中山大學
SUN YAT-SEN UNIVERSITY

国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

Compilation Principle
编 译 原 理

第10讲：语法分析(7)

张献伟

xianweiz.github.io

DCS290, 04/01/2021

Quiz Questions(1)

- 1. LR(0)在实际中很少使用，为什么？

  不进行任何展望(lookahead)，只利用栈顶信息，极易发生冲突

- 2. SLR(1)是怎么在LR(0)基础上改进的？

  展望(lookahead)一个输入token：归约时使用Follow集

- 3. 怎么理解LR分析中的闭包（Closure）操作?

  归类“期待”意义上等价的项目，对非终结符结合产生式不断添加新的项目

- 4. Top-down和Bottom-up两种解析方式，哪个能覆盖更多的文法? 简要解释。

  Bottom-up，可以处理左递归和共同前缀

- 5. 递归下降（Recursive-descent）和预测（Predictive）分析，哪个更高效？为什么?

  预测，基于lookahead来确定使用哪个产生式，避免了回溯

Quiz Questions(2)

- 6. LR(0)解析表(Parse Table)存放的内容是什么？

  Action和Goto两个子表，Action是对终结符的动作表，Goto是非终结符的跳转表

- 7. LL(k)和LR(k)的主要区别是什么？

  LL(k)是自顶向下，产生最左推导；LR(k)是自底向上，产生最有推导的逆

- 8. $A \rightarrow BC\cdot$ 是什么意思?

  LR中的一个完成项目，在解析进展上，已经识别了BC生成的token，可以使用A → BC进行归约

- 9. 语法分析的输入和输出分别是什么？

  输入：词法分析产生的token序列；输出：语法树

- 10. 语法分析的实现通常是表驱动(table-driven)，这种实现总体包括哪些模块？

  输入缓冲区(Input buffer)，解析表(Parse Table)，栈(Stack)，驱动程序(Driver)

图灵奖'2020

Alfred Vaino Aho

Jeffrey David Ullman

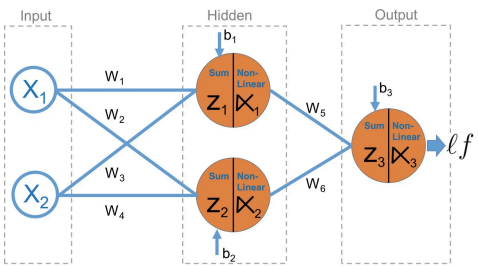
A.M. TURING AWARD HONORS INNOVATORS WHO SHAPED THE FOUNDATIONS OF PROGRAMMING LANGUANGE COMPILERS AND ALGORITHMS

Columbia's Aho and Stanford's Ullman Developed Tools and Seminal Textbooks Used by Millions of Software Programmers around the World

ACM named Alfred Vaino Aho and Jeffrey David Ullman recipients of the 2020 ACM A.M. Turing Award for fundamental algorithms and theory underlying programming language implementation and for synthesizing these results and those of others in their highly influential books, which educated generations of computer scientists. Aho is the Lawrence Gussman Professor Emeritus of Computer Science at Columbia University. Ullman is the Stanford W. Ascherman Professor Emeritus of Computer Science at Stanford University.

Computer software powers almost every piece of technology with which we interact. Virtually every program running our world – from those on our phones or in our cars to programs running on giant server farms inside big web companies – is written by humans in a higher-level programming language and then compiled into

图灵奖'2018&2017&2014



YANN LECUN

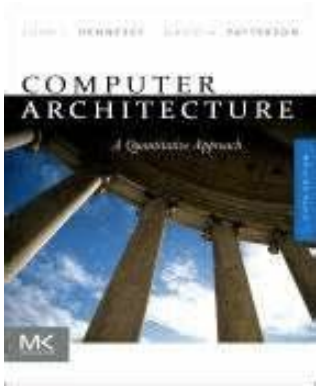
United States – 2018

CITATION

For conceptual and engineering breakthroughs that have made deep neural networks a critical component of computing.

SHORT ANNOTATED BIBLIOGRAPHY | ACM TURING AWARD LECTURE VIDEO | RESEARCH SUBJECTS

Birth: July 8, 1960 in Soisy-sous-Montmorency, France.



DAVID PATTERSON

United States – 2017

CITATION

For pioneering a systematic, quantitative approach to the design and evaluation of computer architectures with enduring impact on the microprocessor industry.

SHORT | ACM TURING | RESEARCH

BIRTH:

MICHAEL STONEBRAKER

United States – 2014

CITATION

For fundamental contributions to the concepts and practices underlying modern database systems.

SHORT ANNOTATED BIBLIOGRAPHY | ACM TURING AWARD LECTURE VIDEO | RESEARCH SUBJECTS | ADDITIONAL MATERIALS | VIDEO INTERVIEW

BIRTH:

October 11, 1943 in Newburyport, Mass.

SLR(1) Parsing

- SLR(1) is a simple improvement over LR(0)
  - LR(0) easily get 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 or multiple reduce items

SLR(1) Grammars[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 table, 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, e.g. 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) Limitations[局限性]

- SLR(1) vs. LR(0)
  - Adding just one token of **lookahead** and using the **Follow** set
  - Greatly expands the class of grammars that can be parsed without conflicts
- When we have a completed item (i.e., dot at the end) such as X –> u·, we know that it is reducible
  - We allow such a reduction whenever 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 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
- **Solution**: consider the context of the state in lookahead

Example

- For input string: id = id, at **I_2** after having reduced id to L
  - Initially, at S_0
  - Move to S_5, after shifting id to stack (S_5 is also pushed to stack)
  - Reduce, and back to S_0, and further GOTO S_2
    - S_5 has a completed item, and next '=' is in Follow(L)
    - S_5 and id are popped from stack, and L is pushed onto stack
    - GOTO(S_0, L) = S_2

```
S' -> S
S -> L = R
S -> R
L -> *R
L -> id
R -> L
```

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
R -> •L
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•
```

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): S => L=R => *R = R
- Shift-reduce conflict
  - SLR parser fails to remember enough info
  - Reduce only after seeing * or =

```
S' -> S
S -> L = R
S -> R
L -> *R
L -> id
R -> L
```

```
I0:  S' -> •S
     S -> •L = R
     S -> •R
     L -> •*R
     L -> •id
     R -> •L

I1:  S' -> S•

I2:  S -> L• = R
     R -> L•

I3:  S -> R•

I4:  L -> *•R
     R -> •L
     L -> •*R
     L -> •id

I5:  L -> id•

I6:  S -> L =•R
     R -> •L
     L -> •*R
     L -> •id

I7:  L -> *R•

I8:  R -> L•

I9:  S -> L = R•
```

SLR(1) Improvement

- 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
- 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
- Just using the entire Follow set is not discriminating enough as the guide for when to reduce
  - 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
- General form of **LR(1) items**[项目]
  - A –> $X_1 \ldots X_i \bullet X_{i+1} \ldots X_j$, a
  - We have states $X_1 \ldots X_i$ on the stack and are looking to put states $X_{i+1} \ldots X_j$ on the stack and then reduce
    - But only if the token following X_j is the terminal *a*
    - *a* is called the lookahead of the configuration
- The lookahead **only** works with completed items[完成项]
  - A –> $X_1 \ldots X_j \bullet$, a
  - All states are now on the stack, but only reduce when next symbol is *a* (*a* is either a terminal or $)
  - Multi lookahead symbols: A -> u•, a/b/c

LR(1) Parsing (cont.)

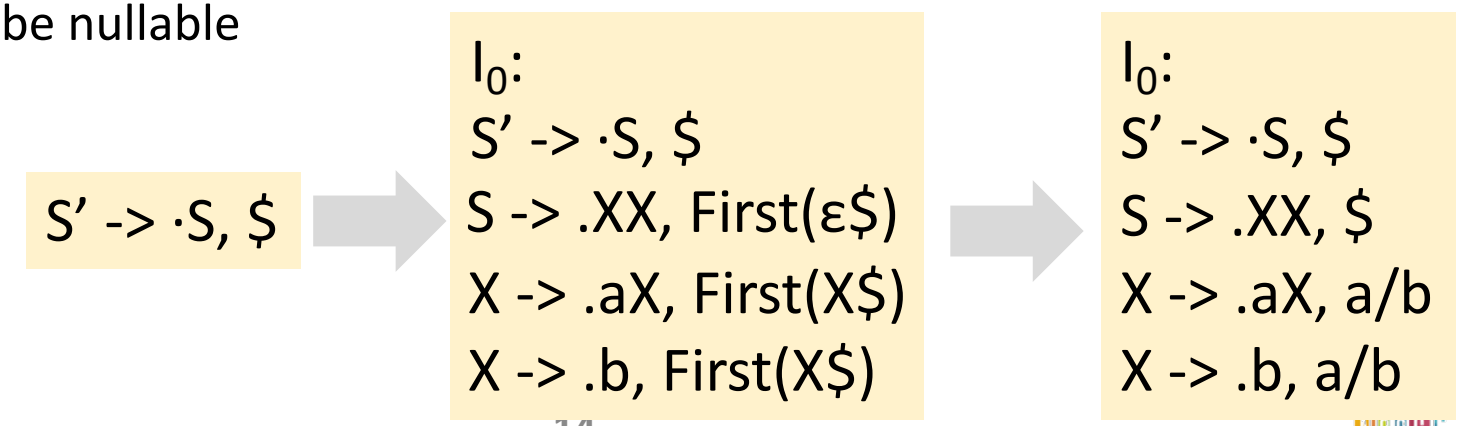
- When to reduce?
  - LR(0): if the configuration set has a **completed item** (i.e., dot at the end)
  - SLR(1): only if the next input token is in the **Follow()** set
  - LR(1): only if the next input token is exactly ***α*** (terminal or $)
  - Trend: more and more precise

- **LR(1) item**s: LR(0) item + lookahead terminals
  - Many differ only in their lookahead components
  - The extra lookahead terminals allow to make parsing decisions beyond the SLR(1) capability, but with a big price
    - More distinguished items and thus more sets
    - Greatly increased Goto and Action table sizes

LR(1) Construction

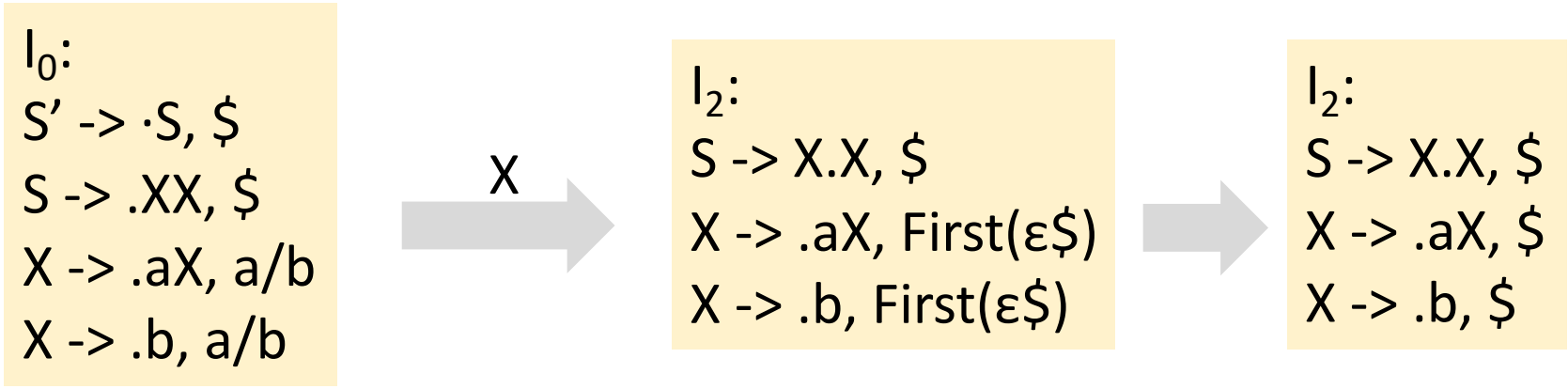
- Configuration sets
  - Sets construction are essentially the same with SLR, but differing on Closure() and Goto()
    - Because with must respect the lookahead
- **Closure()**
  - For each item [A -> u·Bv, a] in *I*, for each production rule B -> w in G', add [B -> ·w, b] to *I*, if
    - b ∈ First(va) and [B -> ·w, b] is not already in I
  - Lookahead is the **First(va)**, which are what can follow B
    - v can be nullable

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

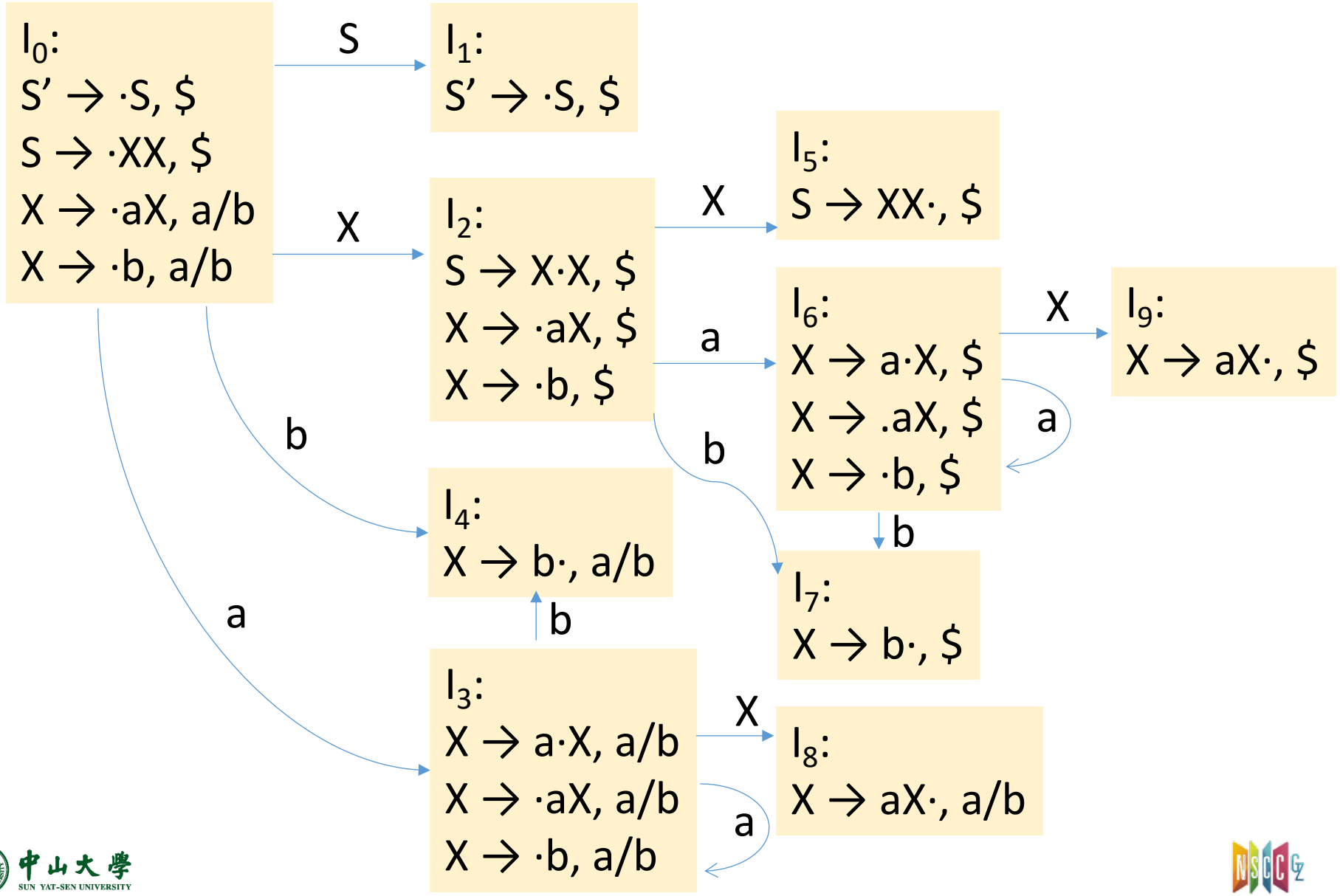


LR(1) Construction (cont.)

- **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** 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



Example

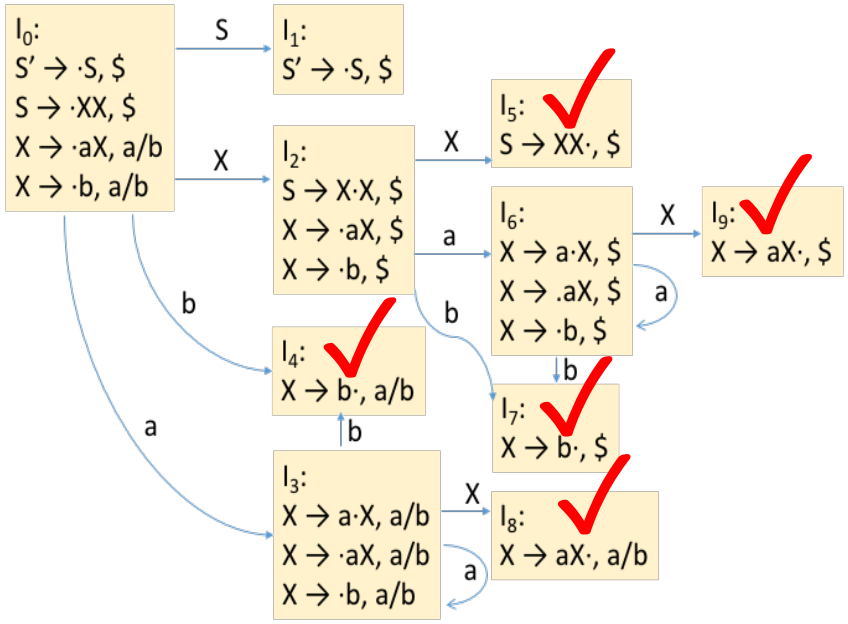


LR(1) Parse Table

- Shift
  - Same as LR(0) and SLR(1)
  - Don't care the lookahead symbols
- Reduce
  - Don't use Follow set (too coarse-grain[粗粒度])
  - Reduce only if input matches lookahead for item
- ACTION and GOTO
  - If [A -> α·aβ, b] ∈ S_i and goto(S_i, a) = S_j, Action[i, a] = sj
    - shift and goto state j if input matches a
    - Same as SLR(1)
  - If [A -> α·, a] ∈ S_i , Action[i, a] = r[R]
    - Reduce R: A -> α if input matches a
    - For SLR, reduced if put input matches Follow(A)

Example

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



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		

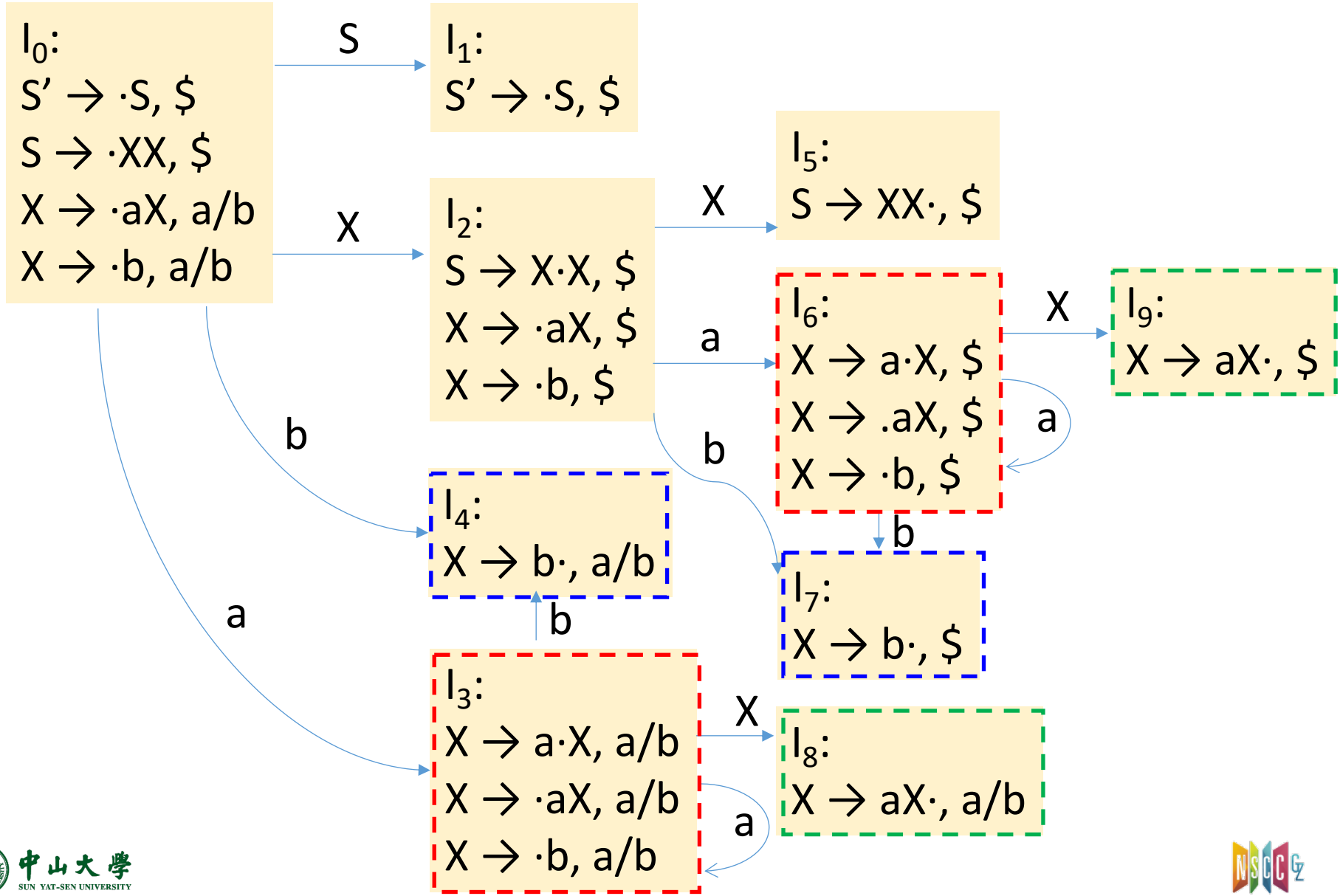
LR(1) Grammars

- 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 if 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

LALR(1) Parser

- LR(1) drawbacks
  - 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)
  - 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

The Example


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

State Merging

- 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 → a·X, a/b
X → .aX, a/b
X → ·b, a/b

I_6:
X → a·X, $
X → .aX, $
X → ·b, $

I_{36}:
X → a·X, a/b/$
X → .aX, a/b/$
X → ·b, a/b/$

I_4:
X → b·, a/b

I_7:
X → b·, $

I_{47}:
X → b·, a/b/$

I_8:
X → aX·, a/b

I_9:
X → aX·, $

I_{89}:
X → aX·, 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)