LL}(1)$ 的分析表]
 - The use of the Parse Table [分析表的使用]
 - The Construction of the Parse Table [分析表的构建]
 - ◆ Table-driven LL(1) Parser Implementation [分析表驱动的 $\text{LL}(1)$ 语法分析实现]
 - ◆ Complexity of LL(1) [$\text{LL}(1)$ 的时间与空间复杂度]

Summary of Syntax analysis

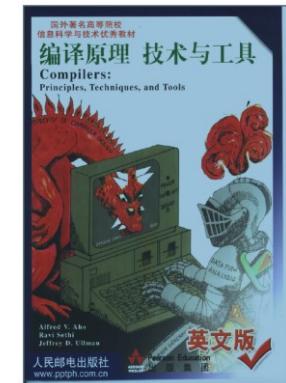


中山大學
SUN YAT-SEN UNIVERSITY

- Bottom-up[自顶向下分析]
 - ◆ Mainly two actions: **Shift** and **Reduce**
 - ◆ LR Parser
 - Table driven, efficient
- Table-driven LR parsers
 - ◆ For components: input buffer, **stack**, **parse table**, parser driver
 - ◆ Action (shift or reduce) is performed based on the top of the stack
 - Symbols and their associated **states** on the stack
 - ◆ Parser Tables contains **ACTION** and **GOTO** tables
 - Table is constructed by identifying the states and their transitions (i.e., **DFA**)
 - LR(0) -> SLR(1) -> LALR(1) -> LR(1)

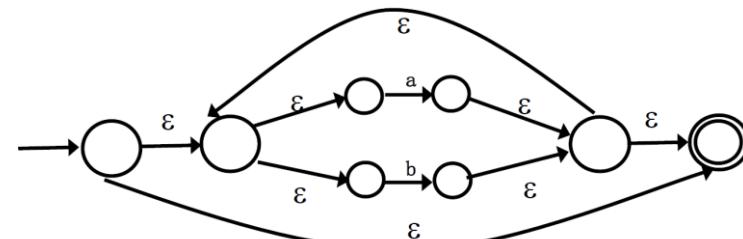
Further Reading

- Dragon Book, 2nd Edition (DBv2)
 - ◆ Comprehensive Reading:
 - Section 4.5- 4.7 for the Bottom-Up parsing.
 - Section 4.8.1-4.8.2 on ambiguities in LR parsing.
 - ◆ Skip Reading:
 - Section 4.8.3 on error recovery in LR parsing.
- Dragon Book, 1st Edition (DBv1)
 - ◆ Comprehensive Reading:
 - Section 4.6 for the OPP details.



Exercise 2

- 正则表达式可以表达语言 $L = \{xnyzn, 1 \leq n \leq 10\}$ 吗?
 - 可以, 任何有穷集都是正规集, 都可以用正规表达式来描述
- 正则表达式可以表达语言 $L = \{x^n y z^n, n > 0\}$ 吗?
 - 不行, 该语言是无穷的, 不能用正规式表达
- 确定有限自动机(DFA)中不存在 ϵ 弧, 所以DFA无法识别空串 ϵ 吗?
 - 错, DFA可以识别空串, 但当且仅当开始状态与终止状态
- 给定正则式 $(a \mid b)^*$, 使用汤普森构造法 (Thompson's Construction) 构造的有限自动机拥有几个状态?
 - 8个



Exercise 2

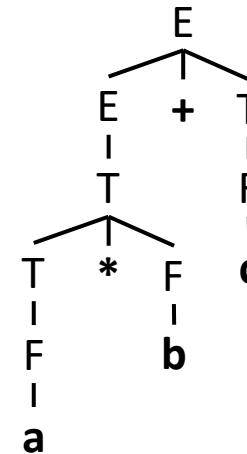
- 对于一个无二义性文法G, 它的一个句子的最左与最右推导产生的分析树一定是一样的吗?
 - 对的, 是这样的。语法树会隐去推导顺序
- 怎么样判断一个文法是否为LL(1)型文法?
 - 对文法中的每个产生式 $A \rightarrow \alpha \mid \beta$:
 - $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$.
 - If $\beta \Rightarrow \epsilon$, then $\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \emptyset$.
- LL(1)语法分析表行数与列数分别由哪些因素确定?
 - 行数是该文法的非终结符个数, 列数为终结符个数+1 (结束符号\$)
- 设句型aaAb 的最右推导为 $S \Rightarrow aBb \Rightarrow aaAb$, 句柄为aA, 该句型的活前缀是?
 - a, aa, aaA

Exercise 2

- 找出以下文法中的句型的所有短语，直接短语，句柄

- 文法: $G(E) : E \rightarrow T \mid E+T; \quad T \rightarrow F \mid T^*F; \quad F \rightarrow (E) \mid a \mid b \mid c$
- 句型: $a * b + c$

- 第一步：画出语法树
- 第二步：找出语法树的叶子节点
 - $a, *, b, +, c$
- 第三步：由叶子节点找出短语
 - a, a^*b, b, c, a^*b+c (why not $*b, b+c, *, +?$)
- 第四步：从短语中找出直接短语
 - a, b, c
- 第五步：从直接短语中找出句柄
 - a



Exercise 2

- 给定一下文法 $G(S)$, 请构建该文法的**LL(1)**分析表

- $S \rightarrow A$
- $A \rightarrow \epsilon$
- $A \rightarrow bbA$

- FIRST & Follow:**

- $S \rightarrow A$: FIRST(α) = {b, ϵ }
- $A \rightarrow \epsilon$: FIRST(α) = { ϵ }
- $A \rightarrow bbA$: FIRST(α) = {b}
- $\text{FOLLOW}(S) = \{\$\}$
- $\text{FOLLOW}(A) = \{\$\}$

- Parse Table:**

	b	\$
S		
A		
	b	\$
S	$S \rightarrow A$	
A	$A \rightarrow bbA$	
	b	\$
S	$S \rightarrow A$	$S \rightarrow A$
A	$A \rightarrow bbA$	$A \rightarrow \epsilon$

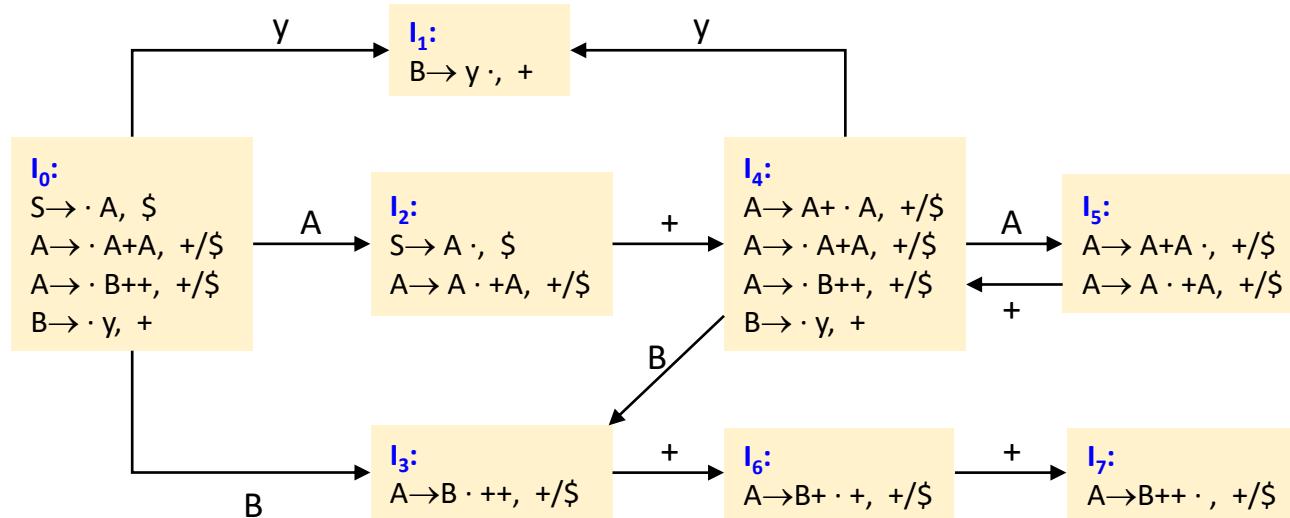
use FIRST

use FOLLOW

Exercise 2

• 给定一下文法 $G(S)$, 请构建识别活前缀的DFA与LR(1)分析表

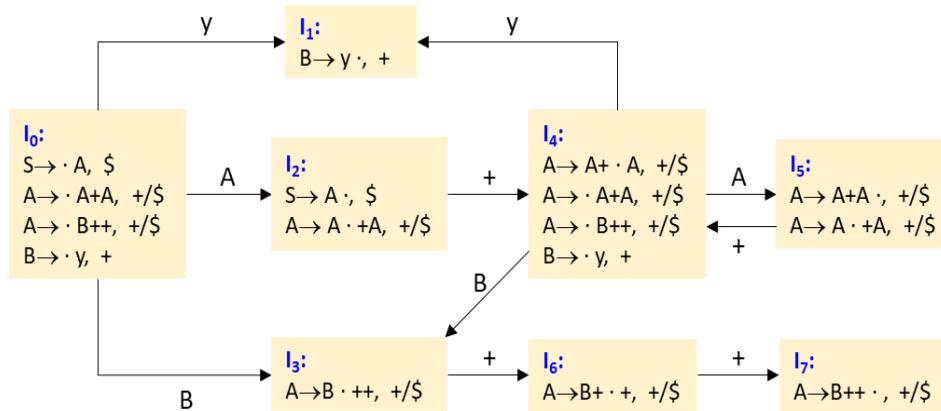
- $S \rightarrow A$
- $A \rightarrow A + A \mid B ++$
- $B \rightarrow y$



Exercise 2

- 给定一下文法 $G(S)$, 请构建识别活前缀的DFA与LR(1)分析表

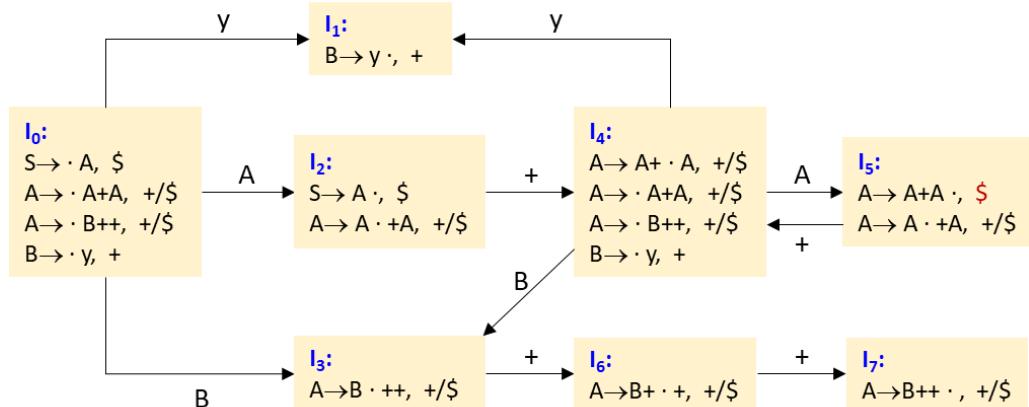
- (1) $S \rightarrow A$
- (2) $A \rightarrow A + A$
- (3) $A \rightarrow B ++$
- (4) $B \rightarrow y$



S	ACTION			GOTO	
	y	+	\$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		r2/s4	r2		
6		s7			
7		r3	r3		

Exercise 2

- 该文法是LR(1)型文法吗？若不是，应如何解决？(有多种选择)



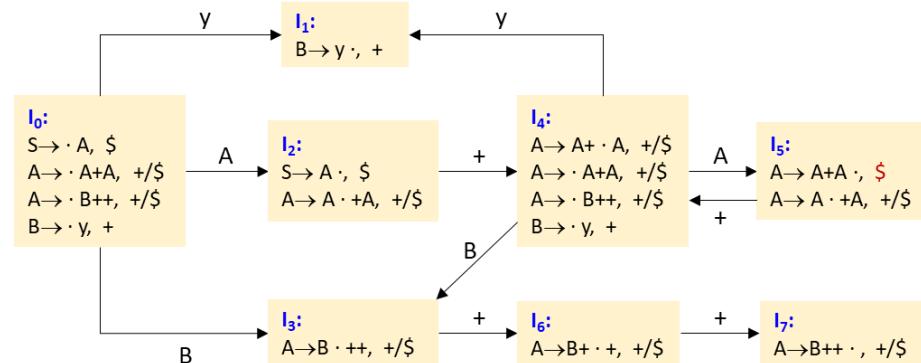
S	ACTION			GOTO	
	y	+	\$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		s4	r2		
6		s7			
7		r3	r3		

Exercise 2

- 对于句子 $y++$, 请给出其分析流程, 并说明 $y++$ 是否为该文法的一个句子?

(1) $S \rightarrow A$; (2) $A \rightarrow A + A$; (3) $A \rightarrow B ++$; (4) $B \rightarrow y$;

Stack	Input	ACTION	GOTO
0 \$	$y++$$		



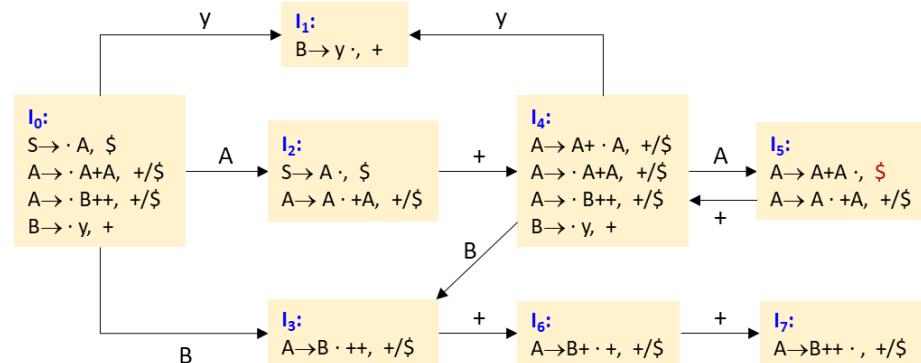
S	ACTION			GOTO	
	y	+	\$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		s4	r2		
6		s7			
7		r3	r3		

Exercise 2

- 对于句子 $y++$, 请给出其分析流程, 并说明 $y++$ 是否为该文法的一个句子?

- (1) $S \rightarrow A$; (2) $A \rightarrow A + A$; (3) $A \rightarrow B ++$; (4) $B \rightarrow y$;

Stack	Input	ACTION	GOTO
<u>0</u>	\$	$y++\$$	s1



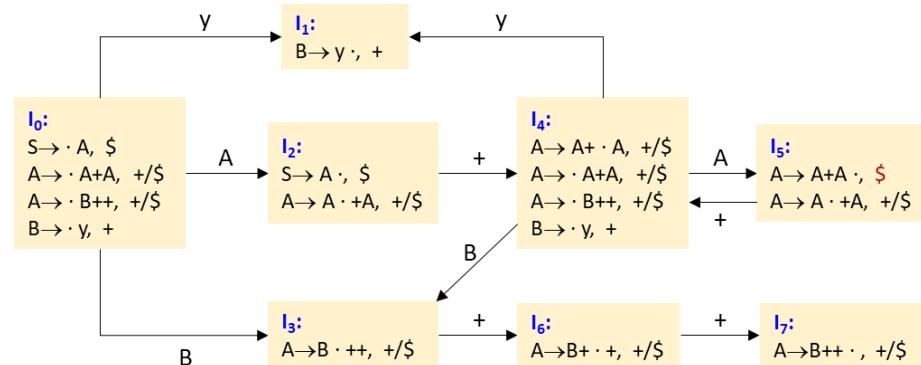
S	ACTION			GOTO	
	y	+	\$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		s4	r2		
6		s7			
7		r3	r3		

Exercise 2

- 对于句子 $y++$, 请给出其分析流程, 并说明 $y++$ 是否为该文法的一个句子?

(1) $S \rightarrow A$; (2) $A \rightarrow A + A$; (3) $A \rightarrow B ++$; (4) $B \rightarrow y$;

Stack	Input	ACTION	GOTO
<u>0</u>	\$	$y+\$$	s1
01	\$y	$\underline{+}+\$$	



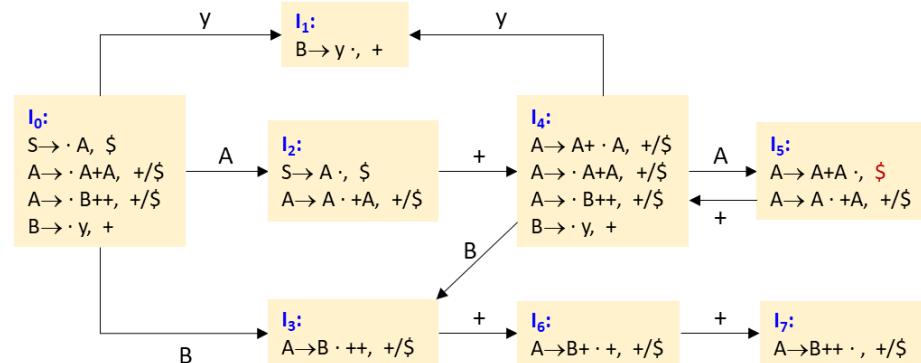
S	ACTION			GOTO	
	y	+	\$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		s4	r2		
6		s7			
7		r3	r3		

Exercise 2

- 对于句子 $y++$, 请给出其分析流程, 并说明 $y++$ 是否为该文法的一个句子?

(1) $S \rightarrow A$; (2) $A \rightarrow A + A$; (3) $A \rightarrow B ++$; (4) $B \rightarrow y$;

Stack	Input	ACTION	GOTO
<u>0</u>	\$	$y++\$$	s1
01	\$y	$\underline{++\$}$	r4($B \rightarrow y$)



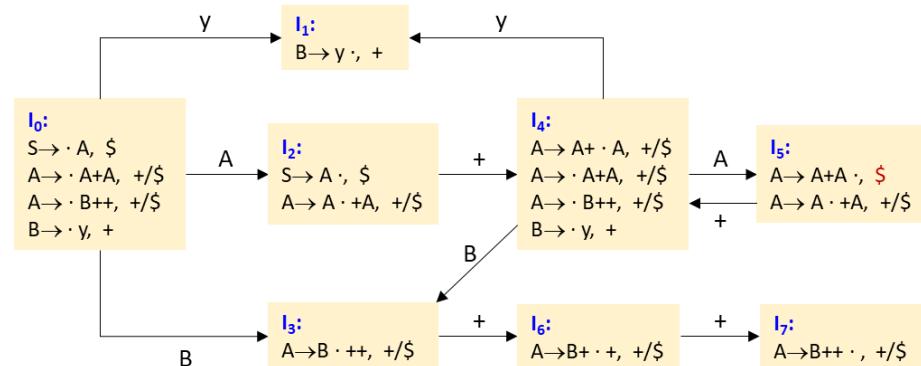
S	ACTION			GOTO	
	y	+	\$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		s4	r2		
6		s7			
7		r3	r3		

Exercise 2

- 对于句子 $y++$, 请给出其分析流程, 并说明 $y++$ 是否为该文法的一个句子?

(1) $S \rightarrow A$; (2) $A \rightarrow A + A$; (3) $A \rightarrow B ++$; (4) $B \rightarrow y$;

Stack	Input	ACTION	GOTO
<u>0</u>	\$	$y++\$$	s1
<u>01</u>	$\$y$	$\underline{++\$}$	r4($B \rightarrow y$)
0	$\$B$	$\underline{++\$}$	3



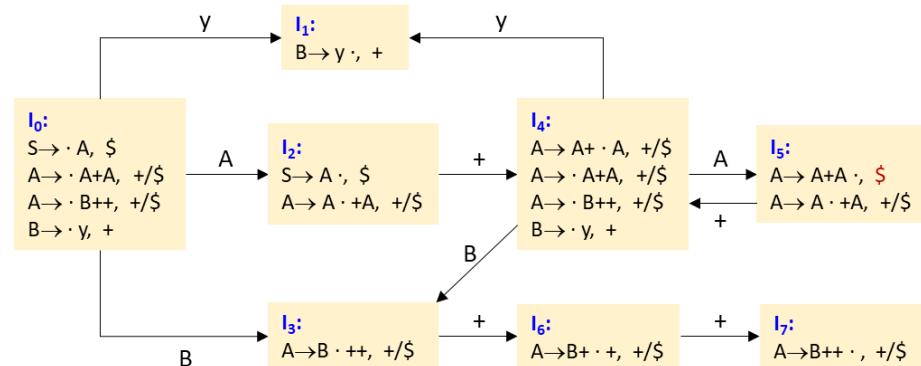
S	ACTION			GOTO	
	y	+	\$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		s4	r2		
6		s7			
7		r3	r3		

Exercise 2

- 对于句子 $y++$, 请给出其分析流程, 并说明 $y++$ 是否为该文法的一个句子?

(1) $S \rightarrow A$; (2) $A \rightarrow A + A$; (3) $A \rightarrow B ++$; (4) $B \rightarrow y$;

Stack	Input	ACTION	GOTO
<u>0</u>	\$	<u>y++\$</u>	s1
<u>01</u>	\$y	<u>++\$</u>	r4($B \rightarrow y$)
0	\$B	<u>+\$</u>	3
<u>03</u>	\$B	<u>+\$</u>	s6



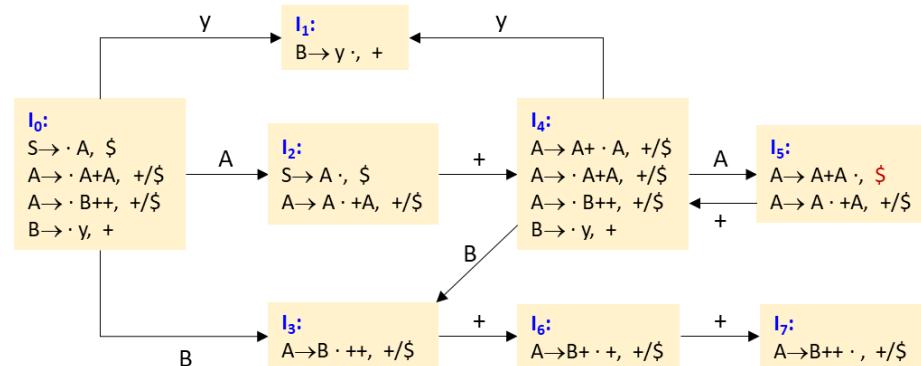
S	ACTION			GOTO	
	y	+	\$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		s4	r2		
6		s7			
7		r3	r3		

Exercise 2

- 对于句子 $y++$, 请给出其分析流程, 并说明 $y++$ 是否为该文法的一个句子?

(1) $S \rightarrow A$; (2) $A \rightarrow A + A$; (3) $A \rightarrow B ++$; (4) $B \rightarrow y$;

Stack	Input	ACTION	GOTO
<u>0</u>	\$	$y++\$$	s1
<u>01</u>	\$y	$\underline{++\$}$	r4($B \rightarrow y$)
0	\$B	$\underline{++\$}$	3
<u>03</u>	\$B	$\underline{++\$}$	s6
<u>036</u>	\$B+	$\underline{+\$}$	s7



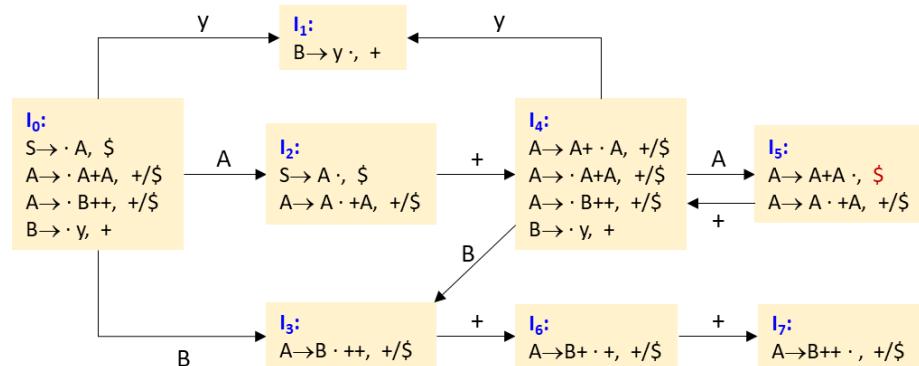
S	ACTION			GOTO	
	y	+	\$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		s4	r2		
6		s7			
7		r3	r3		

Exercise 2

- 对于句子 $y++$, 请给出其分析流程, 并说明 $y++$ 是否为该文法的一个句子?

(1) $S \rightarrow A$; (2) $A \rightarrow A + A$; (3) $A \rightarrow B ++$; (4) $B \rightarrow y$;

Stack	Input	ACTION	GOTO
<u>0</u>	\$	$y++\$$	s1
<u>01</u>	\$y	$\underline{++\$}$	r4($B \rightarrow y$)
0	\$B	$\underline{++\$}$	3
<u>03</u>	\$B	$\underline{++\$}$	s6
<u>036</u>	\$B+	$\underline{+\$}$	s7
<u>0367</u>	\$B++	$\underline{\$}$	r3($A \rightarrow B ++$)



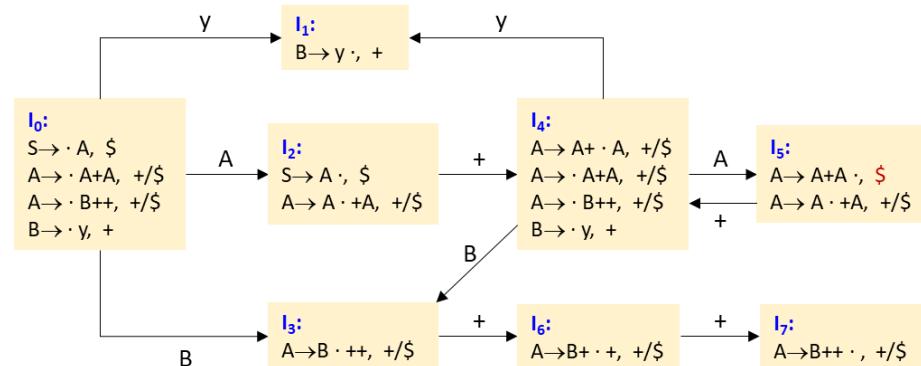
S	ACTION			GOTO	
	y	+	\$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		s4	r2		
6		s7			
7		r3	r3		

Exercise 2

- 对于句子 $y++$, 请给出其分析流程, 并说明 $y++$ 是否为该文法的一个句子?

(1) $S \rightarrow A$; (2) $A \rightarrow A + A$; (3) $A \rightarrow B ++$; (4) $B \rightarrow y$;

Stack	Input	ACTION	GOTO
<u>0</u>	\$	$y++\$$	s1
<u>01</u>	\$y	$\underline{+}+\$$	r4($B \rightarrow y$)
0	\$B	$\underline{+}+\$$	3
<u>03</u>	\$B	$\underline{+}+\$$	s6
<u>036</u>	\$B+	$\underline{+}\$$	s7
<u>0367</u>	\$B++	$\underline{\$}$	r3($A \rightarrow B ++$)
0	\$A	$\underline{\$}$	2



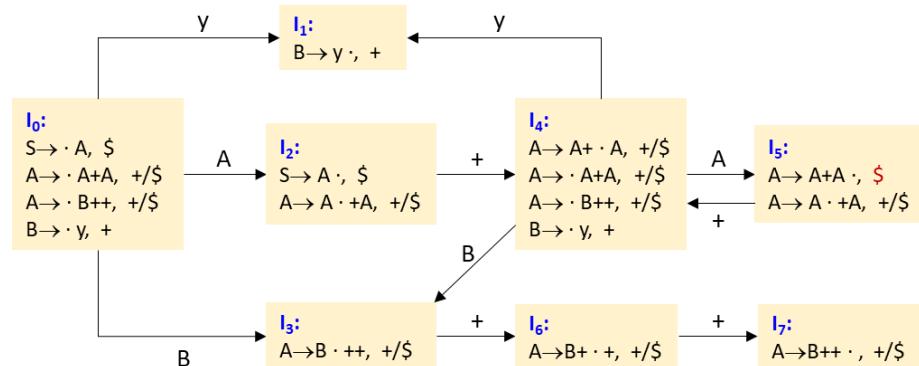
S	ACTION			GOTO	
	y	+	\$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		s4	r2		
6		s7			
7		r3	r3		

Exercise 2

- 对于句子 $y++$, 请给出其分析流程, 并说明 $y++$ 是否为该文法的一个句子?

(1) $S \rightarrow A$; (2) $A \rightarrow A + A$; (3) $A \rightarrow B ++$; (4) $B \rightarrow y$;

Stack	Input	ACTION	GOTO
<u>0</u>	\$	$y++\$$	s1
<u>01</u>	\$y	$\underline{+}+\$$	r4($B \rightarrow y$)
0	\$B	$\underline{+}+\$$	3
<u>03</u>	\$B	$\underline{+}+\$$	s6
<u>036</u>	\$B+	$\underline{+}\$$	s7
<u>0367</u>	\$B++	$\underline{\$}$	r3($A \rightarrow B ++$)
0	\$A	$\underline{\$}$	2
<u>02</u>	\$A	$\underline{\$}$	acc



S	ACTION			GOTO	
	y	+	\$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		s4	r2		
6		s7			
7		r3	r3		