



中山大學  
SUN YAT-SEN UNIVERSITY

计算机学院（软件学院）  
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

# Compilation Principle 编译原理

---

## 第14讲：语法分析(10)

张献伟

[xianweiz.github.io](https://xianweiz.github.io)

DCS290, 4/18/2024



中山大學  
SUN YAT-SEN UNIVERSITY



# Review Questions

- Why LR(0) is of limited usage?

No lookahead, easy to have shift-reduce and reduce-reduce conflicts

- How does SLR(1) improve LR(0)?

Lookahead using the FOLLOW set when reduce happens

- At high level, how does LR(1) improve SLR(1)?

Splitting FOLLOW set (i.e., splitting states) to enforce reduce to consider not only the stack top

- How does LR(1) split the states?

Add lookaheads to each item, i.e., LR(1) item = LR(0) item + lookahead

- How to understand the item  $[A \rightarrow u\bullet, a/b/c]$

Reduce using  $A \rightarrow u$ , ONLY when the next input symbol is  $a/b/c$

# LR(1) Parsing

- 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
  - Many differ only in their lookahead components[仅展望不同]
  - 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 increased GOTO and ACTION table sizes

$S' \rightarrow \cdot S$

LR(0)

3

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

LR(1)

# LR(1) Construction

- Configuration sets

- Sets construction are essentially the same with SLR, but differing on Closure() and Goto()
  - Because we must respect the lookahead

- Closure()

- For 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  $I$ , if
  - $b \in \text{FIRST}(va)$  and  $[B \rightarrow \cdot w, b]$  is not already in  $I$
- Lookahead symbols are the **FIRST(va)**, which are what can follow B
  - $v$  can be nullable

(0)  $S' \rightarrow S$

(1)  $S \rightarrow BB$

(2)  $B \rightarrow aB$

(3)  $B \rightarrow b$

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



$I_0$ :  
 $S' \rightarrow \cdot S, \$$   
 $S \rightarrow \cdot BB, \text{First}(\epsilon \$)$   
 $B \rightarrow \cdot aB, \text{First}(B \$)$   
 $B \rightarrow \cdot b, \text{First}(B \$)$



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

# LR(1) Construction (cont.)

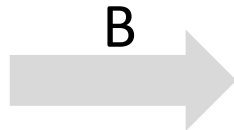
- **Goto(I, X)**

- For item  $[A \rightarrow u \cdot Xv, a]$  in  $I$ ,  $\text{Goto}(I, X) = \text{Closure}([A \rightarrow uX \cdot v, a])$
- 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  $\text{Closure}([S' \rightarrow \cdot S, \$])$
- Construct configuration sets following  $\text{Goto}(I, X)$
- Repeat until no new sets can be added

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

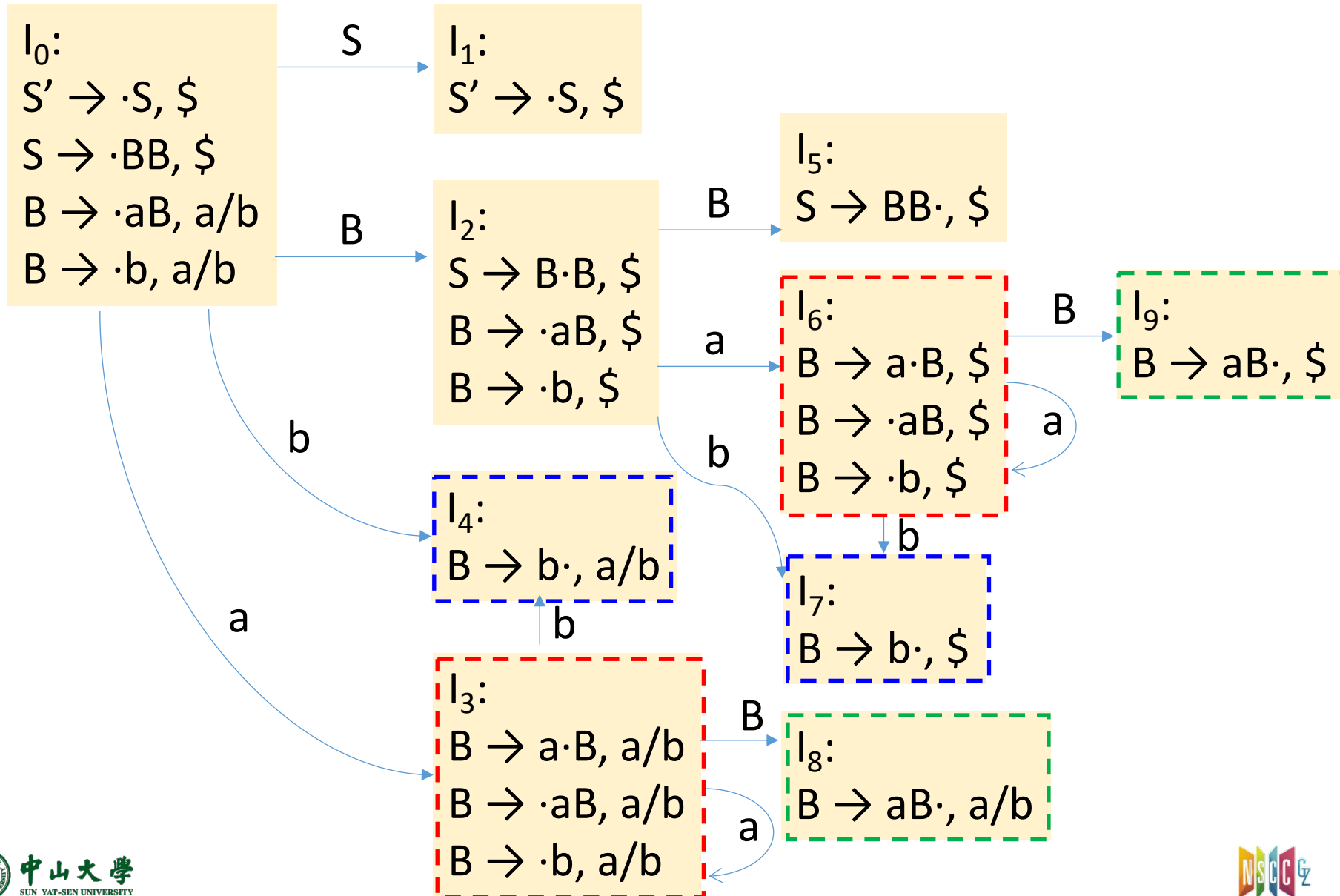


$I_2$ :  
 $S \rightarrow B \cdot B, \$$   
 $B \rightarrow \cdot aB, \text{First}(\epsilon\$)$   
 $B \rightarrow \cdot b, \text{First}(\epsilon\$)$

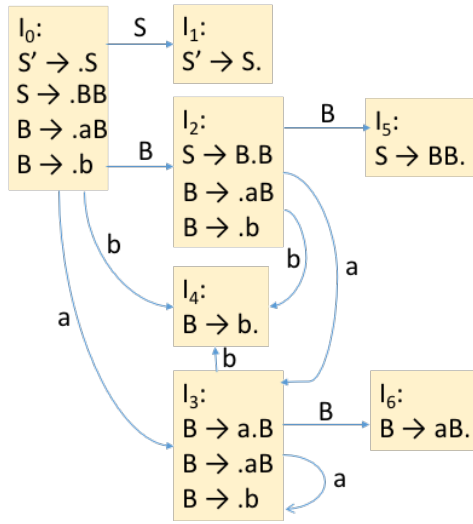


$I_2$ :  
 $S \rightarrow B \cdot B, \$$   
 $B \rightarrow \cdot aB, \$$   
 $B \rightarrow \cdot b, \$$

# The Example



# Example (cont.)



LR(0)

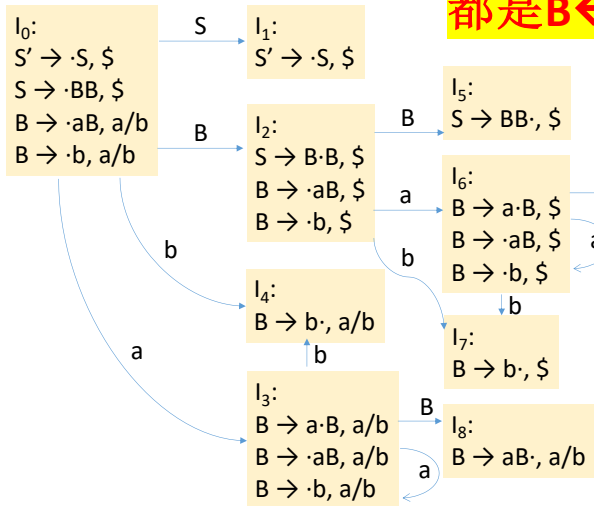
String: bab

#bab\$ => b#ab\$ => B#ab\$ => Ba#b\$  
 0                      0 4                      0 2                      0 2 3  
 => Bab#\$ => BaB#\$ => BB#\$ => S#\$  
          0 2 3 4                      0 2 3 6                      0 2 5                      0 1

String: bab

#bab\$ => b#ab\$ => B#ab\$ => Ba#b\$  
 0                      0 4                      0 2                      0 2 6  
 => Bab#\$ => BaB#\$ => BB#\$ => S#\$  
          0 2 3 7                      0 2 3 9                      0 2 5                      0 1

都是B ← b归约



LR(1)

String: abb

#abb\$ => a#bb\$ => ab#b\$ => aB#b\$  
 0                      0 3                      0 3 4                      0 3 8  
 => B#b\$ => Bb#\$ => BB#\$ => S#\$  
          0 2                      0 2 7                      0 2 5                      0 1

都是B ← aB归约

# LR(1) Parse Table[解析表]

---

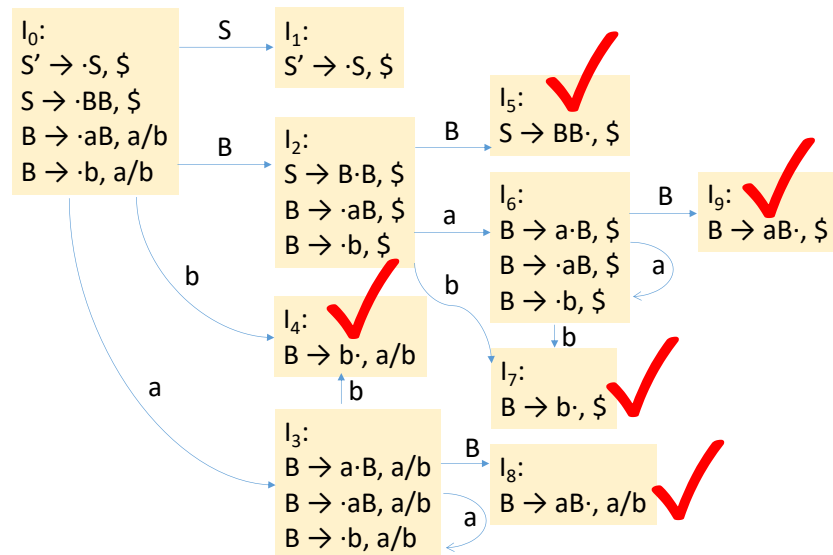
- Shift[移进]
  - Same as LR(0) and SLR(1)
  - Don't care the lookahead symbols
- Reduce[归约]
  - Don't use FOLLOW set (too coarse-grain[太粗粒度])
  - Reduce only if input matches lookahead for item
- ACTION and GOTO[表格]
  - If  $[A \rightarrow \alpha \cdot a \beta, b] \in S_i$  and  $\text{goto}(S_i, a) = S_j$ ,  $\text{Action}[i, a] = s_j$ 
    - Shift  $a$  and goto state  $j$
    - Same as SLR(1)/LR(0)
  - If  $[A \rightarrow \alpha \cdot, a] \in S_i$ ,  $\text{Action}[i, a] = r[R]$ 
    - Reduce using  $R: A \rightarrow \alpha$  if input matches  $a$
    - For SLR, reduced if input matches  $\text{FOLLOW}(A)$



# Example

- (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	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

---

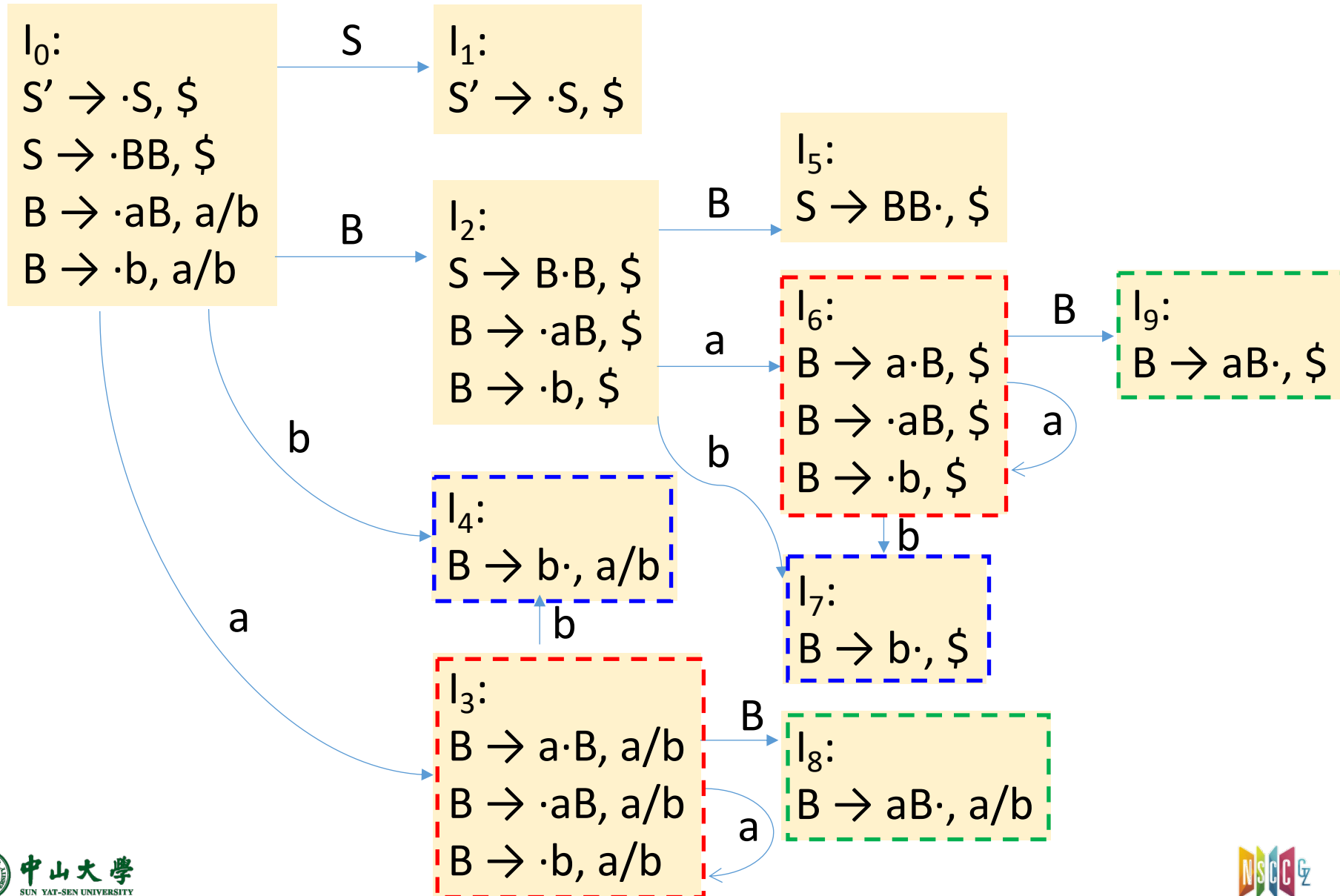
- Every SLR(1) grammar is LR(1), but the LR(1) parser may have **more states** than SLR(1) parser[SLR一定是LR, LR状态更多]
  - LR(1) parser splits states by differing lookaheads, thus it may avoid conflicts that would otherwise result if using the full FOLLOW set
- A grammar is LR(1) if the following two conditions hold for each configurating set[是LR需要满足两个条件]
  - (1) For any item  $[A \rightarrow u \cdot x v, a]$  in the set, with terminal  $x$  (i.e., next shift), there is no item in the set of form  $[B \rightarrow v \cdot, x]$  (i.e., no reduce on  $x$ )
    - 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

# LALR(1) Parser

---

- LR(1) drawbacks[缺点]
  - 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 the number of 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[状态合并]
    - Reduces the #states to the same as SLR(1), but still retains the power of LR(1) lookaheads[LR状态可能过度细分，合并回去，但还是比FOLLOW精细]
  - **Similar states**: have same number of items, the core of each item is identical, and they differ only in their lookahead sets[相似：核心相同，展望不同]

# The 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

$l_3$ :  
 $B \rightarrow a \cdot B, a/b$   
 $B \rightarrow \cdot aB, a/b$   
 $B \rightarrow \cdot b, a/b$

$l_6$ :  
 $B \rightarrow a \cdot B, \$$   
 $B \rightarrow \cdot aB, \$$   
 $B \rightarrow \cdot b, \$$



$l_{36}$ :  
 $B \rightarrow a \cdot B, a/b/\$$   
 $B \rightarrow \cdot aB, a/b/\$$   
 $B \rightarrow \cdot b, a/b/\$$

$l_4$ :  
 $B \rightarrow b \cdot, a/b$

$l_7$ :  
 $B \rightarrow b \cdot, \$$



$l_{47}$ :  
 $B \rightarrow b \cdot, a/b/\$$

$l_8$ :  
 $B \rightarrow aB \cdot, a/b$

$l_9$ :  
 $B \rightarrow aB \cdot, \$$



$l_{89}$ :  
 $B \rightarrow aB \cdot, a/b/\$$

# State Merging (cont.)

State	ACTION			GOTO	
	a	b	\$	S	B
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	B
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: **LALR(1)**

(0)  $S' \rightarrow S$

(1)  $S \rightarrow BB$

(2)  $B \rightarrow aB$

(3)  $B \rightarrow b$

# State Merging (cont.)

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)/SLR(1)

State	ACTION			GOTO	
	a	b	\$	S	B
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	B
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)  $S' \rightarrow S$

(1)  $S \rightarrow BB$

(2)  $B \rightarrow aB$

(3)  $B \rightarrow b$

# Merge Effects[合并效果]

---

- Merging 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
  - We'll see an example



# 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item]

- Because

- Cores must be the same for  $S_i$  and  $S_j$ , and thus one of them must contain  $[A \rightarrow \alpha \cdot, a]$  [这个item总得来自其中一个state吧? ]
- and it must have an item  $[B \rightarrow \beta \cdot a \gamma, c]$  for some  $c$  [否则怎么同core? ]
  - 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

$S' \rightarrow S$

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

$B \rightarrow e$

$C \rightarrow e$

$I_{69}$ :

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

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

next token is c or d,  
reduce to B or C???

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

$I_0$ :  $S' \rightarrow \cdot S, \$$   
 $S \rightarrow \cdot aBc, \$$   
 $S \rightarrow \cdot bCc, \$$   
 $S \rightarrow \cdot aCd, \$$   
 $S \rightarrow \cdot bBd, \$$

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

$I_2$ :  $S \rightarrow a\cdot Bc, \$$   
 $S \rightarrow a\cdot Cd, \$$   
 $B \rightarrow \cdot e, c$   
 $C \rightarrow \cdot e, d$

$I_3$ :  $S \rightarrow b\cdot Cc, \$$   
 $S \rightarrow b\cdot Bd, \$$   
 $C \rightarrow \cdot e, c$   
 $B \rightarrow \cdot e, d$

$I_4$ :  $S \rightarrow aB\cdot c, \$$

$I_5$ :  $S \rightarrow aC\cdot d, \$$

$I_6$ :  $B \rightarrow e\cdot, c$   
 $C \rightarrow e\cdot, d$

$I_7$ :  $S \rightarrow bC\cdot c, \$$

$I_8$ :  $S \rightarrow bB\cdot d, \$$

$I_9$ :  $B \rightarrow e\cdot, d$   
 $C \rightarrow e\cdot, c$

$I_{10}$ :  $S \rightarrow aBc\cdot, \$$

$I_{11}$ :  $S \rightarrow aCd\cdot, \$$

$I_{12}$ :  $S \rightarrow bCc\cdot, \$$

$I_{13}$ :  $S \rightarrow bBd\cdot, \$$

Reduce to B when next token is c

Reduce to C when next token is d

# Example: Error Delay

(0)  $S' \rightarrow S$

(1)  $S \rightarrow BB$

(2)  $B \rightarrow aB$

(3)  $B \rightarrow b$

Input: **aab**\$

State	ACTION			GOTO	
	a	b	\$	S	B
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		

state  $\rightarrow S_0$

symbol  $\rightarrow \$$

aab\$

state  $\rightarrow S_0S_3$

symbol  $\rightarrow \$ a$

ab\$

state  $\rightarrow S_0S_3S_3$

symbol  $\rightarrow \$ a a$

b\$

state  $\rightarrow S_0S_3S_3S_4$

symbol  $\rightarrow \$ a a b$

\$

# Example: Error Delay (cont.)

(0)  $S' \rightarrow S$

(1)  $S \rightarrow BB$

(2)  $B \rightarrow aB$

(3)  $B \rightarrow b$

Input: **aab**\$

State	ACTION			GOTO	
	a	b	\$	S	B
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  $\rightarrow S_0$

symbol  $\rightarrow \$$  aab\$

state  $\rightarrow S_0 S_{36}$

symbol  $\rightarrow \$ a$  ab\$

state  $\rightarrow S_0 S_{36} S_{36}$

symbol  $\rightarrow \$ a a$  b\$

state  $\rightarrow S_0 S_{36} S_{36} S_{47}$

symbol  $\rightarrow \$ a a b$  \$

state  $\rightarrow S_0 S_{36} S_{36} S_{89}$

symbol  $\rightarrow \$ a a B$  \$

state  $\rightarrow S_0 S_{36} S_{89}$

symbol  $\rightarrow \$ a B$  \$

state  $\rightarrow S_0 S_2$

symbol  $\rightarrow \$ B$  \$

# 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[完全就是LR(1)]
  - Usually, there will be states that can be merged and the LALR table will thus have **fewer rows** than LR
- 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[暴力方式]
  - 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)[LALR归约更精细了]
  - A LR(1) grammar may or may not be LALR(1)[LALR合并了状态]
    - Depends on whether merging introduces conflicts
  - A non-SLR(1) grammar may be LALR(1)[LALR能解决SLR冲突]
    - Depends on whether the more precise lookaheads resolve the SLR(1) conflicts
- LALR(1) reaches a good balance between the **lookahead power** and the **table size**
  - Most used variant of the LR family

# LALR Summary[小结]

- LALR(1)是LR(1)和SLR(1)的平衡
  - 文法范围:  $LR > LALR > SLR$
  - 状态数目:  $LR > LALR = SLR$
- 假如一个文法G是LR而非SLR，可能是LALR
  - 非SLR: SLR产生了冲突（依靠FOLLOW集进行归约不够精确）
    - 是LR: 而LR通过精确的lookahead解决了冲突
  - 可能是LALR: LALR对LR进行相似状态合并
    - 若合并后出现了冲突 --> 不是LALR文法
    - 若合并后没有冲突 --> 是LALR文法
      - LALR可以解析文法G，也即解决了SLR原有的冲突
      - 实际上LALR的状态数与SLR相同，但归约动作减少了（也即，对SLR解析表而言，多个移进/归约动作的单元格中的归约被消除了）
    - 如果没有相似状态，则LALR=LR
- 假如一个文法G是SLR
  - 那么G一定也是LR和LALR文法
  - LR的FOLLOW集细分是不必要的，因此LALR合并回了SLR

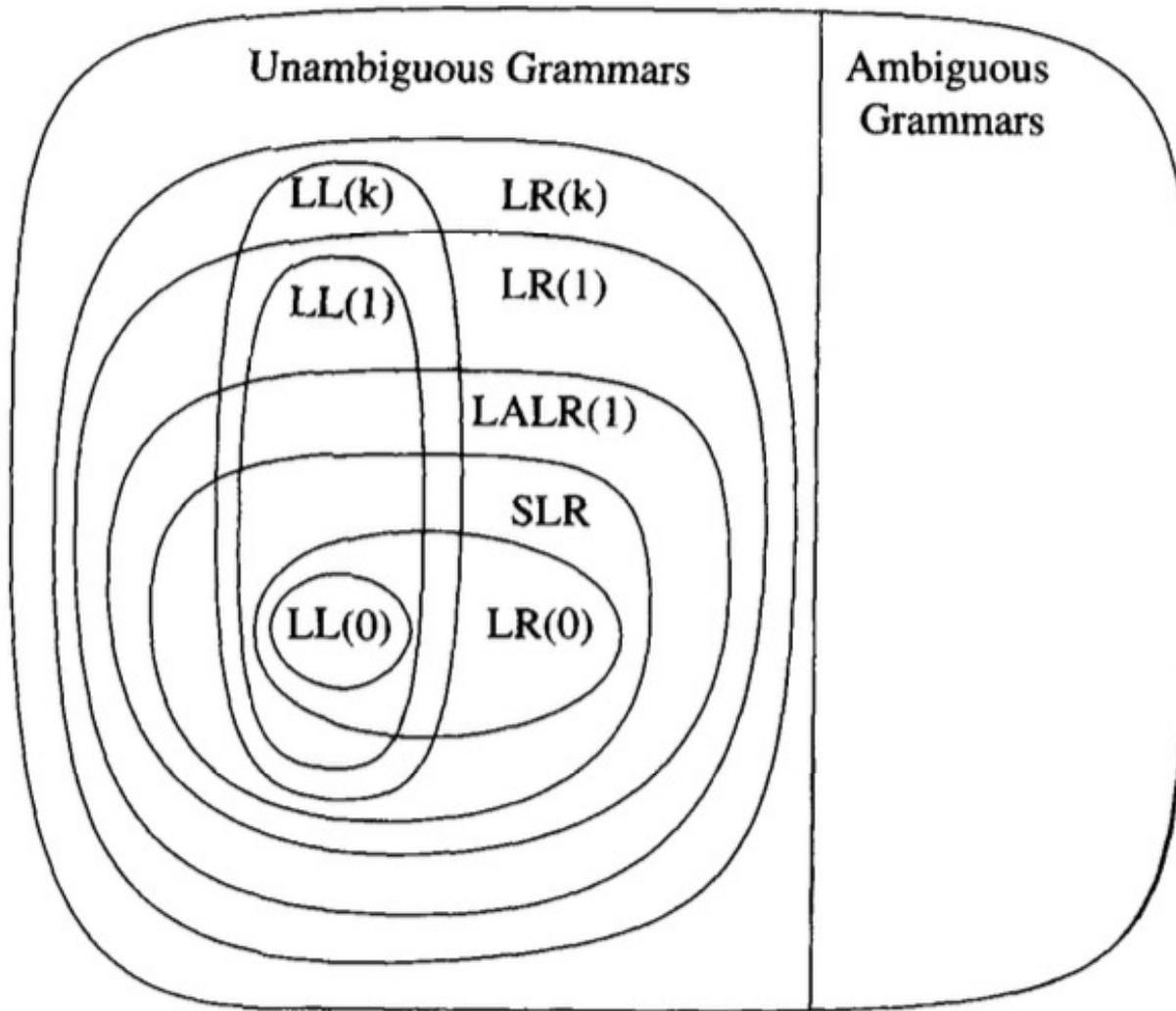


# LL vs. LR Parsing (LL < LR)













---

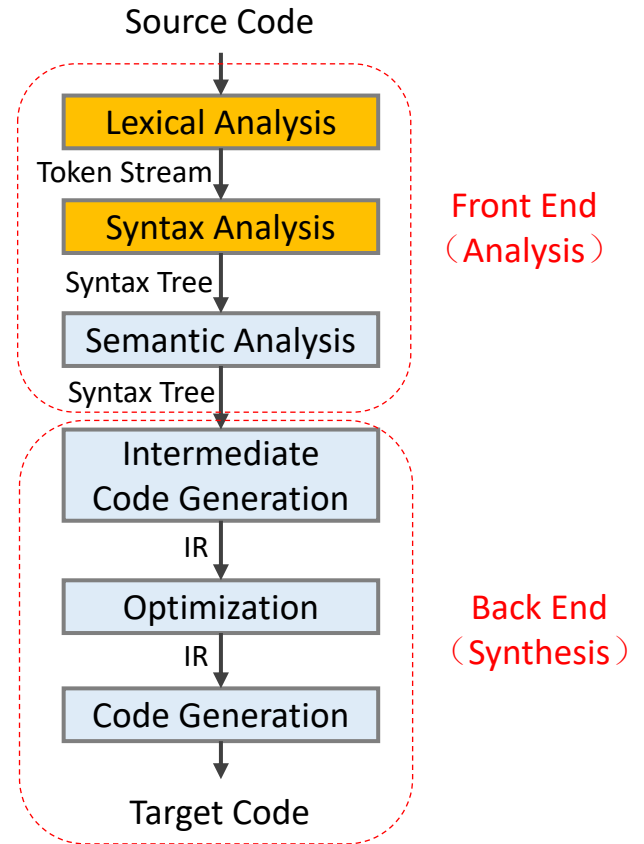
- LL(k) parser, each expansion  $A \rightarrow \alpha$  is decided based on
  - Current non-terminal at the top of the stack[依赖LHS]
    - Which LHS to produce
  - k terminals of lookahead at beginning of RHS[RHS的一点展望]
    - Must **guess** which RHS by peeking at **first few terminals** of RHS
  - 选择依据: LHS + RHS的k个符号 (有限信息)
- LR(k) parser, each production  $A \rightarrow \alpha$  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
  - 选择依据: 整个RHS + LHS后的k个符号 (充足信息)

# Hierarchy of Grammars[文法层级]



# A Long way ...

wk3: 3/14	 Syntax Analysis: Parse, CFG 
wk4: 3/19	 Syntax Analysis: CFG, Parser
wk4: 3/21	 Syntax Analysis: Top-down (1)
wk5: 3/26	 Syntax Analysis: Top-down (2)
wk5: 3/28	 Syntax Analysis: Top-down (3) 
wk6: 4/2	 Syntax Analysis: Bottom-up, LR-parse (1)
wk6: 4/4	Cancelled
wk7: 4/9	 Syntax: LR-parse (2)
wk7: 4/11	 Syntax: LR-parse (3)
wk8: 4/16	 Syntax: LR-parse (4) 
wk8: 4/18	Syntax: LR-parse (5)



# 总结: 语法分析 (1)

---

- 语法分析(Syntax analysis)是编译的第二个阶段
  - 输入: 词法分析产生的token序列
  - 输出: 分析树(parse tree)或抽象语法树(AST)
- 语法指定(Syntax specification)
  - 词法分析使用的RE/FA表达能力不够(e.g., 嵌套结构)
  - 需要使用文法(grammar), 尤其是上下文无关文法(context-free grammar, CFG)
- 文法形式化定义:  $\{T, N, s, \sigma\}$ 
  - T: terminal symbols[终结符] = 词法分析的token, 分析树的叶子节点
  - N: non-terminal symbols[非终结符], 分析树的内部节点
  - s: start symbol[开始符号]
  - $\sigma$ : set of productions[产生式], 形式: LHS  $\rightarrow$  RHS

# 总结: 语法分析 (2)

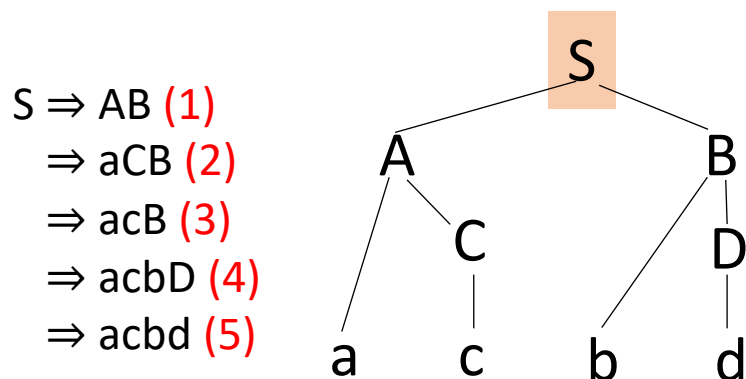
---

- 推导(Derivation)
  - 对产生式的若干次使用 (从LHS到RHS)
    - 从文法开始符号到输入串(input string)
- 归约(Reduce)
  - 推导的逆过程(从RHS到LHS)
    - 从输入串(input string)到文法开始符号
- 分析树(Parse tree)
  - 是推导的图形化表示, 略去了推导中产生式的使用顺序
- 歧义文法(Ambiguous grammar)
  - 某个句子对应多个(最左或最右)分析树
  - 通过指定优先级(precedence)和结合性(associativity)来改写文法以消除歧义

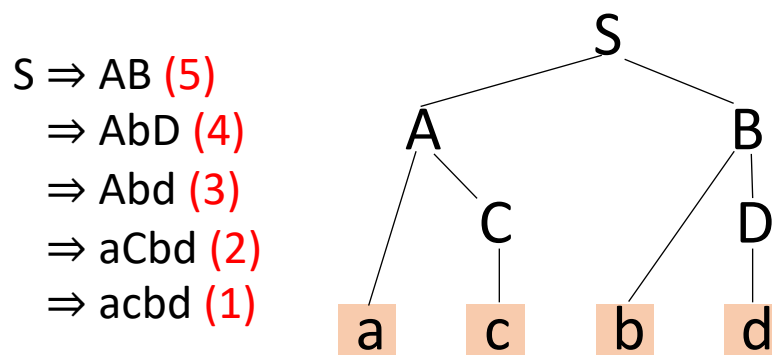
# 总结: 语法分析 (3)

- 语法分析(或解析)就是处理给定文法的输入句子, 构建一个以分析树或抽象语法树表示的推导
  - 自顶向下(Top-down): 从根节点扩展到叶子节点, 每步考虑
    - 替换哪个非终结符?
    - 使用哪个产生式来替换?
  - 自底向上(Bottom-up): 从叶子节点回到根节点
    - 消耗输入token还是归约?
    - 使用哪个产生式来归约?

## Top-down (leftmost derivation)



## Bottom-up (reverse of rightmost derivation)



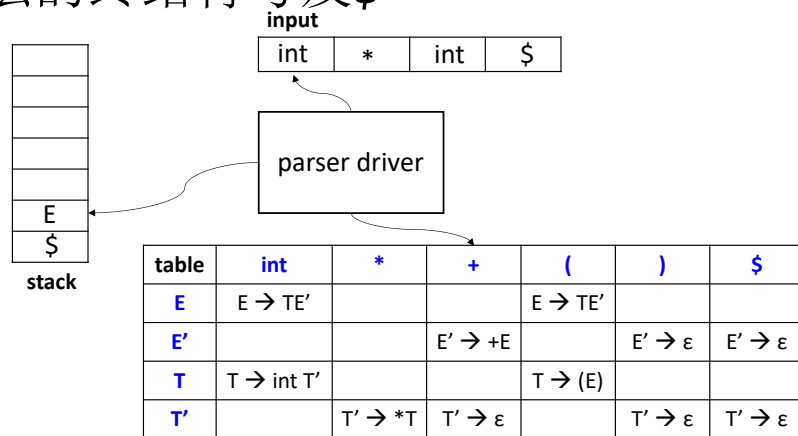
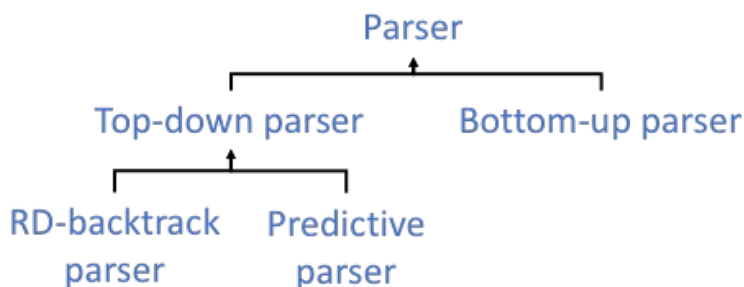
# 总结: 语法分析 (4)

- Top-down分析

- 递归下降分析(Recursive descent): 试错->回溯(backtracking)
  - 消除左递归(Left recursion)
- 预测分析(Predictive): 预测, 无需回溯
  - 消除左递归, 提取左公因子(Left factoring)

- 表驱动的LL(1)分析器

- 四部分: input buffer, stack, parse table, parser driver
- 基于<stack top, current token>来采取操作(expand or match)
- 解析表行为文法的非终结符、列为文法的终结符号及\$
  - 单元格存放一个产生式或空
  - 表格是借助FIRST和FOLLOW集来构建



# 总结: 语法分析 (5)

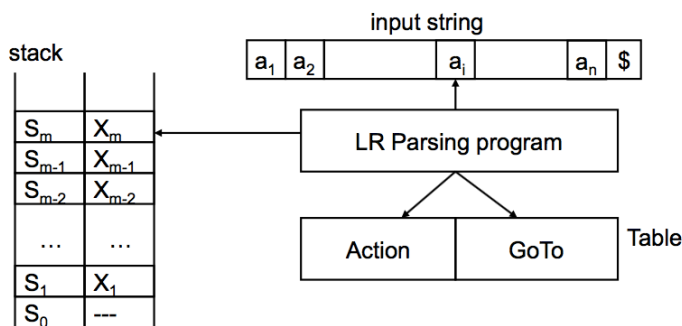
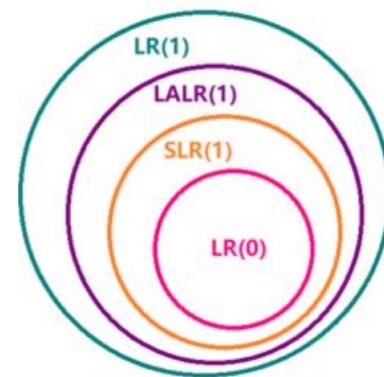
- Bottom-up分析

- 主要有移进(Shift)和归约(Reduce)两个动作
- 实现上主要是LR类型分析器
  - 表格驱动, 高效

- 表驱动的LR分析器

- 四部分: input buffer, stack, parse table, parser driver
- 基于栈顶来采取操作(shift or reduce)
  - 栈保存状态序列和每个状态关联的文法符号
- 解析表包含Action和Goto两个子表
  - 表格是通过识别文法的可能项目集及转换(i.e., DFA)
  - LR(0) -> SLR(1) -> LR(1) -> LALR(1)

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		







中山大學  
SUN YAT-SEN UNIVERSITY

计算机学院（软件学院）  
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

# Compilation Principle 编译原理

---

## 第14讲：语义分析(1)

张献伟

[xianweiz.github.io](https://xianweiz.github.io)

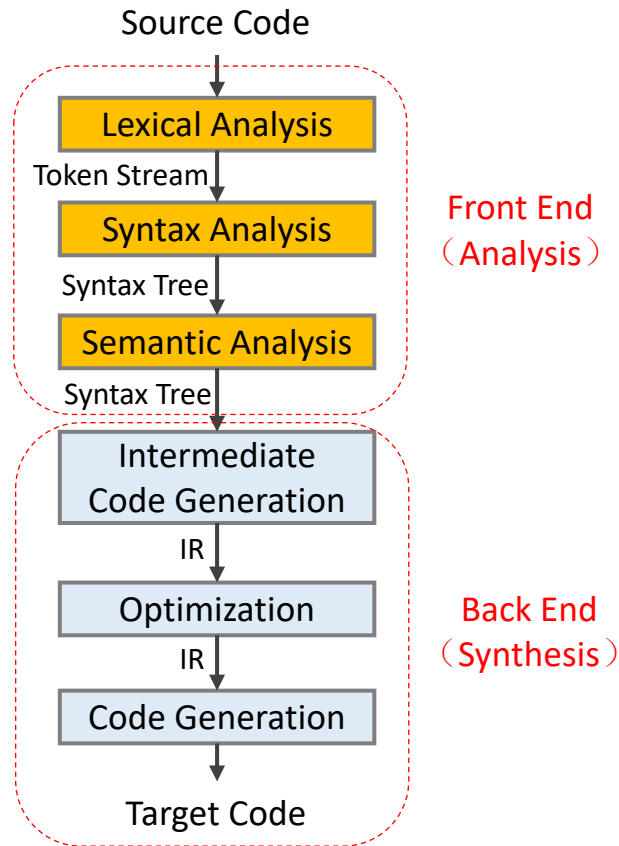
DCS290, 4/18/2024



中山大學  
SUN YAT-SEN UNIVERSITY



# Compilation Phases[编译阶段]



# Compilation Phases (cont.)

- Lexical analysis[词法分析]

- Source code  $\rightarrow$  tokens
- Detects inputs with illegal tokens
- Is the input program **lexically** well-formed?

$x\#y = 1$

- Syntax analysis[语法分析]

- Tokens  $\rightarrow$  parse tree or abstract syntax tree (AST)
- Detects inputs with incorrect structure
- Is the input program **syntactically** well-formed?

$x = 1 \ y = 2$

- Semantic analysis[语义分析]

- AST  $\rightarrow$  (modified) AST + symbol table
- Detects semantic errors (errors in meaning)
- Does the input program has a well-defined **meaning**?

$\text{int } x; y = x(1)$

# Why Semantic Analysis?[语义分析]

- Because programs use **symbols** (a.k.a. identifiers)
  - Identifiers require **context** to figure out the meaning
- Consider the English sentence: “He ate it”
  - This sentence is syntactically correct
  - But it makes sense only in the context of a previous sentence: “Sam bought a pizza.” (what if “Sam bought a car.”?)
- Semantic analysis
  - Associates identifiers with objects they refer to[关联]
    - “He” --> “Sam”
    - “it” --> “pizza”
  - Checks whether identifiers are used correctly[检查]
    - “He” and “it” refer to some object: def-use check
    - “it” is a type of object that can be eaten: type check



# Why Semantic Analysis (cont.)

---

- Semantics of a language is much more difficult to describe than syntax[语义比语法更难描述]
  - Syntax: describes the proper form of the programs[仅形式]
  - Semantics: defines what the programs means (i.e., what each program does when it executes)[到意义]
- Context **cannot** be analyzed using a CFG parser[CFG不能分析上下文信息]
  - Associating IDs to objects require expressing the pattern:  
 $\{wcw \mid w \in (a|b)^*\}$ 
    - The first  $w$  represents the definition of a ID
    - The  $c$  represents arbitrary intervening code
    - The second  $w$  represents the use of the ID

# Semantic Analysis

---

- Deeper check into the source program[对程序进一步分析]
  - Last stage of the front end[前端的最后阶段]
  - Compiler's last chance to reject incorrect programs[最后拒绝机会]
  - Verify properties that aren't caught in earlier phases
    - Variables are declared before they're used[先声明后使用]
    - Type consistency when using IDs[变量类型一致]
    - Expressions have the right types[表达式类型]
      - E.g., string && bool
    - ... ..
- Gather useful info about program for later phases[收集后续信息]
  - Determine what variables are meant by each identifier
  - Build an internal representation of inheritance hierarchies
  - Count how many variables are in scope at each point
  - ... ..