



中山大學
SUN YAT-SEN UNIVERSITY



国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

DCS290

Compilation Principle 编译原理

第四章 语法分析 (4)

郑馥丹

zhengfd5@mail.sysu.edu.cn

CONTENTS

目录

01

自顶向下分析
Top-Down Parsing

02

LL(1)分析
LL(1) Parsing

03

自底向上分析
Bottom-Up Parsing

04

LR分析
LR Parsing

1. 自底向上语法分析[Bottom-Up Parsing]

- 定义:

- 从输入符号串开始, 逐步进行归约, 直至归约到文法的开始符号。

- 语法树的构造:

- 从输入符号串开始, 以它作为语法树的末端结点符号串, 自底向上的构造语法树。

归约过程: cabd |- cAd |- S

用 “|-” 表示归约, 下划线部分为被归约符号

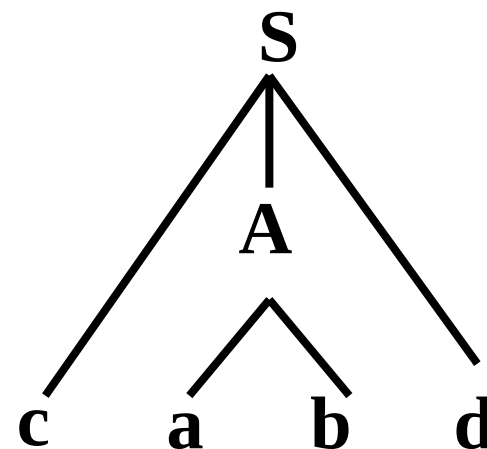
例: 文法G:

$S \rightarrow cAd$

$A \rightarrow ab$

$A \rightarrow a$

识别输入串 $w=cabd$ 是否为该文法的句子



1. 自底向上语法分析[Bottom-Up Parsing]

• 规范归约

- 自底向上分析的**归约**过程**是**自顶向下**推导的逆过程**。
- 最右推导为规范推导，所以**自左向右的归约称为规范归约**。

- 例 文法：

(1) $S \rightarrow aAcBe$	(2) $A \rightarrow b$
(3) $A \rightarrow Ab$	(4) $B \rightarrow d$

输入串 abbcd~~e~~的最右推导(规范推导)的过程为：

$S \Rightarrow aAc\underline{B}e \Rightarrow aA\underline{c}de \Rightarrow a\underline{A}bcde \Rightarrow a\underline{b}bcde$

自底向上分析的规范归约过程为：

$a\underline{b}bcde \mid -a\underline{A}bcde \mid -aA\underline{c}de \mid -aAc\underline{B}e \mid -S$

1. 自底向上语法分析[Bottom-Up Parsing]

- 自底向上分析的主要问题

例：对输入串cabd的两种归约过程

(1) **cabd**|-**cAd**|-S 归约到开始符

(2) **cabd**|-c**A**bd 不能归约到开始符

$S \rightarrow cAd$

$A \rightarrow ab$

$A \rightarrow a$

- 在自底向上的分析方法中，每一步都是从当前串中选择一个子串加以归约，该子串暂称“**可归约串**”。
- 自底向上分析的**主要问题**：**如何确定选用哪个产生式，即，如何确定“可归约串”**
 - ✓ **句柄(最左直接短语)**
 - ✓ **最左素短语**

1. 自底向上语法分析[Bottom-Up Parsing]

- 短语[phrase], 直接短语和句柄[handle]

- 设 $\alpha\beta\delta$ 是文法 $G[S]$ 中的一个句型, 如果有 $S \Rightarrow^* \alpha A \delta$ 且 $A \Rightarrow^+ \beta$, 则称 β 是句型 $\alpha\beta\delta$ 相对于非终结符 A 的**短语[phrase]**。
- 特别的如有 $A \Rightarrow \beta$, 则称 β 是句型 $\alpha\beta\delta$ 相对于规则 $A \rightarrow \beta$ 的**直接短语 (简单短语)**。
- 一个句型的最左直接短语称为该句型的**句柄[Handle]**。
- **句柄就是“可归约串”**。

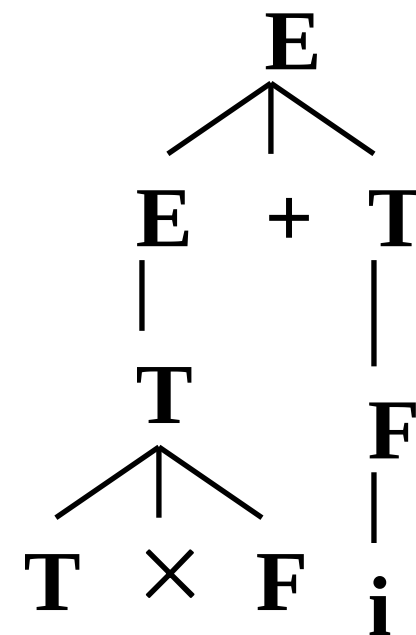
1. 自底向上语法分析[Bottom-Up Parsing]

• 短语[phrase], 直接短语和句柄[handle]

例：文法 $G[E]: E \rightarrow E + T | T, T \rightarrow T \times F | F, F \rightarrow (E) | i$ 的一个句型是 $T \times F + i$ ，相应的语法树见右图：

1. 因为 $E \Rightarrow^* T + i$ 且 $T \Rightarrow T \times F$ ，所以 $T \times F$ 是句型相对于 T 的短语，且是相对于 $T \rightarrow T \times F$ 的直接短语；
2. 因为 $E \Rightarrow^* T \times F + F$ 且 $F \Rightarrow i$ ，所以 i 是句型相对于 F 的短语，且是相对于 $F \rightarrow i$ 的直接短语；
3. 因为 $E \Rightarrow^* E$ 且 $E \Rightarrow^+ T \times F + i$ ，所以 $T \times F + i$ 是句型相对于 E 的短语；
4. $T \times F$ 是最左直接短语，即句柄。

注意：虽然 $F + i$ 是句型 $T \times F + i$ 的一部分，但**不是短语**，因为尽管有 $E \Rightarrow^+ T \times F + i$ ，但是**不存在从文法开始符 $E \Rightarrow^* T \times E$ 的推导**



若有 $S \Rightarrow^* \alpha A \delta$ 且 $A \Rightarrow^+ \beta$ ，则称 β 是句型 $\alpha \beta \delta$ 相对于非终结符 A 的短语。

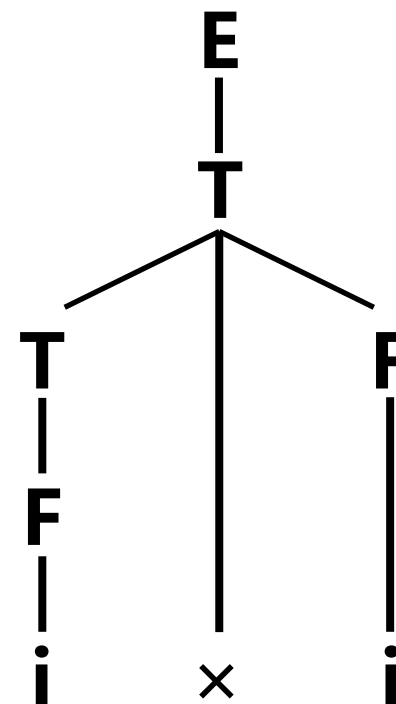
1. 自底向上语法分析[Bottom-Up Parsing]

• 句柄作为可归约串

– 例：文法 $G[E]$: $E \rightarrow E + T \mid T$, $T \rightarrow T \times F \mid F$, $F \rightarrow (E) \mid i$, 识别输入串 $w = i \times i$ 是否为该文法的句子

- 对于句型 $i \times i$, 第1个 i 为句柄 ($E \Rightarrow^* F \times F$, $F \Rightarrow i$)
- 对于句型 $F \times i$, F 为句柄 ($E \Rightarrow^* T \times F$, $T \Rightarrow F$)
- 对于句型 $T \times i$, i 为句柄 ($E \Rightarrow^* T \times F$, $F \Rightarrow i$)
- 对于句型 $T \times F$, $T \times F$ 为句柄 ($E \Rightarrow^* T$, $T \Rightarrow T \times F$)
- 对于句型 T , T 为句柄 ($E \Rightarrow^* T$, $T \Rightarrow T$)

$i \times i \mid -F \times i \mid -T \times i \mid -T \times F \mid -T \mid -E$

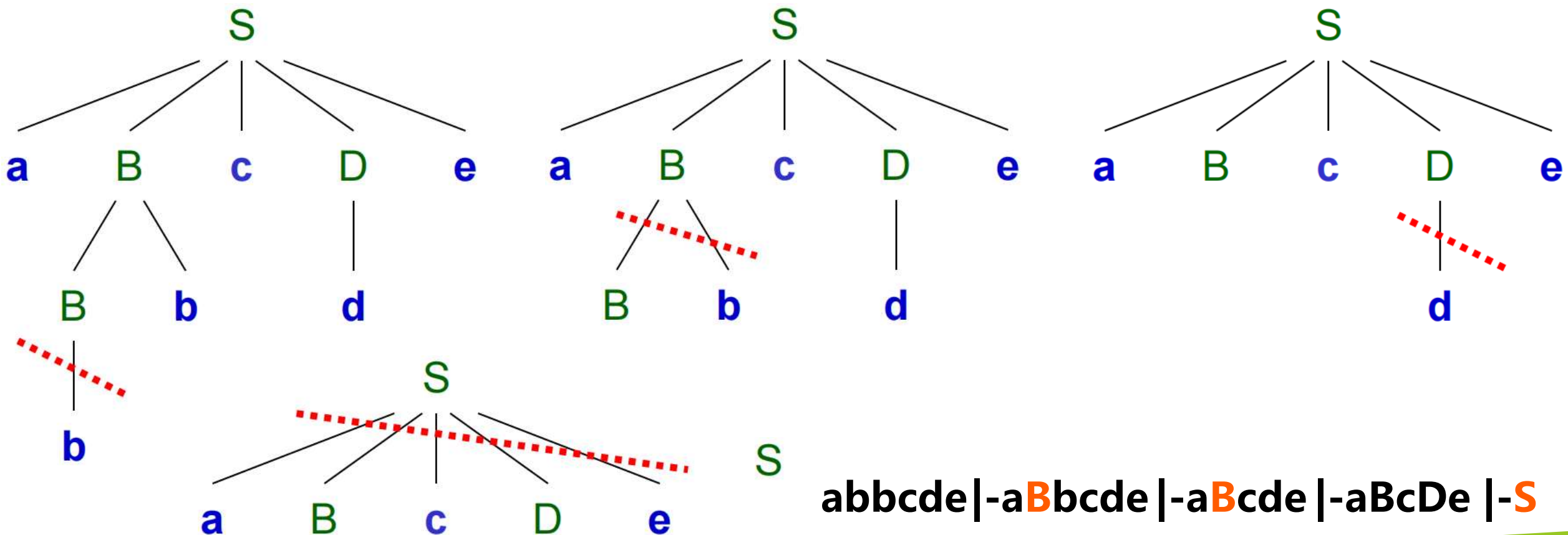


若有 $S \xRightarrow{*} \alpha A \delta$ 且 $A \Rightarrow \beta$, 则称 β 是句型 $\alpha \beta \delta$ 相对于非终结符 A 的直接短语。

1. 自底向上语法分析[Bottom-Up Parsing]

• 句柄剪枝

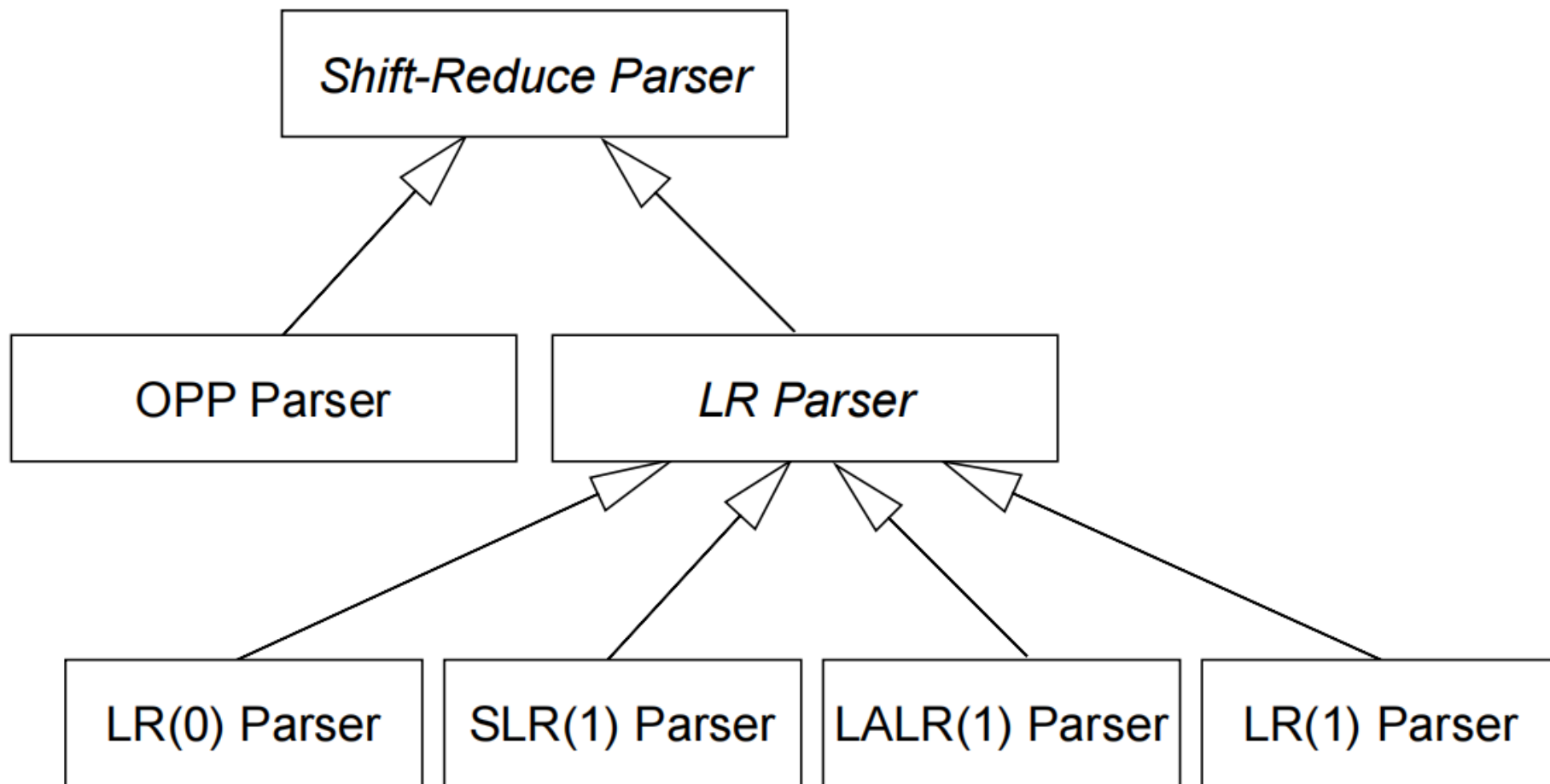
- 例：文法 $G[S]$: $S \rightarrow aBcDe$, $B \rightarrow Bb|b$, $D \rightarrow d$, 识别输入串 $w=abbcde$ 是否为该文法的句子



2. 移进-归约分析[Shift-Reduce Parsing]

- Shift-Reduce分析与LL(1)分析比较
 - 方向对称：
 - LL(1)分析：Top-down
 - Shift-Reduce分析：Bottom-up
 - 同为Table-driven方法
 - LL(1)分析：LL(1)预测分析表
 - Shift-Reduce分析：算符优先关系表/LR(0)/SLR(1)/LALR(1)/LR(1)分析表
 - 栈与输入串
 - LL(1)分析： $[\$S, w\$] \Rightarrow [\$, \$]$
 - Shift-Reduce分析： $[\$, w\$] \Rightarrow [\$S, \$]$
 - 动作
 - LL(1)分析：**match-derive**
 - Shift-Reduce分析：**shift-reduce**

2. 移进-归约分析[Shift-Reduce Parsing]



2. 移进-归约分析[Shift-Reduce Parsing]

- Shift-Reduce分析

- 引进一个先进后出的**符号栈**来存放符号;
- 对输入符号串自左向右进行扫描, 并把当前输入符号下推入栈中 (**移进**), 边移进边分析, 一旦栈顶符号串形成某个句型的**句柄** (为某产生式的右部) 时, 就用相应的非终结符 (产生式的左部) 替换它 (**归约**) 。
- 重复这一过程, 直到输入符号串的末端, **此时如果栈中只剩文法开始符号, 则输入符号串是文法的句子**, 否则不是。 $[\$, w\$] \Rightarrow [\$S, \$]$

2. 移进-归约分析[Shift-Reduce Parsing]

- 例：文法 $G[S]$: $S \rightarrow aBcDe$, $B \rightarrow Bb|b$, $D \rightarrow d$, 对输入串 $abbcde$ 进行移进-归约分析

步骤	符号栈	输入符号串	动作
1	\$	abbcde\$	移进a
2	\$a	bbcdde\$	移进b
3	\$a <u>b</u>	bcde\$	归约 ($B \rightarrow b$)
4	\$aB	bcde\$	移进b
5	\$a <u>Bb</u>	cde\$	归约 ($B \rightarrow Bb$)
6	\$aB	cde\$	移进c
7	\$aBc	de\$	移进d
8	\$aBc <u>d</u>	e\$	归约 ($D \rightarrow d$)
9	\$aBcD	e\$	移进e
10	\$a <u>BcDe</u>	\$	归约 ($S \rightarrow aBcDe$)
11	\$S	\$	接受

2. 移进-归约分析[Shift-Reduce Parsing]

• 例：文法G[E]:
 $E \rightarrow E + E \mid E \times E \mid (E) \mid n$
对输入串 $n + n \times n$
进行移进-归约分析

步骤	符号栈	输入字符串	动作
1	\$	$n + n \times n \$$	移进
2	$\$n$	$+ n \times n \$$	归约 $E \rightarrow n$
3	$\$E$	$+ n \times n \$$	移进
4	$\$E +$	$n \times n \$$	移进
5	$\$E + n$	$\times n \$$	归约 $E \rightarrow n$
6	$\$E + E$	$\times n \$$	归约 $E \rightarrow E + E$
7	$\$E$	$\times n \$$	ERROR
6'	$\$E + E$	$\times n \$$	移进
7'	$\$E + E \times$	$n \$$	移进
8	$\$E + E \times n$	$\$$	归约 $E \rightarrow n$
9	$\$E + E \times E$	$\$$	归约 $E \rightarrow E \times E$
10	$\$E + E$	$\$$	归约 $E \rightarrow E + E$
11	$\$E$	$\$$	接受

3. 算符优先分析[Operator-Precedence Parsing]

- 基本思想：只定义文法中**终结符**之间的优先关系（不考虑非终结符），并由这种关系指导分析过程。
- **不是规范归约。**
- 优点：算符优先分析法分析速度快，特别适用于表达式的分析。
- 缺点：不规范，常常要采用适当措施克服其缺点。
- 前提：**文法必须是算符文法**

3. 算符优先分析[Operator-Precedence Parsing]

- 算符文法[Operator Grammar]

- 设有上下文无关文法 G ，如果 G 中产生式的右部**没有两个非终结符相连**，则称 G 为算符文法
- 算符文法中不含 ϵ 产生式
- 例： $G[E]: E \rightarrow E + E | E \times E | (E) | i$ 是算符文法
- 例： $G[E]: E \rightarrow E A E | (E) | -E | id$ 不是算符文法

3. 算符优先分析[Operator-Precedence Parsing]

- 算符优先

- 优先关系只存在于句型中**相邻**出现的符号
- 算符优先分析法**只考虑终结符之间的优先关系**，不考虑非终结符，所以**两个终结符相邻指它们之间没有其他的终结符**（但可以有非终结符）
- 如： $E+T \times i$ ，则 $+$ 和 $\times$ 相邻， $\times$ 和 i 相邻，但 $+$ 和 i 不相邻
- 算符有三种优先关系：
 - ✓ $a < b$ 表示 a 的优先级低于 b
 - ✓ $a \equiv b$ 表示 a 的优先级等于 b
 - ✓ $a > b$ 表示 a 的优先级高于 b

3. 算符优先分析[Operator-Precedence Parsing]

- 算符优先

- 注意:

- ✓ $<, \equiv, >$ 是偏序关系

- ✓ 不具备自反性: a 不一定 $\equiv a$

- ✓ 不具备对称性: $a < b$ 不意味着 $b > a$, $a \equiv b$ 不意味着 $b \equiv a$

- ✓ 不具备传递性: $a < b$, $b < c$ 不意味着 $a < c$

- ✓ 算法优先关系仅考虑相邻两个终结符之间的关系, 若终结符 a 和 b 在任何句型中都不相邻, 则无法比较它们的优先关系

3. 算符优先分析[Operator-Precedence Parsing]

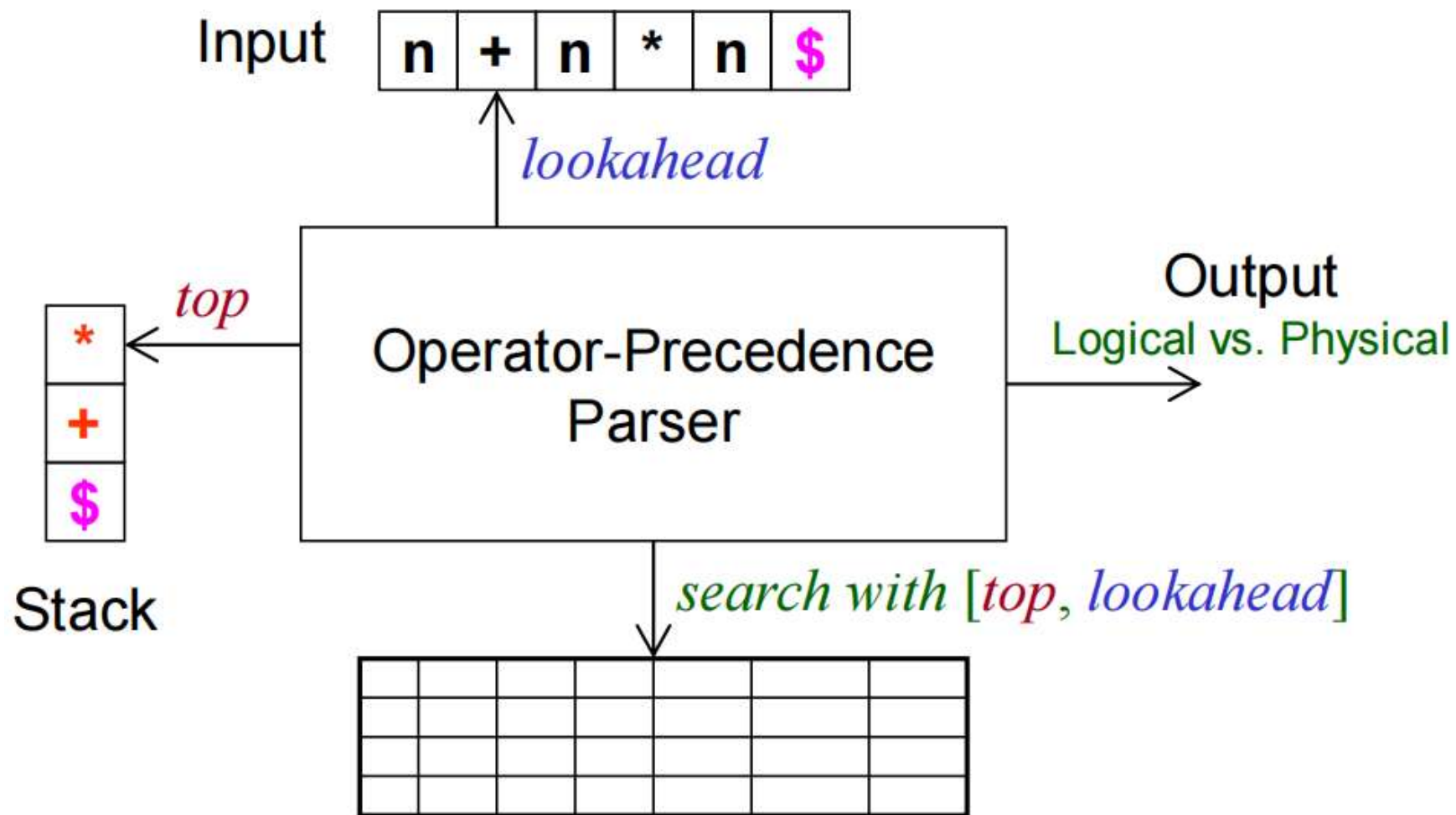
- 同样是Table-driven
- 依赖于算符优先关系表[Operator-Precedence Relation Table]

	id	+	*	\$
id		$\prec$	$\prec$	$\prec$
+	$\prec$	$\prec$	$\prec$	$\prec$
*	$\prec$	$\prec$	$\prec$	$\prec$
\$	$\prec$	$\prec$	$\prec$	

算符优先关系表

3. 算符优先分析[Operator-Precedence Parsing]

- 算符优先分析



3. 算符优先分析[Operator-Precedence Parsing]

- 算法

```
initialize();
for (;;) {
    if (top == $ && lookahead == $) accept();
    topOp = stack.topMostTerminal();
    if (topOp < lookahead || topOp == lookahead) { // shift
        stack.push(lookahead);
        lookahead = scanner.getNextToken();
    } else if (topOp > lookahead) { // reduce
        do {
            topOp = stack.pop();
            寻找最左素短语，用其进行归约
        } while (stack.topMostTerminal() > || == topOp);
    } else error();
}
```

3. 算符优先分析[Operator-Precedence Parsing]

- 最左素短语

- 设有文法 $G[S]$ ，其句型的素短语是一个短语，它至少包含一个终结符，并除自身外不包含其它素短语。
- 最左边的素短语称为最左素短语。
- 因此，找素短语，就要先找短语。

• 最左素短语

– 例：文法 $G[E]$: $E \rightarrow E+T \mid T$, $T \rightarrow T \times F \mid F$, $F \rightarrow (E) \mid i$ 的一个句型为 $T+T \times F+i$

✓ 短语为: T , $T \times F$, i , $T+T \times F$, $T+T \times F+i$

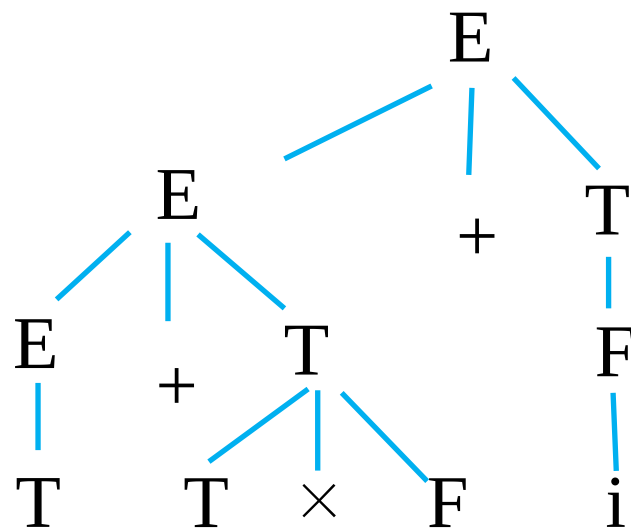
✓ 素短语为: $T \times F$, i

✓ 最左素短语为: $T \times F$

– **规范归约** 归约当前句型中的**句柄**(T)

– **算符优先分析** 归约当前句型中的**最左素短语**($T \times F$)

– **算符优先分析** 去掉了单个非终结符的归约，因为若只有一个非终结符，无法与句型中该非终结符左部和右部的串比较优先关系，也就无法确定如何归约。



素短语：是一个短语，至少包含一个终结符，并除自身外不包含其它素短语。

• 算法

- 将输入符号串 $a_1a_2\dots a_t$ 依次逐个存入符号栈 S 中，直至符号栈顶终结符 a_j 与当前输入符 a_{j+1} 的关系为 $a_j > a_{j+1}$ 为止；
- 最左素短语尾已在符号栈 S 的栈顶，由栈顶向下找最左素短语的头符号，即找到第一个 $<$ 关系 $a_{i-1} < a_i$ ；
- 已找到最左素短语 $N_i a_i N_{i+1} a_{i+1} \dots N_j a_j N_{j+1}$ (其中 $N_k (i \leq k \leq j+1)$ 为非终结符或空)，用相应的产生式进行归约；
- 重复上述过程，当处理到输入符号串尾 $\$$ 时，若栈中只剩： $\$N$ (N 为任何一个非终结符)，则分析成功，否则分析失败。

3. 算符优先分析[Operator-Precedence Parsing]

- 文法G[E]: $E \rightarrow E+T|T$, $T \rightarrow T \times F|F$, $F \rightarrow (E)|i$, 采用算符优先分析法识别输入串 $w=i \times i$ 是否为该文法的句子

步骤	栈	优先关系	当前符号	剩余输入串	动作
1	\$	<	i	+i\$	移进
2	\$i	>	+	i\$	归约 $F \rightarrow i$
3	\$F	<	+	i\$	移进
4	\$F+	<	i	\$	移进
5	\$F+i	>	\$		归约 $F \rightarrow i$
6	\$F+F	>	\$		归约 $E \rightarrow E+T$
7	\$E	=	\$		ACCEPT

	+	×	(	)	i	\$
+	>	<	<	>	<	>
×	>	>	<	>	<	>
(	<	<	<	=	<	
)	>	>		>		>
i	>	>		>		>
\$	<	<	<		<	=

表格中空白的位置
表示error

3. 算符优先分析[Operator-Precedence Parsing]

• 例：文法 $G[E]: E \rightarrow E + E | E \times E | (E) | n$ ，对输入串 $n + n \times n$ 进行移进-归约分析

步骤	符号栈	输入字符串	动作
1	\$	$n + n \times n \$$	移进
2	$\$n$	$+ n \times n \$$	归约 $E \rightarrow n$
3	$\$E$	$+ n \times n \$$	移进
4	$\$E +$	$n \times n \$$	移进
5	$\$E + n$	$\times n \$$	归约 $E \rightarrow n$
6	$\$E + E$	$\times n \$$	移进? 归约?
7	$\$E + E \times$	$n \$$	移进
8	$\$E + E \times n$	$\$$	归约 $E \rightarrow n$
9	$\$E + E \times E$	$\$$	归约?
10	$\$E + E$	$\$$	归约 $E \rightarrow E + E$
11	$\$E$	$\$$	ACCEPT

	+	$\times$	(	)	n	\$
+	>	<	<	>	<	>
$\times$	>	>	<	>	<	>
(	<	<	<	$\equiv$	<	
)	>	>		>		>
n	>	>		>		>
\$	<	<	<		<	$\equiv$

移进

归约 $E \rightarrow E \times E$

3. 算符优先分析[Operator-Precedence Parsing]

- 算法说明:

- 算法中每次都**取终结符进行比较**，当栈顶符号不是终结符时，便取其下一个符号（这时一定是终结符）
- 归约时检查是否有与最左素短语对应的产生式，查看产生式的右部：
 - ✓ 符号个数与该素短语的符号个数相等
 - ✓ **非终结符位置对应，不管其符号名是什么**
 - ✓ 终结符名字和位置都对应相等
- 若有满足以上条件的产生式才归约，否则出错

- 有文法 $G[S]: S \rightarrow (L)|a, L \rightarrow L,S|S$ ，对输入符号串 $(a, (a, a))$ 进行算法优先分析（假设有如下算法优先关系表）

步骤	符号栈	优先关系	输入字符串	动作
1	\$	<	(a, (a, a))\$	移进
2	\$(	<	a, (a, a))\$	移进
3	\$(a	>	, (a, a))\$	归约 $S \rightarrow a$
4	\$(S	<	, (a, a))\$	移进
5	\$(S,	<	(a, a))\$	移进
6	\$(S,(	<	a, a))\$	移进
7	\$(S,(a	>	, a))\$	归约 $S \rightarrow a$
8	\$(S,(S	<	, a))\$	移进
9	\$(S,(S,	<	a))\$	移进

	a	(	)	,	\$
a			>	>	>
(	<	<	=	<	
)			<	<	<
,	<	<	<	<	
\$	<	<			

随堂练习 (8)

- 有文法G[S]: $S \rightarrow (L)|a$, $L \rightarrow L,S|S$, 对输入符号串(a, (a, a))进行算法优先分析 (假设有如下算法优先关系表)

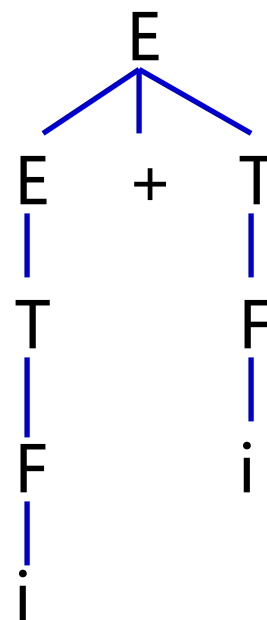
步骤	符号栈	优先关系	输入字符串	动作
10	\$(S,(S, a	>	)\$	归约 $S \rightarrow a$
11	\$(S,(S, S	>	)\$	归约 $L \rightarrow L,S$
12	\$(S,(L	≡	)\$	移进
13	\$(S,(L)	>	)\$	归约 $S \rightarrow (L)$
14	\$(S,S	>	)\$	归约 $L \rightarrow L,S$
15	\$(L	≡	)\$	移进
16	\$(L)	>	\$	归约 $S \rightarrow (L)$
17	\$S		\$	ACCEPT

	a	(	)	,	\$
a			<	<	<
(	<	<	≡	<	
)			<	<	<
,	<	<	<	<	
\$	<	<			

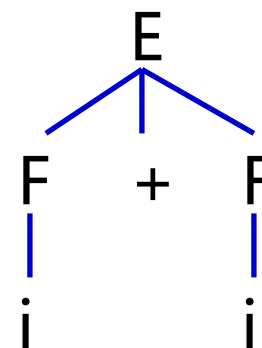
3. 算符优先分析[Operator-Precedence Parsing]

- 规范归约 vs. 算符优先分析

1. 规范归约将输入符号串**归约为文法的开始符号**；算符优先分析将输入符号串**归约为任意一个非终结符**
2. 规范归约每次**归约句柄**；算符优先分析每次**归约最左素短语**，去掉了单个非终结符的归约
3. 规范归约选取右部与句柄一致的产生式进行归约；算符优先分析归约时选择产生式时不管非终结符的名字



采用规范归约
识别 $i+i$ 的过程



采用算符优先分析
识别 $i+i$ 的过程

3. 算符优先分析[Operator-Precedence Parsing]

- 优先函数

- 优先矩阵占用大量空间: $(n+1)^2$
- 在实际应用中用优先函数来代替优先矩阵表示优先关系
- 优先函数: 对给定的优先关系表, 定义两个函数 f , g , 它们应满足:
 - ✓ 当 $a \equiv b$ 则令 $f(a)=g(b)$
 - ✓ 当 $a < b$ 则令 $f(a)<g(b)$
 - ✓ 当 $a > b$ 则令 $f(a)>g(b)$

我们称 f , g 为优先函数, 其取值可用整数表示。

3. 算符优先分析[Operator-Precedence Parsing]

- 优先函数

优先关系表 $(n+1)^2$ 个单元

	i	×	+	#
i		>	>	>
×	<	>	>	>
+	<	<	>	>
#	<	<	<	≡

优先函数表 $2(n+1)$ 个单元

	i	×	+	#
f	6	6	4	2
g	7	5	3	2

- 优点：节省大量内存
- 缺点：原先无优先关系的对，如：(i,i) 变成有关系： $f(i)=6 < g(i)=7$ ，不能准确指出错误位置

3. 算符优先分析[Operator-Precedence Parsing]

- 优先函数

- 用关系图法构造优先函数

- ✓ 对每个终结符 a (包括‘\$’)画出 f_a, g_a 结点,

- 若 $a > b$ 或 $a \equiv b$ 则从 f_a 到 g_b 画一条弧,

- 若 $a < b$ 或 $a \equiv b$ 则从 g_b 到 f_a 画一条弧。

- 函数值: **从该结点出发所能到达的结点(含自身结点)的个数**

3. 算符优先分析[Operator-Precedence Parsing]

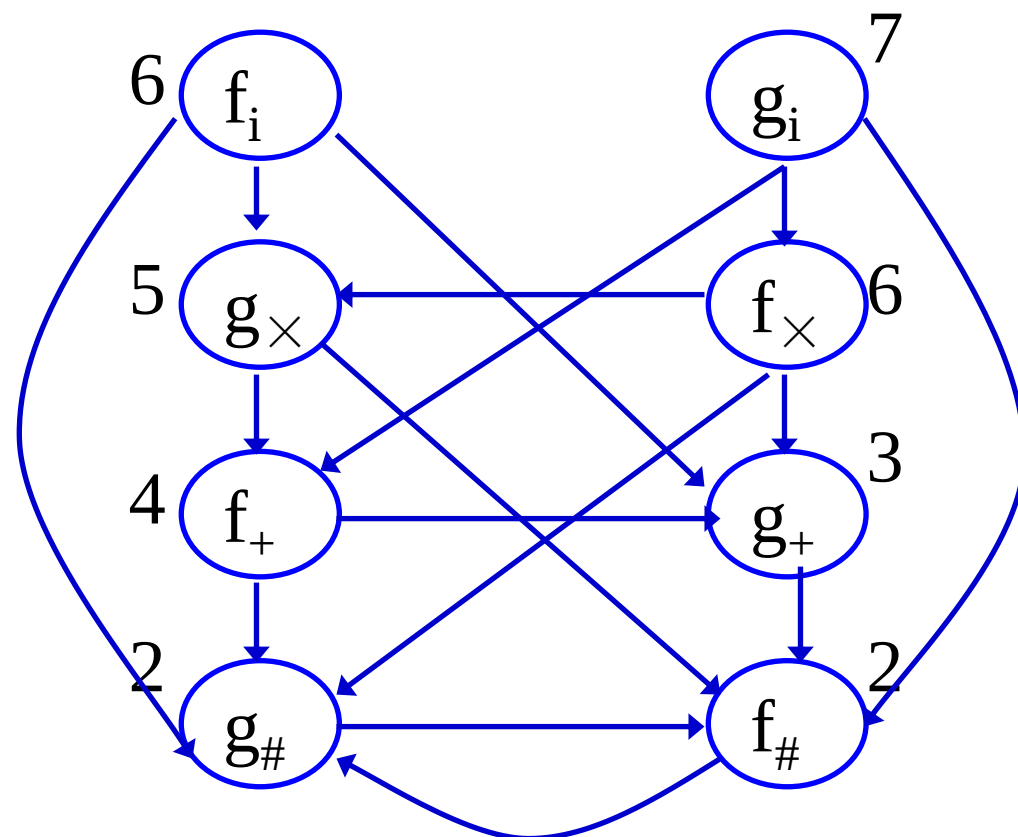
- 例：有优先关系表如下：

	i	×	+	#
i		>	>	>
×	<	>	>	>
+	<	<	>	>
#	<	<	<	≡

优先函数结果为：

	i	×	+	#
f	6	6	4	2
g	7	5	3	2

若 $a > b$ 或 $a \equiv b$ 则从 f_a 到 g_b 画一条弧，
若 $a < b$ 或 $a \equiv b$ 则从 g_b 到 f_a 画一条弧。



从 f_i 出发所能到达的结点有：

$f_i, g_#, f_#, g_×, f_+, g_+$

3. 算符优先分析[Operator-Precedence Parsing]

- 局限性

- 适用范围窄，很多实用语言的文法不是算符优先文法
- 只能处理非常简单的语言
- 有时错误的句子得到了正确的归约，原因：
 - ✓ 去掉了单个非终结符的归约
 - ✓ 归约时，选择产生式不管产生式右部非终结符的名字，只要非终结符位置对应相同就归约

- 商业应用中仍存在

- Sun javac编译器
- 早期的THINK C编译器
- 贝尔实验室的原始C编译器