

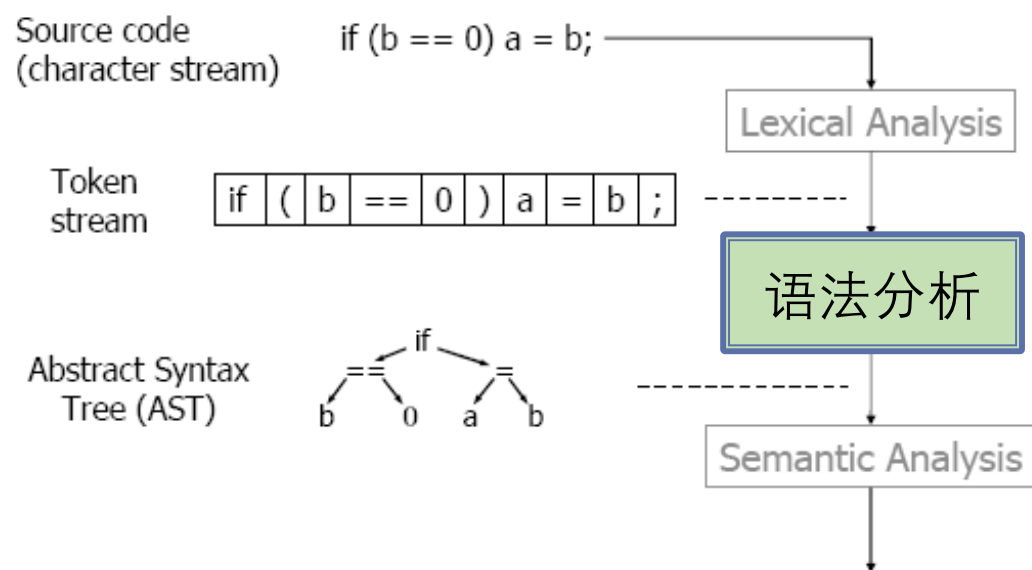
编译原理与技术

--自底向上的语法分析

刘爽

中国人民大学信息学院

■ 语法分析器所处的位置



■ Outline

- 自底向上的语法分析基本问题
 - 移动 – 归约分析法
 - 用栈实现移动归约分析
- 算符优先分析法
 - 算符优先分析法定义、优先分析表的确定
 - 使用算符优先关系进行分析
- LR分析法
 - LR(0), SLR, LR(1)
- 语法分析器的自动产生工具Yacc

■ 自底向上语法分析

- 自底向上的语法分析方法是一种“**移进-归约**”分析法。
 - 最易于实现的一种移进-归约分析方法，叫做**算符优先分析法**，
 - 而更一般的移进-归约分析方法叫做LR分析法，LR分析法可以用作许多自动的语法分析器的生成器。
- 移进-归约分析法为输入串构造分析树时是从叶结点（底端）开始，向根结点（顶端）前进。
 - 我们可以把该过程看成是把输入串 w “**归约**”成文法开始符号的过程。
 - 在每一步归约中，如果一个子串和某个产生式的右部匹配，则用该**产生式的左部符号代替该子串**。
 - 如果每一步都能恰当的选择子串，我们就可以得到**最右推导的逆过程**。

■ 移进-归约分析例子

假定文法为

(1) $S \rightarrow aAcBe$

(2) $A \rightarrow b$

(3) $A \rightarrow Ab$

(4) $B \rightarrow d$

将输入串**abbcd**e归约到**S**.

步骤:

动作:

栈:

1	2	3	4	5	6	7	8	9	10
进	进	归	进	归	进	进	归	进	归
a	b	(2)	b	(3)	c	d	(4)	e	(1)
								e	
			b		c	d	B	B	
	b	A	A	A	A	c	c	c	
a	a	a	a	a	a	a	a	a	S

■ 关键问题

- 如何决定何时进行规约？即用哪一个产生式进行规约？

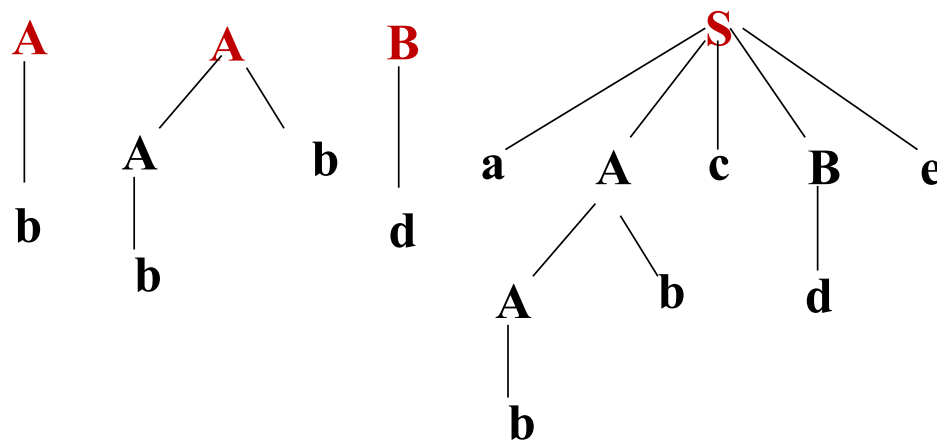


精确定义“可规约串”

不同的可归约串形成了不同的自下而上分析法

- 算符优先分析法中用“最左素短语”；
- 规范归约分析法中用“句柄”；

语法分析过程中, 可用一棵分析树来表示。在自下而上分析过程中, 每一步归约都可以画出一棵子树来归约完成, 形成一棵分析树。



■ 短语、直接短语、句柄的定义：

- 文法 $G[S]$, $\alpha\beta\delta$ 是文法 G 的一个句型, $S \xRightarrow{*} \alpha A \delta$ 且 $A \xRightarrow{+} \beta$
则称 β 是句型 $\alpha\beta\delta$ 相对于非终结符 A 的短语。
若有 $A \Rightarrow \beta$ 则称 β 是句型 $\alpha\beta\delta$ 相对于该规则 $A \rightarrow \beta$ 的直接短语。
- 一个句型的最左直接短语称为该句型的句柄。

!!注意两个条件
缺一不可

例如
文法:

$$\begin{aligned} E &\longrightarrow T \mid E+T \\ T &\longrightarrow F \mid T * F \\ F &\longrightarrow (E) \mid a \end{aligned}$$

$a_1 * a_2 + a_3$ 是文法的句子,
找出此句型的短语, 直接短语
和句柄。

2025/5/28

■ 例子

- 考虑右边所示文法的一个句型 $a1*a2+a3$:
 - $a1, a2, a3, a1*a2, a1*a2+a3$ 都是句型 $a1*a2+a3$ 的短语,
 - $a1, a2$ 和 $a3$ 是直接短语,
 - $a1$ 是最左直接短语,
 - $a2+a3$ 不是句型 $a1*a2+a3$ 短语, 因为 $E \Rightarrow a2+a3$ 但不存在从文法的开始符号 E 到 $a1*E$ 的推导.

$$\begin{aligned} E &\longrightarrow T|E+T \\ T &\longrightarrow F|T*F \\ F &\longrightarrow (E)|a \end{aligned}$$

- 文法 $G[S]$, $\alpha\beta\delta$ 是文法 G 的一个 **句型**, $S \xRightarrow{*} \alpha A \delta$ 且 $A \xRightarrow{+} \beta$ 则称 β 是句型 $\alpha\beta\delta$ 相对于非终结符 A 的 **短语**。
若有 $A \Rightarrow \beta$ 则称 β 是句型 $\alpha\beta\delta$ 相对于该规则 $A \rightarrow \beta$ 的 **直接短语**。
- 一个句型的 **最左直接短语** 称为该句型的 **句柄**。

■ 用子树解释短语，直接短语，句柄:

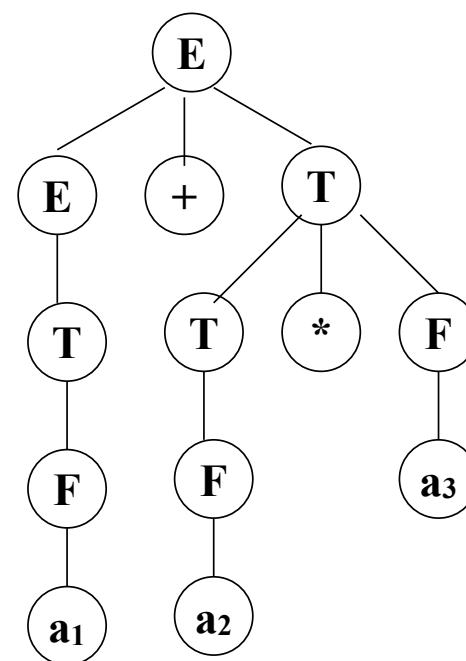
- **短语**: 一棵子树的所有**叶子**自左至右排列起来形成一个相对于**子树根**的短语。
- **直接短语**: 仅有**父子两代**的一棵子树，它的所有叶子自左至右排列起来所形成的符号串。
- **句柄**: 一个句型的分析树中**最左**那棵只有父子两代的子树的所有叶子的自左至右排列。

■ 例子

对表达式文法 $G[E]$ 和句子 $a_1+a_2*a_3$ ，挑选出推导过程中产生的句型中的短语，直接短语，句柄。

E	短语
$\Rightarrow E+T$	<u>$E+T$</u>
$\Rightarrow E+T^*F$	<u>T^*F</u> , $E+T^*F$
$\Rightarrow E+T^*a_3$	<u>a_3</u> , T^*a_3 , $E+T^*a_3$
$\Rightarrow E+F^*a_3$	a_3 , <u>F</u> , F^*a_3 , $E+F^*a_3$
$\Rightarrow E+a_2^*a_3$	a_3 , <u>a_2</u> , $a_2^*a_3$, $E+a_2^*a_3$
$\Rightarrow T+a_2^*a_3$	a_3 , a_2 , <u>T</u> , $a_2^*a_3$, $T+a_2^*a_3$
$\Rightarrow F+a_2^*a_3$	a_3 , a_2 , <u>F</u> , $a_2^*a_3$, $F+a_2^*a_3$
$\Rightarrow a_1+a_2^*a_3$	<u>a_1</u> , a_2 , a_3 , $a_2^*a_3$, $a_1+a_2^*a_3$

$$\begin{aligned} E &\longrightarrow T|E+T \\ T &\longrightarrow F|T^*F \\ F &\longrightarrow (E)|a \end{aligned}$$



■ 规范归约定义

假定 a 是文法 G 的一个句子, 称序列 a_n, a_{n-1}, \dots, a_0 是 a 的一个**规范归约**, 此序列应满足:

- 1) $a_n = a$
- 2) a_0 为文法的开始符, 即 $a_0 = S$
- 3) 对任何 i , $0 < i \leq n$, a_{i-1} 是从 a_i 经把**句柄**替换为相应产生式的左部符号而得到的

规范归约是关于 a 的一个**最右推导**的逆过程。规范归约也称**最左归约**。

在形式语言中, 最右推导常被称为**规范推导**, 由规范推导所得的句型称为**规范句型**。**规范归约的实质**是: 在移进过程中, 当发现栈顶呈现**句柄**时, 就用相应产生式的左部符号进行替换。

■ 回顾例子

对该文法的句子**abbcede**逐步寻找句柄,并用相应产生式的左部符号去替换,得到如下归约过程:(画底线的部分是句柄).

考虑文法:

(1) $S \rightarrow aAcBe$

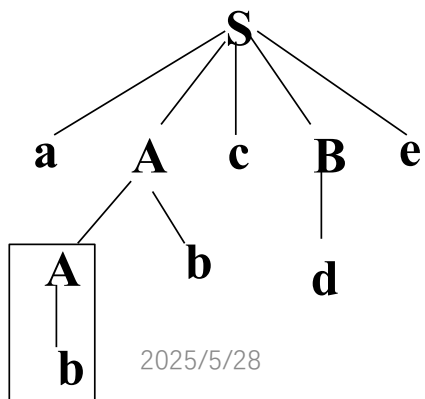
(2) $A \rightarrow b$

(3) $A \rightarrow Ab$

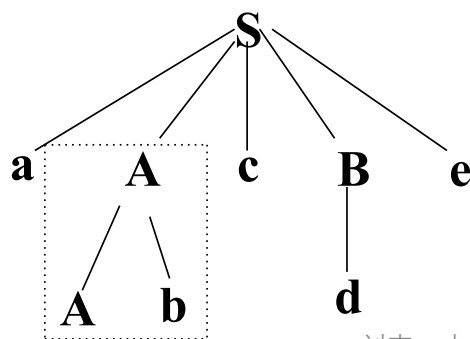
(4) $B \rightarrow d$

将输入串**abbcede**归约到**S**.

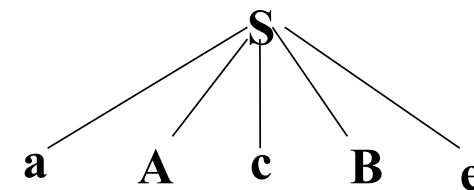
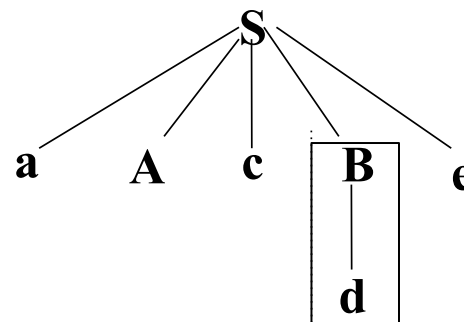
句型	归约过程
<u>ab</u> bcde	(2) A b
a <u>A</u> bcde	(3) A Ab
aAc <u>d</u> e	(4) B d
<u>aAcBe</u>	(1) S aAcBe
S	



2025/5/28



刘爽, 中国人民大学信息学院



12

■ “移进-归约”分析法的栈实现

- 移进-归约分析器使用**符号栈**和**输入缓冲区**。用“#”作为栈底。分析器的工作过程中，对符号栈的使用有四类操作：“移进”，“归约”，“接受”，和“出错处理”。
- 输入串 $i_1*i_2+i_3$ 的分析(规范归约)步骤如下：

$E \longrightarrow E+E$

$E \longrightarrow E * E$

$E \longrightarrow (E) | i$

输入串 $i_1*i_2+i_3$ 的分析(规范归约)步骤如下:

步骤	符号栈	输入串	动作
0	#	$i_1*i_2+i_3\#$	进
1	# i_1	$*i_2+i_3\#$	归 $E \rightarrow i$
2	#E	$*i_2+i_3\#$	进
3	#E*	$i_2+i_3\#$	进
4	#E* i_2	$+i_3\#$	归 $E \rightarrow i$
5	#E*E	$+i_3\#$	归, $E \rightarrow E*E$
6	#E	$+i_3\#$	进
7	#E+	$i_3\#$	进
8	#E+ i_3	#	归, $E \rightarrow i$
9	#E+E	#	归, $E \rightarrow E+E$
10	#E	#	接受

注意:任何可归约串的出现必须在栈顶.

$E \longrightarrow E+E$
 $E \longrightarrow E*E$
 $E \longrightarrow (E)|i$

■ 练习

规范规约:

考虑文法G[S]:

(1) $S \rightarrow aABe$

(2) $A \rightarrow b$

(3) $A \rightarrow Abc$

(4) $B \rightarrow d$

最右推导:

$S \Rightarrow aABe$

$\Rightarrow aAde$

$\Rightarrow aAbcde$

$\Rightarrow abbcde$

步骤

符号栈

输入符号串

动作

■ 练习

规范规约：

考虑文法G[S]：

(1) $S \rightarrow aABe$

(2) $A \rightarrow b$

(3) $A \rightarrow Abc$

(4) $B \rightarrow d$

最右推导：

$S \Rightarrow aABe$

$\Rightarrow aAde$

$\Rightarrow aAbcde$

$\Rightarrow abbcde$

步骤	符号栈	输入符号串	动作
1	#	abbcde#	移进
2	#a	bbcde#	移进
3	#ab	bcde#	归约(2)
4	#aA	bcde#	移进
5	#aAb	cde#	移进
6	#aAbc	de#	归约 (3)
7	#aA	de#	移进
8	#aAd	e#	归约(4)
9	#aAB	e#	移进
10	#aABe	#	归约(1)
11	#S	#	接受

■ 移动归约分析中需要解决的问题

- 定位右句型中将要归约的子串（**可归约串**）
 - 在用堆栈实现时，这个子串总是在栈顶。语法分析器不需要深入到栈中去找句柄
- **选择产生式**对选定的串进行归约
 - 如果选定的子串是多个产生式的右部，要确定选择哪个产生式进行归约
- 移动归约分析过程中的冲突
 - 根据栈中的内容和下一个输入符号不能决定是移动还是归约（**移动-归约冲突**）
 - 不能决定按哪一个产生式进行归约（**归约-归约冲突**）

■ Outline

- 自底向上的语法分析基本问题
 - 移动 – 归约分析法
 - 用栈实现移动归约分析
- 算符优先分析法
 - 算符优先分析法定义、优先分析表的确定
 - 使用算符优先关系进行分析
 - 算符优先分析中的错误恢复
- LR分析法
- 语法分析器的自动产生工具Yacc

■ 算符优先分析法

- 算符优先分析法的关键是定义两个可能相继出现的终结符之间的优先关系，并根据这种优先关系寻找“可规约串”并进行规约（非规范规约）。
- 两个终结符之间的优先关系：
 - $a < b$: a 的优先级低于 b 的优先级
 - $a \equiv b$: a 的优先级等于 b 的优先级
 - $a > b$: a 的优先级高于 b 的优先级

注意：算符的优先关系是**有序**的

- 如果 $a > b$ ，不能推出 $b < a$
- 如果 $a = b$ ，不能推出 $b = a$
- 如果 $a > b$ ，有可能 $b > a$
- 如果 $a > b$ ， $b > c$ ，不一定 $a > c$

■ 优先关系矩阵例子

- $E \rightarrow E + E | E * E | E \wedge E | (E) | i$ 的终结符之间的优先关系可用矩阵表示:

	+	*	^	i	()	#
+	'>	<'	<'	<'	<'	'>	'>
*	'>	'>	<'	<'	<'	'>	'>
^	'>	'>	<'	<'	<'	'>	'>
i	'>	'>	'>			'>	'>
(<'	<'	<'	<'	<'	='	
)	'>	'>	'>			'>	'>
#	<'	<'	<'	<'	<'		='

从表中可以看出:

- 1) 优先级 ^ * + 由高到底
- 2) * + 左结合, ^ 右结合.
- 3) 运算对象 i 要先处理.
- 4) 先括号内后括号外.
- 5) 对于 #, 任何终结符 Q 都高于 #.

■ 算符优先文法定义

- 算符文法的定义：

- 设有一个文法G，如果G中**没有**形如 $A \rightarrow \dots BC \dots$ 的产生式，其中B和C为**非终结符**，则称G为**算符文法**(Operator Grammar)也称OG文法。

- **算符优先文法**的定义：设文法G是不含任何 ε -产生式的**算符文法**，对任何一对终结符a, b, 我们说：

- $a = b$ ：文法中有形如 $A \rightarrow \dots ab \dots$ 或者 $A \rightarrow \dots aBb \dots$ 的产生式
- $a < b$ ：文法中有形如 $A \rightarrow \dots aB \dots$ 的产生式而 $B \xRightarrow{+} b \dots$ 或 $B \xRightarrow{+} Cb \dots$
- $a > b$ ：文法中有形如 $A \rightarrow \dots Bb \dots$ 的产生式，而 $B \xRightarrow{+} \dots a$ 或 $B \xRightarrow{+} \dots aC$

如果G中的任何终结符对(a, b)**至多**只满足 $a = b$, $a < b$, $a > b$ 三种关系之一，则称G是一个**算符优先文法**

■ 算符优先文法例子

- 考虑下述文法,
其是否为算符优先文法?

(1) $E \longrightarrow T \mid E+T$

(2) $T \longrightarrow F \mid T * F$

(3) $F \longrightarrow P \mid P \wedge F$

(4) $P \longrightarrow (E) \mid i$

- $a = ' b$: 文法中有形如 $A \rightarrow \cdots ab \cdots$ 或者 $A \rightarrow \cdots aBb \cdots$ 的产生式
- $a < ' b$: 文法中有形如 $A \rightarrow \cdots aB \cdots$ 的产生式而 $B \Rightarrow b \cdots$ 或 $B \Rightarrow Cb \cdots$
- $a ' > b$: 文法中有形如 $A \rightarrow \cdots Bb \cdots$ 的产生式, 而 $B \Rightarrow \cdots a$ 或 $B \Rightarrow \cdots aC$

由(4) 可得: $(= ')$

由 $E \rightarrow E+T$ 及 $T \overset{+}{\Rightarrow} T * F$ 可得: $+ < ' *$

由 $T \rightarrow F \mid T * F$ 及 $F \Rightarrow P \mid P \wedge F$ 可得: $* < ' \wedge$

由 $E \rightarrow E+T$ 及 $E \Rightarrow E+T$ 可得: $+ > ' +$

由 $F \rightarrow P \wedge F$ 及 $F \Rightarrow P \wedge F$ 可得: $\wedge < ' \wedge$

由 $P \rightarrow (E)$ 以 $E \Rightarrow E+T \Rightarrow T+T \Rightarrow T * F+T$

$\Rightarrow F * F+T \Rightarrow P \wedge F * F+T \Rightarrow i \wedge F * F+T$

可得: $(< ' +, (< ' *, (< ' \wedge, (< ' i$.

同理可得: $+ > '), * > '), \wedge > '), i > ')$.

所以这个文法是算符优先文法.

■ 算符优先文法例子

(1) $E \longrightarrow T | E + T$

(2) $T \longrightarrow F | T * F$

(3) $F \longrightarrow P | P \wedge F$

(4) $P \longrightarrow (E) | i$

	+	*	\wedge	i	()
+	'>	<'	<'	<'	<'	'>
*	'>	'>	<'	<'	<'	'>
\wedge	'>	'>	<'	<'	<'	'>
i	'>	'>	'>			'>
(<'	<'	<'	<'	<'	='
)	'>	'>	'>			'>

■ 算符优先分析法

- 算符优先分析法的关键是定义两个可能相继出现的终结符之间的优先关系，并根据这种优先关系寻找“可规约串”并进行规约（非规范规约）。
- 两个终结符之间的优先关系：
 - $a < b$: a 的优先级低于 b 的优先级
 - $a \equiv b$: a 的优先级等于 b 的优先级
 - $a > b$: a 的优先级高于 b 的优先级

注意：算符的优先关系是**有序**的

- 如果 $a > b$ ，不能推出 $b < a$
- 如果 $a \equiv b$ ，不能推出 $b \equiv a$
- 如果 $a > b$ ，有可能 $b > a$
- 如果 $a > b$ ， $b > c$ ，不一定 $a > c$

■ 算符优先文法定义

- 算符文法的定义：

- 设有一个文法G，如果G中**没有**形如 $A \rightarrow \dots BC \dots$ 的产生式，其中B和C为**非终结符**，则称G为**算符文法**(Operator Grammar)也称OG文法。

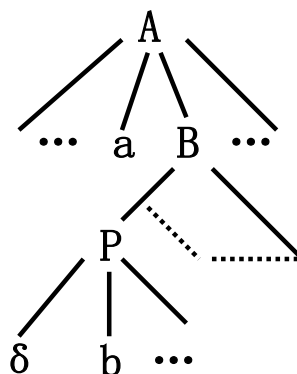
- **算符优先文法**的定义：设文法G是不含任何 ε -产生式的**算符文法**，对任何一对终结符a, b, 我们说：

- $a = b$ ：文法中有形如 $A \rightarrow \dots ab \dots$ 或者 $A \rightarrow \dots aBb \dots$ 的产生式
- $a < b$ ：文法中有形如 $A \rightarrow \dots aB \dots$ 的产生式而 $B \xRightarrow{+} b \dots$ 或 $B \xRightarrow{+} Cb \dots$
- $a > b$ ：文法中有形如 $A \rightarrow \dots Bb \dots$ 的产生式，而 $B \xRightarrow{+} \dots a$ 或 $B \xRightarrow{+} \dots aC$

如果G中的任何终结符对(a, b)**至多**只满足 $a = b$, $a < b$, $a > b$ 三种关系之一，则称G是一个**算符优先文法**

■ 算符优先关系解释I

- $a < ' b$, 则存在语法子树如下

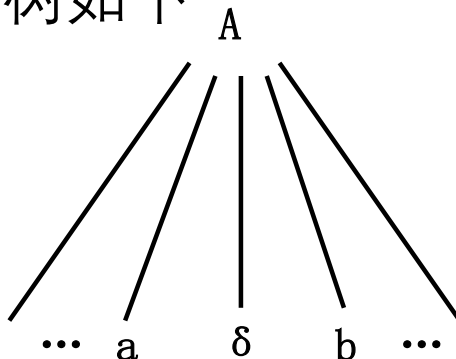


其中, δ 为 ϵ 或者 C ,
 a 和 b 不在同一个句柄中, b 先被归约

- $a < ' b$: 文法中有形如 $A \rightarrow \dots a B \dots$ 的产生式而 $B \Rightarrow b \dots$ 或 $B \Rightarrow C b \dots$

■ 算符优先关系解释II

- $a = ' b$, 则存在语法子树如下

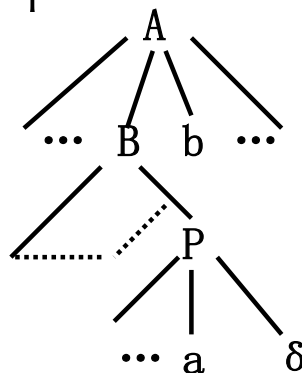


其中, δ 为 ϵ 或者 B ,
 a 和 b 在同一个句柄中, 同时被归约,

- $a = ' b$: 文法中有形如 $A \rightarrow \dots ab \dots$ 或者 $A \rightarrow \dots aBb \dots$ 的产生式

■ 算符优先关系解释III

- $a' > b$, 则存在语法子树如下



其中, δ 为 ϵ 或者 C ,
 a 和 b 不在同一个句柄中, a 先被归约

- $a' > b$: 文法中有形如 $A \rightarrow \cdots Bb \cdots$ 的产生式, 而 $B \Rightarrow \cdots a$ 或 $B \Rightarrow \cdots aC$

■ FIRSTVT & LASTVT

- 构造文法的每个非终结符的首(尾)符号集合:
 - $\text{FIRSTVT}(P) = \{a \mid P \xRightarrow{\pm} a \cdots \text{或} P \xRightarrow{\pm} Qa \cdots, a \in V_T, P, Q \in V_N\}$
 - $\text{LASTVT}(P) = \{a \mid P \xRightarrow{\pm} \cdots a \text{或} P \xRightarrow{\pm} \cdots aQ, a \in V_T, P, Q \in V_N\}$
- 有了这两个集合就可以定义: ' $>$ ' < ' $<$ '而 '=' 可从文法直接找到.
- 如果形如 $A \rightarrow \cdots aB \cdots$ 的产生式, 且任何 $b \in \text{FIRSTVT}(B)$, 满足 $a < ' b$
如果形如 $A \rightarrow \cdots Bb \cdots$ 的产生式, 且任何 $a \in \text{LASTVT}(B)$, 满足 $a ' > b$

■ 构造集合FIRSTVT(P)的算法I:

用下面两条规则构造集合**FIRSTVT(P)** :

- (a) 若有产生式 $P \rightarrow a \dots$ 或 $P \rightarrow Qa \dots$, 则 $a \in \text{FIRSTVT}(P)$.
- (b) 若 $a \in \text{FIRSTVT}(Q)$, 且有产生式 $P \rightarrow Q \dots$, 则 $a \in \text{FIRSTVT}(P)$.

算法思想: 建立一个布尔数组 $F[X, a]$, 使得 $F[X, a]$ 为真的条件是, 当且仅当 $a \in \text{FIRSTVT}(X)$ 。

按规则(a)对每个数组元素 $F[X, a]$ 赋初值。用一个stack把所有初值为真的数组元素 $F[X, a]$ 的符号对 (X, a) 全部放到stack中, 然后对stack进行如下运算:

- 如果stack不空, 则将栈顶逐出, 记此项为 (Q, a) 对于每个形如 $P \rightarrow Q \dots$ 的产生式, 若 $F[P, a]$ 为假, 则变其值为真且将 (P, a) 推进stack.
- 重复上述过程, 直到stack为空。

■ 构造集合FIRSTVT(P)的算法II:

```
1  begin
2      for 每个非终结符X和终结符a do
3          F[X, a]:=FALSE;
4      for 每个形如P→a...或P→Qa...的产生式 do
5          insert(P, a);
6      while 栈非空 do
7          begin
8              把stack的栈顶项,记为 (Q, a), pop出去;
9              for每条形如P→Q...的产生式 do
10                 insert(P, a);
11          end of while;
12 end
```

下面是求F[P, a]的值的算法:

```
1  Proc insert(P, a);
2      if not F[P, a] then
3          begin
4              F[P, a]:=TRUE;
5              把(P,a)推进栈
6          end;
```

FIRSTVT(P)={a | F[P,a] = TRUE }

■ 构造优先表的算法

```
1 for 每条产生式  $P \rightarrow X_1 X_2 \dots X_n$  do
2   for  $i := 1$  to  $n-1$  do
3     begin
4       if  $X_i$ 和 $X_{i+1}$ 均为终结符 then 置  $X_i = X_{i+1}$ ;
5       if  $i \leq n-2$  且  $X_i$ 和 $X_{i+2}$ 都为终结符但 $X_{i+1}$ 为非终结符
6         then 置  $X_i = X_{i+2}$ ;
7       if  $X_i$  为终结符而 $X_{i+1}$  为非终结符 then
8         for FIRSTVT( $X_{i+1}$ )中的每个 $a$  do
9           置  $X_i < a$ 
10      If  $X_i$  为非终结符而 $X_{i+1}$  为终结符 then
11        for LASTVT( $X_i$ )中的每个 $a$  do
12          置  $a > X_{i+1}$ 
13    end
```


■ 算符优先分析—基本定义

- **素短语**:是指这样的一个**短语**,它至少含有一个**终结符**,并且除它自身之外不再含任何更小的素短语。
- **最左素短语**:处于句型最左边的那个**素短语**。
- 算符优先分析法的“可归约串”是**最左素短语**。

例如:考虑下述文法的句型 $P*P+i$ 的素短语.

$$(1) \quad E \longrightarrow T | E+T$$

$$(2) \quad T \longrightarrow F | T * F$$

$$(3) \quad F \longrightarrow P | P \wedge F$$

$$(4) \quad P \longrightarrow (E) | i$$

短语是 $P*P+i$, $P*P$, P 和 i

素短语是 $P*P$ 和 i


最左素短语是 $P*P$

■ 算符优先文法

- 一个算符优先文法的句型（包含在两个#之间）具有以下形式：

$\#N_1a_1N_2a_2 \dots N_na_n N_{n+1}\#$ (*)

- 其中 a_i 是终结符， N_i 是可有可无的非终结符



任何算符文法的句型都具有该形式

- 一个算符优先文法G的任何句型 (*) 的最左素短语是满足如下条件的最左子串 $N_ja_j \dots N_ia_iN_{i+1}$,
 - $a_{j-1} < 'a_j$
 - $a_j = 'a_{j+1}, \dots, a_{i-1} = 'a_i$
 - $a_i ' > a_{i+1}$

■ 使用算符优先关系

- 算符优先关系主要用于界定句型的最左素短语
 - <'标记最左素短语的左端； '='出现在最左素短语的内部； '>'标记最左素短语的右端。
 - 发现最左素短语的过程：
 - 从左端开始扫描串，直到遇到第一个'>'为止。
 - 向左扫描栈，跳过所有的='，直到遇到一个<'为止。
 - 最左素短语包括从上一步遇到的<'右部到第一个'>'左部之间的所有符号，包括介于中间或者两边的非终结符
- 非终结符的处理
 - 因为非终结符不能影响语法分析，所以不需要区分它们，于是只用一个占位符来代替它们

■ 算符优先分析算法

- 算法的主体思想

- 用栈存储已经看到的输入符号，用**优先关系**指导移动-归约语法分析器的动作
 - 如果栈顶的终结符和下一个输入符之间的优先关系是 $<$ 或 $=$ （即**当前输入符号**为最左素短语的左，中端），则进行**移动**；
 - 如果是 $>$ 关系（即**当前输入符号**为**最左素短语**的最右端），则进行**归约**

- 算法描述

- 输入：输入字符串 w 和优先关系表
- 输出：如果 w 是语法产生的一个句子，则输出其用来归约的产生式；如果有错误，则转入错误处理

■ 算符优先分析算法

- 最外层循环(2-17): 读入输入串w
- 内层While循环(5-13): 寻找最左素短语(6-9)进行规约, 直到在当前栈内无法找到新的最左素短语为止

注意: 第10行中, 规约为某个N, 这个N指的是这样一个产生式的左端: 此产生式的右部和 $S[j+1] \cdots S[k]$ 构成如下一一对应关系: 自左至右, 终结符对终结符, 非终结符对非终结符, 而且对应的终结符相同。

```
1.  K:=1; S[k]:='#';
2.  Repeat
3.      把下一输入符号读入a;
4.      If S[k] ∈ VT Then j:=k ELSE j:=k-1;
5.      While S[j] '>' a Do
6.          Repeat
7.              Q:=S[j];
8.              If S[j-1] ∈ VT Then j:=j-1 Else j:=j-2;
9.              Until S[j] < ' Q;
10.         把 S[j+1]...S[k]归约为某个N;
11.         k:=j+1;
12.         S[k]=N;
13.     EndOfWhile
14.     If S[j] < ' a Or S[j]=' a Then
15.         begin k:=k+1; S[k]:=a; end
16.     Else error
17. Until a='#'
```

■ 规范归约和算符优先归约

- 句型 $i+i\#$ 的规范归约过程

步骤	符号栈	剩余输入串	句柄	归约用产生式
1	#	$i+i\#$		
2	$\#i$	$+i\#$	i	$P \rightarrow i$
3	$\#P$	$+i\#$	P	$F \rightarrow P$
4	$\#F$	$+i\#$	F	$T \rightarrow F$
5	$\#T$	$+i\#$	T	$E \rightarrow T$
6	$\#E$	$+i\#$		
7	$\#E+$	$i\#$		
8	$\#E+i$	$\#$	i	$P \rightarrow i$
9	$\#E+P$	$\#$	P	$F \rightarrow P$
10	$\#E+F$	$\#$	F	$T \rightarrow F$
11	$\#E+T$	$\#$	$E+T$	$E \rightarrow E+T$
12	$\#E$	$\#$		接受

$E' \Rightarrow \#E \Rightarrow \#E+T \Rightarrow \#E+F$
 $\Rightarrow \#E+P \Rightarrow \#E+i \Rightarrow \#T+i$
 $\Rightarrow \#F+i \Rightarrow \#P+i \Rightarrow \#i+i$

(1) $E \longrightarrow T | E+T$

(2) $T \longrightarrow F | T * F$

(3) $F \longrightarrow P | P \wedge F$

(4) $P \longrightarrow (E) | i$

■ 规范归约和算符优先归约

- 句型 $i+i\#$ 的算符优先归约过程

	+	i	#
+	'>	'<	'>
i	'>		'>
#	'<	'<	'='

步骤	栈	优先关系	当前符号	剩余符号串	动作
1	#	'<	i	+i#	移进
2	#i	'>	+	i#	归约
3	#P	'<	+	i#	移进
3	#P+	'<	i	#	移进
4	#P+i	'>	#		归约
4	#P+P	'>	#		归约
4	#E	'='	#		接受

- (1) $E \longrightarrow T | E+T$
- (2) $T \longrightarrow F | T * F$
- (3) $F \longrightarrow P | P \wedge F$
- (4) $P \longrightarrow (E) | i$

■ 算符优先分析法优缺点

- 由于算符优先分析并未对文法的非终结符定义优先关系，所以就无法发现由单个非终结符组成的“可归约串”。
- 也就是说，在算符优先归约过程中，我们无法用那些右部仅含一个非终结符的产生式（称为单非产生式，如 $P \rightarrow Q$ ）进行归约。这样，算符优先分析比规范归约要快很多。这既是算符优先分析的优点，也是它的缺点。
- 忽略非终结符在归约过程中的作用，存在某种危险性，可能导致把本来不成句子的输入串误认为是句子。

■ 优先函数

- 优先函数:
 - 在实际实现算法分析时;用两个优先函数 f, g ,把终结符 Q 与两个自然数 $f(Q)$ 和 $g(Q)$ 相对应.
- 使得:
 - 若 $Q1 < Q2$ 则 $f(Q1) < g(Q2)$
 - 若 $Q1 = Q2$ 则 $f(Q1) = g(Q2)$
 - 若 $Q1 > Q2$ 则 $f(Q1) > g(Q2)$
- 函数 f 称为入栈优先函数. 函数 g 称为比较优先函数.

■ 例子

- (1) $E \longrightarrow T|E+T$
- (2) $T \longrightarrow F|T * F$
- (3) $F \longrightarrow P|P^{\wedge} F$
- (4) $P \longrightarrow (E) | i$

	+	*	^	i	()	#
+	'>	<'	<'	<'	<'	'>	'>
*	'>	'>	<'	<'	<'	'>	'>
^	'>	'>	<'	<'	<'	'>	'>
i	'>	'>	'>			'>	'>
(<'	<'	<'	<'	<'	='	
)	'>	'>	'>			'>	'>
#	<'	<'	<'	<'	<'		='

	+	*	^	()	i	#
f	2	4	5	0	6	6	0
g	1	3	6	7	0	7	0

■ 优先函数

• 优先函数的问题

- 优先关系表中的空白项是模糊的
- 每个终结符都对应一对优先函数, 而数是可以比较的, 所以容易掩盖错误
不便区分一目运算“-” “+” 和二目运算“-” “+”.

	a	b
a	='	'>
b	='	='

$f(a)=g(a), f(a)>g(b), f(b)=g(a), f(b)=g(b)$

$f(a)>g(b)=f(b)=g(a)=f(a)$

• 优先关系表的存储方式

- 使得矩阵表示: 准确直观; 需要大量内存空间 $(n+1)^2$
- 优先函数: 需要内存空间小 $2(n+1)$; 不利于出错处理

■ 优先函数的构造算法

- 输入：算符优先表
- 输出：表示输入矩阵的优先函数或指出它不存在
- 方法：
 - 为每个终结符 a （包括 $\#$ ）创建 f_a 和 g_a 。并以其作为结点，画出 $2n$ 个结点
 - 如果 $a > b$ 或者 $a = b$ ，则画一条从 f_a 到 g_b 的箭弧
 - 如果 $a < b$ 或者 $a = b$ ，则画一条从 g_b 到 f_a 的箭弧
 - 如果构造的图中有环路，则不存在优先函数；如果没有环路，则将 $f(a)$ 设为从 f_a 出发所能到达的结点个数（包括初始结点）； $g(a)$ 为从 g_a 出发所能到达的结点个数（包括初始结点）。
 - 检查与原算符优先表是否一致！

■ 构造优先函数-例子

	i	*	+	#
i		>	>	>
*	<	>	>	>
+	<	<	>	>
#	<	<	<	=

	i	*	+	#
f	6	6	4	2
g	7	5	3	2

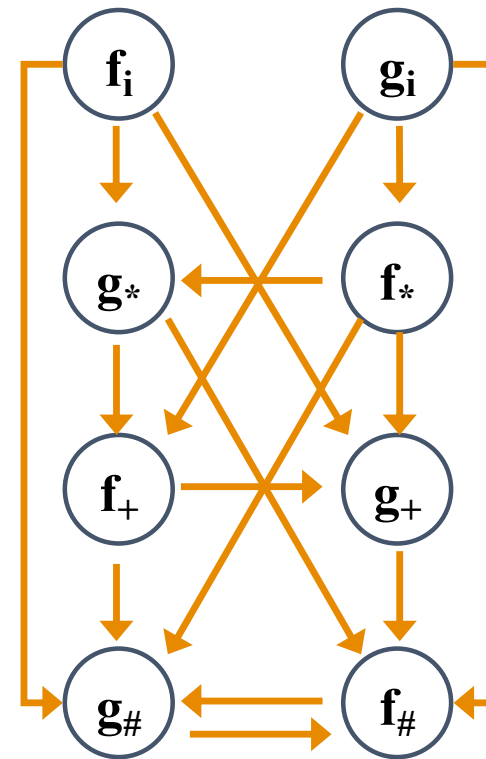
检查是否一致！

构造优先函数-例子

	i	*	+	#
i		>	>	>
*	<		>	>
+	<	<		>
#	<	<	<	

	i	*	+	#
f	7	6	4	2
g	6	5	3	2

检查是否一致!



■ 算符优先分析中的错误恢复

- 算符优先分析器能发现的语法错误
 - 如果栈顶的终结符和当前输入之间没有优先关系（如果用优先函数存储，这个错误不能发现）
 - 如果发现最左素短语，但最左素短语不是任何产生式的右部
- 归约时的错误处理
 - 给出相应的具有描述性的出错信息
 - 试图通过插入、删除来获得句柄

■ 一个加入了出错处理的优先矩阵

- E1: 缺少整个表达式
 - 把id插入输入字符串中
 - 输出诊断信息
 - Missing operand
- E2: 表达式以右括号开始
 - 从输入中删除)
 - 输出诊断信息
 - Unbalanced right parenthesis
- E3: id/)后面跟id/(
 - 把+插入输入字符串
 - 输出诊断信息
 - Missing operator
- E4: 表达式以左括号结束
 - 从栈中弹出(
 - 输出诊断信息
 - Missing right parenthesis

	id	()	#
id	E3	E3	>	>
(<	<	=	E4
)	E3	E3	>	>
#	<	<	E2	E1

■ Outline

- 自底向上的语法分析基本问题
 - 移动 – 归约分析法
 - 用栈实现移动归约分析
- 算符优先分析法
 - 算符优先分析法定义、优先分析表的确定、优先函数的定义
 - 使用算符优先关系进行分析
 - 算符优先分析中的错误恢复
- LR分析法
 - LR(0)
 - SLR
 - LR(1)
- 语法分析器的自动产生工具Yacc

■ LR语法分析器概述

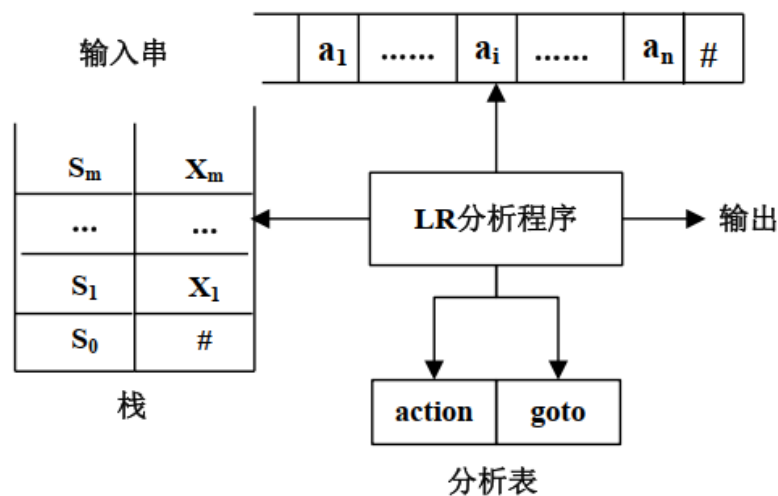
- LR(k)语法分析器适用于一大类上下文无关文法的语法分析。K省略时，默认k是1。
 - L指的是从左向右扫描输入字符串
 - R指的是构造最右推导的逆过程
 - k指的是在决定语法分析动作时需要向前看的符号个数
- 构造LR分析表的三种方法
 - 简单LR方法（SLR），最容易实现，功能最弱
 - 规范LR方法，功能最强，代价最高
 - 向前看的LR方法（LALR），其功能和代价介于前两者之间

■ LR语法分析算法

- LR分析的基本思想是，在**规范归约**过程中，一方面记住已移进和归约出的整个符号串，即记住“历史”，另一方面根据所用的产生式推测未来可能碰到的输入符号，即对未来进行“展望”。
- 当一串貌似句柄的符号串呈现于分析栈的顶端时，我们希望能够根据所记载的“**历史**”和“**展望**”以及“**现实**”的输入符号等三方面的材料，来确定栈顶的符号串是否构成相对某一产生式的句柄。

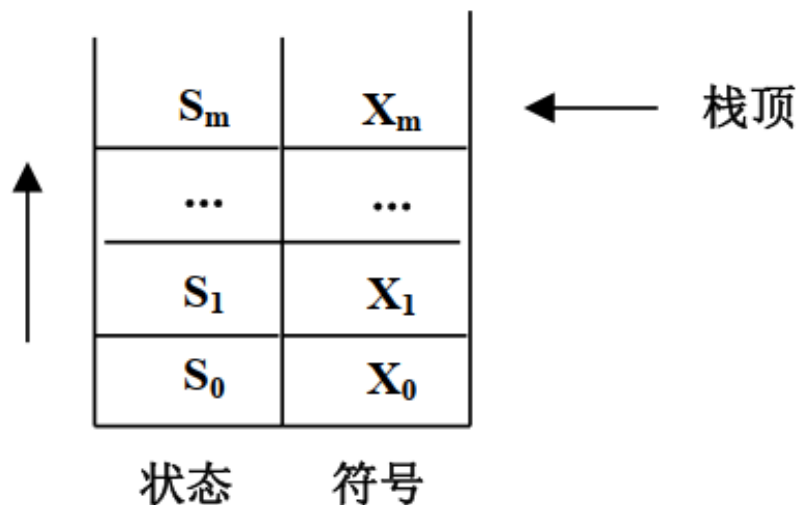
■ LR语法分析器的结构

- 一个LR分析器实质上是一个带先进后出存储器（栈）的确定有限状态自动机(DFA)。



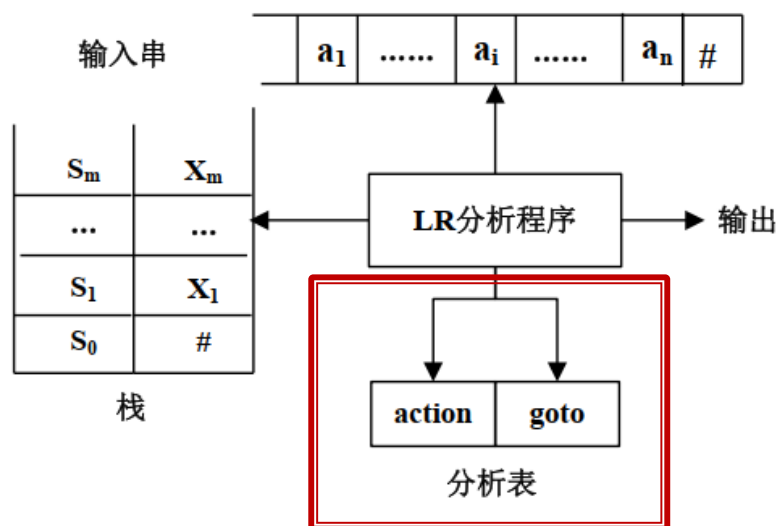
■ LR语法分析器的结构——栈

- 我们将把“历史”和“展望”材料综合的抽象成某些“状态”，存放在栈里。
- 栈里的每个状态概括了从分析开始直到某一归约阶段的全部“历史”和“展望”资料。



■ LR语法分析器的结构——分析表

- ACTION[s, a]规定了当前状态s面临输入符号a时应采取什么动作。
- GOTO[s, X]规定了状态s面对文法符号X（终结符或非终结符）时下一个状态是什么。



动作表: $\text{action}[S_m, a_i]$ 表示下一个输入符号为 a_i , 栈顶状态为 S_m 时,分析算法应执行的动作;
若 $\text{action}[s_m, a_i] = s_i$,表示移进, 然后栈顶状态为 s_i ;
若 $\text{action}[s_m, a_i] = r_j$ 表示使用产生式(j)归约;
若 $\text{action}[s_m, a_i] = \text{acc}$ 表示接受输入串;
若 $\text{action}[s_m, a_i] = \text{err}$ 表示语法错误.

状态转换表: $\text{GOTO}[S_i, X]$ 表示归约出非终结符号X之后, 当前栈顶状态为 S_i 时,分析栈应转换到的下一个状态, 即栈顶的新状态.

■ LR语法分析器的结构——ACTION表

- 每一项ACTION[s,a]所规定的动作是下述四种可能之一。
 - **移进**：把(s,a)的下一个状态 $s' = \text{GOTO}[s,a]$ 和输入符号a推进栈，下一个输入符号变成现行输入符号
 - **归约**：指用某一个产生式 $A \rightarrow \beta$ 进行归约。假若 β 的长度为r，归约的动作是去除栈顶的r个项，使状态 S_{m-r} 变成栈顶状态，然后把 (S_{m-r}, A) 的下一个状态 $s' = \text{GOTO}[S_{m-r}, A]$ 和文法符号A推进栈。
 - **接受**：宣布分析成功，停止分析器的工作。
 - **报错**：发现源程序含有错误，调用出错处理程序。

注意：若a为终结符，则 $\text{GOTO}[S,a]$ 的值已列在 $\text{action}[S,a]$ 的sj之中(状态j)。因此 **GOTO**表仅对所有非终结符A列出 $\text{GOTO}[S, A]$ 的值。

■ LR分析器的工作原理

分析栈中的串和等待输入的符号串构成如下形式的三元组:

$(S_0S_1S_2\dots S_m, \#X_1X_2\dots X_m, a_ia_{i+1} \dots a_n\#)$

一个LR分析器的工作过程就是一步一步的变换三元式直至到“接受”或“报错”为止.

1. 其初态为: $(S_0, \#, a_1a_2 \dots a_ia_{i+1} \dots a_n\#)$
2. 假定当前分析栈的栈顶为状态 S_m ,下一个输入符号为 a_i ,分析器的下一个动作
 - 如果 $\text{action}[S_m, a_i] = \text{移进}$ 且 $S = \text{GOTO}[S_m, a_i]$,则分析器执行移进,三元组变成 $(S_0S_1S_2\dots S_mS, \#X_1X_2\dots X_ma_i, a_{i+1}\dots a_n\#)$ 即分析器将输入符号 a_i 和状态 S 移进栈, a_{i+1} 变成下一个符号;
 - 如果 $\text{action}[S_m, a_i] = \text{归约 } A \rightarrow \beta$,则分析器执行归约,三元组变成 $(S_0S_1S_2\dots S_{m-r}S, \#X_1X_2\dots X_{m-r}A, a_ia_{i+1} \dots a_n\#)$,此处 $S = \text{GOTO}[S_{m-r}, A]$, r 为 β 的长度且 $\beta = X_{m-r+1}X_{m-r+2}\dots X_m$;
 - 若 $\text{action}[S_m, a_i] = \text{acc}$,则接收输入符号串,语法分析完成;
 - 若 $\text{action}[S_m, a_i] = \text{err}$,则发现语法错误,调用错误恢复子程序进行处理。

■ LR语法分析器分析过程举例

$i*i+i$ 的分析过程

文法G:

(1) $E \rightarrow E+T$

(2) $E \rightarrow T$

(3) $T \rightarrow T*F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow i$

状态	ACTION						GOTO		
	i	+	*	()	#	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			

s_j : 把下一状态j和现行输入符号a移进栈

r_j : 按第j个产生式进行规约

acc: 接受

空白格: 出错标志, 报错

步骤	状态	符号	输入串
1	0	#	$i*i+i\#$
2	05	#i	$*i+i\#$
3	03	#F	$*i+i\#$
4	02	#T	$*i+i\#$
5	027	#T*	$i+i\#$
6	0275	#T*i	$+i\#$
7	02710	#T*F	$+i\#$
8	02	#T	$+i\#$
9	01	#E	$+i\#$
10	016	#E+	$i\#$
11	0165	#E+i	$\#$
12	0163	#E+F	$\#$
13	0169	#E+T	$\#$
14	01	#E	$\#$

■ LR文法

- 一个文法，如果能为其构造一张分析表,使得它的每个入口均是唯一确定的，则这个文法为**LR文法**.
- 一个文法，如果能用一个每步顶多**向前检查k个输入符号**的LR分析器进行分析，则这个文法就称为**LR(k)文法**。
 - 我们关注 $k \leq 1$ 的情形

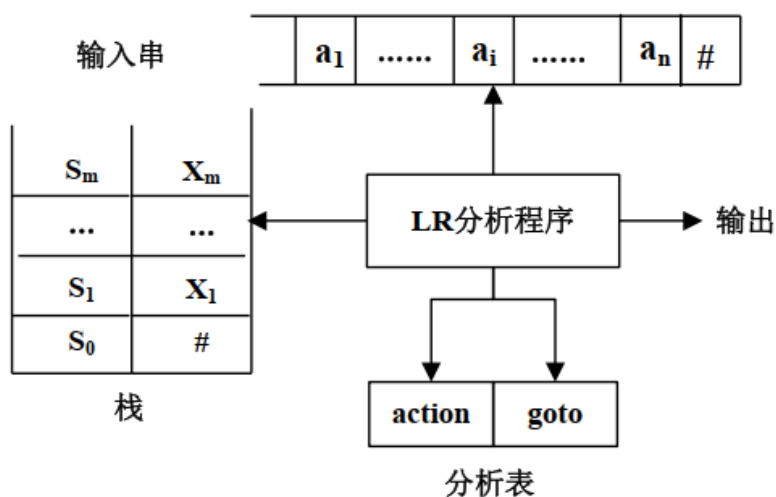
LL文法要求每个非终结符的所有候选首字符均不相同，因为预测程序认为，一旦看到首符之后就看准了该用哪一个产生式进行推导。

LR分析程序只有在看到整个右部所推导的东西之后才认为是看准了归约方向。因此，LR方法比LL方法更加一般化。

■ LR语法分析器的优缺点

- LR分析的**优点**有以下几点：
 - LR语法分析器能识别几乎所有能用**上下文无关文法**描述的程序设计语言的结构。
 - LR分析法是已知的最一般的**无回溯**移动归约语法分析法，而且可以和其他移动归约分析一样被有效地实现。
 - LR分析法分析的文法类是预测分析法能分析的文法类的真超集。
 - 在自左向右扫描输入符号串时，LR语法分析器能及时发现语法错误。
- 这种分析方法的主要**缺点**是，对典型的程序设计语言文法，手工构造LR语法分析器的工作量太大，需要专门的工具。

■ 回顾



- ACTION[s, a]规定了当前状态s面临输入符号a时应采取什么动作。
- GOTO[s, X]规定了状态s面对文法符号X（终结符或非终结符）时下一个状态是什么。
- 我们将把“历史”和“展望”材料综合的抽象成某些“状态”，存放在栈里。

■ LR(0)项目集族和LR(0)分析表的构造

- 前缀：字的前缀是指该字的任意首部。
 - 字abc的前缀有 ϵ 、a、ab、abc。
- 活前缀：指规范句型的一个前缀，这种前缀不含句柄之后任何符号。之所以称为活前缀，是因为在右边添加一些符号之后，就可以使它成为一个规范句型。
- 在LR分析工作过程中的任何时候，栈里的文法符号（自栈底而上） $X_0 X_1 \cdots X_m$ 应该构成活前缀，把输入串的剩余部分配上之后即应成为规范句型。
 - 只要输入串的已扫描部分保持可归约成一个活前缀，那就意味着所扫描过的部分没有错误。

■ LR(0)项目的定义

- 对于一个文法G，我们首先要构造一个NFA，它能识别G的所有活前缀。这个NFA的每一个状态是下面定义的一个“项目”。
- 文法G的每一个产生式的右部添加一个圆点称为G的一个LR(0)项目（简称项目）
 - 例如产生式 $A \rightarrow XYZ$ 对应四个项目：
 $A \rightarrow \bullet XYZ$
 $A \rightarrow X \bullet YZ$
 $A \rightarrow XY \bullet Z$
 $A \rightarrow XYZ \bullet$
产生式 $A \rightarrow \epsilon$ 只对应一个项目 $A \rightarrow \bullet$

■ LR(0)项目的说明

- 直观上说，一个项目指明了在分析过程的某时刻我们看到产生式多大部分。
 - 例如， $A \rightarrow \bullet XYZ$ 这个项目意味着，我们希望能从后面输入串中看到可以从XYZ推出的符号串。
 - $A \rightarrow X \bullet YZ$ 这个项目意味着，我们已经从输入串中看到能从X推出的符号串，我们希望能进一步看到可以从YZ推出的符号串。
- 将每个项目看成NFA的一个状态，我们就可以构造这样一个NFA，用来识别文法所有的活前缀。

■ LR(0)项目间的转换规则

- 如果状态i和j来自**同一个产生式**，而且状态j的圆点只落后于状态i的圆点一个位置，
 - 如状态i为 $X \rightarrow X_1 \cdots X_{i-1} \bullet X_i \cdots X_n$,
 - 状态j为 $X \rightarrow X_1 \cdots X_i \bullet X_{i+1} \cdots X_n$,
 - 那么就从状态i画一条标志为 X_i 的弧到状态j。
- 假若状态i的圆点之后的那个符号为**非终结符**，
 - 如i为 $X \rightarrow \alpha \bullet A \beta$ ，A为非终结符，
 - 就从状态i画 ϵ 弧到所有 $A \rightarrow \bullet \gamma$ 状态。
- NFA的接受状态就是那些圆点出现在最右边的项目。

■ LR(0)项目分类

- 归约项目
 - 凡圆点在最右的项目，如 $A \rightarrow \alpha \bullet$ 称为一个“归约”项目
- 接受项目
 - 对文法的开始符号 S' 的归约项目，如 $S' \rightarrow \alpha \bullet$ 称为“接受”项目。
- 移进项目
 - 形如 $A \rightarrow \alpha \bullet a \beta$ 的项目，其中 a 为终结符，称为“移进”项目。
- 待约项目
 - 形如 $A \rightarrow \alpha \bullet B \beta$ 的项目，其中 B 为非终结符，称为“待约”项目。

■ 识别LR(0)活前缀的NFA例子I

文法G:

$S' \rightarrow E$

$E \rightarrow aA \mid bB$

$A \rightarrow cA \mid d$

$B \rightarrow cB \mid d$

文法的项目有:

1. $S' \rightarrow \bullet E$

2. $S' \rightarrow E \bullet$

3. $E \rightarrow \bullet aA$

4. $E \rightarrow a \bullet A$

5. $E \rightarrow aA \bullet$

6. $A \rightarrow \bullet cA$

7. $A \rightarrow c \bullet A$

8. $A \rightarrow cA \bullet$

9. $A \rightarrow \bullet d$

10. $A \rightarrow d \bullet$

11. $E \rightarrow \bullet bB$

12. $E \rightarrow b \bullet B$

13. $E \rightarrow bB \bullet$

14. $B \rightarrow \bullet cB$

15. $B \rightarrow c \bullet B$

16. $B \rightarrow cB \bullet$

17. $B \rightarrow \bullet d$

18. $B \rightarrow d \bullet$

■ 识别LR(0)活前缀的NFA例子II

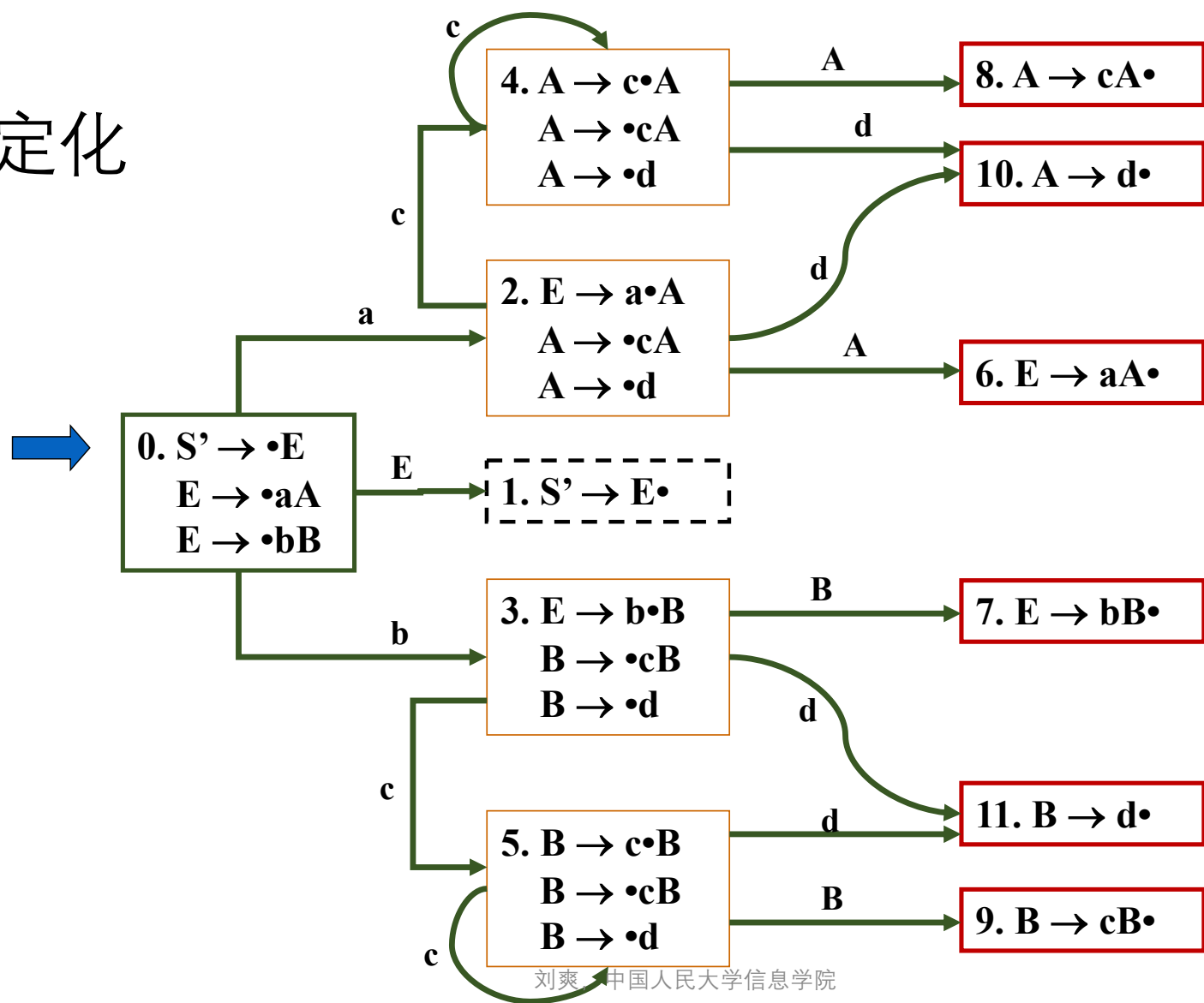
1. $S' \rightarrow \bullet E$
2. $S' \rightarrow E \bullet$
3. $E \rightarrow \bullet aA$
4. $E \rightarrow a \bullet A$
5. $E \rightarrow aA \bullet$
6. $A \rightarrow \bullet cA$
7. $A \rightarrow c \bullet A$
8. $A \rightarrow cA \bullet$
9. $A \rightarrow \bullet d$
10. $A \rightarrow d \bullet$

11. $E \rightarrow \bullet bB$
12. $E \rightarrow b \bullet B$
13. $E \rightarrow bB \bullet$
14. $B \rightarrow \bullet cB$
15. $B \rightarrow c \bullet B$
16. $B \rightarrow cB \bullet$
17. $B \rightarrow \bullet d$
18. $B \rightarrow d \bullet$

- 初态
 - $S' \rightarrow \bullet E$ [1]
- 句柄识别态
 - $E \rightarrow aA \bullet$ [5]
 - $A \rightarrow cA \bullet$ [8]
 - $A \rightarrow d \bullet$ [10]
 - $E \rightarrow bB \bullet$ [13]
 - $B \rightarrow cB \bullet$ [16]
 - $B \rightarrow d \bullet$ [18]
- 句子识别态
 - $S' \rightarrow E \bullet$ [2]



■ 确定化



■ 有效项目

- 有效项目

- 我们说项目 $A \rightarrow \beta_1 \bullet \beta_2$ 对活前缀 $\alpha \beta_1$ 是有效的, 其条件是存在规范推导 $S' \xRightarrow{*} \alpha A \omega \Rightarrow \alpha \beta_1 \beta_2 \omega$
- 事实上, 活前缀 γ 有效项目集, 是从上述的 DFA 的初态出发, 经读出 γ 后而得到的那个项目集状态
- 在任何时候, 分析栈中的活前缀 $X_1 X_2 \cdots X_m$ 的有效项目集正是栈顶状态 S_m 所代表的那个集合

- 例子

- DFA** 的一个活前缀 bc , 使 **DFA** 到达 5, 5 有三个项目
下面说明项目集对 bc 是有效的. 考虑下面三个推导:

- (1) $S' \Rightarrow E \Rightarrow bB \Rightarrow bcB$
- (2) $S' \Rightarrow E \Rightarrow bcB \Rightarrow bccB$
- (3) $S' \Rightarrow E \Rightarrow bB \Rightarrow bcB \Rightarrow bcd$

5. $B \rightarrow c \bullet B$
 $B \rightarrow \bullet c B$
 $B \rightarrow \bullet d$

推导 (1) (2) (3) 分别证明了 $B \rightarrow c \bullet B$
 $B \rightarrow \bullet c B$ $B \rightarrow \bullet d$ 的有效性

■ LR(0)项目集规范族的构造

构成识别一个文法活前缀的DFA的项目集(状态)
的全体称为这个文法的LR(0)项目集规范族

- 用 ε -CLOSURE(闭包)的办法来构造一个文法G的LR(0)项目集规范族。
- 准备工作：
 - 假定文法G是一个以S为开始符号的文法，我们构造一个文法G'，它包含整个G，但它引进了一个不出现在G中的非终结符S'，并加进一个新产生式 $S' \rightarrow S$ ，而这个S'是G'的开始符号。那么我们称G'是G的拓广文法。
这样，便会有一个仅含项目 $S' \rightarrow S \cdot$ 的状态，这就是唯一的“接受”状态。

■ LR(0)项目集规范族的构造

- 假定 I 是文法 G' 的任一项目集，定义和构造 I 的闭包 $CLOSURE(I)$ 的办法是：
 - I 的任何项目都属于 $CLOSURE(I)$;
 - 若 $A \rightarrow \alpha \bullet B \beta$ 属于 $CLOSURE(I)$ ，那么，对任何关于 B 的产生式 $B \rightarrow \gamma$ ，项目 $B \rightarrow \bullet \gamma$ 也属于 $CLOSURE(I)$;
 - 重复执行上述两步骤直至 $CLOSURE(I)$ 不再增大为止。
- 函数 $GO(I, X)$ 是一个状态转换函数。
 - 第一个变元 I 是一个项目集，
 - 第二个变元 X 是一个文法符号。
 - 函数值 $GO(I, X)$ 定义为 $GO(I, X) = CLOSURE(J)$ ，
其中 $J = \{ \text{任何形如 } A \rightarrow \alpha X \bullet \beta \text{ 的项目} \mid A \rightarrow \alpha \bullet X \beta \text{ 属于 } I \}$

直观上说，若 I 是对某个活前缀 γ 的有效项目集，那么 $GO(I, X)$ 便是对 γX 有效的项目集。

■ 构造LR(0)项目集规范族例子

- 初始状态 I_0 的项目集规范族:

$$I_0 = \{ S' \rightarrow \bullet E, E \rightarrow \bullet aA, E \rightarrow \bullet bB \}$$

- I_1 、 I_2 、 I_3 和分别是 $GO(I_0, E)$, $GO(I_0, a)$ 和 $GO(I_0, b)$

- $I_1 = \text{CLOSURE}(S' \rightarrow E \bullet) = \{ S' \rightarrow E \bullet \}$
- $I_2 = \text{CLOSURE}(E \rightarrow a \bullet A) = \{ E \rightarrow a \bullet A, A \rightarrow \bullet cA, A \rightarrow \bullet d \}$
- $I_3 = \text{CLOSURE}(E \rightarrow b \bullet B) = \{ E \rightarrow b \bullet B, B \rightarrow \bullet cB, B \rightarrow \bullet d \}$

文法G:

(0) $S' \rightarrow E$

(1) $E \rightarrow aA$

(2) $E \rightarrow bB$

(3) $A \rightarrow cA$

(4) $A \rightarrow d$

(5) $B \rightarrow cB$

(6) $B \rightarrow d$

■ LR(0)项目集规范族的构造算法

- 通过函数CLOSURE和GO构造一个文法G的拓广的文法G'的LR(0)项目集规范族.

```
Procedure itemsets(G'):
begin
  C:={ closure( {S'→·S})};
  repeat
    for C 中的每个项目集I和G'的每个符号X do
      if GO(I,X) 非空且不属于C then
        GO(I,X)放入C族中
  until C不再增大
end
```

■ LR(0)项目集规范族的讨论

- 同一项目可能对好几个活前缀都是有效的（当一个项目出现在好几个不同的集合中便是这种情况）。
- LR(0)项目集可能出现的冲突
 - 若归约项目 $A \rightarrow \beta_1 \bullet$ 对活前缀 $\alpha\beta_1$ 是有效的，则它告诉我们应该把符号串 β_1 **归约** 为 A 。即把活前缀 $\alpha\beta_1$ 变成 αA 。
 - 若移进项目 $A \rightarrow \beta_1 \bullet \beta_2$ 对活前缀 $\alpha\beta_1$ 是有效的，则它告诉我们，句柄尚未形成，因此，下一步动作应该是**移进**。
 - 但是，可能存在这样的情形，对同一活前缀，存在若干项目对它都是有效的，而且告诉我们应该做的事情各不相同，互相**冲突**。这种冲突通过向前多看几个输入符号，或许能够解决，但有些冲突却是无法解决的。

■ LR(0)文法

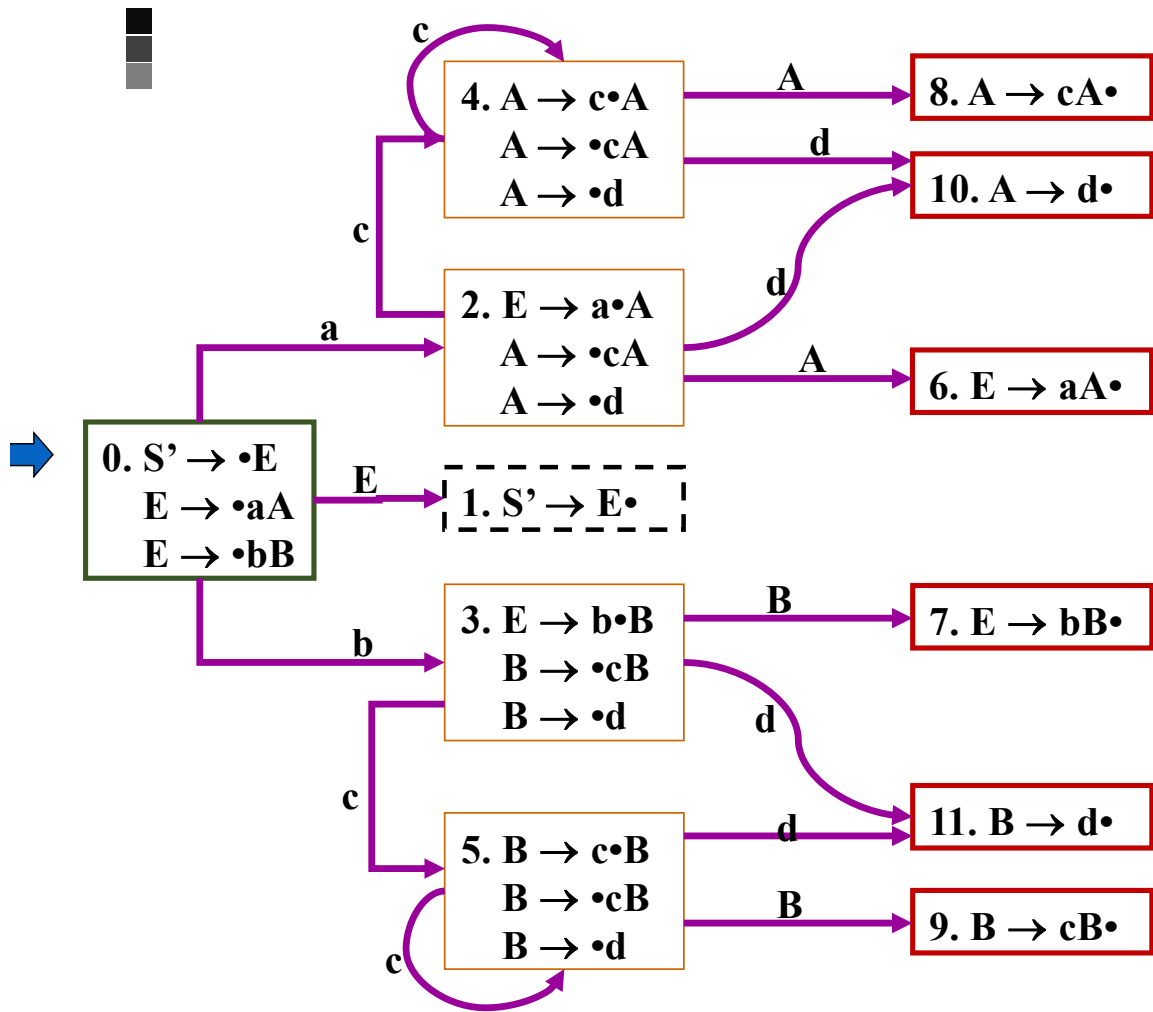
- 假如一个文法G的拓广文法G'的活前缀识别自动机的每个状态（项目集）不存在下述情况：
 - 既含移进项目又含归约项目；
 - 含多个归约项目。

则称G是一个LR(0)文法。换言之，LR(0)文法规范族的每个项目集不包含任何冲突项目。

■ LR(0)分析表的构造

- 假定项目集规范族 $C=\{I_0, I_1, \dots, I_n\}$ 。令每一个项目集 I_k 的下标 k 作为分析器的状态。分析表的ACTION子表和GOTO子表可按如下方法构造
 - (0) 令那个包含项目 $S' \rightarrow \bullet S$ 的集合 I_k 的下标 k 为分析器的初态。
 - (1) 若项目 $A \rightarrow \alpha \bullet a \beta$ 属于 I_k 且 $GO(I_k, a) = I_j$, a 为终结符, 置 $ACTION[k, a]$ 为“把 (j, a) 移进栈”, 简记为“ s_j ”。
 - (2) 若项目 $A \rightarrow \alpha \bullet$ 属于 I_k , 对**任何**终结符 a (或结束符 $\#$), 置 $ACTION[k, a]$ 为“用产生式 $A \rightarrow \alpha$ 进行归约”, 简记为“ r_j ”(假定产生式 $A \rightarrow \alpha$ 是文法 G' 的第 j 个产生式)。
 - (3) 若项目 $S' \rightarrow S \bullet$ 属于 I_k , 则置 $ACTION[k, \#]$ 为“接受”, 简记为“acc”。
 - (4) 若 $GO(I_k, A) = I_j$, A 为非终结符, 则置 $GOTO[k, A] = j$ 。
 - (5) 分析表中凡不能用规则1至4填入信息的空白格均填上“报错标志”。

按上述方法构造的分析表的每个入口都是唯一的。称此分析表是一个**LR(0)**表。使用**LR(0)**表的分析器叫做一个**LR(0)**分析器



文法G:

(0) $S' \rightarrow E$

(1) $E \rightarrow aA$

(2) $E \rightarrow bB$

(3) $A \rightarrow cA$

(4) $A \rightarrow d$

(5) $B \rightarrow cB$

(6) $B \rightarrow d$

状态	Action					GOTO		
	a	b	c	d	#	E	A	B
0	S2	S3				1		
1					acc			
2			S4	S10			6	
3			S5	S11				7
4			S4	S10			8	
5			S5	S11				9
6	r1	r1	r1	r1	r1			
7	r2	r2	r2	r2	r2			
8	r3	r3	r3	r3	r3			
9	r5	r5	r5	r5	r5			
10	r4	r4	r4	r4	r4			
11	r6	r6	r6	r6	r6			

■ 练习：构造LR(0)分析表

- 设有如下文法： $G = \{V_T, V_N, S', P\}$, 其中：
 $V_T = \{\text{var} \text{ : } , \text{id} \text{ real}\}$ （空格是分隔符，逗号是终结符）
 $V_N = \{S' \ S \mid T\}$ （空格是分隔符）

P如下：

- (1) $S' \rightarrow S$
- (2) $S \rightarrow \text{var } I : T$
- (3) $I \rightarrow I, \text{id}$
- (4) $I \rightarrow \text{id}$
- (5) $T \rightarrow \text{real}$

构造该文法的LR(0)分析表

(1) 列出G的所有LR(0)项目

(1) $S' \rightarrow S$	$S' \rightarrow \cdot S$	$S' \rightarrow S \cdot$			
(2) $S \rightarrow \text{var } l : T$	$S \rightarrow \cdot \text{var } l : T$	$S \rightarrow \text{var} \cdot l : T$	$S \rightarrow \text{var } l \cdot : T$	$S \rightarrow \text{var } l : \cdot T$	$S \rightarrow \text{var } l : T \cdot$
(3) $l \rightarrow l, id$	$l \rightarrow \cdot l, id$	$l \rightarrow l \cdot , id$	$l \rightarrow l, \cdot id$	$l \rightarrow l, id \cdot$	
(4) $l \rightarrow id$	$l \rightarrow \cdot id$	$l \rightarrow id \cdot$			
(5) $T \rightarrow \text{real}$	$T \rightarrow \cdot \text{real}$	$T \rightarrow \text{real} \cdot$			

(2) 构造G4的项目集规范族及识别活前缀的DFA



(3) 构造G的LR(0)分析表

	Action						GOTO		
	var	:	,	id	real	#	S	I	T
0									
1									
2									
3									
4									
5									
6									
7									
8									
9									

(2) 构造G的项目集规范族

$I_0 = \{ S' \rightarrow \cdot S, S \rightarrow \cdot \text{var } I : T \}$

$I_1 = \{ S' \rightarrow S \cdot \}$

$I_2 = \{ S \rightarrow \text{var} \cdot I : T, I \rightarrow \cdot I, \text{id}, I \rightarrow \cdot \text{id} \}$

$I_3 = \{ S \rightarrow \text{var } I \cdot : T, I \rightarrow I \cdot, \text{id} \}$

$I_4 = \{ I \rightarrow \text{id} \cdot \}$

$I_5 = \{ S \rightarrow \text{var } I : \cdot T, T \rightarrow \cdot \text{real} \}$

$I_6 = \{ I \rightarrow I, \cdot \text{id} \}$

$I_7 = \{ S \rightarrow \text{var } I : T \cdot \}$

$I_8 = \{ T \rightarrow \text{real} \cdot \}$

$I_9 = \{ I \rightarrow I, \text{id} \cdot \}$

$GO(I_0, S) = I_1$
 $GO(I_0, \text{var}) = I_2$
 $GO(I_2, I) = I_3$
 $GO(I_2, \text{id}) = I_4$
 $GO(I_3, :) = I_5$
 $GO(I_3, \text{id}) = I_6$
 $GO(I_5, T) = I_7$
 $GO(I_5, \text{real}) = I_8$
 $GO(I_6, \text{id}) = I_9$

- (1) $S' \rightarrow S$
- (2) $S \rightarrow \text{var } I : T$
- (3) $I \rightarrow I, \text{id}$
- (4) $I \rightarrow \text{id}$
- (5) $T \rightarrow \text{real}$

(1) 列出G的所有LR(0)项目

$S' \rightarrow \cdot S$ $S' \rightarrow S \cdot$
 $S \rightarrow \cdot \text{var } I : T$ $S \rightarrow \text{var } I : T \cdot$ $S \rightarrow \text{var } I \cdot : T$ $S \rightarrow \text{var } I : \cdot T$ $S \rightarrow \text{var } I : T \cdot$
 $I \rightarrow \cdot I, id$ $I \rightarrow I \cdot , id$ $I \rightarrow I, \cdot id$ $I \rightarrow I, id \cdot$
 $I \rightarrow \cdot id$ $I \rightarrow id \cdot$
 $T \rightarrow \cdot \text{real}$ $T \rightarrow \text{real} \cdot$

(2) 构造G的项目集规范族

$I_0 = \{ S' \rightarrow \cdot S, S \rightarrow \cdot \text{var } I : T \}$ $GO(I_0, S) = I_1$
 $I_1 = \{ S' \rightarrow S \cdot \}$ $GO(I_0, \text{var}) = I_2$
 $I_2 = \{ S \rightarrow \text{var } I : T, I \rightarrow \cdot I, id, I \rightarrow \cdot id \}$ $GO(I_2, I) = I_3$
 $I_3 = \{ S \rightarrow \text{var } I \cdot : T, I \rightarrow I \cdot , id \}$ $GO(I_2, id) = I_4$
 $I_4 = \{ I \rightarrow id \cdot \}$ $GO(I_3, :) = I_5$
 $I_5 = \{ S \rightarrow \text{var } I : \cdot T, T \rightarrow \cdot \text{real} \}$ $GO(I_3, ,) = I_6$
 $I_6 = \{ I \rightarrow I, \cdot id \}$ $GO(I_5, T) = I_7$
 $I_7 = \{ S \rightarrow \text{var } I : T \cdot \}$ $GO(I_5, \text{real}) = I_8$
 $I_8 = \{ T \rightarrow \text{real} \cdot \}$ $GO(I_6, id) = I_9$
 $I_9 = \{ I \rightarrow I, id \cdot \}$

(3) 构造G的LR(0)分析表

	Action						GOTO		
	var	:	,	id	real	#	S	I	T
0	S2						1		
1						Ac c			
2				s4				3	
3		S5	S6						
4	R4	R4	R4	R4	R4	R4			
5					S8				7
6				S9					
7	R2	R2	R2	R2	R2	R2			
8	R5	R5	R5	R5	R5	R5			
9	R3	R3	R3	R3	R3	R3			

写出 var a, b : real 的分析过程

- (1) $S' \rightarrow S$

(2) $S \rightarrow \text{var } I : T$

(3) $I \rightarrow I, \text{id}$

(4) $I \rightarrow \text{id}$

(5) $T \rightarrow \text{real}$

	Action						GOTO		
	var	:	,	id	rea	#	S	I	T
0	S2						1		
1						Acc			
2				s4				3	
3		S5	S6						
4	R4	R4	R4	R4	R4	R4			
5					S8				7
6				S9					
7	R2	R2	R2	R2	R2	R2			
8	R5	R5	R5	R5	R5	R5			
9	R3	R3	R3	R3	R3	R3			

步骤	状态	符号	输入串
1	0	#	var a, b : real#
2	02	#var	a, b : real#
3	024	#var a	, b : real#
4	023	#var I	, b : real#
5	0236	#var I,	b : real#
6	02369	#var I,b	: real#
7	023	#var I	: real#
8	0235	#var I :	real#
9	02358	#var I : real	#
10	02357	#var I : T	#
11	01	#S	#
12	01	#S	#

■ Outline

- 自底向上的语法分析基本问题
 - 移动 – 归约分析法
 - 用栈实现移动归约分析
- 算符优先分析法
 - 算符优先分析法定义、优先分析表的确定、优先函数的定义
 - 使用算符优先关系进行分析
 - 算符优先分析中的错误恢复
- LR分析法
 - LR(0)
 - SLR
 - LR(1)*
- 语法分析器的自动产生工具Yacc

■ LR(0)文法的局限性

- 假定一个LR(0)规范族中含有如下的一个项目集（状态）
 $I = \{X \rightarrow \alpha \bullet b \beta, A \rightarrow \alpha \bullet, B \rightarrow \alpha \bullet\}$
- 第一个项目是移进项目，第二、三个项目是归约项目。这三个项目告诉我们应该做的动作各不相同，互相冲突：
 - 第一个项目告诉我们应该把下一个符号b移进；
 - 第二个项目告诉我们应该把栈顶的 α 归约为A；
 - 第三个项目告诉我们应该把栈顶的 α 归约为B。

SLR分析表的构造

■ SLR语法分析概述

- LR(0)文法的活前缀识别自动机的每一个状态（项目集）都不含冲突的项目。
- 对于某些含有冲突的状态，一种带有简单“展望”的LR分析法，即SLR文法，可以解决冲突。
- SLR文法构造分析表的主要思想是：许多冲突性的动作都可能通过考察有关非终结符的FOLLOW集而获解决。

$$I = \{X \rightarrow \alpha \bullet b \beta, A \rightarrow \alpha \bullet, B \rightarrow \alpha \bullet\}$$

■ SLR基本算法

- 解决冲突的方法是分析所有含A和B的句型，考察集合FOLLOW(A)和FOLLOW(B)，如果这两个集合不相交，而且也不包含b，那么当状态I面临输入符号a时，我们可以使用如下策略：
 - 若 $a=b$ ，则移进。
 - 若 $a \in \text{FOLLOW}(A)$ ，则用产生式 $A \rightarrow \alpha$ 进行归约；
 - 若 $a \in \text{FOLLOW}(B)$ ，则用产生式 $B \rightarrow \alpha$ 进行归约；
 - 此外，报错

$$I = \{X \rightarrow \alpha \bullet b \beta, A \rightarrow \alpha \bullet, B \rightarrow \alpha \bullet\}$$

■ Review:构造结合FOLLOW的算法

对文法G的每个非终结符A构造FOLLOW(A)的办法是：连续应用下列规则,直到每个后随符号集FOLLOW不再增大为止.

- 1) 对于文法的开始符号S, 置#于FOLLOW(S)中;
- 2) 若 $A \rightarrow \alpha B \beta$ 是一个产生式, 则把 $\text{FIRST}(\beta) \setminus \{\varepsilon\}$ 加至FOLLOW(B)中;
- 3) 若 $A \rightarrow \alpha B$ 是一个产生式, 或 $A \rightarrow \alpha B \beta$ 是一个产生式而 $\beta \Rightarrow \varepsilon$ (即 $\varepsilon \in \text{FIRST}(\beta)$), 则把FOLLOW(A)加至FOLLOW(B)中.

■ SLR基本算法

- 假定LR(0)规范族的一个项目集I中
 - 含有m个移进项目
 - $A_1 \rightarrow \alpha \cdot a_1 \beta_1, A_2 \rightarrow \alpha \cdot a_2 \beta_2, \dots, A_m \rightarrow \alpha \cdot a_m \beta_m;$
 - 同时含有n个归约项目
 - $B_1 \rightarrow \alpha \cdot, B_2 \rightarrow \alpha \cdot, \dots, B_n \rightarrow \alpha \cdot,$
 - 如果集合 $\{a_1, \dots, a_m\}, \text{FOLLOW}(B_1), \dots, \text{FOLLOW}(B_n)$ 两两不相交（包括不得有两个FOLLOW集合有 $\#$ ），则隐含在I中的动作冲突可以通过检查现行输入符号a属于上述n+1个集合中的哪个集合而解决：
 - 若a是某个 $a_i, i=1,2,\dots,m$, 则移进。
 - 若 $a \in \text{FOLLOW}(B_i), i=1,2,\dots,n$, 则用产生式 $B_i \rightarrow \alpha$ 进行归约;
 - 此外, 报错

这种冲突的解决方法叫做SLR(1)解决办法。

■ SLR语法分析表的构造算法

- 首先把G拓广为G', 对G'构造LR(0)项目集规范族C和活前缀识别自动机的状态转换函数GO。函数ACTION和GOTO可按如下方法构造:
 - 若项目 $A \rightarrow \alpha \bullet a \beta$ 属于 I_k , $GO(I_k, a) = I_j$, a为终结符, 置ACTION[k,a]为“把状态j和符号a移进栈”, 简记为“sj”;
 - 若项目 $A \rightarrow \alpha \bullet$ 属于 I_k , 那么, 对任何非终结符a, $a \in FOLLOW(A)$, 置ACTION[k,a]为“用产生式 $A \rightarrow \alpha$ 进行归约”, 简记为“rj”; 其中, 假定 $A \rightarrow \alpha$ 为文法G'的第j个产生式
 - 若项目 $S' \rightarrow S \bullet$ 属于 I_k , 则置ACTION[k,#]为可“接受”, 简记为“acc”;
 - 若 $GO(I_k, A) = I_j$, A为非终结符, 则置GOTO[k, A]=j;
 - 分析表中凡不能用规则1至4填入信息的空白格均填上“出错标志”。
 - 语法分析器的初始状态是包含 $S' \rightarrow \bullet S$ 的项目集合的状态

按上述方法构造的含有Action和GOTO的分析表, 若每个入口不含多重定义, 则称它为文法G的一张SLR表。具有SLR表的文法G成为一个SLR(1)文法, 使用SLR表的分析器叫SLR分析器。

■ SLR分析举例

每个SLR(1)文法都是无二义的;
但并非每个无二义的文法都是SLR(1)的。

文法G:

(0) $S' \rightarrow E$

(1) $E \rightarrow E+T$

(2) $E \rightarrow T$

(3) $T \rightarrow T * F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow i$

■ SLR分析举例

每个SLR(1)文法都是无二义的;
但并非每个无二义的文法都是SLR(1)的。

文法G:

- (0) $S' \rightarrow E$
- (1) $E \rightarrow E+T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow T^*F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow i$

$I_0: S' \rightarrow \bullet E$
 $E \rightarrow \bullet E+T$
 $E \rightarrow \bullet T$
 $T \rightarrow \bullet T^*F$
 $T \rightarrow \bullet F$
 $F \rightarrow \bullet (E)$
 $F \rightarrow \bullet i$
 $I_1: S' \rightarrow E \bullet$
 $E \rightarrow E \bullet +T$
 $I_2: E \rightarrow T \bullet$
 $T \rightarrow T \bullet ^*F$
 $I_3: T \rightarrow F \bullet$
 $I_4: F \rightarrow (\bullet E)$
 $E \rightarrow \bullet E+T$
 $E \rightarrow \bullet T$
 $T \rightarrow \bullet T^*F$
 $T \rightarrow \bullet F$
 $F \rightarrow \bullet (E)$
 $F \rightarrow \bullet i$

$I_5: F \rightarrow i \bullet$
 $I_6: E \rightarrow E+ \bullet T$
 $T \rightarrow \bullet T^*F$
 $T \rightarrow \bullet F$
 $F \rightarrow \bullet (E)$
 $F \rightarrow \bullet i$
 $I_7: T \rightarrow T^* \bullet F$
 $F \rightarrow \bullet (E)$
 $F \rightarrow \bullet i$
 $I_8: F \rightarrow (E \bullet)$
 $E \rightarrow E \bullet +T$
 $I_9: E \rightarrow E+T \bullet$
 $T \rightarrow T \bullet ^*F$
 $I_{10}: T \rightarrow T^*F \bullet$
 $I_{11}: F \rightarrow (E) \bullet$

$I_1: S' \rightarrow E \bullet$
 $E \rightarrow E \bullet +T$

$I_2: E \rightarrow T \bullet$
 $T \rightarrow T \bullet ^*F$

$I_9: E \rightarrow E+T \bullet$
 $T \rightarrow T \bullet ^*F$

$FOLLOW(S') = \{ \# \}$
 $FOLLOW(E) = \{ \#,), + \}$

■ SLR分析举例

状态	ACTION						GOTO		
	i	+	*	()	#	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			

文法G:

- (0) $S' \rightarrow E$ (1) $E \rightarrow E+T$
 (2) $E \rightarrow T$ (3) $T \rightarrow T * F$
 (4) $T \rightarrow F$ (5) $F \rightarrow (E)$
 (6) $F \rightarrow i$

$I_1: S' \rightarrow E \bullet$
 $E \rightarrow E \bullet + T$

$I_2: E \rightarrow T \bullet$
 $T \rightarrow T \bullet * F$

$I_9: E \rightarrow E + T \bullet$
 $T \rightarrow T \bullet * F$

$FOLLOW(S') = \{ \# \}$
 $FOLLOW(E) = \{ \#,), + \}$

■ LR(0)分析表 vs SLR分析表

状态	Action					GOTO		
	a	b	c	d	#	E	A	B
0	S2	S3				1		
1					acc			
2			S4	S10			6	
3			S5	S11				7
4			S4	S10			8	
5			S5	S11				9
6	r1	r1	r1	r1	r1			
7	r2	r2	r2	r2	r2			
8	r3	r3	r3	r3	r3			
9	r5	r5	r5	r5	r5			
10	r4	r4	r4	r4	r4			
11	r6	r6	r6	r6	r6			

LR(0)分析表

状态	ACTION						GOTO		
	i	+	*	()	#	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			

SLR分析表

■ SLR语法分析的局限性

- 所有的SLR语法必须满足如下条件
 - $I = \{X \rightarrow \alpha \cdot b\beta, A \rightarrow \alpha \cdot, B \rightarrow \alpha \cdot\}$
若有： $\text{FOLLOW}(A) \cap \text{FOLLOW}(B) = \emptyset$
 $\text{FOLLOW}(A) \cap \{b\} = \emptyset$
 $\text{FOLLOW}(B) \cap \{b\} = \emptyset$
- 状态I面临某输入符号a
 - 1) 若 $a=b$, 则移进
 - 2) 若 $a \in \text{FOLLOW}(A)$, 则用产生式 $A \rightarrow \gamma$ 进行归约
 - 3) 若 $a \in \text{FOLLOW}(B)$, 则用产生式 $B \rightarrow \delta$ 进行归约
 - 4) 此外, 报错

■ SLR文法中可能出现的冲突

文法G':

- (0) $S' \rightarrow S$
- (1) $S \rightarrow L=R$
- (2) $S \rightarrow R$
- (3) $L \rightarrow *R$
- (4) $L \rightarrow i$
- (5) $R \rightarrow L$

$I_0: S' \rightarrow \bullet S$

$S \rightarrow \bullet L=R$

$S \rightarrow \bullet R$

$L \rightarrow \bullet *R$

$L \rightarrow \bullet i$

$R \rightarrow \bullet L$

$I_1: S' \rightarrow S \bullet$

$I_2: S \rightarrow L \bullet =R$

$R \rightarrow L \bullet$

$I_3: S \rightarrow R \bullet$

$I_4: L \rightarrow * \bullet R$

$R \rightarrow \bullet L$

$L \rightarrow \bullet *R$

$L \rightarrow \bullet i$

$I_5: L \rightarrow i \bullet$

$I_6: S \rightarrow L = \bullet R$

$R \rightarrow \bullet L$

$L \rightarrow \bullet *R$

$L \rightarrow \bullet i$

$I_7: L \rightarrow *R \bullet$

$I_8: R \rightarrow L \bullet$

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

$= \in \text{FOLLOW}(R)$

构造规范LR语法分析表

■ 构造规范LR语法分析表

- 针对SLR语法分析的局限性，我们给出如下解决方案：
 - 重新定义项目，使之包含一个终结字符串作为第二个分量，可以把更多的信息并入状态中。
 - LR(k)项目的一般形式也就变成了 $[A \rightarrow \alpha \bullet \beta, a_1 a_2 \cdots a_k]$ ，其中 $A \rightarrow \alpha \bullet \beta$ 是LR(0)项目，每一个a都是终结符或者#。
 - 项目中的 $a_1 a_2 \cdots a_k$ 称为它的**向前搜索字符串**（或展望串）
 - 这样的a的集合是FOLLOW(A)的子集，有可能是真子集
 - 归约项目 $[A \rightarrow \alpha \bullet, a_1 a_2 \cdots a_k]$ 意味着：当它所属的状态呈现在栈顶且后续的k个输入符号为 $a_1 a_2 \cdots a_k$ 时，才可以把栈顶的 α 归约为A。我们只对 $k \leq 1$ 的情形感兴趣 ——LR(1)

■ LR(1)对活前缀有效的定义

- LR(1)的项目 $[A \rightarrow \alpha \cdot \beta, a]$ 对活前缀 γ 有效, 如果存在规范推导 $S \xRightarrow{*} \delta A \omega \Rightarrow \delta \alpha \beta \omega$:
 - $\gamma = \delta \alpha$
 - a 是 ω 的第一个符号, 或者 ω 是空串, a 是 $\#$ 。

- Closure运算的新定义

- 考虑对活前缀 γ 有效的项目集中的项目 $[A \rightarrow \alpha \cdot B \beta, a]$ 必定存在一个最右推导 $S \xRightarrow{*} \delta A \alpha x \Rightarrow \delta \alpha B \beta \alpha x$, 其中 $\gamma = \delta \alpha$
- 假设 $\beta \alpha x$ 能推导出终结符串 bw , 那么对每一个形如 $B \rightarrow \eta$ 的产生式, 存在推导 $S \xRightarrow{*} \gamma B b w \Rightarrow \gamma \eta b w$, 于是 $[B \rightarrow \cdot \eta, b]$ 对 γ 有效, 其中 b 是 $\text{FIRST}(\beta \alpha x)$ 中的任何终结符。根据 FIRST 的定义, $\text{FIRST}(\beta \alpha x) = \text{FIRST}(\beta a)$

假定 I 是文法 G' 的任一项目集, 构造 I 的闭包 $\text{CLOSURE}(I)$ 的方法是:

- I 的任何项目都属于 $\text{CLOSURE}(I)$;
- 若 $[A \rightarrow \alpha \cdot B \beta, a]$ 属于 $\text{CLOSURE}(I)$, $B \rightarrow \eta$ 是一个产生式, 那么, 对 $\text{FIRST}(\beta a)$ 中的每个终结符 b , 将 $[B \rightarrow \cdot \eta, b]$ 加入 $\text{CLOSURE}(I)$;
- 重复上述两步骤直至 $\text{CLOSURE}(I)$ 不再增大为止。

函数值 $\text{GO}(I, X)$ 定义为 $\text{GO}(I, X) = \text{CLOSURE}(J)$, 其中 $J = \{\text{任何形如 } [A \rightarrow \alpha X \cdot \beta, a] \text{ 的项目} \mid [A \rightarrow \alpha \cdot X \beta, a] \text{ 属于 } I\}$

■ 构造FIRST集合的算法 I

对每一个文法符号 $X \in V_T \cup V_N$ 构造 $\text{FIRST}(X)$: 应用下列规则,直到每个集合 FIRST 不再增大为止.

(1)如果 $X \in V_T$,则 $\text{FIRST}(X) = \{X\}$.

(2)如果 $X \in V_N$,且有产生式 $X \rightarrow a \dots$,则把 a 加入到 $\text{FIRST}(X)$ 中;若 $X \rightarrow \varepsilon$ 也是一个产生式,则把 ε 加入到 $\text{FIRST}(X)$ 中.

(3)如果 $X \rightarrow Y \dots$ 是一个产生式且 $Y \in V_N$,则把 $\text{FIRST}(Y) \setminus \{\varepsilon\}$ 加到 $\text{FIRST}(X)$ 中;

如果 $X \rightarrow Y_1 Y_2 \dots Y_k$ 是一个产生式, $Y_1, \dots, Y_{i-1} \in V_N$,而且对任何 $j, j \in [1, i-1]$, $\varepsilon \in \text{FIRST}(Y_j)$, (即 $Y_1 Y_2 \dots Y_{i-1} \xRightarrow{*} \varepsilon$), 则把 $\text{FIRST}(Y_i) \setminus \{\varepsilon\}$ 加到 $\text{FIRST}(X)$ 中; 特别是, 若所有的 $\text{FIRST}(Y_j)$ 均含有 ε , $j=1, 2, \dots, k$, 则把 ε 加到 $\text{FIRST}(X)$ 中.

■ 构造FIRST集合的算法II

对文法G的任何符号串 $\alpha = X_1X_2 \dots X_n$ 构造集合**FIRST(α)**

(1)首先置**FIRST(α) = FIRST(X_1) \setminus \{\epsilon\}**.

(2)如果对任何j, $j \in [1, i-1]$, $\epsilon \in \text{FIRST}(X_j)$, 则把**FIRST(X_i) \setminus \{\epsilon\}**加入到**FIRST(α)**中. 特别是, 若所有的**FIRST(X_j)**均含有 ϵ , $j=1, 2, \dots, n$, 则把 ϵ 加到**FIRST(α)**中.

■ LR(1)分析表构造举例

文法G:

(0) $S' \rightarrow S$

(1) $S \rightarrow CC$

(2) $C \rightarrow cC$

(3) $C \rightarrow d$

■ LR(1)分析表构造举例

文法G:

(0) $S' \rightarrow S$

(1) $S \rightarrow CC$

(2) $C \rightarrow cC$

(3) $C \rightarrow d$

$I_0: S' \rightarrow \bullet S, \quad \#$

$S \rightarrow \bullet CC, \quad \#$

$C \rightarrow \bullet cC, \quad c/d$

$C \rightarrow \bullet d, \quad c/d$

$I_1: S' \rightarrow S\bullet, \quad \#$

$I_2: S \rightarrow C\bullet C, \quad \#$

$C \rightarrow \bullet cC, \quad \#$

$C \rightarrow \bullet d, \quad \#$

$I_3: C \rightarrow c\bullet C, \quad c/d$

$C \rightarrow \bullet cC, \quad c/d$

$C \rightarrow \bullet d, \quad c/d$

$I_4: C \rightarrow d\bullet, \quad c/d$

$I_5: S \rightarrow CC\bullet, \quad \#$

$I_6: C \rightarrow c\bullet C, \quad \#$

$C \rightarrow \bullet cC, \quad \#$

$C \rightarrow \bullet d, \quad \#$

$I_7: C \rightarrow d\bullet, \quad \#$

$I_8: C \rightarrow cC\bullet, \quad c/d$

$I_9: C \rightarrow cC\bullet, \quad \#$

$GO(I_0, S) = I_1;$

$GO(I_0, C) = I_2;$

$GO(I_0, c) = I_3;$

$GO(I_0, d) = I_4;$

$GO(I_2, C) = I_5;$

$GO(I_2, c) = I_6;$

$GO(I_2, d) = I_7;$

$GO(I_3, C) = I_8;$

$GO(I_3, c) = I_3;$

$GO(I_3, d) = I_4;$

$GO(I_6, C) = I_9;$

$GO(I_6, c) = I_6;$

$GO(I_6, d) = I_7;$

■ 规范LR语法分析表的构造

如果该表无冲突，那么称该表为G的规范LR(1)分析表。具有规范的LR(1)分析表的文法称为一个LR(1)文法。

• 步骤

- 构造拓广文法G'的LR(1)项目集规范族 $C=\{I_0, I_1, \dots, I_n\}$
- 从 I_k 构造语法分析器的状态k，状态k的分析动作如下：
 - 如果 $[A \rightarrow \alpha \cdot a \beta, b]$ 在 I_k 中，且 $GO(I_k, a) = I_j$ ，则置 $action[k, a]$ 为 s_j ，即“移动(j, a)进栈”，这里要求a必须是终结符
 - 如果 $[A \rightarrow \alpha \cdot, a]$ 在 I_k 中，则置 $action[k, a]$ 为 r_j ，即按照 r_j 归约，其中j是产生式 $A \rightarrow \alpha$ 的序号
 - 如果 $[S' \rightarrow S \cdot, \#]$ 在 I_k 中，则置 $action[k, \#]$ 为acc，表示接受
 - 状态k的转移按照下面的方法确定：如果 $GO(I_k, A) = I_j$ ，那么 $goto[k, A] = j$
 - 其余表项设为出错
- 初始状态是包含 $[S' \rightarrow \cdot S, \#]$ 的项目集构造出的状态。

■ LR(1)分析表构造举例

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		

$I_0: S' \rightarrow \bullet S, \#$
 $S \rightarrow \bullet CC, \#$
 $C \rightarrow \bullet cC, c/d$
 $C \rightarrow \bullet d, c/d$
 $I_1: S' \rightarrow S \bullet, \#$
 $I_2: S \rightarrow C \bullet C, \#$
 $C \rightarrow \bullet cC, \#$
 $C \rightarrow \bullet d, \#$
 $I_3: C \rightarrow c \bullet C, c/d$
 $C \rightarrow \bullet cC, c/d$
 $C \rightarrow \bullet d, c/d$
 $I_4: C \rightarrow d \bullet, c/d$
 $I_5: S \rightarrow CC \bullet, \#$

$I_6: C \rightarrow c \bullet C, \#$
 $C \rightarrow \bullet cC, \#$
 $C \rightarrow \bullet d, \#$
 $I_7: C \rightarrow d \bullet, \#$
 $I_8: C \rightarrow cC \bullet, c/d$
 $I_9: C \rightarrow cC \bullet, \#$

$GO(I_0, S) = I_1;$
 $GO(I_0, C) = I_2;$
 $GO(I_0, c) = I_3;$
 $GO(I_0, d) = I_4;$
 $GO(I_2, C) = I_5;$
 $GO(I_2, c) = I_6;$
 $GO(I_2, d) = I_7;$

$GO(I_3, C) = I_8;$
 $GO(I_3, c) = I_3;$
 $GO(I_3, d) = I_4;$
 $GO(I_6, C) = I_9;$
 $GO(I_6, c) = I_6;$
 $GO(I_6, d) = I_7;$

文法G:
 (0) $S' \rightarrow S$
 (1) $S \rightarrow CC$
 (2) $C \rightarrow cC$
 (3) $C \rightarrow d$

■ 三种LR分析法的对比

分析法	项目	项目集规范族	Action表 (遇规约项目)
LR(0)	LR(0)	Closure+GO函数	整行填rj
SLR	LR(0)	Closure+GO函数	看Follow集合
LR(1)	LR(1)	Closure+GO函数 Closure 第二个分量算 First(βa)	看项目的第二个分量

■ 二义性文法在LR分析中的应用

• 定理：任何二义性文法不是LR文法，因而也不是SLR或LALR文法

• 算术表达式的二义性文法

• $E \rightarrow E + E \mid E * E \mid (E) \mid id$

• 对应的非二义性文法为

• $E \rightarrow E + T \mid T$
 • $T \rightarrow T * F \mid F$
 • $F \rightarrow (E) \mid id$

$I_7: E \rightarrow E + E \bullet$

$E \rightarrow E \bullet + E$

$E \rightarrow E \bullet * E$

$I_8: E \rightarrow E * E \bullet$

$E \rightarrow E \bullet + E$

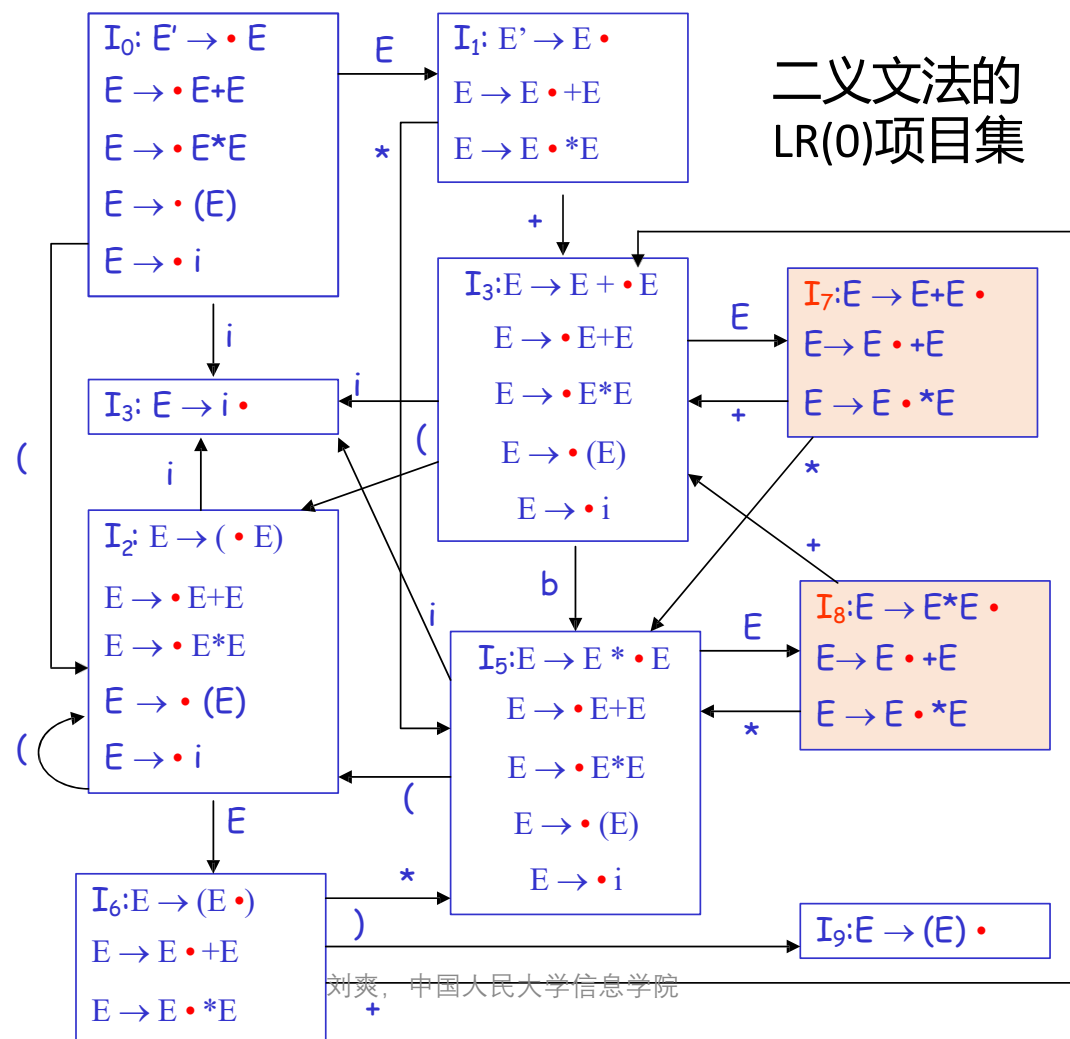
$E \rightarrow E \bullet * E$

对于某些二义文法，
 可以人为地给出优先
 性和结合性的规定，
 从而可以构造出比相
 应非二义性文法更优
 越的LR分析器

二义性文法在LR分析中的应用

文法:

- (1) $E \rightarrow E+E$
- (2) $E \rightarrow E * E$
- (3) $E \rightarrow (E)$
- (4) $E \rightarrow i$



■ 二义性文法在LR分析中的应用

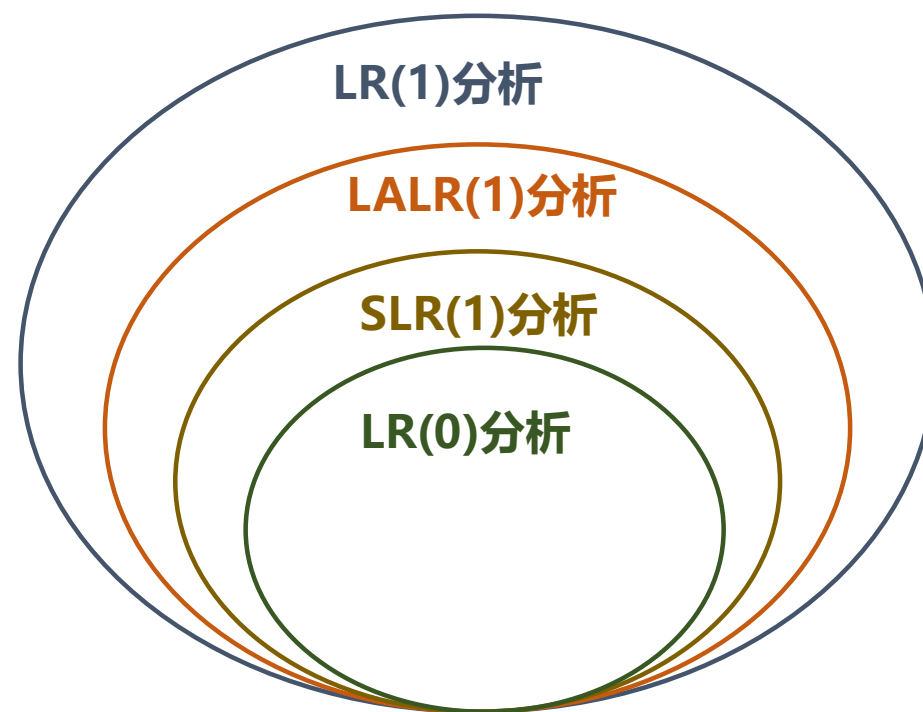
- 定义*优先于+; *、+ 服从左结合，得到二义文法的LR分析表

状态	ACTION						GOTO
	i	+	*	()	#	E
0	S ₃			S ₂			1
1		S ₄	S ₅			Acc	
2	S ₃			S ₂			6
3		r ₄	r ₄		r ₄	r ₄	
4	S ₃			S ₂			7
5	S ₃			S ₂			8
6		S ₄	S ₅		S ₉		
7		r ₁	S ₅		r ₁	r ₁	
8		r ₂	r ₂		r ₂	r ₂	
9		r ₃	r ₃		r ₃	r ₃	

■ LR语法分析中的错误恢复

- 在LR分析过程中，当我们处在这样一种状态下，即输入符号既不能移入栈顶，栈内元素又不能归约时，就意味着发现语法错误。
- 处理的方法分为两类：
 - 第一类多半是用插入、删除或修改的办法。如果不能使用这种办法，则采用第二类办法，
 - 第二类办法包括在检查到某一不合适的短语时，采用局部化的方法进行处理。类似前面讲过的同步符号

■ LR分析

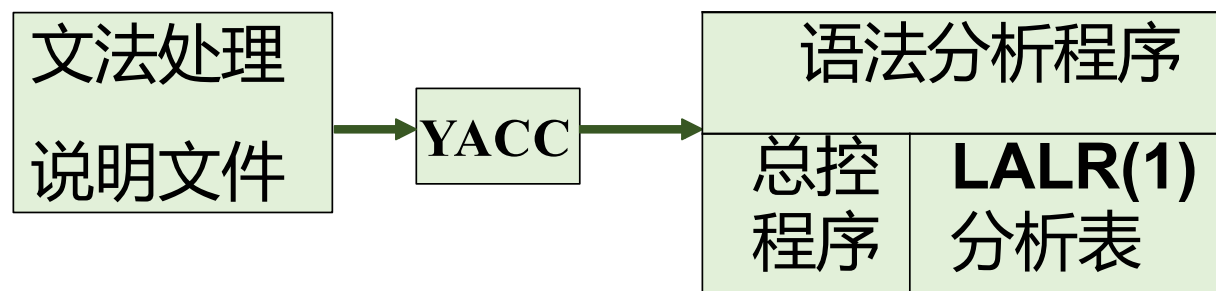


■ Outline

- 自底向上的语法分析基本问题
 - 移动 – 归约分析法
 - 用栈实现移动归约分析
- 算符优先分析法
 - 算符优先分析法定义、优先分析表的确定、优先函数的定义
 - 使用算符优先关系进行分析
 - 算符优先分析中的错误恢复
- LR分析法
- 语法分析器的自动产生工具Yacc

■ 语法分析器自动产生工具Yacc

- 软件工具Yacc是编译程序自动生成器, 是在UNIX中运行的一个实用程序。该程序读用户提供的关于语法分析器的规格说明, 基于LALR(1)语法分析的原理, 自动构造一个 语法分析器; 并且能根据规格说明中给出的语义动作建立规定的翻译。



■ Yacc语言程序的组成

- Yacc语言程序,与LEX规格说明类似,由说明部分、翻译规则和辅助过程组成.
- 各部分之间用双百分号分隔即:

说明部分---可有可无 (包括变量说明,标识符常量说明和正规定义.)

%%

翻译规则 (必须有)

%%

P1 {动作1}

P2 {动作2}

.....

Pn {动作n}

辅助过程---可有可无 (对翻译规则的补充,翻译规则里某些动作需要调用过程,如C语言的库程序.)

■ Yacc例子

- 其中**digit**表示**0...9**的数字,按如下文法写出的**Yacc**规格说明如下:

$E \rightarrow E + T | T$

$T \rightarrow T * F | F$

$F \rightarrow (E) | \text{digit}$

```
(1)  %{  
(2)      # include    <ctype.h>  
(3)      %}  
(4)      token DIGIT (5)  
      %%
```

第(1)-(4)行是说明部分,其中包括可供选择的两部分.

用%{和%}括起来的部分是C 语言程序的正规说明,可以说明翻译规则和辅助过程里使用的变量和函数的类型.

这里用第(2)行的蕴含控制行代替全部说明,具体内容在 ctype.h 里,第(4)行指出DIGIT是token类型的词汇,供后面使用.

■ Yacc例子

```
(6) line      : expr '\n'    { printf("%d\n", $1); }
(7)           ;
(8) expr      : expr '+' term    { $$ = $1 + $3; }
(9)           | term
(10)          ;
(11) term     : term '*' factor  { $$ = $1 * $3; }
(12)          | factor
(13)          ;
(14) factor   : '(' expr ')'     { $$ = $2 ; }
(15)          | DIGIT
(16)          ;
(17)%%
```

第(6)-(16)行是翻译规则,每条规则由 文法的产生式和相关的语义动作组成.

形如: 左部→右部1|右部2|…|右部n 的产生式,在YACC规格说明里写成

```
左部:   右部1 {语义动作1}
        |   右部2 {语义动作2}
        .....
        |   右部n {语义动作n}
        ;
```

在YACC里,用单引号括起来的单个字符看成是终结符号。语义动作是C语言的语句序列。语义动作中,\$\$表示和左部非终结符相关的属性值,\$1表示和产生式第一个文法符号相关的属性值,例如: 语义动作
line: expr '\n' { printf("%d\n", \$1); }

■ Yacc例子

```
(18) yylex( ) {  
(19)         int    c ;  
(20)         c = getchar ( );  
(21)         if ( isdigit ( c ))      {  
(22)             yylval=c-'0';  
(23)             return DIGIT;  
(24)         }  
(25)         return    c;  
(26)     }
```

(18)-(26)行是辅助过程,每个辅助过程都是C语言的函数,对翻译规则的补充,并且,其中**必须**包含名为yylex的**词法分析器**。调用名为yylex()的函数得到一个词汇,该词汇包括两部分的属性,其中值通过Yacc定义的全程变量yylval传递给语法分析器,返回词汇的属性。
(22) yylval=c-'0'; yylex()返回词汇的值。
(23) yylex()返回种别DIGIT;
第(25)行,除了数之外的任何字符, yylex()返回该字符本身。

■ Yacc处理冲突的规则

- 1)产生“归约-归约”冲突时,按照规格说明中产生式的排列顺序,选择排列在前面的产生式进行归约.
- 2)当产生“移进-归约”冲突时,选择执行移进动作.
- 3) Yacc在规格说明部分里,可以规定**终结符号**的优先顺序和结合性.
 - 优先顺序: 按说明终结符的次序,后说明的具有最高的优先顺序. “%prec”说明其后的终结符具有最高的优先顺序.
 - 结合性: “%right”(右结合),
“%left” (左结合),
“%nonassoc”(不具有结合性)
- 4) 应用这些机制,对二义文法,用户可以提供 附加信息 Yacc可以解决冲突.

■ Summary

- 自底向上的语法分析基本问题
 - 移动 – 归约分析法
 - 用栈实现移动归约分析
- 算符优先分析法
 - 算符优先分析法定义、优先分析表的确定、优先函数的定义
 - 使用算符优先关系进行分析
 - 算符优先分析中的错误恢复
- LR分析法
 - LR(0)
 - SLR
 - LR(1) *
- 语法分析器的自动产生工具Yacc

阅读《程序设计语言编译原理》 第三版 第5章