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

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

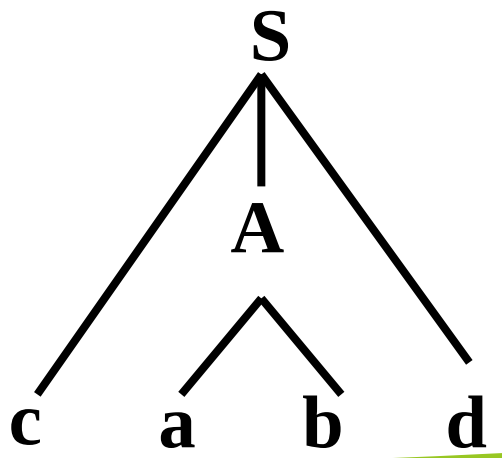
例：文法G：

S → cAd

A → ab

A → 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→aAcBe　　(2)　A→b

    (3)　A→Ab　　(4)　B→d

    输入串 abbcde的最右推导(规范推导)的过程为：

    S=>**aAcBe**=>a**A**cde=>a**Ab**cde=>a**b**bcde

    自底向上分析的规范归约过程为：

    abbcde|-a**A**bcde|-a**A**cde|-aAc**B**e|-**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|-cAbd**　　不能归约到开始符

S → cAd
A → ab
A → 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**

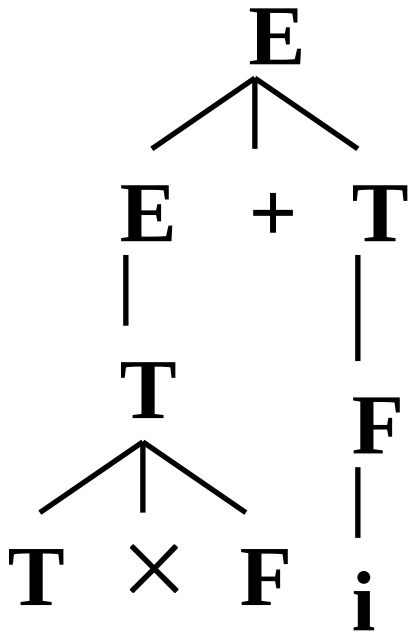
1. 自底向上语法分析[Bottom-Up Parsing]

- **短语[phrase]，直接短语和句柄[handel]**
  - 设αβδ是文法G[S]中的一个句型，如果有S=*>αAδ且A=+>β，则称β是句型αβδ相对于非终结符A的**短语[phrase]**。
  - 特别的如有A=>β，则称β是句型αβδ相对于规则A→β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[handel]**

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



1. 因为E=*> T+i 且 T=>T×F，所以**T×F**是句型相对于T的短语，且是相对于T→T×F的直接短语；
2. 因为E=*> T×F+F 且 F=>i，所以**i**是句型相对于F的短语，且是相对于F→i的直接短语；
3. 因为E=*> E 且E=$^{+}$>T×F+i，所以**T×F+i**是句型相对于E的短语；
4. **T×F**是最左直接短语，即句柄。

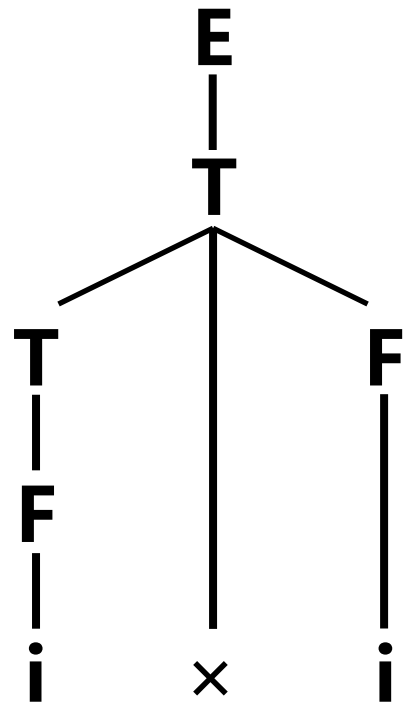
注意：虽然**F+i**是句型T×F+i的一部分，但**不是短语**，因为尽管有E=$^{+}$>F+i，但是**不存在从文法开始符E=* >T×E的推导**

若有S=*>αAδ且A=$^{+}$>β，则称β是句型αβδ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→E+T|T，T→T×F|F，F→(E)|i，识别输入串w=i×i是否为该文法的句子
    - 对于句型i×i，第1个i为句柄(E=>*F×F, F=>i)
    - 对于句型F×i，F为句柄(E=>*T×F, T=>F)
    - 对于句型T×i，i为句柄(E=>*T×F, F=>i)
    - 对于句型T×F，T×F为句柄(E=>*T, T=>T×F)
    - 对于句型T，T为句柄(E=>*T, T=>T)

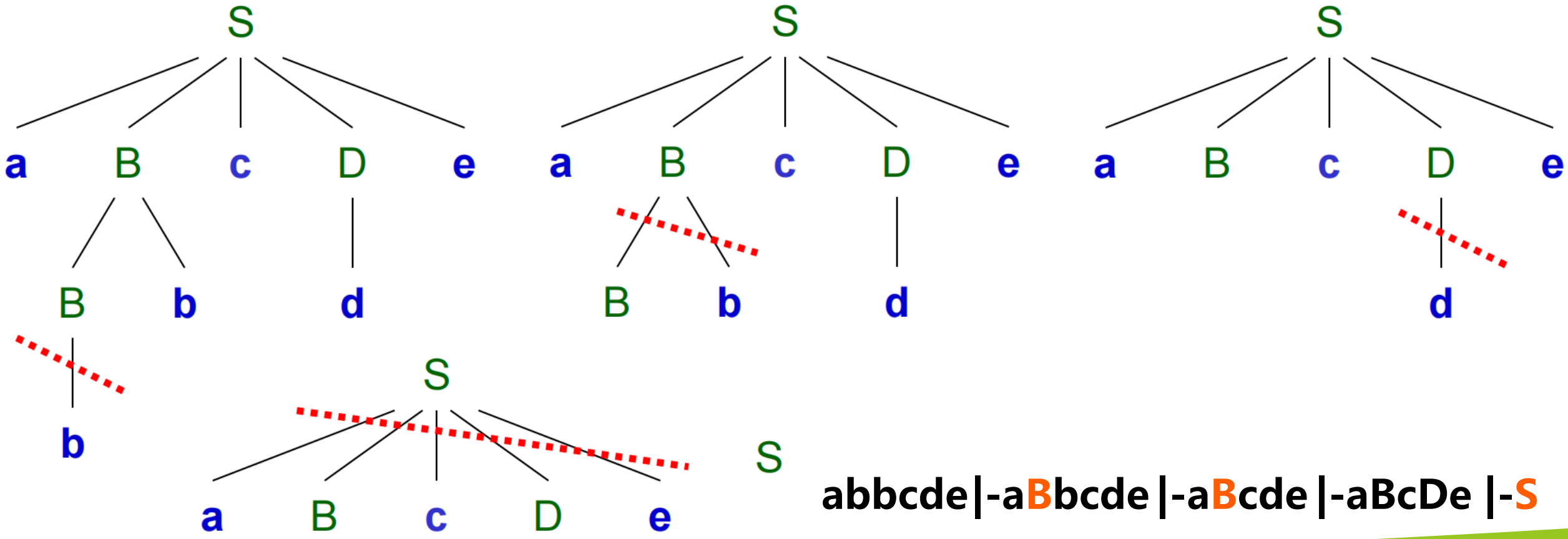
i×i |-F×i |-T×i |-T×F |-T |-E



若有S=*>αAδ且A=>β，则称β是句型αβδ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]

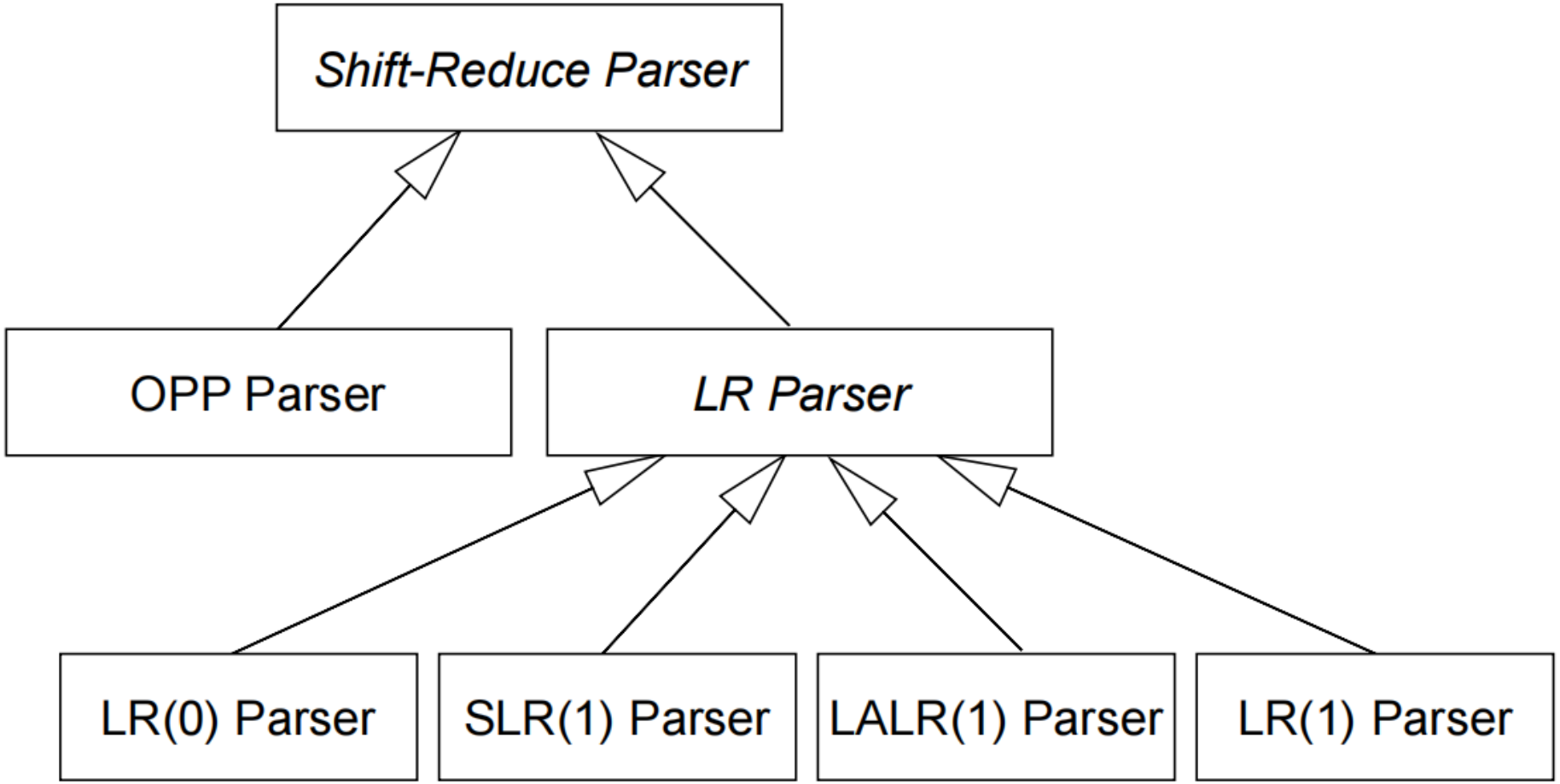
- 句柄剪枝
  - 例：文法G[S]：S→aBcDe，B→Bb|b，D→d，识别输入串w=abbcde是否为该文法的句子



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。**[$, *w*$] ⇒ [$S, $]**

2. 移进-归约分析[Shift-Reduce Parsing]

- 例：文法G[S]：S→aBcDe，B→Bb|b，D→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	$a	bbcde$	移进b
3	$ab	bcde$	归约（B→b）
4	$aB	bcde$	移进b
5	$aBb	cde$	归约（B→Bb）
6	$aB	cde$	移进c
7	$aBc	de$	移进d
8	$aBcd	e$	归约（D→d）
9	$aBcD	e$	移进e
10	$aBcDe	$	归约（S→aBcDe）
11	**$S**	**$**	**接受**

2. 移进-归约分析[Shift-Reduce Parsing]

- 例：文法G[E]:

E→E+E|E×E|(E)|n

对输入串n+n×n

进行移进-归约分析

步骤	符号栈	输入字符串	动作
1	$	n+n×n$	移进
2	$n	+n×n$	归约E→n
3	$E	+n×n$	移进
4	$E+	n×n$	移进
5	$E+n	×n$	归约E→n
6	$E+E	×n$	**归约E→E+E**
7	$E	×n$	**ERROR**
6’	$E+E	×n$	**移进**
7’	$E+E×	n$	移进
8	$E+E×n	$	归约E→n
9	$E+E×E	$	**归约E→E+E?**
10	$E×E	$	**归约E→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[Operater 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→E+E|E×E|(E)|i 是算符文法
  - 例：G[E]：E→EAE|(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×i，则＋和×相邻，×和i相邻，但＋和i不相邻
  - 算符有三种优先关系：
    - ✓ $a \lessdot b$　　表示a的优先级低于b
    - ✓ $a \equiv b$　　表示a的优先级等于b
    - ✓ $a \gtrdot b$　　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：
    - ✓ $\lessdot, \equiv, \gtrdot$ 是**偏序关系**
    - ✓ **不具备自反性**：a不一定≡a
    - ✓ **不具备对称性**：a$\lessdot$b不意味着b$\gtrdot$a，a≡b不意味着b≡a
    - ✓ **不具备传递性**：a$\lessdot$b，b$\lessdot$c不意味着a$\lessdot$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]

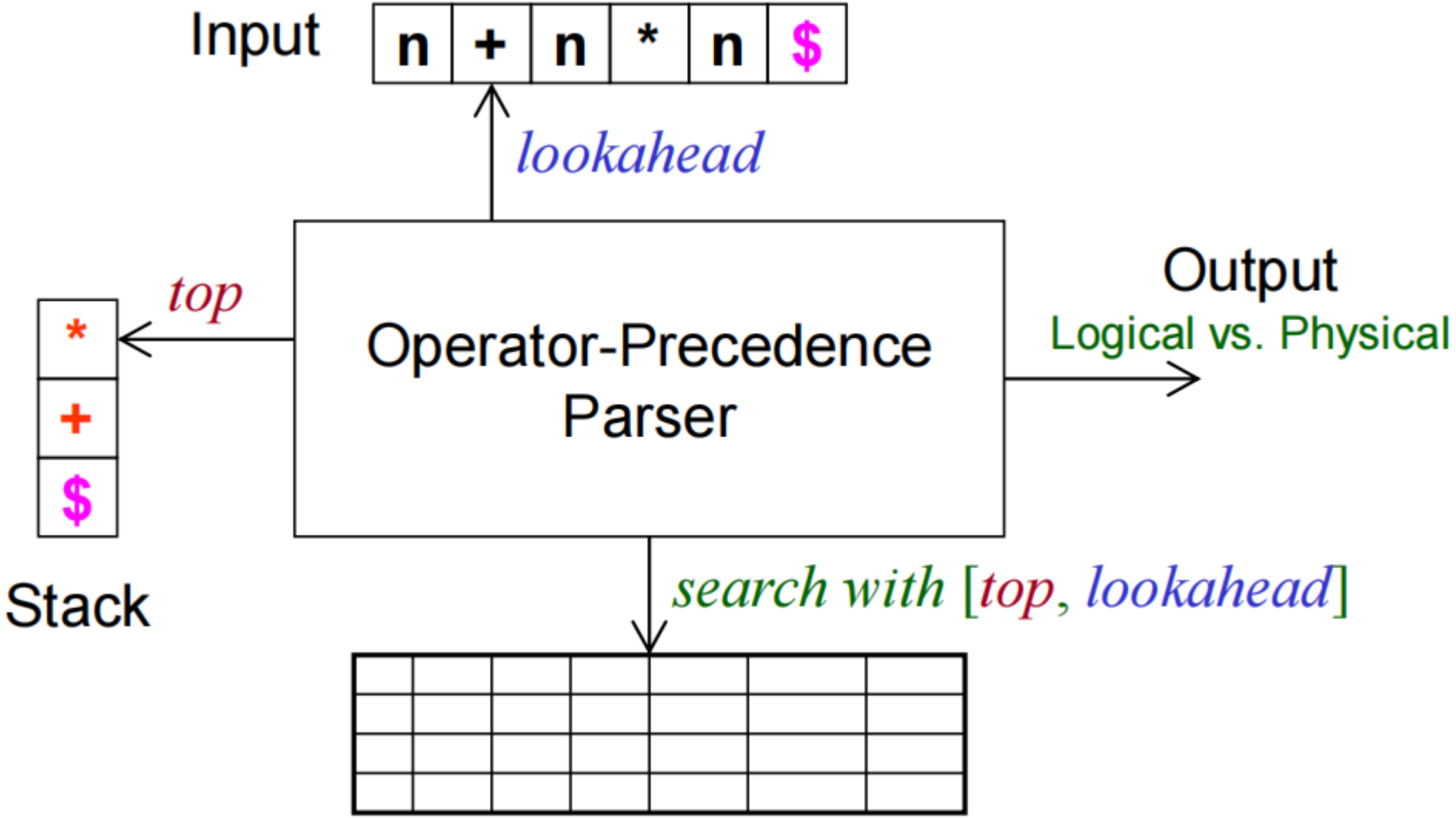
- 同样是Table-driven
- 依赖于算符优先关系表[Operator-Precedence Relation Table]

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

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();
        } while (stack.topMostTerminal() ≻ || ≡ topOp);
    } else  error();
}
```

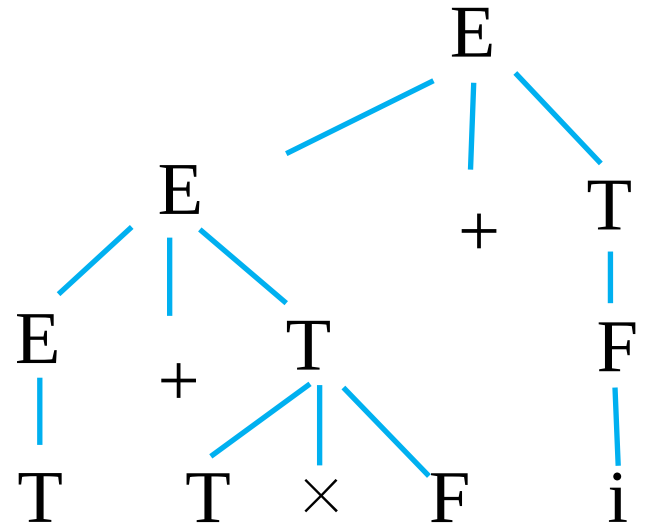
寻找最左素短语，用其进行归约

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。

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

- **最左素短语**
  - 例：文法G[E]：E→E+T|T，T→T×F|F，F→(E)|i 的一个句型为T+T×F+i
    - ✓ 短语为：T，T×F，i，T+T×F，T+T×F+i
    - ✓ 素短语为：T×F，i
    - ✓ 最左素短语为：T×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×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。

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

- 算法
  - 将输入符号串$a_1a_2...a_t$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$a_j \succ a_{j+1}$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$\prec$关系$a_{i-1} \prec a_i$；
  - 已找到**最左素短语**$N_ia_iN_{i+1}a_{i+1}.....N_ja_jN_{j+1}$(其中$N_k(i \leqslant K \leqslant j+1)$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→E+T|T，T→T×F|F，F→(E)|i，采用算符优先分析法识别输入串w=i×i是否为该文法的句子

	+	×	(	)	i	$
+	≻	≺	≺	≻	≺	≻
×	≻	≻	≺	≻	≺	≻
(	≺	≺	≺	≡	≺	
)	≻	≻		≻		≻
i	≻	≻		≻		≻
$	≺	≺	≺		≺	≡

表格中空白的位置表示error

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→i
3	$F	≺	+	i$	移进
4	$F+	≺	i	$	移进
5	$F+i	≻	$		归约 F→i
6	$F+F	≻	$		归约 E→E+T
7	$E	≡	$		ACCEPT

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

- 例：文法G[E]: E→E+E|E×E|(E)|n，对输入串n+n×n进行移进-归约分析

步骤	符号栈	输入字符串	动作
1	$	n+n×n$	移进
2	$n	+n×n$	归约E→n
3	$E	+n×n$	移进
4	$E+	n×n$	移进
5	$E+n	×n$	归约E→n
6	$E+E	×n$	**移进？归约？**
7	$E+E×	n$	移进
8	$E+E×n	$	归约E→n
9	$E+E×E	$	**归约？**
10	$E+E	$	规约E→E+E
11	$E	$	**ACCEPT**

	+	×	(	)	n	$
+	≻	≺	≺	≻	≺	≻
×	≻	≻	≺	≻	≺	≻
(	≺	≺	≺	≡	≺	
)	≻	≻		≻		≻
n	≻	≻		≻		≻
$	≺	≺	≺		≺	≡

移进

归约E→E×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

随堂练习（8）

- 有文法G[S]：S→(L)|a，L→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→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→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→(L)|a，L→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）

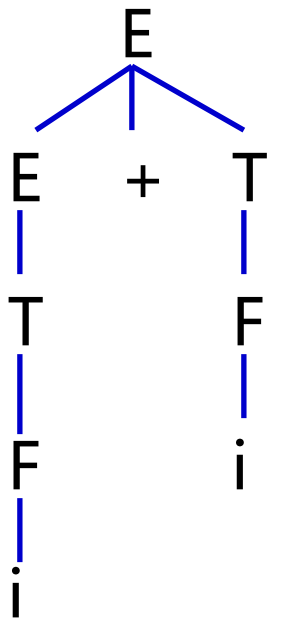
步骤	符号栈	优先关系	输入字符串	动作
10	$(S,(S, **a**	≻	**)**)$	归约S→a
11	$(S,(S**,** S	≻	**)**)$	归约L→L,S
12	$(S,**(**L	≡	**)**)$	移进
13	$(S,(L**)**	≻	**)**$	归约S→(L)
14	$(S**,**S	≻	**)**$	归约L→L,S
15	$**(**L	≡	**)**$	移进
16	$(L**)**	≻	$	归约S→(L)
17	$S		$	**ACCEPT**

	a	(	)	,	$
a			≻	≻	≻
(	≺	≺	≡	≺	
)			≻	≻	≻
,	≺	≺	≻	≻	
$	≺	≺			

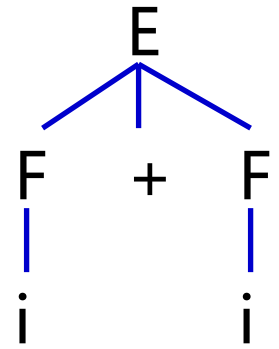
3. 算符优先分析[Operator-Precedence Parsing]

- 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$a \lessdot b$　则令 f(a)<g(b)
    - ✓ 当$a \gtrdot b$　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=b则从f_a到g_b画一条弧，
      - 若a<b或a=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]

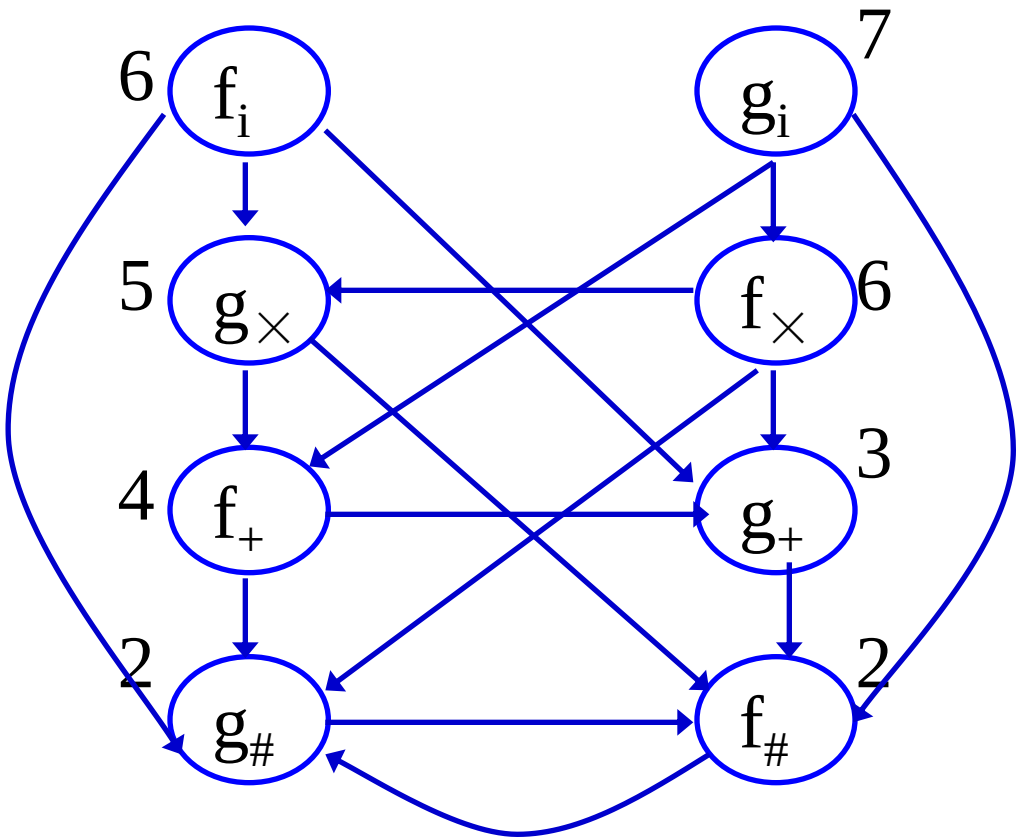
- 例：有优先关系表如下：

	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＝b则从f_a到g_b画一条弧，
若a＜b或a＝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，$g_\#$，$f_\#$，$g_\times$，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