# 编译原理
## 3. 语法分析-自底向上

rainoftime.github.io
浙江大学
计算机科学与技术学院

# 课程内容

1. Introduction
2. Lexical Analysis
3. **Parsing**
4. Abstract Syntax
5. Semantic Analysis
6. Activation Record
7. Translating into Intermediate Code
8. Basic Blocks and Traces
9. Instruction Selection
10. Liveness Analysis
11. Register Allocation
13. Garbage Collection
14. Object-oriented Languages
18. Loop Optimizations

# Recap: Top-Down Parsing

- LL(1): left-to-right scan + leftmost derivation
  - 从左向右读入程序，最左推导，采用1个前看符号

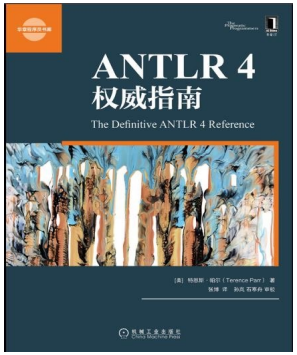
- **LL(1)分析的优点**
  - 运行高效(线性时间)
  - 递归实现符合文法结构、适合手动构造&自动生成

- **LL(1)分析的局限性**
  - 能分析的文法类型受限

# 例: 基于自顶向下分析的Parser生成

- **ANTLR语法分析器的生成器**
  - **Prof. Terence Parr , since 1989**

- **学习资源**
  - The Definitive ANTLR 4 Reference
  - ANTLR v4 with Terence Parr




- **理论基础**
  - **ANTLR3：LL(*)[PLDI11]**
  - **ANTLR4: Adaptive LL(*)[OOPSLA14]**

Terence Parr
Univ. of San Fran.

Sam Harwell
Microsoft

Kathleen Fisher
Tufts University

# Bottom-Up Parsing

- **基于LR(k)文法(*Knuth*, 1963)**
  - 表达力: Every LL(k) grammar is also LR(k)
  - 被Parser自动生成器广泛采用(Yacc, Bison, etc)

- **LR(k) parsing**
  - "L" :left-to-right scanning 自左向右扫描
  - "R" : rightmost derivation in reverse 最右推导的逆
  - "*k*" : 向前看的token个数(当 *k被省略时*, 假设为1)

LR(0), SLR(1), LR(1), LALR(1), …

# The Idea of Bottom-Up Parsing

- 自底向上的语法分析过程可以看成是从串$w$归约为文法开始符号$S$的过程
- **归约**步骤
  - 一个与某产生式体相匹配的特定子串被替换为该产生式头部的非终结符号
- 问题
  - 何时归约 (归约哪些符号串) ？
  - 归约到哪个非终结符号？

# 本讲内容

1. Shift-Reduce
2. LR(0) 分析
3. SLR 分析
4. LR(1) 分析
5. LALR 分析

# 1.移进-规约(Shift-Reduce)
# (Overview of LR)

# An Introductory Example

- LR parsers don't need left-factored grammars and can also handle left-recursive grammars

- Consider the following grammar:

$$E \rightarrow E + ( E ) \mid \text{int}$$

  - Why is this not LL(1)?

- Consider the string: int + ( int ) + ( int )

# 移进-规约 Shift-Reduce

- Idea: Split string into two substrings
  - **Right substring** (a string of terminals) is as yet unexamined by parser
  - **Left substring** has terminals and non-terminals

- The dividing point is marked by a |
  - The | is not part of the string

- Initially, all input is unexamined: | $x_1 x_2 \ldots x_n$

# 例: 移进-规约 Shift-Reduce

$$E \rightarrow E + ( E ) \mid int$$

| int + (int) + (int)$

int + ( int ) + ( int )

# 例: 移进-规约 Shift-Reduce

$$E \rightarrow E + ( E ) \mid int$$

| int + (int) + (int)$ shift

int + ( int ) + ( int )

# 例: 移进-规约 Shift-Reduce

$$E \rightarrow E + ( E ) \mid int$$

| int + (int) + (int)$ shift

int | + (int) + (int)$

int + ( int ) + ( int )

# 例: 移进-规约 Shift-Reduce

$$E \rightarrow E + ( E ) \mid \text{int}$$

| int + (int) + (int)$ shift

int | + (int) + (int)$ red. $E \rightarrow$ int

int + ( int ) + ( int )

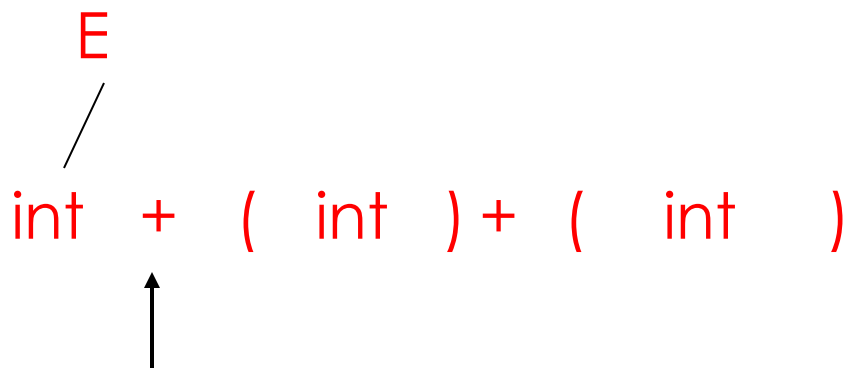
# 例: 移进-规约 Shift-Reduce

$$E \rightarrow E + ( E ) \mid int$$

| int + (int) + (int)$ shift

int | + (int) + (int)$ red. $E \rightarrow int$

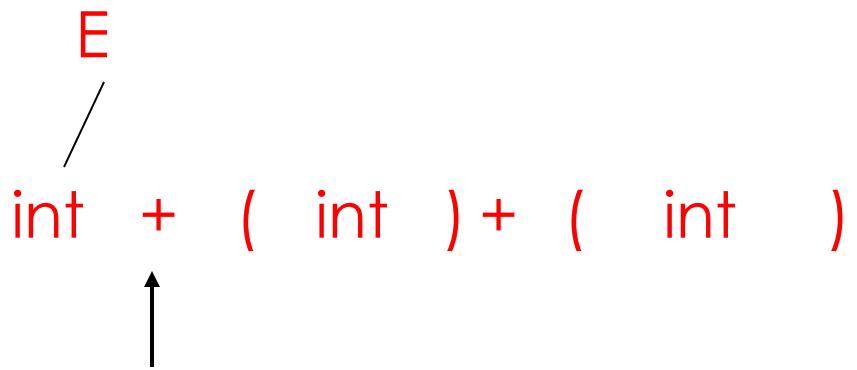
E | + (int) + (int)$



# 例: 移进-规约 Shift-Reduce

$$E \rightarrow E + ( E ) \mid int$$

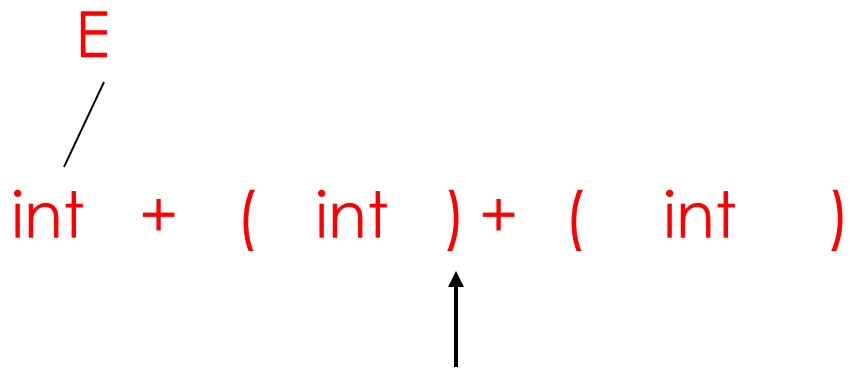
| int + (int) + (int)$ shift
int | + (int) + (int)$ red. $E \rightarrow int$
E | + (int) + (int)$ shift 3 times



# 例: 移进-规约 Shift-Reduce

$$E \rightarrow E + ( E ) \mid int$$

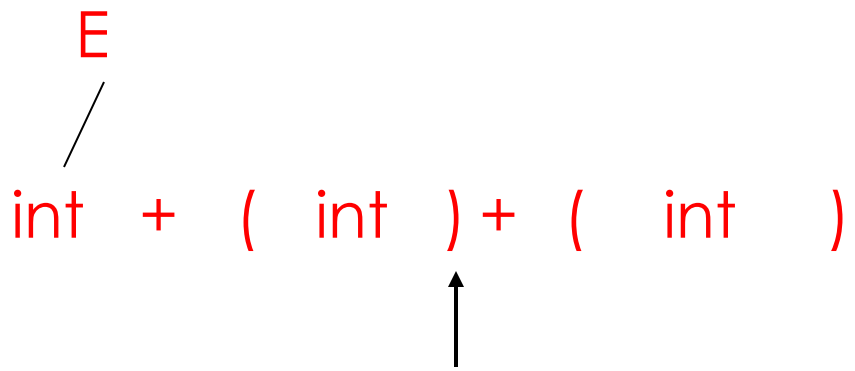
| int + (int) + (int)$ shift
int | + (int) + (int)$ red. $E \rightarrow int$
E | + (int) + (int)$ shift 3 times
E + (int | ) + (int)$



# 例: 移进-规约 Shift-Reduce

$$E \rightarrow E + ( E ) \mid int$$

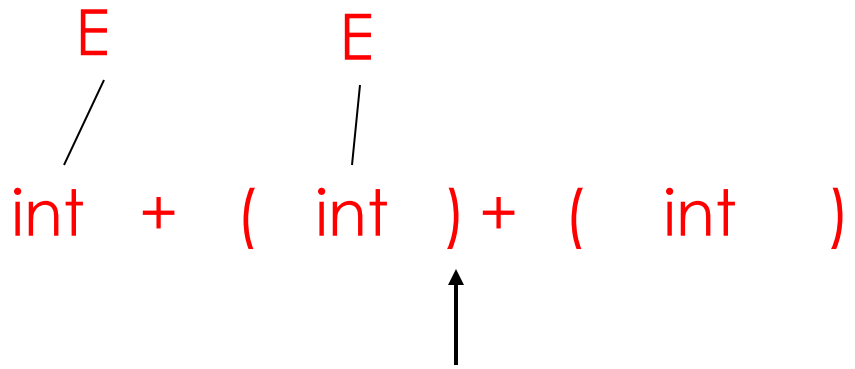
| int + (int) + (int)$ | shift |
|---|---|
| int \| + (int) + (int)$ | red. $E \rightarrow int$ |
| E \| + (int) + (int)$ | shift 3 times |
| E + (int \| ) + (int)$ | red. $E \rightarrow int$ |



# 例: 移进-规约 Shift-Reduce

$$E \rightarrow E + ( E ) \mid int$$

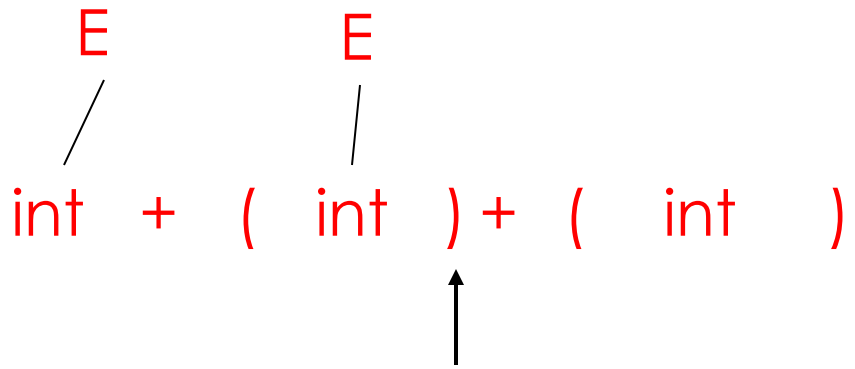
| int + (int) + (int)$ shift
int | + (int) + (int)$ red. $E \rightarrow int$
E | + (int) + (int)$ shift 3 times
E + (int | ) + (int)$ red. $E \rightarrow int$
E + (E | ) + (int)$



# 例: 移进-规约 Shift-Reduce

$$E \rightarrow E + ( E ) \mid int$$

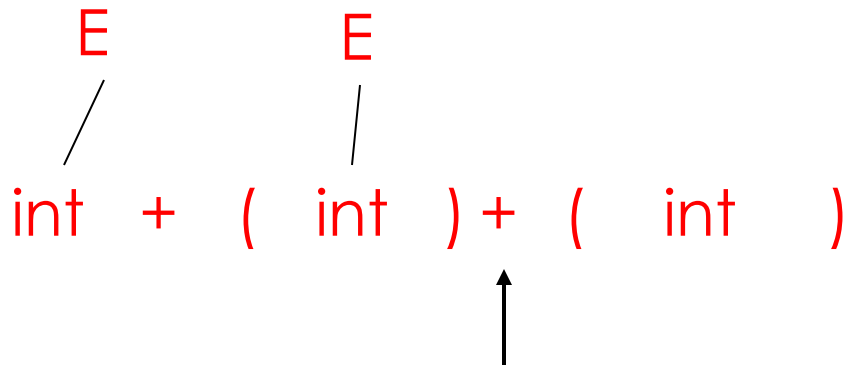
| int + (int) + (int)$ | shift |
|---|---|
| int \| + (int) + (int)$ | red. $E \rightarrow int$ |
| E \| + (int) + (int)$ | shift 3 times |
| E + (int \| ) + (int)$ | red. $E \rightarrow int$ |
| E + (E \| ) + (int)$ | shift |



# 例: 移进-规约 Shift-Reduce

$$E \rightarrow E + ( E ) \mid int$$

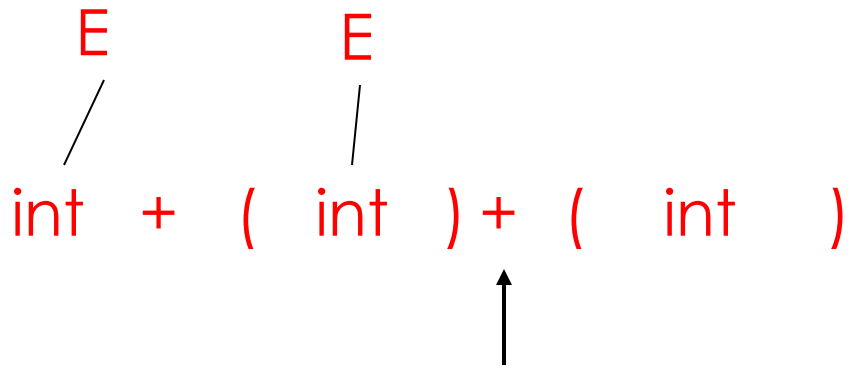
| int + (int) + (int)$ shift
int | + (int) + (int)$ red. $E \rightarrow int$
E | + (int) + (int)$ shift 3 times
E + (int | ) + (int)$ red. $E \rightarrow int$
E + (E | ) + (int)$ shift
E + (E) | + (int)$



# 例: 移进-规约 Shift-Reduce

$$E \rightarrow E + ( E ) \mid int$$

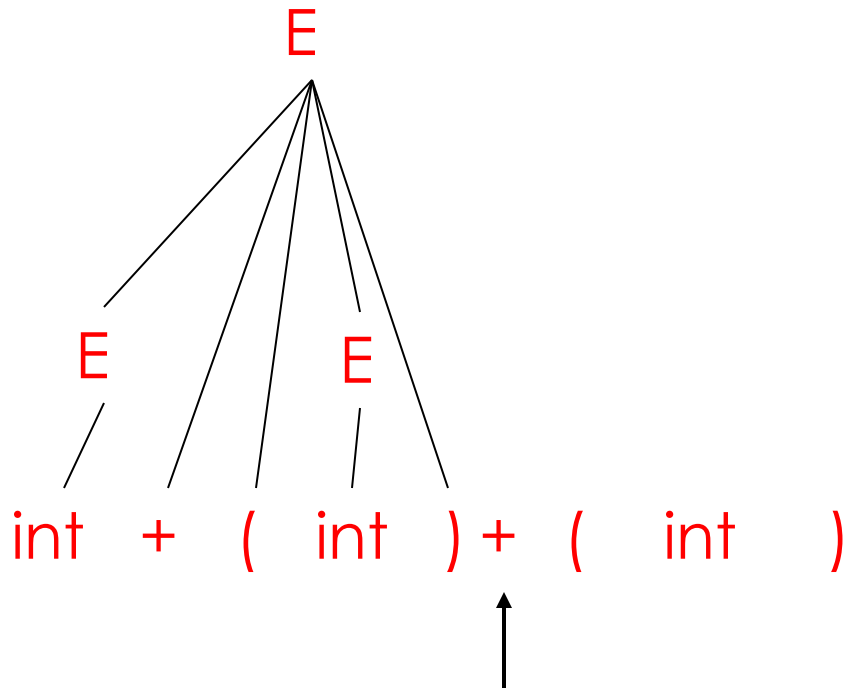
| int + (int) + (int)$ | shift |
|---|---|
| int \| + (int) + (int)$ | red. $E \rightarrow int$ |
| E \| + (int) + (int)$ | shift 3 times |
| E + (int \| ) + (int)$ | red. $E \rightarrow int$ |
| E + (E \| ) + (int)$ | shift |
| E + (E) \| + (int)$ | red. $E \rightarrow E + (E)$ |



# 例: 移进-规约 Shift-Reduce

$$E \rightarrow E + ( E ) \mid int$$

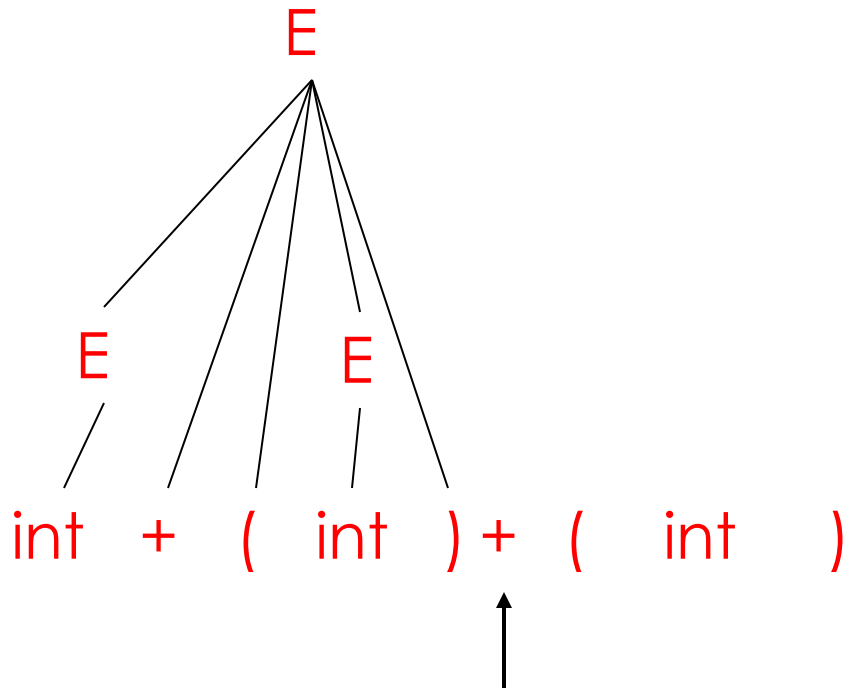
| int + (int) + (int)$ | shift |
|---|---|
| int \| + (int) + (int)$ | red. E → int |
| E \| + (int) + (int)$ | shift 3 times |
| E + (int \| ) + (int)$ | red. E → int |
| E + (E \| ) + (int)$ | shift |
| E + (E) \| + (int)$ | red. E → E + (E) |
| E \| + (int)$ | |



# 例: 移进-规约 Shift-Reduce

$E \rightarrow E + ( E ) \mid int$

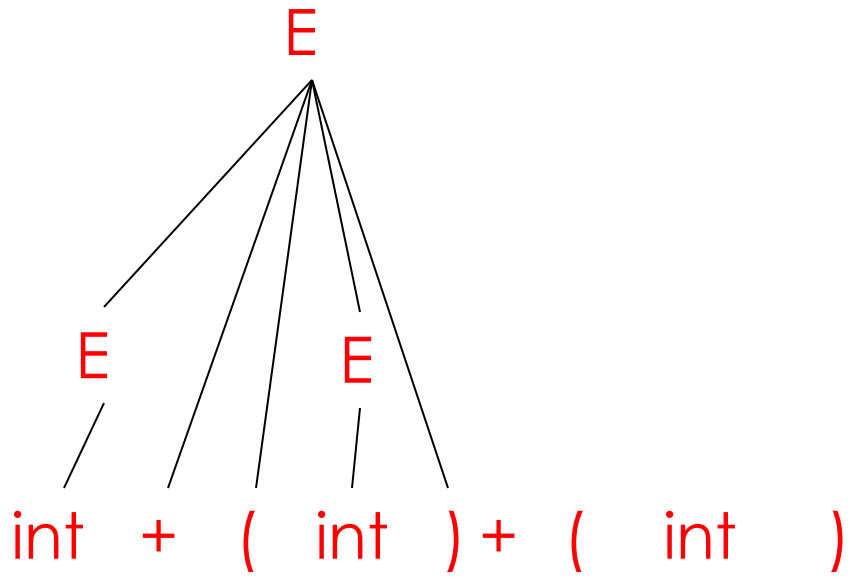
| int + (int) + (int)$ | shift |
|---|---|
| int \| + (int) + (int)$ | red. $E \rightarrow int$ |
| E \| + (int) + (int)$ | shift 3 times |
| E + (int \| ) + (int)$ | red. $E \rightarrow int$ |
| E + (E \| ) + (int)$ | shift |
| E + (E) \| + (int)$ | red. $E \rightarrow E + (E)$ |
| E \| + (int)$ | shift 3 times |



# 例: 移进-规约 Shift-Reduce

$E \rightarrow E + ( E ) \mid int$

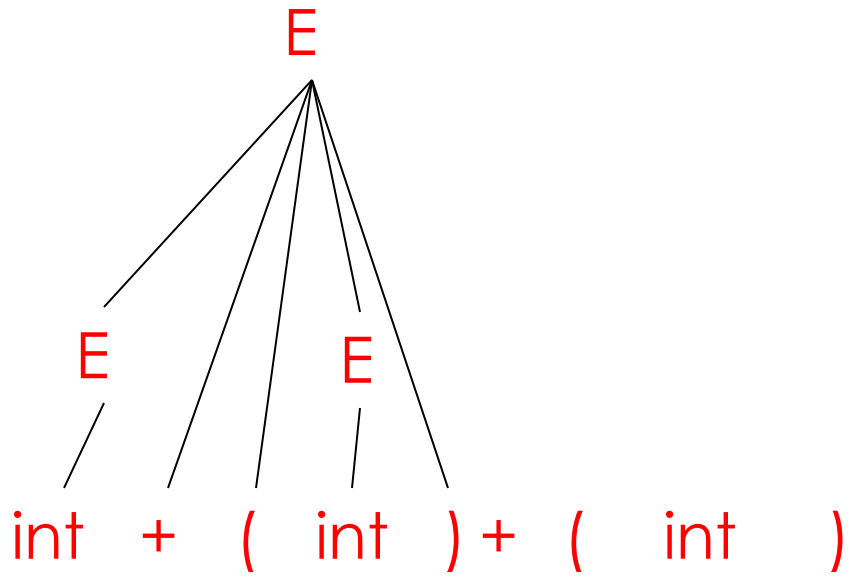
| int + (int) + (int)$ | shift |
|---|---|
| int \| + (int) + (int)$ | red. $E \rightarrow int$ |
| E \| + (int) + (int)$ | shift 3 times |
| E + (int \| ) + (int)$ | red. $E \rightarrow int$ |
| E + (E \| ) + (int)$ | shift |
| E + (E) \| + (int)$ | red. $E \rightarrow E + (E)$ |
| E \| + (int)$ | shift 3 times |
| E + (int \| )$ | |



# 例: 移进-规约 Shift-Reduce

$$E \rightarrow E + ( E ) \mid int$$

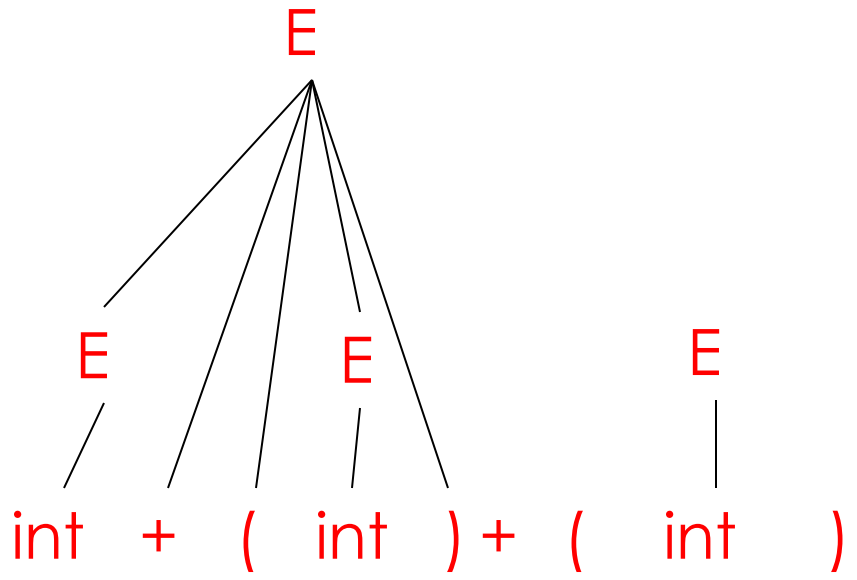
| | |
|---|---|
| I int + (int) + (int)$ | shift |
| int I + (int) + (int)$ | red. E → int |
| E I + (int) + (int)$ | shift 3 times |
| E + (int I ) + (int)$ | red. E → int |
| E + (E I ) + (int)$ | shift |
| E + (E) I + (int)$ | red. E → E + (E) |
| E I + (int)$ | shift 3 times |
| E + (int I )$ | red. E → int |



# 例: 移进-规约 Shift-Reduce

$E \rightarrow E + ( E ) \mid int$

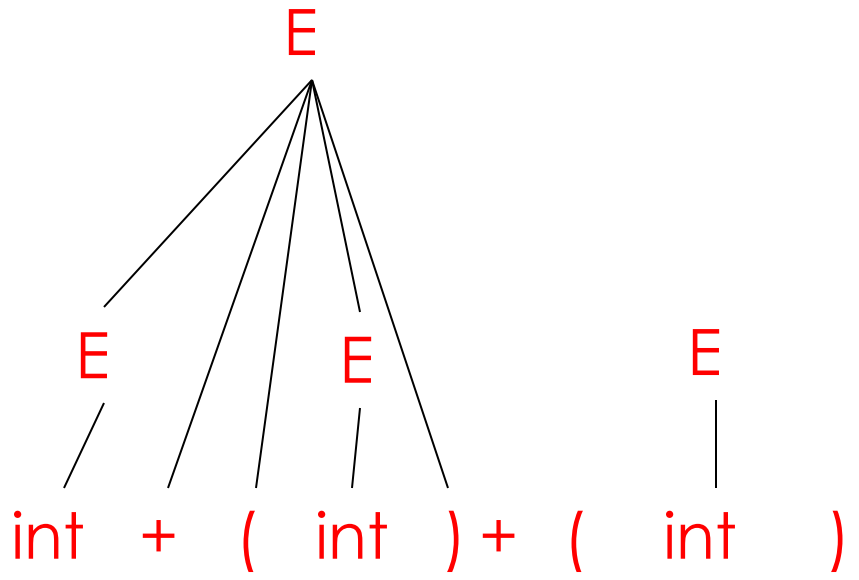
| | |
|---|---|
| I int + (int) + (int)$ | shift |
| int I + (int) + (int)$ | red. E → int |
| E I + (int) + (int)$ | shift 3 times |
| E + (int I ) + (int)$ | red. E → int |
| E + (E I ) + (int)$ | shift |
| E + (E) I + (int)$ | red. E → E + (E) |
| E I + (int)$ | shift 3 times |
| E + (int I )$ | red. E → int |
| E + (E I )$ | |



# 例: 移进-规约 Shift-Reduce

$E \rightarrow E + ( E ) \mid int$

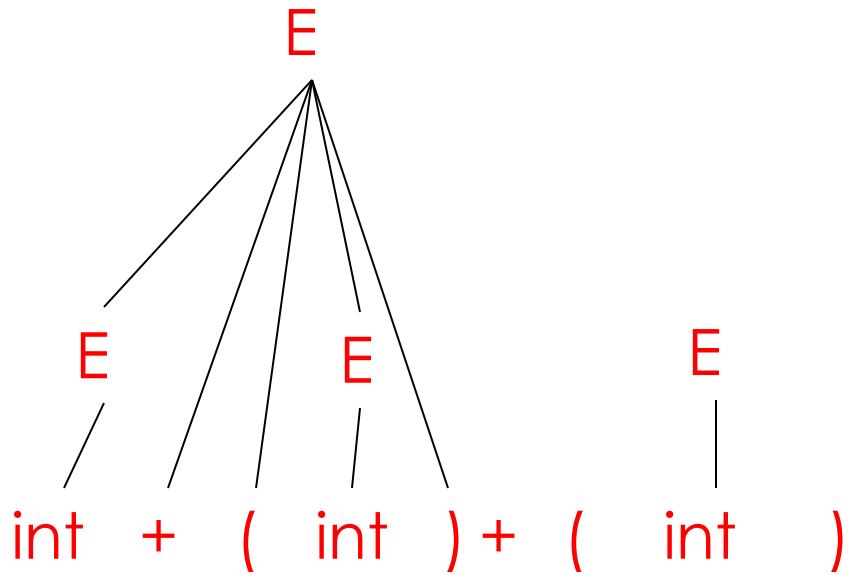
| | |
|---|---|
| ı int + (int) + (int)$ | shift |
| int ı + (int) + (int)$ | red. E → int |
| E ı + (int) + (int)$ | shift 3 times |
| E + (int ı ) + (int)$ | red. E → int |
| E + (E ı ) + (int)$ | shift |
| E + (E) ı + (int)$ | red. E → E + (E) |
| E ı + (int)$ | shift 3 times |
| E + (int ı )$ | red. E → int |
| E + (E ı )$ | shift |



# 例: 移进-规约 Shift-Reduce

$$E \rightarrow E + ( E ) \mid int$$

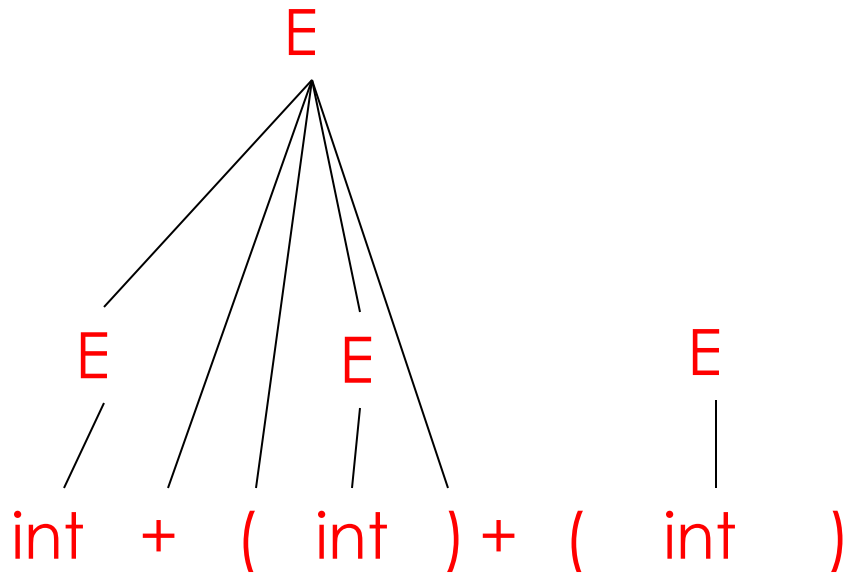
| int + (int) + (int)$ | shift |
|---|---|
| int \| + (int) + (int)$ | red. E → int |
| E \| + (int) + (int)$ | shift 3 times |
| E + (int \| ) + (int)$ | red. E → int |
| E + (E \| ) + (int)$ | shift |
| E + (E) \| + (int)$ | red. E → E + (E) |
| E \| + (int)$ | shift 3 times |
| E + (int \| )$ | red. E → int |
| E + (E \| )$ | shift |
| E + (E) \| $ | |



# 例: 移进-规约 Shift-Reduce

$$E \rightarrow E + ( E ) \mid int$$

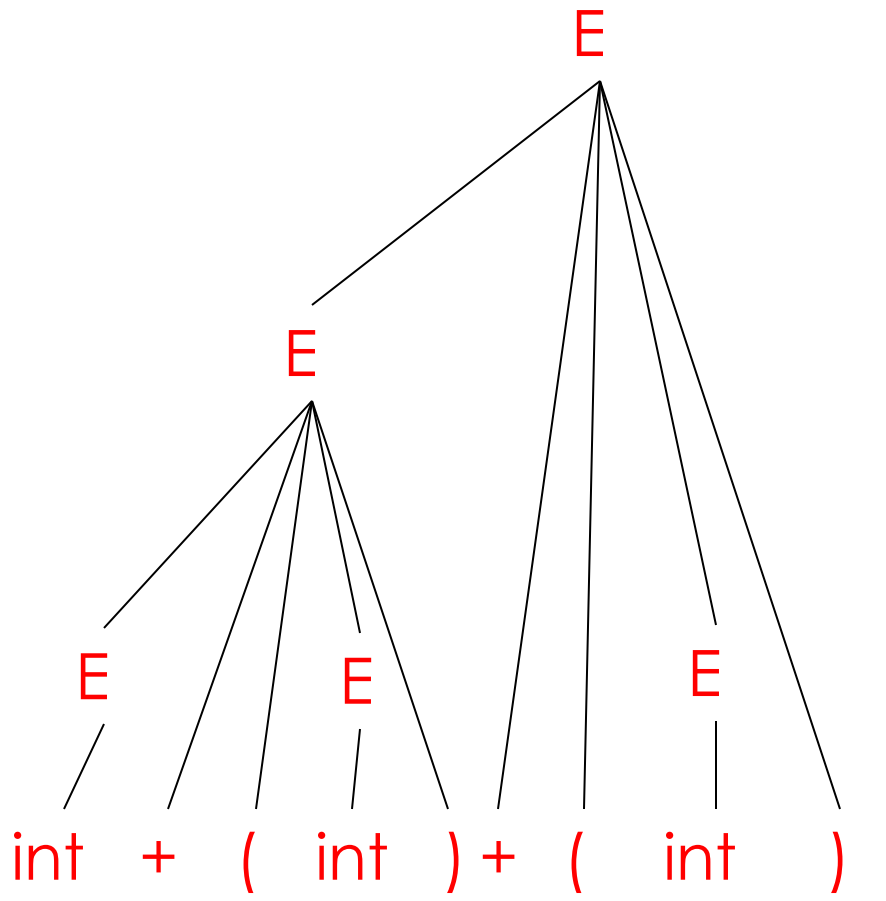
| | |
|---|---|
| I int + (int) + (int)$ | shift |
| int I + (int) + (int)$ | red. E → int |
| E I + (int) + (int)$ | shift 3 times |
| E + (int I ) + (int)$ | red. E → int |
| E + (E I ) + (int)$ | shift |
| E + (E) I + (int)$ | red. E → E + (E) |
| E I + (int)$ | shift 3 times |
| E + (int I )$ | red. E → int |
| E + (E I )$ | shift |
| E + (E) I $ | red. E → E + (E) |



# 例: 移进-规约 Shift-Reduce

$E \rightarrow E + ( E ) \mid int$

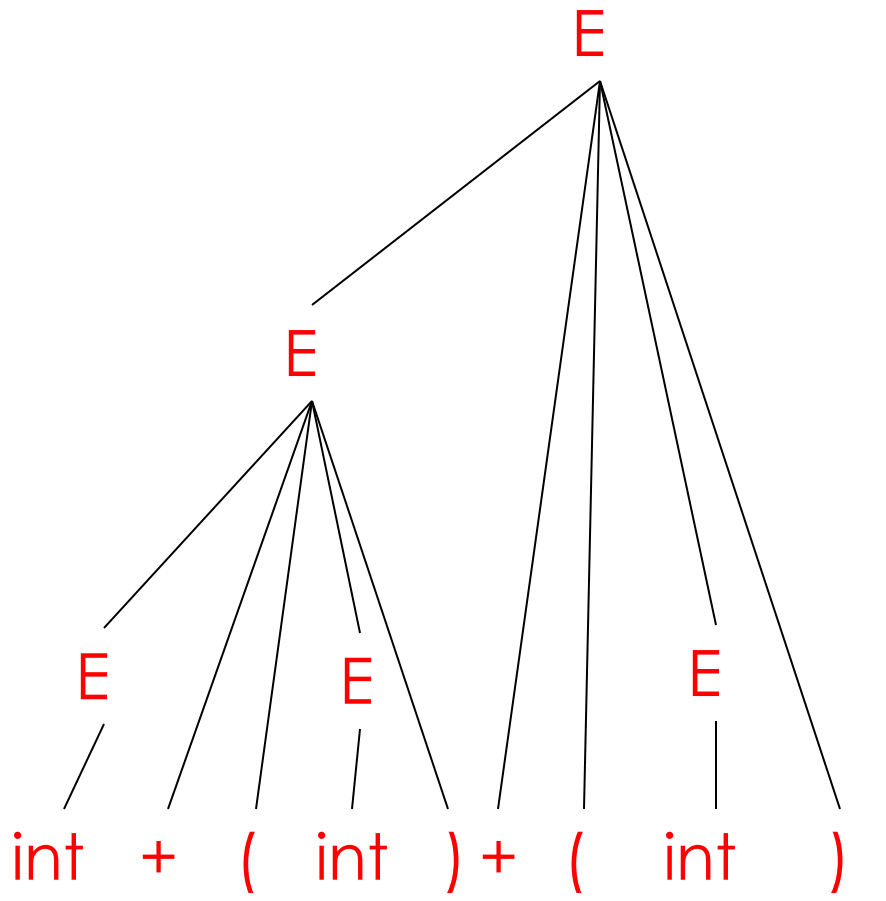
| | |
|---|---|
| I int + (int) + (int)$ | shift |
| int I + (int) + (int)$ | red. E → int |
| E I + (int) + (int)$ | shift 3 times |
| E + (int I ) + (int)$ | red. E → int |
| E + (E I ) + (int)$ | shift |
| E + (E) I + (int)$ | red. E → E + (E) |
| E I + (int)$ | shift 3 times |
| E + (int I )$ | red. E → int |
| E + (E I )$ | shift |
| E + (E) I $ | red. E → E + (E) |
| E I $ | |



# 例: 移进-规约 Shift-Reduce

$E \rightarrow E + ( E ) \mid int$

| int + (int) + (int)$ | shift |
|---|---|
| int \| + (int) + (int)$ | red. E → int |
| E \| + (int) + (int)$ | shift 3 times |
| E + (int \| ) + (int)$ | red. E → int |
| E + (E \| ) + (int)$ | shift |
| E + (E) \| + (int)$ | red. E → E + (E) |
| E \| + (int)$ | shift 3 times |
| E + (int \| )$ | red. E → int |
| E + (E \| )$ | shift |
| E + (E) \| $ | red. E → E + (E) |
| E \| $ | accept |

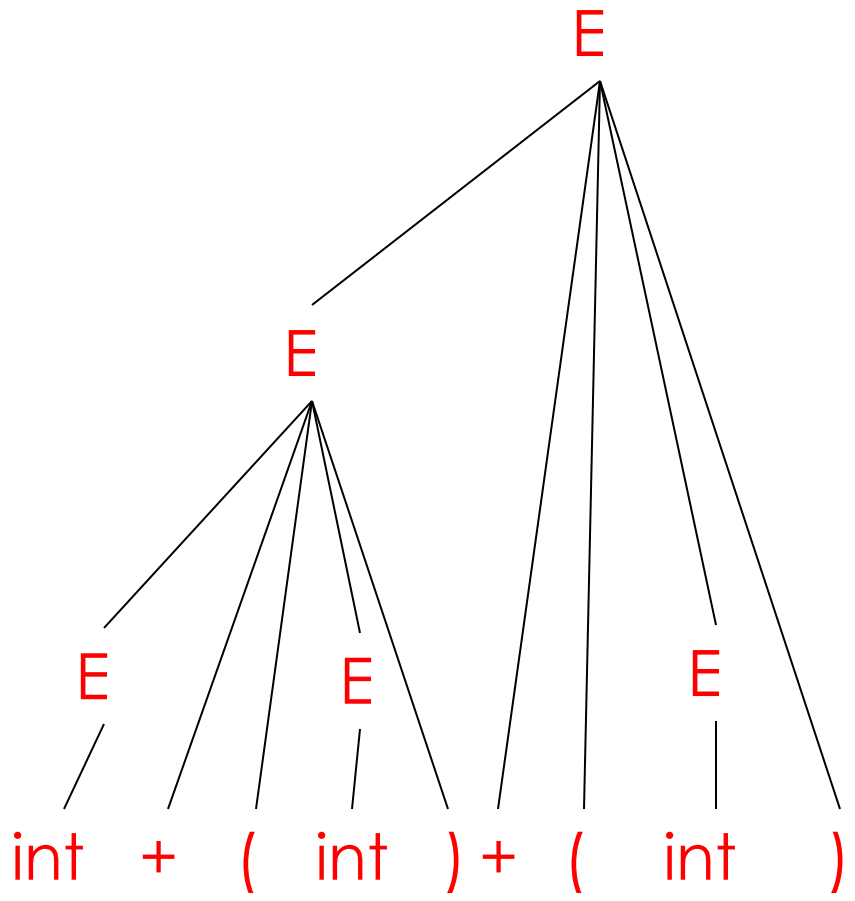



规约到了文法的Start Symbol, i.e., E

# 例: 移进-规约 Shift-Reduce

$$E \rightarrow E + ( E ) \mid int$$

| | |
|---|---|
| ı int + (int) + (int)$ | shift |
| int ı + (int) + (int)$ | red. E → int |
| E ı + (int) + (int)$ | shift 3 times |
| E + (int ı ) + (int)$ | red. E → int |
| E + (E ı ) + (int)$ | shift |
| E + (E) ı + (int)$ | red. E → E + (E) |
| E ı + (int)$ | shift 3 times |
| E + (int ı )$ | red. E → int |
| E + (E ı )$ | shift |
| E + (E) ı $ | red. E → E + (E) |
| E ı $ | accept |




E => E + (E) => E + (int) => E + (E) + (int) => E + (int) + (int) => int + (int) + (int)

**最右推导的逆过程! (Rightmost derivation in reverse)**

# 移进-规约 Shift-Reduce

- LR分析: 最右推导的逆过程
  - **限制了规约方式**(类似LL中的最左推导)

- 最右句型：最右推导过程中出现的句型
  - LR分析的每一步都是最右句型
  - $\alpha \mid \beta$ 是最右句型

- 如何实现?

# LR分析的一般模式: 基于栈的移进-归约

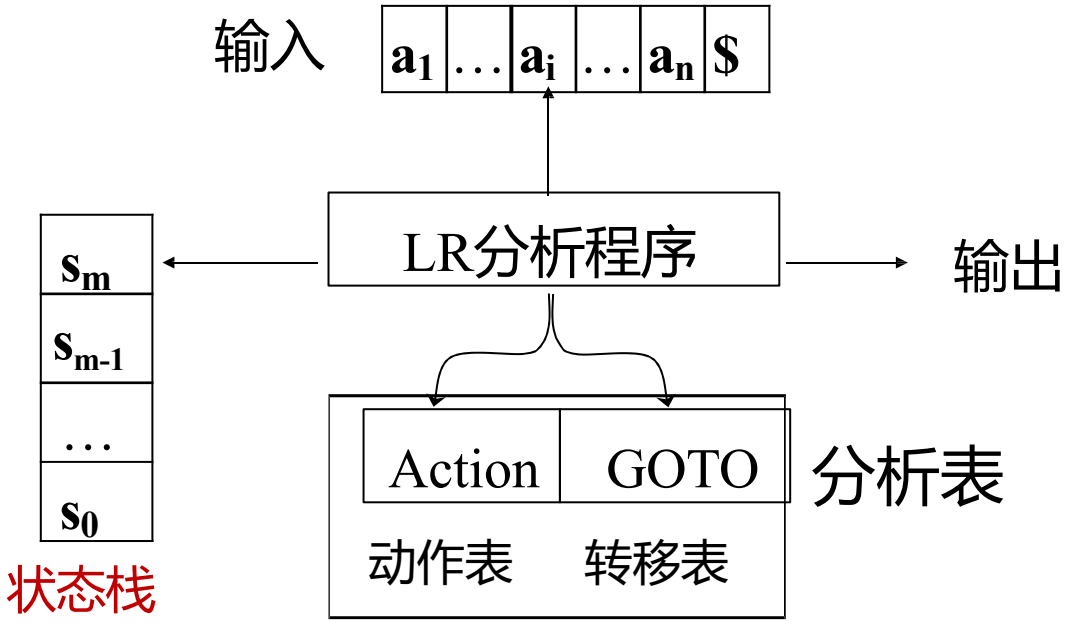
$\alpha$ I $\beta$

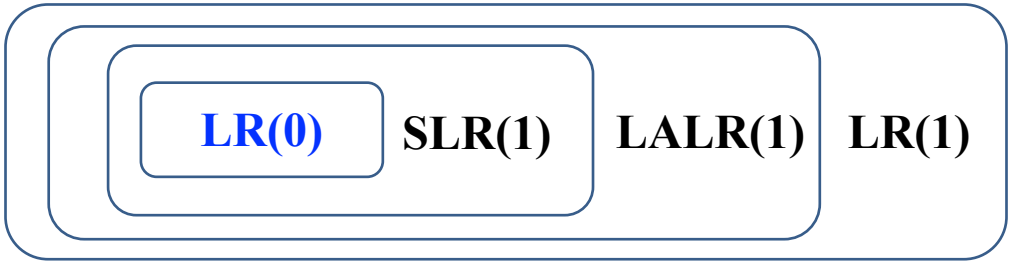
- Two components
  - **Stack**: holds the left-substring $\alpha$ (terminals + nonterminals)
  - **Input stream**: holds the remaining input $\beta$ (terminals)
- Four actions
  - **Shift**: push next input (terminal) on to top of stack
  - **Reduce**:
    - Top of stack should match RHS of rule (e.g., **X -> A B C**)
    - pop the RHS from the top of stack (e.g., **pop C B A**)
    - push the LHS onto the stack (e.g., **push X**)
  - **Error**
  - **Accept:** shift $ and can reduce what remains on stack to the start symbol!

**核心问题: When to shift and when to reduce?**

# 表驱动的LR分析

- 最通用的无回溯移入-规约分析
- 所有的分析器都使用相同的驱动程序
  - 虽然复杂语法对应的表很大，但分析表可以自动生成
  - 分析表随LR分析文法的不同而不同






# 2. LR(0) 分析

- LR(0) Parsing的NFA
- LR(0) Parsing的DFA和分析表

# LR(0) 语法分析思路

- 自底向上分析: 不断地凑出产生式的 RHS

- 假设下一次将会用到的产生式为 $X \rightarrow \alpha\ \beta$, 使用它进行规约前，栈顶可能包含3种情况
  - ……
  - …α…
  - …α β

**凑出这个产生式RHS 的进度不同**

**对于这些情况，希望对其做出不一样的操作**

# LR(0) 语法分析思路

- **关键问题**: 如何知道栈顶的内容可以归约了？
  - 维护一个状态，记录当前识别的进度

- 项/Item: 一个产生式加上在其中某处的一个点
  - 例: 产生式$A \rightarrow \bullet XYZ$有4个Item

    $A \rightarrow \bullet XYZ$，$A \rightarrow X \bullet YZ$，$A \rightarrow XY \bullet Z$，$A \rightarrow XYZ \bullet$
  - $A \rightarrow \alpha \bullet \beta$: 已扫描/归约到了$\alpha$，并期望在接下来的输入中经过扫描/归约得到$\beta$，然后把$\alpha\beta$归约到$A$
  - $A \rightarrow \alpha\beta \bullet$ : 已扫描/归约得到了$\alpha\beta$，可以把$\alpha\beta$归约为$A$

LR(0) Item类似有穷自动机的**状态!**

# LR(0) 语法分析思路

- 项/Item: 一个产生式加上在其中某处的一个点
- 状态跳转:
  - 一个项读入一个符号后，可变为另一个项
  - $A \rightarrow \cdot xyz$ 读入$x$变为 $A \rightarrow x \cdot yz$，类似状态间的跳转

状态+ 跳转→ 自动机

- 文法产生式是有限的，每个产生式右部的长度也是有限的
  - 项的数量也是有限的

有穷自动机，被称为LR(0)自动机

# LR(0) 语法分析思路

- 自底向上分析: 不断地凑出产生式的 RHS
- **关键问题**: 如何知道栈顶的内容可以归约了？
  - 维护一个状态，记录当前识别的进度

➢ **例：** $S \rightarrow bBB$

- ➢ $S \rightarrow \cdot bBB$ ⟵ 移进状态
- ➢ $S \rightarrow b \cdot BB$
- ➢ $S \rightarrow bB \cdot B$

期待B

待约状态

- ➢ $S \rightarrow bBB \cdot$ ⟵ 归约状态

**LR分析器基于这样一些状态来构造自动机进行RHS的识别**

# LR(0) Parsing NFA(非确定性有穷自动机)

- **起始&终结状态**
  - 文法$G$中增加新开始符号$S'$，并加入产生式$S' \rightarrow S\$$
  - 按$S' \rightarrow S\$$进行归约：“将输入符号串归约成为开始符号”
  - 加入S‘方便表示起始和终结状态

**S’ → S $**
**S → ( S )**
**S → a**

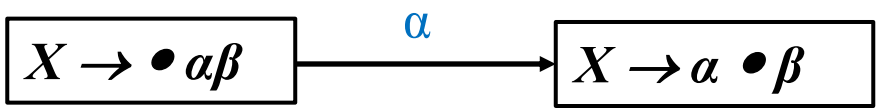
S’ → • S $

S’ → S • $

注：此处的NFA不是指直接用来识别LR(0)语言的自动机（NFA只能识别正则语言，然而正则语言＜LR(0)）。该NFA是用来“记录当前识别进度”的（帮助判断栈顶内容是否可归约了）

# LR(0) Parsing NFA(非确定性有穷自动机)

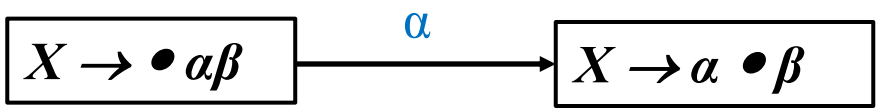
- **状态迁移**
  - LR(0) Item之间会有转换关系，如
  1. $X \to \bullet\, \alpha\beta$ **接收 α后变成** $X \to \alpha \bullet \beta$



# LR(0) Parsing NFA(非确定性有穷自动机)

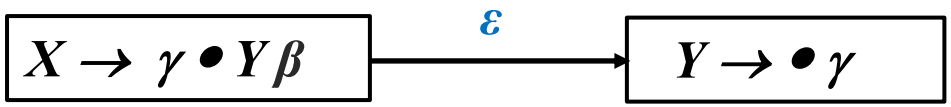
- **状态迁移**
  - LR(0) Item之间会有转换关系，如

1. $X \to \bullet\, \alpha\beta$ **接收 α后变成** $X \to \alpha \bullet \beta$




2. **如果存在产生式** $X \to \alpha Y\beta$, $Y \to \gamma$,

   **那么** $X \to \alpha \bullet Y\beta$ **可以转换到** $Y \to \bullet\, \gamma$



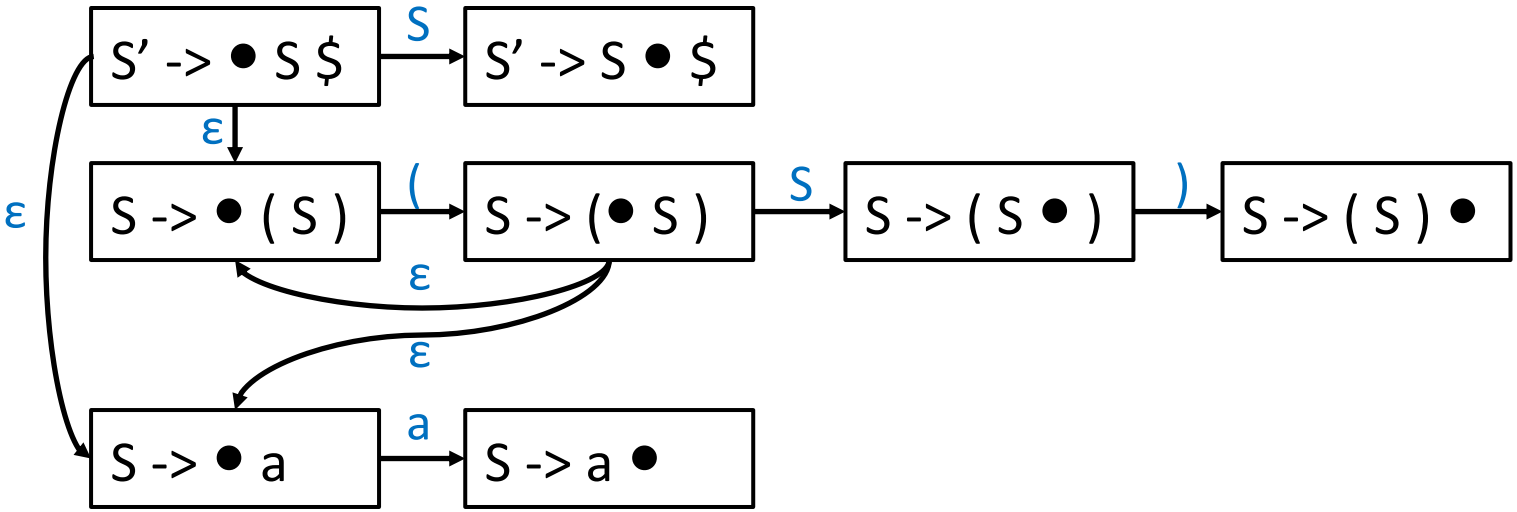

希望看到由$Y\beta$推导出的串，那要先看到由$Y$推导出的串，
因此加上$Y$的各个产生式对应的项

# LR(0) Parsing NFA

- 起始&终结状态
- 状态迁移

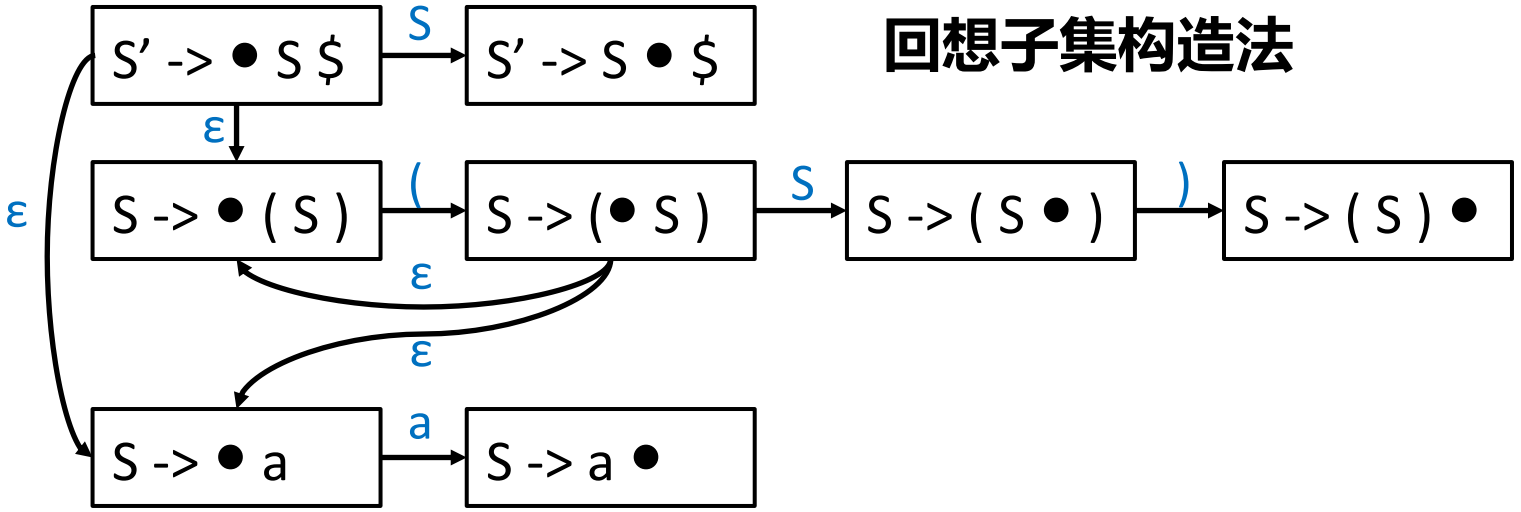
1. $X \to \bullet\ \alpha\beta$ 接收 α后变成$X \to \alpha \bullet \beta$
2. 如果存在产生式 $X \to \alpha Y \beta$, $Y \to \gamma$,
那么$X \to \alpha \bullet Y\beta$可以转换到$Y \to \bullet\ \gamma$

S’ → S $
S → ( S )
S → a



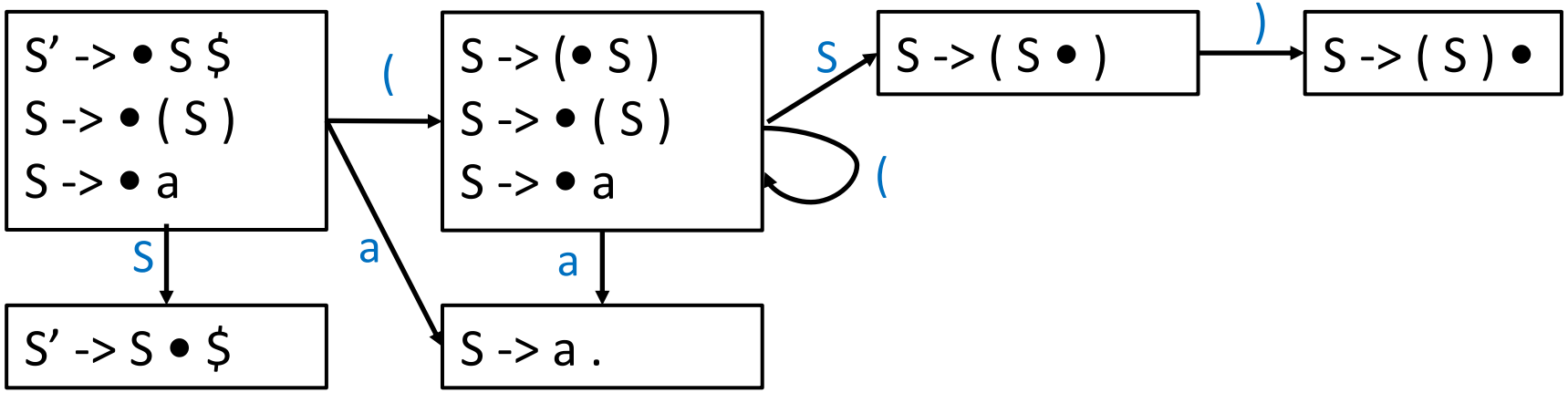
# LR(0) Parsing NFA->DFA!

S’ → S $
S → ( S )
S → a

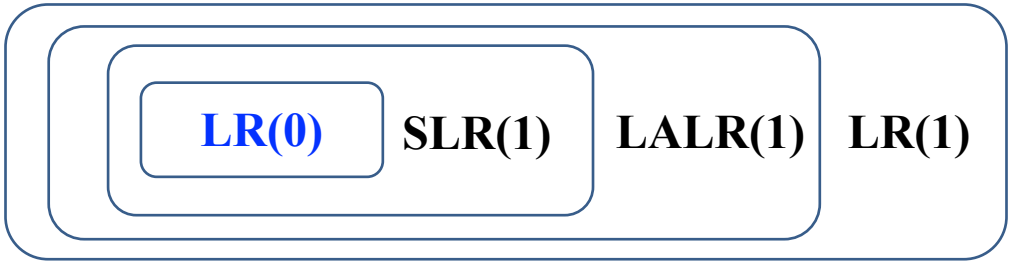



回想子集构造法

The state of the DFA: a set of LR(0) items (“项集”)




**LR(0)分析通常直接构造该DFA**



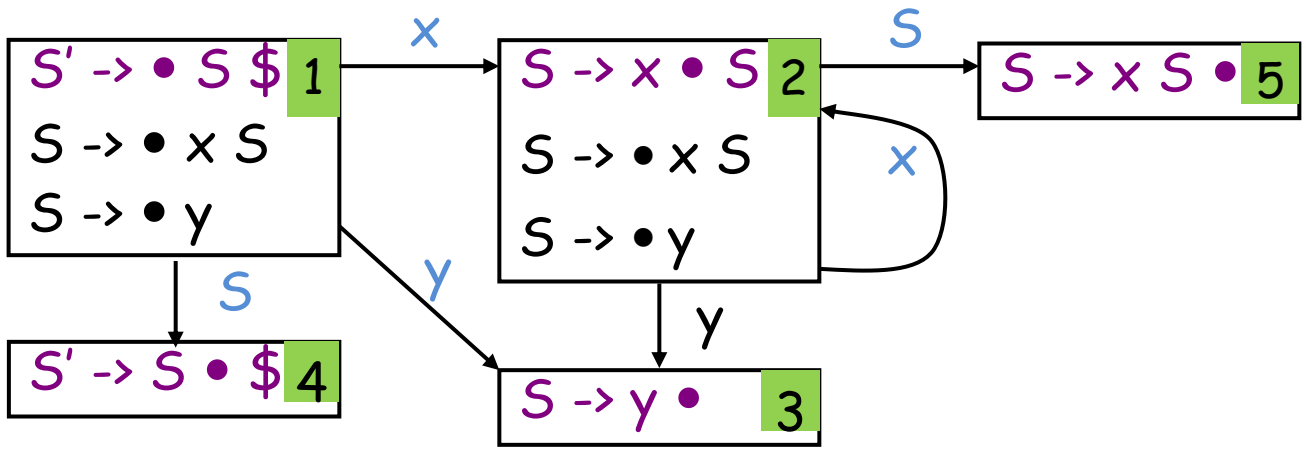

# 2. LR(0) 分析

- LR(0) Parsing的NFA
- LR(0) Parsing的DFA和分析表

# LR(0) Parsing DFA构造

- 如何跳过NFA直接构造以下DFA？

0: S′ -> S$
1: S -> x S
2: S -> y




注：此处的DFA不是指直接用来识别LR(0)语言的自动机（DFA只能识别正则语言，然而正则语言＜LR(0)）。该DFA是用来“记录当前识别进度”的（帮助判断栈顶内容是否可归约了）

# LR(0) Parsing DFA构造: 项集闭包CLOSURE

I: a set of items(项集)

X: a symbol (terminal or non-terminal)

**Closure**($I$) =
  **repeat**
    **for** any item $A \rightarrow \alpha \bullet X \beta$ in $I$
      **for** any production $X \rightarrow \gamma$
        $I \leftarrow I \cup \{X \rightarrow \bullet \gamma\}$
  **until** $I$ does not change.
  **return** $I$

0: S' -> S$
1: S -> x S
2: S -> y

S' -> • S $ 1
S -> • x S
S -> • y

**左边的项集为**

Closure({S' -> • S $})

类似NFA转DFA过程中的ε-Closure

# LR(0) Parsing DFA构造: Goto(状态转换函数)

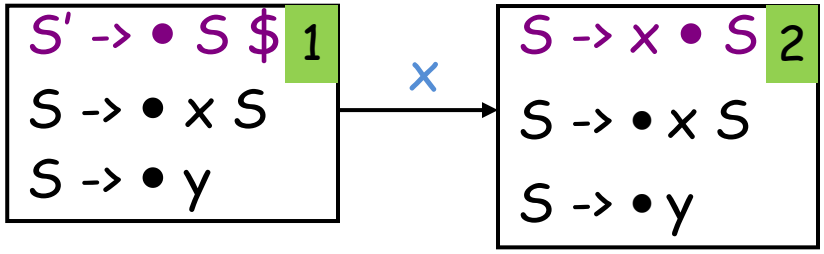
I: a set of items(项集)

X: a symbol (**terminal or non-terminal**)

**Goto**(*I*, *X*) =
  set *J* to the empty set
  **for** any item $A \rightarrow \alpha \bullet X\beta$ in *I*
    add $A \rightarrow \alpha X \bullet \beta$ to *J*
  **return Closure**(*J*)

0: S' -> S$
1: S -> x S
2: S -> y

**项集2 = Goto(项集1, x)**




*I*是一个项集，*X*是一个文法符号，则GOTO(*I*, *X*)定义为*I*中所有形如$A \rightarrow \alpha \cdot X\beta$的项所对应的项$A \rightarrow \alpha X \cdot \beta$ 的集合的闭包

类似NFA转DFA过程中计算新的DFA状态

# LR(0) Parsing DFA构造

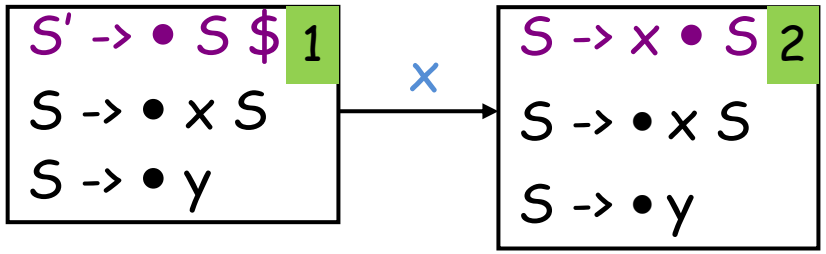
I: a set of items(项集)
X: a symbol (terminal or non-terminal)

T: the set of states
E: the set of (shift or goto) edges

Initialize $T$ to {**Closure**({S′ → • S$})}
Initialize $E$ to empty.
**repeat**
  **for** each state $I$ in $T$
    **for** each item A → α • Xβ in $I$
      let $J$ be ***Goto***$(I, X)$
      $T \leftarrow T \cup \{J\}$
      E ← E ∪ {I →$^{X}$J}
**until** E and T did not change in this iteration

0: S' -> S$
1: S -> x S
2: S -> y

- **起始状态 Closure({S′ → • S$})**





- **不断用Goto计算新状态**

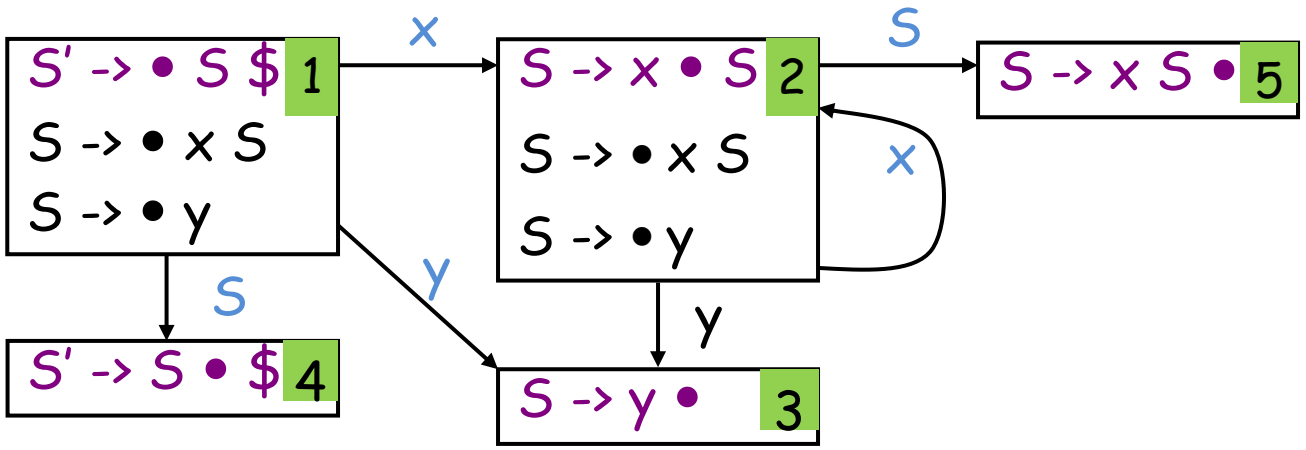
包含S' → S • $的项集所对应的DFA状态为终止状态

# 例: LR(0) Parsing DFA构造

- 考虑以下LR(0)文法

0: S′ -> S$
1: S -> x S
2: S -> y

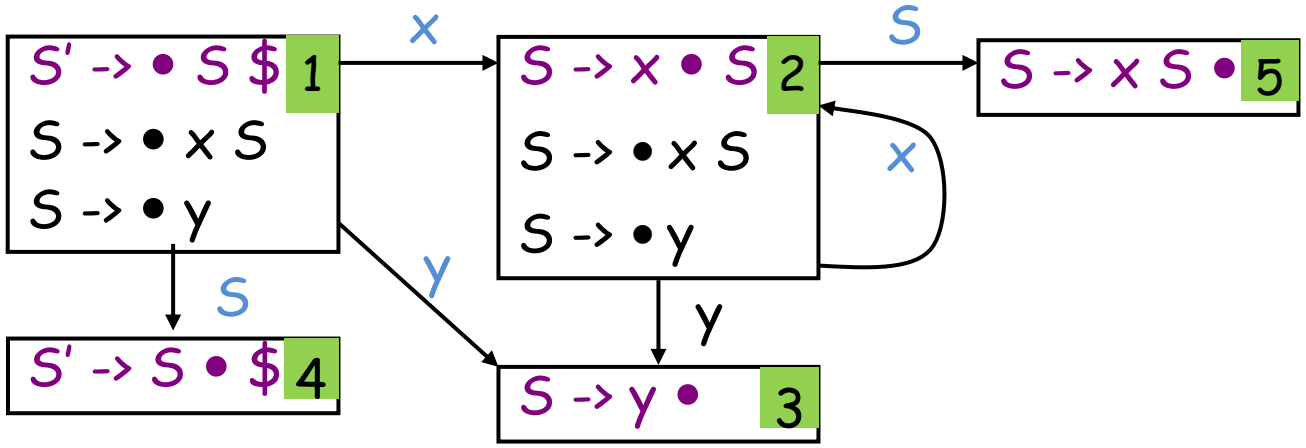
Initialize $T$ to {**Closure**({S′ → .S$})}
Initialize $E$ to empty.
**repeat**
  **for** each state $I$ in $T$
    **for** each item A → α.Xβ in $I$
      let $J$ be ***Goto***$(I, X)$
      $T \leftarrow T \cup \{J\}$
      E ← E ∪ {I $\xrightarrow{X}$ J}
**until** E and T did not change in this iteration



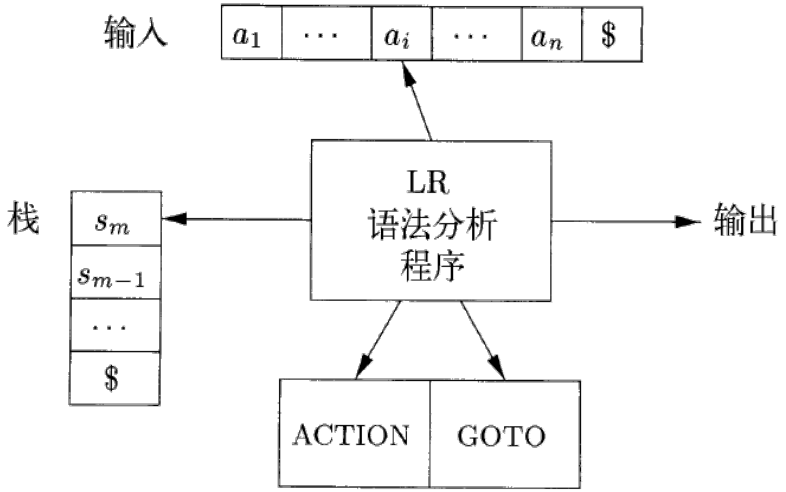
# 从LR(0) Parsing DFA到语法分析表

- 假设已经构造LR(0) 自动机

0: S' -> S$
1: S -> x S
2: S -> y

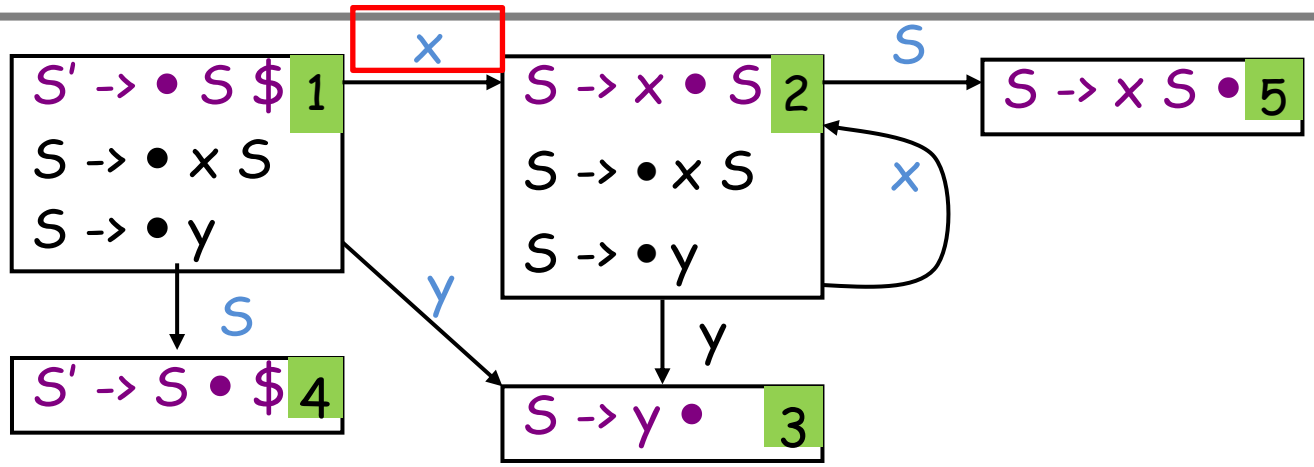



回顾: To generate an LR parser, we must create the Action and GOTO tables



# 从LR(0) Parsing DFA到语法分析表

0: S' -> S$
1: S -> x S
2: S -> y



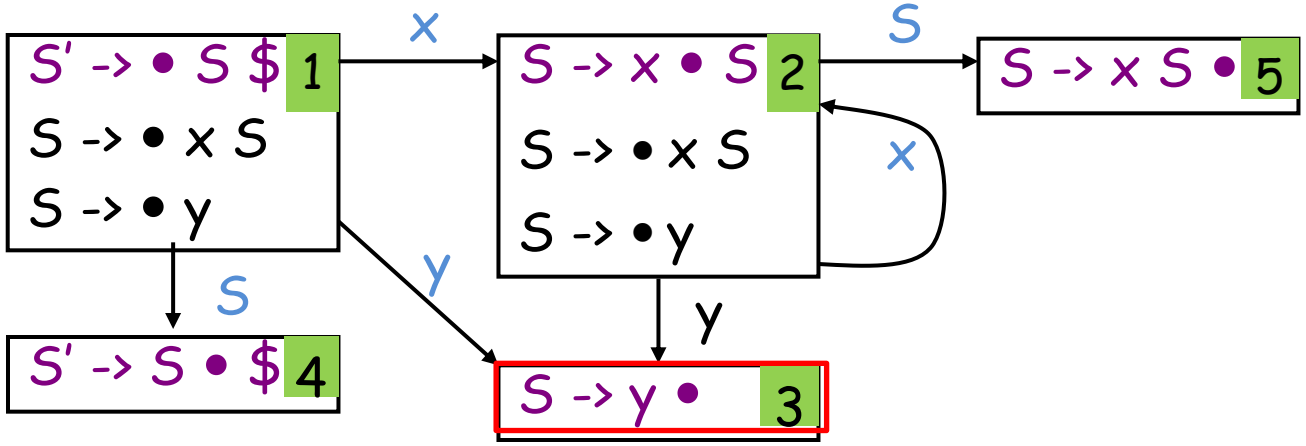

| | Action | | | GOTO |
|---|---|---|---|---|
| State | x | y | $ | S |
| 1 | s2 | s3 | | g4 |
| 2 | s2 | s3 | | g5 |
| 3 | r2 | r2 | r2 | |
| 4 | | | accept | |
| 5 | r1 | r1 | r1 | |

**Shift**: edge labeled with terminal t and from si to sn:

- **T[i, t] = sn** (shift n)

# 从LR(0) Parsing DFA到语法分析表

0: S' -> S$
1: S -> x S
2: S -> y




| | Action | | | GOTO |
|---|---|---|---|---|
| State | x | y | $ | S |
| 1 | s2 | s3 | | g4 |
| 2 | s2 | s3 | | g5 |
| 3 | r2 | r2 | r2 | |
| 4 | | | accept | |
| 5 | r1 | r1 | r1 | |

**Shift**: edge labeled with terminal t and from si to sn:

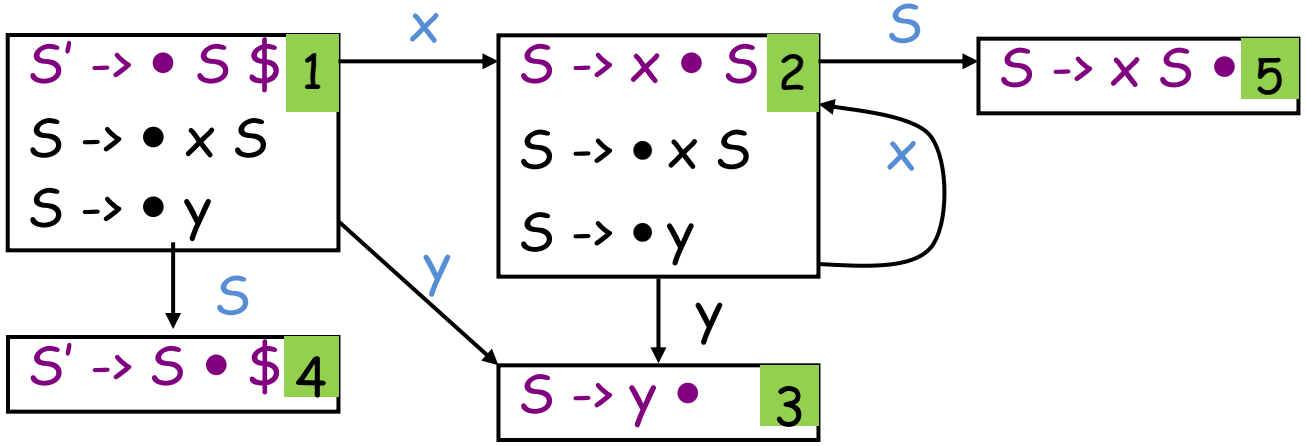
- **T[i, t] = sn** (shift n)

**Reduce:** Item in a state **i** with dot at the end

- **T[i, each terminal] = rk** (reduce k)
- k is the **index of this production**

# 从LR(0) Parsing DFA到语法分析表

0: S' -> S$
1: S -> x S
2: S -> y




| | Action | | | GOTO |
|---|---|---|---|---|
| State | x | y | $ | S |
| 1 | s2 | s3 | | g4 |
| 2 | s2 | s3 | | g5 |
| 3 | r2 | r2 | r2 | |
| 4 | | | accept | |
| 5 | r1 | r1 | r1 | |

**Shift**: edge labeled with terminal t and from si to sn:

- **T[i, t] = sn** (shift n)

**Reduce:** Item in a state **i** with dot at the end

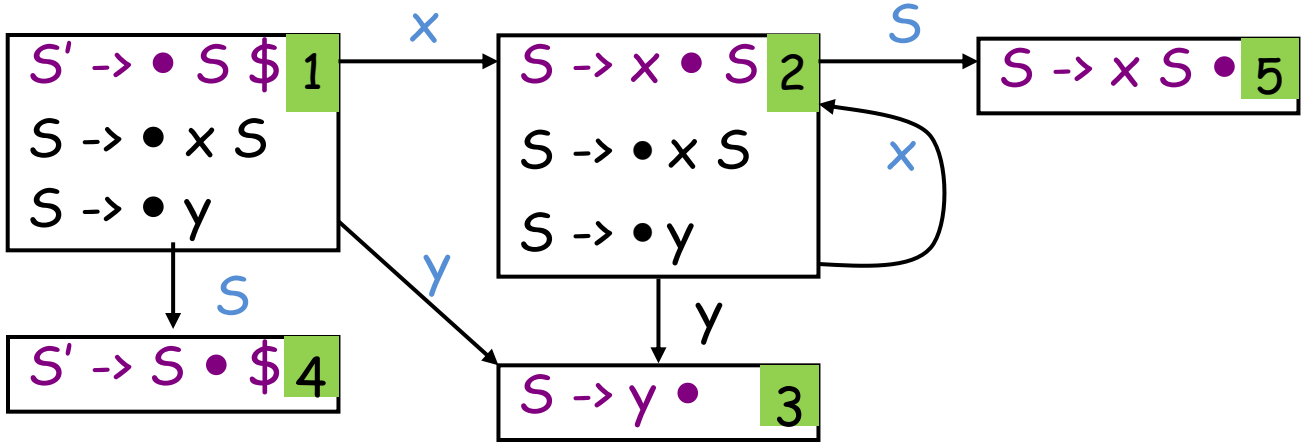
- **T[i, each terminal] = rk** (reduce k)
- k is the **index of this production**

**Accept:** for each state i containing S‘ -> S . $:

- **T[i, $] = accept**

# 从LR(0) Parsing DFA到语法分析表

0: S' -> S$
1: S -> x S
2: S -> y




| | Action | | | GOTO |
|---|---|---|---|---|
| State | x | y | $ | S |
| 1 | s2 | s3 | | g4 |
| 2 | s2 | s3 | | g5 |
| 3 | r2 | r2 | r2 | |
| 4 | | | accept | |
| 5 | r1 | r1 | r1 | |

**Shift**: edge labeled with terminal t and from si to sn:

- **T[i, t] = sn** (shift n)

**Reduce:** Item in a state **i** with dot at the end

- **T[i, each terminal] = rk** (reduce k)
- k is the **index of this production**

**Accept:** for each state i containing S‘ -> S . $:

- **T[i, $] = accept**

**Goto**: edge labeled with non-terminal X and from si to sn:

- **T[i, X] = gn** (goto n)

# LR 语法分析表的一些说明

## 注：下面的“栈”存的是LR Parsing DFA状态

| | Action | | | GOTO |
|---|---|---|---|---|
| State | x | y | $ | S |
| 1 | s2 | s3 | | g4 |
| 2 | s2 | s3 | | g5 |
| 3 | r2 | r2 | r2 | |
| 4 | | | accept | |
| 5 | r1 | r1 | r1 | |

两个部分：动作Action、转换GOTO

- **Action表项的参数：状态*i*，终结符号*a***
  - 移入*j*：*j*是新状态，把*j*压入栈
  - 归约$A \rightarrow \beta$：把$\beta$归约为$A$ (Pop栈顶状态, 并根据GOTO表压入新状态)
  - 接受：接受输入，完成分析
  - 报错：在输入中发现语法错误
- **GOTO表项的参数: 状态i, 非终结符A**
  - 若$\text{Goto}[I_i, A] = I_j$，则$\text{GOTO}[i, A] = j$

Goto函数

GOTO表项

# LR 语法分析算法

**Stack: LR Parsing DFA状态； Input Stream: 输入**

Look up **top stack state** and **input symbol**, to get action

| | |
|---|---|
| **Shift(n):** | • Advance input one token;<br>• Push n on stack. |
| **Reduce (k):** | • Pop stack as many times **as the number of symbols** on the RHS of rule k;<br>• Let X be the LHS symbol of rule k. In the state now on top of stack, look up X to get "goto n"; Push n on top of stack. |
| **Accept:** | Stop parsing, report success. |
| **Error:** | Stop parsing, report failure. |

Reduce(k) // k-th rule $\mathbf{X \rightarrow \alpha}$

**pop** $|\alpha|$ states

**push** GOTO[top(stack), X]

- This is a general algorithm and can be used by other LR parsing
- For LR(0), we do not need to look up input symbol to know whether we should shift or reduce

# LR 语法分析算法

- **输入：文法的LR语法分析表，输入串w**

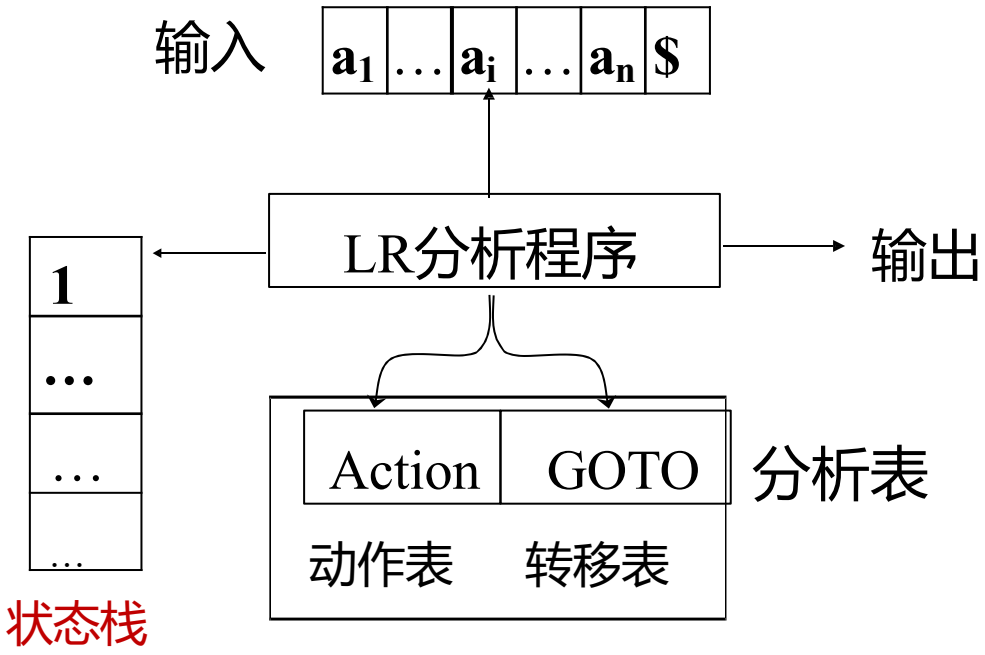
```
令s是栈顶状态, a是w$的第一个符号;
while (1) {  // 一开始s为Parsing DFA的状态1
    if (Action[s, a] = "shift s' ") {
        将s' 压入栈内；
        让a为下一个输入符号；
    } else if (Action[s, a] = "reduce A→β ") {
        从栈顶弹出|β|个状态；
        令s'是当前栈顶状态, 把GOTO[s', A]入栈；
     } else if (Action[s, a] = "Accept") break;
     else error( ); // 后面会讲，实际是调用错误恢复程序;
}
```

# 例: LR(0) 语法分析算法的动作

**注: 前述算法主要考虑状态栈，符号栈信息可相应状态中获取**

0: S' -> S$
1: S -> x S
2: S -> y

| | Action | | | GOTO |
|---|---|---|---|---|
| State | x | y | $ | S |
| 1 | s2 | s3 | | g4 |
| 2 | s2 | s3 | | g5 |
| 3 | r2 | r2 | r2 | |
| 4 | | | accept | |
| 5 | r1 | r1 | r1 | |




Shift: 假设读入y. 根据Action(1, y) = s3

- push状态3到状态栈栈顶

# 例: LR(0) 语法分析算法的动作

Reduce(k) // k-th rule $X \rightarrow \alpha$

**pop** $|\alpha|$ states

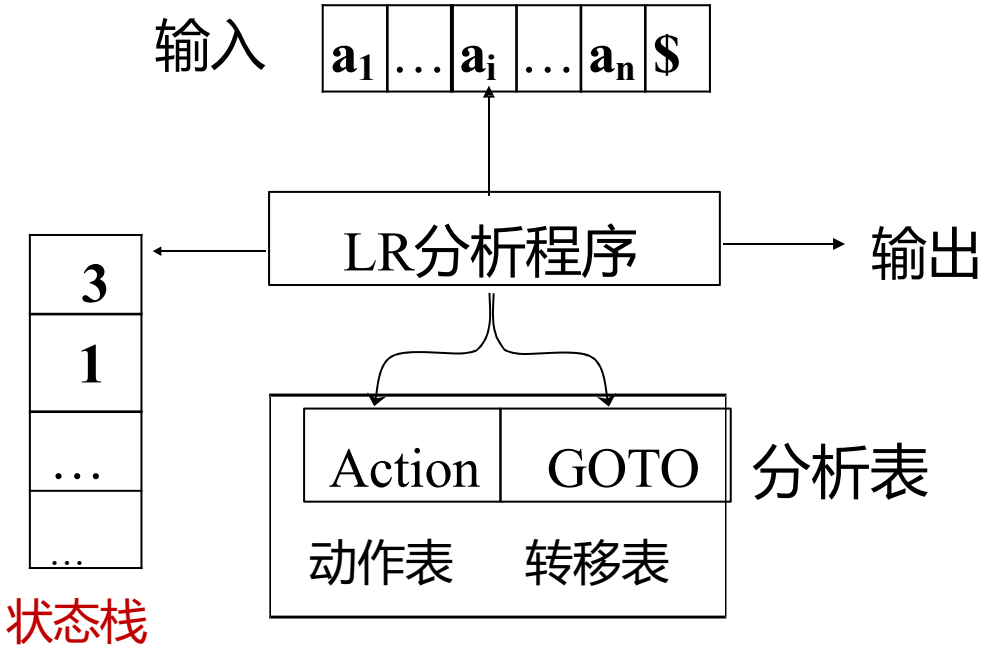
**push** GOTO[top(stack), X]

0: S' -> S$

1: S -> x S

2: S -> y

| | Action | | | GOTO |
|---|---|---|---|---|
| State | x | y | $ | S |
| 1 | s2 | s3 | | g4 |
| 2 | s2 | s3 | | g5 |
| 3 | r2 | r2 | r2 | |
| 4 | | | accept | |
| 5 | r1 | r1 | r1 | |

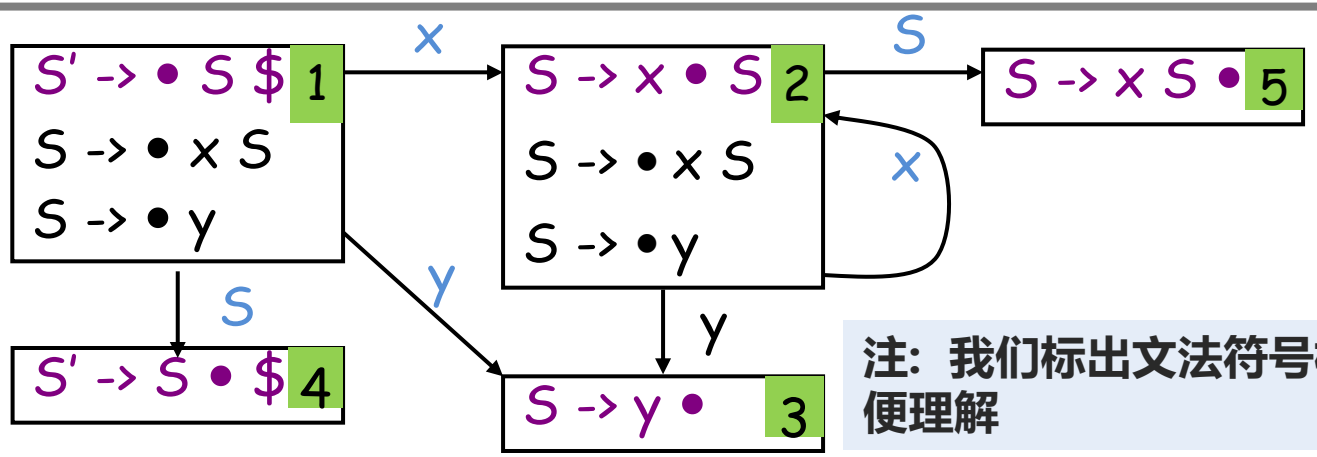



Reduce: 假设下一Token是x, 根据T(3, x) = r2

- Pop状态栈顶|RHS(S->y)| = 1个元素
- 状态栈顶变成1, 根据GOTO(1, S) = g4, push状态4到状态栈上

# 例: 完整的LR(0)语法分析

0: S' -> S$
1: S -> x S
2: S -> y




注: 我们标出文法符号栈以方便理解

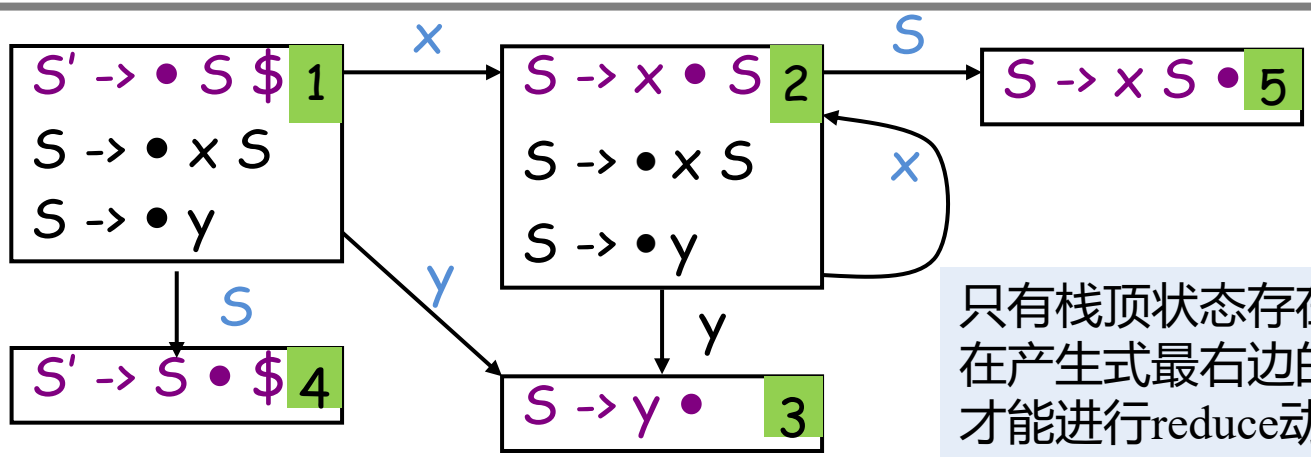
| | Action | | | GOTO |
|---|---|---|---|---|
| State | x | y | $ | S |
| 1 | s2 | s3 | | g4 |
| 2 | s2 | s3 | | g5 |
| 3 | r2 | r2 | r2 | |
| 4 | | | accept | |
| 5 | r1 | r1 | r1 | |

| Stack (states) | Stack (symbols) | Input | Action |
|---|---|---|---|
| 1 | | x x y $ | shift 2 |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

Shift 2: 读入x , 状态栈压入2

# 例: 完整的LR(0)语法分析

0: S' -> S$
1: S -> x S
2: S -> y




只有栈顶状态存在"•"在产生式最右边的Item，才能进行reduce动作

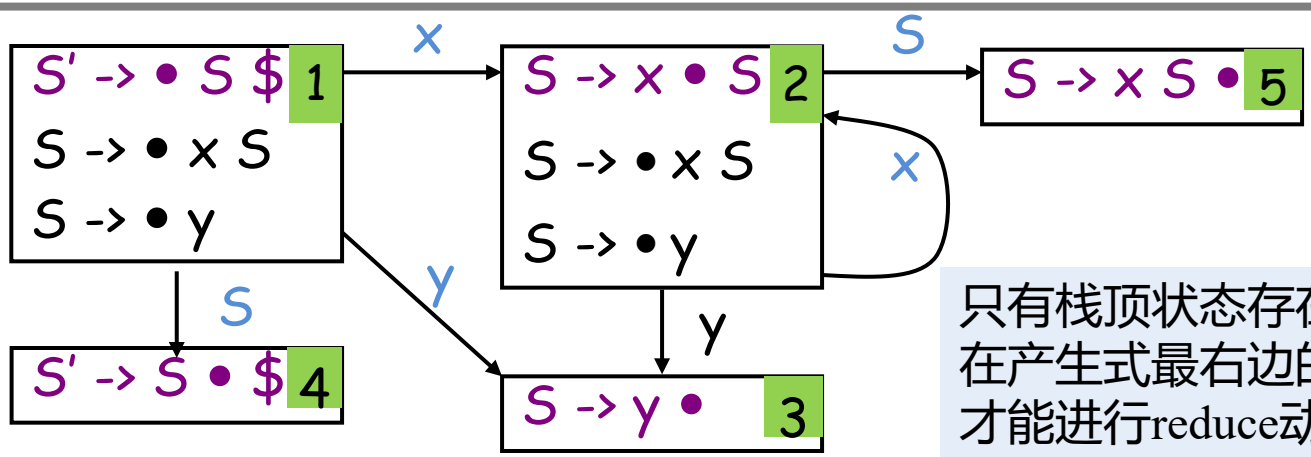
| | Action | | | GOTO |
|---|---|---|---|---|
| State | x | y | $ | S |
| 1 | s2 | s3 | | g4 |
| 2 | s2 | s3 | | g5 |
| 3 | r2 | r2 | r2 | |
| 4 | | | accept | |
| 5 | r1 | r1 | r1 | |

| Stack (states) | Stack (symbols) | Input | Action |
|---|---|---|---|
| **1** | | **x x y $** | **shift 2** |
| **1 2** | **x** | **x y $** | **shift 2** |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

**Shift 2: 读入x , 状态栈压入2**

# 例: 完整的LR(0)语法分析

0: S' -> S$
1: S -> x S
2: S -> y




只有栈顶状态存在"•"在产生式最右边的Item，才能进行reduce动作

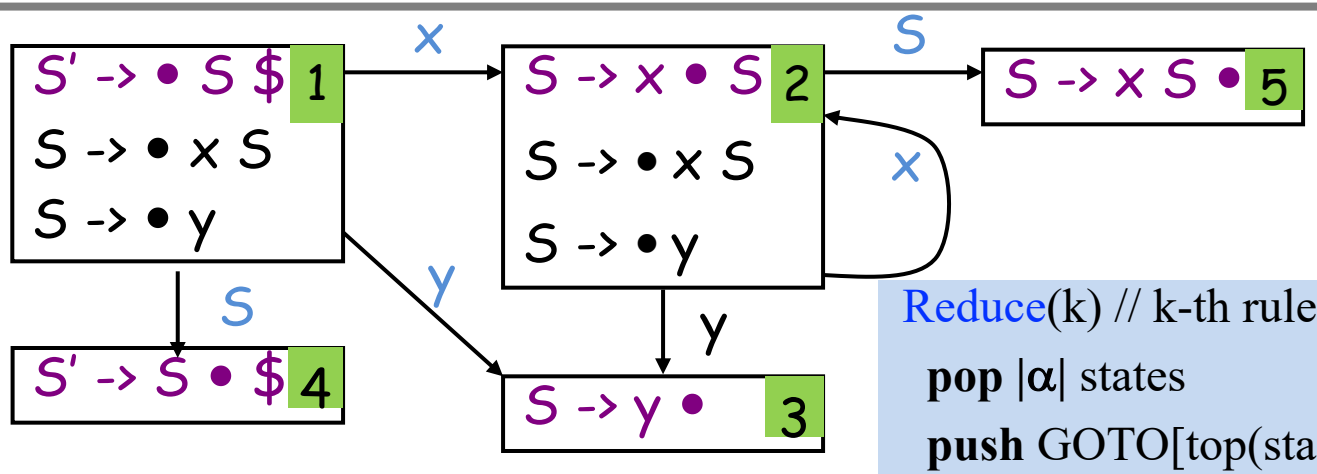
| | Action | | | GOTO |
|---|---|---|---|---|
| State | x | y | $ | S |
| 1 | s2 | s3 | | g4 |
| 2 | s2 | s3 | | g5 |
| 3 | r2 | r2 | r2 | |
| 4 | | | accept | |
| 5 | r1 | r1 | r1 | |

| Stack (states) | Stack (symbols) | Input | Action |
|---|---|---|---|
| **1** | | **x x y $** | **shift 2** |
| **1 2** | **x** | **x y $** | **shift 2** |
| **1 2 2** | **x x** | **y $** | **shift 3** |
| | | | |
| | | | |
| | | | |
| | | | |

**Shift 3: 读入y，状态栈压入3**

# 例: 完整的LR(0)语法分析

0: S' -> S$
1: S -> x S
2: S -> y




Reduce(k) // k-th rule **X** → α
**pop** |α| states
**push** GOTO[top(stack), X]

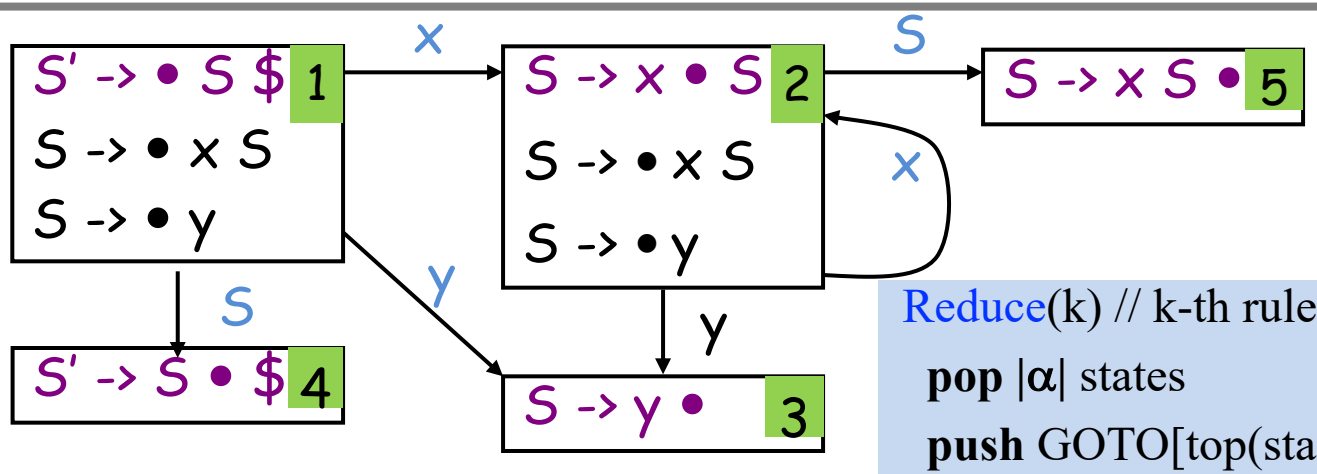
| | Action | | | GOTO |
|---|---|---|---|---|
| State | x | y | $ | S |
| 1 | s2 | s3 | | g4 |
| 2 | s2 | s3 | | g5 |
| 3 | r2 | r2 | r2 | |
| 4 | | | accept | |
| 5 | r1 | r1 | r1 | |

| Stack (states) | Stack (symbols) | Input | Action |
|---|---|---|---|
| 1 | | x x y $ | shift 2 |
| 1 2 | x | x y $ | shift 2 |
| 1 2 2 | x x | y $ | shift 3 |
| 1 2 2 3 | x x y | $ | reduce 2 (S -> y) |
| | | | |
| | | | |
| | | | |

**Reduce: 1. 状态栈弹出3(|RHS|个状态)； 2. 根据Goto(2, S) =5 压入状态5**

# 例: 完整的LR(0)语法分析

0: S' -> S$
1: S -> x S
2: S -> y




Reduce(k) // k-th rule **X → α**
**pop** |α| states
**push** GOTO[top(stack), X]

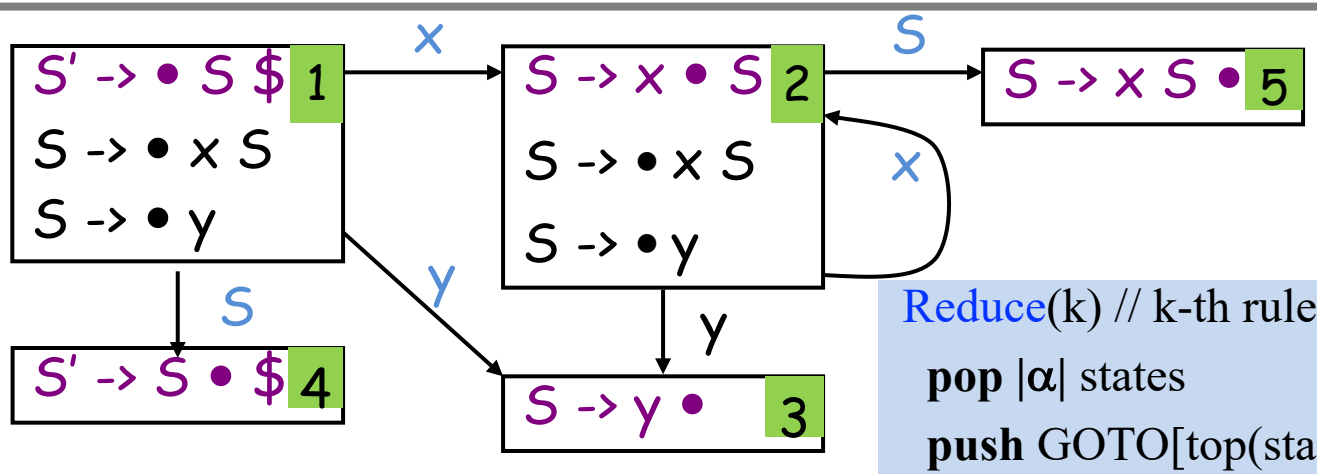
| | Action | | | GOTO |
|---|---|---|---|---|
| State | x | y | $ | S |
| 1 | s2 | s3 | | g4 |
| 2 | s2 | s3 | | g5 |
| 3 | r2 | r2 | r2 | |
| 4 | | | accept | |
| 5 | r1 | r1 | r1 | |

| Stack (states) | Stack (symbols) | Input | Action |
|---|---|---|---|
| 1 | | x x y $ | shift 2 |
| 1 2 | x | x y $ | shift 2 |
| 1 2 2 | x x | y $ | shift 3 |
| 1 2 2 3 | x x y | $ | reduce 2 (S -> y) |
| 1 2 2 5 | x x S | $ | reduce 1 (S -> x S) |
| | | | |
| | | | |

**Reduce: 1. 状态栈弹出5, 2(|RHS|个状态)； 2. 根据Goto(2, S) =5 压入状态5**

# 例: 完整的LR(0)语法分析

0: S' -> S$
1: S -> x S
2: S -> y




Reduce(k) // k-th rule **X** → α
**pop** |α| states
**push** GOTO[top(stack), X]

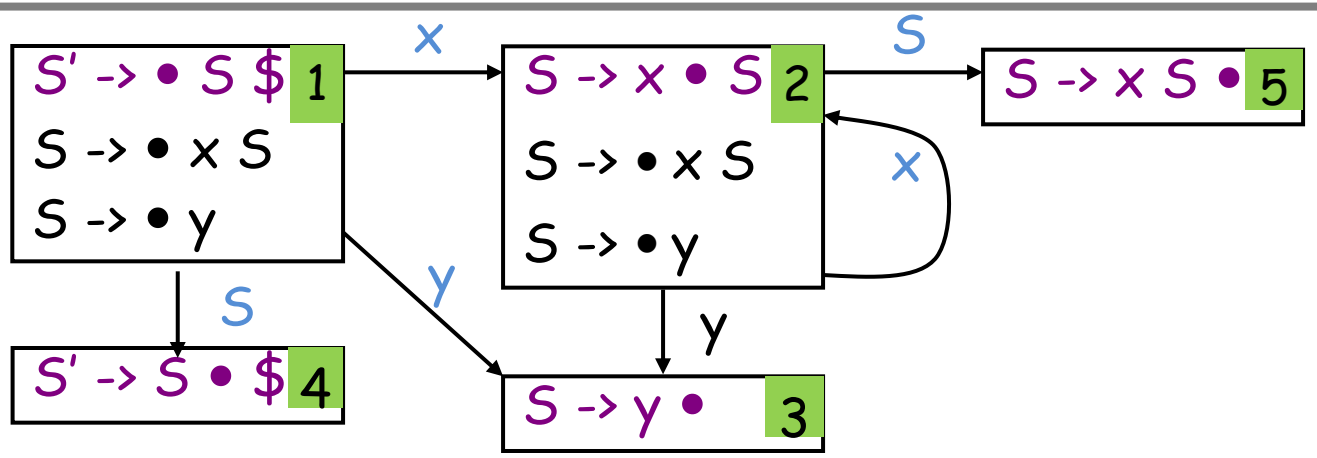
| | Action | | | GOTO |
|---|---|---|---|---|
| State | x | y | $ | S |
| 1 | s2 | s3 | | g4 |
| 2 | s2 | s3 | | g5 |
| 3 | r2 | r2 | r2 | |
| 4 | | | accept | |
| 5 | r1 | r1 | r1 | |

| Stack (states) | Stack (symbols) | Input | Action |
|---|---|---|---|
| 1 | | x x y $ | shift 2 |
| 1 2 | x | x y $ | shift 2 |
| 1 2 2 | x x | y $ | shift 3 |
| 1 2 2 3 | x x y | $ | reduce 2 (S -> y) |
| 1 2 2 5 | x x S | $ | reduce 1 (S -> x S) |
| 1 2 5 | x S | $ | reduce 1 (S -> x S) |
| | | | |

**Reduce: 1. 状态栈弹出5, 2(|RHS|个状态)； 2. 根据Goto(1, S) =4 压入状态4**

# 例: 完整的LR(0)语法分析

0: S' -> S$
1: S -> x S
2: S -> y




| | Action | | | GOTO |
|---|---|---|---|---|
| State | x | y | $ | S |
| 1 | s2 | s3 | | g4 |
| 2 | s2 | s3 | | g5 |
| 3 | r2 | r2 | r2 | |
| 4 | | | accept | |
| 5 | r1 | r1 | r1 | |

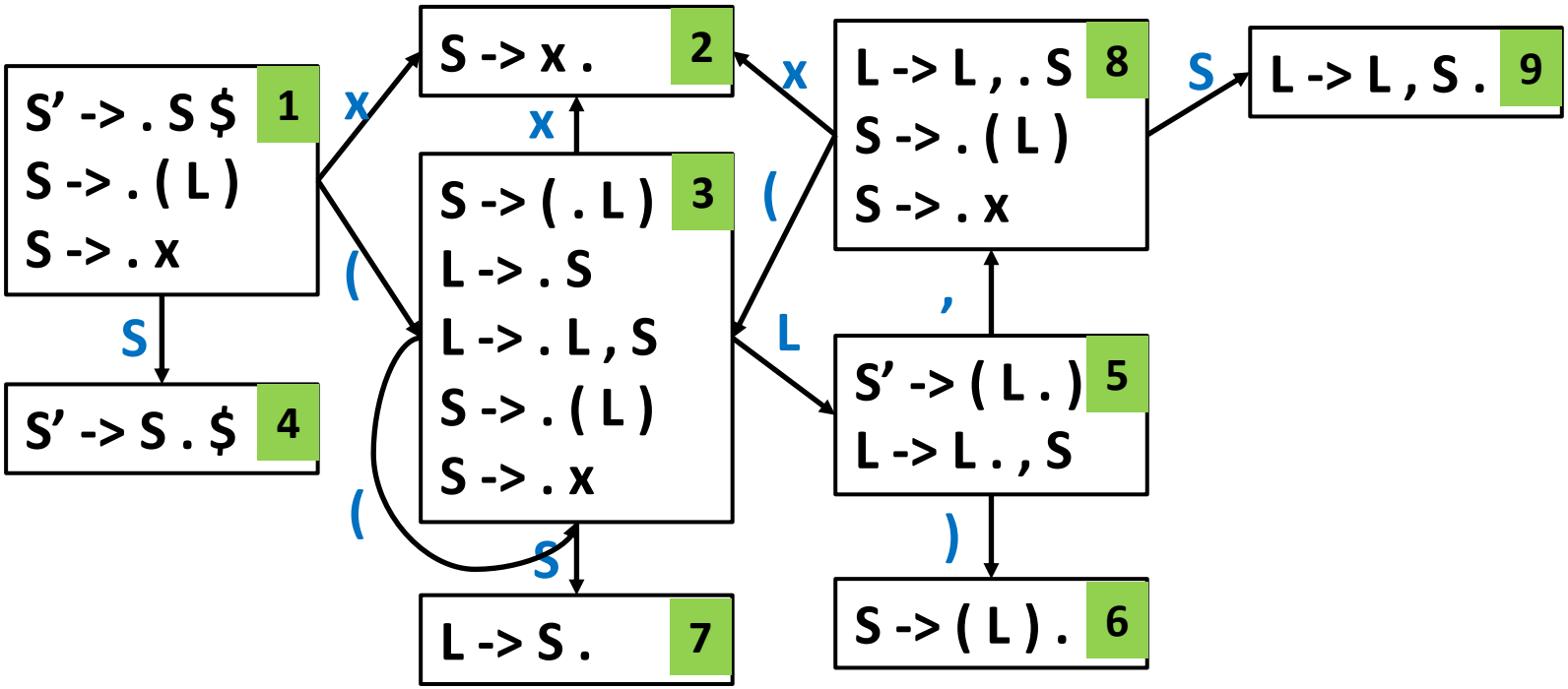
| Stack (states) | Stack (symbols) | Input | Action |
|---|---|---|---|
| 1 | | x x y $ | shift 2 |
| 1 2 | x | x y $ | shift 2 |
| 1 2 2 | x x | y $ | shift 3 |
| 1 2 2 3 | x x y | $ | reduce 2 (S -> y) |
| 1 2 2 5 | x x S | $ | reduce 1 (S -> x S) |
| 1 2 5 | x S | $ | reduce 1 (S -> x S) |
| 1 4 | S | $ | accept |

**Accept:** 栈顶状态4报包含项S' -> S • $，因此Accept

# 练习: LR(0)分析-Parsing DFA构造

0: S' -> S$
1: S -> (L)
2: S -> x
3: L -> S
4: L -> L, S

Initialize $T$ to {**Closure**({S ′ → .S$})}
Initialize $E$ to empty.
**repeat**
  **for** each state $I$ in $T$
    **for** each item A → α.Xβ in $I$
      let $J$ be ***Goto***$(I, X)$
      $T \leftarrow T \cup \{J\}$
      E ← E ∪ {I $\xrightarrow{X}$ J}
**until** E and T did not change in this iteration



# 练习: LR(0)分析-Parsing DFA到语法分析表

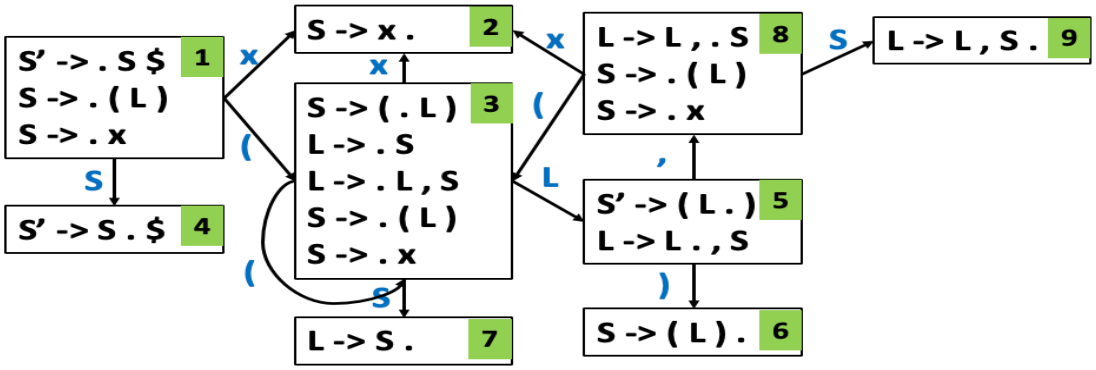
0: S' -> S$

1: S -> (L)

2: S -> x

3: L -> S

4: L -> L, S




| | Action | | | | | GOTO | |
|---|---|---|---|---|---|---|---|
| | ( | ) | x | , | $ | S | L |
| 1 | s3 | | s2 | | | g4 | |
| 2 | r2 | r2 | r2 | r2 | r2 | | |
| 3 | s3 | | s2 | | | g7 | g5 |
| 4 | | | | | a | | |
| 5 | | s6 | | s8 | | | |
| 6 | r1 | r1 | r1 | r1 | r1 | | |
| 7 | r3 | r3 | r3 | r3 | r3 | | |
| 8 | s3 | | s2 | | | g9 | |
| 9 | r4 | r4 | r4 | r4 | r4 | | |

# 练习: LR(0)分析-语法分析表的使用

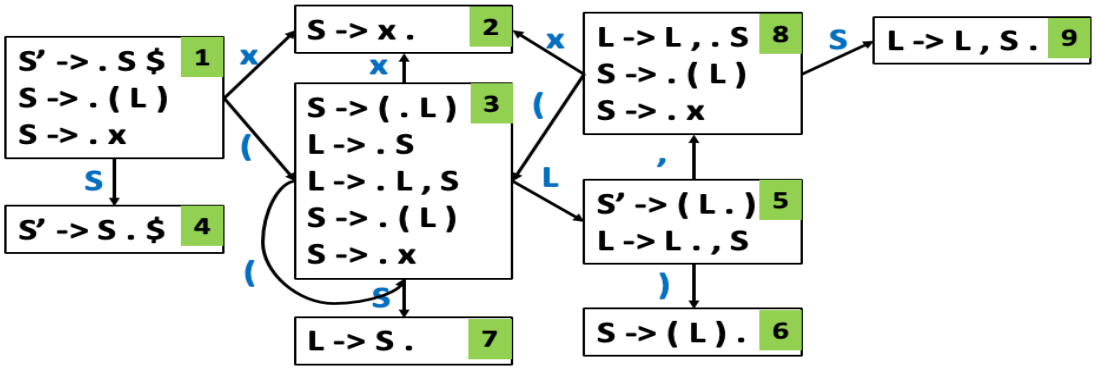
0: S' -> S$

1: S -> (L)

2: S -> x

3: L -> S

4: L -> L, S




| | Action | | | | | GOTO | |
|---|---|---|---|---|---|---|---|
| | ( | ) | x | , | $ | S | L |
| 1 | s3 | | s2 | | | g4 | |
| 2 | r2 | r2 | r2 | r2 | r2 | | |
| 3 | s3 | | s2 | | | g7 | g5 |
| 4 | | | | | a | | |
| 5 | | s6 | | s8 | | | |
| 6 | r1 | r1 | r1 | r1 | r1 | | |
| 7 | r3 | r3 | r3 | r3 | r3 | | |
| 8 | s3 | | s2 | | | g9 | |
| 9 | r4 | r4 | r4 | r4 | r4 | | |

| Stack (states) | Stack (symbols) | Input | Action |
|---|---|---|---|
| 1 | | ( x ) $ | shift 3 |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

# 练习: LR(0)分析-语法分析表的使用

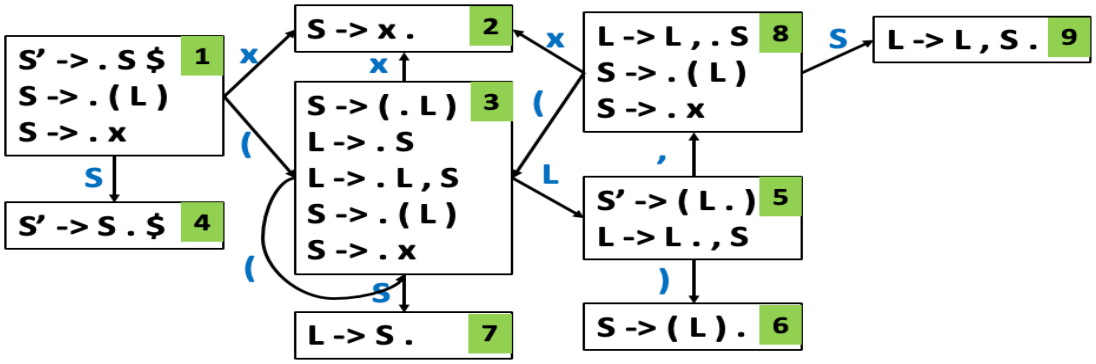
0: S' -> S$

1: S -> (L)

2: S -> x

3: L -> S

4: L -> L, S



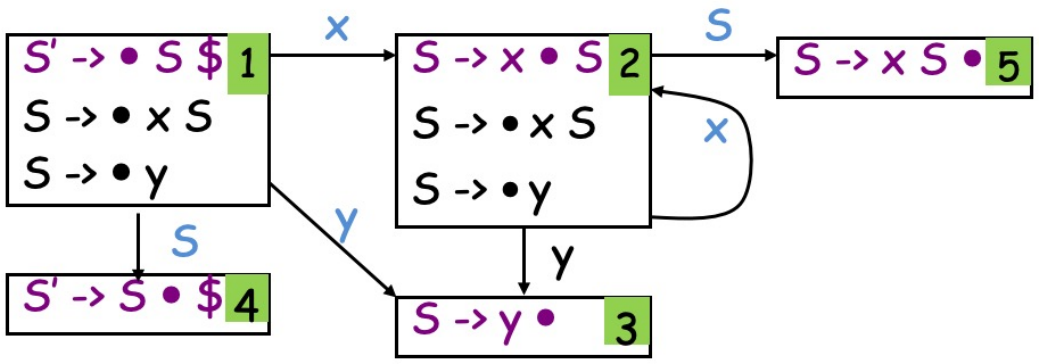

| | Action | | | | | GOTO | |
|---|---|---|---|---|---|---|---|
| | ( | ) | x | , | $ | S | L |
| 1 | s3 | | s2 | | | g4 | |
| 2 | r2 | r2 | r2 | r2 | r2 | | |
| 3 | s3 | | s2 | | | g7 | g5 |
| 4 | | | | | a | | |
| 5 | | s6 | | s8 | | | |
| 6 | r1 | r1 | r1 | r1 | r1 | | |
| 7 | r3 | r3 | r3 | r3 | r3 | | |
| 8 | s3 | | s2 | | | g9 | |
| 9 | r4 | r4 | r4 | r4 | r4 | | |

| Stack (states) | Stack (symbols) | Input | Action |
|---|---|---|---|
| 1 | | ( x ) $ | shift 3 |
| 1, 3 | ( | x ) $ | shift 2 |
| 1, 3, 2 | ( x | ) $ | reduce 2 S -> x |
| 1, 3 | ( S | ) $ | goto 7 |
| 1, 3, 7 | ( S | ) $ | reduce 3 L -> S |
| 1, 3 | ( L | ) $ | goto 5 |
| 1, 3, 5 | ( L | ) $ | shift 6 |
| 1, 3, 5, 6 | ( L ) | $ | reduce 1 S -> ( L ) |
| 1 | S | $ | goto 4 |
| 1, 4 | S | $ | accept |

# 思考题: 关于LR(0)中的“0“?

- 为什么说LR(0)没有Lookahead?
  - LR(0) Item中没有lookahead terminal等信息
  - “是否归约、选择哪个产生式规约”仅由栈顶状态决定

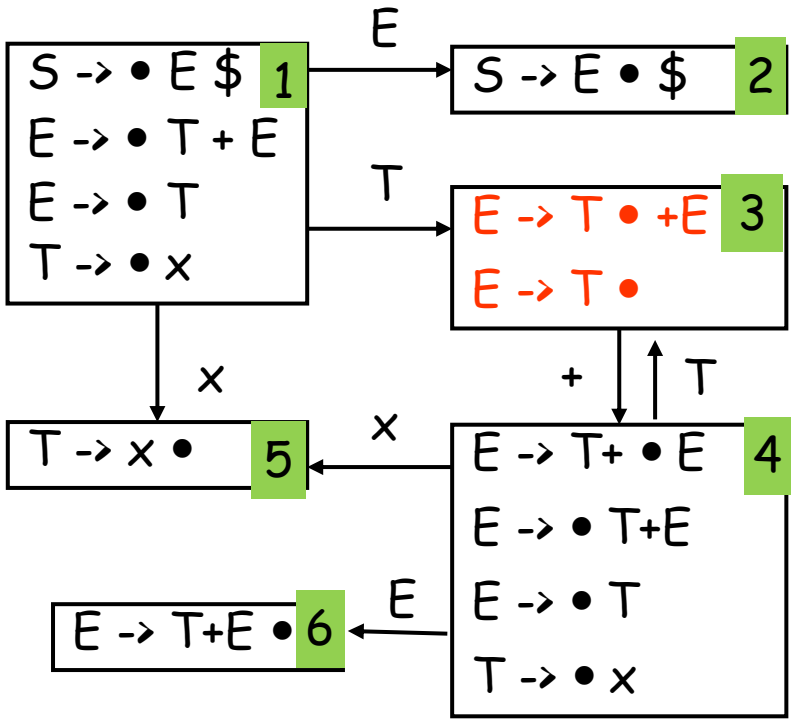
| | Action | | | GOTO |
|---|---|---|---|---|
| State | x | y | $ | S |
| 1 | s2 | s3 | | g4 |
| 2 | s2 | s3 | | g5 |
| 3 | r2 | r2 | r2 | |
| 4 | | | accept | |
| 5 | r1 | r1 | r1 | |



# LR(0)的问题/局限性

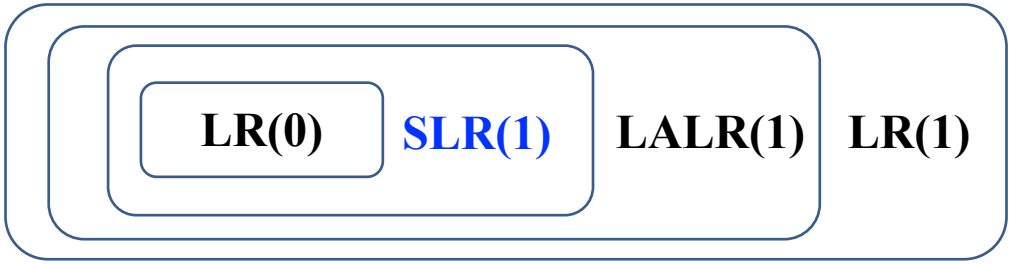
0: S -> E$
1: E -> T+E
2: E -> T
3: T -> x

- **For every item of the form: X -> α • , blindly reduce α to X**
- May lead to conflicts




| | Action | | | GOTO | |
|---|---|---|---|---|---|
| s\t | x | + | $ | E | T |
| 1 | s5 | | | g2 | g3 |
| 2 | | | accept | | |
| 3 | r2 | s4, r2 | r2 | | |
| 4 | s5 | | | g6 | g3 |
| 5 | r3 | r3 | r3 | | |
| 6 | r1 | r1 | r1 | | |

A shift-reduce conflict (on state 3)!




# 3. SLR(1)分析

# SLR(1) Parsing思路

- **利用更多信息来指导规约操作**
- **回顾："LR分析是最右推导的逆过程"**

$$S \Rightarrow_{rm} aABe \Rightarrow_{rm} aEte \Rightarrow_{rm} aDbcte \Rightarrow_{rm} abbcde$$

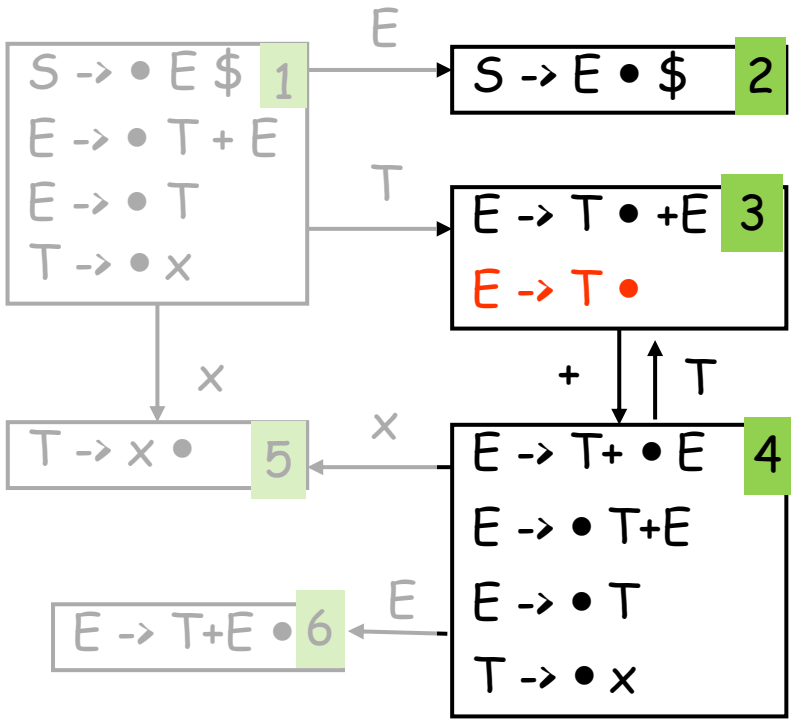
- **因此, 每步归约都应该满足 $t \in Follow(E)$**
  - E是用来归约的产生式的左部
  - t是Next Token

$$Follow(A) = \{a \mid S \Rightarrow^* \ldots Aa \ldots,\ a \in T\}$$

# 例: SLR(1) Parsing思路

0: S -> E$
1: E -> T+E
2: E -> T
3: T -> x

- What are the valid next tokens for r2?
  - **Idea**: we can choose the reduce action only if the next input token t ∈ Follow(E) ={$}
  - **Only T[3, $] can be r2!**




| | Action | | | GOTO |
|---|---|---|---|---|
| State | x | y | $ | S |
| 1 | s2 | s3 | | g4 |
| 2 | s2 | s3 | | g5 |
| 3 | ~~r2~~ | ~~r2~~ | r2 | |
| 4 | | | accept | |
| 5 | r1 | r1 | r1 | |

# SLR(1) Parsing分析表

- SLR(1) Parsing DFA和LR(0)的相同
- 主要区别: 构造分析表时的归约动作
  - SLR: Put reduce actions into the table only where indicated by the FOLLOW set.

SLR(1)

$R \leftarrow \{\}$ // 规约动作集合
**for** each state $I$ in $T$
  **for** each item $A \rightarrow \boldsymbol{\alpha}.$ in $I$
    **for** each token $X$ **in Follow(**$A$**)**
      $R \leftarrow R \cup \{(I, X, A \rightarrow \alpha)\}$

LR(0)

$R \leftarrow \{\}$
**for** each state $I$ in $T$
  **for** each item $A \rightarrow \boldsymbol{\alpha}.$ in $I$
    $R \leftarrow R \cup \{(I, A \rightarrow \alpha)\}$

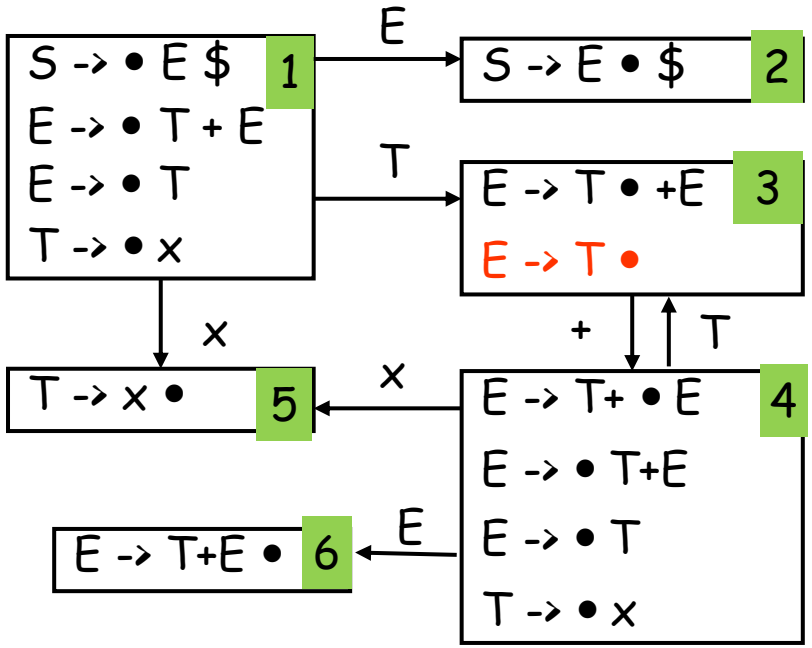
I: the row,
X: the column
$A \rightarrow \alpha$: the production

# 例: 构建SLR(1)分析表

0: S -> E$
1: E -> T+E
2: E -> T
3: T -> x

**Follow (E) = {$}**

Follow (T) = {+, $}




考虑状态3的E -> T. (用”—”划掉的是按LR(0)填的)

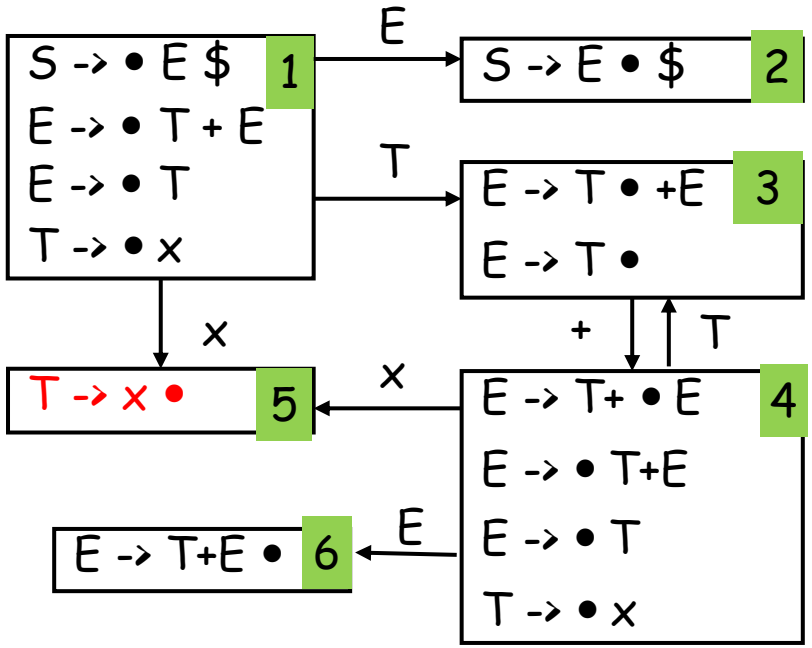
| | Action | | | GOTO | |
|---|---|---|---|---|---|
| s\t | x | + | $ | E | T |
| 1 | s5 | | | g2 | g3 |
| 2 | | | accept | | |
| 3 | ~~r2~~ | s4, ~~r2~~ | r2 | | |
| 4 | s5 | | | g6 | g3 |
| 5 | ~~r3~~ | r3 | r3 | | |
| 6 | ~~r1~~ | ~~r1~~ | r1 | | |

# 例: 构建SLR(1)分析表

0: S -> E$
1: E -> T+E
2: E -> T
3: T -> x

Follow (E) = {$}

Follow (T) = {+, $}




考虑状态5的T -> x.

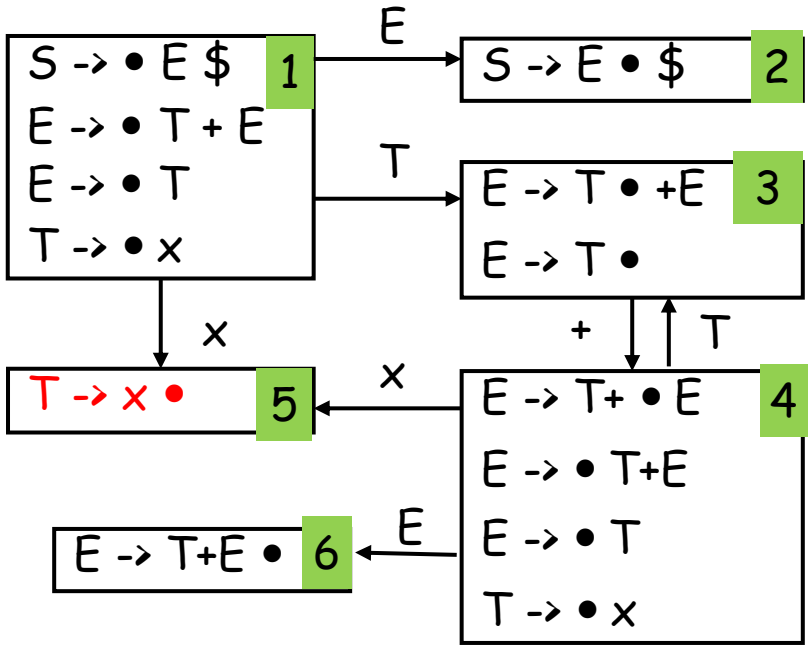
| | Action | | | GOTO | |
|---|---|---|---|---|---|
| s\t | x | + | $ | E | T |
| 1 | s5 | | | g2 | g3 |
| 2 | | | accept | | |
| 3 | ~~r2~~ | s4, ~~r2~~ | r2 | | |
| 4 | s5 | | | g6 | g3 |
| 5 | ~~r3~~ | r3 | r3 | | |
| 6 | ~~r1~~ | ~~r1~~ | r1 | | |

# 例: 构建SLR(1)分析表

0: S -> E$
1: E -> T+E
2: E -> T
3: T -> x

Follow (E) = {$}

Follow (T) = {+, $}




考虑状态6的E -> T + E.

| | Action | | | GOTO | |
|---|---|---|---|---|---|
| s\t | x | + | $ | E | T |
| 1 | s5 | | | g2 | g3 |
| 2 | | | accept | | |
| 3 | ~~r2~~ | s4, ~~r2~~ | r2 | | |
| 4 | s5 | | | g6 | g3 |
| 5 | ~~r3~~ | r3 | r3 | | |
| 6 | ~~r1~~ | ~~r1~~ | r1 | | |

# SLR(1)的问题/局限

- **SLR技术解决冲突的方法**
  - 按照$A \to \alpha$进行归约的条件是：下一个输入符号$x$可以在某个句型中跟在$A$之后，也就是$x \in Follow(A)$

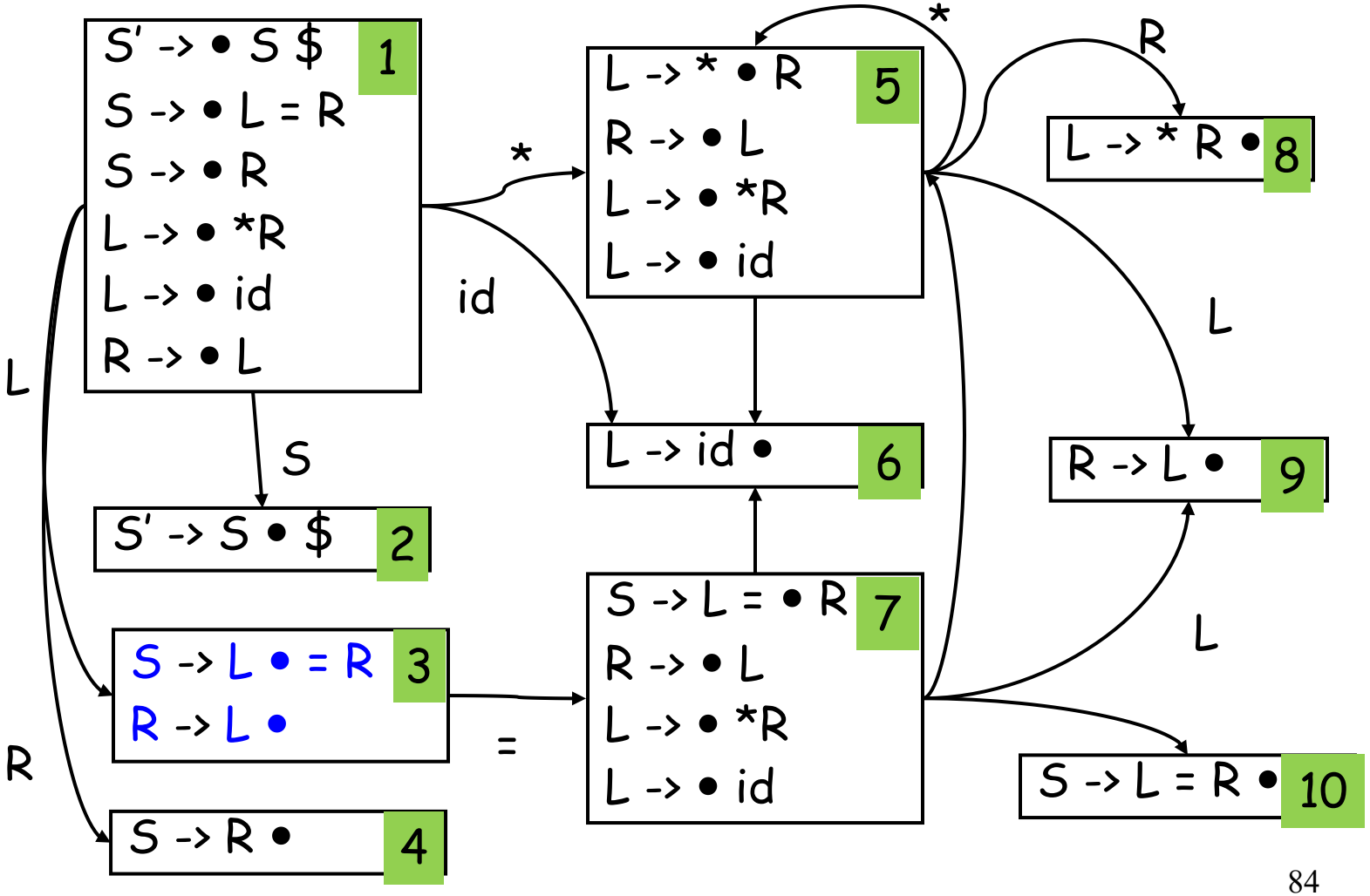
- **LR分析过程是最右推导的逆**
  - 每一步都应该是最右句型
  - 如果$\beta Ax$不是任何最右句型的前缀，那么即使$x$在某个句型中跟在$A$之后，仍不应该按$A \to \alpha$归约
  - (利用Follow集可看作"可能不够精确的近似")

进行归约的条件更加严格才能进一步降低冲突的可能

# SLR(1)的问题/局限

- Follow(R) = Follow(L)
- Thus, there exists shift-reduce conflict in state 2

S' -> S$
S -> L = R
| R
L -> * R
| id
R -> L




**Thank you all for your attention**