第 4 章 语法分析（1）

Syntax Analysis

【对应教材2.2, 4.1-4.3】

内容提要

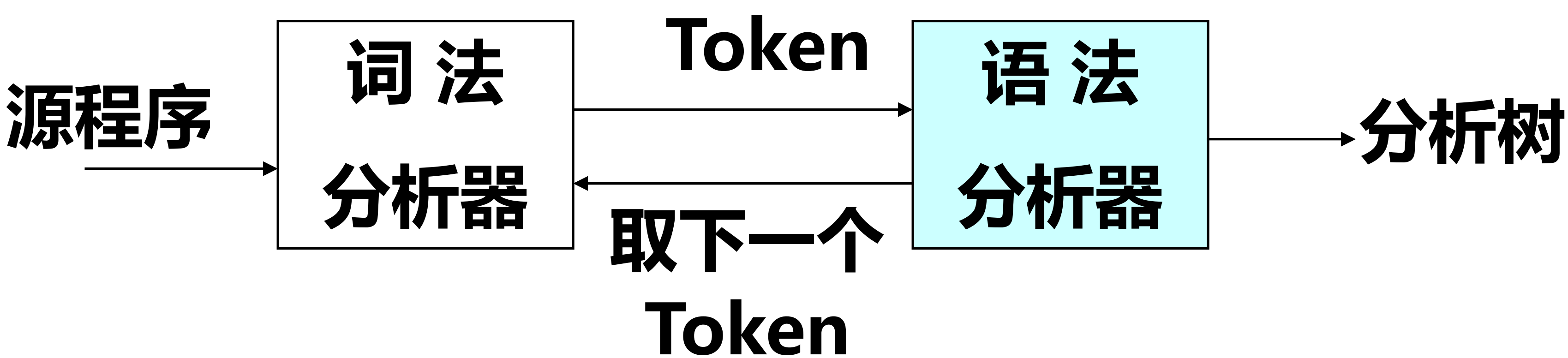
- 语法分析简介
- 上下文无关文法
- 文法的设计方法
- 自顶向下的语法分析
- 自底向上的语法分析
  - 简单LR分析：LR(0), SLR
  - 更强大的LR分析：LR(1), LALR
  - 二义性文法的使用
- 语法分析器生成工具YACC

程序设计语言构造的描述

- 程序设计语言构造的语法可使用**上下文无关文法**或**BNF(Backus-Naur Form)表示法**来描述
  - 文法可以给出精确易懂的语法规则
  - 可以自动构造出**某些**类型文法的语法分析器
  - 文法指出了语言的结构，有助于进一步的语义处理和代码生成
  - 支持语言的演化和迭代

> - **文法：Grammar**
> - **上下文无关文法：Context-Free Grammar, CFG**

语法分析器的作用



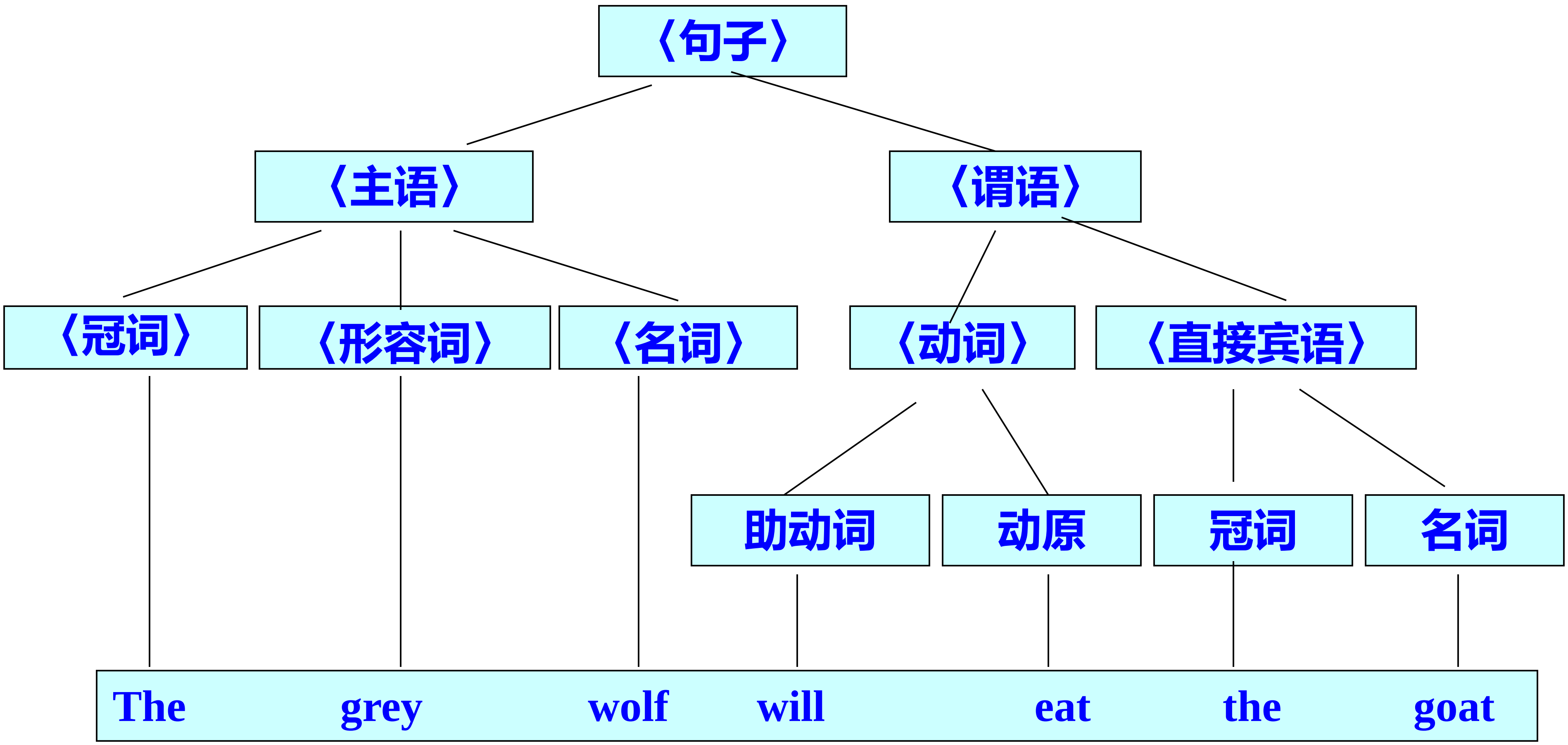
- 功能：根据文法规则，从源程序单词符号串中识别出语法成分，并进行语法检查
- 基本任务：识别符号串S是否为某个合法的语法单元

语法分析器的种类

- 通用语法分析器
  - 可以对任意文法进行语法分析
  - 效率很低，不适合用于编译器
- 自顶向下的语法分析器
  - 从语法分析树的根部开始构造语法分析树
- 自底向上的语法分析器
  - 从语法分析树的叶子开始构造语法分析树
- 后两种方法
  - 总是从左到右、逐个扫描词法单元
  - 只能处理特定类型的文法，但是这些文法足以用来描述常见的程序设计语言

语言和文法基础：例子

分析：The grey wolf will eat the goat



例子

- 为了进行机器分析，“句子由主语后跟随谓语组成”可以表示为：

<句子>→ <主语> <谓语>	(1)
<主语>→ <冠词> <形容词> <名词>	(2)
<冠词>→the	(3)
<形容词> → grey	(4)
<名词>→wolf	(5)
< 谓语>→ < 动词> <直接宾语>	(6)
< 动词>→ <助动词> <动词原形>	(7)
<助动词>→will	(8)
<动词原形> →eat	(9)
<直接宾语>→ < 冠词> <名词>	(10)
<名词> →goat	(11)

句子的语法（Syntax）

- 终结符号集 V_T={the, grey, wolf, will, eat, goat}

- 非终结符号集 V_N= {<句子>，<主语>，<谓语>，<冠词>，<形容词>，<名词>，< 动词>，<直接宾语>，<助动词>，<动词原形>}

- 开始符号 S= <句子>

- 产生规则集 P={<句子>→ <主语><谓语>, ……}

上述句子可根据规则得出

<句子>⇒<主语> <谓语>

⇒ <冠词> <形容词> <名词> <谓语>

⇒ **the** <形容词> <名词> <谓语>

⇒ **the grey**<名词> <谓语>

⇒ **the grey wolf** <谓语>

⇒ **the grey wolf** < 动词> <直接宾语>

⇒ ……

⇒ **the grey wolf will eat the goat**

句子既要符合语法规则又要符合语义规定

<句子> $\stackrel{+}{\Rightarrow}$ **the grey wolf will eat the goat**

the grey wolf will eat the wolf

the grey goat will eat the goat

the grey goat will eat the wolf

符合语法规则且符合语义规定的句子仅是:

the grey wolf will eat the goat

文法（Grammar）的正式定义

文法$G = (V_T, V_N, S, P)$，其中：

- V_T 是一个非空有穷的终结符号(terminal)集合；
- V_N 是一个非空有穷的非终结符号(nonterminal)集合，

  且$V_T \cap V_N = \Phi$；
- $P = \{ \alpha \rightarrow \beta \mid \alpha \in (V_T \cup V_N)^*$且至少包含一个非终结符；$\beta \in (V_T \cup V_N)^* \}$，称为产生式(production)集合;
- $S \in V_N$，称为开始符号(start symbol)。
  - S必须在某个产生式的左部至少出现一次。

产生式可以写成 $A ::= \alpha$ **或** $A \rightarrow \alpha$。

$A \rightarrow \alpha_1$ $A \rightarrow \alpha_2$ **可以缩写为：** $A \rightarrow \alpha_1 | \alpha_2$

上下文无关文法

- Context-free grammar, 简称CFG

- 所有产生式的左边只有一个非终结符号，即
  - 产生式的形式为：$A \rightarrow \beta$
  - 因此不需要任何上下文（context）就可以对A进行推导

- 上下文无关文法描述的语言称为上下文无关语言

例：算术表达式的文法G

G = ({a,+,*,(,)}, {<表达式>，<项>，<因子>}, <表达式>，P)

P：<表达式>→ <表达式> + <项> | <项>
<项>→<项> *<因子> | <因子>
<因子> → (<表达式>) | a

通常可以简写为 (G1[E]):

$$
\begin{aligned}
E &\rightarrow E+T \mid T \\
T &\rightarrow T*F \mid F \\
F &\rightarrow (E) \mid a
\end{aligned}
$$

Expression
Term
Factor

关于文法的一些约定

- 通常可以不用将文法G的四元组显式地表示出来，而只需将产生式写出。
- 一般约定：
  - 第一条产生式的左部是开始符号
  - 用尖括号括起来的是非终结符号，而不用尖括号的是终结符号，或者
  - 大写字母表示非终结符号，小写字母表示终结符号
  - 小写的希腊字母表示（可能为空的）文法符号串
  - 另外也可以把G表示为G[S]，其中S为开始符号。

直接推导(Immediate Derivation)

令$G=(V_T,V_N, S, P)$，若$\alpha \rightarrow\beta\in P$，且

$\gamma, \delta \in(V_T\cup V_N)^*$, 则称 $\gamma\alpha\delta$ 可以**直接推导出** $\gamma\beta\delta$,

表示成 $$\gamma\alpha\delta \Rightarrow \gamma\beta\delta$$

若 $\gamma\alpha\delta$ 直接推导出 $\gamma\beta\delta$，即：

$$\gamma\alpha\delta \Rightarrow \gamma\beta\delta$$

则称 $\gamma\beta\delta$ **直接归约**到 $\gamma\alpha\delta$

归约：reduce (*vi*)、reduction(*n.*) 是推导的逆过程。

推导(Derivation)

一个直接推导序列：

$$\alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n \quad (n > 0)$$

可表示成 $\alpha_0 \Rightarrow^{+} \alpha_n$

$\alpha_0 \Rightarrow^{*} \alpha_n$ **定义为：或者** $\alpha_0 = \alpha_n$

或者 $\alpha_0 \Rightarrow^{+} \alpha_n$

例：

$$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T \Rightarrow a+F \Rightarrow a+a$$

$\alpha A\beta$	$\alpha\gamma\beta$	产生式	α	β
E	$\Rightarrow E+T$	$E\rightarrow E+T$	ε	ε
$E+T$	$\Rightarrow T+T$	$E\rightarrow T$	ε	$+T$
$T+T$	$\Rightarrow F+T$	$T\rightarrow F$	ε	$+T$
$F+T$	$\Rightarrow a+T$	$F\rightarrow a$	ε	$+T$
$a+T$	$\Rightarrow a+F$	$T\rightarrow F$	$a+$	ε
$a+F$	$\Rightarrow a+a$	$F\rightarrow a$	$a+$	ε

最左推导和最右推导

对于w和G，$w \in L(G)$，是否存在$S \overset{+}{\Rightarrow} w$？如何构造这个推导？

例如，G[E] (表达式文法) 和 w = a + a * a

$$E \underset{lm}{\Rightarrow} E+T \underset{lm}{\Rightarrow} T+T \underset{lm}{\Rightarrow} F+T \underset{lm}{\Rightarrow} a+T \underset{lm}{\Rightarrow} a+T*F$$

$$\underset{lm}{\Rightarrow} a+F*F \underset{lm}{\Rightarrow} a+a*F \underset{lm}{\Rightarrow} a+a*a$$

特点：$\alpha A\beta \underset{lm}{\Rightarrow} \alpha\gamma\beta\ (A \to \gamma)$，$\alpha \in V_T^*$ **（最左）**

$$E \underset{rm}{\Rightarrow} E+T \underset{rm}{\Rightarrow} E+T*F \underset{rm}{\Rightarrow} E+T*a \underset{rm}{\Rightarrow} E+F*a$$

$$\underset{rm}{\Rightarrow} E+a*a \underset{rm}{\Rightarrow} T+a*a \underset{rm}{\Rightarrow} F+a*a \underset{rm}{\Rightarrow} a+a*a$$

特点：$\alpha A\beta \Rightarrow \alpha\gamma\beta\ (A \to \gamma)$，$\beta \in V_T^*$ **（最右）**

句型/句子/语言

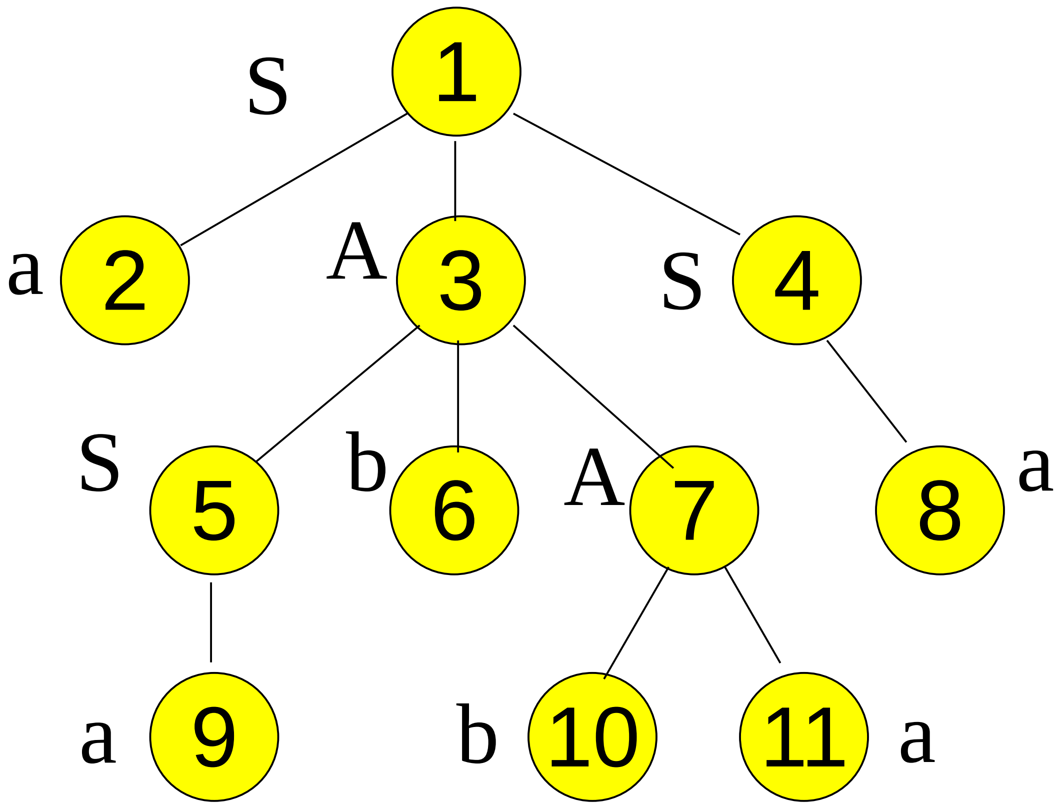
- 句型（sentential form）：
  - 如果$S \Rightarrow^* \alpha$，那么α是文法的句型
  - 句型可能既包含非终结符号，又包含终结符号；
  - 句型也可以是空串
- 句子（sentence）
  - 文法的句子是不包含非终结符号的句型
- 语言
  - 文法G的语言是G的所有句子的集合，记为L(G)
  - w在L(G)中当且仅当w是G的句子，即$S \Rightarrow^* w$

语法分析树（Parse Tree）

- **语法分析树**是推导的一种图形表示形式
  - 根结点的标号是文法的开始符号
  - 每个叶子结点的标号是非终结符号、终结符号或ε
  - 每个内部节点的标号是非终结符号
  - 每个内部结点表示某个产生式的一次应用
  - 内部结点的标号为产生式头，该结点的子结点从左到右对应产生式的右部
- 树的叶子组成的序列是根的文法符号的句型
- 一棵分析树可对应多个推导序列，但是分析树和最左（右）推导序列之间具有一一对应关系

例: G=(V_T,V_N,S,P)，其中

P: $S \rightarrow aAS \mid a \quad A \rightarrow SbA \mid SS \mid ba$



如何画出分析树（1. 自顶向下）

根据推导序列，对每步推导画相应分枝

S

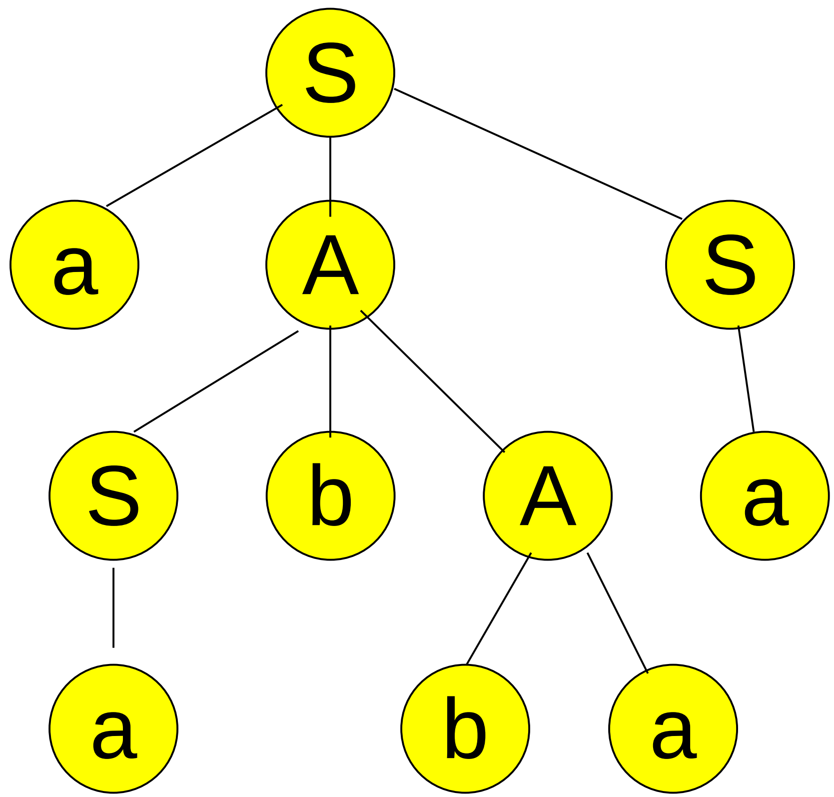
$\Rightarrow aAS$

$\Rightarrow aSbAS$

$\Rightarrow aabAS$

$\Rightarrow aabbaS$

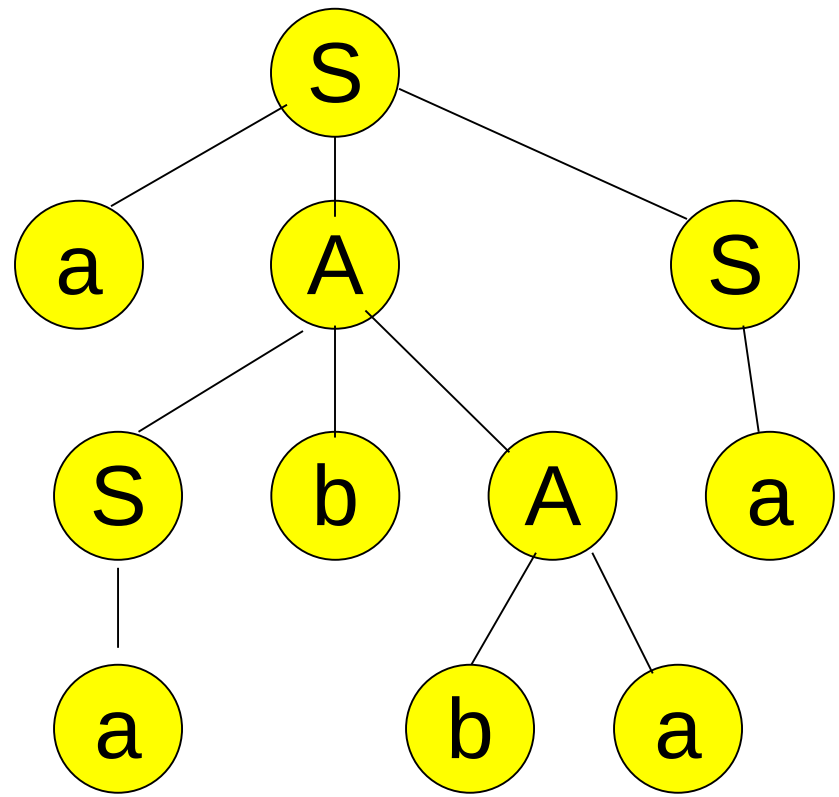
$\Rightarrow aabbaa$



如何画出分析树（2. 自底向上）

根据归约序列，对每步归约画相应分枝

S
$\Rightarrow a\underline{AS}$
$\Rightarrow aA\underline{a}$
$\Rightarrow a\underline{SbA}a$
$\Rightarrow aSb\underline{ba}a$
$\Rightarrow a\underline{a}bbaa$



文法的二义性（Ambiguity）

例：北京市劳模中青年居多。

1. 一个句子的结构可能不唯一；
2. 一个句子对应的分析树可能不唯一。

考虑下面的表达式文法 G2[E],其产生式如下：

$E \rightarrow E+E \mid E*E \mid (E) \mid a$

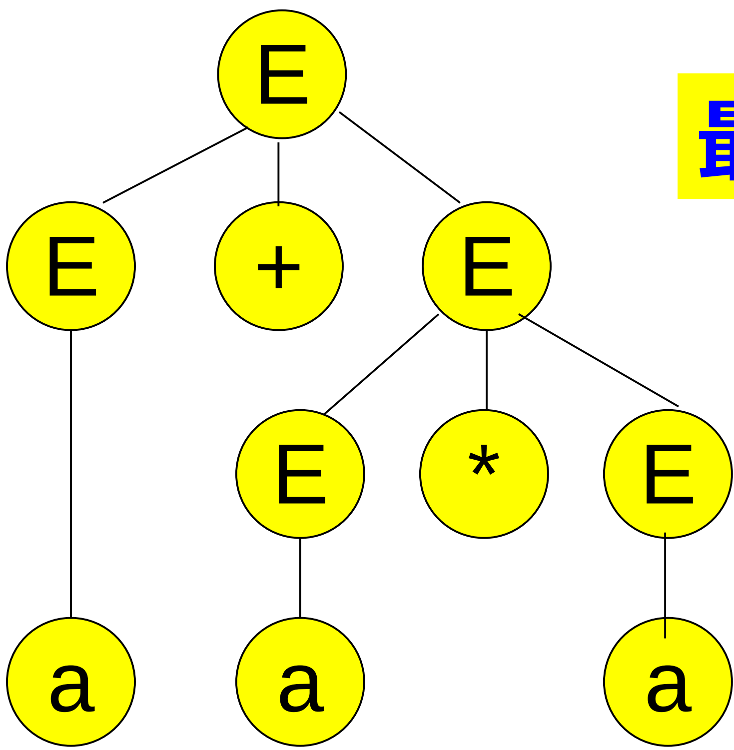
对于句子a+a*a, 有如下两个最左推导：

$E \Rightarrow E+E \Rightarrow a+E \Rightarrow a+E*E \Rightarrow a+a*E \Rightarrow a+a*a$

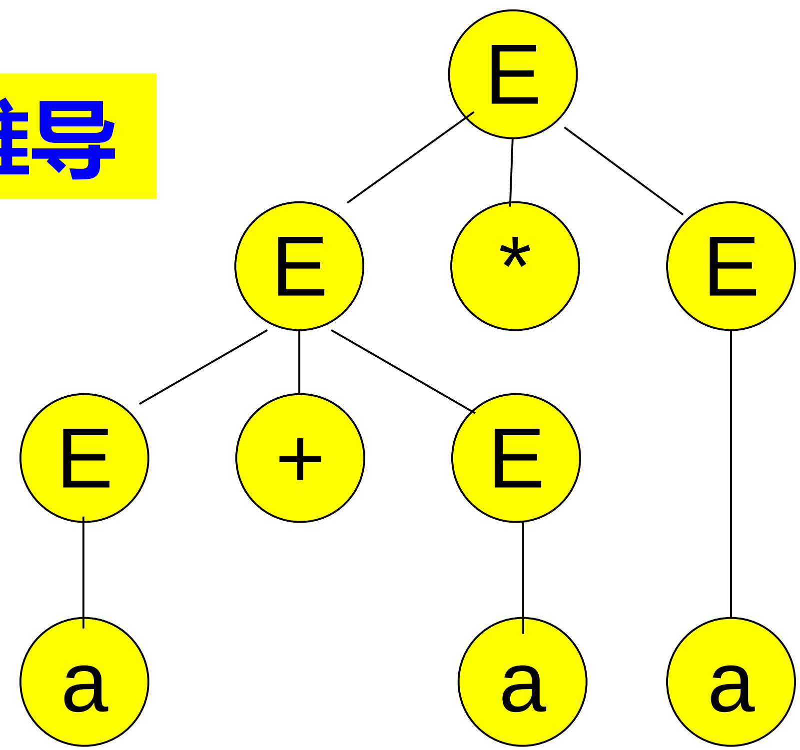
$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow a+E*E \Rightarrow a+a*E \Rightarrow a+a*a$

$E \Rightarrow E+E \Rightarrow a+E \Rightarrow a+E*E \Rightarrow a+a*E \Rightarrow a+a*a$

$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow a+E*E \Rightarrow a+a*E \Rightarrow a+a*a$

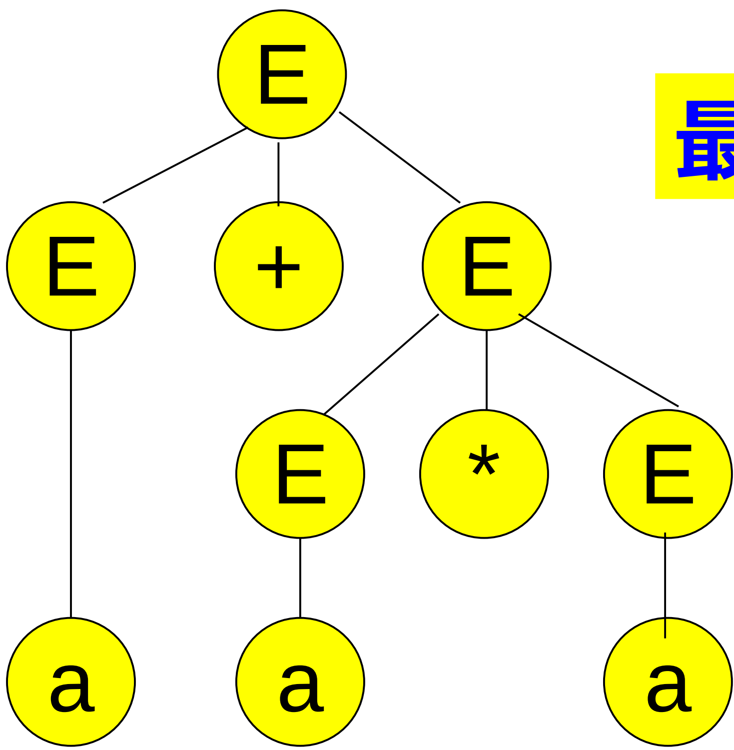


最左推导

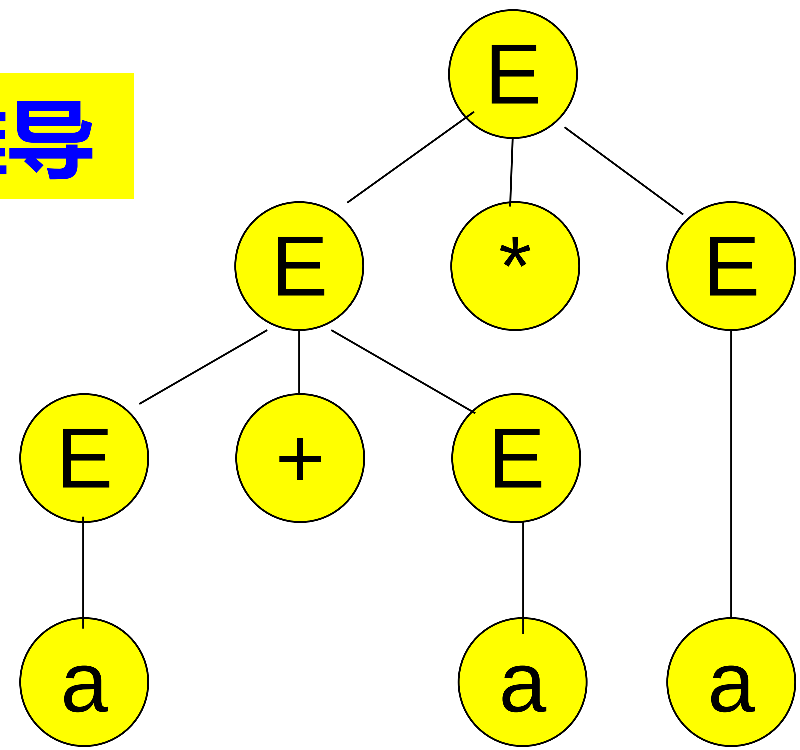


$E \Rightarrow E+E \Rightarrow E+E*E \Rightarrow E+E*a \Rightarrow E+a*a \Rightarrow a+a*a$

$E \Rightarrow E*E \Rightarrow E*a \Rightarrow E+E*a \Rightarrow E+a*a \Rightarrow a+a*a$



最右推导



二义性（或歧义性，Ambiguity)

定义

- 如果一个文法中存在某个句子有两棵分析树，那么该句子是二义性的。
- 如果一个文法产生二义性的句子，则称这个文法是二义性的；
- 否则，该文法是无二义性的。

关于二义性的几点说明-1

1. 一般来说，程序语言存在无二义性文法。
   - 对于表达式来说，文法 G1[E]是无二义性的。
2. 在能驾驭的情况下，可以使用二义性文法。

   对于条件语句，使用二义性文法描述它：

   $$
   \begin{aligned}
   S \rightarrow\ & \textbf{if}\ \ \textbf{expr}\ \ \textbf{then}\ \ S \\
   \mid\ & \textbf{if}\ \ \textbf{expr}\ \ \textbf{then}\ \ S\ \ \textbf{else}\ \ S \\
   \mid\ & \textbf{other}
   \end{aligned}
   $$

   二义性的句子：

   if e1 then if e2 then s1 else s2

关于二义性的几点说明-2

3. 对于任意一个上下文无关文法，不存在一个算法，判定它是无二义性的；但能给出一组充分条件，满足这组充分条件的文法是无二义性的。

4. 存在先天二义性的语言。例如，

$$\{a^i b^i c^j \mid i, j \geq 1\} \cup \{a^i b^j c^j \mid i, j \geq 1\}$$

存在一个二义性的句子$a^k b^k c^k$。

练习-1

- 考虑下面的表达式文法，符号串 $5*3+(2*7)+4$ 对应多少个不同的分析树?
  - $E \rightarrow E*E \mid E+E \mid (E) \mid \text{int}$

- 答案：

  1. {{5 * 3} + (2 * 7) } + 4
  2. {5 *{3 + (2 * 7)}} + 4
  3. {5 * 3} + {(2 * 7) + 4}
  4. 5 * { 3 + {(2 * 7) + 4}}
  5. 5 * {{3 + (2 * 7)} + 4}

练习-2

- 考虑下面的文法，它能生成多少个不同的句子？多少个不同的分析树？
  - **S→A1 | 1B**
  - **A→10 | C | ε**
  - **B→C1 | ε**
  - **C→0 | 1**

- 答案：
  - 5个不同的句子：**1、11、111、01、101**
  - 对应7棵不同的分析树

证明文法生成的语言

- 证明文法G生成语言L可以帮助我们了解文法可以生成什么样的语言。
- 基本步骤：
  - 首先证明$L(G) \subseteq L$
  - 然后证明$L \subseteq L(G)$
  - 一般可以使用数学归纳法
- 证明$L(G) \subseteq L$：
  - 可以按照推导序列长度进行数学归纳
- 证明$L \subseteq L(G)$：
  - 通常可按照符号串的长度来构造推导序列。

文法生成语言的例子（1）

- 文法G：S → (S)S | ε
- 语言L：所有具有对称括号对的串
- L(G) ⊆ L的证明：
  - 归纳基础：推导长度为n =1, S=>ε, 括号对称
  - 归纳步骤：假设长度小于n的推导都能得到括号对称的句子。考虑推导步骤为n的最左推导：

$$S \underset{lm}{\Rightarrow} (S)S \overset{*}{\underset{lm}{\Rightarrow}} (x)S \overset{*}{\underset{lm}{\Rightarrow}} (x)y$$

  - 其中x和y的推导步骤都小于n，因此x和y都是括号对称的句子，因此(x)y也是括号对称的句子
  - => G推导出的所有句子都是括号对称的。

文法生成语言的例子（2）

- **$L \subseteq$ (G)的证明：**
  - 注意：括号对称的串的长度必然是偶数。
- **归纳基础：** 如果括号对称的串的长度为0，显然它可以从S推导得到
- **归纳步骤：** 假设长度小于2n的括号对称的串都能够由S推导得到。假设w是括号对称且长度为2n的串
  - w必然以左括号开头，且w可以写成(x)y的形式，其中x也是括号对称的。因为x、y的长度都小于2n，根据归纳假设，x和y都可以从S推导得到。
  - 因此S=>(S)S =>* (x)y。

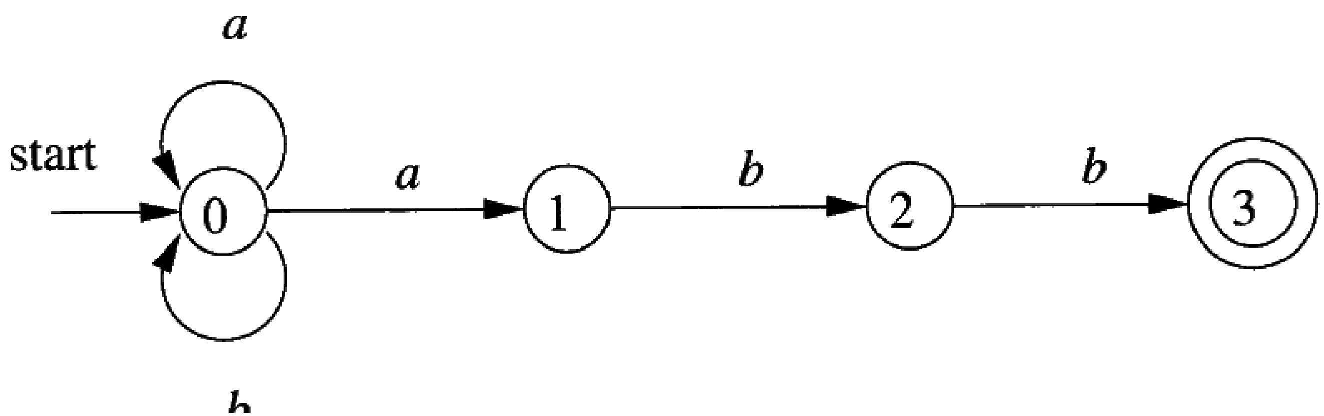
上下文无关文法和正则表达式（1）

- 上下文无关文法比正则表达式的能力更强：
  - 所有的正则语言都可以使用上下文无关文法描述
  - 但是一些用上下文无关文法描述的语言不能用正则文法描述
- 证明：
  - 首先证明：存在上下文无关文法$S \rightarrow aSb \mid ab$ 描述了语言$\{a^nb^n|n>0\}$，但是它无法用DFA识别。
  - 反证法：假设有DFA识别该语言，且有K个状态。那么在识别a^{k+1}...的输入串时，必然两次到达同一个状态。设自动机在第i个和第j个a时进入同一个状态，那么：因为DFA识别L，a^jb^j必然到达接受状态，因此a^ib^j必然也到达接受状态。

上下文无关文法和正则表达式（2）

- 证明（续）
  - 其次证明：任何正则语言都可以表示为上下文无关文法的语言。
  - 任何正则语言都必然有一个等价的NFA。对于任意的NFA构造如下的上下文无关文法：
    - 对NFA的每个状态i，创建非终结符号A_i。
    - 如果有i在输入a上到达j的转换，增加产生式$A_i \rightarrow aA_j$。
    - 如果i在输入ε上到达j，那么增加产生式$A_i \rightarrow A_j$.
    - 如果i是一个接受状态，增加产生式$A_i \rightarrow \varepsilon$
    - 如果i是开始状态，令A_i为所得文法的开始符号。

NFA构造文法的例子



$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$

$A_1 \rightarrow bA_2$

$A_2 \rightarrow bA_3$

$A_3 \rightarrow \varepsilon$

NFA接受一个句子的运行过程实际是文法推导出该句子的过程。（可以考虑**baabb**的推导和接受过程）

非上下文无关的语言结构-1

- 在我们使用的程序语言中，有些语言结构并不能用上下文无关文法描述的。

例1： $L1 = \{ wcw \mid w \in \{a,b\}^+ \}$。例如，aabcaab 就是L1的一个句子。这个语言是检查程序中标识符的声明应先于引用的抽象。

例2： $L2 = \{ a^n b^m c^n d^m \mid n,m \geq 0 \}$，它是检查过程声明的形参个数和过程调用的实参个数一致问题的抽象。

非上下文无关的语言结构-2

- 过程定义和引用的语法并不涉及到参数个数，例如，C语言的函数语句可描述为

$$
\begin{aligned}
\text{s-call} &\rightarrow \text{id (r-list)} \\
\text{r-list} &\rightarrow \text{r-list, r} \\
&\mid \text{r}
\end{aligned}
$$

- 实参和形参个数的一致性检查也是放在语义分析阶段完成的。

文法分类（Chomsky）

0型（任意文法）： $G=(V_T,V_N,S,P)$

规则形式： $\alpha\rightarrow\beta$ $\alpha, \beta\in (V_T\cup V_N)^*, \alpha\neq\varepsilon$

推导：$\gamma\alpha\delta\Rightarrow\gamma\beta\delta$

1型（上下文有关, Context-Sensitive Grammar)

规则形式： $\alpha A\beta \rightarrow \alpha\gamma\beta$

$A \in V_N, \alpha, \gamma, \beta \in (V_T\cup V_N)^*, \gamma\neq\varepsilon$

(注：可以包含$S\rightarrow\varepsilon$, 但此时不允许S出现在产生式右边)

2型（上下文无关, Context-Free Grammar, CFG)

规则形式： $A\rightarrow\beta$ ， $A \in V_N,\ \beta \in (V_T\cup V_N)^*$

3型（正则文法, Regular Grammar)

（右线性）： $A\rightarrow aB$ $A \rightarrow a$

（左线性）： $A\rightarrow Ba$ $A \rightarrow a$ $a \in V_T\cup\{\varepsilon\}$

每一类逐渐对产生式施加限制，表示范围逐步缩小。

在程序语言中的实际应用

- 与词法有关的规则属于正则文法；
- 与局部语法有关的规则属于上下文无关文法；
- 而与全局语法和语义有关的部分往往要用上下文有关文法来描述。
  - 实际上很少使用
- 为简化分析过程，会把描述词法的正则文法从描述语法的上下文无关文法中分离出来。
  - 在分离出正则文法后的上下文无关文法中，这些单词符号是属于终结符号V_T中的符号。
  - 例：表达式文法G1[E]中，a是终结符号。

文法的设计方法

- 文法能够描述程序设计语言的大部分语法
  - 但不是全部，比如：标识符的先声明后使用无法用上下文无关文法描述。
  - 语法分析器接受的语言是程序设计语言的超集。必须通过语义分析来剔除一些符合文法、但不合法的程序。

文法变换方法

- 常见的文法变换方法
  - 消除二义性
  - 消除左递归
  - 提取左公因子

消除文法的二义性（1）

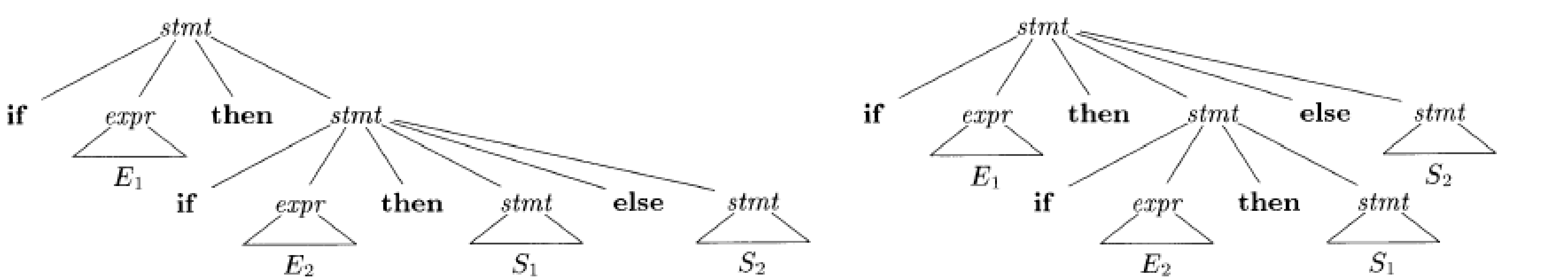
- 一些二义性文法可以被改成等价的无二义性文法
- 例子 (dangling-else):

  stmt* → if *expr* then *stmt

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

  |other

- **if E_1 then if E_2 then S_1 else S_2**有两棵语法树



消除文法的二义性（2）

- 为保证“else和最近未匹配的then匹配”，我们要求在then分支的语句必须是匹配好的。
- 引入matched_stmt表示匹配好的语句，有如下文法：

stmt → *matched_stmt* | *open_stmt*

matched_stmt → if *expr* then *matched_stmt* else *matched_stmt*

| other

open_stmt → if *expr* then *stmt*

| if *expr* then *matched_stmt* else *open_stmt*

- 二义性的消除方法没有规律可循。

例：消除文法的二义性

- 文法G的产生式如下：

  $S \rightarrow aSb \mid bSa \mid SS \mid ab \mid ba$

  - $L(G) = ?$

- G是二义性的，比如 ababab有两个不同的最左推导。
  - S=>SS => abS => abSS => ababS => ababab
  - S=>SS => SSS => abSS => ababS => ababab

等价的上下文无关文法：

$S \rightarrow TS \mid T$　　　$T \rightarrow aB \mid bA$

$A \rightarrow a \mid bAA$　　　$B \rightarrow b \mid aBB$

消除文法中的左递归

- 文法左递归 $A \Rightarrow^{+} A\alpha$
- 直接左递归 $A \rightarrow A\alpha \mid \beta$
  - 串的特点 $\beta\alpha \ldots \alpha$
- 消除直接左递归（变换成右递归）

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

例：

- 算术表达文法G：

$$E \to E + T \mid T \qquad (T + T \ldots + T)$$

$$T \to T * F \mid F \qquad (F * F \ldots * F)$$

$$F \to (E) \mid \mathbf{id}$$

- 消除左递归后的文法G'：

$$E \to TE'$$

$$E' \to + TE' \mid \varepsilon$$

$$T \to FT'$$

$$T' \to * F T' \mid \varepsilon$$

$$F \to (E) \mid \mathbf{id}$$

间接左递归的消除

- 间接左递归

$$S \to Aa \mid b$$

$$A \to Sd \mid \varepsilon$$

- 先变换成直接左递归

$$S \to Aa \mid b$$

$$A \to Aad \mid bd \mid \varepsilon$$

- 再消除左递归

$$S \to Aa \mid b$$

$$A \to bd\,A' \mid A'$$

$$A' \to adA' \mid \varepsilon$$

消除所有左递归的算法

1. 把G的非终结符整理成某种顺序$A_1, A_2, \ldots\ldots A_n$。

2. for i:=1 to n do {

    for j :=1 to i-1 do {

        把每个形如$A_i \rightarrow A_j r$的规则替换成

            $A_i \rightarrow \delta_1 r|\delta_2 r|\ldots\ldots|\delta_k r$

        其中$A_j \rightarrow \delta_1|\delta_2|\ldots\ldots|\delta_k$是当前全部$A_j$的产生式;

    }

    消除A_i规则中的直接左递归

}

3. 化简由2得到的文法即可。

要求无环路
$A \Rightarrow^{+} A$且
无ε-产生式

示例：消除左递归

例：文法G[s]为

$S \rightarrow Ac|c$

$A \rightarrow Bb|b$

$B \rightarrow Sa|a$

该文法无直接左递归，但有间接左递归

$S \Rightarrow Ac \Rightarrow Bbc \Rightarrow Sabc$ **即：** $S \Rightarrow^{+} Sabc$

非终结符顺序重新排列

$B \rightarrow Sa|a$

$A \rightarrow Bb|b$

$S \rightarrow Ac|c$

顺序为：B, A, S

$B \to Sa|a$
$A \to Bb|b$
$S \to Ac|c$

$A \to (\underline{Sa \mid a})\, b \mid b$ 把B的产生式代入A中

$A \to Sab \mid ab \mid b$

$S \to (\underline{Sab \mid ab \mid b})\, c \mid c$ 把A的产生式代入S中

$S \to Sabc \mid abc \mid bc \mid c$

$S \to abcS' \mid bc\, S' \mid cS'$ 消除直接左递归

$S' \to abcS' \mid \varepsilon$

$A \to Sab \mid ab \mid b$

$B \to Sa \mid a$

顺序为：S, A, B

$S \to Ac|c$
$A \to Bb|b$
$B \to Sa|a$

$B \to (\underline{Ac\,|\,c})\ a\ |\ a$　　把S的产生式代入B中

$B \to Aca\ |\ ca\ |\ a$

$B \to (\underline{Bb\,|\,b})\ ca\ |\ ca\ |\ a$　　把A的产生式代入B中

$B \to Bbca\ |\ bca\ |\ ca\ |\ a$

$B \to bcaB'\ |\ caB'\ |\ aB'$　　消除直接左递归

$B' \to bcaB'\ |\ \varepsilon$

$S \to Ac\ |\ c$
$A \to Bb\ |\ b$
$B \to bcaB'\ |\ caB'\ |\ aB'$
$B' \to bcaB'\ |\ \varepsilon$

最后G[s]的产生式

由于对非终结符的排序不同，最后得到的文法在形式上可能是不一样的，但是可以证明它们是等价的。

预测分析法简介

- 试图从开始符号推导出输入符号串；
- 每次为最左边的非终结符号选择适当的产生式；
  - 通过查看下一个输入符号来选择这个产生式。
- 例子：**E→ +EE | -EE | id**；输入为**+ id - id id**
- 当两个产生式具有相同的前缀时无法预测
  - 文法：***stmt* → if *expr* then *stmt* else *stmt* | if *expr* then *stmt***
  - 输入：**if a then …**
- **需要提取公因子!**

提取左公因子

- 含有左公因子的文法

$$A \to \alpha\beta_1 \mid \alpha\beta_2$$

- 提取左公因子

$$A \to \alpha A'$$
$$A' \to \beta_1 \mid \beta_2$$

提取最长公共前缀作为公因子

例：提取左公因子

- 悬空*else*的文法

  stmt → if *expr* then *stmt* else *stmt*
  | if *expr* then *stmt*
  | other

- 提取左公因子

  stmt → if *expr* then *stmt* *optional_else_part*
  | other
  optional_else_part → else *stmt*
  | ε

作业

- 文法分析
  - Ex. 2.2.1, Ex. 4.2.1
- 上下文无关文法
  - Ex. 4.2.3