

语法分析

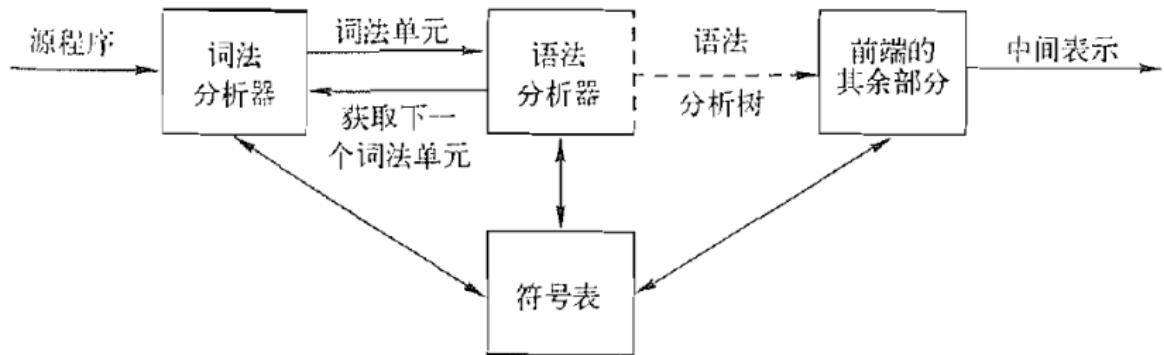
魏恒峰

hfwei@nju.edu.cn

2020 年 12 月 8 日



输入: 词法单元流 & 语言的语法规则



输出: 语法分析树 (Parse Tree)

语法分析举例

```
(Stmt) → <Id> = <Expr> ;
(Stmt) → { <StmtList> }
<Stmt> → if ( <Expr> ) <Stmt>
<StmtList> → <Stmt>
<StmtList> → <StmtList> <Stmt>
<Expr> → <Id>
<Expr> → <Num>
<Expr> → <Expr> <Optr> <Expr>
<Id> → x
<Id> → y
<Num> → 0
<Num> → 1
<Num> → 9
<Optr> → +
<Optr> → *
```

<Stmt>	
if (<Expr>)
if (<Expr> <Optr> <Expr>)
if (<Id> <Optr> <Expr>)
if (x <Optr> <Expr>)
if (x > <Expr>)
if (x > <Num>)
if (x > 9)
if (x > 9) {
	<StmtList> }
if (x > 9) {
	<Stmt> }
if (x > 9) {
	<Id> = <Expr> ;
if (x = <Expr> ;
if (x = <Num> ;
if (x = 0 ;
if (x = 0 ; <Id> = <Expr> ;
if (x = 0 ; y = <Expr> ;
if (x = 0 ; y = <Expr> <Optr> <Expr> ;
if (x = 0 ; y = <Id> <Optr> <Expr> ;
if (x = 0 ; y = y <Optr> <Expr> ;
if (x = 0 ; y = y + <Expr> ;
if (x = 0 ; y = y + <Num> ;
if (x = 0 ; y = y + 1 ; }

语法分析阶段的主题之一：上下文无关文法

```
⟨Stmt⟩ → ⟨Id⟩ = ⟨Expr⟩ ;
⟨Stmt⟩ → { ⟨StmtList⟩ }
⟨Stmt⟩ → if ( ⟨Expr⟩ ) ⟨Stmt⟩
⟨StmtList⟩ → ⟨Stmt⟩
⟨StmtList⟩ → ⟨StmtList⟩ ⟨Stmt⟩
⟨Expr⟩ → ⟨Id⟩
⟨Expr⟩ → ⟨Num⟩
⟨Expr⟩ → ⟨Expr⟩ ⟨Optr⟩ ⟨Expr⟩
⟨Id⟩ → x
⟨Id⟩ → y
⟨Num⟩ → 0
⟨Num⟩ → 1
⟨Num⟩ → 9
⟨Optr⟩ → >
⟨Optr⟩ → +
```

语法分析阶段的主题之二：构建语法分析树

$\langle \text{Stmt} \rangle$

```

if (   <Expr> ) <Stmt>
if ( <Expr> <Optr> <Expr> ) <Stmt>
if ( <Id> <Optr> <Expr> ) <Stmt>
if ( x <Optr> <Expr> ) <Stmt>
if ( x > <Expr> ) <Stmt>
if ( x > <Num> ) <Stmt>
if ( x > 9 ) <Stmt>
if ( x > 9 ) { <StmtList> }
if ( x > 9 ) { <StmtList> } <Stmt>
if ( x > 9 ) { <Stmt> } <Stmt>
if ( x > 9 ) { <Id> = <Expr> ; } <Stmt>
if ( x > 9 ) { x = <Expr> ; } <Stmt>
if ( x > 9 ) { x = <Num> ; } <Stmt>
if ( x > 9 ) { x = 0 ; } <Stmt>
if ( x > 9 ) { x = 0 ; <Id> = <Expr> ; }
if ( x > 9 ) { x = 0 ; y = <Expr> ; }
if ( x > 9 ) { x = 0 ; y = <Expr> <Optr> <Expr> ; }
if ( x > 9 ) { x = 0 ; y = <Id> <Optr> <Expr> ; }
if ( x > 9 ) { x = 0 ; y = y <Optr> <Expr> ; }
if ( x > 9 ) { x = 0 ; y = y + <Expr> ; }
if ( x > 9 ) { x = 0 ; y = y + <Num> ; }
if ( x > 9 ) { x = 0 ; y = y + 1 ; }

```

语法分析阶段的主题之三: 错误恢复



报错、**恢复**、继续分析



上下文无关文法

Definition (Context-Free Grammar (CFG); 上下文无关文法)

上下文无关文法 G 是一个四元组 $G = (T, N, P, S)$:

- ▶ T 是**终结符号** (Terminal) 集合, 对应于词法分析器产生的词法单元;
- ▶ N 是**非终结符号** (Non-terminal) 集合;
- ▶ P 是**产生式** (Production) 集合;

$$A \in N \longrightarrow \alpha \in (T \cup N)^*$$

头部/左部 (Head) A : **单个**非终结符

体部/右部 (Body) α : 终结符与非终结符构成的串, 也可以是空串 ϵ

- ▶ S 为**开始** (Start) 符号。要求 $S \in N$ 且唯一。

$$G = (\{a, b\}, \{S\}, P, S)$$

$$\boxed{\begin{array}{c} S \rightarrow aSb \\ S \rightarrow \epsilon \end{array}}$$

$$G = (\{((),)\}, \{S\}, P, S)$$

$S \rightarrow SS$

$S \rightarrow (S)$

$S \rightarrow ()$

$S \rightarrow \epsilon$

$stmt \rightarrow if\ expr\ then\ stmt$

| $if\ expr\ then\ stmt\ else\ stmt$

| **other**

条件语句文法

悬空 (Dangling)-else 文法

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$
 $S \rightarrow \text{begin } S \ L$
 $S \rightarrow \text{print } E$

$L \rightarrow \text{end}$
 $L \rightarrow ; \ S \ L$
 $E \rightarrow \text{num} \ = \ \text{num}$

约定: 如果没有明确指定, 第一个产生式的头部就是开始符号

关于**终结符号**的约定

1) 下述符号是终结符号:

- ① 在字母表里排在前面的小写字母, 比如 a 、 b 、 c 。
- ② 运算符号, 比如 $+$ 、 $*$ 等。
- ③ 标点符号, 比如括号、逗号等。
- ④ 数字 0 、 1 、 \cdots 、 9 。
- ⑤ **黑体字符串**, 比如 **id** 或 **if**。每个这样的字符串表示一个终结符号。

关于非终结符号的约定

2) 下述符号是非终结符号:

- ① 在字母表中排在前面的大写字母, 比如 A 、 B 、 C 。
- ② **字母 S** 。它出现时通常表示开始符号。
- ③ 小写、斜体的名字, 比如 $expr$ 或 $stmt$ 。

Syntax Semantics

语义: 上下文无关文法 G 定义了一个**语言** $L(G)$

Syntax Semantics

语义: 上下文无关文法 G 定义了一个**语言** $L(G)$

语言是**串**的集合

串从何来?

推导 (Derivation)

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$$

推导即是将某个产生式的左边**替换**成它的右边

每一步推导需要选择**替换哪个非终结符号**, 以及**使用哪个产生式**

推导 (Derivation)

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

$E \implies -E \implies -(E) \implies -(E+E) \implies -(\mathbf{id}+E) \implies -(\mathbf{id}+\mathbf{id})$

推导 (Derivation)

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$$

$E \implies -E \implies -(E) \implies -(E+E) \implies -(\text{id}+E) \implies -(\text{id}+\text{id})$

$E \implies -E$: 经过一步推导得出

$E \stackrel{+}{\implies} -(\text{id}+E)$: 经过一步或多步推导得出

$E \stackrel{*}{\implies} -(\text{id}+E)$: 经过零步或多步推导得出

推导 (Derivation)

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$$

$E \implies -E \implies -(E) \implies -(E+E) \implies -(\text{id}+E) \implies -(\text{id}+\text{id})$

$E \implies -E$: 经过一步推导得出

$E \stackrel{+}{\implies} -(\text{id}+E)$: 经过一步或多步推导得出

$E \stackrel{*}{\implies} -(\text{id}+E)$: 经过零步或多步推导得出

$E \implies -E \implies -(E) \implies -(E+E) \implies -(\textcolor{blue}{E+\text{id}}) \implies -(\text{id}+\text{id})$

Definition (Sentential Form; 句型)

如果 $S \xrightarrow{*} \alpha$, 且 $\alpha \in (T \cup N)^*$, 则称 α 是文法 G 的一个**句型**。

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$$

$E \implies -E \implies -(E) \implies -(E+E) \implies -(\text{id} + E) \implies -(\text{id} + \text{id})$

Definition (Sentential Form; 句型)

如果 $S \xrightarrow{*} \alpha$, 且 $\alpha \in (T \cup N)^*$, 则称 α 是文法 G 的一个**句型**。

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$$

$E \implies -E \implies -(E) \implies -(E+E) \implies -(\text{id} + E) \implies -(\text{id} + \text{id})$

Definition (Sentence; 句子)

如果 $S \xrightarrow{*} w$, 且 $w \in T^*$, 则称 w 是文法 G 的一个**句子**。

Definition (文法 G 生成的语言 $L(G)$)

文法 G 的语言 $L(G)$ 是它能推导出的**所有句子**构成的集合。

$$w \in L(G) \iff S \xrightarrow{*} w$$

关于文法 G 的**两个基本问题**:

- **Membership 问题:** 给定字符串 $x \in T^*$, $x \in L(G)$?
- $L(G)$ 究竟是什么?

给定字符串 $x \in T^*$, $x \in L(G)$?

(即, 检查 x 是否符合文法 G)

给定字符串 $x \in T^*$, $x \in L(G)$?

(即, 检查 x 是否符合文法 G)

这就是**语法分析器**的任务:

为输入的词法单元流寻找推导、**构建语法分析树**, 或者报错

根节点是文法 G 的起始符号

$\langle \text{Stmt} \rangle$	
if (<u>$\langle \text{Expr} \rangle$</u>)	(Stmt)
if (<u>$\langle \text{Expr} \rangle$</u> <u>$\langle \text{Optr} \rangle$</u> <u>$\langle \text{Expr} \rangle$</u>)	(Stmt)
if (<u>$\langle \text{Id} \rangle$</u> <u>$\langle \text{Optr} \rangle$</u> <u>$\langle \text{Expr} \rangle$</u>)	(Stmt)
if (<u>x</u> <u>$\langle \text{Optr} \rangle$</u> <u>$\langle \text{Expr} \rangle$</u>)	(Stmt)
if (<u>x</u> > <u>$\langle \text{Expr} \rangle$</u>)	(Stmt)
if (<u>x</u> > <u>$\langle \text{Num} \rangle$</u>)	(Stmt)
if (<u>x</u> > <u>9</u>)	(Stmt)
if (<u>x</u> > <u>9</u>) {	<u>$\langle \text{StmtList} \rangle$</u>
if (<u>x</u> > <u>9</u>) {	<u>$\langle \text{Stmt} \rangle$</u>
if (<u>x</u> > <u>9</u>) {	<u>$\langle \text{Stmt} \rangle$</u>
if (<u>x</u> > <u>9</u>) { <u>$\langle \text{Id} \rangle$</u> = <u>$\langle \text{Expr} \rangle$</u> ;	(Stmt)
if (<u>x</u> > <u>9</u>) { <u>x</u> = <u>$\langle \text{Expr} \rangle$</u> ;	(Stmt)
if (<u>x</u> > <u>9</u>) { <u>x</u> = <u>$\langle \text{Num} \rangle$</u> ;	(Stmt)
if (<u>x</u> > <u>9</u>) { <u>x</u> = <u>0</u> ;	<u>$\langle \text{Stmt} \rangle$</u>
if (<u>x</u> > <u>9</u>) { <u>x</u> = <u>0</u> ; <u>$\langle \text{Id} \rangle$</u> = <u>$\langle \text{Expr} \rangle$</u> ;	(Expr)
if (<u>x</u> > <u>9</u>) { <u>x</u> = <u>0</u> ; <u>y</u> = <u>$\langle \text{Expr} \rangle$</u> ;	(Expr)
if (<u>x</u> > <u>9</u>) { <u>x</u> = <u>0</u> ; <u>y</u> = <u>$\langle \text{Expr} \rangle$</u> <u>$\langle \text{Optr} \rangle$</u> <u>$\langle \text{Expr} \rangle$</u> ;	(Expr)
if (<u>x</u> > <u>9</u>) { <u>x</u> = <u>0</u> ; <u>y</u> = <u>$\langle \text{Id} \rangle$</u> <u>$\langle \text{Optr} \rangle$</u> <u>$\langle \text{Expr} \rangle$</u> ;	(Expr)
if (<u>x</u> > <u>9</u>) { <u>x</u> = <u>0</u> ; <u>y</u> = <u>$\langle \text{Optr} \rangle$</u> <u>$\langle \text{Expr} \rangle$</u> ;	(Expr)
if (<u>x</u> > <u>9</u>) { <u>x</u> = <u>0</u> ; <u>y</u> = <u>y</u> + <u>$\langle \text{Expr} \rangle$</u> ;	(Expr)
if (<u>x</u> > <u>9</u>) { <u>x</u> = <u>0</u> ; <u>y</u> = <u>y</u> + <u>$\langle \text{Num} \rangle$</u> ;	(Num)
if (<u>x</u> > <u>9</u>) { <u>x</u> = <u>0</u> ; <u>y</u> = <u>y</u> + <u>1</u> ;	

叶子节点是输入的词法单元流

常用的语法分析器以**自顶向下**或**自底向上**的方式构建中间部分

$L(G)$ 是什么?

这是**程序设计语言设计者**需要考虑的问题

$$S \rightarrow SS$$
$$S \rightarrow (S)$$
$$S \rightarrow ()$$
$$S \rightarrow \epsilon$$
$$L(G) =$$

$$S \rightarrow SS$$
$$S \rightarrow (S)$$
$$S \rightarrow ()$$
$$S \rightarrow \epsilon$$
$$L(G) = \{\text{良匹配括号串}\}$$

$S \rightarrow SS$ $S \rightarrow (S)$ $S \rightarrow ()$ $S \rightarrow \epsilon$ $S \rightarrow aSb$ $S \rightarrow \epsilon$ $L(G) =$ $L(G) = \{\text{良匹配括号串}\}$

$$S \rightarrow SS$$
$$S \rightarrow (S)$$
$$S \rightarrow ()$$
$$S \rightarrow \epsilon$$
$$S \rightarrow aSb$$
$$S \rightarrow \epsilon$$

$$L(G) = \{a^n b^n \mid n \geq 0\}$$

$$L(G) = \{\text{良匹配括号串}\}$$

字母表 $\Sigma = \{a, b\}$ 上的所有**回文串** (Palindrome) 构成的语言

字母表 $\Sigma = \{a, b\}$ 上的所有**回文串** (Palindrome) 构成的语言

$$S \rightarrow aSa$$
$$S \rightarrow bSb$$
$$S \rightarrow a$$
$$S \rightarrow b$$
$$S \rightarrow \epsilon$$

字母表 $\Sigma = \{a, b\}$ 上的所有**回文串** (Palindrome) 构成的语言

$$S \rightarrow aSa$$

$$S \rightarrow bSb$$

$$S \rightarrow a$$

$$S \rightarrow b$$

$$S \rightarrow \epsilon$$

$$S \rightarrow aSa \mid bSb \mid a \mid b \mid \epsilon$$

$$\{b^n a^m b^{2n} \mid n \geq 0, m \geq 0\}$$

$\{b^n a^m b^{2n} \mid n \geq 0, m \geq 0\}$

$S \rightarrow bSbb \mid A$

$A \rightarrow aA \mid \epsilon$

$$\{x \in \{a,b\}^* \mid x \text{ 中 } a, b \text{ 个数相同}\}$$

$\{x \in \{a,b\}^* \mid x \text{ 中 } a, b \text{ 个数相同}\}$

$V \rightarrow aVbV \mid bVaV \mid \epsilon$

$$\{x \in \{a,b\}^* \mid x \text{ 中 } a, b \text{ 个数不同}\}$$

$\{x \in \{a,b\}^* \mid x \text{ 中 } a, b \text{ 个数不同}\}$

$S \rightarrow T \mid U$

$T \rightarrow VaT \mid VaV$

$U \rightarrow VbU \mid VbV$

$V \rightarrow aVbV \mid bVaV \mid \epsilon$

$\{x \in \{a,b\}^* \mid x \text{ 中 } a, b \text{ 个数不同}\}$

$S \rightarrow T \mid U$

$T \rightarrow VaT \mid VaV$

$U \rightarrow VbU \mid VbV$

$V \rightarrow aVbV \mid bVaV \mid \epsilon$



练习 (非作业): 证明之

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$

$S \rightarrow \text{begin } S \ L$

$S \rightarrow \text{print } E$

$L \rightarrow \text{end}$

$L \rightarrow ; \ S \ L$

$E \rightarrow \text{num} \ = \ \text{num}$

$$S \rightarrow \text{if } E \text{ then } S \text{ else } S$$
$$S \rightarrow \text{begin } S \ L$$
$$S \rightarrow \text{print } E$$
$$L \rightarrow \text{end}$$
$$L \rightarrow ; \ S \ L$$
$$E \rightarrow \text{num} \ = \ \text{num}$$

顺序语句、条件语句、打印语句



L-System

(注: 这不是上下文无关文法, 但精神上高度一致, 并且更有趣)

variables : A B

constants : + -

start : A

rules : $(A \rightarrow B-A-B)$, $(B \rightarrow A+B+A)$

angle : 60°

A, B : 向右移动并画线

+ : 左转

- : 右转

每一步都并行地应用所有规则

A

B - A - B

A

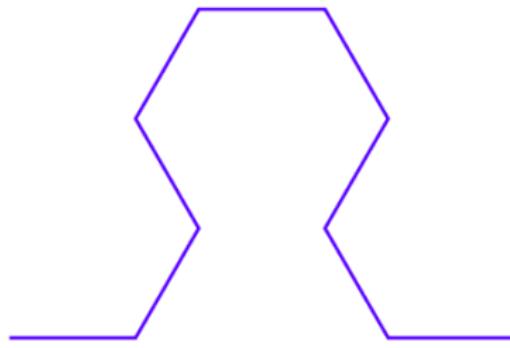
B – *A* – *B*

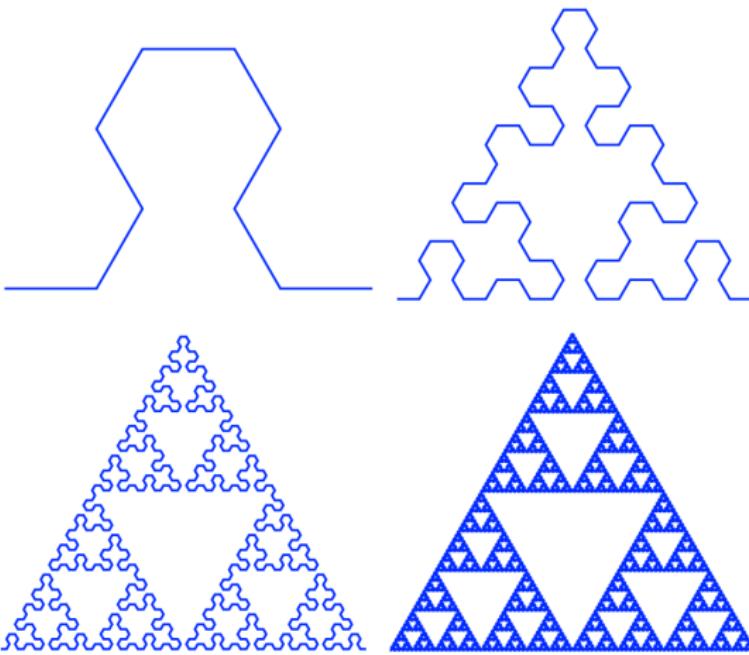
A + *B* + *A* – *B* – *A* – *B* – *A* + *B* + *A*

A

B – *A* – *B*

A + *B* + *A* – *B* – *A* – *B* – *A* + *B* + *A*





Sierpinski arrowhead curve ($n = 2, 4, 6, 8$)

variables : X Y

constants : F + -

start : FX

rules : ($X \rightarrow X+YF+$), ($Y \rightarrow -FX-Y$)

angle : 90°

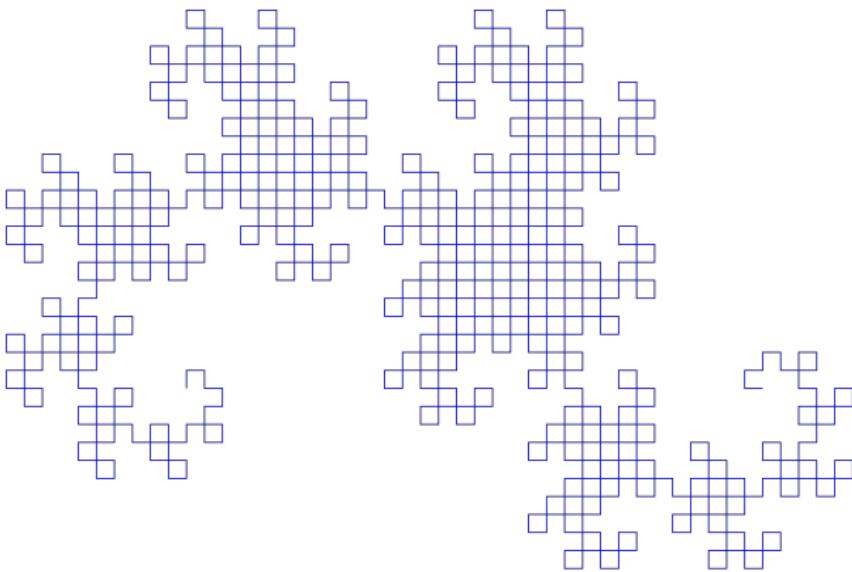
F : 向上移动并画线

+ : 右转

- : 左转

X : 仅用于展开, 在作画时被忽略

每一步都**并行地**应用**所有**规则



Dragon Curve ($n = 10$)

最左 (leftmost) 推导与最右 (rightmost) 推导

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$
$$E \xrightarrow{\text{lm}} -E \xrightarrow{\text{lm}} -(E) \xrightarrow{\text{lm}} -(E + E) \xrightarrow{\text{lm}} -(\mathbf{id} + E) \xrightarrow{\text{lm}} -(\mathbf{id} + \mathbf{id})$$

最左 (leftmost) 推导与最右 (rightmost) 推导

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

$$E \xrightarrow[\text{lm}]{} -E \xrightarrow[\text{lm}]{} -(E) \xrightarrow[\text{lm}]{} -(E + E) \xrightarrow[\text{lm}]{} -(\mathbf{id} + E) \xrightarrow[\text{lm}]{} -(\mathbf{id} + \mathbf{id})$$

$E \xrightarrow[\text{lm}]{} -E$: 经过一步最左推导得出

$E \xrightarrow[\text{lm}]{}^+ -(\mathbf{id} + E)$: 经过一步或多步最左推导得出

$E \xrightarrow[\text{lm}]{}^* -(\mathbf{id} + E)$: 经过零步或多步最左推导得出

最左 (leftmost) 推导与最右 (rightmost) 推导

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$$

$$E \xrightarrow[\text{lm}]{} -E \xrightarrow[\text{lm}]{} -(E) \xrightarrow[\text{lm}]{} -(E + E) \xrightarrow[\text{lm}]{} -(\text{id} + E) \xrightarrow[\text{lm}]{} -(\text{id} + \text{id})$$

$E \xrightarrow[\text{lm}]{} -E$: 经过一步最左推导得出

$E \xrightarrow[\text{lm}]{}^+ -(\text{id} + E)$: 经过一步或多步最左推导得出

$E \xrightarrow[\text{lm}]{}^* -(\text{id} + E)$: 经过零步或多步最左推导得出

$$E \xrightarrow[\text{rm}]{} -E \xrightarrow[\text{rm}]{} -(E) \xrightarrow[\text{rm}]{} -(E + E) \xrightarrow[\text{rm}]{} -(E + \text{id}) \xrightarrow[\text{rm}]{} -(\text{id} + \text{id})$$

Definition (Left-sentential Form; 最左句型)

如果 $S \xrightarrow[\text{lm}]{*} \alpha$, 且 $\alpha \in (T \cup N)^*$, 则称 α 是文法 G 的一个**最左句型**。

$$E \xrightarrow[\text{lm}]{} -E \xrightarrow[\text{lm}]{} -(E) \xrightarrow[\text{lm}]{} -(E + E) \xrightarrow[\text{lm}]{} -(\mathbf{id} + E) \xrightarrow[\text{lm}]{} -(\mathbf{id} + \mathbf{id})$$

Definition (Left-sentential Form; 最左句型)

如果 $S \xrightarrow[\text{lm}]{*} \alpha$, 且 $\alpha \in (T \cup N)^*$, 则称 α 是文法 G 的一个**最左句型**。

$$E \xrightarrow[\text{lm}]{} -E \xrightarrow[\text{lm}]{} -(E) \xrightarrow[\text{lm}]{} -(E + E) \xrightarrow[\text{lm}]{} -(\mathbf{id} + E) \xrightarrow[\text{lm}]{} -(\mathbf{id} + \mathbf{id})$$

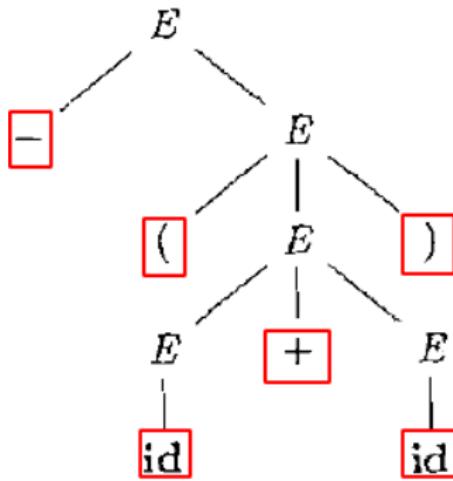
Definition (Right-sentential Form; 最右句型)

如果 $S \xrightarrow[\text{rm}]{*} \alpha$, 且 $\alpha \in (T \cup N)^*$, 则称 α 是文法 G 的一个**最右句型**。

$$E \xrightarrow[\text{rm}]{} -E \xrightarrow[\text{rm}]{} -(E) \xrightarrow[\text{rm}]{} -(E + E) \xrightarrow[\text{rm}]{} -(E + \mathbf{id}) \xrightarrow[\text{rm}]{} -(\mathbf{id} + \mathbf{id})$$

语法分析树

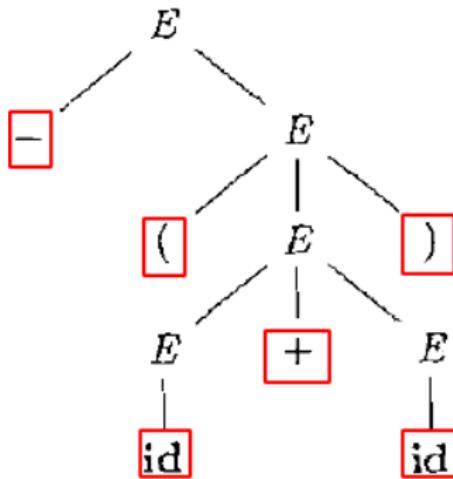
语法分析树是静态的，它不关心动态的推导顺序



一棵语法分析树对应多个推导

语法分析树

语法分析树是静态的，它不关心动态的推导顺序



一棵语法分析树对应多个推导

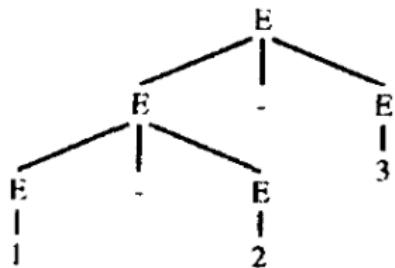
但是，一棵语法分析树与**最左（最右）推导**一一对应

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid \mathbf{id} \mid \mathbf{num}$$

1 – 2 – 3 的语法树?

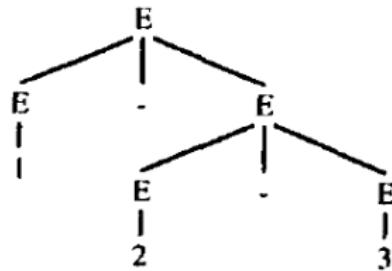
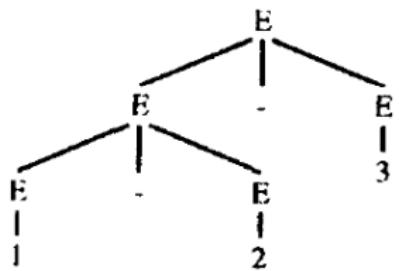
$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid \text{id} \mid \text{num}$$

1 - 2 - 3 的语法树?



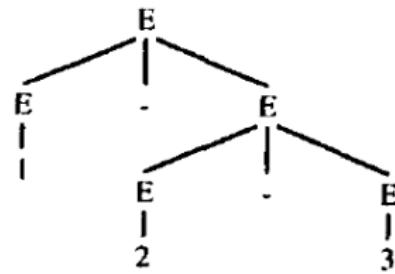
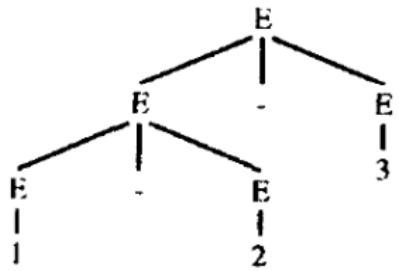
$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid \text{id} \mid \text{num}$$

1 - 2 - 3 的语法树?



$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid \text{id} \mid \text{num}$$

1 - 2 - 3 的语法树?



Definition (二义性(Ambiguous) 文法)

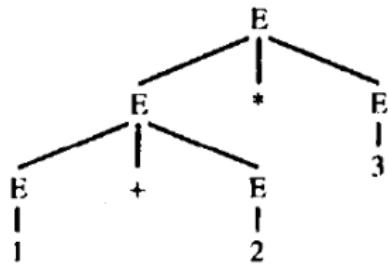
如果 $L(G)$ 中的某个句子有一个以上语法树/最左推导/最右推导, 则文法 G 是二义性的。

$$E \rightarrow E + E \mid E - E \mid E * E \mid E/E \mid (E) \mid \mathbf{id} \mid \mathbf{num}$$

1 + 2 * 3 的语法树?

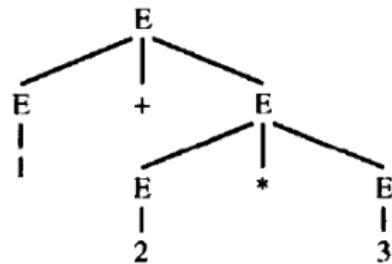
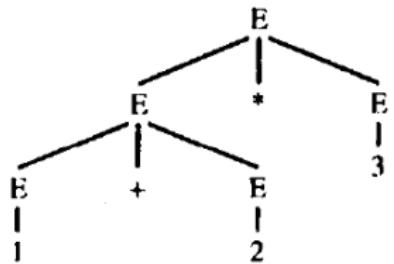
$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid \text{id} \mid \text{num}$$

$1 + 2 * 3$ 的语法树?



$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid \text{id} \mid \text{num}$$

$1 + 2 * 3$ 的语法树?



```
stmt → if expr then stmt
      | if expr then stmt else stmt
      | other
```

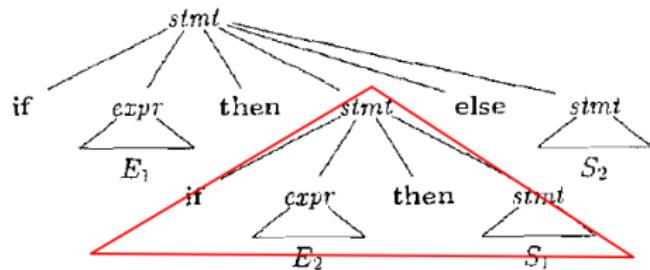
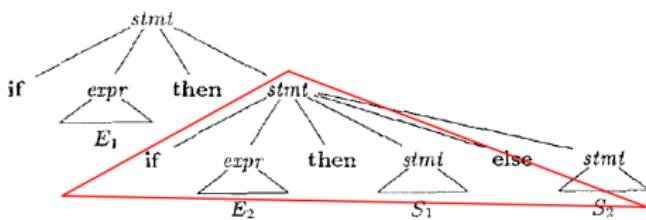
“悬空-else”文法

if E_1 then if E_2 then S_1 else S_2

stmt → if *expr* then *stmt*
 | if *expr* then *stmt* else *stmt*
 | other

“悬空-else”文法

if E_1 **then if** E_2 **then** S_1 **else** S_2



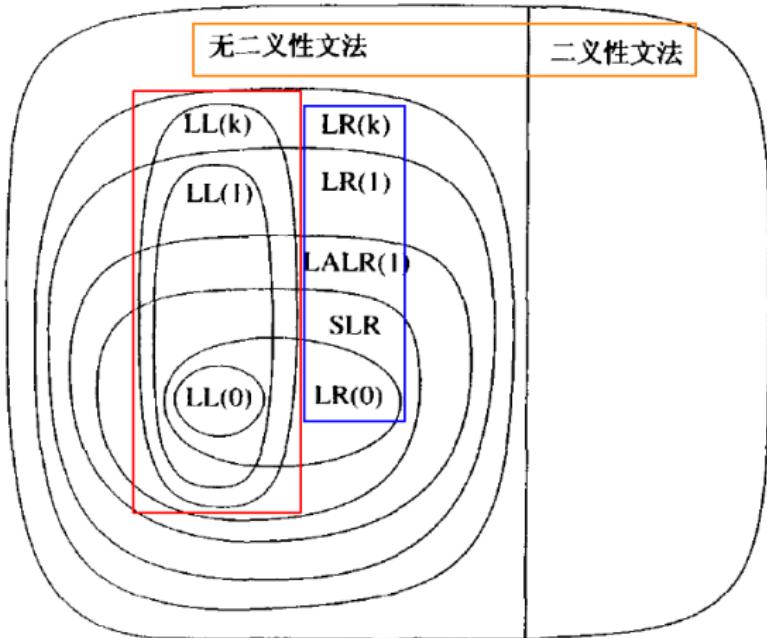
if E_1 **then** (**if** E_2 **then** S_1 **else** S_2)

if E_1 **then** (**if** E_2 **then** S_1) **else** S_2

二义性文法

不同的语法分析树产生不同的语义





所有语法分析器都要求文法是**无二义性**的

二义性文法

Q：如何**识别**二义性文法？

Q：如何**消除**文法的二义性？

二义性文法

Q：如何识别二义性文法？



Q：如何消除文法的二义性？

这是**不可判定**的问题

二义性文法

Q：如何识别二义性文法？



Q：如何消除文法的二义性？

LEARN BY EXAMPLES

这是不可判定的问题

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid \text{id} \mid \text{num}$$

四则运算均是**左结合**的

优先级: 括号最先, 先乘除后加减

二义性表达式文法以**相同的方式**处理所有的算术运算符

要消除二义性, 需要**区别对待**不同的运算符

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid \text{id} \mid \text{num}$$

四则运算均是**左结合**的

优先级: 括号最先, 先乘除后加减

二义性表达式文法以**相同的方式**处理所有的算术运算符

要消除二义性, 需要**区别对待**不同的运算符

将运算的“先后”顺序信息编码到语法树的“层次”结构中

$$E \rightarrow E + E \mid \mathbf{id}$$

$$E \rightarrow E + E \mid \text{id}$$
$$E \rightarrow E + T$$
$$T \rightarrow \text{id}$$

左结合文法

$$E \rightarrow E + E \mid \text{id}$$
$$E \rightarrow E + T$$
$$T \rightarrow \text{id}$$

左结合文法

$$E \rightarrow T + E$$
$$T \rightarrow \text{id}$$

右结合文法

$$E \rightarrow E + E \mid \text{id}$$

$$E \rightarrow E + T$$

$$T \rightarrow \text{id}$$

左结合文法

$$E \rightarrow T + E$$

$$T \rightarrow \text{id}$$

右结合文法

使用左(右)递归实现左(右)结合

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$$

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$$
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \text{id}$$

括号最先，先乘后加文法

$$E \rightarrow E + E \mid E - E \mid E * E \mid E/E \mid (E) \mid \mathbf{id} \mid \mathbf{num}$$
$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T/F \mid F$$
$$F \rightarrow (E) \mid \mathbf{id} \mid \mathbf{num}$$

无二义性的表达式文法

E : 表达式(*expression*); T : 项(*term*) F : 因子(*factor*)

$$E \rightarrow E + E \mid E - E \mid E * E \mid E/E \mid (E) \mid \mathbf{id} \mid \mathbf{num}$$
$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T/F \mid F$$
$$F \rightarrow (E) \mid \mathbf{id} \mid \mathbf{num}$$

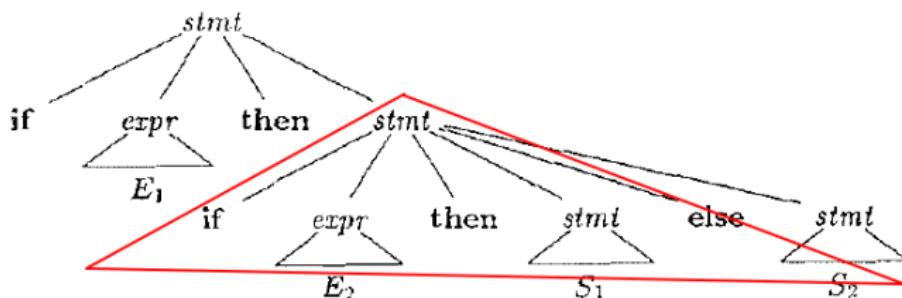
无二义性的表达式文法

E : 表达式(*expression*); T : 项(*term*) F : 因子(*factor*)

将运算的“先后”顺序信息编码到语法树的“层次”结构中

$stmt \rightarrow if\ expr\ then\ stmt$
 | $if\ expr\ then\ stmt\ else\ stmt$
 | **other**

$if\ E_1\ then\ if\ E_2\ then\ S_1\ else\ S_2$



“每个else与最近的尚未匹配的then匹配”

$stmt \rightarrow if\ expr\ then\ stmt$
| $if\ expr\ then\ stmt\ else\ stmt$
| $other$

$stmt \rightarrow matched_stmt$
| $open_stmt$
 $matched_stmt \rightarrow if\ expr\ then\ matched_stmt\ else\ matched_stmt$
| $other$
 $open_stmt \rightarrow if\ expr\ then\ stmt$
| $if\ expr\ then\ matched_stmt\ else\ open_stmt$

基本思想: **then** 与 **else** 之间的语句必须是“已匹配的”

我也看不懂啊

~~“我不想上去课啊妈妈”~~

“清醒一点！你是老师啊！”



我们要证明**两件**事情



我们要证明**两件**事情

$$L(G) = L(G')$$



我们要证明**两件**事情

$$L(G) = L(G')$$

G' 是无二义性的

$stmt \rightarrow if\ expr\ then\ stmt$
| $if\ expr\ then\ stmt\ else\ stmt$
| $other$

$stmt \rightarrow matched_stmt$
| $open_stmt$
 $matched_stmt \rightarrow if\ expr\ then\ matched_stmt\ else\ matched_stmt$
| $other$
 $open_stmt \rightarrow if\ expr\ then\ stmt$
| $if\ expr\ then\ matched_stmt\ else\ open_stmt$

$stmt \rightarrow if\ expr\ then\ stmt$
| $if\ expr\ then\ stmt\ else\ stmt$
| **other**

$stmt \rightarrow matched_stmt$
| $open_stmt$
 $matched_stmt \rightarrow if\ expr\ then\ matched_stmt\ else\ matched_stmt$
| **other**
 $open_stmt \rightarrow if\ expr\ then\ stmt$
| $if\ expr\ then\ matched_stmt\ else\ open_stmt$

$$L(G) \subseteq L(G')$$

$$L(G') \subseteq L(G)$$

$$\begin{aligned}
 stmt &\rightarrow \text{if } expr \text{ then } stmt \\
 &| \quad \text{if } expr \text{ then } stmt \text{ else } stmt \\
 &| \quad \text{other}
 \end{aligned}$$

$ \begin{aligned} stmt &\rightarrow matched_stmt \\ & open_stmt \end{aligned} $	$ \begin{aligned} matched_stmt &\rightarrow \text{if } expr \text{ then } matched_stmt \text{ else } matched_stmt \\ & \quad \text{other} \end{aligned} $
$ \begin{aligned} open_stmt &\rightarrow \text{if } expr \text{ then } stmt \\ & \quad \text{if } expr \text{ then } matched_stmt \text{ else } open_stmt \end{aligned} $	

$$L(G) \subseteq L(G')$$

$$L(G') \subseteq L(G)$$

对推导步数作数学归纳

G' 是无二义性的

```
stmt   →  matched_stmt
       |
       open_stmt
matched_stmt → if expr then matched_stmt else matched_stmt
       |
       other
open_stmt  → if expr then stmt
       |
       if expr then matched_stmt else open_stmt
```

G' 是无二义性的

```
stmt   →  matched_stmt
       |
       open_stmt
matched_stmt → if expr then matched_stmt else matched_stmt
       |
       other
open_stmt  → if expr then stmt
       |
       if expr then matched_stmt else open_stmt
```

每个句子对应的**语法分析树**是唯一的

G' 是无二义性的

```
stmt   →  matched_stmt
       |
       open_stmt
matched_stmt → if expr then matched_stmt else matched_stmt
       |
       other
open_stmt  → if expr then stmt
       |
       if expr then matched_stmt else open_stmt
```

每个句子对应的**语法分析树**是唯一的

只需证明：每个非终结符的“**展开**”方式是唯一的

G' 是无二义性的

```
stmt   →  matched_stmt
       |
       open_stmt
matched_stmt → if expr then matched_stmt else matched_stmt
       |
       other
open_stmt  → if expr then stmt
       |
       if expr then matched_stmt else open_stmt
```

每个句子对应的**语法分析树**是唯一的

只需证明：每个非终结符的“**展开**”方式是唯一的

$$L(\text{matched_stmt}) \cap L(\text{open_stmt}) = \emptyset$$

G' 是无二义性的

```
stmt   → matched_stmt
      |
      open_stmt
matched_stmt → if expr then matched_stmt else matched_stmt
      |
      other
open_stmt   → if expr then stmt
      |
      if expr then matched_stmt else open_stmt
```

每个句子对应的**语法分析树**是唯一的

只需证明：每个非终结符的“**展开**”方式是唯一的

$$L(\text{matched_stmt}) \cap L(\text{open_stmt}) = \emptyset$$

$$L(\text{matched_stmt}_1) \cap L(\text{matched_stmt}_2) = \emptyset$$

G' 是无二义性的

```
stmt   → matched_stmt
      |
      open_stmt
matched_stmt → if expr then matched_stmt else matched_stmt
      |
      other
open_stmt   → if expr then stmt
      |
      if expr then matched_stmt else open_stmt
```

每个句子对应的**语法分析树**是唯一的

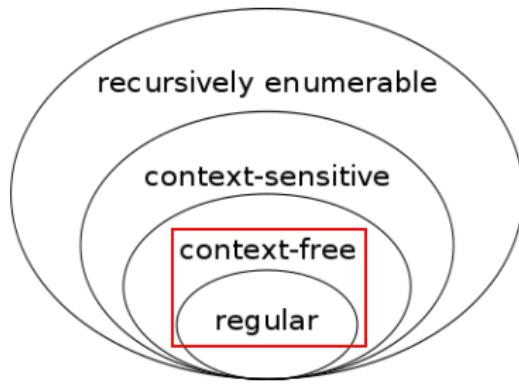
只需证明：每个非终结符的“**展开**”方式是唯一的

$$L(\text{matched_stmt}) \cap L(\text{open_stmt}) = \emptyset$$

$$L(\text{matched_stmt}_1) \cap L(\text{matched_stmt}_2) = \emptyset$$

$$L(\text{open_stmt}_1) \cap L(\text{open_stmt}_2) = \emptyset$$

为什么不使用优雅、强大的**正则表达式**描述程序设计语言的语法？



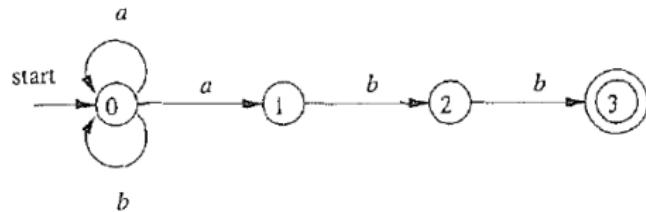
正则表达式的表达能力**严格弱于**上下文无关文法

每个正则表达式 r 对应的语言 $L(r)$ 都可以使用上下文无关文法来描述

$$r = (a|b)^*abb$$

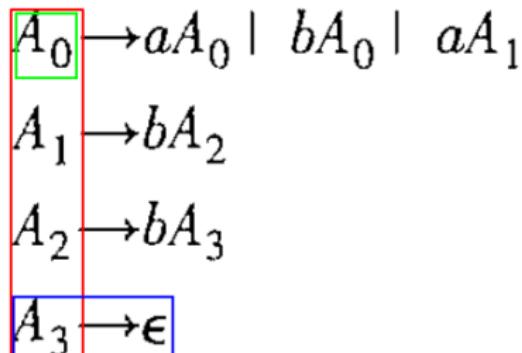
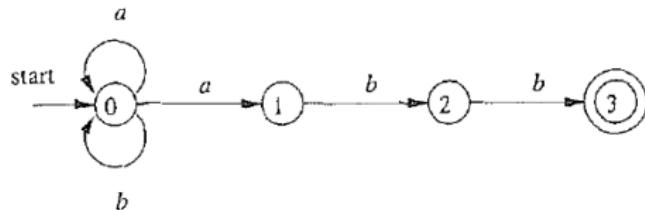
每个正则表达式 r 对应的语言 $L(r)$ 都可以使用上下文无关文法来描述

$$r = (a|b)^*abb$$



每个正则表达式 r 对应的语言 $L(r)$ 都可以使用上下文无关文法来描述

$$r = (a|b)^*abb$$



此外, 若 $\delta(A_i, \epsilon) = A_j$, 则添加 $A_i \rightarrow A_j$

$$S \rightarrow aSb$$
$$S \rightarrow \epsilon$$

$$L = \{a^n b^n \mid n \geq 0\}$$

该语言**无法**使用正则表达式来描述

Theorem

$L = \{a^n b^n \mid n \geq 0\}$ 无法使用正则表达式描述。

Theorem

$L = \{a^n b^n \mid n \geq 0\}$ 无法使用正则表达式描述。

反证法

Theorem

$L = \{a^n b^n \mid n \geq 0\}$ 无法使用正则表达式描述。

反证法

假设存在正则表达式 r : $L(r) = L$

Theorem

$L = \{a^n b^n \mid n \geq 0\}$ 无法使用正则表达式描述。

反证法

假设存在正则表达式 r : $L(r) = L$

则存在**有限**状态自动机 $D(r)$: $L(D(r)) = L$; 设其状态数为 k

Theorem

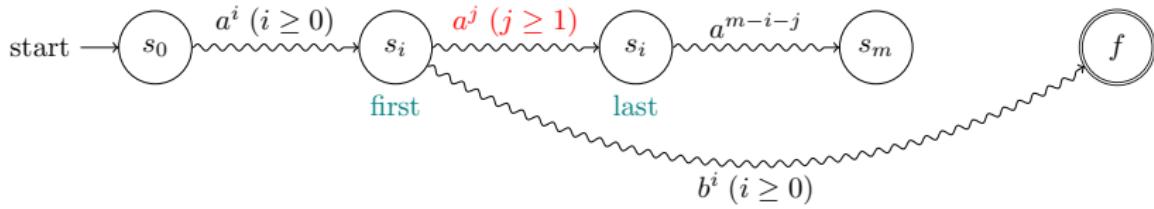
$L = \{a^n b^n \mid n \geq 0\}$ 无法使用正则表达式描述。

反证法

假设存在正则表达式 r : $L(r) = L$

则存在**有限**状态自动机 $D(r)$: $L(D(r)) = L$; 设其状态数为 k

考慮输入 $a^m (m > k)$



Theorem

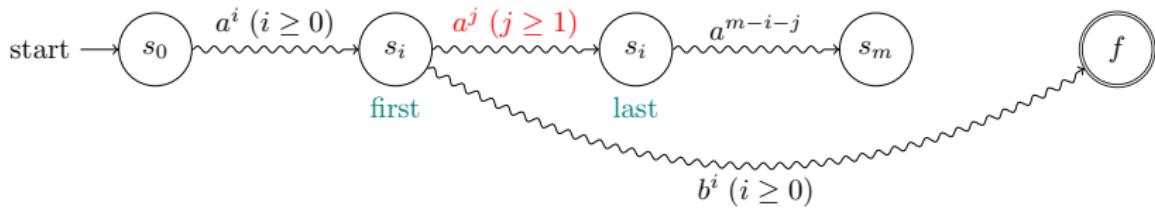
$L = \{a^n b^n \mid n \geq 0\}$ 无法使用正则表达式描述。

反证法

假设存在正则表达式 r : $L(r) = L$

则存在**有限**状态自动机 $D(r)$: $L(D(r)) = L$; 设其状态数为 k

考慮输入 $a^m (m > k)$



$D(r)$ 也能接受 $a^{i+j}b^i$; 矛盾!

$$L = \{a^n b^n \mid n \geq 0\}$$

Pumping Lemma for Regular Languages

$$L = \{a^n b^n \mid n \geq 0\}$$

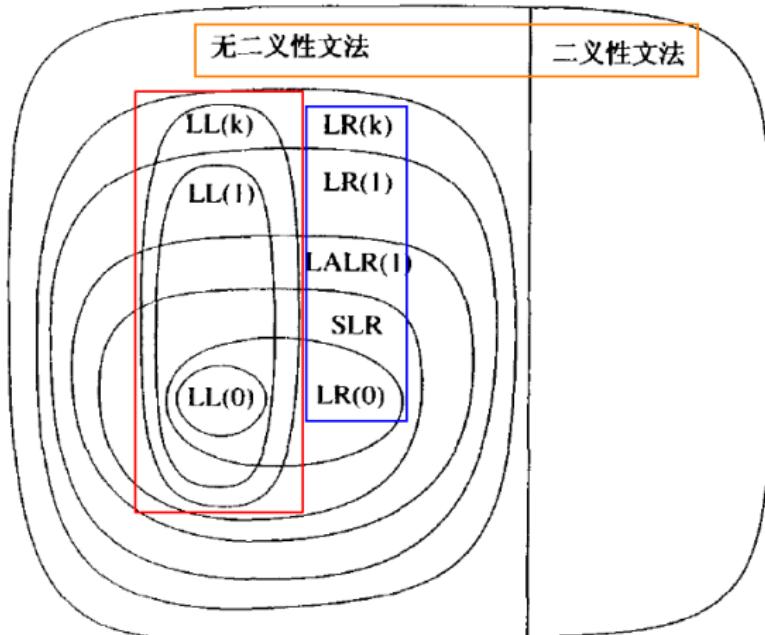
Pumping Lemma for Regular Languages

$$L = \{a^n b^n c^n \mid n \geq 0\}$$

Pumping Lemma for Context-free Languages

只考虑无二义性的文法

这意味着，每个句子对应唯一的一棵语法分析树



今日份主题: $LL(1)$ 语法分析器

自顶向下的、
递归下降的、
预测分析的、
适用于 $LL(1)$ 文法的、
 $LL(1)$ 语法分析器

自顶向下构建语法分析树

根节点是文法的起始符号 S

叶节点是词法单元流 $w\$$

仅包含终结符号与特殊的文件结束符 $\$$

自顶向下构建语法分析树

根节点是文法的起始符号 S

每个中间节点表示对某个非终结符应用某个产生式进行推导

(Q : 选择哪个非终结符, 以及选择哪个产生式)

叶节点是词法单元流 $w\$$

仅包含终结符号与特殊的文件结束符 $\$$

递归下降的实现框架

不需要任何参数

```
void A() {  
    先不考虑这里是如何选择产生式的  
    1)     选择一个 A 产生式,  $A \rightarrow X_1 X_2 \cdots X_k$ ;  
    2)     for ( i = 1 to k ) {  
    3)         if (  $X_i$  是一个非终结符号 )  
    4)             递归下降调用其它非终结符对应的递归函数  
    5)             调用过程  $X_i()$ ;  
    6)         else if (  $X_i$  等于当前的输入符号 a )  
    7)             匹配当前词法单元  
                读入下一个输入符号;  
            }  
        else /* 发生了一个错误 */;  
    }  
}  
} 出现了不期望出现的词法单元
```

为每个非终结符写一个递归函数

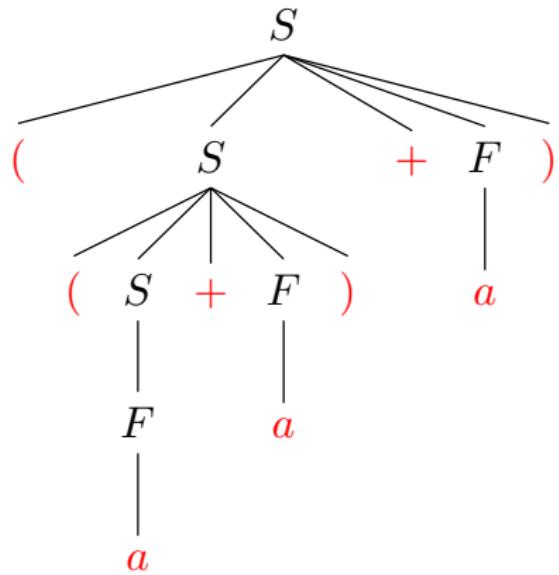
内部按需调用其它非终结符对应的递归函数

$$S \rightarrow F$$
$$S \rightarrow (S + F)$$
$$F \rightarrow a$$

$$w = ((a + a) + a)$$

演示递归下降过程

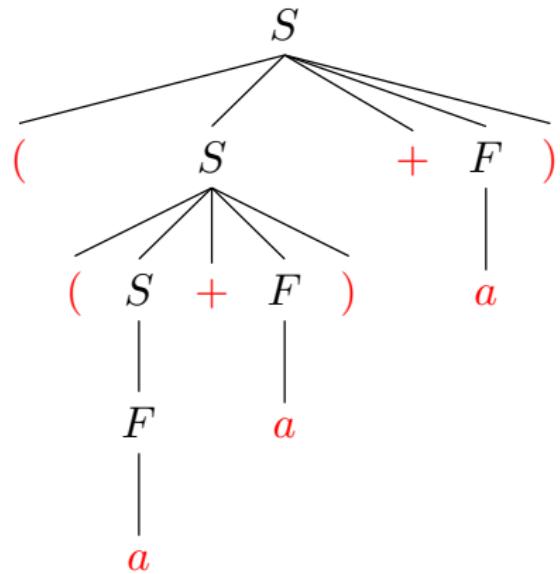
```
S → F  
S → (S + F)  
F → a
```



$$w = ((a+a)+a)$$

演示递归下降过程

```
S → F  
S → (S + F)  
F → a
```

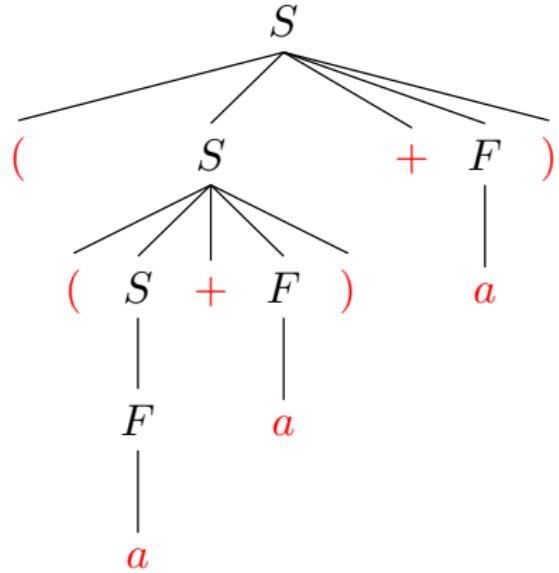


$$w = ((a + a) + a)$$

每次都选择语法分析树**最左边**的非终结符进行展开

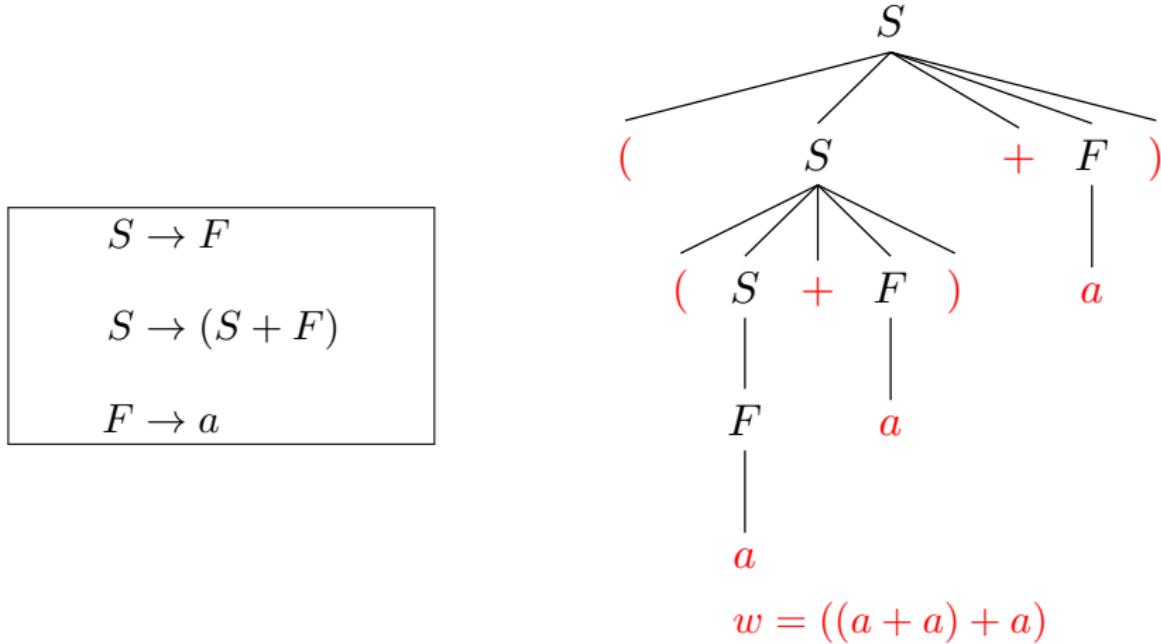
同样是展开非终结符 S ,
为什么前两次选择了 $S \rightarrow (S + F)$, 而第三次选择了 $S \rightarrow F$?

$S \rightarrow F$
 $S \rightarrow (S + F)$
 $F \rightarrow a$



$$w = ((a+a)+a)$$

同样是展开非终结符 S ,
为什么前两次选择了 $S \rightarrow (S + F)$, 而第三次选择了 $S \rightarrow F$?



因为它们面对的**当前词法单元**不同

使用预测分析表确定产生式

$S \rightarrow F$
$S \rightarrow (S + F)$
$F \rightarrow a$

	()	a	+	\$
S	2		1		
F			3		

指明了每个**非终结符**在面对不同的**词法单元或文件结束符**时，
该选择哪个**产生式** (按编号进行索引) 或者**报错**

Definition ($LL(1)$ 文法)

如果文法 G 的预测分析表是无冲突的，则 G 是 $LL(1)$ 文法。

无冲突： 每个单元格里只有一个生成式（编号）

$$S \rightarrow F$$

$$S \rightarrow (S + F)$$

$$F \rightarrow a$$

	()	a	+	\$
S	2		1		
F			3		

对于当前选择的**非终结符**,

仅根据输入中**当前的词法单元**即可确定需要使用哪条**产生式**

递归下降的、预测分析实现方法

$S \rightarrow F$

$S \rightarrow (S + F)$

$F \rightarrow a$

	()	a	+	\$
<i>S</i>	2		1		
<i>F</i>			3		

```
1: procedure MATCH(t)
2:   if token = t then
3:     token ← NEXT-TOKEN()
4:   else
5:     ERROR(token, t)
```

```
1: procedure S()
2:   if token = '(' then
3:     MATCH('(')
4:     S()
5:     MATCH('+')
6:     F()
7:     MATCH(')')
8:   else if token = 'a' then
9:     F()
10:  else
11:    ERROR(token, {'(', 'a'})
```

递归下降的、预测分析实现方法

$S \rightarrow F$

$S \rightarrow (S + F)$

$F \rightarrow a$

	()	a	+	\$
<i>S</i>	2		1		
<i>F</i>			3		

```
1: procedure MATCH(t)
2:   if token = t then
3:     token ← NEXT-TOKEN()
4:   else
5:     ERROR(token, t)
```

```
1: procedure F()
2:   if token = 'a' then
3:     MATCH('a')
4:   else
5:     ERROR(token, {'a'})
```

如何计算给定文法 G 的预测分析表?

$\text{FIRST}(\alpha)$ 是可从 α 推导得到的句型的**首终结符号**的集合

Definition ($\text{FIRST}(\alpha)$ 集合)

对于任意的 (产生式的右部) $\alpha \in (N \cup T)^*$:

$$\text{FIRST}(\alpha) = \left\{ t \in T \cup \{\epsilon\} \mid \alpha \xrightarrow{*} t\beta \vee \alpha \xrightarrow{*} \epsilon \right\}.$$

如何计算给定文法 G 的预测分析表?

$\text{FIRST}(\alpha)$ 是可从 α 推导得到的句型的**首终结符号**的集合

Definition ($\text{FIRST}(\alpha)$ 集合)

对于任意的 (产生式的右部) $\alpha \in (N \cup T)^*$:

$$\text{FIRST}(\alpha) = \left\{ t \in T \cup \{\epsilon\} \mid \alpha \xrightarrow{*} t\beta \vee \alpha \xrightarrow{*} \epsilon \right\}.$$

考虑非终结符 A 的所有产生式 $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_m$,

如果它们对应的 $\text{FIRST}(\alpha_i)$ 集合互不相交,

则只需查看当前输入词法单元, 即可确定选择哪个产生式 (或报错)

如何计算给定文法 G 的预测分析表?

$\text{FOLLOW}(A)$ 是可能在某些句型中紧跟在 A 右边的终结符的集合

Definition ($\text{FOLLOW}(A)$ 集合)

对于任意的 (产生式的左部) 非终结符 $A \in N$:

$$\text{FOLLOW}(A) = \left\{ t \in T \cup \{\$\} \mid \exists s. S \xrightarrow{*} s \triangleq \beta A \textcolor{red}{t} \gamma \right\}.$$

如何计算给定文法 G 的预测分析表?

$\text{FOLLOW}(A)$ 是可能在某些句型中紧跟在 A 右边的终结符的集合

Definition ($\text{FOLLOW}(A)$ 集合)

对于任意的 (产生式的左部) 非终结符 $A \in N$:

$$\text{FOLLOW}(A) = \left\{ t \in T \cup \{\$\} \mid \exists s. S \xrightarrow{*} s \triangleq \beta A \textcolor{red}{t} \gamma \right\}.$$

考虑产生式 $A \rightarrow \alpha$,

如果从 α 可能推导出空串 ($\alpha \xrightarrow{*} \epsilon$),

则只有当当前词法单元 $t \in \text{FOLLOW}(A)$, 才可以选择该产生式

先计算每个符号 X 的 $\text{FIRST}(X)$ 集合

```
1: procedure FIRST( $X$ )
2:   if  $X \in T$  then                                ▷ 规则 1:  $X$  是终结符
3:     FIRST( $X$ ) =  $X$ 
4:   for  $X \rightarrow Y_1Y_2\dots Y_k$  do                ▷ 规则 2:  $X$  是非终结符
5:     FIRST( $X$ )  $\leftarrow$  FIRST( $X$ )  $\cup$  {FIRST( $Y_1$ )  $\setminus$   $\{\epsilon\}$ }
6:     for  $i \leftarrow 2$  to  $k$  do
7:       if  $\epsilon \in L(Y_1 \dots Y_{i-1})$  then
8:         FIRST( $X$ )  $\leftarrow$  FIRST( $X$ )  $\cup$  {FIRST( $Y_i$ )  $\setminus$   $\{\epsilon\}$ }
9:       if  $\epsilon \in L(Y_1 \dots Y_k)$  then                ▷ 规则 3:  $X$  可推导出空串
10:      FIRST( $X$ )  $\leftarrow$  FIRST( $X$ )  $\cup$   $\{\epsilon\}$ 
```

不断应用上面的规则, 直到每个 $\text{FIRST}(X)$ 都不再变化 (闭包!!!)

再计算每个符号串 α 的 FIRST(α) 集合

$$\alpha = X\beta$$

$$\text{FIRST}(\alpha) = \begin{cases} \text{FIRST}(X) & \epsilon \notin L(X) \\ (\text{FIRST}(X) \setminus \{\epsilon\}) \cup \text{FIRST}(\beta) & \epsilon \in L(X) \end{cases}$$

最后, 如果 $\epsilon \in L(\alpha)$, 则将 ϵ 加入 FIRST(α)。

- (1) $X \rightarrow Y$
- (2) $X \rightarrow a$
- (3) $Y \rightarrow \epsilon$
- (4) $Y \rightarrow c$
- (5) $Z \rightarrow d$
- (6) $Z \rightarrow XYZ$

- (1) $X \rightarrow Y$
- (2) $X \rightarrow a$
- (3) $Y \rightarrow \epsilon$
- (4) $Y \rightarrow c$
- (5) $Z \rightarrow d$
- (6) $Z \rightarrow XYZ$

$$\text{FIRST}(X) = \{a, c, \epsilon\}$$

$$\text{FIRST}(Y) = \{c, \epsilon\}$$

$$\text{FIRST}(Z) = \{a, c, d\}$$

$$\text{FIRST}(XYZ) = \{a, c, d\}$$

为每个非终结符 X 计算 $\text{FOLLOW}(X)$ 集合

```
1: procedure FOLLOW( $X$ )
2:   for  $X$  是开始符号 do          ▷ 规则 1:  $X$  是开始符号
3:      $\text{FOLLOW}(X) \leftarrow \text{FOLLOW}(X) \cup \{\$\}$ 
4:   for  $A \rightarrow \alpha X \beta$  do   ▷ 规则 2:  $X$  是某产生式右部中间的一个符号
5:      $\text{FOLLOW}(X) \leftarrow \text{FOLLOW}(X) \cup (\text{FIRST}(\beta) \setminus \{\epsilon\})$ 
6:     if  $\epsilon \in \text{FIRST}(\beta)$  then
7:        $\text{FOLLOW}(X) \leftarrow \text{FOLLOW}(X) \cup \text{FOLLOW}(A)$ 
8:   for  $A \rightarrow \alpha X$  do     ▷ 规则 3:  $X$  是某产生式右部的最后一个符号
9:      $\text{FOLLOW}(X) \leftarrow \text{FOLLOW}(X) \cup \text{FOLLOW}(A)$ 
```

不断应用上面的规则, 直到每个 $\text{FOLLOW}(X)$ 都不再变化 (闭包!!!)

- (1) $X \rightarrow Y$
- (2) $X \rightarrow a$
- (3) $Y \rightarrow \epsilon$
- (4) $Y \rightarrow c$
- (5) $Z \rightarrow d$
- (6) $Z \rightarrow XYZ$

- (1) $X \rightarrow Y$
- (2) $X \rightarrow a$
- (3) $Y \rightarrow \epsilon$
- (4) $Y \rightarrow c$
- (5) $Z \rightarrow d$
- (6) $Z \rightarrow XYZ$

$$\begin{aligned}\text{FOLLOW}(X) &= \{a, c, d, \$\} \\ \text{FOLLOW}(Y) &= \{a, c, d, \$\} \\ \text{FOLLOW}(Z) &= \emptyset\end{aligned}$$

如何根据FIRST 与 FOLLOW 集合计算给定文法 G 的预测分析表?

按照以下规则, 在表格 $[A, t]$ 中填入生成式 $A \rightarrow \alpha$ (编号):

$$t \in \text{FIRST}(\alpha) \quad (1)$$

$$\alpha \xrightarrow{*} \epsilon \wedge t \in \text{FOLLOW}(A) \quad (2)$$

如何根据FIRST 与 FOLLOW 集合计算给定文法 G 的预测分析表?

按照以下规则, 在表格 $[A, t]$ 中填入生成式 $A \rightarrow \alpha$ (编号):

$$t \in \text{FIRST}(\alpha) \quad (1)$$

$$\alpha \xrightarrow{*} \epsilon \wedge t \in \text{FOLLOW}(A) \quad (2)$$

Definition ($LL(1)$ 文法)

如果文法 G 的**预测分析表是无冲突的**, 则 G 是 $LL(1)$ 文法。

“你是电，你是光，你是唯一的神话”

按照以下规则，在表格 $[A, t]$ 中填入生成式 $A \rightarrow \alpha$ (编号):

$$t \in \text{FIRST}(\alpha) \quad (1)$$

$$\alpha \xrightarrow{*} \epsilon \wedge t \in \text{FOLLOW}(A) \quad (2)$$

因其“唯一”，必要变充分

(1) $X \rightarrow Y$

$$\text{FIRST}(X) = \{a, c, \epsilon\}$$

(2) $X \rightarrow a$

$$\text{FIRST}(Y) = \{c, \epsilon\}$$

(3) $Y \rightarrow \epsilon$

$$\text{FIRST}(Z) = \{a, c, d\}$$

(4) $Y \rightarrow c$

$$\text{FIRST}(XYZ) = \{a, c, d\}$$

(5) $Z \rightarrow d$

$$\text{FOLLOW}(X) = \{a, c, d, \$\}$$

(6) $Z \rightarrow XYZ$

$$\text{FOLLOW}(Y) = \{a, c, d, \$\}$$

$$\text{FOLLOW}(Z) = \emptyset$$

	a	c	d	$\$$
X	1, 2	1	1	1
Y	3	3, 4	3	3
Z	6	6	5, 6	

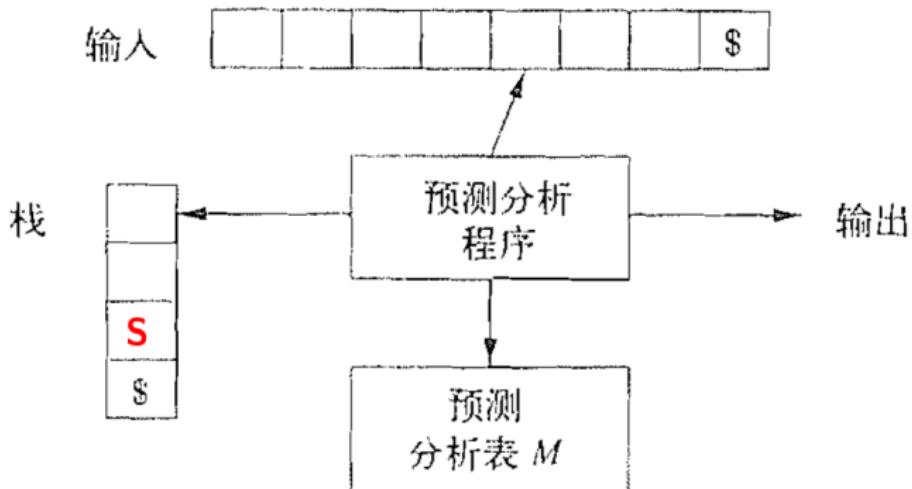
LL(1) 语法分析器

L：从左向右 (left-to-right) 扫描输入

L：构建最左 (leftmost) 推导

1：只需向前看一个输入符号便可确定使用哪条产生式

非递归的预测分析算法



非递归的预测分析算法

设置 ip 使它指向 w 的第一个符号，其中 ip 是输入指针；

令 $X =$ 栈顶符号；

while ($X \neq \$$) { /* 栈非空 */

if (X 等于 ip 所指向的符号 a) 执行栈的弹出操作，将 ip 向前移动一个位置；

else if (X 是一个终结符号) $error()$;

else if ($M[X, a]$ 是一个报错条目) $error()$;

else if ($M[X, a] = X \rightarrow Y_1 Y_2 \dots Y_k$) {

 输出产生式 $X \rightarrow Y_1 Y_2 \dots Y_k$;

弹出栈顶符号；

 将 Y_k, Y_{k-1}, \dots, Y_1 压入栈中，其中 Y_1 位于栈顶。

 }

令 $X =$ 栈顶符号；

}

不是 $LL(1)$ 文法怎么办?

改造它

消除左递归

提取左公因子

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow (E) \mid \text{id} \mid \text{num}$$

E 在**不消耗任何词法单元**的情况下, 直接递归调用 E , 造成**死循环**

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

$$T \rightarrow T * F \mid T / F \mid F$$

$$F \rightarrow (E) \mid \text{id} \mid \text{num}$$

E 在**不消耗任何词法单元**的情况下, 直接递归调用 E , 造成**死循环**

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

$$T \rightarrow T * F \mid T/F \mid F$$

$$F \rightarrow (E) \mid \text{id} \mid \text{num}$$

$$\text{FIRST}(E + T) \cap \text{FIRST}(T) \neq \emptyset$$

不是 $LL(1)$ 文法

消除左递归

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

消除左递归

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

$$E \rightarrow TE'$$

$$E' \rightarrow + TE' \mid \epsilon$$

将左递归转为**右递归**

消除左递归

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

$$E \rightarrow TE'$$

$$E' \rightarrow + TE' \mid \epsilon$$

将左递归转为**右递归**

(注: 右递归对应右结合; 需要在后续阶段进行额外处理)

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \beta_n$$

其中, β_i 都不以 A 开头

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$$

$$E \rightarrow E + \textcolor{red}{T} \mid \textcolor{red}{T}$$
$$T \rightarrow \textcolor{red}{T} * F \mid F$$
$$F \rightarrow (\textcolor{red}{E}) \mid \mathbf{id}$$

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \text{id}$$
$$E \rightarrow TE'$$
$$E' \rightarrow + TE' \mid \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow * FT' \mid \epsilon$$
$$F \rightarrow (E) \mid \text{id}$$

非直接左递归

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sb \mid \epsilon$$
$$S \implies Aa \implies Sda$$

非直接左递归

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Sb \mid \epsilon$$

$$S \implies Aa \implies Sda$$

- 1) 按照某个顺序将非终结符号排序为 A_1, A_2, \dots, A_n .
- 2) **for (从 1 到 n 的每个 i) {**
- 3) **for (从 1 到 $i - 1$ 的每个 j) {**
- 4) 将每个形如 $A_i \rightarrow A_j \gamma$ 的产生式替换为产生式组 $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$,
其中 $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ 是所有的 A_j 产生式
- 5) }
- 6) 消除 A_i 产生式之间的立即左递归
- 7) }

图 4-11 消除文法中的左递归的算法

$$A_k \rightarrow A_l \alpha \implies l > k$$

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sb \mid \epsilon$$
$$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$$
$$S \rightarrow Aa \mid b$$
$$A \rightarrow bdA' \mid A'$$
$$A' \rightarrow cA' \mid ada' \mid \epsilon$$
$$A_k \rightarrow A_l\alpha \implies l > k$$

$$E \rightarrow TE'$$
$$E' \rightarrow + TE' \mid \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow * FT' \mid \epsilon$$
$$F \rightarrow (E) \mid \text{id}$$

$$\text{FIRST}(F) = \{(, \text{id}\}$$

$$\text{FIRST}(T) = \{(, \text{id}\}$$

$$\text{FIRST}(E) = \{(, \text{id}\}$$

$$\text{FIRST}(E') = \{+, \epsilon\}$$

$$\text{FIRST}(T') = \{*, \epsilon\}$$

$$\text{FOLLOW}(E) = \text{FOLLOW}(E') = \{), \$\}$$

$$\text{FOLLOW}(T) = \text{FOLLOW}(T') = \{+,), \$\}$$

$$\text{FOLLOW}(F) = \{+, *,), \$\}$$

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

$$F \rightarrow (E) \mid \text{id}$$

非终结符号	输入符号					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T		$T \rightarrow FT'$		$T \rightarrow FT'$		
T'			$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$	$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

$$\text{FIRST}(F) = \{(, \text{id}\}$$

$$\text{FIRST}(T) = \{(, \text{id}\}$$

$$\text{FIRST}(E) = \{(, \text{id}\}$$

$$\text{FIRST}(E') = \{+, \epsilon\}$$

$$\text{FIRST}(T') = \{*, \epsilon\}$$

$$\text{FOLLOW}(E) = \text{FOLLOW}(E') = \{), \$\}$$

$$\text{FOLLOW}(T) = \text{FOLLOW}(T') = \{+,), \$\}$$

$$\text{FOLLOW}(F) = \{+, *,), \$\}$$

文件结束符 \$ 的必要性

已匹配	栈	输入	动作	
句型	E\$	id + id * id\$		
	TE'\$	id + id * id\$	输出	$E \rightarrow TE'$
	FT'E'\$	id + id * id\$	输出	$T \rightarrow FT'$
	id T'E'\$	id + id * id\$	输出	$F \rightarrow id$
id	T'E\$	+ id * id\$	匹配	id
id	E\$	+ id * id\$	输出	$T' \rightarrow \epsilon$
id	+ TE'\$	+ id * id\$	输出	$E' \rightarrow + TE'$
id +	TE'\$	id * id\$	匹配	+
id +	FT'E'\$	id * id\$	输出	$T \rightarrow FT'$
id +	id T'E\$	id * id\$	输出	$F \rightarrow id$
id + id	T'E\$	* id\$	匹配	id
id + id	* FT'E\$	* id\$	输出	$T' \rightarrow * FT'$
id + id *	FT'E\$	id\$	匹配	*
id + id *	id T'E\$	id\$	输出	$F \rightarrow id$
id + id * id	T'E\$	\$	匹配	id
id + id * id	E\$	\$	输出	$T' \rightarrow \epsilon$
id + id * id	\$	\$	输出	$E' \rightarrow \epsilon$

图 4-21 对输入 $id + id * id$ 进行预测分析时执行的步骤

$$S \rightarrow i \ E \ t \ S \mid i \ E \ t \ S \ e \ S \mid a$$
$$E \rightarrow b$$

提取左公因子

$$S \rightarrow i \ E \ t \ S \ S' \mid a$$
$$S' \rightarrow e \ S \mid \epsilon$$
$$E \rightarrow b$$

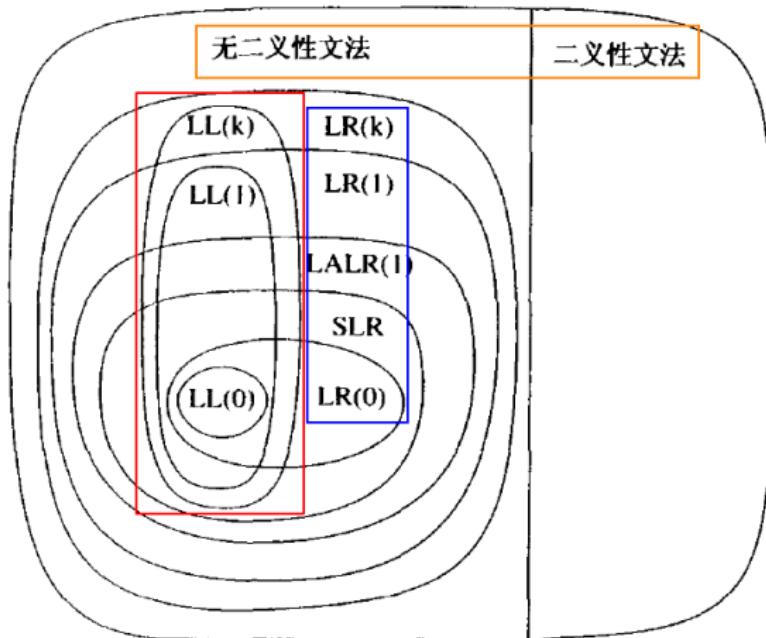
$$S \rightarrow i \ E \ t \ S \mid i \ E \ t \ S \ e \ S \mid a$$
$$E \rightarrow b$$

非终结符号	输入符号					
	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

解决二义性：选择 $S' \rightarrow eS$, 将 else 与前面最近的 then 关联起来

只考虑无二义性的文法

这意味着，每个句子对应唯一的一棵语法分析树



今日份主题: *LR* 语法分析器

自底向上的、
不断归约的、
基于句柄识别自动机的、
适用于 *LR* 文法的、
LR 语法分析器

自底向上构建语法分析树

根节点是文法的起始符号 S

叶节点是词法单元流 $w\$$

仅包含终结符号与特殊的**文件结束符** $\$$

自底向上构建语法分析树

根节点是文法的起始符号 S

每个中间非终结符节点表示使用它的某条产生式进行归约

叶节点是词法单元流 $w\$$

仅包含终结符号与特殊的文件结束符 $\$$

自顶向下的“推导”与自底向上的“归约”

$$E \xrightarrow{\text{rm}} T \xrightarrow{\text{rm}} T * F \xrightarrow{\text{rm}} T * \text{id} \xrightarrow{\text{rm}} F * \text{id} \xrightarrow{\text{rm}} \text{id} * \text{id}$$

(1) $E \rightarrow E + T$

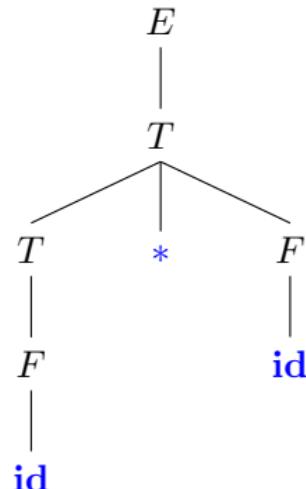
(2) $E \rightarrow T$

(3) $T \rightarrow T * F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow \text{id}$



$$w = \text{id} * \text{id}$$

$$E \Leftarrow T \Leftarrow T * F \Leftarrow T * \text{id} \Leftarrow F * \text{id} \Leftarrow \text{id} * \text{id}$$

“推导” ($A \rightarrow \alpha$) 与 “归约” ($A \leftarrow \alpha$)

$S \triangleq \gamma_0 \implies \dots \gamma_{i-1} \implies \gamma_i \implies \gamma_{r+1} \implies \dots \implies r_n = w$

$S \triangleq \gamma_0 \iff \dots \gamma_{i-1} \iff \gamma_i \iff \gamma_{r+1} \iff \dots \iff r_n = w$

自底向上语法分析器为输入构造**反向推导**

LR 语法分析器

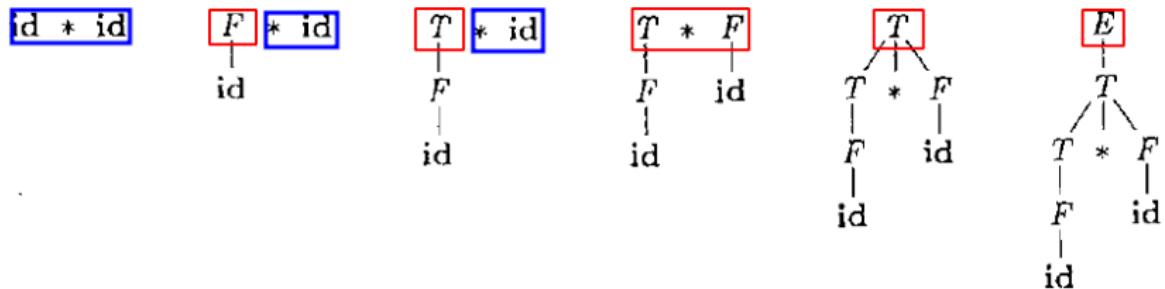
L：从左向右 (Left-to-right) 扫描输入

R：构建反向 (Reverse) 最右推导

“反向最右推导”与“从左到右扫描”相一致

LR 语法分析器的状态

在任意时刻, 语法分析树的上边缘与剩余的输入构成当前句型



LR 语法分析器使用栈存储语法分析树的上边缘

它包含了语法分析器目前所知的所有信息

板书演示“栈”上操作

(1) $E \rightarrow E + T$

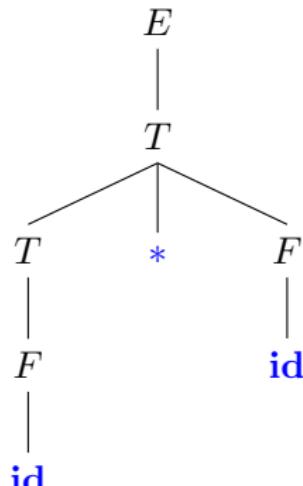
(2) $E \rightarrow T$

(3) $T \rightarrow T * F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow \text{id}$



$$w = \text{id} * \text{id}$$

两大操作: 移入输入符号 与 按产生式归约

直到栈中仅剩开始符号 S , 且输入已结束, 则成功停止

基于栈的 LR 语法分析器

Q_1 : 何时归约? (何时移入?)

Q_2 : 按哪条产生式进行归约?

基于栈的 LR 语法分析器

$$(1) E \rightarrow E + T$$

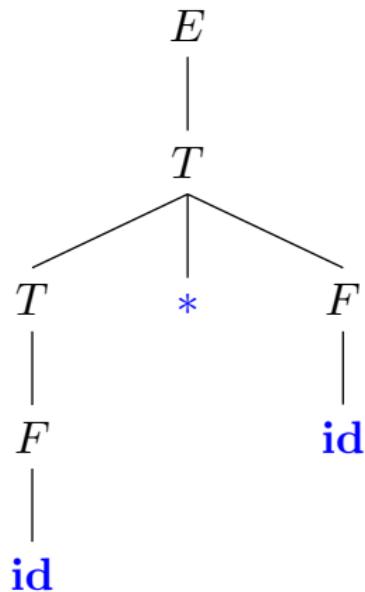
$$(2) E \rightarrow T$$

$$(3) T \rightarrow T * F$$

$$(4) T \rightarrow F$$

$$(5) F \rightarrow (E)$$

$$(6) F \rightarrow \text{id}$$



为什么第二个 F 以 $T * F$ 整体被归约为 T ?

这与 **栈** 的当前状态 “ $T * F$ ” 相关

LR 分析表 指导 LR 语法分析器

状态	ACTION					GOTO			
	id	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4			9	3	
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

在当前状态 (编号) 下, 面对当前文法符号时, 该采取什么动作

ACTION 表指明动作, GOTO 表仅用于归约时的状态转换

状态	ACTION					GOTO		
	id	+	*	()	\$	E	T	F
0	s5		s4			1	2	3
1		s6			acc			
2		r2	s7		r2	r2		
3		r4	r4		r4	r4		
4	s5		s4			8	2	3
5		r6	r6		r6	r6		
6	s5		s4				9	3
7	s5		s4					10
8		s6		s11				
9		r1	s7		r1	r1		
10		r3	r3		r3	r3		
11		r5	r5		r5	r5		

<i>sn</i>	移入输入符号，并进入 状态 n
<i>rk</i>	使用 k 号产生式 进行归约
<i>gn</i>	转换到 状态 n
<i>acc</i>	成功接受，结束
空白	错误

再次板书演示“栈”上操作：移入与归约

(1) $E \rightarrow E + T$

(2) $E \rightarrow T$

(3) $T \rightarrow T * F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow \text{id}$

状态	ACTION					GOTO			
	id	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

$$w = \text{id} * \text{id\$}$$

栈中存储语法分析器的状态（编号），“编码”了语法分析树的上边缘

```

1: procedure LR()
2:   PUSH( $S, s_0$ )                                 $\triangleright$  或 PUSH( $S, \$s_0$ )
3:   token  $\leftarrow$  NEXT-TOKEN()
4:   while (1) do
5:      $s \leftarrow$  TOP( $S$ )
6:     if ACTION[ $s, \text{token}$ ] =  $s_i$  then           $\triangleright$  移入
7:       PUSH( $S, i$ )                                 $\triangleright$  或 PUSH( $S, \text{token}_{s_i}$ )
8:       token  $\leftarrow$  NEXT-TOKEN()
9:     else if ACTION[ $s, \text{token}$ ] =  $r_j$  then       $\triangleright$  归约;  $j : A \rightarrow \alpha$ 
10:    | $\alpha$ | 次 POP( $S$ )
11:     $s \leftarrow$  TOP( $S$ )
12:    PUSH( $S, \text{GOTO}[s, A]$ )  $\triangleright$  转换状态; 或 PUSH( $S, A_{\text{GOTO}[s, A]}$ )
13:    else if ACTION[ $s, \text{token}$ ] = acc then         $\triangleright$  接受
14:      break
15:    else
16:      ERROR(...)

```

行号	栈	= 符号	输入	动作
(1)	0	\$	id * id \$	移入到 5
(2)	0 5	\$ id	* id \$	按照 $F \rightarrow id$ 归约
(3)	0 3	\$ F	* id \$	按照 $T \rightarrow F$ 归约
(4)	0 2	\$ T	* id \$	移入到 7
(5)	0 2 7	\$ T *	句型 id \$	移入到 5
(6)	0 2 7 5	\$ T * id	\$	按照 $F \rightarrow id$ 归约
(7)	0 2 7 10	\$ T * F	\$	按照 $T \rightarrow T * F$ 归约
(8)	0 2	\$ T	\$	按照 $E \rightarrow T$ 归约
(9)	0 1	\$ E	\$	接受

$w = \mathbf{id * id\$}$ 的分析过程

如何构造 LR 分析表?

状态	ACTION					GOTO			
	id	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2	r2	s7			r2	r2			
3	r4	r4			r4	r4			
4	s5		s4				8	2	3
5	r6	r6		r6	r6				
6	s5		s4				9	3	
7	s5		s4					10	
8		s6		s11					
9	r1	s7		r1	r1				
10	r3	r3		r3	r3				
11	r5	r5		r5	r5				

在**当前状态 (编号)**下, 面对**当前文法符号**时, 该采取什么**动作**

状态是什么？如何跟踪状态？

状态	ACTION					GOTO		
	id	+	*	()	\$	E	T	F
0	s5		s4			1	2	3
1		s6			acc			
2	r2	s7		r2	r2			
3	r4	r4		r4	r4			
4	s5		s4			8	2	3
5	r6	r6		r6	r6			
6	s5		s4			9	3	
7	s5		s4				10	
8		s6		s11				
9	r1	s7		r1	r1			
10	r3	r3		r3	r3			
11	r5	r5		r5	r5			

状态是语法分析树的上边缘，存储在栈中

可以用**自动机**跟踪状态变化 (自动机中的路径 \Leftrightarrow 栈中符号/状态编号)

何时归约？使用哪条产生式进行归约？

状态	ACTION					GOTO		
	id	+	*	()	\$	E	T	F
0	s5		s4			1	2	3
1		s6			acc			
2	r2	s7		r2	r2			
3	r4	r4		r4	r4			
4	s5		s4			8	2	3
5	r6	r6		r6	r6			
6	s5		s4			9	3	
7	s5		s4				10	
8	s6			s11				
9	r1	s7		r1	r1			
10	r3	r3		r3	r3			
11	r5	r5		r5	r5			

必要条件：当前状态中，已观察到某个产生式的完整右部

对于 LR 文法，这是当前**唯一**的选择

何时归约？使用哪条产生式进行归约？

Definition (句柄 (Handle))

在输入串的 (唯一) 反向最右推导中, 如果下一步是逆用产生式 $A \rightarrow \alpha$ 将 α 归约为 A , 则称 α 是当前句型的句柄。

最右句型	句柄	归约用的产生式
$id_1 * id_2$	id_1	$F \rightarrow id$
$F * id_2$	F	$T \rightarrow F$
$T * id_2$	id_2	$F \rightarrow id$
$T * F$	$T * F$	$T \rightarrow T * F$
T	T	$E \rightarrow T$

LR 语法分析器的关键就是高效寻找每个归约步骤所使用的句柄。

句柄可能在哪里？

Theorem

存在一种 LR 语法分析方法，保证句柄总是出现在栈顶。

句柄可能在哪里？

Theorem

存在一种 LR 语法分析方法，保证句柄总是出现在栈顶。

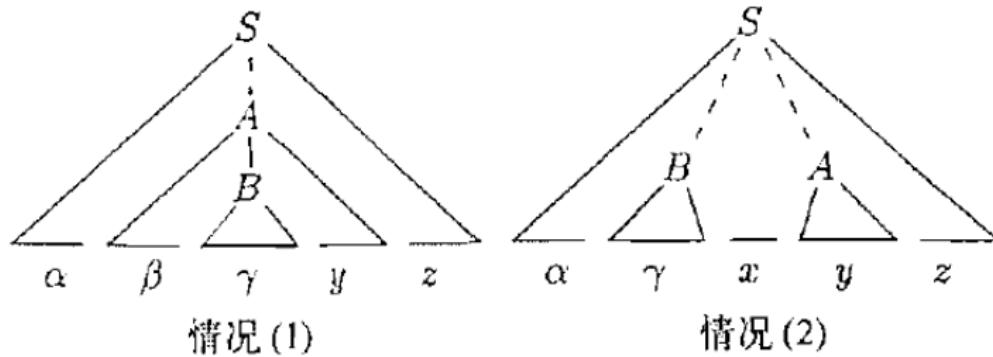


图 4-29 一个最右推导中两个连续步骤的两种情况

$$S \xrightarrow[\text{rm}]{*} \alpha Az \xrightarrow[\text{rm}]{*} \alpha \beta Byz \xrightarrow[\text{rm}]{*} \alpha \beta \gamma yz \quad S \xrightarrow[\text{rm}]{*} \alpha BxAz \xrightarrow[\text{rm}]{*} \alpha Bxyz \xrightarrow[\text{rm}]{*} \alpha \gamma xyz$$

可以用**自动机**跟踪状态变化
(自动机中的路径 \Leftrightarrow 栈中符号/状态编号)

Theorem

存在一种 LR 语法分析方法，保证**句柄总是出现在栈顶**。

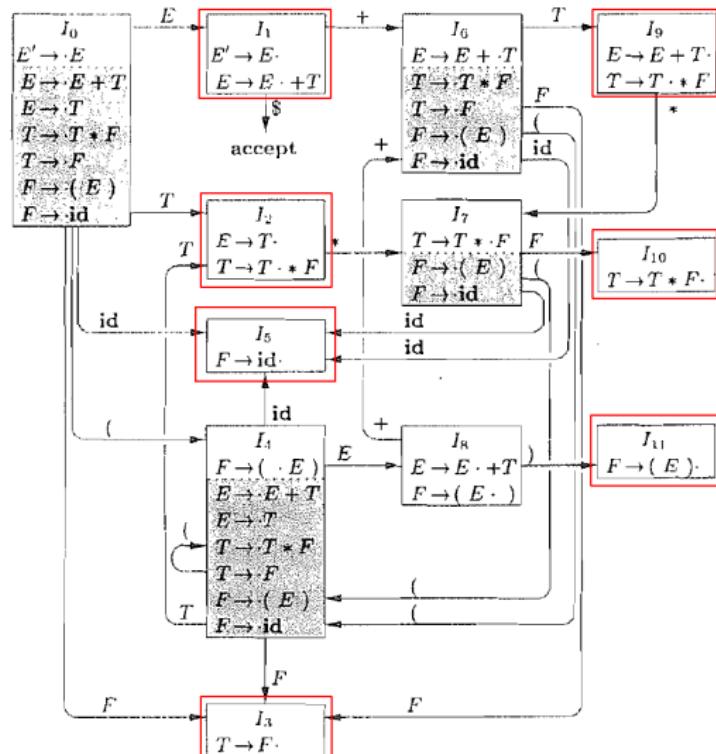
可以用**自动机**跟踪状态变化
(自动机中的路径 \Leftrightarrow 栈中符号/状态编号)

Theorem

存在一种 LR 语法分析方法，保证**句柄总是出现在栈顶**。

希望能够在自动机的当前状态识别可能的句柄

LR(0) 句柄识别有穷状态自动机 (Handle-Finding Automaton)



状态是什么?

状态刻画了“当前观察到的**针对所有产生式的右部的前缀**”

Definition ($LR(0)$ 项 (Item))

文法 G 的一个 $LR(0)$ 项是 G 的某个产生式加上一个位于体部的**点**。

项指明了语法分析器已经观察到了某个产生式的某个前缀

状态刻画了“当前观察到的**针对所有产生式的右部的前缀**”

Definition (*LR(0)* 项 (Item))

文法 G 的一个 *LR(0)* 项是 G 的某个产生式加上一个位于体部的**点**。

项指明了语法分析器已经观察到了某个产生式的某个前缀

$$A \rightarrow XYZ$$

$$[A \rightarrow \cdot XYZ]$$

$$[A \rightarrow X \cdot YZ]$$

$$[A \rightarrow XY \cdot Z]$$

$$[A \rightarrow XYZ \cdot]$$

(产生式 $A \rightarrow \epsilon$ 只有一个项 $[A \rightarrow \cdot]$)

状态刻画了“当前观察到的**针对所有产生式的右部的前缀**”

Definition (项集)

项集就是若干**项**构成的集合。

因此, 句柄识别自动机的一个**状态**可以表示为一个**项集**

状态刻画了“当前观察到的**针对所有产生式的右部的前缀**”

Definition (项集)

项集就是若干**项**构成的集合。

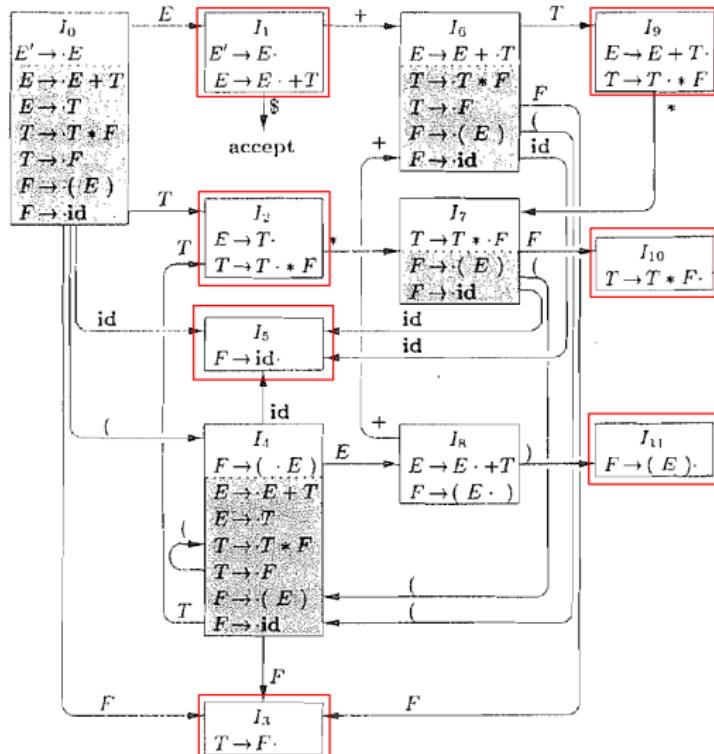
因此, 句柄识别自动机的一个**状态**可以表示为一个**项集**

Definition (项集族)

项集族就是若干**项集**构成的集合。

因此, 句柄识别自动机的**状态集**可以表示为一个**项集族**

LR(0) 句柄识别自动机



项、项集、项集族

Definition (增广文法 (Augmented Grammar))

文法 G 的**增广文法** G' 是在 G 中加入产生式 $S' \rightarrow S$ 得到的文法。

目的: 告诉语法分析器何时停止分析并接受输入符号串

当语法分析器**面对 \$ 且要使用 $S' \rightarrow S$ 进行归约**时, 输入符号串被接受

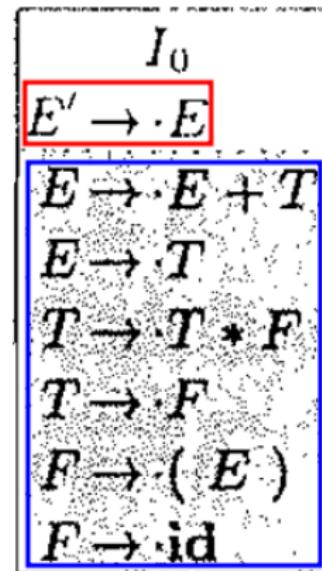
LR(0) 句柄识别自动机



初始状态是什么?

点指示了栈顶, 左边(与路径)是栈中内容, 右边是期望看到的文法符号串

- (0) $E' \rightarrow E$
- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow T * F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow \text{id}$



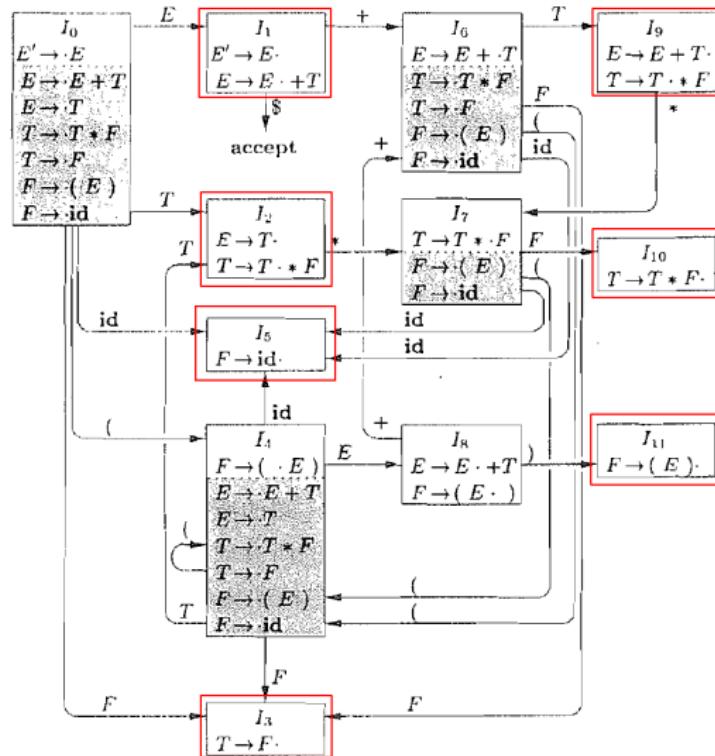
CLOSURE($\{[E' \rightarrow \cdot E]\}$)

$LR(0)$ 句柄识别自动机



状态之间如何转移?

板书演示 $LR(0)$ 句柄识别自动机的构造过程



状态编号约定

```

SetOfItems CLOSURE( $I$ ) {
     $J = I;$ 
    repeat
        for ( $J$  中的每个项  $A \rightarrow \alpha \cdot B\beta$ )
            for ( $G$  的每个产生式  $B \rightarrow \gamma$ )
                if (项  $B \rightarrow \cdot\gamma$  不在  $J$  中)
                    将  $B \rightarrow \cdot\gamma$  加入  $J$  中;
    until 在某一轮中没有新的项被加入到  $J$  中;
    return  $J$ ;
}

```

$$J = \text{GOTO}(I, \textcolor{red}{X}) = \text{CLOSURE}\left(\left\{ [A \rightarrow \alpha X \cdot \beta] \mid [A \rightarrow \alpha \cdot X\beta] \in I \right\}\right)$$
$$(X \in N \cup T)$$

```

void items( $G'$ ) {
     $C = \{\text{CLOSURE}(\{[S' \rightarrow \cdot S]\})\}$ ; 初始状态
    repeat
        for ( $C$ 中的每个项集  $I$ )
            for (每个文法符号  $X$ )
                if ( $\text{GOTO}(I, X)$  非空且不在  $C$  中)
                    下一个状态 将  $\text{GOTO}(I, X)$  加入  $C$  中;
        until 在某一轮中没有新的项集被加入到  $C$  中;
}

```

图 4-33 规范 LR(0) 项集族的计算

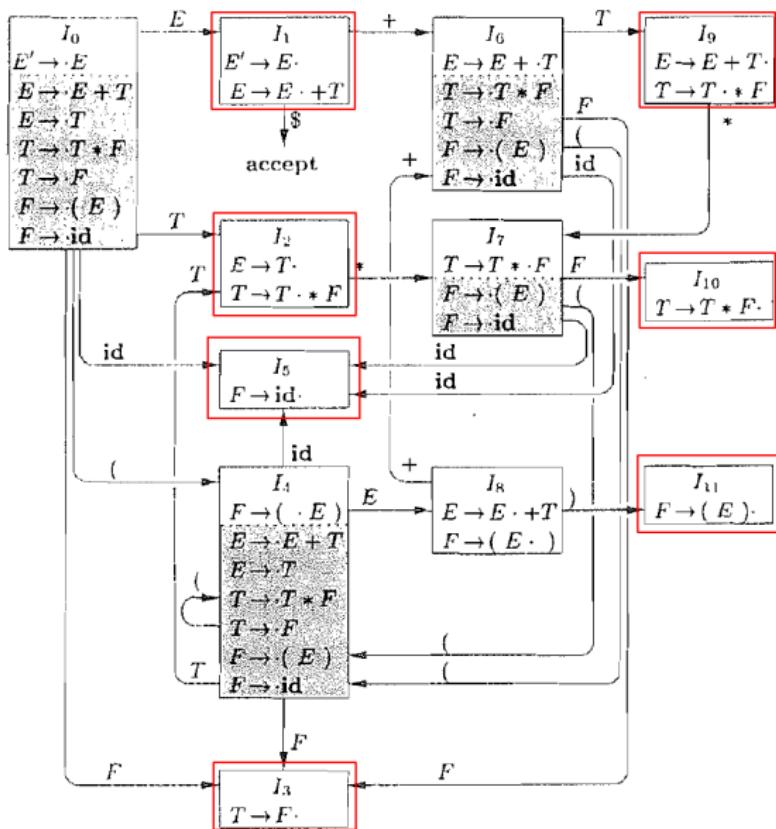
```

void items( $G'$ ) {
     $C = \{\text{CLOSURE}(\{[S' \rightarrow \cdot S]\})\}$ ; 初始状态
    repeat
        for ( $C$ 中的每个项集  $I$ )
            for (每个文法符号  $X$ )
                if ( $\text{GOTO}(I, X)$  非空且不在  $C$ 中)
                    下一个状态 将  $\text{GOTO}(I, X)$  加入  $C$ 中;
        until 在某一轮中没有新的项集被加入到  $C$ 中;
}

```

图 4-33 规范 LR(0) 项集族的计算

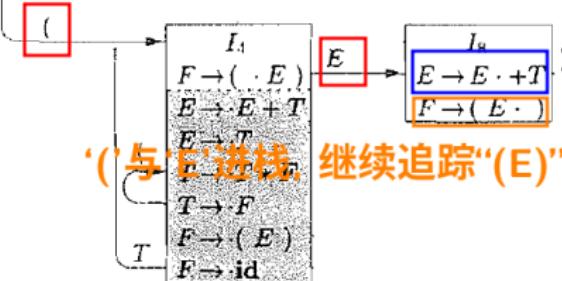
接受状态: $F = \{I \in C \mid \exists k. [k : A \rightarrow \alpha \cdot] \in I\}$



红色框中的状态为 接受状态

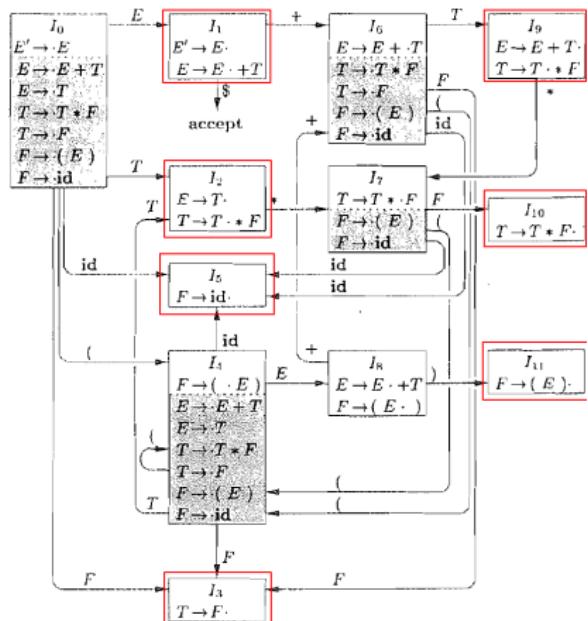
I_0
 $E' \rightarrow \cdot E$
 $E \rightarrow \cdot E + T$
 $E \rightarrow \cdot T$
 $T \rightarrow \cdot T * F$
 $T \rightarrow \cdot F$
 $F \rightarrow (\cdot E)$
 $F \rightarrow \text{id}$

‘(’与‘E’进栈，转而追踪‘E+T’



点指示了栈顶，左边（与路径）是栈中内容，右边是期望看到的文法符号串

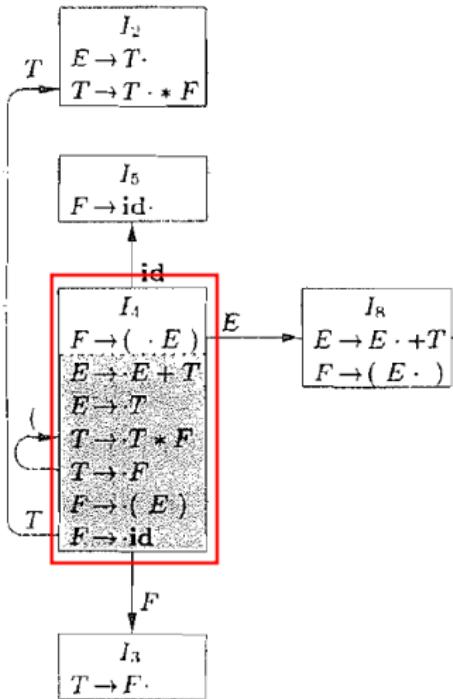
LR(0) 分析表



	ACTION					GOTO			
	<code>id</code>	<code>+</code>	<code>*</code>	<code>(</code>	<code>)</code>	<code>\$</code>	<code>E</code>	<code>T</code>	<code>F</code>
0	s_5					s_4	g_1	g_2	g_3
1		s_6							
2	r_2	r_2	s_7, r_2	r_2	r_2	r_2			
3	r_4	r_4		r_4	r_4	r_4			
4	s_5					s_4	g_8	g_2	g_3
5	r_6	r_6		r_6	r_6	r_6			
6	s_5					s_4	g_9	g_3	
7	s_5					s_4			g_{10}
8		s_6							
9	r_1	r_1	s_7, r_1	r_1	r_1	r_1			
10	r_3	r_3		r_3	r_3	r_3			
11	r_5	r_5		r_5	r_5	r_5			

GOTO 函数被拆分成 ACTION 表 (针对终结符) 与 GOTO 表 (针对非终结符)

(1) $\text{GOTO}(I_i, a) = I_j \wedge a \in T \implies \text{ACTION}[i, a] \leftarrow s_j$



	ACTION						GOTO		
	id	+	*	()	\$	E	T	F
0	s5					s4			
1									acc
2	r2	r2	s7, r2	r2	r2	r2			
3	r4	r4	r4	r4	r4	r4			
4	s5					s4			
5	r6	r6	r6	r6	r6	r6			
6	s5					s4			
7	s5					s4			
8		s6							s11
9	r1	r1	s7, r1	r1	r1	r1			
10	r3	r3	r3	r3	r3	r3			
11	r5	r5	r5	r5	r5	r5			

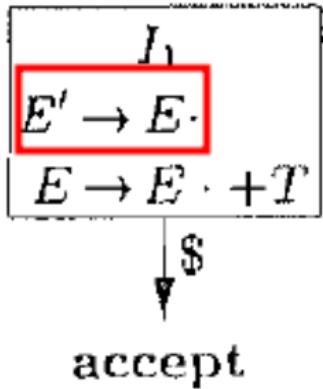
(2) $\text{GOTO}(I_i, A) = I_j \wedge A \in N \implies \text{GOTO}[i, A] \leftarrow g_j$

L_1
 $E \rightarrow T \cdot$
 $T \rightarrow T \cdot * F$

L_{10}
 $T \rightarrow T * F \cdot$

	ACTION						GOTO		
	id	+	*	()	\$	E	T	F
0	$s5$			$s4$			$g1$	$g2$	$g3$
1		$s6$				acc			
2	$r2$	$r2$	$s7, r2$	$r2$	$r2$	$r2$			
3	$r4$	$r4$	$r4$	$r4$	$r4$	$r4$			
4	$s5$			$s4$			$g8$	$g2$	$g3$
5	$r6$	$r6$	$r6$	$r6$	$r6$	$r6$			
6	$s5$			$s4$				$g9$	$g3$
7	$s5$			$s4$					$g10$
8		$s6$				$s11$			
9	$r1$	$r1$	$s7, r1$	$r1$	$r1$	$r1$			
10	$r3$	$r3$	$r3$	$r3$	$r3$	$r3$			
11	$r5$	$r5$	$r5$	$r5$	$r5$	$r5$			

(3) $[k : A \rightarrow \alpha \cdot] \in I_i \wedge A \neq S' \implies \forall t \in T \cup \{\$\}, \text{ACTION}[i, t] = rk$



	ACTION						GOTO		
	id	+	*	()	\$	E	T	F
0	s_5				s_4		g_1	g_2	g_3
1		s_6				<i>acc</i>			
2	r_2	r_2	<i>s7, r2</i>	r_2	r_2	r_2			
3	r_4	r_4	r_4	r_4	r_4	r_4			
4	s_5			s_4			g_8	g_2	g_3
5	r_6	r_6	r_6	r_6	r_6	r_6			
6	s_5			s_4				g_9	g_3
7	s_5			s_4					g_{10}
8		s_6				s_{11}			
9	r_1	r_1	<i>s7, r1</i>	r_1	r_1	r_1			
10	r_3	r_3	r_3	r_3	r_3	r_3			
11	r_5	r_5	r_5	r_5	r_5	r_5			

(4) $[S' \rightarrow S\cdot] \in I_i \implies \text{ACTION}[i, \$] \leftarrow acc$

LR(0) 分析表构造规则

- (1) $\text{GOTO}(I_i, a) = I_j \wedge a \in T \implies \text{ACTION}[i, a] \leftarrow sj$
- (2) $\text{GOTO}(I_i, A) = I_j \wedge A \in N \implies \text{GOTO}[i, A] \leftarrow gj$
- (3) $[k : A \rightarrow \alpha \cdot] \in I_i \wedge A \neq S' \implies \forall t \in T \cup \{\$\}$. $\text{ACTION}[i, t] = rk$
- (4) $[S' \rightarrow S \cdot] \in I_i \implies \text{ACTION}[i, \$] \leftarrow acc$

Definition ($LR(0)$ 文法)

如果文法 G 的 $LR(0)$ 分析表是无冲突的, 则 G 是 $LR(0)$ 文法。

	ACTION						GOTO		
	id	+	*	()	\$	E	T	F
0	s_5			s_4			g_1	g_2	g_3
1		s_6				acc			
2	r_2	r_2	s_7, r_2	r_2	r_2	r_2			
3	r_4	r_4	r_4	r_4	r_4	r_4			
4	s_5			s_4			g_8	g_2	g_3
5	r_6	r_6	r_6	r_6	r_6	r_6			
6	s_5			s_4			g_9	g_3	
7	s_5			s_4				g_{10}	
8		s_6			s_{11}				
9	r_1	r_1	s_7, r_1	r_1	r_1	r_1			
10	r_3	r_3	r_3	r_3	r_3	r_3			
11	r_5	r_5	r_5	r_5	r_5	r_5			

非 $LR(0)$ 分析表/文法

LR(0) 分析表每一行 (状态) 所选用的归约产生式是相同的

	ACTION						GOTO		
	id	+	*	()	\$	E	T	F
0	s_5			s_4			g_1	g_2	g_3
1		s_6				<i>acc</i>			
2	r_2	r_2	s_7, r_2	r_2	r_2	r_2			
3	r_4	r_4	r_4	r_4	r_4	r_4			
4	s_5			s_4			g_8	g_2	g_3
5	r_6	r_6	r_6	r_6	r_6	r_6			
6	s_5			s_4			g_9	g_3	
7	s_5			s_4					g_{10}
8		s_6			s_{11}				
9	r_1	r_1	s_7, r_1	r_1	r_1	r_1			
10	r_3	r_3	r_3	r_3	r_3	r_3			
11	r_5	r_5	r_5	r_5	r_5	r_5			

归约时不需要向前看, 这就是“0”的含义

$LR(0)$ 语法分析器

L : 从左向右 (Left-to-right) 扫描输入

R : 构建反向 (Reverse) 最右推导

O : 归约时无需向前看

$LR(0)$ 自动机与栈之间的互动关系

向前走 \Leftrightarrow 移入

回溯 \Leftrightarrow 归约

自动机才是本质，栈是实现方式
(用栈记住“来时的路”，以便回溯)

SLR(1) 分析表

状态	ACTION					GOTO		
	id	+	*	()	\$	E	T	F
0	s5			s4		1	2	3
1		s6						
2		r2	s7		r2	r2		
3		r4	r4		r4	r4		
4	s5			s4		8	2	3
5		r6	r6		r6	r6		
6	s5			s4			9	3
7	s5			s4				10
8		s6			s11			
9		r1	s7		r1	r1		
10		r3	r3		r3	r3		
11		r5	r5		r5	r5		

归约:

(3) $[k : A \rightarrow \alpha \cdot] \in I_i \wedge A \neq S' \implies \forall t \in \text{FOLLOW}(A). \text{ACTION}[i, t] = rk$

Definition (*SLR(1)* 文法)

如果文法 G 的 ***SLR(1)* 分析表** 是无冲突的, 则 G 是 *SLR(1)* 文法。

无冲突: ACTION 表中每个单元格最多只有一种动作

状态	ACTION					GOTO			
	id	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4			9	3	
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

两类可能的冲突: “移入/归约” 冲突、“归约/归约” 冲突

非 SLR(1) 文法举例

$S \rightarrow L = R \mid R$

$L \rightarrow * R \mid \text{id}$

$R \rightarrow L$

$I_0: \begin{array}{l} S' \rightarrow \cdot S \\ S \rightarrow \cdot L = R \\ S \rightarrow \cdot R \\ L \rightarrow \cdot * R \\ L \rightarrow \cdot \text{id} \\ R \rightarrow \cdot L \end{array}$

$I_1: S' \rightarrow S \cdot$

$I_2: \begin{array}{l} S \rightarrow L \cdot = R \\ R \rightarrow L \cdot \end{array}$

$I_3: S \rightarrow R \cdot$

$I_4: \begin{array}{l} L \rightarrow * \cdot R \\ R \rightarrow \cdot L \\ L \rightarrow \cdot * R \\ L \rightarrow \cdot \text{id} \end{array}$

$I_5: L \rightarrow \text{id} \cdot$

$I_6: \begin{array}{l} S \rightarrow L = \cdot R \\ R \rightarrow \cdot L \\ L \rightarrow \cdot * R \\ L \rightarrow \cdot \text{id} \end{array}$

$I_7: L \rightarrow * R \cdot$

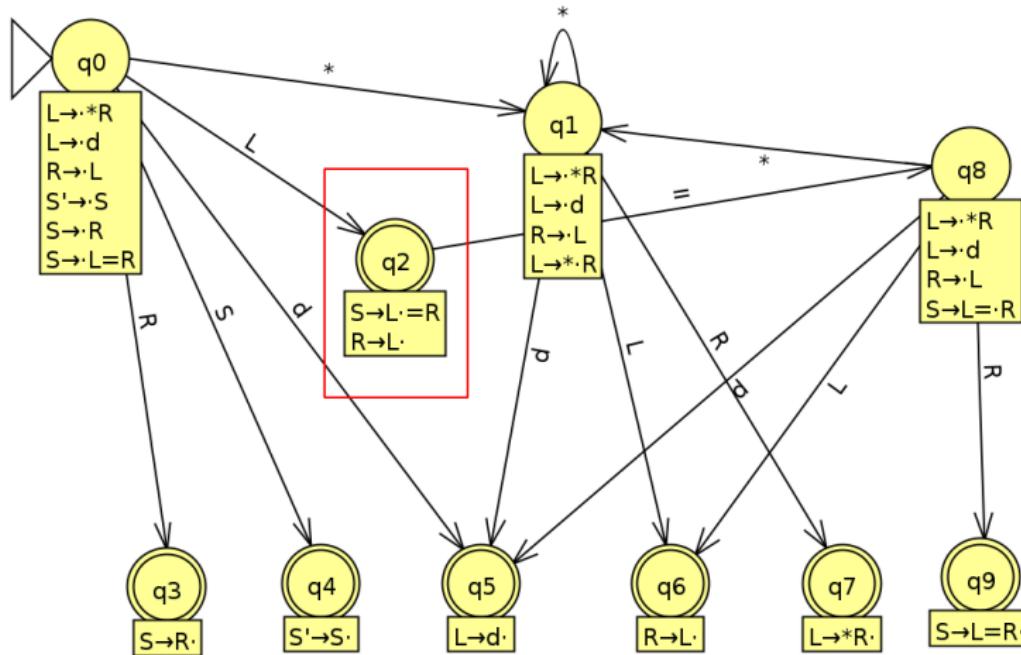
$I_8: R \rightarrow L \cdot$

$I_9: S \rightarrow L = R \cdot$

$[S \rightarrow L \cdot = R] \in I_2 \implies \text{ACTION}(I_2, =) \leftarrow s6$

$= \in \text{FOLLOW}(R) \implies \text{ACTION}(I_2, =) \leftarrow r5$

即使考虑了 $= \in FOLLOW(A)$, 对该文法来说仍然不够
因为, 这仅仅说明在某个句型中, a 可以跟在 A 后面



该文法没有以 $R = \dots$ 开头的最右句型

希望 LR 语法分析器的每个状态能**尽可能精确地**
指明哪些输入符号可以跟在句柄 $A \rightarrow \alpha$ 的后面

希望 LR 语法分析器的每个状态能尽可能精确地
指明哪些输入符号可以跟在句柄 $A \rightarrow \alpha$ 的后面

在 $LR(0)$ 自动机中, 某个项集 I_j 中包含 $[A \rightarrow \alpha \cdot]$
则在之前的某个项集 I_i 中包含 $[B \rightarrow \beta \cdot A\gamma]$ 与 $[A \rightarrow \cdot \alpha]$

这表明只有 $a \in \text{FIRST}(\gamma)$ 时, 才可以进行 $A \rightarrow \alpha$ 归约

希望 LR 语法分析器的每个状态能尽可能精确地
指明哪些输入符号可以跟在句柄 $A \rightarrow \alpha$ 的后面

在 $LR(0)$ 自动机中, 某个项集 I_j 中包含 $[A \rightarrow \alpha \cdot]$
则在之前的某个项集 I_i 中包含 $[B \rightarrow \beta \cdot A\gamma]$ 与 $[A \rightarrow \cdot \alpha]$

这表明只有 $a \in \text{FIRST}(\gamma)$ 时, 才可以进行 $A \rightarrow \alpha$ 归约

但是, 对 I_i 求闭包时, 仅得到 $[A \rightarrow \cdot \alpha]$, 丢失了 $\text{FIRST}(\gamma)$ 信息

Definition ($LR(1)$ 项 (Item))

$$[A \rightarrow \alpha \cdot \beta, a] \quad (a \in T \cup \{\$\})$$

此处, a 是向前看符号, 数量为 1.

Definition ($LR(1)$ 项 (Item))

$$[A \rightarrow \alpha \cdot \beta, a] \quad (a \in T \cup \{\$\})$$

此处, a 是向前看符号, 数量为 1.

思想: α 在栈顶, 且剩余输入中开头的是可以从 βa 推导出的符号串

Definition ($LR(1)$ 项 (Item))

$$[A \rightarrow \alpha \cdot \beta, a] \quad (a \in T \cup \{\$\})$$

此处, a 是向前看符号, 数量为 1.

思想: α 在栈顶, 且剩余输入中开头的是可以从 βa 推导出的符号串

$$[A \rightarrow \alpha \cdot, a]$$

只有下一个输入符号为 a 时, 才可以按照 $A \rightarrow \alpha$ 进行归约

LR(1)句柄识别自动机

$$[A \rightarrow \alpha \cdot B\beta, a] \in I \quad (a \in T \cup \{\$\})$$

```
SetOfItems CLOSURE(I) {
    repeat
        for ( I 中的每个项  $[A \rightarrow \alpha \cdot B\beta, a]$  )
            for (  $G'$  中的每个产生式  $B \rightarrow \gamma$  )
                for ( FIRST( $\beta a$ ) 中的每个终结符号  $b$  )
                    将  $[B \rightarrow \cdot \gamma, b]$  加入到集合  $I$  中;
    until 不能向  $I$  中加入更多的项 ;
    return  $I$ ;
}
```

$$\forall b \in \text{FIRST}(\beta a). [B \rightarrow \cdot \gamma, b] \in I$$

LR(1)句柄识别自动机

```
SetOfItems GOTO( $I, X$ ) {  
    将  $J$  初始化为空集；  
    for ( $I$  中的每个项  $[A \rightarrow \alpha \cdot X\beta, a]$ )  
        将项  $[A \rightarrow \alpha X \cdot \beta, a]$  加入到集合  $J$  中；  
    return CLOSURE( $J$ );  
}
```

$$J = \text{GOTO}(I, X) = \text{CLOSURE}\left(\left\{ [A \rightarrow \alpha X \cdot \beta, a] \mid [A \rightarrow \alpha \cdot X\beta, a] \in I \right\}\right)$$
$$(X \in N \cup T)$$

LR(1)句柄识别自动机

```
void items( $G'$ ) {  
    将  $C$  初始化为 CLOSURE ( $\{[S' \rightarrow \cdot S, \$]\}$ )  
    repeat  
        for (  $C$  中的每个项集  $I$  )  
            for ( 每个文法符号  $X$  )  
                if ( GOTO( $I, X$ ) 非空且不在  $C$  中 )  
                    将 GOTO( $I, X$ ) 加入  $C$  中;  
    until 不再有新的项集加入到  $C$  中;  
}
```

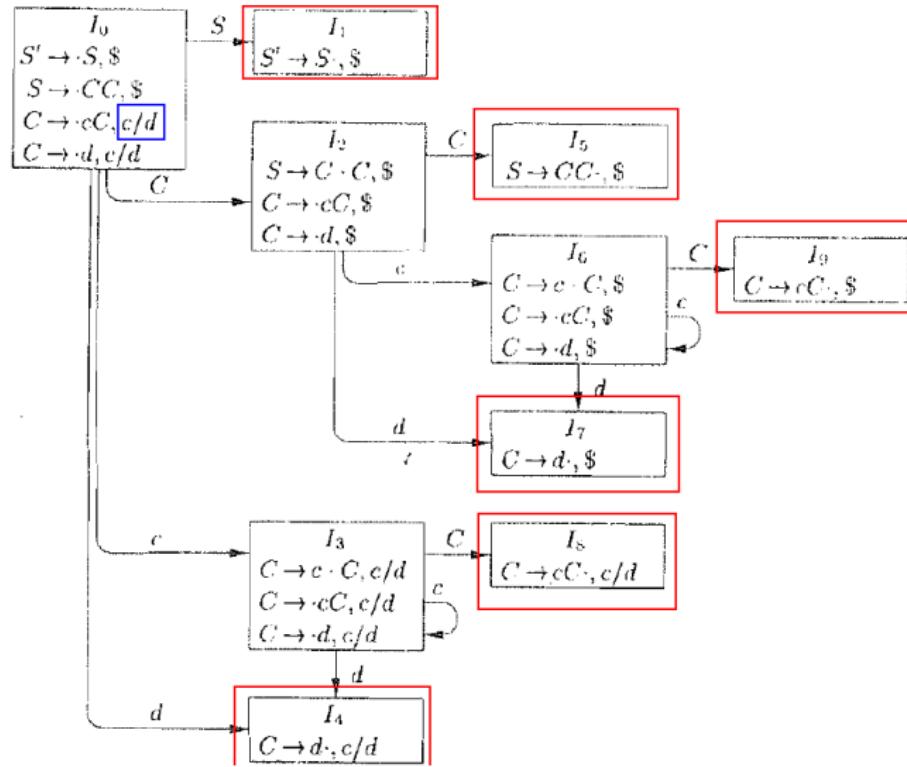
初始状态: CLOSURE($[S' \rightarrow \cdot S, \$]$)

板书演示: LR(1) 自动机的构造过程

$S' \rightarrow S$

$S \rightarrow C \ C$

$C \rightarrow c \ C \mid d$



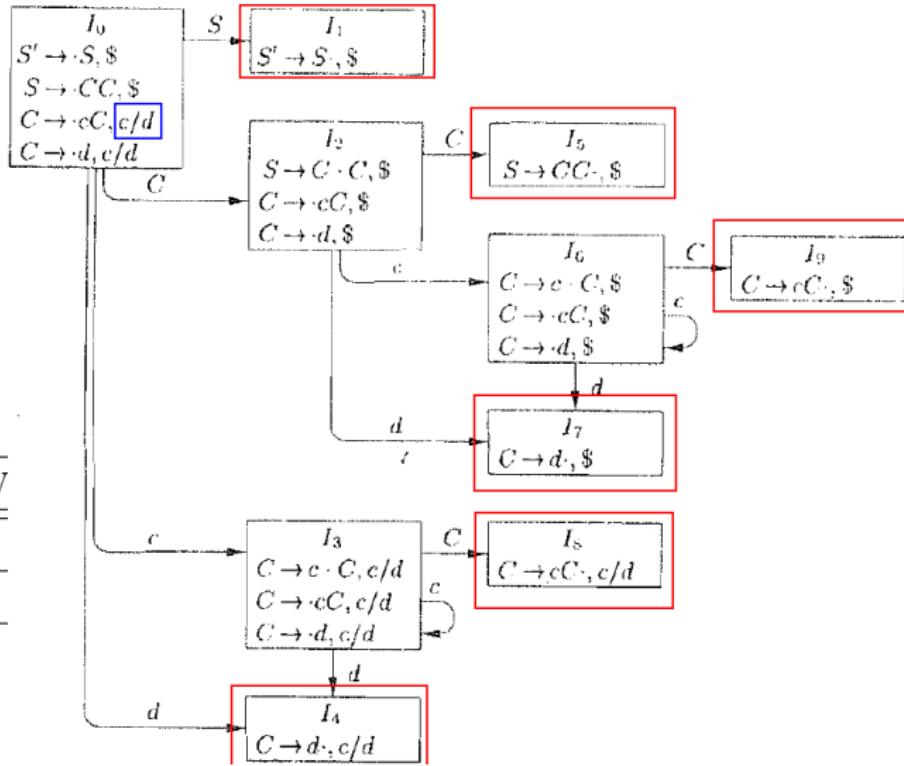
板书演示: LR(1) 自动机的构造过程

$S' \rightarrow S$

$S \rightarrow C \ C$

$C \rightarrow c \ C \mid d$

	FIRST	FOLLOW
S	{c, d}	\$
C	{c, d}	{c, d, \$}



LR(1) 分析表构造规则

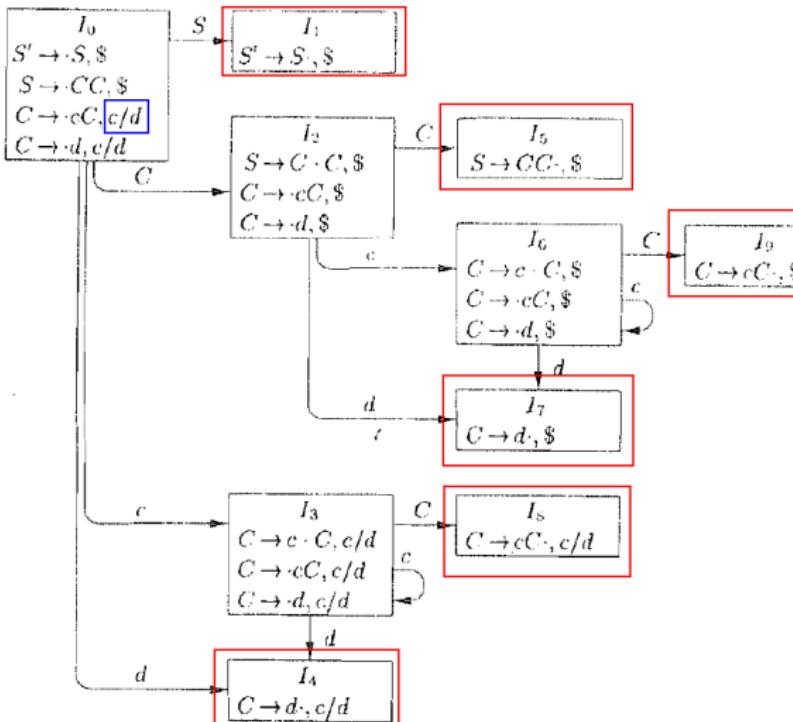
- (1) $\text{GOTO}(I_i, a) = I_j \wedge a \in T \implies \text{ACTION}[i, a] \leftarrow sj$
- (2) $\text{GOTO}(I_i, A) = I_j \wedge A \in T \implies \text{GOTO}[i, A] \leftarrow gj$
- (3) $[k : A \rightarrow \alpha \cdot, a] \in I_i \wedge A \neq S' \implies \text{ACTION}[i, a] = rk$
- (4) $[S' \rightarrow S \cdot, \$] \in I_i \implies \text{ACTION}[i, \$] \leftarrow acc$

LR(1) 分析表构造规则

- (1) $\text{GOTO}(I_i, a) = I_j \wedge a \in T \implies \text{ACTION}[i, a] \leftarrow sj$
- (2) $\text{GOTO}(I_i, A) = I_j \wedge A \in T \implies \text{GOTO}[i, A] \leftarrow gj$
- (3) $[k : A \rightarrow \alpha \cdot, a] \in I_i \wedge A \neq S' \implies \text{ACTION}[i, a] = rk$
- (4) $[S' \rightarrow S \cdot, \$] \in I_i \implies \text{ACTION}[i, \$] \leftarrow acc$

Definition (LR(1) 文法)

如果文法 G 的 **LR(1)** 分析表是无冲突的，则 G 是 $LR(1)$ 文法。



状态	ACTION			GOTO	
	c	d	\$	S	C
0	s3	s4		1	2
1					
2	s6	s7		5	
3	s3	s4		8	
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

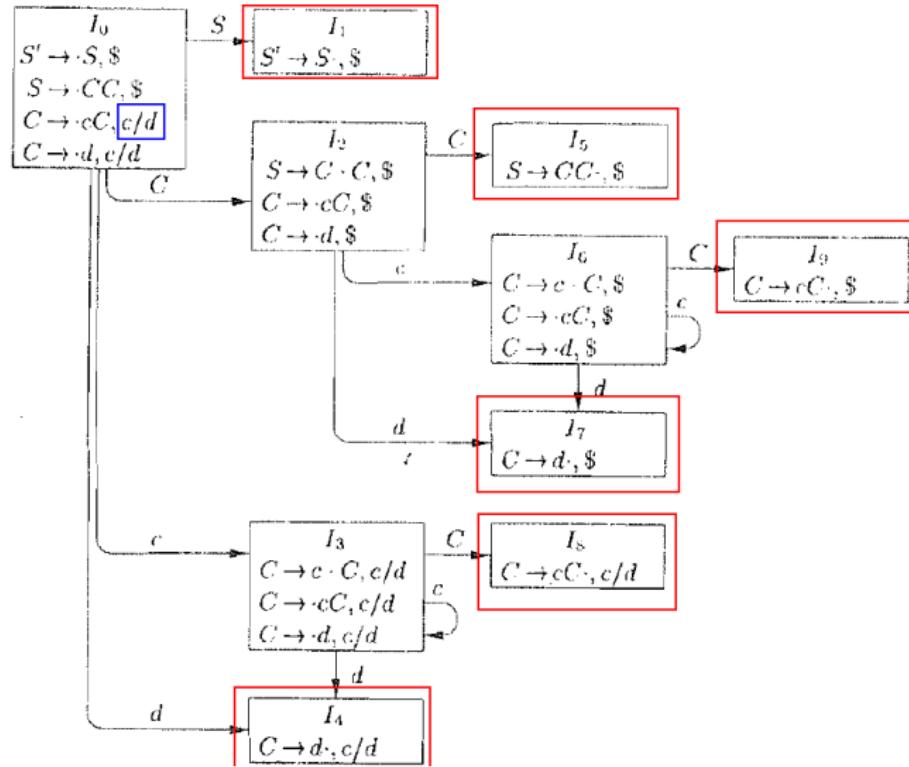
LR(1) 通过不同的向前看符号, 区分了状态对 (3, 6), (4, 7) 与 (8, 9)

$S' \rightarrow S$

$S \rightarrow C \ C$

$C \rightarrow c \ C \mid d$

$L(G) =$

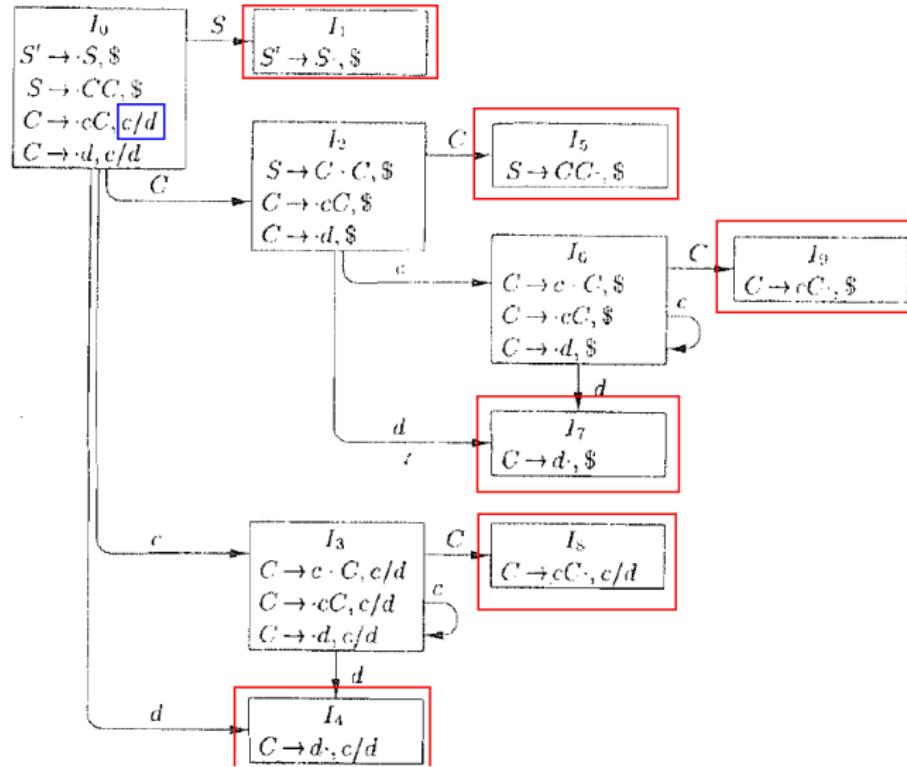


$S' \rightarrow S$

$S \rightarrow C \ C$

$C \rightarrow c \ C \mid d$

$$L(G) = c^*dc^*d$$



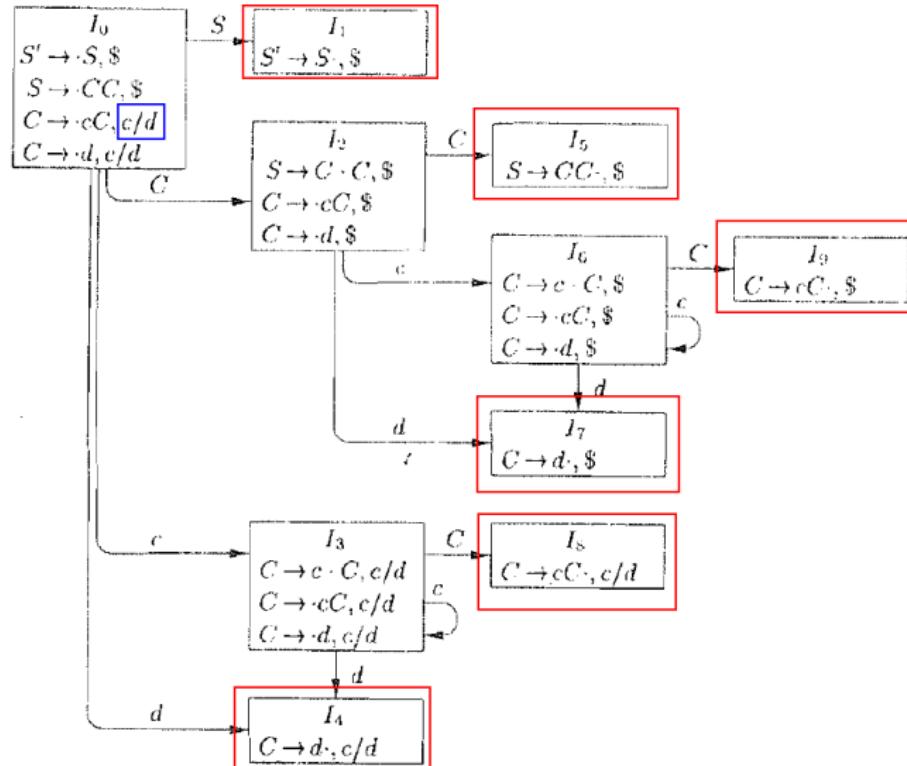
$w = cc d c d \$$

$S' \rightarrow S$

$S \rightarrow C \ C$

$C \rightarrow c \ C \mid d$

$L(G) = c^* d c^* d$



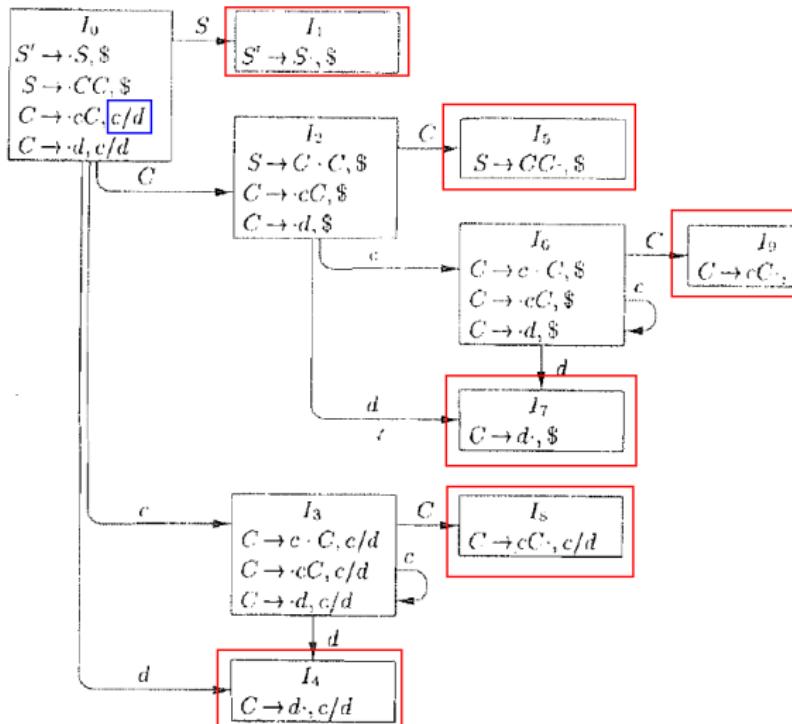
总结: $LR(0)$ 、 $SLR(1)$ 、 $LR(1)$ 的归约条件

$[k : A \rightarrow \alpha \cdot] \in I_i \wedge A \neq S' \implies \forall t \in T \cup \{\$\}$. ACTION[i, t] = rk

$[k : A \rightarrow \alpha \cdot] \in I_i \wedge A \neq S' \implies \forall t \in FOLLOW(A)$. ACTION[i, t] = rk

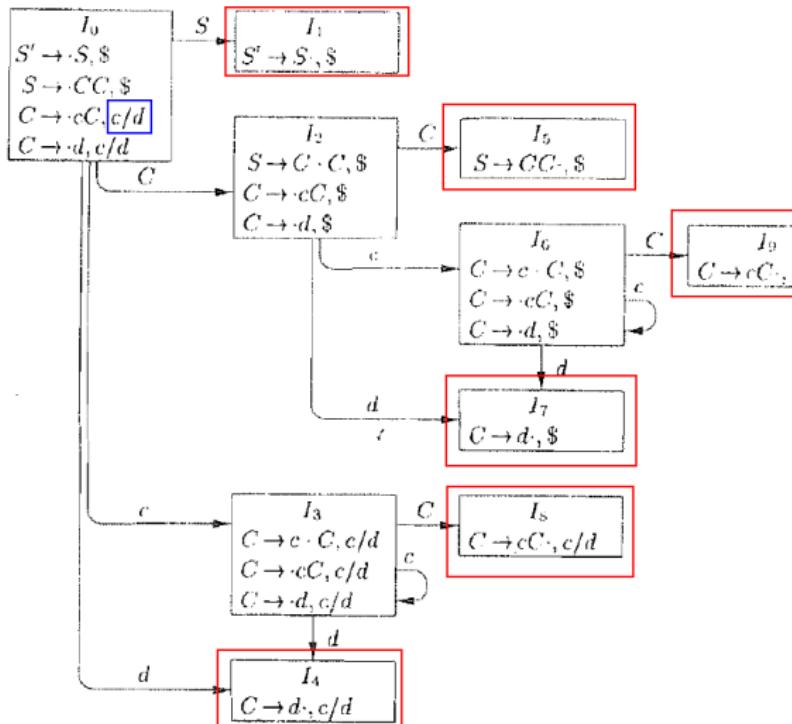
$[k : A \rightarrow \alpha \cdot, a] \in I_i \wedge A \neq S' \implies$ ACTION[i, a] = rk

$LR(1)$ 虽然强大, 但是生成的 $LR(1)$ 分析表可能过大, 状态过多



状态	ACTION			GOTO	
	c	d	\$	S	C
0	s3	s4		1	2
1					
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5				r1	
6	s6	s7			9
7				r3	
8	r2	r2			
9				r2	

$LR(1)$ 虽然强大, 但是生成的 $LR(1)$ 分析表可能过大, 状态过多



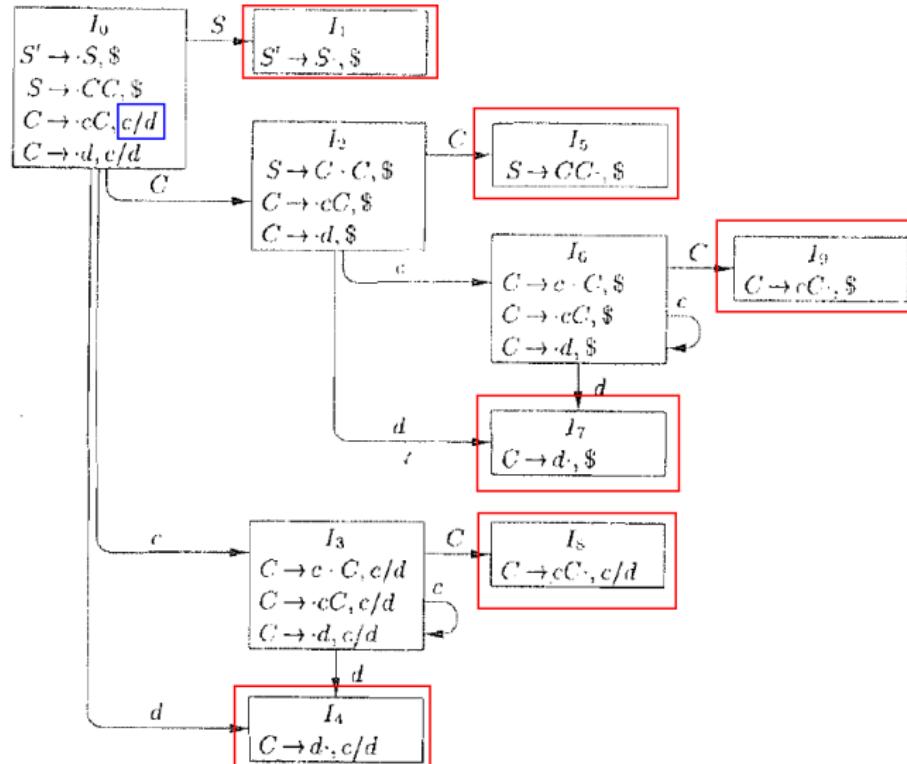
状态	ACTION			GOTO	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

$LALR(1)$: 合并具有相同核心 $LR(0)$ 项的状态 (忽略不同的向前看符号)。

$w = cc d c d \$$

$S' \rightarrow S$
 $S \rightarrow C \ C$
 $C \rightarrow c \ C \mid d$

$L(G) = c^* d c^* d$



Q: 合并 I_4 与 I_7 为 I_{47} ($\{[C \rightarrow d \cdot, c/d/\$]\}$), 会怎样?

Theorem

如果合并后的语法分析器**无冲突**, 则它的行为与原分析器**一致**。

- (1) **接受**原分析器所接受的句子, 且状态转移相同
- (2) **拒绝**原分析器所拒绝的句子, 但可能多一些不必要的**归约**动作

Theorem

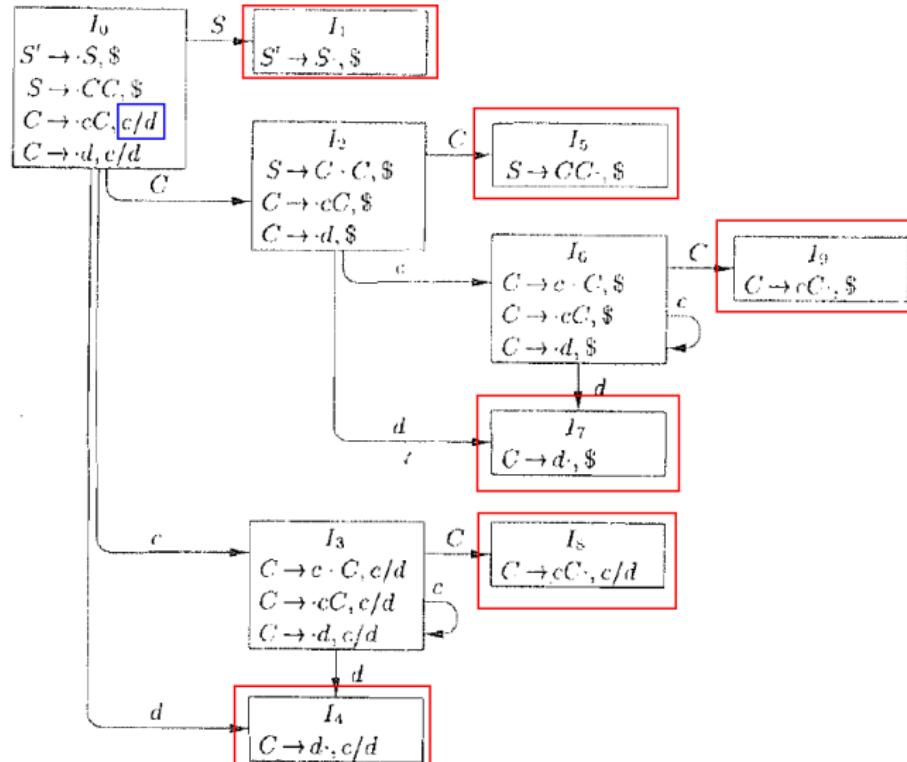
如果合并后的语法分析器**无冲突**, 则它的行为与原分析器**一致**。

- (1) **接受**原分析器所接受的句子, 且状态转移相同
- (2) **拒绝**原分析器所拒绝的句子, 但可能多一些不必要的**归约**动作
 (“实际上, 这个错误会在移入任何新的输入符号之前就被发现”)

$$w = ccd\$$$

$S' \rightarrow S$
 $S \rightarrow C \ C$
 $C \rightarrow c \ C \mid d$

$$L(G) = c^*dc^*d$$



继续合并 (I_8, I_9) 以及 (I_3, I_6)

状态	ACTION			GOTO	
	c	d	\$	S	C
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			

状态	ACTION			GOTO	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47		5	
36	s36	s47			
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

状态	ACTION			GOTO	
	c	d	\$	S	C
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			

状态	ACTION			GOTO	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47		5	
36	s36	s47			
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Q : GOTO 函数怎么办?

状态	ACTION			GOTO	
	c	d	\$	S	C
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			

状态	ACTION			GOTO	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47		5	
36	s36	s47			
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Q : GOTO 函数怎么办?

A : 可以合并的状态的 GOTO 目标 (状态) 一定也是可以合并的

Q : 对于 $LR(1)$ 文法, 合并得到的 $LALR(1)$ 分析表是否会引入冲突?

Q : 对于 $LR(1)$ 文法, 合并得到的 $LALR(1)$ 分析表是否会引入冲突?

Theorem

$LALR(1)$ 分析表**不会**引入移入/归约冲突。

Q : 对于 $LR(1)$ 文法, 合并得到的 $LALR(1)$ 分析表是否会引入冲突?

Theorem

$LALR(1)$ 分析表**不会**引入移入/归约冲突。

反证法

假设合并后出现 $[A \rightarrow \alpha\cdot, a]$ 与 $[B \rightarrow \beta \cdot a\gamma, b]$

则在 $LR(1)$ 自动机中,

存在某状态同时包含 $[A \rightarrow \alpha\cdot, a]$ 与 $[B \rightarrow \beta \cdot a\gamma, c]$

Q : 对于 $LR(1)$ 文法, 合并得到的 $LALR(1)$ 分析表是否会引入冲突?

Theorem

$LALR(1)$ 分析表**可能会**引入归约/归约冲突。

Q : 对于 $LR(1)$ 文法, 合并得到的 $LALR(1)$ 分析表是否会引入冲突?

Theorem

$LALR(1)$ 分析表**可能会**引入归约/归约冲突。

$$L(G) = \{acd, ace, bcd, bce\}$$

$$S' \rightarrow S$$

$$S \rightarrow a A d \mid b B d \mid a B e \mid b A e$$

$$A \rightarrow c$$

$$B \rightarrow c$$

Q : 对于 $LR(1)$ 文法, 合并得到的 $LALR(1)$ 分析表是否会引入冲突?

Theorem

$LALR(1)$ 分析表可能会引入归约/归约冲突。

$$L(G) = \{acd, ace, bcd, bce\}$$

$$S' \rightarrow S$$

$$S \rightarrow a A d \mid b B d \mid a B e \mid b A e$$

$$A \rightarrow c$$

$$B \rightarrow c$$

$$\{[A \rightarrow c \cdot, d], [B \rightarrow c \cdot, e]\}$$

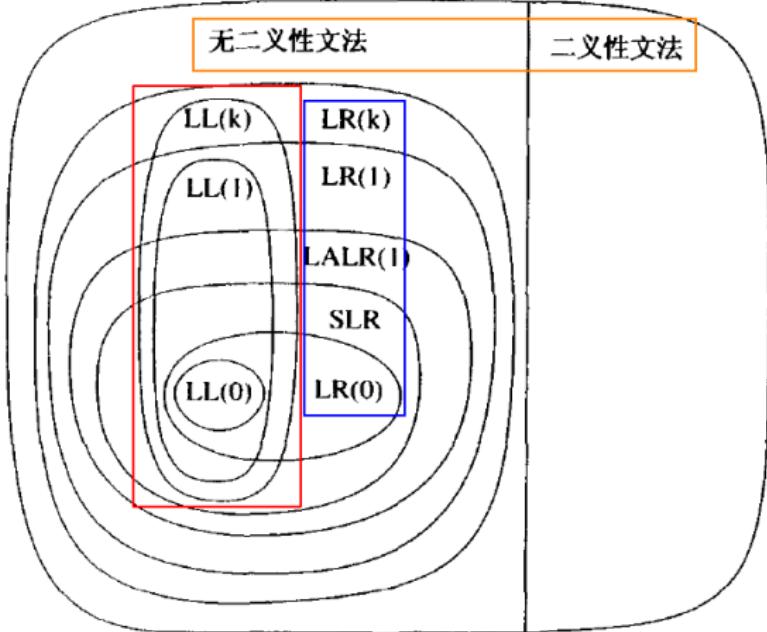
$$\{[A \rightarrow c \cdot, e], [B \rightarrow c \cdot, d]\}$$

$$\{[A \rightarrow c \cdot, \textcolor{red}{d/e}], [B \rightarrow c \cdot, \textcolor{red}{d/e}]\}$$

LALR(1) 语法分析器的优点

状态数量与 *SLR(1)* 语法分析器的状态数量相同

对于 *LR(1)* 文法, 不会产生移入/归约冲突



好消息：善用 *LR* 语法分析器，处理**二义性**文法



表达式文法

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$$
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \text{id}$$
$$E \rightarrow TE'$$
$$E' \rightarrow + TE' \mid \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow * FT' \mid \epsilon$$
$$F \rightarrow (E) \mid \text{id}$$

表达式文法: 使用 $SLR(1)$ 语法分析方法

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$$

$$\begin{array}{ll} I_0: & E' \rightarrow \cdot E \\ & E \rightarrow \cdot E + E \\ & E \rightarrow \cdot E * E \\ & E \rightarrow \cdot (E) \\ & E \rightarrow \cdot \text{id} \end{array}$$

$$\begin{array}{ll} I_1: & E' \rightarrow E \cdot \\ & E \rightarrow E \cdot + E \\ & E \rightarrow E \cdot * E \end{array}$$

$$\begin{array}{ll} I_2: & E \rightarrow (\cdot E) \\ & E \rightarrow \cdot E + E \\ & E \rightarrow \cdot E * E \\ & E \rightarrow \cdot (E) \\ & E \rightarrow \cdot \text{id} \end{array}$$

$$I_3: E \rightarrow \text{id} \cdot$$

$$\begin{array}{ll} I_4: & E \rightarrow E + \cdot E \\ & E \rightarrow \cdot E + E \\ & E \rightarrow \cdot E * E \\ & E \rightarrow \cdot (E) \\ & E \rightarrow \cdot \text{id} \end{array}$$

$$\begin{array}{ll} I_5: & E \rightarrow E * \cdot E \\ & E \rightarrow \cdot E + E \\ & E \rightarrow \cdot E * E \\ & E \rightarrow \cdot (E) \\ & E \rightarrow \cdot \text{id} \end{array}$$

$$\begin{array}{ll} I_6: & E \rightarrow (E \cdot) \\ & E \rightarrow E \cdot + E \\ & E \rightarrow E \cdot * E \end{array}$$

$$\begin{array}{ll} I_7: & E \rightarrow E + E \cdot \\ & E \rightarrow E \cdot + E \\ & E \rightarrow E \cdot * E \end{array}$$

$$\begin{array}{ll} I_8: & E \rightarrow E * E \cdot \\ & E \rightarrow E \cdot + E \\ & E \rightarrow E \cdot * E \end{array}$$

$$\{+, *\} \subseteq \text{FOLLOW}(E)$$

表达式文法: 使用 $SLR(1)$ 语法分析方法

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$$

$$\begin{array}{ll} I_0: & E' \rightarrow \cdot E \\ & E \rightarrow \cdot E + E \\ & E \rightarrow \cdot E * E \\ & E \rightarrow \cdot (E) \\ & E \rightarrow \cdot \text{id} \end{array}$$

$$\begin{array}{ll} I_5: & E \rightarrow E \cdot * E \\ & E \rightarrow \cdot E + E \\ & E \rightarrow \cdot E * E \\ & E \rightarrow \cdot (E) \\ & E \rightarrow \cdot \text{id} \end{array}$$

$$\begin{array}{ll} I_1: & E' \rightarrow E \cdot \\ & E \rightarrow E \cdot + E \\ & E \rightarrow E \cdot * E \end{array}$$

$$\begin{array}{ll} I_6: & E \rightarrow (E) \cdot \\ & E \rightarrow E \cdot + E \\ & E \rightarrow E \cdot * E \end{array}$$

$$\begin{array}{ll} I_2: & E \rightarrow \cdot (E) \\ & E \rightarrow \cdot E + E \\ & E \rightarrow \cdot E * E \\ & E \rightarrow \cdot (E) \\ & E \rightarrow \cdot \text{id} \end{array}$$

$$\begin{array}{ll} I_7: & E \rightarrow E + E \cdot \\ & E \rightarrow E \cdot + E \\ & E \rightarrow E \cdot * E \end{array}$$

$$I_3: E \rightarrow \text{id} \cdot$$

$$\begin{array}{ll} I_8: & E \rightarrow E * E \cdot \\ & E \rightarrow E \cdot + E \\ & E \rightarrow E \cdot * E \end{array}$$

$$\begin{array}{ll} I_4: & E \rightarrow E + \cdot E \\ & E \rightarrow \cdot E + E \\ & E \rightarrow \cdot E * E \\ & E \rightarrow \cdot (E) \\ & E \rightarrow \cdot \text{id} \end{array}$$

$$I_9: E \rightarrow (E) \cdot$$

$$\{+, *\} \subseteq \text{FOLLOW}(E)$$

考虑到结合性与优先级:

状态	ACTION						GOTO
	id	+	*	()	\$	
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

条件语句文法

$stmt \rightarrow if\ expr\ then\ stmt$

| $if\ expr\ then\ stmt\ else\ stmt$

| other

$S' \rightarrow S$

$S \rightarrow i\ S\ e\ S + i\ S + a$

条件语句文法: 使用 $SLR(1)$ 语法分析方法

$$S' \rightarrow S$$

$$S \rightarrow i \cdot S \ e \ S + i \ S + a$$

$I_0:$	$S' \rightarrow \cdot S$ $S \rightarrow \cdot iSeS$ $S \rightarrow \cdot iS$ $S \rightarrow \cdot a$	$I_3:$	$S \rightarrow e \cdot$ $I_4:$ $S \rightarrow iS \cdot eS$ $S \rightarrow iS \cdot$
$I_1:$	$S' \rightarrow S \cdot$	$I_5:$	$S \rightarrow iSe \cdot S$ $S \rightarrow \cdot iSeS$ $S \rightarrow \cdot iS$ $S \rightarrow \cdot a$
$I_2:$	$S \rightarrow i \cdot SeS$ $S \rightarrow i \cdot S$ $S \rightarrow \cdot iSeS$ $S \rightarrow \cdot iS$ \dots $S \rightarrow \cdot a$	$I_6:$	$S \rightarrow iSeS \cdot$

状态	ACTION				GOTO
	i	e	a	$\$$	
0	s2		s3		1
1				ace	
2	s2		s3		4
3			r3	r3	
4			s5	r2	
5	s2		s3		6
6		r1		r1	

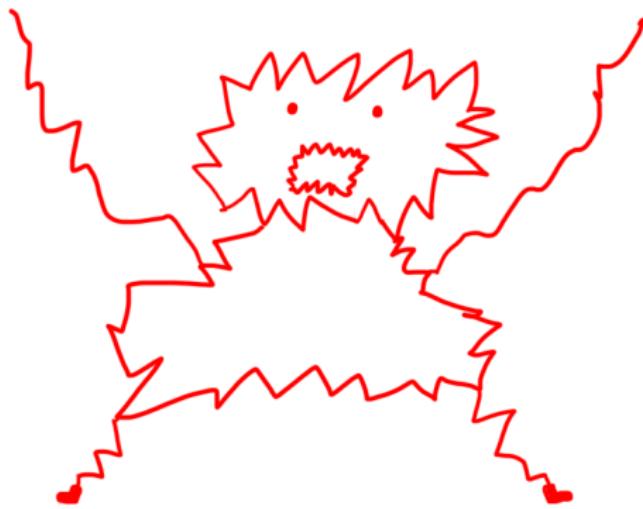
$e \in \text{FOLLOW}(S)$

$\text{ACTION}[4, e] = s5$

语法分析阶段的主题之三: 错误恢复



报错、**恢复**、继续分析



恐慌 (Panic) 模式: 丢弃输入、调整状态、假装成功

分号作为语句分隔符, 可用作**同步单词** (Synchronizing Word)

丢弃输入: 不断调用词法分析器, 直到找到下一个分号

调整状态: 不断出栈, 直到找到一个状态 s 满足

$$\text{GOTO}[s, Stmt] \neq \text{ERROR}$$

假装成功: 将状态 $\text{GOTO}[s, Stmt]$ 压栈, 恢复语法分析过程

分号作为语句分隔符, 可用作**同步单词** (Synchronizing Word)

丢弃输入: 不断调用词法分析器, 直到找到下一个分号

调整状态: 不断出栈, 直到找到一个状态 s 满足

$$\text{GOTO}[s, Stmt] \neq \text{ERROR}$$

假装成功: 将状态 $\text{GOTO}[s, Stmt]$ 压栈, 恢复语法分析过程

终结符 a 作为非终结符 A 的**同步单词** (如, $a \in \text{FOLLOW}(A)$)

分号作为语句分隔符, 可用作**同步单词** (Synchronizing Word)

丢弃输入: 不断调用词法分析器, 直到找到下一个分号

调整状态: 不断出栈, 直到找到一个状态 s 满足

$$\text{GOTO}[s, Stmt] \neq \text{ERROR}$$

假装成功: 将状态 $\text{GOTO}[s, Stmt]$ 压栈, 恢复语法分析过程

终结符 a 作为非终结符 A 的**同步单词** (如, $a \in \text{FOLLOW}(A)$)

可为**多个**非终结符 A 设置相应的同步单词 a



Bison: 语法分析器的生成器

多行表达式文法: 每行都以换行符结束; 允许有空行

```
20  lines : lines expr '\n'  
21  |      | lines '\n'  
22  |      | /* epsilon */  
23  |      | error '\n'  
24  
25  expr : expr '+' expr  
26  | expr '-' expr  
27  | expr '*' expr  
28  | expr '/' expr  
29  | '(' expr ')' '  
30  | '-' expr  
31  | NUMBER  
32  ;
```

file: expr.y

产生式 + 动作: LALR(1) 使用该产生式归约时执行动作

```
20 lines : lines expr '\n'          { printf("%d\n", $2); }
21     | lines '\n'                  { printf("0\n"); }
22     | /* epsilon */
23     | error '\n'                 { yyerror("Wrong expression");
24     |                         | yyerrok; }
25
26 expr : expr '+' expr           { $$ = $1 + $3; }
27     | expr '-' expr           { $$ = $1 - $3; }
28     | expr '*' expr           { $$ = $1 * $3; }
29     | expr '/' expr           { $$ = $1 / $3; }
30     | '(' expr ')'           { $$ = $2; }
31     | '-' expr                %prec UMINUS { $$ = - $2; }
32     | NUMBER                  /* default: { $$ = $1; } */
33 ;
```

类型: 为每个文法符号的**属性值**指定合适的类型

```
5 %union {  
6     int i; /* type for results of expressions */  
7     /* to add type (e.g., struct) for parse tree */  
8 }  
9  
10 %token <i> NUMBER  
11 %type <i> expr /* to use type for parse tree */
```

```
bison -d -v expr.y
```

-d: (definition) 产生 `expr.tab.h` 与 `expr.tab.c`

-v: “warning: 20 **shift/reduce** conflicts” (`expr.output`)

使用结合性/优先级解决移入/归约冲突

```
13 %left '+' '-'
14 %left '*' '/'
15 /* placeholder; for unary minus operator */
16 %right UMINUS
```

原则上，可以为每个**终结符**以及每条**产生式**赋予适当的结合性与优先级

```
26 expr : expr '+' expr          { $$ = $1 + $3; }
27     | expr '-' expr          { $$ = $1 - $3; }
28     | expr '*' expr         { $$ = $1 * $3; }
29     | expr '/' expr         { $$ = $1 / $3; }
30     | '(' expr ')'          { $$ = $2; }
31     | '-' expr               %prec UMINUS { $$ = - $2; }
32     | NUMBER                /* default: { $$ = $1; } */
33 ;
```

使用结合性/优先级解决移入/归约冲突

```
13 %left '+' '-'
14 %left '*' '/'
15 /* placeholder; for unary minus operator */
16 %right UMINUS
```

原则上，可以为每个**终结符**以及每条**产生式**赋予适当的结合性与优先级

```
26 expr : expr '+' expr          { $$ = $1 + $3; }
27     | expr '-' expr          { $$ = $1 - $3; }
28     | expr '*' expr         { $$ = $1 * $3; }
29     | expr '/' expr         { $$ = $1 / $3; }
30     | '(' expr ')'          { $$ = $2; }
31     | '-' expr               %prec UMINUS { $$ = - $2; }
32     | NUMBER                /* default: { $$ = $1; } */
33 ;
```

产生式的结合性与优先级被设定为它的**最右终结符号**的结合性与优先级

使用结合性/优先级解决移入/归约冲突

冲突: “移入 a ” vs. “按 $A \rightarrow \alpha$ 归约”

产生式的优先级高于 a 的优先级

或者优先级相同但产生式是左结合的, 则选择归约

```
1  %{
2      /* generated by `bison -d expr.y` */
3      #include "expr.tab.h"
4  %}
5
6  %%
7  "+" |
8  "-" |
9  "*" |
10 "/" |
11 "(" | /* literal character tokens for the operators */
12 ")" { return yytext[0]; }
13 '\n' { return yytext[0]; }
14 [1-9][0-9]* { yylval.i = atoi(yytext);
15 | | | | return NUMBER; } /* return tokens */
16 [ \t] { } /* ignore whitespaces; DO NOT return them as tokens */
17 . { printf("Mystery character %c\n", *yytext); }
18 %%
```

file: expr.l

```
3 int main(int argc, char** argv) {
4     if (argc <= 1)
5         return 1;
6
7     FILE* f = fopen(argv[1], "r");
8     if (!f) {
9         perror(argv[1]);
10        return 1;
11    }
12
13    yyrestart(f);
14    /* it will call yylex() on demand */
15    yyparse();
16
17    return 0;
18 }
19
20 int yyerror(char* s) {
21     printf("error: %s\n", s);
22 }
```

file: main.c

```
bison -d -v expr.y
```

```
flex expr.l
```

```
gcc main.c lex.yy.c expr.tab.c -lfl -o expr
```

```
./expr expr.cmm
```

使用 错误产生式 (Error Production) 进行错误恢复

$$A \rightarrow \text{error } \alpha \quad \alpha \in N^*$$

```
21 lines : lines expr '\n'          { printf("%d\n", $2); }
22   | lines '\n'                  { printf("0\n"); }
23   | /* epsilon */
24   | error '\n'                { yyerror("Wrong expression");
25                                yyerrok; }
```

使用 错误产生式 (Error Production) 进行错误恢复

$$A \rightarrow \text{error } \alpha \quad \alpha \in N^*$$

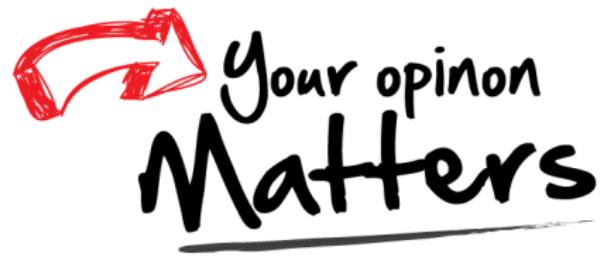
```
21 lines : lines expr '\n'          { printf("%d\n", $2); }
22   | lines '\n'                  { printf("0\n"); }
23   | /* epsilon */
24   | error '\n'                { yyerror("Wrong expression");
25                                yyerrok; }
```

调整状态 不断出栈，直到碰到某状态包含形如 $A \rightarrow \cdot \text{error } \alpha$ 的项；
移入 **error**

丢弃输入 不断调用词法分析器，寻找终结符号串 α 并移入

假装成功 此时栈顶为 **error** α ，归约为 A ；恢复正常

Thank You!



Office 926

hfwei@nju.edu.cn