Lecture 5: Bottom-up Analysis

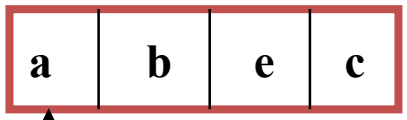
Xiaoyuan Xie 谢晓园
xxie@whu.edu.cn
计算机学院E301

5.1 Motivation Example

5.1 Motivation example

- **自顶向下分析回顾：**

P:
(1) Z → aBeA
(2) A → Bc
(3) B → d
(4) B → bB
(5) B → ε



读头 从左向右移动读头，读入字符串

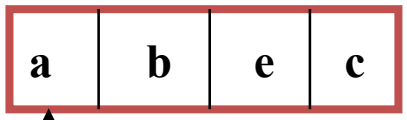
输入（读头所在）	栈内符号	分析	执行推导	匹配
abec	Z	Z的哪一个产生式以a开头？-(1)	aBeA	a
bec	BeA	B的哪一个产生式以b开头？-(4)	bBeA	b
ec	BeA	B的哪一个产生式能够以e开头？-(5)	εeA	e
c	A	A的哪一个产生式能够以c开头？-(2) (5)		

5.1 Motivation example

- **自顶向下分析回顾(续):**

P:
(1) Z → aBeA
(2) A → Bc
(3) B → d
(4) B → bB
(5) B → ε

输入（读头所在）	栈内符号	分析	执行推导	匹配
c	A	A的哪一个产生式能够以c开头？-(2)	Bc	
c	Bc	A的哪一个产生式能够以c开头？-(5)	εc	c



读头　　从左向右移动读头，读入字符串

5.1 Motivation example

- 自底向上分析过程是从所给**输入串**出发，对其进行**最左归约**的过程。

P:

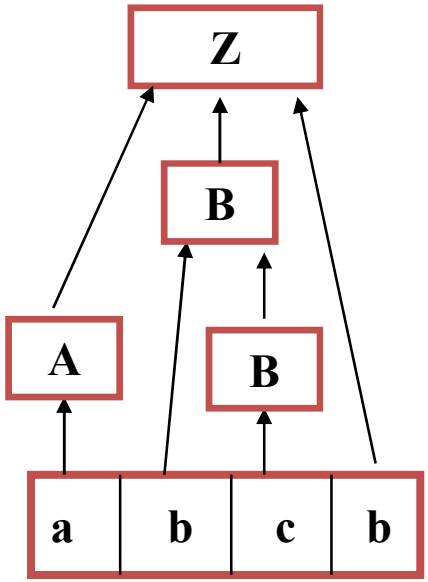
(1) $Z \rightarrow ABb$
(2) $A \rightarrow a$
(3) $A \rightarrow b$
(4) $B \rightarrow b$
(5) $B \rightarrow c$
(6) $B \rightarrow bB$

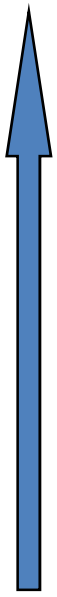
a	b	c	b

符号栈	输入	动作
	abcb	移入
a	bcb	归约(2)
A	bcb	移入
Ab	cb	移入
Abc	b	归约(5)
AbB	b	归约(6)
AB	b	移入
ABb		归约(1)
Z		成功

5.1 Motivation example

- 自底向上归约的过程也是自底向上构建语法树的过程





归约过程

自底向上分析中归约过程的逆过程就是该句子的最右推导;

abcb
---> Abcb
---> AbBb
---> ABb
---> Z

(1) $Z \rightarrow ABb$
(2) $A \rightarrow a$
(3) $A \rightarrow b$
(4) $B \rightarrow b$
(5) $B \rightarrow c$
(6) $B \rightarrow bB$

$Z \Rightarrow_{rm} A\underline{B}b$
$\Rightarrow_{rm} Ab\underline{B}b$
$\Rightarrow_{rm} \underline{A}bcb$
$\Rightarrow_{rm} abcb$

5.1 Motivation example

- **从推导的角度看，语法分析的任务是**
  - 接受一个终结符号串作为输入，找出从文法的开始符号推导出这个串的方法
  - 自顶向下分析

5.1 Motivation example

- **从规约的角度看，语法分析的任务是**
  - 将一个串*w*归约为文法符号的过程
    - 在每一步的归约中，一个与某产生式RHS相匹配的特定子串（**可规约子串**）被替换为该产生式LHS的非终结符号，一次归约实质上是一个推导的反向操作
  - 为一个输入串构造语法分析树的过程
    - 从叶子（输入串中的终结符号，将位于分析树的底端）开始，向上到达根结点
  - 自底向上分析
  - **再看一个例子（example-1）**

5.1 Motivation example

- **分析动作:移入(shift),归约(reduce)**
- **包含以下方法:**
  - 简单优先法; 算符优先法; LR 类的方法;
- **关键问题:**
  - **什么时候进行归约（选择那部分进行规约），按照哪条产生式进行归约**

5.2 自底向上分析法概述

5.2 自底向上分析法概述

- **如何实现?**
  - 让我们看一个例子： “推导 v.s. 规约”

5.3 自底向上分析法概述

分析

- 被归约的对象一定是某一产生式 RHS 的文法符号串 α，将 α 看成是 $T \cup N$ 为字母表的一个正则表达式，$\beta \in (T \cup N)^*$ 是要归约的对象，则在 β 中寻找 α 子串可以用自动机解决.
- 被归约对象之后一定是终结符号串：

推导过程：

$$\begin{aligned} A &\underset{rm}{\overset{*}{\Rightarrow}} \alpha \underline{B} w \\ &\underset{rm}{\Rightarrow} \alpha\beta w \end{aligned}$$

归约过程：

$$\begin{aligned} & \alpha\beta w \\ \underset{rm}{\overset{*}{\Leftarrow}} \quad & \alpha \underline{B} w \end{aligned}$$

- 不确定性：
  - exp PLUS term TIMES ID 可以归约的对象有：

1) exp PLUS term TIMES <u>ID</u>
2) exp PLUS <u>term</u> TIMES ID
3) <u>exp PLUS term</u> TIMES ID

但是只有 (1) 是唯一一个正确的.

5.2 自底向上分析法概述

- **分析**
  - 关键在于寻找“正确的待规约对象”，即它能保证被归约后一定还保持着最右句型 --- 称为“句柄”

1) exp PLUS term TIMES ID $\underset{rm}{\Leftarrow}$ exp PLUS term TIMES fac

2) exp PLUS term TIMES ID $\underset{rm}{\Leftarrow}$ exp PLUS exp TIMES ID

3) exp PLUS term TIMES ID $\underset{rm}{\Leftarrow}$ exp TIMES ID

  - 这里只有1）正确
- **下面的内容将围绕“句柄的定义”，“如何找到句柄”展开**

5.2 自底向上分析法概述：定义

- **短语**　一个短语就是可以被规约为一个非终结符的串
  - 一个句型形如αβγ, 如果存在一个句型αAγ,而且 A⇒$^{+}$β, 则称β为句型αβγ的*短语*;
  - 例如句型AbBb，则bB,AbBb是它的短语，因为
    - 存在句型ABb，ABb ⇒ AbBb，α = A，γ = b;
    - 存在句型Z，Z ⇒ ABb ⇒ AbBb ，　α = ε，　γ = ε;

(1) Z → ABb
(2) A → a
(3) A → b
(4) B → d
(5) B → c
(6) B → bB

- **简单短语**
  - 一个句型形如αβγ, 如果存在一个句型αAγ,而且 A ⇒ β, 则称β为句型αβγ的*简单短语*;

例如句型AbBb，bB是它的简单短语，AbBb不是它的简单短语

如果是一步规约，就是简单短语

5.2 自底向上分析法概述：定义

- **句柄:一个句型的简单短语可能有多个，称*最左简单短语*为句柄 (handler);**
  - 例如：句型abBb，简单短语有两个：a,bB
    - $Z \Rightarrow ABb \Rightarrow aBb \Rightarrow abBb$
  - 最左简单短语a是句柄
- **句柄的唯一性:**
  - 如果一个CFG无二义性,则它的任意一个句型都有唯一的句柄;

5.2 自底向上分析法概述：定义

- **例子**

P:
(1) E → T
(2) E → E + T
(3) T → F
(4) T → T * F
(5) F → (E)
(6) F → i
(7) F → n

句型：　E+(E+T*F)*i

句型的一个推导:
E ⇒ E + T ⇒ E + T*F⇒E+T*i ⇒
E+F*i ⇒ E+(E)*i ⇒ E+(E+T)*i
⇒ E+(E+T*F)*i

短语：E+(E+T*F)*i、(E+T*F)*i 、(E+T*F) 、E+T*F 、T*F、i

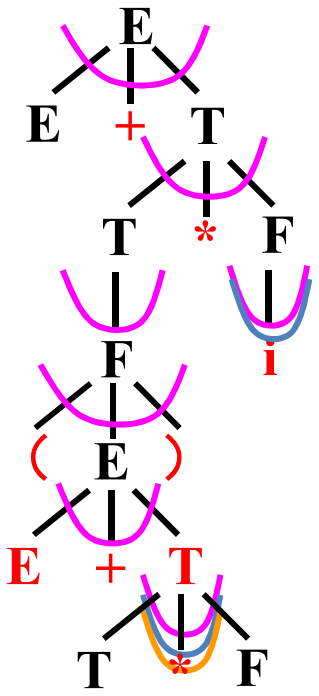
简单短语: T*F、i

句柄: T*F

通过为所给句型建立语法分析树，可以很容易地识别出该句型的所有短语、简单短语和句柄。

5.2 自底向上分析法概述：定义

- **由语法分析树识别短语、简单短语和句柄**



语法分析树的叶子节点构成句型：
E+(E+T*F)*i

每棵子树的叶子节点构成短语：
E+(E+T*F)*i、(E+T*F)*i 、(E+T*F) 、
E+T*F 、T*F、i

每棵简单子树(只有一层的子树)的叶子节点构成简单短语：T*F、i

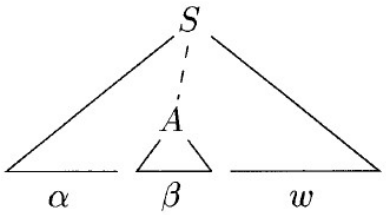
最左简单子树的叶子节点构成句柄： T*F

5.2 自底向上分析法概述：从规约的角度理解

- **换句话说，β是最右句型γ的一个句柄，须满足：**
  - 存在产生式A→β，且
  - 将β规约为A之后得到的串是从开始符合推出γ的某个最右推导序列中出现在位于γ之前的最右句型
- **通俗地说**
  - 句柄是最右推导的反向过程中被规约的那些部分
- **句柄的作用**
  - 对句柄的规约，代表了相应的最右推导中的一个反向步骤

5.2 自底向上分析法概述：从规约的角度理解

- **Bottom-Up Parsing – 归约过程 – 反向最右推导过程 – 句柄剪枝过程**
- **句柄：和某个产生式体相匹配的子串，对它的归约代表了相应的最右推导的一个反向步骤**
- **句柄的形式定义：$S \overset{*}{\underset{rm}{\Rightarrow}} \alpha Aw \underset{rm}{\Rightarrow} \alpha\beta w$，那么紧跟$\alpha$的$\beta$可以一步归约到$A$。 w是所有的终结符号串。**
- **注意：**
  - 归约前后都是最右句型
  - 和某个产生式RHS匹配的最左子串不一定是句柄



5.2 自底向上分析法概述：从规约的角度理解

- 通过“句柄剪枝”可以得到一个反向的最右推导。从句子w开始，令$w=\gamma_n$，γ_n是未知最右推导的第n个最右句型

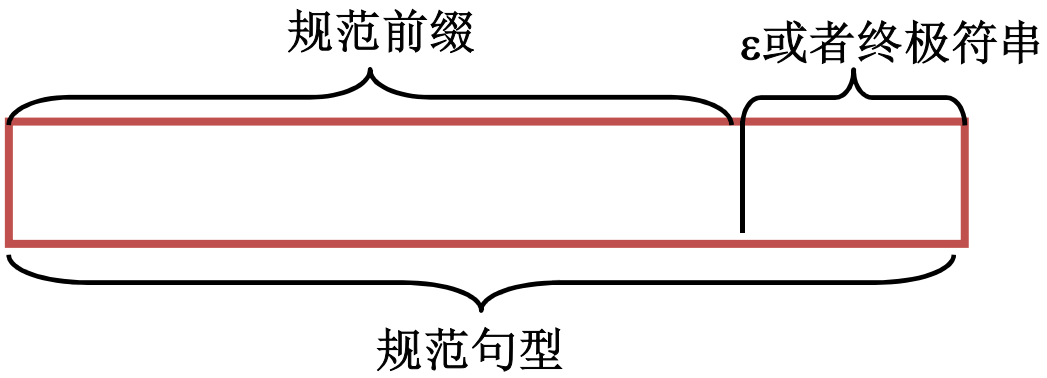
$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

- 以相反的顺序重构这个推导（实际上是重构归约序列），我们在γ_n中寻找句柄β_n，并将替换为相应产生式$A \rightarrow \beta_n$的头部，得到前一个最右句型γ_{n-1}；重复这个过程，直到S
- 和推导类似，如果能重现这个序列，则可以完成语法分析任务（其中可能存在什么问题）
  - 如何找到句柄

5.2 自底向上分析法概述：定义

- **为了帮助识别句柄，我们定义“活前缀/可行前缀”概念**
  - 规范前缀: 一个规范句型的一个前缀称为*规范前缀*, 如果该前缀后面的符号串不包含非终极符;
    - $Z \Rightarrow ABb$：规范前缀为 AB, ABb
    - $Z \Rightarrow^{+} Acb$：规范前缀为 A, Ac, Acb

Why



5.2 自底向上分析法概述：定义

- **为了帮助识别句柄，我们定义“活前缀”概念**
  - 规范活前缀 (viable prefix):满足如下条件之一的规范前缀称为规范活前缀:
    - 该规范前缀不包含简单短语; 或者
    - 该规范前缀只包含一个简单短语,而且是在该规范前缀的最后(这个简单短语就是句柄);
  - 换句话说，规范活前缀不含句柄之后的任何符号

(1) $Z \rightarrow ABb$
(2) $A \rightarrow a$
(3) $A \rightarrow b$
(4) $B \rightarrow d$
(5) $B \rightarrow c$
(6) $B \rightarrow bB$

$Z \Rightarrow ABb$
规范前缀为 AB, ABb
规范活前缀: AB(不包含简单短语)
ABb(包含一个简单短语且在最后)

有何用意?

5.2 自底向上分析法概述：定义

- **规范活前缀 例2：**

(1) Z → ABb
(2) A → a
(3) A → b
(4) B → d
(5) B → c
(6) B → bB

$Z \Rightarrow^+ abcb$

规范前缀为 a, ab, abc, abcb

规范活前缀：a (包含一个简单短语且在最后)

ab, abc, abcd都不是规范活前缀

识别活前缀是假，识别句柄是真

5.2 自底向上分析法概述：定义

- **活前缀有两种类型**
  - 非归态(移入)活前缀：规范活前缀不包含简单短语，而且是在该规范活前缀的最后句柄尚未形成，需要继续移进若干符号后才能形成句柄
  - 归态活前缀：只包含一个简单短语，尾部正好是句柄之尾，此时可以进行规约。规约后又成为另一句型的活前缀

$Z \Rightarrow ABb$　　规范前缀为 AB, ABb

规范活前缀: AB(不包含简单短语) --- 移入型规范活前缀

ABb(包含一个简单短语) --- 归约规范活前缀

与自底向上分析（识别句柄）有何关系?

5.2 自底向上分析法概述：移进归约语法分析框架

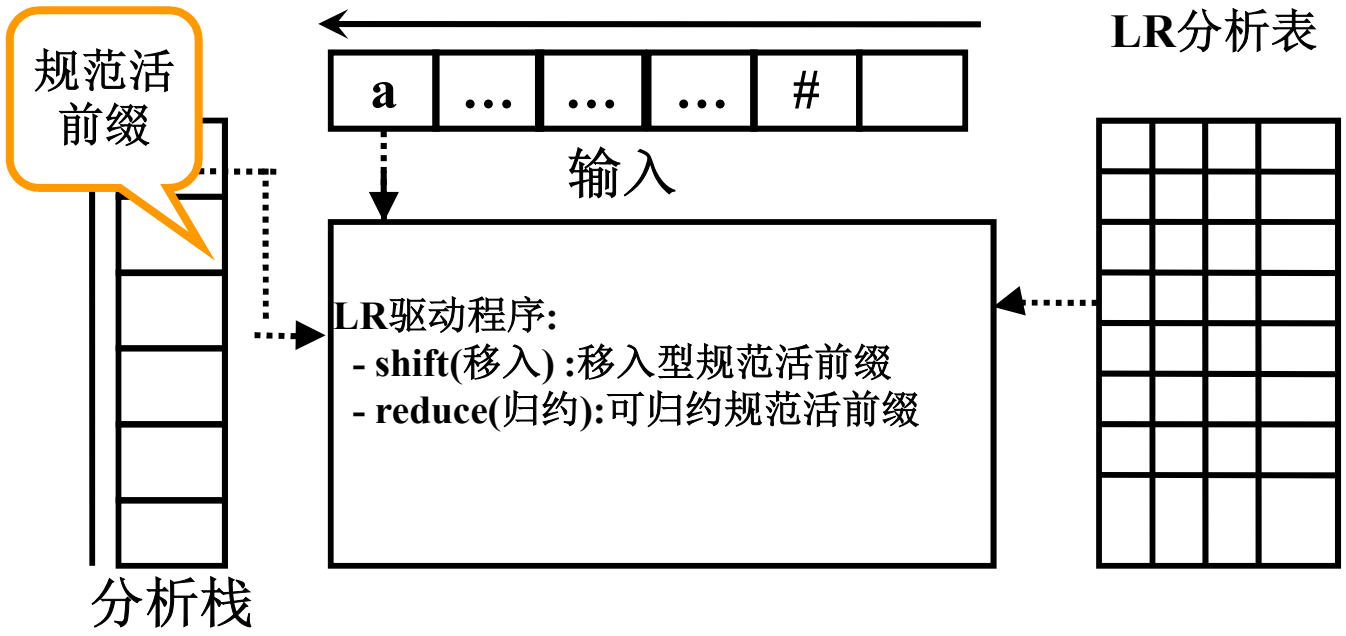
- 自底向上的分析：使用一个栈保存文法符号，一个输入缓冲区存放将要进行分析的剩余符号
  - 初始栈： $　　初始输入 *w*$
- 对输入串的一次从左到右的扫描过程中，语法分析器将零个或多个输入符号移入到栈的顶端，直到它可以对栈顶的一个文法符号串进行归约为止（找到归态活前缀）
  - 它将归约为某个产生式的头。
- 不断重复这个循环，直到它检测到一个语法错误，或者栈中包含了开始符号且输入缓冲区为空。
  - 栈中符号（从底向上）和待扫描的符号组成了一个最右句型
  - 开始时刻：栈中只包含$，而输入为w$
  - 成功结束时刻：栈中$S，而输入$

自顶向下的非递归预测分析中栈是如何工作的？

5.2 自底向上分析法概述: 移进归约语法分析框架

- **主要分析动作**
  - 移入：将下一个输入符号移动到栈顶
  - 归约：将句柄归约为相应的非终结符号
    - **句柄总是在栈顶（与其前面的串形成归态活前缀）**
    - 具体操作时弹出句柄，压入被归约到的非终结符号
  - 接受：宣布分析过程成功完成
  - 报错：发现语法错误，调用错误恢复子程序

5.2 自底向上分析法概述：移进/规约分析法



5.2 自底向上分析法概述: 归约分析过程的例子

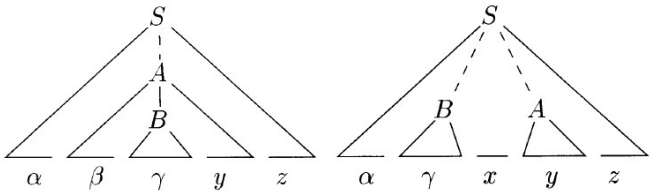
栈	输入	动作
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	移入
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	按照 $F \to \mathbf{id}$ 归约
$ F	$* \mathbf{id}_2$ $	按照 $T \to F$ 归约
$ T	$* \mathbf{id}_2$ $	移入
$ $T *$	$\mathbf{id}_2$ $	移入
$ $T * \mathbf{id}_2$	$	按照 $F \to \mathbf{id}$ 归约
$ $T * F$	$	按照 $T \to T * F$ 归约
$ T	$	按照 $E \to T$ 归约
$ E	$	accept

图 4-28　一个移入－归约语法分析器在处理输入 $\mathbf{id}_1 * \mathbf{id}_2$ 时经历的格局

困难：何时规约，何时移进 --- 及时发现正确的归态活前缀，也即，正确识别句柄。

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid \mathbf{id}$$

5.2 自底向上分析法概述



1）$S \underset{rm}{\overset{*}{\Rightarrow}} \alpha Az \underset{rm}{\Rightarrow} \alpha\beta Byz \underset{rm}{\Rightarrow} \alpha\beta\gamma yz$

2）$S \underset{rm}{\overset{*}{\Rightarrow}} \alpha BxAz \underset{rm}{\Rightarrow} \alpha Bxyz \underset{rm}{\Rightarrow} \alpha\gamma xyz$

图 4-29　一个最右推导中两个连续步骤的两种情况

STACK	INPUT
$\$\alpha\beta\gamma$	$yz\$$
$\$\alpha\beta B$	$yz\$$
$\$\alpha\beta By$	$z\$$

STACK	INPUT
$\$\alpha\gamma$	$xyz\$$
$\$\alpha Bxy$	$z\$$

- 句柄总是出现在栈的顶端，绝不会出现在栈的中间
- 语法分析器进行一次归约以后，都必须接着移入零个或多个符号才能在栈顶找到下一个句柄
- 不需要去栈中间去寻找句柄

5.2 自底向上分析法概述

- **Let us revisit the viable prefix**
  - 见“移进规约分析与活前缀”

5.2 自底向上分析法概述: 移进/归约冲突

- **移入-归约技术并不能处理所有上下文无关文法**
- **某些上下文无关文法（比如二义性文法）**
  - 移入/归约冲突：栈中的内容和接下来的k个输入符号，都不能确定进行移入还是归约操作(不能确定是否是句柄)
  - 归约/归约冲突：存在多个可能的归约到不同非终结符号的归约(不能确定句柄归约到那个非终结符号)

5.2 自底向上分析法概述: 移进/归约冲突

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

| **if** *expr* **then** *stmt* **else** *stmt*

| **other**

栈	输入
··· **if** *expr* **then** *stmt*	**else** ··· $

规约为stmt?
还是移入else?

5.2 自底向上分析法概述: 归约/归约冲突

(1)	*stmt*	→	**id** (*parameter_list*)
(2)	*stmt*	→	*expr* := *expr*
(3)	*parameter_list*	→	*parameter_list* , *parameter*
(4)	*parameter_list*	→	*parameter*
(5)	*parameter*	→	**id**
(6)	*expr*	→	**id** (*expr_list*)
(7)	*expr*	→	**id**
(8)	*expr_list*	→	*expr_list* , *expr*
(9)	*expr_list*	→	*expr*

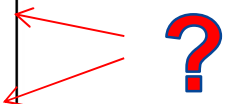


图 4-30　有关过程调用和数组引用的产生式

栈	输入
… **id** (**id**	, **id**) …

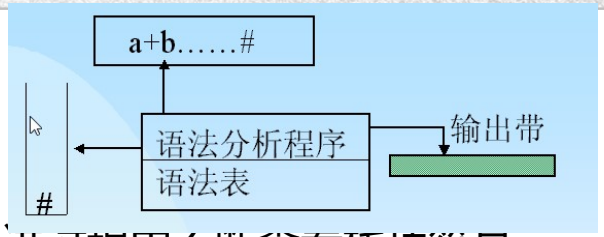
作业

- **教材P153: 4.5.2, 4.5.3**

5.3 LR Family

5.3 LR语法分析技术

- **LR 方法**
  - 工作方式：移进-规约
    - 自左至右将输入串符号一个个移进栈，移进过程中不断查看栈顶符号，一旦形成了某个句型的句柄，就将此句柄用相应产生式左部替换（规约）
    - 规约后若再形成句柄，则继续替换（规约）；直到栈顶不再形成句柄为止。然后继续移进符号，重复上面过程，直至栈顶只剩下文法的开始符号，输入串读完为止 --- 句子识别成功



5.3 LR语法分析技术

- **LR(k)的语法分析概念**
  - L表示最左扫描，R表示反向构造出最右推导
  - k表示最多向前看k个符号
- **当k的数量增大时，相应的语法分析器的规模急剧增大**
  - K=2时，程序设计语言的语法分析器的规模通常非常庞大
  - 当k=0、1时已经可以解决很多语法分析问题，因此具有实践意义
  - 因此，我们只考虑k<=1的情况

5.3 LR语法分析技术

- **LR 文法：**
  - 某一CFG能够构造一张分析表，使得表中每一元素至多只有一种明确动作，则称改文法为LR文法
  - 注：并非所有CFG都是LR文法，但对于多数程序设计语言，一般都可以用LR文法描述；和LL(1)相比，LR分析适应的文法范围要广一些
- **不同的 LR 方法**
  - 不同的LR方法对CFG的要求不一样，能够分析的CFG多少也不一样， $LR(0) \subseteq SLR(1) \subseteq LALR(1) \subseteq LR(1)$

5.3 LR语法分析技术

- **LR 关键问题**
  - 如何判定规范活前缀?
  - 如何判定移入型规范活前缀?
  - 如何判定归约规范活前缀以及用哪一个产生式归约?
  - 如何判定分析结束?
- **构造一个判定归约规范活前缀的自动机 -- LR自动机**
- **由LR自动机构造LR分析表指导LR分析**

5.3 LR语法分析器的优点

- **表格驱动**
  - 虽然手工构造表格工作量大，但表格可以自动生成
- **对于几乎所有的程序设计语言，只要写出上下文无关文法，就能够构造出识别该构造的LR语法分析器**
- **最通用的无回溯移入-归约分析技术，且和其它技术一样高效**
- **可以尽早检测到错误**
- **能分析的文法集合是LL(k)文法的超集**

5.4 LR(0) 分析

5.4 LR(0) 分析法

- **构造LR(0)自动机：目的识别活前缀**
  - LR(0) 项（项目item）：为每个产生式添加圆点，圆点只能出现在产生式的右部符号串的任意位置
    - 产生式S→aAc，有4个LR(0)项：
      - S→ •aAc
      - S → a • Ac
      - S → a A • c
      - S → a Ac •
    - 特别地，空产生式: S → ε, 对应LR(0)项S → •
    - 右部长度为n的产生式有n+1个项；

5.4 LR(0) 分析法

- **项的直观含义**
  - 项A → α • β表示已经扫描/归约到了α，并期望接下来的输入中经过扫描/归约得到β，然后把αβ归约到A
  - 如果β为空，表示我们可以把α归约为A
- **项也可以用一对整数表示**
  - (i,j)表示第i条规则，点位于右部第j个位置

5.4 LR(0) 分析法

- **移入-归约语法分析器如何知道何时该移入、何时该归约呢?**
  - LR语法分析器试图用一些**状态**来表明我们在移进归约语法分析过程中所处的位置，从而做出移入-归约决定
- **项的意义**
  - 指明在语法分析过程中的给定点上，我们已经看到了(分析到了)一个产生式的哪些部分。或者说，如果我们想用这个产生式进行归约，还需要看到哪些文法符号
- **项的集合（项集）对应于一个状态**

5.4 LR(0) 分析法

- **规范LR(0)项集族提供构建LR(0)自动机的基础**
  - LR(0)自动机可用于做出语法分析决定
  - LR(0)自动机中每个状态代表了LR(0)项集族中的一个项集

5.4 LR(0) 分析法

- **构造LR(0)自动机-增广**
  - 首先将文法进行增广，增加产生式S'→ S, 令S'→•S作为初态项
  - 为什么要拓广: 保证文法开始符号不出现在任何产生式右部
    - CFG可能会导致S出现在右部，但是为了避免混淆，保证开始符号不出现在右部 →每当识别出S，就确定完成句子识别
    - 例如S → aSb, S →c, 给定abc，c可以规约为S，结束分析?

5.4 LR(0) 分析法

- **构造以项为状态的自动机**
  - 开始状态S’ → • S
  - 转换
    - A → α • Bβ到B → • γ有一个ε转换
    - 从A → α • Xβ到A → αX • β有一个X转换
  - 接受状态：A → α.，即点在最后的项

5.4 LR(0) 分析法

- **构造以项为状态的自动机**
  - 圆点理解为栈内栈外分界线
  - 例如：S→aAb, A→XYZ

找句柄 →找归态活前缀

aXYZb如果加上dot, 就能立即发现句柄

例如，产生式A → XYZ对应四个项目

- A→•XYZ　预期要归约的句柄是XYZ，但都未进栈
- A→X•YZ　预期要归约的句柄是XYZ，仅X进栈
- A→XY•Z　预期要归约的句柄是XYZ，仅XY进栈
- A→X YZ•　已处于归态活前缀，XYZ可进行归约，这个项目是归约项目。

每个产生式都能获得n+1个项目
我们现在需要的是将所有项目串起来，形成一个完整的，可以识别活前缀的LR(0)自动机

5.4 LR(0) 分析法

- **构造LR(0)自动机**
  - IS 是一个LR(0)项目的集合;
  - X 是一个符号(终极符或非终极符);
  - $IS_{(X)}$表示项目集IS关于符号X的投影: $IS_{(X)} = \{S \rightarrow \alpha X \bullet \beta \mid S \rightarrow \alpha \bullet X\beta \in IS, X \in V_T \cup V_N\}$ ---IS(x)只对IS中圆点后面是X的项目起作用，所起的作用就是将圆点从X的前面移到X的后面

$$IS = \{A \rightarrow A \bullet Bb,\ B \rightarrow a \bullet,\ B \rightarrow b \bullet B,\ Z \rightarrow \bullet cB\}$$

$$X = B$$

$$IS_{(B)} = \{A \rightarrow AB \bullet b,\ B \rightarrow bB \bullet\}$$

5.4 LR(0) 分析法

构造LR(0)自动机

- CLOSURE(I)：I的项集闭包
  - 对应于DFA化算法的 ε-CLOSURE
- GOTO(I,X)：I的X后继
  - 对应于DFA化算法的MOVE(I,X)

5.4 LR(0) 分析法

构造LR(0)自动机

- IS 是 LR(0)项目的集合; CLOSURE(IS)也是一个LR(0)项目集合，按照下面的步骤计算:

  [1] 初始, CLOSURE(IS) = IS

  [2] 对于CLOSURE(IS)中没有处理的LR(0)项目，

  如果该项目的圆点后面是一个非终极符A，

  而且A的全部产生式是{A→α1, …, A→αn}

  则增加如下LR(0)项目到CLOSURE(IS)

  {A→ •α1, …, A→ •αn}

  [3] 重复[2]直到 CLOSURE(IS)收敛;

5.4 LR(0) 分析法

- **构造LR(0)自动机**
  - CLOSURE(IS)示例

$V_T = \{a, b, c\}$
$V_N = \{S, A, B\}$
$S = S$
P:
{ $S' \rightarrow S$
$S \rightarrow aAc$
$A \rightarrow ABb$
$A \rightarrow Ba$
$B \rightarrow b$
}

$IS = \{S' \rightarrow \bullet S\}$
$CLOSURE(IS) = \{S' \rightarrow \bullet S$,
$S \rightarrow \bullet aAc\}$

$IS = \{S \rightarrow a \bullet Ac\}$
$CLOSURE(IS)$
$= \{S \rightarrow a \bullet Ac,$
$A \rightarrow \bullet ABb,\ A \rightarrow \bullet Ba,$
$B \rightarrow \bullet b\}$

5.4 LR(0) 分析法

- 内核项：初始项S’ →•S、以及所有点不在最左边的项
- 非内核项：除了S’ →•S之外、点在最左边的项
- 由于表可能很庞大，实现算法时可以考虑只保存相应的非终结符号；甚至可以只保存内核项，而在要使用非内核项时调用CLOSURE函数重新计算

5.4 LR(0) 分析法

- **构造LR(0)自动机**
  - GOTO函数 （DFA中的弧，表示状态变迁）
    - IS 是 LR(0)项目集合;
    - X 是一个符号(V_T or V_N终极符或非终极符);
    - GOTO(IS, X) = CLOSURE($IS_{(x)}$)
  - 其含义是对任意项目集IS，由于IS中有A→α•Xβ项目，IS(x)中有A→αX•β项目，表示识别活前缀又移进了一个符号X

5.4 LR(0) 分析法

- **LR(0)自动机建立在规范LR(0)项集族上**
  - CFG G={V_T, V_N, S, P} 的LR(0)自动机
  - (∑，SS，S0，TS，Φ)
    - 是否需要 增广产生式 Z → S ?
    - ∑ = $V_T \cup V_N \cup \{\#\}$
    - S0 = CLOSURE(Z → •S)
    - SS
    - TS
    - Φ

    } 构造 LR(0)自动机的过程中逐步生成的

5.4 LR(0) 分析法

构造LR(0)自动机：完整算法

[1] 增广产生式 $Z \rightarrow S$

[2] $\Sigma = V_T \cup V_N \cup \{\#\}$

[3] S0 = CLOSURE($Z \rightarrow \bullet S$)

[4] SS = {S0}

[5] 对于SS中的每一个项目IS, 和每个符号X$\in \Sigma$,

计算IS' = GOTO(IS, X),

如果IS'不为空, 则 建立 IS $\xrightarrow{X}$ IS',

如果IS'不为空且IS'不属于SS,则把IS'加入SS;

[6] 重复[5]直到SS收敛;

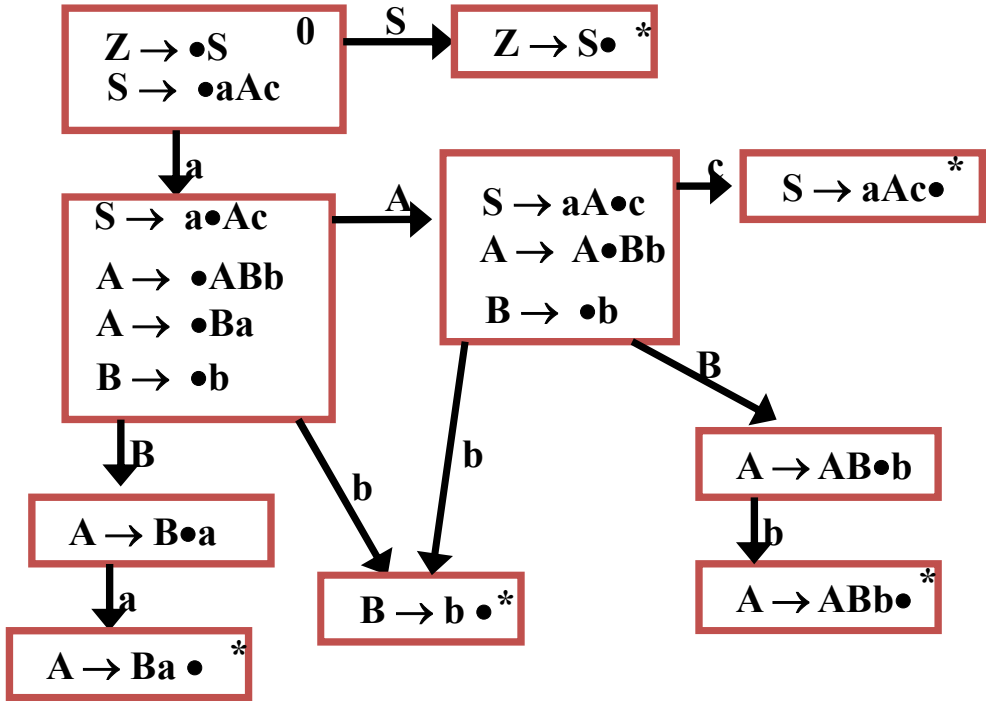
[7] 终止状态:包含形如A $\rightarrow \alpha\bullet$项目的项目集合;

5.4 LR(0) 分析法

■ 构造LR(0)自动机-例1

每个状态有什么含义?

V_T = {a, b, c}
V_N = {S, A, B}
S = S
P:
{ S→ aAc
A → ABb
A → Ba
B → b
}

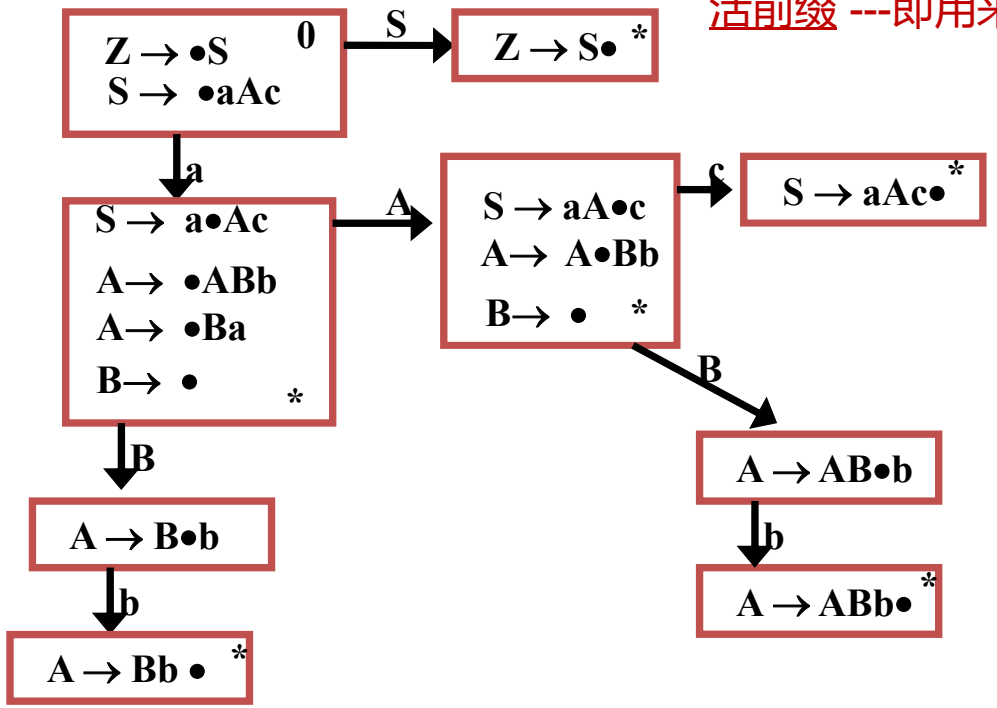


5.4 LR(0) 分析法

■ 构造LR(0)自动机-方法2，例2

一个CFG的LR(0)自动机接受的是该CFG的所有LR0归约规范活前缀 ---即用来识别活前缀

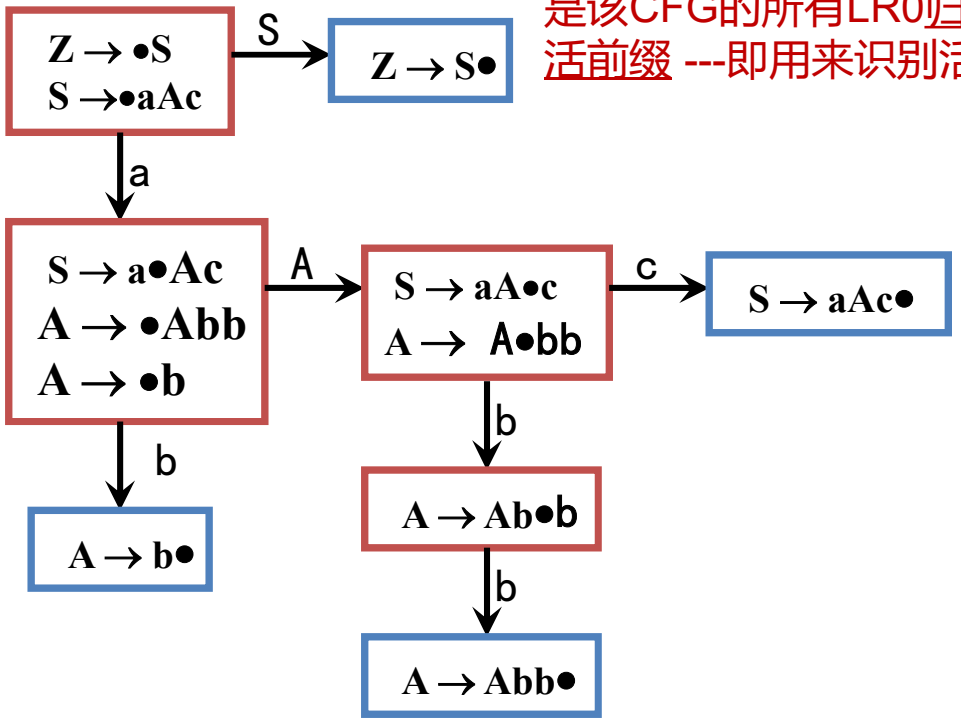
$V_T = \{a, b, c\}$
$V_N = \{S, A, B\}$
$S = S$
P:
{ $S \rightarrow aAc$
$A \rightarrow ABb$
$A \rightarrow Ba$
$B \rightarrow \varepsilon$
}



5.4 LR(0) 分析法

■ 构造LR(0)自动机-例3

$V_T = \{a, b, c\}$
$V_N = \{S, A\}$
$S = S$
P:
{ $S \rightarrow aAc$
$A \rightarrow Abb$
$A \rightarrow b$
}



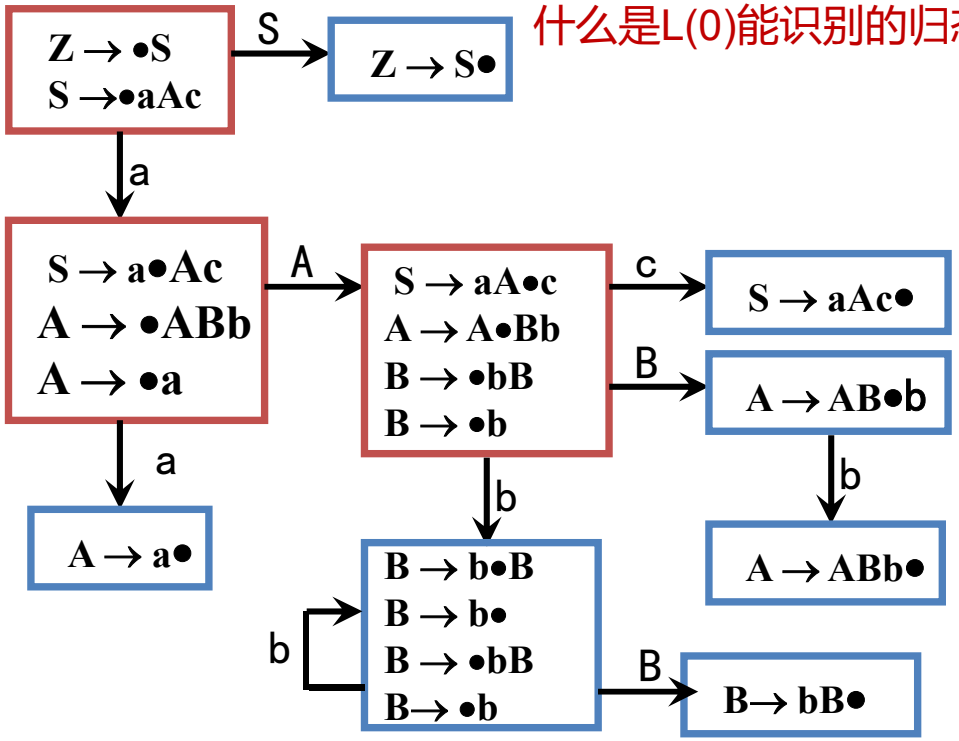
一个CFG的LR(0)自动机接受的是该CFG的所有LR0归约规范活前缀 ---即用来识别活前缀

5.4 LR(0) 分析法

- **构造LR(0)自动机-例4**

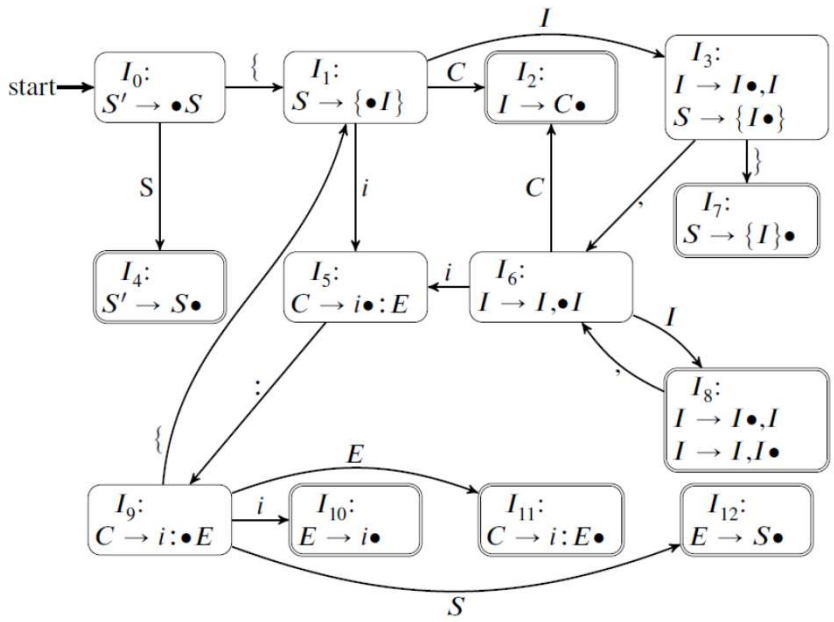
什么是L(0)能识别的活前缀?
什么是L(0)能识别的归态活前缀?

$V_T = \{a, b, c\}$
$V_N = \{S, A, B\}$
$S = S$
P:
{ $S \rightarrow aAc$
$A \rightarrow ABb$
$A \rightarrow a$
$B \rightarrow bB$
$B \rightarrow b$
}



5.4 LR(0) 分析法

■ 构造LR(0)自动机



什么是L(0)能识别的活前缀?

什么是L(0)能识别的归态活前缀?

试求状态I_1所接受的所有的仅以**终结符**组成的活前缀对应的正则表达式

状态I_1所接受的终结符组成的活前缀即是从开始状态I_0到状态I_1所有可能的路经对应的边标记序列.

而经过终结符为状态转移边能到达状态I_1的只有状态I_0, I_1, I_5和I_9. 故对应的正则表达式为“{(i:{)*”

LR(0)自动机刻画了可能出现在语法分析栈中的文法符号串 (即识别活前缀)

5.4 LR(0) 分析法

- **有效项**
  - 如果存在 $S' \underset{rm}{\overset{*}{\Rightarrow}} \alpha Aw \underset{rm}{\Rightarrow} \alpha\beta_1\beta_2 w$ ，那么我们说项$A \rightarrow \beta_1 . \beta_2$对$\alpha\beta_1$有效
- **有效项的意义**
  - 表示$\alpha\beta_1$中的后缀β_1可以形成$A \rightarrow \beta_1 . \beta_2$的句柄
  - 帮助我们决定是进行规约还是移入
  - 如果我们知道$A \rightarrow \beta_1 . \beta_2$对$\alpha\beta_1$有效

1. 如果β_2不等于空，表示句柄尚未出现在栈中，应该移入或者等待归约
2. 如果β_2等于空，表示句柄出现在栈中，应该归约

5.4 LR(0) 分析法

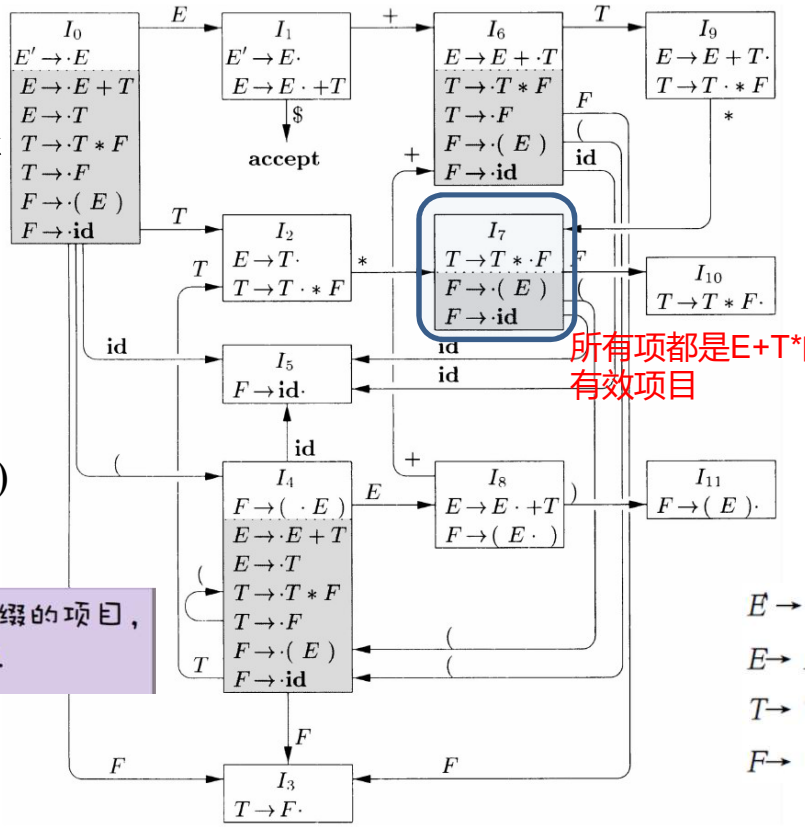
■ 活前缀与有效项目

- 所有的有效项目集合涵盖了从文法开始符号最右推出该活前缀的所有可能.
- 任何一个活前缀的有效项目就是自动机从开始状态出发在接受该活前缀后所能到达的状态对应的项目.
- 分析就是根据状态对应项目和当前的输入选择唯一的移进 - 归约操作.

5.4 LR(0) 分析法

- **E+T*是活前缀**
  - Assume栈内为E+T*，状态机处于 I_7
- **对应的最右推导**
  - E’=>E=>E+T=>E+T*F
  - E’ =>E=>E+T=>E+T*F=>E+T*(E)
  - E’ =>E=>E+T=>E+T*F=>E+T*id

设 $\alpha\beta_1$ 是一个活前缀，设 $A \to \beta_1 \bullet \beta_2$ 是一个以 β_1 为前缀的项目，称 $A \to \beta_1 \bullet \beta_2$ 是 $\alpha\beta_1$ 的一个有效项目 iff $S' \overset{*}{\underset{rm}{\Rightarrow}} \alpha\beta_1\beta_2 w$.



5.4 LR(0) 分析法

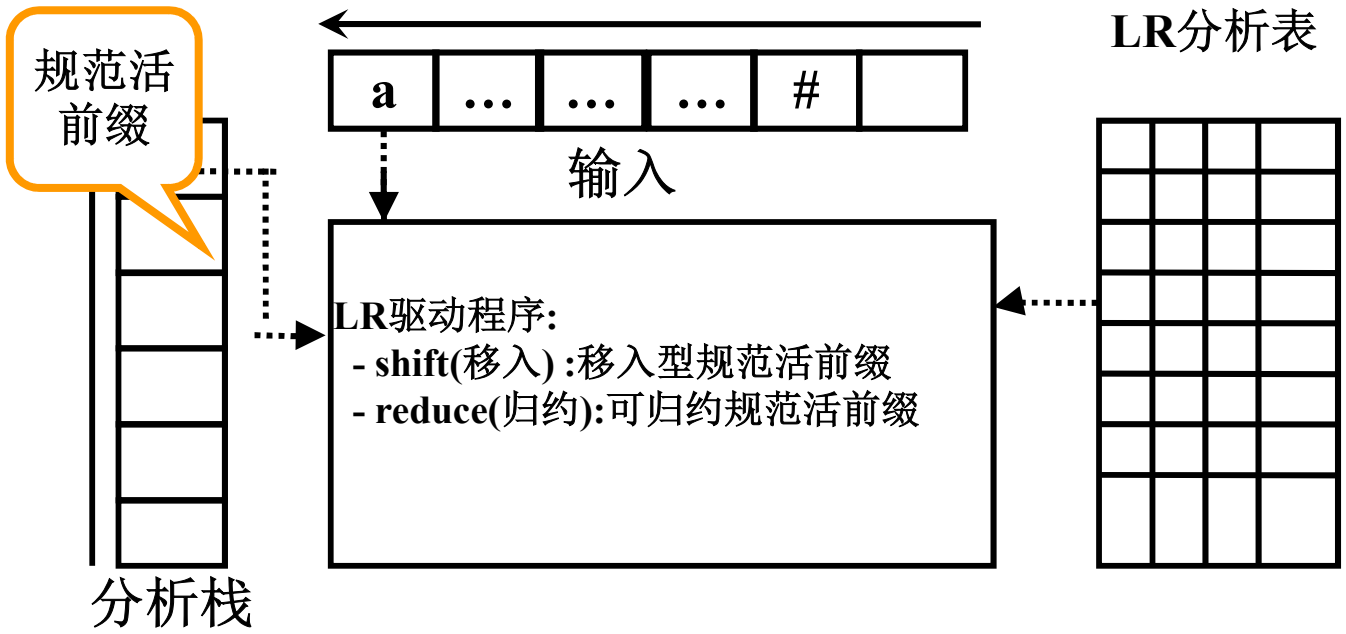
LR(0)分析法,相关概念

- 移入项目: $A \rightarrow \alpha \bullet a\beta$, $a \in V_T$
- 归约项目: $A \rightarrow \alpha \bullet$,
- 接受项目: $Z \rightarrow S \bullet$, ($Z \rightarrow S$是增广产生式)
- 移入状态:包含移入项目的状态
- 归约状态:包含归约项目的状态
- 冲突状态(同一个状态中):
  - 包含不同的归约项目，则称该状态存在归约-归约冲突；
  - 既包含移入项目，又包含归约项目，则称该状态存在移入-归约冲突

5.4 LR(0) 分析法

- **使用LR(0)自动机进行语法分析**
  - 规范LR(0)项集族: 提供构建LR(0)自动机的基础
    - LR(0)自动机可用于做出语法分析决定
    - LR(0)自动机中每个状态代表了LR(0)项集族中的一个项集
  - 符号栈 → 状态栈+符号栈(辅助)

5.4 LR(0) 分析法



Naïve 移进/规约

5.4 LR(0) 分析法

栈	输入	动作
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	移入
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	按照 $F \to \mathbf{id}$ 归约
$ F	$* \mathbf{id}_2$ $	按照 $T \to F$ 归约
$ T	$* \mathbf{id}_2$ $	移入
$ $T *$	$\mathbf{id}_2$ $	移入
$ $T * \mathbf{id}_2$	$	按照 $F \to \mathbf{id}$ 归约
$ $T * F$	$	按照 $T \to T * F$ 归约
$ T	$	按照 $E \to T$ 归约
$ E	$	accept

图 4-28　一个移入 – 归约语法分析器在处理输入 $\mathbf{id}_1 * \mathbf{id}_2$ 时经历的格局

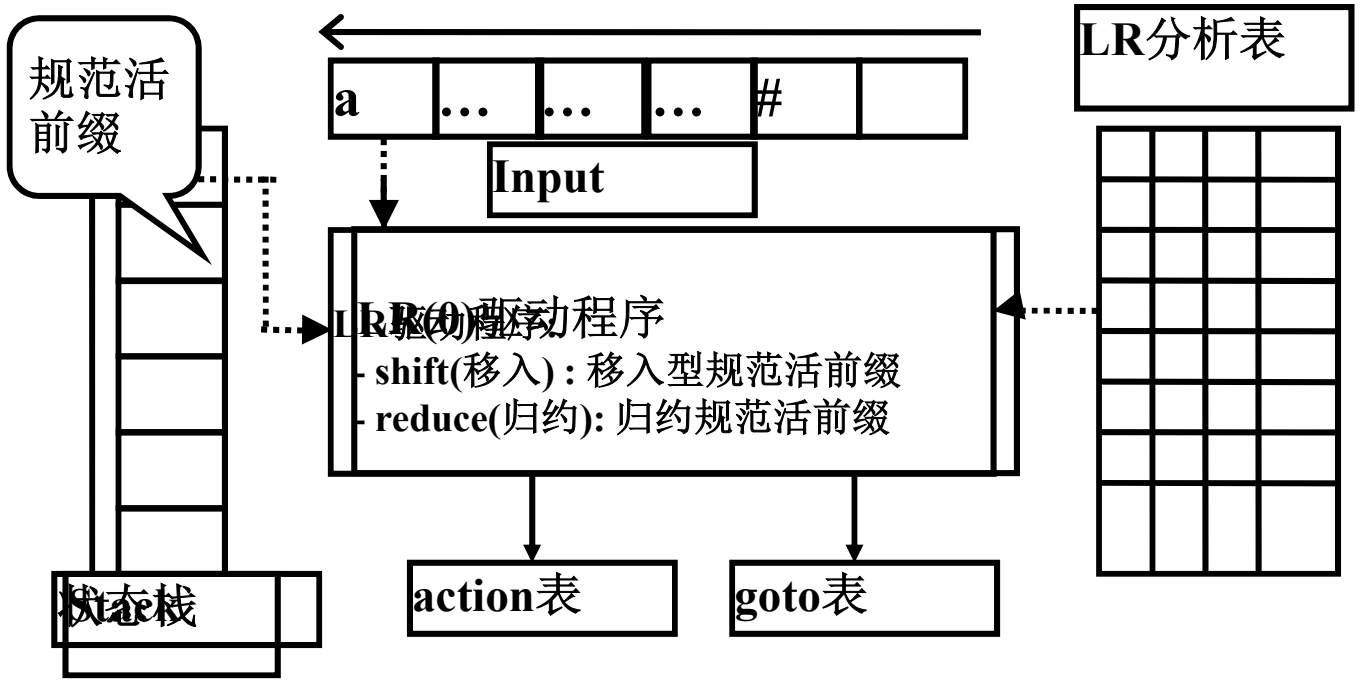
Naïve 移进/规约

$E \to E + T \mid T$

$T \to T * F \mid F$

$F \to (E) \mid \mathbf{id}$

5.4 LR(0) 分析法


规范活
前缀
a
…
…
…
#
Input
LR分析表
LR(0)驱动程序
- shift(移入) : 移入型规范活前缀
- reduce(归约): 归约规范活前缀
action表
goto表

5.4 LR(0) 分析法

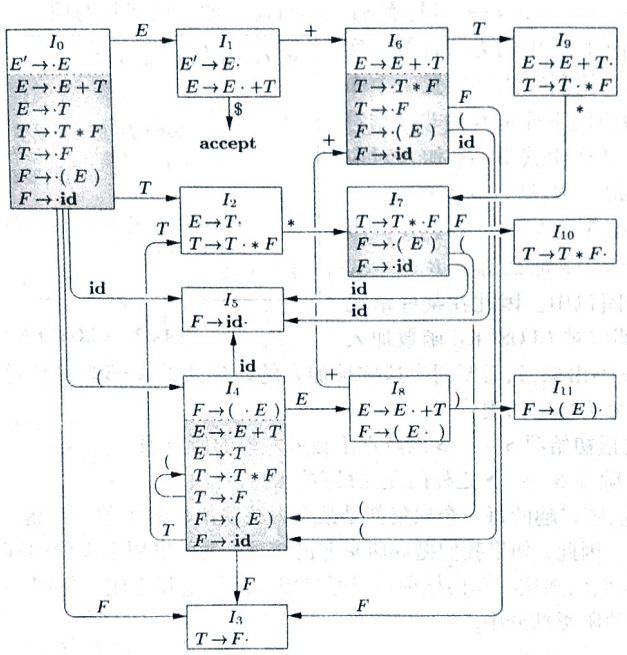


图 4-31　表达式文法(4.1)的 LR(0) 自动机

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

$E \to E + T \mid T$

$T \to T * F \mid F$

$F \to (E) \mid \mathbf{id}$

LR语法分析器的格局包含了栈中内容和余下输入 ($s_0 s_1 \ldots s_m$, $a_i a_{i+1} \ldots a_n \$$)
- 第一个分量是栈中的内容（右侧是栈顶）
- 第二个分量是余下输入

LR语法分析器的每一个状态都对应一个文法符号（s0除外）如果进入状态s的边的标号为X，那么s就对应于X

令Xi为si对应的符号，那么$X_1 X_2 \ldots X_m$ $a_i a_{i+1} \ldots a_n$对应于一个最右句型

5.4 LR(0) 分析法

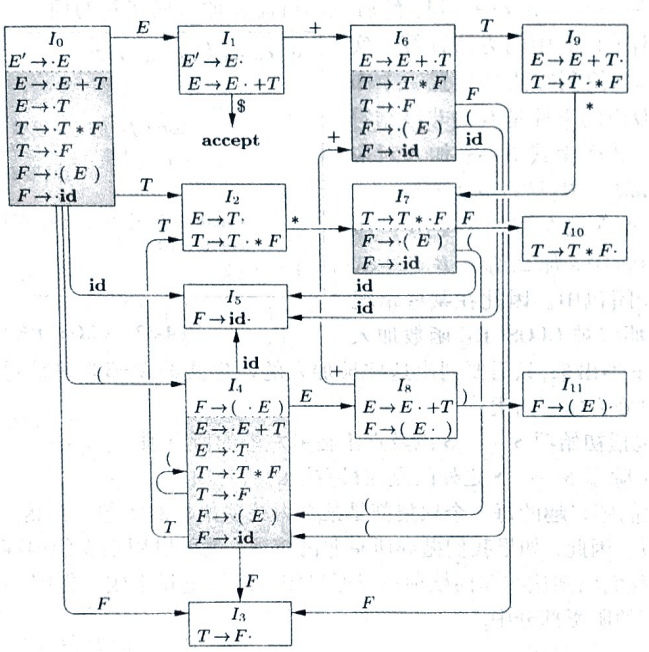


图 4-31　表达式文法(4.1)的 LR(0)自动机

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

对于格局($s_0s_1\ldots s_m$, $a_ia_{i+1}\ldots a_n$\$)，LR语法分析器查询条目ACTION[s_m,a_i]确定相应动作

- 移入s：执行移入动作，将s（对应a）移入栈中，新格局($s_0s_1\ldots s_ms$，$a_{i+1}\ldots a_n$\$)
- 归约A → β，将栈顶的β归约为A，压入状态s: ($s_0s_1\ldots s_{m-r}s$，$a_ia_{i+1}\ldots a_n$\$)，其中r是β的长度，s=GOTO[s_{m-r},A]
- 接受：语法分析过程完成
- 报错：发现语法错误，调用错误恢复例程

5.4 LR(0) 分析法

- **构造LR(0)分析表**
  - Action表

状 态 \ 终结符	a_1	…	#
S_1			
…			
S_n			

$action(S_i,a) = S_j$, 如果S_i到S_j有a输出边

$action(S_i,c) = R_p$, 如果S_i是p-归约状态,$c \in Vt \cup \{\#\}$

$action(S_i,\#) = accept$, 如果S_i是接受状态

$action(S_i,a) = error$, 其他情形

5.4 LR(0) 分析法

- **构造LR(0)分析表**
  - GOTO表

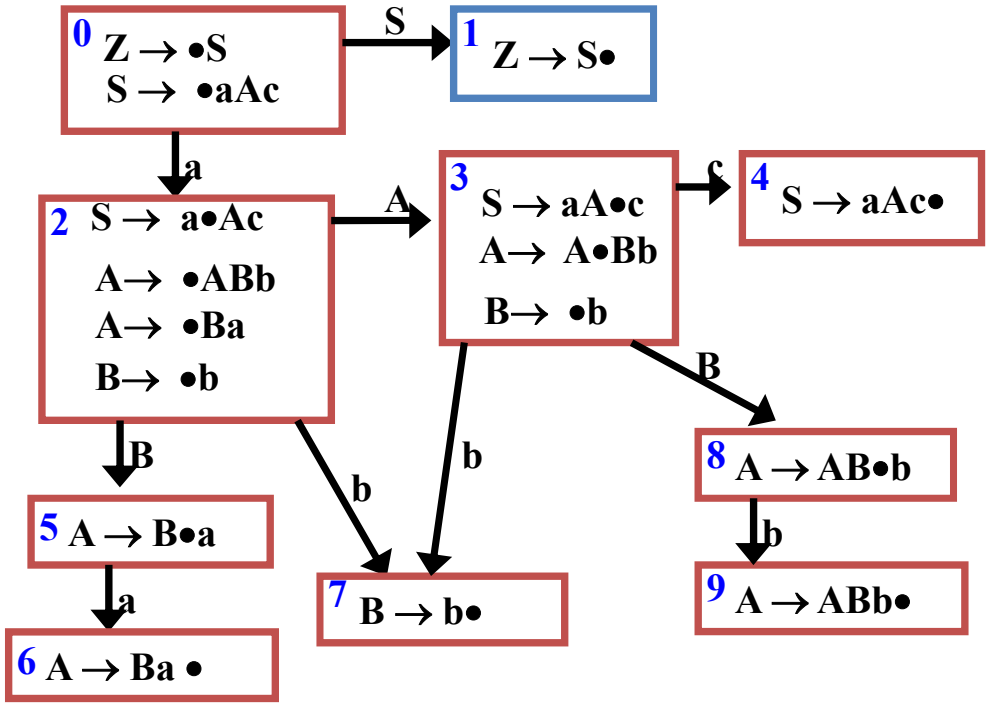
状 态 \ 非终结符	A_1	…	#
S_1			
…			
S_n			

goto (S_i, A) = S_j, 如果S_i到S_j有A输出边
goto (S_i, A) = error,如果S_i没有A输出边

5.4 LR(0) 分析法

- 例子

$V_T = \{a, b, c\}$
$V_N = \{S, A, B\}$
$S = S$
P:
{ (1) S→ aAc
(2)A → ABb
(3)A → Ba
(4)B → b
}



5.4 LR(0) 分析法

先前做法

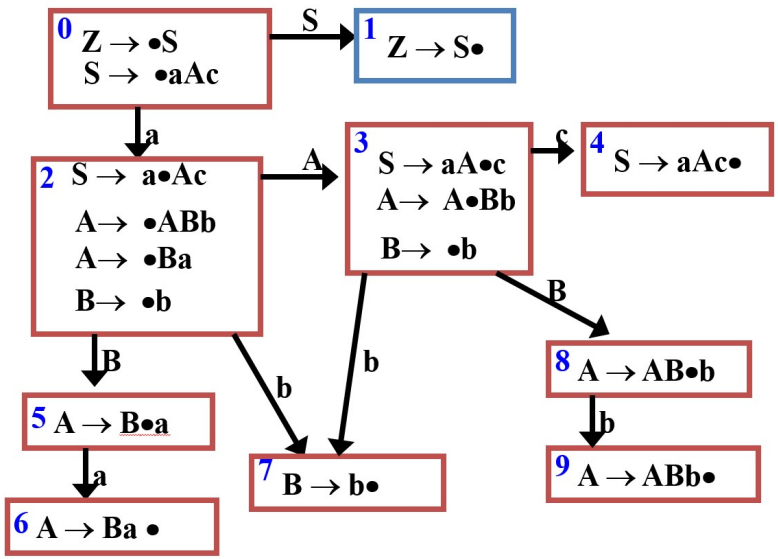
P:

(0) Z → S
(1) S→ aAc
(2)A → ABb
(3)A → Ba
(4)B → b

a	b	a	c

符号栈	输入流	分析动作
	abac#	移入
a	bac#	移入
ab	ac#	归约 (4)
aB	ac#	移入
aBa	c#	归约(3)
aA	c#	移入
aAc	#	归约(1)
S	#	归约(0)
Z	#	接受

5.4 LR(0) 分析法



	action表					goto表		
	a	b	c	#		S	A	B
0	S2					1		
1				accept				
2		S7					3	5
3		S7	S4					8
4	R1	R1	R1	R1				
5	S6							
6	R3	R3	R3	R3				
7	R4	R4	R4	R4				
8		S9						
9	R2	R2	R2	R2				

5.4 LR(0) 分析法

LR(0)分析驱动程序

- 符号约定：
  - S0: 开始状态
  - Stack:状态栈
  - Stack(top):栈顶元素
  - P:产生式
  - | P |:产生式P右部符号个数;
  - P_A:产生式P左部非终极符;
  - Push(S):把状态S压入stack;
  - Pop(n):从stack弹出n个栈顶元素;

初始化: push(S0); a = readOne();
L: Switch *action*(stack(top), a)
Case error: error();
Case accept: return true;
Case Si: push(Si), a=readOne(); goto L;
Case R_P: pop(|P|);
push(*goto*(stack(top), P_A));
goto L;

5.4 LR(0) 分析法

完整过程

a	b	a	c

P: (0) Z → S; (1) S→ aAc; (2)A → ABb;
(3)A → Ba; (4)B → b

	action表					goto表		
	a	b	c	#		S	A	B
0	S2					1		
1				accept				
2		S7					3	5
3		S7	S4					8
4	R1	R1	R1	R1				
5	S6							
6	R3	R3	R3	R3				
7	R4	R4	R4	R4				
8		S9						
9	R2	R2	R2	R2				

状态栈	输入流	分析动作
0	abac#	S2
02	bac#	S7
027	ac#	R4,Goto(2, B)=5
025	ac#	S6
0256	c#	R3,Goto(2, A)=3
023	c#	S4
0234	#	R1, Goto(0, S)=1
01	#	Accept

5.4 LR(0) 分析法

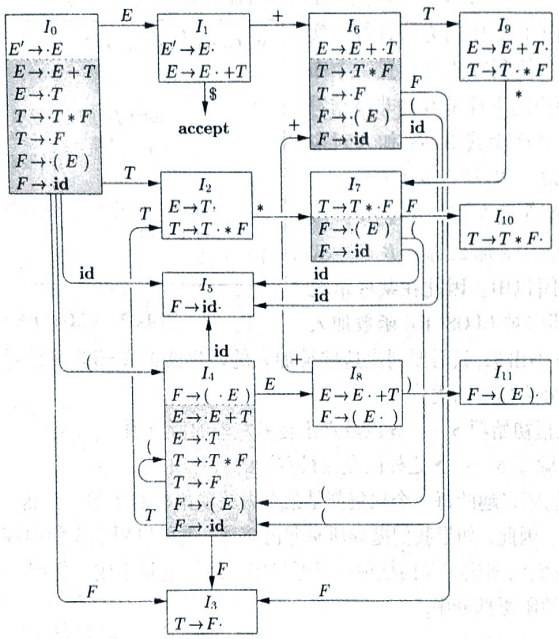


图 4-31　表达式文法(4.1)的 LR(0)自动机

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			

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

	栈	符号	输入	动作
(1)	0		id * id + id $	移入
(2)	0 5	id	* id + id $	根据 $F \to$ id 归约
(3)	0 3	F	* id + id $	根据 $T \to F$ 归约
(4)	0 2	T	* id + id $	移入
(5)	0 2 7	T *	id + id $	移入
(6)	0 2 7 5	T * id	+ id $	根据 $F \to$ id 归约
(7)	0 2 7 10	T * F	+ id $	根据 $T \to T * F$ 归约
(8)	0 2	T	+ id $	根据 $E \to T$ 归约
(9)	0 1	E	+ id $	移入
(10)	0 1 6	E +	id $	移入
(11)	0 1 6 5	E + id	$	根据 $F \to$ id 归约
(12)	0 1 6 3	E + F	$	根据 $T \to F$ 归约
(13)	0 1 6 9	E + T	$	根据 $E \to E + T$ 归约
(14)	0 1	E	$	接受

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid \mathbf{id}$$

5.4 LR(0) 分析法

- **已知LR_0 是CFG *G*的LR(0)自动机**
  - 如果LR_0的任意一个状态都不存在冲突(归约-归约冲突、移入-归约冲突), 则G称为LR(0)文法;
  - 可以推知：在LR(0)文法中，
    - 任意状态或者是移入状态,或者是归约状态
    - 如果是归约状态,一定存在一个唯一的归约项目,该归约项目对应一个产生式p,因此, 该归约状态称为*p-归约状态*

5.4 LR(0) 分析的局限

- LR(0)文法仅凭符号栈里的内容来确定可归约活前缀, 非常容易产生冲突;
  - LR(0)文法易于产生冲突的原因在于在确定分析动作时没有考虑输入串信息。
  - 事实上,只有有限的文法能满足LR(0)文法的条件;
- LR(0)分析不是一种实用的方法，只是为介绍LR分析的思想而引进的;

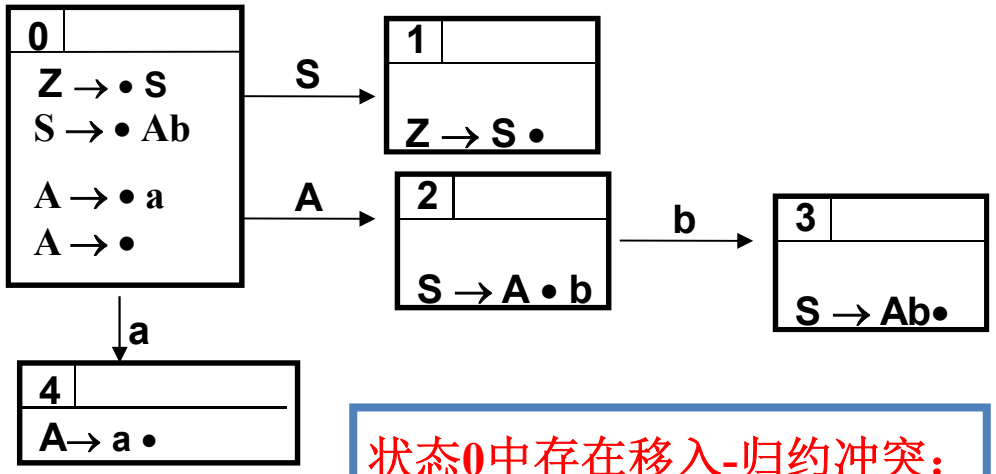
LR (0)自动机的移入-归约冲突

V_T = {a, b}

V_N = {S, A}

S = S

P:
{ (1) S→ Ab
(2) A → ε
(3) A → a
}



状态0中存在移入-归约冲突：
(1) 移入项目:A → • a
(2) 归约项目: A → •

LR(0)自动机的归约-归约冲突

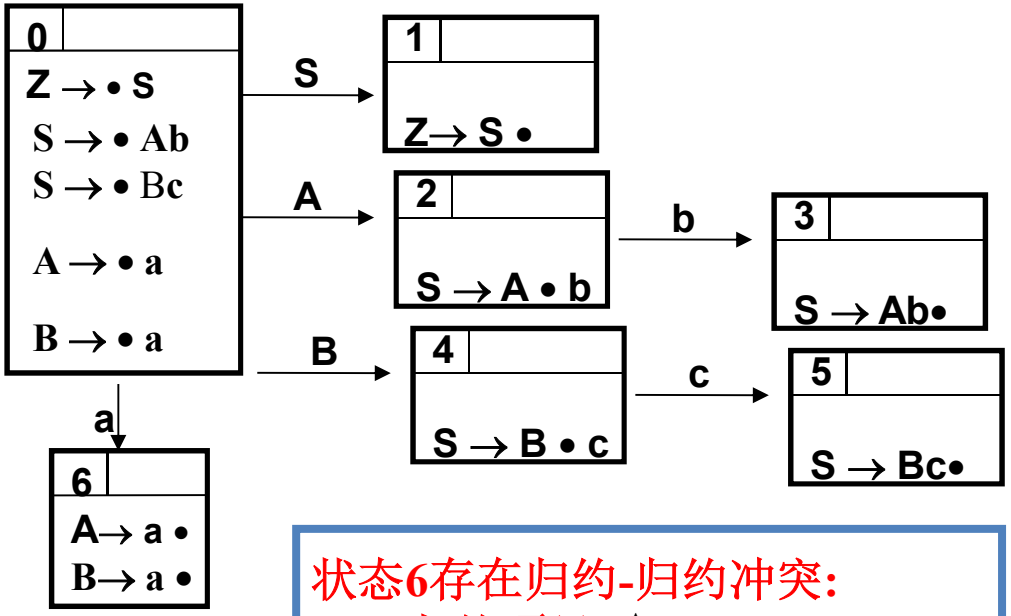
V_T = {a, b, c}

V_N = {S, A, B}

S = S

P:

{(1) S→ Ab
(2) S→ Bc
(3) A → a
(4) B → a
}



状态6存在归约-归约冲突:

(1) 归约项目: A → a •

(2) 归约项目: B → a•

如何消除冲突?

- **改进LR(0)**
  - SLR
  - LR(1)
  - LALR

作业

- 给定文法 G(S): $S \rightarrow S\ S \mid (S) \mid \varepsilon$ 其中(,)是终结符
  - 为该文法构造增广文法G(S')
  - 构造G(S') 的识别活前缀LR(0) 项目自动机
  - 为自动机中每个项目标出核心项目
  - 为什么正则表达式(*S(*所生成的文法符号串一定是活前缀；

5.5 SLR

改进策略之SLR

- **消除冲突：向前看一个输入符号来选择分析动作(考虑现实)**
  - 对于任何形如I={X→γ• bβ，A→ α• ， B→ α•}的LR(0)项目集，
    - 假设下一个输入符号是: a，则存在规约冲突；但
    - 如果Follow(A)∩Follow(B) = Φ, 且b∉Follow(A)，b∉Follow(B)，则可以通过如下方法进行冲突消除

改进策略之SLR

- **消除冲突：向前看一个输入符号来选择分析动作(考虑现实)**
  - 移入-归约冲突(S-R冲突)
    - 移入: 如存在移入项目 $A\rightarrow \alpha\bullet a\beta$
    - 归约: 如果存在归约项目 $B\rightarrow \pi\bullet$, 且 $a \in follow(B)$
  - 归约-归约冲突(R-R冲突)
    - 归约(P1): 如果存在归约项目 $A\rightarrow \pi\bullet$, $a\in follow(A)$, 产生式P1= $A\rightarrow \pi$
    - 归约(P2): 如果存在归约项目 $B\rightarrow \pi'\bullet$, $a\in follow(B)$, 产生式P2= $B\rightarrow\pi'$

LR(0)分析表 (带有S-R 冲突)

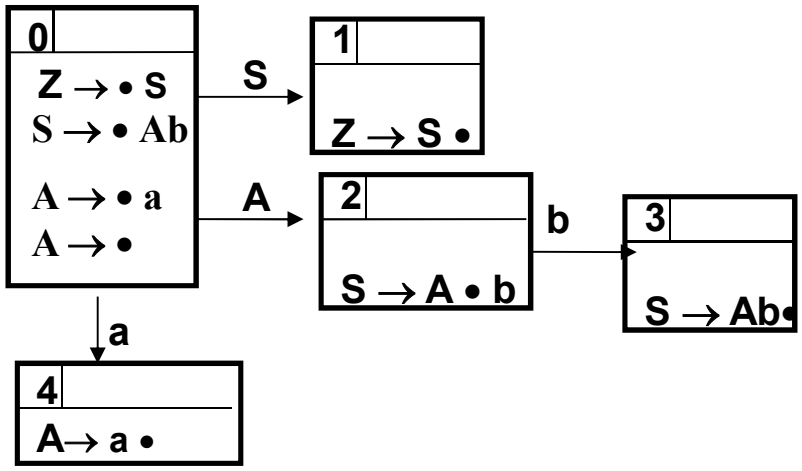
V_T = {a, b}

V_N = {S, A}

S = S

P:

{ (1) S→ Ab
(2) A → ε
(3) A → a
}



状态0中存在移入-归约冲突：
(1) 移入项目: A → • a
(2) 归约项目: A → •

	Action 表				Go to 表	
	a	b	#		S	A
0	S5;R2	R2	R2		1	3
1			Accept			
2		S3				
3	R1	R1	R1			
4	R3	R3	R3			

LR(0)分析表 (没有冲突)

$V_T = \{a, b\}$

$V_N = \{S, A\}$

S = S

P:

{ (1) S→ Ab

(2) A → ε

(3) A → a

}

	Action 表				Goto 表	
	a	b	#		S	A
0	S5	R2			1	3
1			Accept			
2		S3				
3			R1			
4		R3				

冲突用follow(A)解决掉了

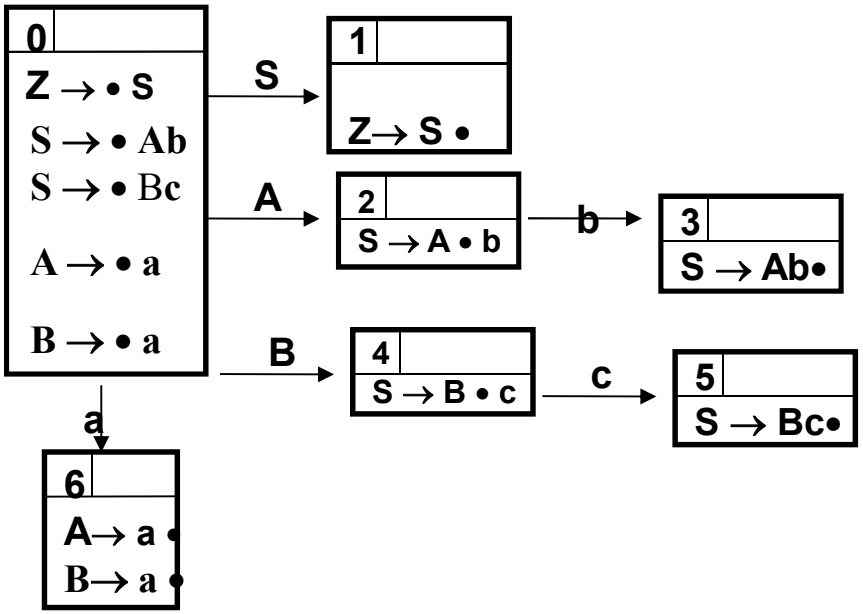
LR(0)分析表 (带有R-R 冲突)

V_T = {a, b, c}

V_N = {S, A, B}

S = S

P:
{(1) S→ Ab
(2) S→ Bc
(3) A → a
(4) B → a
}



	Action 表					Goto 表		
	a	b	c	#		S	A	B
0	S7					1	3	5
1				Accept				
2		S4						
3	R1	R1	R1	R1				
4			S6					
5	R2	R2	R2	R2				
6	R3 R4	R3 R4	R3 R4	R3 R4				

状态6存在归约-归约冲突:
(1) 归约项目: A → a •
(2) 归约项目: B → a•

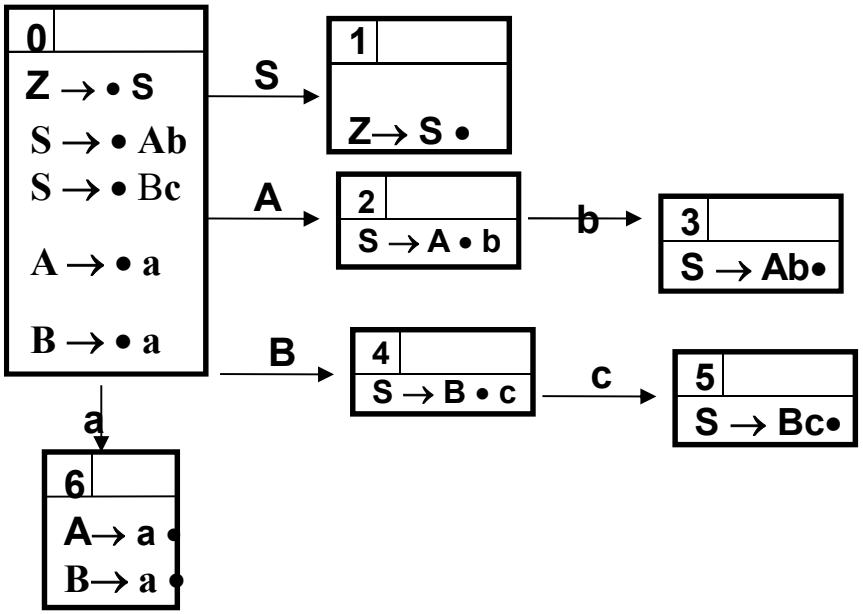
LR(0)分析表 (带有R-R 冲突)

V_T = {a, b, c}

V_N = {S, A, B}

S = S

P:
{(1) S→ Ab
(2) S→ Bc
(3) A → a
(4) B → a
}



	Action 表					Goto 表		
	a	b	c	#		S	A	B
0	S7					1	3	5
1				Accept				
2		S4						
3	R1	R1	R1	R1				
4			S6					
5	R2	R2	R2	R2				
6		R3	R4					

状态6存在归约-归约冲突:
(1) 归约项目: A → a •
(2) 归约项目: B → a•

冲突用 follow(A) 和 follow(B)消除了

SLR(1) 分析

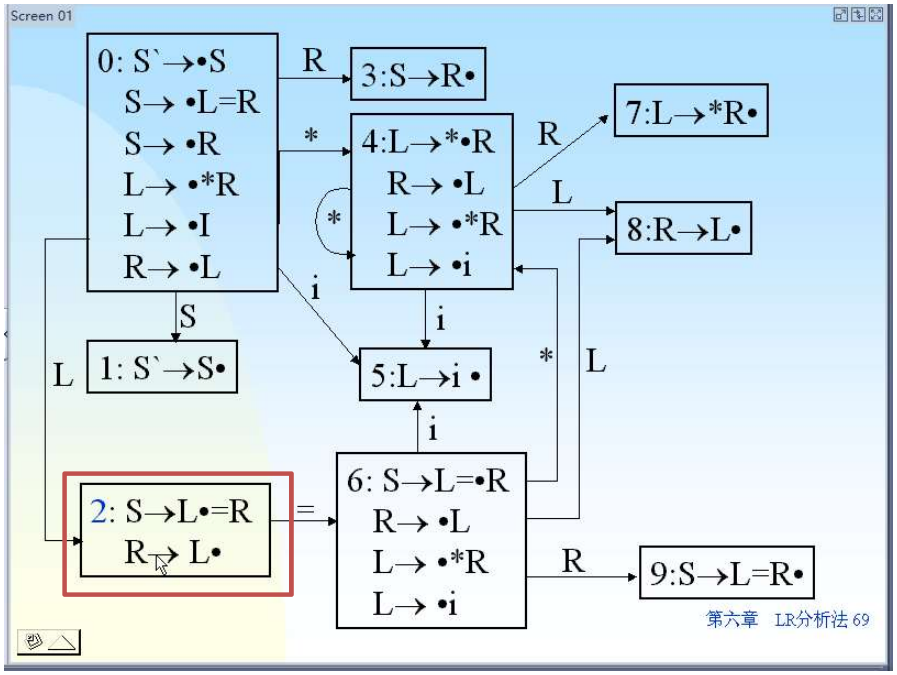
- **SLR(1) 分析的思想**
  - 向前看一个输入符号;
  - 用非终极符的follow集合解决冲突;
  - 如果能将LR(0)自动机中的所有冲突消除掉，则该文法称为*SLR(1)文法*，否则称为非SLR(1)文法。
- **SLR(1)文法**
  - 文法G的SLR(1)自动机与它的LR(0)自动机完全相同
  - SLR(1)分析表中Action表与LR(0)分析表的Action表稍有不同(移入动作没有区别，归约动作有区别)
  - SLR(1)驱动程序与LR(0)驱动程序也相同

非SLR(1)文法的例子

文法 G

$$
\begin{cases} S \rightarrow L{=}R \mid R \\ L \rightarrow *R \mid id \\ R \rightarrow L \end{cases}
$$

First(S)
= First(L)
= First(R)
= {*, id}
Follow(S)
= {$}
Follow(L)
= Follow(R)
= {=, $}

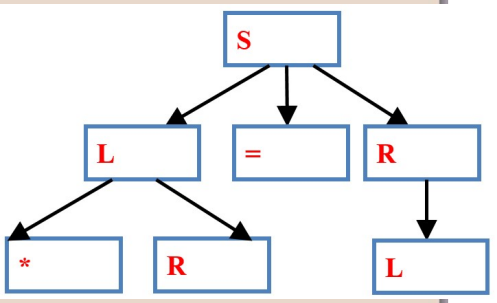


非SLR(1)文法的例子

- 文法 G 不是二义文法，但是也不是 SLR 文法，状态 I_2 有移进 - 归约冲突：$= \in Follow(R)$，所以面对 `=`，可以用 $R \to L$ 归约，也可移进到状态 I_6.
- 原因是 Follow 集作为归约的条件不充分：
  - `=` 只可能出现在右句型 *R 之后：

对于同一个非终极符可能出现在不同的位置，不同位置的后继符号(follow)是不同的，统一看待，仍有可能引起冲突。

$$S \underset{rm}{\Rightarrow} L=R$$
$$\underset{rm}{\Rightarrow} L=L$$
$$\underset{rm}{\Rightarrow} L=id$$
$$\underset{rm}{\Rightarrow} *R=id$$



  - 文法 G 没有 ``R=XXX'' 的右句型，因此，在状态 $I_2$ 下面对 ``='', 只能选择移进.
- 解决方法：将后随符号和最右句型联系起来，精确定位每个项目的后随符号.

SLR(1)分析的局限性

- 对SLR分析的思考
  - 在构造SLR分析表的方法中，若项目集I_k中含有$A\rightarrow\alpha\bullet$，那么在状态k时，只要面临 输入符号$a \in Follow(A)$，就确定采用$A\rightarrow\alpha$产生式进行归约。但是，在某种情况下，当状态k呈现于栈顶时，栈里的符号串所构成的活前缀$\beta\alpha$未必允许把α归约为A。因为可能没有一个规范句型含有前缀βAa。因此此时用$A\rightarrow\alpha$产生式进行归约未必有效。

二义性文法的处理

- **二义性文法都不是LR的**
- **二义性文法却有其存在的必要**
- **对于某些二义性文法**
  - 可以通过消除二义性规则来保证每个句子只有一棵语法分析树
  - 且可以在LR分析器中实现这个规则

优先级/结合性消除冲突

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

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

- **二义性文法的优点**
  - 容易修改算符的优先级和结合性
  - 简洁：较少的非终结文法符号
  - 高效：不需要处理E → T这样的归约

二义性表达式文法的LR(0)项集

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

I_7，I_8中有冲突，在输入+或*时，不能确定是归约还是移入，且不可能通过向前看符号解决

I_0:
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot \mathbf{id}$

I_1:
$E' \rightarrow E \cdot$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

I_2:
$E \rightarrow (\cdot E)$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot \mathbf{id}$

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

I_4:
$E \rightarrow E + \cdot E$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot \mathbf{id}$

I_5:
$E \rightarrow E * \cdot E$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot \mathbf{id}$

I_6:
$E \rightarrow (E \cdot)$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

I_7:
$E \rightarrow E + E \cdot$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

I_8:
$E \rightarrow E * E \cdot$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

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

基于优先级解决冲突

前缀	栈	输入
$E + E$	0 1 4 7	$*$ **id** $

- **设定优先级如下：*的优先级大于+，且+是左结合的，则有**
  - 下一个符号为+时，我们应该将E+E归约为E
  - 下一个符号为*时，我们应该移入*，期待移入下一个符号

解决冲突之后的SLR(1)分析表

- **对于状态7，输入**
  - +时归约
  - *时移入
- **对于状态8**
  - 执行归约

I_7: $E \to E + E\cdot$
$E \to E\cdot + E$
$E \to E\cdot * E$

I_8: $E \to E * E\cdot$
$E \to E\cdot + E$
$E \to E\cdot * E$

状态	ACTION						GOTO
	id	+	*	(	)	$	E
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	

悬空else的二义性

stmt → **if** *expr* **then** *stmt* **else** *stmt*

| **if** *expr* **then** *stmt*

| **other**

$S' \rightarrow S$

$S \rightarrow i\ S\ e\ S \mid i\ S \mid a$

- 栈中内容 if *expr* then *stmt*,

是输入else，还是归约?

- 答案是移入

Follow(S)={e,$}

I_0:
$S' \rightarrow \cdot S$
$S \rightarrow \cdot iSeS$
$S \rightarrow \cdot iS$
$S \rightarrow \cdot a$

I_1:
$S' \rightarrow S\cdot$

I_2:
$S \rightarrow i\cdot SeS$
$S \rightarrow i\cdot S$
$S \rightarrow \cdot iSeS$
$S \rightarrow \cdot iS$
$S \rightarrow \cdot a$

I_3:
$S \rightarrow a\cdot$

I_4:
$S \rightarrow iS\cdot eS$
$S \rightarrow iS\cdot$

I_5:
$S \rightarrow iSe\cdot S$
$S \rightarrow \cdot iSeS$
$S \rightarrow \cdot iS$
$S \rightarrow \cdot a$

I_6:
$S \rightarrow iSeS\cdot$

作业

- 教材P164：4.6.1(1), 4.6.2, 4.6.3
- 教材P164: 4.6.5，4.6.6

5.6 LR(1)

SLR局限性的解决的办法

- **结论**
  - 并非Follow集中的符号都会出现在规范句型中
- **对策**
  - 给每个LR(0)项目添加展望信息，即添加句柄之后可能跟的终结符，因为这些终结符确实是规范句型中跟在句柄之后的
  - 不要看Follow集，而是要看展望符(First集)

解决的办法

- **LR(1) 分析**
  - 基本思想:
    - 对于非终极符的每个不同出现求其*后继终极符*, 称为展望符/向前看符号;
    - 为了让LR分析器的每个状态精确指明哪些输入符号可以跟在句柄α后面，使α可能被规约为A
    - 规范LR(1) 项目 = LR(0)项目+展望符集
      - 形式如[A → α·β,a]，其中A → αβ是一条产生式，a是一个终结符号或者$符
      - a是向前看符号，长度为1

解决的办法

- **LR(1) 项目**
  - 两个部分：(A→ α•β, {a, …})
    - LR(0) 项目: A→ α•β
    - 展望符集: {a, …}，表示非终极符A此次出现的所有可能follow符号，**是FOLLOW(A)的子集**
  - 例如：
    - S→ L•=R , {#}
    - A → α• , {a, b}
  - 展望符集的作用:
    - 对于移入型项目, 不起作用,但是需要保存：如果β≠ε时， a没有任何作用
    - 对于归约型项目, 表示只有当下一个输入符是其中一个展望符时, 才可以进行归约动作：a的意义是若栈顶状态中存在LR(1)项[A → α·,a]， 在输入符号为a时才会进行归约

解决的办法

- **LR(1) 有效项（包含了展望符）**
  - $S' \rightarrow \delta A\omega \rightarrow \delta\alpha\beta\omega$，如果$a \in First(\omega)$，则有 $A \rightarrow \bullet\alpha\beta$, $A \rightarrow \alpha\bullet\beta$, $A \rightarrow \alpha\beta\bullet$ 的展望符a，记$(A \rightarrow \bullet\alpha\beta, a)$, $(A \rightarrow \alpha\bullet\beta, a)$, $(A \rightarrow \alpha\beta\bullet, a)$ 都是有效的LR(1)项目；
    - 如果$a \in Follow(A)$但$a \notin First(\omega)$，a不属于$A \rightarrow \bullet\alpha\beta$, $A \rightarrow \alpha\bullet\beta$, $A \rightarrow \alpha\beta\bullet$的展望符集

解决的办法

- **LR(1) 分析**
  - 分析步骤:
    - 构造 LR(1) 自动机
      - LR(1) 项目 = LR(0)项目+展望符集
    - 生成 LR(1)分析表 (action & goto)
    - LR(1)驱动程序 = LR(0)驱动程序

LR(1) 自动机

- **LR(1) 项目集合*关于符号X的投影***
  - IS 是 LR(1) 项目的集合;
  - X 是一个符号;
  - $IS_{(X)}$ 表示项目集IS关于X的投影:
  - $IS_{(X)} = \{(S \rightarrow \alpha X \bullet \beta, ss) \mid (S \rightarrow \alpha \bullet X \beta, ss) \in IS, X \in V_T \cup V_N\}$

投影对展望符集没有影响!

- **IS = {(A → A•Bb, {a,b}), (B → a•, #), (B → b•B, {b})}**
- **X = B**
- **$IS_{(B)}$ = {(A → AB•b, {a, b}), (B → bB•, {b})}**

LR(1) 自动机

- **LR(1)项目集合的*闭包***
  - IS 是LR(1)项目的集合;
  - CLOSURE(IS)是一个LR(1)项目集合, 按照下面的步骤计算:

    [1]初始, CLOSURE(IS) = IS;

    [2] 对于CLOSURE(IS)没有处理的LR(1)项目,

    如果其形式为 $(B\rightarrow \beta\bullet A\pi, ss)$,

    而且A的全部产生式是$\{A\rightarrow\alpha 1, \ldots, A\rightarrow\alpha n\}$

    则增加如下LR(1)项目到CLOSURE(IS)

    $\{(A\rightarrow \bullet\alpha 1, ss'), \ldots, (A\rightarrow \bullet\alpha n, ss')\}$,

    其中 $ss' = first(\pi)$, 如果符号串π不导出空;

    $ss' = (first(\pi)-\{\varepsilon\}) \cup ss$, 如果符号串$\pi$导出空;

    [3] 重复[2]直到 CLOSURE(IS)收敛;

LR(1) 自动机

- **LR(1)项目集合的*闭包***
  - 示意：LR1Closure.pdf

- 设 $[A \to \alpha \bullet B\beta, a]$ 是有效项目，$B \to \gamma$ 是产生式，则：

$$
\begin{aligned}
S' &\overset{*}{\underset{rm}{\Rightarrow}} \delta Aw \quad (a \in first(w)) \\
&\underset{rm}{\Rightarrow} \delta\alpha B\beta w \\
&\underset{rm}{\Rightarrow} \delta\alpha\gamma\beta w
\end{aligned}
$$

  所以 $[B \to \bullet\gamma, first(\beta a)]$ 是对活前缀 $\delta\alpha$ 一个有效项目.

- $$
\begin{aligned}
&\overline{\{[A \to \alpha \bullet B\beta, a]\}} \\
= \ &\{[A \to \alpha \bullet B\beta, a]\} \cup \{[B \to \bullet\gamma, first(\beta a)]\}
\end{aligned}
$$

Example: 文法 G

$$
\begin{aligned}
&\overline{\{[S' \to \bullet S, \$]\}} \\
= \{ \ &[S' \to \bullet S, \$], [S \to \bullet L{=}R, \$], [S \to \bullet R, \$], \\
&[L \to \bullet{*}R, =], [L \to \bullet id, =], [R \to \bullet L, \$]\}
\end{aligned}
$$

$$
\begin{aligned}
S &\to L{=}R \mid R \\
L &\to {*}R \mid id \\
R &\to L
\end{aligned}
$$

LR(1)自动机

- **goto函数()**
  - IS 是 LR(1) 项目集;
  - X 是一个符号;
  - goto(IS, X) = CLOSURE(IS $_{(X)}$)

LR(1)自动机的构造过程

- [1]增广产生式 $Z \rightarrow S$
- [2]$\Sigma = V_T \cup V_N \cup \{\#\}$
- [3]S0 = CLOSURE($\{(S' \rightarrow \bullet S, \{\#\})\}$)
- [4]ISS = {S0}
- [5]对于ISS中的每一个项目集合IS, 和每个符号$X \in \Sigma$,

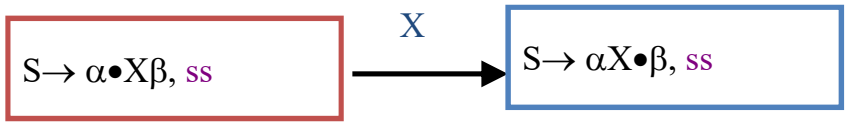
  计算IS' = goto(IS, X),

  如果IS'不为空, 则 建立 IS $\xrightarrow{X}$ IS',

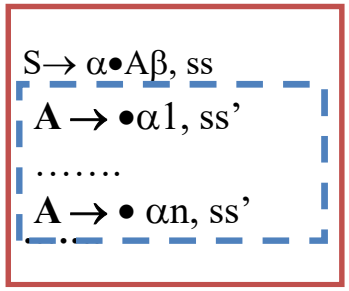
  如果IS'不为空且IS'不属于ISS,则把IS'加入ISS;
- [6]重复[5]直到ISS收敛;

如何计算展望符集?

- **投影得到的项目**
  - **继承**



  - **扩展（闭包产生的新项目）**



ss' = first(β), 如果β不导出空;

ss' = (first(β)-{ε}) ∪ss, 如果β导出空;

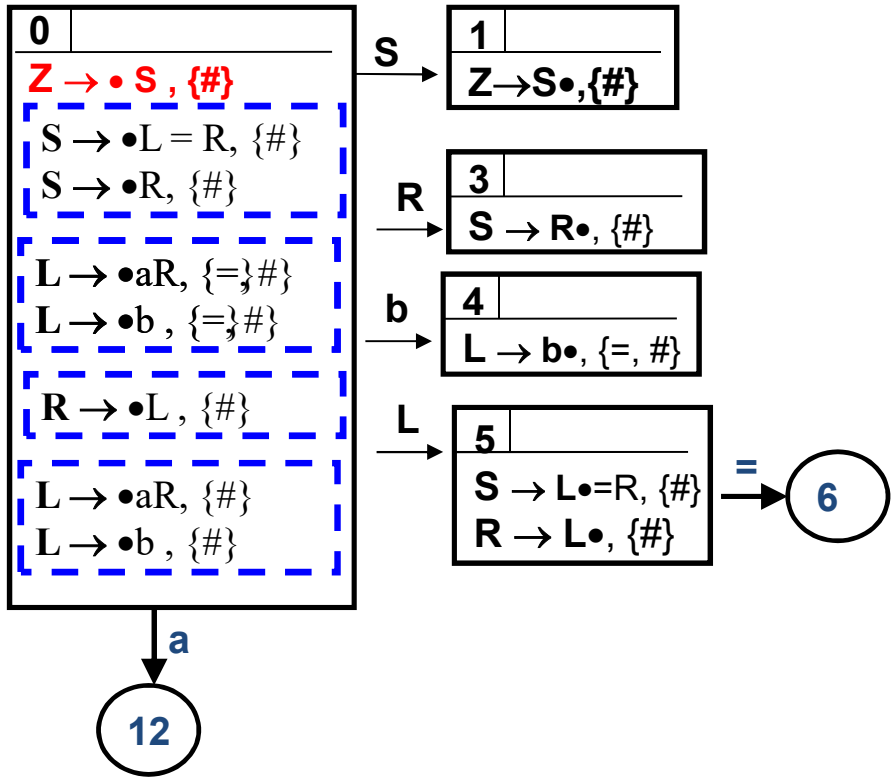
LR(1)自动机的构造

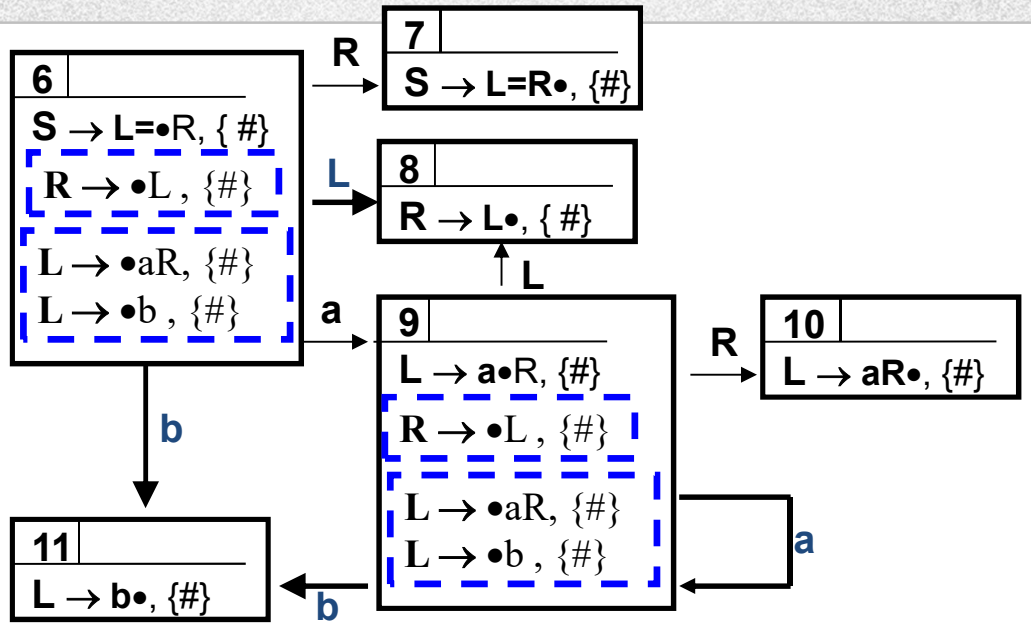
V_T = {a, b, =}

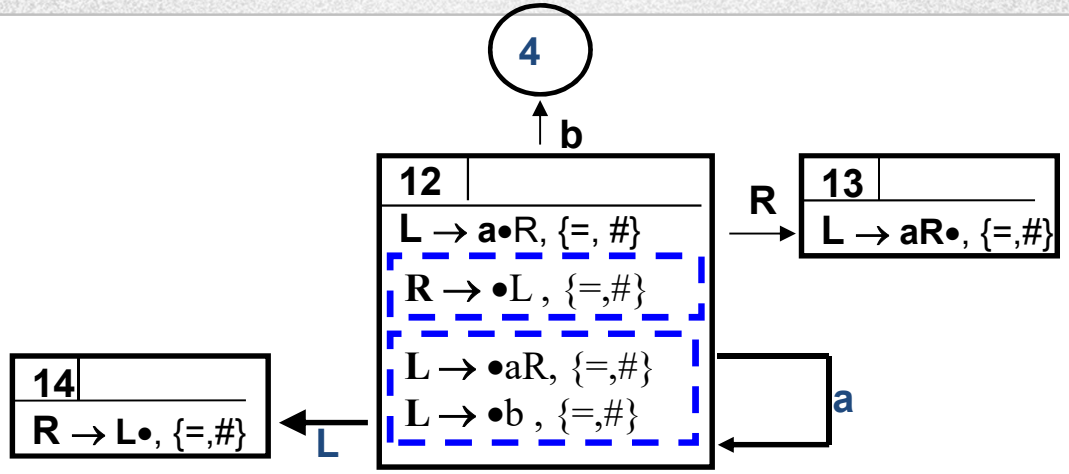
V_N = {S, L, R}

S = S

P:
{(1) S → L = R
(2) S → R
(3) L → aR
(4) L → b
(5) R → L
}




6
S → L=•R, { #}
R → •L , {#}
L → •aR, {#}
L → •b , {#}
R
7
S → L=R•, {#}
L
8
R → L•, { #}
L
a
9
L → a•R, {#}
R → •L , {#}
L → •aR, {#}
L → •b , {#}
R
10
L → aR•, {#}
a
b
11
L → b•, {#}
b


4
b
12
L → a•R, {=, #}
R → •L , {=,#}
L → •aR, {=,#}
L → •b , {=,#}
R
13
L → aR•, {=,#}
a
L
14
R → L•, {=,#}

LR(1) 分析表

- action表

(1)action(S_i,a) = S_j, 如果S_i到S_j有a输出边
(2)action(S_i,a) = R_p, 如果S_i中包含这样LR(1)项目,
(A→α•, ss),其中A→α是产生式P,且a∈ss;
(3)action(S_i,#) = accept, 如果S_i是接受状态
(4)action(S_i,a) = error, 其他情形

终极符 / 状 态	a_1	…	#
S_1			
…			
S_n			

LR(1) 分析表

- goto表

goto (S_i, A) = S_j, 如果S_i到S_j有A输出边
goto (S_i, A) = error,如果S_i没有A输出边

非终极符 / 状态	A_1	…	A_n
S_1			
…			
S_n			

LR(1) 分析表

	Action 表					Goto 表		
	a	b	=	#		S	L	R
0	S12	S4				1	5	3
1				Accept				
3				R2				
4			R4	R4				
5			S6	R5				
6	S9	S11					8	7
7				R1				

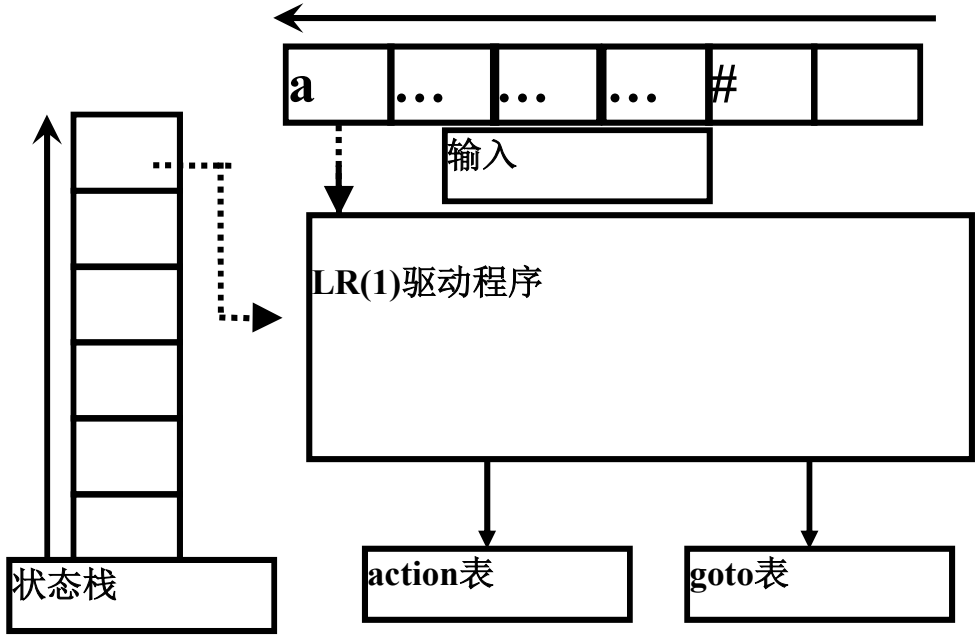
LR(1) 分析表 (接上页.)

	Action 表					Goto 表		
	a	b	=	#		S	L	R
8				R5				
9	S9	S11						10
10				R3				
11				R4				
12	S12	S4					14	13
13			R3	R3				
14			R4	R4				
15								

LR(1) 文法

- **给定一个上下文无关文法 *G***
  - LR_1 是文法G的 LR(1) 自动机
  - A_1 *是G的action表*
  - 如果对于任意一个状态s和任意的一个终极符a, A_1(s,a)只有一个唯一的动作, 则文法G称为*LR(1)文法*;
    - Shift
    - Reduce
    - Accept
    - Error

LR(1) 分析方法



LR(1) 分析驱动程序

- **初始化: push(S0); a = readOne();**
- **L: Switch action(stack(top), a)**
  - Case *error*: error();
  - Case *accept*: return true;
  - Case *Si*: push(Si), a=readOne(); goto L;
  - Case R_P: pop(|P|);
    push(goto(stack(top), P_A));
    goto L;

如何在LR分析时生成语法分析树？

LR(1)分析过程

V_T = {a, b, =}

V_N = {S, L, R}

S = S

P:
{(1) $S \rightarrow L = R$
(2) $S \rightarrow R$
(3) $L \rightarrow aR$
(4) $L \rightarrow b$
(5) $R \rightarrow L$
}

状态栈	输入流	分析动作
0	b=b#	S4
04	=b#	R4, Goto(0,L)=5
05	=b#	S6
056	b#	S11
056(11)	#	R4, Goto(6, L)=8
0568	#	R5 , Goto(6, R)=7
0567	#	R1, , Goto(0, S)=1
01	#	Accept

5.7 LALR(1)

LR(1)分析存在的问题

- 为消除冲突，引入太多的状态;
- 有些状态含有完全相同的LR(0)项目部分，只有展望符部分是不同的;
- LR(1)项目的心：如果(A→α•β, ss) 是一个LR(1)项目，则其中的LR(0)项目部分称为它的心;
- LR(1)自动机状态的心：一个状态所含有的所有LR(1)项目的心;
- 同心状态：如果两个LR(1)状态具有相同的心，则称这两个状态为同心状态。

LALR(1) 分析

- **主要思想**
  - 合并文法G的LR(1)自动机中的同心状态，得到的自动机称为LALR(1)自动机；
  - 若这个得到的LALR(1)自动机没有冲突，则称文法G是LALR(1)文法。
- **LALR(1)分析过程**
  - 构造LALR(1)自动机
  - 构造LALR(1)分析表(同LR(1)分析表构造方法)
  - LALR(1)驱动程序 = LR(1)驱动程序

如何构造LALR(1)自动机

- **第一种途径：**
  - 首先构造LR(1)自动机
  - 然后合并其中的同心状态
  - 该方法简单，但不现实(not practical)!

Step1:构造LR(1)自动机

$V_T = \{a, b, =\}$

$V_N = \{S, L, R\}$

$S = S$

P:

{(1) $S \rightarrow L = R$

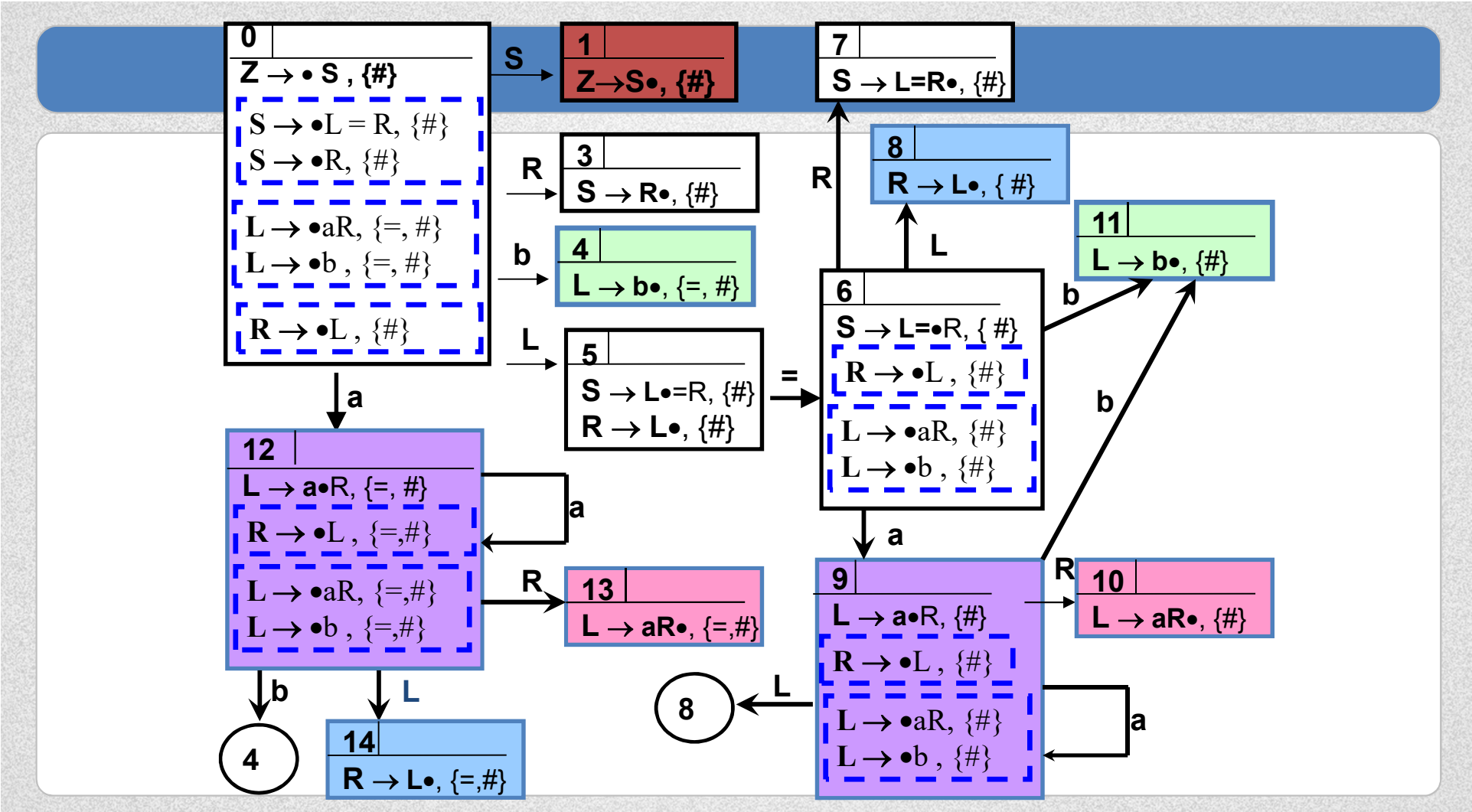
(2) $S \rightarrow R$

(3) $L \rightarrow aR$

(4) $L \rightarrow b$

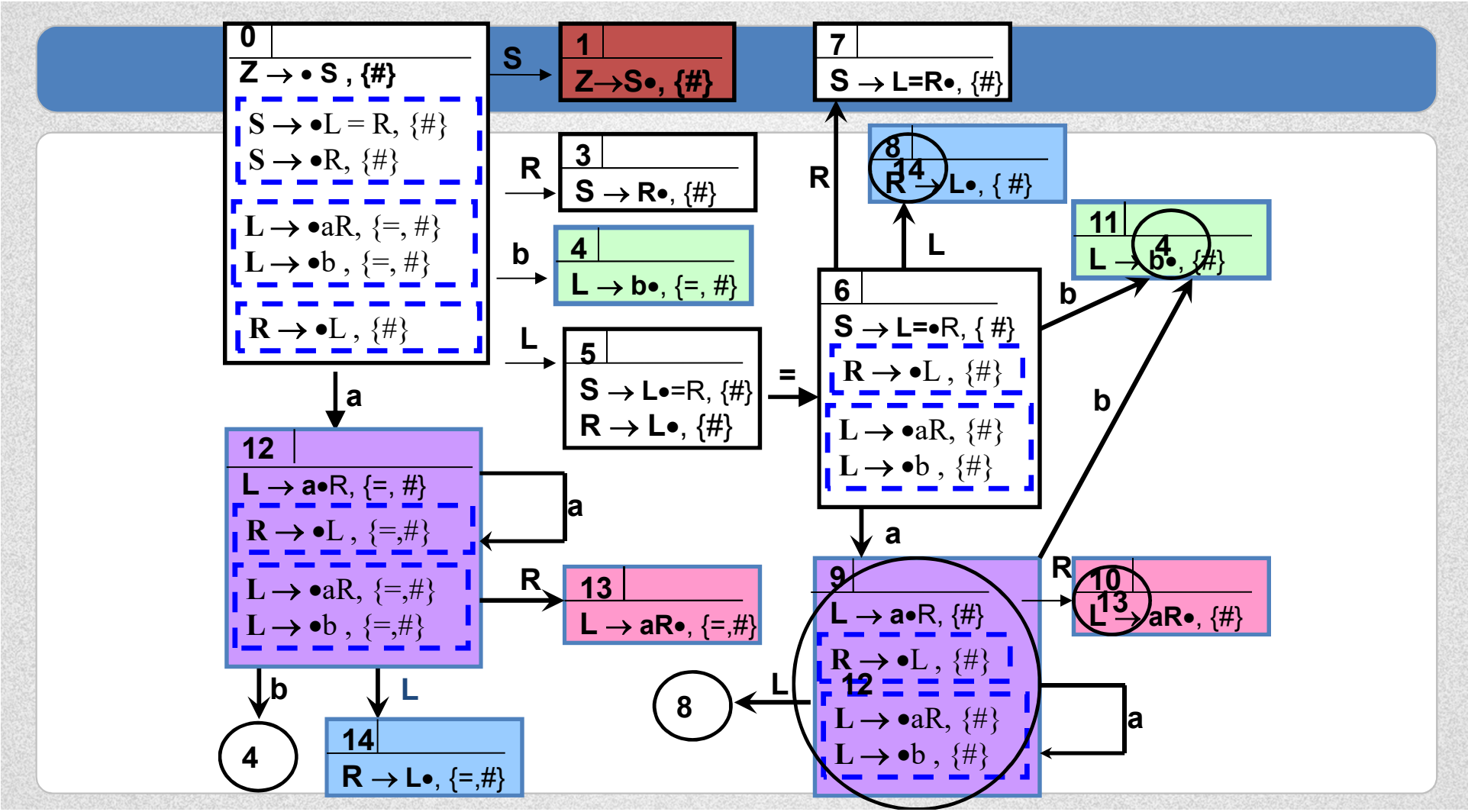
(5) $R \rightarrow L$

}


0
Z → • S , {#}
S → •L = R, {#}
S → •R, {#}
L → •aR, {=, #}
L → •b , {=, #}
R → •L , {#}
S
1
Z→S•, {#}
R
3
S → R•, {#}
b
4
L → b•, {=, #}
L
5
S → L•=R, {#}
R → L•, {#}
=
6
S → L=•R, { #}
R → •L , {#}
L → •aR, {#}
L → •b , {#}
7
S → L=R•, {#}
R
8
R → L•, { #}
L
b
11
L → b•, {#}
b
a
12
L → a•R, {=, #}
R → •L , {=,#}
L → •aR, {=,#}
L → •b , {=,#}
a
R
13
L → aR•, {=,#}
b
L
4
14
R → L•, {=,#}
a
9
L → a•R, {#}
R → •L , {#}
L → •aR, {#}
L → •b , {#}
R
10
L → aR•, {#}
L
8
a
b

Step2: 合并同心状态

- 状态4和11同心
- 状态8和14同心
- 状态10和13同心
- 状态9和12同心


0
Z → • S , {#}
S → •L = R, {#}
S → •R, {#}
L → •aR, {=, #}
L → •b , {=, #}
R → •L , {#}
S
1
Z→S•, {#}
R
3
S → R•, {#}
b
4
L → b•, {=, #}
L
5
S → L•=R, {#}
R → L•, {#}
=
6
S → L=•R, { #}
R → •L , {#}
L → •aR, {#}
L → •b , {#}
7
S → L=R•, {#}
R
8
14
R → L•, { #}
L
11
4
L → b•, {#}
b
b
a
12
L → a•R, {=, #}
R → •L , {=,#}
L → •aR, {=,#}
L → •b , {=,#}
a
R
13
L → aR•, {=,#}
a
9
L → a•R, {#}
R → •L , {#}
12
L → •aR, {#}
L → •b , {#}
R
10
13
L → aR•, {#}
b
L
8
L
b
4
L
14
R → L•, {=,#}

Step2: 合并Action表和GOTO表

算法 4.59 一个简单，但空间需求大的 LALR 分析表的构造方法。

输入： 一个增广文法 G'。

输出： 文法 G' 的 LALR 语法分析表函数 ACTION 和 GOTO。

方法：

1）构造 LR(1)项集族 $C=\{I_0, I_1, \cdots, I_n\}$。

2）对于 LR(1)项集中的每个核心，找出所有具有这个核心的项集，并将这些项集替换为它们的并集。

3）令 $C'=\{J_0, J_1, \cdots, J_m\}$ 是得到的 LR(1)项集族。状态 i 的语法分析动作是按照和算法 4.56 中的方法根据 J_i 构造得到的。如果存在一个分析动作冲突，这个算法就不能生成语法分析器，这个文法就不是 LALR(1)的。

4）GOTO 表的构造方法如下。如果 J 是一个或多个 LR(1)项集的并集，也就是说 $J=I_1 \cup I_2 \cup \cdots \cup I_k$，那么 GOTO($I_1$, X)，GOTO(I_2, X)，…，GOTO(I_k, X)的核心是相同的，因为 I_1、I_2、…、I_k 具有相同的核心。令 K 是所有和 GOTO(I_1, X)具有相同核心的项集的并集，那么 GOTO(J, X)=K。 □

LR(1) 分析表

	Action 表					Goto 表		
	a	b	=	#		S	L	R
0	S12	S4				1	5	3
1				Accept				
3				R2				
4			R4	R4				
5			S6	R5				
6	S9	S11					8	7
7				R1				

LR(1) 分析表 (接上页.)

	Action 表					Goto 表		
	a	b	=	#		S	L	R
8				R5				
9	S9	S11						10
10				R3				
11				R4				
12	S12	S4					14	13
13			R3	R3				
14			R4	R4				
15								

LALR(1) 分析表

	Action 表					Goto 表		
	a	b	=	#		S	L	R
0	S12	S4				1	5	3
1				Accept				
3				R2				
4			R4	R4				
5			S6	R5				
6	S12	S4					14	7
7				R1				
12	S12	S4					14	13
13			R3	R3				
14			R4	R4				

Step2: 合并Action表和GOTO表

- **合并Action表**
  - 出错与出错合并：结果仍为出错，无冲突；
  - 移进与移进合并；
  - 出错与移进合并：不会出现此类情况，因为出错项目与移进项目不同心
  - 移进与规约合并：不会出现此类情况(不可能引入此类冲突)，因为不可能不同心
    - 例如I1：L→ i• a；I2：L→ i•a xxx, 说明合并前已有冲突

Step2: 合并Action表和GOTO表

- **合并Action表**
  - 规约与出错合并：规定它做规约
    - 由此可见，LALR与LR(1)相比，放松了报错条件，但由于移进预测能力没有减弱，所以在下一个符号进栈前总能报错，所以对错误的定位能力没有减弱
  - 规约与规约合并：
    - 按同一个产生式规约，无冲突；
    - 按不同产生式规约，将造成冲突，因此LALR能力弱于LR(1)

Step2: 合并Action表和GOTO表

- **合并时发生规约-规约冲突，例如：语言{acd，ace，bcd，bce}**
  - 可行前缀ac的有效项集{[A⟶c ·,d],[B⟶c ·,e]}
  - 可行前缀bc的有效项集{[A⟶c ·,e],[B⟶c ·,d]}
  - 合并之后的项集为{[A⟶c ·,d/e],[B⟶c ·,d/e]}
  - 当输入为d或e时，用哪个归约?

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

综上所述，可得：只要合并同心之后，不存在按不同产生式的规约-规约冲突，由LR(1)项目集族总能构造成LALR分析表

Step2: 合并Action表和GOTO表

- **合并GOTO表**
  - 直接合并，无冲突，因为一定同心项目的GOTO表指向的项目也同心

LALR(1)构造示例2

- **示例：LALR-process.pdf**

LALR(1) 自动机

- **对于给定的上下文无关文法 *G***
  - G的LALR(1)项目跟LR(1)项目形式相同;
  - LALR(1)自动机中每个状态S中各个项目的展望符集是把LR(1)自动机中所有和S同心的状态的对应项目的展望符集合合并后得到的;
  - 如果每个LALR(1)的状态都用该状态的心(LR(0)项目)替换，则LALR(1)自动机和它的LR(0)自动机相同;
  - G的LALR(1) 自动机的状态数同LR(0)自动机的状态数相同;
  - 能不能用LR(0)自动机构造LALR(1)自动机呢?

如何构造LALR(1)自动机

- **第二种途径：**
  - 首先构造LR(0)自动机
  - 然后为每个状态的每个LR(0)项目计算展望符集；
  - 是实际应用中采用的方法！
  - 关键是如何计算展望符（向前看符号）呢？

如何构造LALR(1)自动机

- **第二种途径：**

算法 4.62 确定向前看符号。

输入：一个 LR(0) 项集 I 的内核 K 以及一个文法符号 X。

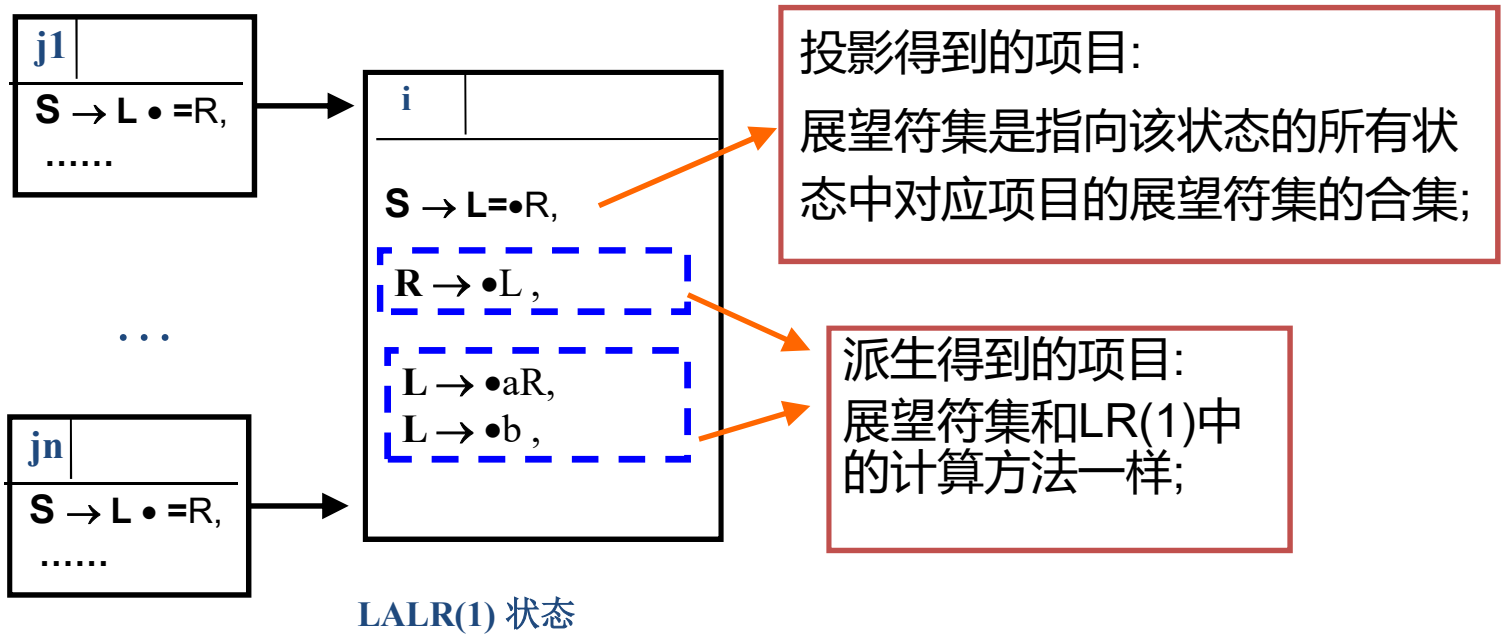
输出：由 I 中的项为 GOTO(I, X) 中内核项自发生成的向前看符号，以及 I 中将其向前看符号传播到 GOTO(I, X) 中内核项的项。

方法：算法在图 4-45 中给出。 □

```
for ( K 中的每个项 A → α·β ) {
    J := CLOSURE({[A → α·β,#]} );
    if ( [B → γ·Xδ,a] 在 J 中，并且 a 不等于 # )
        断定 GOTO(I,X) 中的项 B → γX·δ 的向前看符号 a
        是自发生成的；
    if ( [B → γ·Xδ,#] 在 J 中 )
        断定向前看符号从 I 中的项 A → α·β 传播到了 GOTO(I,X) 中的项
        B → γX·δ 之上；
}
```

图 4-45 发现传播的和自发生成的向前看符号

LALR(1)展望符的计算方法



LALR(1) 状态

构造LALR(1) 自动机

V_T = {a, b, =}

V_N = {S, L, R}

S = S

P:

{(1) $S \rightarrow L = R$

(2) $S \rightarrow R$

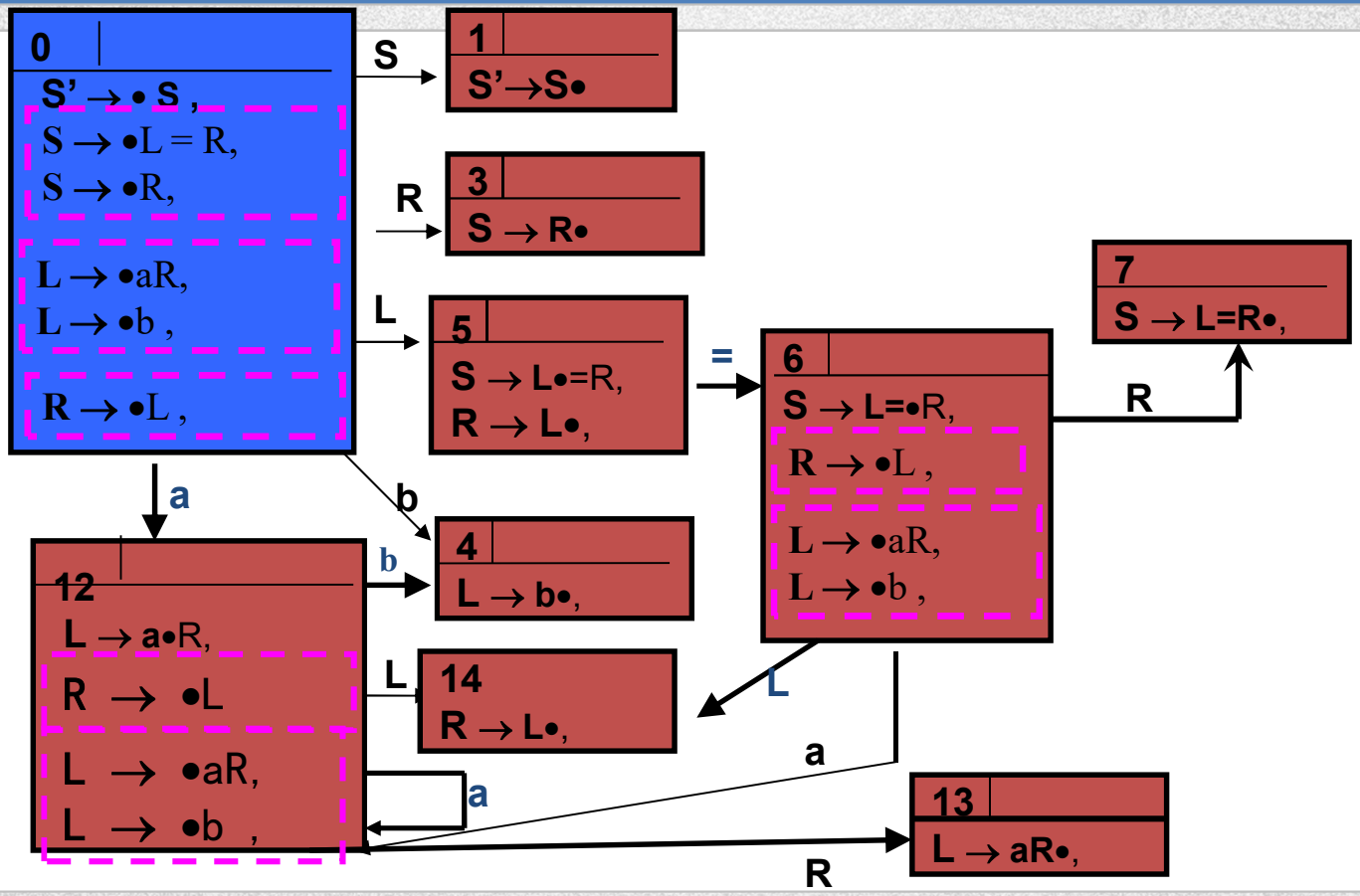
(3) $L \rightarrow aR$

(4) $L \rightarrow b$

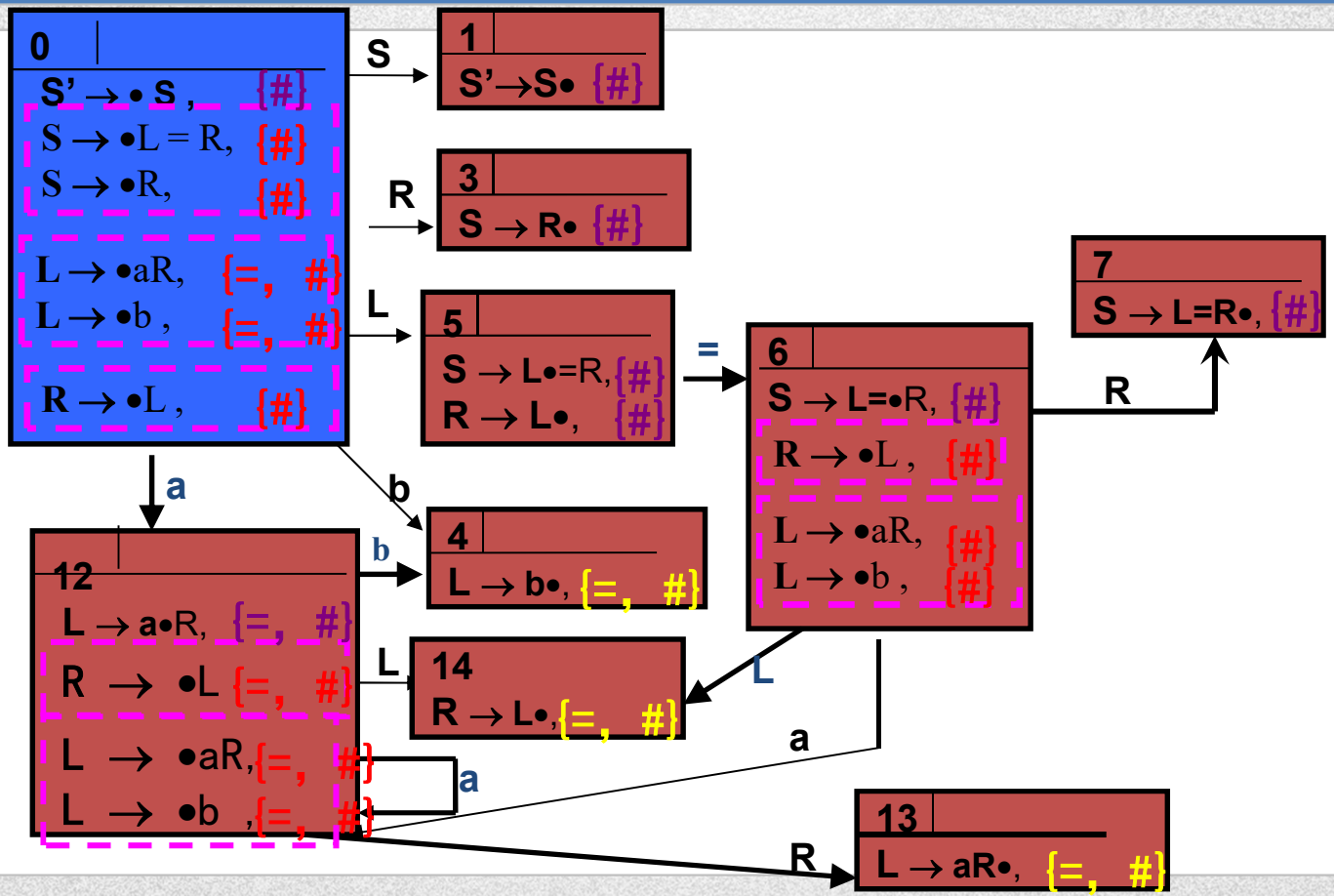
(5) $R \rightarrow L$

}

Step1: 构造LR(0) 自动机



Step2:计算展望符


0
S' → • S , {#}
S → •L = R, {#}
S → •R, {#}
L → •aR, {=, #}
L → •b , {=, #}
R → •L , {#}
1
S'→S• {#}
3
S → R• {#}
5
S → L•=R, {#}
R → L•, {#}
6
S → L=•R, {#}
R → •L , {#}
L → •aR, {#}
L → •b , {#}
7
S → L=R•, {#}
4
L → b•, {=, #}
12
L → a•R, {=, #}
R → •L {=, #}
L → •aR, {=, #}
L → •b , {=, #}
14
R → L•, {=, #}
13
L → aR•, {=, #}
S
R
L
=
R
a
b
b
L
L
a
a
R

合并可能产生的冲突

- 如果文法G的LALR(1) 自动机中没有冲突,则文法G称为*LALR(1) 文法*;
- 合并引起的冲突是指：**本来的LR(1)项集没有冲突**，而合并具有相同核心的项集后有冲突
- 合并同心状态只能产生归约-归约冲突，不会产生移入-归约冲突---已经解释过了

LALR分析器和LR分析器

- **若构造出的LALR分析表中没有冲突，则可以用LALR分析表进行语法分析**
  - 如果是正确的输入串
    - LALR和LR分析器执行完全相同的移入和归约动作序列，只是栈中的状态名字有所不同
  - 如果是错误的输入串
    - 则LALR分析器可能会比LR分析器晚一点报错，定位一样，能力一样
    - LALR分析器不会在LR分析器报错后再移入任何符号，但是可能会继续执行一些归约

LALR分析器和LR分析器

- “i=*i=#” 的LR(1)分析过程

步骤	状态栈	符号栈	输入串
0	0	#	i=*i=#
1	0，5	#i	=*i=#
2	0，2	#L	=*i=#
3	0，2，6	#L=	*i=#
4	0，2，6，12	#L=*	i=#
5	0，2，6，12，10	#L=*i	=#
	报错		

LALR分析器和LR分析器

“i=*i=#” 的LALR(1)分析过程

步骤	状态栈	符号栈	输入串
0	0	#	i=*i=#
1	0，5	#i	=*i=#
2	0，2	#L	=*i=#
3	0，2，6	#L=	*i=#
4	0，2，6，4	#L=*	i=#
5	0，2，6，4，5	#L=*i	=#
6	0，2，6，4，8	#L=*L	=#
7	0，2，6，4，7	#L=*R	=#
8	0，2，6，8	#L=L	=#
9	0，2，6，9	#L=R	=#
	报错		

LR(1)方法遇到输入串有错立即报错，LALR没有立即报错，而是多做了几步规约后才报错，但他们对错误定位能力相同，故LALR报错能力并没有减弱。

LALR分析器和LR分析器

识别能力

- $L(G) = \{c^m d c^n d \mid m, n \in \mathbb{N}\}$.
- LALR 和 LR(1) 分析法均接受正确的输入：c*dc*d.
- LR(1)DFA 的状态 I_4 接受第一个 d，状态 I_7 接受第二个 d.
- 合并后状态 I_{47} 对应的项目 $[C \to d\bullet, \$]$ 对活前缀 cd 不再是有效项目.
- 如果输入是 cd，LR(1) 分析法将在状态 I_4，面对输入 \$，分析器将立刻报错.
- 如果输入是 cd，LALR 分析法将在状态 I_{47}，面对输入 \$，分析器将在两步归约操作后报错.

文法 G

1/ $S \to Cc$ 2/ $C \to cC$ 3/ $C \to d$

LR(1) 分析法

stack			action	rest
I_0	I_3	I_4	error	\$
\$	c	d		

LALR 分析法

stack			action	rest
I_0	I_{36}	I_{47}	reduce	\$
\$	c	d		
I_0	I_{36}	I_{89}	reduce	\$
\$	c	C		
I_0	I_2		error	\$
\$	C			

5.8 综合比较

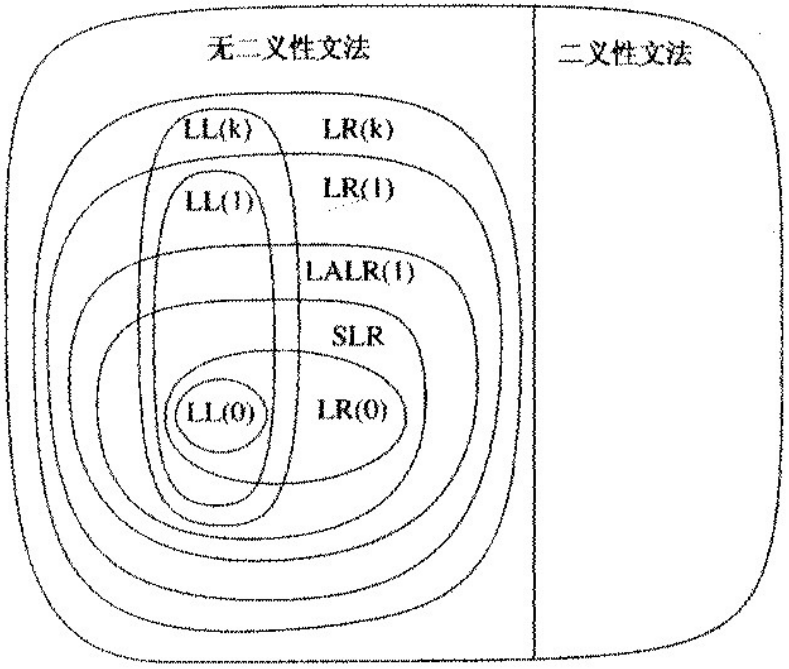
综合比较

- **状态数:**
  - 对LR(1)项集规范族中所谓的同心项集进行合并，从而使得分析表既保持了LR(1)项中向前看符号的信息，又使状态数减少到与SLR分析表的一样多
  - LR(1) > LALR(1) = SLR(1) = LR(0)
- **展望符的确定:**
  - LR(0)没有展望符;
  - SLR(1)取follow集;
  - LR(1)取不同位置的follow集
  - LALR(1)取同心项的展望符的并集;
- **向前看输入符:**
  - SLR(1),LR(1)和LALR(1)向前看一个输入符;
  - LR(0)不看;
- **分析能力: LR(1) ⊃ LALR(1) ⊃ SLR(1) ⊃ LR(0)**

综合比较

- 如果原来LR(1) DFA规范项目集中没有移进-规约冲突，则LALR(1)项目集不可能有移进-规约冲突；即，如果LALR(1)项目集有移进-规约冲突，则LR(1) 一定也有相同的S-R冲突
- 但是LALR(1)可能出现R-R冲突
- LALR分析表移进操作与LR(1)和SLR一样，唯一不同的在于归于操作
- LALR(1) 可能解决SLR的S-R冲突，例如复制表达式文法是LALR文法，而不是SLR文法 --- 绝大多数程序语言是LALR文法

综合比较



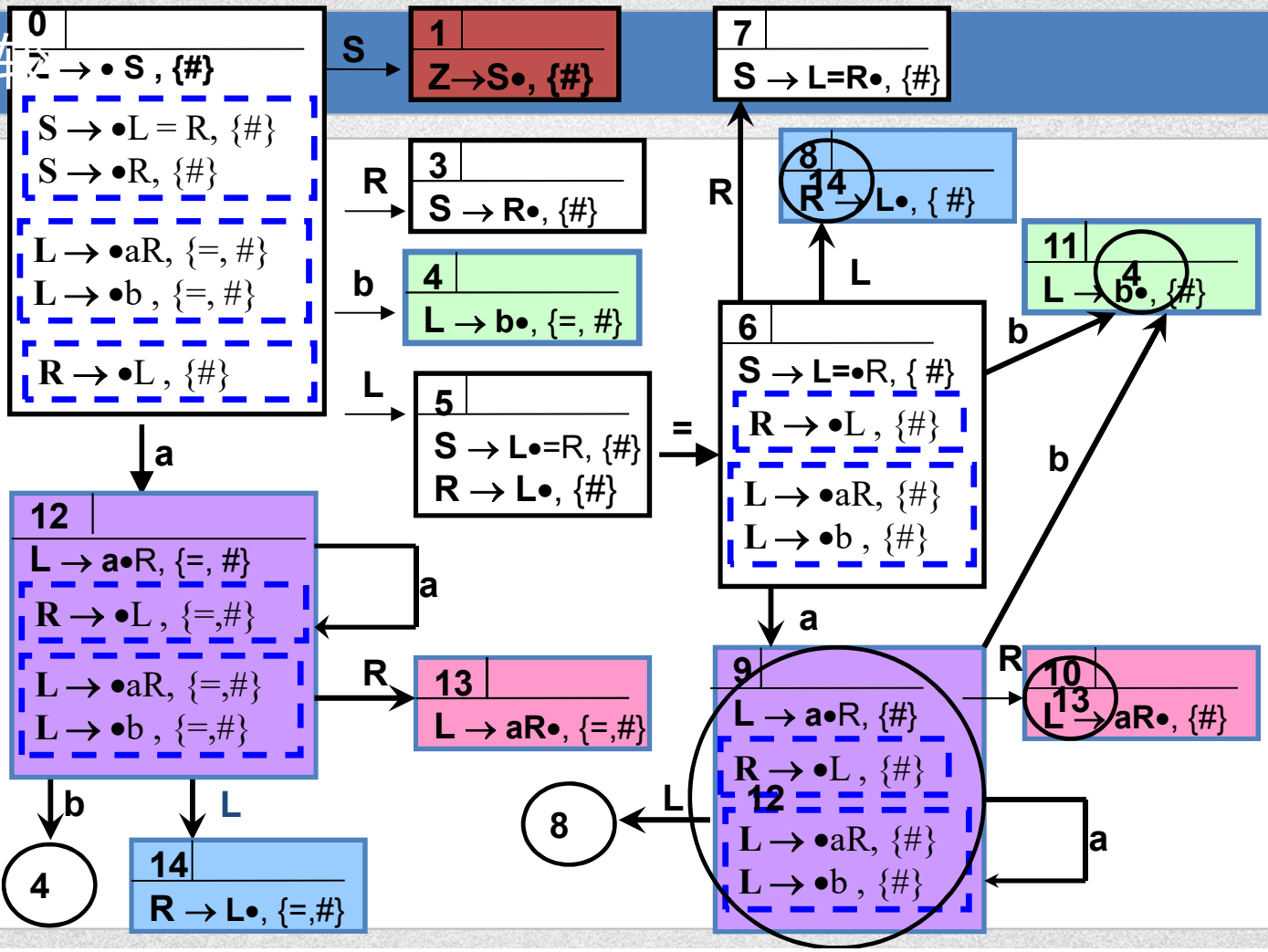
LR(1)-LALR(1) = 规约-规约冲突的情况

综合比较

Example of LALR gammar, but not SLR grammar

- $$G: \begin{cases} S \rightarrow Aa \mid bAc \mid dc \mid bda \\ A \rightarrow d \end{cases}$$
- LR(0) 规范项目集中有：$I = \{S \rightarrow d \bullet c,\ A \rightarrow d\bullet\}$，并且，$c \in Follow(A)$，所以有移进 - 归约冲突.
- LALR 项目集中对应的是：$I = \{[S \rightarrow d \bullet c, \$],\ [A \rightarrow d\bullet, a]\}$，避免了上述冲突，所以是 LALR 文法.

综合比较


0
Z → • S , {#}
S → •L = R, {#}
S → •R, {#}
L → •aR, {=, #}
L → •b , {=, #}
R → •L , {#}
S
1
Z→S•, {#}
R
3
S → R•, {#}
b
4
L → b•, {=, #}
L
5
S → L•=R, {#}
R → L•, {#}
=
6
S → L=•R, { #}
R → •L , {#}
L → •aR, {#}
L → •b , {#}
7
S → L=R•, {#}
R
8
14
R → L•, { #}
L
11
4
L → b•, {#}
b
a
12
L → a•R, {=, #}
R → •L , {=,#}
L → •aR, {=,#}
L → •b , {=,#}
a
R
13
L → aR•, {=,#}
b
L
4
14
R → L•, {=,#}
a
9
L → a•R, {#}
R → •L , {#}
12
L → •aR, {#}
L → •b , {#}
R
10
13
L → aR•, {#}
b
L
8
a

综合比较

Example of LR(1) gammar, but not LALR grammar

- $$G: \begin{cases} S \rightarrow aAa \mid bBb \mid aBb \mid bAa \\ A \rightarrow c \\ B \rightarrow c \end{cases}$$
- LR(1) 规范项目集中有：$I = \{[A \rightarrow c\bullet, a], [B \rightarrow c\bullet, b]\}$，
  $I = \{[A \rightarrow c\bullet, b], [B \rightarrow c\bullet, a]\}$，没有冲突.
- 合并后，LALR 项目集中对应的是：
  $IJ = \{[A \rightarrow c\bullet, a/b], [B \rightarrow c\bullet, a/b]\}$，产生了 R-R 冲突.

综合比较

Example of LL(1) gammar, but not SLR grammar

- $$G: \begin{cases} S \rightarrow AaAb \mid BbBa \\ A \rightarrow \varepsilon \\ B \rightarrow \varepsilon \end{cases}$$
- G 是 LL(1) 文法.
- G 不是 SLR 文法，存在规范 LR(0) 项目集：$I = \{S \rightarrow \bullet AaAb, S \rightarrow \bullet BbBa, A \rightarrow \bullet, B \rightarrow \bullet\}$，有两个归约项目 $A \rightarrow \bullet, B \rightarrow \bullet$，而 $a, b \in Follow(A) = Follow(B)$，从而产生 R-R 冲突.
- 还存在 LL(1) 文法，但不是 LALR 文法.

作业

- **教材P177：4.7.4，4.7.5**

5.9 Error Handling

语法错误的处理

- **错误难以避免**
- **编译器需要具有处理错误的能力**
- **程序中可能存在不同层次的错误**
  - 词法错误；语法错误；语义错误；逻辑错误
- **语法错误相同容易发现，语义和逻辑错误较难精确的检测到**
- **语法分析器中错误处理程序的设计目标**
  - 清晰准确地**报告**出现错误，并指出错误的**位置**
  - 能从当前错误中**恢复**，以继续检测后面的错误
  - 尽可能减少处理正确程序的开销

预测分析中的错误恢复

- **错误恢复**
  - 当预测分析器报错时，表示输入的串不是句子
  - 对于使用者而言，希望预测分析器能够进行恢复处理后继续语法分析过程，以便在一次分析中找到更多的语法错误
  - 但是有可能恢复得并不成功，之后找到的语法错误有可能是假的
  - 进行错误恢复时可用的信息：栈里面的符号，待分析的符号
- **两类错误恢复方法**
  - 恐慌模式
  - 短语层次的恢复

基本思想

- 分析表对应的 Action 空白项表示出错，
- 一般处理方法：
  - 如果出错状态含有项目 $A \to \alpha \bullet \beta$，表示希望形成形如``$\alpha\beta$''的句柄，但是当前的错误输入使得分析器不能移进，此时可采用类似 LL 分析的方法，跳过当前的输入直到非终结符 A 的同步符号出现，用 $A \to \alpha\beta$ 归约继续分析.
  - 如果出错状态含有项目 $A \to \alpha\bullet$，表示当前的输入不是 A 的 Follow 集元素，处理方法，归约该项目，转入和前者一样的处理.

例子

- **E→E+E, E→ E*E, E→(E), E→ id**

	Action						Goto
状态	id	+	*	(	)	$	E
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	r2		r3	r3	

I_0:
$E' \to \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_1:
$E' \to E\cdot$
$E \to E\cdot + E$
$E \to E\cdot * E$

I_2:
$E \to (\cdot E)$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_3:
$E \to \mathbf{id}\cdot$

I_4:
$E \to E + \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_5:
$E \to E * \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_6:
$E \to (E\cdot)$
$E \to E\cdot + E$
$E \to E\cdot * E$

I_7:
$E \to E + E\cdot$
$E \to E\cdot + E$
$E \to E\cdot * E$

I_8:
$E \to E * E\cdot$
$E \to E\cdot + E$
$E \to E\cdot * E$

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

例子

e1：这个例程在状态0、2、4和5上被调用。所有这些状态都期望读入一个运算分量的第一个符号，这个符号可能是 **id** 或左括号，但是实际读入的却是 + 、 * 或输入结束标记。

将状态3(状态0、2、4和5在输入 **id** 上的 GOTO 目标)压入栈中；

发出诊断信息“缺少运算分量。”

e2：在状态0、1、2、4和5上发现输入为右括号时调用这个过程。

从输入中删除右括号；

发出诊断信息“不匹配的右括号。”

e3：当在状态1和6上，期待读入一个运算符却发现了一个 **id** 或左括号时调用。

将状态4(对应于符号 + 的状态)压入栈中。

发出诊断信息“缺少运算符。”

e4：当在状态6上发现输入结束标记时调用。

将状态9(对应于右括号)压入栈中；

发出诊断信息“缺少右括号。”

在处理错误的输入 **id** +)时，语法分析器进入的格局序列显示在图4-54中。 □

4.8.4 4.8节的练习

	Action						Goto
状态	id	+	*	(	)	$	E
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	r2		r3	r3	

I_0:
$E' \to \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_1:
$E' \to E\cdot$
$E \to E\cdot + E$
$E \to E\cdot * E$

I_2:
$E \to (\cdot E)$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_3:
$E \to \mathbf{id}\cdot$

I_4:
$E \to E + \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_5:
$E \to E * \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_6:
$E \to (E\cdot)$
$E \to E\cdot + E$
$E \to E\cdot * E$

I_7:
$E \to E + E\cdot$
$E \to E\cdot + E$
$E \to E\cdot * E$

I_8:
$E \to E * E\cdot$
$E \to E\cdot + E$
$E \to E\cdot * E$

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

例子

分析表的出错处理

1/ E → E+E　2/ E → E*E　3/ E → (E)　4/ E → id

	Action						Goto
状态	id	+	*	(	)	$	E
0	s3	e1	e1	s2	e2	e1	1
1	e3	s4	s5	e3	e2	acc	
2	s3	e1	e1	s2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	s3	e1	e1	s2	e2	e1	7
5	s3	e1	e1	s2	e2	e1	8
6	e3	s4	s5	e3	s9	e4	
7	r1	r1	s5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r2	r3	r3	r3	

例子

- **分析过程：error-handling.pdf**

5.10 YACC

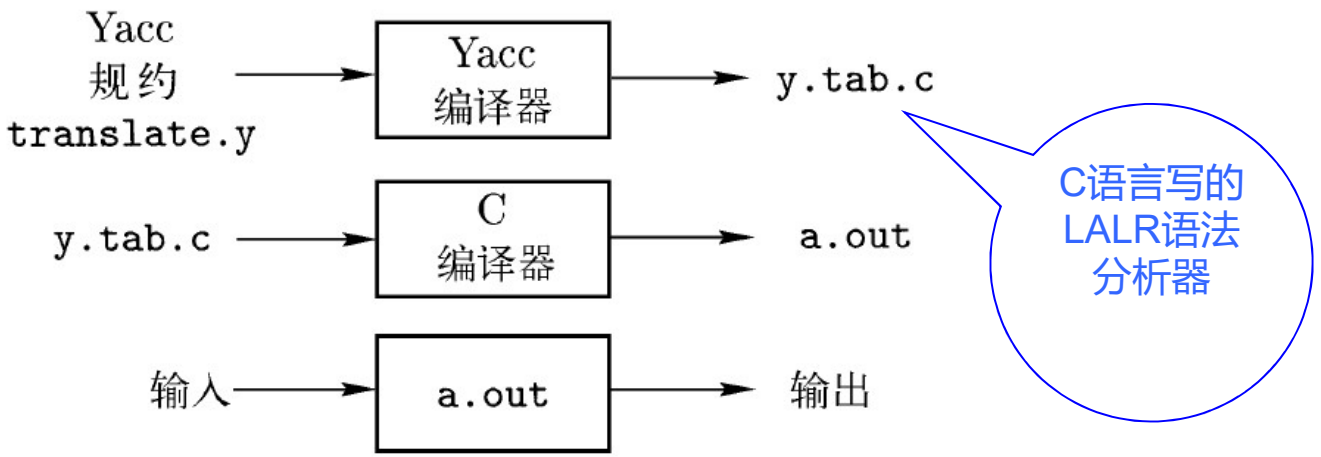
自下而上语法分析器生成工具

Examples

1. yacc(Yet Another Compiler-Compiler): 1975 年由贝尔实验室 Mike Lesk & Eric Schmidt 开发，UNIX 标准实用工具 (utility);
2. byacc: Berkeley YACC: Robert Corbett, 1989 年, yacc compatible, in Free BSD distribution, DOS version in my CD-ROM;
3. bison: Robert Corbett & Richard Stallmen, 1988 年，yacc compatible, in Linux distribution, 最新版本：2.4. 支持 GLR(Generalized LR) 文法，http://www.gnu.org/software/bison/.
4. CUP: LALR Parser Generator in Java, current version 0.11a, 对应的词法分析器生成工具为：JFLex(http://jflex.de/), http://www2.cs.tum.edu/projects/cup/.
5. A. Holub LRpars: See CD-ROM, 支持动态显示分析过程.

语法分析器生成工具YACC

- **YACC的使用方法如下：**



YACC源程序的结构

- **声明**
  - 分为可选的两节：第一节放置C声明，第二节是对词法单元的声明。
- **翻译规则：**
  - 指明产生式及相关的语义动作
- **辅助性C语言例程**
  - 被直接拷贝到生成的C语言源程序中，
  - 可以在语义动作中调用。
  - 其中必须包括yylex()。这个函数返回词法单元，可以由LEX生成

声明
%%
翻译规则
%%
辅助性C语言程序

翻译规则的格式

<产生式头> ： <产生式体>1 {<语义动作>1}
| <产生式体>2 {<语义动作>2}
… … … …
| <产生式体>n {<语义动作>n}
;

- 第一个产生式的头被看作开始符号；
- 语义动作是C语句序列；
- $$表示和产生式头相关的属性值，$i表示产生式体中第i个文法符号的属性值。
- 当我们按照某个产生式归约时，执行相应的语义动作。通常可以根据$i来计算$$的值。
- 在YACC源程序中，可以通过定义YYSTYPE来定义$$，$i的类型。

YACC源程序的例子

```
%{
#include <ctype.h>
%}

%token DIGIT

%%
line   : expr '\n'        { printf("%d\n", $1); }
       ;
expr   : expr '+' term    { $$ = $1 + $3; }
       | term
       ;
term   : term '*' factor  { $$ = $1 * $3; }
       | factor
       ;
factor : '(' expr ')'     { $$ = $2; }
       | DIGIT
       ;
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

作业

- **课程设计：**
  - 期末考试前一天晚上18:00之前提交 （超过ddl提交记为0分）

Thank you!