4.5 自底向上语法分析

- “移进—归约”分析方法，shift-reduce
- 将符号串归约为文法开始符号
- 算法优先分析法、LR分析法
- 每个归约步骤
  - 一个特定的子符号串匹配某个产生式的右部
  - 将这个子符号串替换为产生式左部的NT

例4.21

- 文法　　S □ aABe

　　A □ Abc | b

　　B □ d

- 归约　　abbcde

　　aAbcde

　　aAde

　　aABe

　　S

- 对应最右推导

S $\underset{rm}{\square}$ aABe $\underset{rm}{\square}$ aAde $\underset{rm}{\square}$ aAbcde $\underset{rm}{\square}$ abbcde　　rm

4.5.1 句柄（Handle）

- 符号串的“句柄”
  - 与某个产生式右部匹配的子串
  - 归约为产生式左部□□最右推导逆过程
- 形式化定义：一个最右句型γ的句柄
  - <A□□, □中□的位置>
  - $S \underset{rm}{\Rightarrow^*} \square Aw \Rightarrow_{rm} \square\square w$，□中□之后即为句柄位置，w只包含终结符
- 一个句型可有多个不同句柄
- 非多义性文法的最右句型有唯一句柄

例4.22

(1) E □ E + E　　(3) E □ (E)

(2) E □ E * E　　(4) E □ **id**

$E \underset{rm}{\Rightarrow} \underline{E + E}$

$\underset{rm}{\Rightarrow} E + \underline{E * E}$

$\underset{rm}{\Rightarrow} E + E * \underline{\mathbf{id}_3}$

$\underset{rm}{\Rightarrow} E + \underline{\mathbf{id}_2} * \mathbf{id}_3$

$\underset{rm}{\Rightarrow} \underline{\mathbf{id}_1} + \mathbf{id}_2 * \mathbf{id}_3$

$E \underset{rm}{\Rightarrow} \underline{E * E}$

$\underset{rm}{\Rightarrow} E + \underline{\mathbf{id}_3}$

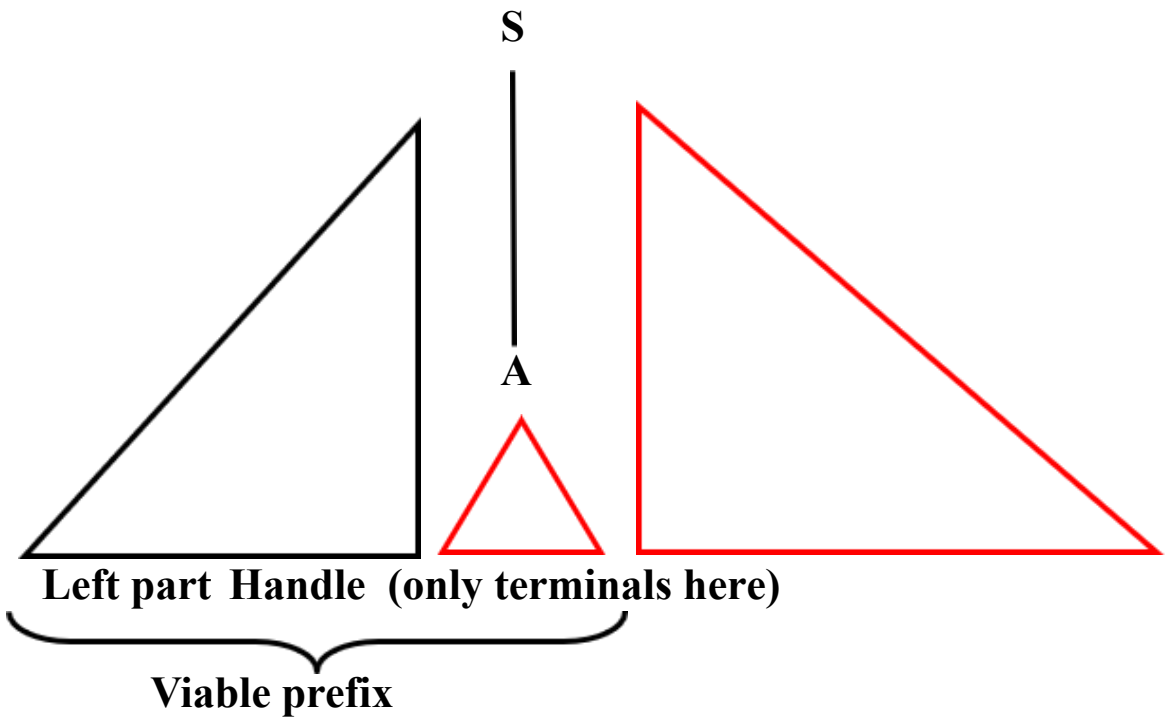
$\underset{rm}{\Rightarrow} \underline{E + E} * \mathbf{id}_3$

$\underset{rm}{\Rightarrow} E + \underline{\mathbf{id}_2} * \mathbf{id}_3$

$\underset{rm}{\Rightarrow} \underline{\mathbf{id}_1} + \mathbf{id}_2 * \mathbf{id}_3$

4.5.2 句柄剪枝(Handle Pruning)

- 子串□产生式左部——语法树剪枝
- 不断剪枝□最右推导的逆过程——归约



例4.23

最右句型	句柄	归约产生式
$\mathbf{id_1} + \mathbf{id_2} * \mathbf{id_3}$	$\mathbf{id_1}$	E □ **id**
$E + \mathbf{id_2} * \mathbf{id_3}$	$\mathbf{id_2}$	E □ **id**
$E + E * \mathbf{id_3}$	$\mathbf{id_3}$	E □ **id**
$E + E * E$	$E * E$	E □ E * E
$E + E$	$E + E$	E □ E + E
E		

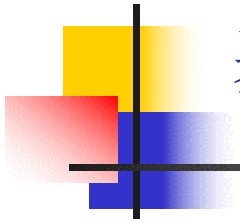
4.5.3 移进－归约分析的实现

- 两个问题
  - 定位句柄
  - 确定用哪个产生式进行归约
- 一般实现方法——栈

1. 将输入符号“移进”栈，直至在栈顶形成一个句柄
2. 将句柄“归约”为相应的非终结符
3. 不断重复，直至栈中只剩开始符号，输入缓冲区为空——接受输入串
4. 错误处理

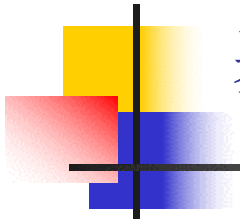
例4.24

栈	输入	动作
$	$id_1 + id_2 * id_3$	移进
$ id_2	$+ id_2 * id_3$	归约E → id
$E	$+ id_2 * id_3$	移进
$E +	$id_2 * id_3$	移进
$E + id_2	$* id_3$	归约E → id
$E + E	$* id_3$	移进
$E + E *	id_3	移进
$E + E * id_3	$	归约E → id
$E + E * E	$	归约E → E * E
$E + E	$	归约E → E + E
$E	$	接受



基本操作

1. 移进（shift）
   - 下个输入符号移到栈顶
2. 归约（reduce）
   - 句柄的右端恰在栈顶，定位句柄左端
   - 确定选用的产生式，用产生式左部非终结符替换栈中的句柄



基本操作

3. 接受（accept）

- 宣布分析成功结束

4. 错误（error）

- 发现语法错误，调用错误恢复函数

句柄必然在栈顶？

- 移进动作在栈内形成句柄
  - 这显然不可能

    若形成($αβy，z$)局面时β成为句柄

    则形成($αβ，yz$)局面时β已可成为句柄
- 归约动作在栈内形成句柄
  - ($αβγδ，y$)归约为($αβγB，y$)后形成句柄β，再

    归约为($αAγB，y$)

    则存在对应推导$S \Rightarrow^{*} \alpha A\gamma By \Rightarrow \alpha\beta\gamma By \Rightarrow^{*} \alpha\beta\gamma\delta y$

    这显然不是一个最右推导，矛盾！

4.5.4 活前缀

- Viable Prefix
- 最右句型的前缀
  - 其右端不会在最右句柄的右端之后
  - 移进—归约分析过程中可出现在栈中
  - 总是可能通过向其末端添加若干终结符，形成最右句型

4.5.5 冲突（例4.25）

- 移进/归约冲突，归约/归约冲突

stmt □ **if *expr* then *stmt***

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

| **other (any other statement)**

Stack Input

$… **if** expr **then** stmt **else** …$

- 归约**if** expr **then** stmt归约为stmt？或是

  移进**else**，归约**if** expr **then** stmt **else** stmt？

  移进/归约冲突

- 解决方法——强制一种动作

例4.26

(1)***stmt*** □ **id (*parameter-list*)**

(2) ***stmt*** □ ***expr* := *expr***

(3) ***parameter-list*** □ ***parameter-list* , *parameter***

(4,5) ***parameter-list*** □ ***parameter*** | **id**

(6,7) ***expr*** □ **id (*expr-list*)** | **id**

(8) ***expr-list*** □ ***expr-list* , *expr*** | ***expr***

- 符号串“A(I, J)”，单词流**id(id, id)**

Stack	Input
$… **id (id**	**, id)** …$

- 归约(5)—过程参数？归约(7)—数组下标？

  归约/归约冲突

- 解决：(1) **id□procid**，区分过程/数组

4.6 算符优先分析方法

- operator-precedence parsing
- 算符文法
  - 无ε产生式
  - 产生式右部不含连续NT
- 例4.27

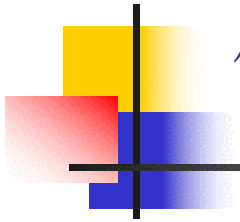
E → EAE | - E | (E) | id

A → - | + | * | / | ↑

E → E - E | E + E | E * E | E ↑ E | E ^ E | - E | (E) | id

算符优先分析方法的特点

- 缺点
  - 多优先级算符例如‘-’难以处理
  - 文法——分析器关系不紧密
  - 仅适用少数文法
- 优点
  - 简单
  - 适合表达式分析



优先级

- 不相交的三种优先级关系—终结符之间
  - a<b，a的优先级低于b
  - a≐b，a的优先级等于b
  - a>b，a的优先级高于b
- 不同于算术关系<、=、 >
  - a<b，a>b可能同时成立
  - 三种关系可能均不成立
- 确定优先关系
  - 传统意义运算符的优先级：+<*，*>+

使用算法计算优先关系

- 非二义性文法，语法树反映优先关系
- 不含ε产生式文法，对任一对终结符a, b

1. a≐b，当且仅当P□…ab…或P□…aQb…
2. a⋖b，当且仅当P□…aR…，且R□b…或R□Qb…
3. a⋗b，当且仅当P□…Rb…，且R□…a或R□…aQ

若算符文法G任何终结符对(a,b)至多只满足

a⋖b，a≐b，a⋗b

之一，则称之为算符优先文法

例

(1) E □ E + T | T

(2) T □ T * F | F

(3) F □ (E) | id

- 传统意义：+⋖*；+ ⋗+；(⋖ +,*；) ⋗ +,*
- 算法计算
  - 由(3)，(≒)（规则1）
  - E□E + T，T □ T * F，+ ⋖ *（规则2）
  - E□E + T，E □ E + T，+ ⋗ +（规则3）
  - F□(E)，E □ E +T， + ⋗)

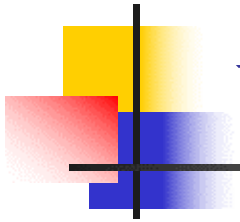
4.6.1 使用算符优先关系

- 目的：确定最右句型的句柄的边界
- ⋖——开始，≐——内部，⋗——结束
- 产生式右部无连续NT——句型同样
- $□_0a_1□_1a_2□_2 \cdots a_n□_n$：$□_i$为ε或单个非终结符，$a_i$为单个终结符
- 去掉所有NT，在相邻T之间（包括$）填写算符优先关系

使用算符优先关系（续）

	id	+	*	$
id		⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	

- **$id+id*id$□$⋖id⋗+⋖id⋗*⋖id⋗$**
- $E + E * E$□ $⋖ + ⋖ * ⋗$



寻找句柄

- 由左至右扫描，寻找第一个>
- 向回（左）扫描，略过=，直至找到<
- <、>之间（不包含）的部分即为句柄
- **$id+id*id$□$<id>+<id>*<id>$—id**
- $E + E * E$□ $< + < * >$—E * E
- 非终结符——占位符
- 使用栈实现非常容易

算法4.5 算符优先分析算法

输入：一个输入串w和一个优先关系表

输出：如果w符合语法，生成语法树框架（所有内部节点表示为占位符）；否则，输出错误

方法：

设置ip指向w$的第一个符号;

repeat forever

if (栈顶为$且ip指向$)

return;

else {

算法4.5（续）

令a为最接近栈顶的终结符，b为ip指向符号

if (a<b 或 a≐b) {

b压栈;

ip前移;

} **else if** (a>b) {

repeat

栈中符号弹出

until 栈顶符号<刚刚弹出符号

} **else** error();

}

例

STACK	INPUT	Remark
$	id + id * id$	$ ⋖ id
$ id	+ id * id$	id ⋗ +
$	+ id * id$	$ ⋖ +
$ +	id * id$	+ ⋖ id
$ + id	* id$	id ⋗ *
$ +	* id$	+ ⋖ *
$ + *	id$	* ⋖ id
$ + * id	$	id ⋗ $
$ + *	$	* ⋗ $
$ +	$	+ ⋗ $
$	$	accept

一系列的弹出操作

——利用某个产生式进行归约

4.6.2 直接构造算符优先关系(表)

- 在理解文法含义情况下直接构造优先关系

1. 若θ_1优先级高于θ_2， $\theta_1 \gtrdot \theta_2$，$\theta_2 \lessdot \theta_1$
2. 若θ_1优先级等于θ_2（或相同的算符）， 若左

   结合，$\theta_1 \gtrdot \theta_2$，$\theta_2 \gtrdot \theta_1$，

   若右结合，$\theta_1 \lessdot \theta_2$，$\theta_2 \lessdot \theta_1$
3. $\theta \lessdot (，(\lessdot \theta，)\gtrdot \theta，\theta \gtrdot)，(\doteq)，(\lessdot (，)\gtrdot)$
4. **id**的优先级高于任何其他符号
5. $具有最低的优先级

例4.28

	+	-	*	/	↑	id	(	)	$
+	⋗	⋗	⋖	⋖	⋖	⋖	⋖	⋗	⋗
-	⋗	⋗	⋖	⋖	⋖	⋖	⋖	⋗	⋗
*	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
/	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
↑	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
id	⋗	⋗	⋗	⋗	⋗			⋗	⋗
(	⋖	⋖	⋖	⋖	⋖	⋖	⋖	≐	
)	⋗	⋗	⋗	⋗	⋗			⋗	⋗
$	⋖	⋖	⋖	⋖	⋖	⋖	⋖		

4.6.3 处理一元算符

- 普通一元算符，¬
  - 对任何算符θ，θ<¬
  - ¬优先级高于θ，¬>θ；否则，¬<θ
- 既是一元算符，也是二元算符，–
  - 词法分析器返回不同单词
  - 仅向前搜索不够，需记住前面符号

4.6.4 优先级函数

- 每个终结符对应两个整型函数
  - f(a) < g(b)□a⋖b
  - f(a) = g(b)□a≐b
  - f(a) > g(b)□a⋗b
  - 错误无法表示，不严重□归约时仍可捕获
  - 不是所有优先关系（表）都存在优先级函数

例4.29

	+	-	*	/	↑	id	(	)	$
f	2	2	4	4	4	6	0	6	0
g	1	1	3	3	5	5	5	0	0

	+	-	*	/	↑	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
↑	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>			>	>
(	<	<	<	<	<	<	<	≐	
)	>	>	>	>	>			>	>
$	<	<	<	<	<	<	<		

算法4.6 构造优先级函数

输入：一个算符优先关系矩阵

输出：对应的优先级函数，或返回不存在信息

方法：

1. 对每个非终结符a或$创建符号$f_a$和$g_a$
2. 将创建的符号划分为尽量多的组：
   - 若a≐b，则f_a和g_b分在同一组
   - 没有≐关系，也可能分在一组
     - 如a≐b，c≐b，则f_a和f_c会在一组
     - 若还有c≐d，f_a和g_d会在一组，而a≐d不一定成立

算法4.6（续）

3. 创建有向图

- 节点——符号分组
- 对任何a、b
  - a<b，g_b所在组向f_a所在组引一条边
  - 若a>b，f_a所在组向g_b所在组引一条边
  - f_a到g_b的边（路径）：f(a)必须大于g(b)，反之亦然。

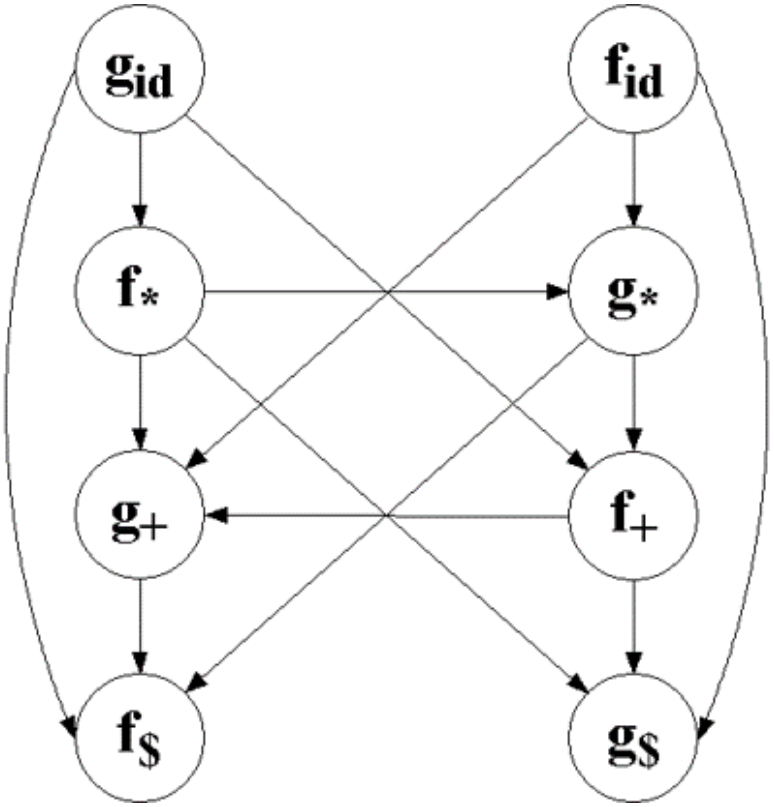
4. 利用有向图获得优先函数

- 有向图存在回路□优先级函数不存在
- 否则
  - $f(a) = f_a$所在组开始的最长路径长度
  - $g(a) = g_a$所在组开始的最长路径长度

例4.30

	id	+	*	$
id		⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	

	id	+	*	$
f	4	2	4	0
g	5	1	3	0



4.6.5 算符优先分析的错误处理

- 两种错误
  1. 栈顶和当前输入两个终结符无优先关系
  2. 句柄与任何产生式右部都不相同
- 只有这两个错误
  3. 考虑算法4.5，总会找到<·——栈底为$，优先级低于任何终结符
  4. 压栈过程中相邻符号必须满足<·、≐，因此，算法肯定可以找到句柄，进行归约

处理归约错误

- 错误诊断信息——句柄“像”哪个产生式右部？
- 句柄abc
  - 右部aEcE：“非法的b”
  - abEdc：“缺少d”
  - 终结符匹配，非终结符不匹配，句柄中无非终结符，右部aEbc：“缺少E”

难点

- 有穷或无穷个符号串“可被弹出”？

  $b_1 \doteq b_2 \doteq \cdots \doteq b_k$

- 有穷，预先计算符号串“最相像”右部
- 有向图
  - 节点□终结符
  - a到b的边□$a \doteq b$
  - 路径□“可被弹出”的符号串
- 开始、结束符号，$a \lessdot b_1$，$b_k \gtrdot c$
- 回路——无穷多个符号串；否则，有穷

例4.31

- **E □ E - E | E + E | E * E | E ↑ E | E ^ E | - E |**
  (E) | id
- (≒)，)外均可作开始符号，(外均可做结束符号
- 长度为1路径：+, -, *, /, ↑, **id**，长度为2：
  ( )

例4.31

- 错误检测
  - 若归约+, -, *, /, ↑，检查两侧是否都有非终结符，若不是，错误信息“缺少运算数”
  - 若归约**id**，检查左侧或右侧没有非终结符。若不是，“缺少运算符”
  - 若归约( )，检查括号之间存在一个非终结符。若不是，“括号间缺少表达式”
- 无穷情况，不能逐个情况考虑，通用函数

处理移进/归约错误

- 栈顶、输入两个终结符无优先关系
- 修改栈、输入——避免无限循环
- 策略一：保证移进
  - 栈顶ab，输入cd，b、c无优先关系
  - a≤c□删除b
  - b≤d□删除c
  - 插入e，使b≤e≤c，一般：$b \leq e_1 \leq e_2 \cdots \leq e_n \leq c$

例4.32

	id	(	)	$
id	e3	e3	⋗	⋗
(	⋖	⋖	≐	e4
)	e3	e3	⋗	⋗
$	⋖	⋖	e2	e1

1. e1：插入**id**□输入缓冲，“缺少运算数”
2. e2：删除)，“未匹配右括号”
3. e3：插入+□输入缓冲，“缺少运算符”
4. e4：弹出(，“缺少右括号”

例4.32（续）

id +)

Stack	Input
$E +	)$

- + ⋗)□归约，+右侧无非终结符□

“缺少运算数”

Stack	Input
$E	)$

- $、)无优先关系□“未匹配)”

Stack	Input
$E	)$

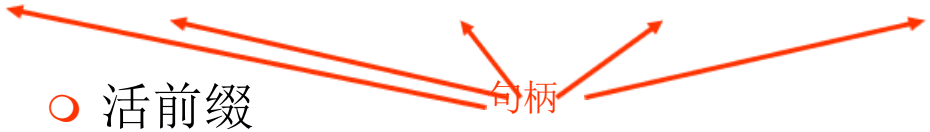
4.7 LR分析方法

例：**S □ aABe**
A □ Abc | b

B □ d

abbcde的最右推导：

S□ aABe □ aAde □ aAbcde □ abbcde



- 活前缀

1. 最右句型的前缀，不超过唯一句柄
2. a, aA, aAd, aAbc, ab, aAb
3. aAde, Abc, aAA不是

移进/归约分析过程

STACK	INPUT	Remark
$	abbcde$	SHIFT
$a	bbcde$	SHIFT
$ab	bcde$	REDUCE
$aA	bcde$	SHIFT
$aAb	cde$	SHIFT (?)
$aAbc	de$	REDUCE
$aA	de$	SHIFT
$aAd	e$	REDUCE
$aAB	e$	SHIFT
$aABe	$	REDUCE
$S	$	ACCEPT

栈中出现的所有符号串都是活前缀

S□ **aABe** □ **aAde** □ **aAbcde** □ **abbcde**

何时移进？何时归约？

- 有时栈顶符号将成为句柄（但还不是）
- 为什么？还未移进足够符号识别句柄
- 正确的移进/归约步骤，保证**栈中符号总是活前缀**

$aAb　　cde$　　移进还是归约?

- 移进——栈中aAbc
- 归约——栈中aAA，不是活前缀！所以正确动作应该是移进

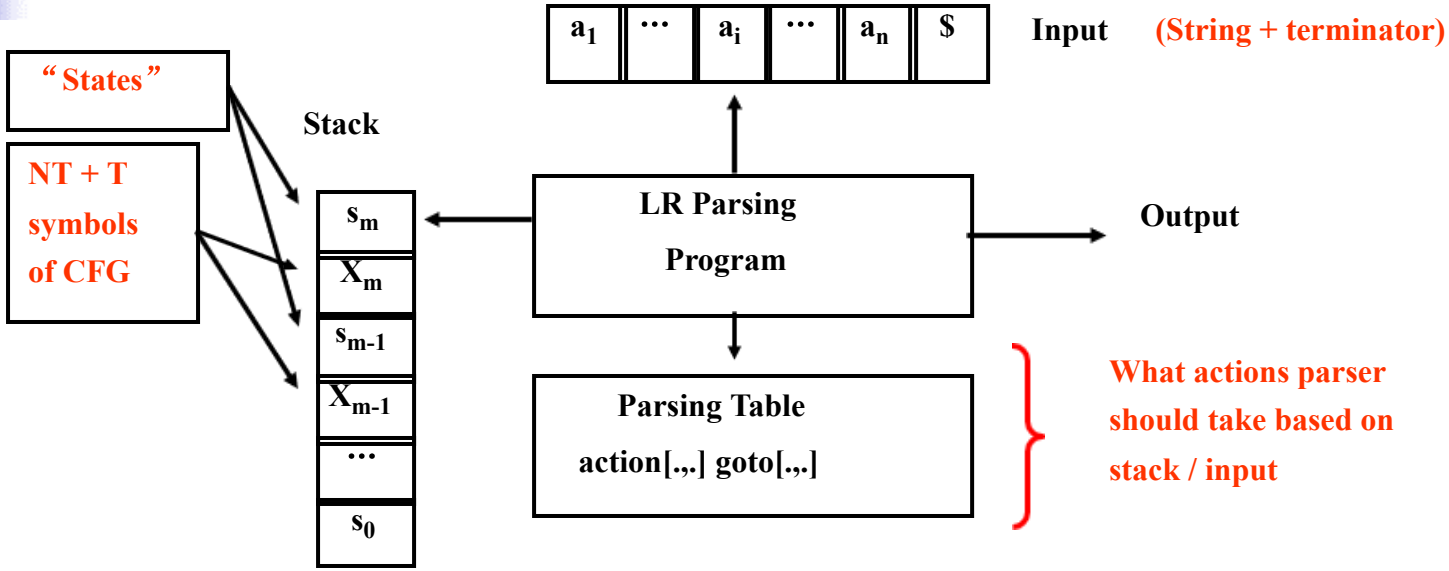
何时移进？何时归约？（续）

- 确定移进/归约，需要
  - 查看栈中一些符号
  - 保证对栈的修改保持“活前缀特性”
- 对于前面例子
  - 任何b若在‘A’右边，不应归约
  - b只有在‘a’之后，才可归约
- 问题：在栈中保存何种信息，才能使得：只查看栈顶数据，即可确定移进/归约？

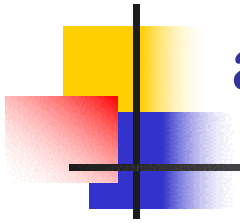
LR分析方法

- LR：L—由左至右扫描，R—最右推导
- LR(1)——1个向前搜索符号
- 优缺点
  - 实际上可识别所有程序语言结构
  - 最通用的非回溯移进-归约分析方法
  - 可分析的文法是预测分析法的超集
  - 错误检测迅速
  - 手工实现繁琐，但已有很多自动生成工具
- SLR，规范LR，LALR

4.7.1 LR分析算法



- 不同LR分析方法算法相同，只是分析表不同
- s_i：状态，概括了栈中位于它下面的所有符号（活前缀）的信息



action表

- action[s_m, a_i]：四种动作
  1. 移进状态s
  2. 利用产生式A □ β进行归约
  3. 接受输入串
  4. 错误

goto表

- goto[s, A]=s’——识别活前缀的DFA的状态转换函数——活前缀（状态）集合实际上是一个正规集，可用DFA识别！
- 初始状态——栈中初始状态
- 格局（configuration）

  $(s_0X_1s_1X_1\cdots X_ms_m, a_ia_{i+1}\cdots a_n\$)$

  $X_1X_1\cdots X_ma_ia_{i+1}\cdots a_n$——最右句型

分析器运行方式

1. action[s_m, a_i]=移进s，进行移进动作，格局变化

   $(s_0X_1s_1X_2\cdots X_ms_ma_is,\quad a_{i+1}\cdots a_n\$)$

2. action[s_m, a_i]=归约A→β（β =$X_{m-r+1}\cdots X_m$），进行归约动作，格局变为

   $(s_0X_1s_1\cdots X_{m-r}s_{m-r}As,\quad a_ia_{i+1}\cdots a_n\$)$

   其中，r为β的长度，s=goto[s_{m-r}, A]

3. action[s_m, a_i]=接受，分析结束

4. action[s_m, a_i]=错误，调用错误恢复函数

算法4.7 LR分析算法

输入：一个输入串w和一个文法G的LR分析表

输出：若w∈L(G)，输出w的一个自底向上分析；

否则，输出错误

方法：初始，栈中只含初始状态s_0，输入缓冲区为w$，分析器按如下算法运行

将ip设置为w$的第一个符号;

repeat forever begin

令s为栈顶状态，a为ip指向符号;

if (action[s, a] = 移进s’) {

将a，s’依次压栈;

算法4.7 LR分析算法（续）

```
        ip指向下一个输入符号;
    }
    else if (action[s, a]=归约A□β) {
        从栈中弹出2*|β|个符号;
        令s’为当前栈顶状态;
        将A和状态goto[s’, A]依次压栈;
        输出A□β;
    }
    else if (action[s, a] = 接受)
        return;
    else error();
}
```

例4.33

(1) E □ E + T

(2) E □ T

(3) T □ T * F

(4) T □ F

(5) F □ (E)

(6) F □ id

- si：移进状态i
- rj：利用产生式j归约
- acc：接受
- 空格：错误

例4.33 分析表

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

例4.33 分析实例

STACK	INPUT	Remark
(1) 0	**id * id + id$**	移进
(2) 0 **id** 5	*** id + id$**	归约F □ **id**
(3) 0 F 3	*** id + id$**	归约T □ F
(4) 0 T 2	*** id + id$**	移进
(5) 0 T 2 * 7	**id + id$**	移进
(6) 0 T 2 * 7 **id** 5	**+ id$**	归约F □ **id**
(7) 0 T 2 * 7 F 10	**+ id$**	归约T □ T * F
(8) 0 T 2	**+ id$**	归约E □ T
(9) 0 E 1	**+ id$**	移进
(10) 0 E 1 + 6	**id$**	移进
(11) 0 E 1 + 6 **id** 5	**$**	归约F □ **id**
(12) 0 E 1 + 6 F 3	**$**	归约T □ F
(13) 0 E 1 + 6 T 9	**$**	归约E □ E + T
(14) 0 E 1	**$**	接受

4.7.2 LR文法

- 可创建LR分析表的文法
- 如何构造LR文法
  - 所有活前缀的集合是正规的
  - 构造DFA识别它
  - 利用DFA辅助分析表的构造
- LR(k)——展望k个符号

  LL(k)——区分右部推导的前k个符号

  LR描述能力更强

$\$\cdots\alpha \quad x\cdots\$$

把αx拼接起来，期望一起“拼装成”“别的什么”

$\$\cdots A \quad x\cdots\$$

试图寻找A□x *

把A“转换成”x

4.7.3 SLR分析

- Simple LR parsing
- 文法G的**LR(0)项目**（简称**项目**，**item**）

  ——产生式，右部某个位置加标记 ‘.’
- A□XYZ有4个项目

  A□·XYZ

  A□X·YZ

  A□XY·Z

  A□XYZ·

项目的含义

- 分析过程中某一时刻，已经（在栈中）**得到了**产生式的**哪些部分**，为形成**句柄**，还需（从输入缓冲区）获取**哪些部分**
  - ·XYZ，栈中还未得到任何内容，“期待”得到XYZ——输入缓冲中内容“拼装成”XYZ
  - X·YZ，已得到X，期待从输入缓冲得到YZ
- A□ε——A□·

构造SLR分析表

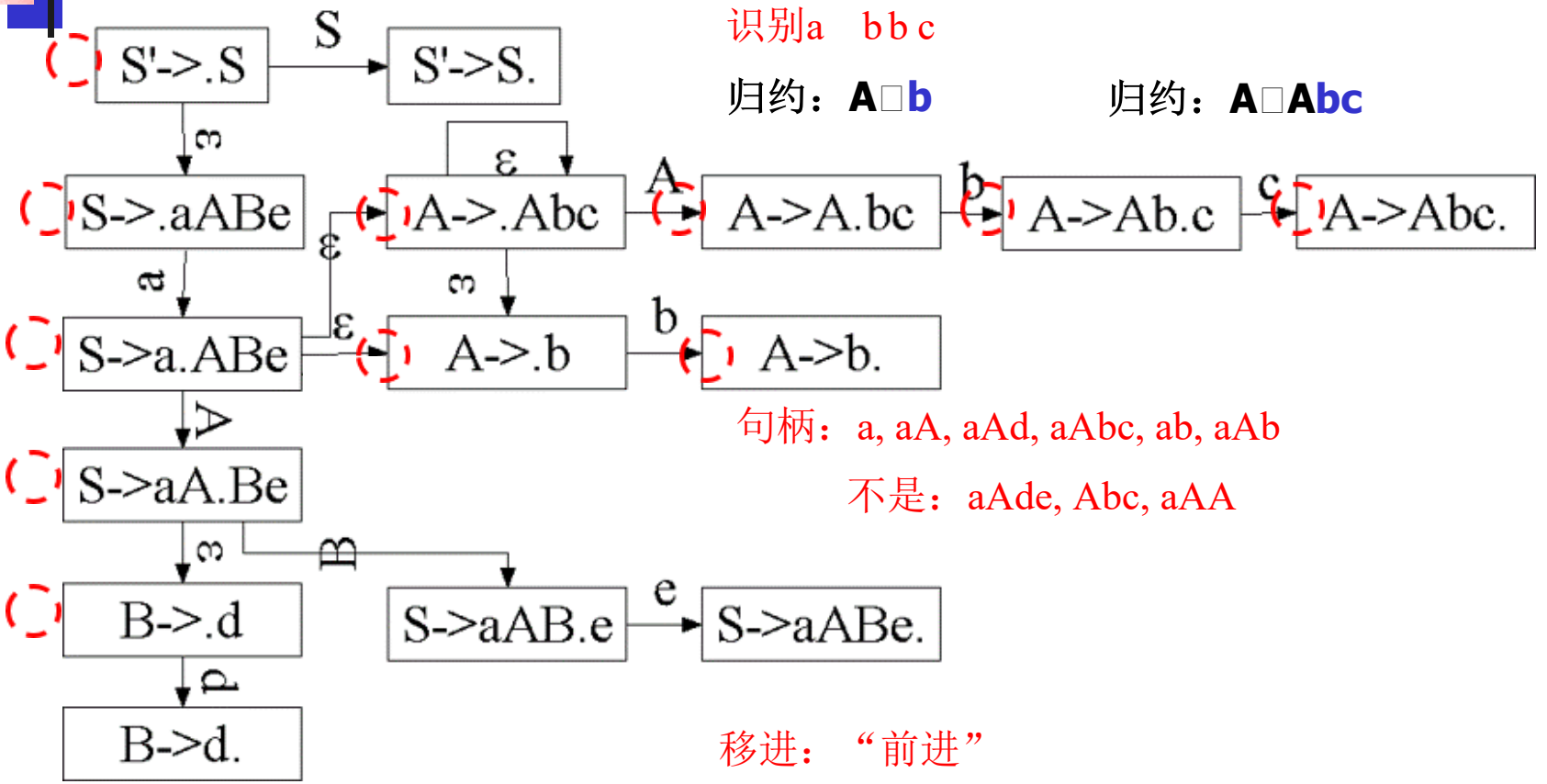
- 核心思想
  - 构造一个DFA来识别活前缀
  - 构造LR(0)项目集

    项目——NFA状态

    LR(0)项目集——DFA状态

    构造项目集——NFA□DFA，子集构造法！
  - NFA
    - 状态——LR(0)项目
    - 边——移进动作
  - 全体LR(0)项目集——LR(0)项目集规范族
  - 拓广文法：S’□S
  - 计算闭包（closure）函数和goto函数

项目——状态的NFA



识别a　b b c

归约：**A□b**　　归约：**A□Abc**

句柄：a, aA, aAd, aAbc, ab, aAb

不是：aAde, Abc, aAA

移进：“前进”

归约：“以退为进”

S□ **aABe** □ **aAde** □ **aAbcde** □ **abbcde**

closure函数的计算

- I为项目集，则closure(I)为按下列规则构造的项目集：
  1. 初始，I中项目都加入closure(I)
  2. A□α·Bβ∈closure(I)，存在产生式B□γ，则将B□·γ添加进closure(I)
  3. 重复2直至没有新的项目加入闭包。

两者间有ε边！

ε-closure闭包！

- A□α·Bβ——期待用输入缓冲中内容拼出Bβ

  存在产生式B□γ □ 先考虑拼出γ，再拼出B

closure函数

```
Function closure(I)
{       J=I;
        repeat
                for J中每个A □□.B□ 和
                    G的每个产生式B□□，B□.□不在J中
                    将B□.□加入J
        until 没有新的项目加入J
        return J
}
```

例4.34

E’ □ E

E □ E + T | T

T □ T * F | F

F □ (E) | id

I = {E’ □ ·E}

closure(I) = {E’ □·E,

E□·E+T, E□·T,

T□·T * F, T□·F,

F□·(E), F □· id }

核心（kernel）项目

- B□·γ（‘·’在最左边）加入closure(I)□
  所有类似产生式（项目）都加入closure(I)
- 实际上不必一一列出，用B表示即可
- 所有项目可分为两大类：
  1. 核心项目：S’□·S和‘·’不在最左边的项目
  2. 非核心项目：其他‘·’在最左边的项目（ε边！）
- 项目集中实际只关心核心项目

goto函数的计算

goto(I, X)=closure({A□□X·□ | A□□·X□ ∈I})

ε_closure(δ(I, X))!

- I：活前缀γ对应的有效项目集□

  goto(I, X)：活前缀γX对应的有效项目集

- 例4.35：I = { E’ □·E, E□E·+T }

  goto(I, +) = { E□E+·T,

  T□·T*F, T□·F,

  F□·(E), F□·**id** }

LR(0)项目集规范族的构造

```
Procedure Items(G’:augmented grammar)
{       C:={ closure [S’ □ .S] }
        repeat
                for C中每个项目集I 和
                        每个文法符号X，goto(I,X)不空且不在C中
                                将goto(I,X)加入C
        until 没有新的项目集加入C
}
```

初始项目集

子集构造法类似过程，由初始项目集开始，对每个项目集和文法符号计算goto函数，直至不再产生新的项目集

例4.36

I_0

E’ □ ·E

E □ ·E + T

E □ ·T

T □ ·T * F

T □·F

F □ ·(E)

F □ ·id

$I_1 = goto(I_0, E)$

E’ □ E·

E □ E ·+ T

$I_3 = goto(I_0, F)$

T □F·

$I_2 = goto(I_0, T)$

E □ T·

T □ T ·* F

例4.36

$I_4 = goto(I_0, ()$

F □ (· E)

E □ ·E + T

E □ ·T

T □ ·T * F

T □·F

F □ ·(E)

F □ ·id

$I_5 = goto(I_0, id)$

F □ id·

$I_6 = goto(I_1, +)$

E □ E + ·T

T □ ·T * F

T □·F

F □ ·(E)

F □ ·id

例4.36（续）

$I_7 = goto(I_2, *)$

T □ T * ·F

F □ ·(E)

F □ ·id

$I_8 = goto(I_4, E)$

F □ (E ·)

E □ E ·+ T

$I_9 = goto(I_6, T)$

E □ E + T·

T □ T· * F

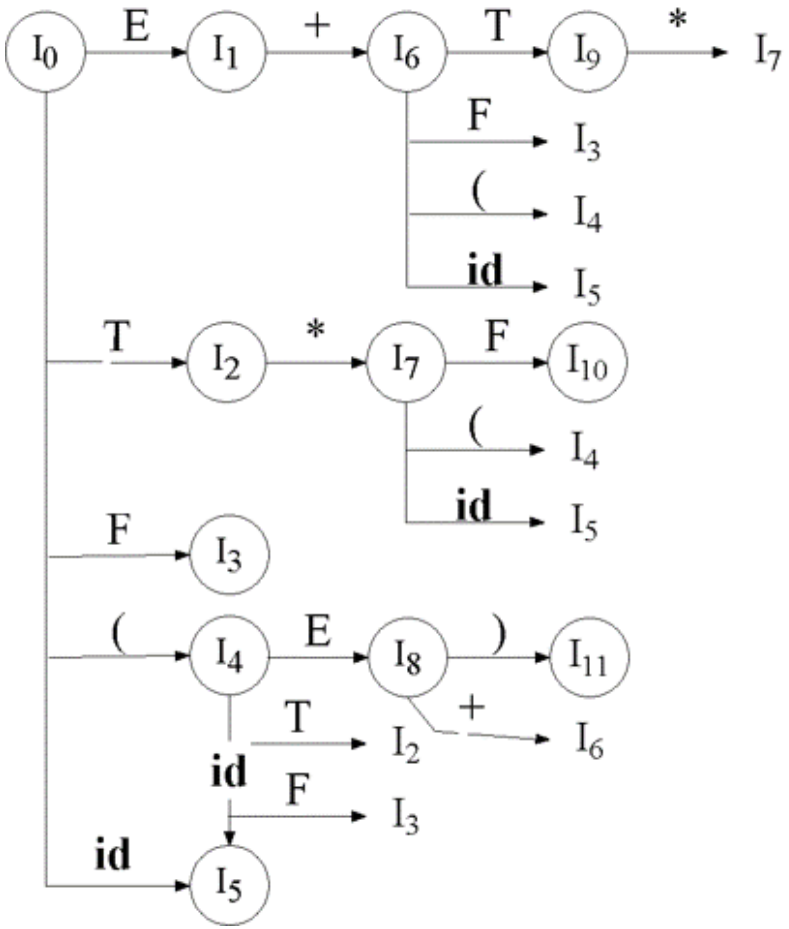
$I_{10} = goto(I_7, F)$

T □ T * F·

$I_{11} = goto(I_8,))$

F □ (E)·

例4.36（续）


I0
E
I1
+
I6
T
I9
*
I7
F
I3
(
I4
id
I5
T
I2
*
I7
F
I10
(
I4
id
I5
F
I3
(
I4
E
I8
)
I11
T
I2
+
I6
id
F
I3
id
I5

例4.36（续）

- I_0初态，所有状态——终态
- 可构造NFA N
  - LR(0)项目□状态
  - 边A□□·X□□A□□X·□标记为X
  - 边A□□·B□□B□·γ标记为ε
  - 项目集的闭包——NFA状态集的ε闭包
  - goto——DFA的状态转换函数
  - items(G’)＝算法3.2

有效项目（valid items）

- 称项目A□□$_1$·□$_2$对活前缀□□$_1$**有效**，

  当存在一个最右推导

  $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha Aw \underset{rm}{\Rightarrow} \alpha \square_1 \square_2 w$

- 一个项目可对很多活前缀有效

有效项目（valid items）

- □□$_1$位于栈顶，帮助确定移进/归约
  - □$_2$□ ε，未形成句柄，移进
  - □$_2$= ε，A□□$_1$为句柄，归约
  - 对同一活前缀，项目集中不同有效项目可能有不同建议——向前搜索解决，或冲突！
- LR分析法核心理论：活前缀γ的有效项目集＝DFA读入γ后达到的状态

例4.37

- E+T*是一个活前缀，对应状态集I_7
- 包括：T □ T*·F

  F □ ·(E)

  F □ ·id　　E+T*的有效项目
- 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

  分别验证了三个项目的有效性
- 无其他有效项目

算法4.8：构造SLR分析表

输入：拓广文法G’

输出：G’的SLR分析表——action，goto函数

方法：

1. 构造G’的LR(0)项目集规范族C={$I_0, I_1, \cdots, I_n$}
2. 由I_i构造状态i：
   a. 若[A□□·a□] 在I_i中，且goto(I_i, a)=I_j，则action[i, a]=“移进j”，此处a为终结符
   b. 若[A□□·] 在I_i中，对所有a∈FOLLOW(A)，令action[i, a]=“归约A□□”，此处A≠S’

算法4.8：构造SLR分析表(续)

c. 若[S’ □S·] 在I_i中，则令action[i, $]=“接受”

3. 利用DFA的状态转移函数构造状态的

   goto函数：goto(I_i, A)=I_j□goto[i, A]=j

4. 2、3未定义的表项为错误项

5. 分析器的初态，设定为包含[S’ □S·]的

   项目集I_0

例4.38

I_0

E’ □ ·E

E □ ·E + T

E □ ·T

T □ ·T * F

T □·F

F □ ·(E)

F □ ·id

□

F □ ·(E), goto(I_0, ( )=I_4□action[0, (]=s4

F □ ·id, goto(I_0, **id**)=I_5□action[0, **id**]=s5

goto[0, E] = 1

goto[0, T] = 2

goto[0, F] = 3

I_1

E’ □ E·

E □ E ·+ T

□

E’ □ E·□action[1, $]=acc

E □ E ·+ T, goto(I_1, +)=I_6□action[1, +]=s6

例4.38（续）

I_2

E □ **T**·

T □ **T** ·* **F**

□

E □ **T**·, FOLLOW(E)={$, +,)}

□action[2,$]=action[2,+]= action[2,)]=r2

T □ **T** ·* **F** , goto(I_2, *)=I_7□action[2,*]=s7

I_3

T □**F**·

□

T □**F**·, FOLLOW(T)={$, +, *,)}

□action[3, $/+/*/)]=r4

例4.38（续）

I_4

F □ (· E)

E □ ·E + T

E □ ·T

T □ ·T * F

T □·F

F □ ·(E)

F □ ·id

□

action[4, (]=s4, action[4, **id**]=s5

goto[4, E] = 8

goto[4, T] = 2

goto[4, F] = 3

例4.38（续）

I_5

F □ id·

□

F □ id·, FOLLOW(F)={$, +, *,)}

□action[5, $/+/*/)]=r6

I_6

E □ E + ·T

T □ ·T * F

T □·F

F □ ·(E)

F □ ·id

□

action[6, (]=s4, action[6, **id**]=s5

goto[6, T] = 9

goto[6, F] = 3

例4.38（续）

I_7

T □ **T** * ·**F**

F □ ·(**E**)

F □ ·**id**

□ action[7, (]=s4, action[7, **id**]=s5

goto[7, F] = 10

I_8

F □ (**E** ·)

E □ **E** ·+ **T**

□ action[8, (]=s11

action[8, +]=s6

I_9

E □ **E** + **T**·

T □ **T**· * **F**

□ **E** □ **E** + **T**·, FOLLOW(E)={$, +,)}

□action[9, $/+/)]=r1

action[9, *] = 7

例4.38（续）

I_{10}

T □ T * F·

□

T □ T * F·, FOLLOW(T)={$, +, *,)}

□action[10, $/+/*/)]=r3

I_{11}

F □ (E)·

□

F □ (E)·, FOLLOW(F)={$, +, *,)}

□action[11, $/+/*/)]=r5

例4.38（续）

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			

例4.39 SLR(1)分析表冲突

- 二义性文法肯定不是SLR(1)文法，非二义性文法不都是SLR(1)文法

(0) S’ □ S

(1) S □ L = R

(2) S □ R

(3) L □ * R

(4) L □ **id**

(5) R □ L

例4.39 SLR(1)分析表冲突(续)

I_0={S’ □·S , S□·L = R , S□·R , L□·* R ,
L□·id , R□·L}

I_1 = {S’ □ S· }

I_2 = {S □ L · = R , R □ L · }

I_3 = {S □ R·}

I_4 = {L □ *·R , R □ ·L , L □ ·* R , L □· id }

I_5 = {L □ id· }

I_6 = {S □ L = · R , R □ ·L , L □ ·* R , L □· id }

I_7 = {L □ * R·}

I_8 = {R □ L ·}

I_9 = {S□L = R·}

例4.39 SLR(1)分析表冲突(续)

- $I_2 = \{S \to L \cdot = R, R \to L \cdot\}$
  - $S \to L \cdot = R \in$ action[2, =] = s6
  - $R \to L \cdot, = \in FOLLOW(R)$
    - action[2, =] = r5
  - 移进/归约冲突
- 分析实例： **id=id**

$0	**id=id$**	**s5**	
$0id5	**=id$**		**r L →id**
$0L2	**=id$**		冲突…

例4.39 SLR(1)分析表冲突(续)

- 应该进行哪个动作？
- R=**id**不是句型！
- =可在R之后，但不是这种情况，只有R在*之后的情况下才会发生——*R=**id**
- 冲突原因

  SLR分析方法——LR(0)+FOLLOW的方法不够强大

  规范LR和LALR均可处理此文法

4.7.4 构造规范LR分析表

- SLR：若项目集I_i包含[A□□·]，且a在FOLLOW(A)中，则状态i进行归约
- 但某些情况，当状态i在栈顶，活前缀βα在栈中时，任何最右句型，都不可能是…βAa…的形式
- 例4.40：
  - 如例4.39，状态2和下一符号=，无法确定如何归约，但不存在句型R=…，状态2只对应活前缀‘L’

LR(1)项目

- 每个项目携带更多信息
  - 方法：分离状态
  - 每个状态明确指出：当哪些符号接在句柄α之后，才可进行α到A的归约

LR(1)项目

- LR(1)项目：产生式+‘·’+终结符

  如 [S □ aA·Be, c]

  - 含义：对一个产生式
    - 栈中已形成哪些部分（aA）+
    - 期待下一步构造出哪些部分（Be）+
    - 向前搜索符c：要进行S□aABe归约，**跟随符号**应是什么
    - 更好的归约判定
- LR(1)项目的核(core)：LR(0)项目

使用LR(1)项目

- 只对归约起作用
  - 当状态i在栈顶，a为下一输入符号
  - 仅当$[A\to\alpha\cdot, a]\in I_i$，才进行$A\to\alpha$归约
  - $[A\to\alpha\cdot, b]$不能指示此情况进行归约
- 有效的概念发生改变
  - $[A\to\alpha\cdot\beta, a]$对活前缀$\gamma$有效，仅当存在推导

    $S\overset{*}{\underset{rm}{\Rightarrow}}\delta Aw\underset{rm}{\Rightarrow}\delta\alpha w$，其中

    1. $\gamma=\delta\alpha$，且
    2. a为w的第一个符号，或w为ε且a为$

例4.41

S□BB

B□aB | b

- 存在最右推导$S \underset{rm}{\Rightarrow} aaBab \underset{rm}{\Rightarrow} aaaBab$

令γ=aaa，δ=aa，A=B，α=a，β=B，w=ab，

则[B□a·B, a]对活前缀aaa有效

- 另存在最右推导$S \underset{rm}{\Rightarrow} BaB \underset{rm}{\Rightarrow} BaaB$

[B□a·B, $]对活前缀Baa是有效的

构造LR(1)项目集规范族

- 基本方法与LR(0)项目集规范族构造类似
- closure、goto的计算有变化
  - [A□α·Bβ, a]在项目集中，项目集对活前缀γ有效，则存在最右推导$S \Box \delta Aax \overset{*}{\underset{rm}{\Box}} \delta\alpha B \Box ax \underset{rm}{}$，其中γ=δα
  - 假定$\Box ax \Box by \overset{*}{\underset{rm}{}}$，则对产生式B□η，存在最右推导$S \Box \overset{*}{\underset{rm}{\gamma}} By \Box \gamma\eta \underset{rm}{by}$，则[B□·η, b]对γ有效
  - 因此，对FIRST(□a)中所有终结符b，将[B□·η, b]加入项目集

算法4.9 构造LR(1)项目集规范族

输入：一个拓广文法G’

输出：LR(1)项目集规范族，项目集对G’的一个或多个活

前缀有效

方法：

```
function closure(I)
{
    repeat
        for I中每个项目[A□α·Bβ, a]，G’中每个产生式B□γ，
                和FIRST(βa)中每个终结符b，
            若[B□·γ, b]不在I中
                将[B□·γ, b]加入I;
    until 没有新的项目加入I;
    return I;
}
```

算法4.9（续）

function goto(I, X)

{

令J为项目[A□αX·β, a]的集合，其中[A□α·Xβ, a]在I中

return closure(J);

}

function items(G’)

{

C = {closure({[S’ □·S, $]})};

repeat

for C中每个项目集I，每个文法符号X，goto(I, X)　　不空且不在C中

将goto(I, X)加入C

until 没有新的项目集加入C

}

例4.42

S’ □ S

S □ CC

C □ cC | d

FIRST(S) = FIRST(C) = {c, d}

FOLLOW(S)={$}

FOLLOW(C)={$,c,d}

I_0 = closure({[S’ □·S, $]}) = {[S’ □·S, $],

[S □·CC, $], [C □·cC, c/d], [C □·d, c/d]}

I_1=goto(I_0, S)={[S’ □S·, $]}

I_2=goto(I_0, C)={[S □C·C, $], [C □·cC, $], [C □·d, $]}

I_3=goto(I_0, c)={[C □c·C, c/d] , [C □·cC, c/d], [C □·d, c/d]}

例4.42（续）

$I_4=goto(I_0, d)=\{[C \to d\cdot, c/d]\}$

$I_5=goto(I_2, C)=\{[S \to CC\cdot, \$]\}$

$I_6=goto(I_2, c)=\{[C \to c\cdot C, \$], [C \to \cdot cC, \$], [C \to \cdot d, \$]\}$

$I_7=goto(I_2, d)=\{[C \to d\cdot, \$]\}$

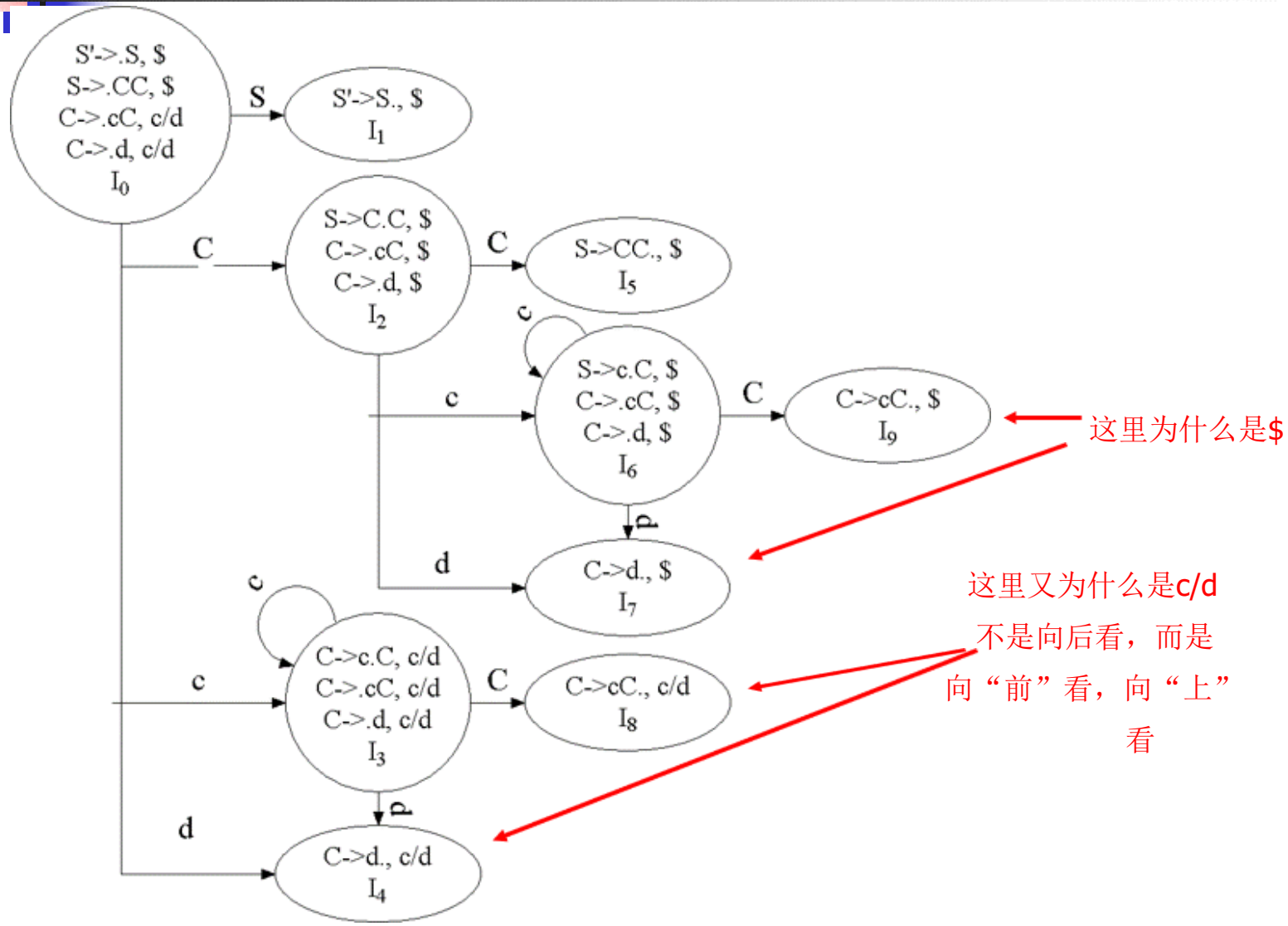
$I_8=goto(I_3, C)=\{[C \to cC\cdot, c/d]\}$

$goto(I_3, c)=I_3$, $goto(I_3, d)=I_4$

$I_9=goto(I_6, C)=\{[C \to cC\cdot, \$]\}$

$goto(I_6, c)=I_6$, $goto(I_6, d)=I_7$

例4.42（续）


S'->.S, $
S->.CC, $
C->.cC, c/d
C->.d, c/d
I0
S
S'->S., $
I1
C
S->C.C, $
C->.cC, $
C->.d, $
I2
C
S->CC., $
I5
c
S->c.C, $
C->.cC, $
C->.d, $
I6
c
C
C->cC., $
I9
d
C->d., $
I7
c
C->c.C, c/d
C->.cC, c/d
C->.d, c/d
I3
c
C
C->cC., c/d
I8
d
C->d., c/d
I4
这里为什么是$
这里又为什么是c/d
不是向后看，而是
向“前”看，向“上”
看

算法4.10 规范LR分析表的构造

输入：一个拓广文法G’

输出：G’的规范LR分析表，函数action和goto

方法：

1. 构造LR(1)项目集规范族C={I_0, I_1, …, I_n}
2. 由I_i构造状态i：
   a. 若[A□□·a□, b]在I_i中，且goto(I_i, a)=I_j，令action[i, a]=“移进j”，此处a为终结符
   b. 若[A□□·, a]在I_i中，A≠S’，则令 action[i, a] = “归约A□□”

算法4.10（续）

c. 若[S’ □S· , $]在$I_i$中，则令

action[i, $] = “接受”

3. 若goto(I_i, A)=I_j，则goto[I_i, A]=j

4. 所有2、3未定义的项为错误项

5. 分析器的初态设定为[S’ □·S , $]所在项目集$I_0$

例4.43

- 例4.42文法规范LR(1)分析表如下所示

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		

4.7.5 构造LALR分析表

- lookahead LR
- 比规范LR分析表小，但对应程序设计语言的翻译，其描述能力足够
- 状态数与SLR相同
  - Pascal语言，几百个
  - 规范LR，几千个

“同心”

- 考虑同心（core，LR(0)项目集）的LR(1)项目集

  S’ □ S

  S □ CC

  C □ cC | d

  I_4和I_7，心C □d·相同，仅搜索符不同

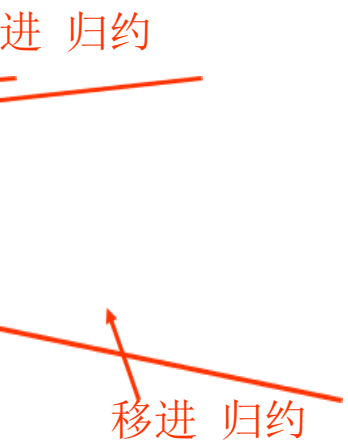
同心集合并

- I_4和I_7区别？
  - 文法生成正规式c^*dc^*d
  - I_4：“第一个d”，后面需接c、d，不能接$
  - I_7：“第二个d”，接受，后接$
  - 合并:I_{47}={[C □d·, c/d/$]}，goto($I_{0/3/2/6}$, d)=$I_{47}$
  - I_{47}对任何输入符号都进行归约——错误不会遗漏，

    ccd、cdcdc，延迟——下个移进时捕捉到
  - 同心集合并，I_{36}，I_{89}

合并后的action和goto

- goto仅依赖心，自然合并，没有问题
- 原LR(1)分析表无冲突，合并后action会产生新的冲突吗？
  - 假定产生移进/归约冲突
    - { [B □β·aγ, b], [A □α·, a], … }
    - 原项目集：{[A □α·, a], [B □β·aγ, c], …}
    - 原LR(1)也有分析表冲突！不可能
  - 可能产生归约/归约冲突



例4.44

- 考虑文法

  S’ □ S

  S □ aAd | bBd | aBe | bAe

  A □ c

  B □ c　　产生符号串acd, ace, bcd, bce

- 构造LR(1)项目集

  - {[A □ c·, d], [B □ c·, e]}对活前缀ac有效
  - {[A □ c·, e], [B □ c·, d]}对活前缀bc有效
  - 无冲突，合并{[A □ c·, d/e], [B □ c·, d/e]}，产生归约/归约冲突

算法4.11 LALR分析表构造算法

输入：一个拓广文法G’

输出：G’的LALR分析表，action和goto函数

方法：

1. 构造LR(1)项目集规范族C={$I_0, I_1, \cdots, I_n$}
2. 将同心集合并
3. 令C’={$J_0, J_1, \cdots, J_m$}为合并后项目集，action函数的构造同算法4.10，若有冲突，失败
4. 令J=$I_1 \cup I_2 \cup \cdots \cup I_k$，$I_1, I_2, \cdots, I_k$心相同，因此goto($I_1$,X), goto($I_2$,X),…,goto($I_k$,X)心也相同，令它们合并结果为K，则goto(J, X)=K

例4.45

- 例4.42文法，三对项目集合并

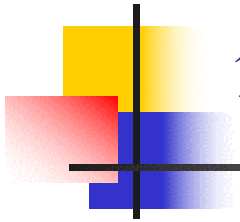
I_{36}={[C □c·C, c/d/$], [C □·cC, c/d/$],[C □·d, c/d/$]}

I_{47}={[C □d·, c/d/$]}

I_{89}={[C □cC·, c/d/$]}

状态	action			goto	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5			r1		
6	r2	r2	r2		

状态	action			goto	
	c	d	$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		



分析过程

- 对正确输入，LALR分析器与规范LR分析器分析过程完全一样
- 对错误输入，LALR会延迟报告错误
  - 例4.42文法，输入串ccd$
  - 规范LR：0 c 3 c 3 d 4，状态4发现错误
  - LALR

    0 c 36 c 36 d 47□0 c 36 c 36 C 89□

    0 c 36 C 89□0 C 2

    状态2发现错误

4.7.6 LALR分析表的高效构造方法

- 状态集I可用其中的核项目表示
- 分析表的计算可仅依赖核项目
  - action
    - 归约A□α，α≠ε，必为核项目
    - 对输入符号a归约A□ε，当且仅当有核项目[B□γ·Cδ, b]， $C \Rightarrow_{rm}^{*} A\eta$，且a∈FIRST(ηδb)
    - 若有核项目[B□γ·Cδ, b]，存在推导$C \Rightarrow_{rm}^{*} ax$，其最后一步不使用ε产生式□移进a
  - goto
    - [B□γ·Xδ, b]—I的核——$\xrightarrow{goto(I,X)}$□[B□γX·δ, b]—核
    - [B□γ·Cδ, b], $C \Rightarrow_{rm}^{*} A\eta$——$\xrightarrow{goto(I,X)}$□[A□X·β, a]

例4.46

- 考虑文法

(0) S’ □ S

(1) S □ L = R

(2) S □ R

(3) L □ * R

(4) L □ **id**

(5) R □ L

例4.46（续）

LR(0)项目集的核，这些核项目的搜索符怎么得到？考察搜索符的产生方式

I_0={S’ □·S}

I_1 = {S’ □ S· }

I_2 = {S □ L · = R , R □ L · }

I_3 = {S □ R·}

I_4 = {L □ *·R }

I_5 = {L □ id· }

I_6 = {S □ L = · R}

I_7 = {L □ * R·}

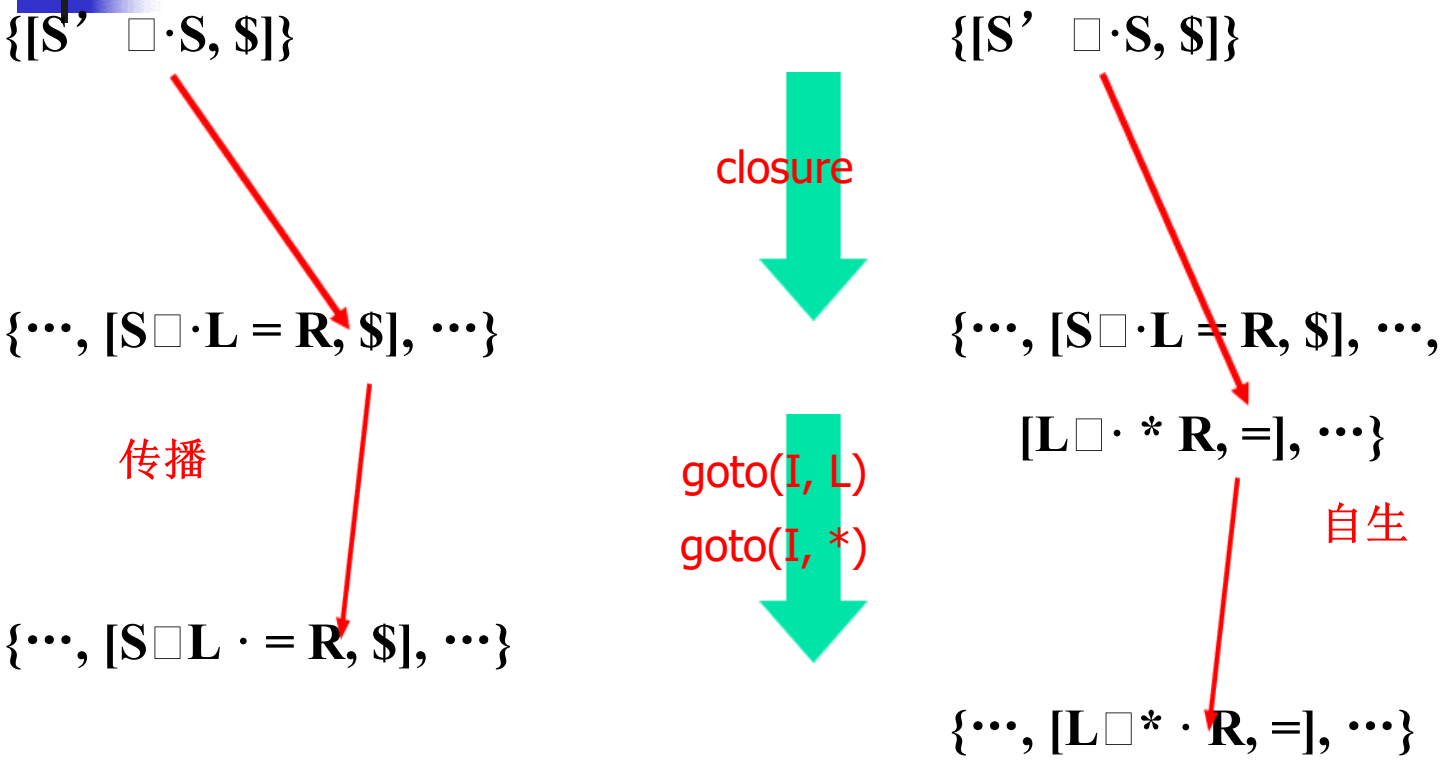
I_8 = {R □ L ·}

I_9 = {S□L = R·}

例4.46（续）

- 扩展搜索符，搜索符如何“传播”？
  - 对LR(0)项目，若B□γ·Cδ在I中，C□Aη，$\overset{*}{\underset{rm}{\Rightarrow}}$存在A□Xβ，则A□X·β在goto(I, X)中
  - 考虑LR(1)项目，[B□γ·Cδ, b]在I中，什么样的搜索符a，使[A□X·β, a]在goto(I, X)中
    - a∈FIRST(ηδ)，自生的（spontaneously）
    - ηδ□*ε，则a=b，传播的（propagate）

例4.46 （续）



算法4.12 判定自生/传播

输入：LR(0)项目集I的核K，文法符号X

输出：goto(I, X)的自生搜索符和传播搜索符

方法：#为伪搜索符

for K中每个项目B□γ·δ {

J’=closure({[B□γ·δ, #]});

if ([A□α·Xβ, a]在J’中，而a≠#)

a为goto(I, X)中项目A□αX·β的自生搜索符;

if ([A□α·Xβ, #]在J’中)

搜索符从I中B□γ·δ传播到goto(I, X)中A□αX·β

}

计算搜索符

- LR(0)初态集核项目S’ □·S具有搜索符$
- 利用算法4.12生成所有自生搜索符
- 将自生搜索符进行传播，直到无法继续传播为止

算法4.13

- 构造LALR(1)项目集规范族的核

输入：拓广文法G’

输出：G’的LALR(1)项目集规范族的核

方法：

1. 构造LR(0)项目集的核
2. 利用算法4.12，对每个LR(0)项目I的核K和每个文法符号X，确定goto(I, X)的核项目的所有自生搜索符，以及K中哪些项目传播搜索符到goto(I, X)的核

算法4.13（续）

3. 初始化搜索符表——表示每个项目集的核项目对应哪些搜索符，初始为自生搜索符集
4. 重复在核项目间传播搜索符：对每个项目i，

   根据2得出它应向哪些项目传播搜索符，将它当前搜索符集并入传播目标项目的搜索符集。重复此过程，直至没有新的传播为止。

例4.47

- 考虑例4.46文法

closure({[S’ □·S,#]})={[S’ □·S, #], [S□·L = R, #],

[S□·R, #], [L□·* R, =/#], [L□·id, =/#], [R□·L, #]}

[L□·* R, =]导致I_4中L□*·R的自生搜索符=

[L□·id, =]导致I_5中L□id·的自生搜索符=

传播： S’ □S·:I_1， S□L·=R:I_2， R□L·:I_2， S□R·:I_3，

L□*·R:I_4， L□id·:I_5

例4.47 （续）

closure({[S □ L · = R, #]})= {[S □ L · = R, #]}

closure({[L □ *·R , #]})={[L □ *·R,#] , [R □ ·L, #] ,

[L □ ·* R, #] , [L □· id, #] }

closure({[S □ L = · R, #]})={[S □ L = · R, #] , [R □ ·L, #] ,

[L □ ·* R, #] , [L □· id, #]}

例4.47 （续）

FROM	TO
I_0: S′ □·S	I_1: S′ □ S·
	I_2: S □ L · = R
	I_2: R □ L ·
	I_3: S □ R·
	I_4: L □ *·R
	I_5: L □ id·
~~I_2: S □ L · = R~~	~~I_6: S □ L = R~~
~~I_4: L □ *·R~~	~~I_4: L □ * R~~
	I_5: L □ id·
	I_7: L □ * R·
	I_8: R □ L ·
~~I_6: S □ L = · R~~	~~I_4: L □ * R~~
	I_5: L □ id·
	I_8: R □ L ·
	I_9: S□L = R·

例4.47 （续）

项目	搜索符			
	初始	第一遍	第二遍	第三遍
I_0: S’ □ ·S	$	$	$	$
I_1: S’ □ S·		$	$	$
I_2: S □ L · = R		$	$	$
I_2: R □ L ·		$	$	$
I_3: S □ R·		$	$	$
I_4: L □ *·R	=	=/$	=/$	=/$
I_5: L □ id·	=	=/$	=/$	=/$
I_6: S □ L = · R			$	$
I_7: L □ * R·		=	=/$	=/$
I_8: R □ L ·		=	=/$	=/$
I_9: S□L = R·				$

例4.47 （续）

状态	action				goto		
	*	=	id	$	S	L	R
0	s4		s5		1	2	3
1				acc			
2		s6		r5			
3				r2			
4	s4		s5			8	7
5		r4		r4			
6	s4		s5			8	9
7		r3		r3			
8		r5		r5			
9				r1			

4.7.7 LR分析表的压缩

- LALR分析表规模
  - 对于程序设计语言的翻译，其文法规模大致为50－100个终结符，100个产生式
  - 得到的LALR分析表规模大致为几百个状态，20000个action函数项

action表的压缩

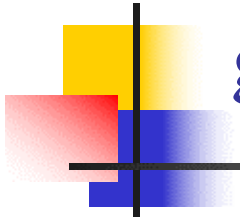
- [状态, 终结符]——二维数组
- 很多行（列表）是相同的
- 状态□指针数组，指向一维数组（行）
- 行□列表——每个状态一个列表
  - (终结符, 动作)，发生频率——位置
  - any—列表中未列出的任何其他终结符

例4.48

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			

例4.48 （续）

- 状态0，4，6，7的action行相同
  - 列表：(**id**, s5), ((, s4), (any, error)
- 状态1：(+, s6), ($, acc), (any, error)
- 8：(+, s6), ( ), s11), (any, error)
- 2：error□r2，错误推迟，(*, s7), (any, r2)
- 9：(*, s7), (any, r1)
- 3：error□r4，(any, r4)，5、10、11类似



goto表的压缩

- 与action表不同，按列压缩，表的每列用列表存储，即每个非终结符一个列表
  - 若非终结符A的列表的某个表项为：

    (current_state, next_state)

    表示在goto表中

    goto[current_state, A] = next_state

goto表的压缩方法的设计思路

- 每列表项（状态）很少
- goto(I_i, A)=I_j，I_j某些项目中，A紧挨在·左边

  状态j出现在第i行第A列

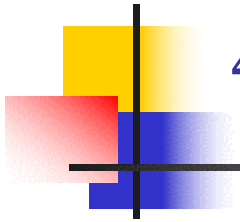
  不存在项目集，对X≠Y，既有项目X在·左边，

  也有项目Y在·左边

  □每个状态只在一列中出现
- 正常项替换error项

例4.49

- F：(7, 10), (any, 3)
- T：(6, 9), (any, 2)
- E：(4, 8), (any, 1)



4.8 使用二义性文法

- 不适合LR，冲突：移进/归约, 归约/归约
- 某些情况下是有用的
  - 表达式，更简洁、更自然
  - 区分普通语法结构和特殊优化情况
- 特殊方法使二义性文法可以使用
- 文法二义性，语言描述无歧义
  - 消除歧义规则□唯一语法树

4.8.1 使用优先级和结合率

E’ □ E

E □ E + T | T

T □ T * F | F

F □ (E) | id

- 原始文法为

1. E□ E+E
2. E□ E*E
3. E□ (E)
4. E□ id

优点：

容易改变运算符优先级

无E□T、E□T额外归约时间

LR(0)项目集规范族

I_0	I_2	I_5	I_8
E’ □ .E	E □ .(E)	E □ E * . E	E □ E * E .
E □ .E + E	E □ .E + E	E □ .E + E	E □ E . + E
E □ .E * E	E □ .E * E	E □ .E * E	E □ E . * E
E □ .(E)	E □ .(E)	E □ .(E)	
E □ .**id**	E □ .**id**	E □ .**id**	

LR(0)项目集规范族

I_1
E’ □ E.
E □ E . + E
E □ E . * E

I_3
E □ id.

I_6
E □ (E .)
E □ E . + E
E □ E . * E

I_9
E □ (E) .

I_4
E □ E + . E
E □ .E + E
E □ .E * E
E □ .(E)
E □ .id

I_7
E □ E + E .
E □ E . + E
E □ E . * E

Follow(E’)=$
Follow(E)=+*)$

SLR分析表

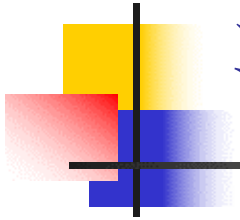
	id	+	*	(	)	$	*E*
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		s4/r1		s5/r1		r1	r1
8		s4/r2		s5/r2		r2	r2
9		r3	r3		r3	r3	

I_7 I_8

E □ E + E . E □ E * E .

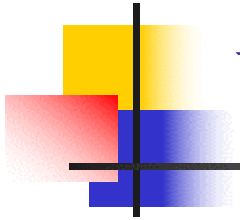
E □ E . + E E □ E . + E

E □ E . * E E □ E . * E



冲突解决方法

- 考虑输入**id+id*id**
  - 分析后**id+id**栈和输入($ 0 E 1 + 4 E 7, * **id** $)
  - 移进/归约（action[7, *]）？
  - *优先级高，移进；反之，归约
- 考虑输入**id+id+id**
  - 分析后**id+id**栈和输入($ 0 E 1 + 4 E 7, + **id** $)
  - 移进/归约（action[7, +]）？
  - +左结合，归约；反之，归约



一般性方法

- 若栈中包含E □ E op E，当前输入符号op
  - 若op为左结合，归约E □ E op E
  - 若op为右结合，移进
- 若栈中包含E □ E op E，当前输入op’
  - 若op优先级高于op’，归约E □ E op E
  - 若op’优先级高于op，移进

4.8.2 “虚悬else”二义性

stmt □ **if** *expr* **then** *stmt*

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

| **other** (any other statement)

- 可改写为

S’ □ S

S □ iSeS | iS | a

LR(0)项目集规范族

I_0

S’ □ .S

S □ .iSeS

S □ .iS

S □ .a

I_1

S’ □S.

I_2

S □ i.SeS

S □ i.S

S □ .iSeS

S □ .iS

S □ .a

I_3

S □ a.

I_4

S □ iS.eS

S □ iS.

I_5

S □ iSe.S

S □ .iSeS

S □ .iS

S □ .a

I_6

S □ iSeS.

Follow(S)= $,e

分析表

	i	e	a	$	*S*
0	s2		s3		*1*
1				acc	
2	s2		s3		*4*
3		r3		r3	
4		s5/r2			r2
5	s2		s3		*6*
6		r1		r1	

- action[4, e]，移进/归约冲突
- else跟随之前最近的if，选择移进！

4.8.3 特殊产生式二义性

1. E□ EsubEsupE

2. E□ EsubE

3. E□ EsupE

4. E□ {E}

5. E□ c

- 产生式1为特殊情况： ，而不是 a_i a_i^2

LR(0)项目集规范族

I_0

E’ □ ·E

E□ ·EsubEsupE

E□ ·EsubE

E□ ·EsupE

E□ ·{E}

E□ ·c

$I_1 = goto(I_0, E)$

E’ □ E·

E□ E·subEsupE

E□ E·subE

E□ E·supE

$I_2 = goto(I_0, ()$

E□ {·E}

E□ ·EsubEsupE

E□ ·EsubE

E□ ·EsupE

E□ ·c

$I_3 = goto(I_0, c)$

E□ c·

$I_4 = goto(I_1, sub)$

E□ Esub·EsupE

E□ Esub·E

E□ ·EsubEsupE

E□ ·EsubE

E□ ·EsupE

E□ ·{E}

E□ ·c

$I_5 = goto(I_1, sup)$

E□ Esup·E

E□ ·EsubEsupE

E□ ·EsubE

E□ ·EsupE

E□ ·{E}

E□ ·c

LR(0)项目集规范族（续）

$I_6 = goto(I_2, E)$

E□ E·subEsupE

E□ E·subE

E□ E·supE

E□ {E·}

$I_{10} = goto(I_7, sup)$

E□ EsubEsup·E

E□ Esup·E

E□ ·EsubEsupE

E□ ·EsubE

E□ ·EsupE

E□ ·{E}

E□ ·c

$I_7 = goto(I_4, E)$

E□

E·subEsupE

E□

EsubE·supE

E□ E·subE

E□ EsubE·

E□ E·supE

$I_9 = goto(I_6, \})$

E□ {E}·

$I_8 = goto(I_5, E)$

E□

E·subEsupE

E□ E·subE

E□ E·supE

E□ EsupE·

$I_{11} = goto(I_{10}, E)$

E□ EsubEsupE·

E□ EsupE·

E□ E·subEsupE

E□ E·subE

E□ E·supE

冲突解决方法

- I_7, I_8, I_{11}，对**sub**和**sup**，移进/归约冲突
  - sub和sup的优先级和结合率未指定
  - 优先级相等，右结合，移进！
- I_{11}对}和$，归约/归约冲突
  - E□ E **sub** E **sup** E？ E□ E **sup** E？
  - 产生式1为特殊情况，优先考虑
  - 归约E□ E **sub** E **sup** E！

分析表

	sub	**sup**	{	}	c	$	*E*
0			s2		s3		*1*
1	s4	s5				acc	
2			s2		s3		*6*
3	r5	r5		r5		r5	
4			s2		s3		*7*
5			s2		s3		*8*
6	s4	s5		s9			
7	s4	s10		r2		r2	
8	s4	s5		r3		r3	
9	r4	r4		r4		r4	
10			s2		s3		*11*
11	s4	s5		r1		r1	

4.8.4 LR分析器错误处理

- 分析表中空位
- LR分析器一发现不能继续，就报告错误
- 规范LR分析器在错误发生后，立即报告
- SLR和LALR分析器可能会进行若干归约后报告，但不会继续移进符号

Panic模式的实现

- 向下扫描栈，对某个特定非终结符A，找到一个状态s，对于A有goto函数
- 丢弃输入符号，直至遇到a∈FOLLOW(A)
- 将goto[s, A]压栈，继续
- A：表示语法结构，表达式、语句…
- 含义：
  - 某个语法结构A分析了一部分，遇到错误
  - 丢弃已分析部分（栈），未分析部分（输入）
  - 假装已找到A的一个实例，继续分析

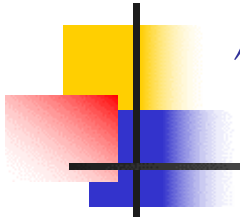
短语级模式的实现

- 对分析表每个错误项，根据语言使用特性分析错误原因，设计恢复函数
- 与算符优先分析器比较
  - 容易实现，无需考虑错误归约
  - 分析表空项填入错误处理函数即可
  - 错误处理函数插入、删除、替换栈和输入
  - 也要避免无限循环

例4.50

E □ E + E　　| E * E | (E) | **id**

状态	action						goto
	id	**+**	*****	**(**	**)**	**$**	**E**
0	s3	e1	e1	s2	e2	e1	1
1	e3	s4	s5	e3	e2	e1	
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	r3	r3	r3	r3	



例4.50（续）

e1：状态0, 2, 4, 5期待运算数，但遇到+、*或$

压栈**id**和状态3（goto[0/2/4/5, id]），

输出错误信息“缺少运算数”

e2：状态0, 2, 4, 5遇到‘)’

删除‘)’，输出错误信息“不匹配的右括号”

e3：状态1, 6期待运算符，但遇到id或(

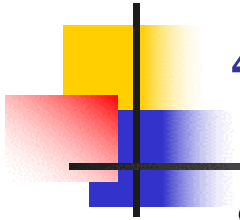
压栈+和状态4，输出错误信息“缺少运算符”

e4：状态6期待运算符或)，但遇到$

压栈)和状态9，输出错误信息“缺少)”

例4.50（续）

STACK	INPUT	Remark
$0	id +)$	
$0 id 3	+)$	
$0 E 1	+)$	
$0 E 1 + 4	)$	
$0 E 1 + 4	$	e2:“未匹配右括号” 将其删除
$0 E 1 + 4 id 3	$	e1:“缺少运算数” 压栈id和状态3
$0 E 1 + 4 E 7	$	
$0 E 1	$	



4.9 Yacc

```
%{
#include <ctype.h>
#include <stdio.h>
#include "expr_lexer.h"
%}

%include {     //将代码加入生成的头文件中
#ifndef YYSTYPE
#define YYSTYPE double
#endif
}
```

Yacc程序（续）

```
%token NUMBER  //定义单词
%left  '+'   '-'        //定义优先级（由低到高）、结合率
%left  '*'   '/'
%right UMINUS

%%              //定义段结束

lines     :     lines expr '\n'   { printf("%g\n", $2); }
          |     lines '\n'
          |
          ;
```

Yacc程序（续）

```
expr :      expr '+' expr     { $$ = $1 + $3; }
        |       expr '-' expr     { $$ = $1 - $3; }
        |       expr '*' expr     { $$ = $1 * $3; }
        |       expr '/' expr     { $$ = $1 / $3; }
        |       '(' expr ')'  { $$ = $2; }
        |       '-' expr %prec UMINUS { $$ = -$2; }
        |       NUMBER
        ;



%%      //规则段结束
```

Yacc程序（续）

```
//程序段
int main(void)
{
    return yyparse();
}
```

Lex程序（续）

```
%{
#include "expr_parser.h"
extern YYSTYPE   yylval;
%}

number    [0-9]+\.?|[0-9]*\.[0-9]+
%%        //定义段结束

[ \t]  { }
{number}   { sscanf(yytext, "%lf", &yylval); return NUMBER; }
\n | . { return yytext[0]; }

%%        //规则段结束
```

4.9.4 Yacc的错误恢复

- 程序员决定哪些NT A可能会产生错误
- 将A □ **error** α加入文法
  - α——文法符号串，**error**——Yacc保留字

Yacc的错误恢复

- 当语法分析时遇到错误，Yacc处理如下：
  - 从栈中弹出状态，直至遇到项目集中含有形如A □ · **error** α的项目为止
  - 将**error**“移进”栈
  - 若α为ε，立即用A □ **error** α进行归约
  - 否则，移进/归约，在栈顶形成**error** α，然后利用A □ **error** α进行归约
  - 恢复正常语法分析

例4.52 台式计算器的错误恢复

- 加入产生式

  lines : error ‘\n’ { yyerror(“重新输入上一行”);
  yyerrok; }

- 当输入有误
  - Yacc弹出状态，直至遇到lines □ · **error**

    ‘\n’——状态0——将上一行输入全部丢弃
  - 将error移进，跳过输入符号，直至‘\n’——开始新的一行表达式的输入
  - yyerrok——Yacc的错误恢复库函数

/