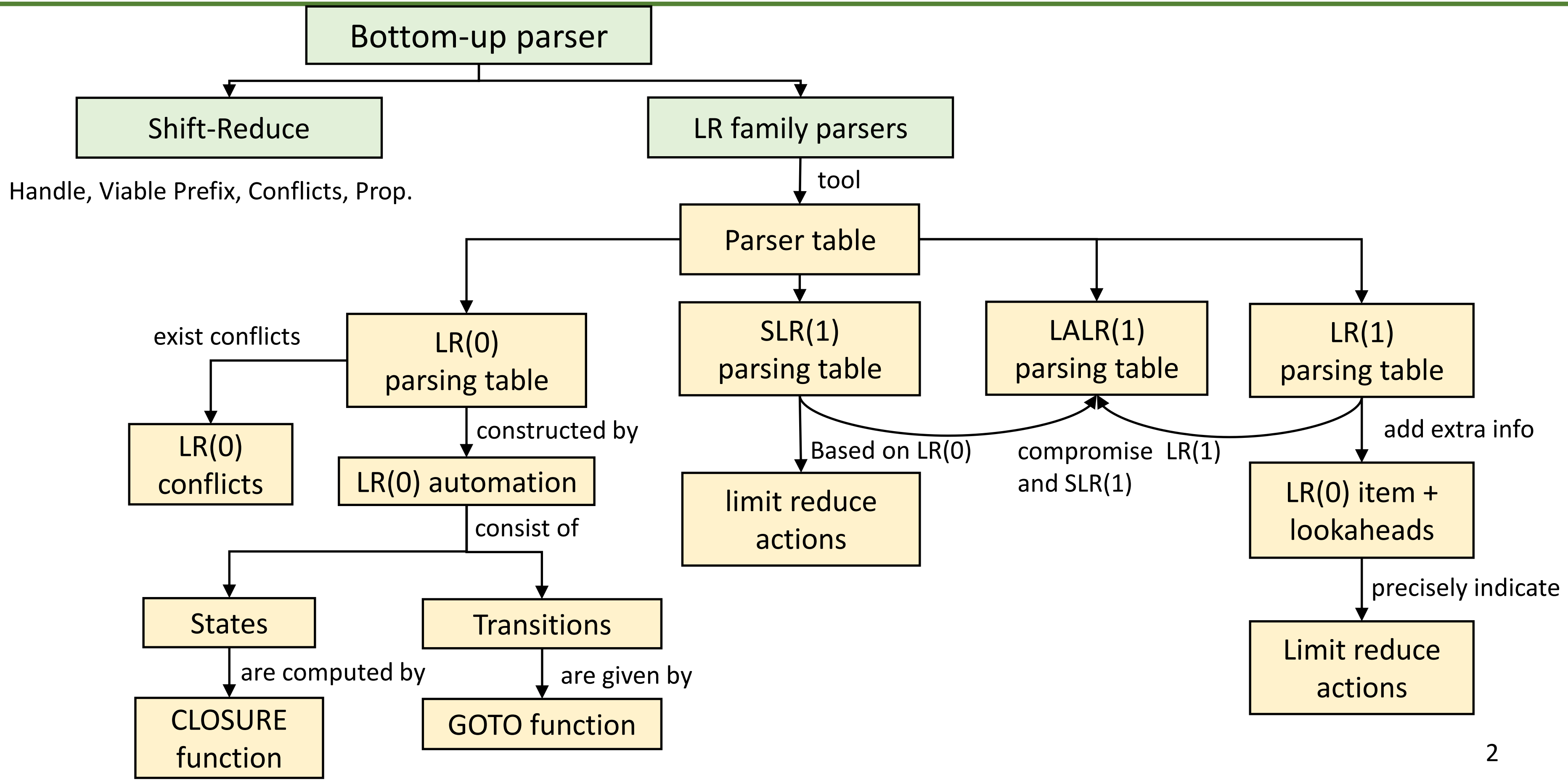
编译原理
Complier Principles

Lecture 5
Syntax Analysis: Bottom-Up

赵帅

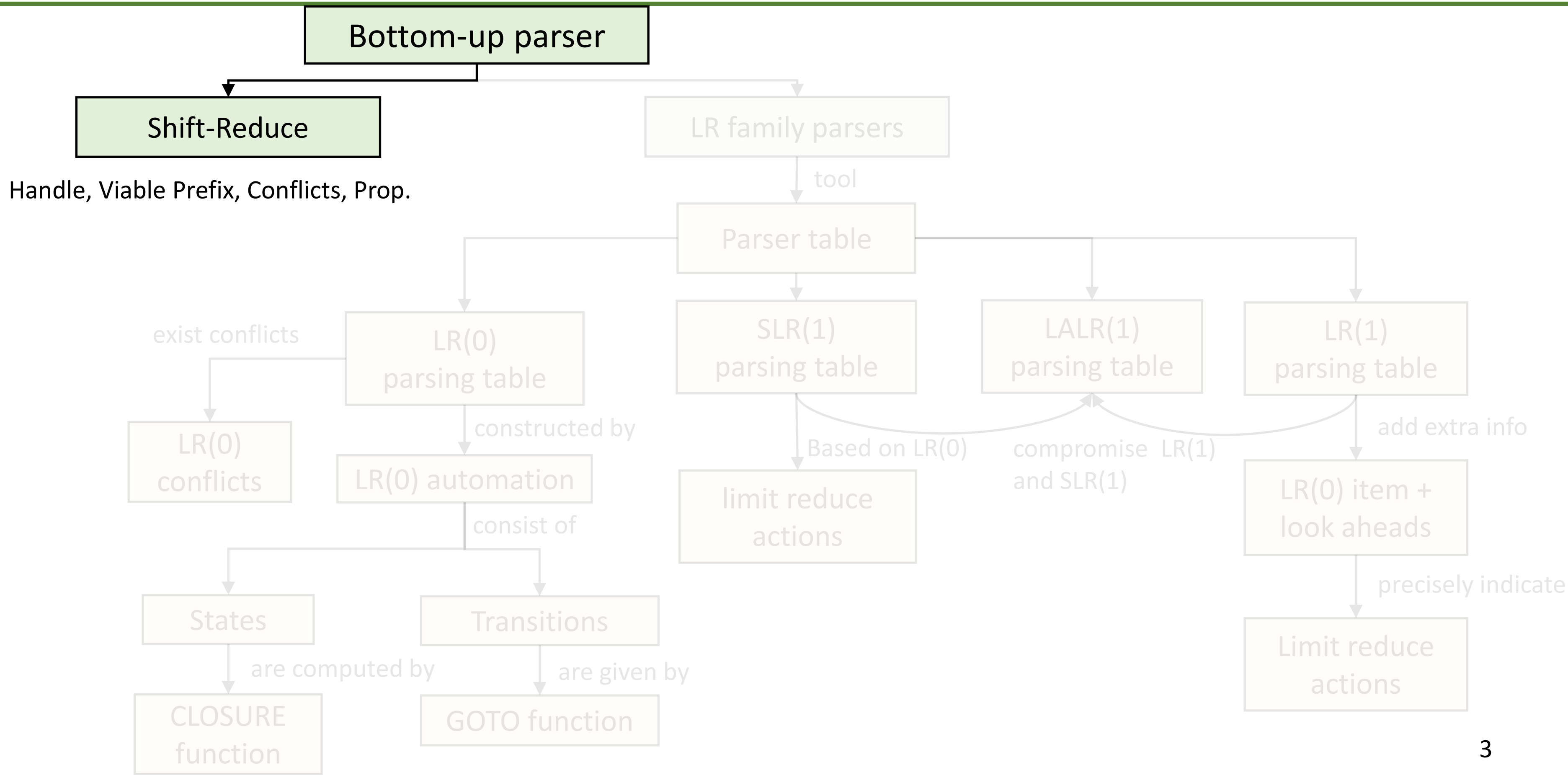
计算机学院
中山大学

Contents



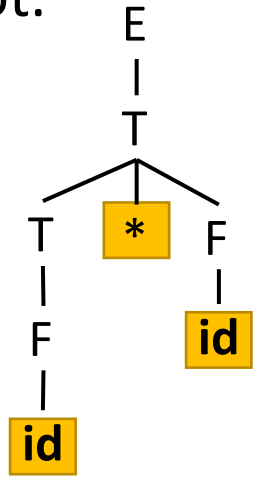
Bottom-up parser
Shift-Reduce
LR family parsers
Handle, Viable Prefix, Conflicts, Prop.
tool
Parser table
exist conflicts
LR(0) parsing table
SLR(1) parsing table
LALR(1) parsing table
LR(1) parsing table
LR(0) conflicts
constructed by
LR(0) automation
Based on LR(0)
compromise LR(1) and SLR(1)
add extra info
limit reduce actions
LR(0) item + lookaheads
consist of
States
Transitions
precisely indicate
are computed by
are given by
Limit reduce actions
CLOSURE function
GOTO function

Contents


Bottom-up parser
Shift-Reduce
Handle, Viable Prefix, Conflicts, Prop.
LR family parsers
tool
Parser table
exist conflicts
LR(0) parsing table
SLR(1) parsing table
LALR(1) parsing table
LR(1) parsing table
LR(0) conflicts
constructed by
LR(0) automation
Based on LR(0)
compromise LR(1) and SLR(1)
add extra info
limit reduce actions
LR(0) item + look aheads
consist of
precisely indicate
States
Transitions
Limit reduce actions
are computed by
are given by
CLOSURE function
GOTO function

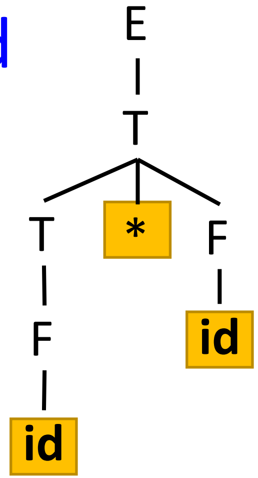
Bottom-up parsing[自下而上分析]

- 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 → E+T | T; T → T*F | F; F → (E) | **id**,   id * id
  - ◆ The rightmost derivation:
    - ▫ E ⇒ T ⇒ T * F ⇒ T * **id** ⇒ F * **id** ⇒ **id** * **id**
  - ◆ The leftmost reduction:
    - ◆ **id** * **id** ⇒ F * **id** ⇒ T * **id** ⇒ T * F ⇒ T ⇒ E



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]

SUN YAT-SEN UNIVERSITY

- 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 → aAcBe;
- A → b;
- A → Ab;
- B → d.

Input String:

- abbcde$

Stack	Input	Action
$	abbcde$	Shift
$a	bbcde$	Shift
$ab	bcde$	Reduce A → b
$aA	bcde$	Shift
$aAb	cde$	Reduce A → Ab
$aA	cde$	Shift
$aAc	de$	Shift
$aAcd	e$	Reduce B → d
$aAcB	e$	Shift
$aAcBe	$	Reduce S → 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 → aAcBe;
A → b;
A → Ab;
B → d.

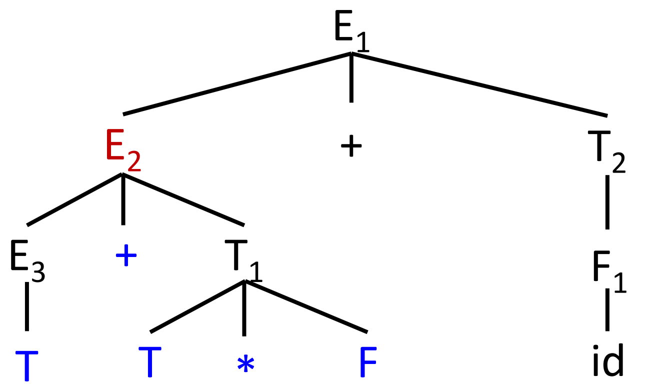
Handle [句柄]

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

G(E):

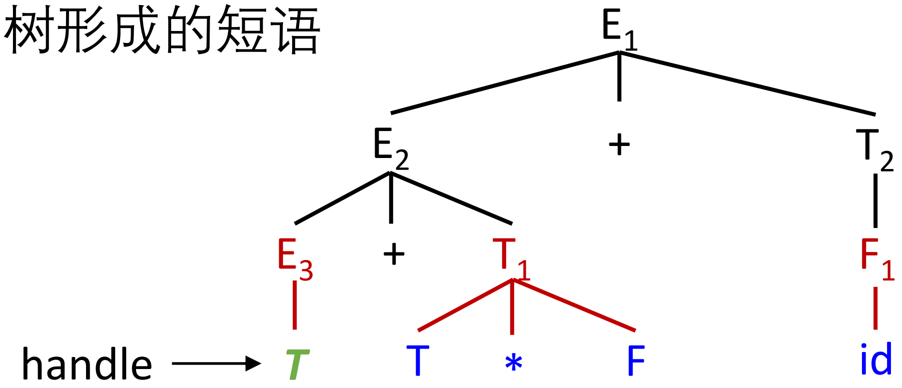
$E \Rightarrow T \Rightarrow T * F \Rightarrow T * id \Rightarrow F * id \Rightarrow id * id$

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



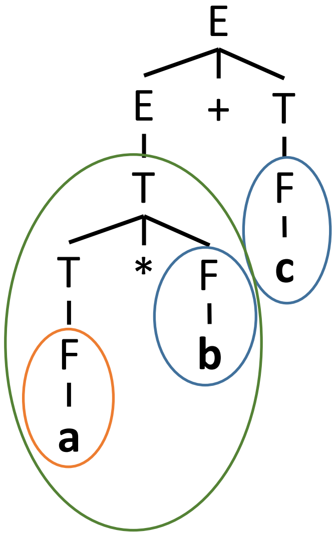
Handle [句柄]

- **Simple Phrase**[直接短语]: If $\alpha_1 A \alpha_2$ is a right sentential form of G(S), and $S \overset{*}{\Rightarrow} \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$, 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 → T | E+T; T → F | T*F; F → (E) | **a** | **b** | **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):

  $E \rightarrow E+T|T$

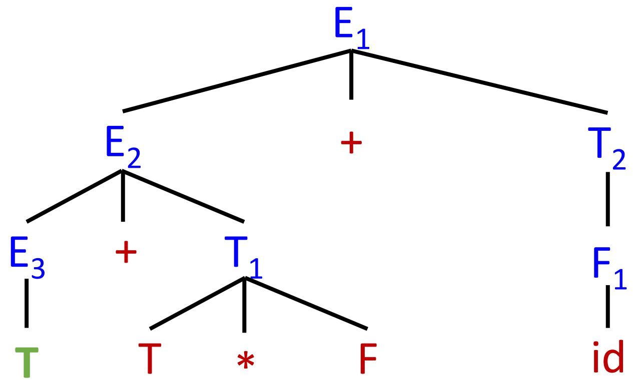
  $T \rightarrow T*F \mid F$

  $F \rightarrow (E) \mid id$

- 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

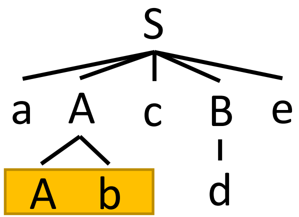
Simple Phrase



Example

- G(S): S → aAcBe; A → b; A → Ab; B → d.

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



Reduce A → Ab?
OR
Reduce A → b?

Ab is Handle, so we choose A → 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 ⇒ aEb ⇒ aaEb, the handle is **aE**, so the viable prefix is a, aa, aaE, but not aaEb.

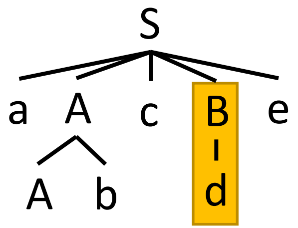
Stack	Input
$...	Handle...$
$...H	andle...$
$...Ha	ndle...$
$...Han	dle...$

Viable Prefix [活前缀]

- **Example 2:** G(S): S → aAcBe; A → b; A → Ab; B → d.

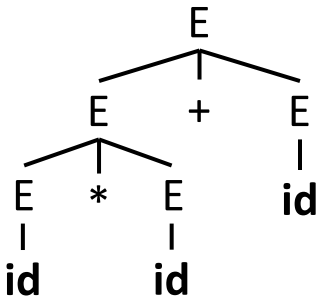
- “aAcde” is a right sentential form, it is split between the stack and the input buffer, aA, aAc, aAcd are all viable prefix of aAcde.

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



Conflicts[冲突]

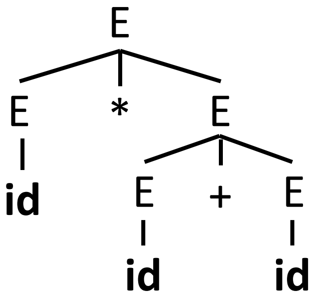
- Conflicts arise with ambiguous grammars.
- Consider G(E):
- E → E*E | E+E | (E) | **id,**
- sentential form: **id * id + id**



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 → E*E | E+E | (E) | **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	$E*E	+**id**$	**Shift**
6	$E*E+**id**	$	Reduce E → **id**
7	$E*E+E	$	Reduce E → E+E
8	$E*E	$	Reduce E → E*E
9	$E	$	**ACCEPT**

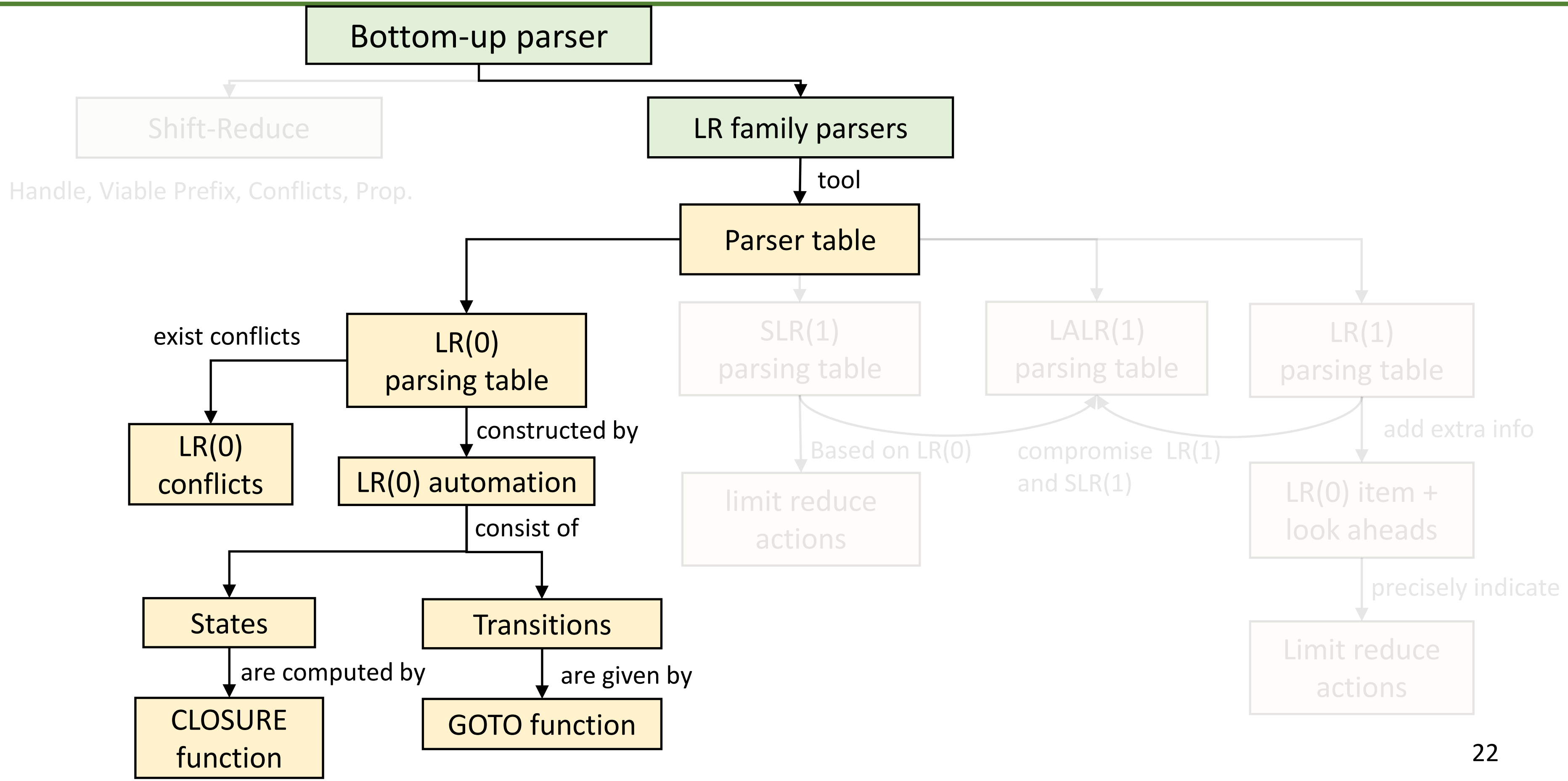
Conflicts[冲突]

- Conflicts arise with ambiguous grammars.
  - Grammar: E → E*E | E+E | (E) | **id**
  - sentential form: **id * id + id**
- In the steps marked blue, both *shift* or *reduce by E → 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)

Properties of Bottom-up Parsing

- **Handles** always appear **at the top** of the stack
  - ◆ Never in middle of stack
  - ◆ Justifies use of stack in shift–reduce parsing
- Results in an easily generalized **shift-reduce** strategy
  - ◆ If there is no handle at the top of the stack, shift
  - ◆ If there is a handle, reduce to the non-terminal
  - ◆ Easy to automate the synthesis of the parser using a table
- 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



Bottom-up parser
Shift-Reduce
Handle, Viable Prefix, Conflicts, Prop.
LR family parsers
tool
Parser table
SLR(1) parsing table
LALR(1) parsing table
LR(1) parsing table
Based on LR(0)
compromise LR(1) and SLR(1)
add extra info
limit reduce actions
LR(0) item + look aheads
precisely indicate
Limit reduce actions
exist conflicts
LR(0) parsing table
LR(0) conflicts
constructed by
LR(0) automation
consist of
States
Transitions
are computed by
are given by
CLOSURE function
GOTO function

Types of Bottom-Up Parsers

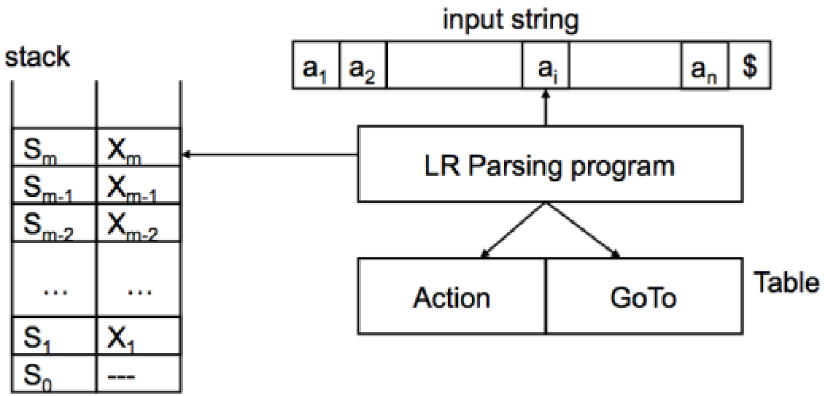
- Different types of bottom-up parsers:
  - Simple precedence parsers
  - Operator precedence parsers
  - Recursive ascent parsers
  - LR family parsers
  - others...
- Here, we will mainly focus on the **LR family parsers**, which are a type of bottom-up parser that analyzes deterministic context-free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

LR(k) Parser

中山大學 SUN YAT-SEN UNIVERSITY

- **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**, $S_0S_1...S_m$ (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 ($X_1X_2...X_m\ a_i...a_n$) 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

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.

State	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

(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1

Possible Actions[可能的动作]

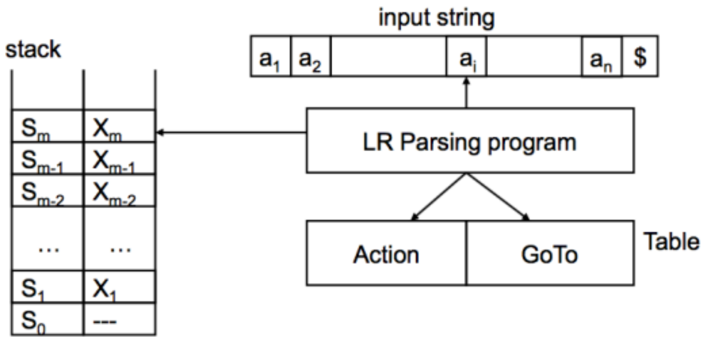
- ACTION[S_m, a_i] has four possible actions:
  - **Shift** (s_x): the handle is not completed loaded in the stack
    1. Shift the next input symbol onto the top of the stack
    2. Add State x to the stack
  - **Reduce** (r_y): the handle is at the top of the stack
    1. Pop the handle and all its associated states.
    2. Reduce the handle by the y^{th} 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



(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 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		

Parser Actions

Initial

S_0					
$					

$a_1a_2...a_n\$$

General

S_0	S_1	S_2	…	S_m	
$	X_1	X_2	…	X_m	

$a_ia_{i+1}...a_n\$$

- If 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

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

Parser Actions

- If ACTION[s_m, a_i] = r_x, then do **reduce**:
  - ◆ Pops k (states, symbols) from stack
  - ◆ Pushes A on stack
  - ◆ GOTO[S_{m-k}, A] = S_j, then

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

General

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

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

Pop

S_0	S_1	S_2	...	S_{m-k}	
\$	X_1	X_2	...	X_{m-k}	

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

Push & GOTO

S_0	S_1	S_2	...	S_{m-k}	S_j
\$	X_1	X_2	...	X_{m-k}	A

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

LR Parsing Program

- Input: string ω and parse table with ACTION/GOTO
- Output: reduction steps ω's bottom-up parse, or error
- Initial: s_0 on the stack, $\omega\$$ 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 -> β */
                pop |β| (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 → E * B

  (2) E → E + B

  (3) E → B

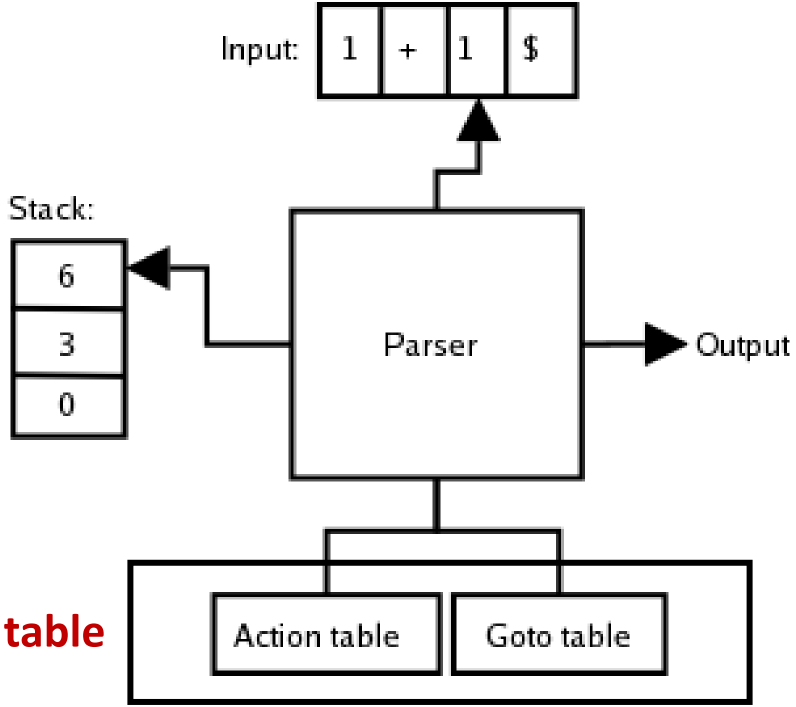
  (4) B → 0

  (5) B → 1

- to parse the following input:

  1 + 1

Parse table



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

Example: Parse Table

- Grammar:
  - (1) E → E * B
  - (2) E → E + B
  - (3) E → B
  - (4) B → 0
  - (5) B → 1

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 → E * B (2) E → E + B (3) E → B (4) B → 0 (5) B → 1

Stack		input	ACTION	GOTO
0	$	**1**+1$	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

中山大學 SUN YAT-SEN UNIVERSITY

- (1) E → E * B (2) E → E + B (3) E → B (4) B → 0 (5) B → 1

Stack		input	ACTION	GOTO
0	$	**1**+1$	s2	
0<u>2</u>	$1	<u>+</u>1$	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 → E * B   (2) E → E + B   (3) E → B   (4) B → 0   (5) B → 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 → E * B (2) E → E + B (3) E → B (4) B → 0 (5) B → 1

Stack		input	ACTION	GOTO
0	$	**1**+1$	s2	
02	$1	**+**1$	r5	4
04	$B	**+**1$	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 → E * B (2) E → E + B (3) E → B (4) B → 0 (5) B → 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

中山大學 SUN YAT-SEN UNIVERSITY

- (1) E → E * B (2) E → E + B (3) E → B (4) B → 0 (5) B → 1

Stack		input	ACTION	GOTO
0	$	**1**+1$	s2	
02	$1	**+**1$	r5	4
04	$B	**+**1$	r3	3
03	$E	**+**1$	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

中山大學 SUN YAT-SEN UNIVERSITY

- (1) E → E * B (2) E → E + B (3) E → B (4) B → 0 (5) B → 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	

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

中山大學 SUN YAT-SEN UNIVERSITY

- (1) E → E * B (2) E → E + B (3) E → B (4) B → 0 (5) B → 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

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

中山大學 SUN YAT-SEN UNIVERSITY

- (1) E → E * B (2) E → E + B (3) E → B (4) B → 0 (5) B → 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 → E * B   (2) E → E + B   (3) E → B   (4) B → 0   (5) B → 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

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 → E * B (2) E → E + B (3) E → B (4) B → 0 (5) B → 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

中山大學 SUN YAT-SEN UNIVERSITY

- (1) E → E * B   (2) E → E + B   (3) E → B   (4) B → 0   (5) B → 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?

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	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		

Items and the LR(0) Automaton

- LR(0) Parsing
  - The LR parser using LR(0) parsing table is LR(0) parser
  - The grammar for which an LR(0) parsing table can be constructed is said to be LR(0) grammar
  - LR(0) parser uses only the content of stack to determine handle, it doesn't need input token as lookahead
  - Almost all “real” grammars are not LR(0)
  - But LR(0) method is a good starting point for learning LR parsing

State in LR Parsing[状态]

- How does a shift-reduce parser know when to shift and when to reduce? [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?
- 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 -> T
- T -> F | T * F
- F -> id

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2\,\$$	shift
$\$\,\mathbf{id}_1$	$* \mathbf{id}_2\,\$$	reduce by $F \to \mathbf{id}$
$\$\,F$	$* \mathbf{id}_2\,\$$	reduce by $T \to F$
$\boxed{\$\,T}$	$* \mathbf{id}_2\,\$$	shift
$\$\,T *$	$\mathbf{id}_2\,\$$	shift
$\$\,T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$\,T * F$	$\$$	reduce by $T \to T * F$
$\$\,T$	$\$$	reduce by $E \to T$
$\$\,E$	$\$$	accept

Item[项目]

- 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 → XYZ** has the following four items:
  - **A → • XYZ**
  - **A → X • YZ**
  - **A → XY • Z**
  - **A → XYZ •**
- Rules of the form A → ε have only a single item A → •

The meaning of Items

- A → • XYZ
  - Indicates that we hope to see a string derivable from XYZ next on the input
- A → X • 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 → XY • Z
- A → XYZ •
  - Indicates that we have seen the body XYZ and that it may be time to reduce XYZ to A

State[状态]

- Example:
  - Suppose we are currently in this position: **A → X • YZ**
  - We have just recognized X and expect the upcoming input to contain a sequence derivable from YZ (say, Y → u|w)
  - Y is further derivable from either u or w:
    - **A → X • YZ**
    - **Y → • u**
    - **Y → • 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 (•S) and move to an item with a dot after S (S•)
  - ◆ 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'-> •S becomes the first item in the start state of the FA

Grammar

(0) E → E * B (1) E → E + B
(2) E → B (3) B → 0 (4) B → 1

⇨

Augmented Grammar

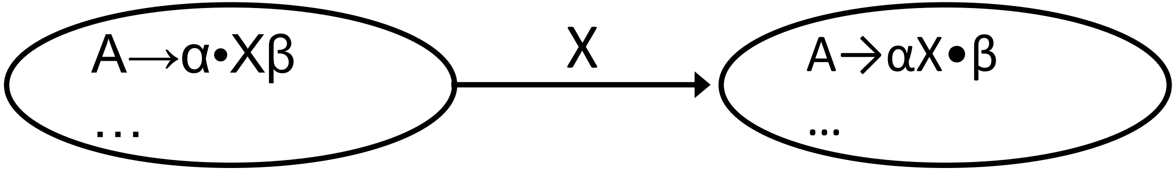
(0) S → E (1) E → E * B (2) E → E + B
(3) E → B (4) B → 0 (5) B → 1

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’ → ·S S → ·BB B → ·aB B → ·b

(0) S’ → S
(1) S → BB
(2) B → aB
(3) B → b

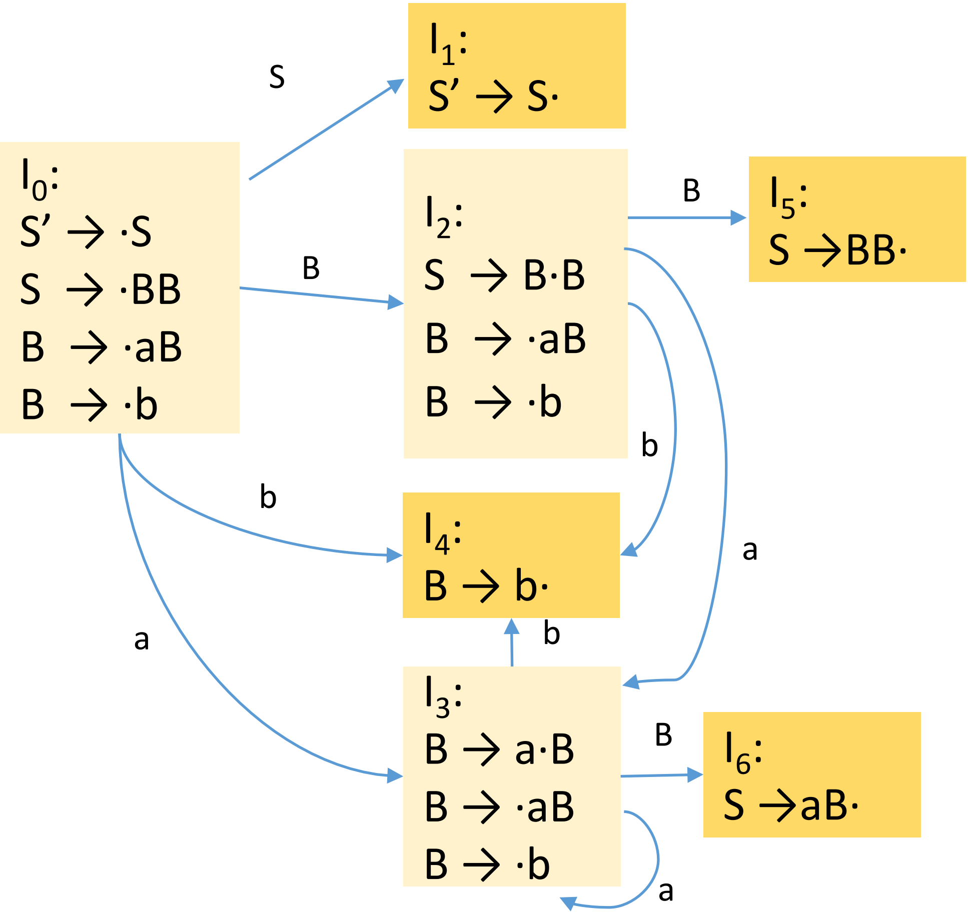
- How to construct transition form one state to another? -- goto operation
  - ◆ A transition on symbol X from state i to state j



Example

Grammar

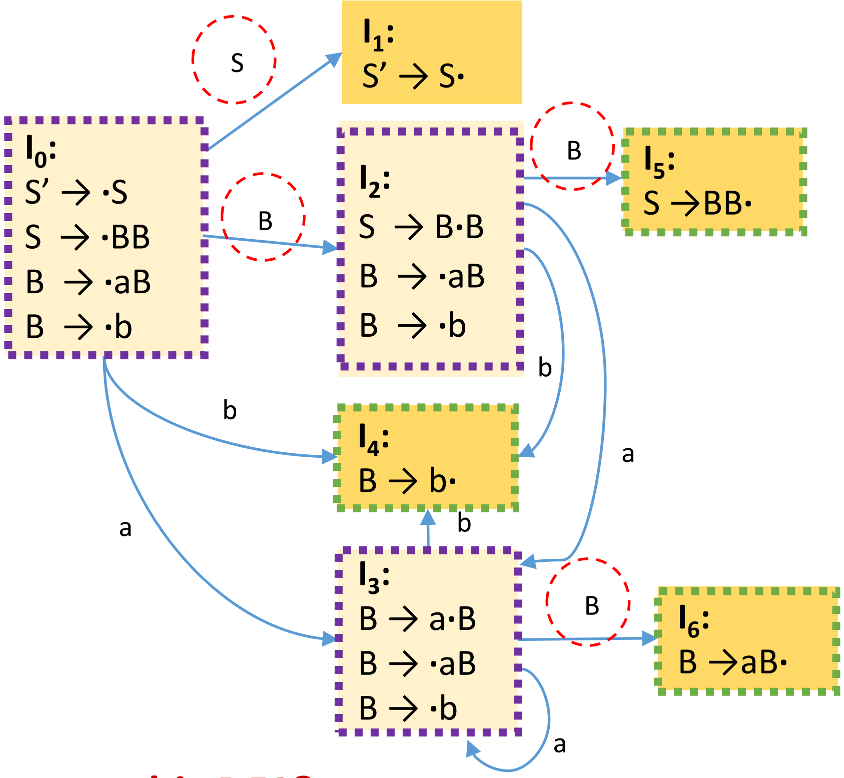
(0) S' -> S
(1) S -> BB
(2) B -> aB
(3) B -> b



Example (cont.)

Grammar

(0) S’ -> S
(1) S -> BB
(2) B -> aB
(3) B -> 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 ***CLOSURE(I)*** is the set of items constructed from *I* by the two rules:
  1. Initially, add every item in *I* to CLOSURE(*I*)
  2. For any item $A \rightarrow \alpha \cdot B\beta$ in 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 CLOSURE(*I*)

  ◆ Apply the above rules until no more new items can be added to *CLOSURE(I)*

CLOSURE(S’ → ·S)

Grammar：
S’ → S
S → BB
B → aB
B → b

S’ → ·S

S’ → ·S
S → ·BB

CLOSURE(S’ → ·S)：
S’ → ·S
S → ·BB
B → ·aB
B → ·b

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 GOTO(I, X) specifies the transition from the state for *I* under input *X*

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

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

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

$I_0 \xrightarrow{B} I_2$

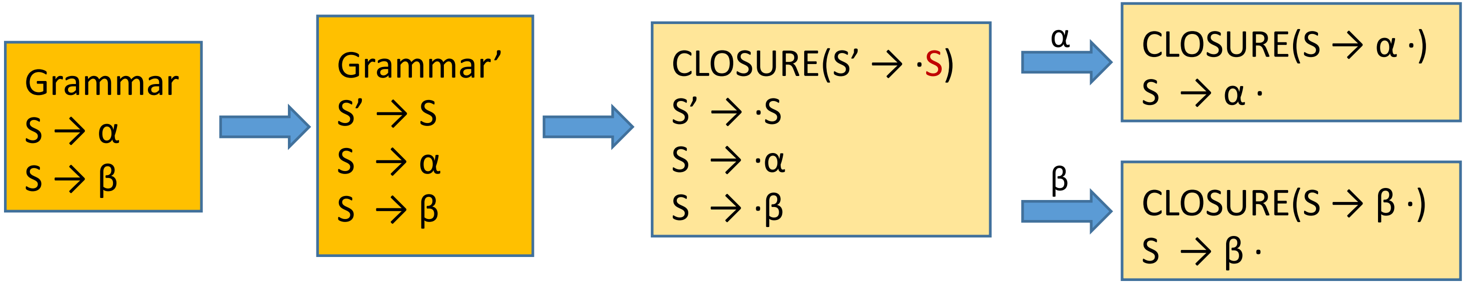
I_2:
$S \rightarrow B\cdot B$
$B \rightarrow \cdot aB$
$B \rightarrow .b$

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

GOTO(I_0, B)

Construct LR(0) States[构造LR(0)状态集]

- **Create augmented grammar G' for G:** [增广文法]
  - Given G: S → α | β, create G': S' → S , S → α | β
  - Creates a single rule S' → S that when reduced, signals acceptance
- **Create 1st state by performing a closure on initial item S'→ ·S** [初始状态]
  - Closure({S'→ ·S}) = {S'→ ·S, S → ·α, S → ·β}
- **Create additional states by performing a GOTO on each symbol** [添加状态]
  - Goto({S'→ ·S, S → ·α, S → ·β}, α) =Closure({S → α·}) = {S → α·}
- **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
  - ◆ 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)
  - ◆ 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' → ·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 some 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

S_0= Closure({S'→ · S})
 = {S'→ · S, S → · BB, B → · aB, B → · b} (I_0)

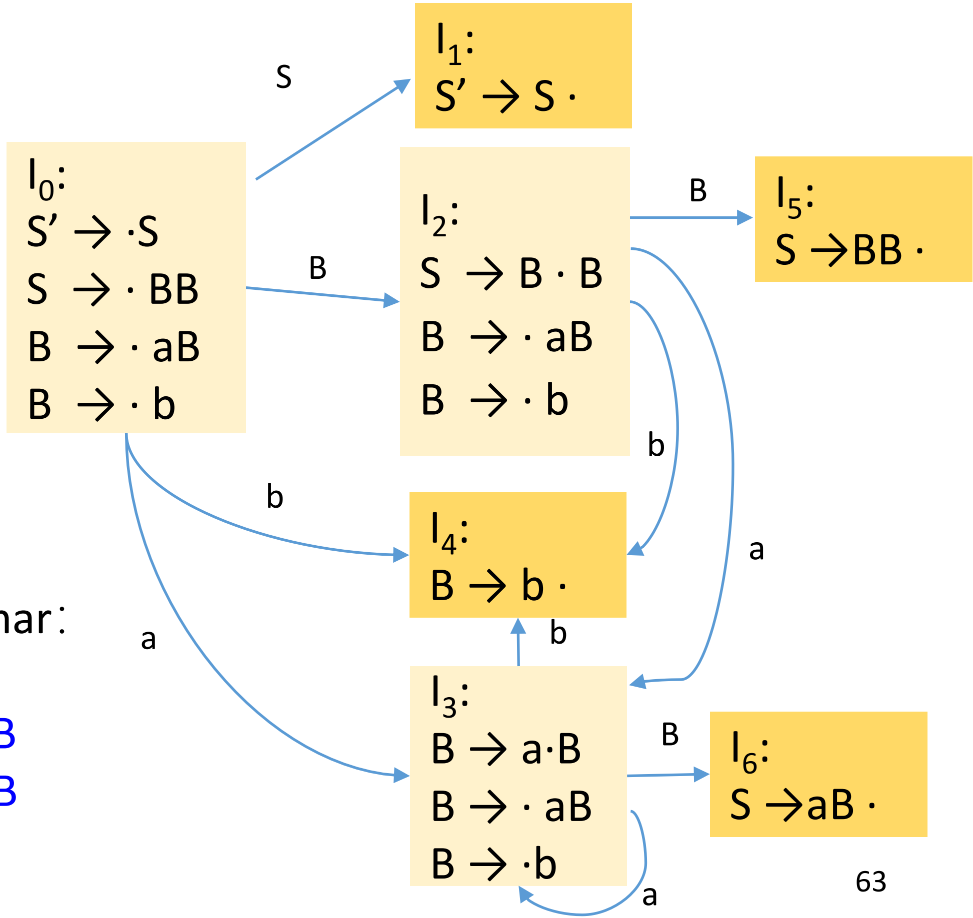
goto(S_0, B) = closure({S → B · B})
S_2= {S →B · B, B → · aB, B → · b} (I_2)

goto(S_0, a) = closure({B →a · B})
S3= {B →a · B, B → · aB, B → · b} (I_3)

goto(S_0, b) = closure({B → b ·})
S4= {B→ b ·} (I_4)

…

Grammar:
S' → S
S → BB
B → aB
B → b



Build Parse Table from DFA

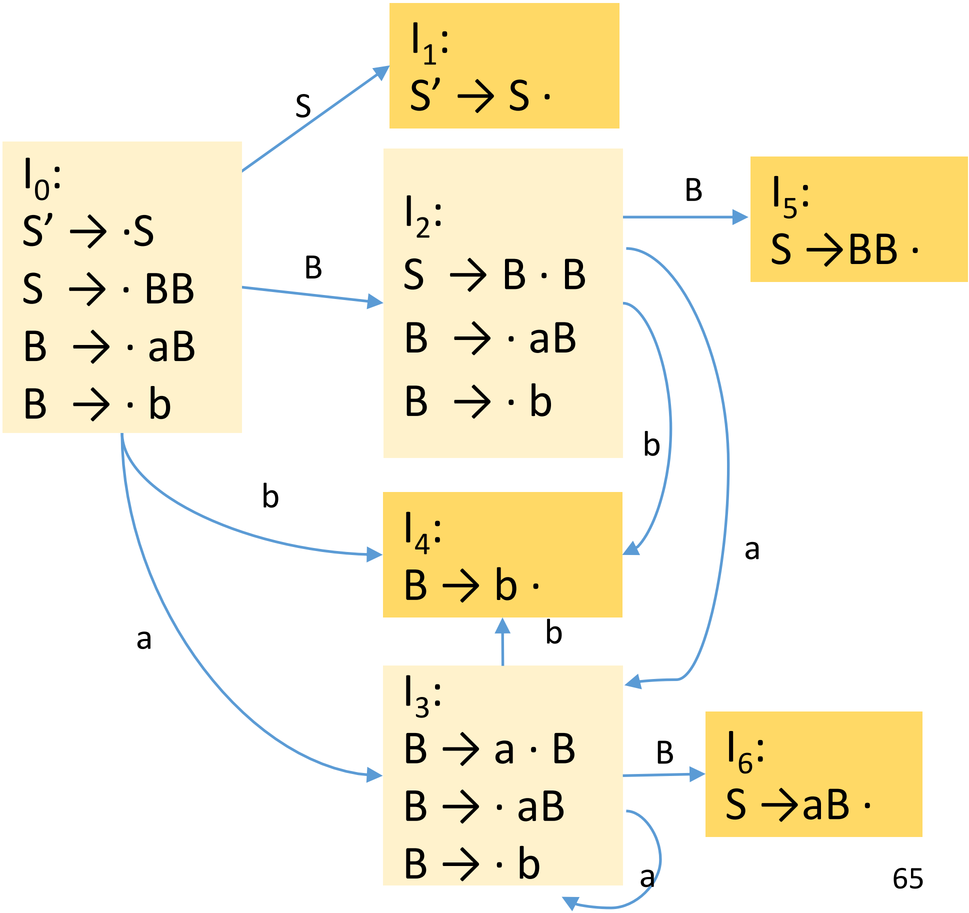
- **ACTION** $[S_i, a]$:
  - If $[A \rightarrow \alpha \cdot a\beta]$ is in S_i and $goto(S_i, a) = S_j$, where "a" is a terminal, then $ACTION[S_i, a] = S_j$ (shift j)
  - If $[A \rightarrow \alpha \cdot]$ is in S_i and $A \rightarrow \alpha$ is j^{th} rule, then $ACTION[S_i, a] = r_j$ (reduce j)
  - If $[S' \rightarrow S \cdot]$ is in S_i, then $ACTION[S_i, \$] = accept(acc)$
  - 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 $goto(S_i, A) = S_j$ then $GOTO[S_i, A] = j$

The example

Grammar： (0) S' → S
(1) S → BB
(2) B → aB
(3) B → 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
  - DFA on input α terminates 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) configurating set can only have exactly one reduce item and cannot have both shift and reduce items
  - If the grammar contains the production $A \rightarrow \varepsilon$, then the item $A \rightarrow \cdot\varepsilon$ 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：
E'→ E
E → E+T | T | V=E
T → (E) | i | i[E]
V → i

I_0:
E'→ ·E
E → ·T
E → ·E+T
E → ·V=E
T → ·(E)
T → ·i
T → ·i[E]
V → ·i

$\xrightarrow{i}$

I_1:
T → i ·
T → i · [E]
V → i ·

LR(0) Limitations[局限性]

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

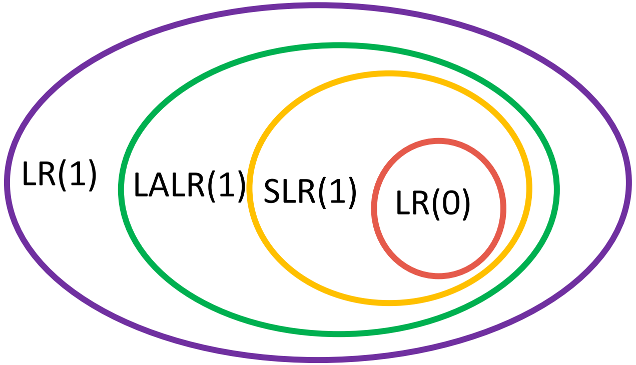
$A \to \alpha\cdot\alpha\beta$
$B \to r\cdot$

$A \to \beta\cdot$
$B \to r\cdot$

  - If the item is complete ($A \to \alpha\cdot$), 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

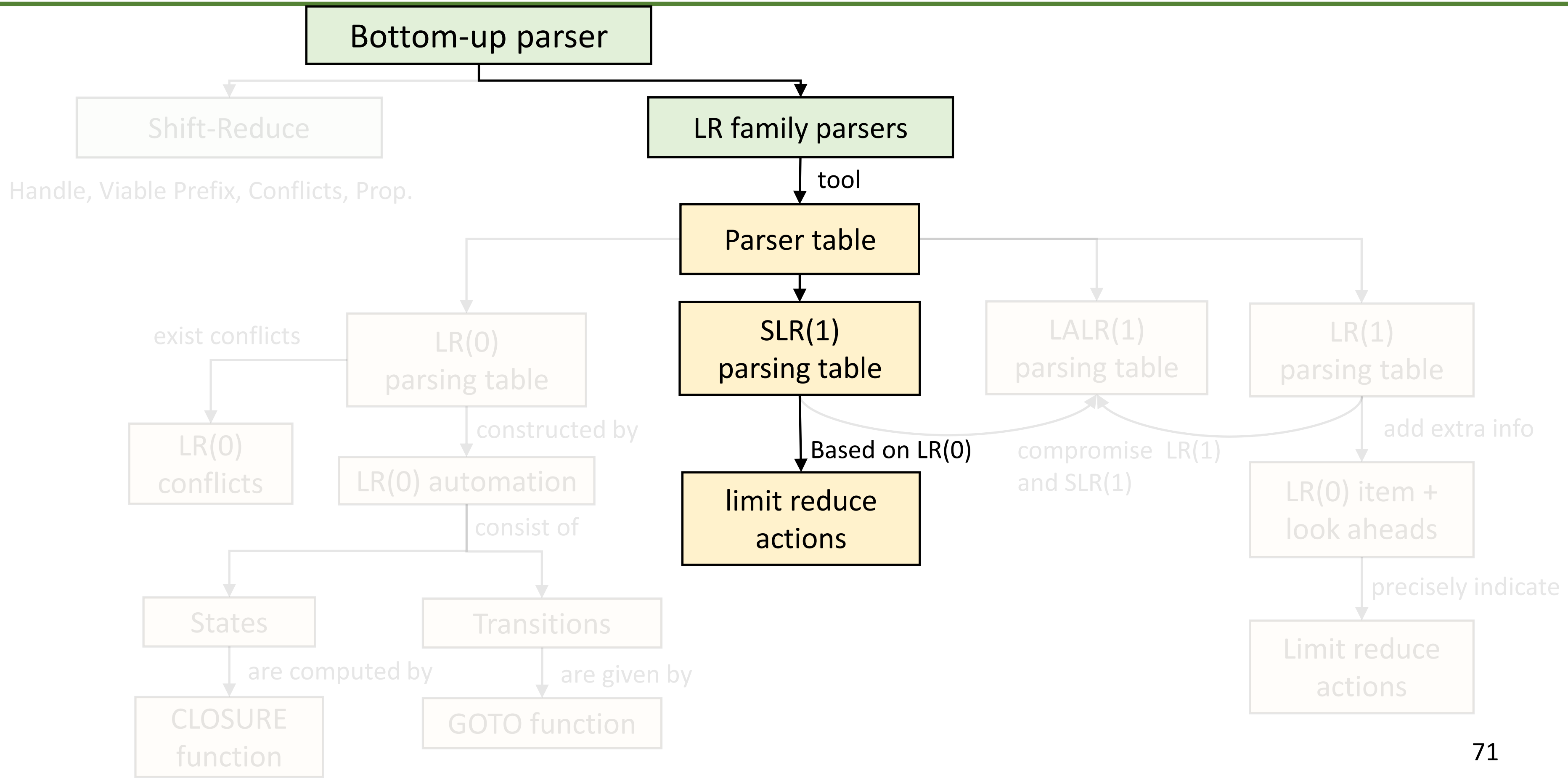
LR(0) Summary

- 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 [考虑展望]
  - ◆ LR(1) : simple LR(1), use FOLLOW[归约用FOLLOW]
  - ◆ SLR(1) : use dedicated symbols[比FOLLOW更精细]
  - ◆ LALR(1) : balance SLR(1) and LR(1)[折衷]

Contents

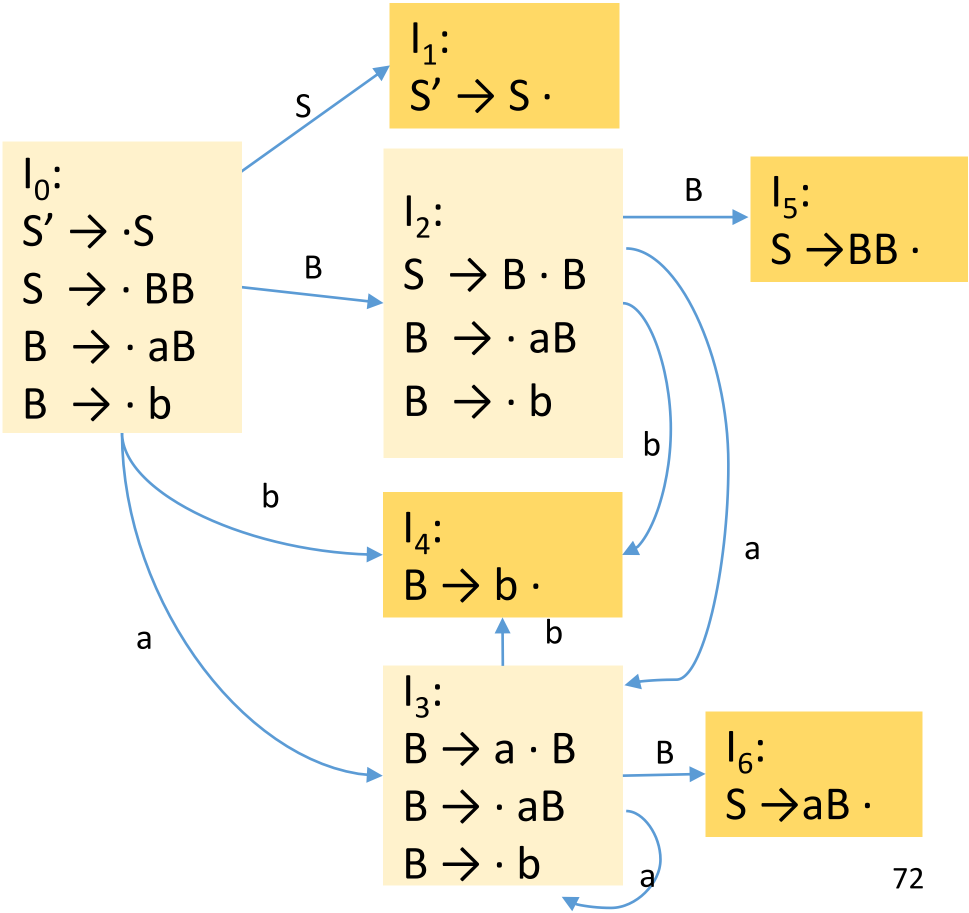


Bottom-up parser
Shift-Reduce
LR family parsers
Handle, Viable Prefix, Conflicts, Prop.
tool
Parser table
exist conflicts
LR(0) parsing table
SLR(1) parsing table
LALR(1) parsing table
LR(1) parsing table
LR(0) conflicts
constructed by
Based on LR(0)
compromise LR(1) and SLR(1)
add extra info
LR(0) automation
limit reduce actions
LR(0) item + look aheads
consist of
precisely indicate
States
Transitions
Limit reduce actions
are computed by
are given by
CLOSURE function
GOTO function

LR(0) Example (revisit)

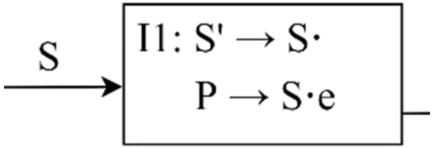
Grammar: (0) S' → S
(1) S → BB
(2) B → aB
(3) B → b

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) configurating 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
  - Follow(T) = { +,),], $ }, Follow(V) = { = }
    - If input t in Follow(T), it will reduce to T
    - If input t in Follow(V), it will reduce to V
    - Else the input [will shift

FOLLOW(A) = $\{a \mid S \stackrel{*}{\Rightarrow} \ldots Aa \ldots, a \in V_T\}$, including \$[结束标记]

Grammar：
E' → E
E → E+T | T | V=E
T → (E) | id | id[E]
V → id

I_0:
E' → · E
E → · T
E → · E+T
E → · V=E
T → · (E)
T → · id
T → · id[E]
V → · id

id

I_1:
T → id ·
T → id · [E]
V → id ·

SLR(1) Grammars[SLR(1)文法]

- A grammar is SLR(1) if the following two conditions hold for each configurating set:
  1. For any item A → u·xv in the set, with terminal x, there is no complete item B → w· in that set with x in Follow(B)
     - In the tables, this translates no shift-reduce conflict on any state
  2. For any two complete items A → u· and B → v· in the set, the follow sets must be disjoint, i.e., Follow(A) ∩ 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 along 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 –> u· ,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 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 : id = 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
    1. S_5 has a completed item, and next input = is in Follow(L)
    2. S_5 and id are popped from stack , and L is pushed onto stack
    3. GOTO(S_0, L) = S_2

I_0:
S' → ·S
S → · L = R
S → · R
L → · *R
L → · id
R → · L

I_1:
S' → S ·

I_2:
S → L · = R
R → L ·

I_3:
S → R ·

I_4:
L → * · R
L → · id
L → · *R
L –> · id

I_5:
L → id ·

I_6:
S → L = · R
R → · L
L → · *R
L → · id

I_7:
L → *R ·

I_8:
R → L ·

I_9:
S → L = R ·

$I_0 \xrightarrow{id} I_5$

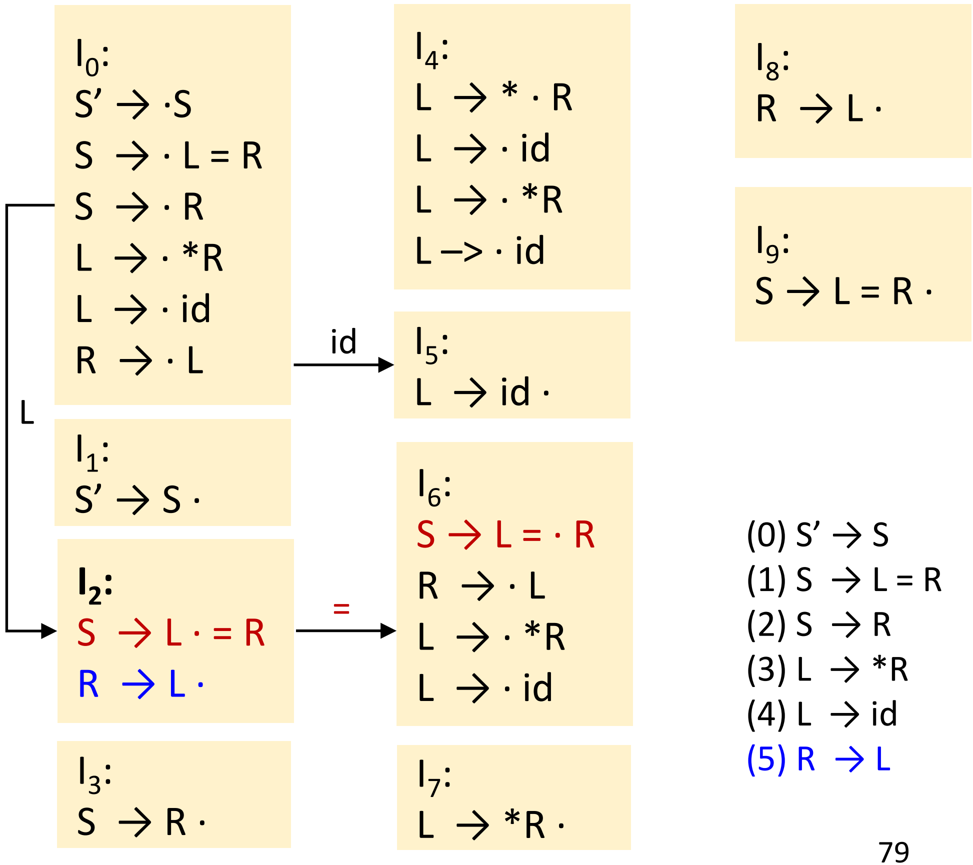
$I_0 \xrightarrow{L} I_2$

(0) S' → S
(1) S → L = R
(2) S → R
(3) L → *R
(4) L → id
(5) R → L

https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf

SLR(1) Limitation Example (Cont.)

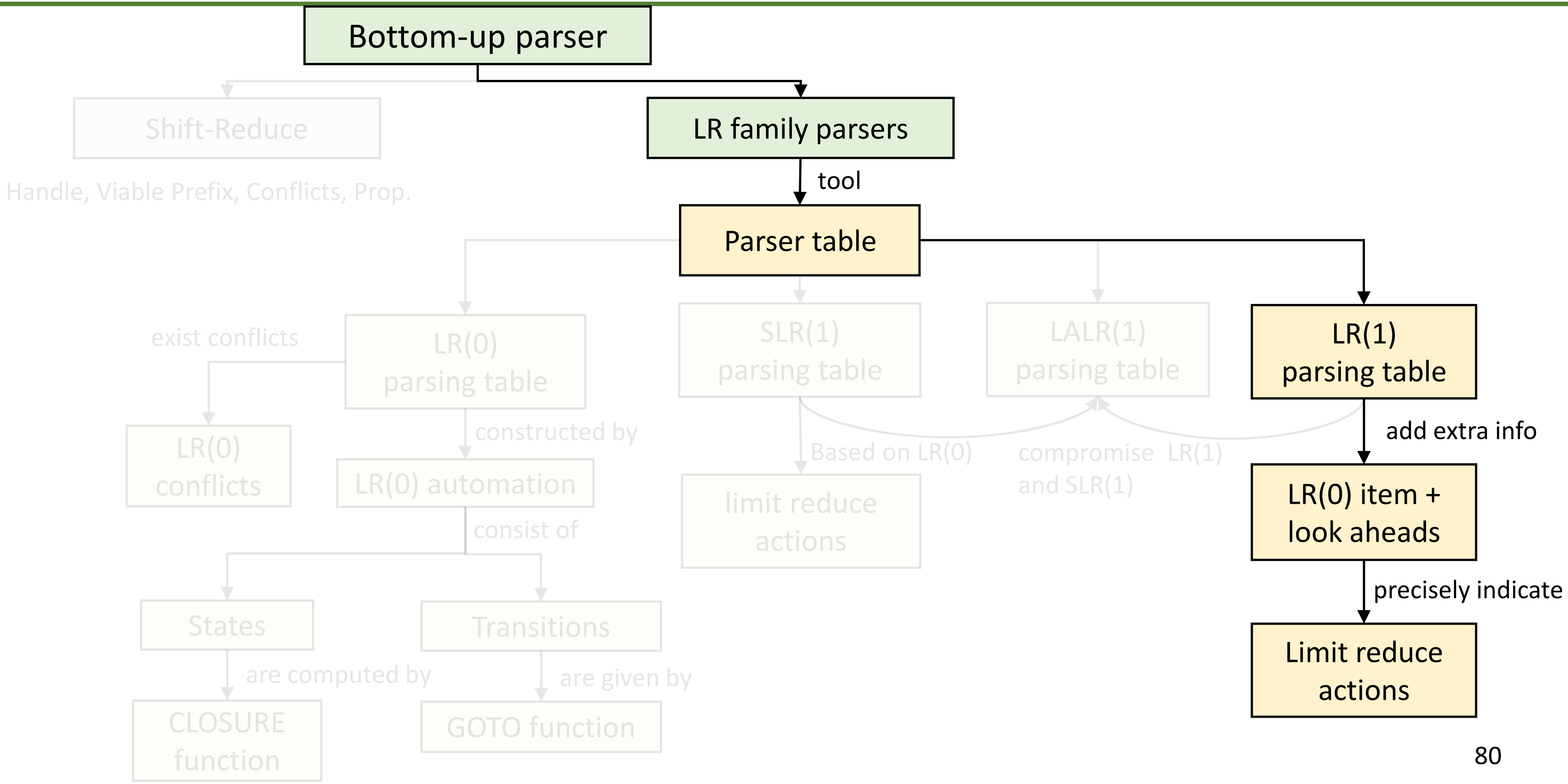
- Choices upon seeing = coming up in the input:
  - Action[2, =] = s6 :
    - Move on to find the rest of assignment
  - Action[2, =] = r5 :
    - = ∈Follow(R)
    - Leads to a wrong direction!
- This is a shift-reduce conflict
  - Should only reduce when we have seen * or =, i.e., when the next input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



(0) S’ → S
(1) S → L = R
(2) S → R
(3) L → *R
(4) L → id
(5) R → L

https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf

Contents



Bottom-up parser
Shift-Reduce
Handle, Viable Prefix, Conflicts, Prop.
LR family parsers
tool
Parser table
LR(0)
parsing table
exist conflicts
LR(0)
conflicts
constructed by
LR(0) automation
consist of
States
Transitions
are computed by
are given by
CLOSURE
function
GOTO function
SLR(1)
parsing table
Based on LR(0)
limit reduce
actions
LALR(1)
parsing table
compromise LR(1)
and SLR(1)
LR(1)
parsing table
add extra info
LR(0) item +
look aheads
precisely indicate
Limit reduce
actions

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 [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 [额外使用栈信息，FOLLOW是input buffer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**: $A \rightarrow X_1 \ldots X_i \bullet X_{i+1} \ldots X_j$, a
  - We have states $X_1 \ldots X_i$ on the stack
  - suppose $X_{i+1} \ldots X_j$ 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]
  - $A \rightarrow X_1 \ldots X_j \bullet$, 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' -> ·S	S' -> ·S, $
LR(0)	LR(1)

LR(1) Construction

中山大學 SUN YAT-SEN UNIVERSITY

- 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 -> u·Bv, a] in *I*, for each production rule B -> w in G', add [B -> ·w, b] to *I*:
    - **b ∈ First(va)** && [B -> ·w, b] is not in *I*
    - v can be nullable
  - Lookahead is the **First(va)**, which are the symbols that can follow B

FIRST(α) = {a | α ⇒ a…, a ∈ VT}, including ε[空串]

(0) S' -> S
(1) S -> XX
(2) X -> aX
(3) X ->b

S' -> ·S, $

→

I_0:
S' -> ·S, $
S -> .XX, First(ε$)
X -> .aX, First(X$)
X -> .b, First(X$)

→

I_0:
S' -> ·S, $
S -> .XX, $
X -> .aX, a/b
X -> .b, a/b

LR(1) Construction (cont.)

- **Goto(I, X)**
  - For item [A -> u·Xv, a] in *I*, Goto(I, X) = Closure ([A -> uX·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’ -> ·S, $])
  - Construct configuration sets following Goto(I, X)
  - Repeat until no new sets can be added

I_0 :
S’ -> ·S, $
S -> .XX, $
X -> .aX, a/b
X -> .b, a/b

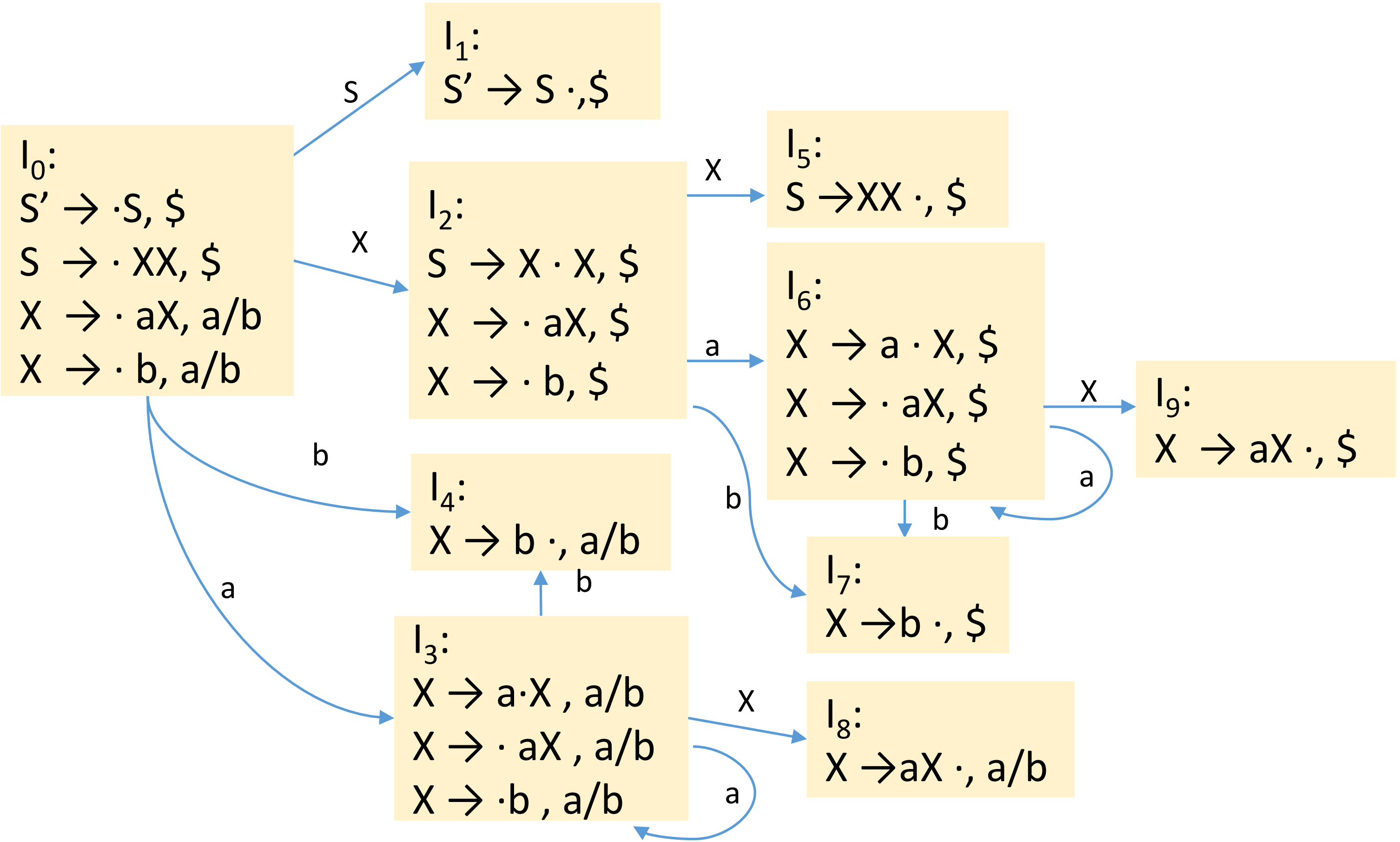
X →

I_2 :
S -> X.X, $
X -> .aX, First(ε$)
X -> .b, First(ε$)

→

I_2 :
S -> X.X, $
X -> .aX, $
X -> .b, $

Example



I_1:
$S' \to S \cdot, \$$
I_0:
$S' \to \cdot S, \$$
$S \to \cdot XX, \$$
$X \to \cdot aX, a/b$
$X \to \cdot b, a/b$
I_2:
$S \to X \cdot X, \$$
$X \to \cdot aX, \$$
$X \to \cdot b, \$$
I_5:
$S \to XX \cdot, \$$
I_6:
$X \to a \cdot X, \$$
$X \to \cdot aX, \$$
$X \to \cdot b, \$$
I_9:
$X \to aX \cdot, \$$
I_4:
$X \to b \cdot, a/b$
I_7:
$X \to b \cdot, \$$
I_3:
$X \to a \cdot X, a/b$
$X \to \cdot aX, a/b$
$X \to \cdot b, a/b$
I_8:
$X \to aX \cdot, a/b$
S
X
a
b

LR(1) Parse Table[解析表]

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

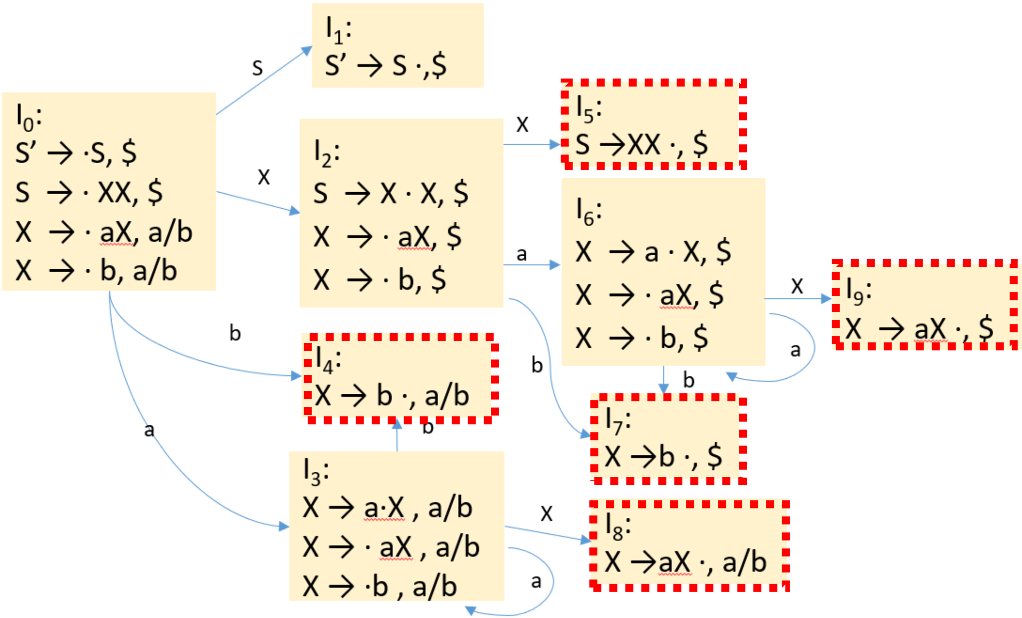
Example

(0) S' -> S

(1) S -> XX

(2) X -> aX

(3) X -> b

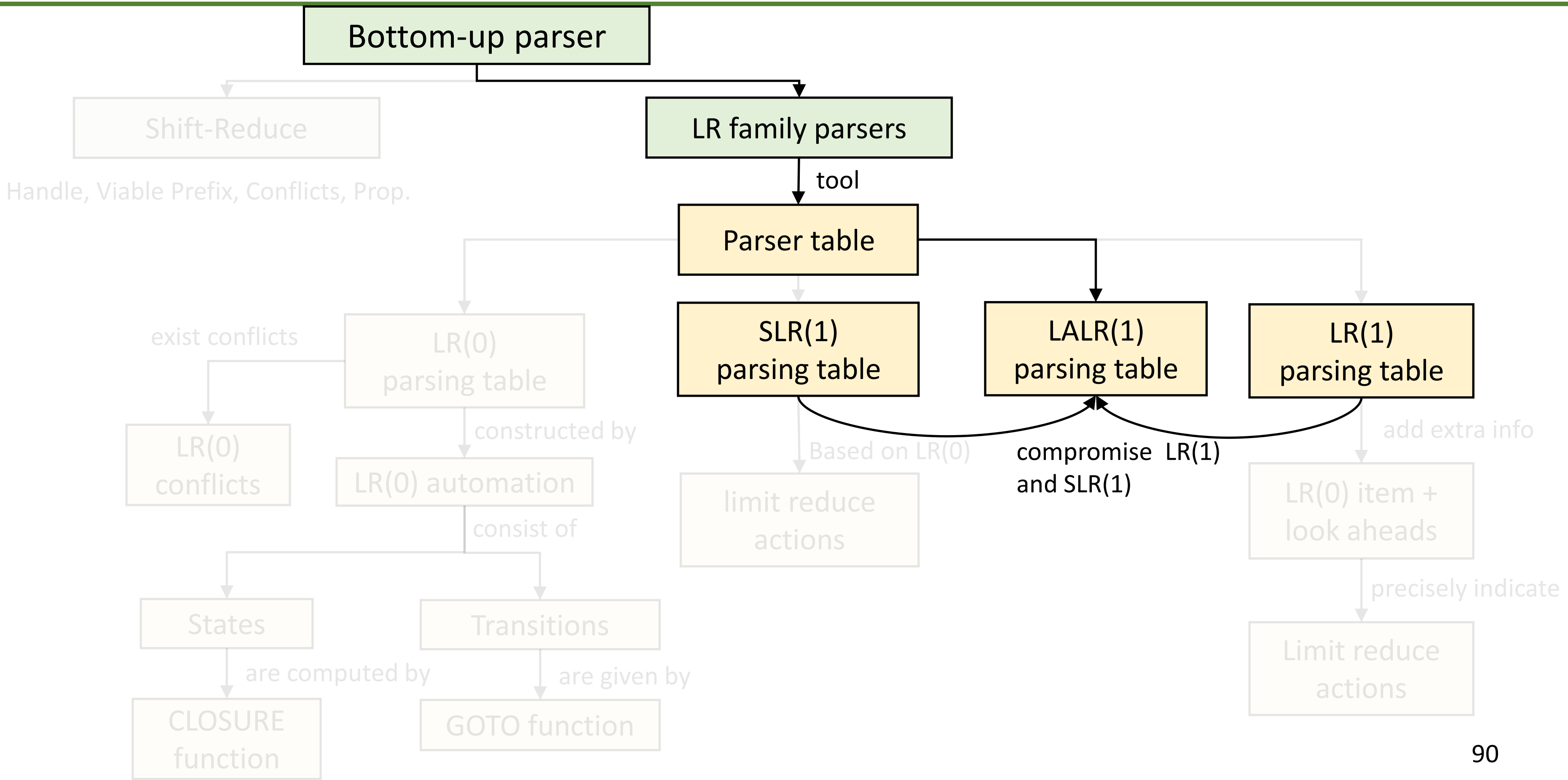


Stat e	Action			Goto	
	a	b	$	S	X
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

LR(1) 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 → u·xv, a] in the set, with terminal x, there is no item in the set of form [B → v·, 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 → u·, a] and [B → v·, a]
    - This translates to no reduce-reduce conflict on any state

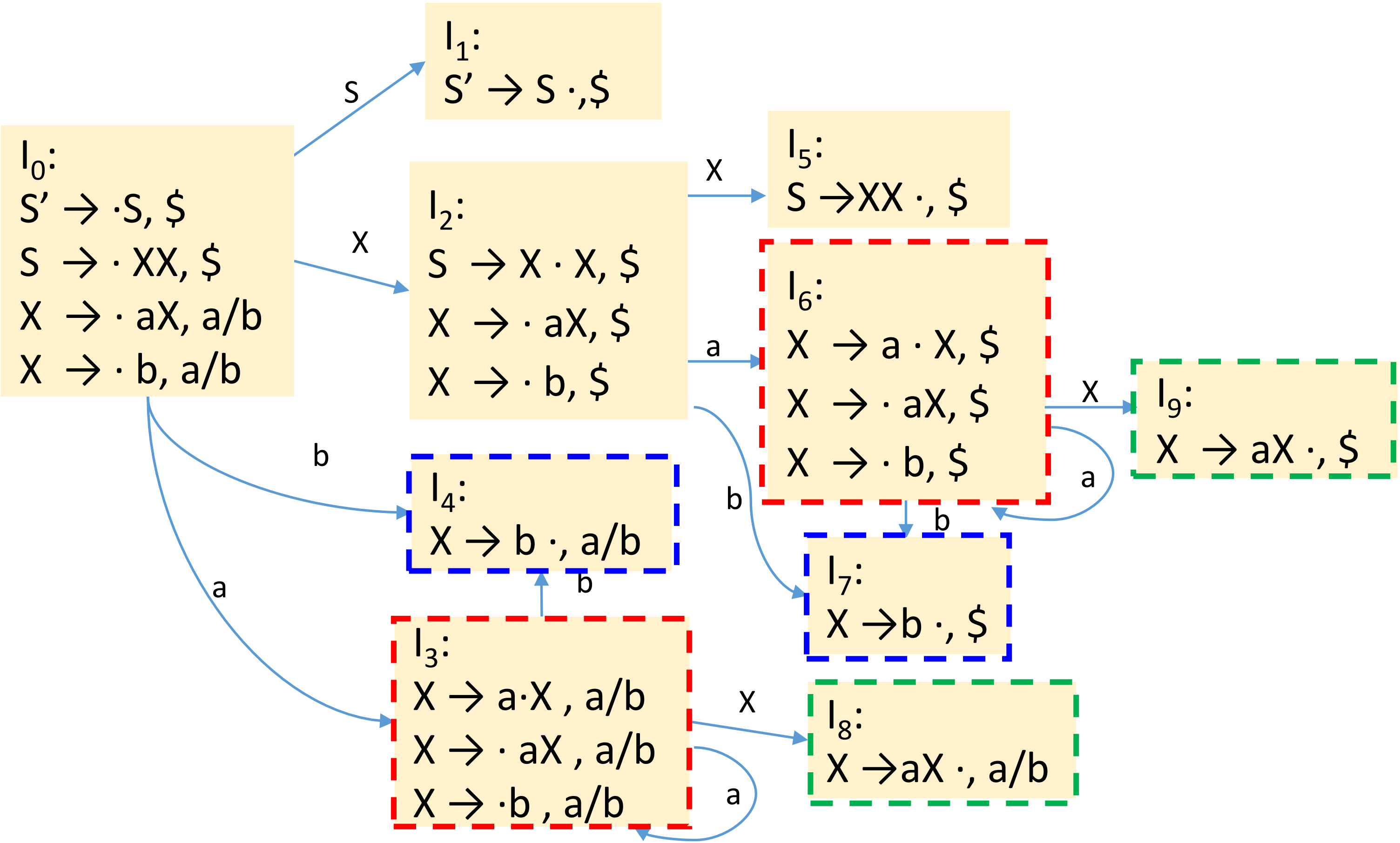
Contents


Bottom-up parser
Shift-Reduce
LR family parsers
Handle, Viable Prefix, Conflicts, Prop.
tool
Parser table
exist conflicts
LR(0) parsing table
SLR(1) parsing table
LALR(1) parsing table
LR(1) parsing table
LR(0) conflicts
constructed by
Based on LR(0)
compromise LR(1) and SLR(1)
add extra info
LR(0) automation
limit reduce actions
LR(0) item + look aheads
consist of
precisely indicate
States
Transitions
Limit reduce actions
are computed by
are given by
CLOSURE function
GOTO function

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) lookaheads
  - Similar states: have same number of items, the core of each item is identical, and they differ only in their lookahead sets

Example



I0:
S' → ·S, $
S → · XX, $
X → · aX, a/b
X → · b, a/b
I1:
S' → S ·,$
I2:
S → X · X, $
X → · aX, $
X → · b, $
I5:
S →XX ·, $
I6:
X → a · X, $
X → · aX, $
X → · b, $
I9:
X → aX ·, $
I4:
X → b ·, a/b
I7:
X →b ·, $
I3:
X → a·X , a/b
X → · aX , a/b
X → ·b , a/b
I8:
X →aX ·, a/b
S
X
X
a
b
X
a
b
b
a
b
X
a

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.)

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		

LR(1)

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		

LALR(1)

Grammar

(0) S’ -> S

(1) S -> XX

(2) X -> aX

(3) X -> b

Merge Effects[合并效果]

- Merging of states can introduce **conflicts**[引入归约-归约冲突]
  - 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
  - Can introduce reduce-reduce (r-r) conflicts
    - LR was introduced to split the Follow set on reduce action
    - Merging reverts the splitting
- **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 -> α·, a] reduce on input a

  [B -> β.aγ, 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item]

- Because:
  - Cores must be the same for S_i and S_j, and thus one of them must contain [A -> α·, a]
  - and it must have an item [B -> β.aγ, c] for some 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

中山大學 SUN YAT-SEN UNIVERSITY

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

S' –> S
S –> aBc | bCc | aCd | bBd B –> e
C –> e

I_{69}:
C → e·, c/d
B → e·, d/c

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

I_0: S' -> •S, $
S -> •aBc, $
S -> •bCc, $
S -> •aCd, $
S -> •bBd, $

I_1: S' -> S•, $

I_2: S -> a•Bc, $
S -> a•Cd, $
B -> •e, c
C -> •e, d

I_3: S -> b•Cc, $
S -> b•Bd, $
C -> •e, c
B -> •e, d

I_4: S -> aB•c, $

I_5: S -> aC•d, $

I_6: B -> e•, c
C -> e•, d

I_7: S -> bC•c, $

I_8: S -> bB•d, $

I_9: B -> e•, d
C -> e•, c

I_{10}: S -> aBc•, $

I_{11}: S -> aCd•, $

I_{12}: S -> bCc•, $

I_{13}: S -> bBd•, $

https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/140%20LALR%20Parsing.pdf

Example: Error Delay

中山大學
SUN YAT-SEN UNIVERSITY

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' -> S

(1) S -> XX

(2) X -> aX

(3) X -> b

Input

aab$

State S_0

Symbol \$ aab\$

State $S_0\ S_3$

Symbol \$ a ab\$

State $S_0\ S_3\ S_3$

Symbol \$ a a b\$

State $S_0\ S_3\ S_3\ S_4$

Symbol \$ a a b \$

Example: Error Delay(cont.)

Grammar

(0) S' -> S (1) S -> XX (2) X -> aX (3) X -> 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	$

LALR Table Construction[解析表构建]

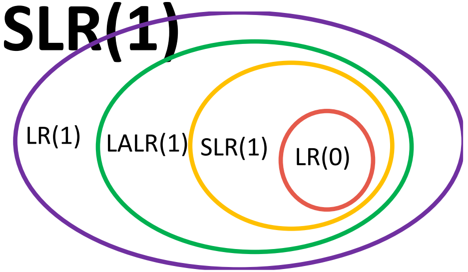
- LALR(1) parsing table is built from the configuration sets in the same way as LR(1)[同样方法构建的项目集]
  - The lookaheads determine where to place reduce actions
  - If there are no mergable states, the LALR(1) table will be identical to the LR(1) table and we gain nothing[退化为LR(1)]
  - Usually, there will 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

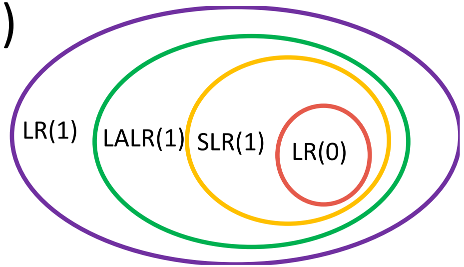
LALR(1) Grammars

- 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: LR > LALR > SLR
  - Number of states in the table: LR > LALR = SLR



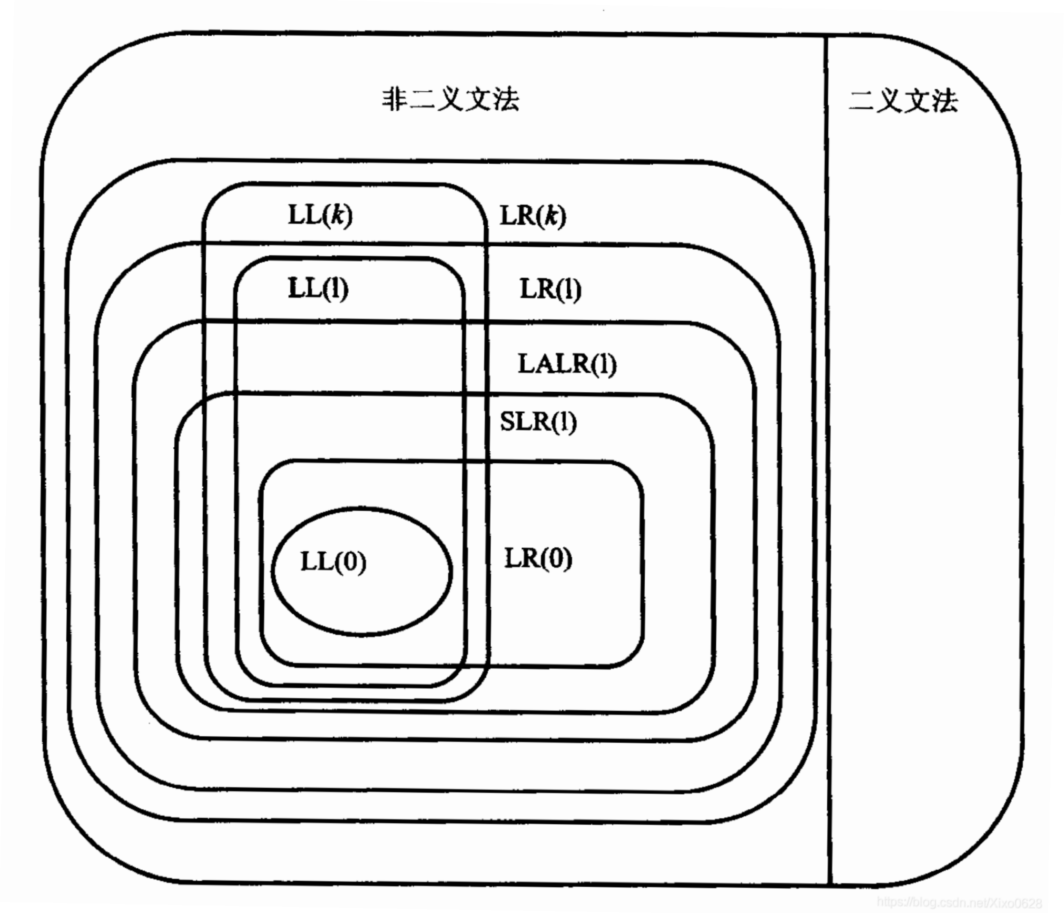
- If a grammar G is LR but is not SLR
  - cannot simply rely on the Follow set to resolve conflicts
    - Need more precise decision on reduction -- lookahead
  - LR grammars have a large parsing table with many states, we can merge some of the states, i.e., LALR
    - No Confliction after merge --> G is LALR
      - The confliction in SLR is resolved.
      - LALR have the same number of sates with SLR, but less reduction actions
    - If no states can be merged, then LALR=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 lookaheads, that is, LALR=SLR

LL vs. LR Parsing (LL < LR)

- LL(k) parser, each expansion A -> α 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 -> α· 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]



Summary of Syntax analysis

- 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, δ)
  - 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 reserve 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?

- Parser
  - Top-down parser
    - RD-backtrack parser
    - Predictive parser
  - Bottom-up parser

Summary of Syntax analysis

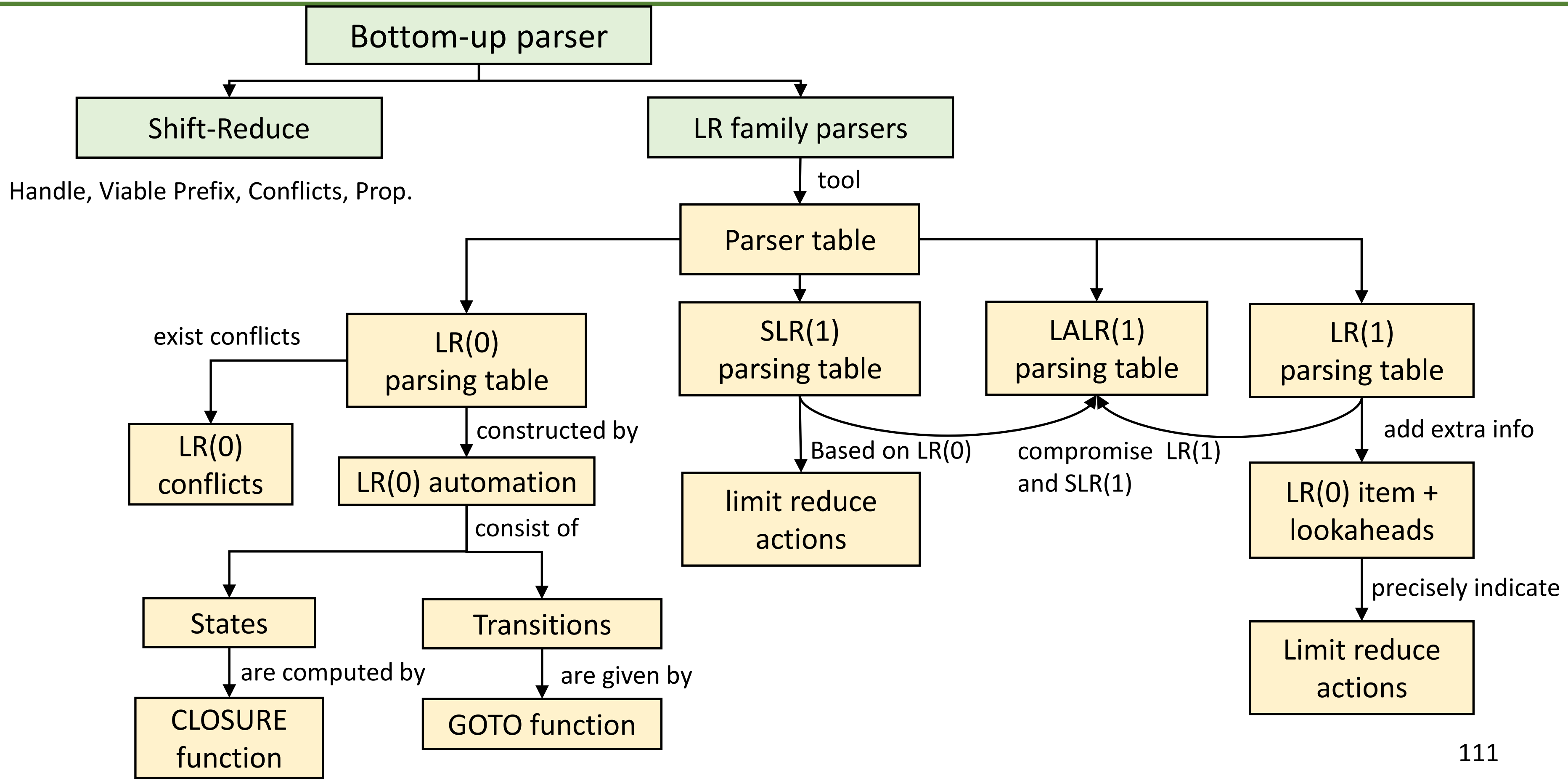
中山大學 SUN YAT-SEN UNIVERSITY

- 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) [LL(1)/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 [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LL(1)语法分析实现]
    - Complexity of LL(1) [LL(1)的时间与空间复杂度]

Summary of Syntax analysis

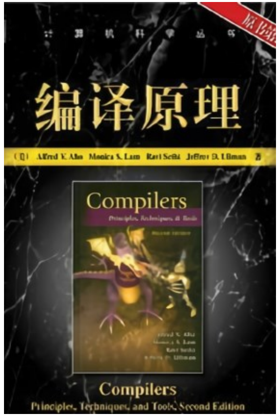
- Bottom-up[自顶向下分析]
  - Mainly two actions: Shift and Reduce
  - LR Parser
    - Table driven, efficient
- Table-driven LR parsers
  - For components: input buffer, stack, parse table, parser driver
  - Action (shift or reduce) is performed based on the top of the stack
    - Symbols and their associated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

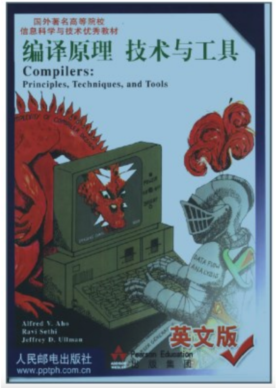
Summary


Bottom-up parser
Shift-Reduce
LR family parsers
Handle, Viable Prefix, Conflicts, Prop.
tool
Parser table
exist conflicts
LR(0) parsing table
SLR(1) parsing table
LALR(1) parsing table
LR(1) parsing table
LR(0) conflicts
constructed by
LR(0) automation
Based on LR(0)
compromise LR(1) and SLR(1)
add extra info
limit reduce actions
LR(0) item + lookaheads
consist of
States
Transitions
precisely indicate
are computed by
are given by
CLOSURE function
GOTO function
Limit reduce actions

Further Reading

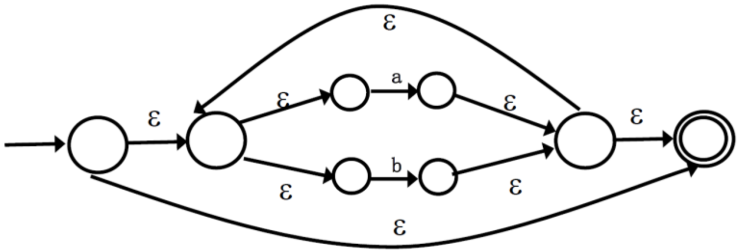
- Dragon Book, 2nd Edition (DBv2)
  - ◆ Comprehensive Reading:
    - ◻ Section 4.5- 4.7 for the Bottom-Up parsing.
    - ◻ Section 4.8.1-4.8.2 on ambiguities in LR parsing.
  - ◆ Skip Reading:
    - ◻ Section 4.8.3 on error recovery in LR parsing.
- Dragon Book, 1st Edition (DBv1)
  - ◆ Comprehensive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





Exercise 2

- 正则表达式可以表达语言$L = \{xnyzn, 1 \le n \le 10\}$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$L = \{x^n yz^n, \mathrm{n} > 0\}$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| 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



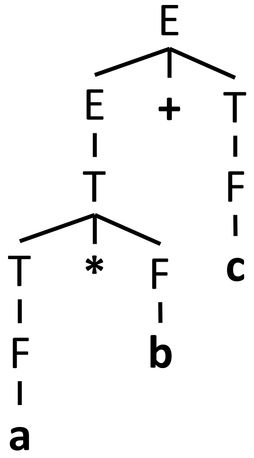
Exercise 2

- 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 $A \to \alpha \mid \beta$:
    - $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$.
    - If $\beta \Rightarrow \varepsilon$, then $FIRST(\alpha) \cap FOLLOW(A) = \emptyset$.
- 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⇒aBb⇒ 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 → T | E+T; T → F | T*F; F → (E) | **a** | **b** | **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 → A
  - A → ε
  - A → bbA

	b	**$**
S		
A		

- FIRST & Follow:
  - S → A: FIRST(α) = {b, ε}
  - A → ε: FIRST(α) = {ε}
  - A → bbA: FIRST(α) = {b}
  - FOLLOW(S) = {$}
  - FOLLOW(A) = {$}

	b	**$**
S	S → A	
A	A → bbA	

use FIRST

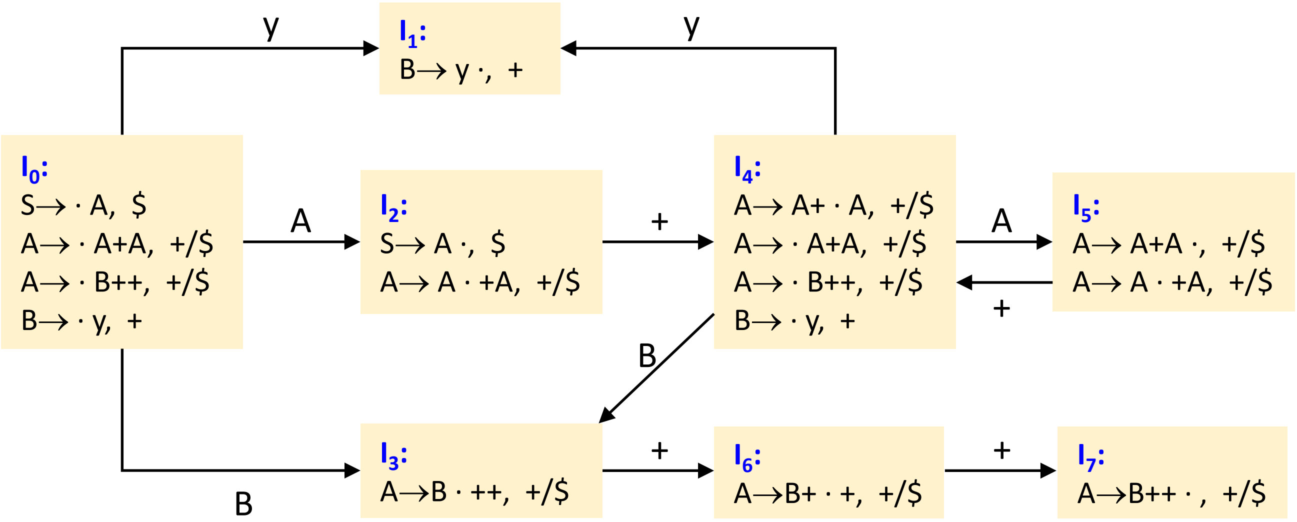
- Parse Table:

	b	**$**
S	S → A	S → A
A	A → bbA	A → ε

use FOLLOW

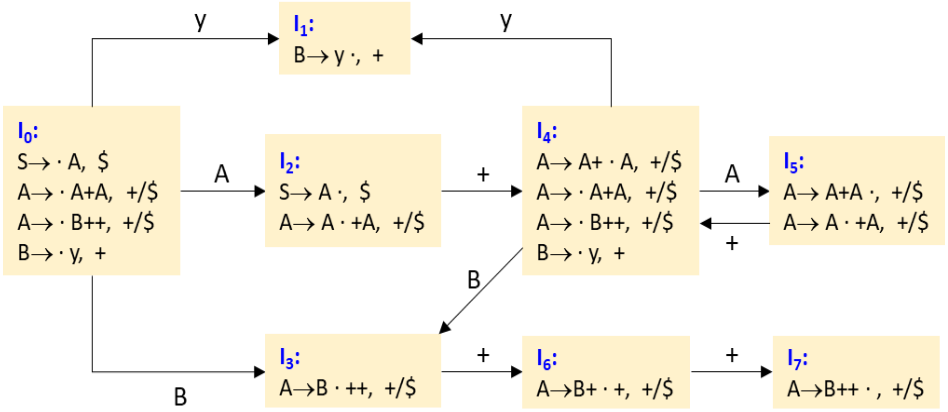
Exercise 2

- 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 → A
  - A → A + A | B + +
  - B → y



Exercise 2

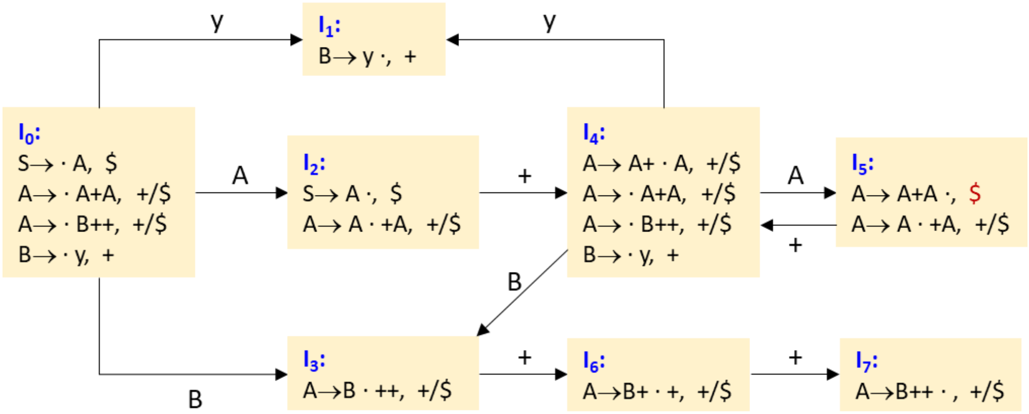
- 给定一下文法G(S)，请构建识别活前缀的DFA与**LR(1)**分析表
  - (1) S → A
  - (2) A → A + A
  - (3) A → B + +
  - (4) B → y



S	ACTOION			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)



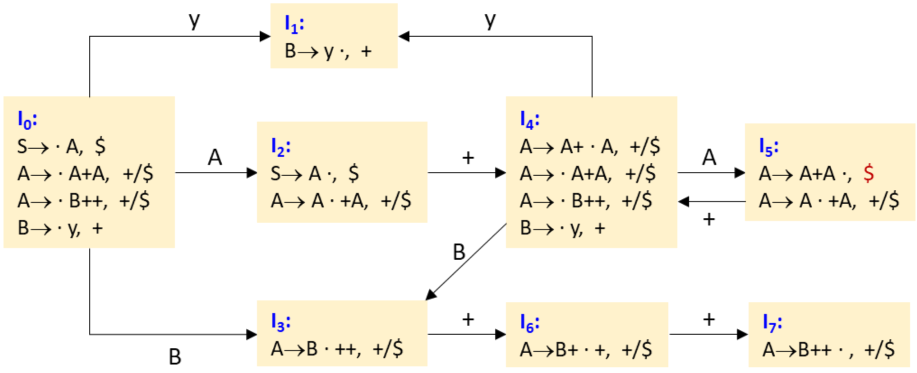
S	ACTOION			GOTO	
	y	+	$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		s4	r2		
6		s7			
7		r3	r3		

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 → A; (2) A → A + A; (3) A → B + +; (4) B → y;

Stack	Input	ACTION	GOTO
0 $	y++$		



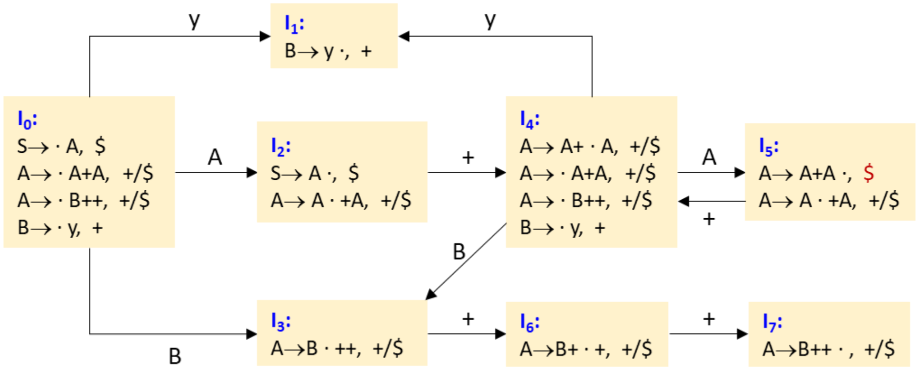
S	ACTOION			GOTO	
	y	+	$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		s4	r2		
6		s7			
7		r3	r3		

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 → A; (2) A → A + A; (3) A → B + +; (4) B → y;

Stack		Input	ACTION	GOTO
0	$	y++$	s1	



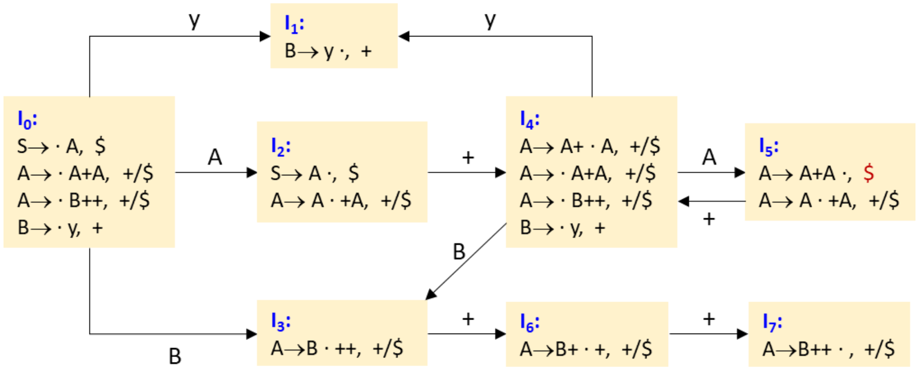
S	ACTOION			GOTO	
	y	+	$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		s4	r2		
6		s7			
7		r3	r3		

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 → A; (2) A → A + A; (3) A → B + +; (4) B → y;

Stack		Input	ACTION	GOTO
0	$	y++$	s1	
01	$y	++$		



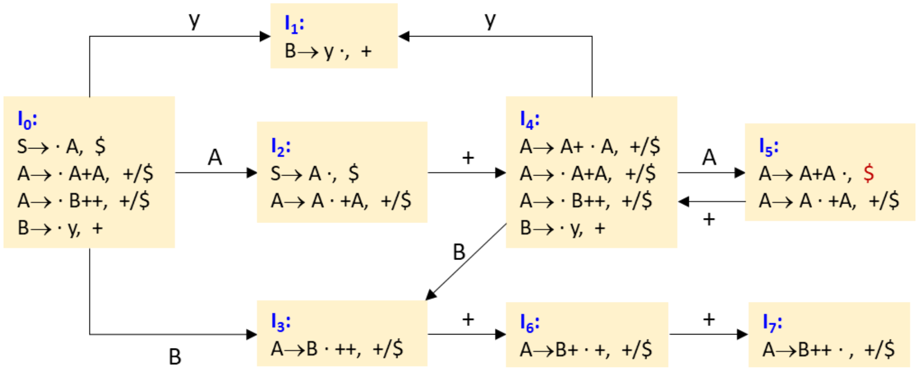
S	ACTOION			GOTO	
	y	+	$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		s4	r2		
6		s7			
7		r3	r3		

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 → A; (2) A → A + A; (3) A → B + +; (4) B → y;

Stack		Input	ACTION	GOTO
0	$	y++$	s1	
01	$y	++$	r4(B → y)	



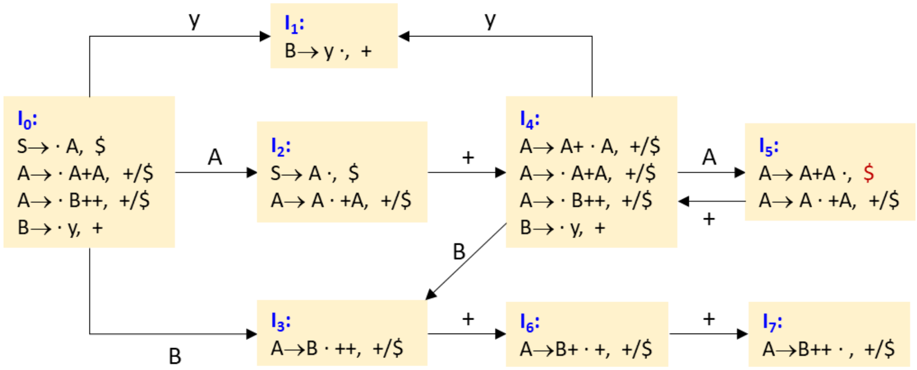
S	ACTOION			GOTO	
	y	+	$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		s4	r2		
6		s7			
7		r3	r3		

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 → A; (2) A → A + A; (3) A → B + +; (4) B → y;

Stack		Input	ACTION	GOTO
0	**$**	**y++$**	**s1**	
01	$y	++$	r4(B → y)	
0	$B	++$		3



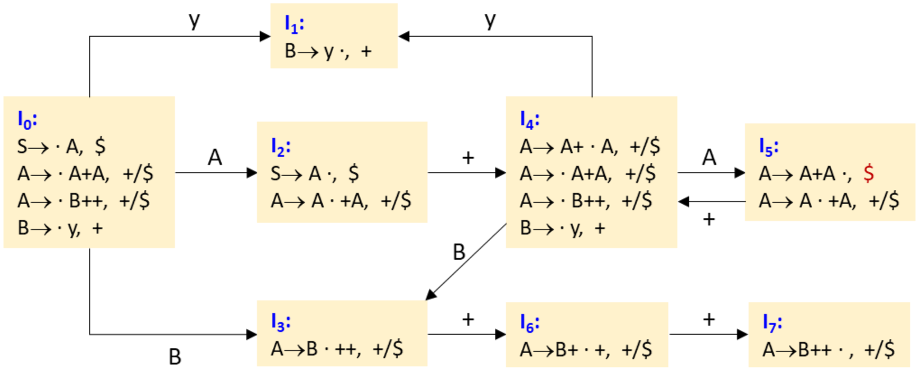
S	ACTOION			GOTO	
	y	+	$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		s4	r2		
6		s7			
7		r3	r3		

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 → A; (2) A → A + A; (3) A → B + +; (4) B → y;

Stack		Input	ACTION	GOTO
0	$	y++$	s1	
01	$y	++$	r4(B → y)	
0	$B	++$		3
03	$B	++$	s6	



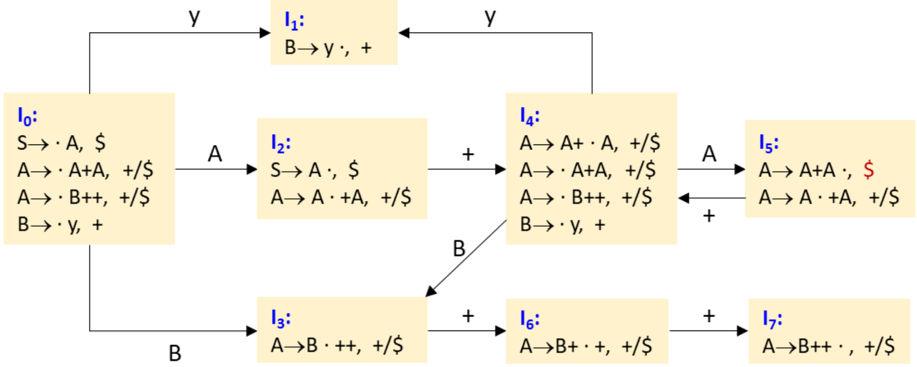
S	ACTOION			GOTO	
	y	+	$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		s4	r2		
6		s7			
7		r3	r3		

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 → A; (2) A → A + A; (3) A → B + +; (4) B → y;

Stack		Input	ACTION	GOTO
0	$	y++$	s1	
01	$y	++$	r4(B → y)	
0	$B	++$		3
03	$B	++$	s6	
036	$B+	+$	s7	



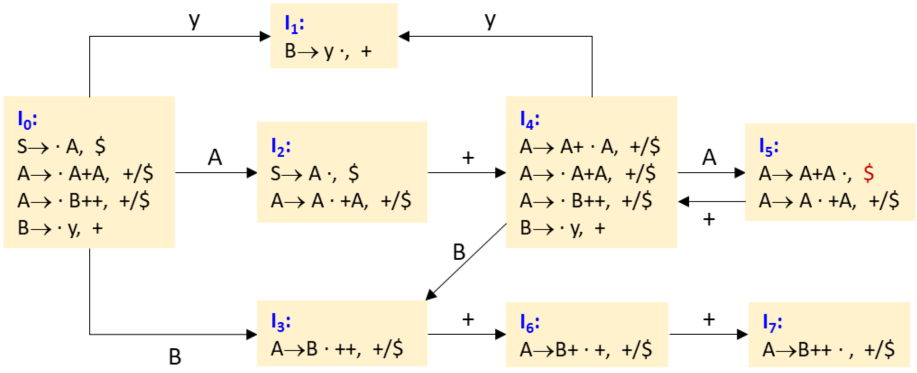
S	ACTOION			GOTO	
	y	+	$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		s4	r2		
6		s7			
7		r3	r3		

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 → A; (2) A → A + A; (3) A → B + +; (4) B → y;

Stack		Input	ACTION	GOTO
0	**$**	**y++$**	**s1**	
01	$y	++$	r4(B → y)	
0	$B	++$		3
03	$B	++$	s6	
036	$B+	+$	s7	
0367	$B++	$	r3(A → B + +)	



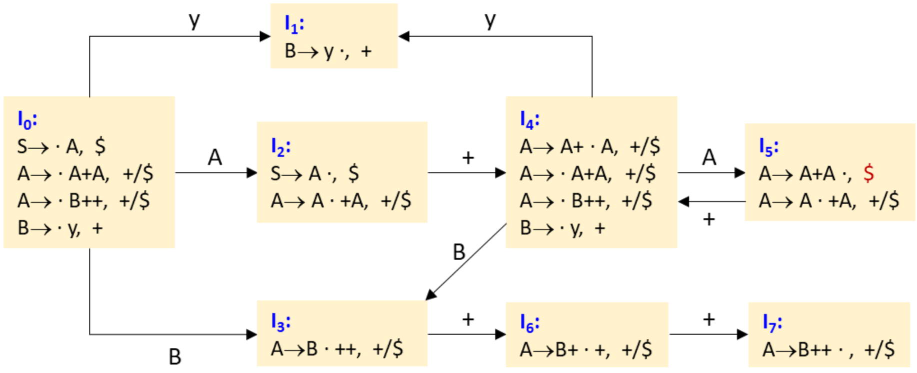
S	ACTOION			GOTO	
	y	+	$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		s4	r2		
6		s7			
7		r3	r3		

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 → A;　(2) A → A + A;　(3) A → B + +;　(4) B → y;

Stack		Input	ACTION	GOTO
0	**$**	**y++$**	**s1**	
01	$y	++$	r4(B → y)	
0	$B	++$		3
03	$B	++$	s6	
036	$B+	+$	s7	
0367	$B++	$	r3(A → B + +)	
0	$A	$		2



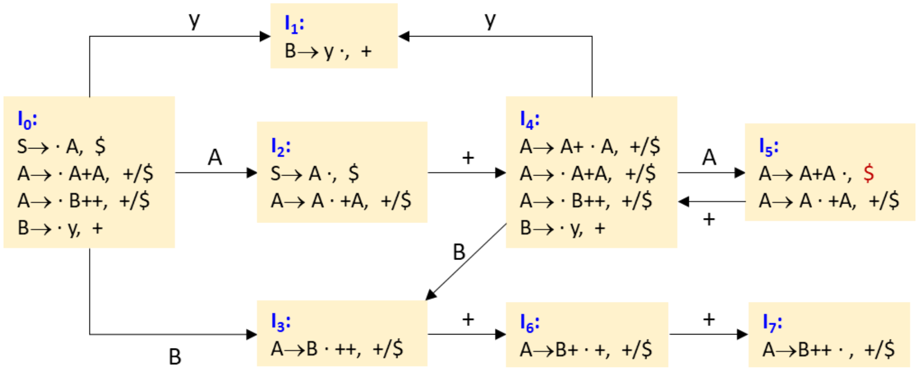
S	ACTOIN			GOTO	
	y	+	$	A	B
0	s1			2	3
1		r4			
2		s4	acc		
3		s6			
4	s1			5	3
5		s4	r2		
6		s7			
7		r3	r3		

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 → A; (2) A → A + A; (3) A → B + +; (4) B → y;

Stack		Input	ACTION	GOTO
0	$	y++$	s1	
01	$y	++$	r4(B → y)	
0	$B	++$		3
03	$B	++$	s6	
036	$B+	+$	s7	
0367	$B++	$	r3(A → B + +)	
0	$A	$		2
02	$A	$	acc	



S	ACTOION			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		