# 第四章 自下而上的语法分析

* 基本思想:
* 从输入串开始，逐步进行“规约”，直到文法的开始符号。即从构造语法树的末端开始。直至根节点。

* 主要方法：LR分析法，从左至右扫描，自下而上规约。

G(S):

(1) S→SAS

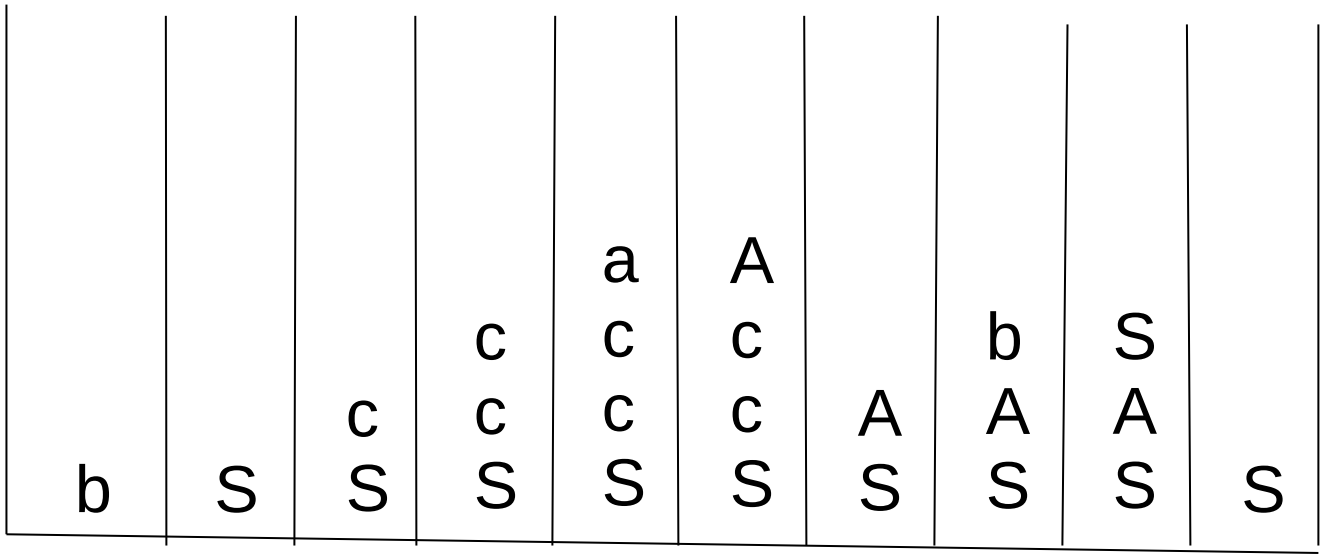
(2) S→b

(3) A→ccA

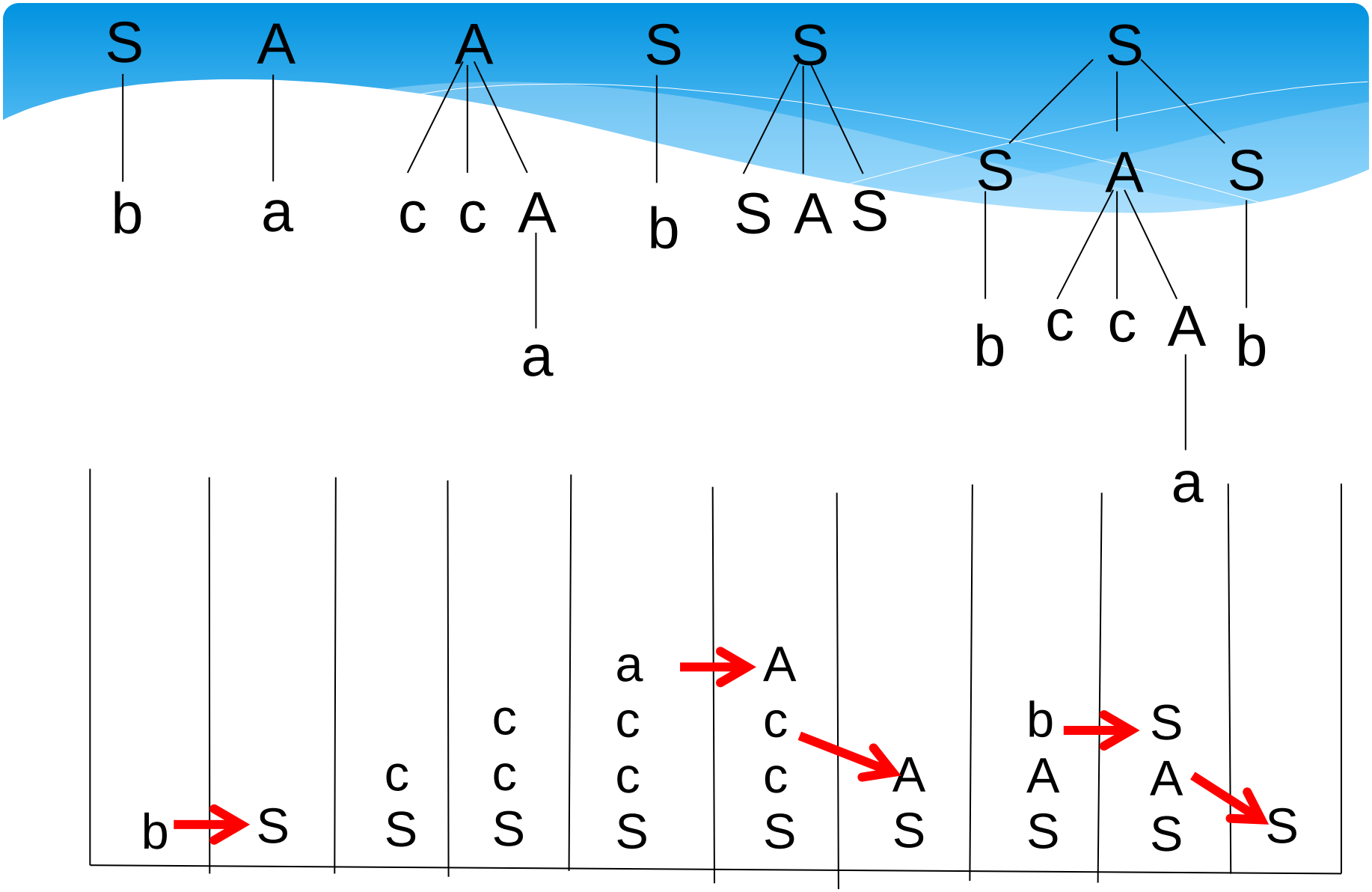
(4) A→a

**输入串bccab**

规约过程

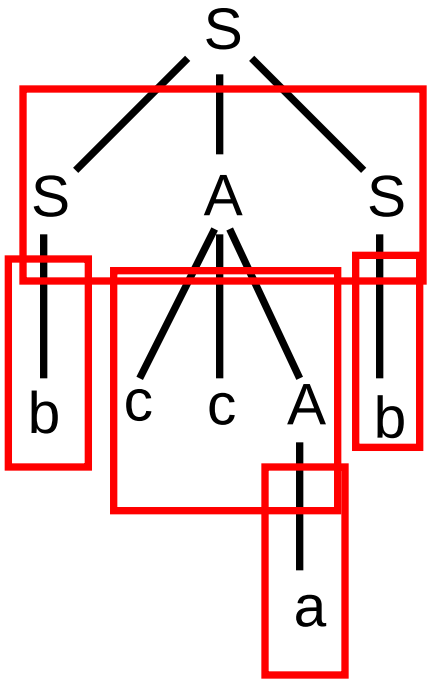



符号栈


S A A S S S
b a c c A b S A S
a
S
S A S
b c c A b
a
b S
c
c S
c
c S
a A
c
c S
A
S
b S
A
A S
S
S

# 剪枝过程

* 语法树的剪枝过程：



# 自下而上可能遇到的问题

* 冲突

规约与规约的冲突

移进与规约的冲突

- 例：设文法G(S)：

(1) S → aAcBe

(2) A → b

(3) A → Ab

(4) B → d

归约与归约的冲突

移进与归约的冲突

试对abbcde进行“移进－归约”分析。

| S A B | a b b c d e |
|---|---|

$S \rightarrow F$

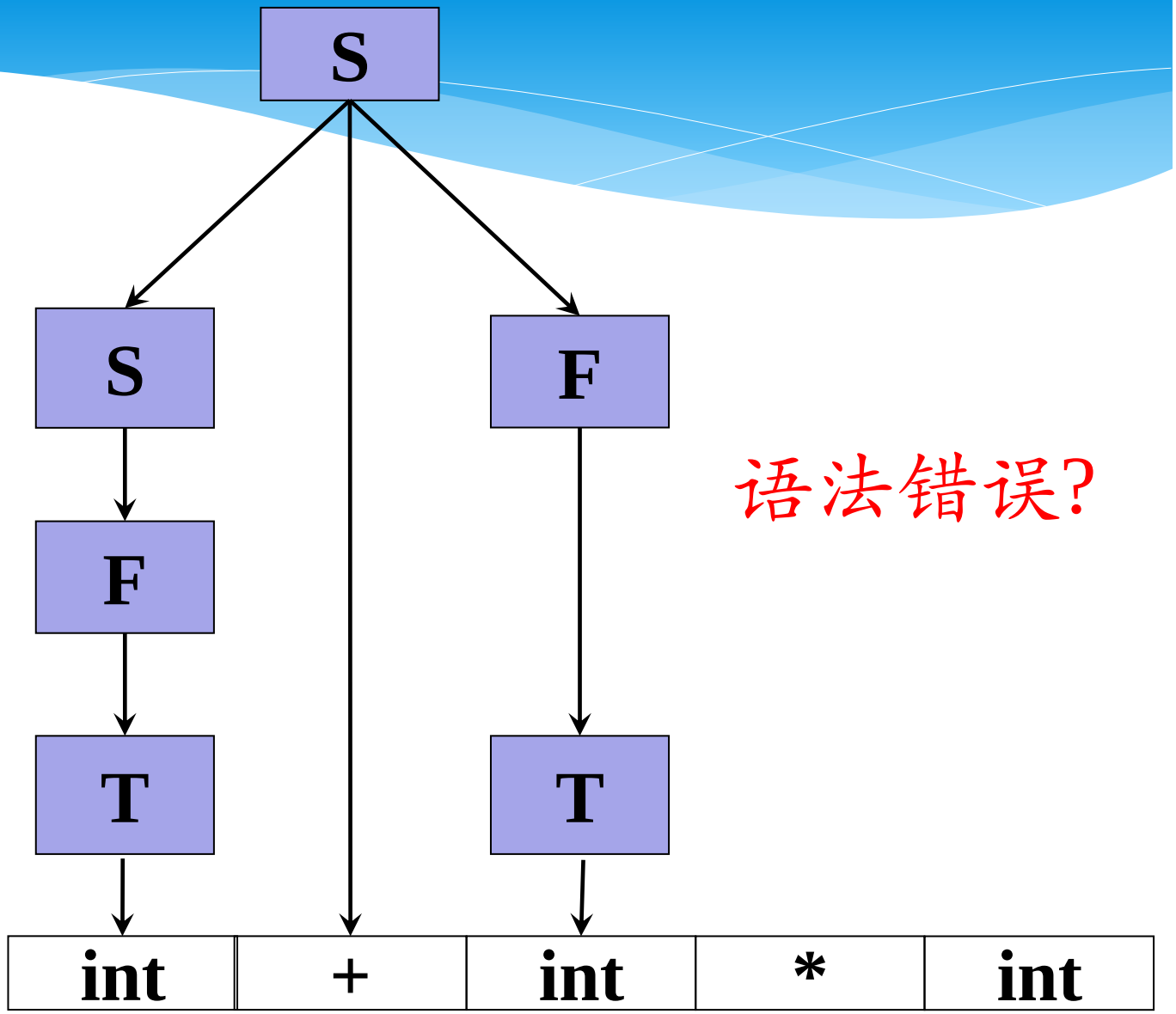
$S \rightarrow S + F$

$F \rightarrow F * T$

$F \rightarrow T$

$T \rightarrow int$

$T \rightarrow (S)$




语法错误?

# 自下而上分析的关键问题

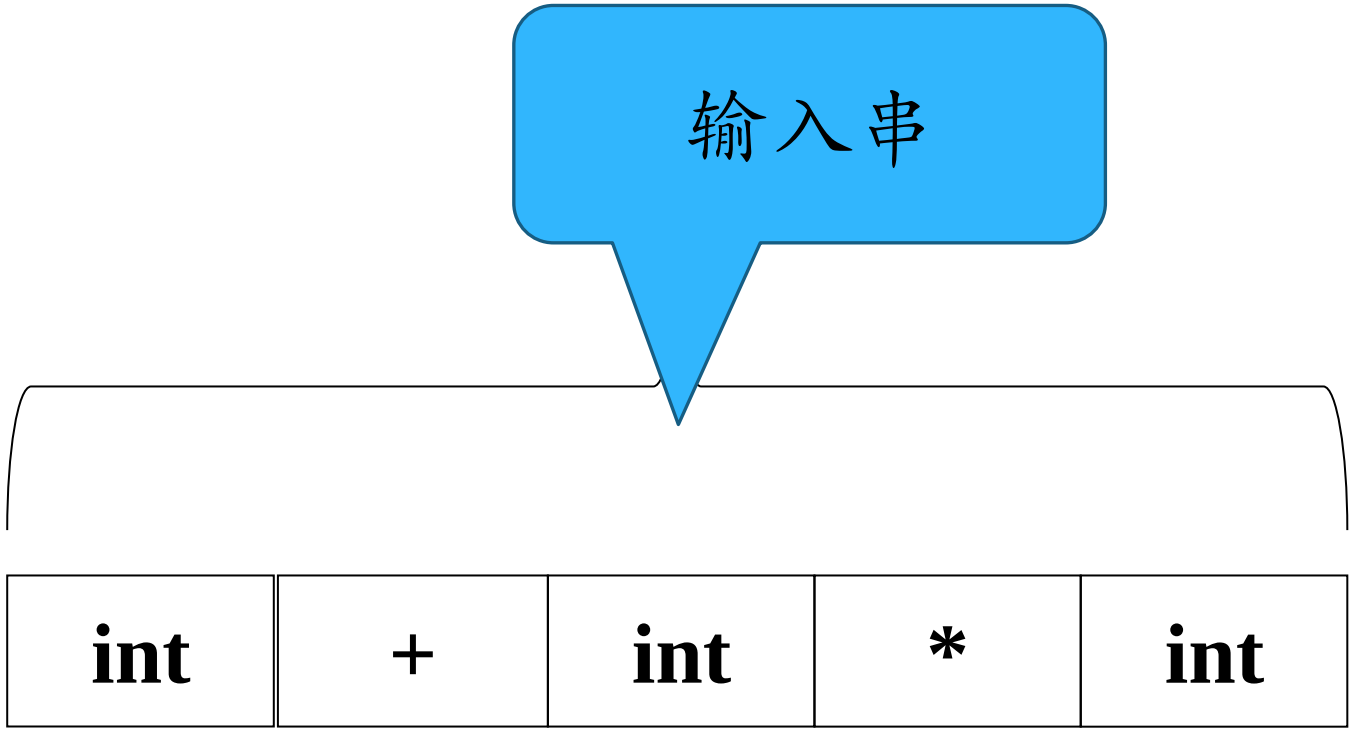
如何判断:

栈顶符号串已经形成可归约串

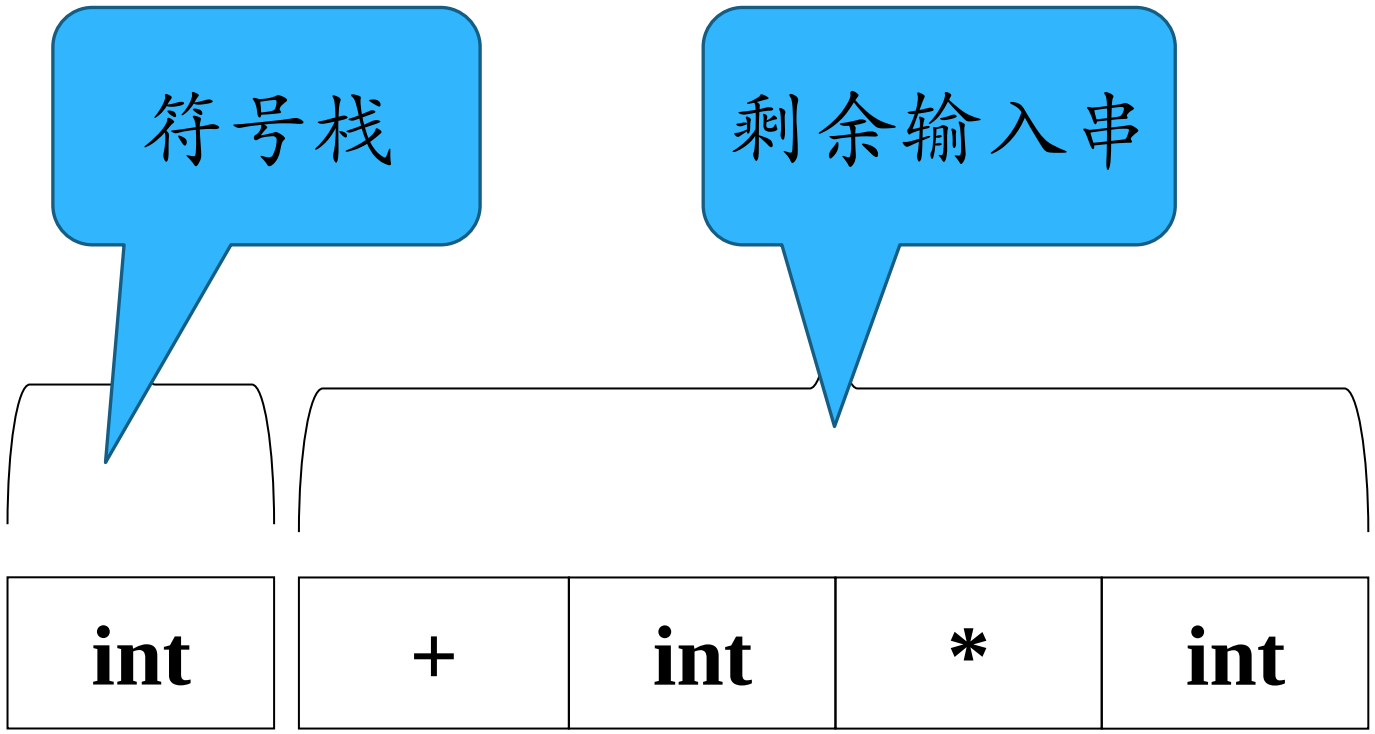
如何进行:

归约

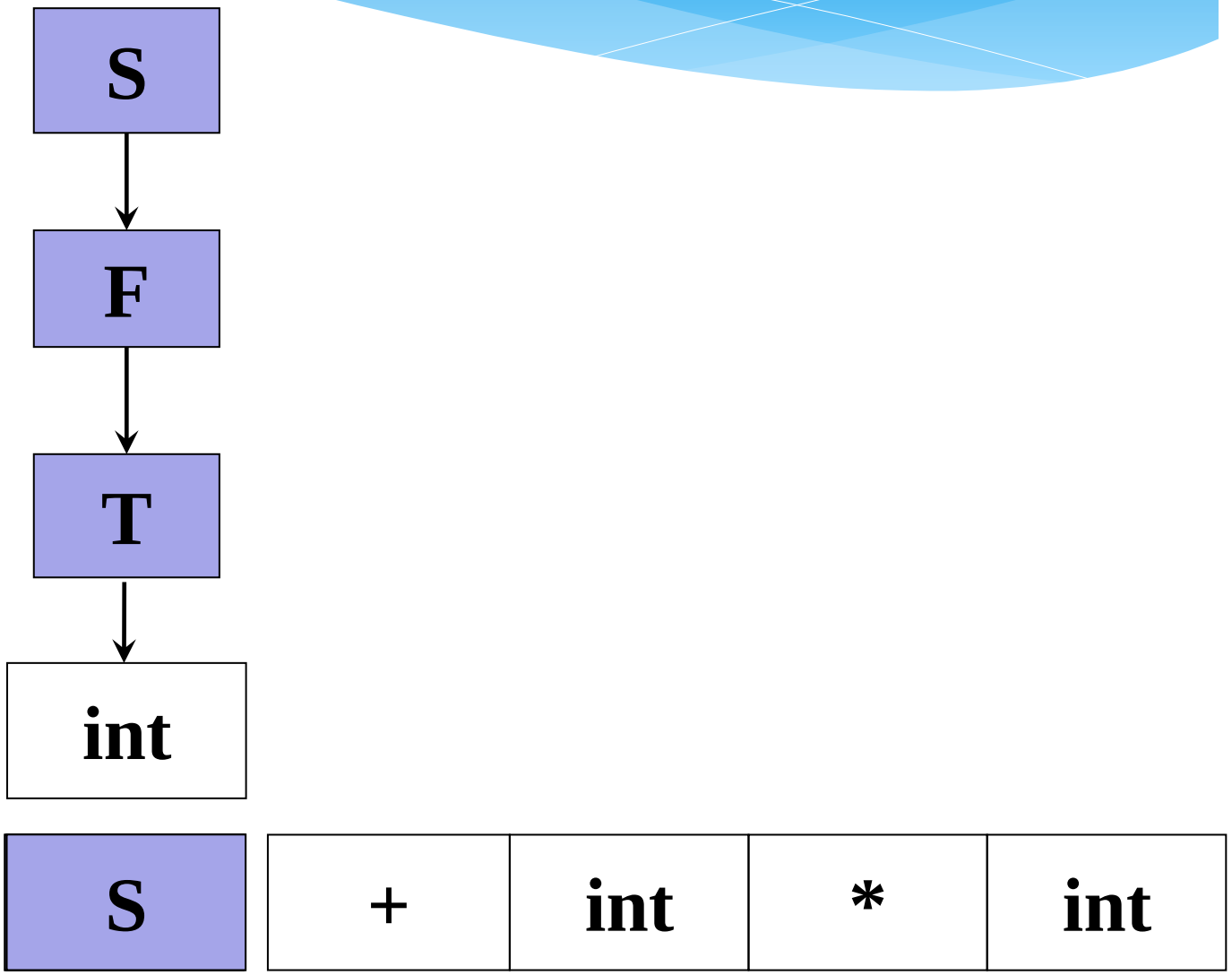
S → F
S → S + F
F → F * T
F → T
T → int
T → (S)



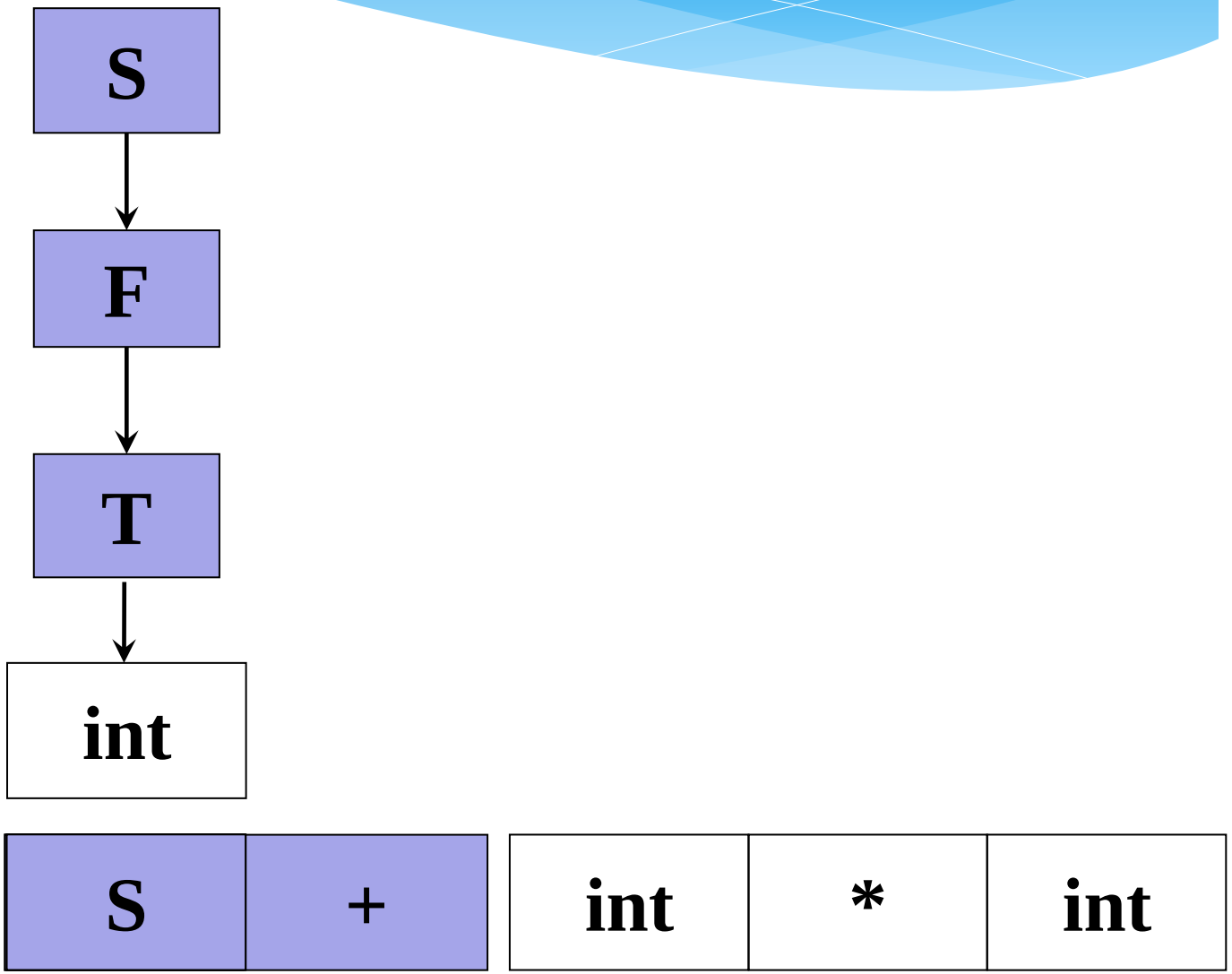
S → F
S → S + F
F → F * T
F → T
T → int
T → (S)



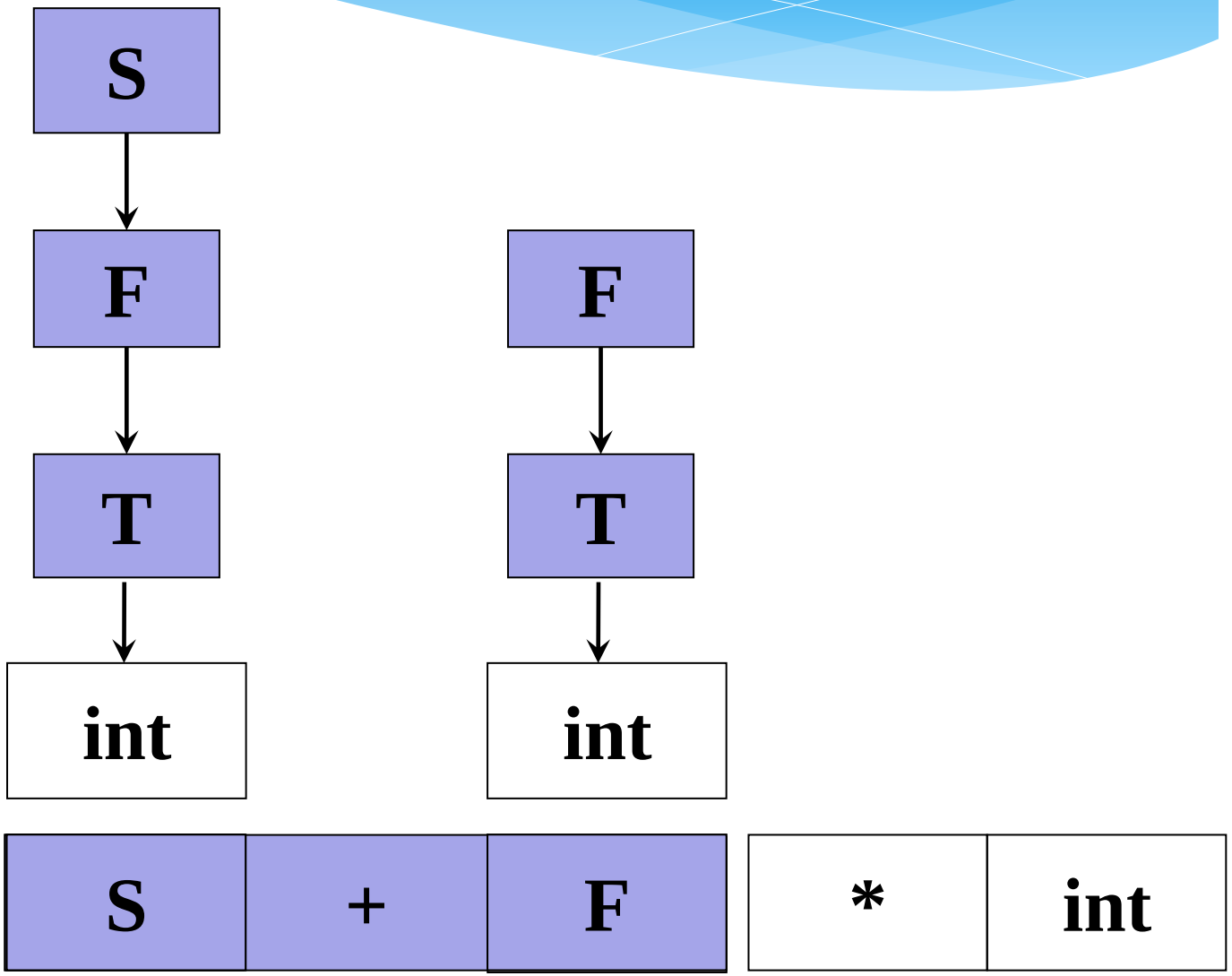
S → F
S → S + F
F → F * T
F → T
T → int
T → (S)



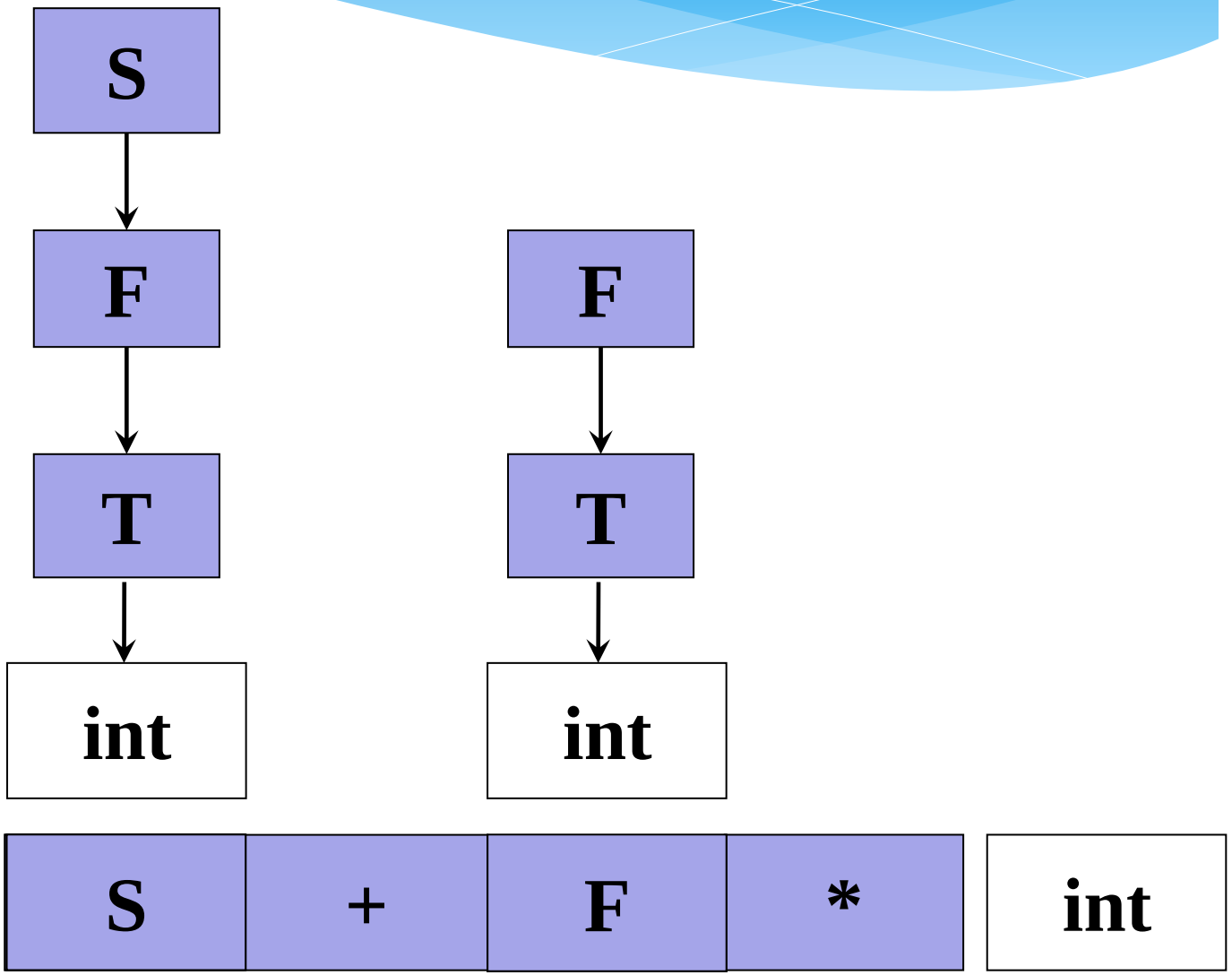
S → F
S → S + F
F → F * T
F → T
T → int
T → (S)



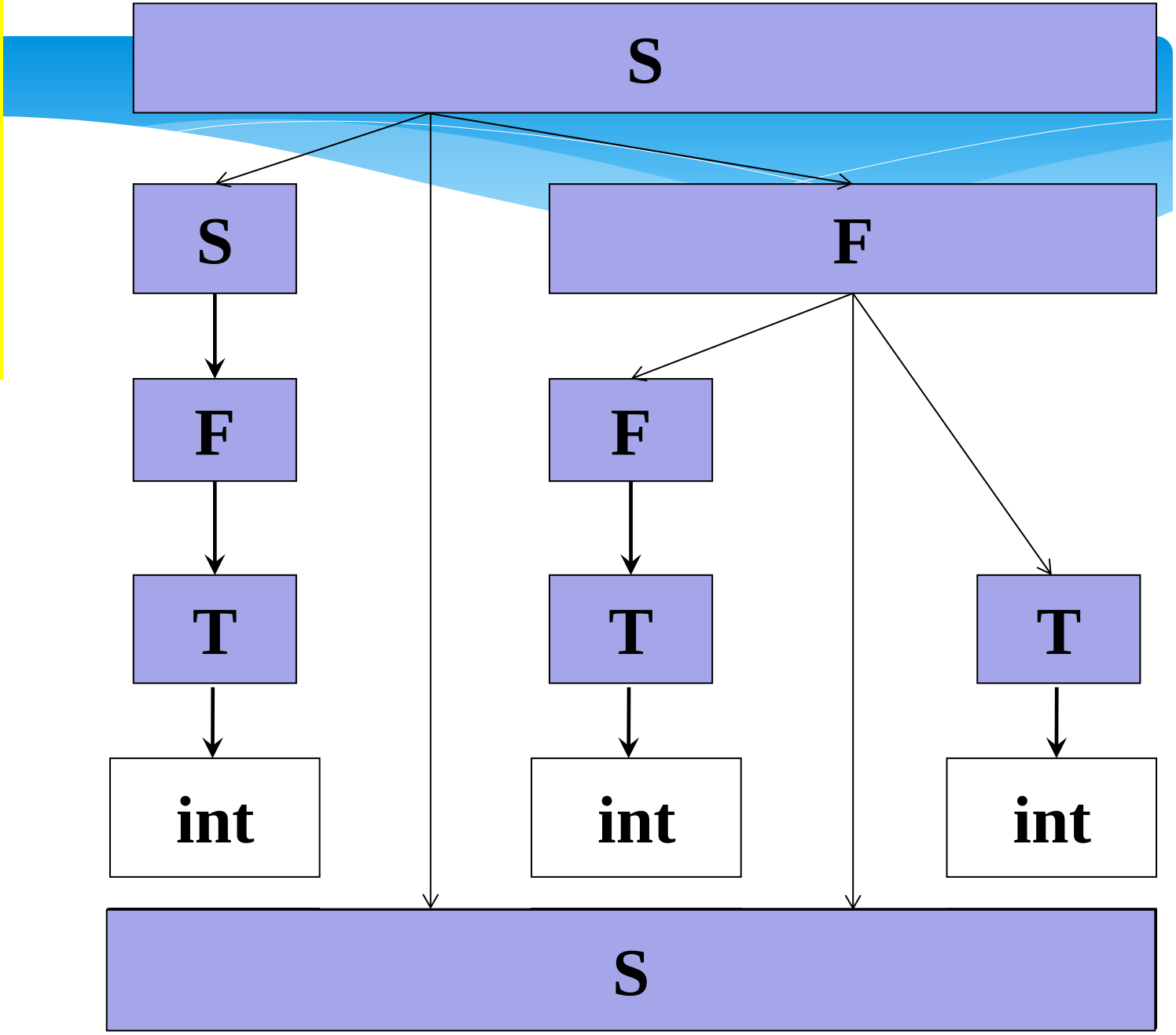
S → F
S → S + F
F → F * T
F → T
T → int
T → (S)



S → F
S → S + F
F → F * T
F → T
T → int
T → (S)



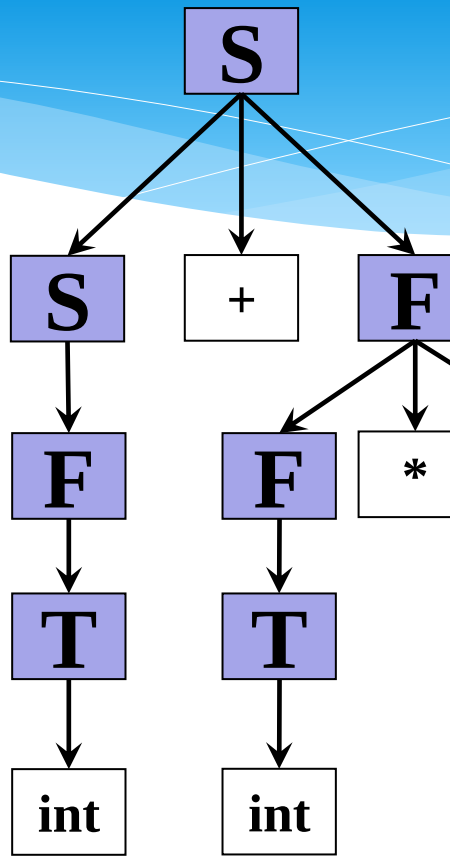
S → F
S → S + F
F → F * T
F → T
T → int
T → (S)



| int | + | int | * | int |
|---|---|---|---|---|

Reduced string

| int |
|---|
| T |
| F |
| S |
| S | + |
| S | + | int |
| S | + | T |
| S | + | F |
| S | + | F | * |
| S | + | F | * | int |
| S | + | F | * | T |
| S | + | F |
| S |




| int |
|---|
| T |
| F |

| int |
|---|
| T |

| int |
|---|
| F*T |
| S+F |

S → F　S → S + F　F → F * T

F → T　T → int　T → (S)

S+F 规约为 S?

# LR分析法

* 句柄?

定义：S是文法G的开始符号，假定$\alpha A\delta$是文法G的一个句型，如果有：

$$S \overset{*}{\Rightarrow} \alpha A\delta \text{ 且 } A \overset{+}{\Rightarrow} \beta ,$$

则称β是句型αβδ关于非终结符A的短语。

特别地如果存在产生式A-> β,则称β 是句型αβδ 关于非终结符A的直接短语。

一个句型的最左直接短语称为该句型的句柄。

例：$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow i$

句型 $i_1*i_2+i_3$ 的短语、直接短语、句柄

**短语：**

$i_1$，$i_2$，$i_3$，$i_1*i_2$，$i_1*i_2+i_3$

**直接短语：**

$i_1$，$i_2$，$i_3$

**句柄：**

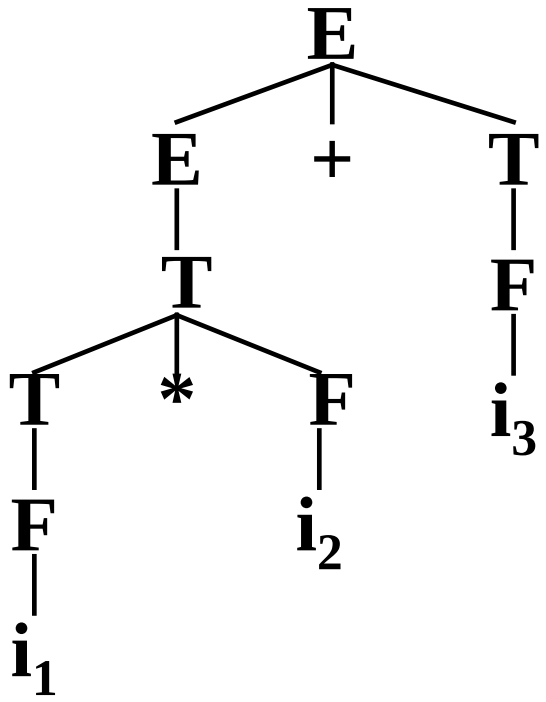
$i_1$，

$$E \overset{*}{\Rightarrow} F*i_2 + i_3$$
$$E \overset{*}{\Rightarrow} i_1*F + i_3$$
$$E \overset{*}{\Rightarrow} i_1*i_2 + F$$
$$E \overset{*}{\Rightarrow} E + i_3$$
$$E \overset{*}{\Rightarrow} E$$

例：E → E+T | T

T → T*F | F

F → i



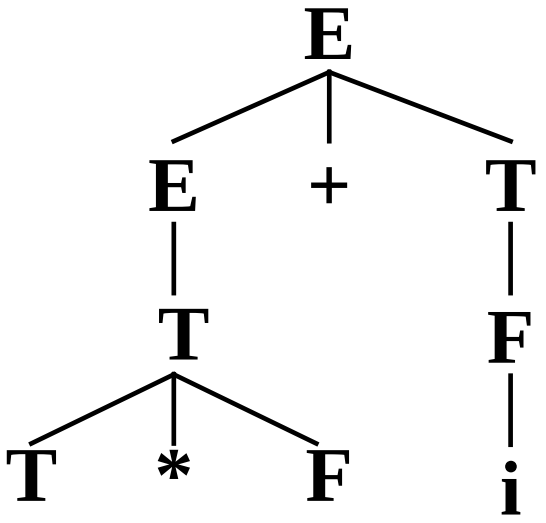

- 在一个句型对应的语法树中
  - 以某非终结符为根的两代及以上的子树的所有末端结点从左到右排列就是相对于该非终结符的一个短语；
  - 如果子树只有二代，则该短语就是直接短语；
  - 最左直接短语—句柄。

例：E → E+T | T

T → T*F | F

F → i

和句型 T*F+i：




短语：

T*F,　　i,　　T*F+i

直接短语：

T*F,　　i

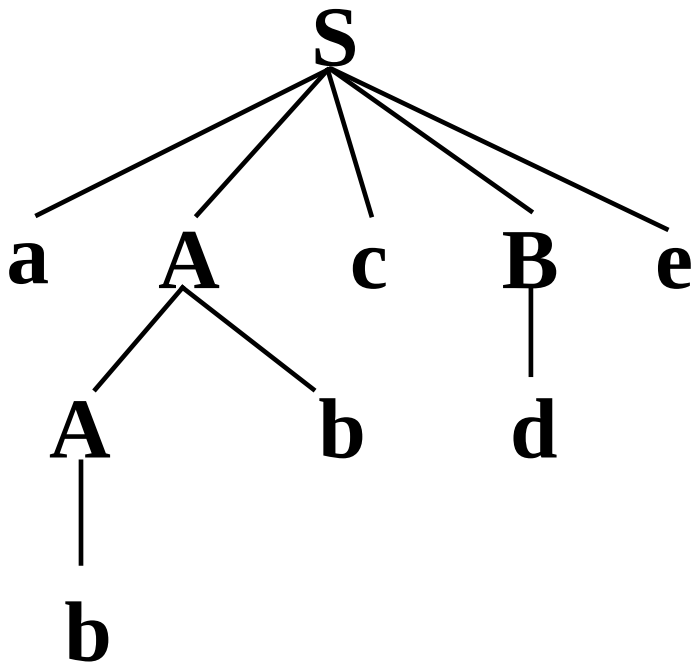
句柄：

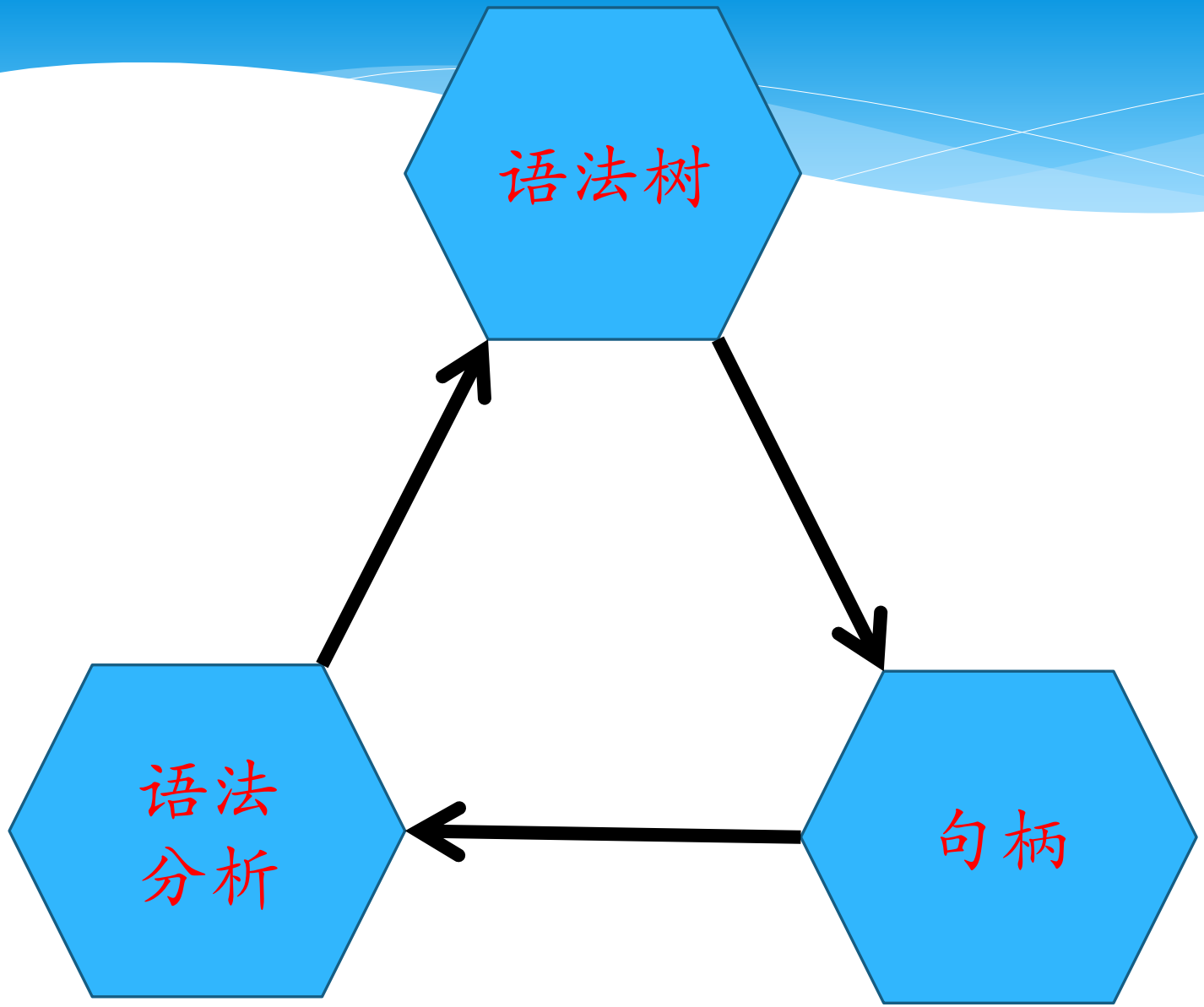
T*F,

# 句柄的用途

* 用来对句子进行规约

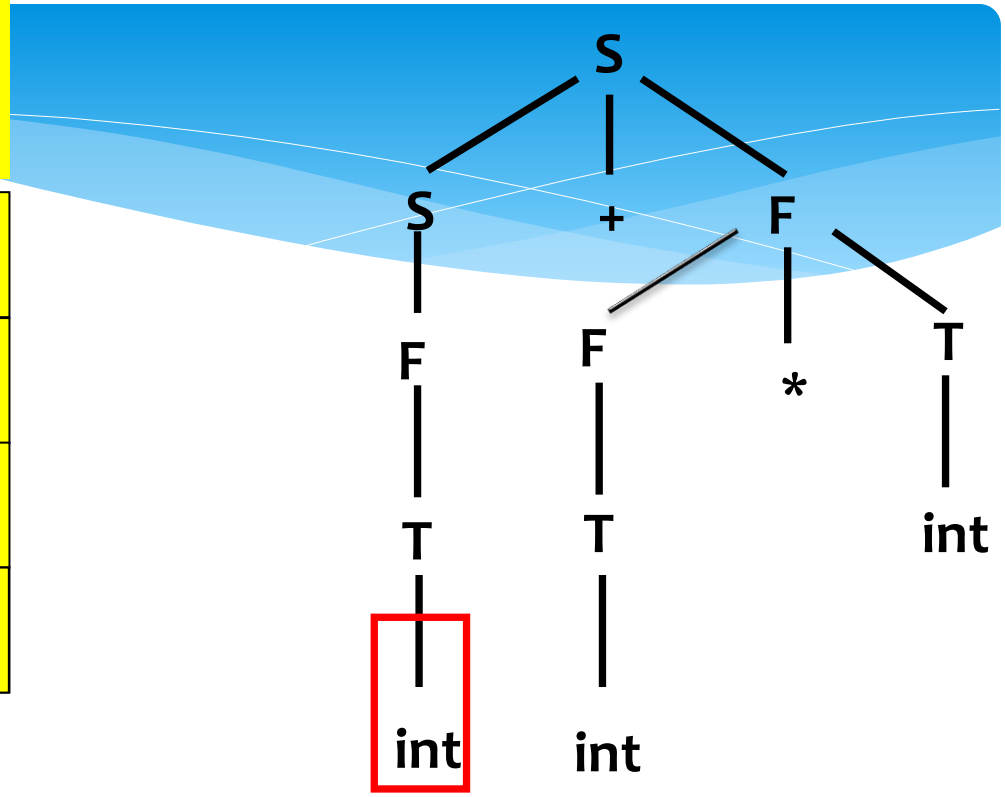
| 句型 | 句柄 | 规约规则 |
| --- | --- | --- |
| abbcde | b | A->b |
| aAbcde | Ab | A->Ab |
| aAcde | d | B->d |
| aAcBe | aAcBe | S->aAcBe |
| S | | |




语法树
句柄
语法
分析

S → F  S → S + F  F → F * T
F → T  T → int

| |
|---|
| S → •S+F |
| S → •F |
| F → •T |
| T → int• |

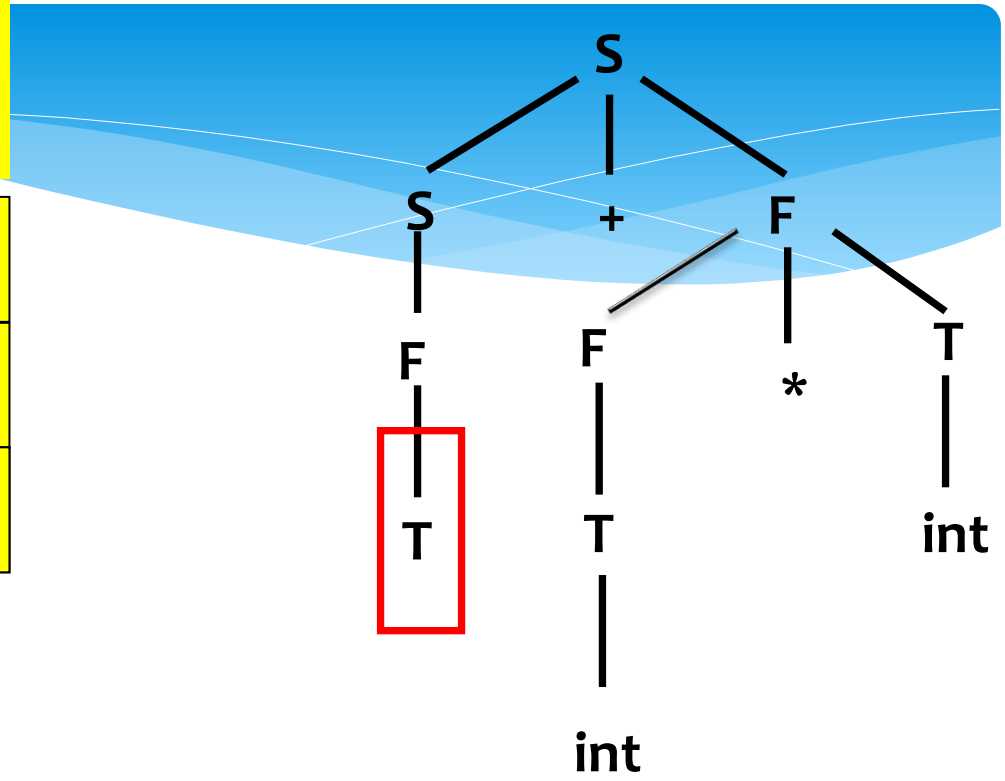



T+int*int

int+int*int

$S \rightarrow F \quad S \rightarrow S + F \quad F \rightarrow F * T$
$F \rightarrow T \quad T \rightarrow int$

| |
|---|
| $S \rightarrow \bullet S+F$ |
| $S \rightarrow \bullet F$ |
| $F \rightarrow T\bullet$ |



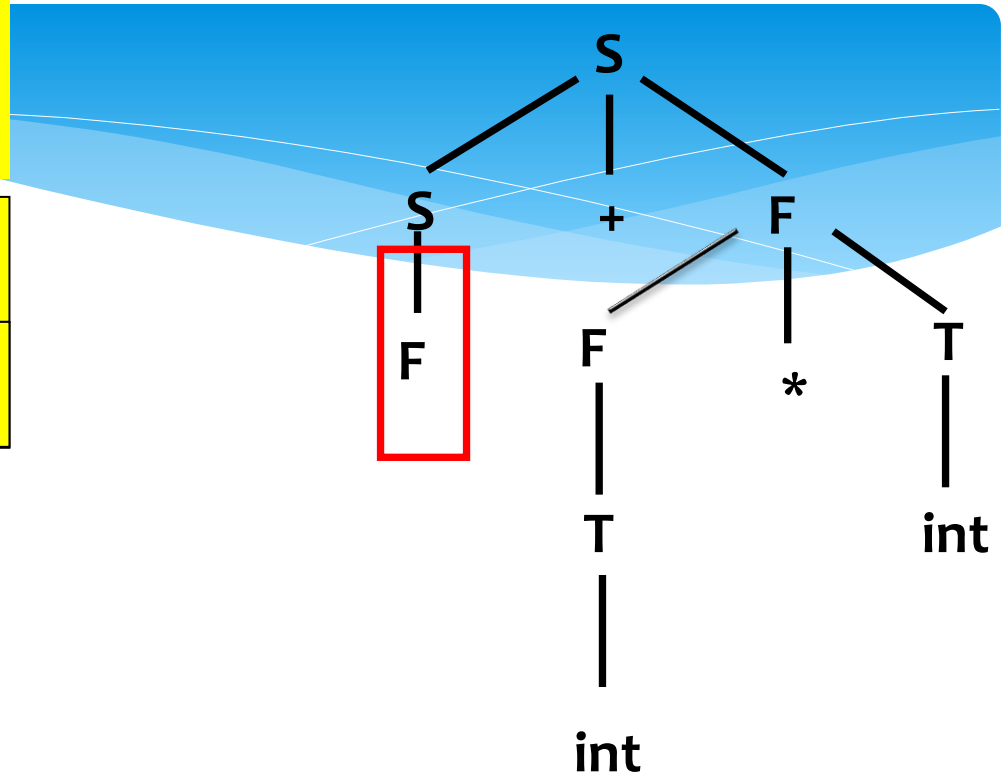

**F+int*int**

**T+int*int**

**int+int*int**

$S \rightarrow F \quad S \rightarrow S + F \quad F \rightarrow F * T$
$F \rightarrow T \quad T \rightarrow int$

| $S \rightarrow \bullet S+F$ |
| --- |
| $S \rightarrow F\bullet$ |




**S+int*int**

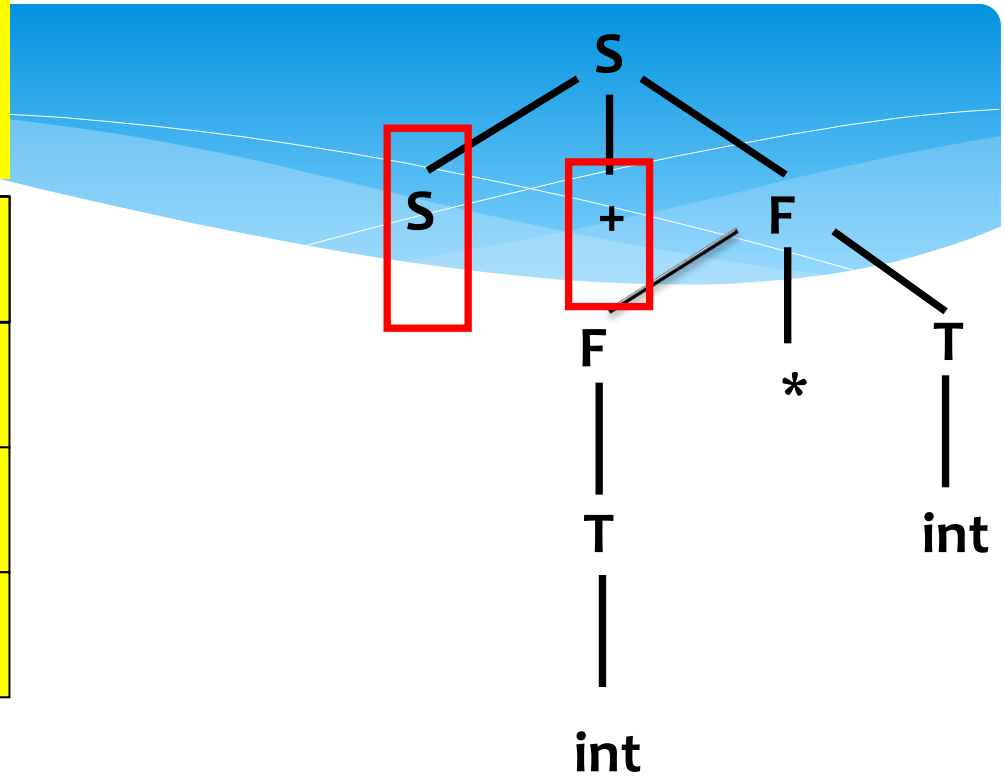
**F+int*int**

**T+int*int**

**int+int*int**

$S \rightarrow F \quad S \rightarrow S + F \quad F \rightarrow F * T$
$F \rightarrow T \quad T \rightarrow int$

| |
|---|
| $S \rightarrow S+\bullet F$ |
| $F \rightarrow \bullet F*T$ |
| $F \rightarrow \bullet T$ |
| $T \rightarrow \bullet int$ |




S+int*int

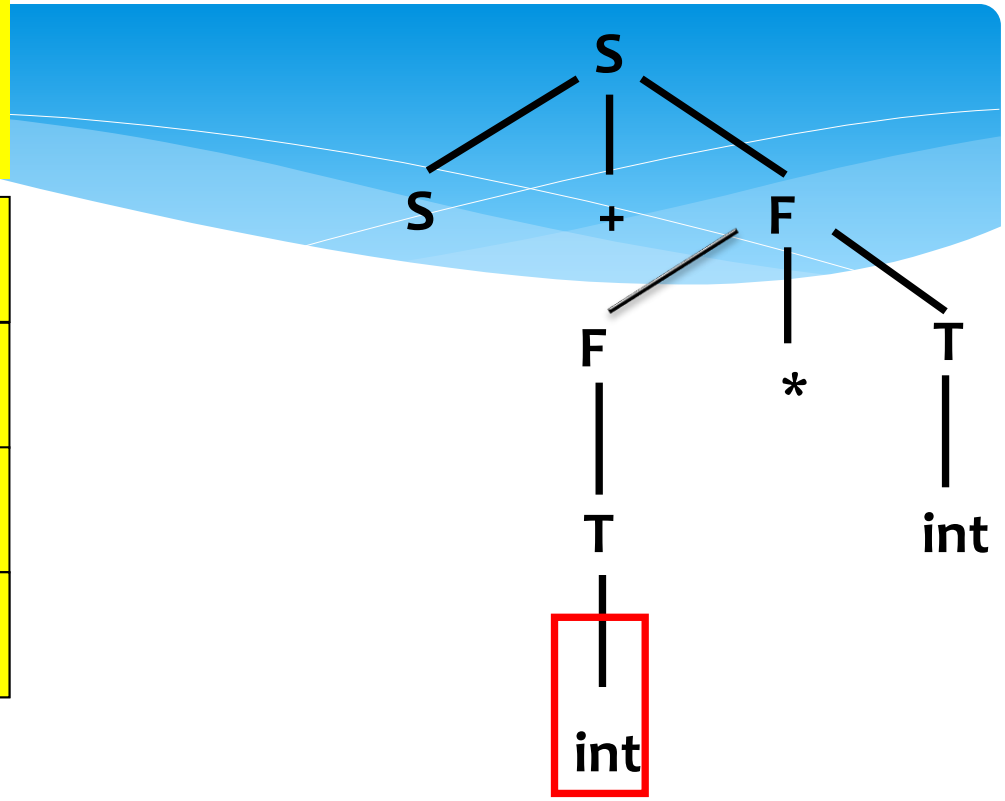
F+int*int

T+int*int

int+int*int

S → F  S → S + F  F → F * T
F → T  T → int

| |
|---|
| S → S+•F |
| F → •F*T |
| F → •T |
| T → int• |




S+T*int

S+int*int

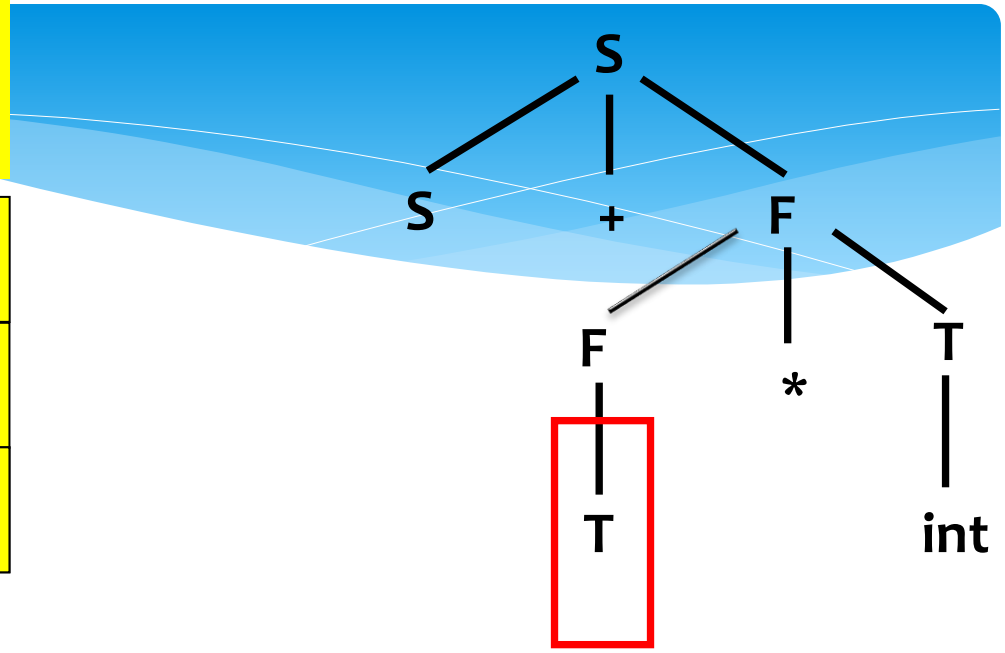
F+int*int

T+int*int

int+int*int

S → F  S → S + F  F → F * T
F → T  T → int

| |
|---|
| S → S+•F |
| F → •F*T |
| F → T• |




S+F*int

S+T*int

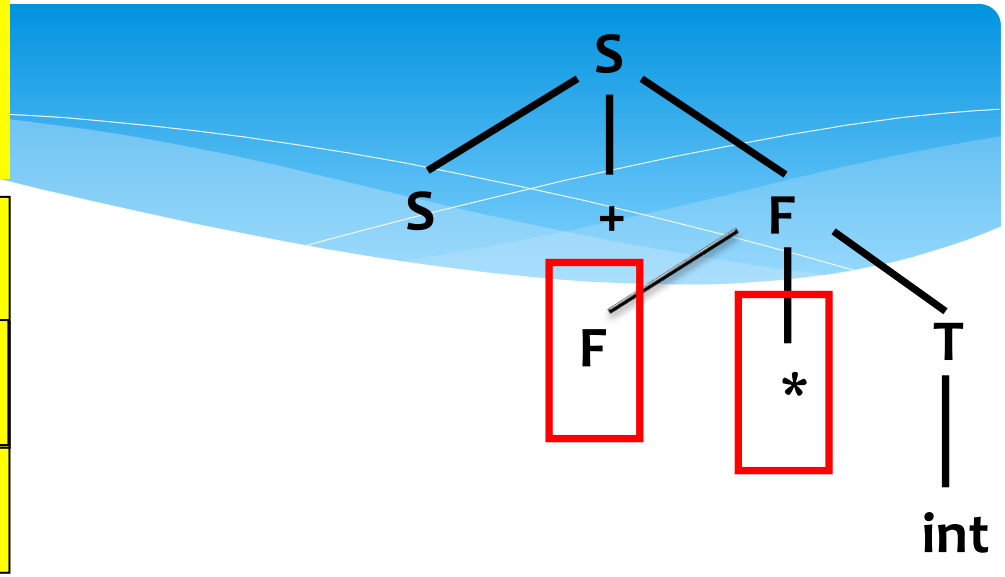
S+int*int

F+int*int

T+int*int

int+int*int

$S \rightarrow F \quad S \rightarrow S + F \quad F \rightarrow F * T$
$F \rightarrow T \quad T \rightarrow int$




$S \rightarrow S+\bullet F$

$F \rightarrow F*\bullet T$

$T \rightarrow \bullet int$

S+F*int

S+T*int

S+int*int

F+int*int

T+int*int

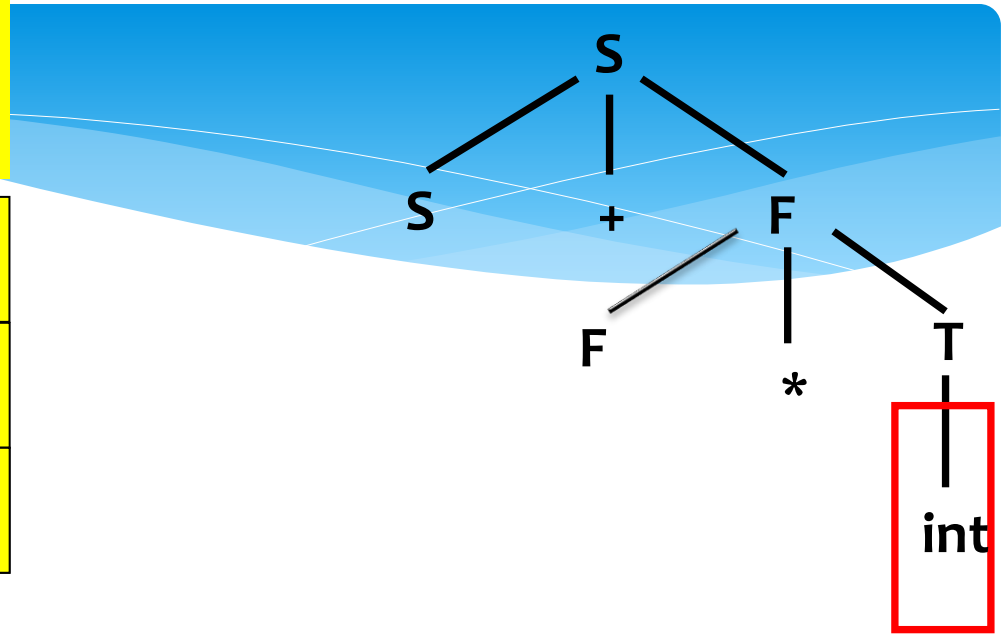
int+int*int

S+F is reduced?

No !

$S \rightarrow F \quad S \rightarrow S + F \quad F \rightarrow F * T$
$F \rightarrow T \quad T \rightarrow int$

| |
|---|
| $S \rightarrow S+\bullet F$ |
| $F \rightarrow F*\bullet T$ |
| $T \rightarrow int\bullet$ |




S+F*T

S+F*int

S+T*int

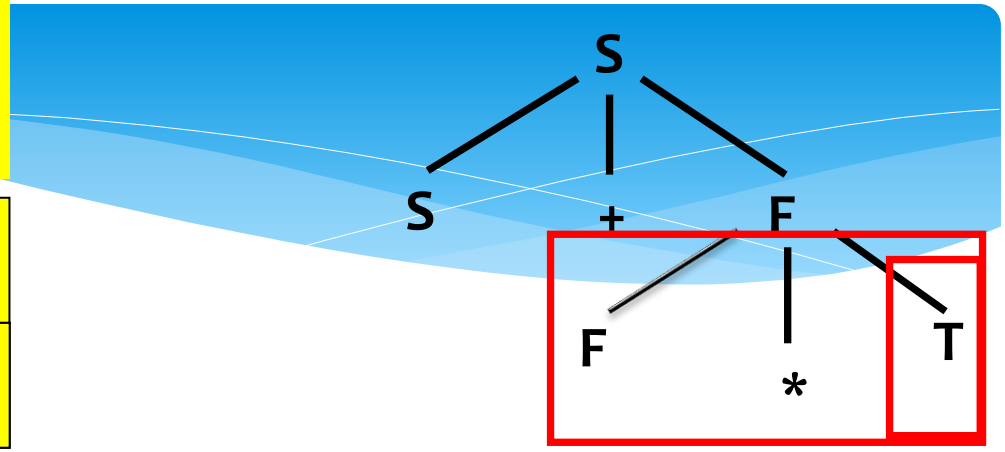
S+int*int

F +int*int

T+int*int

int+int*int

$S \rightarrow F \quad S \rightarrow S + F \quad F \rightarrow F * T$
$F \rightarrow T \quad T \rightarrow int$




$S \rightarrow S+\bullet F$

$F \rightarrow F*T\bullet$

S+F*T

S+F*int

S+T*int

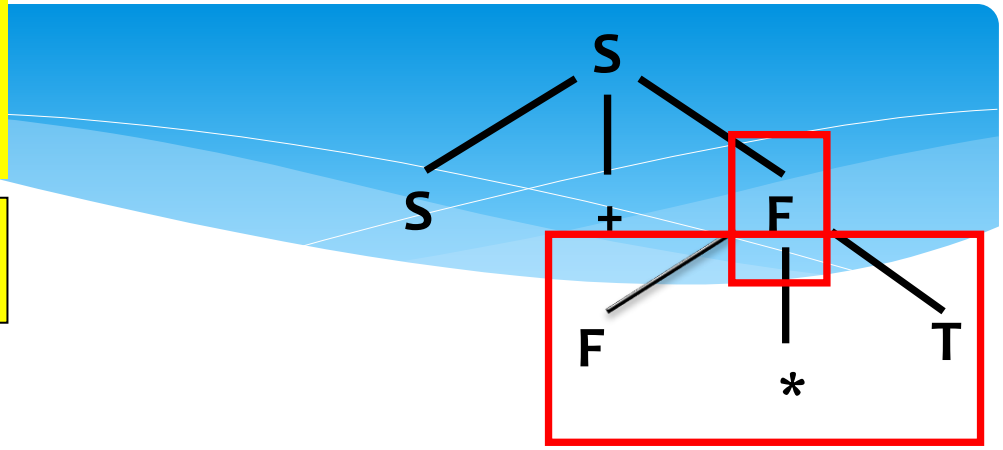
S+int*int

F +int*int

T+int*int

int+int*int

$S \rightarrow F \quad S \rightarrow S + F \quad F \rightarrow F * T$
$F \rightarrow T \quad T \rightarrow int$




$S \rightarrow S+\bullet F$

S+F

S+F*T

S+F*int

S+T*int

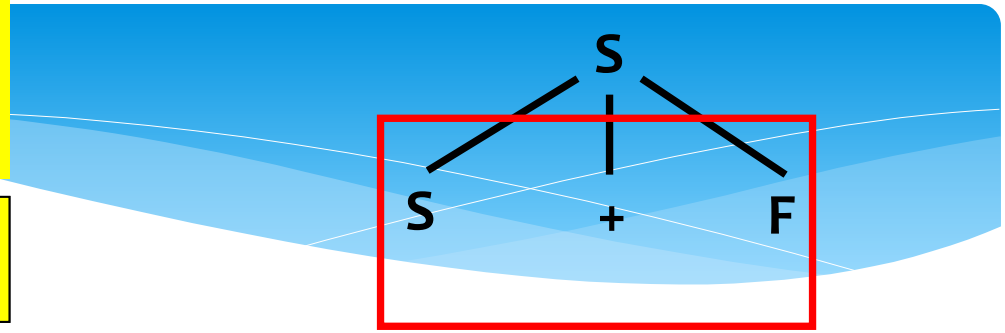
S+int*int

F +int*int

T+int*int

int+int*int

$S \rightarrow F \quad S \rightarrow S + F \quad F \rightarrow F * T$
$F \rightarrow T \quad T \rightarrow int$




S

$S \rightarrow S\bullet$

S+F

S+F*T

S+F*int

S+T*int

S+int*int

F +int*int

T+int*int

int+int*int

# LR分析法

- LR分析法从左向右扫描输入串
- 分析栈中符号，及向前搜索K个输入符号 以确定是否已在栈顶形成句柄，从而决定应采取的动作。
- LR(K)分析法
- 只考虑K=0、1的情况。

* LR(0)、SLR(1)、LR(1)、LALR(1)

# 自下而上分析法--步骤

* **移进**

  读入一个单词并压入栈中，输入串指针后移。

* **规约**

  检查栈顶若干个符号，判断是否可以规约，若能，则以相应产生式左部替换该符号串。

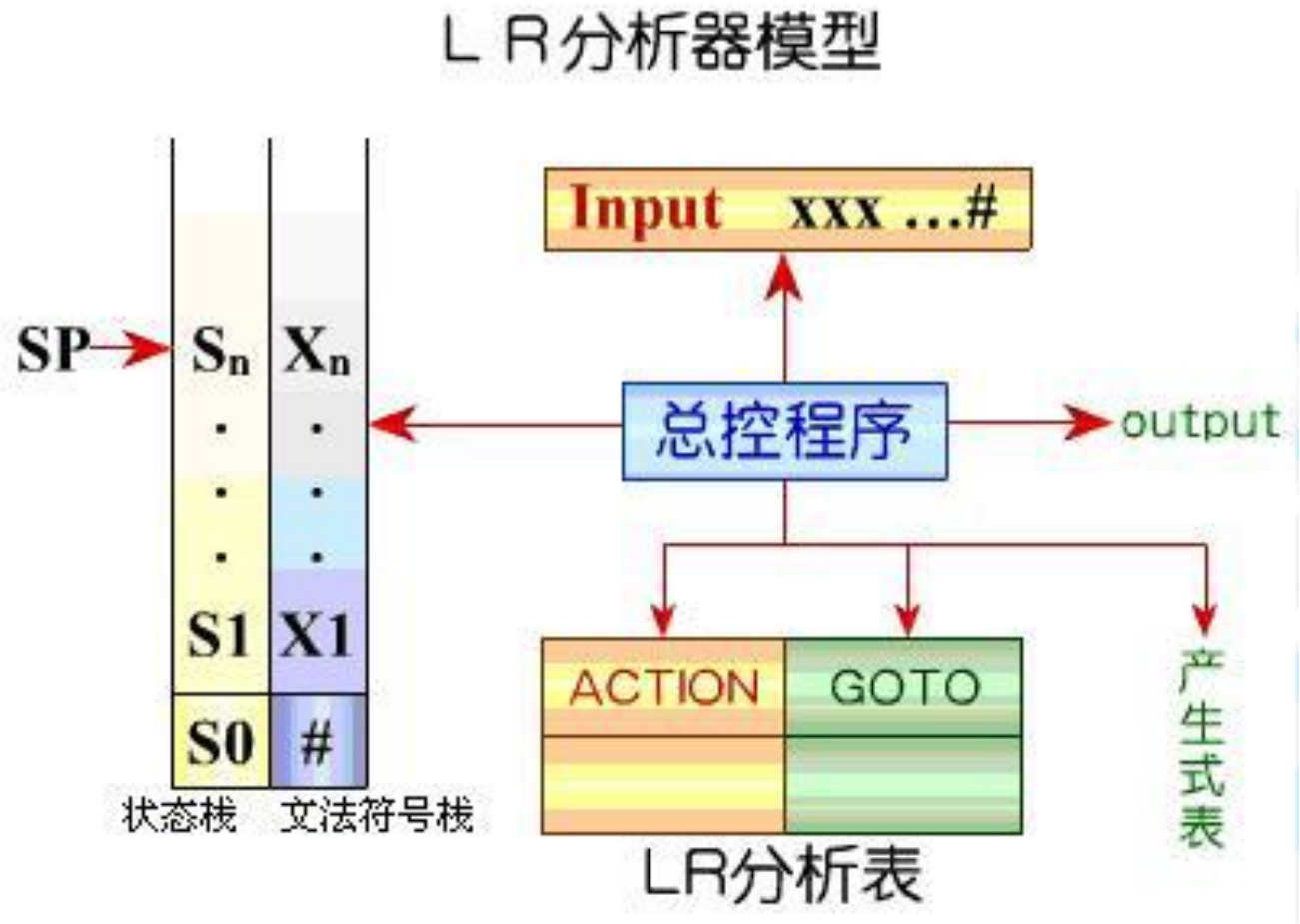
* **识别成功**

  栈内只剩下栈底符号以及文法开始符号，输入串指针处于输入语句的结束符。则语法分析成功。

* **识别失败**

  当输入串指针处于输入语句的结束符时，不能达到第三步的状态，则表示语法错误，转入出错处理程序。

# LR分析器的结构




分析栈

分析表

总控程序

# 分析栈

* 包含状态栈和符号栈

* 状态栈：分析的状态信息
  历史和展望

* 符号栈：分析过程中移进和规约的
  文法符号信息

| | |
|---|---|
| $S_m$ | $X_m$ |
| ⋮ | ⋮ |
| $S_1$ | $X_1$ |
| $S_0$ | # |

状态栈 符号栈
分析栈

# 分析表

* 包括action和goto两个子表
* (1)action[s, a]

  在状态 s下, 当前输入符号为 a 时应采取的分析动作:

  移进, 归约, 接收, 出错。
* (2) goto[s, X]表

  状态及非终结符的二维矩阵

  在状态 s下, 归约后的符号 X 入栈的状态。

| action | goto |
| --- | --- |

LR分析表

# ACTION 子表

* 每一项ACTION[s，a]所规定的四种动作：

1. 移进 (shift) 把(s，a)的下一状态 s’和输入符号 a 推进栈，下一输入符号变成当前输入符号。

2. 归约 (reduce) 指用某产生式 A → β进行归约。若β的长度为 t，归约动作是：去除栈顶 t 个项，使状态 $s_{m-t}$ 变成栈顶状态，然后把 ($s_{m-t}$, A) 的下一状态 $s'$=GOTO[$s_{m-t}$ , A] 和文法符号 A 推进栈.

3. 接受 (accept) 宣布分析成功，停止分析器工作.

4. 报错 (error) 语法出错，记录出错处理程序入口。

# 总控程序

* 根据当前状态栈栈顶符号s以及当前输入符号a查询分析表action[s,a]，根据情况作如下操作：

（1）若action[s,a]=$s_j$，则将状态j推入栈顶，输入指针指向下一个输入符号。

（2）若action[s,a]=$r_j$，则按照第j个产生式$A \rightarrow \beta$规约，假设|β|=t，则将状态栈顶的t个状态出栈，再根据当前栈顶状态$s_i$以及规约后的非终结符A，查goto表，若goto[$s_i$，A]=k，则将状态k推入栈顶。

（3）若action[s,a]=acc，则分析成功，输入串被接受。

（4）若action[s,a]或goto[$s_i$,A]为空白，则转出错处理。

# LR分析表及产生式排序

(1)E→E+T
(2)E→T
(3)T→T*F
(4)T→F
(5)F→(E)
(6)F→i

教材第93页
图4-21

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

# 输入串： i+i*i

ACTION[0, i]=S5

| | 状态栈 | 符号栈 | 剩余输入串 | 动作 |
|---|---|---|---|---|
| 1 | 0 | # | i+i*i# | S5 |
| 2 | 05 | # i | +i*i# | R6 F → i |
| 3 | 03 GOTO[0,F]=3 | # F | +i*i# | R4 T → F |
| 4 | 02 GOTO[0,T]=2 | # T | +i*i# | R2 E → T |
| 5 | 01 GOTO[0,E]=1 | # E | +i*i# | S6 |
| 6 | 016 | # E + | i*i# | S5 |
| 7 | 0165 | # E + i | *i# | R6 F → i |
| 8 | 0163 | # E + F | *i# | R4 T → F |

## 输入串： i+i*i

| | State stack | Symbol stack | Input | Action |
|---|---|---|---|---|
| 8 | 0163 | # E + F | *i# | R4 T → F |
| 9 | 0169 | # E + T | *i# | S7 |
| 10 | 01697 | # E + T * | i# | S5 |
| 11 | 016975 | # E + T * i | # | R6 F → i |
| 12 | 0169710 | # E + T * F | # | R3 T → T*F |
| 13 | 0169 | # E + T | # | R1 E → E+T |
| 14 | 01 | # E | # | ACC |
| 15 | | | | |

* 定义：对于一个文法，如果能够构造一张分析表，使得它的每个入口均是唯一确定的，则这个文法就称为**LR文法**。
* 定义：一个文法，如果能用一个每步顶多向前检查k个输入符号的LR分析器进行分析，则这个文法就称为**LR(k)文法**.
* LR文法不是二义的，二义文法肯定不会是LR的。

# 自下而上分析方法分类

* 算符优先分析法：

  每一步规约，都以栈顶符号串是否已经形成最左素短语为标准。

* LR分析法：

  每一步规约，都以栈顶符号串是否已经形成句柄为准。

# 算符优先分析法

算符优先分析法是根据算符之间的优先关系设计的一种简单、直观的自下而上的语法分析分析，易于手工实现。

只适用于算符优先文法，特别适合于分析程序设计语言中的各类表达式。

将句型中的终结符号当做“算符”，借助于算符之间的优先关系确定可规约串。

# 算符优先分析法

## 算符文法

对于文法G[S]，如果其任一产生式的右部都不含两个相继或并列的非终结符，即不含如下形式的产生式：

$$P \rightarrow \cdots QR \cdots$$

则称文法G为算符文法。

算符文法句型的一般形式为：
$\#N_1a_1N_2a_2 \cdots N_na_nN_{n+1}\#$，其中每个 $a_i$ 都是终结符，每个$N_n$都是非终结符或者ε。

任意两个终结符之间至多只有一个非终结符。

# 算符优先分析法

## 算符优先关系

假定文法G是一个不包含ε产生式的算符文法。G的终结符号对a 和b的优先关系包含三种形式：

(1) $a \asymp b$

当且仅当文法G中含有形如$P \to \cdots ab \cdots$或$P \to \cdots aQb \cdots$的产生式

(2) $a \lessdot b$

当且仅当文法G中含有形如$P \to \cdots aR \cdots$，且$R \overset{+}{\Rightarrow} b \cdots$或$R \overset{+}{\Rightarrow} Qb \cdots$

(3) $a \gtrdot b$

当且仅当文法G中含有形如$P \to \cdots Rb \cdots$，且$R \overset{+}{\Rightarrow} \cdots a$或$R \overset{+}{\Rightarrow} \cdots aQ$

# 算符优先分析法

## 算符优先文法

定义：如果算符文法G中的任意一对终结符a和b，至多只满足下述三种关系之一：

$$a \asymp b,\ a \lessdot b,\ a \gtrdot b$$

则称G是一个算符优先文法。

$E \to E+E$
$E \to E*E$
$E \to (E)$
$E \to i$

例：证明文法G(E)不是算符优先文法。

因为：$E \to E+E$，且$E \Longrightarrow E*E$，根据规则2有 $+ \lessdot *$；

因为：$E \to E*E$，且$E \Longrightarrow E+E$，根据规则3有 $* \lessdot +$；

所以文法G(E)不是算符优先文法。

# 算符优先分析法

## 算符优先关系表

表示算符之间的优先关系的表格。

垂直维度表示位于左侧的算符，水平维度表示位于右侧的算符。

| | + | * | ↑ | i | ( | ) | # |
|---|---|---|---|---|---|---|---|
| + | ≻ | ≺ | ≺ | ≺ | ≺ | ≻ | ≻ |
| * | ≻ | ≻ | ≺ | ≺ | ≺ | ≻ | ≻ |
| ↑ | ≻ | ≻ | ≺ | ≺ | ≺ | ≻ | ≻ |
| i | ≻ | ≻ | ≻ | | | ≻ | ≻ |
| ( | ≺ | ≺ | ≺ | ≺ | ≺ | ≍ | |
| ) | ≻ | ≻ | ≻ | | | ≻ | ≻ |
| # | ≺ | ≺ | ≺ | ≺ | ≺ | | ≍ |

$E' \rightarrow \#E\#$

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow P \uparrow F \mid P$

$P \rightarrow (E) \mid i$

# 算符优先分析法

## 算符优先关系表

注意：

(1) 终结符a和b之间可能没有优先关系，空白表示错误关系

(2) 相同的终结符之间的优先关系不一定是≍

(3) 如果有a≍b，不一定有b≍a

(4) 如果有a≺b，不一定有b≻a

(5) 如果有a≺b ≺c，不一定有 a≺c

| | + | * | ↑ | i | ( | ) | # |
|---|---|---|---|---|---|---|---|
| + | ≻ | ≺ | ≺ | ≺ | ≺ | ≻ | ≻ |
| * | ≻ | ≻ | ≺ | ≺ | ≺ | ≻ | ≻ |
| ↑ | ≻ | ≻ | ≺ | ≺ | ≺ | ≻ | ≻ |
| i | ≻ | ≻ | ≻ | | | ≻ | ≻ |
| ( | ≺ | ≺ | ≺ | ≺ | ≺ | ≍ | |
| ) | ≻ | ≻ | ≻ | | | ≻ | ≻ |
| # | ≺ | ≺ | ≺ | ≺ | ≺ | | ≍ |

# 算符优先分析法

## 算符优先分析过程

开始状态：

分析栈中为：#，剩余输入串为：输入串#

分析规则：

(1) 如果分析栈栈顶的终结符优先级别低于下一输入符号，或与之优先级别相同，则应将下一输入符号移进分析栈，等待规约；

(2) 如果分析栈栈顶的终结符优先级别高于下一输入符号，则表明栈中形成了当前优先级别最高的串，应进行规约。

结束状态：

剩余输入串为：#，分析栈中为：#S，表示分析成功，否则分析失败。

# 算符优先分析法

## 算符优先分析过程

(1) 从左至右扫描输入符号并移入分析栈，并查算符优先分析表；

(2) 如果分析栈栈顶的终结符优先级别高于下一输入符号，则表明栈中形成了当前优先级别最高的串，应进行规约。

结束状态：

注意：算法优先分析过程中规约后的非终结符名称并不重要，可统一用某一个大写字母表示。

# 算符优先分析法

## 算符优先分析过程

对表达式文法G，当输入串为#i+i*i#时的分析过程：

$S' \rightarrow \#S\#$
$S \rightarrow S+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow P \uparrow F \mid P$
$P \rightarrow (S) \mid i$

| | 分析栈 | 剩余输入串 | 动作 |
|---|---|---|---|
| 0 | # | i+i*i# | 初始状态，#进栈 |
| 1 | #i | +i*i# | #<i，i进栈 |
| 2 | #N | +i*i# | #<i>+，规约为N |
| 3 | #N+ | i*i# | #<+，+进栈 |
| 4 | #N+i | *i# | +<i，i进栈 |
| 5 | #N+N | *i# | +<i>*，规约为N |
| 6 | #N+N* | i# | +<*，*进栈 |

| | + | * | ↑ | i | ( | ) | # |
|---|---|---|---|---|---|---|---|
| + | ≻ | ≺ | ≺ | ≺ | ≺ | ≻ | ≻ |
| * | ≻ | ≻ | ≺ | ≺ | ≺ | ≻ | ≻ |
| ↑ | ≻ | ≻ | ≺ | ≺ | ≺ | ≻ | ≻ |
| i | ≻ | ≻ | ≻ | | | ≻ | ≻ |
| ( | ≺ | ≺ | ≺ | ≺ | ≺ | ≍ | |
| ) | ≻ | ≻ | ≻ | | | ≻ | ≻ |
| # | ≺ | ≺ | ≺ | ≺ | ≺ | | ≍ |

# 算符优先分析法

## 算符优先分析过程

对表达式文法G，当输入串为#i+i*i#时的分析过程：

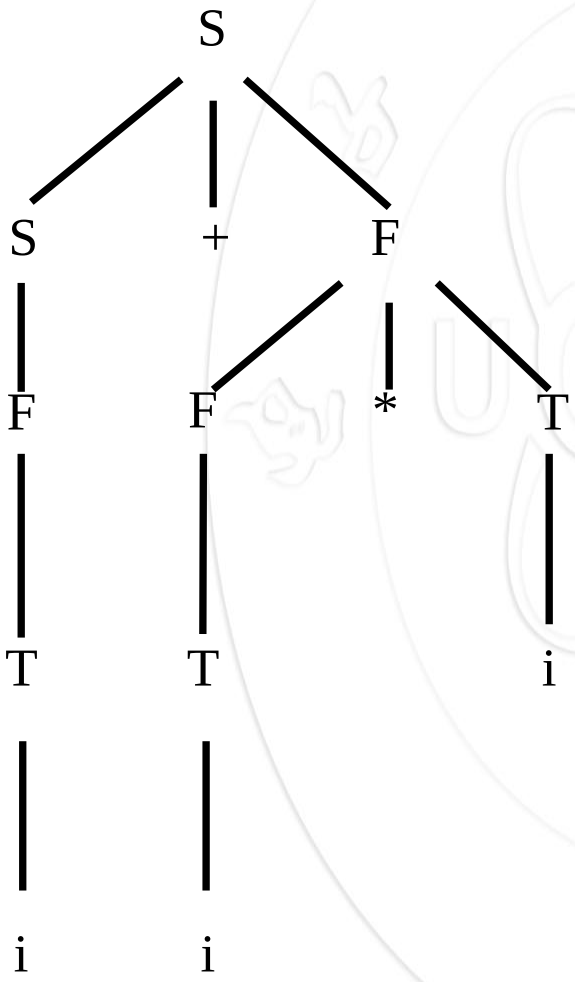
$S' \rightarrow \#S\#$
$S \rightarrow S+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow P \uparrow F \mid P$
$P \rightarrow (S) \mid i$

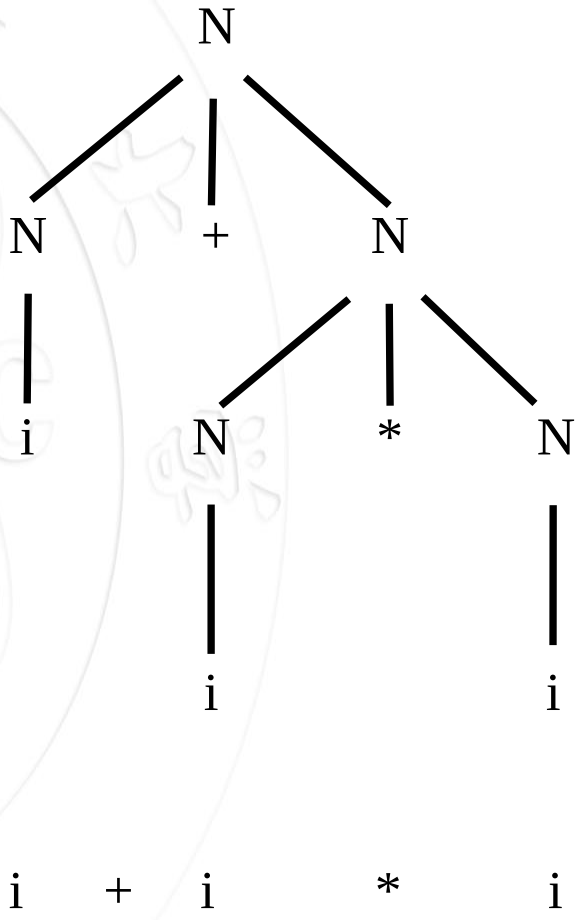
| | 分析栈 | 剩余输入串 | 动作 |
|---|---|---|---|
| 6 | #N+N* | i# | +<*，*进栈 |
| 7 | #N+N*i | # | *<i，i进栈 |
| 8 | #N+N*N | # | *<i>#，规约为N |
| 9 | #N+N | # | +<*>#，规约为N |
| 10 | #N+i | *i# | #<+>#，规约为N |
| 11 | #N | # | +<i>*，规约为N |
| 12 | | | 分析结束 |

| | + | * | ↑ | i | ( | ) | # |
|---|---|---|---|---|---|---|---|
| + | ⋗ | ⋖ | ⋖ | ⋖ | ⋖ | ⋗ | ⋗ |
| * | ⋗ | ⋗ | ⋖ | ⋖ | ⋖ | ⋗ | ⋗ |
| ↑ | ⋗ | ⋗ | ⋖ | ⋖ | ⋖ | ⋗ | ⋗ |
| i | ⋗ | ⋗ | ⋗ | | | ⋗ | ⋗ |
| ( | ⋖ | ⋖ | ⋖ | ⋖ | ⋖ | ≒ | |
| ) | ⋗ | ⋗ | ⋗ | | | ⋗ | ⋗ |
| # | ⋖ | ⋖ | ⋖ | ⋖ | ⋖ | | ≒ |

# 算符优先分析法

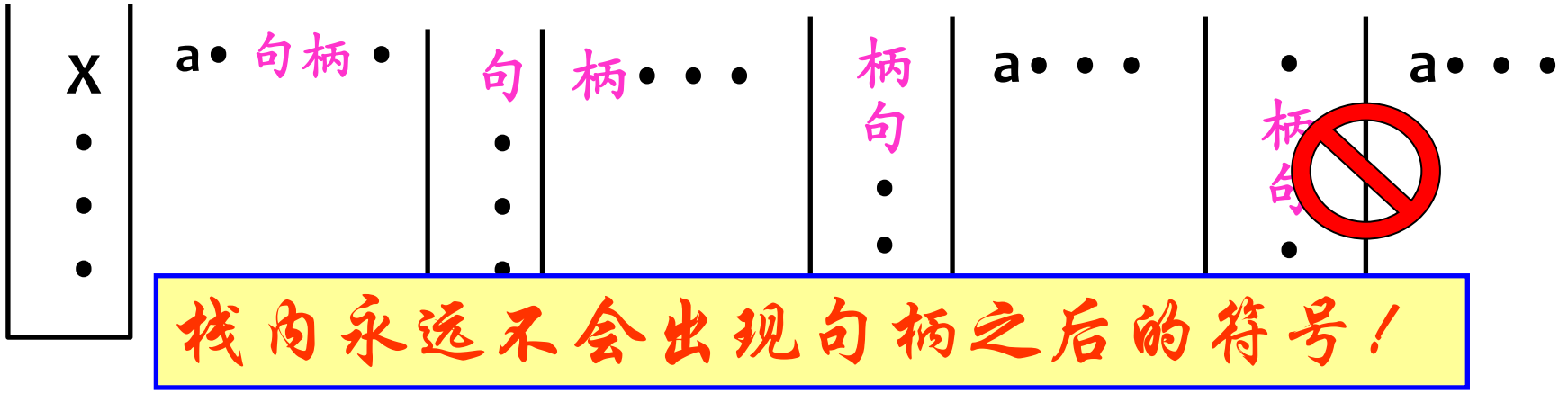
## 算符优先分析语法树



# 分析表的构造

* 规范归约过程中
  * 栈内的符号串和扫描剩下的输入符号串构成了一个规范句型
  * 栈内的如果出现句柄，句柄一定在栈的顶部



* 规约符号串总是在栈顶
* 句柄之后的待入栈符号总是终结符
* 规范句型在符号栈中的符号串是规范句型的前缀。

* 为了知道何时可以进行规约，引入活前缀的概念

# 准备一：活前缀

字符串的前缀、真前缀?

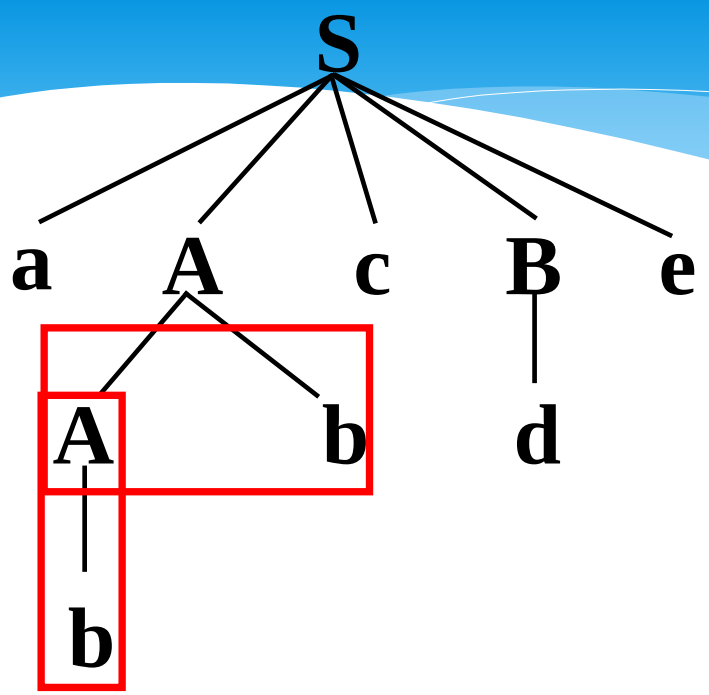
定义：规范句型中不含句柄之后任何符号的一个前缀。（即：从第一个符号开始，到句柄结束，构成了一个符号串，这符号串的前缀叫做该规范句型的活前缀）

例：若$A\rightarrow\alpha\beta$是文法的一个产生式，$\alpha\beta$是句柄，且有

$$S \overset{*}{\Rightarrow} \delta A\omega \Rightarrow \delta\alpha\beta\omega$$

则$\delta\alpha\beta$的任何前缀都是规范句型$\delta\alpha\beta\omega$的活前缀，

包括：$\varepsilon$、$\delta$、$\delta\alpha$、$\delta\alpha\beta$ ……




δ: a

句柄: b

ω: bcde

活前缀

ε

a

ab

活前缀

δ: a

句柄: Ab

ω: cde

ε

a

aA

aAb

(1) 规范句型的活前缀不含句柄后的任何符号。

(2) 活前缀与句柄之间的三种关系：

* 活前缀不含句柄αβ的任何符号
  （此时,期待从剩余输入串中识别由句柄αβ推导出的符号串）
* 活前缀只含句柄αβ的真前缀
  （此时,句柄αβ中α已识别，并出现在栈顶，期待从剩余输入串中识别由β推导出的符号串）
* 活前缀已含句柄αβ的全部符号
  （此时,句柄αβ已全部出现在栈顶，下一步应将αβ归约为A）

# 准备二：项目

项目：在产生式右部添加一个圆点，用于表示活前缀和句柄之间的三种关系：

$$A\rightarrow\bullet\alpha\beta、A\rightarrow\alpha\bullet\beta、A\rightarrow\alpha\beta\bullet$$

项目的分类：

*归约项目：形如$A\rightarrow\alpha\beta\bullet$

*移进项目：形如$A\rightarrow\alpha\bullet\beta$，$\beta=a\cdots$，$a\in V_T$

*待约项目：形如$A\rightarrow\alpha\bullet\beta$，$\beta=B\cdots$，$B\in V_N$

*接受项目：形如$S\rightarrow\alpha\beta\bullet$，S为开始符号。

*如果开始符号有两个产生式，如何表示接受项目？

# 准备三：拓广文法

* 对文法G(S)，增加产生式S′→S，把开始符号重新表示为S′，形成拓广文法G(S’)。
* 在该拓广文法中，项目S′→S•是唯一的接受项目。
* 文法的项目有哪些呢?

# 求拓广文法的项目

有拓广文法G(S’)：S’->S　　S->BB　　B->aB
　　B->b，文法的项目有：

1、S’->•S　　6、S->B•B

2、S->•BB　　7、B->a•B

3、B->•aB　　8、B->b•

4、B->•b　　9、S->BB•

5、S’->S•　　10、B->aB•

# 准备四：识别活前缀的状态转换图

构造状态转换图：

* 一个状态对应一个项目
* 连线表示在当前状态（项目）下，如果从输入串中识别了某个符号，则转换到另一个状态（项目）。
* 状态（项目）S′→ •S作为初态，对应“空”活前缀；
* 状态（规约项目A→αβ•或接受项目S→αβ•）作为终态，能够识别其它活前缀。

# 构造方法

1. 若状态 i 为 $X \to X_1 \dots X_{i-1} \bullet X_i \dots X_n$，
   状态 j 为 $X \to X_1 \dots X_{i-1} X_i \bullet X_{i+1} \dots X_n$，输入为$X_i$
   则从状态 i 画一条标志为 $X_i$ 的有向边到状态 j；

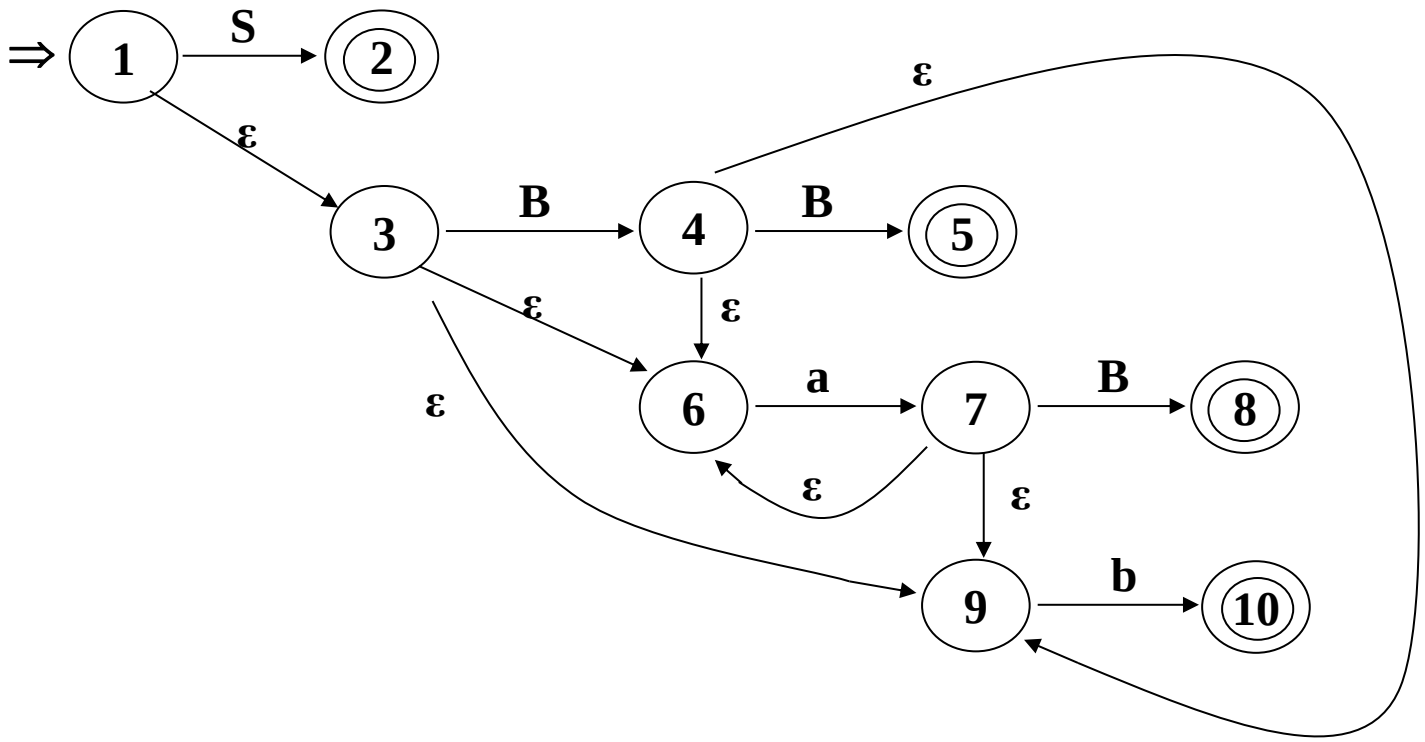
2. 若状态 i 为 $X \to \alpha \bullet A\beta$，A为非终结符，并有产生式 $A \to \alpha_1 | \alpha_2 | \dots | \alpha_n$，则从状态 i 画一条 ε 边到所有状态 $A \to \bullet\alpha_k$。

3. 由于存在ε 边，因此这个状态转换图为NFA。

4. NFA确定化。

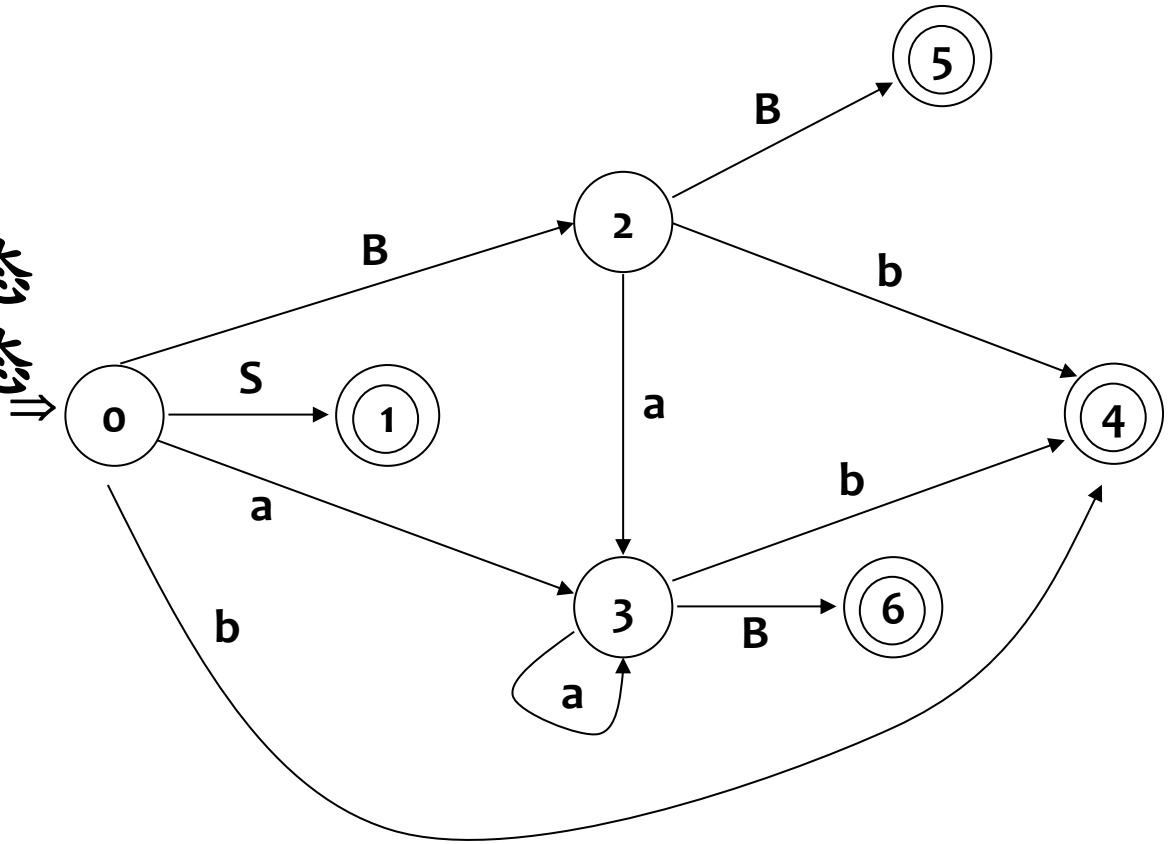
# 有拓广文法G(S')：S'->S　S->BB　B->aB　B->b

* **(1)S'->•S (2)S'->S• (3)S->•BB (4)S->B•B (5) S->BB• (6)B->•aB (7)B->a•B (8)B->aB• (9)B->•b (10)B->b•**



# NFA化简

子集构造法
ε闭包
a弧转换
构造新的状态
删除多余状态
和等价状态



# 项目的有效性定义

1. 对于项目A→α•β, 如果有

$$S \Rightarrow \delta A\omega \Rightarrow \delta\alpha\beta\omega$$

则称项目A→α•β对活前缀δα有效。

即：从初态到达状态A→α•β时，δα已识别出，希望继续从输入串中识别由β推出的串。

# 项目的有效性

2. 若A → α•Bβ对活前缀 δα 有效，且有产生式B→η，则 B → •η 对活前缀 δα 也是有效的。

因为：S’ ⇒δAω⇒δαBβω⇒δαBω’ ⇒δαηω’

即：S’ ⇒δαBω’ ⇒δαηω’

所以，项目B→•η (B→ ε •η)对活前缀 δα (δαε ) 也有效。(A : B; α : ε; β : η; δ : δα)

* 文法G(S′)：S′→S　　S→bB　　B→cB | d
* 判断项目：B → • cB，B → c • B，B → • d 对活前缀 bc的有效性

S’⇒ S ⇒ bB ⇒ bcB

(B→c•B，　δ=b，　δα=bc, β=B)

S’⇒ S ⇒ bB ⇒ bcB ⇒bccB

(B→•cB，　δ=bc, δα=bc, β=cB)

S’⇒ S ⇒ bB ⇒ bcB ⇒ bcd

(B→•d，　δ=bc，δα=bc, β=d)

3. 有效项目集：对活前缀δα有效的项目的集合称为对δα的有效项目集。

4. 有效项目集闭包closure(I)的求法

设I是文法G的一个项目集，通过以下步骤构造有效项目集闭包closure(I)：

① 对 i∈I，都有 i∈closure(I)；

② 若A→α•Bβ ∈ closure(I)，且B→η为文法G的一个产生式，则B→•η ∈ closure(I)；

③ 重复2，直至closure(I)不再增大。

## 5. 有效项目集之间的转换关系：

状态转换函数：go(I, X), X∈V*

如果：A→α•Xβ对δα有效

即：S'⇒δAω⇒δαXβω

所以：A→αX•β对δαX有效

定义：go(I, X) = closure( { A→αX•β | A→α•Xβ∈I} )

## 6. 项目集规范族

**定义：一个文法G的所有有效项目集组成的集合，叫做该文法G的项目集规范族。**

**项目集规范族C的求法：**

```
begin
    C := { closure( {S'→•S} ) };
    repeat
        for（C中每一项目集I和符号X, X∈V*）
         { if（go(I, X) ∉ C）
                then 把go(I, X)加入C中 }
    until C不再增大
end
```

0. S'→S
1. S→BB
2. B→aB
3. B→b

**go(I, X) = closure( { A→αX•β | A→α•Xβ∈I} )**

初始时：C = { $I_0$ }=closure(S '->•S)

={ S'->•S  S->•BB  B->•aB  B->•b  }

go($I_0$,S)={closure(S'->S•)  }={S'->S•} =$I_1$

go($I_0$,B)={closure({S→B•B}) ={S->B•B B->•aB B->•b  }=$I_2$

go($I_0$,a)={closure({B→a•B}) ={B->a•B B->•aB B->•b  }=$I_3$

go($I_0$,b)={closure({B→b•})  ={B->b•} =$I_4$

go($I_1$,S)={} go($I_1$,B)={} go($I_1$,a)={} go($I_1$,b)={} 终结状态

go($I_2$,B)={closure({B→BB•}) ={B->BB•} =$I_5$

go($I_2$,a)={closure({B→a•B}) ={B->a•B B->•aB B->•b  }=$I_3$

go($I_2$,b)={closure({B→b•})  ={B->b•} =$I_4$

**0. S'→S**
**1. S→BB**
**2. B→aB**
**3. B→b**

**go(I, X) = closure( { A→αX•β | A→α•Xβ∈I} )**

go($I_3$,B)={closure({B→aB•}) }={B->aB•}=$I_6$

go($I_3$,a)={closure({B→a•B}) } =$I_3$

go($I_3$,b)={closure({B→b•}) } =$I_4$

go($I_4$,S)={} go($I_4$,B)={} go($I_4$,a)={} go($I_4$,b)={} 终结状态

go($I_5$,S)={} go($I_5$,B={} go($I_5$,a)={} go($I_5$, b)={}) 终结状态

go($I_6$,S)={} go($I_6$,B={} go($I_6$,a)={} go($I_6$, b)={}) 终结状态

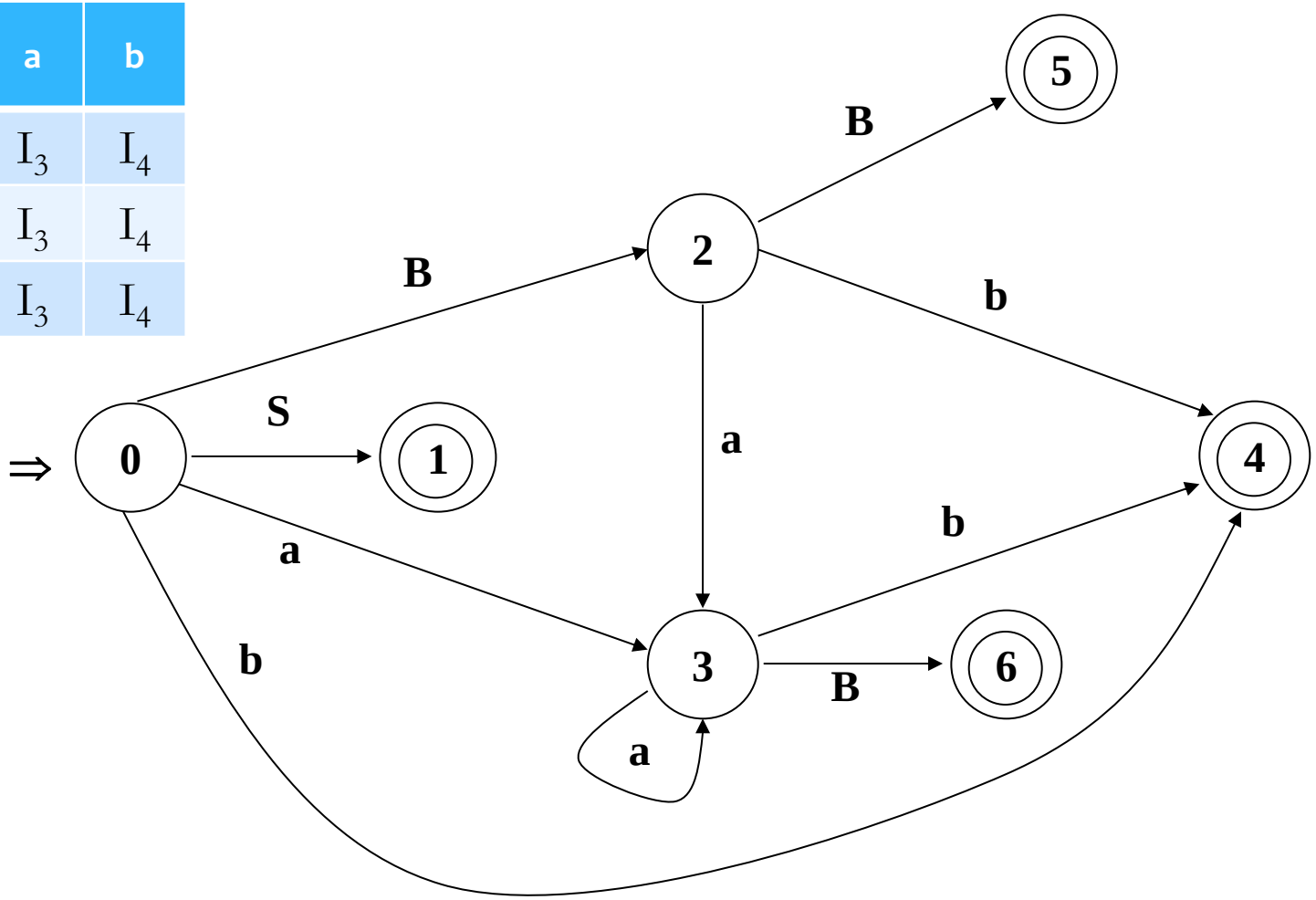
* 结果：$C=\{I_0, I_1, I_2, I_3, I_4, I_5, I_6\}$

$go(I_0,S)=I_1$
$go(I_0,B)=I_2$
$go(I_0,a)=I_3$
$go(I_0,b)=I_4$
$go(I_2,B)=I_5$
$go(I_2,a)=I_3$
$go(I_2,b)=I_4$
$go(I_3,B)=I_6$
$go(I_3,a)=I_3$
$go(I_3,B)=I_4$

| 转换表 | S | B | a | b |
|---|---|---|---|---|
| $I_0$ | $I_1$ | $I_2$ | $I_3$ | $I_4$ |
| $I_1$ | | | | |
| $I_2$ | | $I_5$ | $I_3$ | $I_4$ |
| $I_3$ | | $I_6$ | $I_3$ | $I_4$ |
| $I_4$ | | | | |
| $I_5$ | | | | |
| $I_6$ | | | | |

# 识别活前缀的状态转换图（DFA）

| | S | B | a | b |
|---|---|---|---|---|
| $I_0$ | $I_1$ | $I_2$ | $I_3$ | $I_4$ |
| $I_2$ | | $I_5$ | $I_3$ | $I_4$ |
| $I_3$ | | $I_6$ | $I_3$ | $I_4$ |



# LR(0)分析表的构造

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

$I_0$={S ' ->•S
S->•BB
B->•aB
B->•b}
$I_1$={S ' ->S•}
$I_2$={S->B•B
B->•aB
B->•b}
$I_3$={B->a•B
B->•aB
B->•b}
$I_4$={B->b•}
$I_5$={S->BB•}
$I_6$={B->aB•}

**0. S'→S**
**1. S→BB**
**2. B→aB**
**3. B→b**

- 若$go(I_i, a) = I_j$，则 $action[i, a] = s_j$；
- 若$go(I_i, A) = I_j$，则 $goto[i, A] = j$；
- 若$A→α• \in I_i$，$A→α$为第j个产生式，
- 则对任何终结符a或#，$action[i, a/\#] = r_j$；
- 若$S'→S• \in I_i$，则 $action[i, \#] = acc$。

$go(I_0,S)=I_1$
$go(I_0,B)=I_2$
$go(I_0,a)=I_3$
$go(I_0,b)=I_4$
$go(I_2,B)=I_5$
$go(I_2,a)=I_3$
$go(I_2,b)=I_4$
$go(I_3,B)=I_6$
$go(I_3,a)=I_3$
$go(I_3,b)=I_4$

| 状态 | action | | | goto | |
|---|---|---|---|---|---|
| | a | b | # | S | B |
| 0 | $s_3$ | $s_4$ | | 1 | 2 |
| 1 | $r_0$ | $r_0$ | acc | | |
| 2 | $s_3$ | $s_4$ | | | 5 |
| 3 | $s_3$ | $s_4$ | | | 6 |
| 4 | $r_3$ | $r_3$ | $r_3$ | | |
| 5 | $r_1$ | $r_1$ | $r_1$ | | |
| 6 | $r_2$ | $r_2$ | $r_2$ | | |

# 输入串bab的分析过程

| | 状态栈 | 符号栈 | 输入串 | 动作 |
|---|---|---|---|---|
| 1 | 0 | # | b a b # | s4 4 |
| 2 | 0 | # | b a b # | \|r3\|=1 2 |
| 3 | 0 4 | # B | a b # | s3 3 |
| 4 | 0 2 | # B | a b # | s4 4 |
| 5 | 0 2 3 | # B a | b # | \|r3\|=1 6 |
| 6 | 0 2 3 4 | # B a B | # | \|r2\|=2 5 |
| 7 | 0 2 3 6 | # B B B | # | \|r1\|=2 1 |
| 8 | 0 2 5 | # B B | # | acc |

$I_0$: S′→·S
S→·S+T
S→·T
T→·T*F
T→·F
F→·(S)
F→·i

$I_1$=go($I_0$,S):
S′→S·
S→S·+T

$I_2$=go($I_0$,T)
S→T·
T→T·*F

$I_3$=go($I_0$,F)
T→F·

$I_4$=go($I_0$,()
F→(·S)
S→·S+T
S→·T
T→·T*F
T→·F
F→·(S)
F→·i

$I_5$ =go($I_0$,i)
F→i·

$I_6$=go($I_1$,+)
S→S+·T
T→·T*F
T→·F
F→·(S)
F→·i

$I_7$=go($I_2$,*)
T→T*·F
F→·(S)
F→·i

$I_8$=go($I_4$,S)
F→(S·)
E→E·+T

$I_9$=go($I_6$,T)
S→S+T·
T→T·*F

$I_{10}$=go($I_7$,F)
T→T*F·

$I_{11}$=go($I_8$,))
F→(S)·

(0)S’→S
(1)S→S+T
(2)S→T
(3)T→T*F
(4)T→F
(5)F→(S)
(6)F→i

$I_0$={S' ->•S
S->•S+T
S->•T
T->•T*F
T->•F
F->•(S)
F->•i}

$I_1$={S ' ->S•
S->S • +T}

$I_2$={S ->T•
T->T•*F}

$I_3$={T->F•}

$I_4$={F->(•S)
S->•S+T
S->•T
T->•T*F
T->•F
F->•(S)
F->•i}

$I_5$={F->i•}

$I_6$={S ' ->S+•T
T->•T*F
T->•F
F->•(S)
F->•i}

$I_7$={T->T*•F
F->•(S)
F->•i}

$I_8$={F->(S•)
S->S•+T}

$I_9$={S->S+T•
T->T•*F

$I_{10}$={T->T*F•}

$I_{11}$={F->(S)•}

go($I_0$,S)=$I_1$
go($I_0$,T)=$I_2$
go($I_0$,F)=$I_3$
go($I_0$,()=$I_4$
go($I_0$,i)=$I_5$
go($I_1$,+)=$I_6$
go($I_2$,*)=$I_7$
go($I_4$,S)=$I_8$
go($I_4$,T)=$I_2$
go($I_4$,F)=$I_3$
go($I_4$,()=$I_4$
go($I_4$,i)=$I_5$
go($I_6$,T)=$I_9$
go($I_6$,F)=$I_3$
go($I_6$,()=$I_4$
go($I_6$,i)=$I_5$
go($I_7$,F)=$I_{10}$
go($I_7$,()=$I_4$
go($I_7$,i)=$I_5$
go($I_8$,))=$I_{11}$
go($I_8$,+)=$I_6$
go($I_9$,*)=$I_7$

| | ACTION | | | | | | goto | | |
|---|---|---|---|---|---|---|---|---|---|
| | i | + | * | ( | ) | # | S | T | F |
| 0 | s5 | | | s4 | | | 1 | 2 | 3 |
| 1 | r0 | r0 s6 | r0 | r0 | r0 | acc | | | |
| 2 | r2 | r2 | r2 s7 | r2 | r2 | r2 | | | |
| 3 | r4 | r4 | r4 | r4 | r4 | r4 | | | |
| 4 | s5 | | | s4 | | | 8 | 2 | 3 |
| 5 | r6 | r6 | r6 | r6 | r6 | r6 | | | |
| 6 | s5 | | | s4 | | | | | |
| 7 | s5 | | | s4 | | | | | |
| 8 | | s6 | s7 | | s11 | | | | |
| 9 | r1 | r1 | r1 | r1 | r1 | r1 | | | |
| 10 | r3 | r3 | r3 | r3 | r3 | r3 | | | |
| 11 | r5 | r5 | r5 | r5 | r5 | r5 | | | |

$I_0$={S' ->•S

S->•S+T

$I_2$={S ->T•

I5={F->i•}

F->•(S)

F->•i}

* goto($I_1$,+)=$I_6$ $s_6$ S->S•+T∈$I_1$
* {S'->S•}∈$I_1$ $r_0$

* goto($I_2$,*)=$I_7$ $s_7$ T->T•*F∈$I_2$
* {S->T•}∈ $I_2$ $r_2$

如果项目集中存在以下形式的项目：

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

则会出现“移进-归约”冲突，以及“归约-归约”冲突，冲突原因在于下一个字符的处理存在多种可能。

* 若一个文法G的LR(0)分析表不含多重入口，则该文法G称为LR(0)文法。
* 上述文法不是LR(0)文法!
* LR(0)文法非常少，大多数上下文无关文法都不是LR(0)文法;
* 增加向前“展望”一个符号，则该文法G称为SLR(1)文法。

# 冲突原因分析

* ∀ I={X→α • aβ，A→α•，B →α•}

若采用X->α•aβ进行移进处理，则下一个字符必须为 a

若采用A->α•进行规约处理，则下一个字符必须为 FOLLOW(A)

若采用B->α•进行规约处理，则下一个字符必须为 FOLLOW(B)

因此为了避免“规约-移进”冲突以及“规约-规约冲突”，下一个字符必须满足以下条件：

$\{a\}$ FOLLOW(A) FOLLOW(B)两两不相交

# SLR（Simple LR）分析法

* 如果项目集中存在以下形式的项目：

$$I = \{X \to \alpha \cdot a\beta, \quad A \to \alpha \cdot, \quad B \to \alpha \cdot\}$$

则会出现“移进-规约”冲突，以及“规约-规约”冲突。

* 若{a}、FOLLOW(a)、FOLLOW(B)两两不相交，则可用如下方法解决，该方法称为SLR(1)方法：

* 当前状态是$I = \{X \to \alpha \cdot a\beta, A \to \alpha \cdot, B \to \alpha \cdot\}$,当前输入符号是b。

* 1）若b=a, 则移进输入符号b到符号栈中；
* 2）若$b \in FOLLOW(A)$，则用$A \to \alpha$进行规约；
* 3）若$b \in FOLLOW(B)$，则用$B \to \alpha$进行规约；
* 4）其他情况按出错处理。

# SLR(1)分析表的构造方法

(1) C={ $I_0, I_1, \ldots, I_n$ }，$I_i$对应状态 i，其中，
$I_0$=closure({S'→•S})为唯一初态

(2) 对每个$I_i$，b为向前展望的下一个字符。

- 若 $A \rightarrow \alpha \bullet a\beta \in I_i$，$go(I_i, a) = I_j$，则 action[i, a] = $s_j$；
- 若 $A \rightarrow \alpha \bullet \in I_i$，$A \rightarrow \alpha$ 为第 j 个产生式，对 $\forall b \in$ FOLLOW(A)，action[i, b] = $r_j$；
- 若S'→S• $\in I_i$，则 action[i, #] = acc。

(3) 若$go(I_i, A) = I_j$，则 goto[i, A] = j；

(4) 凡不能用规则(2)登记的表项均为“错误”。

# SLR(1)分析表的构造

拓广文法G(S')

(0)S'→S

(1)S->S+T

(2)S->T

(3)T->T*F

(4)T->F

(5)F->(S)

(6)F->i

FOLLOW(S')={ # }

FOLLOW(S) ={ # + ) }

FOLLOW(T) ={ # + ) * }

FOLLOW(F) ={ # + ) * }

$I_0$={S'->•S
S->•S+T
S->•T
T->•T*F
T->•F
F->•(S)
F->•i}

$I_1$={S'->S•
S->S • +T}

$I_2$={S ->T•
T->T•*F}

$I_3$={T->F•}

$I_4$={F->(•S)
S->•S+T
S->•T
T->•T*F
T->•F
F->•(S)
F->•i}

$I_5$={F->i•}

$I_6$={S'->S+•T
T->•T*F
T->•F
F->•(S)
F->•i}

$I_7$={T->T*•F
F->•(S)
F->•i}

$I_8$={F->(S•)
S->S•+T}

$I_9$={S->S+T•
T->T•*F

$I_{10}$={T->T*F•}

$I_{11}$={F->(S)•}

$go(I_0,S)=I_1$ $go(I_4,i)=I_5$
$go(I_0,T)=I_2$ $go(I_6,T)=I_9$
$go(I_0,F)=I_3$ $go(I_6,F)=I_3$
$go(I_0,()=I_4$ $go(I_6,()=I_4$
$go(I_0,i)=I_5$ $go(I_6,i)=I_5$
$go(I_1,+)=I_6$ $go(I_7,F)=I_{10}$
$go(I_2,*)=I_7$ $go(I_7,()=I_4$
$go(I_4,S)=I_8$ $go(I_7,i)=I_5$
$go(I_4,T)=I_2$ $go(I_8,))=I_{11}$
$go(I_4,F)=I_3$ $go(I_8,+)=I_6$
$go(I_4,()=I_4$ $go(I_9,*)=I_7$

| | ACTION | | | | | | goto | | |
|---|---|---|---|---|---|---|---|---|---|
| | i | + | * | ( | ) | # | S | 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 | | | |

* 上面的分析表不含有多重入口，则表明这是一个SLR(1)分析表。这个文法是一个SLR(1)文法。

# LR(1)分析表的构造

## SLR(1)分析的缺点

SLR(1)分析法能够解决LR(0)分析法中的部分移进规约冲突，但是大多数实用的程序设计语言还是无法满足SLR(1)文法的条件，因此无法使用SLR(1)方法解决项目集规范族中的移进-规约冲突以及规约-规约冲突。

# LR(1)分析表的构造

$I_0$: S′→·S
S→·L=R
S→·R
L→·*R
L→·i
R→·L

$I_1$=GOTO($I_0$,S)
S′→S·

$I_2$=GOTO($I_0$,L)
S→L·=R
R→L·

$I_3$=GOTO($I_0$,R)
S→R·

$I_4$=GOTO($I_0$,*)
L→*·R
L→·*R
L→·i
R→·L

$I_5$=GOTO($I_0$,i)
L→i·

$I_6$=GOTO($I_2$,=)
S→L=·R
L→·*R
L→·i
R→·L

$I_7$=GOTO($I_4$,R)
L→*R·

$I_8$=GOTO($I_4$,L)
R→L·

$I_9$=GOTO($I_6$,R)
S→L+R·

(0)S′→S
(1)S→L=R
(2)S→R
(3)L→*R
(4)L→i
(5)R→L

GOTO($I_0$,S)=$I_1$
GOTO($I_0$,L)=$I_2$
GOTO($I_0$,R)=$I_3$
GOTO($I_0$,*)=$I_4$
GOTO($I_0$,i)=$I_5$
GOTO($I_2$,=)=$I_6$
GOTO($I_4$,R)=$I_7$
GOTO($I_4$,L)=$I_8$
GOTO($I_4$,i)=$I_5$
GOTO($I_6$,R)=$I_9$
GOTO($I_6$,L)=$I_8$
GOTO($I_6$,i)=$I_5$

FOLLOW(S′)={#　　}

FOLLOW(S) ={#　　}

FOLLOW(R) ={#　=　　}

FOLLOW(L) ={#　=　　}

# LR(1)分析表的构造

## SLR(1)分析的缺点

$I_2=GOTO(I_0,L)$

$S \rightarrow L \cdot = R$

$R \rightarrow L \cdot$

I2存在移进-规约冲突，且

$$FOLLOW(R) \cap \{=\} = \{=\} \neq \phi$$

因此文法不是SLR(1)文法，无法使用SLR(1)分析法。

原因在于当下一个输入符号$b \in FOLLOW(R)$时，使用相应产生式进行规约后不一定形成规范句型。

解决办法：当满足一定条件时才进行规约。

# LR(1)分析表的构造

## LR(1)项目

一般形式：[A→α•β，a]

其中： A→α•β是一个LR(0)项目，$a \in V_T$, 称为向前搜索字符。

通过增加一个终结符以明确指出：对于产生式A→αβ，只有当β后跟哪些终结符时才允许将αβ规约为A。

这种方法可以保证规约后分析栈中呈现出下一句型的活前缀。

# LR(1)分析表的构造

## LR(1)项目有效性

一个LR(1)项目 [A→α•Bβ，a] 对活前缀δα是有效的，如果存在规范推导：

$$S \overset{*}{\Rightarrow} \delta A\omega \Longrightarrow \delta\alpha B\beta\omega$$

其中，搜索字符 a = FIRST(ω)；当ω 为ε 时，a 为“#”。

如果另外存在B→γ的产生式，将存在另一个规范推导：

$$S \overset{*}{\Rightarrow} \delta A\omega \Longrightarrow \delta\alpha B\beta\omega \Longrightarrow \delta\alpha\gamma\beta\omega$$

因此，LR(1)项目[B→•γ，b] 对活前缀δα也是有效的，其中b = FIRST(βω)；即 b = FIRST(βa)。

# LR(1)分析表的构造

## LR(1)项目集规范族

与LR(0)项目集规范族相似，同样需要构建Closure(I)以及GOTO(I, X)函数。

构造LR(1)项目集规范族以项目[S′→•S, #]为初始集的初始项目，然后利用Closure函数对其求闭包得到初态项目集，进而应用GOTO函数构建新的项目集，直至项目集规范族不再增大为止。

# LR(1)分析表的构造

## 求LR(1)项目集的闭包Closure(I)的方法

(1) 将I中的所有项目都加入Closure(I)中；

(2) 若项目[A→α•Bβ，a] ∈ Closure(I)，对于任何b ∈ FIRST(βa)，若[B→•γ, b] 不在Closure(I)中，则将[B→•γ, b]加进去。

(3) 重复执行步骤(2)，直到Closure(I)不再增大为止。

# LR(1)分析表的构造

## 求LR(1)项目集的初态项目集合

(0)S′→S
(1)S→BB
(2)B→bB
(3)B→a

I0 = closure([S′→•S, #])

=

{ S′→ • S, #　S→ • BB, # B→ • bB, a/b B→ • a, a/b }

因为FIRST(ε#) = #

因为FIRST(B#) = {a, b}

# LR(1)分析表的构造

## 有效项目集状态转换函数

假设LR(1)项目集I中的项目形如[A→α•Xβ, a]，则项目集I的状态转移函数定义为：

$$GOTO(I, X) = closure(\{[A \rightarrow \alpha X \bullet \beta, a]\})$$

# LR(1)分析表的构造

## 构造LR(1)项目集规范族及识别活前缀的DFA

(1) C = Closure({[S′→•S, #]});

(2) 重复执行动作(3)，直到C不再增大为止；

(3) for (C中的每一个项目集I和G’中每一个符号X)

if (GOTO(I, X) ≠ $\phi$且GOTO(I, X) ∉ C)

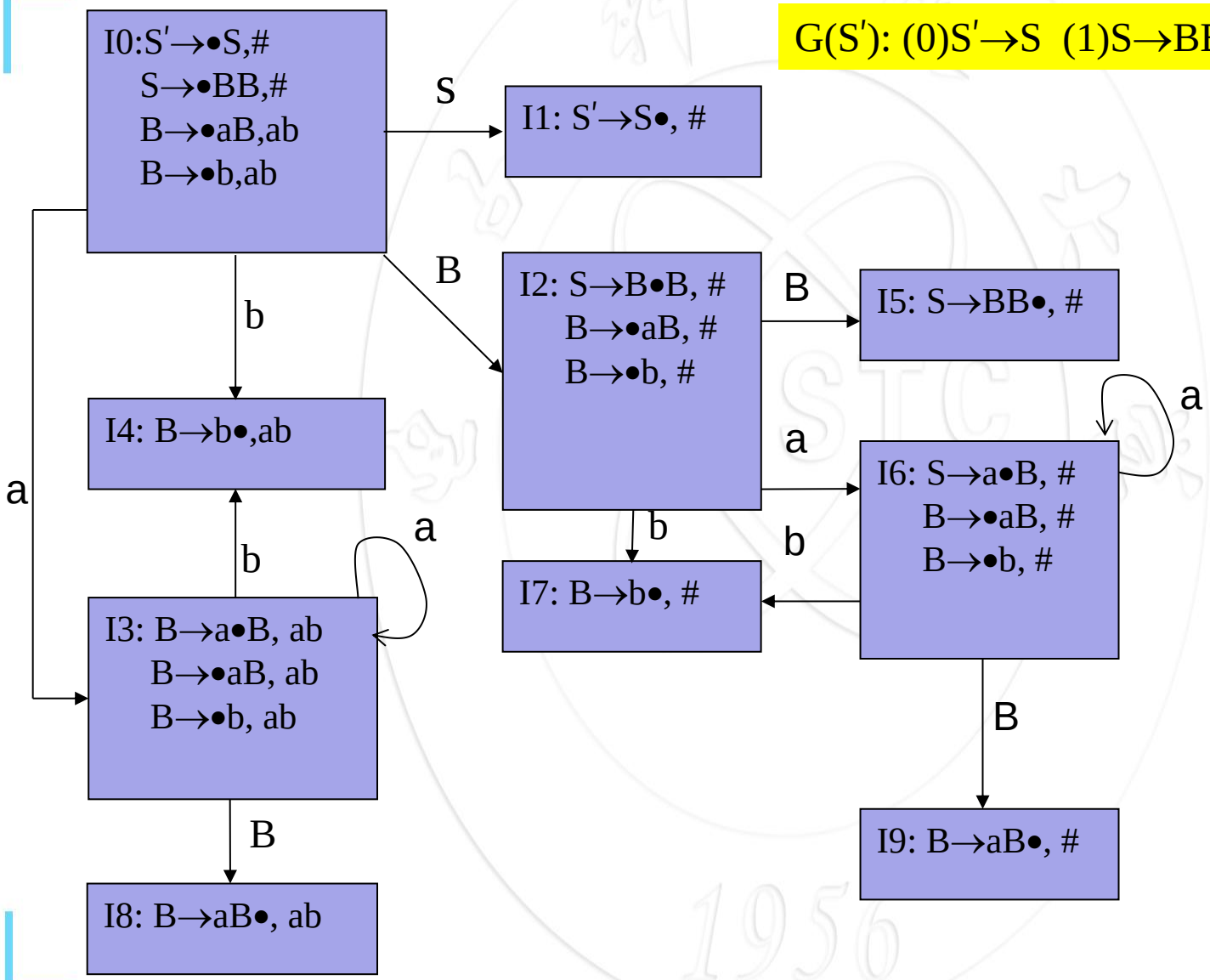
{

将GOTO(I, X)加入C中；

在I和GOTO(I, X)中间添加标记为X的弧线；

}

# LR(1)分析表的构造




G(S′): (0)S′→S (1)S→BB (2)B→aB (3)B→b

GOTO($I_0$,S)=$I_1$
GOTO($I_0$,B)=$I_2$
GOTO($I_0$,a)=$I_3$
GOTO($I_0$,b)=$I_4$
GOTO($I_2$,B)=$I_5$
GOTO($I_2$,a)=$I_6$
GOTO($I_2$,b)=$I_7$
GOTO($I_3$,B)=$I_8$
GOTO($I_3$,a)=$I_3$
GOTO($I_3$,b)=$I_5$
GOTO($I_6$,B)=$I_9$
GOTO($I_6$,a)=$I_6$
GOTO($I_6$,b)=$I_7$

# LR(1)分析表的构造

## 规范的LR(1)分析表的构造

假定LR(1)项目集规范族C={$I_0$, $I_1$,……, $I_n$}，令每个项目集$I_k$的下标k为分析器的一个状态，G'的LR(1)分析表含有状态0，1，……，n。

1. 令含有项目[S'→•S, #]的状态为$I_0$(初态)。ACTION表和GOTO表可按如下方法构造;

2. 若项目[A→α•, b]属于$I_k$，那么置ACTION[k, b]为“用产生式A→α进行规约”，简记为“$r_j$”;其中，假定A→α为文法G'的第j个产生式;

3. 若项目[A→α•Aβ, b]属于$I_k$且GOTO($I_k$, b)= $I_j$, 则置ACTION[k, b]为“把状态j和符号b移进栈”，简记为“$s_j$”;

4. 若项目[S'→•S, #]属于$I_k$，则置ACTION[k, #]为“接受”，简记为“acc”;

5. 若GOTO ($I_k$, A)= $I_j$，A为非终结符，则置GOTO(k, A)=j;

6. 分析表分析中凡不能用规则1至5填入信息的空白格均置上“出错标志”。

# LR(1)分析表的构造

## 规范的LR(1)分析表的构造

按上述算法构造的含有ACTION和GOTO两部分的分析表，如果每个入口不含多重定义，则称它为文法G的一张规范的LR(1)分析表。

具有规范的LR(1)表的文法G称为一个LR(1)文法。

# LR(1)分析表的构造

## LR(1)文法

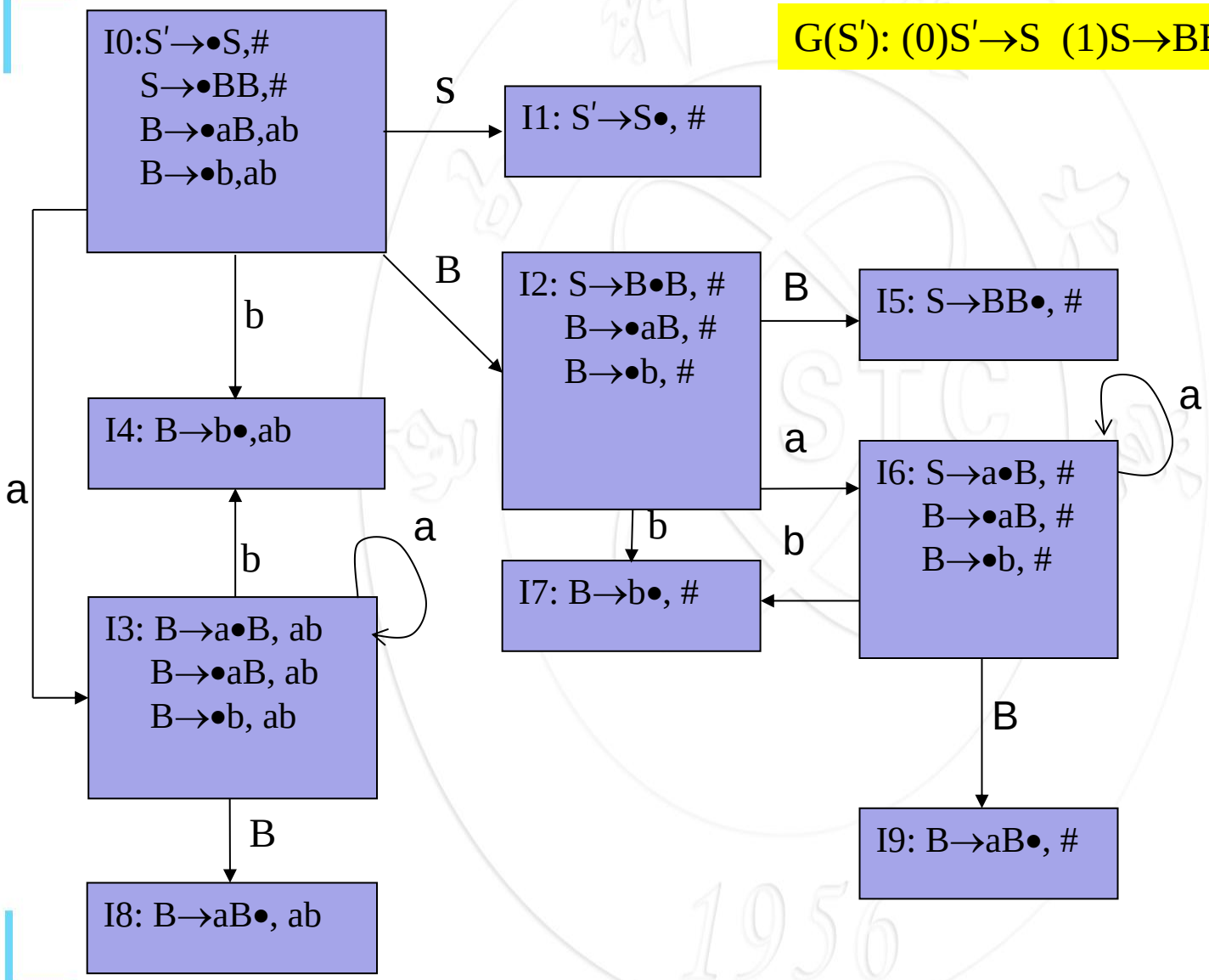
LR(1)文法满足下面两个条件：

1. 如果一个项目集里有项目 [A→u•xv, a]，x是终结符，那就不会有项目[B→u•, x];

2. 项目集里所有归约项目的向前搜索字符不相交，即不能同时含有项目

[A→u•, a] 和[B→v •, a]

# LR(1)分析表的构造




G(S′): (0)S′→S (1)S→BB (2)B→aB (3)B→b

GOTO($I_0$,S)=$I_1$
GOTO($I_0$,B)=$I_2$
GOTO($I_0$,a)=$I_3$
GOTO($I_0$,b)=$I_4$
GOTO($I_2$,B)=$I_5$
GOTO($I_2$,a)=$I_6$
GOTO($I_2$,b)=$I_7$
GOTO($I_3$,B)=$I_8$
GOTO($I_3$,a)=$I_3$
GOTO($I_3$,b)=$I_5$
GOTO($I_6$,B)=$I_9$
GOTO($I_6$,a)=$I_6$
GOTO($I_6$,b)=$I_7$

# LR(1)分析表的构造

| 状态 | ACTION | | | GOTO | |
|---|---|---|---|---|---|
| | a | b | # | S | B |
| 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 | | |

I6: S→a•B, #
B→•aB, #
B→•b, #

GOTO($I_6$,B)=$I_9$
GOTO($I_6$,a)=$I_6$
GOTO($I_6$,b)=$I_7$
GOTO($I_0$,b)=$I_4$

I9: B→aB•, #

G(S′): (0)S′→S
(1)S→BB
(2)B→aB
(3)B→b

# LR(1)分析表的构造

## LR(1)分析与SLR(1)分析对比

(1) LR(1)比SLR(1)的分析能力强!

(2)每个SLR(1)文法都是LR(1)的

(3)一个SLR(1)文法的规范LR分析器比其SLR(1)分析器的状态要多。

# LR(1)分析表的构造

## LR(1)分析与SLR(1)分析对比

G(S′): (0)S′→S
(1)S→BB
(2)B→aB
(3)B→b

**SLR(1)项目集：**

$I_0$={S′→•S　S→•BB　B→•aB　B→•b}
$I_1$={S′→S•}
$I_2$={S→B•B　B→•aB　B→•b}
$I_3$={B→a•B　B→•aB　B→•b}
$I_4$={B→aB•}
$I_5$={B→b•}
$I_6$={S→BB•}

**LR(1)项目集：**

$I_0$={S′→•S, #　S→•BB, #　B→•aB, ab　B→•b, ab}
$I_1$={S′→S•, #}
$I_2$={S→B•B, #　B→•aB, #　B→•b, #}
$I_3$={B→a•B, ab　B→•aB, ab　B→•b, ab}
$I_4$={B→b•, ab}
$I_5$={B→BB•, #}
$I_6$={B→a•B, #　B→•aB, #　B→•b, #}
$I_7$={B→b•, #}
$I_8$={B→aB•, ab}
$I_9$={B→aB•, #}

# LR(1)分析表的构造

## LR(K)分析

有时LR分析需要向前查看K个输入才能够确定冲突解决策略，称为LR(K)分析。LR(K)分析每个项目需要附带有K个终结符，项目的一般形式为：

$$[A \rightarrow \alpha \bullet \beta,\ a_1 a_2 \ldots a_K]$$

其中$A \rightarrow \alpha \bullet \beta$是一个LR(0)项目，$a_i \in V_T$, $a_1 a_2 \ldots a_K$是项目的向前搜索字符串，也称为展望串。

# LALR(1)分析表的构造

LR(1)分析与SLR(1)分析对比

(1) LR(1)比SLR(1)的分析能力强!

(2)每个SLR(1)文法都是LR(1)的

(3)一个SLR(1)文法的规范LR分析器比其SLR(1)分析器的状态要多。

**LR(1)项目集:**

$I_0$={S′→•S, #　S→•BB, #　B→•aB, ab　B→•b, ab}

$I_1$={S′→S•, #}

$I_2$={S→B•B, #　B→•aB, #　B→•b, #}

$I_3$={B→a•B, ab　B→•aB, ab　B→•b, ab}

$I_4$={B→b•, ab}

$I_5$={B→BB•, #}

$I_6$={B→a•B, #　B→•aB, #　B→•b, #}

$I_7$={B→b•, #}

$I_8$={B→aB•, ab}

$I_9$={B→aB•, #}

# LALR(1)分析表的构造

## LALR(1)分析

LR(1) 项目集规范族中，如果存在两个项目集除向前搜索字符不同外，其他部分都是相同的，则称这样的两个LR(1)项目是同心的，相同部分称为同心项目集的心。

如果将同心的LR(1)合并，心仍为相同的一个LR(0)项目集。

# LALR(1)分析表的构造

## 同心集的合并

将LR(1) 的同心项目集的向前搜索字符进行并运算，项目集其他部分不变，合并同心集后的GOTO函数也自动合并。

例：$I_3$={B→a•B, ab　B→•aB, ab　B→•b, ab}

　　$I_6$={B→a•B, #　B→•aB, #　B→•b, #}

合并后：$I_{36}$={B→a•B, ab#　B→•aB, ab#　B→•b, ab#}

若合并LR(1)项目集规范族中的同心集后没有产生新的冲突，则称为LALR(1)项目集。 Lookahead LR

# LALR(1)分析表的构造

## LALR(1)分析

通过构建LALR(1)项目集，然后采用与LR(1)相同的方法构建LALR(1)分析表。

1.构造文法G的规范 LR(1) 状态.

2.合并同心集（除搜索符外两个集合是相同的）的状态.

3.新 LALR(1) 状态的GO函数是合并的同心集状态的GO函数的并.

4. LALR(1)分析表的action 和 goto 构造方法与LR(1)分析表一样.

经上述步骤构造的表若不存在冲突，则称它为G的LALR(1)分析表。

存在这种分析表的文法称为LALR(1)文法。

# LALR(1)分析表的构造

文法G(S′)的LR(1)分析表

G(S′): (0)S′→S
(1)S→BB
(2)B→aB
(3)B→b

| 状态 | ACTION | | | GOTO | |
|---|---|---|---|---|---|
| | a | b | # | S | B |
| 0 | S3 | S4 | | 1 | 2 |
| 1 | | | acc | | |
| 2 | S6 | S7 | | | 5 |
| 3 | S3 | S4 | | | 8 |
| 4 | r3 | r3 | | | |
| 5 | | | r1 | | |
| 6 | S6 | S7 | | | 9 |
| 7 | | | r3 | | |
| 8 | r2 | r2 | | | |
| 9 | | | r2 | | |

# LALR(1)分析表的构造

LALR(1)分析表

合并同心集

S3+S6

S4+S7

S8+S9

| 状态 | ACTION | | | GOTO | |
|---|---|---|---|---|---|
| | a | b | # | S | B |
| 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 | | |

G(S′): (0)S′→S
(1)S→BB
(2)B→aB
(3)B→b

# LALR(1)分析表的构造

| 状态 | ACTION | | | GOTO | |
|---|---|---|---|---|---|
| | a | b | # | S | B |
| 0 | S3 | S4 | | 1 | 2 |
| 1 | | | acc | | |
| 2 | S6 | S7 | | | 5 |
| 3 | S3 | S4 | | | 8 |
| 4 | r3 | r3 | | | |
| 5 | | | r1 | | |
| 6 | S6 | S7 | | | 9 |
| 7 | | | r3 | | |
| 8 | r2 | r2 | | | |
| 9 | | | r2 | | |

| 状态 | ACTION | | | GOTO | |
|---|---|---|---|---|---|
| | a | b | # | S | B |
| 0 | S3,6 | S4,7 | | 1 | 2 |
| 1 | | | acc | | |
| 2 | S3,6 | S4,7 | | | 5 |
| 3,6 | S3,6 | S4,7 | | | 8,9 |
| 4,7 | r3 | r3 | r3 | | |
| 5 | | | r1 | | |
| 8,9 | r2 | r2 | r2 | | |

# LALR(1)分析法

LALR的功能和代价介于SLR和规范LR之间，可用于大多数程序设计语言的文法，并可高效地实现。

对同一个文法来说，LALR分析表和LR(0)、SLR分析表具有相同数目的状态。

LALR分析表比LR(1)分析表小得多，分析能力也若一些，但能够应用于一些SLR(1)不能应用的情况。

# 常用LR分析方法


* LR(0)
* SLR(1)
* LALR(1): Look-Ahead LR
* LR(1)
* 分析能力：
  - LR(1) > LALR(1) > SLR(1) > LR(0)

# LR分析中的错误处理

## LR分析中的错误处理方法

在LR分析过程中，当我们处在这样一种状态下，即输入符号既不能移入栈顶，栈内元素又不能归约时，就意味着发现语法错误。发现错误后，便进入相应的出错处理子程序。处理的方法分为两类：

第一类多半使用插入、删除或修改的办法。如在语句a[1,23.14；中插入一个]。

第二类处理办法包括在检查到某一不合适的短语时，它不能与任一非终结符可能推导出的符号串相匹配。

可以构建包含出错处理子程序的LR分析表.

# LR分析中的错误处理

## 包含出错处理子程序的分析表

| 状态 | ACTION | | | | | | GOTO |
|---|---|---|---|---|---|---|---|
| | i | + | * | ( | ) | # | E |
| 0 | $S_3$ | $e_1$ | $e_1$ | $S_2$ | $e_2$ | $e_1$ | 1 |
| 1 | $e_3$ | $S_4$ | $S_5$ | $e_3$ | $e_2$ | acc | |
| 2 | $S_3$ | $e_1$ | $e_1$ | $S_2$ | $e_2$ | $e_1$ | 6 |
| 3 | $r_4$ | $r_4$ | $r_4$ | $r_4$ | $r_4$ | $r_4$ | |
| 4 | $S_3$ | $e_1$ | $e_1$ | $S_2$ | $e_2$ | $e_1$ | 7 |
| 5 | $S_3$ | $e_1$ | $e_1$ | $S_2$ | $e_2$ | $e_1$ | 8 |
| 6 | $e_3$ | $S_4$ | $S_5$ | $e_3$ | $S_9$ | $S_4$ | |
| 7 | $r_1$ | $r_1$ | $S_5$ | $r_1$ | $r_1$ | $r_1$ | |
| 8 | $r_2$ | $r_2$ | $r_2$ | $r_2$ | $r_2$ | $r_2$ | |
| 9 | $r_3$ | $r_3$ | $r_3$ | $r_3$ | $r_3$ | $r_3$ | |

E→E+E
E→E*E
E→(E)
E→i

# LR分析中的错误处理

## 包含出错处理子程序的分析表

它能识别二义文法所定义的语言。表中某些状态(如状态8,9等)遇到某些输入符号就进行特定的某种归约(如状态8为$r_2$，状态9为$r_3$)，这些状态遇到不合法的输入符号时，本应转向对应的出错处理子程序，而现在我们也把它们进行相同的归约，这样就缩减了分解表所占的空间。当然，如果有错，虽然先进行了某些归约，但在移入下一输入符号以前，错误终将被发现，只是发现的时间推迟了。

# LR分析中的错误处理

包含出错处理子程序的分析表

e1: /*处在状态0,2,4,5时，要求输入符号为一运算量的首符，如i或左括号。当遇到‘+’、‘*’或‘#’等，调用此程序*/

将一假i置于栈内，上盖以状态3；

给出错误信息：“缺少运算量”。

e2: /*当处在状态0,1,2,4,5而遇到右括号时，调用此程序*/

将下一输入符号(右括号)删除；

给出错误信息：“右括号不匹配”。

# LR分析中的错误处理

包含出错处理子程序的分析表

e3: /*当处在状态1或6时，要求输入符号为运算符，但当遇到i或左括号时，调用此程序 */

将‘+’纳入栈顶，上盖以状态4;

给出错误信息：“缺少运算符”。

e4:/* 当处在状态6时，要求输入符号为运算符或右括号，但此时遇到#，调用此程序 */

将‘)’纳入栈顶，上盖以状态9;

给出错误信息：“缺少右括号”。

# LR分析中的错误处理

| | 分析栈 | 符号栈 | 剩余输入串 | ACTION[7, #]=r1 |
|---|---|---|---|---|
| 1 | 0 | # | i+)# | S3 |
| 2 | 03 | # i | +)# | R4 E→i |
| 3 | 01 | # E | +)# | S4 |
| 4 | 014 | # E+ | )# | e2 /*)被e2子程序删除*/ |
| 5 | 014 | # E+ | # | e1 /*e1子程序将i压入栈中*/ |
| 6 | 0143 | # E+i | # | R4 E→i |
| 7 | 0147 | # E+E | # | R1 E→E+E |
| 8 | 01 | # E | # | ACTION(1,#) = acc |

# 本章小结

- 文法和语法
  - 定义、上下文无关文法、推导和规约、语法树
- 自上而下的语法分析
  - 回溯 -- 左递归、公共左因子、ε
  - FIRST、FOLLOW集合、LL(1)文法
  - 递归下降分析法
  - 预测分析法
- 自下而上的语法分析
  - 规范推导、规范规约
  - 短语、直接短语、句柄
  - LR(0)、SLR(1)