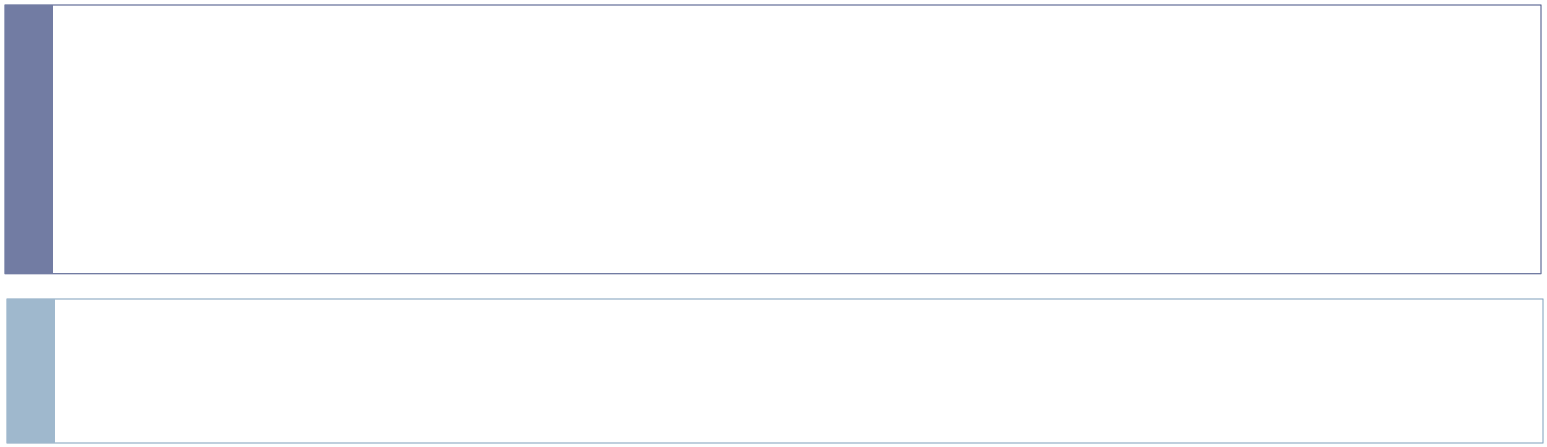
Chapter 5 语法分析-自下而上分析



Outlines

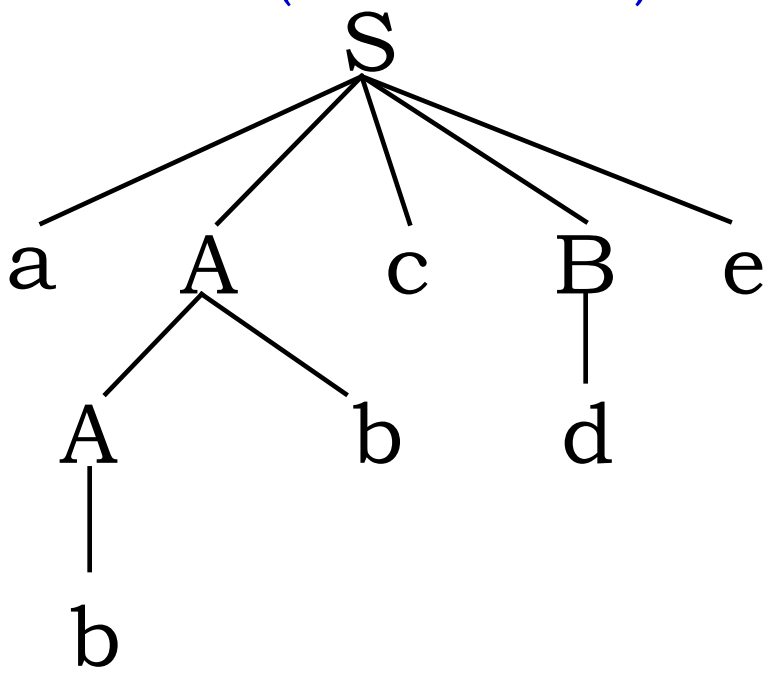
- 自下而上分析基本问题
  - 归约
  - 规范归约
  - 符号栈的使用
  - 语法树的表示
- 算符优先分析法
- LR分析法

Bottom-up

- 自下而上分析
  - 从输入字符的角度而言
    - 从输入开始
    - 逐步进行“归约”
    - 直至归约到文法的开始符号
  - 从语法树的角度而言
    - 从语法树的末端开始
    - 步步向上“归约”
    - 直到根结

例1 abbcde

aAbcde (A → b)
aAcde (A → Ab)
aAcBe (B → d)
S (S → aAcBe)



Bottom-up

- 自下而上分析法
  - 是一种“移进-归约”（shift-reduce）法
  - 基本思想
    - 用一个寄存符号的先进后出栈
    - 把输入符号一个一个地移进到栈里
    - 当栈顶形成某个产生式的候选式时，把栈顶的这一部分替换成(归约为)该产生式的左部符号

Example

- 例2：设文法G[S]：

$$S \rightarrow aABe$$

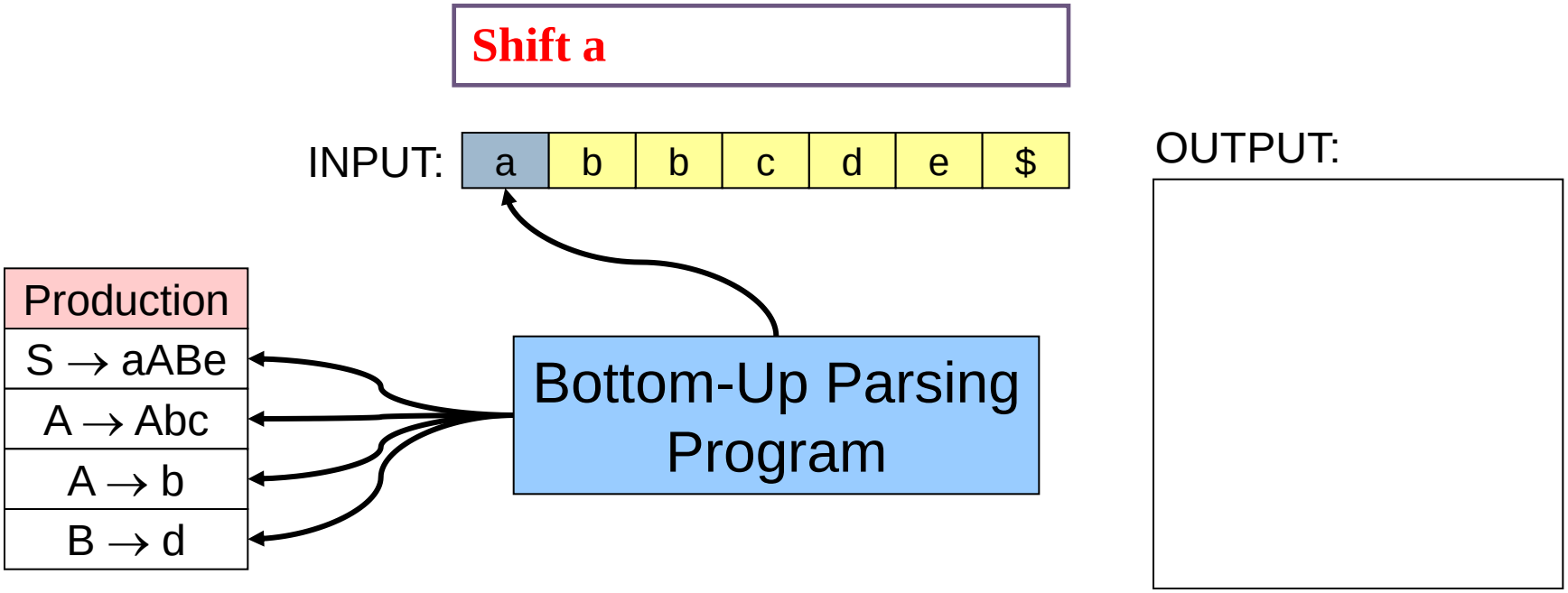
$$A \rightarrow Abc \mid b$$

$$B \rightarrow d$$

试对abbcde进行“移进-归约”分析。

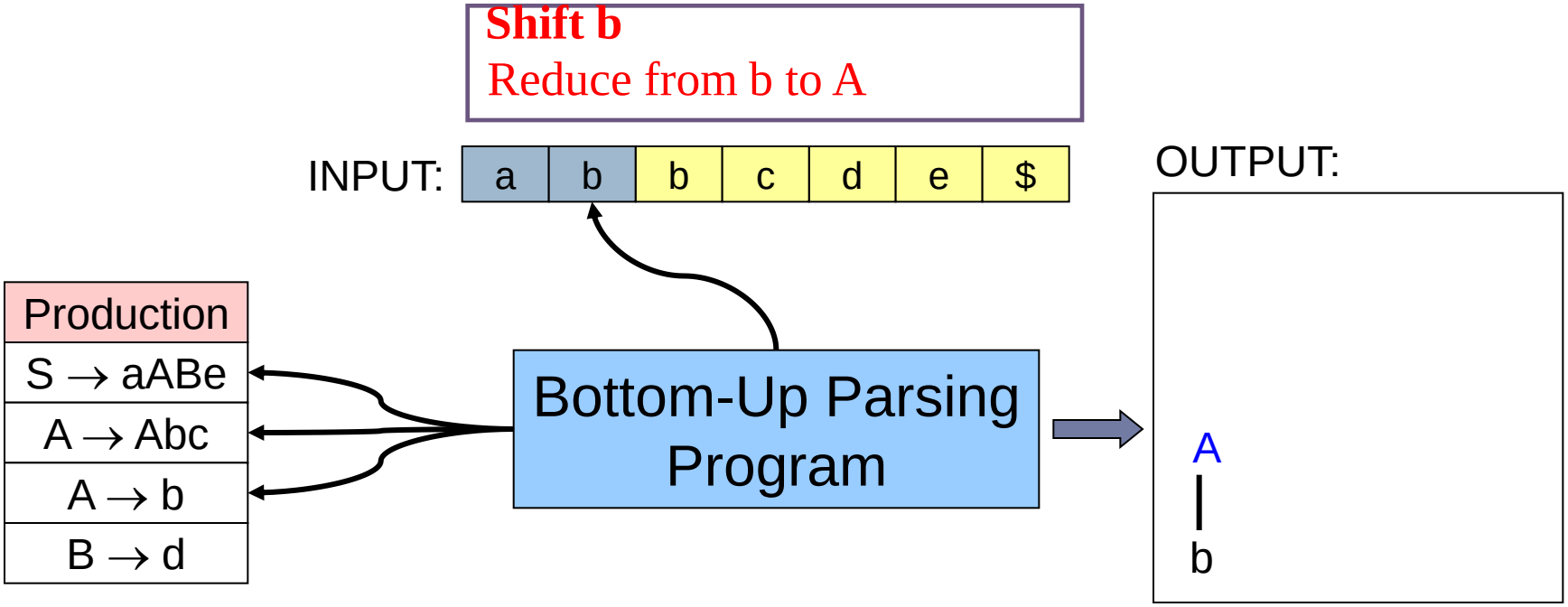
Example

Bottom-Up Parser Example



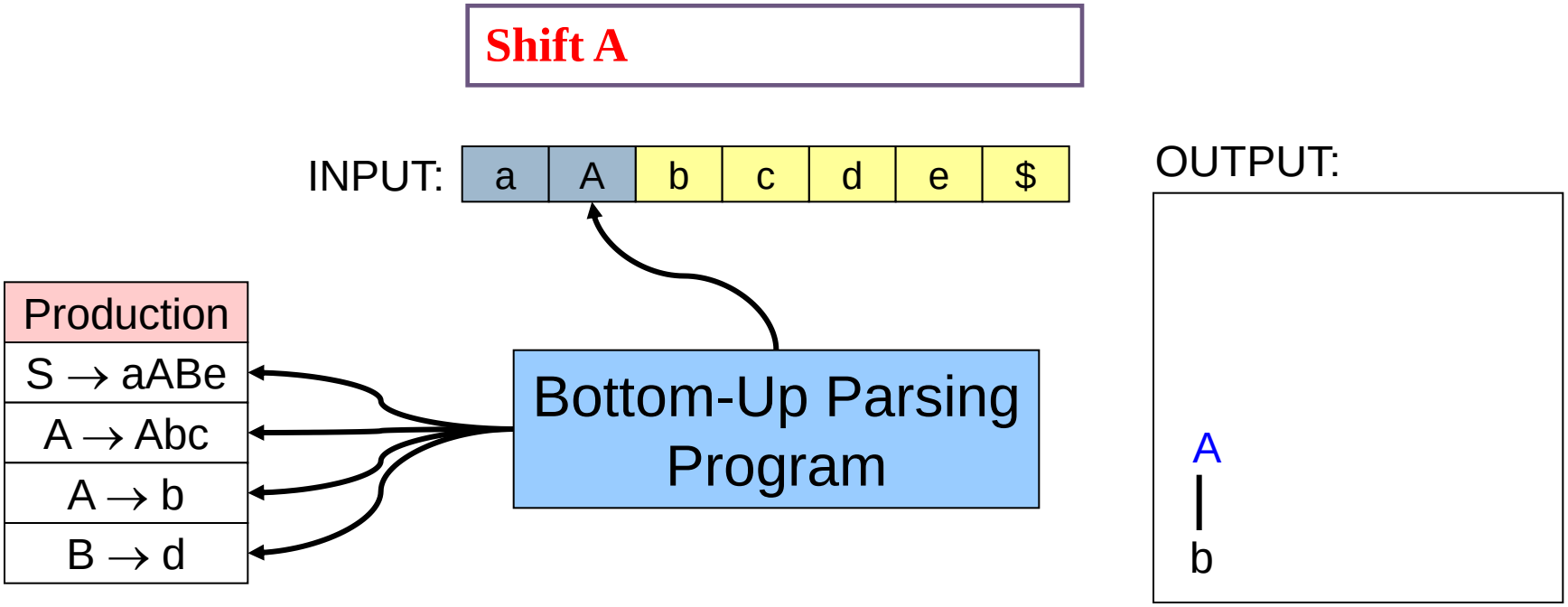
Example

Bottom-Up Parser Example



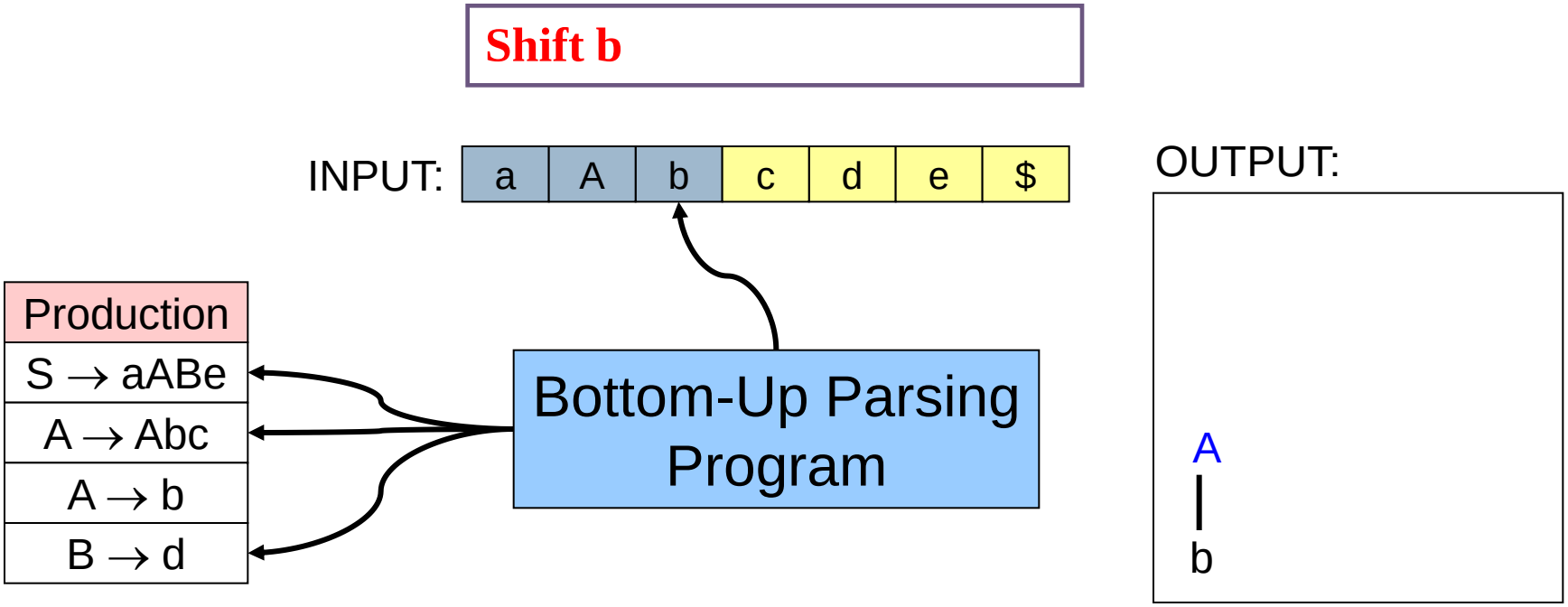
Example

Bottom-Up Parser Example



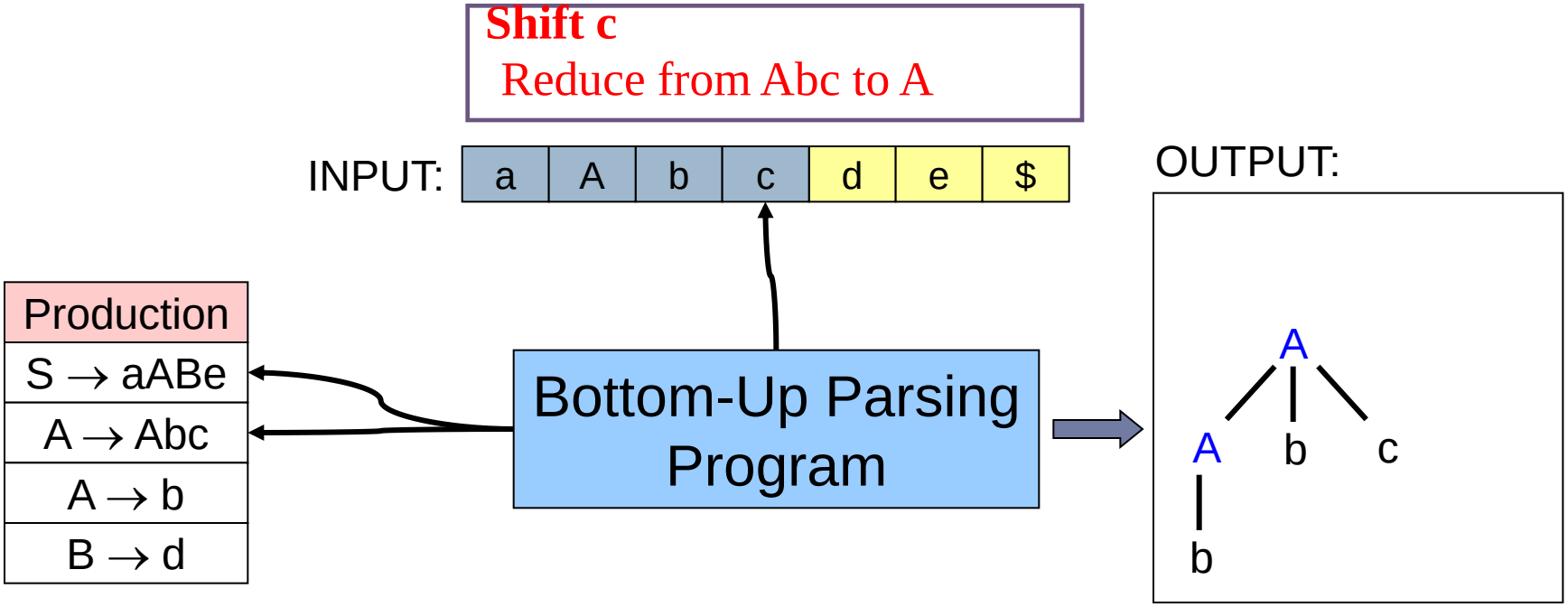
Example

Bottom-Up Parser Example



Example

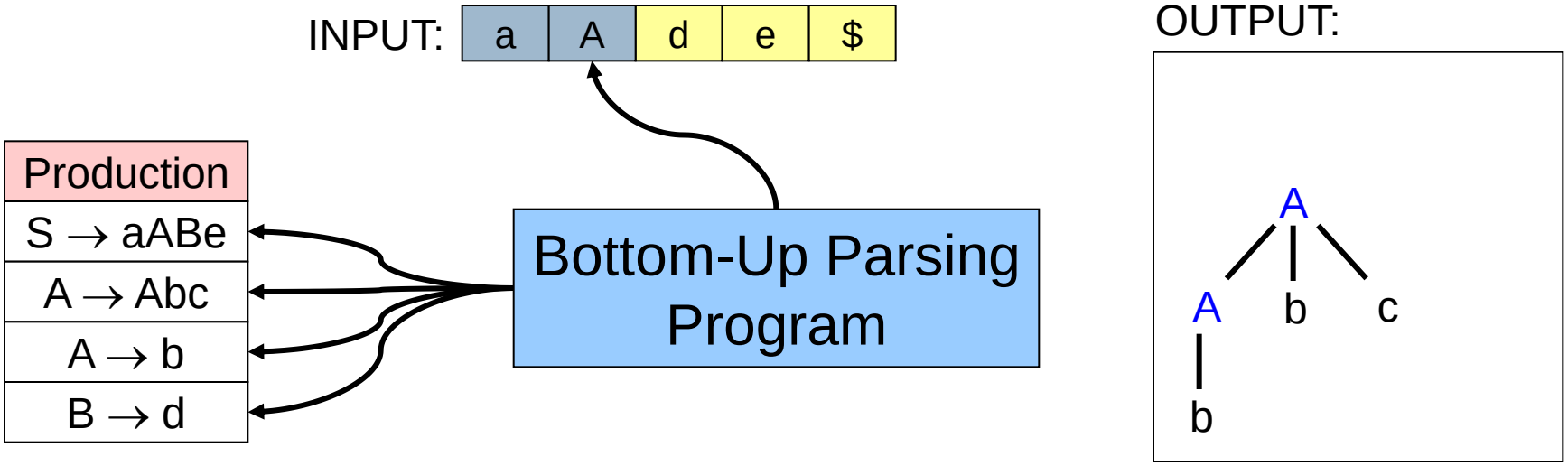
Bottom-Up Parser Example



Example

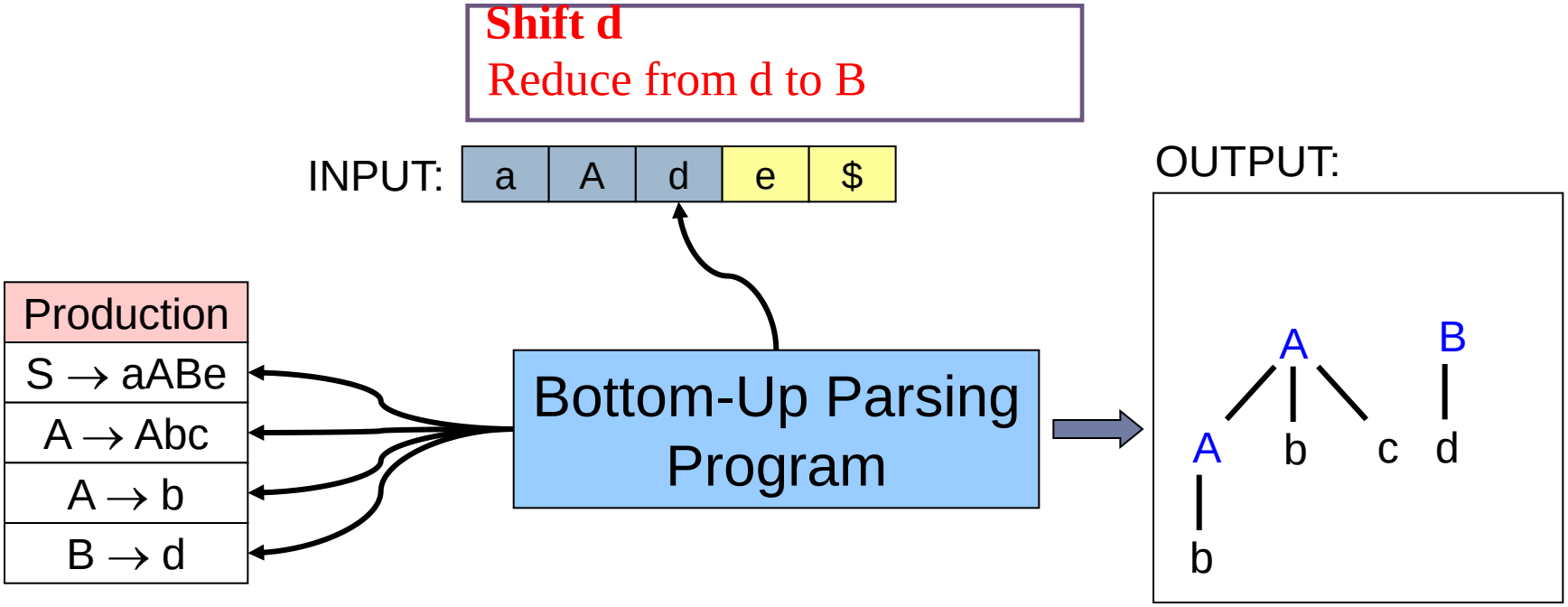
Bottom-Up Parser Example

Shift A



Example

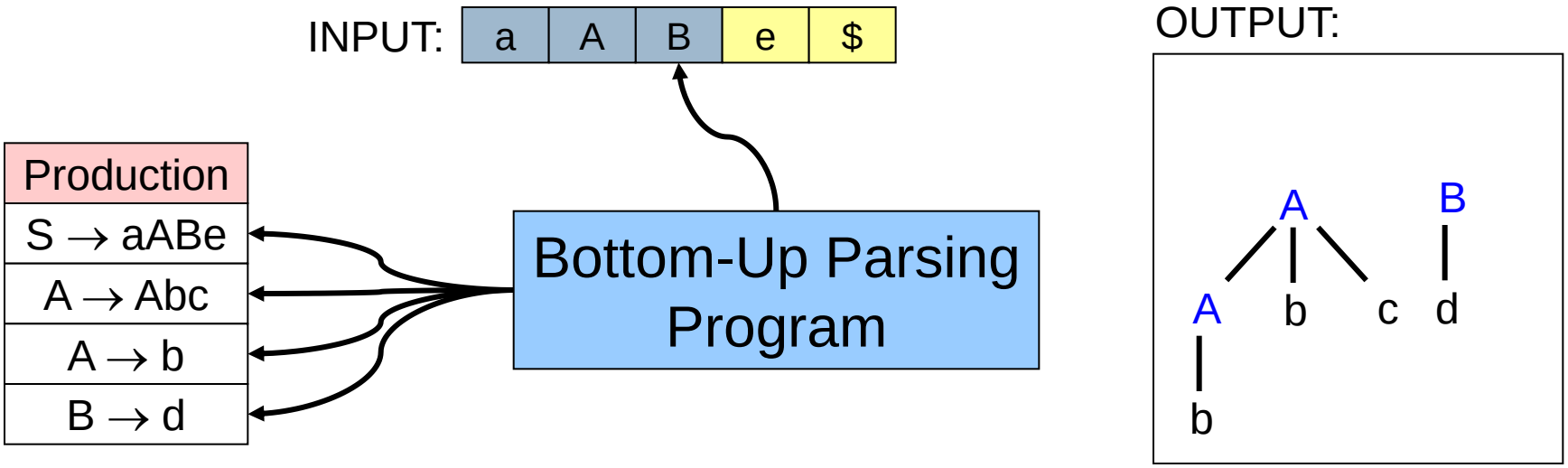
Bottom-Up Parser Example



Example

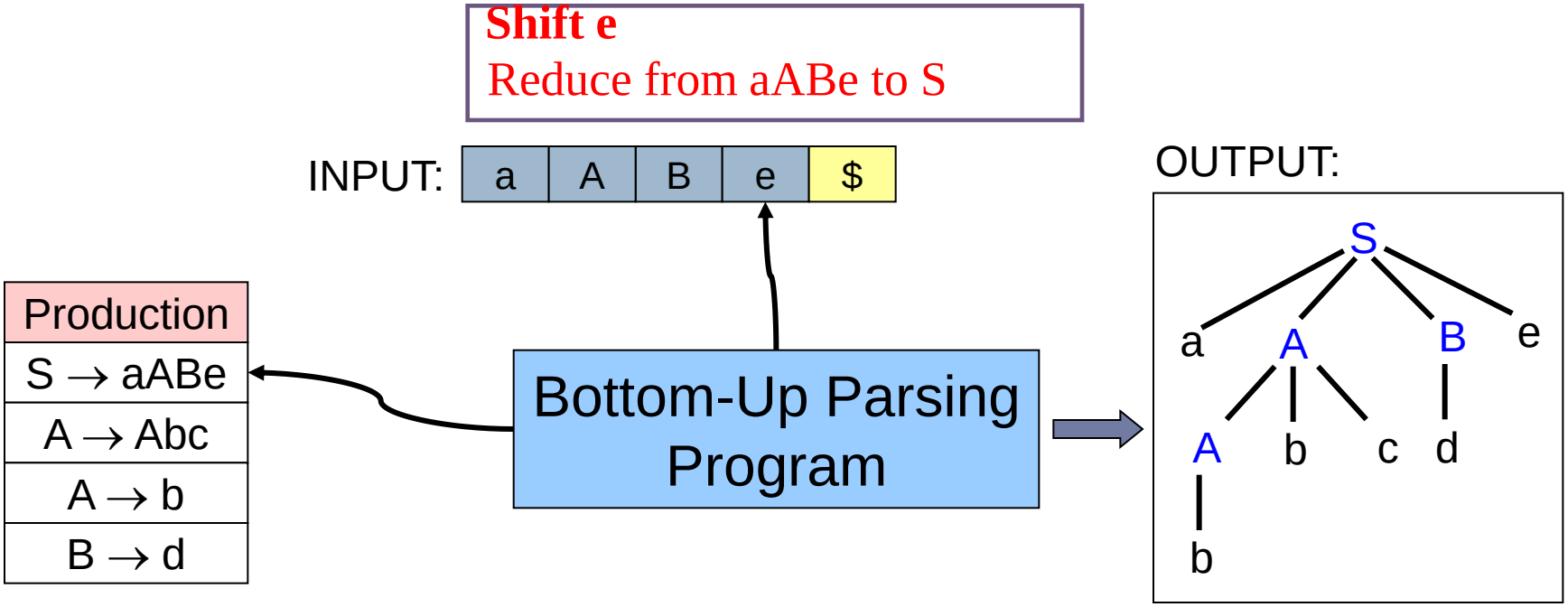
Bottom-Up Parser Example

Shift B



Example

Bottom-Up Parser Example



Example

Bottom-Up Parser Example

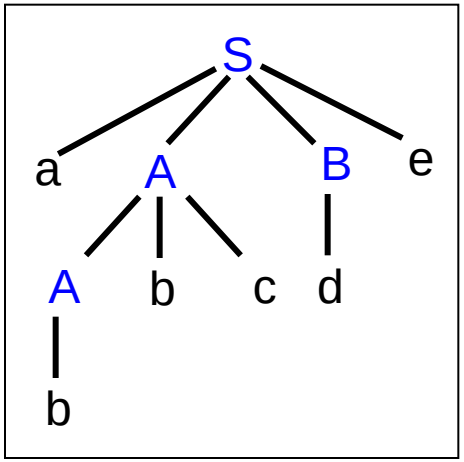
Shift S
Hit the target $

INPUT: S $

Production
S → aABe
A → Abc
A → b
B → d

Bottom-Up Parsing Program

OUTPUT:



Accept!

Example

- 例3：设文法G[S]：

$$S \rightarrow aAcBe$$

$$A \rightarrow Ab$$

$$A \rightarrow b$$

$$B \rightarrow d$$

试对abbcde进行“移进-归约”分析。

Example

$S \rightarrow aAcBe$

$A \rightarrow Ab$

$A \rightarrow b$

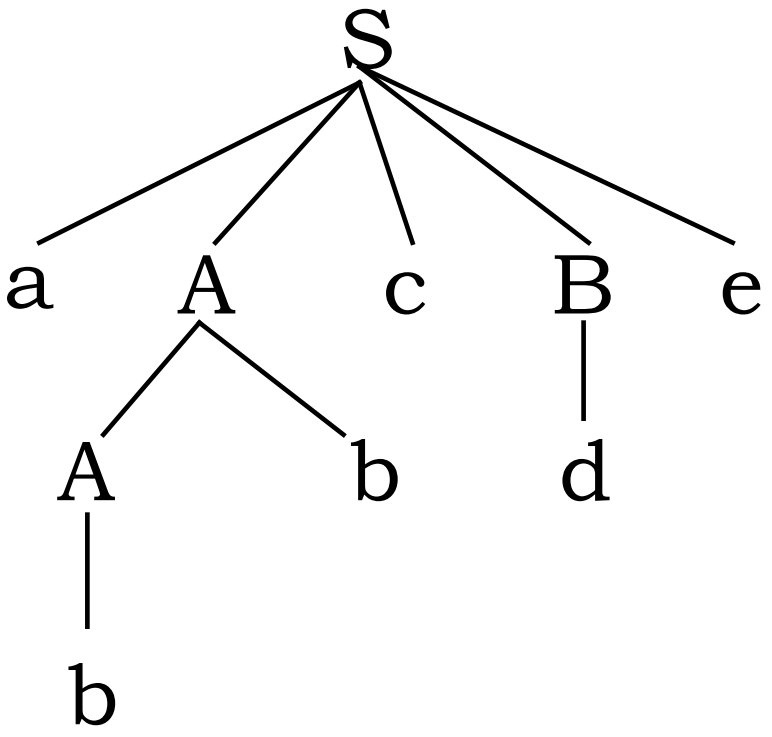
$B \rightarrow d$

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

Shift-reduce

- 语法分析对符号栈的使用有四类操作
  - 移进
  - 归约
  - 接受
  - 出错处理

Parse tree



- 自下而上分析过程
  - 边输入单词符号，边归约
- 核心问题
  - 识别可归约串
  - 常用方法
    - 算符优先分析法
      - 不是一种规范归约法
    - LR分析法
      - 一种规范归约法

Shift-reduce

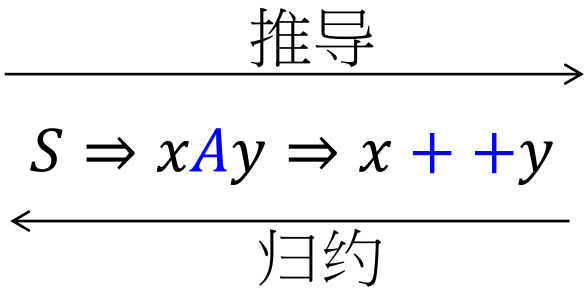
- 定义：令G是一个文法，S是文法的开始符号，假定αβδ是文法G的一个句型，如果有

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

- 则β称是句型αβδ相对于非终结符A的短语。
- 特别是，如果有A⇒β,则称β是句型αβδ相对于规则A→β的**直接短语**。一个句型的最左直接短语称为该句型的**句柄**。

Shift-reduce

- S→xAy, A→+|++
- x++y 应该归约成xA+y还是xAy？

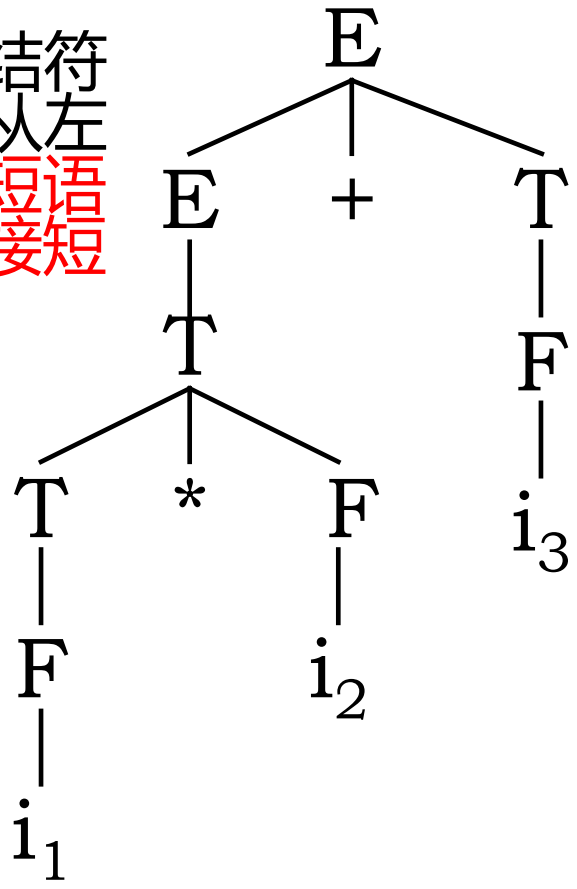


$$S \overset{*}{\Rightarrow} \alpha A \delta \overset{+}{\Rightarrow} \alpha \beta \delta$$

归约之后为一个句型

Shift-reduce

- 例4：考虑文法G[E]：E→T|E+T
  T→F|T*F　F→(E)|i和句型$i_1*i_2+i_3$：
- 在一个句型对应的语法树中，以某非终结符为根的两代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，如果子树只有两代，则该短语就是直接短语。
  - $E \Rightarrow E+T \Rightarrow E+F \Rightarrow E+i_3 \Rightarrow T+i_3 \Rightarrow T*F+i_3 \Rightarrow T*i_2+i_3 \Rightarrow F*i_2+i_3 \Rightarrow i_1*i_2+i_3$
  - 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



Shift-reduce

- 例：考虑文法G[E]：
- E→T|E+T
- T→F|T*F
- F→(E)|i

- 和句型1）E+T*F+i
- 2）E+T*F

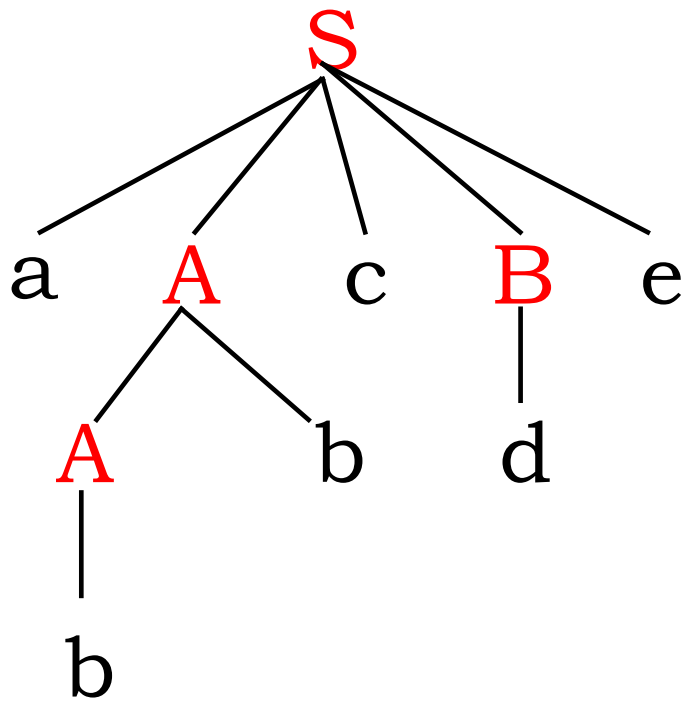
Shift-reduce

- 可用句柄来对句子进行归约
- 例5：设文法G[S]：

  (1) S → aAcBe　(2) A → b

  (3) A → Ab　　(4) B → d

句型	归约规则
abbcde	(2) A → b
aAbcde	(3) A → Ab
aAcde	(4) B → d
aAcBe	(1) S → aAcBe
S	



Shift-reduce

- 定义：假定α是文法G的一个句子，我们称序列

  $\alpha_n, \alpha_{n-1}, \cdots, \alpha_0$

  是一个规范归约，如果此序列满足：

  - 1 $\alpha_n = \alpha$
  - 2 α_0为文法的开始符号，即α_0=S
  - 3 对任何i，$0 \le i \le n$， α_{i-1}是从α_i经把句柄替换成为相应产生式左部符号而得到的。

Shift-reduce

- 把上例倒过来写，则得到：
  - S ⇒ aAcBe⇒ aAcde ⇒ aAbcde ⇒ abbcde
  - 显然这是一个最右推导
- 规范归约
  - 也称最左归约
  - 是关于α的一个最右推导的逆过程
  - 规范归约的中心问题
    - 确定句型的句柄
- 最右推导
  - 在形式语言中，常被称为规范推导
- 规范句型
  - 由最右推导（规范推导）推出的句型

Shift-reduce

- 从分析树上直观地看，“剪句柄”的方法十分简单。但是若在语法分析器中实现剪句柄，则有两个问题必须解决：
  - ① 确定右句型中将要归约的子串(确定句柄)；
  - ② 确定如何选择正确的产生式进行归约。
- 具体实现采用**移进—归约**方法，用一个栈“记住”将要归约句柄的前缀，并用一个分析表来确定何时栈顶已形成句柄，以及形成句柄后选择哪个产生式进行归约。

Shift-reduce

- 在**移进—归约分析模式**中，符号栈的使用有以下四种操作形式。
  - ① 移进(shift)：把当前输入中的下一个终结符移进栈；
  - ② 归约(reduce)：句柄在栈顶已形成，用适当产生式左部代替句柄；
  - ③ 接受(accept)：宣告分析成功；
  - ④ 报错(error)：发现语法错误，调用错误恢复例程。

Shift-reduce

例6： G(S): S → aABe, A → Abc | b, B → d, 输入序列**abbcde**

步骤	栈内容	当前输入	动作
(0)	#	abbcde#	shift
(1)	#a	bbcde#	shift
(2)	#ab	bcde#	reduced by A→b
(3)	#aA	bcde#	shift
(4)	#aAb	cde#	shift
(5)	#aAbc	de#	reduced by A→Abc
(6)	#aA	de#	shift
(7)	#aAd	e#	reduced by B→d
(8)	#aAB	e#	shift
(9)	#aABe	#	reduced by S→aABe
(10)	#S	#	accept

Operator precedence analysis

- 考虑二义文法文法G(E)：

$$E \rightarrow i \mid E+E \mid E-E \mid E*E \mid E/E \mid (E)$$

  - 它的句子有几种不同的规范归约
- 归约
  - 即计算表达式的值
  - 归约顺序不同，则计算的顺序不同，结果也不同
  - 起决定作用的是相邻的两个算符之间的优先关系
    - 如果规定算符的**优先次序**，并按这种规定进行归约，则归约过程是唯一的

Operator precedence analysis

- 算符优先分析法
  - 就是定义算符之间的某种优先关系，借助于这种关系寻找“可归约串”和进行归约
  - 是一种自下而上的归约过程
  - 特别有利于表达式分析，宜于手工实现

  - **不是一种规范归约法**
    - 这种归约未必是严格的最左归约

Operator precedence analysis

- 定义任何两个可能相继出现的终结符a与b的优先关系：

三种关系

$a \lessdot b$　a的优先级低于b

$a \doteq b$　a的优先级等于b

$a \gtrdot b$　a的优先级高于b

- 注意：与数学上的<、>、=不同，$a \lessdot b$并不意味着$b \gtrdot a$

Operator precedence analysis

- 一个文法，如果它的任一产生式的右部都不含两个相继(并列)的非终结符，即不含如下形式的产生式右部：

  …QR…

  则我们称该文法为算符文法。
- 约定：
  - a、b代表任意终结符；
  - P、Q、R代表任意非终结符；
  - ‘…’代表由终结符和非终结符组成的任意序列，包括空字。

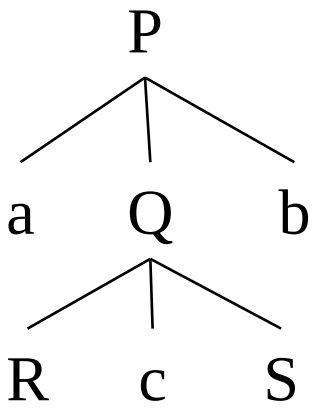
Operator precedence analysis

假定G不含ε-产生式，ab为终结符

$a \doteq b \Leftrightarrow P \to \cdots ab \cdots$ 或 $P \to \cdots aQb \cdots$

$a \lessdot b \Leftrightarrow P \to \cdots aR \cdots$ 且 $(R \overset{+}{\Rightarrow} b \cdots$ 或 $R \overset{+}{\Rightarrow} Qb \cdots)$

$a \gtrdot b \Leftrightarrow P \to \cdots Rb \cdots$ 且 $(R \overset{+}{\Rightarrow} \cdots a$ 或 $R \overset{+}{\Rightarrow} \cdots aQ)$



$a \doteq b$

$a \lessdot c$

$c \gtrdot b$

Operator precedence analysis

如果一个算符文法G中的任何终结符对(a，b)至多只满足下述三关系之一：

$$a \doteq b，\ a \lessdot b，\ a \gtrdot b$$

则称G是一个**算符优先文法**。

Operator precedence analysis

- 例7:考虑下面的文法G[E]:

  (1) E→E+T | T　　(2) T→T*F | F

  (3) F→P↑F | P　　(4) P→(E) | i

  - 由第(4)条规则，有 (≒)；
  - 由规则E→E＋T和T⇒T*F，有 ＋ ⋖ *；
  - 由(2)和(3)，可得* ⋖ ↑；
  - 由(1)E→E＋T和E ⇒ E+T，可得+ ⋗ +；
  - 由(3)F→P↑F和F ⇒ P↑F，可得↑ ⋖ ↑。
  - 由(4)P→(E)和 E⇒E+T⇒T+T⇒T*F+T⇒F*F+T
    ⇒P↑F*F+T⇒i↑F*F+T

    有 (⋖ +、(⋖ *、(⋖↑和(⋖ i。

Operator precedence analysis

- **优先关系表**
  注意：#E#

	+	*	↑	i	(	)	#
+	⋗	⋖	⋖	⋖	⋖	⋗	⋗
*	⋗	⋗	⋖	⋖	⋖	⋗	⋗
↑	⋗	⋗	⋖	⋖	⋖	⋗	⋗
i	⋗	⋗	⋗			⋗	⋗
(	⋖	⋖	⋖	⋖	⋖	≐	
)	⋗	⋗	⋗			⋗	⋗
#	⋖	⋖	⋖	⋖	⋖		≐

Operator precedence analysis

- 从算符优先文法G构造优先关系表的算法：
  - 通过检查G的每个产生式的每个候选式，可找出所有满足a≒b的终结符对。
  - 确定满足关系<·和>的所有终结符对：

首先需要对G的每个非终结符P构造两个集合FIRSTVT(P)和LASTVT(P)：

$$FIRSTVT(P) = \{a | P \overset{+}{\Rightarrow} a\cdots \text{或} P \overset{+}{\Rightarrow} Qa\cdots, a \in V_T \text{而} Q \in V_N\}$$

$$LASTVT(P) = \{a | P \overset{+}{\Rightarrow} \cdots a \text{或} P \overset{+}{\Rightarrow} \cdots aQ, a \in V_T \text{而} Q \in V_N\}$$

Operator precedence analysis

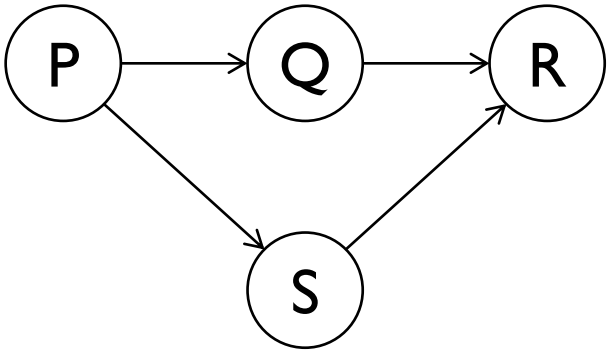
- 构造集合FIRSTVT(P)的算法：
  按其定义，可用下面两条规则来构造集合FIRSTVT(P)：
  - 1. 若有产生式P→a…或P→Qa…，则a∈FIRSTVT(P)；
  - 2. 若a∈FIRSTVT(Q)，且有产生式P→Q…，则a∈FIRSTVT(P)。

Operator precedence analysis

- 构造集合FIRSTVT(P)的算法：
- 计算Firstvt的传播

$$P \to Qa|S$$
$$Q \to Rb$$
$$S \to Rc$$
$$R \to d|b$$

非终结符Firstvt关系



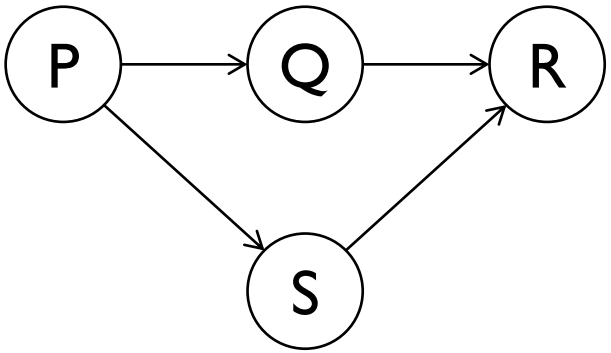
Stack	P,a Q,b S,c R,d R,b	Q,b S,c R,d R,b	P,b S,c R,d R,b	S,c R,d R,b
P	a	a	a,b	a,b
Q		b	b	b
S				c
R				

Operator precedence analysis

- 构造集合FIRSTVT(P)的算法：
- 计算Firstvt的传播

$$P \to Qa|S$$
$$Q \to Rb$$
$$S \to Rc$$
$$R \to d|b$$

非终结符Firstvt关系



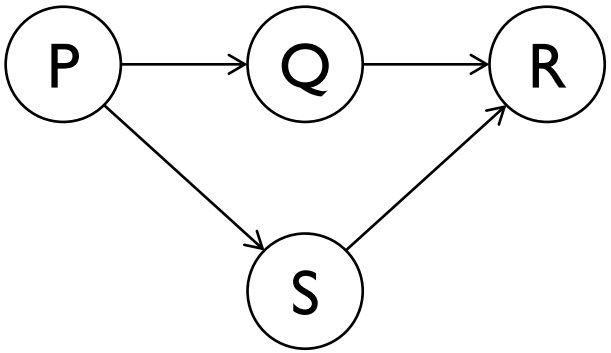
Stack	S,c R,d R,b	P,c R,d R,b	R,d R,b	Q,d S,d R,b
P	a,b	a,b,c	a,b,c	a,b,c
Q	b	b	b	b,d
S	c	c	c	c
R			d	d

Operator precedence analysis

- 构造集合FIRSTVT(P)的算法：
- 计算Firstvt的传播

$$P \to Qa|S$$
$$Q \to Rb$$
$$S \to Rc$$
$$R \to d|b$$

非终结符Firstvt关系



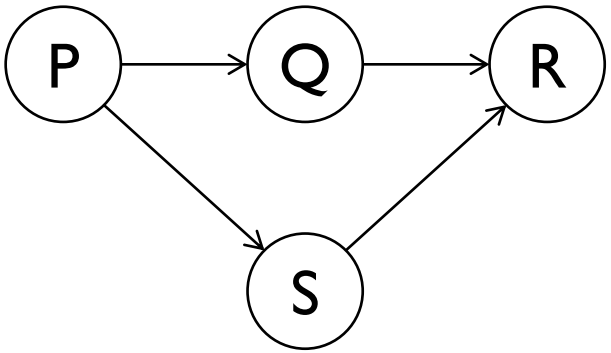
Stack	Q,d S,d R,b	P,d S,d R,b	S,d R,b	R,b
P	a,b,c	a,b,c,d	a,b,c,d	a,b,c,d
Q	b,d	b,d	b,d	b,d
S	c	c	c,d	c,d
R	d	d	d	b,d

Operator precedence analysis

- 构造集合FIRSTVT(P)的算法：
- 计算Firstvt的传播

$$P \to Qa|S$$
$$Q \to Rb$$
$$S \to Rc$$
$$R \to d|b$$

非终结符Firstvt关系



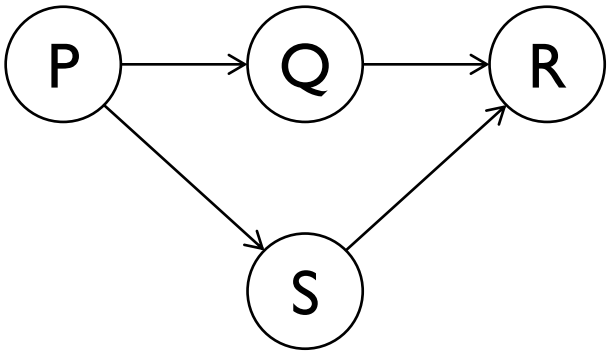
Stack	R,b	S,b Q,b	Q,b	
P	a,b,c,d	a,b,c,d	a,b,c,d	a,b,c,d
Q	b,d	b,d	b,d	b,d
S	c,d	b,c,d	b,c,d	b,c,d
R	b,d	b,d	b,d	b,d

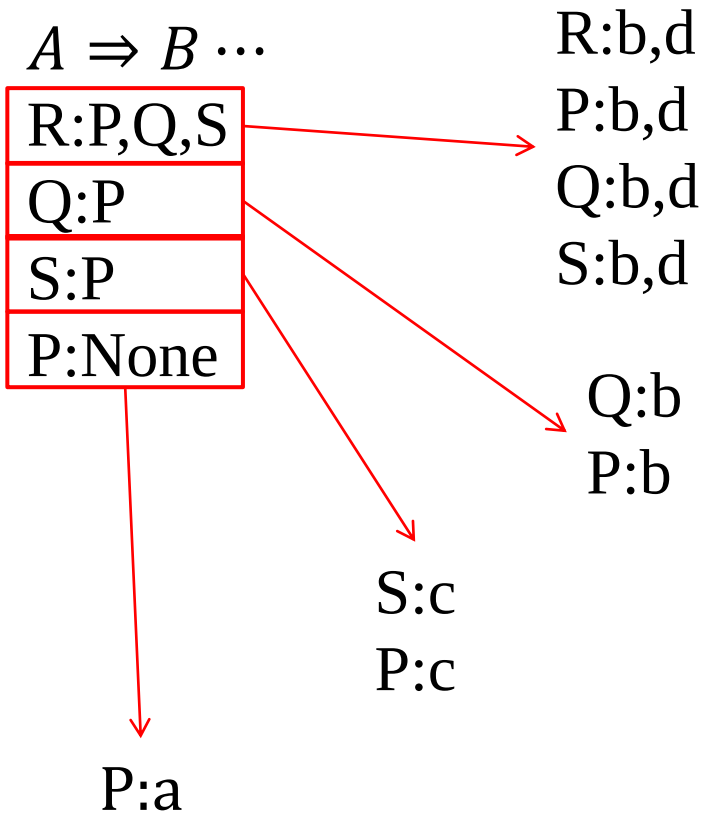
Operator precedence analysis

- 构造集合FIRSTVT(P)的算法：Alter
- 计算Firstvt的传播

$$P \to Qa|S$$
$$Q \to Rb$$
$$S \to Rc$$
$$R \to d|b$$

非终结符Firstvt关系





Operator precedence analysis

- 构造集合LASTVT(P)的算法：
  按其定义，可用下面两条规则来构造集合LASTVT(P)：
  - 1. 若有产生式P→…a或P→…aQ，则a∈LASTVT(P)；
  - 2. 若a∈LASTVT(Q)，且有产生式P→ … Q，则a∈LASTVT(P)。

Operator precedence analysis

- 有了这两个集合之后，就可以通过检查每个产生式的候选式确定满足关系$\lessdot$和$\gtrdot$的所有终结符对

- 如：假定有个产生式的一个候选形为

  …aP…

  那么，对任何b∈FIRSTVT(P)，有 a $\lessdot$ b。
- 假定有个产生式的一个候选形为

  …Pb…

  那么，对任何a∈LASTVT(P)，有 a $\gtrdot$ b。

Operator precedence analysis

- 使用每个非终结符P的FIRSTVT(P)和LASTVT(P)，就能够构造文法G的优先表。构造优先表的算法是：
  - FOR 每条产生式$P \to X_1X_2 \ldots X_n$ DO
    FOR i:=1 TO n-1 DO
    BEGIN
    IF X_i和X_{i+1}均为终结符 THEN 置$X_i \doteq X_{i+1}$
    IF $i \le n-2$且X_i和X_{i+2}都为终结符
    但X_{i+1}为非终结符 THEN 置$X_i \doteq X_{i+2}$；
    IF X_i为终结符而X_{i+1}为非终结符 THEN
    FOR FIRSTVT(X_{i+1})中的每个a DO
    置 $X_i \lessdot a$；
    IF X_i为非终结符而X_{i+1}为终结符 THEN
    FOR LASTVT(X_i)中的每个a DO
    置 $a \gtrdot X_{i+1}$
    END

Operator precedence analysis

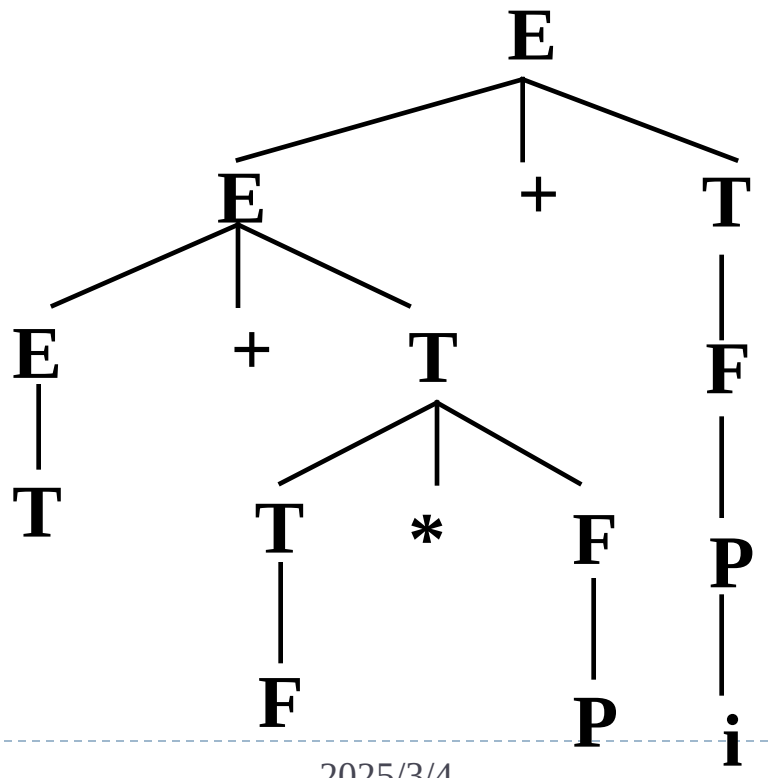
- 定义：令G是一个文法，S是文法的开始符号，假定αβδ是文法G的一个句型，如果有

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

- 则β称是句型αβδ相对于非终结符A的短语。
- 所谓素短语是指这样一个短语,它至少含有一个终结符,并且,除自身之外,不再含任何更小的素短语
- 句型最左边的那个素短语叫最左素短语。

Operator precedence analysis

- 考虑下面的文法G(E)：
  (1) E→E+T | T　　(2) T→T*F | F
  (3) F→P↑F | P　　(4) P→(E) | i
- 句型：T+F*P+i
- 短语：T+F*P+i, T, F,
  P, F*P, i, T+F*P
- 直接短语：T, F, P, i
- 句柄：T
- 素短语：F*P, i
- 最左素短语：F*P



Operator precedence analysis

- 算符优先文法句型(括在两个#之间)的一般形式写成：

  $$\#N_1a_1N_2a_2\ldots N_na_nN_{n+1}\#$$

  其中，每个a_i都是终结符，N_i是可有可无的非终结符。

- 定理： 一个算符优先文法G的任何句型的最左素短语是满足如下条件的最左子串 $N_ja_j\ldots N_ia_iN_{i+1}$，

  $$a_{j-1} \lessdot a_j \doteq a_{j+1} \doteq \cdots \doteq a_i \gtrdot a_{i+1}$$

Example

- 例8:考虑下面的文法G[S]:

S→a|∧|(T)　　T→T,S|S

1. 计算文法G的FIRSTVT和LASTVT;
2. 构造优先关系表;
3. G是算符优先文法吗?
4. 对 (a,(a,a))#进行归约

Example

S→a　S→∧　S→(T)

T→T,S　　T→S

非终结符	FIRSTVT集	LASTVT集
S	{ a ∧ (}	{ a ∧) }
T	{ a ∧ (, }	{ a ∧) , }

Example

	a	∧	(	)	,	#
a				⋗	⋗	⋗
∧				⋗	⋗	⋗
(	⋖	⋖	⋖	≒	⋖	
)				⋗	⋗	⋗
,	⋖	⋖	⋖	⋗	⋗	
#	⋖	⋖	⋖			≒

(a , (a , a)) #

(S , (a , a)) #

(S , (S , a)) #

(S , (S , S)) #

(S , (T)) #

(S , S) #

(T) #

S #

- (3) 输入串(a,(a,a))# 的算符优先分析过程为：

栈	当前字符	剩余输入串	动作
#	(	a,(a,a))#	Move in
#(	a	,(a,a))#	Move in
#(a	,	(a,a))#	Reduce: S→a
#(N	,	(a,a))#	Move in
#(N,	(	a,a))#	Move in
#(N,(	a	,a))#	Move in
#(N,(a	,	a))#	Reduce: S→a
#(N,(N	,	a))#	Move in
#(N,(N,	a	))#	Move in
#(N,(N,a	)	)#	Reduce: S→a
#(N,(N,N	)	)#	Reduce: T→T,S
#(N,(N	)	)#	Move in
#(N,(N)	)	#	Reduce: S→(T)
#(N,N	)	#	Reduce: T→T,S
#(N	)	#	Move in
#(N)	#		Reduce: S→(T)
#N	#		

LR

- LR分析法
  - The "L" is for left-to-right scanning of the input,
  - The "R" for constructing a rightmost derivation in reverse
  - 用于大多数CFG的语法分析

- 特点（相对于算符优先分析法或其它“移进 - 归约”技术）
  - 使用广泛、效率不差
  - 扫描时就能发现错误，并能准确指出出错位置
  - 工作量相当大
    - 故采用YACC等语法分析程序自动产生器

LR

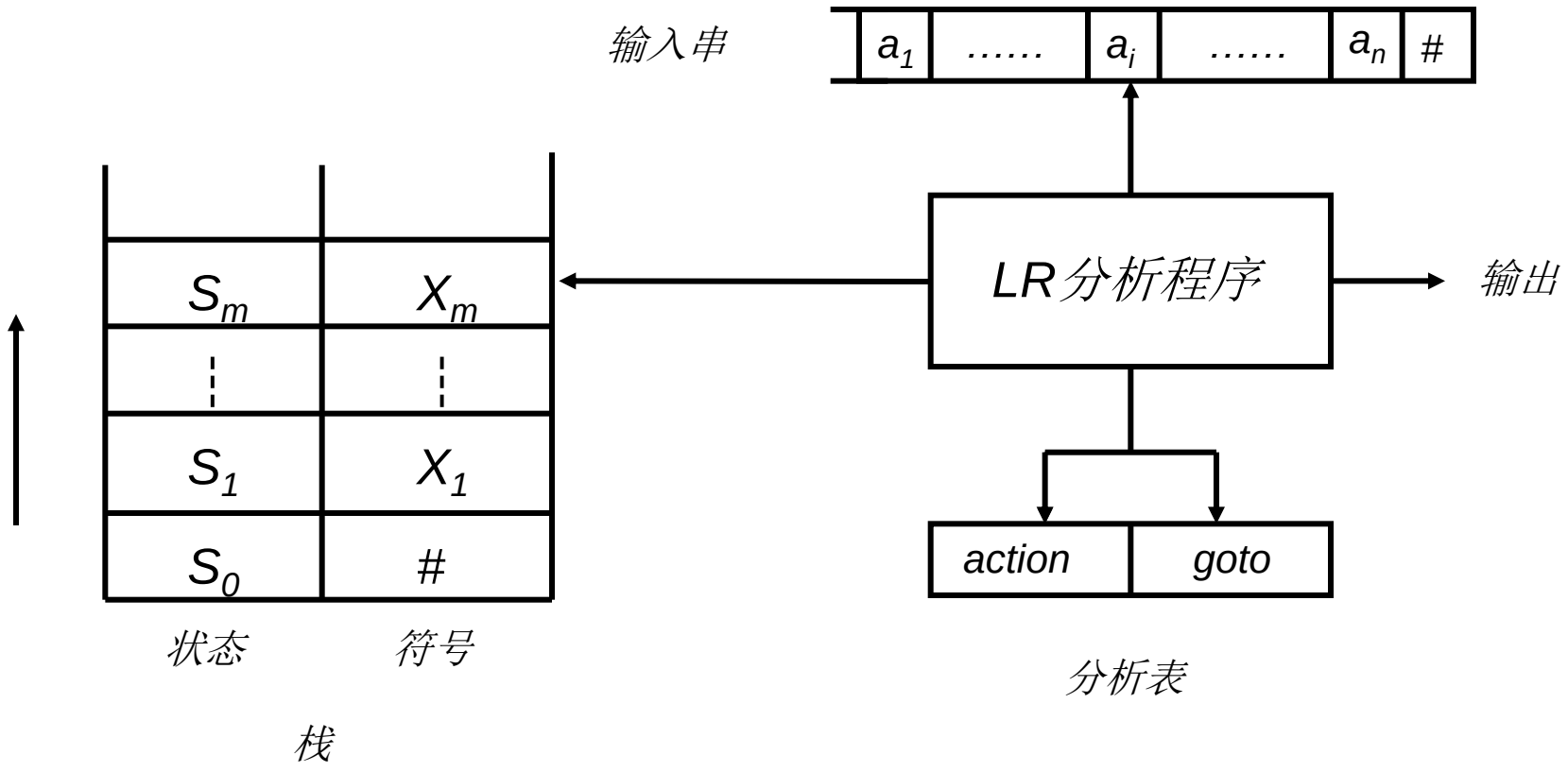
- 基本思想：在规范归约过程中，
  - 一方面记住“历史”
    - 记住已移进和归约出的整个符号串
  - 另一方面“展望”未来
    - 根据所用的产生式推测未来可能碰到的输入符号
- 当一串貌似句柄的符号串呈现于分析栈的顶端时
  - 根据 “历史”、“展望”和“现实”的输入符号等三方面信息
  - 来确定栈顶的符号串是否构成相对某一产生式的句柄

LR

- 一个LR分析器实质上是一个带先进后出存储器（栈）的确定状态有限状态自动机
  - 将“历史”与“展望”信息抽象成为某些状态
  - 先进后出存储器（分析栈）用于存放状态
  - 栈里的每个状态都概括了从分析开始直到某一归约阶段的全部“历史”和“展望”资料
  - 任何时候，栈顶的状态都代表整个历史和已推测的展望
- LR分析器的每一步工作都是由栈顶和现行输入符号所唯一决定的

LR

- LR分析程序的实质：分析栈 + DFA



LR table

- 分析表由ACTION表和GOTO表两部分组成。
  - ACTION(s，a)：表示当状态s面临输入a时的动作
  - GOTO(s，x)：表示面对文法符号x的下一状态

  - ACTION[s，a]表中的动作种类：
  - ① 移进
  - ② 归约
  - ③ 接受
  - ④ 报错